Java aktuell

iJUG Verbund
www.ijug.eu

Container
Kubernetes und Alternativen im Detail

CI/CD
Continuous Integration, Delivery und Deployment

Agilität
Agile Entwicklung und Gamification

Ausgabe 02/2021

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 02

Java aktuell

Mehr Informationen
zum Magazin und
Abo unter:

**https://www.ijug.eu/
de/java-aktuell**

iJUG
Verbund
www.ijug.eu

Liebe Leser der Java aktuell,

in dieser Ausgabe haben unsere Autoren eine bunte und wissenswerte Mischung von Fachartikeln aus den Themenbereichen Cloud und Container für Sie vorbereitet. Am Anfang bringen Andreas Badelt und Markus Karg Sie wie gewohnt auf den aktuellen Stand, was in der Java- und Eclipse-Welt so passiert ist.

Unsere Reise in die Schwerpunktthemen beginnt mit Thomas Michaels Wegweiser durch das DevOps-Labyrinth. Schritt für Schritt begleitet er Java-Entwickler darin mit dem Ziel, eine Anwendung in der Cloud betreiben zu können. Im Anschluss beleuchtet Matthias Häußler den Inbegriff der Container-Orchestrierung näher: Kubernetes. Der Autor beschreibt die notwendigen Voraussetzungen und Kenntnisse und vergleicht den bekannten Dauerbrenner mit den alternativen Open-Source-Technologien. Im Java-Universum gibt es weitere Container-Systeme, die einen Blick wert sind. Nils Bokermann stellt ab Seite 25 „Alternativen zum Monolithen Kubernetes" vor. Im folgenden Praxisbericht schickt Markus Lohn den Oracle Service Bus (OSB) in Rente und ersetzt ihn mit einer Microservices-Architektur unter OpenShift. Er beschreibt im Detail, ob und weshalb sich die Umstellung gelohnt hat.

Aufgrund steigender Anforderungen an Sicherheit und Compliance ist Policy-as-Code entstanden. Die Idee dahinter ist, Policies in einer höheren Programmiersprache zu schreiben und zu automatisieren. Andreas Zitzelsberger stellt ab Seite 35 das Open-Source-Werkzeug Open Policy Agent (OPA) zur Umsetzung von Policy-as-Code vor.

Panik vor Continuous Integration und Delivery? Unsere Autoren geben Ihnen nützliche Informationen an die Hand, damit auch Sie diese Prozesse erfolgreich einsetzen können. Los geht es mit Christian Uhl. Er hat reichlich Projekterfahrung mit CD gesammelt und kann sich nicht mehr vorstellen, jemals wieder darauf zu verzichten. Anhand eines praxisnahen Projektberichts erläutert er einen sicheren Weg zu einer CD-Pipeline und stellt deren Vorteile. Im darauffolgenden Artikel befassen sich Johannes Schnatterer und Daniel Huchthausen mit dem Einsatz von GitOps im CD-Prozess. Die beiden Autoren zeigen anhand von Beispielen dessen Stärken auf und vergleichen ihre Methode mit „klassischer" Continuous Delivery. Auch Dominik Röschke und Nikolas May tragen einen Teil zur Thematik bei, indem die beiden die Continuous-Integration-Server (CI) GitLab CI, CircleCI und Drone CI beleuchten und miteinander vergleichen. Ist einer dieser Server die ultimative Lösung?

Distributed Tracing wird, vor allem bei Microservices-Anwendungen, zur Überwachung von Anwendungen eingesetzt und hilft bei der Suche nach Fehlern und Performance-Schwachstellen. Gunnar Hilling zeigt ab Seite 55, wie Sie Distributed Tracing mit Jaeger und Java umsetzen können. IT-Projekte im Bankenumfeld sind oft besonders sensibel. Dr. Stefan Koch hat ein solches bei der Umstellung zu einem agilen Entwicklungsmodell begleitet. Im Artikel teilt er seinen persönlichen Rückblick im Hinblick auf Migration und Aspekte der Implementierung. Den „zwei Seiten eines Daten-Projekts" widmet sich Mark Keinhörster ab Seite 65. Er stellt verschiedene Werkzeuge vor, die für eine Optimierung des Projekts vom Feature-Engineering bis hin zur Visualisierung sorgen.

Zum Abschluss dieser Ausgabe erläutert Roman Simschek Gamification näher. Anhand seines Trainingsseminars zeigt er, wie man den Teilnehmenden mithilfe einer spielerischen Herangehensweise die Themen Agilität und Scrum näherbringen kann.

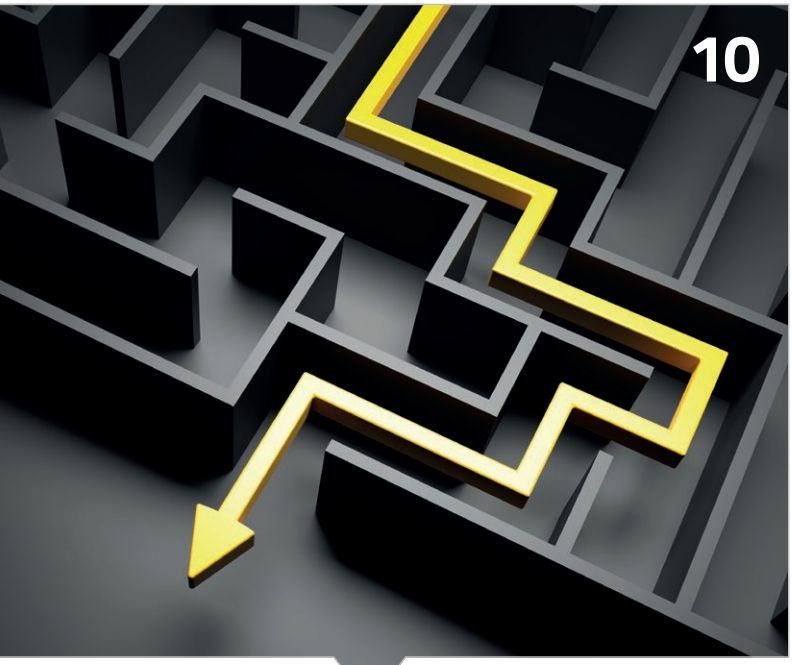
Wir wünschen Ihnen viel Spaß beim Lesen!

Ihre

Lisa Damerow

Redaktionsleitung Java aktuell

Inhalt



Einstieg in die Cloud

Projekterfahrung beim Wechsel von OSB zu OpenShift

Keine Panik vor Continuous Deployment

Agilität im Bankenumfeld integrieren

Das Java-Tagebuch

Andreas Badelt, stellv. Leiter der DOAG Java Community

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

1. Oktober 2020

Reaktion auf zunehmende Reaktivität

Nicht direkt Java-bezogen, jedoch auch in der Java-Welt relevant: Auf den Siegeszug des „Reactive Manifesto" folgen die „Reactive Principles". Diese sollen ein Begleitwerk („Companion") zum Manifesto sein, quasi die Interpretationshilfe zum Gesetzbuch. Veröffentlicht wurden sie von der im September 2019 gegründeten Reactive Foundation, hinter der einige Personen stecken, die auch am Manifest beteiligt waren, aber auch mehrere große Firmen *[1]*.

20. Oktober 2020

OSGi Alliance geht in Eclipse auf

Die altehrwürdige OSGi Alliance (20 Jahre fühlen sich in der IT-Branche immer noch wie mehrere Zeitalter an) löst sich auf und übergibt ihre Projekte der Eclipse Foundation. Letztere scheint sich zu einer Art schwarzem Loch im Java-System zu entwickeln, das nach und nach alle Projekte anzieht. Ich hoffe, dass die Analogie nicht allzu weit trägt; bislang erstrahlen die neuen Projekte ja allesamt, statt hinter dem Ereignishorizont zu verschwinden. Gerade für die OSGi Alliance gibt es ja auch eine langjährige Verbindung zu Eclipse, nicht nur weil die Plug-ins der Eclipse IDE als OSGi-Bundles realisiert sind.

23. Oktober 2020

EclipseCon rein virtuell

Auch die EclipseCon wurde von der Pandemie getroffen, hat allerdings die Zeit genutzt, um eine reine Online-Konferenz auf die Beine zu stellen, mit 2.600 Teilnehmenden in den vergangenen vier Tagen. Obwohl die Anreise nach Ludwigsburg diesmal offensichtlich keine Hürde war, kam die größte Gruppe (406) laut Eclipse-Foundation-Zählung aus Deutschland, gefolgt von 386 Teilnehmenden aus den USA und 335 aus Indien. Die Video-Mitschnitte praktisch aller Sessions sind auf YouTube verfügbar *[2]*.

28. Oktober 2020

MicroProfile 4.0 mit erstem Release Candidate

MicroProfile 4.0 RC 1 ist vor ein paar Tagen erschienen und bringt eine Reihe von Aktualisierungen der Einzelspezifikationen: Config 2.0, Open API 2.0, OpenTracing 2.0, Metrics 3.0, Health 3.0, REST Client 2.0 und JWT Authentication 1.2 (wobei Letztere auch unter den Namen „JWT Token" und „JWT RBAC" herumzugeistern scheint). Auf höherer Ebene dreht sich die Diskussion gerade darum, ob 4.0 ein „Platform Specification Release" werden soll oder ein reines Marketing-Release (also eine Sammlung von Einzel-Spezifikationen, die nur als „MicroProfile 4.0" beworben wird).

Ersteres erfordert nach den Richtlinien der Eclipse Foundation, dass zum Release-Zeitpunkt mindestens eine kompatible Implementierung angegeben werden muss, die auch alle optionalen Features implementiert und anhand eines einzigen TCK geprüft wird – die Rolle, die GlassFish bei Jakarta EE spielt. Unter anderem aus Gründen der Neutralität zwischen den vielen beteiligten Herstellern will man dies offensichtlich erst mal vermeiden – es geht klar in Richtung Marketing-Release, ohne „MicroProfile compatible"-Logo.

Die Diskussion ist durch die Umstrukturierung des MicroProfile-Projekts zu einem Spezifikations-Projekt innerhalb der Eclipse Foundation überhaupt erst angestoßen worden. Ein Nebeneffekt dieser Umstrukturierung ist, dass aus dem Namen Eclipse MicroProfile demnächst wieder „nur" MicroProfile wird. Muss man nicht im Detail verstehen, ist aber auf jeden Fall kürzer.

11. November 2020

KI für (bald arbeitslose) Entwickelnde

Es geht weiter mit künstlicher Intelligenz in der Software-Entwicklung. Mit DiffBlue kann der Traum des faulen Programmierers wahr werden (meine Worte, nicht der Marketing-Text): Unit-Tests auf Knopfdruck. Das von Forschern aus Oxford geschriebene Tool nebst Startup existiert schon seit mehreren Jahren, ich bin erst jetzt darüber gestolpert. Per Maven oder IntelliJ-Plug-in (rechte Maustaste – „write tests") werden Testklassen erzeugt, die im Großen und Ganzen aussehen wie handgeschrieben. Ok, wie das dem Test-driven Development dient, klären wir später. Oder die KI klärt das für uns, indem sie demnächst erst den Test schreibt und dann den Produktiv-Code. Ich hoffe, das ist jetzt hinreichend absurd, damit wir uns noch ein paar Jahrzehnte lang darauf einstellen können.

14. November 2020

Neue Spring-Versionen

Wie immer kurz nacheinander sind das neue Spring Framework 5.3 und das darauf aufbauende Spring Boot 2.4 fertig geworden. Beide unterstützen jetzt neben 8 und 11 das aktuelle Java 15. Das Framework kommt mit einer Vielzahl kleinerer Änderungen, unter anderem ist der Kern des R2DBC-Supports (nein, nicht der aus Star Wars, sondern die Reactive Relational Database Connectivity) jetzt direkt im Framework verankert und Kotlin-Multi-Plattform- aka „Multi-Format Reflectionless"-Serialisierung wird (erst mal nur für JSON) in den Web- und Messaging-Libraries unterstützt.

Spring Boot kommt mit einer Reihe von Neuerungen im Konfigurationsmanagement. Eine wesentliche ist der Umgang mit Konfigurationsdateien hinsichtlich Spezifität und Reihenfolge; beispielsweise kann eine externe application.properties-Datei eine spezifischere

application-prod.properties im Jar-File überschreiben. Hier könnte eine Migration nötig sein – für den Übergang lässt sich jedoch das „Legacy Processing" aktivieren. Wenn tatsächlich viel mit Konfigurationsprofilen gearbeitet wird, könnten die neuen „Profile Groups" ganz interessant sein. Origin Chains für die Konfigurationsitems können darüber hinaus die Suche nach dem richtigen Eintrag erleichtern. Ansonsten gibt es einen neuen „Startup Endpoint", der, wie der Name verspricht, Informationen über den Startup-Prozess preisgibt, etwa über Beans mit langen Initialisierungszeiten.

16. November 2020

Azul Zulu setzt auf Apple Silicon

Azul Systems nutzt den Hype um Apple Silicon: Der JVM-Hersteller hat einen OpenJDK-Build für die neuen Macs mit Apples eigenen ARM-basierten Chips angekündigt. Der zugehörige JEP 391 „Port the JDK to macOS/AArch64" ist bereits im August angelegt worden, bislang aber noch keinem zukünftigen Release zugeordnet (für 16 wird es wohl nichts mehr). Ist vielleicht auch nicht unbedingt nötig, Azul will jedenfalls auch ältere Releases (8, 11, 13 und 15) auf die neue Architektur portieren. Auch hier ist die Nutzung generell frei, Geld verdient Azul dann mit optionalem Support.

19. November 2020

Jakarta: Enter MicroProfile

Jakarta EE und MicroProfile nähern sich personell weiter aneinander an. Im Steering Committee von Jakarta wurde beschlossen, dass alle strategischen Mitglieder des Jakarta-Projekts auch Mitglieder der MicroProfile Working Group werden sollen – so dies nicht bereits der Fall ist. Damit das Engagement nicht am Budget scheitert, sollen sie den neuen Mitgliedsbeitrag für 2021 aus dem eigenen Jakarta-Topf erhalten. Das heißt: Passend zum nahenden Black Friday gibt's jetzt die „zwei für eins"-Aktion bei Eclipse.

22. November 2020

Jakarta EE 9

Jakarta EE 9 ist endlich da. Die „New Features"-Liste wird bekanntermaßen diesmal eingespart, aber unter der Haube hat der wichtige Wechsel der Package-Namen von „javax" zu „jakarta" stattgefunden. Damit sind die (marken-)rechtlichen Altlasten über Bord und der Blick kann nach vorne gerichtet werden. Zunächst soll möglichst schnell 9.1 erscheinen, mit verpflichtendem JDK 11 Support für alle Implementierungen. Die reine Unterstützung mit 9.1 ist natürlich nur die halbe „Miete" – beginnend mit EE 10 soll dann die aktive Nutzung neuer Konstrukte in den APIs in den Fokus rücken. Was EE 10 dann bringen wird, ist noch nicht klar. Aber die „Jakarta EE Ambassadors" arbeiten schon seit einiger Zeit an einem Plan inklusive eines „Contribution Guide" für alle Interessierten *[3]*: Neben den schon früher erwähnten Themen „CDI Alignment" und „Java SE Alignment" sowie dem Schließen von Spezifikationslücken (OAuth, JSON, AMQP/Kafka/MQTT, Microservices-Profile usw.) geht es natürlich um neue und aktualisierte Einzel-Spezifikationen wie NoSQL und MVC.

27. November 2020

JDK 16 Early Access

Die Entwicklung am JDK 16 geht mit den Early-Access-Builds in die Zielgerade. Hinzugekommen sind unter anderem ein weiteres Inkubator-Update des Foreign-Memory-API (erstmals in Java 14) und darauf aufbauend nun das Foreign-Linker-API für statisch typisierten „pure Java"-Zugriff auf nativen Code (ebenfalls als Inkubator- Feature, sprich das API kann sich je nach Feedback noch stark ändern oder auch ganz wegfallen). Records hingegen verlassen in Release 16 den Inkubator, mit einer geplanten – allerdings nicht nur Records betreffenden – Änderung: Innere Klassen werden in Zukunft statische Members haben dürfen, was dann auch bedeutet, dass sie Record-Klassen deklarieren dürfen.

30. November 2020

GraalVM Native Images für Spring Boot

GraalVM-Unterstützung für Spring hatte ich wohl noch nicht erwähnt, aber zum Glück wird man ja jeden Tag mit Twitter-Meldungen vollgezwitschert, und ein paar relevante sind immer dabei. Um ein GraalVM Native Image mit Spring Boot 2.3 oder jünger zu bauen, reichen jetzt die spring-graalvm-native-Library als Dependency und das Boot-Maven-Plug-in. Mit „mvn package:build-image" wird daraus ein Image – Buildpacks sei Dank! Wer noch nicht mit BuildPacks zu tun hatte – dieser Blog-Eintrag kam praktisch gleichzeitig angezwitschert und bietet eine gute Einführung (nicht nur für Spring Boot) *[4]*.

1. Dezember 2020

Quarkus.io 1.10

Quarkus.io 1.10 ist freigegeben worden, mit einer Reihe kleinerer Änderungen: beispielsweise JSON als Default Content Type, ein verbesserter CodeStart jetzt als Default zur Generierung neuer Projekte und kleinere Verbesserungen für die Integration mit großen Cloud-Ökosystemen.

3. Dezember 2020

Virtuelle Konferenzen in Corona-Zeiten

Konferenzen in Pandemie-Zeiten sind ein großes Thema. Aber die Technologie macht große Fortschritte. Komplett ersetzen lässt sich das persönliche Konferenz-Erlebnis vor Ort nicht, es gibt jedoch auch ohne Virtual Reality einige Ansätze und Plattformen, die zumindest ein bisschen „Konferenz-Feeling" aufkommen lassen. Eine davon hat eine Gruppe von iJUG-Aktiven ausprobiert, die eine

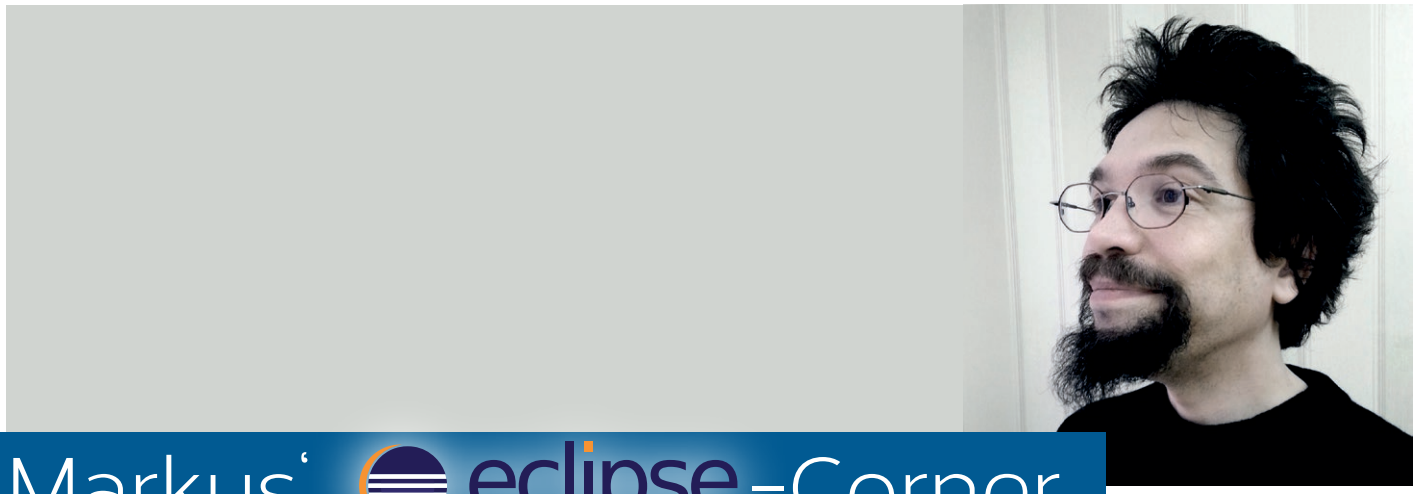
Neuauflage der Cyberland unter dem Namen „Cyberland 2D" *[5]* organisiert haben. „Gather.town" bietet ein Konferenzzentrum-Layout, in dem sich der eigene Avatar nicht nur in Konferenzräume bewegen lässt, sondern sich auch in der Lobby an Themen-Tische begeben und auf andere Avatare treffen kann. Insbesondere Letzteres macht das Ganze spannend: Von allen Avataren in der „virtuellen Nähe" – beziehungsweise den Personen dahinter – sieht man, sofern eingeschaltet, die Kamera in einem kleinen Fenster und kann ins Gespräch kommen. Das Ganze erfordert ein wenig Übung und das grundsätzliche Miteinander im virtuellen Raum will auch erst gelernt sein. Aber es ist deutlich spannender als reine Konferenz-Sessions per Video. Fortsetzung folgt im nächsten Jahr!

Referenzen:

[1] *https://principles.reactive.foundation*
[2] *https://www.eclipsecon.org/2020/*
[3] *https://bit.ly/3lV3z8w*
[4] *https://blog.codecentric.de/en/2020/11/buildpacks-spring-boot/*
[5] *https://cyberland.ijug.eu*

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

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



Markus' eclipse-Corner

So, nun ist also Jakarta EE 9 endgültig veröffentlicht. Seit dem 8. Dezember oder äh… dem 20. November oder… welches Datum gilt denn nun eigentlich? Nach all den monatelangen Verschiebungen, für die übrigens die Programmierer selbst nichts können, da ausschließlich von den beteiligten Arbeitgebern (allen voran Oracle, IBM und Red Hat) verursacht, setzt die Eclipse Foundation (EF) nun noch einen drauf und verwirrt uns mit einer Auswahl willkürlich gesetzter, alternativer Veröffentlichungstermine. Da sie nun allesamt um sind: Hurra!

Jakarta EE 9 kommt mit keinerlei neuen Features, einem Haufen Arbeit für die Anwender bei der Migration (Umstellung der Package-Namen) und null Nutzen dank keinerlei neuen Features. Das muss einem Konsortium von Multi-Milliarden-Unternehmen erst mal einer nachmachen. Warum sollte man sich dieses Release denn bitteschön antun?

Weil uns keine andere Wahl bleibt. Jakarta EE ist nun mal die einzige unternehmensübergreifende Industrienorm für die Ansteuerung beliebiger Application-Server. Punkt. Alles andere sind herstellerspezifische Produkte, die uns dem Vendor-Lock-in, als der direkten Abhängigkeit von der Gnade eines bestimmten Anbieters, aussetzt – das hatten wir mit Oracle und wollen es nie wieder. Und sobald die Hersteller ihre zertifizierten Produkte auf Version 9 upgraden, ist Schicht mit javax.* – dann heißt es schlicht: „Class not found!" Also beugen wir uns alle brav dem total unnötigen Package-Wechsel (ausgelöst durch den schieren Unwillen der Eclipse Foundation, Irrsinnssummen für das Vorhandensein der Wortmarke „Java" im Package-Präfix javax.* an Oracle abzudrücken) und investieren Tage bis Wochen in Fleißarbeit (nein, keine IDE kann das vollautomatisch, ohne dabei nette Fehler zu generieren, die man erst im Testlauf bemerkt – mehr dazu in diesem Live-Hack von Ivar Grimstad *[1]*). „Wenn's schee macht!" *[2]*, um hier mal eine große, deutsche Schauspielerin zu zitieren.

Aber auch ein zweiter Aspekt sollte nicht unberücksichtigt bleiben: Jakarta EE 9 ist das letzte Release, das ohne Features daherkommt, ebenso wie das letzte, das auf Java 8 basiert. Der Plan für Jakarta EE sieht vor, dass die folgenden Releases uns endlich eine Fülle neuer, moderner Features bringen, auf die wir uns alle seit Jahren freuen, und dies auf einer modernen SE-Version aufgesetzt wird (ja stimmt, das war schon für EE 9 versprochen und wurde klammheimlich gestrichen, weil die großen Hersteller es nicht schafften, den Code auf Java SE 11 zum Laufen zu bekommen – peinlich, echt peinlich!) Das heißt, bald nutzen wir das „var"-Schlüsselwort ebenso wie vermutlich „record" und weitere Nettigkeiten, zudem moderne Garbage Collectors, hoffentlich Fibers und Value Types, und vieles mehr. Eine Verschlankung der Server, zugunsten höherer Performance und besser lesbaren Codes. Und wenn es gut läuft, packt die EF das Ganze mit einer JVM aus dem Fundus von Eclipse Adoptium alias AdoptOpenJDK zusammen. Klingt doch gut. Um Versprechungen war die EF noch nie verlegen.

Um also eine weitere, bekannte Persönlichkeit zu paraphrasieren: „Nach dem Release ist vor dem Release!" *[3]* Auch in dieser Ausgabe wieder der übliche Gruppeneinlauf: Je mehr JUG-Mitglieder nicht nur der kostenlosen Brezel wegen zu den regelmäßigen Talks kommen, sondern auch ihre Freizeit in einen aktiven Beitrag zu Jakarta EE oder einem der zertifizierten Open-Source-Produkte stecken, desto mehr gewinnen wir alle. Programmieren in Java macht Spaß und jeder, der den Inhalt der Java aktuell versteht, ist in der Lage, etwas beizusteuern. Also meldet euch bei mir *[4]*, ich zeige euch, wie Open Source funktioniert!

Referenzen

[1] Ivar Grimstad: Live-Migration einer Jakarta-EE-8-Anwendung auf Jakarta EE 9: *https://youtu.be/3ClvncBrKJw*

[2] Annemarie Wendl in einer bekannten Buttermilch-Werbung: *https://de.wikipedia.org/wiki/Annemarie_Wendl*

[3] Sepp Herberger: „Nach dem Spiel ist vor dem Spiel": *https://de.wikiquote.org/wiki/Diskussion:Sepp_Herberger*

[4] E-Mail: markus@headcrashing.eu, Twitter: @mkarg, YouTube: *https://www.youtube.com/user/headcrashing*

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.

Ariadnefaden – der Weg Richtung DevOps ist ein Labyrinth

Thomas Michael, GOD mbh

Wir wollen in die Cloud – irgendwie mit Docker und natürlich automatisiert. Daher muss man sich in Zukunft mehr um Continuous Integration/Deployment kümmern und viel Neues lernen. Daraus ergeben sich viele Fragezeichen. Wo fängt man als Java-Entwickler an? Wo liest man sich ein? Welche Technologien muss man lernen? Welche nicht? Während man früher nur Java-Code und zusätzlich ein wenig Konfigurationsdateien in XML schrieb, wird heute viel mehr von einem Entwickler erwartet. Nachfolgend werden die notwendigen Technologien und Tools vorgestellt, um unser Ziel – automatisiert in die Cloud zu kommen – umsetzen zu können (siehe Abbildung 1). Am Ende haben wir eine hochverfügbare, skalierbare Applikation in der Cloud!

Abbildung 1: Überblick über das Labyrinth (© Thomas Michael)

Status Quo

Als Java-Entwickler kennt man sein Werkzeug und Umfeld. Den Java-Code schreibt man in seiner Lieblings-IDE. Am Wegesrand finden sich relationale und nicht relationale Datenbanken, die problemlos genutzt werden können und man stolpert nicht mehr über Frameworks wie Hibernate oder Spring. In Sichtweite sind HTML, JavaScript oder diverse andere UI-Frameworks, mit denen alle Daten einfach angezeigt werden. Dieses Umfeld beherrschen wir. Wir bauen unsere Software mit einem Build-Framework wie Gradle oder Maven, beispielsweise ein Jar- oder War-File, lassen dieses bei uns lokal laufen, testen und debuggen, beheben Fehler und am Ende stellen wir den Code einem Versionsverwaltungstool wie Git oder SVN (Apache Subversion) zur Verfügung. Im Anschluss läuft alles erneut in einer Continuous-Integration- beziehungsweise -Deployment-Umgebung und unser Code landet irgendwie in der Produktion. Für das „Wie" ist man als Java-Entwickler nur selten verantwortlich. Wie in *Abbildung 2* illustriert, übergibt man sein Artefakt an die Kollegen aus der Infrastruktur- oder Systemadministration, kümmert sich um das nächste Feature oder um den nächsten Bug und kommt aus diesem Kreislauf nicht heraus.

Vor dem Labyrinth

Einen Server zu betreiben bedeutet Arbeit: Sicherheitslücken, Patchen, Netzwerkfreigaben, Updates, Ausfallsicherheit, IP-Adressen zuweisen. Routing-Tabellen, Backup, Monitoring und vieles mehr. Und das Artefakt muss auch noch laufen! Will man das als Java-Entwickler auch noch alles lernen und machen?

Deutlich einfacher ist es, dafür einen Cloud-Anbieter zu nehmen. Mit CloudFoundry *[1]* und Heroku *[2]* gibt es unter anderem zwei PaaS-Anbieter (Platform-as-a-Service). Sie stellen eine Plattform zur Verfügung, um „einfach" nur das Jar oder einen Container, wie Docker, mit der App laufen zu lassen.

Mit Azure von Microsoft *[3]* oder AWS von Amazon *[4]* gibt es Anbieter, die noch mehr bereitstellen, als nur das Jar oder den Container laufen zu lassen. Dort kann man ganze Netzwerke und die unterschiedlichsten Services buchen, Datenbanken erstellen und auch seine Applikationen laufen lassen.

Nachfolgende Beispiele beziehen sich meist auf Docker oder die Cloud von AWS.

Am Ende des Labyrinths

Am Ausgang des Labyrinths erwartet uns eine Anwendung für die Cloud. Die Applikation deployt automatisiert, ist hochverfügbar und skalierbar. Wie sieht der Weg durch das Labyrinth aus? Was ist unser Ariadnefaden?

Mit einem neuen AWS-Konto *[4]* bekommt man für ein Jahr zum Ausprobieren ein kostenloses Kontingent, um seine Anwendung laufen zu lassen. Java-Anwendungen können direkt als Jar im Service Elastic Beanstalk *[5]* laufen. Docker-Container können sowohl im Service Elastic Beanstalk als auch in diversen AWS-Container-Services wie Fargate oder ECS laufen.

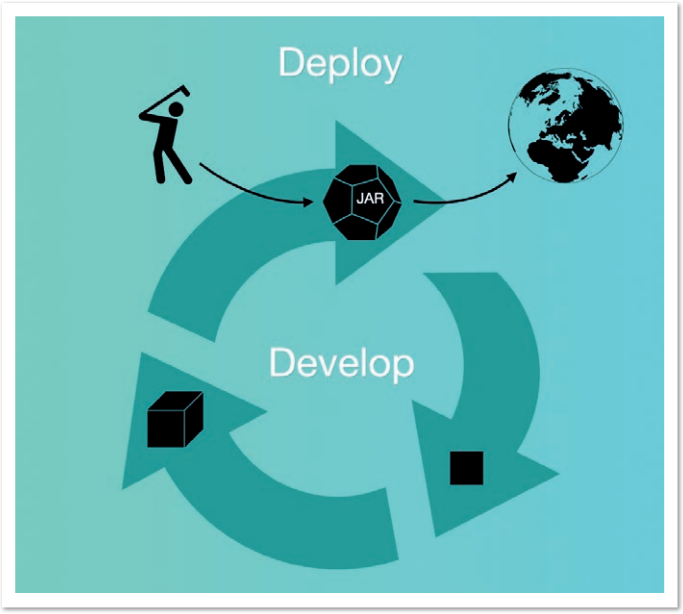


Abbildung 2: Kreislauf der Entwicklung (© Thomas Michael)

Der Mitspieler

Als konkrete Anwendung nehmen wir eine Spring-Boot-Applikation mit einer SQL-Datenbank. Diese Applikation soll in der Cloud laufen, automatisch deployen, hochverfügbar sein und bei Bedarf skalieren können. Auf dem Weg zu diesem Ziel werden DevOps-Methoden nötig sein.

Zur Automatisierung in AWS wird der Service CloudFormation *[6]* genutzt. Eine gern genutzte Alternative ist Terraform *[7]*, da mit der Terraform-Cloud unabhängig gearbeitet werden kann. Terraform ist nicht Bestandteil dieses Labyrinths, aber eine dringende Leseempfehlung!

Schritt für Schritt durch das Labyrinth

Eine Applikation sollte für den Anwender immer erreichbar sein, da er sie sonst nicht nutzen kann. Hat die Applikation zu viele Anwender, werden mehr Instanzen benötigt. Kurz: Eine Anwendung muss hochverfügbar sein und skalieren können.

Und wie wird eine Anwendung ausgeliefert? Wo stehen die Server und wie werden diese konfiguriert? Was muss man dafür beherrschen? Glücklicherweise haben sich darüber schon andere Gedanken gemacht. Eine Applikation sollte dem 12-Faktoren-Prinzip *[8]* entsprechen, dann ist sie „cloud ready" *[9]*, wie es CloudFoundry ausdrückt. Diese Prinzipien sind der Ariadnefaden, der uns durch das Labyrinth führt. Ein erster Überblick über diese Prinzipien:

1. Eine gemeinsame Codebasis (zum Beispiel Git)
2. Abhängigkeiten sind explizit deklariert (etwa über Gradle oder Maven)
3. Die Konfiguration ist in Umgebungsvariablen abgelegt
4. Unterstützende Dienste wie eine Datenbank sind als Ressourcen hinzu konfiguriert und damit leicht austauschbar
5. Strikte Trennung von der Build-, Release- und Run-Phase (beispielsweise ein Jar bauen und dann kombiniert mit der Konfiguration für die Test-Stage starten)
6. Die App wird als ein eigener Prozess ausgeführt
7. Dienste werden über Portbindung exportiert
8. Skalierbarkeit durch Nebenläufigkeit
9. Starten und Stoppen der App ist problemlos und möglich schnell
10. Möglichst gleiche Applikation in den Umgebungen DEV, Test und Produktion durch Nutzung einer Continuos Deployment Pipeline
11. Die Logs werden rausgestreamt
12. Admin- und Managementvorgänge sind einmalige Vorgänge

```
FROM java:8
WORKDIR /
ADD target/my-app-0.1.0.jar myApplication.jar
EXPOSE 8080
CMD java - jar myApplication.jar
```

Listing 1: Inhalt eines Dockerfile, um ein Jar im Container zu haben

```
docker build . -t somename
```

Listing 2: Bauen des Dockerfile

```
docker run somename
```

Listing 3: Starten des Docker-Containers

Aus den Prinzipien ergeben sich weitere Fragen und erste Anpassungen. Wie werden die Prinzipien angewandt? Vieles ist für uns Entwickler selbstverständlich, wie eine gemeinsame Codebasis (Prinzip 1) oder dass ein Buildtool wie Maven die externen Abhängigkeiten auflöst (Prinzip 2) und am Ende eine Pipeline das Artefakt baut (Prinzip 10). Ein zukünftiger DevOps muss allerdings größer denken.

Gehen wir Schritt für Schritt, Prinzip für Prinzip vor.

Schritt 1 (Prinzip 1)

In den meisten Projekten ist eine Versionsverwaltung bereits vorhanden. Ansonsten kann auf GitHub *[10]* oder bei AWS unter CodeDeploy *[11]* sehr einfach ein Git-Repository angelegt werden. Eigene Git-Kenntnisse können über Gitkatas *[12]* vertieft werden.

Ein Repository ist später Grundvoraussetzung für eine automatisierte Pipeline.

Schritt 2 (Prinzip 2)

Der Einsatz eines Build-Tools wie Maven oder Gradle garantiert, dass die Anwendung immer gleich gebaut wird, unabhängig vom Entwicklungs-PC. Damit erhalten wir ein reproduzier- und versionierbares Artefakt.

Als DevOps sind wir aber auch für die Laufzeitumgebung zuständig. Dank Container-Tools wie Docker können wir beschreiben, welche Software auf der Laufzeitumgebung sein soll, und damit ist auch die Laufzeitumgebung ein reproduzier- und versionierbares Artefakt. Als Beispiel ist in *Listing 1* ein Docker-Container im Dockerfile beschrieben, der eine Java-8-Laufzeitumgebung bekommt, dann mit dem Befehl `ADD` aus dem `target`-Order das gebaute Artefakt in den Container kopiert und es in myApplication.jar umbenennt. Dieses wird am Ende gestartet. Da die Applikation auf dem Port 8080 läuft, wird dieser Port nach außen gegeben. Mehr dazu unter Schritt 7 (Prinzip 7). Mit *Listing 2* wird der Container gebaut und mit *Listing 3* kann er separat gestartet werden.

Schritt 3 (Prinzip 3)

Die Konfiguration ist in Umgebungsvariablen abgelegt, sodass die Maschine, auf der die Applikation läuft, ihre eigene Konfiguration in die Anwendung geben kann. Ein Server, egal ob virtuell oder echt, weiß anhand seiner Umgebungsvariablen, welche Stage er repräsentiert. Mit *Listing 4* kann man auf einem Linux-System manuell eine Umgebungsvariable anlegen. Für Docker kann man das im Dockerfile wie in *Listing 5* machen.

Schritt 4 (Prinzip 4)

Die Datenbank zum Testen kann lokal vorliegen, kann aber auch als Docker-Container definiert werden. Dies hat den Vorteil, dass ein Testsystem dann Anwendung und Datenbank baut. Um die Datenbank aktuell zu halten, sind Tools wie Flyway *[13]* oder Liquibase *[14]* mehr als sinnvoll, um auch das Datenbankschema versioniert zu haben.

Wichtig ist, dass der Code die Dienste nicht direkt nutzt, sondern über die Umgebungsvariablen einbindet oder über von außen konfigurierte Ressourcen.

Schritt 5 (Prinzip 5)

Dieser Schritt profitiert von den vorangegangenen Schritten!

- Das Jar ist gebaut
- Die Ressourcen, wie die Datenbank, werden über Umgebungsvariablen eingelesen

In Spring Boot gibt es den sehr einfachen Weg, über Profile die unterschiedlichen Stages in einem Profil abzubilden. Dann ist es die Aufgabe der jeweiligen Stage, das richtige Profil zu nutzen. Noch flexibler ist man, wenn man für die Cloud ein allgemeines, von der Stage unabhängiges Cloud-Profil erstellt. Dieses Profil nimmt die Werte aus den Umgebungsvariablen, wie in *Abbildung 3* dargestellt, an. Damit konfiguriert die Laufzeitumgebung die Anwendung!

```
export DB_URL="jdbc:mysql://localhost:3306/myapp"
```

Listing 4: Festlegen der URL für die Datenbank als Umgebungsvariable

```
FROM java:8
WORKDIR /
ADD target/my-app-0.1.0.jar myApplication.jar
ENV DB_DRIVER=com.mysql.jdbc.Driver
ENV DB_URL=jdbc:mysql://localhost:3306/myapp
ENV DB_USERNAME=username
ENV DB_PASSWORD=password
EXPOSE 8080
CMD java - jar myApplication.jar
```

Listing 5: Im Dockerfile fest die Umgebungsvariablen hinterlegen.

```
spring:
  profiles: cloud
  datasource:
    driverClassName: ${DB_DRIVER}
    url: ${DB_URL}
    username: ${DB_USERNAME}
    password: ${DB_PASSWORD}
  jpa:
    hibernate.ddl-auto: update
```

Abbildung 3: Ausschnitt aus der application.yaml (© Thomas Michael)

```
1  AWSTemplateFormatVersion: '2010-09-09'
2  Transform: AWS::Serverless-2016-10-31
3  # Parameters for application
4  Parameters:
5    myDatabaseUrl:
6      Type: String
7      Description: url for connection to db
```

Abbildung 4: Eingangsparameter eines Cloudformation-Templates für die Datenbank-URL (© Thomas Michael)

```
Resources:
  myJavaApp:
    Type: AWS::ElasticBeanstalk::Application
  myJavaAppTemplate:
    Type: AWS::ElasticBeanstalk::ConfigurationTemplate
    Properties:
      ApplicationName:
        Ref: myApp
      Description: AWS ElasticBeanstalk Sample Configuration Template
      OptionSettings:
      - Namespace: 'aws:elasticbeanstalk:application:environment'
        OptionName: DB_URL
        Value: !Ref myDatabaseUrl
      - Namespace: aws:autoscaling:asg
        OptionName: MinSize
        Value: '2'
      - Namespace: aws:autoscaling:asg
        OptionName: MaxSize
        Value: '6'
      - Namespace: aws:elasticbeanstalk:environment
        OptionName: EnvironmentType
        Value: LoadBalanced
```

Abbildung 5: Nutzung des Eingabeparameters im Cloudformation-Template (© Thomas Michael)

Um in der Cloud zu laufen, kann mit AWS eine Elastic-Beanstalk-Applikation definiert werden. Dabei werden die Umgebungsvariablen, wie im Beispiel in *Abbildung 4*, in Abhängigkeit von einem Eingabeparameter gesetzt. Das Cloudformation-Template erwartet, dass der Benutzer den Wert „myDatabaseUrl" bei der Benutzung des Templates mit angibt. In *Abbildung 5* wird der Wert referenziert und an die Umgebungsvariable `DB_URL` gebunden. So kann dieses Template für unterschiedliche Stages genutzt werden. Das Artefakt, in diesem Fall ein Cloudformation-Template, wird nur einmal gebaut, heißt hier „Package" und wird in allen Stufen wiederverwendet.

Wie kommen die Umgebungsvariablen in den Docker-Container? Sie fest im Dockerfile zu hinterlegen ist keine dynamische Lösung. Für Docker kann man beim Starten das Environment mit angeben, wie in *Listing 6* zu sehen.

Schritt 6 (Prinzip 6)

Docker-Container laufen in einem eigenen Prozess und können schnell gestartet und gestoppt werden, Jars ebenso.

Mit Kubernetes lassen sich mehrere Container starten, stoppen und orchestrieren. Auch dort ist die Konfiguration wieder in YAML. Empfehlung: Auf der Webseite *[16]* die interaktiven Tutorials zu Kubernetes durcharbeiten.

Hilfreiche Linux-Befehle für das Beobachten und Stoppen von Prozessen sind zum Beispiel `ps`, `top` und `kill`.

Schritt 7 (Prinzip 7)

Eine gestartete Webanwendung läuft auf einem Port, beispielsweise dem Port 8080. Lokal gestartet ist sie dort erreichbar über http(s)://localhost:8080/. Damit die gleiche Anwendung, gestartet im Docker-Container, erreichbar ist, muss dem Container mitgeteilt werden, welcher Port von außen erreichbar sein soll. Dies macht man im Dockerfile *(siehe Listing 7)*.

Beim Starten des Containers kann dieser Port an einen Port der eigenen Maschine gebunden werden, in *Listing 8* wird der Port 80 direkt an den Port 8080 des Docker-Image gebunden. Dazu sollte man die Liste *[15]* der frei verfügbaren und standardisierten Ports kennen.

Durch die Portbindung kann man mehrere Container mit dem Jar starten, sie an unterschiedliche Ports binden und sich selbst einen eigenen, simplen Load Balancer davor bauen.

Oder man nutzt die Möglichkeiten, die die Cloud mit Elastic Load Balancers *[16]* bietet, und stellt fest, dass es Load Balancer für unterschiedliche Bereiche gibt. Damit sind wir auf dem Weg zu einer hochverfügbaren Anwendung, die die Lasten verteilt.

