jax®

Java-Trends 2023: State of the Art

JAX Whitepaper

Quarkus, Kotlin, Kubernetes, Continuous Integration, Continuous Delivery, API Design, Dev(Ops) Experience, Cloud-Native, Vulnerability-Management

jaxcon jax.konferenz JAX TV jax

jax.de

Inhalt

Kotlin-Docker-Image-Erstellung ohne Dockerfiles und Kubernetes-Auslieferung

Kotlin-Kubernetes-Backend-App

Dieser Artikel stellt eine Kombination aus Werkzeugen vor, die Kotlin-Quellcode kontinuierlich in Docker Images verpackt und die Docker Images direkt danach an Kubernetes ausliefert.

von Lothar Schulz

Softwareentwickler:innen arbeiten am performantesten, wenn sie sich auf das Lösen von Businessproblemen fokussieren können. Fokus erreichen sie idealerweise mit so wenigen Unterbrechungen ihrer Arbeitsumgebung wie möglich. Meiner Beobachtung nach arbeiten Kotlin-Entwickler:innen sehr häufig in diesem Kontext:

- Kotlin-Quellcode in einer integrierten Softwareentwicklungsumgebung (IDE [1])
- Kommandozeile
- verteiltes Codeverwaltungssystem

Jede Änderung dieses Kontexts führt zu Verzögerungen, die im Allgemeinen nicht gewollt sind.

Nur in Ausnahmefällen wird Backend-Software heutzutage nicht als OCI-Image [2] ausgeliefert. Container sind häufig Docker-Container [3]. Diese können mit Dockerfiles [4] erzeugt werden (Docker Images werden mit Dockerfiles erzeugt und ausgeführte Docker Images werden als Docker-Container bezeichnet). Dockerfiles verwenden eine einfache DSL, um ein Docker Image zu erstellen. Die Dockerfile DSL ist im Wesentlichen eine Sequenz von Befehlen, die nötig sind, um ein Docker Image wiederholbar zu erstellen. Dieses Vorgehen würde den oben beschriebenen Kontext ändern. Aus diesem Grund nutze ich einen Weg, Docker Images ohne Dockerfiles und ihre DSL zu erzeugen.

Docker-Image-Erzeugung mit JIB und Gradle

Gradle [5] ist eines der am weitesten verbreiteten Build-Systeme im Kotlin-Bereich. Dazu hat sicherlich der Schwenk von Groovy auf die Gradle Kotlin DSL [6] beigetragen, für die der gleiche Kotlin-Compiler genutzt wird wie für Kotlin-Quellcode. Alle Vorteile wie Codevervollständigung, Typsicherheit, Compilerfehler und -warnungen können von Entwickler:innen genutzt werden.

JIB [7] erlaubt, OCI-Images mit Kotlin (und Java) zu erstellen, ohne einen Docker Daemon zu benötigen. Somit ist auch kein Dockerfile nötig und die Entwickler:innen können im oben beschriebenen Kontext bleiben. JIB stellt ein Gradle-Plug-in [8] zur Verfügung. Dieses integriert sich nahtlos in die Gradle-Build-Descriptor-Datei (Listing 1).

Um nun ein Docker Image zu erzeugen, kann das Gradle-JIB-Ziel des Gradle Wrapper verwendet werden:

```
./gradlew jib
```

Die obige JIB-Konfiguration enthält interessanterweise nicht nur ein Basis-Docker-Image [9], sondern auch

Listing 1: build.gradle.kts (Ausschnitt)

```
plugins {
  ...
  id("com.google.cloud.tools.jib") version "3.3.0"
}

jib {
  from {
    image "openjdk:17@sha256:528707081fdb9562eb819128a9f85ae7fe00
                                                0e2fbaeaf9f87662e7b3f38cb7d8"
  }
  container {
    ports = listOf("8080")
    mainClass = this@Build_gradle.mainClassStr
  }
}
```

einen spezifischen Image Digest. Ohne diesen Digest besteht die Gefahr, dass das Erstellen des Image nicht exakt wiederholbar ist. Hintergrund dafür sind mögliche Basis-Image-Updates mit einem veränderten *latest*-Tag [10]. Werden nun gleiche Quellcodeversionen mit verschiedenen Basis-Images gebaut, sind die resultierenden Docker Images ebenfalls unterschiedlich.

Beim Fehlen einer Image-Digest-Definition informiert JIB mit einer Warnung darüber, die mit der Spezifikation des Image Digest nicht mehr auftritt. Mit diesem Set-up zum Erzeugen von Docker Images bleiben Kotlin-Entwickler:innen im oben beschriebenen Kontext.

Kontinuierliche Kubernetes-Auslieferung

Docker Images können lokal gebaut werden und mit kubectl zu Kubernetes ausgeliefert werden. Um neue Quellcodeänderungen kontinuierlich in Kubernetes zu testen, müsste nach jeder Quellcodeänderung ein neues Docker Image gebaut und im Anschluss an Kubernetes ausgeliefert werden.

Skaffold [11] ist ein System für die automatisierte und wiederholbare Auslieferung zu Kubernetes, das durch Quellcodeänderungen getriggert werden kann. Ebenso wie das Erstellen des Docker Image mit dem Gradle Wrapper auf der Kommandozeile kann auch Skaffold von der Kommandozeile aus gesteuert werden. Skaffold erfordert dazu Konfigurationsdateien, die ähnlich wie Kubernetes-YAML-Konfigurationen aufgebaut sind:

```
apiVersion: skaffold/v2beta29
kind: Config
build:
  artifacts:
    - image: gcr.io/ktor-jib/kjib-image
      jib: {}
```

Listing 2: deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web
  labels:
    app: web
spec:
  selector:
    matchLabels:
      app: web
  template:
    metadata:
      labels:
        app: web
    spec:
      containers:
        - name: web
          image: gcr.io/ktor-jib/
                 kjib-image
          ports:
            - containerPort: 8080
          readinessProbe:
            tcpSocket:
              port: 8080
            initialDelaySeconds: 5
            periodSeconds: 10
            failureThreshold: 2
          livenessProbe:
            tcpSocket:
              port: 8080
            initialDelaySeconds: 15
            periodSeconds: 20
            failureThreshold: 2
```

Skaffold benötigt Information über das Docker Image, das bei Quellcodeänderungen neu zu bauen ist. Das wird mit dem *image* Key in der vorletzten Zeile des obigen Abschnitts definiert. Zum Bauen des Docker Images soll JIB verwendet werden, das wird in der letzten Zeile definiert. Die entsprechenden Kubernetes-Konfigurationsdateien müssen das gleiche Docker Image referenzieren, z. B. so wie in Listing 2.

Skaffold erwartet Kubernetes-Konfigurationsdateien ohne weitere Konfiguration im Unterordner *k8s*. Somit können ggf. abweichende Kubernetes-Konfigurationen für andere Auslieferungs-Use-Cases in anderen Ordnern gespeichert werden.

Der Projekt-Quellcode ist mit allen bisherigen Beispielen auf GitHub [12] und Codeberg [13] verfügbar. Als lokale Kubernetes-Umgebung wird Minikube [14] verwendet sowie EKS [15] in der Cloud. Egal, in welcher Umgebung Kubernetes betrieben wird, die Auslieferung mit Skaffold muss für dieses Projekt möglich sein:

```
skaffold dev
```

Mit dem *dev*-Befehl von Skaffold wird ein kontinuierliches Auslieferungsszenario gestartet. Skaffold reagiert nun auf Quellcodeänderungen (egal ob in der IDE oder aus dem Terminal), indem das Erstellen des definierten Docker Images gestartet wird. Dazu wird das Gradle-JIB-Plug-in genutzt. Ist das Docker Image erstellt, startet die Kubernetes-Auslieferung. Sobald die Auslieferung zu Kubernetes stabilisiert ist, wird der eingehende Verkehr auf die neue Auslieferung geleitet und die Quellcodeän-

derungen sind aktiv. Eine stabile Auslieferung wird sehr oft mittels zweier Sonden (Probes) definiert (Listing 3):

- Liveness Probe
- Readiness Probe

Die Liveness Probe überprüft den Zustand des Docker-Containers. Schlägt das fehl, wird der Container neu gestartet. Die Readiness Probe überprüft, ob die Applikation im Container auf Anfragen reagiert. Ist das nicht der Fall, werden keine Anfragen an die Applikation gesendet.

Listing 3: Ausschnitt deployment.yaml

```
readinessProbe:
  tcpSocket:
    port: 8080
  initialDelaySeconds: 5
  periodSeconds: 10
  failureThreshold: 2
livenessProbe:
  tcpSocket:
    port: 8080
  initialDelaySeconds: 15
  periodSeconds: 20
  failureThreshold: 2
```

Listing 4: web.sh

```
ip=$(minikube ip)
port=$(kubectl get svc web -o=jsonpath='{.spec.ports[0].nodePort}')

while :
do
  echo "curl http://$ip:$port: "
  curl http://$ip:$port || true
  echo " "
  sleep 4s
done
```

Listing 5: Skaffold-Kommandozeile (Beispielausgabe)

```
Watching for changes...
Generating tags...
 - gcr.io/ktor-jib/kjib-image -> gcr.io/ktor-jib/kjib-image:40bddde-dirty
Checking cache...
 - gcr.io/ktor-jib/kjib-image: Found. Tagging
Tags used in deployment:
 - gcr.io/ktor-jib/kjib-image -> gcr.io/ktor-jib/kjib-image:59b464bbd59a0
                27842cd5e51a50d396c8141ec78b328aad6a0ef0130a8c314f0
Starting deploy...
 - deployment.apps/web configured
Waiting for deployments to stabilize...
 - deployment/web: waiting for rollout to finish: 1 old replicas are
                                                    pending termination...
 - deployment/web is ready.
Deployments stabilized
```

Damit die beiden Probes erfolgreich sein können, müssen Anfragen an das Kubernetes Deployment sowie die Applikation gestellt werden. Dafür habe ich ein kleines Shellskript geschrieben, das alle vier Sekunden eine HTTP-Abfrage an den bereitgestellten HTTP-Endpunkt stellt (Listing 4). Eine erfolgreiche Auslieferung erzeugt eine Ausgabe ähnlich zu Listing 5. Das gezeigte Beispiel zeigt, wie kontinuierliche Kubernetes-Auslieferung gelingen kann, ohne dass Kotlin-Entwickler:innen den gewohnten Kontext aus IDE und Kommandozeile verlassen müssen. Der Fokus liegt hier auf der Kommandozeile, da verschiedene IDE-Präferenzen unterstützt werden sollen.

Lothar Schulz ist ein Softwaremanager mit umfangreicher Erfahrung in vielen Bereichen. Er liebt es, zu coden.

Links & Literatur

[1] https://de.wikipedia.org/wiki/Integrierte_Entwicklungsumgebung

[2] https://opencontainers.org

[3] https://www.docker.com

[4] https://www.docker.com/blog/speed-up-your-development-flow-with-these-dockerfile-best-practices/

[5] https://gradle.org

[6] https://docs.gradle.org/current/userguide/kotlin_dsl.html

[7] https://github.com/GoogleContainerTools/jib

[8] https://github.com/GoogleContainerTools/jib/tree/master/jib-gradle-plugin; ein Maven-Plug-in existiert ebenfalls: https://github.com/GoogleContainerTools/jib/tree/master/jib-maven-plugin

[9] https://docs.docker.com/glossary/#base-image

[10] https://forums.docker.com/t/docker-latest-tag-to-check-if-there-is-any-changes-on-the-upstream-hub/46012

[11] https://skaffold.dev

[12] https://github.com/lotharschulz/ktorjib

[13] https://codeberg.org/lotharschulz/ktorjib

[14] https://minikube.sigs.k8s.io/docs/

[15] https://aws.amazon.com/eks/

Dev(Ops) Experience Cloud-Native

Umwege zum Glück

Der wachsende Marktanteil von Cloudsystemen zeigt ganz klar, dass mehr und mehr Softwaresysteme in der Cloud betrieben werden. Als Voraussetzung hierfür erfolgen immer mehr Entwicklungen Cloud-Native. Der Begriff Dev(Ops) Experience Cloud-Native fokussiert auf „Entwicklung für die Cloud" und „Deployment in der Cloud". Manchmal ist bei der Unterscheidung nicht ganz klar, ob es eher Dev oder eher Ops ist, weswegen es sich eher um eine Dev(Ops) Experience handelt.

von Michael Hofmann

Für ein Entwicklerteam, das für die Cloud entwickelt, ergeben sich eine ganze Reihe neuer Herausforderungen, angefangen bei einem adaptierten Entwicklungsprozess bis hin zu neuen Methodiken und Frameworks. Aber auch eine Verschiebung der Aufgabenverteilung steht zur Debatte. In vielen Projekten ist man noch in der Findungsphase, welche neue Aufgaben von den Entwicklern übernommen werden. Oder ist es besser, wenn diese zusätzlichen Aufgaben von den Operations-Kollegen übernommen werden? Zu guter Letzt muss man sich auch noch über neue Tools Gedanken machen, um die neuen Arbeitsschritte im Entwicklungsprozess zu unterstützen.

Das Thema ist sehr umfangreich und bei einigen Teilgebieten befindet sich die DevOps-Community noch am Anfang oder mitten im Umbruch. Von daher ist es schwer, alle notwendigen Aspekte zu beleuchten. Mit diesem Schwerpunkt wollen wir versuchen, ein paar Themen herauszugreifen und näher zu betrachten. Manches davon wird tiefergehend beschrieben, andere Dinge können nur oberflächlich erwähnt werden. In der Vorabstimmung mit den Autorenkollegen hat sich ganz klar gezeigt, wie vielfältig das Themengebiet ist. Wir hoffen trotzdem, dass der ein oder andere Hinweis in den Artikeln hilft, die eigene Dev(Ops) Experience für die Cloud-Native-Entwicklung zu verbessern.

Entwicklungsprozess

Um Anwendungen für die Cloud zu entwickeln, kommen, neben neuen Architekturansätzen, auch neue Anforderungen aus der Infrastruktur auf die Entwickler zu. Um dem gerecht zu werden, sind neue Teilschritte im Entwicklungsprozess notwendig. Die Anwendung muss in einem Docker-Container lauffähig sein, der selbst wiederum in einer Orchestrierungsumgebung wie Kubernetes betrieben wird. Somit erweitert sich der herkömmliche Entwicklungsprozess um die Teilschritte „Docker Build and Push" und „Deploy to (local) K8S". Damit sieht der neue Entwicklungsprozess für eine Cloud-Native-Anwendung aus wie in **Abbildung 1** gezeigt.

Dieser erweiterte Entwicklungsprozess führt unweigerlich dazu, dass neue Tools eingesetzt werden müssen. Sei es, weil die alten Werkzeuge nicht mehr passen, oder weil es neue Teilschritte im Prozess gibt, die vorher nicht notwendig waren. Das Ziel sollte immer sein, eine möglichst hohe Automatisierung zu erreichen, um weiterhin effizient entwickeln zu können. Auf das Thema der Toolauswahl kommen wir später nochmal zurück. Auch das Thema „Deploy to (local) K8S" wird in einem der folgenden Abschnitte genauer beleuchtet.

DevOps-Aufgabenverteilung

Neben dem neuen Entwicklungsprozess ergeben sich auch neue Aufgaben. Früher hat man beispielsweise eine

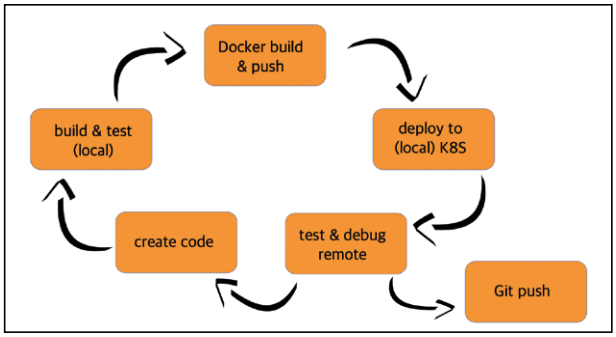

Abb. 1: Cloud-Native-Entwicklungsprozess

Installationsanweisung in Prosa an die Ops-Kollegen übergeben. Auf Basis dieser Beschreibung haben sie dann versucht, ein Deployment und einen Betrieb der Anwendung zu bewerkstelligen. Als Folge dieses Vorgehens war das Ergebnis oft fehlerhaft, da die Prosabeschreibungen falsch verstanden wurden oder missverständlich formuliert waren.

Aus diesen Fehlern hat man glücklicherweise gelernt. Mit deklarativen Manifest-Dateien, wie sie beispielsweise bei Kubernetes zum Einsatz kommen, wird ganz klar festgelegt, was die Anwendung für einen erfolgreichen Betrieb von der Laufzeitumgebung benötigt. Missverständnisse sind hierbei ausgeschlossen.

Durch diesen neuen Ansatz ergibt sich zwangsläufig eine neue Aufgabe. Es stellt sich die Frage: Wer im Projekt/Unternehmen kümmert sich um die Erstellung dieser Manifest-Dateien? Auf der einen Seite ist es ganz klar die Aufgabe des Entwicklers. Er hat das Wissen, welche Ressourcen (Datenbankverbindungen, Umgebungsvariablen, CPU- und Speicherverbrauch …) seine Anwendung benötigt. Auf der anderen Seite beinhalten diese Manifest-Dateien auch viele Aspekte, die aus der Betriebsplattform (Autoscaling, Storage, Load Balancing, Networking ...) kommen. Hierfür ist Wissen notwendig, das früher nur bei den Ops-Kollegen zu finden war und heute oft bei den sogenannten Cloud-Plattform-Teams beheimatet ist. Das Wissen aus den beiden Welten Dev und Ops muss in diese Manifest-Dateien einfließen. Daher finden in den Projekten/Unternehmen intensive Diskussionen statt, wie diese Aufgabenverteilung am besten erfolgen kann.

Die Lösungsansätze reichen hierbei von „das Ops-Team stellt nur Kubernetes als Plattform bereit“, über „das Ops-Team entwickelt Basis-Manifest-Dateien zur Unterstützung des Dev-Teams“ bis hin zu „das DevOps-Team ist alleine zuständig und verantwortlich für die Anwendung“. Auf jeden Fall muss ein Schulterschluss mit den Ops-Kollegen/dem Plattform-Team stattfinden, um einen individuellen und praktikablen Ansatz zu finden. Was begrüßenswert ist, da es der DevOps-Philosophie entspricht.

Technisch betrachtet bietet beispielsweise Helm [1] – als sogenannter Package-Manager – die Möglichkeit, Base-Charts zu erstellen. Dabei werden Basis-Manifest-Dateien zur Verfügung gestellt, die dann in die eigenen Helm-Charts eingebunden und auch entsprechend parametrisiert werden können. Es entsteht hierbei eine mehrstufige hierarchische Chart-in-Chart-Struktur. Die Kunst dabei ist es, den richtigen Ansatz für die folgenden Fragen zu finden: Was alles soll im Base-Chart vordefiniert werden, was davon kann parametrisiert werden und was wird dem Verwender des Base-Charts selbst überlassen? Welche Base-Chart-Struktur ist für unser Projekt/Unternehmen passend?

Wird die Erstellung der jeweiligen Base-Charts als InnerSource-Projekt organisiert, erhält man in der Regel sehr schnell einen geeigneten Satz an Charts, die den Entwicklerteams das Leben stark vereinfachen.

Abhängigkeiten zum Cloud-Provider

Jeder der bekannten Cloud-Provider (AWS, Azure, Google, ...) hat unterschiedliche X-as-a-Service-Angebote im Programm. D. h., es werden verschiedene Cloud-Computing-Services zur Verfügung gestellt, um den Einstieg, aber auch den Umgang mit der Cloud zu erleichtern. Je mehr X-as-a-Service-Dienste in Anspruch genommen werden, desto weniger muss man sich selbst damit beschäftigen. Damit verschiebt sich die zuvor skizzierte Aufgabenverteilung weiter in Richtung Cloud-Provider und entlastet das Ops-Team von solchen Aufgaben.

Auf den ersten Blick scheinen diese Angebote verlockend, doch man muss sich darüber im Klaren sein, dass damit eine starke Abhängigkeit zum Cloud-Provider eingegangen wird. Die Serviceangebote gestalten sich sehr proprietär und ein Wechsel des Cloud-Providers ist folglich nicht mehr ganz so leicht.

Um dieses Problem in den Griff zu bekommen, werden immer öfter sogenannte Multi-Cloud-Strategien umgesetzt. Aus demselben Grund gibt es mittlerweile gesetzliche Vorgaben und Richtlinien, die einen Wechsel des Cloud-Providers jederzeit ermöglichen sollen.

Twelve-Factor App [2]

Die vor Jahren entstandene Methode der „Twelve-Factor App“ gibt hilfreiche Empfehlungen, was bei der Erstellung einer Software-as-a-Service-Anwendung beachtet werden sollte. Die folgenden Grundsätze bilden die Basis für die zwölf Faktoren:

- deklarative Formate für die automatisierte Installation
- eindeutiger Kontrakt mit dem Betriebssystem für maximale Portabilität
- Deployment in modernen Cloud-Plattformen
- minimaler Unterschied zwischen Entwicklungs- und Produktionsumgebung für Continuous Deployment
- Flexibilität bei Änderungen des Toolings, der Architektur oder beim Deployment

Aus diesen Grundsätzen wurden zwölf konkrete Faktoren mit ganz klaren Handlungsanweisungen abgeleitet. Befolgt man diese Anweisungen, erhält man eine Anwendung, die den Anforderungen der Cloud-Systeme entspricht. Die Aufzählung und Beschreibung dieser zwölf Faktoren würde den Rahmen dieses Artikels sprengen, von daher wird jedem Entwicklerteam empfohlen, die zwölf Faktoren nachzulesen und in den eigenen Projekten zu beachten.

Einen speziellen Faktor, nämlich Nummer zehn, „Dev/prod parity“ werden wir später im Artikel noch aufgreifen und näher beleuchten.

Entwickler-Frameworks und JVM

Neben der Operations-Seite hat sich auch auf der Entwicklerseite einiges im Cloud-Umfeld getan. Spring hat schon sehr früh begonnen, neue Frameworks für die Entwicklung in der Cloud zu erstellen. Bei den Java-EE- bzw. Jakarta-EE-Entwicklern hat es leider ein wenig länger ge-

dauert, bis auch für sie neue Frameworks entstanden sind. Erst mit der Schaffung von MicroProfile [3] konnten Jakarta-EE-Anwendungen für die Cloud entwickelt werden.

Leider ist es so, dass Spring- und auch MicroProfile-Anwendungen sehr lange Startzeiten aufweisen. Auf der anderen Seite ist ein großer Benefit der Cloud, dass Lastspitzen sehr elegant durch Autoscaling abgefangen werden können. Um diesen Nachteil bei JVM-basierten Systemen zu minimieren, sind neue Technologien entstanden. Oracle hat deswegen mit der Entwicklung der GraalVM [4] begonnen. Kernstück ist ein Compiler, der den Java-Code nativ übersetzt und somit einen schnell ausführbaren und kompakten Binärcode erzeugt. Den Kampf gegen die langsamen Startzeiten hat auch Quarkus [5] aufgenommen. Quarkus bezeichnet sich als Kubernetes-nativen Java-Stack, der maßgeschneidert ist für OpenJDK, HotSpot und GraalVM. Außerdem ist Quarkus ein MicroProfile-zertifizierter Applikationsserver, der die native Kompilierung durch die GraalVM unterstützt und somit hochperformanten und ressourcenschonenden Anwendungscode erzeugt. Mit diesem Themenblock beschäftigt sich ein weiterer Artikel in diesem Schwerpunkt.

Tools

Steht man am Anfang seiner Reise in die Cloud-Entwicklung oder will sich einen aktuellen Überblick verschaffen, welche Werkzeuge verfügbar sind, so bietet die CNCF Landscape [6] einen guten Ausgangspunkt. Besonders unter den Rubriken „Application Definition & Image Build“ und „Continuous Integration & Delivery“ findet man eine große Auswahl hilfreicher und notwendiger Tools, die den Tätigkeitsbereich Dev(Ops) abdecken. Entscheidet man sich für eins der hier aufgeführten Tools, hat man gute Chancen, dass dieses Tool auch morgen noch aktuell und verfügbar ist. Schließlich will man eine etablierte und funktionierende Toolchain nicht immer wieder verändern müssen.

Bei der Auswahl der Tools sollte man sich neben der sogenannten Experience auch genauer ansehen, wie es mit der Integrierbarkeit der Tools in den vorhandenen Entwicklungsprozess aussieht. Werden standardisierte Übergabeschnittstellen in Richtung vorgelagertes bzw. nachgelagertes Tool unterstützt oder ist die Integration in den Gesamtprozess eher hakelig? Denn bei aller Risikoabsicherung durch die CNCF Landscape kann es immer wieder zu Situationen kommen, in denen ein Tool durch ein anderes ersetzt werden muss. Sei es, weil das Tool sein End of Life erreicht hat oder weil es mittlerweile ein besseres Werkzeug gibt, das an die Stelle des alten Tools treten soll.

In den letzten Jahren hat sich in der Unternehmenspolitik bezüglich der Toolauswahl einiges verändert. Früher gab es immer wieder mal das „Diktat von oben“, welche Tools man als Entwickler verwenden darf. Diese Vorgaben werden Gott sei Dank immer weniger. Schließlich sollte doch der Entwickler entscheiden, mit welchen Tools er seine tägliche Arbeit erledigen will. Schlussendlich setzt sich hier ein sogenannter Tool-Darwinismus durch. Es spricht sich unter den Entwicklern schnell herum, welches Tool wirklich gute Dienste leistet. Am Ende bleibt dann oft ein einziges Tool übrig, das sich als Standard innerhalb des Projekts oder der Firma etabliert. Die Unternehmensvorgaben sind damit aber noch nicht ganz vom Tisch. Schließlich sind sie definitiv sinnvoll, wenn es sich um Compliance-Vorgaben handelt oder man gewisse Standards in der Qualität der Software einhalten will (Stichwort SonarQube [7]).

Wie schon in der Vergangenheit, gibt es auch heute noch Toolhersteller, die kommerzielle Tools für diesen Zweck anbieten. Die Basis hierfür sind meist Open-Source-Projekte. Darauf aufbauend werden weiterführende spezielle Funktionen kommerziell angeboten. In der Regel sind die Basisfunktionen aus den Open-Source-Projekten ausreichend. Von daher muss sich jedes Team selbst ein Bild davon machen, ob sich diese kommerziellen Angebote lohnen oder ob man erstmal mit der Open-Source-Basis starten will.

Tools im Entwicklungsprozess

Aus Sicht der Dev(Ops) Experience enthalten die Rubriken „Application Definition & Image Build“ und „Continuous Integration & Delivery“ die wichtigsten Tools. Hier werden die Tools gruppiert, mit denen ein Entwickler am unmittelbarsten zu tun hat. Doch bevor wir uns mit den zugehörigen Tools beschäftigen, sollten wir nochmal einen Blick auf den Entwicklungsprozess werfen.

Fasst man den neuen Entwicklungsprozess zusammen, dreht es sich im Grunde um folgende Herausforderung: „Wie bekomme ich meinen Code möglichst schnell in die Cloud?“. Der schnelle Roundtrip ist jetzt noch wichtiger als früher, da die zusätzlichen Schritte die Roundtrip-Zeit erhöhen. Fehlerhafte Zwischenergebnisse, die wegen des umfangreicheren Prozesses öfter auftreten können, führen ebenfalls dazu, dass der Roundtrip sich verlängert.

Die hierfür notwendigen Artefakte (Docker Image, Manifest-Dateien) werden in der Regel vom Entwickler erstellt. Deswegen ist es naheliegend, dass es gut wäre, wenn diese Artefakte auf dem lokalen Entwicklungsrechner erstellt und getestet werden können. Der Entwickler muss prüfen können, ob seine Helm Charts die richtigen Manifest-Dateien erzeugen oder ob das Docker Image den notwendigen Voraussetzungen entspricht. Finden diese Schritte nur auf dem CI-Server statt, so ist die Durchlaufzeit erhöht und die Effizienz nicht besonders hoch. Der Königsweg sieht so aus, dass auf dem Entwicklungsrechner und auf dem CI-Server dieselben Werkzeuge verwendet werden. Die klassischen Überraschungen, dass der Build lokal funktioniert, aber die Ausführung auf dem CI-Server Fehler wirft, werden somit auf ein Minimum reduziert.

Ein neuer notwendiger Schritt im Entwicklungsprozess ist die Erstellung des Docker Image. Am einfachsten gelingt sie, indem man in Gradle oder Maven ein entsprechendes Plug-in verwendet. Doch hier ist Vorsicht geboten: In den letzten Jahren hat hier ein großes Plug-in-Sterben stattgefunden. Von anfangs mehr als zehn verfügbaren Plug-ins sind nur noch wenige übrig. Die bekanntesten sind docker-maven-plugin [8] und jib [9].

Bei Gradle sieht es ähnlich traurig aus: Docker Gradle Plugin von Palantir [10], Benjamin Muschko [11] oder jib [12] sind die letzten verfügbaren Alternativen.

Auf der anderen Seite gibt es einige Tools, die als sogenannte Stand-alone-Tools die Erstellung des Docker Image übernehmen können. Hierzu zählen neben Docker selbst [13] Kaniko [14] und Buildah [15], um nur einige Vertreter zu nennen. Dabei stellt sich aber sofort die Frage, wie man diese auf dem Entwicklerrechner im Anschluss an den Build der Anwendung automatisiert ausführen kann. Auf dem CI-Server ist das einfacher umzusetzen.

CI/CD-Server

Angepasst an die neuen Möglichkeiten bzw. Anforderungen der Cloud, sind in den letzten Jahren neue CI/CD-Server entstanden. Argo, Flux, Keptn, JenkinsX und Tekton, um nur ein paar Vertreter aus dem Fundus der CNCF Landscape zu nennen. Mit einem speziellen Vertreter aus dieser Kategorie (Argo) beschäftigt sich ein weiterer Artikel innerhalb dieses Schwerpunkts.

Test und Debugging

Der Test und das Debugging der Anwendung können zu großen Teilen lokal ausgeführt werden. Es gibt aber durchaus die Notwendigkeit, die Anwendung in der Zielumgebung, also im Kubernetes-Cluster, zu testen und evtl. dort auch gleich zu debuggen. Zwischen lokaler Entwicklungsumgebung und Cluster gibt es einige Unterschiede, die zu unterschiedlichem Verhalten der Anwendung führen können, wie zum Beispiel:

- Betriebssystem
- JDK-Version
- Umgebungsvariablen, Volume Mounts, DNS …
- Backing Service mit Clusterkonfiguration (lokal nur Single-Server-Konfiguration)

Sobald sich die Anwendung im Cluster anders verhält als beim lokalen Test, wird es aufwendig, die Ursache für dieses Verhalten zu analysieren. Glücklicherweise gibt es mehrere Ansätze, die hierbei helfen können. Diese sind:

- synchrones Deployment
- Source Reload
- Swap Deployment

Im Folgenden werden diese drei Ansätze genauer beschrieben.

Synchrones Deployment

Der erste Ansatz ist das synchrone Deployment. Hierbei wird beispielsweise mit Skaffold [16] der Code lokal und im Cluster synchronisiert. Sobald sich der Code lokal ändert, beginnt Skaffold im Hintergrund mit den notwendigen Build-Schritten und kümmert sich auch gleich noch um das Deployment im Cluster. Langlaufende Tests während des Builds kann man hierbei vorübergehend deaktivieren. Dadurch hat man nach wenigen Sekunden den neuen Codestand im Cluster deployt und kann sofort mit einem Test prüfen, ob damit der Fehler behoben wurde. Gerade in der Anfangsphase, bei der man sich als Entwickler um das Docker Image oder um Kubernetes Manifest-Dateien kümmern muss, ist dieses Tool sehr hilfreich.

Source Reload

Eine andere Möglichkeit für den schnellen Durchlauf von Compile zu Deployment bietet das Source Reload. Hierbei werden neue *Class*-Files, die lokal kompiliert wurden, in den laufenden Kubernetes Pod hochgeladen. Der Application Server im Pod tauscht diese *Class*-Files im laufenden Betrieb ohne Neustart des Servers aus. Beim darauffolgenden Testaufruf wird der neue Code ausgeführt und behebt hoffentlich den Bug.

Hierfür sind ein paar Voraussetzungen notwendig. Zum einen muss der Application Server diese Funktionalität anbieten, was bei den gängigsten Servern (Tomcat, OpenLiberty, ...) durchaus der Fall ist. Ein kurzer Blick in die Doku sollte das bestätigen und die notwendigen Einstellungen erklären. Zum anderen benötigt man noch ein Tool, das den Filetransfer der *Class*-Files ermöglicht. Hier kommt Ksync [17] ins Spiel. Ksync synchronisiert das lokale Filesystem, in dem die *Class*-Files nach dem Compile abgelegt werden, mit dem Filesystem im laufenden Kubernetes Pod. Sobald Ksync die neuen Dateien im Pod abgelegt hat, werden diese vom Application Server geladen. Der ganze Vorgang geht sehr schnell vonstatten, da es sich in der Regel nur um ein paar kleine Dateien handelt, die ohne großen Zeitaufwand in den Cluster kopiert werden können. Das Ganze würde auch mit dem WAR-File funktionieren, ist aber wegen des zusätzlichen

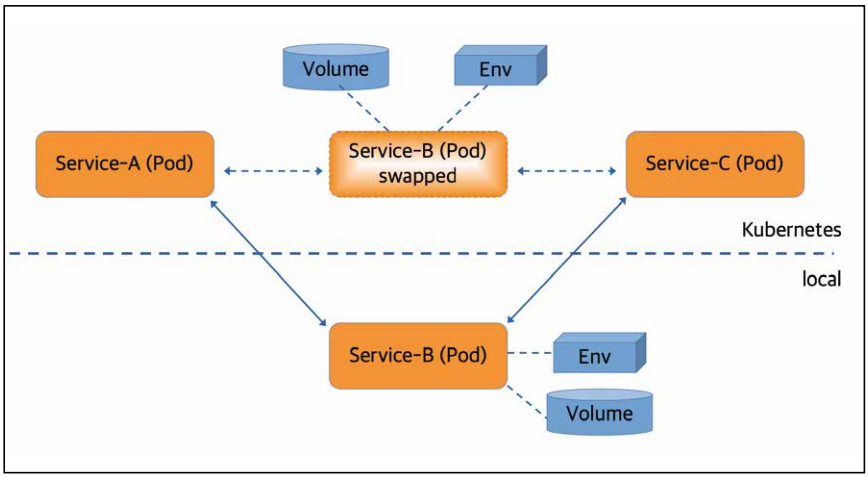

Abb. 2: Swap Deployment

Build-Schritts und der Größe des WAR-Files ein wenig langsamer.

Quarkus verfügt über ein sogenanntes Live Coding, das denselben Ansatz verfolgt. Ein passend konfigurierter App-Server tauscht empfangene *Class*-Files ohne Restart on the fly aus. Der notwendige lokale Sync-Prozess wird von Quarkus auch gleich mitgeliefert. Damit ermöglicht Quarkus einen Source Reload ganz ohne die Notwendigkeit weiterer Tools.

Ein Code-Replace ohne Neustart hat den Vorteil, dass eine existierende Remote-Debug-Verbindung nicht abbricht. Damit ergibt sich eine Experience für den Entwickler, die fast identisch ist mit einer lokalen Entwicklung, sogar beim Debuggen.

Swap Deployment

Der dritte Ansatz geht noch einen Schritt weiter. Hierbei wird der Kubernetes Pod, in dem die Anwendung läuft, durch einen bidirektionalen Proxy ersetzt. Dieser Proxy wird dabei so konfiguriert, dass er die Requests an die lokal laufende Instanz der Anwendung weiterleitet. Das lokale Netzwerk auf dem Entwicklungsrechner wird ebenfalls verändert. Somit können die Responses wieder zurück in den Kubernetes-Cluster geleitet werden. Diese werden vom Proxy empfangen und entsprechend weitergeleitet. Das Diagramm in **Abbildung 2** soll diese Situation veranschaulichen. Als zusätzliches Feature – neben der Manipulation der lokalen Netzwerkeinstellungen – werden die Volumes und die Umgebungsvariablen aus dem Kubernetes Pod auch für die lokale Umgebung zur Verfügung gestellt. Geeignete Tools für das Swap Deployment sind telepresence [18] und das Visual-Studio-Code-Plug-in Bridge to Kubernetes [19].

Durch die lokal laufende Anwendung wird bei diesem Ansatz keine hundertprozentige Parity (siehe Faktor 10 der Twelve-Factor App) erreicht. Naturgemäß gibt es Unterschiede im Betriebssystem und im JDK. Die restliche Parity wird allerdings eingehalten. Im Vergleich hierzu erreichen die beiden anderen Ansätze volle 100 Prozent Parity, da die Tests im Kubernetes-Cluster selbst ausgeführt werden.

Eine Konsequenz dieses Ansatzes muss noch erwähnt werden. Das Ersetzen des laufenden Pods mit dem Proxy ist für alle Entwickler, die auf den Cluster Zugriff haben, transparent. D. h., sobald ein Entwicklerkollege einen Request gegen diesen (ausgetauschten) Pod abschickt, landet dieser auch in der eigenen lokalen Entwicklungsumgebung. Mittlerweile können beide Tools dieses Verhalten kompensieren, indem sie den Proxy parallel zum Original-Pod betreiben. Auf diese Weise werden die Requests der Entwicklerkollegen nicht mehr zur lokalen Instanz geleitet. Leider ist bei telepresence hierfür eine kommerzielle Version des Tools notwendig. Bridge to Kubernetes bietet das im Open-Source-Tool mit an.

Lokaler Kubernetes-Cluster

Normalerweise orientieren sich die Kubernetes-Cluster in der Cloud, in den vorgelagerten Stages, an den Vorgaben, die man für die produktive Stage benötigt. D. h., die Cloud-Cluster für die unterschiedlichen Stages sind mit Zugriffsbeschränkungen ausgestattet (RBAC, Security Constraints) und besitzen einen Network-Provider, der die Netzwerkzugriffe innerhalb des Clusters regelt. Darüber hinaus existieren auch noch weitere Einschränkungen, die aus der produktiven Stage übertragen wurden.

„Nur mal schnell was ausprobieren“ ist auf solchen Systemen nicht ganz einfach. Dafür eignen sich lokale Kubernetes-Cluster, die auf dem Entwicklerrechner gestartet werden, sehr viel besser. Auf diesem lokalen Kubernetes hat der Entwickler volle Rechte und kann sehr einfach ein paar Dinge ausprobieren. Einfach mal hinter die Kulissen schauen, um zu verstehen, wie der Kubernetes-Cluster eigentlich arbeitet, ist damit viel leichter. Hinzu kommt, dass der Entwickler-Roundtrip schneller vonstattengehen kann, wenn der CI/CD-Server für das Deployment ausgelassen werden kann. Vorausgesetzt, man hat sich für den CI/CD-Teil an die Empfehlung von weiter oben im Artikel gehalten (möglichst identische Tools für CI/CD). Das soll nicht heißen, dass nur noch im lokalen Cluster entwickelt werden soll, denn schließlich muss die Anwendung auch irgendwann auf der produktiven Stage in der Cloud lauffähig sein.

Lokale Kubernetes-Cluster sind mit Minikube [20], DockerDesktop [21] oder Kind [22] möglich, um nur eine kleine Auswahl zu nennen. Mittlerweile gibt es noch eine ganze Reihe mehr solcher Cluster, manche sogar mit einer Spezialisierung für gewisse Zwecke. Eine eigene Internetrecherche ist hier sicherlich lohnend.

Branch-Deployment

Die Verbindung von GitOps mit der Cloud und den neuen CI/CD-Servern ermöglicht jetzt ein sogenanntes Bran-

ch Deployment. Build Pipelines können dafür genutzt werden, einen persönlichen Git-Branch in der Cloud zu deployen, ohne dabei mit dem eigentlichen Deployment aus dem Main Branch zu kollidieren. Auf diese Weise kann ein Entwickler seine Änderungen in der Cloud testen, ohne den anderen Kollegen dabei in die Quere zu kommen. Der Trick dabei ist die dynamische Erstellung eines privaten Kubernetes Namespace, der nur für den gewählten Git-Branch genutzt wird. Hier können individuelle Tests erfolgen, und wenn diese abgeschlossen sind, sollten die temporären Artefakte automatisch wieder aufgeräumt werden. Beim Merge des privaten Branch sollte die Pipeline auch den zugehörigen privaten Kubernetes Namespace wieder entfernen.

Fazit

Dieser kleine Rundumblick durch die Dev(Ops) Experience kann im Grunde nur zeigen, wie umfangreich das Thema geworden ist. Im Vergleich zu früheren Entwicklungen ist die Komplexität massiv gestiegen und damit auch die Anforderungen an die Entwicklerteams. Eine Vielzahl neuer Teilaufgaben muss von diesen oder neu zu schaffenden (DevOps-)Teams gestemmt werden. Der Wechsel von herkömmlicher Entwicklung in Richtung Cloud beschert den Entwicklerteams in der Regel eine sehr steile Lernkurve.

Der Wechsel in die Cloud-Entwicklung kann ohne passende Werkzeuge nicht gelingen. Von daher ist es auch kein Wunder, dass es in diesem Bereich in den letzten Jahren eine wahre Explosion an Tools gegeben hat. Aus heutiger Sicht ist dieser Veränderungsprozess auch noch lange nicht abgeschlossen.

In vielen Bereichen bleibt durch den notwendigen Einsatz neuer Tools kein Stein auf dem anderen. Oftmals entsteht eine völlig neue Toolchain, die aufeinander abgestimmt funktionieren muss und sogar einen Austausch von einzelnen Komponenten verkraften soll.

Das alles zusammengenommen ergibt für uns Dev(Ops)-Mitarbeiter eine ganze neue Dev(Ops) Experience mit neuen Möglichkeiten, aber auch neuen Herausforderungen. Im Guten wie im Schlechten. Auf jeden Fall bleibt es spannend.

Michael Hofmann ist freiberuflich als Berater, Coach, Referent und Autor tätig. Seine langjährigen Projekterfahrungen in den Bereichen Softwarearchitektur, Java Enterprise und DevOps hat er im deutschen und internationalen Umfeld gesammelt.

Links & Literatur

[1] https://helm.sh
[2] https://12factor.net
[3] https://microprofile.io
[4] https://www.graalvm.org
[5] https://quarkus.io
[6] https://landscape.cncf.io
[7] https://www.sonarqube.org
[8] https://github.com/fabric8io/docker-maven-plugin
[9] https://github.com/GoogleContainerTools/jib/tree/master/jib-maven-plugin
[10] https://github.com/palantir/gradle-docker
[11] https://github.com/bmuschko/gradle-docker-plugin
[12] https://github.com/GoogleContainerTools/jib/tree/master/jib-gradle-plugin
[13] https://docs.docker.com/develop/develop-images/build_enhancements/
[14] https://github.com/GoogleContainerTools/kaniko
[15] https://buildah.io
[16] https://skaffold.dev
[17] https://github.com/ksync/ksync
[18] https://www.telepresence.io
[19] https://docs.microsoft.com/en-us/visualstudio/bridge/overview-bridge-to-kubernetes
[20] https://minikube.sigs.k8s.io/docs/start/
[21] https://www.docker.com/products/docker-desktop/
[22] https://kind.sigs.k8s.io

Kubernetes Continuous Delivery mit ArgoCD

Komfort und gute Diagnosemöglichkeiten

Der Einsatz von Kubernetes verspricht mehr Komfort und schnelleres Feedback für Entwickler. Dabei sind viele Nutzer bestrebt, bisher bekannte Verfahren und Werkzeuge auf Kubernetes eins zu eins zu übertragen. Vor allen Dingen, wenn bereits ein flexibler Build-Server wie Jenkins eingesetzt wird, kommt oft gar nicht erst die Idee auf, ob es noch bessere Verfahren geben könnte. Dieser Artikel stellt die Motivation und Hintergründe für ergänzende Werkzeuge vor und zeigt, wie diese effektiv eingesetzt werden können.

von Thomas Kruse

Wer kennt das nicht: Das Projekt ist zu 80 Prozent rum, es müssen nur noch die meisten Features implementiert werden. Dazu gesellen sich dann Last-Minute Changes der Stakeholder, die sich auch mit Verweis auf agiles Vorgehen und festen Sprint Scope nicht aufschieben lassen. Neben diszipliniertem Vorgehen gibt es hier zwei Dinge, die unverzichtbar sind: Automatisierung und schnelles Feedback.

Ersteres ist zum Glück regelmäßig in Form von CI-Server und Testautomatisierung vorzufinden. Es scheint sich herumgesprochen zu haben, dass das dem Stand der Technik entspricht und für langlebige Qualitätssoftware genauso unverzichtbar ist wie das tägliche Zähneputzen für ein gesundes Gebiss. Dabei greifen diese Punkte auch ineinander: Durch automatisierte Tests, z. B. mit JUnit, wird schnelles Feedback ermöglicht und das Risiko reduziert, dass durch menschliche Fehler bei manuellen Tests Testfälle ausgelassen oder falsch ausgeführt werden. Mitunter vergisst ein Entwickler komplett, die Tests auszuführen. Hier springt der CI-Server ein: Er kann bei Änderungen in der Quellcodeverwaltung reagieren und führt die automatischen Tests selbstverständlich aus. Damit sorgt er wiederum für schnelles Feedback für die Entwickler.

Die meisten Projekte setzen inzwischen auf ein Verfahren, bei dem neue Features zunächst auf einem Branch entwickelt werden und die Änderungen dann – möglichst bald – auf einem Integrations-Branch wie *master* oder *main* zusammengeführt werden. Moderne CI-Server sind in der Lage, die Branches zu erkennen und auch dort automatisch Builds inkl. Tests auszuführen. Damit ist schnell klar, ob das Feature wirklich done ist und alle Tests grün sind, bevor die Änderungen zusammengeführt werden.

Dem Stakeholder imponieren grüne Tests und Code-Coverage in der Regel eher weniger: Er möchte seine Features ausprobieren können. Nur so kann er sich schnell wieder neue Changes für die Entwickler ausdenken oder auf neue Featureideen kommen.

Es liegt also die Idee nahe, auch hier auf Automatisierung zu setzen. Oft wurden dazu früher dedizierte Instanzen der Anwendung genutzt, die in dedizierten Umgebungen inkl. Umsystemen bereitgestellt wurden. Das konnten etwa so etwas wie „Prod“, „Pre-Prod“, „Dev“ oder „D“, „T“, „P“ Stages sein. Solche Umgebungen wurden mit mehr oder weniger viel Aufwand vorbereitet: Virtuelle Maschinen, DNS-Einträge, Application-Server, Datenbanken und Ähnliches gehörten dazu. Das Deployment des gebauten Artefakts der Anwendung wurde oft durch den Build-Server mit übernommen. Je nach Anwendungstyp wurden diverse

Verfahren eingesetzt: SSH/SCP aus dem Build heraus, mit einem technischen User auf den virtuellen Maschinen – oder auch spezielle Verfahren wie Tomcat Cargo, womit ein neues Artefakt auf dem Tomcat-Application-Server ausgerollt werden kann.

Es ist schnell einzusehen, dass dieses Verfahren nicht gut skaliert. So wird das Deployment von Feature-Branches schwer umsetzbar. Damit kann das Potenzial von schnellem Feedback nicht mehr voll ausgeschöpft werden: Erst nach der Integration auf einem deploy-fähigen Branch kann von der Fachabteilung Feedback eingeholt werden. Muss danach noch einmal an dem Feature gearbeitet werden und erfolgt das erneut auf einem Branch, wiederholt sich die Situation.

Es wäre hier wünschenswert, wenn jeder Branch nach jedem *git push* wieder in Augenschein genommen werden könnte. Meiner Meinung nach liefert Kubernetes als API-zentrische Plattform hier ein ebenso großes Potenzial wie es früher die Einführung von JUnit und CI-Servern brachte. Potenzial allein reicht jedoch nicht: Kubernetes kann man als Betriebssystem eines Rechenzentrums oder auch der Cloud auffassen. Doch mit Continuous Integration oder Continuous Delivery hat Kubernetes erst einmal nichts zu tun, dazu bedarf es weiterer Werkzeuge.

Schauen wir uns zunächst an, was wir für das Deployment einer Anwendung in Kubernetes so benötigen.

Bereitstellung einer Anwendung in Kubernetes

Kubernetes betreibt containerbasierte Anwendungen. Das bedeutet, dass wir unsere Anwendung zu einem Container-Image, zum Beispiel mit Docker, Kaniko oder Jib verpacken müssen. Weiterhin wird eine Registry benötigt, aus der die Kubernetes Nodes dann das Anwendungs-Image beziehen können. Auch dieser Aspekt ist nicht in Kubernetes selbst gelöst und es gibt diverse Produkte, mit denen das umgesetzt werden kann. Die simpelste Form von Workload in Kubernetes ist der sogenannte Pod. Im einfachsten Fall ist ein Pod genau ein Container. Pro Pod gibt es eine eigene IP-Adresse aus dem Cluster-IP-Bereich, sodass auf der Netzwerkebene Portkonflikte vermieden werden. Ein Pod bietet jedoch keine höherwertigen Eigenschaften, wie zum Beispiel einen automatischen Ersatz, wenn die Maschine, auf der der Pod eigentlich laufen soll, ausfällt. Daher wird in der Regel ein ReplicaSet oder Deployment genutzt. Ein ReplicaSet stellt sicher, dass stets eine gewisse Anzahl von Pods vorhanden ist. Ein Deployment kann mehrere ReplicaSets verwalten und damit einen unterbrechungsfreien Rollout einer neuen Anwendungsversion realisieren.

Damit innerhalb des Clusters der Zugriff auf die Pods funktioniert, gibt es das *Service*-Objekt. Es stellt innerhalb des Clusters einen DNS-Namen, eine feste virtuelle Service-IP und auch einfache Load-Balancing-Funktionalität bereit. Für den Zugriff von außen auf die Anwendung im Cluster wird typischerweise der Service nach außen bereitgestellt. Das kann über den Typ *LoadBalancer* mit einer dedizierten IP erfolgen oder bei HTTP-Anwendungen auch sehr gut mit einem Ingress. Ein Ingress untersucht eingehende HTTP-Anfragen anhand von Regeln und ruft dann z. B. in Abhängigkeit vom angefragten Hostnamen einen zugehörigen Service auf.

All diese Objekte werden in Kubernetes durch YAML- oder JSON-Konfigurationen definiert. Egal mit welchem Werkzeug, VS Code, kubectl oder dem grafischen Kubernetes-Dashboard, stets finden unter der Haube Zugriffe auf den Kubernetes-API-Server statt, bei dem diese YAML-Dokumente ausgetauscht werden. Klingt nach optimalen Voraussetzungen für eine Automatisierung.

Zentrale Rolle: Der CI-Server

Zunächst muss die jeweilige Anwendung als Container-Image bereitgestellt werden. Dabei gibt es einige Dinge zu beachten: Docker hat für Container-Images das Schema eingeführt, dass jedes Image ein Repository als Namen hat. Das Repository beinhaltet typischerweise auch den Hostnamen der zu verwendenden Registry. Zumindest bei Docker wird ohne expliziten Hostnamen stets Docker Hub als Registry angesprochen. Dazu kommt noch ein optionales Tag, mit dem typischerweise die Version gekennzeichnet wird. Das Tag hat an sich keine Semantik und kann frei gewählt werden. Wird es ausgelassen, so wird implizit *latest* verwendet. Hier wartet bereits ein Stolperstein: Wird mit demselben Tag ein aktualisiertes oder geändertes Image bereitgestellt, kann es zu Inkonsistenzen kommen: Hat ein Node ein Image einmal bezogen, wird es in der Regel nicht erneut geladen oder auf Aktualisierungen geprüft. Wird auf einem anderen Node, aus welchen Gründen auch immer, zu einem späteren Zeitpunkt das Image bezogen, handelt es sich nun um gemischte Versionen. Daher bietet sich

an, dass im Tag etwas Eindeutiges wie ein Zeitstempel oder der git-Commit-Hash, aus dem das Image gebaut wurde, verwendet wird.

Ein Image kann weiterhin Metadaten als sogenannte Label erhalten. Das sind schlichte Key-Value-Zeichenketten. Hier bietet es sich an, den Build-Zeitpunkt, das Quellcoderepository, den Branch und den Commit-Hash als Metadaten einzufügen. Das verbessert die Nachvollziehbarkeit, selbst wenn lediglich das Image zur Verfügung steht.

Es hat sich auch bewährt, in der Anwendung selbst Build-Zeitpunkt, Branch, Commit-Hash und ggf. git-Tag bereitzustellen, um diese für Tester und Nutzer bspw. zur besseren Kommunikation bei Fehlermeldungen anzuzeigen.

Der eigentliche Image Build ist recht unspektakulär, jedoch ist es wichtig, dass es eine gute Testabdeckung gibt und die Tests vor der Bereitstellung des Image zur Absicherung auch ausgeführt werden. Dabei kann im Build ruhig von einer möglichen Parallelisierung Gebrauch gemacht werden, doch der finale Push in die Registry sollte nur bei korrekten Anwendungsversionen erfolgen. Die prinzipiellen Kommandos werden nachfolgend exemplarisch gezeigt (Maven-basierter Build mit Testausführung und Container-Image-Build mit Jib).