Ein hilfreiches Tool unter Linux: `lsof` – List open files. Mit *Listing 9* kann man sehen, ob der verwendete Port bereits verwendet wird. Oft startet eine Anwendung nicht, wenn der Port bereits belegt ist.

```
docker run -e  DB_DRIVER=com.mysql.jdbc.Driver
058afd8971d1
```

Listing 6: Starten eines Docker-Image mit Setzen einer Umgebungsvariablen

```
EXPOSE 8080
```

Listing 7: Snippet aus Dockerfile, Portfreigabe des Docker-Containers

```
docker run -p 80:8080
```

Listing 8: Starten eines Docker-Containers mit Portbindung

```
lsof -i :8080
```

Listing 9: Verwendung von lsof

Schritt 8 (Prinzip 8)

Gute Beispiele für die Umsetzung von Skalierbarkeit sind zum einen Kubernetes oder AWS mit Elastic Beanstalk.

In AWS definiert man in CloudFormation eine AWS-Elastic-Beanstalk-Applikation und definiert gleich mit, wie viele Instanzen der Applikation minimal (2) und maximal (6) vorhanden sein sollen *(siehe Abbildung 5)*.

Für einen allerersten Einblick ist dies ausreichend. Wer es beliebig komplex möchte, mit unterschiedlichen Health-Checks, Speicherauslastungen und unterschiedlichen Load Balancers, kann dies für AWS *[16]* nachlesen.

Schritt 9 (Prinzip 9)

Wenn alle Prinzipien eingehalten sind, ist dieser Punkt erfüllt.

Schritt 10 (Prinzip 10)

In den vorherigen Schritten wurde die Basis dafür gelegt, dass die Applikation nur einmal gebaut wird, per Docker oder CloudFormation verpackt und über die Umgebungsvariablen seine Stage und damit seine Konfiguration erkennt. In *Listing 10* wird ein CloudFormation Template mit Package gebaut. Alle dazugehörigen Ressourcen werden in AWS in einem S3-Bucket gespeichert. S3 steht für „Simple Storage Service". Zum Verständnis reicht es, dass man sich einen S3-Bucket wie einen riesigen Datenspeicher vorstellt.

In *Abbildung 6* ist zu sehen, wie eine Pipeline dazu aussieht. Der Entwickler pusht seinen Code ins Repository. Dann startet die Pipeline, baut den Java-Code und das CloudFormation-Template dazu. Danach deployt die Pipeline die Anwendung auf dem Testsystem mit

```
aws cloudformation package --template-file elasticBeanstalk.yaml --output-template-file output.yaml --s3-bucket
$templateBucketName
```

Listing 10: Bauen des ElasticBeanstalk-CloudFormation-Templates

```
aws cloudformation deploy --template-file output.yaml --stack-name myFirstApp --parameter-overrides
myDatabaseUrl=jdbc:mysql://qsserver/myapp stage=qs
```

Listing 11: Deployen des vorher gebauten Templates auf der QS-Umgebung

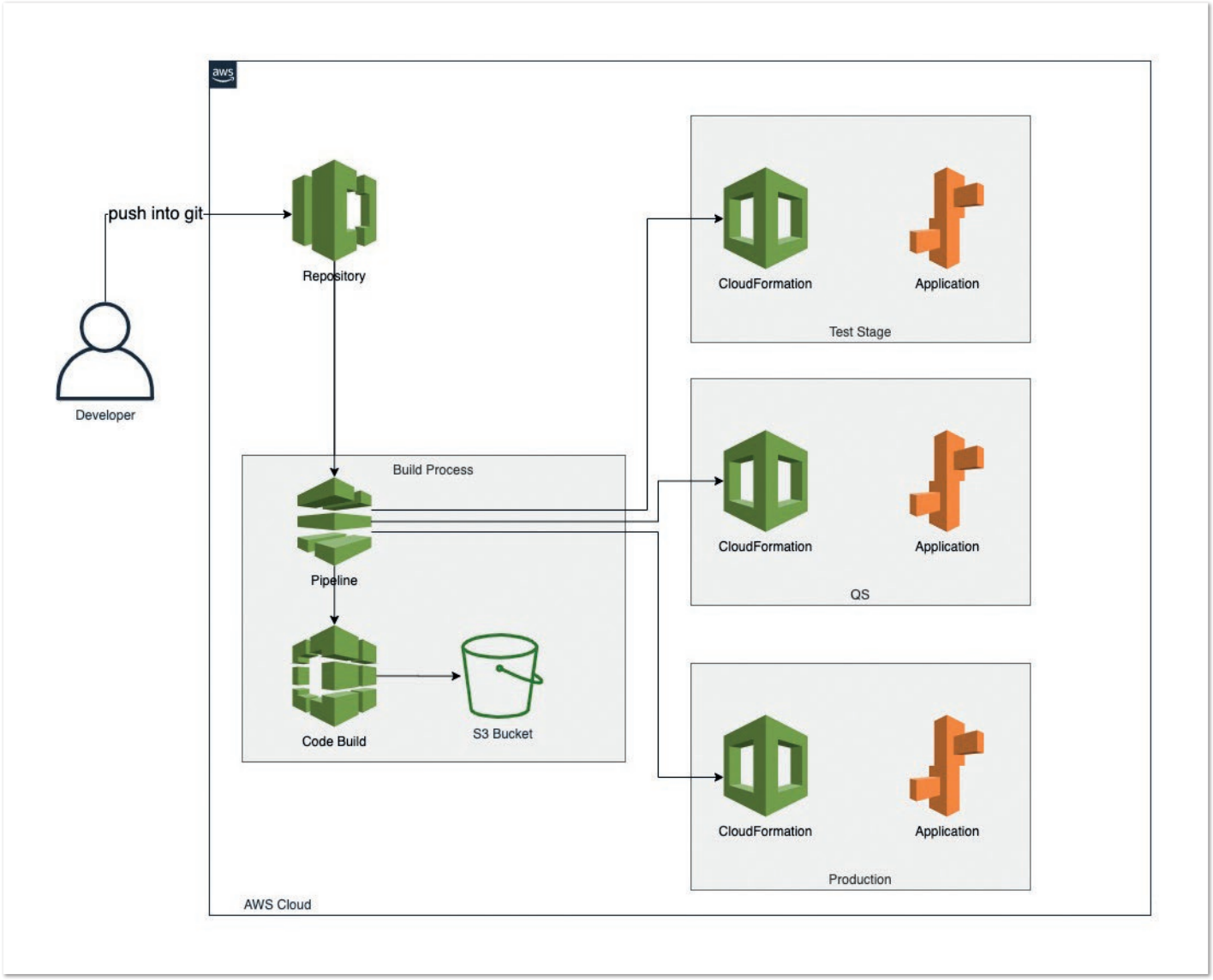


Abbildung 6: Anzeige der Stages für Test, QS und Produktion (© Thomas Michael)

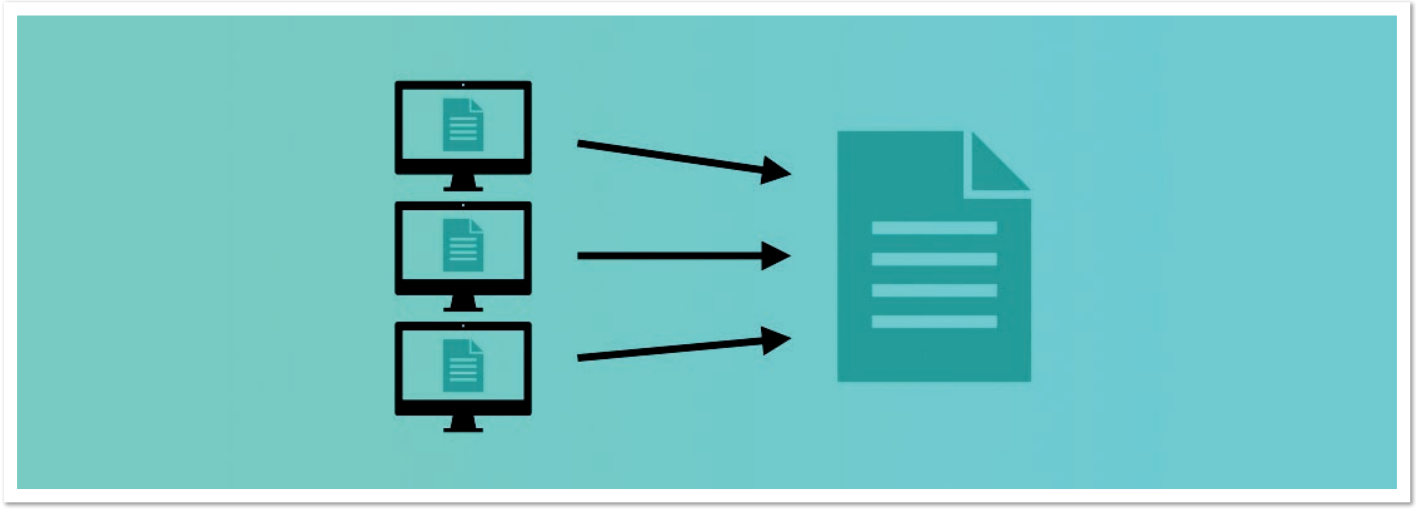


Abbildung 7: Logs von mehreren Servern werden zentral gesammelt (© Michael Thomas)

der Testkonfiguration und so weiter. Die Pipeline ist sehr vereinfacht dargestellt. Dies könnte beispielsweise ein Jenkins, ein Bamboo oder die AWS-eigene Code-Pipeline sein.

Schritt 11 (Prinzip 11)

Aufgrund von Hochverfügbarkeit und Skalierbarkeit läuft die Anwendung auf mehreren Instanzen. Jede Instanz schreibt ihre eigenen Logfiles. Fehlersuche ist unnötig schwer. Daher ist es das Ziel, alle Logs zentral einzusammeln, wie an *Abbildung 7* skizziert.

Für Docker mit Logging Drivers *[17]* oder in AWS mit CloudWatch *[18]* lassen sich die Logs aggregieren und analysieren.

Hilfreiches Linux-Wissen an dieser Stelle sind die nachfolgenden Kommandozeilentools:

- `scp`, um Daten auf einen entfernten Server zu kopieren
- `ssh`, um sich auf einem entfernten Server einzuloggen
- `less` oder `more`, um sich (Log)-Dateien anzugucken
- `grep`, um Dateien mit gesuchten Inhalten zu finden

Schritt 12 (Prinzip 12)

Dieses Prinzip bedeutet, dass auch einmalige Vorgänge, wie beispielsweise eine Migration, auch Teil der Anwendung sind. Damit sind sie auch Teil der Versionierung und man muss sich darum kümmern, dass sie automatisiert und einmalig auf jeder Stage laufen.

Ein Beispiel dafür ist eine Datenbank-Migration. Durch Tools wie Flyway und Liquibase kann die Datenbankmigration automatisiert werden. Beim Starten der Anwendung wird überprüft, ob die Datenbank auf dem richtigen Stand ist; falls nicht, werden entsprechende Skripte ausgeführt.

Fazit

Mit dem Einhalten der 12-Faktor-Prinzipien lernt man die notwendigen DevOps-Methoden, um eine Anwendung bereit für die Cloud zu machen. Natürlich ist das DevOps-Feld noch deutlich größer, aber dies ist ein Anfang, auf dem aufgebaut werden kann, Schritt für Schritt.

Quellen

[1] *https://docs.cloudfoundry.org/devguide/deploy-apps/prepare-to-deploy.html*
[2] *https://www.heroku.com*
[3] *https://azure.microsoft.com/de-de/*
[4] *https://portal.aws.amazon.com/billing/signup#/start*
[5] *https://aws.amazon.com/de/elasticbeanstalk/*
[6] *https://docs.aws.amazon.com/de_de/AWSCloudFormation/latest/UserGuide/Welcome.html*
[7] *https://www.terraform.io*
[8] Hintergrund zu 12-Faktor-App: *https://12factor.net. Adam Wigging (2017)*
[9] *https://docs.cloudfoundry.org/devguide/deploy-apps/deploy-app.html*
[10] *https://github.com*
[11] *https://eu-central-1.console.aws.amazon.com/codesuite/codedeploy/start*
[12] *https://github.com/schauder/gitkata*
[13] *https://flywaydb.org*
[14] *https://www.liquibase.org*
[15] *https://de.wikipedia.org/wiki/Liste_der_standardisierten_Ports*
[16] *https://docs.aws.amazon.com/de_de/elasticbeanstalk/latest/dg/using-features.managing.elb.html*
[17] *https://docs.docker.com/config/containers/logging/configure/*
[18] *https://docs.aws.amazon.com/cloudwatch/index.html*

Thomas Michael
GOD mbH
Thomas.Michael@god.de

Thomas Michael ist Softwareentwickler aus Leidenschaft. Nach dem Informatikstudium widmete er sich ganz der Software-Entwicklung in Braunschweig und hat damit sein Hobby zum Beruf gemacht. Nach dem ersten Kontakt mit der Cloud lassen ihn die Möglichkeiten nicht mehr los. Als AWS Developer arbeitet er bei der GOD mbH in Braunschweig an der erfolgreichen Umsetzung von Projekten mit Cloud-Technologien für interne und externe Kunden. Privat arbeitet er an kleineren Springboot- oder Angular-Anwendungen, die immer öfter durch serverless Lambda ersetzt werden.

Kubernetes – UX und Optionen für Java-Entwickler

Matthias Häußler, Novatec Consulting GmbH

Kubernetes hat sich in den letzten Jahren als der Standard für Container-Orchestrierung etabliert und ist aus der Entwicklung und insbesondere dem Betrieb von modernen, verteilten und Container-basierten Anwendungen nur noch schwer wegzudenken. In diesem Artikel versuche ich zu beschreiben, welche zusätzlichen Kenntnisse zum Deployment und Betrieb von Anwendungen auf Kubernetes notwendig sind, und vergleiche diese mit den verwandten Open-Source-Technologien Cloud Foundry und Knative.

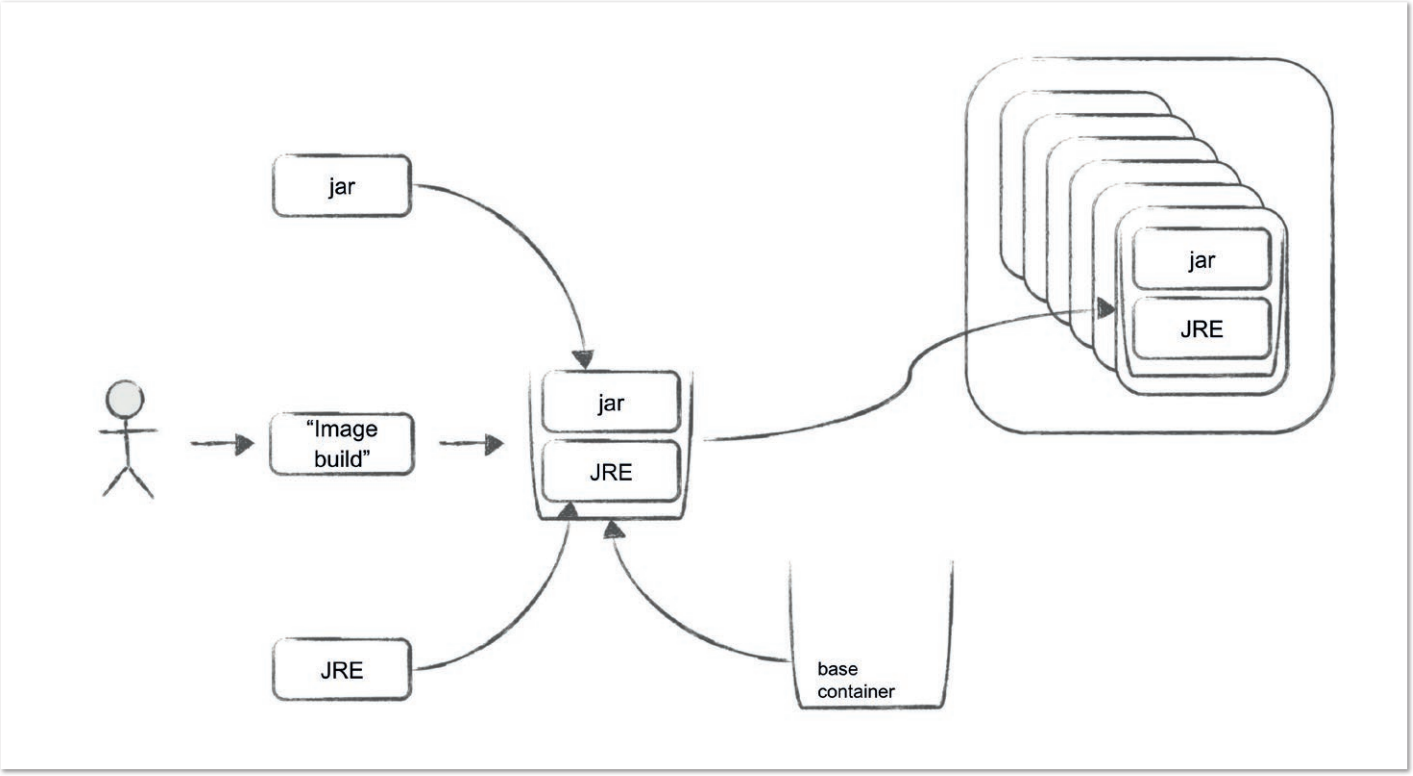


Abbildung 1: Zusammenspiel Container, Image, Build und Registry (© Matthias Häußler)

In der sogenannten Cloud-nativen Welt ist Kubernetes omnipräsent. Es ist sowohl als lokale On-Premises-Lösung einsetzbar als auch bei allen namhaften Cloud-Providern als Managed Service leicht verfügbar. Manuelle sowie automatische Skalierung von Workloads, Load Balancing, Service Discovery und dynamische Updates einer Anwendung ohne Unterbrechung der Verfügbarkeit gehören zum kleinen Einmaleins der Technologie. Weiterhin bietet Kubernetes einfache Möglichkeiten für benutzerdefinierte Erweiterungen. Es ist das zentrale Projekt der Cloud Native Computing Foundation (CNCF) und gibt sehr viel Nährboden für ein ganzes Ökosystem an weiteren zusammenhängenden Technologien *[1]*.

Im Gegenzug sieht es sich oft Kritik ausgesetzt: Kubernetes sei nicht einfach in seiner Handhabung, insbesondere die initiale Lernkurve sei steil. Durch die weite Verbreitung haben sich daher auch die Anforderungen diesbezüglich an Entwickler verändert und vergrößert. Wo früher das theoretische Ende der Verantwortlichkeit beim Build und Test des Codes lag, wird heutzutage zumindest ein Grundverständnis von Dingen wie Dockerfile-Syntax, Pods, Ingress, ConfigMaps etc. erhofft und sogar erwartet.

In den nachfolgenden Abschnitten ist eine Beschreibung der Kubernetes-Technologie zur Referenz aufgezeigt. Im Anschluss werden die Technologien Cloud Foundry/Knative dazu in Relation gesetzt.

Kubernetes (und Docker)

Im folgenden Abschnitt wird dargestellt, welche Schritte notwendig sind, um den Quelltext einer Anwendung vom Repository in die Laufzeit von Kubernetes zu bringen. Der Vorgang orientiert sich an einer einfachen Spring-Boot-Web-Anwendung.

Der erste Schritt, der in jedem Fall ausgeführt werden muss, ist das Erstellen eines Container-Image mit der darin kompilierten beziehungsweise lauffähigen Anwendung. Kubernetes ist per se „anwendungsagnostisch", das heißt, das Eintrittsticket zur Plattform ist ein Container-Image. Was genau darin enthalten ist, ist aus Sicht der Plattform erst einmal nicht relevant.

Die traditionelle Vorgehensweise ist hier zuerst ein Maven- oder Gradle-Build einer jar-Datei, das dann in einem Dockerfile referenziert und in ein Container-Image verpackt wird. Das Image besteht abschließend aus einer Basis-OS-Schicht, einer Java Runtime und der jar-Datei *(siehe Abbildung 1)*. Es gibt auch Alternativen, die diese Schritte kombinieren und automatisieren können. Beispielsweise Maven-/Gradle-Plug-ins wie fabric8 *docker-maven-plugin* *[2]* oder Spring Boot ab Version 2.3 mit Integration der Paketo-Technologie *[3]*.

Es bleibt festzuhalten, dass Schritte und Entscheidungen notwendig sind, bevor überhaupt mit Kubernetes begonnen werden kann, und es seitens Kubernetes keine Vorgabe für eine Build- sowie Container-Strategie gibt.

Sobald das Image gebaut ist, kann es über eine Image Registry wie zum Beispiel dem Docker Hub *[4]* einer Kubernetes-Umgebung zur Verfügung gestellt werden.

Um Kubernetes zu nutzen, benötigt es ein gewisses Verständnis der Basis-Artefakte. Zu Beginn sei hier der Pod erwähnt. Ein *Pod* ist in Kubernetes die kleinste einsetzbare Einheit. Abgeleitet aus der Tierwelt, indem ein Docker-Container gern als Wal darstellt dargestellt wird, ist ein Pod eine Herde von Walen. Sinnbildlich bedeutet es in Kubernetes ein Konstrukt von einem bis n Containern.

Wichtig ist hierbei zu wissen, dass Kubernetes sämtliche Container innerhalb eines Pod nur als Ganzes steuern kann, das heißt gemeinsam

starten und stoppen. Es können daher beispielsweise keine Container innerhalb eines Pod unabhängig voneinander behandelt oder skaliert werden. Ein Pod benötigt daher immer die genaue Information über die Container Images und Registry *(siehe Abbildung 2)*.

Die Anzahl der skalierten Pods wird durch sogenannte *Sets* konfiguriert. Die einfachste Variante ist ein *ReplicaSet* für die Skalierung von zustandslosen Anwendungscontainer. Ein *StatefulSet* ermöglicht die Skalierung von Anwendungen mit Zustand in einer Form, dass der Zustand nicht in den Pods, sondern via Set zentral verwaltet und von allen zugehörigen Instanzen geteilt wird. Ein *DaemonSet* stellt sicher, dass eine Pod-Instanz auf jedem physischen Knoten des Clusters am Laufen sein soll.

ReplicaSets haben als übergeordnetes Objekt ein Deployment, das deklarativ Updates von ReplicaSets und damit auch Pods beschreibt *(siehe Abbildung 3)*. Wenn eine neue Image-Version für einen Container vorliegt oder es neue Variablen zur Konfiguration gibt, wird das Deployment ein neues ReplicaSet erstellen. Es wird versuchen, nahtlos die Instanzen im alten ReplicaSet herunterzufahren und die neu konfigurierten Instanzen im neuen Set zu starten. Das Deployment verwaltet also den Lebenszyklus der Anwendung über verschiedene Versionen hinweg *(siehe Abbildung 4)*.

Schließlich soll die Funktionalität einer deployten Anwendung auch entweder innerhalb des Clusters für andere Anwendungen oder auch extern verfügbar gemacht werden. Für diesen Zweck gibt es

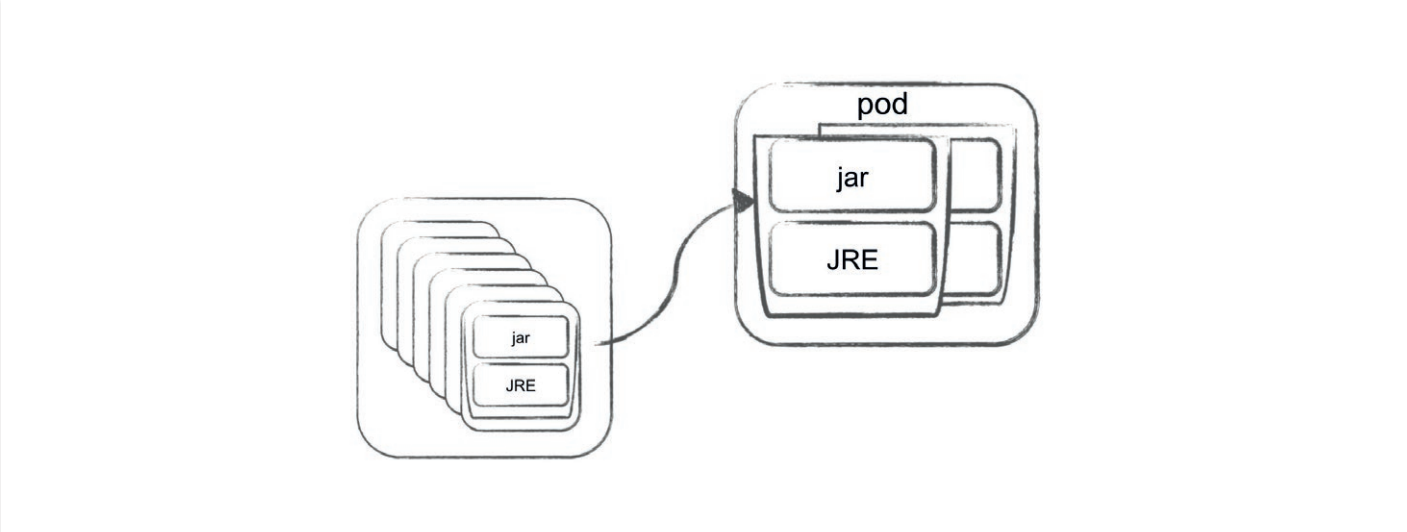


Abbildung 2: Pod referenziert Images aus der Container Registry (© Matthias Häußler)

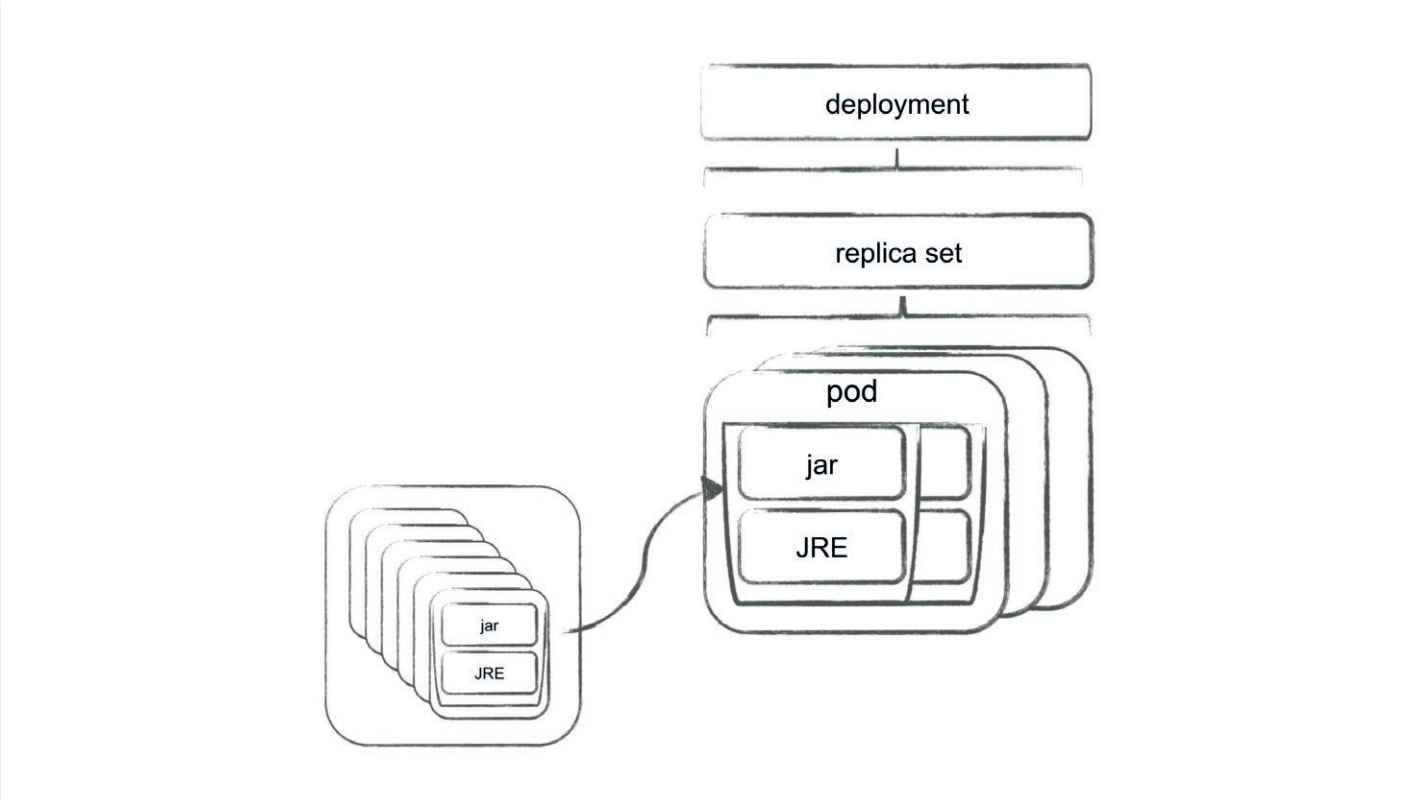


Abbildung 3: Deployment – ReplicaSet – Pod (© Matthias Häußler)

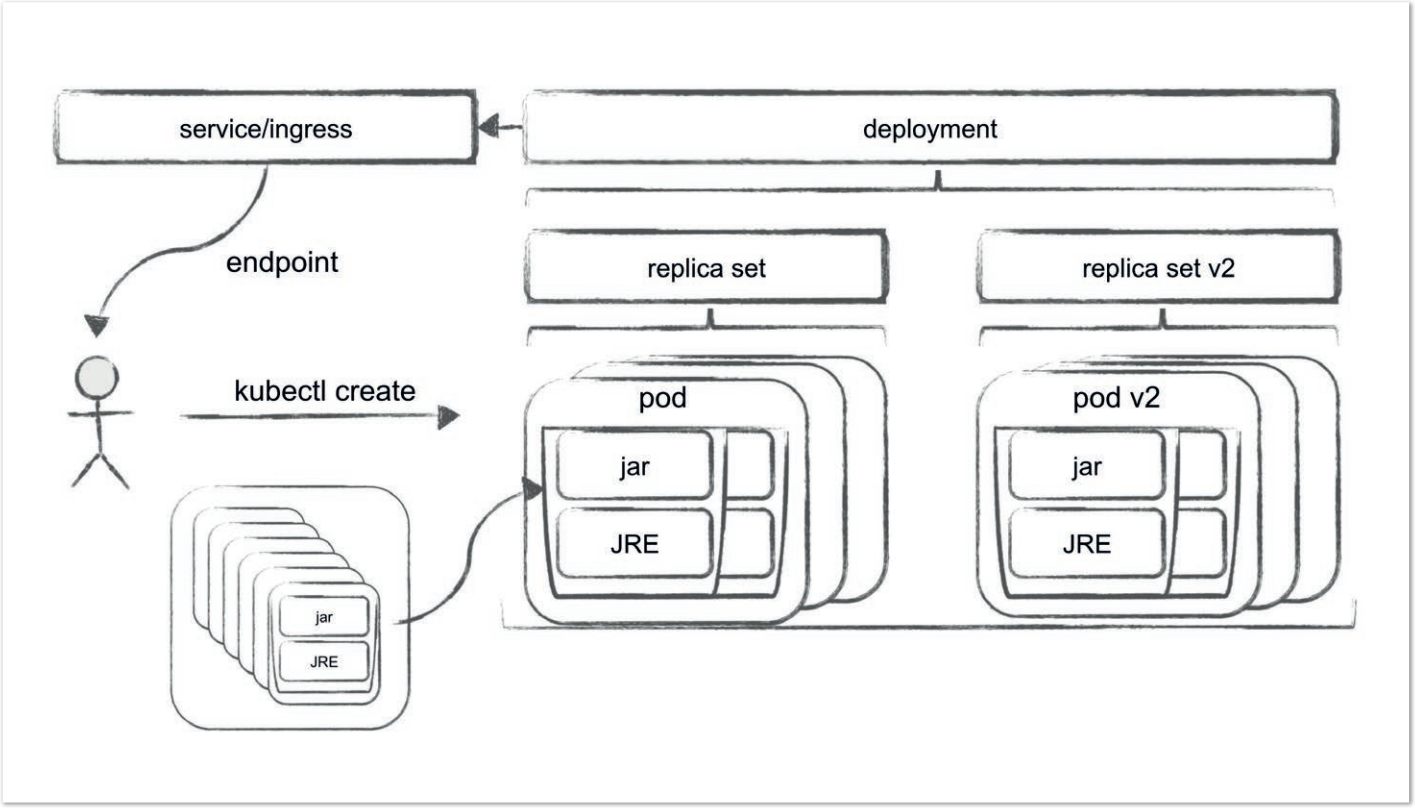


Abbildung 4: Service/Ingress – Deployment – ReplicaSet – Pod (© Matthias Häußler)

sogenannte Services von unterschiedlicher Klassifizierung. Eine *ClusterIP* gibt einem Deployment eine interne IP und DNS-Namen; sie dient somit zur Kommunikation von Komponenten innerhalb des Clusters. Für externen Zugriff gibt es die Varianten *NodePort* und *LoadBalancer*.

Zusätzlich gibt es die Möglichkeit, einen Ingress zu verwenden. Das ist ein API-Objekt, das ebenfalls Netzwerkverkehr extern verwaltet und von dort auch via URL Zugriff bietet *(siehe Abbildung 4)*.

Wie im vergangenen Abschnitt (wahrscheinlich) gut erkennbar, ist es schwierig, das Basis-Konzept von Kubernetes kurz zusammenzufassen. Zwei Dinge werden daher offensichtlich: Zum einen braucht es für die korrekte Verwendung der Technologie ein gewisses Grundwissen. Kubernetes ist eine Automatisierungsplattform. Falsche Konfiguration bedeutet hier leider auch, dass falsch automatisiert wird. Zum anderen sieht man auch am Klang der Namen, dass der Fokus hier sehr stark auf Infrastruktur und nicht auf der Anwendung liegt. Genau genommen gibt es kein Artefakt oder Konfigurationsobjekt, das einfach nur „Application" heißt.

Grundsätzlich weiß Kubernetes über die Container erst einmal nur, ob sie laufen oder eben nicht, aber nicht, in welchem Zustand die Anwendung innerhalb ist. Eine genauere Kenntnis über die Anwendung verschaffen sogenannte „Probes". In einer *ReadinessProbe* wird beschrieben, welcher Zustand eintreten muss, damit ein Workload als „ready" deklariert wird und Kubernetes anfängt, Netzwerklast darauf zu lenken. Für das Beispiel einer Spring-Boot-Anwendung mit Actuator-Komponente *[5]* würde das wie in *Listing 1* gezeigt aussehen.

```
readinessProbe:
  httpGet:
    path: /actuator/health
    port: 8080
    …
```

Listing 1: ReadinessProbe (gekürzt)

In einer vereinfachten Variante dargestellt bedeutet dies, dass die Anwendung erst dann als „ready" gilt, wenn der Endpunkt `/actuator/health` unter Port 8080 einen Status 200 der HTTP-Abfrage liefert.

Dieses Beispiel finde ich sehr sinnbildlich für den Umgang mit Kubernetes. Die Technologie ist sehr mächtig und lässt wenig Wünsche beim funktionalen Umfang offen. Es gibt auch sehr wenig, was nicht konfigurierbar ist. Allerdings braucht es umfassende Kenntnis, um das alles richtig anwenden zu können.

Cloud Foundry

Cloud Foundry hat im Vergleich zu Kubernetes den Anspruch, eine PaaS-Lösung („Platform as a Service") zu sein, die oberhalb der Container-Abstraktionsschicht agiert und damit Fokus auf die Anwendungsebene bietet. Für Entwickler bedeutet das, sämtliche Container-Build- und Betriebsaktivitäten liegen außerhalb ihrer Verantwortung. Diese Aufgaben werden von der Plattform erledigt. Was übrig bleibt, ist die Bereitstellung des Codes.

Von Beginn an hat Cloud Foundry das Ziel, nicht als reine Technologie, sondern als Plattform mit dem Fokus auf „Developer Experience" zu wirken. Mit dem Grundprinzip, dass, je weniger sich die Entwickler mit wiederholenden Infrastruktur- und Konfigurationsaufgaben auseinandersetzen müssen, desto mehr Zeit für die eigentliche Aufgabe bleibt – die Entwicklung von Anwendungsfunktionalität.

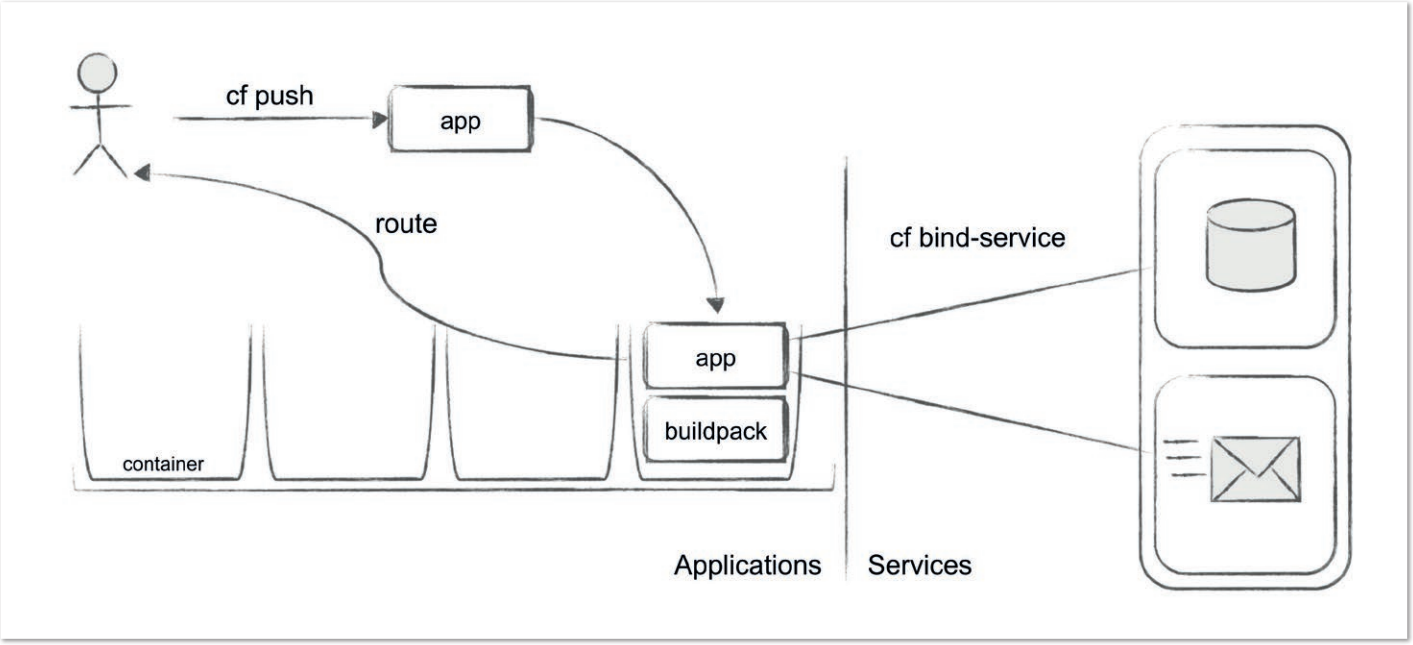


Abbildung 5: Cloud Foundry (© Matthias Häußler)

Trotz allem muss das Innenleben funktionieren. Da Cloud Foundry offiziell bereits seit 2011 existiert (und damit Jahre vor Docker und Kubernetes) hat es in dieser Zeit einige technologische Änderungen im Inneren durchlaufen, ohne dabei die externe Schnittstelle zur Entwicklung zu ändern oder Nutzungsfreundlichkeit zu opfern. Nach vielen Jahren mit eigener Technologie für Container und Orchestrierung gibt es seit Kurzem zwei Projekte, die genau dafür Docker und Kubernetes verwenden – KubeCF *[6]* und cf-for-k8s *[7]*.

Der Ansatz ist hierbei der Versuch einer losen Kopplung zwischen der Leistungsfähigkeit von Kubernetes und ebenjener „Developer Experience" von Cloud Foundry. Die weitere Beschreibung der Plattform orientiert sich auch an diesen Varianten für einen sinnvollen Vergleich.

Das Grundprinzip hat sich dabei jedoch nicht wesentlich verändert. Die Plattform orientiert sich sehr stark an einem 12-factor-Modell *[8]* mit zustandslosen Anwendungen („Apps") und zustandsbehafteten Diensten („Services"). Die Apps werden von der Plattform automatisiert und die Konfigurationsinformationen der Services dynamisch injiziert.

Um eine Anwendung zu deployen, gibt es den wahrscheinlich bekanntesten Cloud-Foundry-Befehl `cf push`, der einige Unterschritte in sich bündelt:

Der Code wird zur Plattform hochgeladen, dort wird geprüft, welche Programmiersprache oder welches Framework vorliegt. Wird dieses als kompatibel erkannt, wird ein zugehöriger sogenannter Buildpack verwendet, der als Laufzeitumgebung für den Code fungiert. Buildpacks sind ein Konzept, das ursprünglich von Heroku stammt, von Cloud Foundry adoptiert wurde und schließlich in das Cloud-Native-Buildpacks-Projekt geflossen ist *[9]*. Ein Vorteil und eine Einschränkung zugleich, die Plattform kümmert sich um eine standardisierte Umgebung, funktioniert aber auch nur, wenn passende Buildpacks vorliegen – was allerdings bei JVM-basierten Sprachen keinen Grund zur Sorge darstellt.

Nachdem ein Container-Image-Konstrukt von Basis-Image, Buildpack und Anwendungscode vorliegt, wird dieses an die Orchestrierung übergeben. Abschließend wird für den Zugriff auf die Anwendung eine sogenannte „Route" erzeugt, die einen Zugriff über eine URL ermöglicht *(siehe Abbildung 5)*. In Kubernetes-/Docker-Sprache (und auch in der technischen Umsetzung) passiert hier also Folgendes:

- Erstellung eines Docker-Image mit einem gewarteten Basis-Image, Buildpack und Anwendungscode
- Push des Image in eine interne Container Registry
- Deployment der Anwendung in Kubernetes mit Konfiguration der Pods
- Erstellung eines ClusterIP-Service
- Erstellen einer Ingress-Regel zum externen Zugriff via URL auf die Anwendung

Für die Java-Entwicklung besteht bei der *cf-for-k8s*-Variante ein weiterer Vorteil. Hier wurde das bereits genannte Open-Source-Projekt Paketo für die Erstellung der Container-Images integriert. Im Vergleich zu früheren Versionen der Buildpacks, die fertige jar-Dateien als Übergabe erwartet haben, beinhaltet diese Technologie auch das Kompilieren des Java-Codes. Das heißt, es ist möglich, den `cf push`-Befehl direkt auf ein Maven *pom.xml* oder *Gradle File* anstatt eines „fat jar" zu richten. Somit werden nicht nur Container-Image-Builds, sondern auch der Code-Build automatisiert. Die Vorteile gegenüber Kubernetes sind damit:

- Fokus auf einfache Handhabung,
- Container-Image-Bau und -Betrieb liegen vollkommen innerhalb der Plattform

Knative

Knative hat im Vergleich zu Cloud Foundry keine Vergangenheit vor Docker und Kubernetes. Der Name beschreibt die Herkunft sehr gut, es ist Kubernetes-nativ, das heißt, es setzt direkt darauf auf und erweitert die Funktionalität und Handhabung.