```
./mvnw package verify --batch-mode --errors --fail-at-end --show-version
./mvnw jib:build
```

Automatisches Deployment in Kubernetes

Kubernetes bietet dank des API-zentrischen Ansatzes einfache und flexible Optionen zur Integration. Selbst wenn das gewählte CI-Produkt keine Kubernetes-Integration anbietet, kann im einfachsten Fall das *kubectl*-Kommando im Build integriert werden. Dabei hat der Build-Server Zugriff auf wichtige Metadaten und kann diese verwenden, um daraus dynamisch die Zielkonfiguration in Kubernetes zu ermitteln.

Als Grundlage der Konfiguration werden auch hier Kubernetes-Manifeste verwendet, die zusammen mit dem Quellcode im git abgelegt werden. Zum Beispiel könnte das in einem Ordner wie *manifests* oder *k8s* geschehen. Typischerweise ändert sich hauptsächlich die Image-Version einer Anwendung bei sukzessiven Builds. Der Build-Server muss daher nach einem initialen Deployment nur die Image-Version in einem Deployment anpassen. Kubernetes selbst kümmert sich dann um das Rollout der neuen Version. Dem Build-Server müssen dazu lediglich Credentials zur Verfügung stehen, um auf den Kubernetes Namespace die entsprechende Operation durchführen zu dürfen. Exemplarisch wird dieses Vorgehen nachfolgend am Beispiel von Jenkins gezeigt (Aktualisierung einer Anwendungsversion mit kubectl im Build):

```
stage('Deploy to cluster') {
  withKubeConfig([credentialsId: 'jenkins-secret', serverUrl: env.K8S_URL]) {
    sh "kubectl -n demo set image deployment sample app=
                                    $env.DOCKER_REGISTRY/sample:${dockerTag}"
}
```

Einige Punkte sind bei diesem Verfahren nicht optimal gelöst: Die tatsächlich verwendete Konfiguration befindet sich lediglich in Kubernetes. Das reduziert die Nachvollziehbarkeit. Ein Rollback zu einer vorherigen Version wird durch die eingesetzten Verfahren nicht unterstützt. Zudem erhält der Build-Server Rechte auf der Zielumgebung.

Listing 1: Exemplarische Aktualisierung des GitOps-git-Repositorys

```
git clone git@git.trion.de:gitops/k8s.git
cd k8s/base
kustomize edit set image registry.trion.de/
                                app-sample:latest=*:${COMMIT_SHA}
cd ..
kustomize build overlays/prod > app.yaml
git add .
git status #debugausgabe im build
git commit -m "Update for app ${COMMIT_SHA}"
git push
```

Listing 2: Kubernetes CronJob mit periodischem Abgleich