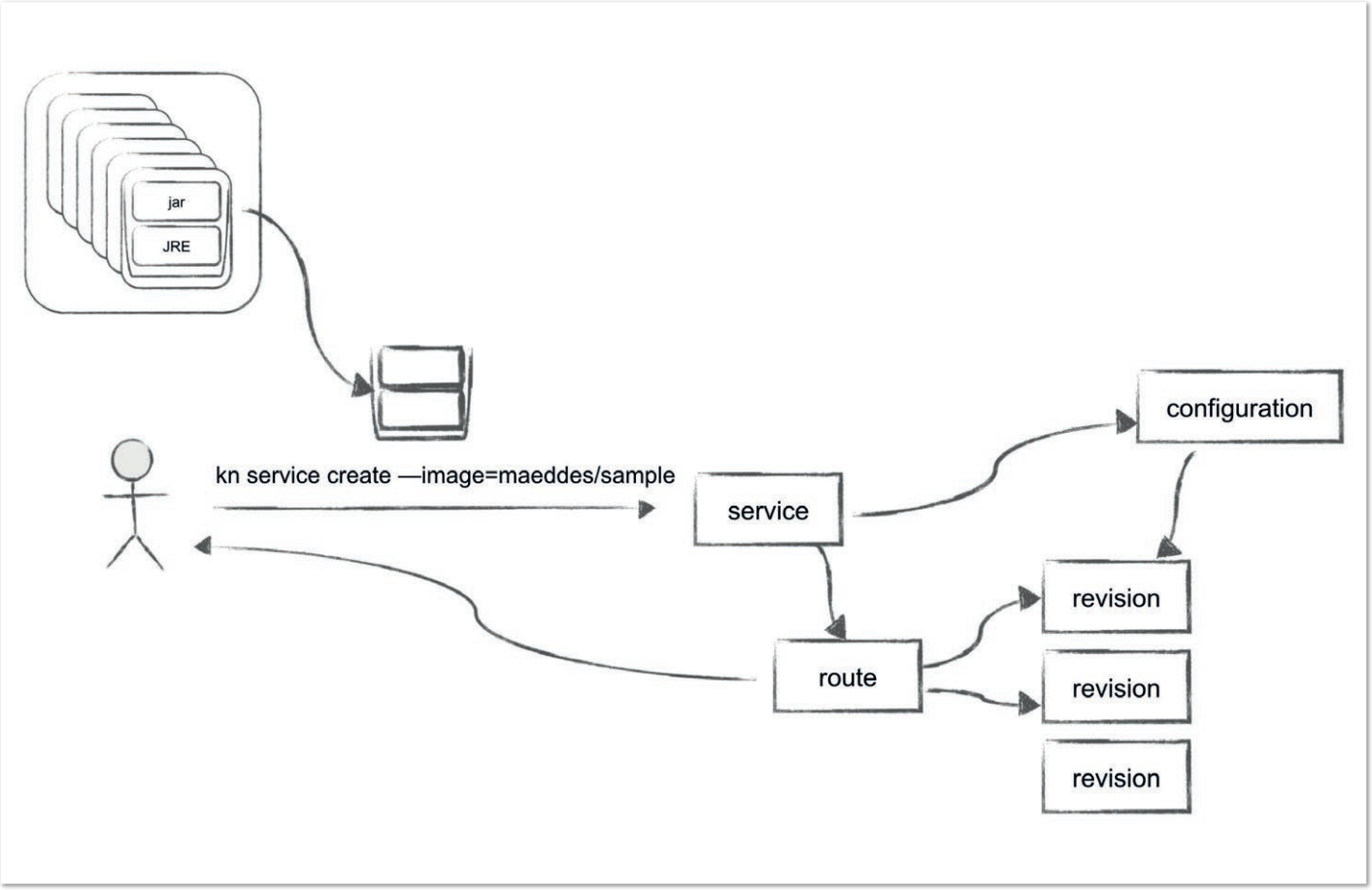


Abbildung 6: Knative Flow (© Matthias Häußler)

Knative besteht aus zwei Hauptprojekten. Die Kernkomponente nennt sich „Serving" und dient dem Deployment und der Laufzeitumgebung für Cloud-native Apps oder auch Functions. Damit ist „Serving" auch die Komponente, die hier zum Vergleich herangezogen wird. Die zweite Komponente „Eventing" hat, wie der Name vermuten lässt, den Fokus auf Event-basierte Anwendungen und ist damit etwas zu spezialisiert für diesen High-Level-Vergleich *[10]*.

An der Begrifflichkeit „Functions" ist die Ausrichtung der Technologie erkennbar. Während es bei Kubernetes um Container und bei Cloud Foundry um Anwendungen ging, steht hier eine noch höhere Abstraktionsschicht mit im Fokus – sogenannte „Serverless Functions". Dabei geht es im Kern darum, Funktionen auf der Plattform nur dann auszuführen, wenn sie aufgerufen werden, und möglichst schnell die Skalierung an den Bedarf anzupassen. Das beinhaltet auch die Skalierung auf null Instanzen, damit keine Apps im Leerlauf Ressourcen verbrauchen.

Um diese Funktionalität technisch überhaupt zu ermöglichen, benötigt Knative genauere Informationen über den Netzwerkverkehr innerhalb der Komponenten eines Clusters. Dazu gibt es per Dokumentation eine Auswahl an Technologien. Eine gängige Variante ist Istio *[11]*. Damit ist es Knative nicht nur möglich, den Netzwerkverkehr zu analysieren, sondern auch, ihn zu steuern.

Die Ausrichtung hin zu einer Serverless-Plattform bringt auch andere Anforderungen an den Code mit sich. Während es bei Cloud Foundry und Kubernetes ratsam ist, Code zustandslos aufzubauen und möglichst 12-factor-konform zu haben, kommt bei Knative noch die zusätzliche Anforderung einer schnellen Startzeit hinzu.

Allerdings ist es trotz stärkerem Fokus auf höhere Abstraktionsschichten auch hier, wie bei Kubernetes, so, dass der Code vor dem Deployment in ein Container-Image verpackt sein muss. Im Falle von Java-Entwicklung können hier Technologien wie Native Images der GraalVM *[12]* ihre Stärke sehr gut ausspielen. Für das Deployment gibt es, ähnlich zu Cloud Foundry, eine einfache CLI *[13]*, mit der eine containerisierte App oder Function einfach deployt werden kann.

Der Aufruf `kn create` führt hinter den Kulissen auch einige Schritte aus. Zunächst wird ein Knative-Service erstellt, der sich um den Lebenszyklus der neu erstellten Anwendung kümmert. Das beinhaltet Erstellung und Steuerung der Artefakte *Route* und *Configuration*. Während sich die Route um den Netzwerkverkehr zur Anwendung kümmert, deckt *Configuration* – wie der Name vermuten lässt – deren Konfiguration ab.

Eine *Revision* ist eine bestimmte Version der Anwendung. Bei einer Änderung der Konfiguration, wie zum Beispiel beim Update des Container-Image, wird daher eine neue Revision erstellt *(siehe Abbildung 6)*.

Neben der dynamischen Skalierung auf Basis von Netzwerklast gibt es bei Knative das sogenannte „Weighted/Percentage Routing". Hier lässt sich der Netzwerkverkehr dahingehend formen, dass explizit angegeben werden kann, wie viel Prozent der Aufrufe

die unterschiedlichen Revisionen bekommen. Die Konfiguration kommt hierbei aus dem Service-Objekt, die Route steuert den Fluss des Netzwerkverkehrs. Damit stellen sich in Knative folgende Vorteile gegenüber Kubernetes dar:

- Einfache, intuitive CLI
- Dynamische Skalierung basierend auf Netzwerklast
- Prozentuales Routing zwischen Revisionen der Anwendung

Fazit

Bei einem Artikel dieser Größenordnung ist es schwer, dem Funktionsumfang der genannten Technologien wirklich gerecht zu werden. Kubernetes, Cloud Foundry und Knative können deutlich mehr, als hier beschrieben. Des Weiteren ist dieser Vergleich sicherlich nicht umfassend, was Alternativen und Verbesserungen für die „User Experience" rund um das Thema Java-Entwicklung für Kubernetes angeht.

Insbesondere OpenShift *[14]* ist ein Kandidat, der sehr gut in den Vergleich passt. Das Fehlen hier entspricht jedoch keinesfalls einer Wertung, sondern meiner unglücklicherweise mangelnden Erfahrung damit.

Dennoch kann bereits an den gezeigten Beispielen gut verdeutlicht werden, wie leicht es möglich ist, hilfreiche Verbesserungen sowohl an der Nutzungsfreundlichkeit als auch beim Funktionsumfang von Kubernetes zu erzielen.

Das Schöne daran ist, dass es keine Entweder-oder-, sondern eine Und-Entscheidung ist. Die gezeigten Ansätze von Knative und Cloud Foundry setzen direkt auf Kubernetes auf, anstatt es zu ersetzen. Das heißt, es ist möglich, parallel das Kubernetes-API und das CF- oder Knative-API zu verwenden. Theoretisch ist es auch denkbar, alle drei Technologien in einem Cluster zu verwenden. Inwieweit sich die Technologien weiterverbreiten können oder welche Technologien sich noch dazugesellen, bleibt abzuwarten. Jedenfalls gibt es für die Technologien hilfreiche Tutorials für ein einfaches Setup, daher mein Vorschlag: Einfach mal ausprobieren.

Quellen

[1] Cloud Native Computing Foundation (CNFC) Landscape: *https://landscape.cncf.io/*
[2] Paketo Buildpacks, *https://paketo.io/*
[3] Fabric8 docker-maven-plugin, *https://github.com/fabric8io/docker-Maven-plugin*
[4] Docker Hub, *https://docs.docker.com/docker-hub/*
[5] Spring Boot Actuator, *https://docs.spring.io/spring-boot/docs/current/reference/html/production-ready-features.html*
[6] *https://kubecf.io/*
[7] *https://www.cloudfoundry.org/technology/cf-for-k8s/*
[8] 12-factor app, *https://12factor.net/de/*
[9] Cloud Native Buildpacks, *https://buildpacks.io/*
[10] Knative Dokumentation, *https://knative.dev/docs/*
[11] Istio, *https://istio.io/*
[12] GraalVM, *https://www.graalvm.org*
[13] Knative Client CLI, *https://knative.dev/docs/install/install-kn/*
[14] OpenShift, *https://openshift.io/*

Matthias Häußler
Novatec Consulting GmbH
matthias.haeussler@novatec-gmbh.de

Matthias Häußler ist Principal Advocate bei der Novatec Consulting GmbH, Dozent für Cloud Native Software Development an der Hochschule für Technik in Stuttgart und Organisator des Cloud Foundry Meetup Stuttgart. Er berät Kunden rund um das Thema Cloud, unterstützt aktiv Implementierungen und Migrationen und gibt Schulungen zu den Technologien. Er hält regelmäßig Vorträge und Workshops auf nationalen sowie internationalen Konferenzen und Meetups.

Alternativen zum Monolithen Kubernetes

Nils Bokermann

Moderne, verteilte Architektur löst Monolithen und große Systeme ab. Damit einhergehend ändert sich auch die Deployment-Architektur. Häufig wird die neue Welt direkt mit Kubernetes in Verbindung gebracht. Dass es neben Kubernetes andere Möglichkeiten gibt, die auch eine iterative Migration ermöglichen, wird in diesem Artikel gezeigt.

Als Erstes muss die Behauptung über den Monolithen Kubernetes klargestellt werden. Es sieht auf den ersten Blick doch so aus, als sei Kubernetes der Inbegriff einer Microservices-Architektur. Auf den zweiten Blick stellt man allerdings fest, dass die einzelnen Services derart verwoben sind, dass man hier eigentlich von einem Deployment-Monolithen sprechen kann.

Diese Architektur führt wiederum dazu, dass die Einführung von Kubernetes zu einer Big-Bang-Migration führt. Dabei wird nicht nur die Software-Architektur verändert, sondern auch viele betriebliche Belange wie Netzwerke, Storage und Deployment. Im klassischen Rechenzentrumsbetrieb werden diese Gewerke durch verschiedene Abteilungen oder Gruppen abgedeckt. Diese Gewerke müssen sich nun an ein gemeinsames Tooling und an Software-defined Networks anpassen.

Geht das vielleicht auch anders? Kann man nicht schrittweise im Rahmen einer iterativen Migration vorgehen? Um es vorwegzunehmen: Ja, man kann – und im Folgenden wird ein möglicher Weg aufgezeigt.

Um die folgenden Schritte in einem betrieblichen Kontext zu verstehen, werden zwei gegensätzliche Betriebskonzepte vorgestellt.

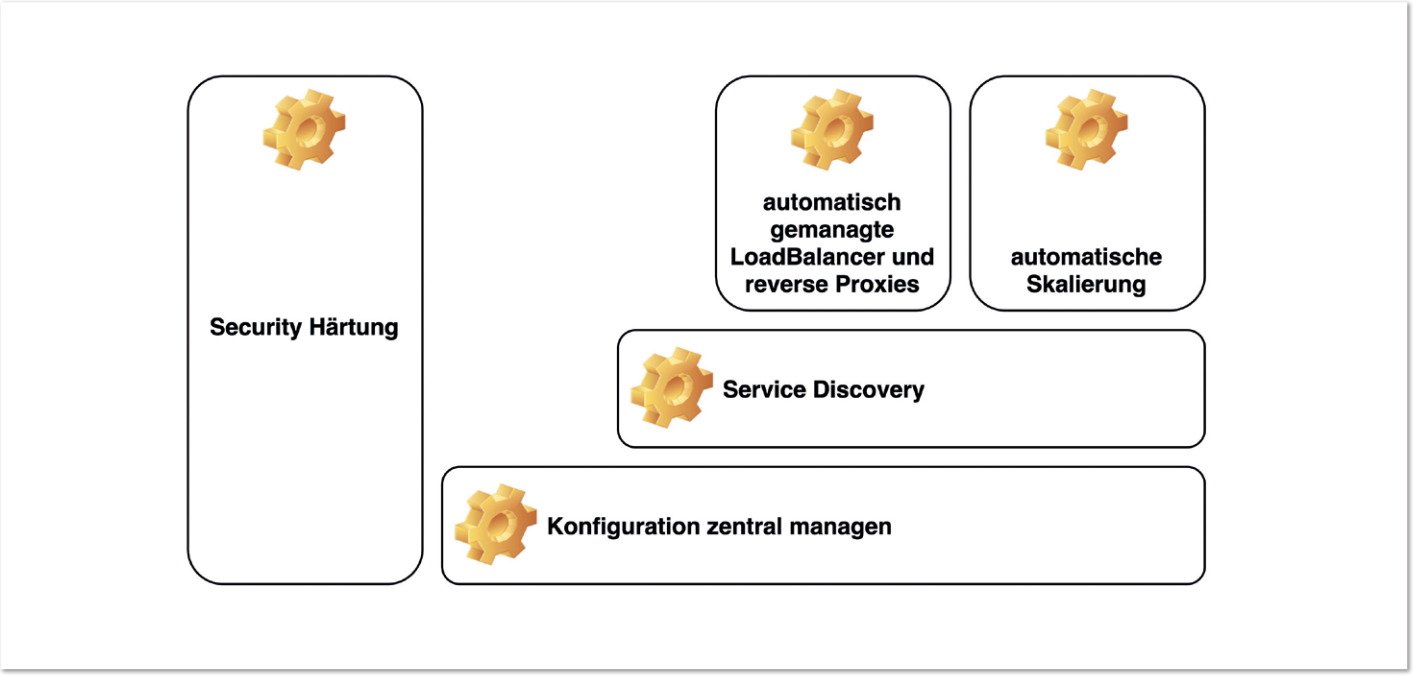


Abbildung 1: Modulübersicht (© Nils Bokermann)

Es soll im Weiteren zwischen einem manuell gemanagten Betrieb und einem automatisch gemanagten Betrieb unterschieden werden. Beim manuell gemanagten Betrieb stehen die Konfiguration und das Deployment-Ziel zum Deployment-Zeitpunkt fest. Damit kann die Konfiguration vor dem Deployment auf das entsprechende System gepusht werden. Im Gegensatz dazu wird bei einem automatisch gemanagten Betrieb das Deployment-Ziel erst während des Deployments festgelegt. Damit ist eine proaktive Verteilung von Konfiguration nicht mehr sinnvoll möglich. Der Service muss also beim Start seine Konfiguration aus einer Konfigurationsdatenbank ziehen. Die beiden Betriebsansätze unterscheiden sich also vornehmlich in der Art des Konfigurationsmanagements. Während der manuell gemanagte Betrieb nach dem Push-Prinzip verteilt, geschieht dies im automatisch gemanagten Betrieb nach dem Pull-Prinzip.

In *Abbildung 1* finden sich die Bausteine, die für eine Transition in einen automatisiert gemanagten Betrieb in Betracht kommen. Als Basis dient die zentrale Verteilung von Konfiguration (Konfiguration zentral managen). Damit kann ein Verteilen von Environment-Dateien auf verschiedene Systeme abgelöst werden. Eine weitere Vereinfachung der Konfiguration kann durch die Einführung einer Service-Discovery-Lösung geschaffen werden. Sind die Informationen über Services im Service Discovery vorhanden, kann mittels dieser Information auch Load-Balancing und Reverse-Proxy konfiguriert werden. Damit ist auch eine dynamische Verteilung von Services auf Systeme und eine automatische Skalierung möglich. Die Security-Härtung wird in diesem Artikel an den Schluss gestellt, obwohl sie von allen anderen Maßnahmen unabhängig ist und demnach auch zu jeder Zeit in der Transition eingeführt werden kann.

Beispiel-Code

Im Folgenden werden immer wieder Code-Snippets gezeigt, die zu einer einfachen Client-Server-Anwendung gehören. Diese Anwendung findet sich in den verschiedenen Transitionsstufen auf GitHub *[1]*.

Konfiguration zentral managen

Im ersten Schritt der Transition wird es darum gehen, die Shell-Environment-Verteilung loszuwerden. Damit sind Konfigurationswerte gemeint wie:

- URLs zu anderen Services,
- Flags,
- Feature-Switches oder
- Credentials

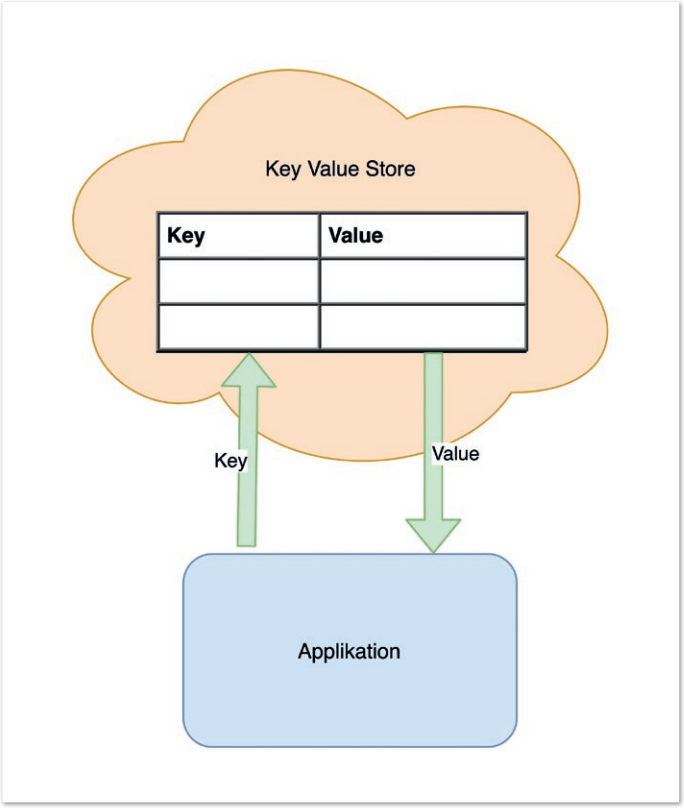


Abbildung 2: Key-Value Store (© Nils Bokermann)

```
public HeroServiceRestClient(
           @Value("${backend.url}") String backendUrl,
           @Value("${backend.username}") String username,
           @Value("${backend.password}") String password) {
     …
}
```

Listing 1: Klassische Value-Injection in Spring Boot

```
public HeroServiceRestClient(
              ConfigurableEnvironment ce) {
    String beUrl = ce.getRequiredProperty("backend.url");
    String username = ce.getProperty("backend.username");
    String password = ce.getProperty("backend.password");
    …
    }
```

Listing 2: Value-Injection mittels ConfigurableEnvironment

Zu diesem Zweck wird ein Key-Value-Store an die Services angeschlossen, in dem die ehemaligen Environment-Variablen zentral gespeichert werden. Zum Abruf wird der Value zu einem bekannten Key angefragt *(siehe Abbildung 2)*.

Als Beispiel für einen Key-Value-Store wird HashiCorp Consul *[2]* genutzt. An diesem Beispiel werden die Umstellungen an Software-Architektur vorgestellt und die Einflüsse auf den Betrieb aufgezeigt. Wenn in der bestehenden (Spring-Boot-)Software die Konfiguration via `application.properties` „injected" wurde, ist nur eine Erweiterung der Dependencies um `spring-cloud-starter-consul-config` notwendig. Es ist allerdings zu Debugging-Zwecken ratsam, zumindest an einer Stelle im Code die Änderungen von *Listing 1* zu *Listing 2* zu übernehmen. Durch die Injection des `ConfigurableEnvironment` kann überprüft werden, welche Datenquellen angezogen werden.

Auf der Seite des Betriebs ändern sich in diesem Schritt mehrere Dinge. Zum einen muss eine neue Software installiert und konfiguriert werden. Zum anderen wird der Verteilungsmechanismus für die Environment-Variablen umgestellt. Eine sinnvolle Verteilung der Daten nach Service-Gruppen und Schutzbedarf sollte in diesem Schritt bedacht oder zumindest begonnen werden.

Wie Ihnen sicher aufgefallen ist, sind die Credentials einfach so mit allen anderen Environment-Variablen umgezogen. Aus Security-Gesichtspunkten sollten diese natürlich einem besonderen Schutz unterliegen. Wenn die Credentials allerdings vorher genau wie alle anderen Variablen im Shell-Environment gelegen haben, ist die Überführung in den zentralen Key-Value-Store immerhin keine Verschlechterung. Existiert jedoch schon eine entsprechender Secret-Store, so sollte dieser auch weiter genutzt werden.

Service Discovery

Unter dem Begriff Service versteht man in diesen Zusammenhang jede Art von Dienst, die über Netzwerk angesprochen wird. Von der Datenbank über REST-Services, Docker-Container, Application-Server und FatJar-Anwendungen.

Um die Service Discovery nutzen zu können, muss sich der Service-Geber (Server) an der Service Registry anmelden (siehe *Abbildung 3*, Schritt 1). Danach kann der Service-Nehmer (Client) auf die Service Registry zugreifen und analog zum Key-Value-Store über den Service-Namen die aktuelle URL und gegebenenfalls Meta-Daten abfragen (siehe *Abbildung 3*, Schritte 2 und 3). Die Nutzung erfolgt dann analog zur Nutzung der statisch konfigurierten URL aus dem vorigen Beispiel.

Da HashiCorp Consul neben dem Key-Value-Store auch die Möglichkeit hat, als Service Registry zu fungieren, wird diese Funktionalität genutzt.

Die Software-Änderung im Bereich des Service-Gebers beschränkt sich, dank Spring-Boot, wiederum auf das Hinzufügen einer Dependency. Es wird einfach das Paket `spring-cloud-starter-consul-discovery` hinzugefügt. Zusätzlich sollte der Health-Check-Endpunkt für Consul gepflegt werden, damit nur Endpunkte verteilt werden, die gestartet und aktiv sind. In *Listing 3* wird diese Konfiguration beispielhaft für Spring-Boot-Actuator gezeigt.

Auf der Service-Nehmer-Seite wird etwas mehr Aufwand nötig. Die Services werden nicht durch das `ConfigurableEnvironment` oder `@Value`-Annotationen „injected", sondern müssen über den DiscoveryClient bezogen werden. In *Listing 4* wird die Service-URL für den Service „Backend" aus dem `DiscoveryClient` bezogen.

Auf der Betriebsseite ist für diese Stufe der Transition wenig zu tun. Der Consul-Service ist bereits im Rechenzentrum verteilt und liefert Daten aus. Auch der Rückbau der statisch konfigurierten URLs kann und sollte erst nach dem Ausrollen der Service Discovery durchgeführt werden. Solange die Services ihren Standort nicht verändern, bleibt die statische Information valide. Nach Umstellung der Clients sollten die nicht mehr benötigten URLs aus dem Key-Value-Store entfernt werden, um Seiteneffekte zu entdecken und Verwirrung zu vermeiden.

Reverse Proxy

Bis zu diesem Schritt wurden Reverse Proxies und Loadbalancer durch das Betriebsteam händisch provisioniert. Nachdem nun aber alle notwendigen Informationen in der Service Registry vorliegen, können diese auch genutzt werden, um zum Beispiel den Reverse Proxy zu konfigurieren.

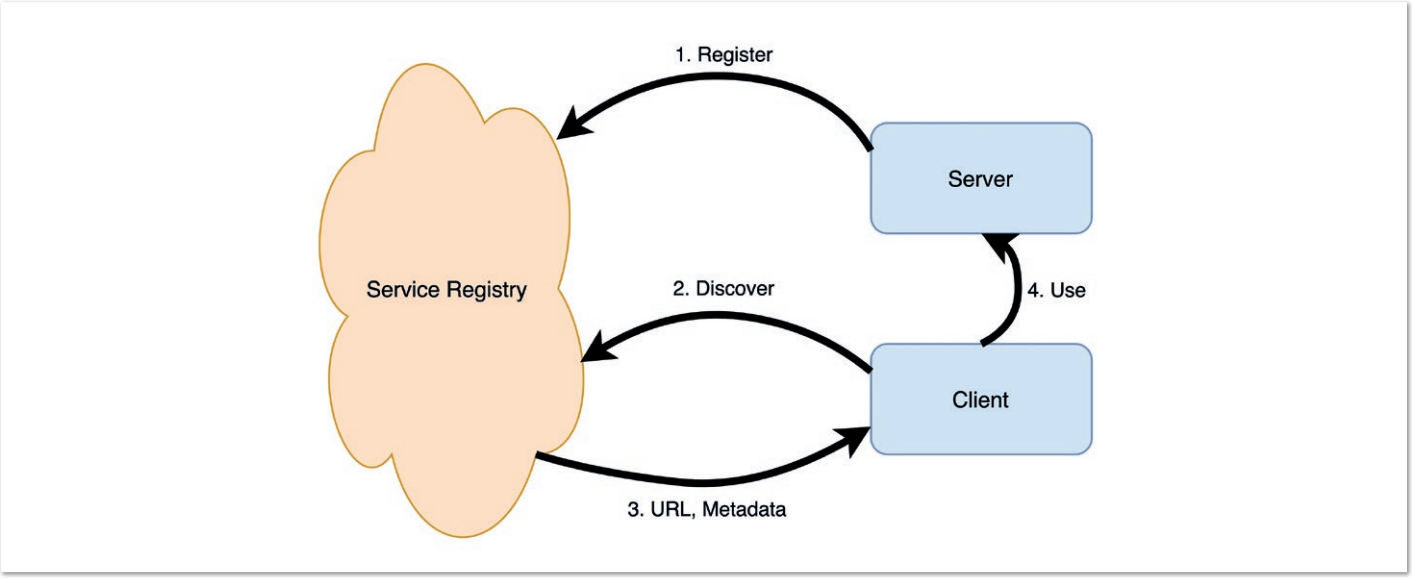


Abbildung 3: Service Discovery Flow (© Nils Bokermann)

```
spring.cloud.consul.discovery.healthCheckPath=/actuator/health
spring.cloud.consul.discovery.healthCheckInterval=5s
```

Listing 3: Konfiguration des Health-Endpunkts in den application.properties.

Am Beispiel von `traefik` *[3]* wird die Konfiguration eines Reverse Proxy vorgestellt. Die Vorbereitungen für diesen Schritt wurden auf der Software-Seite schon durch die Anbindung an die Service Discovery getroffen. Hier ist also die Betriebsabteilung gefragt, die Reverse Proxies umzustellen.

Eine minimale Konfiguration findet sich in *Listing 5*. Diese Konfiguration eignet sich dazu, in einer Entwicklungsumgebung erste Schritte zu gehen, ist jedoch nicht für einen produktiven Betrieb geeignet. Hierfür müssen weitere Regeln erstellt werden, da in dieser Konfiguration unter anderem auch der Consul-Service exponiert wird und damit das komplette Environment, also auch die Passwörter unserer Services.

Automatische Skalierung

Als nächster logischer Schritt steht die Deployment-Verteilung an. Auch zu diesem Zweck wird wiederum auf ein Tool aus dem Hause HashiCorp zurückgegriffen: HashiCorp Nomad *[4]*. Nomad ist ein

```
private String retrieveURL(
    DiscoveryClient discoveryClient) {
    Optional<ServiceInstance> serviceOptional =
       discoveryClient.getInstances("Backend")
            .stream().findAny();
    if (serviceOptional.isPresent()) {
        ServiceInstance serviceInstance =
             serviceOptional.get();
        URI uri = serviceInstance.getUri();
        return uri.toString();
    }
    return null;
}
```

Listing 4: Nutzung des DiscoveryClient, um einen dynamischen Endpunkt zu ermitteln

„Workload Orchestrator", es verteilt Applikationen auf zur Verfügung stehende Systeme. Dazu wird ein regelbasiertes System genutzt, das aufgrund von Last- und Speicher-Anforderungen der Applikation entscheidet, wohin die Applikation deployt wird.

Diese Form der Automatisierung hat diverse Auswirkungen auf die Software-Architektur. Durch die automatische Verteilung und Optimierung kann eine Applikation auf einem System beendet werden, um auf einem anderen System neu gestartet zu werden. Die Applikationen müssen also tolerant gegen einen Stopp per Unix-Signal sein. Es kann dabei genauso vorkommen, dass die Applikation auf zwei Systemen gleichzeitig läuft und Anfragen an beide Instanzen der Applikation verteilt werden. Lokale Datenhaltung oder Singletons können daher zu Problemen führen.

Für die Clients ergeben sich darüber hinaus noch weitere Anforderungen. Sie müssen in der Lage sein, auf fehlende Services fehlertolerant zu reagieren, zum Beispiel durch Retry-Mechanismen oder durch gecachte Default-Werte. Diese Lösungsansätze sind offensichtlich fachlich getrieben und daher kann es keine allgemeingültige (technische) Lösung geben.

Um einen möglichen Retry eines Requests zu gestatten, muss darauf geachtet werden, dass Schnittstellen idempotent gestaltet werden.

Die Nomad-Konfiguration *(siehe Listing 6)* bildet die Komplexität und Flexibilität einer Rechenzentrumsverwaltung ab. Nomad verwaltet eine dreistufige Hierarchie aus `jobs`, `groups` und `tasks`. Ein `task` ist das Deployment einer ausführbaren Einheit, also einer Applikation, eines Docker-Containers oder auch eines Skripts.

Innerhalb des `task` wird ein `service` definiert, der der Service Registry bekannt gemacht wird. Ein `check` ist optional, aber überaus sinnvoll,

```
# Consul als Provider anschliessen
[providers.consulCatalog]
  # Alle Service Exponieren (Nur für Entwicklungszwecke)
  exposedByDefault = true
  # Abfrage-Häufigkeit. Abwägung zwischen traffic und Ausfallsicherheit
  refreshInterval = "30s"
  # Für jeden Service einen Hostname mit seinem Namen
  defaultRule = "Host(`{{ .Name }}`)"
# Consul-Server
[providers.consulCatalog.endpoint]
    scheme = "http"
    address = "127.0.0.1:8500"
```

Listing 5: Erweiterung um den DiscoveryClient

```
# Job ist die toplevel Struktur-Einheit
job "frontend" {
  region = "global"
  datacenters = ["dc1"]

  type = "service"

  update {
    stagger      = "30s"
    max_parallel = 1
  }

# Group ist die zweite Stufe der Strukturierung
  group "webs" {
    count = 1

# Task ist die dritte Stufe der Strukturierung
    task "frontend" {
      # Nomad unterstützt verschiedene “driver”.
      driver = "java"

      # Die Konfiguration ist spezifisch für den „driver“
      config {
        jar_path    = "local/frontend-1.0.0-SNAPSHOT.jar"
        args        = ["--backend.instanceName=backend-docker-backend"]
      }

      artifact {
        source      = "https://github.com/sanddorn/InfrastructureAsMicroservice/releases/download/1.0.0-SNAPSHOT/
frontend-1.0.0-SNAPSHOT.jar"
        options {
          checksum = "md5:892e87bd35f7ca7c989b9ad38df85632"
        }
      }

      service {
        port = "http"

        check {
          type     = "http"
          path     = "/actuator/health"
          interval = "30s"
          timeout  = "2s"
        }
      }

      resources {
        cpu    = 500 # MHz
        memory = 512 # MB

        network {
          port "http" {
       static = 8080
          }
        }
      }
    }
  }
}
```

Listing 6: Nomad-Deployment-Konfiguration

um den Betrieb der Applikation automatisiert sicherzustellen. Soll der Service-Port dynamisch, und nicht wie im Beispiel statisch, konfiguriert werden, ist das möglich. Die gewählte Adresse und der Port liegen dann im Start-Environment der Applikation. Weiterführende Informationen sind auf der Nomad-Website zu finden *[5]*.

Security-Härtung

Es ist offensichtlich, dass Credentials nicht im Klartext gespeichert werden sollten. Zuvor, bei der Einführung des Key-Value-Stores, wurde dieses Sicherheitsrisiko bewusst in Kauf genommen. An dieser Stelle werden wir durch die Einführung einer Secret-Management-Lösung dieses Problem beheben. Die Einführung einer Secret-Management-Lösung lässt sich zu jeder Zeit während der Transition angehen, also auch als ersten Schritt.

Mit HashiCorp Vault *[6]* steht eine Secret-Management-Lösung zur Verfügung, die sich ähnlich einfach, wie auch schon HashiCorp Consul, in das Spring-Boot-Universum einführen lässt. Neben der zusätzlichen Dependency (`spring-cloud-starter-vault-config`) muss hier allerdings noch eine Bootstrap-Konfiguration angelegt werden *(siehe Listing 7)*. Die Applikationskonfiguration (`application.properties`) reicht hierbei nicht, da auch Credentials verwaltet werden können, die schon im Bootstrap der Spring-Boot-Anwendung genutzt werden (beispielsweise Datenbank-Credentials oder TLS-Keys und Zertifikate).

Innerhalb der Demo-Anwendung wird ein Vault-Token benutzt, das ein vollständiges Credential ist. In einem sicheren Applikationsbetrieb sollte eine andere Art der Vault-Absicherung gewählt werden. Für die Applikationsentwicklung ist die Einbindung der Secret-Management-Lösung damit abgeschlossen; die kompliziertere Aufgabe ist es, das Rechte-Management sowie die Replikation innerhalb der Secret-Management-Lösung zu konfigurieren und zu implementieren.

Die neue „Silver Bullet"?

Die „Silver Bullet" ist diese Lösung nicht. Es gibt diverse andere Produkte auf dem Software-Markt, mit denen eine ähnliche Lösung aufzubauen ist. Da die HashiCorp-Module sich aber so gut in die Spring-Boot-Welt einbauen lassen, lag es nahe, diese auch für einen kurzen Abriss zu nutzen.

Kubernetes diskret?

War das jetzt Kubernetes in einzelnen Bausteinen? So halb. Auf der einen Seite ist die komplette Netzwerk-Schicht außen vor geblieben und kann nach bekannter Art mit Switch, Router und Firewall weiter betrieben werden. Auf der anderen Seite gibt es hier die Möglichkeit, nicht nur Container zu deployen, sondern auch andere Arten von Software, solange sie aus einer Shell heraus gestartet werden können.

```
spring:
  cloud:
    vault:
      token: ${VAULT_TOKEN}
      host: localhost
      port: 8200
      scheme: http
```

Listing 7: bootstrap.yml zur Konfiguration des Secret-Managements

Für den Fall, dass wir die Applikationen in Container verpacken können, ist die entstandene Landschaft darauf vorbereitet, auch innerhalb von Kubernetes-Clustern betrieben zu werden. Die Hürde, die jetzt noch durch die Umsetzung von Namespaces, Network-Policies und Pod-Security-Policies zu nehmen ist, wird deutlich leichter fallen als zu Beginn der Transition. Fraglich bleibt jetzt allerdings, ob ein Umstieg auf Kubernetes wirtschaftlich sinnvoll ist.

Wie geht's weiter?

In diesem Artikel wird jeder einzelne Transitionsschritt nur angerissen und die Folgen auf die Software-Architektur tiefer, die Folgen auf die Betriebsstruktur nur sehr oberflächlich behandelt. Die betrieblichen und Security-Aspekte für die Einführung der jeweiligen Services (Consul, Nomad, traefik und Vault) würden den Rahmen eines Überblick-Artikels bei Weitem sprengen.

Quellen

[1] Nils Bokermann, *https://github.com/sanddorn/InfrastructureAsMicroservice*
[2] *https://www.consul.io*
[3] *https://traefik.io/traefik/*
[4] *https://www.nomadproject.io*
[5] *https://www.nomadproject.io/docs/job-specification/service*
[6] *https://www.vaultproject.io*

Nils Bokermann
Freiberufler
nils.bokermann@bermuda.de

Nils Bokermann ist als freiberuflicher IT-Consultant unterwegs und berät seine Kunden in Architektur- und Methodik-Fragen. Er stellt gerne Hype-getriebene Entscheidungen infrage und ist daher gerne auch als „Advocatus Diaboli" bekannt.

Enterprise Service Bus in Rente, oder: die neue Art, Systeme zu integrieren

Markus Lohn, esentri AG

Das Verlassen der Komfortzone ist häufig ein schwieriges Unterfangen. Bisher nutzten wir den Oracle Service Bus (OSB) [1] als Werkzeug für die Entwicklung und den Betrieb von Schnittstellen bei einem mittelständischen Kunden. Doch dieses Vorgehen funktionierte für uns in diesem Kontext nicht mehr optimal. Seit geraumer Zeit implementieren wir nun Schnittstellen auf Basis einer Microservices-Architektur und mit Containern. Den Betrieb und die Überwachung dieser Services managen wir durch eine Container-Plattform auf Basis von Red Hat OpenShift [2]. In diesem Artikel erfahren Sie, warum sich der Kurswechsel mehr als gelohnt hat und wie die neue Technik mehr Geschwindigkeit und Qualität in Implementierung und Betrieb brachte.

Es hat sich gelohnt, weil mehr Entscheidungsfreiheit die Agilität im Projekt erhöhte.

Mehr Entscheidungsfreiheit wird durch die Themen Technologie, Minimalismus und Entwicklungsumgebung begünstigt. Wir entwickeln und betreiben jede Schnittstelle in einem eigenen Container *(siehe Abbildung 1)*. Container-Technologien, beispielsweise Docker, sind seit einigen Jahren in der IT etabliert. Innerhalb eines Containers kann jede beliebige Technologie ausgeführt werden. Beliebt ist die Nutzung von Node.js oder Java in Form einer Spring-Boot-Applikation. Somit besteht die Möglichkeit, die passende Technologie für die zu lösende Aufgabe auszuwählen. In diesem Projekt implementierten wir die Schnittstellen auf Basis von Red Hat Fuse *[3]* und Spring Boot. Im Vergleich ist vor allem Java- und XML-Wissen in einem OSB-Projekt gefragt. Andere Technologien sind nur sehr schwer integrierbar. Der OSB benötigt eine Oracle-WebLogic-Server(WLS)-Infrastruktur. WebLogic Server ist ein Java-Server und implementiert Java EE. Daher sind alle Technologien aus dem Java-Ökosystem grundsätzlich anwendbar. Eine Migration auf einen anderen OSB- oder Java-EE-Server ist theoretisch möglich, praktisch aber immer mit einer Neuentwicklung oder erheblichem Aufwand verbunden.

WLS stellt eine Vielzahl von Komponenten zur Verfügung, zum Beispiel JMS-Server, HTTP-Server, EJB-Server, MBean-Server, Libraries, Applikationen und mehr. Viele dieser Komponenten werden nicht in jedem Anwendungsfall benötigt, aber dennoch immer geladen. Das verursacht eine lange Startzeit und verbraucht unnötig Ressourcen. Innerhalb eines Containers wird nur das genutzt, was im Dockerfile definiert wurde. Beispielsweise ist für eine Spring-Boot-Applikation lediglich eine Java-Runtime erforderlich.

Mit dem Oracle JDeveloper ist die IDE in einem OSB-Projekt fest vorgegeben und kann nicht frei gewählt werden. Die Integration von Apache Maven im JDeveloper mit OSB-Projekten ist nicht optimal gelungen. Es sind doch einige Anpassungen erforderlich, damit ein OSB-Projekt über Maven in eine CI/CD-Pipeline integriert werden kann (`-Djava.security.egd=file:///dev/urandom`, Projektstruktur standardisieren, clean konfigurieren, Config-Pläne integrieren etc.) Für die Entwicklung von Services mit Red Hat Fuse kann jede beliebige Java-IDE verwendet werden. Die Nutzung eines speziellen Maven-Plug-ins für Fuse ist nicht erforderlich.

Bei der Auswahl der Technologie hat man beim Einsatz von Containern weniger Einschränkungen. Ferner nutzt man nur die Komponenten, die tatsächlich für die Problemlösung erforderlich sind. Die Entwicklung macht einfach mehr Spaß und die Flexibilität wird erhöht, da kein technologischer Zwang besteht. Durch schnellere Bereitstellung und Feedback kann eine agile Vorgehensweise viel leichter unterstützt und umgesetzt werden.

Wie schnell können neue Kollegen die Technologie lernen und anwenden? Hat sich das im Projekt ausgewirkt?

Es hat sich gelohnt, weil neue Kollegen viel schneller integriert wurden. Zum einen wird das am Zeitaufwand für den Aufbau einer Entwicklungsumgebung deutlich. Die Entwicklungsumgebung für Fuse mit OpenShift aufzubauen benötigt nur zwei Schritte, die in weniger als zwei Stunden durchgeführt werden können. Der Aufruf eines Installers und die Konfiguration der OpenShift-Plattform erfolgen direkt aus der IDE. Im Bereich OSB sind viele Schritte (JDeveloper, Datenbank, WLS und OSB installieren) erforderlich, die unter Umständen mehrere Stunden Aufwand benötigen. Natürlich existieren Lösungen, um den Prozess und die notwendige Zeit zu reduzieren, zum Beispiel durch Einsatz von Vagrant *[4]*. Jedoch ist der initiale Aufwand erheblich und an die Zeit für Fuse kommen auch diese Verbesserungen nicht heran.