```
apiVersion: batch/v1
kind: CronJob
metadata:
  namespace: gitops-job
  name: gitops-job
spec:
  schedule: "*/5 * * * *"
  concurrencyPolicy: Forbid
  jobTemplate:
    spec:
      backoffLimit: 0
      template:
        spec:
          restartPolicy: Never
          serviceAccountName: gitops
          containers:
          - name: app
            image: alpine/k8s:1.22.6
            command:
            - sh
            - -e
            - -c
            args:
            - >
              git clone
                git@git.trion.de:gitops/k8s.git;
              kubectl apply -f app.yaml;
```

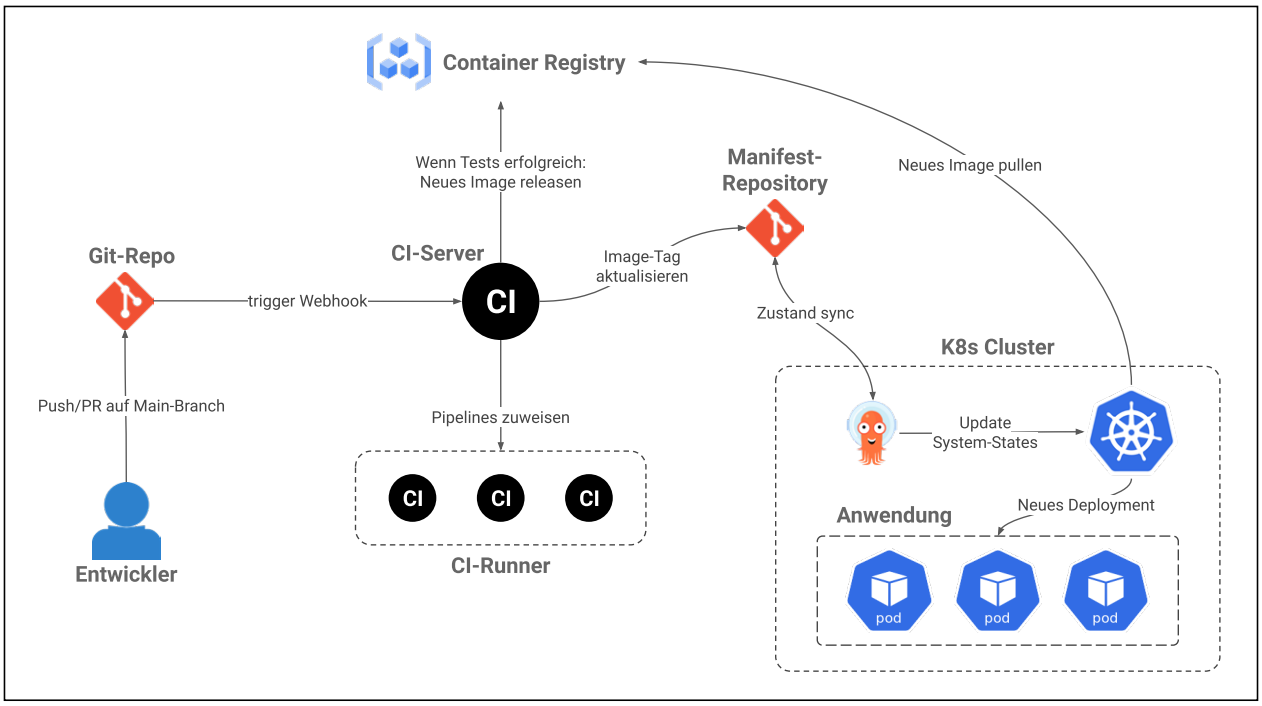

Abb. 1: Übersicht GitOps mit CI-Server und ArgoCD in Kubernetes

Eine Trennung nach Verantwortlichkeiten wäre hier wünschenswert: Der Build-Server soll lediglich getestete Artefakte als Container-Image bereitstellen, das Continuous Deployment wird in einem separaten Prozess verortet.

Trennung von CI und CD mit GitOps

GitOps beschreibt im Wesentlichen, dass git nicht nur für die Entwicklung von Artefakten verwendet, sondern aktiv in Betriebsprozessen (Ops) eingesetzt wird. Typischerweise findet sich dabei Infrastructure as Code, also die deklarative oder konfigurative Beschreibung von Betriebskomponenten. Die Umsetzung kann dabei mit diversen Werkzeugen erfolgen.

Dabei ist nicht definiert, ob dasselbe git-Repository, das auch für die Quellcodeverwaltung der Anwendung genutzt wird, für den GitOps-Prozess zu nutzen ist oder ein separates Repository verwendet werden muss. Auch der Umgang mit verschiedenen Umgebungen und das Staging/die Promotion einer Anwendung sind nicht festgelegt. Für eine bessere Übersichtlichkeit und Trennung der Verantwortlichkeiten empfiehlt es sich jedoch, zumindest zwischen dem Quellcoderepository für das Artefakt und dem für den GitOps-Prozess zu differenzieren. Zum Einsatz von Branches oder einer Ordnerstruktur für die verschiedenen Umgebungen lässt sich keine pauschale Empfehlung aussprechen. Beide Verfahren haben ihre jeweiligen Vor- und Nachteile.

Selbst ohne spezielle Tools lässt sich ein simples GitOps-Verfahren bauen. Das ist zum Verständnis der Abläufe auch ein guter Ansatz.

In Listing 1 ist ein simpler Ablauf zur Aktualisierung des GitOps-Repositorys aus dem Build-Job heraus zu sehen. Dabei wird zusätzlich *kustomize* verwendet, um die Manifeste für verschiedene Umgebungen rendern zu können.

Listing 2 zeigt nun, wie mit einem Kubernetes *CronJob* ein periodischer Abgleich aussehen könnte. Auch hier vorausgesetzt, dass die Rechte vorhanden sind. Wobei hier schon die Trennung auffällt: Der Build-Server erhält lediglich den Zugriff auf das GitOps-Repository, nicht auf den oder die Kubernetes-Cluster.

Damit haben wir einen rudimentären GitOps-Ablauf erstellt. Es fehlt jedoch für eine gute Developer Experi-

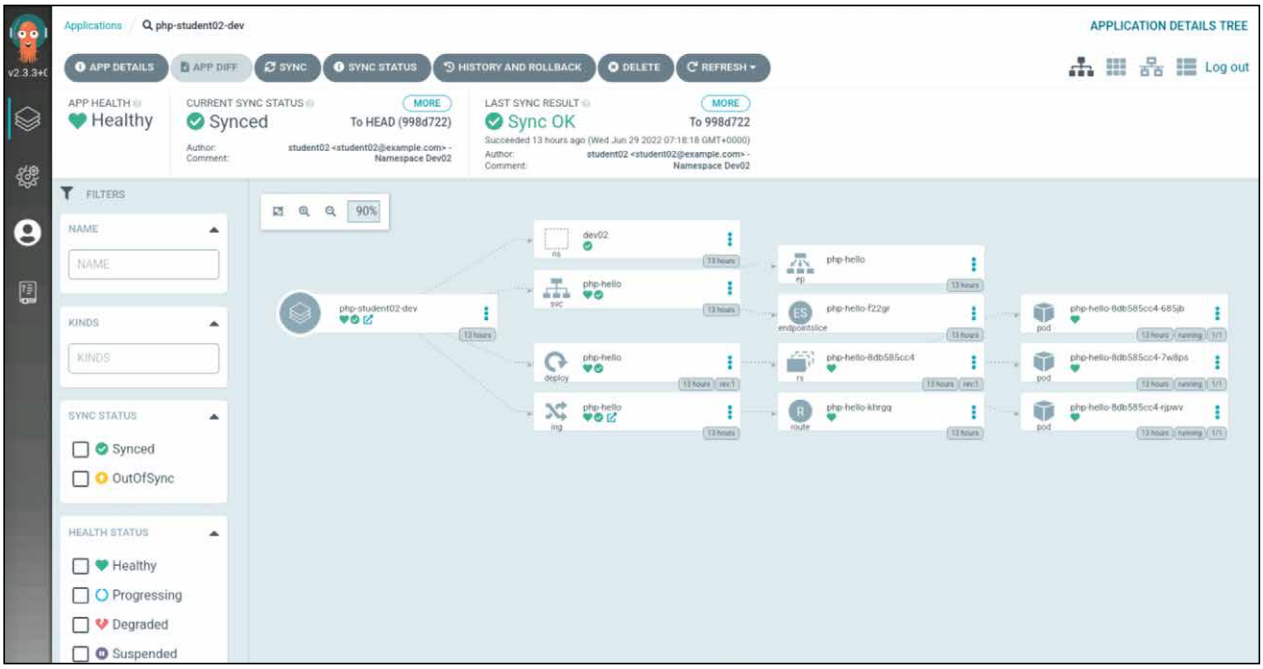

Abb. 2: Mehrere Anwendungen unter Verwaltung von ArgoCD

ence ein UI und es fehlen auch Funktionen: Wie kann ein Rollback aussehen? Wie können die Cronjobs bzw. wie kann eine Alternative verwaltet werden? Dazu betrachten wir im nächsten Abschnitt ArgoCD.

ArgoCD als Kubernetes-CD-Werkzeug

ArgoCD liefert die fehlenden Features und bringt sogar noch darüber hinausgehende Flexibilität mit. So gibt es ein Berechtigungssystem, mit dem eingeschränkt werden kann, welche Anwendungen auf welche Kubernetes Namespaces zugreifen und welche Objekte welche Operationen ausführen dürfen. In **Abbildung 1** ist der konzeptionelle Aufbau eines GitOps-Prozesses mit zwei getrennten git-Repositorys und ArgoCD als Werkzeug zu sehen..

ArgoCD besticht dabei durch eine wirklich gute Weboberfläche. Auch wenn mehrere Anwendungen verwaltet werden, wie in **Abbildung 2** zu sehen ist, ist doch stets eine gute Übersichtlichkeit gegeben. Dabei gibt es noch weitere Ansichtsvarianten, die vor allem bei sehr vielen Anwendungen in kompakter Form einen schnellen Überblick ermöglichen.

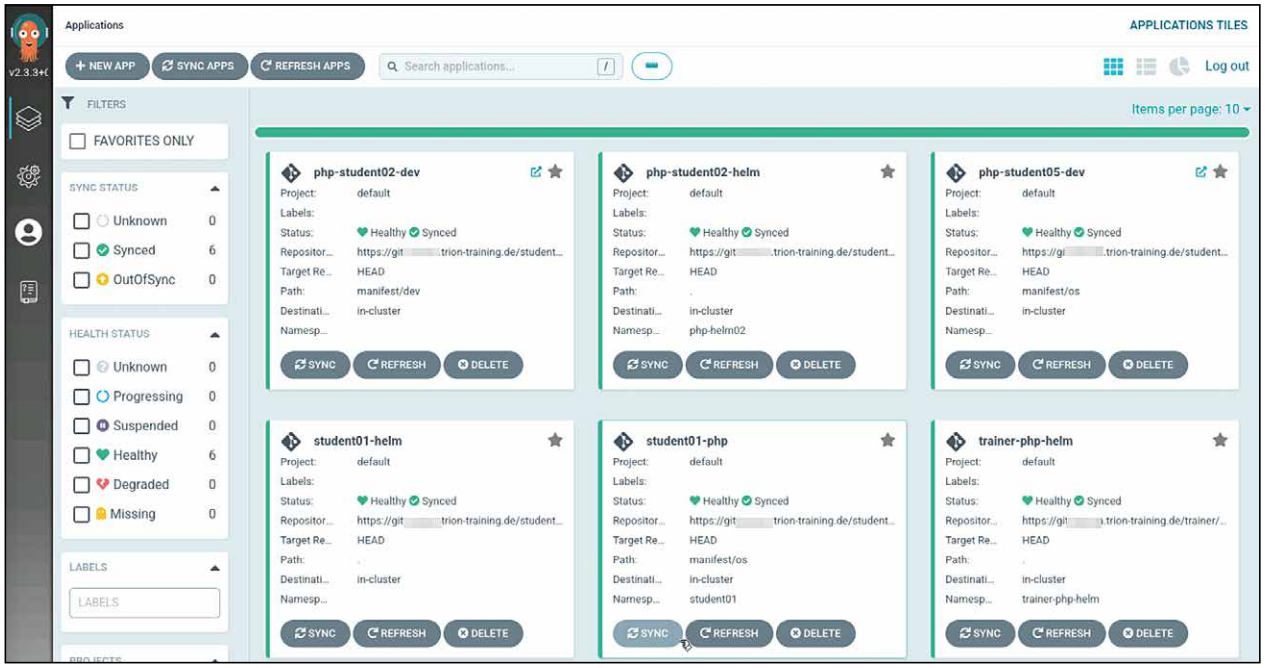

Abb. 3: Detaillierte Visualisierung von Kubernetes-Objekten und ihren Zusammenhängen in einer Anwendung in ArgoCD

ArgoCD ist dabei ein Kubernetes-natives Projekt: Es versteht die unterschiedlichen Kubernetes-Objekte, arbeitet jedoch auch ausschließlich mit Kubernetes zusammen.

In **Abbildung 3** ist zu sehen, wie ArgoCD den Zustand einer Anwendung im Kubernetes-Cluster inkl. der Zusammenhänge zwischen den einzelnen Objekten visualisiert. Sind Health-Checks definiert oder lässt sich der Zustand anderweitig ermitteln, so wird auch das optisch wiedergegeben.

Um die Diagnose zu erleichtern, bietet die ArgoCD-Oberfläche auch Zugriff auf die Kubernetes Events, detaillierte Statusinformationen und die Logdateien der betreffenden Pods. Damit ist auch ohne zusätzliche Infrastruktur schon ein guter Schritt für die Developer Experience getan. Oft reichen diese Mittel sogar aus, und der Sprung in die zentralen Logging- und Observability-Systeme oder die Verwendung von kubectl auf der Kommandozeile erübrigen sich.

Die Einrichtung von ArgoCD ist sehr simpel und in der Getting-Started-Anleitung unter [1] gut beschrieben. Nach dem Log-in können Anwendungen über die grafische Oberfläche angelegt werden. Dabei ist das GitOps-Repository anzugeben und der damit zu verknüpfende Cluster und das ArgoCD-Projekt auszuwählen. Bei einer Standardinstallation hat das Defaultprojekt alle Rechte, und die Verbindung wird zu demselben Cluster hergestellt, in dem ArgoCD selbst läuft. Es ist also sehr einfach, direkt zu starten.

Bei ArgoCD-Anwendungen kann der Administrator sich entscheiden, ob ArgoCD automatisch die Synchronisierung vornehmen oder lediglich auf Änderungen der Zielkonfiguration prüfen soll – und dies im UI anzeigt. Je nach Umgebung kann das eine oder andere Verfahren gewünscht sein: Für Entwicklungs-Stages soll möglichst schnell der aktuelle Stand bereitgestellt werden, sodass eine automatische Synchronisierung sinnvoll ist. Für produktive Umgebungen soll das ggf. ein bewusster manueller Schritt sein, der durch einen Administrator ausgeführt wird. Genauso kann ArgoCD auch wieder Objekte entfernen, wenn diese nicht mehr benötigt werden. Bei diesem Pruning gilt es ebenfalls zu entscheiden, ob es nur manuell oder automatisiert erfolgen soll.

ArgoCD verwaltet seinen eigenen Zustand ebenfalls als Kubernetes-CRD-Objekte. Damit ist es möglich, auch automatisiert – über das Kubernetes API – Anwendungen zur Verwaltung durch ArgoCD anzulegen oder zu entfernen. Somit lassen sich Konstrukte – wie ein automatisches Deployment einer Anwendung je Branch – automatisieren. Neben reinen Kubernetes-Manifesten unterstützt ArgoCD auch Helm als Kubernetes-Paketmanager und das leichtgewichtige kustomize. Mit Helm lassen sich Anwendungen statt aus git- auch aus Helm-Repositorys inkl. Konfiguration von Variablen steuern. Werden lediglich verschiedene Varianten, z. B. für verschiedene Umgebungen mit unterschiedlicher Skalierung oder Ingress-Konfiguration, benötigt, ist kustomize ein guter Einstieg im Vergleich zu dem sehr flexiblen, aber auch komplexeren Helm.

Fazit

Mit ArgoCD lassen sich transparente und revisionssichere Abläufe umsetzen, um Anwendungen in Kubernetes mit GitOps-Verfahren zu betreiben. Wer als Entwickler einmal den Komfort und die guten Diagnosemöglichkeiten von ArgoCD kennengelernt hat, der möchte nur ungern wieder darauf verzichten.

Wer bisher noch keinen Einstieg in GitOps-Verfahren gefunden hat, kann mit ArgoCD z. B. in einem Minikube innerhalb weniger Minuten loslegen. Und stellt sich ArgoCD anschließend nicht als das präferierte Werkzeug heraus, so lassen sich die Prinzipien und gesammelten Erfahrungen auch leicht auf andere Produkte übertragen. Das Investment zahlt sich insofern in jedem Fall aus und man wird mit einer ausgezeichneten Developer Experience belohnt.

Übrigens: Rund um Argo tummeln sich dazu noch weitere Werkzeuge, die einen Blick wert sind: ArgoEvents und ArgoWorkflows. Damit lassen sich weitergehende Automatisierungen und Integrationen erstellen, die auch gut mit ArgoCD kombinierbar sind.

Thomas Kruse begann seine Karriere 1998 als freiberuflicher Berater. Heute ist er bei der trion development GmbH [2] und unterstützt Unternehmen als Architekt, Trainer und Coach für Projekte, die Cloud- und Java-Technologien einsetzten. In seiner Freizeit engagiert er sich für Open-Source-Projekte und organisiert die Java User Group und die Frontend Freunde in Münster.

tk@trion.de

Links & Literatur

[1] https://argo-cd.readthedocs.io/en/stable/getting_started/

[2] https://www.trion.de

Infrastrukturautomatisierung mit Docker Images und Helm Charts

CI für Infrastructure as Code

Dank der immer größeren Verbreitung von Continuous Delivery gewinnt die Automatisierung von Infrastruktur immer mehr an Bedeutung. Die Idee, Infrastruktur mit Hilfe von Quelltexten zu beschreiben, findet immer mehr Anhängerschaft, und daraus ergeben sich neue Herausforderungen für Operations, aber auch für Developer, die mit Infrastruktur in Berührung kommen. Dieser Artikel beschreibt, welchen Herausforderungen Operations und Developer sich bei der Arbeit mit Infrastructure as Code stellen müssen und wie sie aus den Erfahrungen der klassischen Softwareentwicklung lernen können, diese zu meistern.

von Sandra Parsick

Infrastruktur kommt in der Continuous-Delivery-Beschreibung das erste Mal in den Kategorien Konfigurationsmanagement und automatische Verteilung vor (**Abb. 1**).

Mit Konfigurationsmanagement ist die Automatisierung der Infrastrukturbereitstellung gemeint. Bei automatischer Verteilung geht es um die Automatisierung der Installation von Softwareartefakten.

Der Begriff Infrastruktur ist weit gefächert. Sie lässt sich in mehrere Schichten aufteilen (**Abb. 2**). Die Basis bilden virtuelle Maschinen (VMs) bzw. Server. Darauf bauen Betriebssysteme (OS) auf. Je nachdem, ob die Infrastruktur klassisch oder eher moderner aufgestellt ist, kommen Laufzeitumgebungen oder Plattformen ins Spiel. Am Ende geht es um die Bereitstellung der Anwendungen.

Für die Automatisierung der einzelnen Schichten sind oft unterschiedliche Werkzeuge verantwortlich. Operations, die ihre Infrastruktur automatisieren wollen, müssen anfangen, Code zu schreiben. Jedes Werkzeug bringt zwar seine eigene Codesyntax mit, doch ergeben sich bei der Arbeit mit diesen Werkzeugen ähnliche Fragestellungen, die sich auch Developer in der klassischen Softwareentwicklung gestellt und dafür auch Antworten gefunden haben.

Für Infrastructure as Code brauchen die Operations einen Entwicklungsprozess. Wenn Developer Infrastructure as Code betreiben, müssen sie sich fragen, wie sie

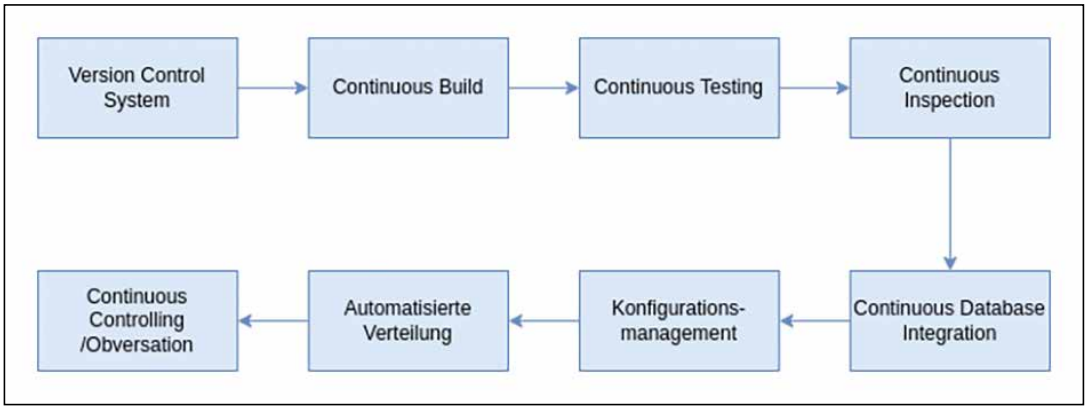

Abb. 1: Continuous-Delivery-Überblick

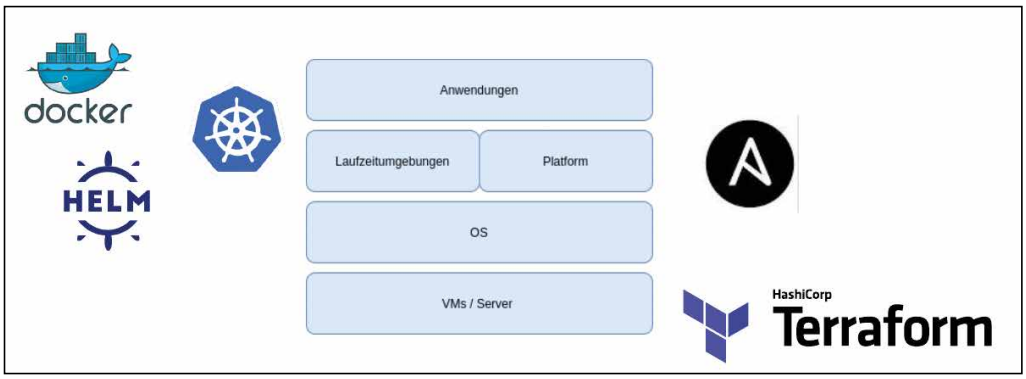

Abb. 2: Infrastruktur-Stack

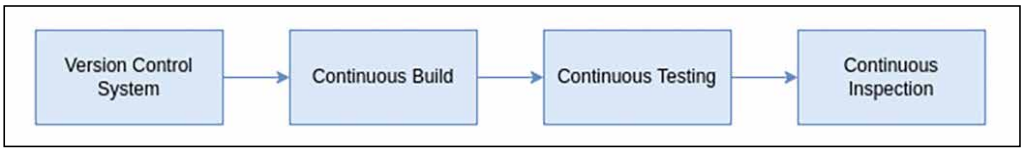

Abb. 3: Überblick Continuous Integration

diesen neuen Aspekt in ihren Entwicklungsprozess integrieren.

Ein genauer Blick auf die Continuous Delivery Pipeline ergibt, dass die ersten vier Kategorien Version Control System, Continuous Build, Continuous Testing und Continuous Inspection Methodiken beschreiben, wie die Developer die Qualität des Codes verbessern können.

Das Version Control System kümmert sich darum, wie Quelltext strukturiert und versioniert abgelegt werden kann. Mit Continuous Build stellen die Developer sicher, dass der geschriebene Quelltext immer lauffähig ist. Continuous Testing hilft, sicherzustellen, dass der geschriebene Quelltext gemäß einer Erwartungshaltung funktioniert, und Continuous Inspection stellt mit statischer Codeanalyse sicher, dass der Quelltext bestimmten Regelsätzen entspricht. Diese vier Kategorien werden auch unter dem Begriff Continuous Integration zusammengefasst (**Abb. 3**).

In den nächsten Abschnitten werden typische Fragestellungen aus der Arbeit mit Infrastructure as Code vorgestellt und wie dabei Methodiken aus Continuous Integration helfen können. Als Beispiel dient der Anwendungsfall, in dem Developer Container und Kubernetes-Deployment-Skripte für ihre Anwendung mit Hilfe von Helm Charts [1] und Docker [2] bereitstellen sollen.

```
.
├── deployment
│   └── helm-charts
│       ├── local-values.yaml
│       └── spring-boot-demo
│           ├── Chart.yaml
│           ├── templates
│           │   ├── deployment.yaml
│           │   ├── _helpers.tpl
│           │   ├── ingress.yaml
│           │   ├── service.yaml
│           │   └── tests
│           │       └── test-connection.yaml
│           └── values.yaml
├── LICENSE
├── pom.xml
├── README.md
├── spring-boot-example.iml
└── src
    ├── main
    │   ├── docker
    │   │   └── Dockerfile
    │   ├── java
    │   └── resources
    └── test
        └── java
```

Abb. 4: Ordnerstruktur mit Helm Charts und Dockerfile

```xml
<plugin>
    <groupId>io.fabric8</groupId>
    <artifactId>docker-maven-plugin</artifactId>
    <version>0.40.0</version>
    <executions>
        <execution>
            <id>docker-build</id>
            <goals>
                <goal>build</goal>
                <goal>push</goal>
            </goals>
        </execution>
    </executions>
    <configuration>
        <images>
            <image>
                <name>spring-boot-demo:latest</name>
                <build>
                    <dockerFile>Dockerfile</dockerFile>
                    <assembly>
                        <descriptorRef>artifact</descriptorRef>
                    </assembly>
                </build>
            </image>
        </images>
        <pushRegistry>localhost:6000</pushRegistry>
    </configuration>
</plugin>
```

Abb. 5: Integration des Docker Image Build in Maven

Version Control System

Wenn Operations oder auch Developer mit Infrastructure as Code anfangen, dann liegen die Skripte oft in einem Verzeichnis auf einer geteilten Maschine, vielleicht auch in einem Wiki. Bei Änderungen gibt es keine Historie, und bei Fehlern ist es schwierig, auf einen lauffähigen Stand zurückzugehen.

Die Einführung eines Versionskontrollsystems (VCS) wie Subversion oder Git hilft bei dieser Fragestellung. Als gute Praxis hat sich herausgestellt, alles, was zu einem Projekt gehört, im VCS abzulegen. Dazu gehören neben dem Programmcode auch Build-Skripte, Deployment-Skripte und – abgesehen von Secrets – alles, was dazu notwendig ist, aus dem aktuellen Stand des Codes im Repository ein lauffähiges Artefakt zu erstellen und es in Produktion zu bringen. Trennt man Artefaktcode von Bereitstellungscode, ist man sich nie sicher, ob die beiden Stände noch zusammenpassen. Jeder im Team checkt seine Änderungen häufig ein und alle gemeinsam sind für den Quelltext verantwortlich.

Im Beispiel (**Abb. 4**) werden die Dockerfiles und die Helm Charts mit dem Programmcode der Anwendung in dasselbe Repository abgelegt und mit diesem versioniert.

Continuous Build

Auch wenn alles im VCS liegt, kann es passieren, dass der Quelltext nicht lauffähig ist. In der klassischen Softwareentwicklung hilft hier Continuous Build. Der Build-Lauf wird automatisiert und somit kann jeder im Team lokal überprüfen, ob der Quelltext lauffähig ist. Durch die Automatisierung des Build-Laufs kann auch jede Änderung im VCS durch automatisiertes Bauen überprüft werden. Im Beispiel integrieren die Developer den Docker Image Build (**Abb. 5**) und die Paketierung der Helm-Charts (**Abb. 6**) in ihr Build-Werkzeug.

Es besteht die Möglichkeit, Docker Image Build und Helm-Charts-Paketierung ohne die Build-Werkzeug-Integration regelmäßig auszuführen. Die Erfahrung zeigt aber, dass Developer das gerne vernachlässigen und dadurch erst vom CI-Server Feedback erhalten, ob der Image Build oder die Paketierung funktioniert hat.

Ein weiterer Grund ist, dass abstrakt gesehen ein Docker Image ein anderes Format für ein Deployment-Artefakt ist und Helm Charts ihr eigenes Auslieferungsformat für die Deployment-Skripte mitbringen. Jedes Build-Werkzeug hat u. a. die Zielsetzung, ein fertiges Deployment-Artefakt zu erzeugen. In diesem Fall sind das ein Docker Image und die fertige Paketierung der Deployment-Skripte.

Die Überwachung im CI-Server ist somit leicht umzusetzen. Bei der Integration in das Build-Werkzeug geschieht es automatisch, wenn das Anwendungsrepository schon unter CI-Überwachung stand.

```xml
<plugin>
    <groupId>io.kokuwa.maven</groupId>
    <artifactId>helm-maven-plugin</artifactId>
    <version>6.3.0</version>
    <configuration>
        <chartDirectory>${project.basedir}/deployment/helm-charts</chartDirectory>
        <chartVersion>${project.version}</chartVersion>
        <helmVersion>3.8.1</helmVersion>
    </configuration>
    <executions>
        <execution>
            <id>build-chart</id>
            <phase>package</phase>
            <goals>
                <goal>package</goal>
            </goals>
        </execution>
        <execution>
            <id>upload-chart</id>
            <phase>deploy</phase>
            <goals>
                <goal>upload</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

Abb. 6: Integration der Helm-Charts-Paketierung in Maven

Continuous Testing

Auch wenn der Quelltext dank Continuous Build immer lauffähig ist, kann es passieren, dass er nach Änderungen nicht immer

```yaml
schemaVersion: 2.0.0

fileExistenceTests:
  - name: 'application'
    path: '/application/'
    shouldExist: true

metadataTest:
  exposedPorts: ["8080"]
  workdir: "/application"
```

Abb. 7: Beispiel für einen Docker-Image-Test mit Container Structure Test

```
→ container-structure-test test --image sparsick/spring-boot-demo --config spring-boot-test.yaml

=============================================
====== Test file: spring-boot-test.yaml ======
=============================================
=== RUN: File Existence Test: application
--- PASS
duration: 0s
=== RUN: Metadata Test
--- PASS
duration: 0s

=============================================
================== RESULTS ==================
=============================================
Passes:      2
Failures:    0
Duration:    0s
Total tests: 2

PASS
```

Abb. 8: Testlauf von Container Structure Test

```
apiVersion: v1
kind: Pod
metadata:
  name: "{{ include "spring-boot-demo.fullname" . }}-test-connection"
  labels:
    {{- include "spring-boot-demo.labels" . | nindent 4 }}
  annotations:
    "helm.sh/hook": test
spec:
  containers:
    - name: wget
      image: busybox
      command: ['wget']
      args: ['{{ include "spring-boot-demo.fullname" . }}:{{ .Values.service.port }}/actuator/health']
  restartPolicy: Never
```

Abb. 9: Beispiel für einen Helm-Chart-Test

das tut, was er soll. Der Grund dafür ist, dass die Developer nach Änderungen im Quelltext nicht immer testen, ob er immer noch so funktioniert wie erwartet. Continuous Testing adressiert genau dieses Problem. Dabei hat es sich als gute Praxis erwiesen, dass zum Quelltext Tests geschrieben und automatisiert und wiederholbar ausgeführt werden. Jeder im Team ist in der Lage, die Tests lokal auf dem eigenen Rechner auszuführen.

Bei Docker Images möchten die Developer testen, ob die Struktur im Image der Erwartung entspricht. Dabei kann das Testwerkzeug Container Structure Test [3] helfen (**Abb. 7**). Es bietet Möglichkeiten, die Dateistruktur und Dateiinhalte im Image dahingehend zu testen. Auch ist es möglich, zu prüfen, ob bestimmte Metadaten richtig gesetzt wurden. Der Testlauf erfolgt über einen CLI-Aufruf (**Abb. 8**).

Helm bringt seine eigene Möglichkeit mit, Tests zu definieren und auszuführen (**Abb. 9**). Dabei handeln es sich um sogenannte Smoke-Tests, das heißt, sie überprüfen oberflächlich, ob die Anwendung in Kubernetes läuft. Dafür definieren die Developer einen Test-Pod, der außerhalb der eigentlichen Anwendung testet, ob sie in Kubernetes läuft. Auch hier erfolgt der Testlauf über einen CLI-Aufruf (**Abb. 10**).

Für solche Tests stellt sich für die Developer die Frage, wie sie lokal ausgeführt werden können, da sie dafür einen Kubernetes-Cluster brauchen. Mittlerweile gibt es Werkzeuge, wie zum Beispiel minikube [4], die mit Hilfe von Containern einen lokalen Kubernetes-Cluster bereitstellen können. Welches dieser Werkzeuge zum Einsatz kommen soll, hängt stark davon ab, wie der produktive Kubernetes-Cluster installiert ist. Zum Beispiel ist es interessant, welche Implementierung für den Ingress-Controller benutzt wird. Zwar sind das API und einige Funktionalitäten eines Ingress-Controllers standardisiert. Aber jede Implementierung (z. B. nginx oder Traefik) bringen ihre eigenen Features und somit Konfigurationsmöglichkeiten mit. Je nach Antwort muss der lokale Cluster entsprechend konfiguriert werden, oder es wird ein Werkzeug gewählt, das in seiner Default-Konfiguration die passenden Implementierungen nutzt. Das gilt auch für andere Kubernetes-Objekte.

Damit die Developer die Helm Charts lokal ausführen können, benötigen sie auf den lokalen Cluster angepasste Value-Dateien. Diese werden am besten mit im Repository abgespeichert (**Abb. 11:** *local-values.yaml*).

```
→ helm test spring-boot-demo-instance
NAME: spring-boot-demo-instance
LAST DEPLOYED: Sun Jul 24 15:57:40 2022
NAMESPACE: default
STATUS: deployed
REVISION: 3
TEST SUITE:     spring-boot-demo-instance-spring-boot-demo-helm-chart-test-connection
Last Started:   Mon Jul 25 07:43:40 2022
Last Completed: Mon Jul 25 07:44:14 2022
Phase:          Failed
Error: pod spring-boot-demo-instance-spring-boot-demo-helm-chart-test-connection failed
```

Abb. 10: Testlauf eines Helm-Chart-Tests

```
→ helm upgrade -i spring-boot-demo-instance spring-boot-demo -f local-values.yaml
Release "spring-boot-demo-instance" has been upgraded. Happy Helming!
NAME: spring-boot-demo-instance
LAST DEPLOYED: Sun Jul 24 15:57:40 2022
NAMESPACE: default
STATUS: deployed
REVISION: 3
```

Abb. 11: Lokale Helm-Chart-Installation

```
→ hadolint Dockerfile
Dockerfile:3 DL3041 warning: Specify version with `dnf install -y <package>-<version>`.
Dockerfile:5 DL3042 warning: Avoid use of cache directory with pip. Use `pip install --no-cache-dir
<package>`
```

Abb. 12: Linting mit hadolint

```
→ helm lint spring-boot-demo -f local-values.yaml
==> Linting spring-boot-demo
[INFO] Chart.yaml: icon is recommended

1 chart(s) linted, 0 chart(s) failed
```

Abb. 13: Linting mit Helm

Continuous Inspection

Wenn mehrere Personen im Team am Quelltext arbeiten, ist es irgendwann sinnvoll, einen Styleguide zu etablieren. Dazu kommt, dass das Team möchte, dass in Codereviews nicht immer dieselben Sachen angemerkt werden müssen und der Fokus auf inhaltlichen statt auf stilistischen Fragen liegen kann.

Hierbei kann Continuous Inspection mit dem Einsatz automatisierter statischer Codeanalyse (Linting) helfen. Beim Linting analysieren sogenannte Linter den Quelltext anhand festgelegter Regelsätze. Diese können aus der Community kommen oder das Team erstellt eigene. Ein weiterer Vorteil ist, dass die Linter auch gleich einen Syntaxcheck machen können, ohne dass der Quelltext ausgeführt werden muss.

Bei Docker Images kann das Werkzeug hadolint [5] helfen. Es überprüft das Dockerfile dahingehend, ob es die Best Practices für Dockerfiles [6] befolgt und gibt bei Abweichungen Meldung (**Abb. 12**).

Helm bringt seinen eigenen Linter mit. Dieser überprüft u. a., ob alle Values im Template aufgelöst werden können und ob die generierten Kubernetes-Deskriptoren eine korrekte YAML-Syntax haben (**Abb. 13**).

Fazit

Auch wenn Infrastructure as Code für viele Developer etwas Neues ist, müssen sie ihre über mehrere Jahre gelernten Methodiken nicht über Bord werfen, sondern können sie auch auf Infrastrukturcode übertragen. Auch Operations profitieren von dieser Erfahrung und müssen keine neuen Entwicklungsprozesse erfinden, sondern können aus der klassischen Softwareentwicklung lernen.

Sandra Parsick ist Java Champion und arbeitet als freiberufliche Softwareentwicklerin und Consultant im Java-Umfeld. Seit 2008 beschäftigt sie sich mit agiler Softwareentwicklung in verschiedenen Rollen. Ihre Schwerpunkte liegen im Bereich der Java-Enterprise-Anwendungen, Cloud, Software Craftsmanship und in der Automatisierung von Softwareentwicklungsprozessen. Darüber schreibt sie gerne Artikel und spricht auf Konferenzen. In ihrer Freizeit engagiert sie sich in verschiedenen Programmkomitees und Communitygruppen.

mail@sandra-parsick.de

@SandraParsick www.sandra-parsick.de

Links & Literatur

[1] https://helm.sh

[2] https://docs.docker.com

[3] https://github.com/GoogleContainerTools/container-structure-test

[4] https://minikube.sigs.k8s.io/docs/

[5] https://github.com/hadolint/hadolint

[6] https://docs.docker.com/develop/develop-images/dockerfile_best-practices/

Erste Schritte mit Quarkus

Senkrechtstarter

Quarkus hat die Herzen vieler Entwickler:innen im Sturm erobert. Das liegt unter anderem daran, dass hier Produktivität und Ease of Development großgeschrieben werden. Mit dem Framework zu arbeiten, macht schlichtweg eine Menge Spaß. Zudem gelingt der Einstieg in der Regel recht schnell, da überwiegend bekannte APIs und Technologien zum Einsatz kommen. Dieser Artikel begleitet auf den ersten Schritten mit Quarkus.

von Thilo Frotscher

Betrachtet man die Marktentwicklung der vergangenen Jahre, so hat sich für die Implementierung neuer Java-Anwendungen ganz eindeutig Spring Boot zum Favoriten der meisten Teams entwickelt. Das lag sicherlich unter anderem daran, dass es bereits zuvor eine große Basis an Spring-Entwickler:innen gab und Spring Boot bezüglich Produktivität noch einmal eine beträchtliche Verbesserung darstellte. Ein nicht zu unterschätzender Faktor war jedoch sicherlich auch die Tatsache, dass sich die Java-EE-Plattform als wichtigster Wettbewerber aufgrund des Umzugs zur Eclipse Foundation über mehrere Jahre praktisch nicht weiterentwickelt hat.

Der aktuelle Stand der inzwischen zu Jakarta EE [1] umbenannten Plattform gilt daher weithin als technisch angestaubt (wenngleich es nun endlich mit der Weiterentwicklung losgehen soll). Gleichzeitig gibt es jedoch eine unfassbar große Menge von Entwickler:innen, die über viele Jahre umfangreiches Know-how zu Java EE aufgebaut haben. Und es gibt eine ebenfalls riesige Anzahl existierender Java-EE-Anwendungen, die weiterentwickelt und gepflegt werden müssen. Für diese stellt Spring Boot in vielen Fällen keinen sinnvollen Zukunftspfad dar. Denn eine Migration dieser existierenden Anwendungen auf ein komplett anderes Framework wäre schlichtweg zu aufwendig. Und nicht zuletzt gibt es durchaus auch einige Kritikpunkte, was die Eignung von Spring-Boot-Anwendungen für Cloud oder Serverless anbelangt.

In dieser Gemengelage entstand in den vergangenen Jahren eine Vielzahl neuer Java-Frameworks, die sich zum Ziel setzten, die Entwicklung zeitgemäßer Anwendungen mit hoher Produktivität zu vereinen. Zeitgemäß bedeutet in diesem Zusammenhang, dass aktuelle Trends wie Cloud, Serverless, Docker und Kubernetes entweder gut unterstützt werden oder dem Einsatz des Frameworks zumindest nicht entgegenstehen. Eines der spannendsten dieser Frameworks ist Quarkus aus dem Hause Red Hat. Es vereint hohe Produktivität mit aktuellen Technologien und legt großen Wert darauf, Entwickler:innen die Arbeit so einfach und angenehm wie möglich zu machen. Vieles davon mag wirken, als wäre es von Spring Boot abgekupfert, was jedoch nicht unbedingt negativ zu bewerten ist.

Ganz im Gegenteil ist es sogar von Vorteil, wenn sinnvolle und bewährte Konzepte übernommen werden, anstatt das Rad immer wieder neu zu erfinden. So können sich Entwickler:innen leichter zwischen Projekten aus beiden Welten hin und her bewegen. Ein weiterer Vorteil von Quarkus besteht darin, dass keine proprietären APIs zu erlernen sind. Stattdessen vereint das Framework wohlbekannte und weitverbreitete Technologien wie JAX-RS, CDI und JPA mit den neuen Möglichkeiten des MicroProfile [2]. Zusätzlich wird über sogenannte Quarkus Extensions eine Vielzahl von bekannten und verbreiteten Technologien integriert, wie beispielsweise Hibernate, Flyway, Kafka, Elasticsearch, Redis, Neo4J, Vault, AWS oder Vert.x, um nur einen kleinen Teil zu nennen.

Eben diese Extensions sind ein wichtiger Bestandteil des Quarkus Frameworks. Sie sind überwiegend so gestaltet, dass nach ihrer Hinzunahme zu einem Projekt die jeweils integrierte Technologie sich so nahtlos wie möglich einfügt. Notwendige Konfigurationen werden automatisch so vorbelegt, dass Entwickler:innen keinen oder nur minimalen zusätzlichen Aufwand haben, um die Extension direkt einzusetzen. Selbstverständlich

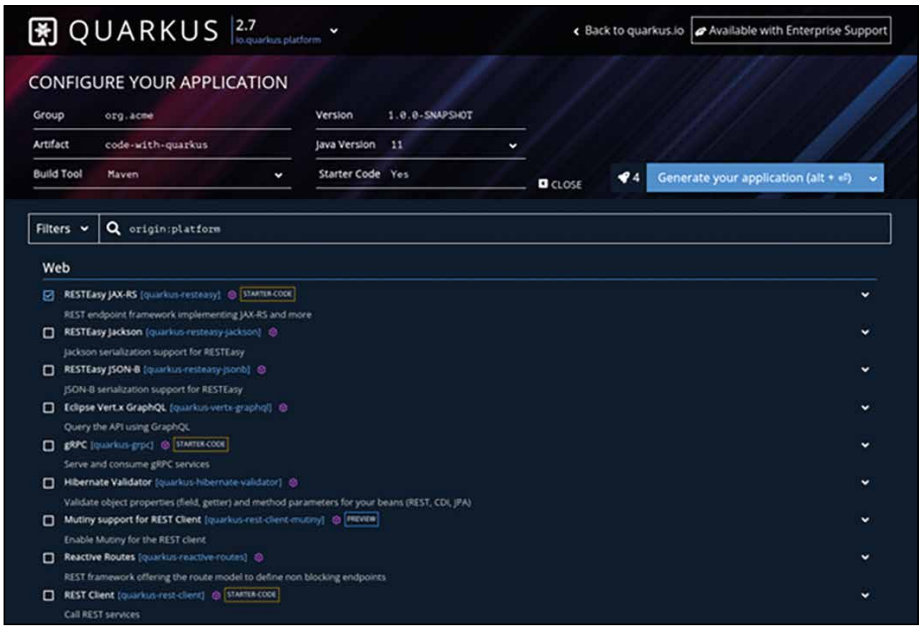

Abb. 1: Browserbasierte Konfiguration eines neuen Quarkus-Projekts

können solchen Konventionen und Standardeinstellungen bei Bedarf überschrieben werden. Hier wird Ease of Development großgeschrieben und so dafür gesorgt, dass Einsteiger:innen sich schnell mit Quarkus zurechtfinden.

Los geht's

Es gibt unterschiedliche Wege, um ein neues Quarkus-Projekt zu beginnen. Einen leichten Einstieg stellt beispielsweise die Webseite unter [3] dar (**Abb. 1**), die mit dem „spring intializr" vergleichbar ist. Hier können gewünschte Eigenschaften der neuen Quarkus-Anwendung eingestellt werden, etwa Group und Artifact ID, das zu verwendende Build-Tool (Maven oder Gradle) und die Java-Version (11 oder 17). Zusätzlich besteht die Möglichkeit, benötigte Quarkus Extensions auszuwählen. Im Anschluss kann nach einem Klick auf den Button GENERATE YOUR APPLICATION ein *zip*-Archiv mit einem entsprechenden Projektgerüst heruntergeladen werden.

Eine alternative Möglichkeit, um neue Projekte zu beginnen, besteht darin, an der Kommandozeile das Quarkus-Maven-Plug-in aufzurufen, um ein initiales Projektgerüst zu generieren. In diesem Fall werden die gewünschten Eigenschaften des neuen Projekts als Kommandozeilenparameter angegeben. Mit Hilfe des Parameters *buildTool* kann dabei erreicht werden, dass Gradle anstelle von Maven als Build-Tool für das neue Projekt verwendet wird. Im weiteren Verlauf gehen wir davon aus, dass Maven zum Einsatz kommt:

```
mvn io.quarkus.platform:quarkus-maven-plugin:2.7.5.Final:create \
  -DprojectGroupId=org.acme \
  -DprojectArtifactId=code-with-quarkus \
  -Dextensions="resteasy"
```

Schließlich bietet Quarkus auch noch ein eigenes Kommandozeilenwerkzeug (CLI), das beispielsweise mit SDKMAN!, Homebrew oder JBang installiert werden kann. Anschließend wird es durch Eingabe von *quarkus* an der Kommandozeile gestartet. Mit seiner Hilfe können ebenfalls Gerüste für neue Projekte erzeugt werden. Daneben unterstützt das Werkzeug auch bei anderen Aufgaben während der Entwicklungsphase, etwa beim Kompilieren eines Projekts oder dem Hinzufügen und Verwalten von Quarkus Extensions.

Nachdem man sich für einen der obigen Ansätze entschieden hat, könnte ein erstes Quarkus-Projekt mit der Extension *quarkus-resteasy* erstellt werden. Diese Extension sorgt für eine Unterstützung von JAX-RS, es könnte damit also eine Anwendung mit HTTP API implementiert werden, beispielsweise ein Microservice. Als JAX-RS-Implementierung kommt hinter den Kulissen RESTEasy zum Einsatz, wie der Name der Extension vermuten lässt.

In **Abbildung 1** ist zu erkennen, dass einige Extensions sogenannten „Starter-Code" mitbringen. Dabei handelt es sich um einfache Beispiele als Startpunkt für eigene Anwendungen, die im generierten Projektgerüst enthalten sind. Die Extension *quarkus-resteasy* bietet solchen Starter-Code. Generiert man ein Projekt mit dieser Extension, dann enthält es unter anderem die JAX-RS-Klasse *GreetingResource* sowie eine dazugehörige Testklasse namens *GreetingResourceTest* auf Basis des Test-Frameworks REST Assured. Die Klasse *GreetingResourceIT* dient dem Test der gleichen Ressource, wird jedoch dann verwendet, wenn die Anwendung in eine native Binärdatei kompiliert wird. Dazu später mehr.

Neben diesen Beispielklassen werden auch verschiedene Dockerfiles generiert, mit deren Hilfe die Anwendung später in ein Docker Image gepackt werden kann. Die Dockerfiles unterscheiden sich unter anderem darin, dass sie entweder für die traditionelle Ausführung der Anwendung in einer JVM vorgesehen sind oder für die Ausführung als native Binärdatei. Weiterhin finden sich im Projekt eine *index.html*-Datei, die zunächst vor allem Informationen über die generierte Anwendung enthält, sowie der Maven-Wrapper *mvnw*, der dafür sorgt, dass die richtige Maven-Version verwendet oder falls notwendig installiert wird.

Das generierte Projekt ist direkt lauffähig. Während man aktiv am Code arbeitet, empfiehlt es sich, die Anwendung im Quarkus Development Mode zu starten. Mit Maven gelingt das beispielsweise durch den Befehl

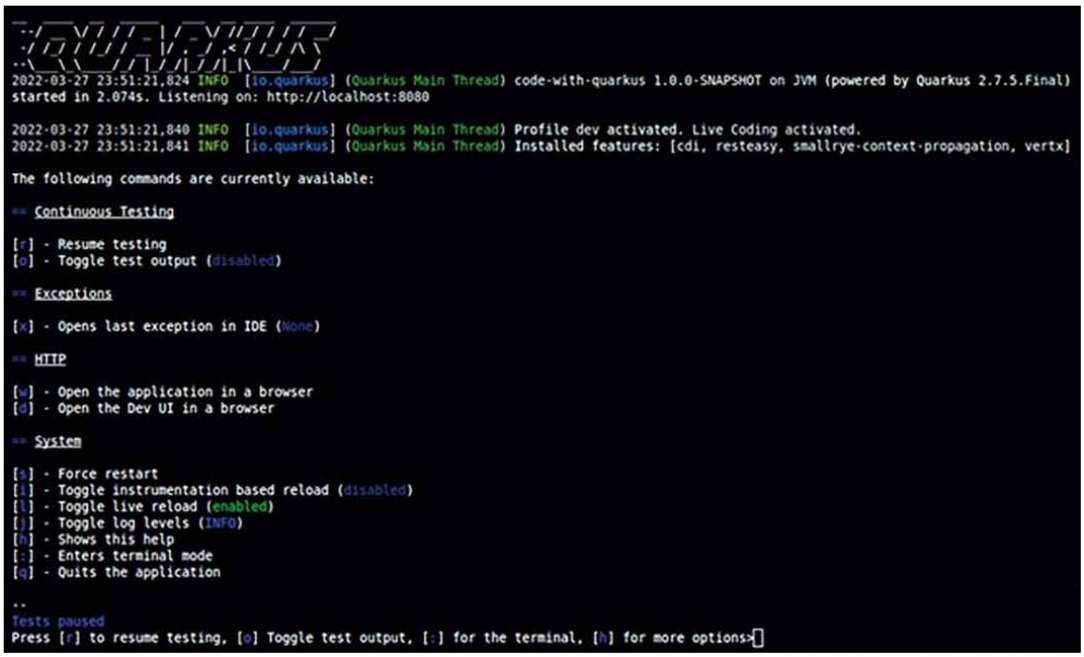

Abb. 2: Quarkus DEV Mode

```
./mvnw quarkus:dev
```

Nach dem Start der Anwendung im Development Mode (**Abb. 2**) kann sie bei Bedarf über Port 5005 mit einem Debugger verbunden werden. Insbesondere ermöglicht dieser Modus aber ein sogenanntes Hot Deployment mit Hintergrundkompilierung. Das bedeutet, dass in der IDE vorgenommene Codeänderungen automatisch in die Anwendung einfließen, ohne dass sie neu gestartet werden müsste. Die Änderungen wirken sich direkt in der laufenden Anwendung aus. Auf diese Weise erhalten Entwickler:innen sehr schnelles Feedback dazu, ob Änderungen im Programmcode das gewünschte Resultat zeigen oder nicht. Das Ganze funktioniert sogar aus der Ferne: Im Remote Development Mode kann die Quarkus-Anwendung in einer Containerumgebung ausgeführt werden. Änderungen im lokalen Dateisystem werden dann unmittelbar in den Container überführt. Aus offensichtlichen Gründen sollte der Remote Development Mode nicht in einem produktiven System eingesetzt werden.

Nachdem die Anwendung mit dem generierten Beispielcode gestartet wurde, ist sie standardmäßig auf Port 8080 erreichbar. Es könnte daher beispielsweise mit dem Werkzeug *curl* ein HTTP GET Request an *http://localhost:8080/hello* gesendet werden. Der URL-Pfad *hello* wird dabei von der generierten Beispielklasse *GreetingResource* definiert. Die Anwendung antwortet dann erwartungsgemäß mit dem Text *Hello RESTEasy*. Die Vorteile des Development Mode lassen sich leicht demonstrieren, indem man diesen Text in *GreetingResource* ändert. Bei nochmaligem Versand eines GET Request an den gleichen URL antwortet die Anwendung dann mit dem neuen Text, ohne dass ein zwischenzeitlicher Neustart notwendig gewesen wäre. Die Codeänderung wurde direkt in die laufende Anwendung übernommen.

Sollte diese Änderung aber nicht möglicherweise Auswirkungen auf den generierten Test haben? Ein Blick in die Klasse *GreetingResourceTest* offenbart, dass dieser nun tatsächlich fehlschlagen müsste, da er weiterhin den Text *Hello RESTEasy* in der Response erwartet. Zum Ausführen des Tests genügt das Drücken der Taste R für „*Re-Run all tests*“ in dem Fenster, in dem die Anwendung im Development Mode läuft. In der Folge ist dann das Fehlschlagen des Tests dort sichtbar.

Der Development Mode hält noch eine ganze Reihe weiterer nützlicher Funktionen bereit, die durch Drü-

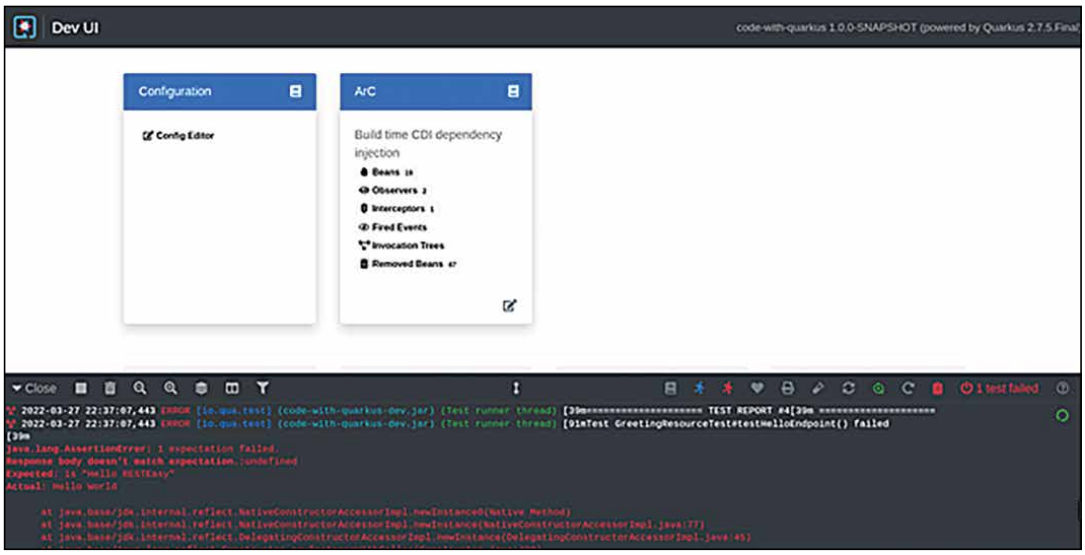

Abb. 3: Quarkus Dev UI

cken der Taste H angezeigt werden. So kann etwa der Log Level geändert, die fehlgeschlagenen Tests wiederholt oder die Anwendung neu gestartet werden. Es lohnt sich, diese Funktionen im Einzelnen auszuprobieren. Sie können bei der täglichen Entwicklungsarbeit sehr hilfreich sein und in Summe eine Menge Zeit sparen.

Eine weitere Funktion des Development Mode (Taste W) öffnet die laufende Anwendung im Browser. Hier wird nun die bereits angesprochene generierte Seite *index.html* angezeigt. Diese müsste in einem echten Projekt natürlich durch eine tatsächliche Anwendungsstartseite ersetzt werden. Ein Drücken der Taste D öffnet schließlich das Quarkus Dev UI im Browser (**Abb. 3**). Hier werden überwiegend die gleichen Funktionen geboten wie an der Konsole, einige davon jedoch komfortabler oder mit detaillierten Möglichkeiten.

Nach diesen ersten Schritten gibt es nun schier grenzenlose Möglichkeiten, verschiedene Quarkus Extensions auszuwählen und schrittweise der Anwendung hinzuzufügen. Dabei wird an vielen kleinen Dingen immer wieder deutlich, dass sich das Entwicklungsteam von Quarkus umfangreiche Gedanken darüber macht, wie sich die Arbeit mit dem Framework möglichst einfach gestalten lässt. So genügt beispielsweise das Hinzufügen von Extensions wie *smallrye-openapi*, *smallrye-health* oder *smallrye-metrics*, um eine Unterstützung typischer Microservices-Anforderungen einzuschalten. Hierbei handelt es sich um Implementierungen von drei APIs aus dem MicroProfile.

Nachdem die Extensions dem Quarkus-Projekt hinzugefügt wurden, steht jeweils ein neuer HTTP-Endpunkt automatisch zur Verfügung, über den eine OpenAPI-Beschreibung der Anwendung, eine Reihe von Metriken oder Health-Informationen angefordert werden können. Erneut handelt es sich um sinnvoll gewählte Startpunkte, die durch einfache Erweiterungen des Programmcodes oder der Konfiguration angepasst und weiter verfeinert werden können. Wählt man dagegen die Extensions für Hibernate sowie einen PostgreSQL-JDBC-Treiber aus und startet die Anwendung anschließend im Development Mode, wird automatisch ein Docker-Container mit PostgreSQL gestartet, das notwendige Docker Image bei Bedarf zuvor heruntergeladen.

Hat die Anwendung einen gewissen Entwicklungsstand erreicht, sollte sie irgendwann verpackt werden, etwa um sie in einer Testumgebung in Betrieb zu nehmen. Das Verpacken kann durch Aufruf des folgenden Kommandos erreicht werden:

```
./mvnw clean package
```

Anschließend findet sich im *target*-Verzeichnis des Projekts eine JAR-Datei, die die Klassen und Ressourcen der Anwendung enthält. Im Unterverzeichnis *target/quarkus-app* ist zusätzlich ein Runnable JAR der Anwendung abgelegt, das mit

```
java -jar target/quarkus-app/quarkus-run.jar
```

gestartet werden kann. Es handelt sich dabei allerdings nicht um ein sogenanntes Uber JAR. Die Abhängigkeiten der Anwendung sind also nicht enthalten. Sie finden sich stattdessen im Ordner *quarkus-app/lib*. Die Erzeugung von Uber JARs ist jedoch ebenfalls möglich und wird mit Hilfe der Konfigurationsoption *quarkus.package.type=uber-jar* unterstützt.

Native Binärdateien

Wie eingangs erwähnt unterstützt Quarkus die Kompilierung von Anwendungen in native Binärdateien. Sie werden dann nicht mehr in einer JVM ausgeführt, sondern eben als native Anwendungen auf dem jeweiligen Betriebssystem. Das bringt eine ganze Reihe von Vorteilen mit sich. Insbesondere starten native Anwendungen deutlich schneller und zeichnen sich durch einen geringeren Ressourcenverbrauch aus. Daher sind sie für Container und Serverless-Betrieb besser geeignet als traditionelle Java-Anwendungen.

Allerdings haben diese Vorteile auch einen Preis. Zum einen entfällt bei nativen Anwendungen offensichtlich das über viele Jahre gehegte Merkmal „Write once, run anywhere“.

Das ist im Zeitalter von Containern jedoch sicherlich in den meisten Fällen zu verschmerzen. Weiterhin wird bislang nur maximal Java 11 für die Kompilierung in native Anwendungen unterstützt. Am schwersten wiegt aber vermutlich, dass einige wichtige Java-Features im nativen Betrieb entweder gar nicht mehr oder nur unter bestimmten Bedingungen funktionieren. Hierzu zählen insbesondere Reflection oder das Laden von Klassen zur Laufzeit. Dies gilt es nicht nur im eigenen Anwendungscode zu beachten, sondern insbesondere auch in sämtlichen verwendeten Bibliotheken. Hier kommt es also gegebenenfalls darauf an, dass diese bereits speziell für den nativen Betrieb mit Quarkus angepasst wurden. Ein Beispiel dafür ist die Unterstützung von CDI. Für das Zusammenspiel mit Quarkus muss hier eine spezielle Implementierung namens Quarkus ArC zum Einsatz kommen. Sie basiert auf der CDI-2.0-Spezifikation und unterstützt diese auch weitestgehend, jedoch nicht in vollem Umfang. Details hierzu finden sich in der Dokumentation.

Um ein Quarkus-Projekt in eine native Anwendung zu kompilieren, wird eine Distribution der GraalVM [4] benötigt, die zuvor installiert werden müsste. Falls das nicht möglich oder gewünscht ist, bietet Quarkus eine sehr elegante Lösung, um dennoch zum Ziel zu kommen: Der Build-Vorgang kann in einen Container ausgelagert werden, der die GraalVM enthält. Das geschieht durch den Aufruf von

```
./mvnw package -Dnative -Dquarkus.native.container-build=true
```

Je nach Hardwareausstattung des jeweiligen Rechners kann der Build-Vorgang einige Minuten in Anspruch nehmen. Im Anschluss findet sich im *target*-Verzeichnis des Quarkus-Projekts eine native Anwendungsdatei, deren Dateiname auf *-runner* endet. Sie kann direkt aufgerufen werden und startet die Quarkus-Anwendung in beeindruckender Geschwindigkeit (**Abb. 4**). Mit Hilfe der eingangs generierten Dockerfiles kann die native Anwendung schließlich auf einfache Weise in ein Docker Image verpackt werden.

Fazit

Quarkus zählt sicherlich zu den spannendsten Frameworks, die in jüngster Vergangenheit im Java-Universum entstanden sind. Folgerichtig hat es in kurzer Zeit viel Aufmerksamkeit erlangt und eine erstaunliche Fangemeinde um sich geschart. Zahlreiche auf Quarkus basierende Projekte – teils von beachtlicher Größe – sind bereits in Entwicklung oder im produktiven Einsatz. Das Framework glänzt mit hoher Produktivität und starkem Fokus auf bestmögliche Unterstützung für Entwickler:innen. Es ist umfangreich dokumentiert, unterstützt aktuelle Technologien, wird stetig weiterentwickelt und sehr regelmäßig aktualisiert. Kurzum: Es macht einfach unglaublich viel Spaß, mit Quarkus zu arbeiten. Sein größter Trumpf ist jedoch vermutlich die Tatsache, dass es auf weithin bekannten Technologien basiert und daher nicht erfordert, proprietäre APIs zu erlernen. Insbesondere Entwickler:innen mit Vorkenntnissen in JAX-RS, CDI und JPA werden sich ausgesprochen schnell zurechtfinden. Somit kann Quarkus als Zukunftspfad für Teams und Unternehmen dienen, die in der Vergangenheit umfangreiches Java-EE-Know-how aufgebaut haben. Denn Quarkus bietet die Möglichkeit, existierende Anwendungen mit überschaubarem Aufwand zu modernisieren. Und für neue Projekte ist es zu einem sehr ernstzunehmenden Konkurrenten für Spring Boot geworden.

Thilo Frotscher arbeitet als freiberuflicher Softwarearchitekt und Trainer. Als Experte für Java, APIs und Systemintegration unterstützt er seine Kunden überwiegend durch Entwicklungstätigkeiten, Reviews oder die Durchführung von Schulungen, u. a. zu Quarkus. Einen weiteren Tätigkeitsschwerpunkt bildet die Beratung beim Design von HTTP-Schnittstellen. Thilo ist (Co-)Autor mehrerer Bücher in den Bereichen Java-Enterprise-Anwendungen, (Web-)Services und Systemintegration, hat zahlreiche Fachartikel verfasst und spricht regelmäßig auf Fachkonferenzen und Schulungsveranstaltungen, sowie bei Java User Groups.

Links & Literatur

[1] Jakarta EE: https://jakarta.ee/

[2] MicroProfile: https://microprofile.io

[3] https://code.quarkus.io

[4] GraalVM: https://www.graalvm.org/

Die Zukunft der Webentwicklung

Ein Blick in die Glaskugel

Eine neue, frische Generation von JavaScript Frameworks ist aufgekommen, deren Vertreter beispielsweise Astro, Qwik, Marko und SolidJS sind. Dieser Artikel stellt die Konzepte und die neuen Technologien vor.

von Manfred Steyer

Es gibt Milchprodukte, die länger halten als so manches JavaScript-Framework. Mit dieser pointierten Aussage lässt sich gut der Zustand der JavaScript-Welt des letzten Jahrzehnts beschreiben. Seit ein paar Jahren ist die JavaScript-Welt jedoch ziemlich stabil. Die drei großen Technologien Angular, React und Vue dominieren das Geschehen.

Das heißt aber nicht, dass die Welt stillsteht. Gerade in der letzten Zeit sind einige neue Frameworks mit frischen Ideen aufgekommen. Manche Leute sprechen hierbei von einer neuen Welle an Frameworks, andere sogar von einer neuen Generation.

Vertreter dieser Strömung sind zum Beispiel Astro [1], Qwik [2], Marko [3] und SolidJS [4]. Auch wenn die Verbreitung dieser neuen Frameworks nicht einmal ansatzweise an jene der großen Drei herankommt, lohnt sich ein Blick auf ihre Konzepte, um sich ein Bild über eine mögliche Zukunft der Webentwicklung zu machen.

In diesem Artikel zeige ich, was die neuen Technologien ausmacht und welche Probleme sie lösen, aber auch, welche Konsequenzen damit einhergehen. Dazu müssen als Erstes verschiedene Arten des Renderings unterschieden werden.

Die verwendeten Beispiele gibt es wie immer in meinem GitHub-Account [5], [6].

Clientseitiges Rendering in SPAs

Single Page Applications (SPAs) verlagern das gesamte Rendering auf die Clientseite, also in den Browser. Das bessert die Reaktionszeiten und erhöht somit die Benutzungsfreundlichkeit enorm. Da eine SPA beim Start jedoch JavaScript Bundles und Daten laden muss, verzögert clientseitiges Rendering gleichzeitig auch den Programmstart. Wie **Abbildung 1** zeigt, verstreicht somit einiges an Zeit bis zum First Meaningful Paint (FMP).

Die initiale Verzögerung von ein paar wenigen Sekunden mag bei Geschäftsanwendungen nicht ins Gewicht fallen, bei denen es wichtiger ist, dass sich die Anwendung nach dem Start flüssig verhält. Bei öffentlichen Portalen, bei denen die Conversion (der Wandel von Interesse in Aktion) im Vordergrund steht, sieht das anders aus. Hier zählt jede Millisekunde, um die Absprungraten gering zu halten.

Außerdem können Suchmaschinen nach wie vor mit HTML besser umgehen als mit JavaScript. Deswegen nutzen Portale, die auf SPA-Frameworks basieren, in

Abb. 1: Clientseitiges Rendering

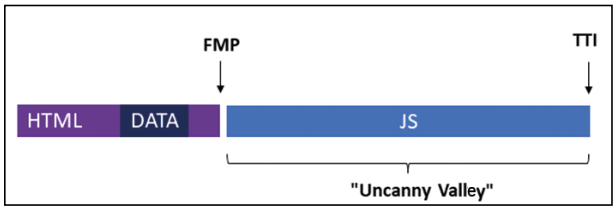

Abb. 2: Serverseitiges Rendering

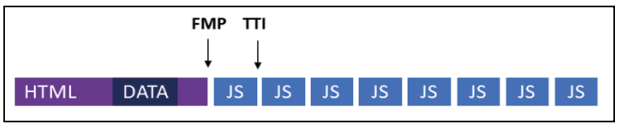

Abb. 3: Progressive Hydration

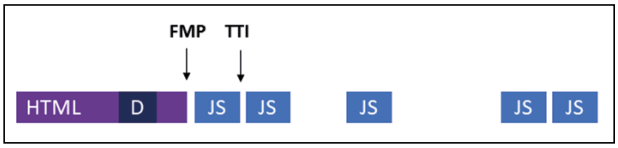

Abb. 4: Partial Hydration

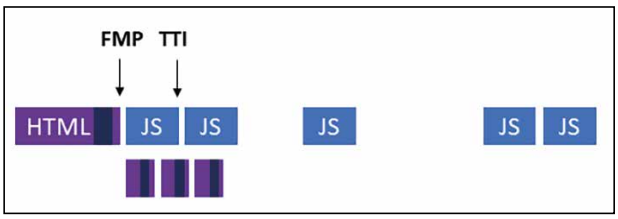

Abb. 5: Streaming

der Regel serverseitiges Rendering (SSR). **Abbildung 2** veranschaulicht den Unterschied zum clientseitigen Rendering von SPAs.

Bei SSR enthält das ausgelieferte HTML bereits sämtliche Informationen, die die jeweilige Seite ausmachen. Der First Meaningful Paint (FMP) findet also früher statt. Um die Seite interaktiv werden zu lassen, muss sie jedoch die einzelnen JavaScript Bundles laden. Sie erwecken das statische HTML zum Leben. Das wird auch Hydration genannt. Damit die JavaScript Bundles die Daten, die bereits beim serverseitigen Rendering genutzt wurden, nicht erneut laden müssen, verstaut der Server sie in einem unsichtbaren Bereich der Seite, z. B. in einem *Script*-Tag.

Obwohl bei SSR der FMP früher stattfindet, dauert es länger bis zur sogenannten Time to Interactive (TTI), da sie das Laden der Bundles abwarten muss. Ein weiterer Grund liegt darin, dass das initiale HTML nun größer ist. Es liegt also zwischen FMP und TTI eine Zeitspanne, in der die Anwendung zwar interaktiv wirkt, es aber nicht ist. Hierbei ist die Rede vom Uncanny Valley – frei übersetzt: unheimliches Tal.

Partial und Progressive Hydration als Lösung

Der viel beachtete Artikel „When everything's important, nothing is!“ [7] hat die im letzten Abschnitt diskutierte Problematik bereits 2016 aufgezeigt und als Lösung Progressive Booting vorgeschlagen. Allerdings war es mit den damaligen Frameworks nicht ohne Weiteres möglich, dieses Verfahren umzusetzen, das man heute besser unter dem Namen Progressive Hydration kennt.

Die Idee dahinter ist, dass die Seite ihre einzelnen Bereiche nach und nach, also fortschreitend (engl. „progressive“), hydriert. Somit kann der gerade sichtbare Bereich der Seite recht schnell interaktiv werden (**Abb. 3**).

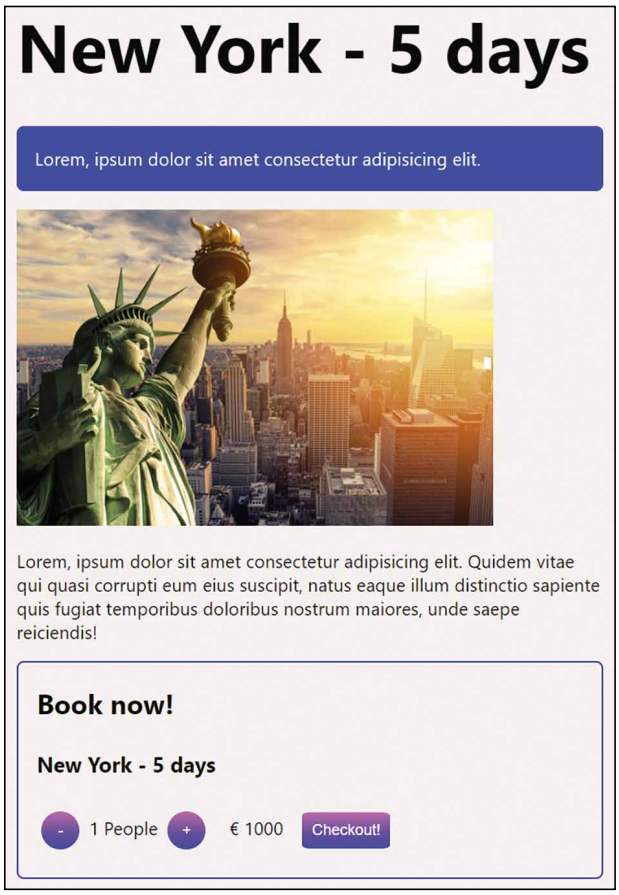

Abb. 6: Island Architecture in Astro

Manche Teile der Seite müssen eventuell gar nicht hydriert werden. Dabei kann es sich um Teile handeln, die der Benutzer nicht verwendet, aber auch um solche, die von Haus aus statisch sind. Beispiele sind Navigationsleisten, Fußzeilen, aber auch Contentbereiche in Artikeln. Schafft man es, diese Bereiche zu identifizieren, verringert sich der Aufwand für das Hydrieren weiter. Hierbei ist von Partial Hydration die Rede (**Abb. 4**).

Noch etwas schneller wird es, wenn auch das HTML in mehreren Teilen zum Client gestreamt wird. Der erste Teil enthält zum Beispiel sämtliche statischen Teile einschließlich der Platzhalter für Inhalte, die aus einer Datenbank kommen. Sobald diese Inhalte geladen wurden, sendet der Server sie über eine offengehaltene HTTP-Verbindung zum Browser (**Abb. 5**).

Umsetzung der Hydration-Spielarten

Die meisten der aktuell aufkommenden Frameworks unterstützen zumindest einige der hier gezeigten Hydration-Spielarten. Astro zum Beispiel bietet mit seiner sogenannten Island Architecture Unterstützung für Progressive und Partial Hydration. Die Idee dahinter ist, dass eine Seite aus interaktiven Inseln besteht, die in statische Contentbereiche eingebettet sind. Lediglich Letztere gilt es zu hydrieren. In **Abbildung 6** handelt es sich nur bei der Box im unteren Bereich um solch eine interaktive Insel.

Abhängig von den gewählten Einstellungen hydriert Astro die einzelnen Inseln zum Beispiel erst dann, wenn sie in den sichtbaren Bereich gescrollt werden. Alternativ dazu lassen sie sich mit freien Ressourcen vorladen. Astro lässt sich aber auch anweisen, ganz wichtige Inseln sofort beim Server anzufordern. Daneben besteht die Möglichkeit, das Hydrieren von einer Media Query und somit unter anderem von der aktuellen Bildschirmauflösung, abhängig zu machen.

Einen ähnlichen Weg geht übrigens React, das als eines der großen Drei hier mithalten kann, mit seinen Server Components. React-basierte Metaframeworks wie Next.js vereinfachen die Nutzung dieser Idee.

Das Framework Marko geht hier noch einen Schritt weiter. Bei diesem Framework versucht der zugrundeliegende Compiler, die statischen und dynamischen Seitenteile selbst zu identifizieren. Qwik hingegen untergliedert einzelne Komponenten sogar in mehrere Bundles und versucht, sie so spät wie möglich zu laden. Die Seite fordert zum Beispiel den Code eines Click Handlers erst beim Klick auf die jeweilige Schaltfläche an.