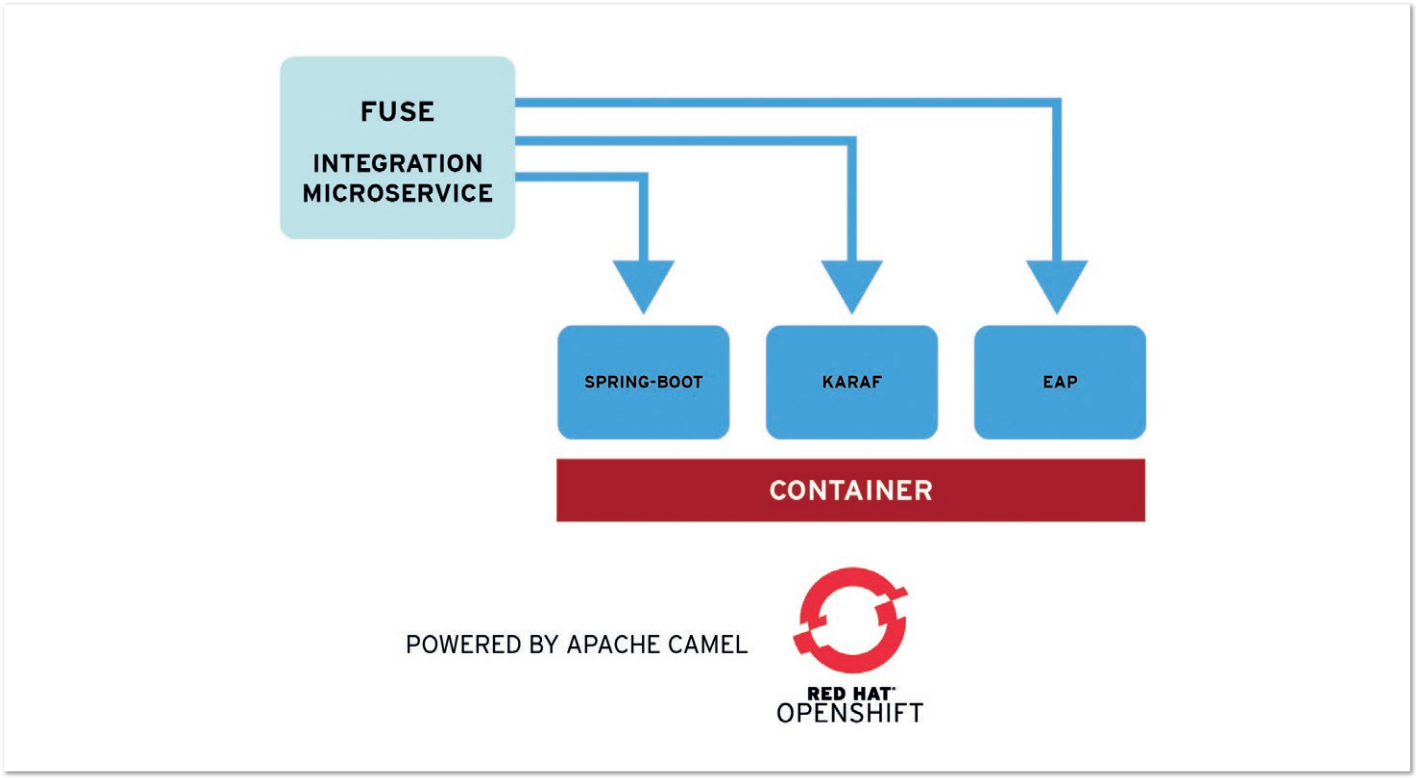


Abbildung 1: Fuse-Integration mit Spring Boot, running on OpenShift (© Red Hat)

Zum anderen wird die Einarbeitung durch eine standardisierte Projektstruktur erleichtert. Fuse nutzt Apache Maven als Buildwerkzeug und nutzt die vorgegebene Projektstruktur von Maven. Jeder Entwickler mit Maven-Erfahrung findet sich im Projekt sofort zurecht. In einem OSB-Projekt gibt es einen solchen Standard nicht. Die Projektstruktur muss definiert und durch Richtlinien sowie automatisierte Werkzeuge geprüft und sichergestellt werden.

Ferner unterstützt Fuse durch die Nutzung einer Java-DSL (Domain-specific Language) vor allem Entwickler beim Einarbeiten und Verständnis der Logik. Im OSB liegen Quelltexte ausschließlich als XML-Dateien vor. Ohne die Nutzung von Oracle JDeveloper, der die XML-Dateien in einer grafischen Anzeige aufbereitet, wäre eine Einarbeitung nicht denkbar.

In diesem Projekt integrierten wir sowohl junge Kollegen mit wenig Projekterfahrung als auch erfahrene Seniors. Insgesamt betrachtet benötigten die neuen Kollegen zirka einen Tag für ein Onboarding im Projekt. Bereits am zweiten Tag wurden User Stories aus dem Backlog erfolgreich bearbeitet. In OSB-Projekten konnten wir das bis dato so nicht bewerkstelligen.

Durch das schnellere Aufsetzen der Entwicklungsumgebung und den standardisierten Aufbau von Fuse-Projekten konnten Entwickler viel schneller im Projekt produktiv werden. Die Einarbeitung in die Technologie wird durch die große, engagierte Community und die vielen Ressourcen im Netz erheblich unterstützt und verkürzt. Als Ergebnis konnten sich die Entwickler schneller im Projekt integrieren und auf die Lösung von Fachproblemen fokussieren.

Wie aber war die allgemeine Entwicklungsexperience? Wurde die erhoffte Akzeptanz erreicht?

Es hat sich gelohnt, weil die Entwicklung neuer Integrationen „leichter von der Hand ging". Wiederverwendung konnte in Fuse mit bekannten Mitteln, zum Beispiel Vererbung, Auslagerung in Libraries oder Copy & Paste, sehr einfach erreicht werden. Im OSB funktioniert das zwar auch, aber über die Anlage eines separaten OSB-Projekts. In diesem neuen OSB-Projekt können dann beispielsweise XSDs, WSDLs, XLSTs oder Alerts zusammengefasst werden. Das Referenzieren dieser Artefakte muss dann jedoch über relative Pfade sichergestellt werden. Das hatte natürlich Auswirkungen auf Entwicklung und Betrieb.

Refactoring kann mit Fuse einfach in der genutzten IDE ausgeführt werden. Die Neuorganisierung von Packages oder die Umbenennungen von Klassen stellen keine Probleme dar und werden optimal durch die Entwicklungsumgebung unterstützt. Ab der Version 12c wurde das Refactoring im OSB erheblich verbessert, vor allem in Bezug auf Namespaces. Aber dennoch kann es vereinzelt heute noch notwendig werden, Dinge manuell in XML-Dateien zu ändern.

Eine Fuse-Spring-Boot-Applikation kann im Vergleich zum OSB viel schneller getestet werden. Eine Spring-Applikation kann auf der Entwicklungsumgebung in weniger als 30 Sekunden gestartet werden. Das ist mit dem OSB nie erreichbar. Ferner hatten wir immer wieder Probleme mit Git und dem Merge von XML-Dateien im OSB. Bei Fuse mit Java konnten wir diese Probleme nicht mehr feststellen.

Die Unterstützung von Patterns ist bei Fuse fester Bestandteil im Framework. Im Bereich „Integration" existiert ein Standardwerk von Gregor Hope – „Enterprise Integration Patterns". Für viele der beschriebenen Patterns gibt es in Fuse eine eigene Komponente, zum Beispiel Routing Slip, Content Enricher. Im Bereich OSB gibt es keine speziellen Komponenten, die Patterns implementieren. Einige Integrationspatterns werden im Produkt direkt unterstützt. Andere Patterns müssen manuell nachprogrammiert werden, wie Content Enricher oder Wire Tap.

Durch die Anwendung von Integration- und Software-Patterns sowie die sehr gute Werkzeugunterstützung wurde die tägliche Arbeit jedes Entwicklers unterstützt und erleichtert. Generell war die Akzeptanz und Zufriedenheit bei den Projektmitgliedern insgesamt höher als zuvor.

Hatte das Vorgehen auch Auswirkungen auf die Qualität?

Es hat sich gelohnt, weil wir die Qualität enorm gesteigert haben. Vorgaben hinsichtlich der Gestaltung von Source Code oder Architektur konnten wir durch Einsatz entsprechender Werkzeuge, automatisiert für alle Schnittstellen mit Fuse, prüfen und sicherstellen. Die Prüfungen wurden auch in den Build-Prozess integriert. Für OSB-Projekte existiert hier keine Werkzeugunterstützung. Somit sind solche Vorgaben zunächst einmal nur manuell durchführbar und damit ist der gesamte Prozess anfällig für Fehler.

Der große Vorteil bei Fuse besteht darin, jede einzelne Komponente durch einen Unit-Test prüfen zu können. Das Mocking von Endpunkten ist bereits im Framework enthalten, deshalb entfällt die Nutzung eines zusätzlichen Mocking-Werkzeugs. Daher kann eine Schnittstelle ausgiebig geprüft werden. Vor allem für Mappings von Daten war das ein wirkungsvolles Instrument. Im OSB ist kein Testframework integriert. Es muss auf jeden Fall auf ein zusätzliches Werkzeug zurückgegriffen werden. In der Vergangenheit nutzten wir hierzu immer SoapUI. Allerdings sind das praktisch nur Integrationstests, keine Komponententests. Diese Tests lassen sich nicht immer einfach umsetzen, wenn die Eingangs- und Ausgangsnachrichten sehr komplex sind.

Zusätzlich steigerten wir die Qualität durch Nutzung von Werkzeugen zur statischen Codeanalyse. In unserem konkreten Fall nutzten wir PMD und stellten sicher, dass alle Blocker entfernt wurden. Für den OSB existieren keine Werkzeuge für die Codeanalyse.

Viele Entwicklungsvorgaben in den Projekten konnten automatisiert geprüft werden. Ein großer Vorteil war die Möglichkeit, die Komponenten einer Schnittstelle unabhängig voneinander testen zu können. Vor allem Mappings änderten sich im Laufe des Projekts häufiger. Durch die automatisierten Tests konnte die Qualität auch bei vielen Änderungen sichergestellt werden.

Für die Entwicklung war der Wechsel enorm lohnend. Wie stellte sich das nun für den Betrieb dar?

Es hat sich gelohnt, weil das Betriebsmodell durch Container bestens unterstützt wird. Die IT hat das Ziel, die Verantwortlichkeiten stärker zwischen Plattform und Anwendungen zu trennen. Auf organisatorischer Ebene ist das schon seit geraumer Zeit gelungen. Durch Nutzung einer geeigneten Plattform sollte das auch auf technischer Ebene erreicht werden.

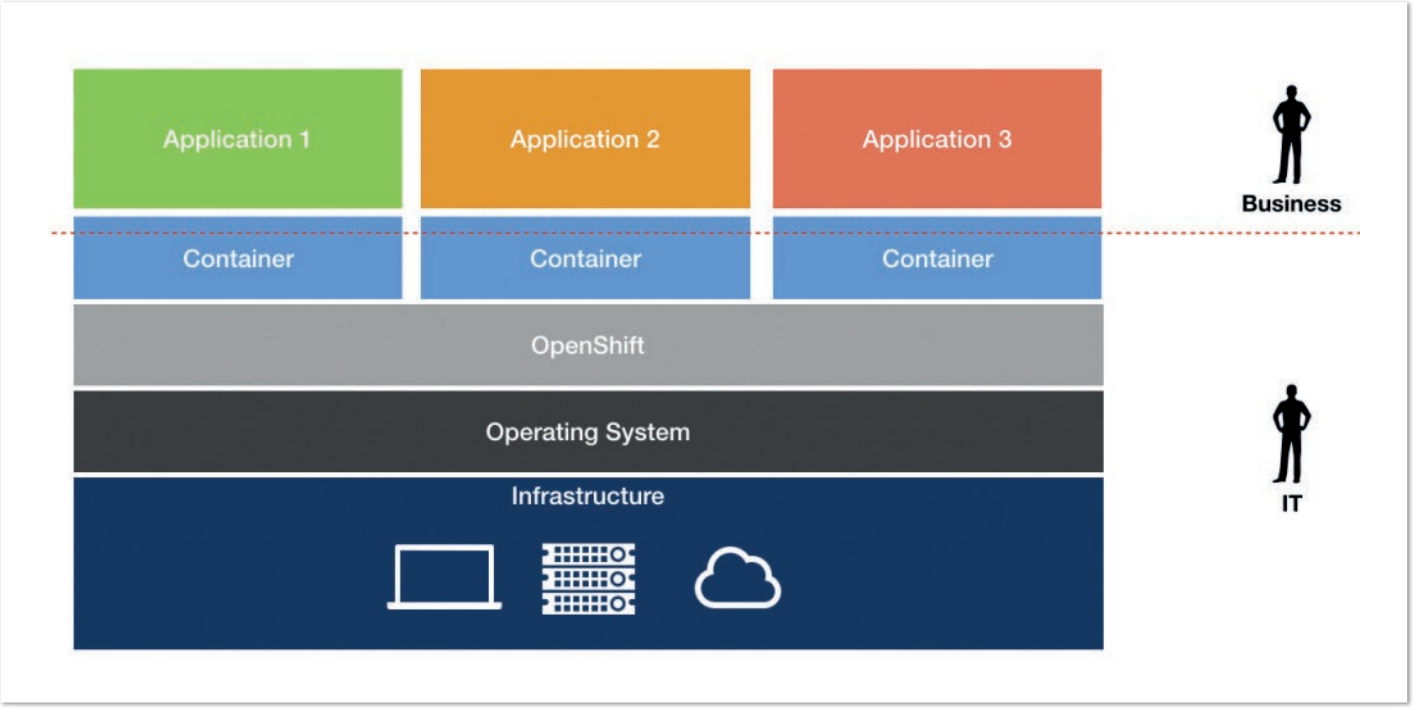


Abbildung 2: Separation of Responsibilities (© Markus Lohn)

Mit Containern kann diese Trennung technisch unterstützt werden *(siehe Abbildung 2)*. Die IT des Kunden verwaltet die Plattform für Container und stellt sicher, dass diese stabil läuft und bei Bedarf skaliert. Die Verantwortung der Anwendungen in den Containern übernehmen definierte Ansprechpartner aus den Fachabteilungen. Somit ist schon allein durch Einsatz einer Container-Management-Plattform eine logische und technische Trennung der Verantwortlichkeiten sichergestellt. Insbesondere können durch die Konfiguration von Namespaces (Projekte) die Anwendungen nach fachlichen Gesichtspunkten gruppiert und unterschiedliche Sicherheitsstufen konfiguriert werden.

Im OSB ist die Trennung der Verantwortlichkeiten so nicht darstellbar. Eine logische Gruppierung der OSB-Services nach fachlichen Gesichtspunkten ist ebenfalls nicht möglich. Obwohl der Oracle Enterprise Manager unterschiedliche Rollen kennt, kann jeder Berechtigte im OSB alle Projekte im Dashboard sehen und verwalten. Insbesondere hat jeder Berechtigte Zugriff auf die erstellten Logfiles, was unter Umständen nicht gewünscht ist. Insbesondere beim Restart eines Managed Servers sind immer alle OSB-Projekte betroffen. Bei Fuse kann jede einzelne Schnittstelle separat verwaltet werden, ohne andere Schnittstellen zu betreffen.

Fazit

Wir haben erfolgreich die Integrationsstrategie von einem zentralen zu einem dezentralen Ansatz vollzogen. Die Nutzung von modernen Entwicklungswerkzeugen und -prinzipien hat erheblich zur Akzeptanz beigetragen. Die Qualität konnte enorm durch den Einsatz von Standardverfahren verbessert werden. Ferner werden nur die absolut notwendigen Komponenten für die Lösung eines Problems herangezogen. Besonders das Betriebsmodell wird durch die neue Plattform auch von technischer Seite optimal unterstützt. Insgesamt betrachtet gibt es viele Alternativen um das Thema Integration zu lösen. Dabei muss nicht immer zwingend ein ESB oder eine komplexe Integrationsumgebung zum Einsatz kommen.

Referenzen

[1] *https://www.oracle.com/de/middleware/technologies/service-bus.html*
[2] *https://www.openshift.com/*
[3] *https://www.redhat.com/de/technologies/jboss-middleware/fuse*
[4] *https://github.com/markuslohn/vagrant-oracle-soa12213-dev*

Markus Lohn
esentri AG
markus.lohn@esentri.com

Markus Lohn sorgt für das reibungslose Zusammenspiel von Anwendungen und Daten und schafft damit die Grundlage für Innovation und Vorsprung. Er unterstützt Kunden bei der Konzeption und Umsetzung von komplexen Integrationsstrategien, API-Management- und Workflow-Lösungen. Mit jahrelanger Erfahrung als Berater ist er fit in den Bereichen Java EE, SOA/Integration, API-Management und Container-Plattformen. Dogmatische Diskussion vermeiden und die Ausrichtung der Entwicklung am tatsächlichen Bedarf sind ihm besonders wichtig.

Policy-as-Code für Cloud-native Anwendungen mit OPA

Andreas Zitzelsberger, QAware

Wir erleben heute mehr Verteilung, höhere Komplexität und mehr Eigenverantwortung für Entwicklungsteams und dementsprechend steigende Sicherheits- und Compliance-Anforderungen. Der Wunsch, diese Anforderungen transparent, nachvollziehbar und wartbar umzusetzen, hat Policy-as-Code entstehen lassen: die Idee, Policies in einer höheren Programmiersprache zu schreiben und zu automatisieren. Der Open Policy Agent (OPA) ist ein prominentes Werkzeug zur Umsetzung von Policy-as-Code aus dem Cloud-native-Ökosystem. Im Folgenden wird OPA vorgestellt und gezeigt, wie Cloud-native Anwendungen mit OPA gesichert werden können.

Der Open Policy Agent (OPA) ist eine leichtgewichtige und vielseitig verwendbare Open Source Policy Engine und bringt mit Rego eine eigene Programmiersprache für Policies mit. Eine Policy in diesem Sinne ist eine Abbildung von strukturierten Bewegungs- und Stammdaten auf ein strukturiertes Ergebnis. Im einfachsten Fall ist das Ergebnis ein Boolean, es kann sich allerdings auch um einen komplexen Objektbaum handeln. Leichtgewichtig bedeutet, dass OPA in seiner Funktionalität stark fokussiert ist und als wenige Megabyte großes Executable sowohl zentral als auch als Sidecar neben Anwendungs-Instanzen betrieben werden kann. Die Kommunikation mit OPA erfolgt über ein REST-API. OPA ist zustandslos und skaliert deshalb trivial horizontal.

OPA löst das Problem der Entscheidungsfindung auf generische Art und Weise, kümmert sich jedoch nicht um das Durchsetzen der Entscheidungen. OPA ist daher keine Business Rules Engine wie Apache Drools oder Camunda, könnte aber in einer Business Rules Engine die Auswertung der Kriterien übernehmen. Diese Fokussierung macht OPA zu einem vielseitig einsetzbaren und nutzbringenden Halbzeug. Gründer des Projekts und treibende Kraft hinter OPA ist die Firma Styra, die um OPA herum ein kommerzielles Produkt zur Verwaltung und Pflege von Policies anbietet.

Wofür eignet sich OPA?

Der Parade-Use-Case für OPA ist Autorisierung. Bei der Autorisierung geht es darum, auf Basis einer attestierten Identität (Authentifizierung) und der mit dieser Identität verbundenen Eigenschaften

(Claims) eine Zutrittsentscheidung zu treffen. Dabei hilft OPA, regelbasierte Autorisierung (RuBAC, Rule-based Authorisation) zu implementieren, die vielseitigste und mächtigste Art der Autorisierung. Wenn, wie in den meisten Organisationen, Rollen für die Autorisierung vorhanden sind, hilft RuBAC, unterschiedliche Dimensionen in den Rollen zu verknüpfen, sie fachlich zu halten und die oft beobachtete Explosion der Rollenzahl zu vermeiden. Mit RuBAC sind aber auch fortschrittlichere Use-Cases möglich, zum Beispiel eine On-Call Policy, die den Mitgliedern einer On-Call-Gruppe nur außerhalb der Geschäftszeiten Zugriff auf ein System gewährt.

Bei der Autorisierung sind einfache Ja/Nein-Entscheidungen gefragt, mit OPA lässt sich allerdings noch mehr machen, etwa Quotas berechnen, Daten filtern oder am anderen Ende der Komplexitätsskala Vertragskonditionen einer Versicherung errechnen. Da OPA komplexe Eingabedaten verarbeiten kann, eignet es sich auch zur Prüfung von Daten, zum Beispiel von strukturierten Konfigurationsdateien. Hierfür gibt es zwei Teilprojekte von OPA, die später in diesem Artikel vorgestellt werden: *Gatekeeper* *[1]* und *Conftest* *[2]*.

Beim Einsatz von OPA muss man sich zwischen zwei wesentlichen Szenarien entscheiden: einem verteilten oder einem zentralen Einsatz. Der verteilte Einsatz als Sidecar hilft, dem CAP-Theorem ein kleines Schnippchen zu schlagen. Die Policies können in diesem Fall auch zentral verwaltet werden, entweder über die OPA-APIs oder etwa als Kubernetes Config Maps. Die Auswertung erfolgt jedoch lokal und resilient gegen Ausfälle zentraler Systeme. Opfert man also ein klein wenig Konsistenz, kann man in der Verfügbarkeit einen großen Sprung machen. Der zentrale Einsatz dagegen bietet unbedingte Konsistenz und ermöglicht, OPA separat zu skalieren – auf Kosten der Verfügbarkeit.

Die Rego-Sprache

Rego ist eine OPA-eigene, von Datalog/Prolog inspirierte Domain-specific Language (DSL) für Policies. Als wohlgestaltete DSL ist Rego eine willkommene Abwechslung zum YAML-Wahnsinn der letzten Jahre. Für imperative Java-Programmierer ist die Sprache gewöhnungsbedürftig, wer allerdings mal Logikprogrammierung gemacht hat, fühlt sich schnell zuhause. Rego ist keine generische Programmiersprache wie Java. So gibt es keine Schleifen, dafür aber List Comprehensions und, abgesehen von der Möglichkeit, externe Web-Services abzufragen, keine Konstrukte mit Nebenwirkungen.

Rego hat Konstanten, die üblichen skalaren Datentypen sowie Arrays, Sets und Objects (Maps). Daneben gibt es Referenzen, die auch *null* sein können. Genauso gibt es den üblichen Satz an logischen und arithmetischen Operatoren, aber auch High-Level-Operatoren wie die Schnittmenge, Vereinigung und Differenz zwischen Sets. Es können Funktionen definiert werden und es gibt eine reichhaltige Standard-Library. Das Regelwerk kann mithilfe von Packages strukturiert werden.

Interessant wird es dort, wo die Ähnlichkeit zur gewohnten imperativen Programmierung aufhört. Das wichtigste Konstrukt in Rego sind Regeln. Regeln folgen dem einfachen Format: `DEFINITION GUARD*` *(siehe Listing 1)*.

Die Guards werden dabei „ver-*oder*-t". Wenn eine der Guards `true` ergibt, so ist die Expression definiert (und per Konvention *true*), an-

```
# DEFINITION     GUARD
cat_alive { true; false}
```

Listing 1: Einfaches Beispiel für eine Regel in Rego

```
ok_roles := [ "developer", "admin" ]

allow[role] {
role := ok_roles[i]
role == input.roles[_]
}
```

Listing 2: Eine Regel mit freien Variablen als Existenzquantoren

sonsten undefiniert (nicht `false`). Die Ausdrücke in einer Guard werden entweder durch Zeilenumbrüche oder Semikolons getrennt. Die Ausdrücke werden dabei „ver-*und*-et". Im obigen Beispiel ist also `cat_alive` *undefined*. Die Definition kann etwas anderes sein, zum Beispiel eine Wertzuweisung oder eine Funktionsdeklaration. Auch in diesen Fällen gilt, dass die Variable beziehungsweise die Funktion nur definiert ist, wenn die Guard `true` ergibt.

Das nächste Beispiel in *Listing 2* ist praxisnäher und zeigt, wie freie Variablen funktionieren. Die Regel enthält hier zwei freie Variablen. _ wirkt dabei als Wildcard, der Wert an dieser Stelle interessiert uns nicht. Die zweite Variable `i` ist der Index im Rollen-Array. Die Regel ist definiert, wenn es eine Belegung der Variablen gibt, die alle Bedingungen erfüllt. In diesem Fall geben wir die gefundene Rolle aus. Wir haben also Existenzquantoren implementiert. Allquantoren können durch die Negation von Existenzquantoren implementiert werden.

Um nun alle passenden Rollen zurückzugeben, können wir zu einer Comprehension greifen *(siehe Listing 3)*. Die Comprehension erzeugt ein Array von Werten, die die Bedingungen erfüllen.

Die Beispiele *(Listing 2 und Listing 3)* zeigen, wie sehr sich Logikprogrammierung von imperativer Programmierung unterscheidet. Zum Verständnis hilft es, im Kopf den Schalter umzulegen – weg von der intuitiven Ausführung einzelner Statements nacheinander und hin dazu, den Code als eine Menge von Einschränkungen zu verstehen, die gleichzeitig erfüllt sein müssen.

Außerdem sollte man sich beim Bauen der Policies nicht verkünsteln und die Regeln les- und wartbaren Codes beachten: Saubere Strukturen, durchdachter Einsatz von Packages, Funktionen und voneinander abhängenden Regeln, sprechende Bezeichner und Kommentare sind wichtig. Wenn man sich daran hält, dann sind die entstehenden Policies kompakt und gerade auch von Nichttechnikern gut les- und verstehbar.

Wenn Sie tiefer einsteigen möchten: Die Rego-Sprache sowie die Standard-Library sind sehr gut und ausführlich dokumentiert *[3]* und der Rego Playground *[4]* bietet die Möglichkeit, Rego direkt im Browser auszuprobieren.

Entwicklung von Policies

OPA bietet gute Grundlagen, um Policies zu entwickeln:

```
ok_roles := [ "developer", "admin" ]

allow[roles] {
roles := [r | r := ok_roles[i]; r == input.roles[_]]
}
```

Listing 3: Eine Regel mit freien Variablen und einer Comprehension

```
test_developer_allowed {
  allow with input as { "roles" : [ "developer" ]}
}

test_manager_denied {
  not allow with input as { "roles" : [ "manager" ]}
}
```

Listing 4: Testfälle in OPA

1. Der REPL-Loop
OPA bietet einen sogenannten Read-Eval-Print-Loop (REPL), also eine Shell, um interaktiv Policies auszuführen. Im einfachsten Fall wird die Shell mit `opa run` gestartet und mit CTRL-D beendet, und schon kann man loslegen. Es ist auch möglich, einen Pfad oder eine Datei mit Policies anzugeben, um dann interaktiv mit den Policies zu arbeiten.

2. Entscheidungs-Logs
OPA kann Entscheidungs-Logs ausgeben, was per Konfiguration an-und ausgeschaltet werden kann, beispielsweise auf der Kommandozeile: `opa run --set decision_logs.console=true`. Diese Logs helfen sowohl bei der Fehlersuche als auch, um Strukturen zu explorieren. Die Entscheidungs-Logs enthalten die vollständigen Eingabedaten. Wenn also etwa JWT-Token an OPA übergeben werden, so tauchen diese auch im Log auf, wenn nicht die eingebaute Maskierungsfunktion verwendet wird.

3. Tracing von Policy-Auswertungen
Mit dem Parameter `explain` beim Stellen von Anfragen an das REST-API kann die Auswertung der Regeln im Detail nachvollzogen werden.

4. Automatische Policy-Tests
OPA bringt ein Framework für Tests mit, das natürlich auch wieder in OPA implementiert ist (denn was sind Tests anderes als Policies?) Um Tests zu erstellen, muss man lediglich ein paar Konventionen folgen. Test-Module werden mit dem Dateinamen-Suffix `_test.rego` versehen. Die Testfälle selbst werden als Regeln mit dem Präfix `test_` angelegt. Das `with`-Schlüsselwort wird verwendet, um Bewegungs- und Stammdaten zu mocken *(siehe Listing 4)*.

Die Tests werden mit `opa test` ausgeführt; `opa --test --coverage` gibt die Testabdeckung aus und `opa test --coverage --format=json` liefert die Testabdeckung im JSON-Format für die maschinelle Weiterverarbeitung etwa in einer CI/CD Pipeline.

5. Benchmarking
OPA hat einfache Benchmark-Fähigkeit eingebaut. Mit `opa bench` kann die Laufzeit einzelner Regeln oder von Testfällen erhoben werden.

6. IDEs
Es gibt Plug-ins für VS Code, IntelliJ IDEA und natürlich auch VIM und andere Editoren mit Syntax-Highlighting, Code-Formatierung, Testunterstützung und Auswertung von Policies.

Integration von OPA

OPA ist plattformunabhängig ausgelegt und hat ein REST-API. Da OPA in Go geschrieben ist, steht alternativ ein natives Go-API zur Verfügung. Policies können auch in Wasm (WebAssembly) kompiliert und dort direkt ausgeführt werden, dabei steht jedoch nur ein eingeschränkter Umfang der Standardbibliothek zur Verfügung.

Der wichtigste Teil der APIs ist das Data-API. Dieses ermöglicht unter anderen, mit einem einfachen POST-Request ein sogenanntes Input-Dokument mit Bewegungsdaten zu übergeben und die Ergebnisse der Regeln eines Package abzufragen. Diese Methode werden wir auch im später folgenden Beispiel verwenden. Das Data-API ermöglicht außerdem, Stammdaten zu verwalten. Daneben gibt es:

- Das Query-API für das Stellen einfacher Anfragen ohne Bewegungsdaten und für Ad-hoc-Queries, bei denen OPA-Ausdrücke ausgewertet werden.
- Das Compile-API, das die partielle Auswertung von Policies erlaubt, womit komplexe Use-Cases und Skalierung ermöglicht wird.
- Das Health-API, das Health Checks zur Verfügung stellt.
- Einen Prometheus-Endpoint, um Performance-KPIs zu erheben.

Die APIs ermöglichen aber noch mehr Use-Cases, wie das Verwalten von Stammdaten, das Ausführen von Ad-hoc-Queries, die partielle Auswertung von Policies etc. OPA kann optional auch vordefinierte APIs konsumieren:

- Über das Bundle-API können Policies automatisch von einem zentralen REST-Service bezogen werden.
- OPA kann regelmäßig Status-Updates an eine Implementierung des Status-Service-API schicken.
- Für Auditierbarkeit in der Tiefe gibt es das Decision-Log-Service-API. OPA schickt detaillierte Entscheidungslogs an dieses API.

OPA Conftest

Conftest ist ein Kommandozeilen-Werkzeug, mit dem sich beliebige Konfigurationsdateien in einer Vielzahl von Sprachen prüfen lassen. *Conftest* kann sehr einfach als Teil einer Continuous Integration oder Continuous Deployment Pipeline eingesetzt werden oder als Git Commit Hook Entwicklern schnelles Feedback geben. Besonders wertvoll wird *Conftest* in Zusammenhang mit einem GitOps-/Infrastructure-as-Code-Ansatz. Damit können die Infrastrukturdefinitionen und die Konfiguration selbst geprüft und auditiert werden.

OPA Gatekeeper

OPA *Gatekeeper* prüft Änderungen an einer Kubernetes-Laufzeitumgebung gegen Policies. *Gatekeeper* wird dazu in Kubernetes als Admission Controller installiert und kann sowohl eigehende Änderungen prüfen und verhindern als auch Auskunft darüber geben, welche Inhalte eines Clusters gegen Policies verstoßen, etwa um ein Cluster zu auditieren oder Policy-Änderungen vorzubereiten. Im Gegensatz zu OPA selbst läuft *Gatekeeper* nicht direkt mit Rego-Policies, sondern mit dem OPA Constraint Framework. Das Framework macht Policies in Form sogenannter Constraint Templates zugänglich. Die Policy kann dann über eine einfache YAML-Konfiguration referenziert werden. *Gatekeeper* bringt bereits eine breite und erweiterbare Policy-Library mit.

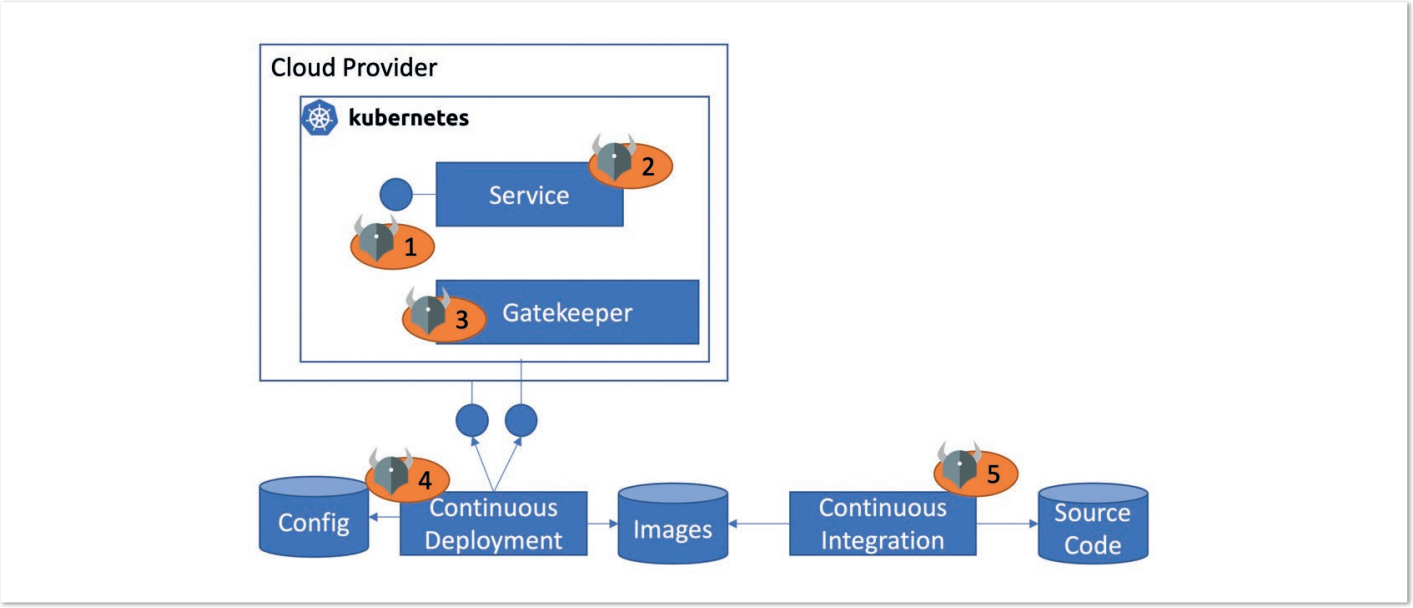


Abbildung 1: Wie sich OPA in einer Cloud-Umgebung mit GitOps einsetzen lässt (© Andreas Zitzelsberger)

```
http.authorizeRequests()
    .anyRequest()
    .authenticated()
    .accessDecisionManager(new UnanimousBased(singletonList(new OpaVoter(OPA_URL))))
.and().oauth2Login();
```

Listing 5: HTTP-Security-Konfiguration mit Delegation an OPA

Conftest und *Gatekeeper* sind Werkzeuge, die als zentrale Bausteine Sinn machen, um über Entwicklungsteams hinweg Vorgaben durchzusetzen. Dabei gilt: Weniger ist besser. Lieber wenige, sauber durchdachte Policies, die nur an den Stellen Aktionen verhindern, an denen es für unbedingt notwendig erachtet wird. Häufig ist es besser, mit Warnungen und Benachrichtigungen zu arbeiten.

Absicherung einer Cloud-nativen Anwendung

Abbildung 1 zeigt eine moderne Umgebung mit Kubernetes bei einem Cloud Provider, GitOps und CI/CD. OPA kann an mehreren Stellen Nutzen stiften: Zur…

- Autorisierung von Zugriffen auf die Anwendung,
- Implementierung von Policies in der Anwendung,
- Absicherung von Cluster-Zugriffen mit dem Gatekeeper,
- Prüfung und Auditierung von Infrastructure-as-Code in einer CD-Pipeline,
- Prüfung und Auditierung der Anwendungskonfiguration in einer CI-Pipeline.

Als konkretes Beispiel wollen wir einen einfachen Service mit Java/Spring Boot absichern. Das vollständige Beispiel findet sich auf GitHub *[5]*. Bevor wir uns Gedanken über Autorisierung und Policies machen, brauchen wir attestierte Identitäten der uns aufrufenden Nutzer und Services.

Für Nutzerauthentifizierung ist OpenID Connect mit dem Authorization Code Flow empfehlenswert, worauf wir auch im Beispiel aufbauen. OpenID Connect ist ebenfalls für Service-Authentifizierung geeignet. Oft wird dafür auch mTLS verwendet, was Spring Security ebenfalls unterstützt. Wenn Sie ein Service Mesh wie Istio einsetzen oder eingehende TLS-Verbindungen bereits beim Ingress terminieren, so werden die Identitäten gerne in einem HTTP-Header übergeben. Gebräuchlich sind etwa der `X-Remote-User`-Header für Nutzer-IDs oder der `X-Forwarded-Client-Cert`-Header für komplette Zertifikate. Auch das ist mit Spring Security einfach möglich.

Im allerersten Schritt können wir mit dem Spring Initializr *[6]* einen neuen Service mit den Dependencies Spring Web, Security und OAuth2 Client erzeugen. Empfehlenswert ist außerdem Spring Actuator, damit der Health Check nicht selbst gebaut werden muss.

Dann konfigurieren wir den OAuth2/OpenID Connect Client. Im Beispiel verwenden wir GitHub als OpenID Connect Provider, dazu ist ein GitHub-Account notwendig. Sie können sich auf GitHub *[7]* eine neue OAuth-Anwendung anlegen. Die Zugangsdaten tragen Sie in die Spring-Konfiguration unter `spring.security.oauth2.client.registration.github.clientId` und `spring.security.oauth2.client.registration.github.clientSecret` ein.

```
anon_paths := ["/", "/webjars/*"]
auth_paths := ["/norris/fact", "/user/name" ]

allow { regex.match(anon_paths[_], input.path) }
allow {
    input.auth.authenticated == true
    regex.match(auth_paths[_], input.path)
}
```

Listing 6: access-policy.rego

```
package main

deny[msg] {
  input.spring.security.oauth2.client.registration.google
  msg := "Google is disallowed as an identity provider"
}
```

Listing 7: config-policy.rego

```
$ conftest -p config-policy.rego test src/main/resources/application.yaml
FAIL - src/main/resources/application.yaml - Google is disallowed as an identity provider
```

Listing 8: conftest sagt: kein Google für unseren Service

Dann brauchen wir einen `WebSecurityConfigurerAdapter`. In der Methode *configure* aktivieren wir den OAuth2-Login und delegieren den kompletten Autorisierungsprozess an OPA, indem wir einen `AccessDecisionVoter` implementieren. Ein `AccessDecisionVoter` entscheidet auf Basis des Ergebnisses der Authentifizierung und des aktuellen Requests, ob Zugriff gewährt wird. Unser Voter reicht in einem POST-Aufruf die HTTP-Methode, den Aufrufpfad, den HTTP-Header und die Nutzer-Identität im Input-Dokument an OPAs Data-API weiter und gewährt Zugriff abhängig von der Entscheidung von OPA. Wir tragen unseren `OpaVoter` als einzigen Voter in die `WebSecurityConfig` ein *(siehe Listing 5)*.

Nun können wir unsere Policy anlegen. Im Beispiel verwenden wir eine einfache Policy, die anonymen Zugriff auf die Index-Seite und statische Assets sowie authentifizierten Zugriff auf zwei Web Services erlaubt *(siehe Listing 6)*.

Um das Ganze auszuprobieren, starten wir lokal OPA als Server mit dem Befehl `opa run -s access-policy.rego` und dann den Spring Boot Service mit `./gradlew bootRun` und können unseren neuen Service unter `http://localhost:8080` testen.

Aufbauend darauf können wir nun beliebig Policies auf Basis des eingehenden Requests gestalten. Da im Input-Dokument auch die Claims des ID-Tokens vorliegen, können diese in den Policies verwendet werden. Möchten Sie zum Beispiel mit GitHub den Zugriff auf Angehörige einer Firma einschränken, so können Sie das mit dem GitHub-spezifischen „Company"-Claim tun. Analog können wir die weit verbreiteten „Roles"- oder „Groups"-Claims verwenden, wenn der Identity Provider wie Keycloak diese Claims unterstützt. Genauso lassen sich die oben angerissenen Use-Cases mit mTLS, Identität im Header oder Bearer-Token umsetzen.

Im nächsten Schritt wollen wir OPA *Conftest* einsetzen, um unsere Anwendungskonfiguration zu prüfen. Nehmen wir an, wir wollten Google als Identity Provider verbieten. Dann müssen wir eine entsprechende Policy formulieren *(siehe Listing 7)*. Im Anschluss rufen wir *Conftest* auf *(siehe Listing 8)*.

Gatekeeper werden wir im Beispiel nicht betrachten, das bleibt als Übung für motivierte Leser. Zwei Hinweise dazu: *Gatekeeper* ist ein Kubernetes Admission Controller, deshalb muss es im Ziel-Cluster möglich sein, einen Admission Controller zu installieren, was in manchen Kubernetes-Umgebungen unterbunden wird. Ein Admission Controller bietet eine wunderbare Möglichkeit, sich in den Fuß zu schießen, da es ohne Weiteres möglich ist, Policies zu formulieren, die einen Cluster unbenutzbar machen. Es ist natürlich auch möglich, Policies zu formulieren, die solche gefährlichen Änderungen verhindern. Für die ersten Gehversuche empfiehlt sich auf jeden Fall eine dedizierte Testumgebung, lokal zum Beispiel mit Minikube, oder ein Katacoda-Tutorial *[8]*.

Fazit

OPA ist ein klar fokussiertes und durchdachtes Werkzeug, dass in seiner Domäne alles richtig macht. OPA kann durch das Ermöglichen von Policy-as-Code mehr Sicherheit und Nachvollziehbarkeit bringen und in Form von *Conftest* und *Gatekeeper* sicheres DevOps erlauben. Dabei hilft die Rego-Sprache, Policies aus dem Dunkel verstreuter Implementierungen zu holen und Transparenz auch herzustellen. Dabei bleibt OPA angenehm einfach zu nutzen und zu integrieren.

Quellen

[1] OPA Gatekeeper: *https://github.com/open-policy-agent/gatekeeper*
[2] OPA Conftest: *https://www.conftest.dev/*
[3] OPA-Dokumentation: *https://www.openpolicyagent.org/docs*
[4] Rego Playground: *https://play.openpolicyagent.org/*
[5] Beispiel-Anwendung: *https://github.com/az82/opa-secure-microservice*
[6] *https://start.spring.io/*
[7] *https://github.com/settings/applications*
[8] OPA Gatekeeper Katacoda Tutorial: *https://katacoda.com/austinheiman/scenarios/open-policy-agent-gatekeeper*

Andreas Zitzelsberger
QAware GmbH
andreas.zitzelsberger@qaware.de

Andreas Zitzelsberger ist Technischer Geschäftsbereichsleiter bei QAware sowie Entwickler und Architekt aus Leidenschaft. Seine Schwerpunkte sind der Bau von Cloud-nativen Anwendungen und Sicherheit in der Cloud.