Streaming wird mittlerweile ebenfalls von React, aber auch von Qwik, SolidJS und Marko unterstützt. Die Vorgehensweise ist bei diesen Frameworks sehr ähnlich: Die Daten der zu streamenden Seitenteile repräsentiert das Framework durch Promise-Objekte oder asynchrone Ladefunktion. Sobald die Daten zur Verfügung stehen, rendert der Server die betroffenen Teile und sendet sie an den Client, der sie in den Platzhalter einfügt.

Smarte Compiler

Der Einsatz von Compilern für Frontend-Frameworks ist nichts Neues. Angular nutzt zum Beispiel schon länger einen Compiler, um HTML-Templates in möglichst effizienten JavaScript-Code zu überführen. Der Compiler von Svelte wandelt den geschriebenen Code sogar größtenteils in natives JavaScript um, sodass das Endergebnis möglichst klein und performant ist.

Wie oben bereits erwähnt, hilft der Compiler in Marko und in Qwik dabei, eine Anwendung in kleinere Teile zu zerlegen, die bei Bedarf geladen werden. Ein einfaches Qwik-basiertes Beispiel dazu findet sich in Listing 1.

Die Endung *$* bei *onClick$* veranlasst den Qwik-Compiler, den Event Handler in eine eigene Datei auszulagern sowie Code einzuführen, der diese Datei bei Bedarf lädt. Dabei kümmert sich der Compiler auch darum, die nötigen Zustände – hier die Werte von state und options – ins HTML zu serialisieren, sodass sie nach dem Laden von *incCount* zur Verfügung stehen.

Dem dahinterliegenden mentalen Modell zufolge hält Qwik die Ausführung von Komponenten nach dem serverseitigen Rendern an und setzt sie bei Bedarf beim Client fort. Deswegen ist hier auch nicht von Hydration, sondern von Resumability die Rede.

Reaktivität und Änderungsverfolgung

Ein weiteres Merkmal der jungen Frameworks besteht darin, dass sie reaktive Konzepte für eine zielgerichtete Änderungsverfolgung (Change Detection) implementieren. Während die großen Drei nach Änderungen immer ganze Komponenten im DOM aktualisieren und dabei häufig auch sämtliche Parent-Komponenten ins Visier nehmen, erlauben es reaktive Building-Blocks, lediglich betroffene Teile einzelner Komponenten zu untersuchen.

Listing 1

```
<p>
  <span>Count: {state.count}</span>.
  <button class="ml10 counter" onClick$={() =>
                                  incCount(state, options)}>[...]</button>
  <span class="ml20">€ {state.price}</span>
</p>
```

Listing 2

```
// Liste im Store verwalten
const [state, setState] = createStore({ list: ['a', 'b', 'c'] });

// Auf Liste im Store zugreifen
<For each={state.list}>{item => /*...*/}</For>
```

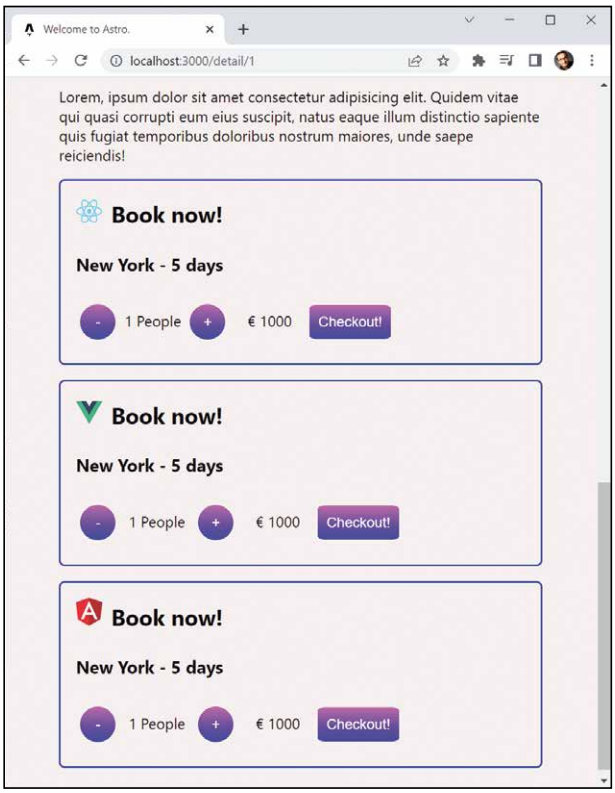

Abb. 7: Inseln basierend auf unterschiedlichen Frameworks

Die Konsequenz daraus ist auch, dass diese Frameworks stärker vorgeben, wie der Zustand der Anwendung zu verwalten ist. Qwik, Marko und SolidJS, aber auch Svelte kommen deswegen mit einem eigenen Store-Konzept. Listing 2, das aus der SolidJS-Dokumentation übernommen wurde, veranschaulicht das.

Im Gegensatz zu RxJS ist die Reaktivität bei diesen Building-Blocks implizit. Es fällt also auf den ersten Blick gar nicht auf, dass die Anwendung reaktiv ist. Vielmehr haben Entwickler:innen das Gefühl, mit herkömmlichen Komponenten zu arbeiten. Sie bekommen also kein neues Paradigma aufgezwungen.

Meta-Frameworks und Convention over Configuration

Meta-Frameworks sind mittlerweile eigentlich nichts Neues mehr: React hat Next.js, Vue hat Nuxt.js und für Svelte gibt es SvelteKit. Diese Frameworks automatisieren immer wiederkehrende Aufgaben rund um Hydration und geben Konventionen vor. Dieser Tradition folgt auch Qwik mit Qwik City.

Astro hingegen ist per Design selbst ein Meta-Framework: Es fokussiert sich auf das Rendering der statischen Seitenteile und delegiert den Umgang mit interaktiven Inseln an andere Technologien, die es über ein Adapterkonzept einbindet. Zur Veranschaulichung enthält die Astro-basierte Seite in **Abbildung 7** drei Inseln, die mit drei unterschiedlichen Frameworks entwickelt wurden.

Astro erlaubt es somit, Progressive und Partial Hydration bereits heute zu nutzen, ohne auf die Vorzüge etablierter Technologien verzichten zu müssen. Das ist ein schlauer Schachzug, zumal jüngere Frameworks mit dem Featureumfang der großen Drei nicht mithalten können. Gerade Pakete zur Verwaltung umfangreicher Formulare, wie man sie in großen Unternehmensanwendungen benötigt, sucht man da derzeit noch vergebens.

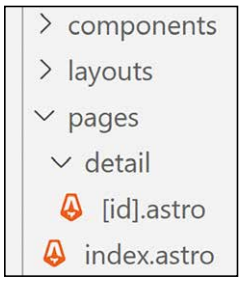

Abb. 8: Dateibasiertes Routing

Ein weiteres Merkmal, das sich in nahezu allen Meta-Frameworks findet, ist dateibasiertes Routing. Anstatt Routingregeln programmatisch zu definieren, lassen sie sich anhand von Konventionen aus der Ordnerstruktur ableiten. Ein Beispiel dafür findet sich in **Abbildung 8**. Die dort gezeigte Projektstruktur veranlasst Astro, die Route *detail/7* mit der Datei *[id].astro* zu assoziieren. Die 7 übergibt es dabei als Parameter *id*.

Bewertung

Bei den neuen Frameworks geht es in erster Linie um Performance – allem voran um Startperformance. Dazu setzen sie auf Progressive sowie Partial Hydration, aber auch Streaming sowie auf die Generierung kleinerer Bundles durch smarte Compiler. Die Konsequenz daraus ist, dass Backend und Frontend wieder stärker zusammenwachsen. Beide werden mit JavaScript entwickelt und sind aufeinander abgestimmt.

Das schränkt nicht nur die Framework-Auswahl im Backend ein, sondern macht auch die Umsetzung komplexer. Nicht alle Bibliotheken laufen problemlos sowohl im Browser als auch in einem serverseitigen Prozess. Das liegt daran, dass im JavaScript-basierten Backend einige Browser-APIs gar nicht verfügbar sind und einzelne Mechanismen wie Cookies oder Redirects ganz anders funktionieren.

Deswegen werden Hydration-Spielarten nur dann genutzt, wenn sie auch wirklich benötigt werden, nämlich bei öffentlichen Webportalen, bei denen Conversion und SEO im Vordergrund stehen. Für Geschäftsanwendungen, die sich hinter einer Log-in-Maske verbergen, ist man nach wie vor mit klassischem, clientseitigem Rendering besser beraten.

Neben der Startperformance adressieren die neuen Frameworks auch die Performance zur Laufzeit, indem sie reaktive Building Blocks einsetzen. Das erlaubt es dem Framework, Änderungen gezielt zu erkennen und in die Seite einzubringen.

Die Konsequenz daraus ist, dass die Frameworks vorgeben, wie Komponentenzustände zu verwalten sind. In der Regel bekommen Entwickler:innen ein Store-Konzept vorgegeben. Das ist jedoch keine allzu große Einschränkung für neue Lösungen, sodass es sich hierbei aus Anwendungssicht um einen Quick Win handelt.

Ähnlich ist es mit den Meta-Frameworks, die Konventionen vorgeben und die Hydration-Spielarten ab Werk liefern. Ein lästiges Konfigurieren ist somit nicht notwendig. Auch die Idee der dateibasierten Routen scheint in die Kategorie Quick Wins zu fallen.

Wie geht es weiter?

Derzeit scheint die Vormachtstellung der großen Drei – Angular, React und Vue – ungebrochen zu sein. Gerade jene, die Geschäftsanwendungen schreiben, dürften wenig Interesse an einer Umstellung auf andere Frameworks haben. Die Aufwand-Nutzen-Abwägung spricht (derzeit) nicht dafür.

Aber gerade für öffentliche Portale scheinen die neuen Frameworks verlockend zu sein. In diesem Bereich sehe ich zwei Szenarien: Es wäre zum einen möglich, dass sich eines oder ein paar der neuen Frameworks zum De-facto-Standard für öffentliche Portale und längerfristig sogar für Webanwendungen jeglicher Art entwickeln.

Da das jedoch mit viel Aufwand verbunden ist, könnten stattdessen auch die großen Drei die neuen Strömungen aufnehmen. Beispielsweise hat React bereits verschiedene Hydration-Spielarten an Bord und das Angular-Team hat angekündigt, hier in der nächsten Zeit nachziehen zu wollen. Auch die Ideen für eine gezieltere Change Detection könnte das Angular-Team in der nächsten Zeit aufgreifen, wenn es Alternativen zur Arbeit mit zone.js anbietet.

Als Mittelweg bieten sich auch Meta-Frameworks wie Astro an, die es erlauben, interaktive Inseln mit dem Framework der Wahl zu schreiben. Bestehende Entwicklungen sowie bestehendes Wissen lassen sich somit wiederverwenden.

Fazit

Die neuen Strömungen bringen einige neue Ideen ins Spiel und begünstigen in erster Linie öffentliche Portale. Diejenigen, die Geschäftsanwendungen schreiben, sind nach wie vor mit den großen Drei gut beraten. Außerdem ist es sehr wahrscheinlich, dass die großen Drei die Konzepte, die sich bewähren, aufgreifen. Die neuen Frameworks dienen in diesem Fall als Inkubatoren für neue Entwicklungen. Konkurrenz belebt eben – zu unserem Vorteil – das Geschäft!

Manfred Steyer ist Trainer und Berater mit Fokus auf Angular, Google Developer Expert und Trusted Collaborator im Angular-Team. Er schreibt für O'Reilly, das deutsche Java Magazin und Heise Developer. Unter www.ANGULARarchitects.io bieten er und sein Team tiefgehende Angular-Schulungen und Beratung an.

Links & Literatur

[1] https://astro.build

[2] https://qwik.builder.io

[3] https://markojs.com

[4] https://www.solidjs.com

[5] https://github.com/manfredsteyer/astro-last-minute

[6] https://github.com/manfredsteyer/qwik-checkout

[7] https://aerotwist.com/blog/when-everything-is-important-nothing-is/

Automatisiertes Management von Schwachstellen

Vulnerability-Management

Bei der Erstellung von Softwaresystemen können ungewollt Sicherheitsschwachstellen mit eingebaut werden. Über diese Schwachstellen können Angreifer Kontrolle über das System gewinnen oder Daten daraus abziehen. Es stehen jedoch zahlreiche Hilfsmittel zur Verfügung, um das zu vermeiden.

von Stefan Fleckenstein

Eine gute Übersicht über die am häufigsten auftretenden Sicherheitsprobleme geben die OWASP Top 10 [1]. Diese Schwachstellen möglichst früh im Entwicklungsprozess zu erkennen, bringt einige Vorteile:

1. Wenn mögliche Schwachstellen direkt im Entwicklungszyklus angezeigt werden, lernen die Entwickler:innen, diese in Zukunft direkt zu vermeiden. Das macht das System sicherer und spart Geld, weil weniger Fehler beseitigt werden müssen.
2. Selten durchgeführte, aufwendige Penetrationstests passen nicht zu agilen Vorgehensmodellen mit kurzen Auslieferungszyklen. Wenn man sich allein auf Penetrationstests verlässt, können in dazwischenliegenden Releases Schwachstellen in Produktion gebracht werden, die Angriffspunkte für mögliche Attacken sein können.

Schwachstellen können mit jedem Commit ins System kommen. Auch kann sich die Gefahrenlage mit der Zeit ändern, z. B. wenn neue Schwachstellen in genutzten Bibliotheken öffentlich bekannt werden. Deshalb ist es wichtig, Schwachstellenmanagement als einen kontinuierlichen Prozess aufzusetzen. Viele Tests zur Identifikation von Schwachstellen können automatisiert und damit organisch in den Entwicklungsprozess integriert werden. Dieser Artikel zeigt, welche Arten von automatisierten Testverfahren es gibt und stellt eine Auswahl von Werkzeugen mit Fokus auf Open-Source-Tools vor. Wir schauen uns außerdem an, wie sich diese Werkzeuge mit wenig Aufwand in den CI/CD-Prozess integrieren lassen, und beleuchten, wo ihre Grenzen und Fallstricke liegen.

Die Testarten

Es steht eine große Zahl an Open-Source- und kommerziellen Werkzeugen zur Verfügung, die man manuell zur Prüfung von Schwachstellen aufrufen kann, die sich aber zum größten Teil auch sehr gut automatisieren lassen. Diese Werkzeuge können in sechs Gruppen eingeordnet werden:

1. *Software Composition Analysis (SCA)*, Prüfung der Abhängigkeiten: Moderne Systeme werden nicht von Grund auf neu geschrieben, sondern viele Basisfunktionen werden als Bibliotheken genutzt. Das gilt nicht nur für den Anwendungscode, sondern im Fall von Docker auch für Betriebssystemfunktionen und Programme. Diese Bibliotheken können bekannte Schwachstellen haben, die sich für Angriffe nutzen lassen.

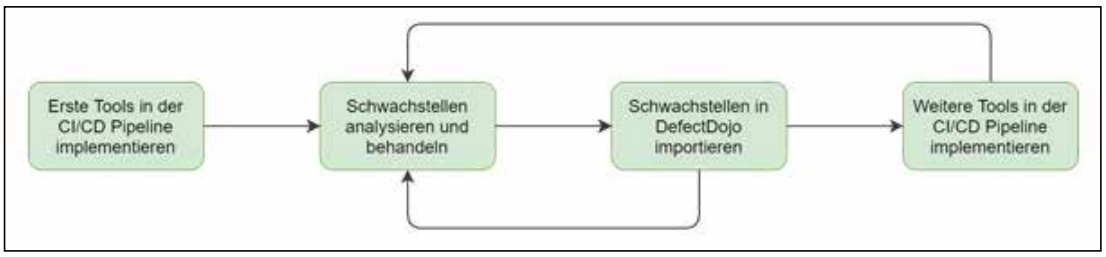

Abb. 1: Inkrementelle Implementierung von Schwachstellentests

2. *Static Application Security Testing (SAST)*, statische Codeanalysen: Im Code lassen sich durch regelbasierte Suchen viele Probleme erkennen, z. B. Injections oder schwache Verschlüsselung. Es existieren Werkzeuge für alle gängigen Programmiersprachen und auch Infrastructure as Code (Dockerfiles, Helm Charts, Terraform, …).
3. *Secrets Detection (SD)*, Suche nach Geheimnissen: Geheimnisse wie Passwörter oder API-Keys dürfen nicht mit dem Code in Repositories eingecheckt werden, und es gibt Werkzeuge, die z. B. Git Repositories über die gesamte Versionshistorie nach solchen Geheimnissen durchsuchen.
4. *Infrastrukturtests:* Auch die laufende Infrastruktur wie z. B. die Konfiguration von Cloud-Infrastrukturen, Kubernetes-Cluster, aber auch klassische Rechenzentrumsinfrastrukturen lassen sich sowohl mit Innensichten (Prüfungen, die innerhalb der Infrastruktur laufen) als auch Außensichten (Prüfungen von außen über das Internet) auf Schwachstellen überprüfen.
5. *Dynamic Application Security Testing (DAST)*, Dynamische Tests: Blackbox-Sicherheitstests, bei denen die Tests durch Angriffe auf eine Anwendung (typischerweise Webanwendungen oder APIs) von außen durchgeführt werden. Die Tests können passiv sein und nur nach Auffälligkeiten suchen oder aktive Angriffe auf das System durchführen.
6. *Interactive Application Security Testing (IAST):* IAST arbeitet innerhalb einer Anwendung durch Instrumentierung des Codes, um Probleme zu erkennen und zu melden, während die Anwendung läuft.

In diesem Artikel konzentrieren wir uns auf die ersten vier Gruppen von Prüfungen, die alle einen statischen Charakter haben. Sie lassen sich sehr gut in den Build-Prozess integrieren.

Integration in CI/CD Pipelines

Ideal ist eine Integration dieser Tests direkt in die CI/CD Pipeline, sodass sie mit jedem Build durchlaufen werden und das Team so immer den aktuellen Stand kennt und schnell reagieren kann. Glücklicherweise sind quasi alle Testwerkzeuge für den Einsatz in Pipelines vorbereitet, sodass bei der Integration wenig Aufwand entsteht. Mit einem inkrementellen Ansatz, wie in **Abbildung 1** gezeigt, können schnell erste Resultate gezeigt werden, die die Sicherheit des Systems verbessern.

Zum Start werden erste Werkzeuge in die Pipeline integriert, z. B. zur Prüfung der Abhängigkeiten im JavaScript-Code des Frontends oder zur statischen Analyse der Dockerfiles. Die Ausgabe der Ergebnisse erfolgt in einem lesbaren Format in der Konsole oder als Dateiexport. Damit kann das Team anfangen, die Ergebnisse der Prüfungen zu sichten und Schwachstellen zu beseitigen.

Die Analyse der Ergebnisse auf Basis von Konsolenausgaben oder Dateien ist mühsam. Häufig sind auch Ergebnisse zu finden, die im aktuellen Projektkontext False Positives darstellen, aber bei jedem Lauf der Pipeline wieder gemeldet werden. DefectDojo [2] ist ein Open-

Source-Programm, das die Ergebnisse der Schwachstellentests einlesen kann und in einer Benutzeroberfläche darstellt. Darin können dann die Verwaltung und das Reporting der Schwachstellen erfolgen.

In weiteren Schritten werden zusätzliche Werkzeuge in die Pipeline aufgenommen, die mehr Informationen zu Schwachstellen erzeugen, bei deren Beseitigung das System noch sicherer gemacht wird.

Software Composition Analysis (SCA)

Viele Codebestandteile für neue Anwendungen stammen heute aus Bibliotheken. Diese können bekannte Schwachstellen enthalten, die Angreifer ausnutzen können. Das prüft man mit SCA. Die dafür verwendeten Tools nutzen verschiedene Quellen, z. B. die CVE-Datenbank [3] oder GitHub Advisories [4]. Kommerzielle Toolhersteller pflegen zusätzlich ihre eigenen Datenbanken mit Schwachstellen. Es stehen Tools für Anwendungen (Tabelle 1) und Infrastructure as Code (Tabelle 2) zur Verfügung.

Wie man mit den Ergebnissen umgeht

Die Aktualisierung einer betroffenen Komponente auf eine neuere Version ist häufig der einfachste und beste Weg, eine gefundene Schwachstelle zu beseitigen. Manchmal ist das aber nicht möglich, weil zum Beispiel die Schwachstelle in der Komponente noch nicht gefixt ist oder weil die Komponente eine transitive Abhängigkeit ist. In diesen Fällen gibt es mehrere Möglichkeiten, mit der Schwachstelle umzugehen:

Programmiersprache	Werkzeug	Link
Java u. a.	OWASP Dependency Check	https://jeremylong.github.io/DependencyCheck
JavaScript	npm audit	https://docs.npmjs.com/cli/v7/commands/npm-audit
.NET	dotnet list package --vulnerable	https://devblogs.microsoft.com/nuget/how-to-scan-nuget-packages-for-security-vulnerabilities

Tabelle 1: SCA-Tools für Anwendungen

Infrastruktur	Werkzeug	Link
Docker Images	Trivy	https://github.com/aquasecurity/trivy
Docker Images	Grype	https://github.com/anchore/grype

Tabelle 2: SCA-Tools für Infrastructure as Code

Programmiersprache	Werkzeug	Link
Java	FindSecBugs	https://find-sec-bugs.github.io
JavaScript	ESLint	https://eslint.org
.NET	Security Code Scan	https://security-code-scan.github.io
Verschiedene	Semgrep	https://semgrep.dev
Verschiedene	GitLab SAST	https://docs.gitlab.com/ce/user/application_security/sast

Tabelle 3: SAST-Tools für Programmiersprachen

Infrastruktur	Werkzeug	Link
Verschiedene	Checkov	https://www.checkov.io
Verschiedene	KICS	https://kics.io
Verschiedene	Terrascan	https://docs.accurics.com/projects/accurics-terrascan/en/latest
Terraform	tfsec	https://tfsec.dev
Kubernetes	Kubesec	https://kubesec.io

Tabelle 4: SAST-Tools für Infrastructure as Code

Werkzeug	Link
detect-secrets	https://github.com/Yelp/detect-secrets
truffleHog3	https://github.com/feeltheajf/trufflehog3
Gitleaks	https://github.com/zricethezav/gitleaks

Tabelle 5: Werkzeuge für Secrets Detection

1. Sicherstellen, dass die anfällige Funktion der Komponente nicht im System verwendet wird.
2. Einkapseln der anfälligen Funktion der Komponente im eigenen Code, damit die Schwachstelle nicht mehr von Angreifern genutzt werden kann.
3. Explizite Akzeptanz des Risikos.

Static Application Security Testing (SAST)

Programmierer können unbewusst Schwachstellen in ihren Code einbauen. Durch ein manuell gebautes SQL für eine Performanceoptimierung kann schnell eine Angriffsfläche für SQL Injections entstehen. Werden Ein- und Ausgaben in Weboberflächen nicht sorgfältig geprüft, kann ein Einfallstor für Cross-Site Scripting entstehen. Oder sensitive Daten sind durch falsche Konfiguration der Kryptografie nur schwach verschlüsselt. Mit regelbasierten Suchen lassen sich viele dieser Probleme erkennen. Es existieren Werkzeuge für alle gängigen Programmiersprachen (Tabelle 3) und Infrastructure as Code (Tabelle 4).

Wie man mit den Ergebnissen umgeht

Wenn eine gefundene Schwachstelle nicht als False-Positive-Meldung eingestuft wird, ist es immer sinnvoll, den Source Code entsprechend zu ändern, um die Schwachstelle zu entfernen. Die Werkzeuge geben dabei häufig Hilfestellung, indem sie Beispiele für eine korrekte Implementierung enthalten.

Secrets Detection (SD)

Geheimnisse wie Passwörter oder API-Keys sind schnell einmal im Code-Repository eingecheckt. Beispiele dafür sind Informationen, die im Code hart verdrahtet wurden oder ein versehentlicher Commit einer Konfigurationsdatei. Wenn das Code-Repository nicht sehr strikt beschränkt, wer darin lesenden Zugriff hat, können dadurch möglicherweise weitreichende Angriffsstellen entstehen, mit denen z. B. Unbefugte direkt auf Datenbanken oder Schnittstellen zugreifen können. Solche Daten dürfen also nicht mit dem Code in Repositories eingecheckt werden, und es gibt Werkzeuge, die z. B. Git Repositories über die gesamte Versionshistorie nach solchen Geheimnissen durchsuchen (Tabelle 5).

Wie man mit den Ergebnissen umgeht

Es reicht nicht aus, das Geheimnis aus dem Code oder der Konfigurationsdatei zu entfernen und eine neue Version in das Repository einzuchecken. Da die Information weiterhin in den vorherigen Versionen sichtbar ist, müssen die Passwörter oder API-Keys selbst in den Konfigurationen geändert werden. Das kann viel Aufwand bedeuten, ist aber der einzige Weg, die Systeme in diesen Fällen sauber abzusichern.

Infrastrukturtests

Die vorher beschriebenen Tests werden in die CI/CD-Pipeline eingebunden und liefern Ergebnisse direkt im Entwicklungsprozess. Es kann aber auch die installierte Infrastruktur selbst geprüft werden. Die eingebauten Werkzeuge der großen Cloud-Provider sowie Open-Source-Tools erzeugen eine Innensicht auf mögliche Schwachstellen in der Konfiguration (Tabelle 6).

Wie man mit den Ergebnissen umgeht

Wie auch bei den anderen Prüfungen muss zunächst eine Bewertung erfolgen, wie relevant die Meldung im aktuellen Kontext ist. Eventuell soll eine im Internet zu erreichende Ressource tatsächlich öffentlich zu erreichen sein, oder es ist eine Fehlkonfiguration, die behoben werden muss. Wenn die Einrichtung der Umgebung mit Infrastructure as Code geschieht, können bereits in einer Entwicklungs- oder Testumgebung viele Probleme gefunden werden, was dann zu einer sicheren Einrichtung der Produktionsumgebung führt. Dennoch

Tabelle 6: Tools für Infrastrukturtests

Infrastruktur	Werkzeug	Link
AWS	Prowler	https://github.com/toniblyx/prowler
AWS	Security Hub	https://aws.amazon.com/security-hub/
Azure	Security Center	https://azure.microsoft.com/services/security-center/
Kubernetes	kube-bench	https://github.com/aquasecurity/kube-bench
Kubernetes	kube-hunter	https://github.com/aquasecurity/kube-hunter

sollten in allen Fällen die Schwachstellentests auch auf der produktiven Umgebung durchgeführt werden.

Grenzen und Fallstricke

Mit den hier vorgestellten Vorgehensweisen und Werkzeugen kann man die Sicherheit eines Softwaresystems stark verbessern. Man muss sich aber auch der Grenzen und Fallstricke bewusst sein.

Sicherheit im Design

Auch wenn die genannten Werkzeuge ein integraler Bestandteil der Entwicklung sein sollten, fängt Sicherheit dennoch früher im Entwicklungsprozess an. Die Anforderungen an Sicherheit müssen klar definiert sein. Um diese zu ermitteln, helfen Schutzbedarfs- und Bedrohungsanalysen und daraus abgeleitete Risiken. Die Sicherheitsanforderungen werden in den User Stories und der Definition of Done verankert und sind damit auch Bestandteil der Architektur des Systems.

Automatisierte Prüfungen vs. Sicherheitstests

Die automatisierten Prüfungen können nur einen Teil der Schwachstellen erkennen. Wenn die Sicherheitsanforderungen aber in den User Stories und der Definition of Done beschrieben sind, wird es ein natürlicher Prozess, diese Anforderungen auch mit den verschiedenen im Projekt etablierten Testmethoden zu überprüfen, von Unit-Tests über Integrationstests bis Ende-zu-Ende-Tests.

Auch Penetrationstests haben weiterhin ihre Berechtigung. Mit einer Mischung aus automatisierter Schwachstellenerkennung und expliziten Tests der Sicherheitsanforderungen verlieren Penetrationstests aber ihren Schrecken und werden mehr zu einer Bestätigung, dass die Maßnahmen für sicheres Design und Entwicklung funktionieren.

Listing 1

```
plugins {
  ...
  id "org.owasp.dependencycheck" version "6.5.0.1"
  id "com.github.spotbugs" version "4.7.9"
}

...

dependencies {
  ...
  spotbugsPlugins 'com.h3xstream.findsecbugs:findsecbugs-plugin:1.11.0'
}

dependencyCheck {
  format='ALL'
}

spotbugs {
  ignoreFailures = true
}
```

Auswahl der Tools

Die in den vorherigen Kapiteln aufgezählten Tools sind nur eine kleine Auswahl. Es gibt viele weitere Tools, sowohl als Open Source als auch kommerzielle Produkte. Open-Source-Tools können schon gute Ergebnisse liefern, kommerzielle Produkte haben jedoch häufig bessere Analysemethoden und Managementoberflächen, für die man allerdings auch bezahlen muss.

Listing 2

```
variables:
  DD_PRODUCT_TYPE_NAME: "Research and Development"
  DD_PRODUCT_NAME: "Example Service"
  DD_ENGAGEMENT_NAME: "GitLab"

stages:
  - check
  - import

checkVulnerabilities:
  stage: check
  image: openjdk:11-jdk
  script:
    - chmod u+x gradlew
    - ./gradlew dependencyCheckAnalyze spotbugsMain
  artifacts:
    paths:
     - build/reports/
    when: always
    expire_in: 1 day

import_findsecbugs:
  stage: import
  image: maibornwolff/dd-import:1.0.3
  needs:
    - job: checkVulnerabilities
      artifacts: true
  variables:
    GIT_STRATEGY: none
    DD_TEST_NAME: "FindSecBugs"
    DD_TEST_TYPE_NAME: "SpotBugs Scan"
    DD_FILE_NAME: "build/reports/spotbugs/main.xml"
  script:
    - /usr/local/dd-import/bin/dd-reimport-findings.sh

import_dependency_check:
  stage: import
  image: maibornwolff/dd-import:1.0.3
  needs:
    - job: checkVulnerabilities
      artifacts: true
  variables:
    GIT_STRATEGY: none
    DD_TEST_NAME: "Dependency Check"
    DD_TEST_TYPE_NAME: "Dependency Check Scan"
    DD_FILE_NAME: "build/reports/dependency-check-report.xml"
  script:
    - /usr/local/dd-import/bin/dd-reimport-findings.sh
```

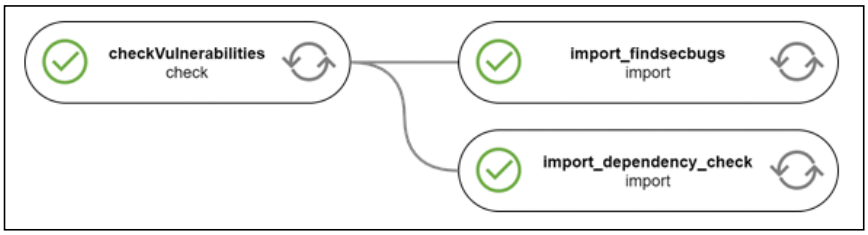

Abb. 2: Pipeline mit Schwachstellentests

Wie viel Aufwand man in die Auswahl von Tools investiert und ob kommerzielle Produkte notwendig sind, hängt letztendlich von den Sicherheitsanforderungen ab. Für niedrige bis mittlere Anforderungen wird ein kurzer Auswahlprozess von Open-Source-Tools vollkommen ausreichen, damit kann man mit wenig Investition die Sicherheit deutlich verbessern. Bei höheren Sicherheitsanforderungen, oder wenn man nicht für ein einzelnes Projekt, sondern auf Unternehmensebene denkt, ist auch ein höherer Aufwand für die Werkzeugauswahl und die Investition in kommerzielle Produkte gerechtfertigt.

False Positives

False Positives sind gemeldete Schwachstellen, die im aktuellen Kontext keine Beeinträchtigung der Sicherheit darstellen. Insbesondere die statischen Codeanalysen neigen dazu, viele False-Positive-Ergebnisse zu produzieren, aber auch bei den anderen Klassen der Werkzeuge treten sie auf. Bei der Konfiguration der Tools gibt es zwei Möglichkeiten, ihre Anzahl zu reduzieren:

1. Ausschluss von Code, der zur Laufzeit nicht ausgeführt wird (z. B. Testcode oder Eingaben für die Codegenerierung) und eingebundenem Code wie JavaScript-Bibliotheken (z. B. jQuery oder Bootstrap)
2. Anpassung der Regelsätze der Scan-Engine; man kann zum einen Regeln für Bedingungen deaktivieren, die im aktuellen Kontext als sicher gelten können, oder bestehende Regeln ändern, damit sie besser zum Code passen

Wird DefectDojo zum Management der Schwachstellen eingesetzt, können die False Positives auch dort markiert werden. Damit entfällt ihre Behandlung für die Zukunft.