Continuous Deployment ganz ohne Angst

Christian Uhl, Personio GmbH

Continuous Delivery ist ein relativ bekanntes Konzept – wird aber vielerorts noch skeptisch betrachtet. Nachdem ich Continuous Delivery erfolgreich bei einem Unternehmen eingeführt habe und lange mit einer solchen Infrastruktur Software geliefert habe, kann ich mir nicht mehr vorstellen, darauf zu verzichten. In diesem Artikel will ich einen sicheren Weg hin zu einer Continuous Deployment Pipeline aufzeigen und die Vorteile herausstellen.

Die Runway als Metapher

Im Startup-Umfeld, besonders bei fremdfinanzierten, wird gern die Metapher eines „Runway" (Startbahn für Flugzeuge) verwendet. Man kann damit gut die Kernproblematik ausdrücken: Es gibt eine klar begrenzte Strecke, die man zur Beschleunigung nutzen kann – dann hebt man entweder ab (man hat genug Umsatz, um sich von selbst in der Luft zu halten) oder man knallt gegen einen Berg (die Finanzreserven sind aufgebraucht). So nützlich diese Idee ist, greift sie meiner Meinung nach zu kurz – ein Runway ist schnurgerade, und das echte Leben ist selten so. Fast jede Unternehmung geht nicht ganz so vonstatten, wie es am Anfang geplant war, und man muss im Zweifel einen schnellen Haken schlagen können, indem man sein Produkt an die Bedürfnisse des Markts anpasst.

Dies ist nicht nur in den Anfangsphasen eines Startups notwendig – jedes digitale Produkt ist dem Markt ausgesetzt, und die Rahmenbedingungen können sich plötzlich ändern – ein disruptives Konkurrenzprodukt, veränderte gesetzliche Rahmenbedingungen oder technologische Innovationen bedrohen etablierte Produkte und Unternehmen genauso.

Um diesen Anforderungen an Agilität gerecht zu werden, sollte man sich meiner Meinung nach die Umgebung so professionell einrichten, dass keinerlei „Verschwendung" im Prozess mehr entsteht – um möglichst viel seiner eingesetzten Kapazität in wertschöpfende Funktionalität investieren zu können. Continuous Deployment ist ein wichtiger Baustein eines solchen Setups, denn es ermöglicht uns, die Feedback-Zyklen extrem klein zu halten.

Je schneller eine Funktionalität vom Rechner des Entwicklers zu den Benutzern einer Software kommt, desto schneller hat man auch wieder wertvolles Feedback in den Händen – die Gefahr, das falsche Produkt zu bauen, sinkt.

Begriffsklärung: Continuous Was-denn-noch?

Es gibt mehrere aufeinander aufbauende Konzepte im Namensraum „Continuous":

Continuous Integration

Die älteste Idee, die bis in das Jahr 1991 und Grady Booch zurückreicht, beschreibt die Vorgehensweise, den Code, den verschiedene Entwickler im selben Projekt erstellen, regelmäßig zu mergen und zu kompilieren. Die Idee dahinter ist, dass die verschiedenen Code-Kopien (etwa Branches) nicht zu weit voneinander abweichen und man sich den späteren, sehr großen Integrationsaufwand spart, der ansonsten entstehen würde – die sprichwörtliche „Merge Hell". Extreme Programming hat diese Idee weiter aufgenommen und empfiehlt, dies mehrfach täglich zu tun. Das konkrete Ergebnis von CI ist aber kein kompiliertes Softwarepaket beziehungsweise Anwendung, sondern nur ein Hauptzweig der Codebasis, der immer kompilierbar ist und die neuesten Änderungen beinhaltet. Als Werkzeug bietet es sich an, eine dedizierte Maschine zu betreiben, die auf Codeänderungen im Hauptzweig lauscht und bei Änderungen die Kompilier- und Testprozesse ausführt, zum Beispiel Jenkins, Bamboo, TravisCI, GitLab CI und viele weitere.

Continuous Delivery

In einer Continuous-Delivery-Umgebung wird die Anwendung nicht nur kompiliert, sondern auch paketiert und in einen auslieferungsfähigen Zustand gebracht (etwa in einen Container gepackt und in einer Container Registry abgelegt). Eine Deployment-Pipeline übernimmt die Schritte nach Kompilierung und Test; sie kann die Software schon in einer produktionsähnlichen Umgebung installieren. Für die finale Veröffentlichung ist aber noch menschlicher Eingriff notwendig – am besten in Form eines einzelnen Knopfes, den man drücken muss, um die getestete Version zu deployen.

Continuous Deployment

Die Königsdisziplin: Die Software wird nach dem Durchlaufen aller vorherigen Prozessschritte von Continuous Integration und Continuous Delivery direkt in die Produktionsumgebung installiert und ist für die Kunden und Konsumenten verfügbar. Es ist kein menschlicher Eingriff mehr vonnöten.

Das Optimierungsdilemma

Es gibt verschiedene Metriken, mit denen man die Verlässlichkeit seines Softwareentwicklungsprozesses messen kann. Ich möchte zwei davon herausgreifen, weil sie das Optimierungsdilemma schön illustrieren:

Die meisten Software-Release-Prozesse außerhalb von Continuous Deployment versuchen, die *„Mean Time between Failures"* (MTBF) zu verbessern. Damit kann man messen, wie groß der Median des Zeitraums zwischen zwei schwerwiegenden Problemen der Software ist. Eine Nicht-Verfügbarkeit der Anwendung, Datenverlust oder ein Bug, der wichtige Arbeitsvorgänge unmöglich macht, können hier als Fehler zählen.

Um diesen Wert zu verbessern, versucht man, die Zeitspanne zwischen zwei Fehlern möglichst groß zu halten – indem man Änderungen selten, und nur nach genauen und umfangreichen Tests und Verifikationen, vornimmt. Diese Maßnahmen reduzieren die totale Anzahl von Fehlern, können aber die Zeit, die ein System im Fehlerfall verbringt (bis eine Korrektur veröffentlicht wurde) stark verlängern.

Ein weiteres Konzept dazu ist *„Mean Time to Recovery"* (MTTR). Dieser Wert beschreibt den Median der Zeit, die benötigt wird, um von einem fehlerhaften Zustand des Systems wieder zu einem korrekten Zustand zu kommen, etwa indem man einen Bugfix veröffentlicht. Leider stehen beide Metriken häufig miteinander in Konflikt: Viele Maßnahmen, die die MTTR positiv beeinflussen, können einen negativen Einfluss auf die MTBF haben und umgekehrt.

Die gute Nachricht ist allerdings, dass Continuous Deployment nicht nur die relative, sondern sogar die absolute Häufigkeit von Fehlern im System reduziert. Die statistische Untersuchung dazu findet sich im Buch „Accelerate: The Science of Lean Software and DevOps" *[1]*.

Wir deployen freitags!

Es ist fast schon ein wiederkehrender Witz in vielen Unternehmen, dass freitags keine Änderungen ausgespielt werden. In einem unsicheren Umfeld, bei dem man nicht weiß, ob ein Deployment Schaden anrichten kann, ist das mit Sicherheit auch die richtige Entscheidung – niemand möchte am Sonntagmorgen aus dem Bett geklingelt werden, weil die Software Fehler wirft.

Eine „Wir-deployen-nicht-am-Freitag"-Regel sollte aber nie als etwas Finales verstanden werden – sondern als Warnsignal. Sie bedeutet,

	Elite	High	Mid	Low
Deployment-Frequenz	> 1 pro Stunde	1/Stunde bis 1/Tag	1/Woche bis 1/Monat	1/Woche bis 1/Monat
Lead Time	< 1 Stunde	1 Tag bis 1 Woche	1 Woche bis 1 Monat	1 Monat bis 6 Monate
MTTR	< 1 Stunde	< 1 Tag	< 1 Tag	1 Woche bis 1 Monat
Change Failure Rate	0 - 15 %	0 - 15 %	0 - 15 %	46 - 60 %

Tabelle 1: Eine höhere Deployment-Frequenz korreliert mit einer verbesserten MTTR und einer geringeren Change Failure Rate

	Elite	High	Mid	Low
Neue Funktionalität	50 %	50 %	40 %	30 %
Nachbearbeitung	19,5 %	20 %	20 %	20 %
Security	5 %	5 %	5 %	10 %
Fehlerbehebung	10 %	10 %	10 %	20 %
Support	5 %	10 %	10 %	15 %

Tabelle 2: Eine höhere Deployment-Frequenz (für die Gruppen „Elite" und „High") bedeutet auch mehr Zeit für neue Funktionalität zur Verfügung zu haben – und damit eine höhere Produktivität des Teams

dass der Release-Prozess zu unsicher ist und man sich nicht darauf verlassen kann. Das kann zum Beispiel bedeuten, dass die Tests nicht aussagekräftig genug sind, um zu beweisen, dass die Produkte-Iteration keine Regression enthält oder dass das Deployment fehlschlagen könnte. Man sollte also das Vorhandensein dieser Regel zum Anlass nehmen, den eigenen Prozess nochmal kritisch zu hinterfragen und so lange zu verbessern, bis die Regel ihre Berechtigung verliert – und abgeschafft werden kann.

Messbar positive Effekte von Continuous Deployment

Das schnelle und sichere Ausrollen von Software ist nicht nur angenehm für die Entwickler und Nutzer gleichermaßen, sondern hat messbar positive Effekte auf den Unternehmenserfolg, wie man aus den beiden Tabellen *[2]* entnehmen kann.

Ein weiterer sehr interessanter Fakt aus den Untersuchungen ist die Tatsache, dass eine Organisation, die auf einem fortgeschrittenen Delivery-Level arbeitet *(siehe Tabelle 2)*, deutlich mehr Zeit für „New Work", also wertschöpfende Arbeit, zur Verfügung hat *[2]*.

Der Weg zum kontinuierlichen Release

Automatisierung – für alles!

Der erste Schritt hin zu kontinuierlichen Deployments müssen zuerst vollständig automatisierte Deployments sein – und das ist auch für alle, die den Weg nicht ganz bis zum Schluss mitgehen möchten, eine sinnvolle Investition. Automatisierte Deployments sind sicherer und schneller als manuelle Handgriffe. Gerade wenn etwas schiefgelaufen ist und man ganz schnell um 21:00 Uhr am Freitag noch einen Bugfix in Produktion bringen muss, möchte man einen sicheren und erprobten Prozess haben. Ansonsten läuft man Gefahr, in der Hektik einer gefährlichen Situation mit einem fehlerhaften Hotfix noch größeren Schaden anzurichten.

Wer seiner Continuous-Integration-Umgebung noch nicht traut, kann jederzeit von seiner lokalen Maschine aus deployen – Hauptsache, es ist nur ein `./deploy`-Kommando.

Tests unter Kontrolle bringen

Die meisten Softwareprojekte haben mittlerweile eine gewisse Testabdeckung – aber die Gretchenfrage ist in diesem Fall immer: Kann man den Tests vertrauen? Bedeutet ein fehlerfreier Lauf der Testrunners, dass die Software auch garantiert und zu 100 Prozent funktioniert? Dies bedeutet nicht unbedingt, eine Testabdeckung von 100 Prozent zu erreichen – aber die wichtigen Pfade sollten abgedeckt sein, nicht nur im Happy Path (also in dem Fall, in dem alles klappt), sondern auch typische Fehlerszenarien sollten überprüft sein.

Weiterhin darf die Testlaufzeit nicht eskalieren. Wenn die Pipeline drei Stunden dauert, limitiert das die Frequenz, in der deployt werden kann, zu sehr. Außerdem möchte man im Fall eines Hotfixes nicht so lange warten und die Verlockung, die Tests „dieses eine Mal" zu überspringen, ist zu groß.

Ein besserer Review-Prozess

In meinem bisherigen Werdegang waren die größten Probleme nach einem Release selten, dass das Feature nicht funktioniert hat (auch wenn ich mehr als genug Bugs für ein ganzes Leben veröffentlicht habe), sondern viel schwerwiegender war, dass das Feature nicht wie gewünscht umgesetzt wurde. Je besser und schneller die Feedback-Zyklen sind, desto früher lässt sich so etwas erkennen. Ich habe gute Erfahrung damit gemacht, jede potenzielle Änderung auf Knopfdruck in eine eigene Umgebung veröffentlichen zu können – die besten Konversationen hat man zwischen Entwicklung und Produkt immer, wenn man gemeinsam auf funktionierende Software blickt. Selbstverständlich ist der Release hinter einem Feature Toggle eine genauso gute Lösung, wenn man eine gute Feature-Toggle-Infrastruktur besitzt.

Observability einführen

Wenn man konstant automatisiert neue Änderungen an der Software ausspielt, muss man auch zwingend wissen, wie sich diese Änderungen ausgewirkt haben, denn niemand kann bei einer solchen Geschwindigkeit noch alles manuell überprüfen. Es ist also

notwendig, den Systemstatus automatisiert im Blick zu behalten und bei Problemen sofort Alarm zu schlagen (oder den letzten Deploy zurückzunehmen, siehe „Canary Releases").

Der Begriff „Observability" ist derzeit noch umkämpft und wird viel von Marketing getrieben, aber die Kernidee ist, dass man jederzeit weiß, ob das System in Ordnung ist und bei Problemen sofort informiert wird – und dann die Werkzeuge zur Verfügung hat, um dem Problem auf den Grund zu gehen. Typischerweise werden hier Logs, Metriken und Traces verwendet.

Welche Tools konkret verwendet werden, ist Geschmackssache. Der wirklich wichtige Teil ist, dass die generierten Daten auch ausgewertet werden, dem jeweiligen Entwicklungsteam zur Verfügung stehen und das Alerting auch zu den Entwicklern durchschlägt. Das Konzept des „You build it, you run it" zahlt sich auf Dauer aus: Nur ein Team, das auch „fühlt", was die Software in Produktion tut, kann langfristig die Qualität auf ein Niveau, das Continuous Deployment erlaubt, heben und halten.

Eine nette Methodik, die mir begegnet ist und die ich aufgegriffen habe, ist die Rolle der „Cops of the Day". Ein zufälliges Paar von Entwicklern geht jeden Morgen eine Checkliste durch, um zu sehen, ob sich das System in den letzten 24 Stunden gut benommen hat und ob es Anzeichen für spätere Probleme gibt – etwa steigende Antwortzeiten, Exceptions in den Logs oder langsam volllaufende Queues.

Ein weiterer Trick, der mir geholfen hat, eine stärkere Verbindung und eine Brücke zwischen dem Entwicklungsteam und der Software in Produktion zu schaffen, ist das „ChatOps"-Konzept – das Verbinden der Software mit dem Kommunikationstool der Firma (beispielsweise Slack oder Microsoft Teams). Gerade Notifikationen über interessante Business-Events (Benutzer registriert sich, ein Kauf wurde getätigt, eine Subscription wurde verlängert etc.) sind einfach zu integrieren und können eine Verbindung zwischen der täglichen Arbeit des Teams und den Effekten schaffen. Dies führt zu mehr Empathie für die Produktionsumgebung und langfristig zu höherer Qualität.

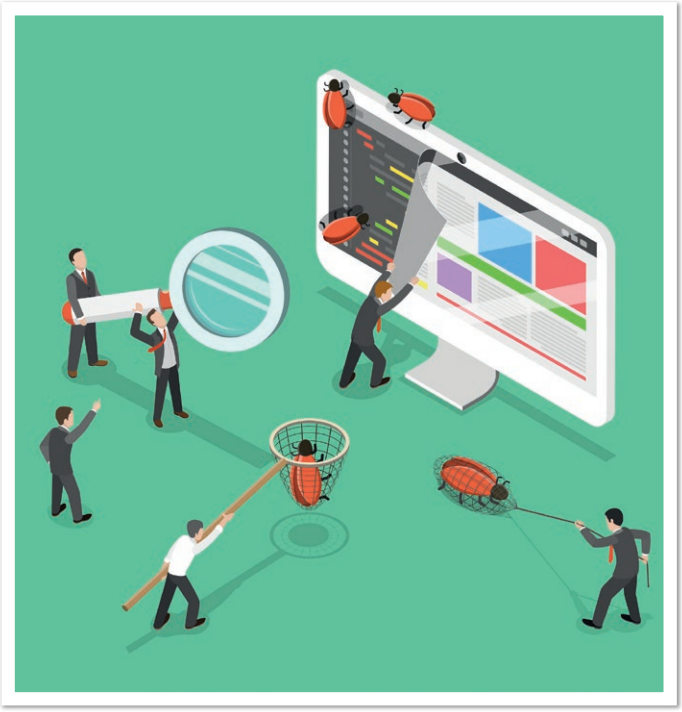
Ansätze, die mir in früheren Teams geholfen haben, sind parallelisierende Testrunner zu verwenden, die verschiedenen Test-Schichten (Unit, Integration und End-to-End) unabhängig voneinander in Build-Steps abzubilden und mit dynamisch provisionierten Build-Maschinen zu arbeiten. Eine Spot-Instanz nur für die Dauer eines Builds kostet nicht viel Geld und man kann eine größere Instanz wählen.

Den Release-Prozess verbessern

Der Release-Prozess ist natürlich Kernstück jeder Continuous-Deployment-Strategie. Viele Schwächen im Prozess sind durchaus akzeptabel bei einer zweiwöchentlichen Release-Kadenz, aber bei 20 Deploys am Tag schon nicht mehr – und das kann ein einzelnes, gut eingespieltes Team durchaus erreichen. Im Folgenden diskutieren wir verschiedene Konzepte, die in Summe einen resilienten Release-Prozess ergeben, der Continuous Deployment erlaubt.

Rolling Update

Ein Deploy einer Anwendung, die nur als eine Instanz gleichzeitig laufen kann, geht immer mit einer kurzen Downtime einher. Dieses Problem potenziert sich bei Continuous Deployments und führt zu nicht akzeptablen Unterbrechungen für die Nutzer. Daher ist es unumgänglich, sicherzugehen oder die Architektur soweit zu verändern, dass die Anwendung in mehreren Instanzen gleichzeitig laufen kann. Dann kann man ein rollendes Update durchführen: Während die alte Instanz noch läuft und Anfragen bearbeitet, startet die neue Version schon – nach erfolgreichem Start wird der Traffic umgeleitet und die alte Version heruntergefahren. Wenn man mehrere Instanzen einer Anwendung betreibt, kann man auch Stück für Stück die Instanzen tauschen. So ergibt sich keine Einschränkung der Nutzer, auch bei häufigen Releases.



Wichtig ist hier, die eventuell anfallenden Datenbank-Migrationen im Griff zu behalten, denn es greifen über einen gewissen Zeitraum beide Versionen auf die Datenbank zu. Es empfiehlt sich, die Migration von der Software-Änderung zu trennen und es auf zwei Releases aufzuteilen, um die Kompatibilität zu wahren.

Rights-Based Releases/Feature Toggles

Ein sehr mächtiges Konzept, das Continuous Deployment erst ermöglicht, ist es, Deployment und Release voneinander zu entkoppeln. Deployment bedeutet, dass eine Codeänderung in Produktion läuft – aber sie muss deswegen noch nicht von Kunden benutzt werden – das ist das eigentliche Release.

Hierfür helfen Feature Toggles. Sie erlauben es, zwei parallele Codepfade in der Anwendung zu haben; je nach Einstellung des „Schalters" wird zwischen diesen Pfaden umgeschaltet. Dieses Schalten kann im Idealfall entkoppelt von Deployments über eine Konfigurationsoberfläche erfolgen. So kann man auch im Fehlerfall oder bei Nichtgefallen sehr schnell wieder zum vorherigen Verhalten der Anwendung zurückkehren.

Eine darauf aufbauende Idee sind rechtebasierte Releases. Eine Änderung an der Anwendung ist zuerst über das Feature Toggle versteckt und wird dann progressiv mehr und mehr Nutzergruppen zur

Verfügung gestellt – beispielsweise erst interne Anwender, dann ausgesuchte Beta-Kunden und erst dann die gesamte Nutzerbasis.

In allen Fällen ist zu beachten, dass die Toggle-Pfade implementiert und hinterher auch wieder aufgeräumt werden müssen. Jedes Toggle-basierte Deployment erzeugt technische Schuld, die wieder abgearbeitet werden muss.

Dark Releases

Manchmal kann es hilfreich sein, eine neue Funktionalität zuerst entkoppelt von ihrer Integration zu deployen, um blockierende Abhängigkeiten aufzulösen. Ein einfaches Beispiel ist der Start eines neuen Service in einer Microservices-Architektur – die Funktionalität des Service wird noch von niemandem verwendet, aber die ersten Iterationen werden dennoch schon deployt. Damit kann das Team sehr schnell iterieren, aber man verschiebt den Integrationszeitpunkt, beispielsweise von Frontend und Backend, nach hinten – mit den Problematiken, die im Kapitel über Continuous Integration erwähnt wurden.

Canary Deployments

Auch die beste Testabdeckung hilft nicht immer und in der Realität wird es immer wieder Bugs geben, für die es keinen Test gibt. Um Fehler, die es tatsächlich in die Produktion schaffen, in ihrer Wirkung zu beschränken, kann man mit Canary Releases als Erweiterung zu Rolling Updates arbeiten. Bei einem Deployment einer Anwendung, die in mehreren Instanzen läuft, wird zuerst nur eine Instanz mit der neuen Version gestartet. Diese erhält einen Teil des Traffics und wird von der Deployment-Engine überwacht – gibt die neue Version schlechte Signale von sich (etwa mehr Exceptions oder HTTP-500-Statuscodes), wird das Deployment abgebrochen und das Entwicklungsteam informiert. So wird sichergestellt, dass die Probleme nur kurz und isoliert auftauchen.

Die Angst loswerden

Wenn man nun den Release- und Deployment-Prozess mit den vorher erwähnten Maßnahmen hinreichend verlässlich bekommen hat, empfehle ich, es noch eine Weile bei Continuous Delivery zu belassen und manuell ein Deployment vorzunehmen, so oft man kann. Ziel der Übung ist es, das Drücken des Knopfes ein absolut langweiliges Event werden zu lassen, denn die Automatisierung funktioniert ja. Am besten überlässt es man der Person, die Continuous Deployment skeptisch gegenübersteht, den Knopf die ganze Zeit zu drücken.

Mit der Zeit wird das von ganz allein zu langweilig und aufwendig. Jetzt kann man ganz entspannt die manuelle Hürde wegnehmen und die Vorteile des Continuous-Deployment-Prozesses genießen.

Fazit

Es bedarf also recht vieler Schritte und Prozessverbesserungen, um Continuous Deployment einführen zu können – aber alle sind bereits als Best Practices anerkannt, die auch ohne Continuous Deployment deutliche Verbesserungen im Softwareprozess sind, von denen man individuell profitiert. Wenn man es dann geschafft hat, kann ich berichten, dass man nie wieder zurück möchte. Die früheren Arten, Software auszuliefern, fühlen sich archaisch an und stehen einem bei dem im Weg, worum es eigentlich geht: Richtig gute Software entwickeln!

Quellen

[1] Nicole Forsgren, Jez Humble and Gene Kim (2018): *Accelerate: The Science of Lean Software and DevOps.* IT Revolution, Portland OR

[2] DORA: The State of Devops Report: *https://inthecloud.withgoogle.com/state-of-devops-18/dl-cd.html*

Christian Uhl
Personio GmbH
Christian.uhl@personio.de

Christian war einige Jahre lang als Senior Software Consultant bei der codecentric AG und als Head of Engineering bei Matmatch GmbH tätig, bevor er zu Personio kam. Mittlerweile hat er eine große Vielfalt an verteilten Systemen gesehen und eine Vielzahl von Fehlern gemacht, die er jetzt teilen kann. Er interessiert sich besonders für verteilte Systeme und Microservices-Architekturen und ist immer neugierig, wie all diese Teile zusammenpassen.

Coding Continuous Delivery: CIOps vs. GitOps mit Jenkins

Johannes Schnatterer und Daniel Huchthausen, Cloudogu GmbH

Continuous Delivery (CD) hat sich im Umfeld agiler Softwareentwicklung als adäquates Vorgehen erwiesen, qualitativ hochwertige Software in kurzen Zyklen zuverlässig und wiederholbar zu veröffentlichen. Der Einsatz von Containern und Cloud, beispielsweise auf Plattformen wie Kubernetes (K8s), bietet viele Möglichkeiten, um CD-Prozesse robuster und einfacher zu gestalten. Eine solche Möglichkeit ist GitOps. In diesem Artikel werden die Unterschiede zwischen klassischen CD-Pipelines (CIOps) und GitOps-Prozessen anhand von konkreten Beispielen aufgezeigt.

Die Automatisierung bei CD erfolgt mittels Pipelines in Continuous-Integration-(CI)-Servern wie Jenkins. Dabei gibt es zwei Anwendungsfälle, bei denen der Einsatz von Containern Vorteile bietet:

1. Bei der Ausführung der Pipeline können Tools ohne weitere Konfiguration im CI-Server in Containern ausgeführt werden. In Containern können Anwendungen außerdem zum Test isoliert ausgeführt werden, um beispielsweise Port-Konflikte zu vermeiden
2. Container-Images sind ein standardisiertes Artefakt, in das die Anwendung durch die Pipeline verpackt wird

Diese Images können auf vielen verschiedenen Betriebsumgebungen deployt werden, da mittlerweile sowohl Docker-Container und Images als auch das API der Registry durch die Open Container Initiative (OCI) standardisiert sind. In den letzten Jahren haben sich besonders im DevOps-Umfeld Container Orchestration Platforms als flexibles Mittel für Deployments von OCI-Images erwiesen. Dabei hat sich K8s als De-facto-Standard herauskristallisiert, weshalb sich dieser Artikel exemplarisch auf das Deployment auf K8s beschränkt.

Klassische CD-Pipelines – CIOps

Bei einer „klassisch" umgesetzten CD-Pipeline führt der CI-Server aktiv das Deployment in die Betriebsumgebung durch *(siehe Abbildung 1)*. Zur Abgrenzung von später entstandenen Methoden wie GitOps (siehe unten) wird dieses Vorgehen auch als „CIOps" bezeichnet. Manchmal wird es dabei als Antipattern dargestellt *[1]*. Das Verfahren hat sich allerdings jahrelang in der Praxis bewährt und es spricht generell nichts dagegen, es weiterhin zu verwenden.

Eine einfach umsetzbare Logik zur Automatisierung des Deployments in einer CIOps-Pipeline mit Staging- und Produktionsumgebungen ist die Verwendung von Branches in Git.

Dafür nutzen viele Teams Feature Branches oder Git Flow, in denen der integrierte Entwicklungsstand auf dem Develop-Branch zusammenfließt und der Main- (oder Master-) Branch die produktiven Versionen enthält. Darauf kann einfach eine CD-Strategie aufgebaut werden: Jeder Push auf Develop führt zu einem Deployment auf die Staging-Umgebung, jeder Push auf Main geht in Produktion. So steht stets die letzte integrierte Version auf Staging für funktionale oder manuelle Tests bereit. Durch einen Pull Request (PR) oder Merge auf Main wird dann das Deployment in Produktion angestoßen. Zudem ist ein Deployment pro Feature Branch denkbar.

Der Nachteil dieses Vorgehens ist, dass für jedes Deployment ein Build im CI-Server durchlaufen werden muss. Das macht den Prozess langsamer. Denn generell soll ja dasselbe Artefakt auf allen Stages deployt werden. Es wäre also gar kein neuer Build, Test oder gar Versionsname nötig.

Eine solche Deployment-Logik lässt sich mit Jenkins Pipelines einfach realisieren, da der Branch-Name in Multibranch-Builds aus dem Environment abgefragt werden kann. Ein ausführliches Beispiel, das eine vollständige Implementierung mit Jenkins zeigt, ist in Java aktuell 04/2018 beschrieben *[2]*, das vollständige Jenkinsfile kann bei GitHub eingesehen werden *[3]* (Branch „11").

GitOps vs. CIOps

Mittlerweile gibt es im K8s-Umfeld eine Alternative zu CIOps: GitOps. Hier prüft eine im K8s-Cluster laufende, Cloud-native Anwendung (der „GitOps-Operator") kontinuierlich den tatsächlichen Zustand des Clusters gegen den Wunschzustand, der in einem Git-Repository beschrieben ist. Deployments werden durch einen Push auf dieses Repository, beispielsweise durch die Annahme eines PR, ausgelöst *(siehe Abbildung 2)*. Durch GitOps ergeben sich einige Vorteile:

- Weniger schreibender Zugriff von außen auf den Cluster nötig, da der GitOps-Operator Deployments von innerhalb des Clusters durchführt
- Keine Credentials im CI Server, da kein Zugriff auf den Cluster benötigt wird
- Infrastructure as Code (IaC) bietet Vorteile für Auditierung und Reproduzierbarkeit. Außerdem sind Cluster und Git automatisch synchronisiert
- Der Zugriff auf Git ist oft organisatorisch einfacher als der auf den API-Server. Möglicherweise entfällt die Notwendigkeit einer Firewall-Freischaltung

Rolle des CI-Servers bei GitOps

Für das Deployment von Third-Party-Anwendungen (die nicht selbst entwickelt werden), ist ein CI-Server nicht mehr unbedingt nötig. Bei selbst geschriebenen Anwendungen sind nach wie vor Build, Tests, etc. auszuführen. Dies übernimmt weiterhin der CI-Server, genauso wie das Pushen des Images in eine Registry *(siehe Abbildung 3)*. Außerdem kann der CI-Server eingesetzt werden, um einige der Herausforderungen von GitOps zu lösen:

- Lokale Entwicklung mit GitOps ist weniger effizient (Betrieb des Operators, Deployment und Debugging sind umständlicher).
- Manuelle Implementierung von Staging kann umständlich sein (für jede Stage muss ein PR erstellt werden).
- Oft wird bei GitOps der Infrastruktur-Code in einem zentralen Repository gesammelt. Dies bietet den Vorteil, dass der gesamte Zustand des Clusters an einer Stelle gespeichert ist. Der Nachteil: Die Trennung von Anwendungs- und Infrastruktur-Code auf zwei Repositories ist aufwendiger zu warten, beispielsweise bei Review, Versionierung und lokaler Entwicklung.

Durch Unterstützung des CI-Servers kann erreicht werden, dass beides im Repository der Anwendung (im Folgenden als App-Repository bezeichnet) verbleibt. Der CI-Server pusht dann den Infrastruktur-Code in das GitOps-Repository *(siehe Abbildung 4)*.

GitOps am Beispiel

Die Implementierung eines GitOps-Flows, wie in *Abbildung 4* gezeigt, klingt zunächst sehr einfach zu implementieren. Doch wie so oft liegt auch hier der Teufel im Detail: Zum einen warten Herausforderungen bei der Implementierung, zum anderen fallen schnell weitere Punkte auf, die durch die Pipeline automatisiert werden können. So kann am Ende die zunächst einfach erscheinende Implementierung einer solchen Pipeline doch aufwendig werden.

Die größte Herausforderung ist, dass mehrere gleichzeitige Builds laufen können, die in dasselbe GitOps-Repository schreiben. Eine zuverlässige Fehlerbehandlung solcher Concurrency-Issues führt zu

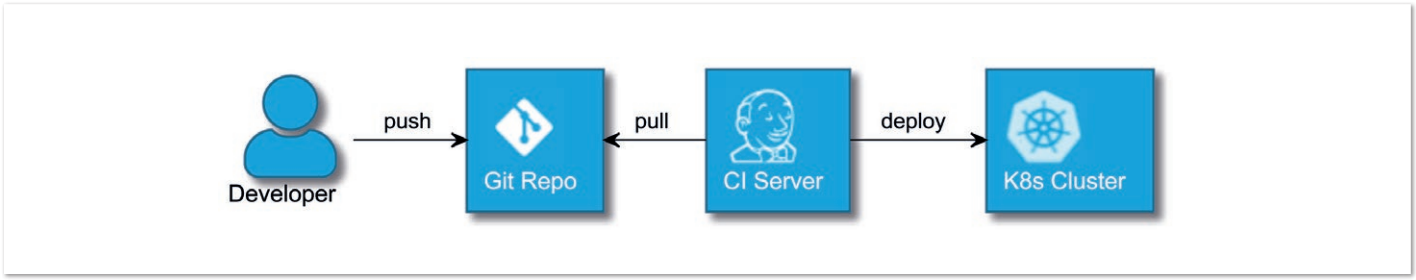


Abbildung 1: „Klassische" CD-Pipeline – CIOps (© Cloudogu)

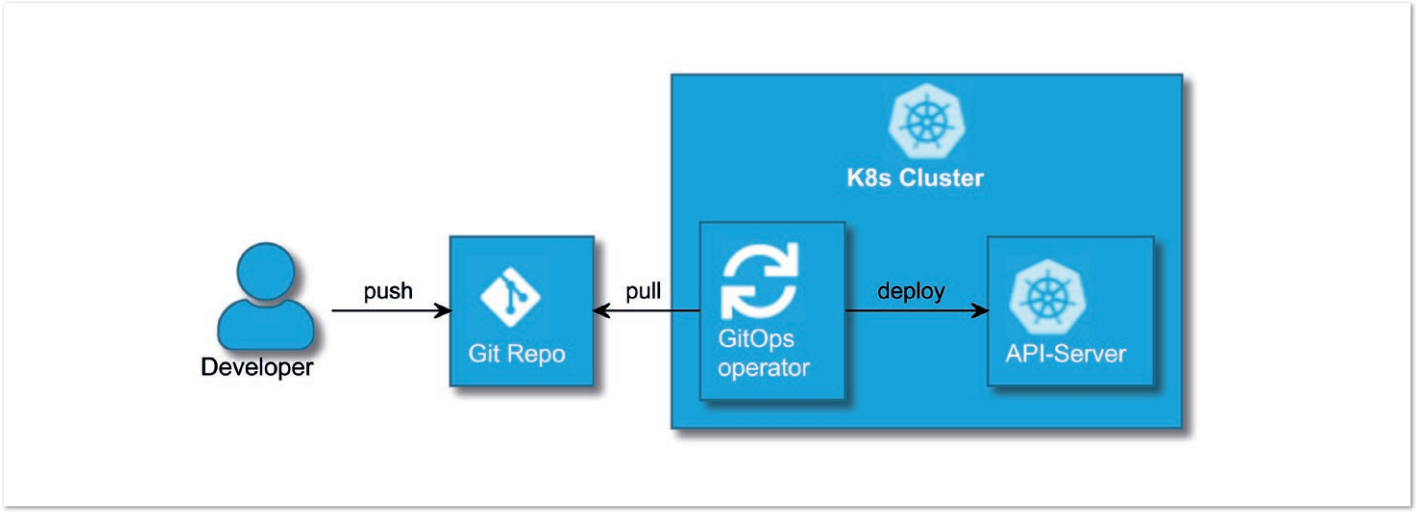


Abbildung 2: Einfaches Deployment mittels GitOps (© Cloudogu)

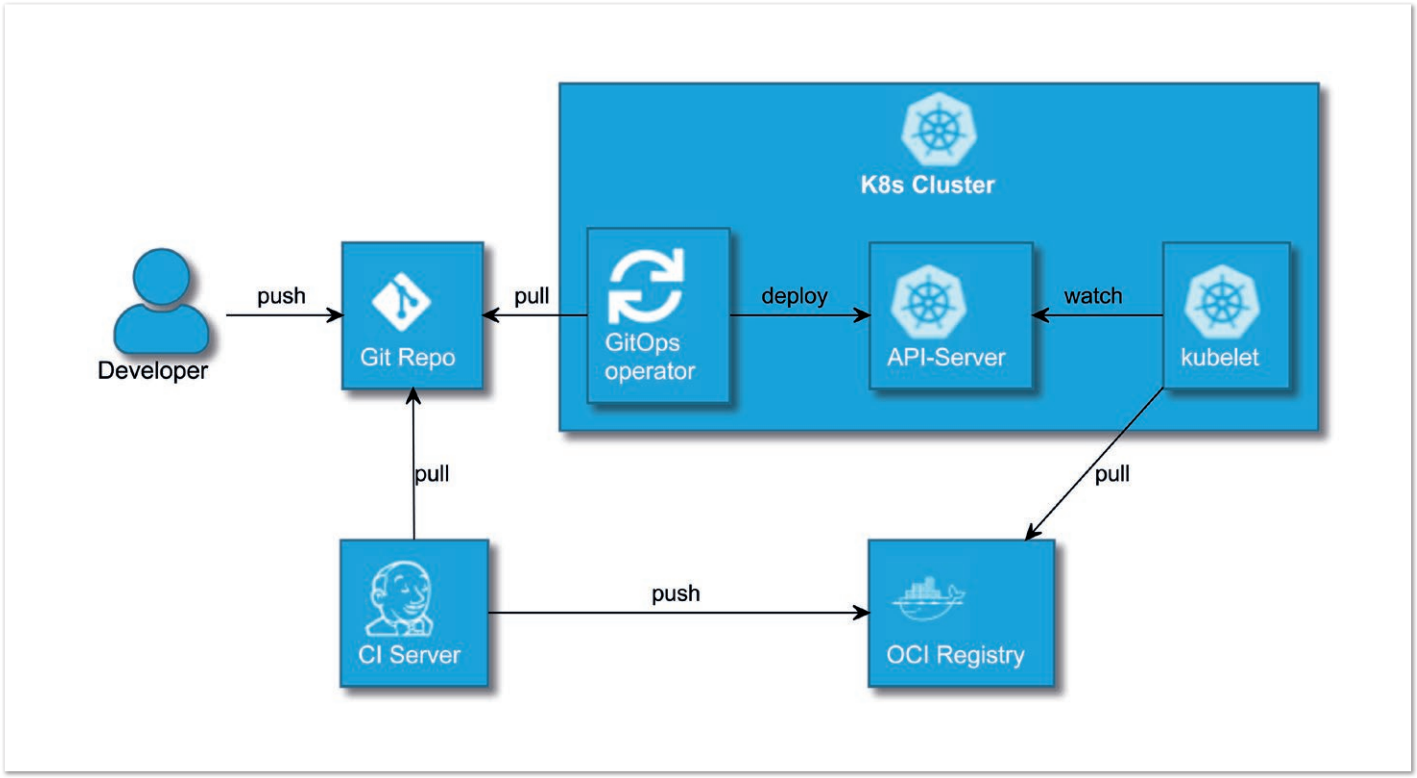


Abbildung 3: GitOps-Deployment von selbst entwickelten Images (© Cloudogu)

überraschender Komplexität. Wenn die Pipeline dann einmal grundlegend funktioniert, kann der Entwicklungsprozess durch weitere Automatisierung effizienter gestaltet werden. Beispiele für solche Erweiterung folgen später.

Konkrete Beispiele für GitOps-Flows bietet der GitOps-Playground *[4]*, mit dem in einem lokal ausführbaren Cluster verschiedene GitOps-Operator, wie Flux (GitOps Toolkit) und ArgoCD (GitOps Engine) im Zusammenspiel mit Jenkins ausprobiert werden können.

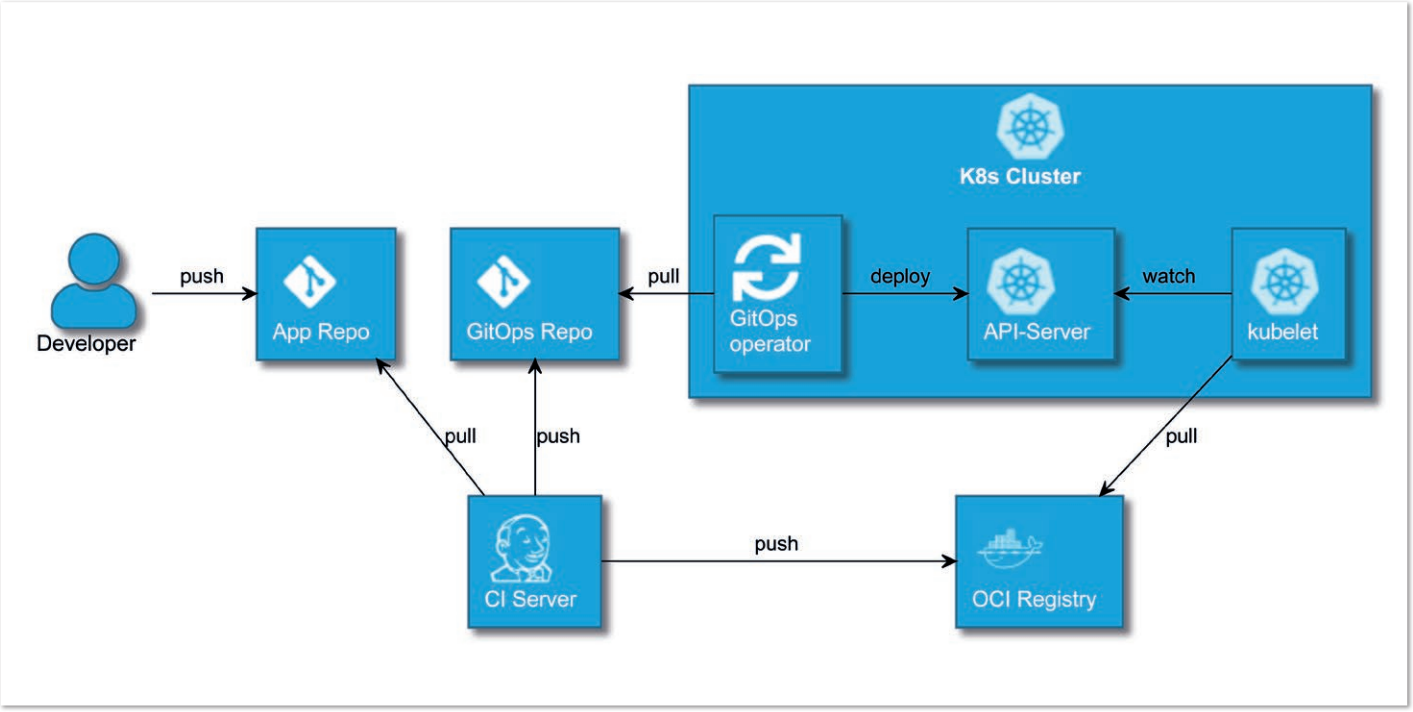


Abbildung 4: Deployment mit App-Repository und GitOps-Repository (© Cloudogu)

Darin enthalten ist auch eine Jenkins Pipeline (Datei „applications/petclinic/fluxv1/plain-k8s/Jenkinsfile"). Diese ist ähnlich aufgebaut wie das erwähnte CIOps-Beispiel. In zwei Punkten unterscheiden sich die Pipelines jedoch grundlegend:

- Die YAMLs werden in das GitOps-Repository gepusht, statt auf den Cluster angewandt
- Die verschiedenen Stages werden komplett im GitOps-Repository realisiert (nicht im App-Repository). Dadurch ist hier kein CI-Server mehr notwendig. Der GoLive ist schneller

Die grundlegende Ordnerstruktur des GitOps-Repository zeigt *Abbildung 5*: Auf oberster Ebene gibt es einen Ordner pro Stage. Darin enthalten ist je ein Ordner pro Applikation. Das Deployment unterscheidet sich dann leicht, je nachdem wie Stages gelöst sind:

- Staging Namespaces im selben Cluster (siehe GitOps-Playground): Es gibt nur einen GitOps-Operator und dieser deployt alle im GitOps befindlichen K8s-Ressourcen in den Cluster. Dabei muss jeweils auf die Angabe des richtigen Namespaces in den K8s-Ressourcen geachtet werden. Auf diese Weise ist es im GitOps-Playground gelöst
- Alternativ können auch Staging Cluster verwendet werden. In diese wird der GitOps-Operator so konfiguriert, dass er entsprechend seiner Stage alles deployt, was im jeweiligen Ordner liegt. Beispiel: Der GitOps-Operator im Staging-Cluster deployt nur K8s-Ressourcen aus dem „Staging"-Ordner

Der Ablauf der erwähnten Pipeline aus dem GitOps-Playground ist wie folgt:

1. Push auf den Main Branch des App-Repository löst den GitOps-Prozess aus
2. GitOps-Repository klonen
3. Staging: Image-Version in Deployment-YAML aktualisieren, in den Applikationsordner der Stage kopieren und auf den Main Branch des GitOps-Repositories pushen.
4. Produktion: Wie in Schritt 3, nur dass die Änderungen im Ordner "Production" gemacht werden und auf einen speziell für die Anwendung erstellten Branch im GitOps-Repo gepusht werden. Anschließend wird ein PR auf den Main Branch geöffnet.