Bewertung und Priorisierung

Idealerweise wird man die Prüfungen zum Start des Projekts in die CI/CD-Pipeline einbauen. So kann man gefundene Probleme schnell beseitigen und es entstehen gar nicht erst lange Listen an abzuarbeitenden Problemen. Werden die Prüfungen erst aufgesetzt, wenn die Codebasis schon größer ist, erhält man typischerweise zunächst einmal eine längere Liste von gefundenen Schwachstellen. Diese müssen nach Relevanz bewertet und priorisiert werden. Dabei helfen die von den Werkzeugen vergebenen Schweregrade, sodass man sich von kritischen Meldungen zu den weniger schwerwiegenden durcharbeiten kann. Diese Schritte stellen einen Initialaufwand für das Projekt dar, der nicht unterschätzt werden darf und eingeplant werden muss.

Praktisches Beispiel

Wie sieht das Ganze in der Praxis aus? Eine kurze GitLab Pipeline zeigt, wie man eine Analyse der Abhängigkeiten und eine statische Codeanalyse mit wenig Aufwand einbinden kann.

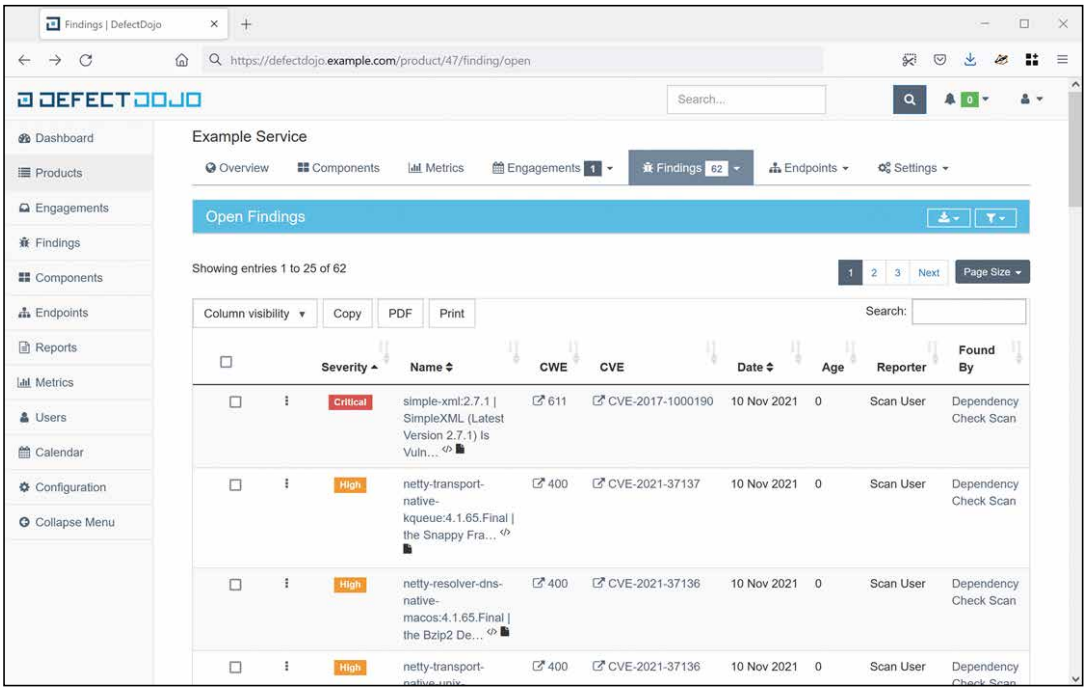

Abb. 3: Verwaltung der Schwachstellen mit DefectDojo

Unser Beispiel-Service ist eine Spring-Boot-Anwendung, die mit Gradle gebaut wird. In der Datei *build.gradle* müssen wir nur einige zusätzliche Zeilen einfügen, um den OWASP Dependency Check für die Analyse der Abhängigkeiten und FindSecBugs für die statische Codeanalyse aufzurufen (Listing 1).

In unserer GitLab CI/CD Pipeline führen wir drei zusätzliche Schritte ein, die in **Abbildung 2** zu sehen sind:

1. *checkVulnerabilities* führt sowohl die Prüfung der Abhängigkeiten als auch die statische Codeanalyse durch. Diese Prüfungen könnte man natürlich auch direkt beim Bauen der Anwendung durch Gradle aufrufen lassen. Die Berichte werden im Ordner *build\reports* abgelegt, dessen Inhalte als Artefakte für die nachfolgenden Schritte gespeichert werden.
2. *import_findsecbugs* und *import_dependency_check* importieren die gefundenen Schwachstellen in DefectDojo. Dazu nutzen wir den dd-import-Wrapper [5], der die benötigten Konfigurationen in DefectDojo bei Bedarf selbst anlegt und die API-Parameter als Umgebungsvariablen abfragt (Listing 2), was sehr gut zu der Definition von CI/CD Pipelines passt.

Damit stehen alle Ergebnisse in DefectDojo zur Auswertung und zur weiteren Behandlung bereit (**Abb. 3**). Typische Maßnahmen sind:

- Die Auffälligkeit bleibt offen und soll im Code oder durch Aktualisierung einer Bibliothek beseitigt werden.
- Die Auffälligkeit ist ein False Positive und kann geschlossen werden.
- Die Auffälligkeit wird durch andere Maßnahmen mitigiert und kann geschlossen werden.
- Das durch die Auffälligkeit entstandene Risiko wird akzeptiert.

Diese Entscheidungen bleiben stabil, wenn bei weiteren Läufen der Pipeline die Ergebnisse neu importiert werden, d. h. geschlossene Auffälligkeiten bleiben mit ihrer Begründung geschlossen. Zusätzlich werden Auffälligkeiten automatisch geschlossen, wenn sie beispielsweise durch eine Änderung nicht mehr im Code enthalten sind und somit keine Meldung mehr entsteht. So reduziert sich die Menge der offenen Auffälligkeiten mit der Zeit und reflektiert, dass die Sicherheit des Systems wächst.

Fazit

Automatisierte Schwachstellentests können gut in bestehende CI/CD Pipelines integriert werden und helfen, frühzeitig mögliche Sicherheitsprobleme zu vermeiden. Initialaufwände entstehen bei Auswahl und Konfiguration der Tools und bei ersten Prüfungen. Diese Aufwände schwingen sich aber schnell ein und das Reagieren auf Schwachstellen wird Teil der laufenden Entwicklung.

Anwendungen, die geschäftskritische Vorgänge abwickeln oder sensible Daten verwalten, sollten Schwachstellentests der hier vorgestellten Kategorien einsetzen. Mit weiteren sicherheitsrelevanten Maßnahmen wie Architektur- und Codereviews, funktionalen Sicherheitstests und Penetrationstests ergibt sich eine optimale Absicherung der Anwendung gegenüber Angriffsversuchen.

Stefan Fleckenstein ist Bereichsleiter Cybersecurity bei MaibornWolff. Nach mehr als 20 Jahren in der Softwareentwicklung wurde Informationssicherheit ein Hauptthema seiner Arbeit. Weitere Schwerpunkte seiner Arbeit sind Secure-Software-Development-Lifecycles, Securityaudits von Software-Entwicklungsprojekten und Vulnerability-Management. In seiner Freizeit ist Stefan Fleckenstein Moderator und Entwickler für das Open-Source-Vulnerability-Management-Tool DefectDojo.

www.defectdojo.org

Links & Literatur

[1] https://owasp.org/Top10

[2] https://www.defectdojo.org

[3] https://www.cve.org

[4] https://github.com/advisories

[5] https://github.com/MaibornWolff/dd-import

Mit Team Programming echte Zusammenarbeit erreichen

Gemeinsam sind wir stark

Seit einigen Jahren macht eine Programmierpraktik von sich reden, bei der die Entwickler:innen nicht separat Aufgaben lösen, sondern das gesamte Team gemeinsam an einer Lösung arbeitet. Auf diese Weise versteht mehr als eine Person den Code und kann (und mag) ihn später auch pflegen. Diese Art der Zusammenarbeit bringt außerdem andere positive Effekte mit sich, etwa beim Teambuilding. Erfahrungen aus fünf Jahren Coaching zu Team Programming, vor Ort und remote.

von Thomas Much

Als vor über zwei Jahrzehnten das Extreme Programming mit seinen Ideen und Programmierpraktiken aufkam, war das Pair Programming revolutionär – zwei Entwickler:innen arbeiten zusammen an einer Aufgabe, die sonst eine:r allein gelöst hat. „Endlich Zusammenarbeit!“, jubelten die einen. „Was für eine Verschwendung“, fluchten die anderen. Und zwar nicht nur das Management, das auf Effizienz und Optimierungen achtet, sondern auch Entwickler:innen, die vielleicht lieber allein arbeiten.

Mit der Zeit entwickelten sich die Programmierpraktiken weiter, und seit einigen Jahren wird nun das Team Programming (auch bekannt als Mob oder Ensemble Programming) propagiert, bei dem das ganze Team zusammen eine Aufgabe löst. Das mag verrückt klingen, funktioniert aber erstaunlich gut – und bügelt ein paar Schwachstellen aus, die viele Entwickler:innen beim Pair Programming empfinden. Zudem ergeben sich einige positive Wirkungen fast nebenbei, die man sonst mühsam zu erreichen versucht. Sei es, dass eine Gruppe von Entwickler:innen überhaupt als Team zusammenfindet, sei es, dass das Team lernt, Konflikte konstruktiv auszutragen und gemeinsame Konventionen zu entwickeln.

Dabei gibt es kein richtig oder falsch. Beim Team Programming geht es vor allem darum, dass das Team besser zusammenarbeitet, was auch immer das für das jeweilige Team bedeutet. Das ist absolut wichtig, aber es haben sich ein paar Tipps herauskristallisiert, wie man an das Team Programming herangehen sollte, damit es sich möglichst bald gut anfühlt (was wichtig für die Akzeptanz ist) und man es mit Mehrwert einsetzen kann. Anschließend kann man damit experimentieren und es Schritt für Schritt für das aktuelle Team anpassen.

Ein bisschen Geschichte

1999 beschrieb Kent Beck die Werte, Prinzipien und Praktiken des Extreme Programming, darunter Pair Programming. Spätestens 2003 kam die Idee auf, Pair Programming mit mehr als zwei Beteiligten durchzuführen. Allerdings fand dieser Ansatz kaum Beachtung – vielleicht, weil schon die Vorstellung, zwei Entwickler:innen mit einer gemeinsamen Aufgabe zu betreuen, noch zu neu war. Vielleicht auch, weil das Vorgehen dieser Zusammenarbeit noch nicht so gut beschrieben war, dass sich ein Mehrwert erkennen bzw. erleben ließ.

2011 überlegte Woody Zuill mit seinem Team bei Hunter Industries, wie sich ein Projekt, das unterbrochen worden war, neu aufsetzen ließe. Aufbauend auf TDD, Coding Dojos und Pair Programming lernte das Team, konsequent, intensiv und direkt zusammenzuarbeiten und sich ständig in kleinen Schritten zu verbessern. Ganz nebenbei entstand so das, was das Team „Mob Programming“ nannte.

Nachdem Woody Zuill 2014 „Mob Programming – A Whole Team Approach“ auf einer Konferenz vorgestellt

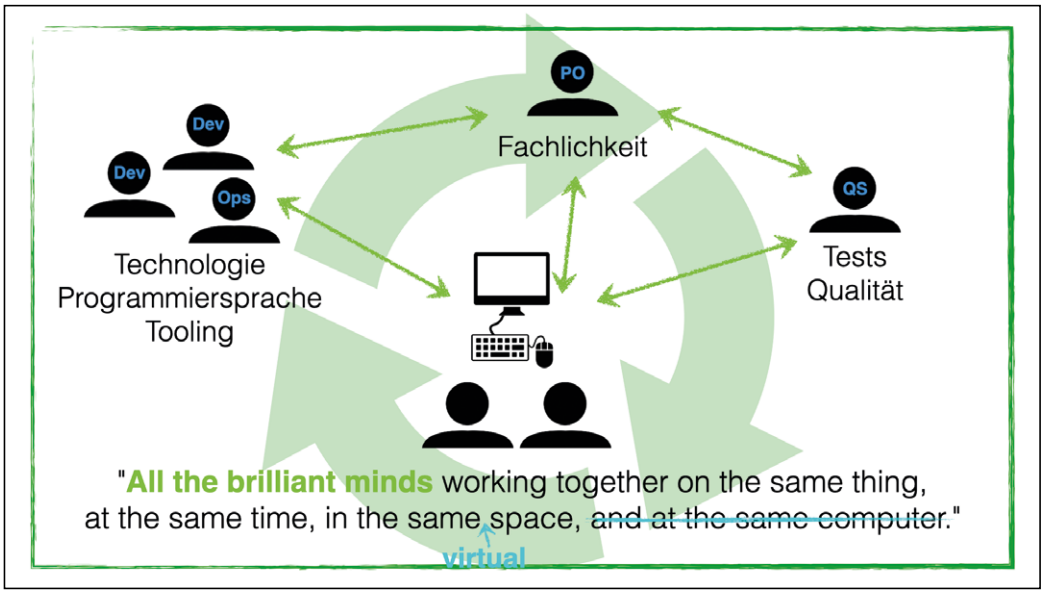

Abb. 1: Die Grundidee des Team Programming – alle relevanten Personen arbeiten gleichzeitig zusammen

hatte [1], wurde es von immer mehr Teams und Firmen eingesetzt und angepasst, beispielsweise für die Remotearbeit.

2016 bekam ich irgendwo im Netz diese neue Idee mit. Und weil ich damals schon länger als Pair-Programming-Coach unterwegs war, suchte ich (erfolgreich!) nach einem Team, das Mob Programming mit mir ausprobieren würde. Seitdem ist Mob bzw. Team Programming meine bevorzugte Art, Teams an echte Zusammenarbeit heranzuführen, weil es mit ein bisschen Moderation meiner Erfahrung nach leichter erlernt werden kann als Pair Programming. Und das profitiert auch oft von funktionierendem Team Programming – letztlich ist ein Pair ja auch ein Team, wenn auch ein kleines.

Was ist Team Programming?

Die Idee des Team Programming ist, dass alle, die wir zu Umsetzung einer Aufgabe benötigen, gleichzeitig und gemeinsam daran arbeiten. Dadurch wird das Warten auf Wissensträger:innen, Antwortgeber:innen, die Bearbeitung eines Pull Requests etc. vermieden. Und anstatt viele verschiedene Aufgaben anzufangen, haben wir plötzlich die Möglichkeit, die wichtigste (am höchsten priorisierte) Aufgabe zügig fertig zu bekommen. Woody Zuill nennt diese Art der Zusammenarbeit „all the brilliant people working on the same thing, at the same time, in the same space, and on the same computer“ (all die großartigen Menschen arbeiten an derselben Sache, zur selben Zeit, im selben Raum und am selben Computer). Und man sieht bereits in **Abbildung 1**, dass daran nicht nur Entwickler:innen beteiligt sind, sondern durchaus auch Product Owner (PO), Qualitätsspezialist:innen (QS) und andere Rollen.

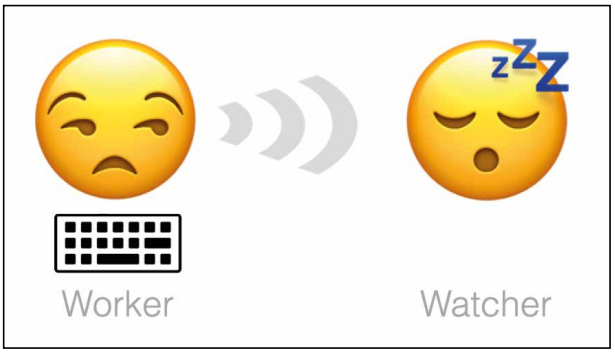

Abb. 2: Wenn der Navigator zum Zuschauer verkommt – das Driver-Observer-Antipattern

Der ursprüngliche Name „Mob“ war wohl eher als Witz des Teams gedacht, das einen lustigen Namen für sich gesucht hatte. „Mob Programming“ wird aber leider häufig zu „Mobbing“ abgekürzt, was nicht nur im Deutschen ein extrem negativer Begriff ist, sondern auch im Englischen (auch wenn es dort mit „harrassment“ und „bullying“ diverse andere Begriffe für Mobbing gibt).

Weltweit wurde und wird daher nach besseren Namen gesucht. „Ensemble Programming“ taucht derzeit häufiger auf. Woody Zuill nannte diese Praktik schon recht früh „Whole Team Programming“. Gelegentlich kürzt er es zu „Team Programming“ ab, weshalb ich hier diese Bezeichnung nutze. Aber der Name ist nur Schall und Rauch, es kommt auf andere Dinge an.

Worauf es ankommt

Grundlegend ist die aktive Beteiligung aller Teilnehmer:innen an der Team-Programming-Session, was nicht nur die Aufmerksamkeit, sondern auch die Kreativität fördert. Das stellen wir folgendermaßen sicher:

- Der Platz an der Tastatur wird regelmäßig gewechselt.
- An der Tastatur programmiert man nicht selbst aktiv, sondern lässt sich vom Rest anleiten.
- Es sollten nicht zu wenige und vor allem nicht zu viele Teilnehmer:innen sein.
- Das Team reflektiert regelmäßig seine Arbeitsweise.

Warum ist das alles wichtig? Wer schon einmal Teil eines Pairs war, bei dem eine:r die ganze Zeit redet, tippt und sich an die Tastatur „klammert“, während

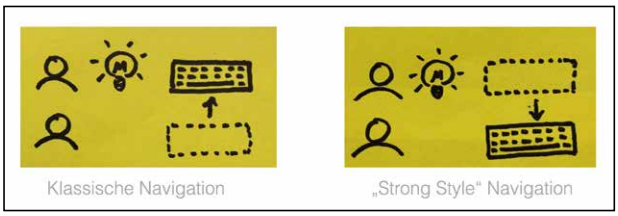

Abb. 3: Klassische Navigation (aktiver Driver) vs. Strong Style Navigation

der/die andere im Wesentlichen nur zuschauen darf, weiß, wie zäh und ermüdend der Versuch sein kann, zusammenzuarbeiten. Und für die „Vortragende" ist es irgendwann auch frustrierend, weil man merkt, dass der andere nicht mehr bei der Sache ist. Hier funktionieren die beiden Rollen „Driver" und „Navigator" des Pair Programming vermutlich nicht gut, denn es fehlt Abwechslung und die echte Beteiligung beider Entwickler:innen. Man nennt diese Variante, die man meiner Erfahrung nach leider erstaunlich häufig antrifft, das „Driver-Observer-Antipattern" oder „Worker-Watcher-Antipattern" (**Abb. 2**). Kurzzeitig mag das helfen, weil es vielleicht schon eine Verbesserung des Miteinanders darstellt. Aber mittelfristig ist es oft Zeitverschwendung, weil das Vorgeführte am „Watcher" oft nur vorbeirauscht und nicht wirklich aufgenommen oder gelernt wird. Das heißt, ein regelmäßiger Tausch der Rollen zwischen den beteiligten Entwickler:innen, das Weitergeben der Tastatur, ist entscheidend – auch beim Team Programming.

Daher nutzen wir dafür feste, kurze Zeitfenster (Timeboxes), nach denen der Platz an der Tastatur weitergegeben wird. Bei einer Session vor Ort startet man am besten mit fünf bis maximal zehn Minuten. Bei Remote-Sessions sind die Zeitfenster eher ein bisschen länger (10–15 Minuten), weil das Weitergeben der virtuellen „Tastatur" etwas länger dauert (dazu später mehr). Weil es sehr menschlich ist, dass man die Zeit vergisst, gerade bei so kurzen Zeitblöcken, sollten wir uns per Timer an den Wechsel erinnern lassen. Das funktioniert zudem prima als spielerische Ermahnung für diejenigen, die gerne länger als abgesprochen an der Tastatur festhalten wollen. Als Timer kann man vor Ort problemlos sein Smartphone nutzen. Hilfreich ist es allerdings, wenn der Timer für alle sichtbar ist. Das ermöglichen Webtimer, die man einfach im Browser auf einem separaten Bildschirm laufen lassen kann [2], [3] und [4]. Bei Remote-Sessions wird der URL des Webtimers in den Session-Chat geschrieben, sodass jede:r Teilnehmer:in den Timer auf dem eigenen Gerät laufen lassen kann.

Zeitfenster von fünf oder zehn Minuten klingen vielleicht lächerlich kurz. Wer jetzt denkt: „In der kurzen Zeit schaffe ich doch gar nichts an der Tastatur ..." – hat recht! Denn darauf kommt es gar nicht an, und das ist ein weiteres wichtiges Kriterium für gelungenes Team Programming. Die meisten von uns sind darauf konditioniert, möglichst schnell viel Code zu tippen. Das Problem dabei: Für die, die beobachten, ist das oft zu schnell, um den Code wirklich zu verstehen. Einen solchen Driver kann man schlecht navigieren, weil man Probleme hat, den schnellen Eingaben und Sprüngen zwischen Dateien und Fenstern zu folgen.

Vor einigen Jahren hat Llewellyn Falco daher die Strong Style Navigation beschrieben, ursprünglich noch für das Pair Programming: Jede Idee muss, um zu Code zu werden, durch fremde Hände gehen. Wenn ich eine Idee habe, tippe ich also nicht mehr selbst. Sondern wenn ich eine Idee habe, nimmst du die Tastatur und ich erkläre dir meine Idee so gut, dass du sie tippen kannst. **Abbildung 3** zeigt den Unterschied.

Die Person an der Tastatur wird also nicht selbst aktiv, sondern lässt sich vom Rest anleiten. Das wird zu Anfang oft noch sehr grundlegend sein, manchmal geht es um einzelne Zeichen oder Tastaturkürzel. Bald, wenn das Team etwas länger zusammengearbeitet hat, wird diese Navigation auf einem höheren, abstrakteren Niveau durchgeführt. Diese Vorgehensweise nimmt etwas Geschwindigkeit aus dem Coding, genauer: Die Geschwindigkeit wird an die Zuschauenden bzw. Lernenden angepasst, die plötzlich zu den Steuernden werden. Aus Gehetze wird so mit etwas Übung ein zügiger Marsch des gesamten Teams.

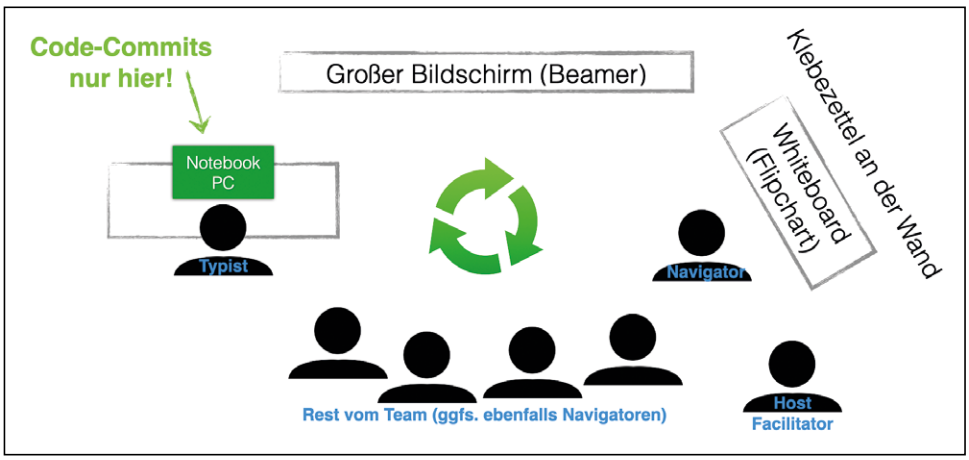

Abb. 4: Typisches Set-up einer Team-Programming-Session vor Ort

Die Rolle des Drivers nennt man beim Strong Style daher auch gern „Typist" (Protokollant), der die Ideen des Teams in Code tippt. Programmiert bzw. entwickelt wird die Software dabei letztlich von den Navigatoren, also den übrigen, nicht tippenden Teilnehmer:innen. Strong Style Navigation ist der Standard beim Team Programming. Gerade am Anfang soll-

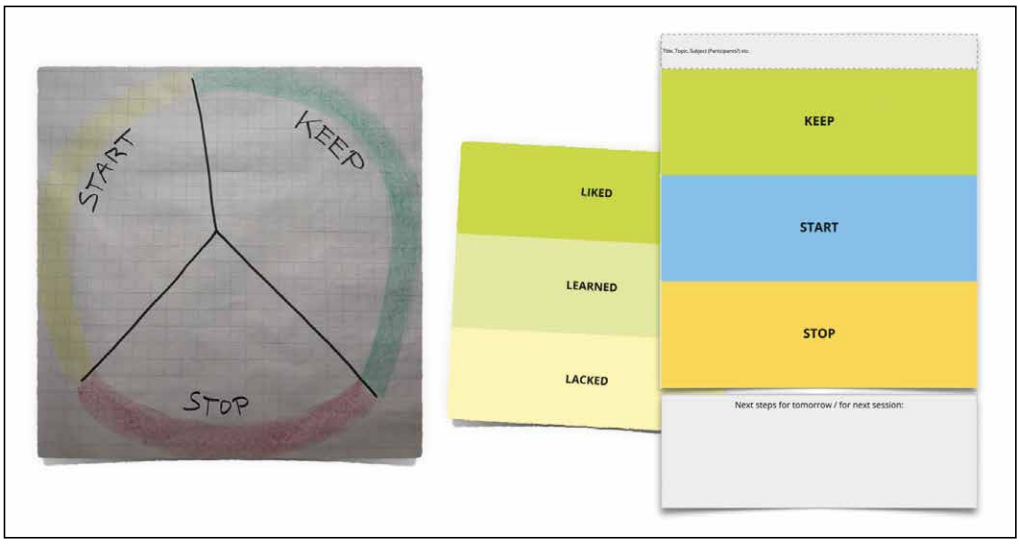

Abb. 5: Mögliche Vorlagen für Mini-Retros, am Flipchart vor Ort oder auf einem Online-Whiteboard

te man das sehr ernst nehmen – und auch später nur aus gutem Grund davon abweichen. Ein typisches Set-up des Team Programming für ein so zusammenarbeitendes Team zeigt **Abbildung 4**.

Letztlich entscheidet auch die Anzahl der Teilnehmer:innen darüber, wie gut die Session abläuft. Als ideal hat sich eine Größe von vier bis sechs Personen erwiesen. Darunter tendiert die kleine Gruppe zu sehr zum Groupthink (Gruppendenken), d. h., man trifft tendenziell schlechtere Entscheidungen, weil eine abweichende Meinung zu schnell untergeht. Bei mehr als sechs Personen wird es schnell zäh. Als Faustregel habe ich bei vielen Teams erlebt, dass vor Ort jede:r Teilnehmer:in mindestens einmal pro halbe Stunde an der Tastatur sein sollte, remote mindestens einmal pro Stunde.

Team Programming wird also nicht immer vom gesamten Team durchgeführt, vor allem nicht bei großen Teams. Gruppen ab acht Mitgliedern verlieren schnell die Interaktion und die Beteiligung aller. Ich durfte auch schon Sessions mit zehn und mehr Teilnehmer:innen begleiten. Das ist vor Ort leichter als remote. So oder so sollten die Sessions dann aber nicht zu lang sein, weil vermutlich nicht alle an die Tastatur kommen werden. Eine so große Session eignet sich z. B. als Ersatz für eine einstündige Präsentation – aber nicht als reine Vorführung, sondern interaktiv.

Mit der idealen Anzahl Teilnehmer:innen von vier bis sechs dauert eine Team-Programming-Session meistens zwei Stunden oder mehr, damit sich die Rüstzeit lohnt (Zurechtfinden in der Aufgabe, Zusammenfinden der aktuellen Teilnehmer:innen etc.). Wenn die Aufgabe allen klar ist oder die meisten schon daran gearbeitet haben, kann man auch eine Stunde effektiv zusammen nutzen. Ansonsten wird das Team oft von sich aus länger gemeinsam programmieren wollen (einen halben oder ganzen Tag).

Am Ende einer Session (bzw. eines Tages) führt das Team eine Mini-Retro durch, um die Arbeitsweise jeden Tag Schritt für Schritt ein bisschen zu verbessern. So werden Probleme schnell erkannt und aus dem Weg geräumt, bevor sie Frust erzeugen. Vor allem kann das Team die positiven Aspekte erkennen und stärken. Dafür sammeln wir in einer Mini-Retro, was heute gut war und was wir in der nächsten Session anders oder neu ausprobieren wollen. Es gibt dafür kein festes Format, vielleicht wird die Retrospektive auch nur mündlich durchgeführt. Typische Fragen für ein Flipchart bzw. ein Online-Board zeigt **Abbildung 5**.

Kurz bedeutet, dass die Mini-Retro in der Regel fünf bis maximal fünfzehn Minuten dauert. Manchmal wird es aus gutem Grund länger dauern, manchmal ist vielleicht gar keine Retro erforderlich. Aber gerade am Anfang sollte man darauf achten, dass das Team regelmäßig darüber nachdenkt, was es an der eigenen Arbeitsweise stärken und was es anpassen möchte (z. B. die Timebox-Länge, die Werkzeuge, das Mobiliar bzw. den Raum o. Ä.).

Wenn niemand weiterweiß

Am besten ist es, wenn das Team für die ersten Team Programming Sessions Aufgaben auswählt, bei denen mindestens eine:r oder zwei aus dem Team wissen, wie man die Aufgabe löst, und somit die anderen anleiten können. Es fühlt sich gut an, wenn man nicht ständig alle Grundlagen in der Suchmaschine erfragen muss – der Einstieg ist deutlich leichter.

Aber auch für später ist das wichtig: Wenn niemand im Team die Kompetenzen hat, um die anstehende Aufgabe umzusetzen, laden wir die nötigen Kolleg:innen aus anderen Teams zu unserer Session ein. Wenn wir solche Kolleg:innen nicht finden, könnten einige wenige (zur Not eine:r) aus unserem Team mit einem Prototyp bzw. Spike die neue Technologie bzw. die Umsetzbarkeit der Aufgabe damit erforschen. In der Team Programming Session wird dann aber nicht einfach der fertige Prototyp vorgestellt bzw. dessen Code übernommen. Stattdessen beginnt das Team wieder ganz von vorn, aber mit dem Wissen aus dem Prototyp, d. h., dass das Team unter Anleitung die grundlegenden, wichtigen Schritte selbst erfahren kann.

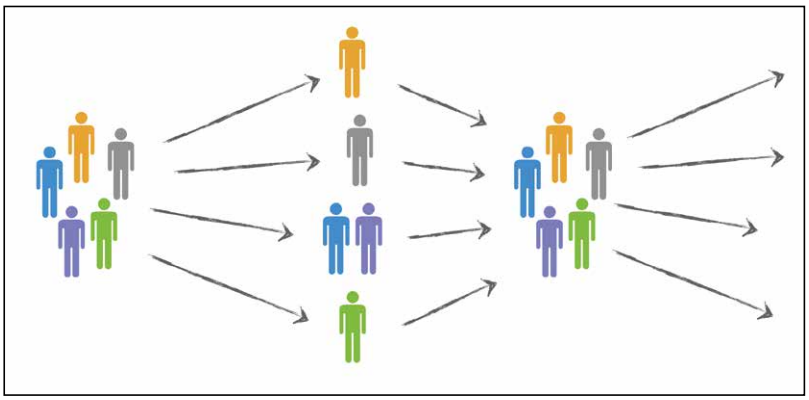

Abb. 6: Zusammen zu starten, ermöglicht sinnvolleres Aufteilen der Aufgabe und leichteres Zusammenbringen der Ergebnisse

Wenn man merkt, dass keine:r mehr weiterweiß und eine gemeinsame Recherche nicht in akzeptabler Zeit die nächsten möglichen Schritte aufzeigen kann, sollte man aufhören und die Team Programming Session vertagen. In der Zwischenzeit kann jede:r für sich ein bisschen recherchieren. Aber Achtung: Recherchieren heißt nicht fertigprogrammieren! Es hilft dem Team nichts, wenn jemand in der folgenden Session eine fertige Lösung präsentiert. Hinter dieser Einzellösung steht das Team dann im Zweifelsfall nicht.

Es ist vollkommen in Ordnung, mögliche Lösungsideen für sich allein auszuprobieren und die Aufgabe vielleicht sogar zu lösen. Am Anfang der nächsten Session teilt man diesen Erfolg dem Team mit – und leitet das Team dann an, das noch mal bei null startet. Das wird vielleicht nicht die „perfekte“, selbst gebastelte Lösung liefern, aber das Team versteht die ersten grundlegenden Schritte und kann bei der weiteren Umsetzung deutlich fundierter mitdiskutieren und -arbeiten. Nach dem Erleben der grundlegenden Schritte könnte sich das Team auch einfach die vollständigere Lösung gemeinsam anschauen, besser verstehen als zuvor – und gegebenenfalls dann doch übernehmen, nur jetzt mit der nötigen Überzeugung.

Worauf sollten wir noch achten?

Am Anfang jeder Team Programming Session ist es sinnvoll, dass sich alle, die dabei sind, Klarheit über die anstehende Aufgabe verschaffen. Wir gehen die Aufgabe (z. B. in Form einer User Story) gemeinsam durch, diskutieren, ob die Akzeptanzkriterien verständlich sind und planen dann den ersten Schritt bzw. die ersten Schritte. Vor allem setzen wir uns ein gemeinsames Ziel für diese Session. Das kann beispielsweise sein, dass wir das Set-up auf allen Rechnern der Teilnehmer:innen passend für die Aufgabe erweitern. Oder dass wir uns bewusst nur den ersten Schritt (und nicht auch den zweiten oder dritten) vornehmen, um schädlichen Stress zu vermeiden. Diese Abstimmung bzw. Planung sollte nur wenige Minuten dauern, idealerweise maximal ein oder zwei Zeitrahmen. Denn im Verlauf des Team Programming ändern sich die Ziele mit ziemlicher Sicherheit – wir müssen nur daran denken, sie immer wieder gemeinsam mit allen neu abzustimmen.

So eine Neujustierung der (Zwischen-)Ziele kann prima rund um die Pausen stattfinden. Bevor wir in eine Pause gehen, fragen wir: Wo stehen wir? Und vor allem: Was ist der nächste konkrete Schritt? Nach der Pause lohnt es sich, noch mal mit „Was ist nun der nächste, kleine Schritt?“ zu starten. Wenn das jemand anders als vor der Pause beantwortet, merkt das Team auch gleich, ob alle ein ähnliches Verständnis der Aufgabe haben.

Pausen machen wir meistens einmal pro Stunde, z. B. zehn Minuten. Aber damit kann und sollte das Team experimentieren, denn die Notwendigkeit und Dauer von Pausen wechselt mit der Aufgabe, der Anzahl der Teilnehmer:innen, der Remote- bzw. Vor-Ort-Situation etc. Wichtig ist nur, dass man die regelmäßigen, kurzen Pausen nicht vergisst – was beim Coden leicht passiert. Die Mittagspause sollte nicht die einzige Unterbrechung des Arbeitstages sein, wenn man so intensiv zusammenarbeitet. Eine Pause ist sinnvoll, bevor die Aufmerksamkeit zu sehr nachlässt. Nötigenfalls lassen wir uns per Timer daran erinnern.

Alternativ kann man eine ganz andere Möglichkeit für Pausen eröffnen – mit „dynamischem“ Team Programming. Wer nicht gerade an der Tastatur sitzt und nicht den aktivsten Part der Diskussion trägt, kann die Session jederzeit für eine Pause verlassen. Es sind ja noch genügend andere anwesend, die den Typisten oder die Typistin navigieren können. Und das klappt nicht nur bei Pausen, sondern auch mit längeren Terminen, die eine einzelne Person wahrnehmen muss (Meetings etc.). Nach dem Termin kehrt man zur Team-Session zurück, schaut den Rest des laufenden Zeitrahmens zu – und sitzt danach an der Tastatur und lässt sich anleiten. So steigt man schnell wieder in die Session ein, ohne viel zu fragen und ohne dass die anderen alles noch einmal erklären müssen.

Man kann Team Programming für alle Aufgaben, d. h. zu 100 Prozent anwenden. Es gibt weltweit Teams, die es mit Erfolg einsetzen. Aber das ist gar nicht nötig. Die positiven Effekte von Team Programming zeigen sich schon, wenn man auch nur wenige Male pro Woche als Team zusammen entwickelt. Viel wichtiger ist das gemeinsame Starten. Wenn die Umsetzung einer Aufgabe beginnt, werden oft die Grundlagen des Designs, der Konfiguration etc. gelegt – und es tauchen oft die spannendsten Entscheidungen auf. Wenn das Team diese Grundlagen gemeinsam aufbaut, ist es viel leichter, sich bei Bedarf aufzuteilen, in mehrere kleine Team-Sessions, Pairs oder als einzelne Entwickler:innen. Wenn das gesamte Team später wieder zusammenkommt, sind die Entwicklungen, die inzwischen passiert sind, viel leichter zu verstehen, weil alle den Anfang mitbekommen haben (**Abb. 6**).

Manchmal ist es schwierig, auf die eigentliche Aufgabe fokussiert zu bleiben, wenn alle plötzlich viele kleine, gute Ideen einbringen. Um sich nicht zu verzetteln und unsere Köpfe freizubekommen („Wenn wir das nicht jetzt machen, vergessen wir es"), schreiben wir unsere vielen Ideen gern auf. Zunächst sehr unstrukturiert, nach und nach entwickeln sich eine Struktur (z. B. nächste Schritte, in dieser Aufgabe nicht vergessen, Ideen für andere Stories etc.) und eine Priorisierung. Wichtig ist, dass wir den nächsten konkreten Schritt, den wir umsetzen wollen, für alle sichtbar hingeschrieben haben. Manchmal machen wir das mit Zetteln auf einem (Online-)Whiteboard. In letzter Zeit nutzen wir zunehmend einfach eine To-do-Textdatei, die auf oberster Ebene des Projekts mit eingecheckt und bei Beendigung der Aufgabe aufgeräumt und gelöscht wird. Damit merken wir uns alle Ideen strukturiert, aber formlos, und vermeiden somit unnötige Bürokratie, die durch das Anlegen von Unteraufgaben im Ticketsystem entstünde.

Wenn das Team gelernt hat, in der Strong Style Navigation zu arbeiten, sich also die Typist:in von den anderen leiten lässt, ermöglicht es diese Arbeitsweise auch, den PO, eine UX-Kolleg:in oder die Teamleitung dazuzunehmen. Das klingt und ist erst einmal ungewohnt, kann aber eine tolle Dynamik in die Session bringen. Wenn die Entwickler:innen das zu lösende Problem jemandem erklären, von dem sie denken, dass er oder sie nicht programmieren kann, habe ich mehr als einmal erlebt, dass sie sich gegenseitig plötzlich sehr gute Einblicke und Hintergrundinformationen gegeben haben, die zuvor als selbstverständlich angenommen worden waren. Am besten startet man aber zunächst mit den Entwickler:innen und übt den Ablauf, vor allem, wenn die Aufgaben eher Coding-lastig sind. Wenn das halbwegs rundläuft, kann das Team problemlos ein oder zwei Wenig- oder Nicht-Coder dazunehmen. Diese sollten bei Coding-Aufgaben aber besser in der Minderheit sein, oder das Team gibt der Session bewusst einen anderen Schwerpunkt bzw. ein anderes Ziel.

Warum funktioniert das?

Durch Team Programming kann ich anderen beim Denken und Machen zuschauen. Implizites Wissen und Können werden explizit gemacht. Als Typist:in kann ich es selbst unter Anleitung ausprobieren. Und dann kann ich versuchen, andere anzuleiten, es sind ja genügend Kolleg:innen in der Session dabei, die mein Anleiten verbessern können. So entsteht eigenes Wissen und vor allem Können.

Das ist erst einmal ungewohnt und löst oft Ängste aus: „Ich muss an die Tastatur, während alle mir zuschauen." Wenn man aber noch einmal verdeutlicht, dass die Tastatur auch schnell wieder bei der nächsten Person ist, trauen sich dann doch die meisten, mitzumachen. Und im Netz finden sich diverse Berichte, dass gerade dieses ungewohnte Format auch für eher introvertierte Entwickler:innen nach einer Eingewöhnung sehr bereichernd ist.

Denn es hat sich gezeigt, dass ein Team, wenn es in so einer sachbezogenen Runde – nötigenfalls unter Anleitung – respektvoll und sachlich zu diskutieren lernt, einen Raum der (psychologischen bzw. emotionalen) Sicherheit schaffen kann, einen geschützten Raum, in dem es in Ordnung ist, eine vermeintlich „dumme" Frage zu stellen. Weil das Team plötzlich merkt, dass mehr als eine:r diese Frage hat.

Ich stehe mit meinem Problem nicht allein da. Andere springen mir bei oder verstehen mich besser und können mein Anliegen dem Rest erklären, wenn ich das selbst nicht schaffe. Kritische Situationen durch Missverständnisse und ungewollt persönlich empfundene Aussagen werden so durch den Rest des Teams entschärft.

Ich traue mich, zu fragen, weil ich merke, dass einige meiner Fragen selbst von den Profis im Team nicht so ohne Weiteres beantwortet werden können. Und weil ich erlebe, dass auch die vermeintlichen Rockstar-Entwickler:innen nach Dingen fragen, die ich kenne und die ich ganz einfach finde, bei denen ich also aushelfen kann.

Das gemeinsame Machen mit allen kleinen Fragen und Problemen ist der essenzielle Unterschied zu einem „Show and Tell"-Format, der wichtige Unterschied zum nur Zusammensitzen und Reden. Das Team löst nicht nur die Aufgabe gemeinsam, es hilft sich gegenseitig bei (Set-up-)Problemen, zeigt IDE-Tastaturkürzel, hilfreiche Suchbegriffe für die Internetsuchmaschine, fachliche Besonderheiten, die so nicht im Unternehmens-Wiki stehen. Dadurch wird nicht nur implizites Wissen weitergegeben, sondern auch viel Können – also das, was gute Entwickler:innen ausmacht. Und gerade beim impliziten Wissen tun wir uns oft schwer, es angemessen zu dokumentieren; nämlich so, dass es nicht veraltet, und so, dass es gelesen wird. Hier ist es meist viel einfacher,

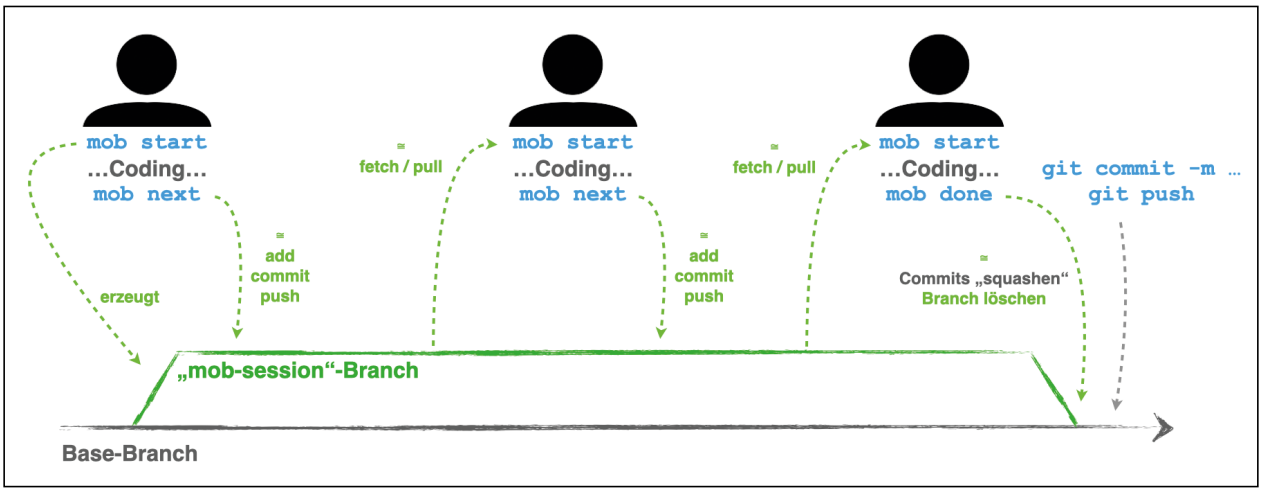

Abb. 7: Schematischer Ablauf der Verwendung von mob in einer Remote-Team-Programming-Session

jemandem zuzuschauen, Fragen zu stellen und es selbst auszuprobieren.

Weil jeder im Team mal tippt, mal mitdiskutiert, entsteht der Code wirklich gemeinsam. Es geht nicht mehr um „meinen Code“ und „deinen Code“. Dieses Co-Authoring oder Co-Creating ist eine sehr starke Methode der Kollaboration.

Dafür muss das Team oft die neuen Rollen lernen (Typist:in als Protokollant:in). Einige müssen lernen, Geduld zu haben, respektvoll auf Missverständnisse und Fehler hinzuweisen. Andere müssen lernen, den Mut zu haben, sich einzubringen, auch mal Neues auszuprobieren. Aber wenn ein Team das konsequent und anfangs mit der nötigen Geduld übt, werden alle sehr schnell bessere Sparring-Partner:innen für die gemeinsame Entwicklung, egal ob mit Team Programming oder bei Einzelaufgaben. Und für Rockstar-Entwickler:innen, die eher ungeduldig und oft genervt davon sind, dass sie alles dreimal erzählen müssen, hat Team Programming den angenehmen Nebeneffekt, dass man es nur einmal in der Teamsession erzählen muss. Häufig ist ein:e Kolleg:in dabei, die/der es sofort versteht und das Team nun beim zweiten Mal anleiten kann.

Es steckt noch sehr viel mehr im Team Programming, was Stoff genug für einen eigenen Artikel ist: Durch die regelmäßige intensive (und anfangs idealerweise moderierte) Zusammenarbeit mit wechselnden Team-Session-Teilnehmer:innen lernt das Team, Konflikte zügig, vertrauensvoll und konstruktiv zu lösen. Man kann beobachten, dass die Phasen des Team-Buildings häufiger, dafür zunehmend weniger kritisch durchlaufen werden. Und ganz nebenbei lösen sich diverse Dysfunktionen, die man in Gruppen antrifft, durch das Erleben der Zusammenarbeit auf.

Nebenbei passt diese Arbeitsweise wunderbar zu IT Kanban mit einer Limitierung des Work in Progress (WIP). Beim Team Programming haben wir implizit ein sehr geringes WIP-Limit – bei einem kleinen Team, das sämtliche Aufgaben mit Team Programming durchführt, haben wir sogar ein WIP-Limit von 1, d. h., die eine aktuelle Aufgabe wird erst fertig gestellt, bevor sich das Team die nächste Aufgabe vornimmt. Dieser Fokus bringt mehr Ruhe ins Team und dadurch hoffentlich bessere Ergebnisse für unsere Kund:innen.

Remote – das neue Normal

Auch wenn bisher schon ab und zu von Besonderheiten beim Remote-Team-Programming die Rede war, gibt es dabei eine besondere Schwierigkeit: Wie geben wir die Tastatur (den aktuellen Arbeitsstand) weiter? Denn wenn das Zeitfenster der derzeitigen Typist:in abgelaufen ist, kompiliert der Codestand vielleicht noch gar nicht. Oder die Tests sind nicht (mehr) grün. Oder die Aufgabe ist noch nicht fachlich korrekt gelöst.

Vor Ort macht das nichts. Es geht einfach der:die Nächste an die Tastatur am selben Rechner und das Team setzt die Arbeit nahtlos fort. Committet bzw. gepusht wird der Code erst, wenn das Team den für sich üblichen Abschluss gefunden hat.

Die Idee eines zentralen, gemeinsam genutzten Rechners kann man auch für die Remotesituation umsetzen. Auf diesem Rechner wird die gemeinsam genutzte IDE konfiguriert und alle, die tippen, verbinden sich per Remote-Session mit diesem Rechner – der vielleicht nur virtuell in der Cloud existiert. Alternativ nutzt man eines der zahlreichen IDE-Plug-ins, mit denen die IDE einer Entwickler:in für alle Teilnehmer der Team-Programming-Session freigegeben wird. Alle Teilnehmer:innen arbeiten remote auf dem freigebenden Rechner, von dem aus der Code am Ende committet wird. Beispiele für solche Plug-ins sind Code With Me (IntelliJ), Live Share (Visual Studio Code) oder Code Together (neben den beiden genannten IDEs auch für Eclipse und im Browser verfügbar).

Der Vorteil ist, dass man bei diesen Varianten schnell einsteigen kann; man benötigt nur einen Remote-Session-Client oder eine passende IDE mit Plug-in, schon kann man auf dem passend konfigurierten Rechner mitcoden. Set-up- bzw. Infrastrukturprobleme, die gerade am Anfang viel Zeit kosten, blendet man damit aus.

Das ist aber auch der größte Nachteil: Wenn man nicht aufpasst, dass die IDE in jeder Folgesession von einer anderen Teilnehmer:in freigegeben wird, ist das Team schnell von dem einen Rechner abhängig, auf dem alles läuft. Außerdem gibt es bei den Remote-Client-Sessions bzw. IDE-Freigaben häufiger Latenzprobleme bei Tastatureingaben, z. B. bei der Code Completion. Und sobald man ein anderes Tool außerhalb der IDE verwendet, muss man doch wieder auf Screen Sharing zurückgreifen.

Eine einfache und doch elegante Lösung ist das mob-Tool [5]. Damit arbeitet jede:r Typist:in lokal und ohne Latenzen auf ihrem komplett eingerichteten System. Die IDEs und übrigen Tools können gleich sein, müssen es aber nicht; das ist eine Teamentscheidung. Der Arbeitsstand wird über ein Git-Remote-Repository ausgetauscht.

Das mob-Tool fasst die nötigen Git-Kommandos zu wenigen Befehlen zusammen, mit denen die Team-Programming-Session durchgeführt wird – im Wesentlichen *mob start*, *mob next* und *mob done*. Das Tool legt am Anfang automatisch einen temporären Git-Remote-Branch an, auf den es die lokalen Änderungen der aktuellen Typist:in am Ende des Zeitfensters pusht. Die nächste Person an der Tastatur pullt den – eventuell gar nicht lauffähigen – Arbeitsstand von dort usw. Erst am Ende der Team-Programming-Session werden die vielen kleinen Commits aus dem temporären Branch zu einem einzigen Commit auf dem Basis-Branch konsolidiert („gesquasht“) und der temporäre Branch automatisch vom mob-Tool gelöscht. **Abbildung 7** verdeutlicht den Ablauf.

Die/der jeweilige Typist:in teilt dabei den eigenen Bildschirm. Alle anderen Teilnehmer:innen haben idealerweise die Kamera eingeschaltet, damit das Team auch Mimik und Gestik zur Kommunikation nutzen kann – wie bei einer Session vor Ort. Das mob-Tool bringt mittlerweile auch einen eigenen Webtimer mit, der automatisch von den Toolkommandos gesteuert wird [4].

Diese Form des Team Programming für verteilte Teams wurde bereits lange vor der Coronapandemie von einem Team als Remote Mob Programming entwickelt und auf seiner Webseite [6] bzw. in einem Buch [7] beschrieben. Beides ist lesenswert, denn dort finden sich unabhängig vom Team Programming gute Ideen zur Remotearbeit.

Wer sich mit dem „Remote Manifest“ von GitLab beschäftigt, wird einen wichtigen Unterschied feststellen. Das Manifest erachtet asynchrone Kommunikation wichtiger als synchrone, Niedergeschriebenes wichtiger als begleitetes Erleben [8]. Remote Team Programming geht den umgekehrten Weg. Wie beim Team Programming vor Ort haben wir eher synchrone Kommunikation, eher gemeinsames Machen und Erleben, weniger Wartezeiten, weniger implizites Können. Das ist ein anderer Kompromiss, denn während das Remote Manifest u. a. darauf abzielt, Mitglieder eines Teams über die ganze Welt verteilt arbeiten zu lassen, geht synchrone Arbeit natürlich nur, wenn das verteilte Team in ähnlichen Zeitzonen tätig ist.

Wie immer geht es dabei nicht um ein Entweder-oder, sondern um den Schwerpunkt bzw. die bevorzugte Arbeitsweise. Eines der Teams, die ich zuletzt coachen durfte, war über Deutschland und Indien verteilt. Es hat für sich den Kompromiss gefunden, an drei Tagen in der Woche jeweils zwei Stunden Remote Team Programming zu machen. Gegen Ende des Arbeitstages in Indien finden sich zwei überlappende Stunden mit dem Arbeitstag in Deutschland. Nach der gemeinsamen Session machen die Kolleg:innen in Deutschland weiter und pflegen die To-do-Datei im Projekt, damit die Kolleg:innen in Indien am anderen Morgen an der passenden Stelle weitermachen können [9].

Und künftig hybrid?

Derzeit ist Remotearbeit im Homeoffice gerade im IT-Bereich oft zu 100 Prozent erlaubt und gewünscht. Aber auch nach der Pandemie werden wir vermutlich bzw. hoffentlich nicht zu 100 Prozent ins Büro zurückkehren. Firmen und Beschäftigte haben in den vergangenen Monaten unerwartet viele Vorteile durch die Remotearbeit für sich entdeckt. Spannend wird dann die Hybridarbeit, wenn ein Team teilweise zu Hause, teilweise im Büro arbeitet – und das an verschiedenen Wochentagen noch mit unterschiedlicher Besetzung.

Wenn während einer Team-Programming-Session ein Teil des Teams im Homeoffice arbeitet und ein anderer Teil im Büro gemeinsam an einem Rechner in einem Meetingraum sitzt, wird es schwierig, alle zu integrieren. Vermutlich werden zwei Gruppen entstehen, die jeweils untereinander diskutieren. Und vermutlich werden die Remotekolleg:innen abgehängt, weil sie die Gespräche vor Ort nicht mitbekommen. Eventuell ist es dann sinnvoller, zwei separate Sessions zu verschiedenen Themen durchzuführen.

Für wirkliche Zusammenarbeit scheint derzeit der Grundsatz „Eine:r remote – alle remote“ zu gelten, damit wir asymmetrische Information vermeiden. Bereits vor der Pandemie durfte ich solche Sessions begleiten. In diesem Fall haben sich die Kolleg:innen vor Ort entweder an ihren Büroplätzen mit Headset und Rechner am Screen Sharing beteiligt. Oder sie haben sich in einem Raum versammelt, jede:r mit eigenem Rechner (für die Kamera und das Screen Sharing), dann aber bevorzugt mit gutem Raummikro und gutem Lautsprecher. Außerdem mussten wir daran denken, dass die Remotekolleg:innen Klebezettel vor Ort nicht sehen können, d. h., wir haben auch in dieser Situation ein Online-Whiteboard genutzt. Das funktioniert erstaunlich gut! Gerade auch, wenn Team Programming nicht zu 100 Prozent, sondern vielleicht einmal am Tag oder ein paarmal in der Woche für zwei bis drei Stunden genutzt wird – eher wie ein anberaumtes Meeting, wenn auch mit anderem Ablauf.

Wie lernt mein Team das?

Ein Team kann sich Team Programming selbst beibringen, beispielsweise auf Grundlage von Artikeln wie diesem, zumal man auch beim Team Programming besser nicht in den Kategorien richtig oder falsch denken sollte. Wichtig ist, dass das Team regelmäßig seine Arbeitsweise reflektiert und sich schrittweise verbessert. Es hat sich aber gezeigt, dass es für einen guten Start hilfreich ist, wenn man Team Programming mit jemandem kennenlernt, der/die es bereits als erfolgreich erlebt hat. Komische Momente, ruckeln, Wartezeiten, gefühlte Konflikte etc. können mit vorhandener Erfahrung besser einsortiert werden, was tatsächlichen Konflikten vorbeugt und das Team schneller zu einem Ablauf bringt, der sich gut anfühlt.

Hilfreich ist auch, wenn die Entwickler:in mit Team-Programming-Erfahrung nicht selbst Teil des programmierenden Teams ist, sondern eher die Rolle einer Moderatorin oder eines Moderators einnimmt. So ein:e Moderator:in muss nicht zwingend von außerhalb des Unternehmens kommen, sollte aber dem Team und den fachlichen Aufgaben nicht zu nahestehen. Die/Der Moderator:in achtet beispielsweise auf die Kommunikation der Teilnehmer:innen, auf die Einhaltung der Zeitfenster und unterstützt das Üben der Strong-Style-Arbeitsweise. Denn gerade das Sich-selbst-Zurücknehmen als Typist:in ist anfangs für viele ungewohnt und schlägt schnell in aktives, selbst gesteuertes Programmieren um. Und die/der Moderator:in kann, weil sie/er selbst nicht mit der konkreten Lösung der Aufgabe befasst ist, leichter an die rechtzeitige Beendigung der aktuellen Session denken, um eine Mini-Retro durchzuführen. Das geht im Eifer des Gefechts sonst allzu leicht unter.

Nach einer teamabhängigen Anfangsphase können die Teams die Sessions natürlich auch allein durchführen. Von Zeit zu Zeit kann eine Begleitung durch Außenstehende aber immer wieder wertvolle Beobachtungen liefern, mit denen sich das Team dann selbst weiterentwickeln kann.

Als Programmieraufgabe für die ersten Team-Programming-Sessions kann man eines der zahlreichen im Netz vorhandenen Coding Dojos verwenden. Mehr Spaß macht es aber meiner Erfahrung nach, wenn das Team eine echte Aufgabe aus dem eigenen Projekt bzw. Produkt umsetzen kann. Dann sollte sich das Team am besten eine Aufgabe suchen, die nicht zu klein ist – sonst ist sie zu schnell erledigt und wir können die Typist-Rotation nicht üben. Die Aufgabe sollte auch nicht zu groß sein, damit wir Fortschritte erleben können. Große Aufgaben brauchen oft noch viel Recherche, und gemeinsame Recherche ist gerade zu Anfang schwierig. Wie weiter oben schon beschrieben: Am besten wählt das Team eine Aufgabe, von der ein oder zwei aus dem Team schon wissen, worum es geht und wie man die Aufgabe üblicherweise löst. Programmieraufgaben sind besser geeignet als Konfigurations-, Infrastruktur- oder Adminaufgaben. Das geht zwar auch alles, aber es ist damit anfangs schwerer, in die Typist-Rotation zu kommen. Alternativ übt man diese mit gemeinsamer Dokumentation etwa im zentralen Wiki – was übrigens auch eine gute Übung für Nichtprogrammierer:innen ist.

Welche Formate sind zum Erlernen von Team Programming geeignet? In den letzten Jahren habe ich z. B. Tagesworkshops mit einem ca. einstündigen Kick-off zu den Grundlagen, zwei Team-Programming-Sessions

von jeweils ca. zwei Stunden Dauer und einem Abschluss im Retro-Stil durchgeführt. Das gab den Teams bereits einen guten Impuls für weitere, eigene Versuche. Oder aber ich durfte Teams ein, zwei oder idealerweise vier Wochen lang begleiten. Dadurch konnten wir die Team-Programming-Sessions nach und nach in den vorhandenen (Sprint-)Ablauf einbringen und die tatsächlichen Teamaufgaben gemeinsam lösen. Was nach relativ viel Moderator- bzw. Coachingeinsatz klingt, aber nachhaltig dazu geführt hat, dass die Teams das gemeinsame Programmieren als brauchbares Werkzeug erlebt haben und nun bei Bedarf bewusst einsetzen.

Es gibt diverse Bücher darüber, was Team Programming ist, wie man es im eigenen Team verankert oder coacht, z. B. [10], [11], [12] und [13]. Die Erfinder des Mob Programming veröffentlichen auf ihrem YouTube-Kanal „Mob Mentality Show“ [14] sehenswerte Beiträge, außerdem pflegen sie eine Liste von weiteren Ressourcen im Netz [15].

Fazit

Team Programming habe ich in den letzten Jahren als große Bereicherung erlebt. Kaum ein Team konnte daraus nicht den einen oder anderen Nutzen ziehen – wenn es sich einmal darauf eingelassen hatte, etwas Neues auszuprobieren. Am Ende einer Team-Programming-Session haben wir statt Endlosdiskussionen oft nicht nur lauffähigen Code, der uns in Richtung der Lösung der eigentlichen Aufgabe bringt. Häufig werden dabei auch viele kleine Missverständnisse und Unklarheiten beseitigt und es wird dadurch ganz nebenbei, aber spürbar zum Team-Building beigetragen.

Wer nun Team Programming im eigenen Team ausprobieren möchte, sollte immer daran denken: Dieser Artikel beschreibt zwar die Grundlagen, die viele Teams weltweit als hilfreich empfinden – aber jedes Team ist anders. Wenn man auf Basis dieser Grundlagen einmal erlebt hat, wie gut Team Programming funktioniert, kann und sollte man damit experimentieren, um das, was dem Team hilft, zu behalten und zu stärken, und das, was als nicht so gut empfunden wird, anzupassen und vielleicht schon nach der nächsten Session neu zu bewerten. Team Programming ist eine relativ neue Programmierpraktik, bei der sich in den kommenden Jahren noch viel tun dürfte. Vor zwei Jahren konnte sich kaum jemand vorstellen, dass wir so intensiv remote zusammen programmieren würden. Und wer weiß, welche neuen Ideen uns in den kommenden Jahren beim hybriden Team Programming Fortschritte bringen werden. Eines aber eint all diese Ansätze: zusammen zur selben Zeit an einer gemeinsamen Aufgabe zu arbeiten. Und das ist für viele Teams ein großer Schritt nach vorn.

Thomas Much ist Technical Agile Coach für Die Techniker und unterstützt Teams mit seinen Coaching-Kolleg:innen dabei, bei der Zusammenarbeit und den agilen Programmierpraktiken immer ein bisschen besser zu werden – u. a. durch die Förderung von Pair und Team Programming.

thomas.much@tk.de @thmuch

Links & Literatur

[1] https://www.agilealliance.org/resources/experience-reports/mob-programming-agile2014/

[2] https://mobti.me

[3] https://cuckoo.team

[4] https://timer.mob.sh

[5] https://mob.sh

[6] https://remotemobprogramming.org

[7] https://leanpub.com/remotemobprogramming

[8] https://about.gitlab.com/company/culture/all-remote/

[9] https://linkedin.com/pulse/building-teams-mob-programming-new-normal-arjun-rc-he-him-/

[10] https://leanpub.com/mobprogramming

[11] https://leanpub.com/techagilecoach

[12] https://ensembleprogramming.xyz

[13] https://pragprog.com/titles/mpmob/code-with-the-wisdom-of-the-crowd/

[14] https://www.youtube.com/channel/UCgt1lVMrdwlZKBaerxxp2iQ/videos

[15] https://trello.com/b/1lfMkCOh/software-profession-resources

Der Weg durch den API-Dschungel

„Na, dann baut doch ein API!"

Die Aussage in der Headline ist leichter gesagt als getan. Denn es muss eine Reihe von Entscheidungen getroffen werden, um das „perfekte" API zu bauen. In diesem Artikel wollen wir einen Weg durch den Dschungel der Fragen weisen, die aufkommen, wenn man ein API planen und umsetzen möchte.

von Renke Grunwald und Arne Limburg

Es gibt verschiedene Situationen, in denen die Entwicklung eines API notwendig ist. Häufig handelt es sich dabei um eine Reaktion auf eine fachliche Anforderung. Wer mehr zur proaktiven Entwicklung eines API lesen will, dem sei der Artikel von Lars Röwekamp in diesem Heft ans Herz gelegt.

Der Anstoß für ein API ist normalerweise, dass Informationen von A nach B fließen müssen. A und B können dabei beliebige Komponenten sein, also z. B. Microservices, Web-Frontends, Rich Clients, Mobile Clients usw. Anders als häufig behauptet ist es für die Gestaltung eines API von erheblicher Bedeutung, um was für Komponenten es sich handelt und in welcher Beziehung sie zueinander stehen.

In welche Richtung die Daten fließen sollen, ist dabei in der Regel recht schnell klar. Bei einer Leseoperation bietet der Provider der Daten das API an, bei einer Schreiboperation der Consumer. Um herauszufinden, worum es sich handelt, sollte man betrachten, welches der Systeme das führende System der Informationen ist, die fließen sollen.

Lässt es sich nicht leicht identifizieren, sollte zunächst die Frage gestellt werden, ob es überhaupt eine Information gibt, die fließen muss, oder ob die Anforderung, ein API zu bauen, aus einer Fehlannahme resultiert. Letzteres kommt häufiger vor als man denkt. Der Aufbau und vor allem die Pflege eines API bedeuten immer Aufwand. Bevor also ein API erstellt wird, sollte immer klar sein, dass es auch tatsächlich benötigt wird.

Dazu muss man sich fragen: Wer hat eine Information, die jemand anders benötigt, und wer ist derjenige, der die Information benötigt? Nur wenn es einen Anbieter der Information und auch mindestens einen Konsumenten der Information gibt, ist ein API sinnvoll. Denn aufgrund des erwähnten Aufwands gilt immer: Keine Schnittstelle ist besser als eine Schnittstelle.

Um die Datenkonsistenz des gesamten Systems jederzeit zu gewährleisten, sollten schreibende Operationen immer nur am führenden System durchgeführt werden, das dafür ein API bieten sollte. Dieses System sollte dann natürlich auch immer den aktuellen Stand des Datensatzes haben. Oder anders formuliert: Erst wenn die Daten im führenden System aktualisiert sind, gelten sie für das Gesamtsystem als erfolgreich aktualisiert.

Im Gegensatz dazu können lesende Informationen auch bei Caches oder bei aggregierenden Services abgerufen werden. Dann sollte man sich aber immer klar machen, dass die Informationen ggf. veraltet sind und wie damit umgegangen werden soll.