Nachdem diese Pipeline durchlaufen ist, wird die Anwendung vom GitOps-Operator zum Review auf Staging deployt. Darüber hinaus existiert ein PR, der bei Annahme direkt (ohne CI-Server) zu einem Deployment in Produktion führt.

Die oben beschrieben Concurrency-Issues können zwischen dem Klonen des Repository und den Pushes auftreten: Wenn zwischenzeitlich das remote Repository geändert wurde, scheitert der Push und damit der Build. Das macht die Entwicklung komplizierter und langsamer. Die Pipeline im Beispiel hat daher einen einfachen Retry-Mechanismus. Scheitert der Push, erfolgt ein Pull und ein erneuter Push. Diese Lösung ist nicht perfekt, da bei Konflikten der Build trotzdem scheitert. Unter gewissen Umständen kann sogar eine Inkonsistenz entstehen: Beim Pull könnte ein Fast Forward Merge gemacht werden, der die Änderungen aus dem Build mit denen aus einem anderen „vermischt". Hier wäre es also sicherer, nach dem Pull nicht einfach zu pushen, sondern einen Reset auf den Remote-Stand zu machen und die Änderungen erneut durchzuführen.

An anderer Stelle zeigt sich die Pipeline schon recht ausgefeilt. So werden die Commits, die der Job am GitOps-Repository vorgenommen hat, für mehr Transparenz in der Jenkins Job Description angezeigt. Für ein effizienteres Review des PR wird Folgendes in die Commit Message im GitOps-Repository geschrieben (*Abbildung 6* zeigt dies am Beispiel mit SCM-Manager):

- Autor des ursprünglichen Commit im App-Repository
- Autor wird beibehalten, aber „Jenkins" wird Committer. Damit

ist klar, von wem diese Änderung stammt, aber auch, dass der Commit automatisiert erstellt wurde
- Link auf die Issue-ID, geparst aus der ursprünglichen Commit Message. Damit ist eine direkte Verbindung zur Fachlichkeit im Issue Tracker gegeben
- Link auf den ursprünglichen Commit im App-Repository. Damit lässt sich mit einem Klick auf den Source Code der Anwendung wechseln
- Ein Staging-Commit wird jeweils markiert (nicht in der Abbildung zu sehen)

Derzeit sind im GitOps-Playground noch weitere Features in Arbeit, die den Prozess effizienter gestalten. Möglicherweise sind diese zum Zeitpunkt der Veröffentlichung des Artikels schon verfügbar:

- Fail Early: statische YAML-Analyse durch den CI-Server. Damit möglichst selten der aufwendige Weg der Fehlersuche im Log des GitOps-Operators gegangen werden muss, werden die YAML-Files auf syntaktische Korrektheit geprüft, beispielsweise mit dem Tool „yamlint". Ein weiterer Schritt ist das Prüfen der K8s-Ressourcen gegen das K8s-Schema. Dies kann mit dem Tool „kubeval" erfolgen. Bei Helm-Charts mit eigenem Schema könnte auch gegen dieses geprüft werden (mittels „helm lint")
- Automatisch erstelle PRs können durch weitere Informationen angereichert werden. Beispielsweise kann ein bereits bestehender PR ergänzt werden, wenn weitere Commits darauf gemacht werden. Zudem kann ein Link auf den erstellenden Jenkins Job in den Kommentaren erzeugt werden
- Ein Weg, um Konfigurationsdateien oder Scripts in den Cluster zu bringen, ist es, diese als inline YAML zu verpacken, beispielsweise in eine Config Map. Dieses Verfahren hat den Nachteil, dass es in dieser Form kein Syntax-Highlighting oder Linting gibt. Dadurch kommt es häufiger zu vermeidbaren Fehlern oder ineffizientem „hin- und herkopieren". Dieser Nachteil kann mittels Automatisierung behoben werden: Der CI-Server übernimmt das Verpacken einer „echten" Datei in YAML. Dadurch besteht bei der Entwicklung die Möglichkeit, auf dieser Datei zu arbeiten und dort das gewohnte Highlighting und Linting zu bekommen
- Häufig besteht der Bedarf vor Abschluss der Entwicklung eines Features, dieses in der Staging-Umgebung manuell zu testen. Mithilfe der Pipeline lässt sich dies ohne (verfrühten) Merge auf den Main-Branch und ohne PR für Produktion realisieren. Dies kann durch Build-Parameter in Jenkins implementiert werden. Ein solcher Parameter kann beim manuellen Anstoßen eines Builds gesetzt werden. Die Pipeline kann auf den Parameter reagieren, indem ins Staging gepusht, aber kein PR für Produktion erstellt wird
- Eine größere Anzahl Stages kann durch weitere Branches und PRs realisiert werden

```
├── production
│   └── application
│       ├── deployment.yaml
│       └── ...
└── staging
    └── application
        ├── deployment.yaml
        └── ...
```

Abbildung 5: Mögliche Ordnerstruktur eines GitOps-Repositories (© Cloudogu)

Templating-Tools

Die vorgestellten Beispiele zu CIOps und GitOps zeigen, wie einfache K8s-Ressourcen auf den Cluster mit der jeweiligen Methode angewendet werden können. Nachteil dabei ist, dass die K8s-Ressourcen komplett redundant für jede Stage gespeichert werden müssen. In der Praxis werden daher oft Templating-Tools eingesetzt, die eine Parametrisierung einer einzigen Quelle (ohne Redundanz) ermöglichen. Helm, der offizielle Package Manager für K8s, ist eine gängige Lösung. Mit Helm können nicht nur Third-Party-Packages deployt werden. Seine Templating-Funktion kann auch für die lokale Entwicklung genutzt werden.

Für die lokale Entwicklung gibt es einige Alternativen zu Helm, wie das „Template-freie" Tool Kustomize, das mit sogenannten Overlays arbeitet, die mit dem Patch-Mechanismus auf eine Basisdatei angewendet werden.

Bei CIOps lassen sich Templating-Tools relativ einfach anwenden. Die Tools stehen als Kommandozeilenwerkzeug zur Verfügung, das in der Pipeline aufrufbar ist. Ein Beispiel dafür steht bei GitHub zur Verfügung *[3]* (Branch „12"). Hier wird das Helm-Binary als Container ausgeführt, sodass keine weitere Konfiguration seitens des Jenkins-Masters erforderlich ist. Einige weitere wertvolle Erkenntnisse aus der Praxis:

- Durch die Nutzung von `helm upgrade --install` muss nicht aufwendig zwischen Erstinstallation und Upgrade unterschieden werden
- Die in allen Helm-Paketen (sogenannten Charts) vorgeschriebene „values.yaml" beschreibt Standardwerte, eine weitere `values-`Datei pro Stage setzt die jeweils spezifischen Werte. Diese Datei muss dem Helm-Befehl per `-value`-Parameter übergeben werden, die Standard „values.yaml" wird implizit immer angezogen

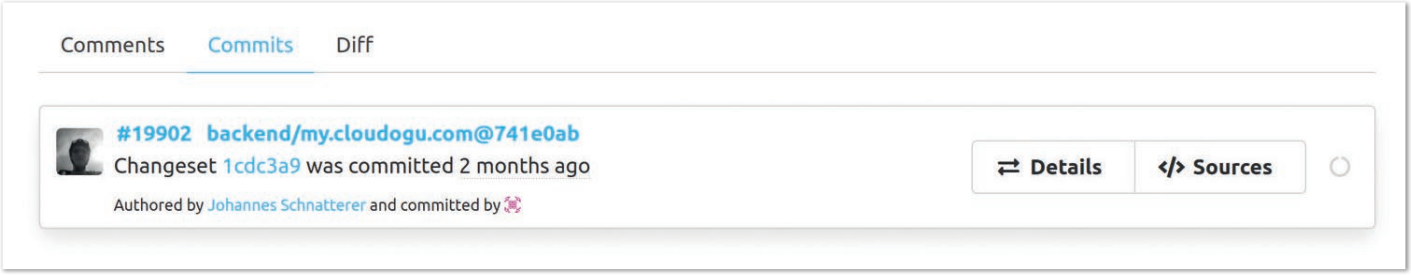


Abbildung 6: Beispiel eines vom CI-Server erstellten Commit im GitOps-Repository (© Cloudogu)

- Name und Version des Image können bequem mittels Parameter gesetzt werden, beispielsweise `--set 'image.tag=...`

Bei der Verwendung von Templating-Tools mit GitOps wartet zu Beginn eine große Herausforderung: Wie kann der imperative Aufruf (beispielsweise `helm upgrade`) in eine deklarative Form gebracht werden, die im GitOps-Repository abgelegt werden kann? Die Lösung: weitere Operatoren in K8s. Für die weit verbreiteten Tools Helm und Kustomize existieren solche Operatoren bereits, für anderen Templating-Tools nicht unbedingt. Auch hier gibt es ein Praxisbeispiel im GitOps-Playground *[4]* (Datei: applications/nginx/fluxv1/Jenkinsfile). Darin wird eine statische HTML-Seite mit dem Webserver NGINX ausgeliefert. Dieses Beispiel käme auch ohne Pipeline aus, allerdings mit den oben erwähnten Nachteilen:

- die HTML-Datei müsste inline in einer YAML-Datei gepflegt werden,
- für die lokale Entwicklung wäre ein Helm-Operator notwendig und
- die values.yamls müssten in pro Stage komplett redundanten HelmRelease YAMLs beschrieben werden.

Insofern ist an dieser Stelle die Verwendung einer Pipeline auch beim GitOps-Deployment von Third-Party-Anwendungen vorteilhaft. Beim Blick auf die beiden Jenkinsfiles aus dem GitOps-Playground fällt auf, dass die Pipelines für die beiden unterschiedlichen Anwendungsfälle „K8s-Ressourcen" und Helm zu großen Teilen gleich sind. Hier bietet sich das Auslagern in eine Jenkins Shared Library an. Diese ist in Arbeit und wird ihren Weg in den GitOps-Playground finden.

Abschließend sei angemerkt, dass die Nutzung eines Helm-Operators auch ohne GitOps Vorteile haben kann: Die Quelle und Version des Charts sind als IaC (in YAML) deklariert, statt innerhalb eines Jenkinsfiles. Dieses wird einfach auf den Cluster angewendet. In der Pipeline wird kein Helm Binary mehr benötigt. Das gleiche Vorgehen funktioniert auch in der lokalen Entwicklung.

Fazit

Die Mehrwerte von CD stehen außer Frage. Dieser Artikel zeigt anhand beispielhafter CD-Implementierungen mit K8s- und Helm-Deployment, dass die Realisierung sowohl mit CIOps als auch GitOps gut mit Jenkins möglich ist. Die Frage „CIOps oder GitOps" ist also ein Implementierungsdetail. Beides kann hervorragend in der Praxis funktionieren. Wer bereits bestehende CD-Prozesse hat, sollte nur umstellen, wenn die Vorteile von GitOps im jeweiligen Anwendungsfall große Mehrwerte bringen. Nicht zu unterschätzen ist dabei der Aufwand für die Migration: Wer viele Pipelines hat, muss auch viele Pipelines migrieren. Für Neueinsteiger bietet es sich aufgrund der vielen Vorteile an, direkt mit GitOps zu starten. Allerdings wird die bereits steile Lernkurve dadurch noch steiler. Die vollständigen Beispiele können bei GitHub in den Repositories *[3]* (CIOps) und *[4]* (GitOps) gefunden werden.

Was dieser Artikel nicht betrachtet, sind die Unterschiede verschiedener GitOps-Operatoren. Dies ist ein Thema für sich. Ein erster Schritt, sich diesem praktisch zu nähern, kann der GitOps-Playground *[4]* sein.

Quellen

[1] *https://www.weave.works/blog/kubernetes-anti-patterns-let-s-do-gitops-not-ciops*

[2] *https://cloudogu.com/de/blog/continuous_delivery_4_de*

[3] *https://github.com/cloudogu/jenkinsfiles*

[4] *https://github.com/cloudogu/k8s-gitops-playground/tree/405d*

Johannes Schnatterer

Cloudogu GmbH

johannes.schnatterer@cloudogu.com

Seit mehr als zehn Jahren verfolgt Johannes mit seiner Tätigkeit in den Bereichen Dev, Ops und Architektur das Ziel, wartbare, sichere Anwendungen schnellstmöglich in Produktion zu bringen. Dabei greift er zu Methoden wie Continuous Delivery & GitOps, Clean Code, IaC, O11y, Dokumentation und Pragmatismus. Er ist begeistert von Open Source, allen Cloud-native-Themen und der lebhaften Community. Als Autor, Trainer und Consultant lehrt und lernt er gerne.

Daniel Huchthausen

Cloudogu GmbH

daniel.huchthausen@cloudogu.com

Daniel Huchthausen ist Consultant bei der Cloudogu GmbH aus Braunschweig. Er steht sowohl Kunden als auch Kollegen als Scrum Master und als Experte für Anforderungs- und Testmanagement sowie Prozessanalyse und -optimierung zur Verfügung.

Ex Machina: Automation Divided

Dominik Röschke und Nikolas May, MATHEMA Software GmbH

In diesem Artikel werfen wir einen Blick auf einige Continuous-Integration-Server (CI) und beleuchten ihre Vor- und Nachteile genauer. Am Schluss werden Empfehlungen für mögliche Einsatzgebiete gegeben.

Ex Machina: aus der Maschine. Seinen Ursprung hat der Begriff im altgriechischen Theater, vor allem in Tragödien. Tragische Konflikte konnten nicht von den beteiligten Menschen gelöst werden, dies musste durch Einmischung der Götter von außen geschehen. Meist durch den Einsatz einer Theatermaschine: Deus ex Machina – Gott aus einer (Theater-)Maschine.

Auch als Softwareentwickler haben wir häufiger Probleme, die wir nicht – oder nur schwer – allein lösen können. Continuous Integration (CI) ist eines dieser Probleme. Würde man die CI-Arbeit per Hand erledigen, so würde dies für jeden Commit bedeuten, dass dieser manuell gebaut, getestet und gegen den Haupt-Branch integriert werden müsste. Abgesehen davon, dass dies ein stark repetitives und fehleranfälliges Vorgehen wäre, würde es den eigentlichen Handlungsstrang, die Softwareentwicklung, nur sehr langsam vorantreiben.

Warum sich also nicht den Trick seiner Vorfahren zunutze machen und die Ex Machina zu Hilfe nehmen: einen CI-Server. Dieser übernimmt dann das Bauen, Testen und Integrieren der Software automatisch. Nachdem sich Continuous Integration inzwischen sehr weit verbreitet hat, gibt es auch eine Vielzahl verschiedener CI-Server, jeder mit seinen eigenen Vor- und Nachteilen.

Unter anderem zu nennen sind GitLab CI, Drone CI und CircleCI, auf die wir uns in diesem Vergleich konzentrieren werden. Natürlich gibt es noch eine große Auswahl an anderen Produkten (Jenkins, Teamcity, Concourse CI, Travis-CI, Bamboo usw.), die man noch beleuchten könnte. Aus Platzgründen werden wir bei den genannten drei CI-Servern bleiben.

GitLab CI

GitLab CI hat 2014 seinen Anfang als Git Repository gemacht und wurde kontinuierlich erweitert, unter anderem auch durch eine integrierte CI-Lösung – GitLab CI/CD. Diese ist inzwischen in der Version 13 verfügbar und bringt einige spannende Features mit:

- **Auto DevOps:** Bei Aktivierung verspricht GitLab, automatisch zu erkennen, welche Art von Software im Repository liegt, und dafür die CI/CD-Features zu aktivieren. Dies umfasst das Bauen, Testen, Deployen und Monitoren der Software.
- **Review-Apps:** Im Zusammenspiel mit einem Kubernetes-Cluster kann GitLab automatisch für jeden offenen Branch eine eigene Review-Umgebung aufsetzen. Dies ermöglicht, dass bereits vor einem Merge auf einfachste Art und Weise eine Version von Nicht-Entwickelnden ausprobiert werden kann.
- **Integrierte Docker-Registry**

Auto DevOps und Review-Apps funktionieren im Optimalfall Plug-and-Play und erzeugen keinen zusätzlichen Konfigurationsaufwand. In der Praxis hat sich allerdings gezeigt, dass gerade bei komplexeren Softwareprojekten das Tooling doch überfordert ist und man die Pipeline per Hand konfigurieren muss.

Möchte man GitLab manuell konfigurieren, beginnt alles mit der .gitlab-ci.yml *(siehe Listing 1)*, die versioniert im Repository mitabgelegt wird. Hier wird in verständlicher Pipeline-Syntax beschrieben, wie das Projekt zu bauen ist. Wie man darin erkennen kann, lassen sich Dependencies auf sehr einfache Art und Weise cachen, um Build-Times zu reduzieren.

Die Vorteile von GitLab CI/CD sind unter anderem die Integration in das eigene Repository und die vollautomatische Konfiguration einer CI-Pipeline. Auch, dass man auf *gitlab.com* eigene CI-Runner anbinden kann, ist unglaublich praktisch. So kann sogar der eigene Entwicklungsrechner zum CI-Runner werden, falls die 200 Freiminuten aufgebraucht sind. Alles was dafür benötigt wird, sind Docker und das GitLab-Runner-Binary. Review-Apps sind, gerade in Zeiten von vermehrter Remote-Arbeit, auch ein nicht zu unterschätzendes Mittel, um Kommunikation und Zusammenarbeit zu vereinfachen.

Ein Nachteil von GitLab CI ist die vergleichsweise kleine Community: 4.300 Fragen auf Stack Overflow verglichen mit 43.000 Fragen zu Jenkins. Das wird aber durch eine sehr umfangreiche Dokumentation sehr gut ausgeglichen.

GitLab gibt es gehostet in vier verschiedenen Preiskategorien. Alternativ kann es mit ein bisschen Konfigurationsaufwand selbst gehostet werden, auch hier gibt es eine Free und eine Premium Edition, die vor allem zusätzliche Organisationsfeatures bietet.

CircleCI

Bei CircleCI handelt es sich um eine 2011 gegründete, cloudbasierte Continuous-Integration- und Delivery-Plattform. Es unterstützt den CI/CD-Prozess durch Build-Pipelines. Als Laufzeitumgebungen für die Pipelines werden Docker, Linux, Windows und MacOS angeboten. Von Haus aus wird eine Anbindung an GitHub

```
image: docker:latest
services:
  - docker:dind

# Definition des Dependency-Caches
cache:
  paths:
    - .m2/repository

variables:
  MAVEN_OPTS: "-Dmaven.repo.local=$CI_PROJECT_DIR/.m2/repository"

stages:
  - build

maven-build:
  image: maven:3.6-amazoncorretto-11
  stage: build
  script: "mvn package -B"
  artifacts:
    paths:
      - target/*.jar
```

Listing 1: Ein simples Beispiel einer GitLab-CI-Pipeline, die mit Maven ein Projekt baut und die Dependencies des Builds in einem Cache aufbewahrt

```
version: 2
jobs:
 build: #Definiton des Jobs mit Namen build
   docker:
     - image: circleci/openjdk:8-jdk # Dockerimage in dem gearbeitet wird
   working_directory: ~/repo

   steps: #
     - checkout # Repository auschecken
     - restore_cache:
         keys:
         - maven-dependencies-{{ checksum "pom.xml" }} # Cache wiederherstellen
         - maven-dependencies- #Fallback, falls sich die Checksumme geändert hat
     - run: mvn dependency:go-offline #Maven dependencies herunterladen
     - save_cache: #Lokales Maven Repository cachen
         paths:
           - ~/.m2
         key: maven-dependencies-{{ checksum "pom.xml" }}
     - run: mvn package #Test ausführen
     - store_test_results:  #läd die Testergebnisse hoch
          path: petclinic/target/surefire-reports
     - persist_to_workspace:  #Persistiert die Dateien in einem zwischen den Jobs geteilten Workspace
         root: .
         paths:
           - petclinic/target/petclinic.war
```

Listing 2: Ein simples Beispiel einer CircleCI-Pipeline, die mit Maven ein Projekt baut und die Dependencies des Builds in einem Cache aufbewahrt

und Bitbucket ermöglicht. Zudem gibt es für eine Vielzahl von Programmiersprachen und Build-Tools Beispiel-Build-Pipelines.

CircleCI hält seine Konfiguration als YAML-Datei, die sich unter .circleci/config.yml befindet *(siehe Listing 2)*. Somit wird die Pipeline-Definition im Repository versioniert und kann bei jedem Commit ausgewertet und berücksichtigt werden. Zudem werden für jeden Build ein neuer Docker-Container beziehungsweise eine virtuelle Maschine erzeugt. Das hat zur Folge, dass man immer auf einer frischen Instanz arbeitet, die sich in einem wohldefinierten Zustand befindet.

Um die Build-Zeiten dennoch auf eine akzeptable Geschwindigkeit zu bringen, bietet CircleCI integriertes Caching *(siehe auch Listing 2)*. Mittels `save_cache` können zum Beispiel Dependencies gecacht werden, die bei einem erneuten Build mittels `restore_cache` wiederhergestellt werden. Somit müssen nur noch bei Änderungen die Abhängigkeiten aktualisiert werden. In der unter *Listing 2* gezeigten Pipeline verringert das Caching die Build-Zeiten des Maven-Projekts von 75 auf 13 Sekunden.

CircleCI bietet die Möglichkeit, Pipeline-Elemente über sogenannte Orbs wiederzuverwenden. Damit können öfter auftretende Build-Schritte vereinheitlicht und über Parametrisierung gesteuert werden. Die so erstellten Orbs werden im sogenannten Orb Registry gespeichert und unterliegen dem Open-Source-Gedanken. Dies hat den Vorteil, dass man sich vieler bereits existierender Orbs bedienen kann *[1]* und nicht das Rad neu erfinden muss.

Sollte es bei CircleCI trotz der umfangreichen Log- und Analysemöglichkeiten zu einem Fehler kommen, der nicht nachgestellt werden kann, besteht die Möglichkeit, eine SSH-Verbindung zum Build-Job aufzubauen, um diesen zu debuggen.

Alles in allem ist CircleCI ein mächtiger CI-Service, der eine Vielzahl an vordefinierten Funktionen und Orbs mitbringt. Des Weiteren

```
kind: pipeline
name: default

steps:
- name: test
  image: maven:3-jdk-10
  commands:
  - mvn package -B
```

Listing 3: Eine simple Drone-CI-Pipeline, die ein Maven-Projekt baut

besteht die Möglichkeit, CircleCI On-Premises zu hosten, wenn man beispielsweise aus regulatorischen Gründen darauf angewiesen ist. Als Negativpunkte sind zum einen die Beschränkung der Versionsverwaltungen auf Bitbucket und GitHub anzumerken. Zum anderen dürfte das Preismodell abschrecken. Hierfür werden im Cloud-Modell 15,00 Euro pro User zuzüglich Rechenzeit berechnet. Im „On-Premises"-Betrieb werden 35,00 Euro pro User fällig. Eine gute Nachricht für alle Open-Source-Projekte: CircleCI schenkt euch Rechenzeit im Wert von 240,00 Euro pro Monat!

Drone CI

Drone CI ist ein relativ neuer CI-Server (die Version 1.0 wurde am 7. November 2018 veröffentlicht) und ist auf eine Container-native CI-Pipeline ausgelegt. Drone selbst lässt sich am einfachsten direkt über Docker oder Kubernetes hosten. Für ein schmerzfreies Setup werden vorgefertigte Helm-Charts zur Verfügung gestellt. Drone wurde Anfang August 2020 von Harness gekauft und bietet inzwischen auch eine CI-as-a-Service-Lösung unter *drone.io* an. Diese ist besonders für Test- und Open-Source-Projekte geeignet, da sie 5.000 Builds im Jahr für jedes Projekt anbietet und für Open-Source-Projekte kostenlos ist.

Drone verpackt jeden Pipeline-Step in einen eigenen Container und führt diesen auf konfigurierten Docker-Runnern aus. Eine beispielhafte Drone-CI-Pipeline, um ein simples Java-Projekt zu bauen, sieht man in *Listing 3*.

	GitLab CI	CircleCI	Drone CI
gehostet	200 Freiminuten, danach kostenpflichtig	2.500 Credits pro Woche, danach kostenpflichtig	5.000 Gratis-Builds im Jahr, danach kostenpflichtig
selbst gehostet	Gratis mit kostenpflichtigen Zusatzfunktionen	Kostenpflichtig, 35 Euro pro User im Monat	Gratis
unterstützte VCS	GitLab, mit „GitLab Premium" jeder Git-Server	Bitbucket, GitHub	GitHub, GitLab, Gogs, Gitea, Bitbucket
Unterstützte Build-Umgebungen	Linux, Mac, Windows	Docker, Linux VM, macOS VM, Windows VM	Linux, Windows

Tabelle 1

Dadurch, dass jeder Pipeline-Step in einem eigenen Docker-Container auf Docker-Runnern ausgeführt wird, ist eine Skalierung des CI-Service unglaublich einfach. Falls sich Builds aufstauen, kann ohne viel Aufwand ein neuer Runner hinzugefügt werden. Auch Drone CI bietet die Möglichkeit, Build-Dependencies zu cachen, allerdings nur über ein Plug-in. Drone-Plug-ins sind weitere Docker-Container, die eine Pipeline um zusätzliche Features erweitern. Beispielsweise durch einen Dependency-Cache.

Wie bereits eingangs erwähnt, ist Drone unglaublich einfach zu verwenden, wenn Docker-Infrastruktur zur Verfügung steht. Falls nicht, sollte man Drone allerdings auch nicht verwenden, da es voll und ganz dafür ausgelegt ist, Software in Containern zu bauen und auszuliefern.

Fazit

Leider gibt es keine Ex Machina, die jegliche Probleme löst, ohne fehl am Platz zu wirken. So überlege man sich, wie es wirken würde, wenn die Göttin Athene die Probleme von Friedrich Dürrenmatts „die Physiker" lösen würde. Ähnlich verhält es sich auch bei unseren CI-Servern. Jeder der genannten Server hat eine Daseinsberechtigung, da er sich in seinen Bereichen besonders auszeichnet *(siehe Tabelle 1)*.

CircleCI bietet eine leichtgewichtige, skalierbare Lösung, die dank des Caching und der performanten Maschinen schnelle Build- und Deployment-Zeiten gewährt. Durch die Integration mit Bitbucket und GitHub bekommt man fast eine Out-of-the-Box-Erfahrung. Der einzige Wermutstropfen mag hier der Preis sein, den man allerdings gegen die Betriebskosten für die entsprechende Infrastruktur rechnen sollte.

Falls bereits Container-Infrastruktur vorhanden ist, ist Drone CI ein schnell aufgesetzter, leichtgewichtiger CI-Service, der sich perfekt für Cloud-native Anwendungen eignet. Hier liegt allerdings auch der große Nachteil von Drone CI: Befindet man sich noch in einer Welt voller Legacy-Anwendungen und hat sich noch nie mit Containern beschäftigt, ist ein anderer CI-Service vermutlich besser geeignet.

GitLab bietet mit GitLab CI eine gut integrierte CI-Erweiterung für das eigene Repository. Vor allem wenn man kein externes Repository in GitLab CI integrieren muss, lohnt es sich das eigene Projekt mit GitLab CI auszuprobieren. Kaum ein anderes CI-System bietet so viele Features ohne Konfigurationsaufwand. Das einzige nennenswerte Manko sind die Kosten, falls zusätzliche Organisationsfeatures benötigt werden, und die enttäuschten Erwartungen, wenn Auto DevOps doch nicht für das eigene Projekt funktioniert.

Referenzen

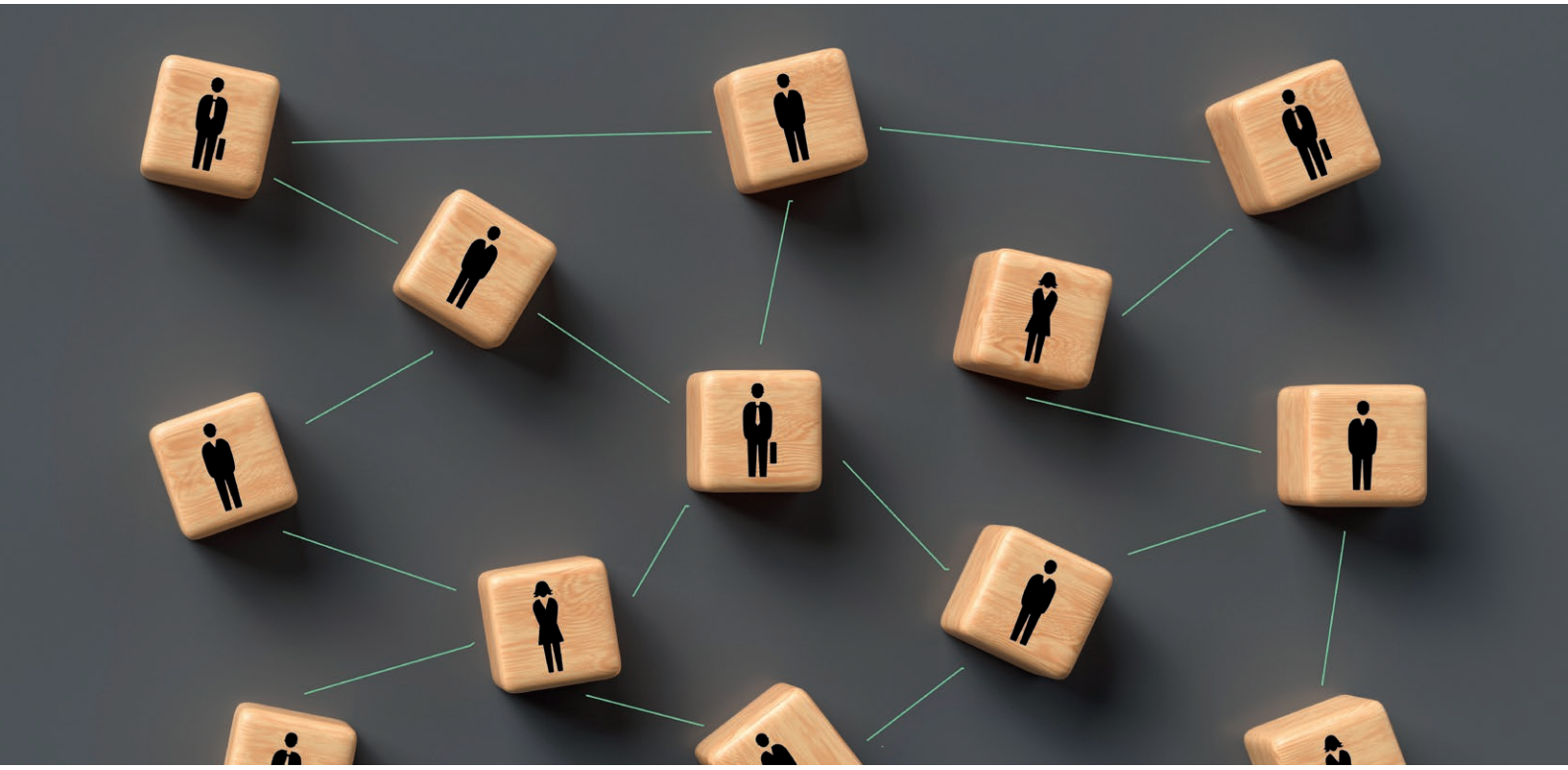
[1] *https://circleci.com/developer/orbs*

Dominik Röschke
MATHEMA Software GmbH
dominik.roeschke@mathema.de

Dominik Röschke arbeitet seit 2017 als Software-Entwickler für die MATHEMA Software GmbH. Hierbei unterstützt er Kunden, wertschaffende und kundenorientierte Software zu schreiben und zu betreuen. Als Speaker und auf Barcamps liegt sein Augenmerk auf dem Austausch mit anderen Entwickelnden und dem Kennenlernen neuer Technologien.

Nikolas May
MATHEMA Software GmbH
nikolas.may@mathema.de

Nikolas May arbeitet als Consultant für die MATHEMA Software GmbH. Seit 2008 beschäftigt er sich mit der Entwicklung von Softwarelösungen für Unternehmen. Zusätzlich arbeitet er seit 2013 mit Java EE und unterstützt Kunden bei der Entwicklung und Integration von Enterprise-Anwendungen. Sein aktuelles Steckenpferd ist DevOps.



Distributed Tracing mit Jaeger und Java

Gunnar Hilling, Hilling IT GmbH

Nicht zuletzt durch die Popularität von Microservices-Architekturen hat das Thema „Distributed Tracing" an Wichtigkeit gewonnen. Musste man hierfür früher auf teure und aufwendig zu wartende Werkzeuge wie AppDynamics [1] oder Dynatrace [2] zurückgreifen, sind in den letzten Jahren gleich mehrere produktionsreife, freie Implementierungen entstanden, die Tracing-Daten zentral sammeln, aufbereiten und visualisieren können.

Wenn heute eine Applikation neu- oder weiterentwickelt wird, so wird häufig eine Microservices-Architektur zugrunde gelegt. Ohne hier Vor- und Nachteile diskutieren zu wollen, kann jedoch festgehalten werden, dass die Verteilung eines einzigen Aufrufs über mehrere Systeme das Auffinden der Ursachen für Fehler und Performance-Probleme erschwert. Dies ist nicht nur bei Microservices-Anwendungen so, sondern gilt heute generell für die allermeisten Systeme, denn kaum eine komplexe Anwendung dürfte komplett gekapselt in einer VM oder in einem Container laufen.

Hier stellt sich die Frage, wie man einen verteilten Methodenaufruf verfolgen kann. Woher rühren eventuelle Fehler? Wo geht die Zeit für einen Aufruf tatsächlich verloren? *Distributed Tracing* kann diese Fragen beantworten und beschäftigt sich hierbei im Kern mit zwei Begriffen:

- Ein *Span* stellt die Daten eines spezifischen Methodenaufrufs dar. Er besitzt einen Namen, über den er auch technisch identifiziert werden kann. Bei der Erfassung eines Span werden die Start- und Endzeiten des Aufrufs gespeichert. Spans können dabei auch verschachtelt sein, es gibt Eltern-Kind-Beziehungen zwischen Spans.
- Ein *Trace* ist ein Ausführungspfad von Services im beobachteten System und besteht aus einem oder mehreren *Spans*. *Traces* werden implizit durch zusammengehörende Spans definiert. Die Spans bilden dabei einen gerichteten Graphen.

In der Auswertung bekommen wir später dann zunächst die Traces angezeigt, die inklusive aller zugehörigen Spans dargestellt und analysiert werden können. Die Auswertung, Visualisierung und Beobachtung von Traces ermöglicht es, Ursachen für Fehler sowie Performance-Probleme in verteilten Systemen effizient zu finden. Ebenso wird die Überprüfung von Optimierungen deutlich vereinfacht.

Ansatz

Um dies zu ermöglichen, benötigen wir folgende Komponenten:

- ein Protokoll, um die Span-Informationen zwischen den Systemen zu übertragen. Dies geschieht bei HTTP-Aufrufen über zusätzliche Header, bei Jaeger *[3]* standardmäßig durch den Header *uber-trace-id*
- ein API, um Traces zu erzeugen und abzuschließen sowie diese Informationen weiterzuleiten. Dieses wird durch den OpenTracing-Standard *[4]* definiert
- einen Server, der die Analyse übernimmt und die Daten sinnvoll aufbereitet und visualisiert

Durch die Standardisierung gibt es die Wahl zwischen mehreren Open-Source-Implementierungen. Für die Beispiele werden wir Jaeger verwenden, da es bei Quarkus *[5]*, das als Java-Framework verwendet wird, der Standard ist. Eine aktuelle Übersicht über die Implementierungen ist auf der Homepage des OpenTracing-Projekts zu finden *[6]*.

Kurze Einführung in das OpenTracing-API

Es gibt im Wesentlichen drei Möglichkeiten, Spans und damit Traces zu erstellen:

- Das API wird direkt aufgerufen und erlaubt die maximale Kontrolle über die Erzeugung und die Eigenschaften der Spans. Diese Methode sollte in der Praxis typischerweise in Bibliotheken verwendet werden, sodass man im eigentlichen Anwendungscode mit dem API kaum in Berührung kommt.
- Es existiert ein Framework, das die Verwaltung der Spans für uns übernimmt. Diesen Ansatz werden wir anhand der Integration in MicroProfile 3.3 *[7]* kennenlernen. Der Vorteil ist ein höherer Abstraktionsgrad. Außerdem können Spans automatisch erstellt werden. Dies ist beispielsweise bei REST-Aufrufen sinnvoll. Für Verfeinerungen sind hier Annotations verwendbar.
- Mit einem Java-Agent können Anwendungen, die direkt keine Unterstützung für OpenTracing bieten, nachträglich instrumentiert werden. Java-Agents werden typischerweise von den kommerziellen Werkzeugen verwendet. Sie sind auch für OpenTracing verfügbar, sollen aber nicht weiter besprochen werden, da es eher darum geht, Anwendungen von vornherein „fit" für verteiltes Tracing zu machen.

Zunächst soll die direkte Nutzung des API aus einer MicroProfile-Anwendung heraus vorgestellt werden. Anschließend werden die vereinfachten Integrationsmöglichkeiten im Rahmen einer Quarkus-Anwendung betrachtet.

Beispiele

Um den Code mitsamt einem funktionierenden Jaeger-Backend besser nachvollziehen zu können, kann dieser von GitHub geklont werden: Unter *[8]* liegen die Sources und auf oberster Ebene außerdem eine *docker-compose.yml*, über die direkt die benötigten Docker-Container gestartet werden können, um den fertigen Code selbst ausprobieren zu können. Sollen die lokal gebauten Images verwendet werden, müssen die Tags in der *docker-compose.yml* entsprechend angepasst werden.

Alle Beispiele werden als REST-Anwendungen über HTTP am einfachsten über `curl` aufgerufen. Bei einer vorhandenen Docker-Installation können alle Container über `docker-compose up` gestartet werden.

Die direkte Nutzung des API wird im Modul *quarkus-hello* in der Klasse *DemoResource* dargestellt. Die eigentliche „Geschäftsmethode" `DemoResource.demo()` ruft jeweils fünf Mal drei interne Methoden auf, die mit kleinen Unterschieden einen neuen Span erstellen. Durch die JAX-RS-Annotationen und die Docker-Konfiguration lässt sich die Methode per `curl http://localhost:8080/demo/` aufrufen.

Die erste Methode `DemoResource.createChildSpan()` erzeugt zunächst einen neuen Span *(siehe Listing 1)*. Dieser erhält einen Namen und einen eigenen `Tag` mit dem Namen „mytag". Sowohl den Namen als auch den `Tag` mit Wert findet man später in den Trace-Informationen wieder. Anschließend wird der Span gestartet und direkt oder nach einer kurzen Wartepause – um Arbeit zu simulieren – beendet.

Durch den Aufruf von `scopeManager().activate()` wird der neue Span als aktiver Span im aktuellen Thread gesetzt, sodass er später über den ScopeManager wieder referenziert werden kann oder wie hier, zu Demo-Zwecken, zusammen mit dem aktuellen Scope automatisch beendet wird, wenn der Scope geschlossen wird, nämlich am Ende des `try-with-resources`-Blocks. Ein Span besteht unter anderem aus:

- einem Funktionsnamen („Operation")
- Start- und Ende-Zeitstempel
- optionalen Tags (Key-Value-Paare)
- optionalen Logs (evenfalls Key-Value-Paare)
- einem SpanContext zur eindeutigen Identifikation
- optionalen Referenzen auf verwandte Spans, genauer auf deren SpanContext

Der aktuelle Span kann innerhalb einer Anwendung über den Tracing-*Scope* gespeichert und geladen werden, es ist also immer maximal ein Span aktiv.

Nach dem Start der Container und dem Aufruf der `demo`-Methode per `curl (http://localhost:8080/demo)` können wir uns das

```
Span span = tracer.buildSpan("child span")
        .withTag("mytag", "a child span")
        .withTag("index", index)
        .start();
try (Scope scope = tracer.scopeManager().activate(span, true)) {
    sleep(10L * index);
}
```

Listing 1: Händische Span-Erzeugung

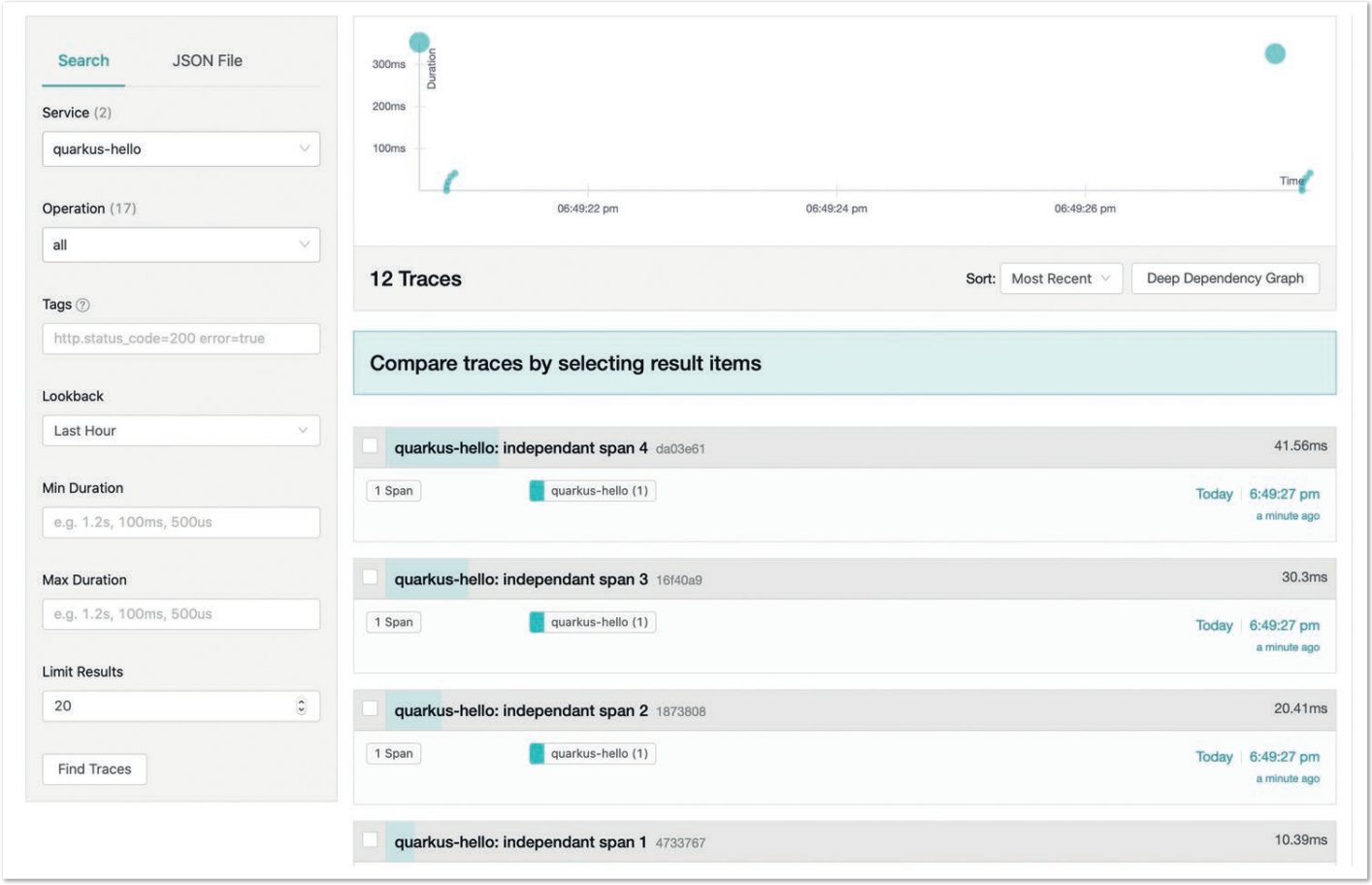


Abbildung 1: Startseite der Jaeger-Oberfläche (© Gunnar Hilling)

Ergebnis im Jaeger-Frontend anschauen. Hierzu rufen wir im Browser die URL http://localhost:16686 auf, wählen links oben den Service „quarkus-hello" und klicken links unten auf „Find Traces". Wir erhalten eine Übersicht ähnlich *Abbildung 1*. Die Suchoptionen benötigen wir hier zunächst nicht.

Im Beispiel wurde die URL zwei Mal aufgerufen, daher existieren auf der Zeitleiste jetzt zwei „Cluster". Für das Ergebnis der gerade vorgestellten Methode `DemoResource.createChildSpan()` klicken wir auf einen der dicken Kreise im oberen Bereich und erhalten jetzt eine Liste der Traces und zugehörigen Spans *(siehe Abbildung 2)*. Im unteren Bereich werden wie erwartet die im Code erzeugten Spans in zeitlicher Abfolge inklusive der hinzugefügten *Tags* dargestellt. Durch die Länge der Balken ist direkt die Ausführungszeit ablesbar.

Es stellt sich die Frage, warum ein umschließender Span auf oberster Ebene existiert: Dieser wird automatisch von der JAX-RS-Implementierung in Quarkus erzeugt, da es ja gerade Sinn und Zweck ist, ein verteiltes Tracing zu realisieren. Daher muss beim Aufruf einer Funktion von „außen" ein eventuell existierender Span automatisch aus den HTTP-Headers extrahiert werden. Ein neu erzeugter Span wird automatisch als „child-span" angesehen und entsprechend in der Hierarchie unter dem darüberliegenden Span eingeordnet. Ein `child-span` gehört logisch zum `parent-span`. Dieser hängt daher aus Tracing-Sicht von der Beendigung aller `child-spans` ab.

Bildet dieses Verhalten die Programmlogik nicht richtig ab, da beispielsweise ein asynchroner Prozess angestoßen wird, der unabhängig laufen soll, kann alternativ für den erzeugten Span der `parent-span` ignoriert werden und stattdessen das *Tag* `follows_from` anstatt `child_of` gesetzt werden (Methode `DemoResource.createReferencingSpan()`). Leider wird dieser momentan identisch visualisiert, wie man in *Abbildung 2* sehen kann.

In `DemoResource.createIndependantSpan()` wird die dritte Möglichkeit gezeigt. Hier wird der aktive Span ignoriert und keine Referenz gesetzt, daher tauchen die hier erzeugten Spans unabhängig auf der Übersicht auf, nämlich als kleine „Punkte" im unteren Bereich der Zeitleiste in *Abbildung 1*.

Wie funktioniert jetzt aber das Thema „Distributed" in diesem einfachen Beispiel? Nun, zunächst gar nicht, da der Aufrufer, nämlich `curl`, noch keine Trace-Informationen übermittelt. Informationen über Spans müssen beim verteilten Tracing an zwei Stellen übermittelt werden:

- Beim Erstellen und Beenden eines Span über das API wie im ersten Beispiel müssen diese Informationen an den Tracing-Server, im Beispiel Jaeger, übermittelt werden.
- Wird ein weiteres System aufgerufen, während ein Span aktiv ist, muss die Information über den aktiven Span an dieses System zusammen mit dem Aufruf übermittelt werden. Dies geschieht über einen HTTP-Header. Die hier verwendete Bibliothek verwendet dafür standardmäßig noch das Jaeger-eigene Format mit dem Header `uber-trace-id`. Jaeger unterstützt auch die Protokolle *Zipkin* und den noch relativ neuen Standard *W3C Trace-Context [9]*.

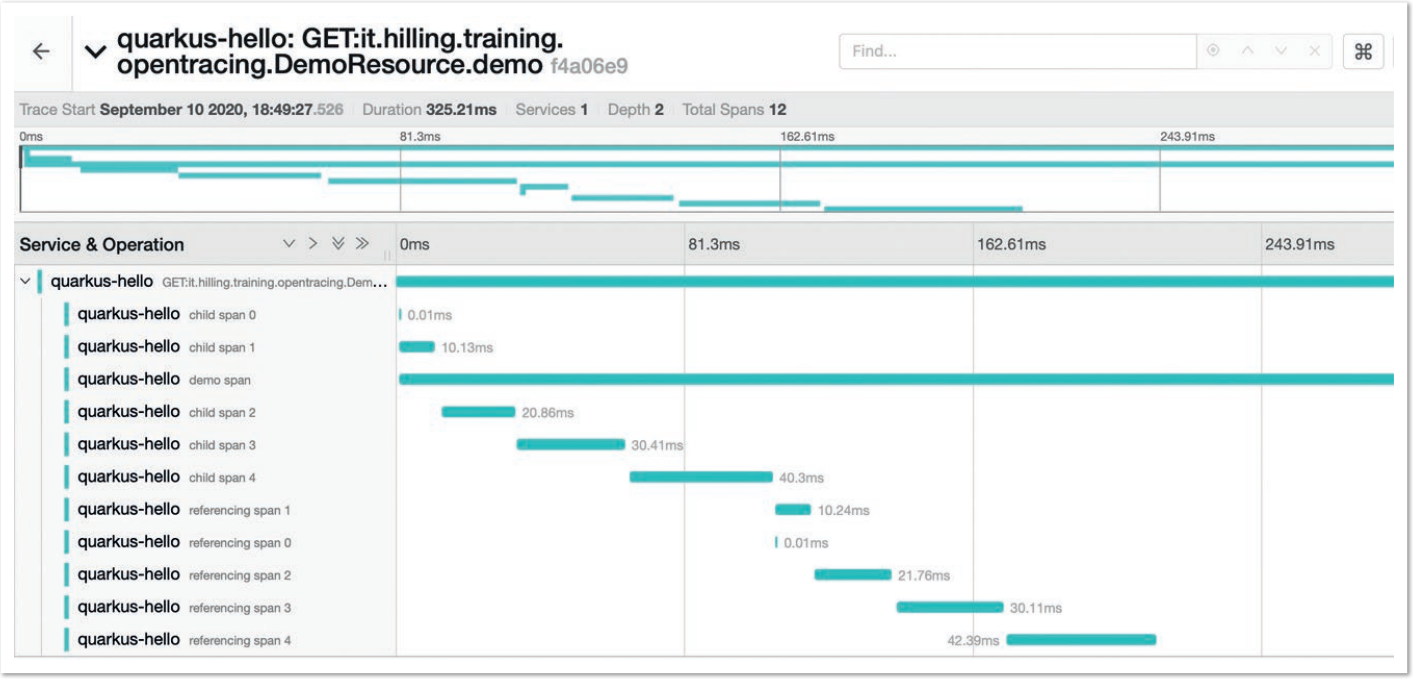


Abbildung 2: Trace der Methode `DemoResource.createChildSpan()` (© Gunnar Hilling)

Diese Header werden durch die MicroProfile-Implementierung clientseitig automatisch gesetzt und auf dem Server automatisch ausgewertet. Voraussetzung hierfür ist natürlich die Verwendung der Standard-Bibliotheken. Im Beispiel wird JAX-RS verwendet und die Quarkus-MicroProfile-Implementierung sorgt dafür, dass der Span für den REST-Aufruf automatisch erstellt wird *(siehe Listing 2)*: Man benötigt keinen zusätzlichen Code!

Wenn beim Aufruf der REST-Ressource der Parameter `world` übergeben wird, wird ein zweiter REST-Service, der in einem weiteren Container läuft, aufgerufen (`curl localhost:8080/hello/world`). Die Auswertung ist in *Abbildung 3* dargestellt. Dort ist zu sehen, dass die Spans automatisch um einige Informationen zu den Aufrufen ergänzt werden. Die unterschiedlichen Systeme werden für eine bessere Übersicht farblich unterschiedlich dargestellt. Zu sehen ist außerdem ganz unten ein Datenbank-Aufruf, zu dem wir gleich noch kommen werden.

Für den REST-Aufruf werden sowohl client- als auch serverseitig Spans erzeugt, sodass auch die Zeiten für den Aufbau der Verbindung und die Verarbeitung in den Stacks auf beiden Seiten sichtbar werden. Auf der Seite des `world`-Service ist außerdem ein manueller API-Aufruf eingebaut (in `WorldResourceImpl.world()`), der den Span `worldimpl` erzeugt.

Der interne Aufruf der Datenbank-Ressource erzeugt ebenfalls einen Span, da die Methode `DatabaseResource.translate()` annotiert ist *(siehe Listing 3)*. Das Ganze funktioniert natürlich nur, weil es sich um eine CDI-Komponente handelt und die Annotation einen Interceptor *[10]* aktiviert.

Die Auswertung ist so schon ziemlich vollständig und erlaubt das einfache Auffinden von Bottlenecks und Fehlern in verteilten Systemen. Nun gibt es allerdings noch eine typische „Problemzone", die hier noch nicht mit einbezogen wurde, nämlich den Datenbank-Aufruf.

Da Datenbanken OpenTracing typischerweise nicht direkt unterstützen – dazu fehlt auch ein einheitliches Protokoll –, gibt es als Ausweg in Quarkus die Möglichkeit, diesen Support durch einen speziellen JDBC-Treiber „nachzurüsten". Hierzu wird die Abhängigkeit auf *opentracing-jdbc* hinzugefügt und die Datenbank-URL zu *jdbc:tracing:postgresql://postgresql:5432/gunnar* geändert. Dies aktiviert den OpenTracing-Treiber, der den eigentlichen Aufruf umschließt und dabei Aufruf-Informationen inklusive der Dauer des Aufrufs sammelt.

```
@RestClient
WorldResource worldResource;

@Path("/{name}")
@GET
@Produces(MediaType.TEXT_PLAIN)
public String hello(@PathParam("name") @NotNull String name) {
    LOG.info("called hello");
    if(name.equals("world")) {
        name = worldResource.world();
    }
    return "hello " + name;
}
```

Listing 2: Automatische Span-Erzeugung und -Weiterleitung

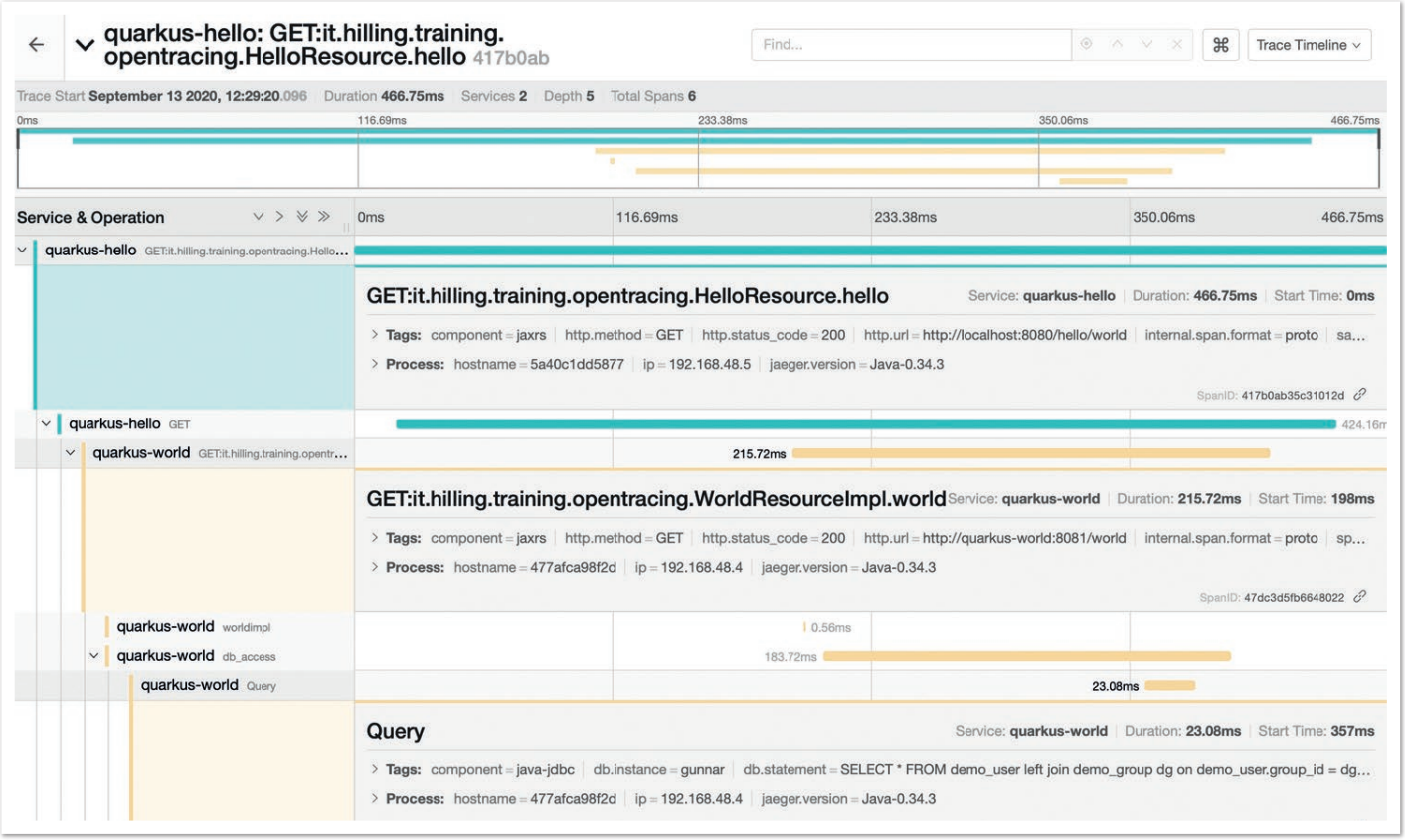


Abbildung 3 (© Gunnar Hilling)

Performance

Das Thema „Performance" ist beim Tracing ähnlich wichtig wie beim Logging. An dieser Stelle müssen wir zwei Aspekte unterscheiden:

1. Wie stark beeinflusst der Tracing-Code unsere Anwendungen?
2. Wie viele zusätzliche Ressourcen benötigt unsere Tracing-Infrastruktur?

Zum ersten Punkt kann gesagt werden, dass die Auswirkungen nur knapp messbar sein dürften. Der eigentlich Code-Aufwand ist minimal, vor allem, wenn man Tracing mit Logging vergleicht: Während Log-Aufrufe in fast jeder Klasse vorkommen dürften, muss der Tracing-Code nur dort angestoßen werden, wo tatsächlich ein Span erstellt oder weitergegeben werden soll. Kritisch ist hier die Kommunikation mit dem Tracing-Agent.

An dieser Stelle ist die Infrastruktur auf zwei Arten optimiert: Zum einen wird für die Kommunikation mit dem Tracing-Agent UDP verwendet, sodass diese Aufrufe jedenfalls nicht blockieren werden. Zum anderen wird, damit die Pakete trotzdem zuverlässig übermittelt werden, auf jedem physikalischen Host ein Tracing-Agent installiert, der die Informationen lokal puffert und asynchron zur zentralen Aufbereitung weiterleitet *[11]*.

Der Ressourcen-Aufwand für den zentralen Server ist ein anderes Thema, da er nicht zuletzt von der Anzahl der insgesamt vorhandenen Systeme abhängt, die ihre Daten senden. Ein probates Mittel, um den Gesamtaufwand zu senken, ist die Verwendung von Sampling bei der Erstellung von Spans. Hierbei wird nur ein festgelegter Prozentsatz der Spans erstellt und übermittelt.

```
@Traced(operationName = "db_access")
public String translate(String arg) {
```

Listing 3: Dieser Methodenaufruf erzeugt durch die Annotation einen zusätzlichen Child-Span.

Für ein Beispiel wurde die CPU-Last mit 10.000 Aufrufen für das oben dargestellte Szenario zwei Mal ermittelt. Die Aufrufe wurden mithilfe von Apache Bench durchgeführt *[12]*. Zunächst werden alle Spans erstellt, im zweiten Durchlauf ist das Quarkus-Profil „sampled" aktiv, bei dem nur 20 Prozent der Spans tatsächlich gemessen werden.

Die Messung *(siehe Tabelle 1)* erhebt nicht den Anspruch auf wissenschaftliche Korrektheit, aber der Trend beim CPU-Verbrauch des Jaeger-Containers ist wohl eindeutig erkennbar. Natürlich ist es beim Sampling nur noch möglich, Trends für die Aufrufzeiten zu ermitteln; „Ausreißer", die man gerne analysiert hätte, können durchaus verloren gehen. Hilfreich ist die Möglichkeit, die Sampling-Konfiguration zentral einzustellen und zu ändern *[13]*. Hierdurch können dann auch gezielt bestimmte Services genauer unter die Lupe genommen werden.

Standardisierung

Niemand begibt sich gerne ohne Not in ein „Vendor Lock-in". Die gute Nachricht ist hier, dass der Trend klar in Richtung offene Standards geht, sodass unsere Investitionen in Code und Infrastruktur in Zukunft nicht verloren sein werden:

- OpenTracing definiert clientseitige Schnittstellen für populäre Sprachen wie Java, JavaScript, Go und Python. Ebenso wird die Semantik der dort definierten Objekte geklärt

Container	CPU-Last
Alle Spans	
opentracing-quarkus_jaeger_1	**16,9 %**
opentracing-quarkus_postgresql_1	35,33 %
opentracing-quarkus_quarkus-hello_1	151 %
opentracing-quarkus_quarkus-world_1	88 %
Sampling 20 %	
opentracing-quarkus_jaeger_1	**2,6 %**
opentracing-quarkus_postgresql_1	24,48 %
opentracing-quarkus_quarkus-hello_1	183 %
opentracing-quarkus_quarkus-world_1	150 %

Tabelle 1: „ab -n 10000 -c 50 localhost:8080/hello/world"

- Eclipse MicroProfile unterstützt OpenTracing
- OpenTracing definiert mehrere Möglichkeiten zur Übertragung der Kontexte zwischen beteiligten Systemen. Praktisch relevant sind vor allem Jaeger („uber-trace-id"), Zipkin („x-b3-traceid") und der neue Standard des W3C („traceparent"). Viele Tracer unterstützen mehrere Formate, Jaeger alle drei
- Das OpenTelemetry-Projekt zielt darauf ab, in Zukunft die Interoperabilität auch auf Protokollebene und im Backend des Tracings weiter zu verbessern

Ein willkommener Nebeneffekt

Da im Rechenzentrum sicher schon ein zentrales Logging über Graylog, ELK oder Ähnliches etabliert ist, kann Distributed Tracing jetzt auch helfen, den Nutzen dieser Logs deutlich zu erhöhen.

Wie in *Abbildung 4* zu sehen ist, werden dem Logging-Context automatisch die aktuelle `traceId` und `spanId` hinzugefügt. Hierdurch ist es endlich möglich, verteilte Log-Ausgaben einfach und einheitlich einem einzelnen Aufruf oder Event zuzuordnen.

Selbst wenn vollständiges Tracing aller Aufrufe nur auf den Staging-Umgebungen aktiviert sein sollte, so bietet dies doch für die Entwicklung riesige Vorteile beim Auffinden von Fehlern.

Zu guter Letzt

Mit OpenTracing lassen sich Informationen flexibel und effizient zentral sammeln, die hilfreich für die Bewertung und das Monitoring von verteilten Systemen sind. Durch die Standardisierung ist zu erwarten, dass in Zukunft wohl alle relevanten Komponenten – auch in unterschiedlichen Programmiersprachen – den OpenTracing-Standard unterstützen werden. Dies wird mittelfristig dazu führen, dass der Support für OpenTracing so selbstverständlich in Frameworks, Servern und Bibliotheken enthalten sein wird wie der für Logging.

Wir Entwickler und Administratoren können uns freuen, da wir so in Zukunft deutlich bessere, einfachere und preiswertere Mittel zur Verfügung haben, unsere Systeme zu überwachen und Fehler zu finden.

Ein großer Vorteil beim verteilten Tracing ist, dass es mit relativ wenig Aufwand und Risiko in bereits vorhandene Systeme eingebaut werden kann. Es ist nicht nur in Cloud-, sondern auch in „klassischen" Umgebungen einsetzbar und kann auch durchaus Stück für Stück in die bestehende Landschaft nachgerüstet werden.

Referenzen

[1] *https://www.appdynamics.com*
[2] *https://www.dynatrace.com*
[3] *https://www.jaegertracing.io*
[4] *https://opentracing.io*
[5] *https://quarkus.io*
[6] *https://opentracing.io/docs/supported-tracers/*
[7] *https://microprofile.io*
[8] *https://github.com/guhilling/opentracing-quarkus.git*
[9] *https://github.com/w3c/trace-context*
[10] *https://docs.jboss.org/weld/reference/latest/en-US/html_single/#interceptors*
[11] *https://www.jaegertracing.io/docs/1.19/architecture/*
[12] *https://httpd.apache.org/docs/2.4/programs/ab.html*
[13] *https://www.jaegertracing.io/docs/1.19/sampling/*
[14] *https://opentelemetry.io/docs/concepts/what-is-opentelemetry*

Gunnar Hilling
Hilling IT GmbH
gunnar@hilling.de

Gunnar Hilling ist Physiker, hat sich aber nach seinem Studium frühzeitig der IT zugewandt. Dabei betätigte er sich zunächst in maschinennaher Programmierung auf verschiedenen UNIX-Systemen und war Ende der 90er als Assistent an der Universität Stuttgart. Seit der Jahrtausendwende dreht sich seine jetzt selbstständige Arbeit hauptsächlich um Java, Java-Enterprise und seit einigen Jahren auch gerne um das große Thema DevOps und Cloud. Nebenbei ist er der Lehre treu geblieben und arbeitet als Trainer unter anderem für Red Hat.

```
quarkus-hello_1  | 11:54:06 INFO  traceId=f640dbdb2af27563, spanId=f640dbdb2af27563, sampled=true [it.hi
.tr.op.HelloResource] (executor-thread-1) called hello
quarkus-world_1  | 11:54:06 INFO  traceId=f640dbdb2af27563, spanId=ff99ad79cb936eb, sampled=true [it.hi.
tr.op.WorldResourceImpl] (executor-thread-1) called world
```

Abbildung 4: Log-Ausgabe der Beispiel-Container (© Gunnar Hilling)

Gestern Wasserfall – Heute agil

Dr. Stefan Koch, ORDIX AG

Heutzutage werden in der IT-Entwicklung agile Vorgehensmodelle angestrebt. Sie sollen ausgeklügelte Wasserfallmodelle ablösen, die ein Garant für Software- und Prozessqualität sind. Wie funktioniert die organisatorische und technologische Umstellung einer zentralen IT-Anwendung in einer großen deutschen Handelsbank? Ist das agile Versprechen kürzerer Entwicklungszeiten mit Hochverfügbarkeit, Datensicherheit und einem hohen funktionalen Qualitätsanspruch vereinbar? Dieser Artikel ist ein subjektiver Rückblick auf den Migrationsprozess und Aspekte der Implementierung.

Qualitätsanspruch von IT-Anwendungen im Bankenumfeld

Wenn es um Geld geht, hört der Spaß auf – deshalb ist der Qualitätsanspruch von IT-Anwendungen im Bankenwesen immer schon besonders hoch gewesen. Im Folgenden werden einige Aspekte herausgegriffen, die auf den ersten Blick einer Agilitätsoffensive im Wege stehen.

Die funktionale Qualität nimmt einen sehr hohen Stellenwert ein. Einige Vorreiter agiler IT-Entwicklung sind gegenüber Fehlern bewusst tolerant: Der beste Test ist der Betrieb der Anwendung. Bei der Nutzung der Anwendung treten Fehler auf. Diese müssen nur schnell entdeckt und behoben werden. Im Bankenumfeld ist die Fehlertoleranz meist deutlich geringer: Bankkunden tolerieren Fehler bei Bankanwendungen viel weniger als beispielsweise bei Streamingdiensten. Darüber hinaus entstehen im Fehlerfall in der IT-Landschaft fehlerhafte Daten, die durch andere Anwendungen weiterverarbeitet werden. Der Aufwand für die Beseitigung des resultierenden Datenschiefstands ist enorm.

Um Fehler schnell beheben zu können, müssen sie einerseits entdeckt, andererseits auch analysiert werden. Aus Perspektive des Entwicklungsteams ist dafür ein direkter Zugriff auf die produktive Umgebung hilfreich: Anhand der Log-Dateien und der Produktionsdaten kann der Entwickler direkt nachvollziehen, was schiefgelaufen ist. Dieses Vorgehen ist aufgrund der Vertraulichkeit der Daten im Bankenumfeld undenkbar. Die Trennung von Entwicklung und Betrieb ist eine wesentliche organisatorische Maßnahme, um den Kreis der Personen zu verringern, die Zugriff auf Kundendaten haben.

Die Nachvollziehbarkeit des Softwarestands ist für die Release-Steuerung zentral. Kommt es beispielsweise zu einem schwerwiegenden Fehler im Betrieb, so soll durch ein Hotfix-Release der Fehler schnellstmöglich behoben werden. Das Hotfix-Release soll aber nur genau diesen einen Fehler beheben und keine weiteren Features beinhalten.

Hochverfügbarkeit und automatische Softwareinstallation (Deployment) sind seit mehr als einem Jahrzehnt im Bankenumfeld Standard. Durch den Einsatz von Applikationsservern und oft individuell entwickelten Deployment-Schnittstellen ist so beispielsweise das Ausrollen eines ganzen Releases, bestehend aus einer Vielzahl von Anwendungen, vollautomatisch möglich. Maßnahmen der Hochverfügbarkeit verbessern nicht nur die Verfügbarkeit im Falle eines Defekts, sondern ermöglichen auch Wartungsmaßnahmen im laufenden Betrieb. Agile Entwicklung von IT mit der Auflösung monolithischer Anwendungen darf hinter dem Erreichten nicht zurückbleiben.

Cloud fördert Agilität

Um die Agilität des Software-Entwicklungsprozesses zu unterstützen, wurde die Plattform der Software ausgetauscht. Anstelle eines Applikationsservers steht eine Cloud (Container-as-a-Service, CaaS) zur Verfügung. In der Cloud werden Anwendungen in einem sogenannten Container ausgeführt. Der Container besteht aus dem Prozess zusammen mit seiner Prozessumgebung. Ähnlich einer ausführbaren Datei ist das Docker-Image die Grundlage für einen Container. Die Entwickler stellen das Docker-Image für die Anwendung zusammen. Das Image enthält alle Software, die für die Ausführung der Anwendung erforderlich ist.

Die Cloud – im konkreten Fall OpenShift – stellt für ihre Container Dienste und Ressourcen bereit und sorgt für Skalierbarkeit und Hochverfügbarkeit *[1]* – all das unabhängig von der in der Entwicklung verwendeten Technologie. Durch den Einsatz dieser Cloud bekommen Entwickler die Möglichkeit, die eingesetzten Tech-nologien (Programmiersprachen, Frameworks, Applikationsserver, Datenbanken, Messaging-Systeme oder sonstige Komponenten) ausschließlich nach den Bedürfnissen der Anwendung auszusuchen. Diese Freiheit können die Entwickler für die Suche nach kreativen und effizienten Lösungen nutzen. Auch bei Veränderun-gen der eingesetzten Technologien verändert sich die Cloud als Ausführungsumgebung nicht.

Qualitätsgesicherter Entwicklungsprozess

Eine Cloud-Infrastruktur bereitzustellen, ist eine inzwischen wohlbekannte Aufgabe – darin aber die oben aufgeführten Qualitätsansprüche für den Software-Entwicklungsprozess zu integrieren, eine andere. Dazu müssen gezielt die Freiheiten des Entwicklers eingeschränkt und ein bankenkonformer Prozess installiert werden. Konkret geht es darum, wie das Ergebnis der Entwicklung in die Produktion kommt. Technisch wird dieser Prozess mit dem Begriff Continuous Delivery (CD) bezeichnet. Dieser sollte möglichst ohne manuellen Eingriff, also weitgehend automatisiert, erfolgen.

Die konkrete Implementierung von Continuous Delivery ist individuell von den konkreten Qualitätsansprüchen abhängig. *Abbildung 1* beschreibt den in der Bank installierten Prozess mit den beteiligten Komponenten und Akteuren.

Funktionale Qualität

Die funktionale Qualität wird dadurch abgesichert, dass die Software bis zum Release mehrere Stufen (Stages) durchlaufen muss. Jede Stufe prüft eine gewisse Qualität der Software. Nur wenn die Qualität erfolgreich nachgewiesen wurde, kommt die Software in die nächste Stufe. Für jede Stufe oder Stage wird eine eigene Umgebung in Form einer Cloud-Instanz eingesetzt *(siehe Abbildung 1)*: DEV-Cloud, System-Test-Cloud, Integration-Test-Cloud und PROD-Cloud.

In der DEV-Cloud wird der aktuelle Entwicklungsstand der Software ausgeführt. Die Software lässt sich bauen und hat alle Modultests erfolgreich bestanden. In die nachfolgende Stufe kommen jene Entwicklungsstände, die möglicherweise als Release verwendet werden können – die Release-Kandidaten. In der System-Test-Cloud wird die Software einem Systemtest unterzogen. In der Integration-Test-Cloud wird die Software auf Integrierbarkeit und sonstige nicht-funktionale Anforderungen überprüft. Softwarestände, die diese Überprüfungen bestanden haben, können anschließend in die Produktion (PROD-Cloud) gebracht werden.

Nachvollziehbarkeit des Software-Stands

Ist eine Software in der Produktion, so ist (nicht nur) im Fehlerfall wichtig zu wissen, welche Version des Quellcodes in Betrieb gebracht wurde. Der Zusammenhang zwischen Quellcode- und Release-Version muss aus diesem Grund im Continuous-Delivery-Prozess dokumentiert werden. In *Abbildung 1* sind dafür folgende Komponenten verantwortlich: der Build-Server, Source2Image-Container sowie die DEV- und Release-Repositories.

Der Source2Image-Container ist ein Prozess, der aus einer Version des Quellcodes ein Docker-Image erstellt. Den Quellcode bezieht der Source2Image-Container direkt aus dem Source-Code-Repository. Damit ist sichergestellt, dass nur versionierter Quellcode aus dem Source-Code-Repository verwendet wird. Der Build-Server dokumentiert diesen Build-Prozess: Die Nachvollziehbarkeit, welches Docker-Image aus welcher Quellcode-Version erstellt wurde, ist damit gewährleistet.

Der Source2Image-Container legt das erstellte Docker-Image in das DEV-Repository ab. Um in der Test- und schließlich auch in der Produktionsumgebung installiert zu werden, wird das Image aus dem DEV-Repository in das Release-Repository kopiert. Die eigentliche Installation wird durch ein Ticketsystem gesteuert. Ein Ticket

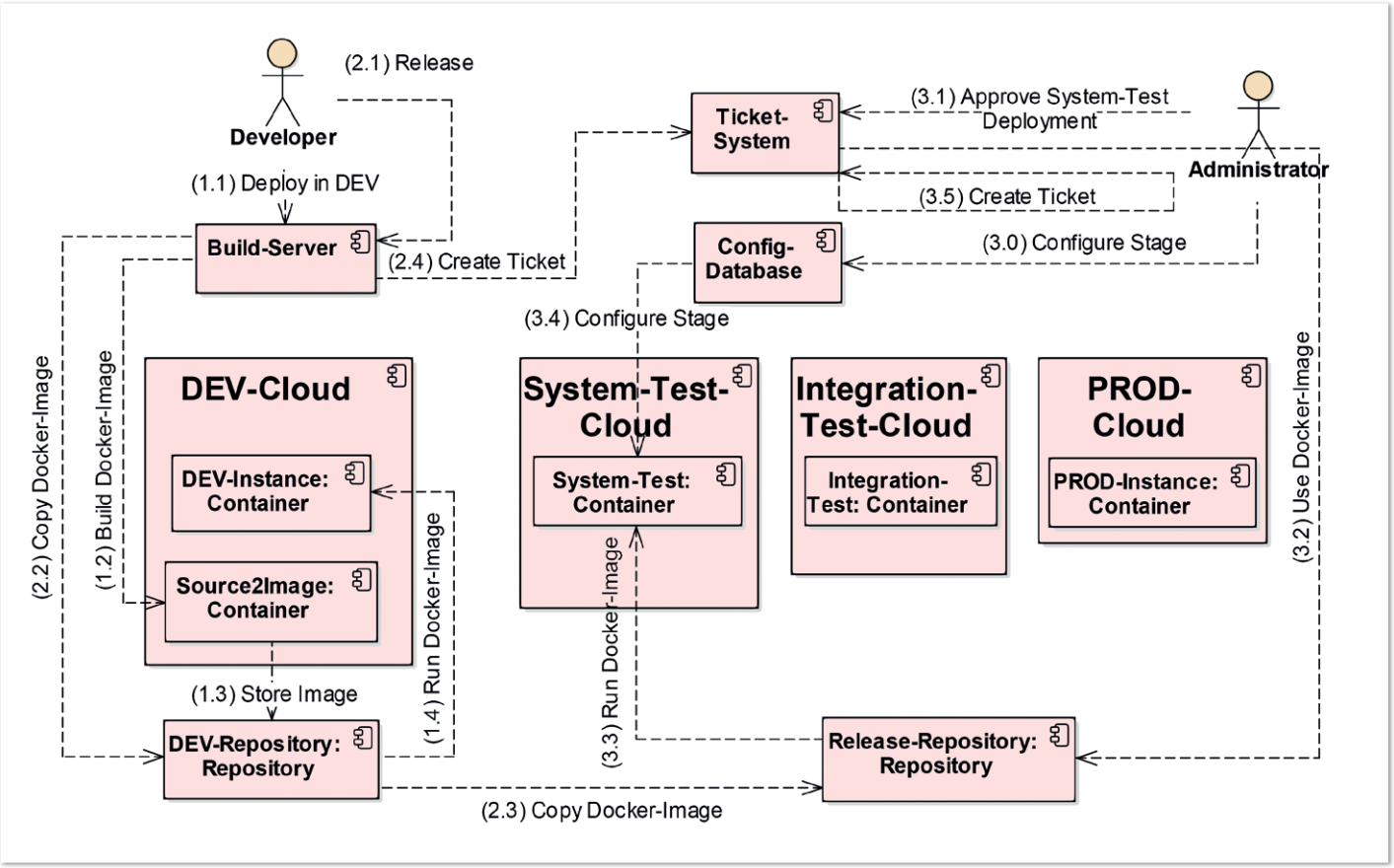


Abbildung 1: Überblick über den Continuous-Delivery-Prozess (© Dr. Stefan Koch)

enthält die Information, welches Docker-Image auf welche Umgebung installiert werden soll. Das Ticketsystem erweitert damit auch die Nachvollziehbarkeit: Welche Quellcode-Version wurde in welche Stage installiert?

Erfahrungen

Agile Softwareentwicklung in einer Cloud klingt nach einer großen Autonomie einer kleinen Gruppe von Entwicklern. In der Zeit der Migration vom Applikationsserver in die Cloud hat es sich jedoch anders angefühlt. Die Implementierung von Continuous Delivery (CD) wird als Service für mehrere Entwicklungsprodukte der Bank zur Verfügung gestellt. Das erspart den Entwicklungsteams auf der einen Seite, selbst einen solchen Prozess zu entwickeln, und gibt andererseits die Sicherheit, dass der Prozess bankenkonform ist. Im Gegenzug muss das Entwicklungsteam zunächst den CD-Prozess verstehen und dann die Anwendung darin integrieren. Die Aufgabe der Entwickler besteht darin, über Maven-Plug-ins den Prozess *(1.1) Deploy in DEV* und *(2.1) Release* aus *Abbildung 1* zu konfigurieren. Dabei definieren die Entwickler auch, welche Attribute in der Config-Database hinterlegt werden. Alle übrigen Prozesse stellt die Bank als Service zur Verfügung.

Auch die technologische Unabhängigkeit, die der Cloud-Einsatz prinzipiell ermöglicht, ist relativ. Basis-Images werden durch die Bank zur Verfügung gestellt. Im konkreten Fall wird als Ersatz für den Applikationsserver ein vorkonfigurierter Apache-Tomcat eingesetzt.

Da der CD-Prozess als Service bereitgestellt wurde, konnte sich das Entwicklungsteam auf die Kernthemen für die Umstellung konzentrieren: Migration der Ausführungsumgebung von einem vollwertigen Applikationsserver auf Apache-Tomcat sowie die Integration der Anwendung in den bestehenden CD-Prozess. Für die Integration steht dem Entwicklungsteam jederzeit Support durch den CD-Service zur Verfügung – dieser war gerade in den Anfängen auch dringend erforderlich.

Sicherlich gibt es nach wie vor Optimierungspotenzial: Wie viele Container werden in der Produktion benötigt, mit welchen CPU- und Speicherressourcen sind diese auszustatten? Die Cloud-Umgebung liefert für diesen Lernprozess die erforderliche Freiheit. Es können vorübergehend einfach zusätzliche Produktivinstanzen zur Verfügung gestellt werden. Diese Flexibilität hat sich auch ausgezahlt, als die Last durch besondere Umstände kurzfristig vervielfacht wurde.

Agilität bedeutet eine Verkürzung der Release-Zyklen. Anwendungsfeatures lassen sich so zeitnah in die Produktion bringen. Die zur Verfügung gestellten Umgebungen unterstützen zunächst aber genau ein Release. Aufgrund der Abhängigkeiten zu anderen Anwendungen der Bank ist es jedoch wünschenswert, Features längerfristig vorzubereiten – hier stößt der Anspruch der Agilität auf die Realität, dass Veränderungen an verteilten Anwendungen längerfristig geplant und vorbereitet werden.

Zusammenfassung

Die Migration vom klassischen Wasserfallmodell in ein agiles Vorgehensmodell unter Nutzung einer Cloud wurde erfolgreich durchgeführt. Software- und Prozess-qualität werden weiterhin uneingeschränkt eingehalten. Der Qualitätsanspruch steht aber in Konkurrenz zu Zielen der Agilität. So ist ein vollautomatisiertes Continuous Delivery nicht möglich, da einerseits ein Hand-Over zwischen Entwicklern und Administratoren aus Datenschutzgründen erforderlich ist, andererseits ein Change-Prozess notwendig ist, nicht zuletzt, um bestehenden Abhängigkeiten zwischen Anwendungen Rechnung zu tragen.

Durch den Einsatz der Cloud-Lösung wird die Agilität des gesamten Software-Entwicklungsprozesses gefördert. Das Entwicklungsteam erhält Freiheitsgrade bei der Auswahl des Technologie-Stacks. Die Betriebsweise lässt sich einfach auf Laständerungen anpassen. Der Aufwand für die Implementierung eines bankenkonformen Continuous Delivery darf aber nicht unterschätzt werden.

Referenzen

[1] Dokumentation zu OpenShift: *https://docs.openshift.com/*

Dr. Stefan Koch
ORDIX AG
info@ordix.de

Dr. Stefan Koch ist als Principal Consultant bei der ORDIX AG seit 1998 beschäftigt. Als Bereichsleiter für Application Development betreut Dr. Stefan Koch eine Vielzahl der Entwicklungsprozesse des Unternehmens. Neben seiner Referententätigkeit im Seminarzentrum der ORDIX AG ist er auch auf Konferenzen als Speaker tätig.

Machine Learning in Produktion: Die zwei Seiten eines Daten-Projekts

Mark Keinhörster, codecentric AG

Vom Feature-Engineering über das Training bis hin zur Visualisierung; Tools und Frameworks für Data Scientists gibt es zuhauf. Aber was kommt nach einem Proof-of-Concept? Da Data-Science-Experimente eher experimenteller Natur sind, stellen sie klassische IT-Projekte vor Herausforderungen. Neben der Anwendung selbst müssen nun auch die trainierten Modelle sowie deren Code für Training und Evaluation den Ansprüchen der modernen Softwareentwicklung gerecht werden.

Data-Science (DS) und Machine-Learning (ML) sind in der Wirtschaft mittlerweile angekommen und immer mehr Unternehmen nutzen neuronale Netze, Entscheidungsbäume und Konsorten zur Implementierung neuer Features oder ganzer Geschäftsideen. Das verwundert wenig, denn moderne Frameworks wie TensorFlow, Keras oder Pytorch gestalten den Einstieg verhältnismäßig leicht, ohne tief in die Mathematik der Algorithmen einsteigen zu müssen. Rund um die Thematik entwickelt sich ein großes Ökosystem aus Tools und Frameworks, die die Nutzer vom Feature-Engeering über das Training der Modelle bis hin zur Visualisierung der Ergebnisse unterstützen. Leider ist die Entwicklung von ML-Modellen, das ML-Engineering, immer noch vom experimentierfreudigen Data-Science-Ansatz beeinflusst. Hyperparameter werden ad hoc justiert

und Datensätze unterschiedlich vorverarbeitet, bis die Modellqualität ausreichend ist. Das mag in kleineren Projekten funktionieren, skaliert bei längeren Laufzeiten und größeren Teams jedoch nur noch unzureichend. Das Ergebnis sind häufig schwer zu reproduzierende Resultate und frustrierte Entwickler.

Was wir haben und was wir wollen

Die Grundvoraussetzungen für Data-Science-Projekte sind mit einer Projektidee und einem großen Datenbestand mittlerweile oft gegeben. Mit diesen Voraussetzungen macht das Team sich auf den Weg, um möglichst schnell Projektziele zu erreichen: Modell-Vorhersagen und, daraus resultierend, Profite aus einem potenziellen Produkt. In vielen schnelllebigen Experimenten werden Hypothesen in Form von Modellen aufgestellt und verworfen, bis erste verwertbare Resultate erzielt werden. Die Experimente folgen dabei einem klaren Schema:

- Daten beschaffen und speichern,
- Daten säubern,
- Daten vorverarbeiten,
- Modell trainieren,
- Ergebnisse evaluieren.

Aber wie geht es weiter, wenn das Experiment geglückt ist? Die konkrete Frage, die über Daten-Projekten und deren Erfolgen schwebt, ist: Was muss noch für einen wirtschaftlichen Nutzen passieren? Dabei stellen sich für die Entwickler und Data Scientists weitere Fragen, die beantwortet werden müssen. Beispielsweise, woher die echten Daten kommen und welche Bereiche des Unternehmens dafür involviert werden müssen. Jedoch auch, wie die Vorhersagen zum Nutzer kommen und wie die Nutzer möglichst früh mit einbezogen werden können.

Die zwei Seiten eines Datenprojekts

Das zuvor beschriebene Experiment ist die explorative Seite eines Datenprojekts, in der sich ein Data Scientist am wohlsten fühlt. Dort geht es primär um die Identifizierung neuer Features im Dataset, die Optimierung des Modells oder die Visualisierung der Ergebnisse. Daneben gibt es noch eine andere, auf Automation bedachte Seite, auf der sich Data- und ML-Engineers Gedanken über den Schnitt einzelner Module für eine möglichst hohe Testabdeckung und einen robusten Betrieb machen sowie Artefakte über Pipelines gebaut und bereitgestellt werden. Damit ist klar, in einem Datenprojekt werden nicht nur ML-Modelle entwickelt, sondern es entsteht ein automatisiertes ML-System mit dem Ziel, konkrete und messbare Mehrwerte aus den Daten zu generieren.

Abbildung 1 zeigt den Data-Science-Zyklus eines ML-Systems. In jedem groben Schritt von der Vorbereitung bis hin zur Vorhersage werden die Daten und die entstehenden Modelle auf Validität geprüft und versioniert. Bei der Datenbeschaffung wird immer wieder kontrolliert, ob die Annahmen über den Datensatz, wie beispielsweise die Verteilung der Klassen oder Grenzwerte in einzelnen Spalten, noch korrekt sind oder ob sich der Datensatz über die Zeit rapide verändert hat. Genauso verhält es sich mit einem neuen Modell und dessen Vorhersagen. Nur wenn die Vorhersagen weiterhin auf dem Referenzdatensatz, dem Ground-Truth, korrekt sind, wird das neue Modell für die Nutzer bereitgestellt.

Die Komplexität von ML-Systemen wird jedoch häufig unterschätzt. Dabei bestätigten wissenschaftliche Ausarbeitungen von Google bereits 2015, dass der eigentliche ML-Code nur den kleinsten Teil an einem ML-System ausmacht. Das Sammeln und Verifizieren der Daten, das Konfigurationsmanagement, die Infrastruktur für den Betrieb sowie die Überwachung haben einen weitaus größeren Anteil. Aus diesem Grund bestehen Data-Projekte zu gleichen Teilen aus Data-Science- und aus Data-Engineering-Kompetenzen.

Modellqualität validieren

Ein trainiertes Modell arbeitet am zuverlässigsten direkt nachdem es in Produktion deployt wurde. Zu diesem Zeitpunkt entsprechen die Trainingsdaten noch am ehesten den Eingaben, die während des Produktionsbetriebs entstehen. Über die Zeit können sich die Eingaben jedoch verändern. Dies kann verschiedenste Gründe haben:

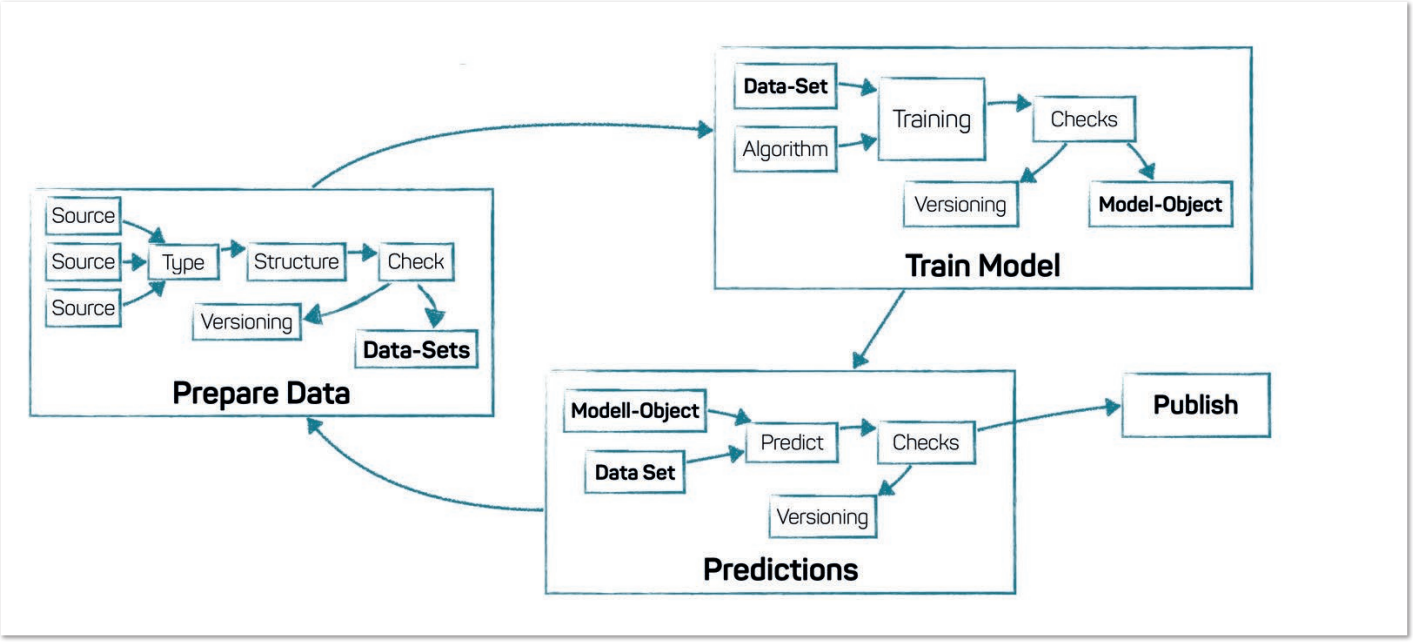


Abbildung 1: Zyklus eines ML-Systems (© codecentric AG)

- Der Fachbereich entscheidet sich dafür, weitere Dokumentenklassen in die Applikation zu integrieren.
- Das Nutzerverhalten bei der Benutzung des Webshops wird durch neue Werbemaßnahmen beeinflusst.
- Oder es waren noch nicht genug Trainingsdaten vorhanden und es wurde lediglich mit den Daten aus dem Winter trainiert.

Einige dieser Versätze, auch Skews genannt, sind leichter zu erkennen, andere schwieriger. Beispielsweise lässt sich ein geändertes Schema (Schema-Skew) meist schnell durch konkrete Fehler in der Verarbeitung erkennen und lösen. Schwieriger wird es bei Feature-Skews, die auftreten, wenn in der Produktion Feature-Werte auftauchen, die es beim Training gar nicht gab. Dies kann an den Daten liegen (im Sommer gibt es nur positive Temperaturen) oder aber auf Bugs im Code hindeuten, wie beispielsweise die Zentrierung eines Wertes um null beim Training, die im Produktionscode vergessen wurde. Neben dem Schema- und Feature-Skew kann es noch Versätze in der Verteilung der Features zwischen den Trainings- und Echtdaten geben (Distribution Skew). Ein Fehler, der häufig bei falschen Sampling-Methoden entsteht, um stark unbalancierte Datensätze abzufangen.

TensorFlow bietet mit „TensorFlow Data Validation" ein entsprechendes Framework an, das sich gerade für Modelle im TensorFlow-/Keras-Universum anbietet.

Modell-Management und Reproduzierbarkeit

Neben den „harten Zahlen", die in den Daten und den Modellen liegen, hat der Experimentieransatz auch Auswirkungen auf den Softwareentwicklungsprozess. Denn in Datenprojekten existieren nicht nur Code und Konfigurationen, sondern auch Trainings- und Testdaten sowie Artefakte für Modelle, wie beispielsweise die Modellarchitektur, Hyperparameter oder Vorhersageergebnisse *(siehe Abbildung 2)*.

Die Verwaltung der Artefakte von Hand ist mühselig, unübersichtlich und schwierig nachzuvollziehen. Denn mit jeder kleinen Änderung entsteht bereits ein neues Experiment mit neuen Ergebnissen und einem neuen deploybaren Artefakt. Damit muss jede Änderung am Dataset, an den Parametern oder an der Architektur, wie es auch im Software-Engineering zum Standard gehört, nachvollziehbar und reproduzierbar versioniert werden. Häufig reicht ein kurzer Blick auf die aktuelle Datenlage mit `ls -1` in der Commandline, um zu zeigen, wie die Verwaltung der Artefakte gehandhabt wird *(siehe Listing 1)*.

Das liegt daran, dass sich die meisten Versionskontrollsysteme (VCS), wie beispielsweise Git, mit Binärdateien wie Trainingsdaten und Modellgewichten schwertun. Außerdem fehlt die Integration der Ergebnisse, um eine schnelle Vergleichbarkeit zwischen verschiedenen Versionen zu schaffen. Das Tool Data Version Control (DVC) schafft hier Abhilfe. DVC ist ein offenes Versionskontrollsystem für ML-Projekte, das sich nahezu nahtlos in den Git Workflow integriert und Funktionalitäten wie die Versionierung großer Binärdateien oder leichtgewichtige Pipelines für Reproduzierbarkeit und Deployments bereitstellt.

Schon die Initialisierung von DVC im Repository schreibt sich wie ein Git-Befehl von der Tastatur *(siehe Listing 2)*.

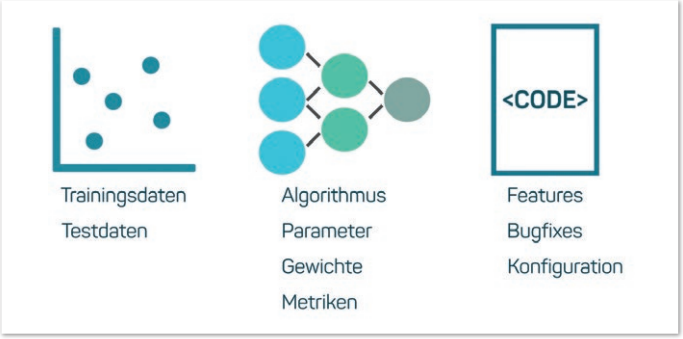


Abbildung 2: Artefakte in datengetriebenen Projekten (© Martin Fowler, https://martinfowler.com/articles/cd4ml.html)

```
$ ls -1
data_2020_10_01.tgz
data_2020_10_01_v2.tgz
data_2020_10_01_v2_test.tgz
data_2020_10_01_v2_re_test_calibration.tgz
data_2020_10_01_v3.tgz
data_2020_10_01_v3_final.tgz
data_2020_10_01_v3_final_final.tgz
data_2020_10_01_v3_DIESMAL_RICHTIG.tgz
...
```

Listing 1: Daten „versionieren"

```
$ dvc init
$ git status
    new file:   .dvc/.gitignore
    new file:   .dvc/config
$ git add .dvc
$ git commit -m "Initialize dvc"
```

Listing 2: DVC-Repo initialisieren

Die Konfigurationen, die DVC für das Tracking benötigt, werden im Unterordner .dvc gespeichert und mit Git versioniert. Experimente werden, wie Features in der Software-Entwicklung, in Git-Branches verwaltet. Große Dateien verwaltet DVC über einen externen Speicherort. Im Repository finden sich lediglich Referenzen auf die Binärdateien. Die Daten selbst werden lokal in einem Cache und bei Bedarf auch in der Cloud persistiert *(siehe Abbildung 3)*. Hier wird eine große Menge an Optionen geboten: Von lokalen Ordnern über Fernzugriffe per SSH bis hin zu Blob-Storages aller bekannten Provider unterstützt DVC alles, was das Entwicklerherz begehrt. Auch hier weist der Befehl in *Listing 3* wieder eine große Ähnlichkeit zu Git auf.

DVC bietet noch weitaus mehr als nur die Versionierung großer Dateien. Mit `dvc run` können beispielsweise ganze Trainingspipelines gebaut werden, die mit `dvc metrics` über alle Branches hinweg verglichen werden können.

Bereitstellung von ML-Modellen

Ein weiterer großer Punkt, der für alle Datenprojekte gleichermaßen wichtig ist, ist die Bereitstellung der Modelle. Doch Frameworks gibt es zuhauf und von PyTorch bis SciKit-Learn haben alle ihre Vor- und Nachteile. Aber eines haben sie gemein: Sie nutzen ihre eigenen Speicherformate. Das führt dazu, dass frühe technologische Entscheidungen nur schwer rückgängig gemacht werden können.

Während SciKit-Learn zum Beispiel für einen einfachen Einstieg bekannt ist, bietet TensorFlow mit Keras ein deutlich mächtigeres API für Deep-Learning-Modelle. Dabei gehen nicht nur die einzelnen Potenziale der Frameworks verloren, auch im Deployment und Betrieb müssen häufig Abstriche gemacht werden. SciKit-Learn-Modelle werden häufig direkt im Code integriert oder es wird ein individueller REST-Service um das Modell geschrieben. TensorFlow enthält mit TensorFlow Serving hingegen einen robusten Modelserver, mit dem Modelle als Service bereitgestellt werden können. Einen gemeinsamen Nenner zu finden, ist jedoch schwierig.

ONNX – ein Format, um sie alle zu einen

Aus diesem Grund ist der Open-Neural-Network-Exchange-Standard (ONNX) entwickelt worden. Mit dem Ziel, ohne viel Aufwand ein Modell mit SciKit-Learn zu entwickeln und anschließend beispielsweise mit TensorFlow Serving im Produktionsbetrieb bereitzustellen. Der Modellstandard beschreibt die Spezifikation eines Berechnungsgraphen, durch den Informationen als sogenannte Tensoren (mehrdimensionale Vektoren) fließen. Die detaillierte Spezifikation mit allem, was dazu gehört, kann im GitHub-Repository von ONNX *[1]* nachgelesen werden.

Grundsätzlich wird ONNX dazu verwendet, ein Modell von einer Technologie zur anderen zu konvertieren. Beispielsweise von SciKit-Learn nach TensorFlow, um eine bereits bestehende TensorFlow-Serving-Infrastruktur zu nutzen. Mittlerweile entsteht jedoch rund um den Standard ein breites Ökosystem, dass allen voran von Microsoft getrieben wird. Beispielsweise entsteht mit dem ONNX-Runtime-Server, der zu dem Zeitpunkt des Artikels noch im Beta-Stadium war, eine vielseitige Alternative zu TensorFlow Serving, die mit ONNX als Standardformat arbeitet.

Für jede unterstützte Technologie stellt ONNX eine eigene Bibliothek zur Konvertierung bereit. Für SciKit-Learn-Modelle ist dies *sklearn-onnx*. *Listing 4* zeigt beispielhaft den Export eines Entscheidungsbaums aus SkiKit-Learn in das generische ONNX-Format.

Die Funktion `convert_sklearn` exportiert das Modell. Einzig die Eingabevektoren für den Zielgraphen müssen vorher definiert werden. In diesem Fall besteht der Vektor aus einer Instanz mit zwei Fließ-

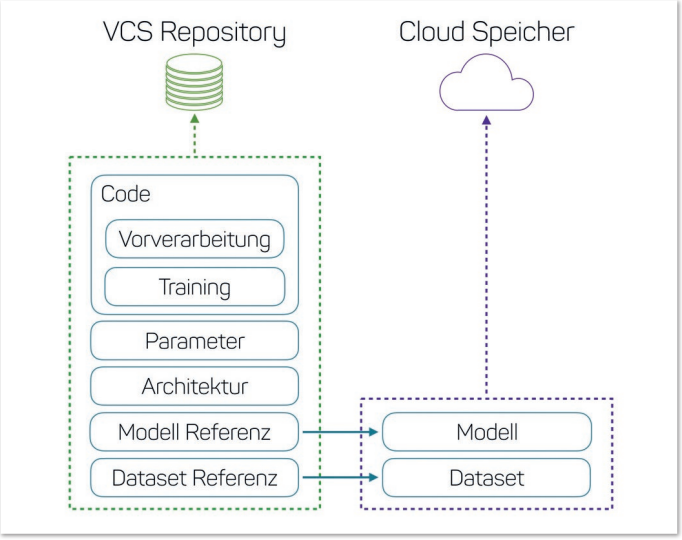


Abbildung 3: Aufteilung der Daten in VCS und Cloudspeicher (© Mark Keinhörster)

```
$ dvc remote add -d local_storage /data/...
Setting 'local_storage' as a default remote.
```

Listing 3: Remote konfigurieren

kommazahlen. Der Eingabetyp ist ein Tupel, bestehend aus Name und Datentyp. Im Fall von *Listing 4* ist `float_input` der Variablenname im Graph und `FloatTensorType` der Datentyp für Fließkommazahlen. Weitere unterstützte Frameworks sind unter anderem CoreML, Keras, TensorFlow, Pytorch, MATLAB oder MXNET.

Service oder Laufzeitumgebung?

Nach Training und Export tun sich verschiedene Wege zur Bereitstellung auf. Der wohl simpelste Ansatz ist die direkte Integration des Modells in den Applikationscode. Für diesen Ansatz stellt Microsoft die *ONNX-Runtime* bereit. Damit wird das Training und die Evaluation des Modells dem jeweiligen Framework überlassen und die Inferenz wird einheitlich über die Runtime ermöglicht, unabhängig vom Framework. Damit können beispielsweise schnelle Proof of

```
from skl2onnx import convert_sklearn
onx = convert_sklearn(entscheidungsbaum,
                      initial_types=[('float_input',
                                      FloatTensorType([1, 2]))])
with open("model.onnx", "wb") as f:
    f.write(onx.SerializeToString())
```

Listing 4: Exportieren des Entscheidungsbaums

```
import onnxruntime as rt
instanz = [1.5, 1.0]
session = rt.InferenceSession("model.onnx")
eingabe = session.get_inputs()[0].name
ergebnis = session.run(None,
                       input_feed={
                           eingabe: datensatz.astype(np.float32)
                           })[0]
```

Listing 5: Inferenz mit der ONNX-Runtime

Concepts mit einfachen Modellen entwickelt werden, die im weiteren Verlauf durch komplexere, beispielsweise mit Keras entwickelte Deep-Learning-Modelle ersetzt werden. Wie ein Aufruf des Modells über die Runtime funktioniert, zeigt *Listing 5*.

Erst einmal wird das Modell mithilfe der `InferenceSession` von der Festplatte geladen. Mit `run` wird die Vorhersage für einen Datensatz durchgeführt. Damit die Eingaben korrekt zugeordnet werden können, wird über `input_feed` der Name der Eingabevariablen (`float_input`, wie in *Listing 4* beschrieben) mitgegeben.

Im Gegensatz zur direkten Integration des ML-Modells in die Business-Logik sieht der klassische Ansatz die Bereitstellung über einen Modelserver vor. Der lädt ein exportiertes Modell in den Hauptspeicher und stellt verschiedene API-Schnittstellen zur Inferenz, wie beispielsweise REST oder GRPC, bereit. Dadurch müssen sich die Entwickler weder um das Laden oder Verwerfen von Modellversionen kümmern, noch müssen sie sich Gedanken um die Implementierung der Endpunkte machen. Dafür ist die Zieltechnologie jedoch festgesetzt und das Team und der Betrieb entsprechend eingeschränkt. Dies ist insbesondere dann wichtig, wenn das Modell auch auf mobilen Endgeräten oder in Cloud-Umgebungen ausgeführt werden soll.

Ein neuer, modernerer Weg ist die Paketierung und Bereitstellung des ML-Codes als Docker-Container. Tools wie BentoML oder Cortex verpacken ML-Modelle Technologie-agnostisch und mit wenig Gluecode in Container, die ein einheitliches REST-API exponieren. Neben der reinen Containerisierung bieten diese Tools zumeist auch Front-Ends für das Modellmanagement an. In Verbindung mit dem einheitlichen Format ONNX lassen sich so Modelle ohne Einschränkung in der Cloud, im Cluster – oder bei Verzicht auf die Paketierung – auch auf mobilen Endgeräten bereitstellen, ohne auf die Vorzüge eines etablierten Modelservers zu verzichten.

Feedback und Monitoring

Nun sind der Entwicklungsprozess strukturiert, die Modelle trainiert und validiert sowie erste Modellversionen über APIs bereitgestellt. Der Produktionsbetrieb gleicht jedoch immer noch einem Blindflug, denn bislang wird noch nicht kontrolliert, ob das Modell auf den Produktionsdaten wirklich arbeitet wie angenommen. Der schnellste Weg für ein erstes fachliches Monitoring ist das Loggen von Samples aus dem Produktionsbetrieb. Darauf können grundlegende Metriken bereitgestellt werden, die als Indikatoren für ein gesundes oder fehlerhaftes Modell dienen. Beispielsweise fachliche Indikatoren wie die Verteilung der Klassen innerhalb der Predictions oder die Grundverteilung der Inputs (Feature- und Distribution-Skew).

Daneben sollten natürlich auch technische Metriken wie die durchschnittliche Anzahl der Vorhersagen pro Stunde, die Cycle-Time oder die Ressourcen-Auslastung des Hosts beobachtet werden. Wirklich interessant wird es, wenn der Fachbereich Feedback zur Modellperformance geben kann. Leider ist der Weg von mitgeschnittenen Samples bis hin zur Analyse durch den Fachbereich lang. Er lohnt sich jedoch, weil die Erkenntnisse für neue Trainingsdurchläufe zurück in die Trainingsdaten wandern und so zukünfige Modelle verbessern. Abhilfe können hier Feedback-Kanäle in Form kleiner Anwendungen schaffen, die es dem Nutzer ermöglichen, die Ergebnisse der Modelle in der Applikation zu bewerten. Das bedeutet jedoch oft zusätzlichen Code und mehr Implementierungsarbeit, weshalb solche Kanäle häufig schlicht nicht vorgesehen sind.

Fazit

Trotzdem die Zahl der Datenprojekte stetig wächst und der Anteil an Machine-Learning-Modellen in diesen Projekten immer größer wird, ist es immer noch schwierig, Praktiken, Vorgehen und Toolsets mit den hohen Anforderungen des Software-Engineerings in Einklang zu bringen. Glücklicherweise gibt es in nahezu allen großen Teilbereichen datengetriebener Projekte mittlerweile Tools, die hohe Automatisierungs- und Standardisierungsgrade ermöglichen. Werkzeuge wie TensorFlow Model Validation, DVC, ONNX und BentoML schaffen Stacks, die sich auf die Operationalisierung von ML-Projekten fokussieren und damit eine standardisierte und zielgerichtete Softwareentwicklung ermöglichen, wie man sie aus klassischen Softwareprojekten kennt. Dabei ist DVC ein praktisches Werkzeug für die Daten- und Modellverwaltung, das reproduzierbare Experimente ermöglicht und gleichzeitig die Schmerzen der Datenpersistenz und des Datenaustauschs im Team erheblich mindert. ONNX ermöglicht den Einsatz heterogener Frameworks für Data Scientists und erleichtert gleichzeitig das Deployment.

Referenzen:

[1] *https://github.com/onnx/onnx*

Mark Keinhörster
codecentric AG
mark.keinhoerster@codecentric.de

Mark ist im Big-Data-Zoo zu Hause und bringt Erfahrungen mit Hadoop und Apache Spark mit. Außerdem beschäftigt er sich mit Docker, Cloud-Technologien und Machine Learning.

Agile Gamification

Roman Simschek, Agile Heroes GmbH

Welche Möglichkeiten kann man nutzen, um möglichst viele Skeptiker von agilen Methoden überzeugen zu können? Diese Frage wird im heutigen Zeitalter von zahlreichen Unternehmen, Teams und Einzelpersonen gestellt. Eine kurze Antwort hierauf würde lauten: durch Gamification. Hierbei handelt es sich um eine spielerische Herangehensweise an das Thema Scrum. Aber was versteckt sich hinter diesem Begriff und wie kann man diese Methodik anwenden? Eine aufschlussreiche Erklärung bietet der nachfolgende Artikel.

Gamification ist ein agiler, aber klar strukturierter und moderierter Prozess mit integrierten spielerischen Elementen. Dadurch können Fragestellungen aus der Geschäftswelt bearbeitet und beantwortet werden. Die Methode eignet sich bestens in Unternehmen, Teams oder sogar bei Einzelpersonen. Durch den spielerischen Einsatz von Legoteilen sollen neue Ideen generiert und Probleme gelöst werden. Gamification soll dazu dienen, Missverständnisse von Scrum zu beseitigen.

Im Großteil unserer zahlreichen Seminare und Trainings zum Thema Agilität und Scrum machen unsere Teilnehmer nämlich kontinuierlich ähnliche Erfahrungen: Falls Scrum im Unternehmen nicht erfolgreich sein sollte, dann liegt es häufig nicht daran, dass die Methode nicht ausreichend gereift ist. Der Grund ist vielmehr, dass diejenigen, die

das Scrum-Framework anwenden sollen, sich nicht von der Methode überzeugen lassen. Hinzu kommt, dass es oftmals auch an der nötigen Motivation fehlt, sich auf die Methode einzulassen.

Basierend auf dieser Erkenntnis haben wir für unsere Kunden ein Gamification-Format entwickelt, dessen Ziel es ist, Agilität und Scrum spielerisch zu vermitteln. Anhand verschiedener Methoden können sich Teilnehmer nun selbst von den Vorteilen des Scrum-Frameworks überzeugen. Die Trainingsteilnehmer sollen hierbei Agilität am eigenen Leibe erleben und spielerisch lernen, wie Scrum am besten angewendet werden soll. Wir haben die Erfahrung gemacht, dass Teilnehmer, die Scrum live erlebt haben, besser verstehen, warum diese Methode sinnvoll ist und dass sie richtig angewendet auch funktioniert. Das Motto hierbei lautet: Scrum live und real miterleben. Dieses Motto wird dadurch gestärkt, dass der Lerneffekt, der auf der bereits weitverbreiteten Erkenntnis basiert, dass, wenn der Mensch etwas selbst anwenden kann, der Lerneffekt viel höher und nachhaltiger ist, als wenn das Wissen nur auditiv vermittelt wird. Wie eine Hands-on-Experience aussehen kann, wollen wir im Folgenden noch genauer erläutern.

Was ist eine Gamification?

Bei einer Gamification eines komplexeren Themas geht es im Wesentlichen darum, Inhalte oder Wissen in spielerischer Form zu vermitteln, damit alle Teilnehmer zukünftig Scrum richtig anwenden können und wollen. Zusätzlich wird oft auch ein wettbewerblicher Charakter, beziehungsweise auf Neudeutsch eine „Challenge", zwischen einzelnen Spielteilnehmern oder Teams integriert. Im Vergleich zu der Methode des Dozierens führt diese Methodik unumgänglich zu einem weitaus besseren Verständnis des Themas Scrum, da die Teilnehmer dazu gezwungen sind, sich kontinuierlich mit dem Scrum-Framework auseinanderzusetzen und dieses korrekt anzuwenden.

Wie könnte also eine erfolgreiche Verknüpfung der Themen Gamification, Agilität und Scrum aussehen? Das Format, das wir entwickelt haben, heißt „SCRUM Hero Island". Es handelt sich hierbei um eine Gamification, bei der Scrum mit Lego simuliert wird. Die Teilnehmer des Trainings werden zunächst in mehrere Scrum-Teams mit jeweils zwischen vier und sechs Personen eingeteilt *(siehe Abbildung 1)*. Diese Teams teilen sich anschließend untereinander auf die einzelnen Rollen gemäß dem Scrum-Framework auf: Scrum Master, Product Owner und Development-Team *(siehe Abbildung 2)*. Darüber hinaus besteht bei SCRUM Hero Island die Möglichkeit, Scrum mit einem oder auch mit mehreren Teams zu simulieren. Die Teilnehmer müssen hierbei selbstorganisiert vorgehen, was bereits ein wichtiges Grundprinzip des Scrum-Frameworks darstellt. Hierzu erhält jedes Team verschiedenste Spielutensilien wie eine Spiellandkarte, Lego, Post-its und eine agile Uhr. Sobald sich alle Teams gefunden haben, alle Rollen verteilt wurden und jedes Team mit den nötigen Utensilien ausgestattet wurde, geht es schon los…

Wie funktioniert SCRUM Hero Island?

Sobald alle Teams ausreichend vorbereitet sind, beginnt nun im nächsten Schritt die Simulation von SCRUM Hero Island. Letztlich basiert die Methodik von Gamification auf einer Fantasiegeschichte. Das bedeutet, dass es sich bei SCRUM Hero Island nur um eine fiktive Darstellung handelt, die beispielhaft für ein besseres Verständnis der Teilnehmer genutzt wird. Die Geschichte von SCRUM Hero Island

Abbildung 1: Spielerische Teamfindung (© Agile Heroes)

Abbildung 2: Die Rollen eines SCRUM-Teams (© Agile Heroes)

Abbildung 3: Stakeholder (© Agile Heroes)

handelt von dem erfahrenen Scrum-Coach Jeff, der jahrelang Unternehmen weltweit bezüglich agiler Methoden und Scrum beraten hat.

Jeff kennt sich also bestens mit dem Thema Scrum aus, möchte sich aber fortan mehr mit seinen privaten Interessen als mit der Arbeit und der agilen Beratung auseinandersetzen. Jeff hat den Entschluss gefasst, sich nun von seinem über Jahre ersparten Geld eine Trauminsel südlich von Hawaii zu kaufen. Der erfolgreiche Coach tauft die Insel „Agility Island". Auf dieser Insel möchte er nun für sich und seine Familie sein Traumressort bauen lassen: eine moderne Traumvilla mit Außenpool, Bootsanlegeplatz, einem schönen, majestätischen Garten, einem Grillplatz und noch vielem mehr. Hier kommen

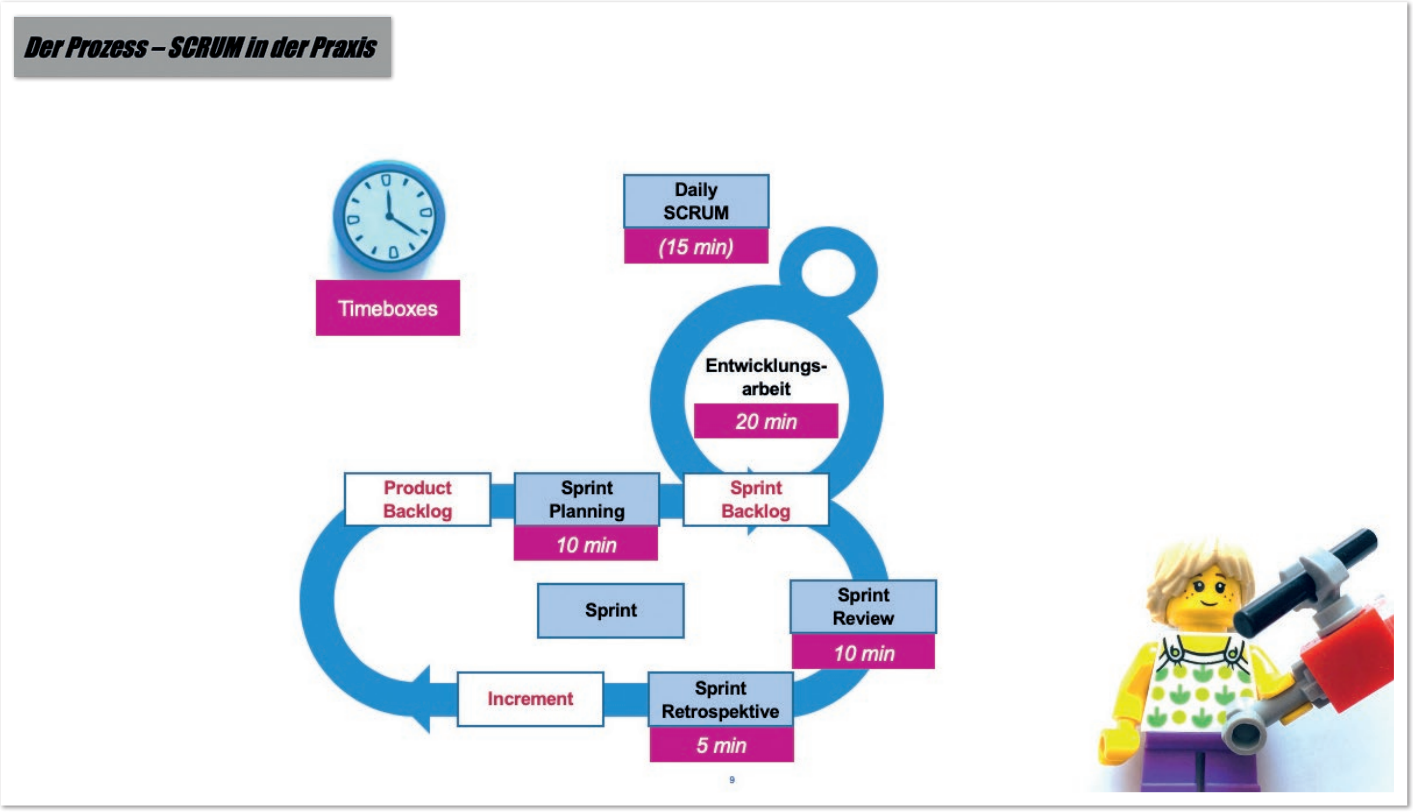


Abbildung 4: Scrum-Prozess – Sprint (© Agile Heroes)

nun unsere Teilnehmer ins Spiel, denn dies ist der Auftrag an die einzelnen Teams: der Bau von Jeffs Traumressort mit all seinen Wünschen auf Agility Island.

Jeff und seine Familie, seine Frau Pamela, sein Sohn Kenny und der Hund Scrummy, haben immer wieder neue Ideen, wie ihr Traumressort aussehen soll und wie genau ihre Träume verwirklicht werden sollen. Jeff und seine Familie stellen im Rahmen der Gamification die Stakeholder im Sinne von Scrum dar *(siehe Abbildung 3)*. Die Teilnehmer sind also darauf fokussiert, dass jegliche Vorstellungen und Wünsche der Stakeholder von ihnen erfüllt werden.

Eines der wichtigsten Merkmale von Scrum ist das iterative Vorgehen in Sprints *(siehe Abbildung 4)*. Man arbeitet demnach inkrementell durch crossfunktionale Teams und versucht am Ende eines Sprints, ein potenziell marktfähiges Inkrement zu erstellen, zu dem es individuelle Abnahmekriterien gibt, die zuvor erfüllt sein müssen. Nutzt man Gamification als Vermittlungskanal, könnten die Anforderungen an die Teams nun folgendermaßen aussehen: In bis zu fünf aufeinanderfolgenden Sprints arbeiten die Teams real mit Lego verschiedene Aufträge von Jeff und seiner Familie auf Agility Island ab.

Hierbei entstehen realitätsgetreu immer wieder neue Herausforderungen. Anforderungen und Abnahmekriterien ändern sich, neue Ideen kommen dazu und Hindernisse treten auf. Beispielsweise könnte sich Jeff nach Sprint 3 noch einen Whirlpool auf der Terrasse wünschen oder es könnte zu plötzlichen Problemen mit Materiallieferungen kommen, was wiederum einige Lieferschritte und Abhängigkeiten beeinflussen könnte. Die Teilnehmer sind also dazu gezwungen, Scrum in realitätsnahen Situationen anzuwenden.

Abbildung 5: Entwicklungsphase (© Agile Heroes)

Abbildung 6: Sprint Review (© Agile Heroes)

Während der SCRUM-Hero-Island-Simulation durchläuft das Team alle Events, die das Scrum-Framework bietet: Sprint Planning, Daily Scrum, Sprint Review und Sprint-Retrospektive. Das Sprint Planning dient hierbei der Planung des angefangenen Sprints, das Daily Scrum bietet dem Development-Team eine Plattform, um sich untereinander für die Lieferung des Produkts koordinieren zu können, die Sprint Review *(siehe Abbildung 6)* dient dem Scrum-Team und den Stakeholdern zum Identifizieren möglicher Verbesserungen für den folgenden Sprint und das gewünschte Produkt. Die Sprint-Retrospektive thematisiert das Miteinander des Scrum-Teams und individuelle Wünsche an deren Zusammenarbeit.

Während der Entwicklungsphase *(siehe Abbildung 5)* können sich die Teams spielerisch an der Entwicklung des Traumhauses von Jeff austoben. Nach jedem einzelnen Sprint macht sich eine Verbesserung der Teams bezüglich des Verständnisses des Scrum-Frameworks bemerkbar. Die Teams verbessern sich stetig in Bezug auf ihre Zusammenarbeit und die Performance. Durch die spielerische Herangehensweise der Gamification-Methode fällt es den Teilnehmern merkbar leichter, sich in die vorgegebene Simulation hineinzuversetzen und Scrum gleichzeitig anwenden zu können. So lernt jedes Team und jeder Einzelne, was die Vorteile inkrementellen Arbeitens sind und wie das Scrum-Framework am besten in der Praxis umgesetzt werden kann. Die Teilnehmer lernen, wieso es wichtig ist, die einzelnen Elemente von Scrum umzusetzen, sie fühlen sich durch die Hands-on-Erfahrung besser auf zukünftige agile Praxisprojekte vorbereitet und können auch komplexere Themen mit einem gesteigerten Selbstbewusstsein angehen.

Die Gamification-Methode zielt also darauf ab, dass Teilnehmer das Scrum-Framework auch tatsächlich anwenden können, obwohl eine Simulation nicht immer alle tiefgehenden theoretischen Themen des Scrum-Frameworks abdecken kann.

Was ist unsere Erfahrung aus der Praxis?

Was ist die Erfahrung des Formats Gamification mit Lego bei der Einführung von Scrum? Wie schon erwähnt ist es oft so, dass es in Organisationen einzelne Personen gibt, die Treiber dafür sind, Scrum als Projektmanagementmethode einzusetzen. Die Frage ist jedoch: Wie kann der Rest des Teams oder des Unternehmens überzeugt werden? Oft liegt der Schlüssel darin, Stakeholder davon zu überzeugen, dass Scrum die richtige Methode für ein Projekt ist oder aber einzelne Teams von der Nutzung von Scrum zu überzeugen, die schon seit Jahren in eingefahrenen Methoden und Pfaden arbeiten.

Hier stellt die Gamification ein gutes und erfolgreiches Instrument dar. Die Vorteile liegen darin, dass alle Teilnehmer spielerisch an agile Methoden gewöhnt werden und jeder Teilnehmer dazu ermutigt wird, aktiv zu partizipieren. Durch den positiven Teamspirit und die ansteckende Wirkung davon, mehrere Sprints mit Lego zu durchlaufen, entsteht eine Neugier auf mehr und Teilnehmer arbeiten plötzlich flexibler und motivierter an der Problemlösung. Viele Teilnehmer berichten, dass sie am liebsten noch tagelang weitergespielt hätten. Dies ist ein guter Grundstein, um in einem weiteren Schritt Scrum als Methode auf konkrete Situationen und Projekte anzuwenden.

Insofern sehen wir die Gamification als eine Möglichkeit, einen ersten Schritt zu gehen und Neugier sowie Bereitschaft auf Agilität in Organisationen zu schaffen. Zusätzlich empfehlen wir eine Betreuung bei der Einführung von Scrum durch einen Agile Coach oder einen erfahrenen Scrum Master, denn deren Aufgabe ist das Unterstützen von Scrum-Teams bei der Einführung und Nutzung des Scrum-Frameworks und sie können darüber hinaus einzelne Teams betreuen, beraten und überwachen, falls einzelne Scrum-Aspekte nicht zielführend ausgeführt werden. Auch hier kann natürlich Lego teilweise zum Einsatz kommen.

Es ist jedoch wichtig zu erwähnen, dass Lego nur selektiv eingesetzt werden sollte und das Modell der Gamification nicht dauerhaft Erfolg bringt, da es sich um eine Simulation handelt, die lediglich der spielerischen Vermittlung von Scrum dient. Beides ist gut, um initial und situativ eingesetzt zu werden. Man sollte Scrum jedoch möglichst schnell auf reale Situationen und Themen anwenden, um einen langfristigen Erfolg aus der agilen Methode generieren zu können.

Roman Simschek
Agile Heroes GmbH
rsimscheck@agile-heroes.de

Roman Simschek ist Partner und Gründer der Agile Heroes GmbH, einem Unternehmensberatungs- und Trainingsinstitut, spezialisiert auf Projektmanagement, insbesondere agiles Projektmanagement mit Scrum. Er betreut Kunden in Deutschland, Österreich und der Schweiz.

Mitglieder des iJUG

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

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: Stefan Kinnen. 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: Mylène Diacquenod
Redaktionsleitung: Lisa Damerow
Kontakt: redaktion@ijug.eu

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

Titel, Gestaltung und Satz:
Caroline Sengpiel,
DOAG Dienstleistungen GmbH

Fotonachweis:
Titel: Bild © Andrii Stepaniuk | *https://de.123rf.com*
S. 10: Bild © Andreas Prott | *https://stock.adobe.com*
S. 18: Bild © Maslakhatul Khasanah | *https://de.123rf.com*
S. 25: Bild © grandfailure | *https://stock.adobe.com*
S. 31: Bild © vska | *https://de.123rf.com*
S. 35: Bild © melpomen | *https://de.123rf.com*
S. 40: Bild © syaraku | *https://de.123rf.com*
S. 43: Bild © Andrii Torianyk | *https://de.123rf.com*
S. 45: Bild © phonlamaiphoto | *https://stock.adobe.com*
S. 51: Bild © rudall30| *https://stock.adobe.com*
S. 55: Bild © fotogestoeber | *https://stock.adobe.com*
S. 61: Bild © JesusmGarcia | *https://stock.adobe.com*
S. 62: Bild © Pavel Lunevich | *https://de.123rf.com*
S. 65: Bild © Chalermsuk Bootvises | *https://de.123rf.com*
S. 70: Bild © ribkhan | *https://stock.adobe.com*

Anzeigen:
Julia Bartzik, DOAG Dienstleistungen GmbH
Kontakt: anzeigen@doag.org

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

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

Inserentenverzeichnis

iJUG Verbund

Werden Sie Mitglied im iJUG!

Ab 15,00 EUR im Jahr erhalten Sie

30 % Rabatt auf Tickets der JavaLand

Jahres-Abonnement der Java aktuell

Mitgliedschaft im Java Community Process

www.ijug.eu

JavaLand

16. - 17. März 2021
als Online-Veranstaltung

Programm und Tickets unter

www.javaland.eu

8 Streams mit über 120 Vorträgen

Premium Sponsor

IBM

Gold Sponsors

aiticon DEVELOPING YOUR WEB

KARAKUN

OPEN KNOWLEDGE

REWE digital

tarent

Silver Sponsors

adesso business. people. technology.

AUSY IT starts with people

CGI

codecentric

cronn wir entwickeln software_

DW

GFT

Gradle

ilum:e

QAWARE SOFTWARE ENGINEERING

VORWERK

Präsentiert von: DOAG Heise Medien

Community Partner: iJUG Verbund

www.javaland.eu