Wenn festgelegt wurde, welches System die Schnittstelle bereitstellt, muss geklärt werden, welche Daten fließen sollen. Um den Wartungsaufwand möglichst gering zu halten, sollte hier darauf geachtet werden, dass zunächst nur die Daten über die Schnittstelle zur Verfügung gestellt werden, die auch tatsächlich benötigt werden. Erweitert werden kann eine Schnittstelle jederzeit, wie wir später sehen werden. Der früher häufig verwendete Ansatz, einfach alle vorhandenen Daten zu übertragen, sollte unbedingt vermieden werden. Erstens führt das zu unnötigem Netzwerk-Traffic und zweitens wird es dadurch erschwert, die Schnittstelle später zu ändern. Das ist insbesondere dann ärgerlich, wenn in der weiteren Entwicklung Daten geändert werden müssen, die zunächst gar nicht benötigt wurden.

Nachdem die Daten der Schnittstelle geklärt wurden, sollte der Workflow geklärt werden, in dem die Daten benötigt und abgerufen werden. Bei lesenden Operationen sollte insbesondere geklärt werden, ob der Konsument die Daten abrufen wird, wenn er sie benötigt oder ob der Anbieter des API den Konsumenten proaktiv informiert, wenn sich die Daten ändern. Im ersten Fall wäre eine synchrone Schnittstelle sinnvoll, im zweiten

Fall eine asynchrone. Die Entscheidung, ob ein synchrones oder asynchrones API sinnvoll ist, ist im Einzelfall nicht einfach zu treffen.

Bei lesenden Operationen ist eine synchrone Abfrage zunächst der intuitive Weg: Man benötigt Daten, also fragt man sie an. In einem verteilten System hat dieses Vorgehen aber zwei Nachteile. Zum einen besteht immer das Risiko, dass der aufzurufende Service nicht verfügbar ist und man in seinem Use Case ganz ohne Daten dasteht. Zum anderen ist ein synchroner Aufruf auch weniger performant, weil die Anfrage immer über mindestens zwei Systeme gehen muss.

Die Alternative dazu wäre die asynchrone Kommunikation, bei der das führende System bei jeder Änderung der Daten ein Event wirft. Der Service, der die Daten benötigt, würde dann auf diese Events hören und intern einen Cache pflegen, den er aktuell hält. Benötigt er die Daten, kann er aus dem internen Cache bedient werden, wodurch die beiden Nachteile der synchronen Kommunikation nicht vorhanden sind.

Auch bei schreibenden Operationen ist ein synchroner Aufruf oft intuitiver, z. B. weil man ein direktes Feedback über Erfolg oder Misserfolg der Schreiboperation erhält und dann dementsprechend weiterarbeiten kann. Aber auch hier hat man das Problem, dass das aufzurufende System nicht verfügbar sein könnte.

Wählt man die asynchrone Variante, bei der man einen Command über eine Messaging-Infrastruktur verschickt, so ist es beim Senden des Commands unerheblich, ob das aufnehmende System verfügbar ist. Bei diesem Vorgehen ist natürlich unbedingt die Hochverfügbarkeit der Messaging-Infrastruktur sicherzustellen. Nachteil der asynchronen Variante ist, dass der Erfolg der Operation auch asynchron kommuniziert wird (über das oben erwähnte Event, dass sich Daten geändert haben). Wenn der Use Case also auf dem Erfolg der Schreiboperation basiert, muss dieser in zwei asynchrone Codeabschnitte aufgeteilt werden, was die Wartung erschwert.

Die Wahl der Technologie

Während die eigentliche Funktionalität des API natürlich im Fokus steht, muss letztlich auch die Entscheidung getroffen werden, mit welcher Technologie es umgesetzt wird. Um eine geeignete API-Technologie auszuwählen, ist es ratsam, sich zunächst einige Fragen bezüglich der Anforderungen an das API zu stellen.

Wir werden uns einige mögliche Fragen genauer anschauen und je nach Beantwortung passende Technologien vorstellen. Da es eine Fülle von API-Technologien gibt, werden wir uns auf einige bekanntere Vertreter konzentrieren. Die Auswahl ist also bewusst nicht vollständig.

Wie einfach soll das API konsumierbar sein?

Um ein API für so viele Konsumenten wie möglich bereitzustellen, sollten die technischen Hürden so gering möglich sein – auch mit dem bewussten Verzicht auf mögliche Vorteile, die eine komplexere API-Technologie mit sich bringen kann.

Technologien wie etwa GraphQL oder gRPC erkaufen sich ihre Vorzüge durch technologische Komplexität, die es zunächst zu überwinden gilt. Eine schnelle und einfach Anbindung an diese API-Technologien ist damit durchaus schwierig. Die Verwendung von technologiespezifischen Bibliotheken ist bei diesen Technologien absolut notwendig. Man tauscht hier gewissermaßen technologische Komplexität gegen Mächtigkeit der Technologie.

Aufgrund geringer Komplexität und einfachem Verständnis haben sich indessen JSON-basierte HTTP APIs durchgesetzt. Sowohl das Modell in Form von JSON als auch Verwendung der verschiedenen HTTP-Verben für einen bestimmten Pfad sind konzeptionell einfach zu begreifen. Wenn zusätzlich noch etablierte Designprinzipien wie etwa das RESTful oder gar oData angewandt werden, kann ein Consumer sich fast vollends auf die eigentliche Funktionalität des API fokussieren, ohne durch technische Komplexität abgelenkt zu werden.

Praktisch jede Programmierumgebung bietet die Möglichkeit an, HTTP-Anfragen zu versenden und anschließend zu verarbeiten. Die Nachvollziehbarkeit ist durch ein rein textuelles Format ebenfalls gegeben. Mit gängigen Tools wie Postman oder curl lassen sich APIs nahezu spielend erkunden, ohne dass aufwendige Vorbereitungen getroffen werden müssen.

Die weite Verbreitung von HTTP, vor allem als Standardprotokoll in Webbrowsern, bringt ein enormes Wissen, sowohl in Form von Dokumentation als auch bei vorhandenem Entwicklerwissen, mit sich. Bei anderen Technologien, gerade wenn diese eher neuartig sind oder nur eine gewisse Nische abdecken, ist das oft nicht der Fall; hier muss bei Problemen oft teurer Expertenrat eingeholt werden.

Für viele Fälle ist ein RESTful API im Zweifel ein guter Kompromiss, mit dem wenig falsch gemacht wird. Die Wahl spezieller API-Technologien sollte wohlüberlegt und durch Anforderungen begründet sein.

Gibt es technologische Einschränkungen?

Die beste API-Technologie bringt wenig, wenn ein Konsument diese nur eingeschränkt oder überhaupt nicht verwenden kann. Die technologischen Einschränkungen, insbesondere des Konsumenten, nehmen erheblichen Einfluss auf die Wahl der Technologie.

So war es mit dem damals aufkommen Hype für Node.js-Entwickler sicher reizvoll ein GraphQL-basiertes API bereitzustellen. Wurde das API jedoch hauptsächlich von Java-Konsumenten verwendet, sorgte das sicher für einigen Frust bei den Entwicklern, da die GraphQL-Unterstützung im Java-Ökosystem initial noch nicht weit vorangeschritten war. Solche Einschränkungen können sich – wie im Fall von GraphQL – mit der Zeit durchaus lockern, aber bei der Entscheidung einer Technologie sollte natürlich stets der Ist-Zustand im Vordergrund stehen. Etwaige Vorteile für den Produzenten übertrumpfen selten die dabei entstehenden Nachteile bei den Konsumenten.

Ähnlich ist es bei Technologien wie gRPC, die aufgrund technischer Gegebenheiten für bestimmte Umgebungen eher ungeeignet sind. So können browserbasierte Konsumenten die Effizienz von gRPC nur begrenzt ausschöpfen, da ein Browser nur eingeschränkten Einfluss auf die HTTP/2-Kommunikation bietet. Zwar könnte der Konsument das gRPC API grundsätzlich (mit Einschränkungen) nutzen, es stellt sich jedoch die Frage, ob eine andere Technologie möglicherweise besser geeignet ist.

Genauso gibt es Technologien, die gänzlich ungeeignet sind, da ein Konsument diese nicht direkt verwenden kann. Der direkte Einsatz von Kafka oder vielen anderen Event-Messaging-Technologien im Browser ist schlichtweg nicht möglich, da der Browser keine beliebigen TCP-Verbindungen zu einem Broker aufbauen kann. Aus Sicht des Konsumenten wäre es daher notwendig, dass der Producer stattdessen ein WebSocket- oder ein Server-sent Event API bereitstellt, da diese Protokolle nativ vom Browser unterstützt werden.

Bei der Wahl der API-Technologie müssen daher immer die technischen Gegebenheiten einbezogen werden, auch wenn das bedeutet, dass auf die vom Produzenten präferierte Technologie zugunsten der für den Konsumenten passenden Technologie verzichtet wird. Hier lohnt oft der Dialog mit möglichen Konsumenten, um einen guten Überblick potenzieller Einschränkungen zu erhalten.

Wie effizient muss das API sein?

Ein funktionierendes API ist wichtig, aber nicht selten muss eine Schnittstelle auch effizient sein. Eine Anfrage, die mehrere Sekunden dauert oder unverhältnismäßig große Datenmengen überträgt, kann für viele Konsumenten unzumutbar sein.

Ist der Konsument etwa ein mobiles Endgerät, dem nur eine langsame Internetverbindung zur Verfügung steht, ist die Menge übertragener Daten ein entscheidender Faktor. Die Wahl der API-Technologie kann hier massiven Einfluss haben. Bei textuellen Formaten wie JSON kann durch Komprimierung die übertragene Menge an Daten erheblich reduziert werden, aber den häufig noch kleineren Fußabdruck von Binärformaten wie etwa Protobuf zu unterbieten, wird dennoch schwierig sein.

Genauso lässt sich etwa mit GraphQL einfacher ein Overfetching verhindern, da der Konsument bestimmt, welche Teilmenge an Informationen er benötigt. In klassischen JSON-basierten HTTP APIs hingegen ist das ohne Mehraufwand nicht möglich. Hier gibt es keine Standardlösung, um festzulegen, in welcher Form der Payload zurückgegeben wird. Deshalb muss im Zweifel auf eigene Konventionen und Implementierungen gesetzt werden, um Overfetching zu vermeiden, etwa über einen expliziten Parameter oder Content Negotiation. Ein solche Funktionalität ist bei GraphQL von Haus aus dabei.

Dennoch bieten gerade etablierte Technologien oft viele Tricks an, um die Effizienz eines API zu erhöhen. So bietet HTTP eine Vielzahl von Möglichkeiten, um Caching innerhalb von APIs zu ermöglichen – mit dem großen Vorteil, dass viele HTTP-Clients das automatisch unterstützen. Die Benutzung der relevanten Cachingfunktionalitäten ist in HTTP-basierten Technologien wie GraphQL oder JSON-RPC indessen nicht ohne weiteres möglich, da diese HTTP eher als Mittel zum Zweck ansehen und sich weniger an die zugrundeliegende Semantik wie die angedachte Verwendung der HTTP-Verben halten.

Wie stark dürfen Producer und Consumer gekoppelt sein?

Eine weitere relevante Frage ist, wie sehr Produzenten und Konsumenten gekoppelt sein dürfen. Je nach angestrebter Kopplung oder eben der Vermeidung dieser bieten sich bestimmte Technologien eher an als andere.

Kennt ein Produzent die konkreten Use Cases der Konsumenten, ist ein eher allgemeiner Ansatz, der von RESTful APIs verfolgt wird, mitunter hinderlich. Stattdessen könnte das API spezielle Endpunkt anbieten, die direkt auf den jeweiligen Use Case der Konsumenten zugeschnitten sind. So kann auf einfache Weise das oben erwähnte Overfetching verhindert werden, da nur genau die benötigten Daten übertragen werden. Zusätzlich müssen sich die Entwickler weniger Gedanken beim Design der verfügbaren Ressourcen machen. Das sorgt erfahrungsgemäß für hitzige Diskussionen innerhalb eines Teams, besonders, wenn vermeintliche REST-Regeln gebrochen werden.

Gerade Technologien wie JSON-RPC und gRPC stellen hier eine gute Wahl dar, da diese bewusst eher in Operationen (Verben) als in Ressourcen (Nomen) denken. Ein vereinfachter Vergleich von RESTful und JSON-RPC ist nachfolgend zu sehen. Stornierung als Ressource in einem RESTful API:

```
POST /orders/123/cancellation
```

Stornierung als Operation in einem JSON-RPC API:

```
{"jsonrpc": "2.0", "method": "cancelOrder", "params": {"orderId": 123}
```

Eine stärkere Kopplung muss nicht notwendigerweise mit Nachteilen daherkommen. Werden etwa Frontend

und Backend vom selben Team entwickelt – und auch im selben Rhythmus ausgeliefert – ergeben sich aus der hohen Kopplung selten Probleme. Durch die Kopplung können bewusst Aspekte wie Versionierung und mögliche Breaking Changes außer Acht gelassen werden.

Im Gegensatz dazu, bieten sich RESTful APIs vor allem dann an, wenn der Produzent bewusst wenig Kenntnisse von den möglichen Use Cases potenzieller Konsumenten hat. Hier steht eher die Flexibilität des API im Vordergrund. Geringe Kopplung ist vor allem bei getrennten Teams oder öffentlichen APIs ein Muss.

Ein gutes Beispiel ist das damalige Twitter-API, das bewusst nicht auf konkrete Use Cases zugeschnitten, sondern sehr generisch gehalten war und daher zur Entwicklung unzähliger Twitter-Clients mit teils innovativen Features führte. Wäre das API nur auf die Use Cases der Twitter-eigenen Anwendung ausgelegt gewesen, gäbe es wenig Spielraum, um neuartige Wege zur Benutzung von Twitter zu erfinden.

Wie wichtig ist das Ökosystem der API-Technologie?

Ein Aspekt, der die Entscheidung für eine API-Technologie ebenfalls beeinflussen kann, ist das Ökosystem, das sich um dieses entwickelt hat. Ein gutes Ökosystem hilft nicht nur dem Produzenten bei der Implementierung des API, sondern durchaus auch dem Konsumenten beim Anbinden und der Verwendung der Schnittstelle.

Weitverbreitete API-Technologien haben naturgemäß eine größere Fülle an Tools. So gibt es für HTTP-basierte APIs und im speziellen für RESTful APIs viele hilfreiche Tools. Von interaktiven HTTP-Clients bis zu ausgereiften Mocking-Tools ist quasi alles dabei. Diese Tools können das Arbeiten mit und an einem API enorm erleichtern und sind meist ein wichtiger Baustein bei der Entwicklung der Schnittstelle.

Bei neueren Technologien steht der Toolaspekt oft bewusst im Fokus. Gewissermaßen haben sich die Werkzeuge hier nicht um die bestehende Technologie entwickelt, wie es etwa bei HTTP der Fall ist, sondern waren direkt bei ihrer Entstehung ein Ziel oder gar eine notwendige Vorrausetzung.

So bietet das GraphQL-Ökosystem mit GraphiQL einen interaktiven API-Browser an, der sich auf einfache Weise in der DEV-Stage als Teil des Frontends integrieren lässt. Also genau da, wo auch die eigenen Anfragen gemacht werden. Jeder der GraphQL kennt, wird in der Regel auch mit GraphiQL in Berührung gekommen sein. Das zeigt den engen Verbund zwischen Technologie und dem dazugehörigen Ökosystem.

Ein weiteres Beispiel ist die Codegenerierung, die bei der Verwendung von gRPC ein essenzieller Bestandteil ist. Die Qualität der Codegeneratoren ist dementsprechend sehr hoch. Bei Ansätzen wie etwa OpenAPI, das für den allgemeinen Einsatz bei HTTP-basierten APIs gedacht ist, ist Codegenerierung ein eher nebensächlicher Aspekt, der sich mitunter in der Qualität der darauf basierenden Codegeneratoren niederschlägt.

Zwar muss das Ökosystem nicht der entscheidende Faktor bei der Wahl einer API-Technologie sein, kann aber dennoch den Ausschlag geben, wenn sich mehrere Technologiealternativen gleichermaßen anbieten. Ein gesundes Ökosystem ist auch ein Zeichen dafür, dass eine Technologie etabliert ist und von vielen Entwicklern geschätzt wird.

Wie dokumentiert man ein API?

Nachdem die Wahl einer API-Technologie gefallen ist und die Schnittstelle geplant wurde, stellt sich noch eine weitere Frage: Wie stellt man sicher, dass ein Konsument es auch korrekt nutzt? Der Idealfall ist natürlich, dass sich die korrekte Nutzung des API direkt aus dem API ergibt. Einen solchen Ansatz versuchen HATEOAS-basierte APIs zu realisieren. HATEOAS steht dabei für Hypermedia as the Engine of Application State, was nicht weniger bedeutet, als dass der Zustand des Use Cases und die daraus resultierenden möglichen Operationen direkt aus der vorherigen Kommunikation hervorgehen. Eine weitere Dokumentation ist dann im Idealfall nicht notwendig. Da die Realisierung dieses Ansatzes recht aufwendig ist, bietet er sich nur in einer begrenzten Menge von Fällen an, und zwar dann, wenn mögliche Operationen tatsächlich extrem von vorhergehenden Aktionen abhängen, wie es häufig bei Konfigurationen der Fall ist („erst wenn ein Haken gesetzt wurde, stehen weitere Haken zur Verfügung“ oder „wenn diese Auswahl getroffen wurde, ist jene Auswahl nicht mehr möglich“). HATEOAS bietet in solchen Fällen eine gute Möglichkeit, dem Client die Businesslogik zugänglich zu machen, ohne dass dieser sie im Vorfeld kennen und implementieren muss. Er muss lediglich so generisch aufgebaut sein, dass er flexibel auf Antworten reagiert.

Eine solche Implementierung ist sowohl für Consumer als auch für Provider des API aufwendig zu realisieren. In den meisten Fällen bietet sich daher eine klassische Dokumentation des API an.

Je komplexer das API ist, umso detaillierter sollte dabei die Spezifikation sein. Fehlt sie, führt das schnell zum Frust bei den Konsumenten, insbesondere dann, wenn jede Anfrage mit einem unfreundlichen 400-Statuscode beantwortet wird.

Eine Spezifikation für ein API lässt sich grob in zwei Bestandteile gliedern: die eigentlichen Operationen, die bereitgestellt werden, und das Modell, auf denen sie operieren. Je nach Schnittstelle kann entweder die Beschreibung der Operationen oder des Modells wichtiger sein. Idealerweise beschreiben Spezifikationen sowohl das Modell als auch Operationen so ausführlich, dass es für einen Konsumenten bei der Benutzung des API zu keinen Missverständnissen kommt.

Wie auch bei anderen Entscheidungen, die bei der Erstellung eines API getroffen werden müssen, kann die Spezifikation der Schnittstelle auf verschiedenen Wegen erfolgen. Wir werden uns einige mögliche Varianten anschauen und auf Vor- und Nachteile eingehen.

Unabhängig davon, wie die Spezifikation der Schnittstelle umgesetzt wird, fällt die Kommunikation zwischen Produzenten und Konsumenten nicht weg und bleibt ein wichtiges Mittel, um den Erfolg eines API zu garantieren. Ein gutes API entsteht fast nie im Vakuum. Nur weil eine Schnittstelle gut spezifiziert ist, bedeutet das nicht, dass die Anforderungen des Konsumenten vollumfänglich erfüllt werden. Die Spezifikation kann jedoch als gutes Medium zum formalen Austausch zwischen Produzenten und Konsumenten dienen, um so iterativ das passende API zu entwickeln, mit dem alle Beteiligten zufrieden sind.

Bewusster Verzicht auf eine Spezifikation

Gänzlich auf eine Spezifikation jeglicher Art zu verzichten, klingt zunächst fahrlässig (sofern man nicht das oben erwähnte HATEOAS anwendet). Es zu tun, kann sich jedoch anbieten, wenn sowohl Produzent und Konsument im selben Team entwickelt werden, speziell wenn einzelne Entwickler ein neues Feature gleichzeitig auf beiden Seiten entwickeln.

Die Spezifikation des API existiert hierbei lediglich implizit in Form des Codes im Produzenten. Der Konsument muss demnach ein tieferes Verständnis des Produzenten haben, um die Funktionsweise des API zu verstehen und nachvollziehen zu können.

Bei einem größeren Team oder gar unterschiedlichen Teams für Produzenten und Konsumenten gerät dieser Ansatz jedoch schnell an seine Grenzen. Das Verständnis eines API an das Wissen um die Interna des Produzenten zu koppeln widerspricht oft dem eigentlichen Anlass, überhaupt erst eine Schnittstelle bereitzustellen. Nicht zuletzt ist es oft das Ziel eines API, genau diese Details zu verbergen.

Es wird insbesondere dann kritisch, wenn ein Konsument sich auf ein ursprünglich nicht angedachtes Verhalten des Produzenten verlässt, das dieser zukünftig nicht ohne weiteres ändern kann, ohne damit aus Sicht des Konsumenten Breaking Changes zu verursachen.

Auf eine Spezifikation zu verzichten, ist daher eine Entscheidung, die wohlüberlegt sein muss und nur bei bestimmten Teamkonstellationen oder einfachen APIs sinnvoll ist.

Shared Library/Client-SDK

Ein weiterer Ansatz ist die Bereitstellung einer Shared Library, in der das Modell des API als verwendbarer Code vorliegt (z. B. als Java-Klassen). So eine Shared Library kann sowohl vom Konsumenten als auch dem Produzenten verwendet werden, um sicherzustellen, dass stets das richtige Modell verwendet wird. Dieser Ansatz verhindert viele vermeidbare Fehler, da ein Konsument auf das oft fehleranfällige Spiegeln des Modells im eigenen Code verzichten kann.

Um den Konsumenten weiter zu entlasten, bietet es sich an, neben dem eigentlichen Modell auch die Operationen in Form von Code bereitzustellen. Das geschieht oft in Form eines Client-SDK, das die Operationen des API direkt als Funktionen/Methoden in der jeweiligen Programmiersprache enthält.

Durch ein Client-SDK kann sich ein Konsument deutlich mehr auf den eigentlichen Mehrwert des API konzentrieren, da Technologiedetails wie etwa HTTP-Header, Dateiformat etc. bewusst hinter Funktionen/Methoden versteckt werden. Der Produzent hingegen ist deutlich flexibler bei der technischen Umsetzung der Schnittstelle, da diese durch das Client-SDK vor dem Consumer versteckt werden kann.

Die eigentliche Dokumentation erfolgt in diesem Ansatz häufig durch programmiersprachenspezifische Mittel. So würde ein Client-SDK für Java mit Hilfe von JavaDoc dokumentiert sein, um den Konsumenten maximalen Komfort bei der Verwendung zu bieten.

Ein großer Nachteil einer Shared Library oder eines Client-SDK ist die Vorrausetzung, dass sowohl Produzent als auch der Konsument dieselbe Programmiersprache verwenden. Zwar kann der Produzent Client-SDKs für verschiedene Zielsprachen anbieten, verliert dadurch jedoch mitunter den möglichen Vorteil der gleichzeitigen Verwendung des Modells im Produzenten und Konsumenten.

Ein weiter negativer Aspekt ist die engere Kopplung zwischen Produzent und Konsument. Durch das vorgegebene Modell und die Operationen ist es für den Konsumenten mitunter nicht mehr möglich, nur von bestimmten Teilen des Modells abzuhängen. Das erschwert möglicherweise die Veränderung des API, ohne dabei die Konsumenten zu beeinträchtigen, selbst dann, wenn diese eigentlich von der Änderung nicht betroffen wären.

Schemabasierte Spezifikation (Code-First vs. Schema-First)

Da der Produzent eines API häufig die potenziellen Konsumenten nicht kennt, sind die zuvor beschriebenen Ansätze, die entweder Kenntnis über den Code des Pro-

duzenten oder die Verwendung derselben Programmiersprache vorrausetzen, nicht immer geeignet. Es bedarf in diesen Fällen also einer Spezifikation, die unabhängig von konkreten technischen Details des Produzenten und des Konsumenten die verfügbaren Operationen und das dazugehörige Modell beschreibt: ein sogenanntes Schema.

Der wohl bekannteste Vertreter im Bereich der schemabasierten API-Spezifikation ist OpenAPI, früher auch unter dem Namen Swagger bekannt. OpenAPI erlaubt die Spezifikation der Operationen und des Modells einer HTTP-basierten Schnittstelle inklusive dazugehöriger Dokumentation in Form eines JSON- oder YAML-basierten Schemas.

Eine solche OpenAPI-Spezifikation dient dem Konsumenten in erste Linie als detaillierte Beschreibung eines API. Statt eines selbst ausgedachten Formats bietet OpenAPI ein einheitliches Vorgehen, um die Schnittstelle im gewünschten Detailgrad zu spezifizieren.

Als offener Standard bietet OpenAPI nicht nur dem Entwickler des API Vorteile, sondern dient auch als Grundlage für zahlreiche Tools, die direkt OpenAPI-basierte Spezifikation verarbeiten können und basierend darauf nützliche Funktionalitäten anbieten. Einige Beispiele:

- bessere IDE-Unterstützung
- Visualisierung und interaktiver API-Browser (via Swagger UI)
- automatische Codegenerierung des Modells oder gar eines gesamten Client-SDK
- Mock-Server

Natürlich gibt es neben OpenAPI zahlreiche andere Schemaformate, um Schnittstellen zu spezifizieren und dokumentieren, alle jeweils mit ihren eigenen Tools. Um einige bekannte Vertreter zu nennen:

- AsyncAPI – ähnlich wie OpenAPI, nur für die Beschreibung von Event-basierten Schnittstellen
- GraphQL Schema Definition Language
- Protocol Buffers
- WSDL
- JSON Schema (ist auch Teil von OpenAPI)

Welcher konkrete Ansatz zum Einsatz kommen sollte, hängt naturgemäß von der ausgewählten API-Technologie ab. Ist die Entscheidung gefallen, das API mit Hilfe eines Schemas zu spezifizieren, stellt sich schnell die Frage, wie genau dieses Schema letztlich erzeugt wird. Hier lassen sich generell zwei Vorgehensweisen unterschieden: Code-First und Schema-First.

Code-First

Bei Code-First liegt, wie der Name schon andeutet, der Fokus auf dem Code. Das Schema ergibt sich aus der Implementierung des Produzenten. Mit Hilfe geeigneter Tools wird das Schema basierend auf dem Code – optional gepaart mit toolspezifischen Annotationen (z. B. Java Annotations oder spezielle Kommentare) – generiert. Die Erzeugung des Schemas kann dabei sowohl zur Compile- als auch zur Runtime erfolgen, je nachdem, wie genau ein Konsument auf das Schema zugreifen wird. Ein Beispiel mit OpenAPI ist in Listing 1 zu finden. (Es handelt sich dabei um einen Spring-MVC-basierten HTTP-Endpunkt in Kotlin, der mit Swagger-Annotationen erweitert ist; hieraus kann eine entsprechende Operation im Open-API-Format generiert werden.)

Der Code stellt also die Wahrheit bezüglich der Spezifikation des API dar. Anders als beim bewussten Verzicht auf eine Spezifikation müssen die Konsumenten keine Interna des Produzenten kennen, da das automatisch generierte Schema eine vollumfängliche Beschreibung der Schnittstelle darstellt.

Ein großer Vorteil von Code-First ist die fehlende Notwendigkeit, händisch ein Schema anzulegen. Der Produzent kann ausschließlich im Code arbeiten und muss eventuell punktuell die toolspezifischen Annotationen verwenden. Tieferes Verständnis für das verwendete Schemaformat ist nicht notwendig. Das Schema ist gewissermaßen einfach ein weiteres Build-Artefakt, das als Teil der Anwendung erzeugt wird. So ersteht etwa neben einem JAR auch eine entsprechende Schemadatei, die über eine dafür vorgesehene Schema-Registry oder auch manuell an Konsumenten verteilt wird.

Listing 1

```
@ApiOperation(
  value = "Add comment to a post",
  nickname = "addComment",
  response = AddCommentResponseDto::class,
)
@ApiResponses(
  value = [
    ApiResponse(
      code = 201,
      message = "Comment added successfully",
      response = AddCommentResponseDto::class
    )
  ]
)
@RequestMapping(
  method = [RequestMethod.POST],
  value = ["/posts/{postId}/comments"],
  produces = ["application/json"],
  consumes = ["application/json"]
)
fun addComment(
  @ApiParam(value = "The ID of the post the comment is added to",
            required = true) @PathVariable("postId") postId: kotlin.String,
  @ApiParam(value = "Data required to add a comment", required=true)
     @Valid @RequestBody addCommentRequestDto: AddCommentRequestDto
): ResponseEntity<AddCommentResponseDto> {
  // Implementation goes here
}
```

Generell ist Code-First dann sinnvoll, wenn die Konsumenten wenig Einfluss auf das API nehmen und der Produzent entscheidet, wie die Schnittstelle aussieht. Das gilt oft dann, wenn es viele verschiedene Konsumenten gibt, der Produzent sie aber nicht explizit kennt. Ein gutes Beispiel ist hier erneut das bereits erwähnte Twitter-API. Einzelne Konsumenten des Twitter-API haben üblicherweise wenig Einfluss auf den Funktionsumfang des API und sind nicht an seiner Entwicklung beteiligt.

Schema-First

Das genaue Gegenteil von Code-First stellt Schema-First dar. Statt das Schema nur als Mittel zum Zweck zu betrachten und den Konsumenten die Benutzung des API zu erleichtern, dient es als eine Art Vertrag, an den sich der Produzent hält und auf den sich der Konsument verlassen kann. Da durch dieses Vorgehen das Schema die Wahrheit darstellt, nimmt das Schema einen ähnlichen Stellenwert wie der eigentliche Code ein und sollte dementsprechend mit derselben Sorgfalt gepflegt werden.

Schema-First ist dann ratsam, wenn ein API kollaborativ entwickelt wird und sowohl Produzent als auch Konsument Einfluss auf die Schnittstelle nehmen sollen. Das Schema dient hierbei als neutrales Medium, um gemeinsam das API zu erarbeiten, ohne dabei von technischen Details wie der verwendeten Programmiersprache abgelenkt zu sein. Beispiele für ein OpenAPI-basiertes Schema sind in Listing 2 (Auszug aus einem OpenAPI-basierten Schema, das die Operation zum Hinzufügen von Kommentaren spezifiziert) und Listing 3 (Auszug aus einem eines OpenAPI-basierten Schema, das das Modell für das Hinzufügen von Kommentaren spezifiziert) zu finden.

Listing 2

```
/posts/{postId}/comments:
  post:
    summary: Add comment to a post
    operationId: addComment
    requestBody:
      description: Data required to add a comment
      required: true
      content:
        application/json:
          schema:
            $ref: "#/components/schemas/AddCommentRequest"
    responses:
      '201':
        description: Comment added successfully
        content:
          application/json:
            schema:
              $ref: "#/components/schemas/AddCommentResponse"
    parameters:
      - name: postId
        description: The ID of the post the comment is added to
        in: path
        required: true
        schema:
          type: string
```

Haben Produzent und Konsument sich auf ein Schema geeinigt, können beide unabhängig voneinander mit der Implementierung beginnen. Da das Schema bereits vor der Implementierung festgelegt werden kann, kann der Konsument etwa mit Hilfe von Mocks frühzeitig anfangen, auf dem API basierende Features zu entwickeln.

Dieser Ansatz hat sich in der Praxis insbesondere bei der Kommunikation zwischen Backend und Frontend bewährt. Neue Features können zügig implementiert werden, ohne auf die Implementierung des API warten zu müssen. Das ist gerade im Frontend-Bereich nicht unwichtig, da es hier von enormem Vorteil ist, wenn man Features frühzeitig sehen und anfassen kann. Stellt sich dann heraus, dass das Feature in dieser Form keinen Sinn ergibt, kann die im besten Fall noch nicht begonnene Arbeit im Backend abgebrochen werden.

Erwähnenswert sind im Kontext von Schema-First insbesondere API-Technologien wie GraphQL und gRPC, bei denen das Schema ein integraler und oft auch verpflichtender Bestandteil ist. Bei anderen technologischen Ansätzen wie RESTful APIs ist der Einsatz eines Schemas eher ein optionaler, aber sinnvoller Zusatz.

Eine interessante Kombination aus Schema-First und Code-First bietet die Microprofile-Implementierung des OpenAPI-Standards [1]. Hier kann mit einem Schema-First-Ansatz begonnen werden, der dann später im Code durch Annotations ergänzt bzw. überschrieben wird. Das tatsächlich entstehende Schemadokument ist dann eine Kombination aus beiden Ansätzen.

Wir gehen wir mit Veränderungen am API um?

Wenn ein API erstmal in Betrieb ist und von Konsumenten aktiv verwendet wird, beginnt die Herausforderung der Weiterentwicklung. Einerseits möchte man das API den sich ändernden Businessanforderungen anpassen und basierend auf den neuen Anforderungen das bestmögliche API zur Verfügung stellen. Andererseits möchte man die bereits existierenden Konsumenten natürlich behalten und nicht durch Breaking Changes vergraulen. Deshalb sollte man dafür sorgen, dass das alte API weiterhin funktioniert. Tut man das, ist man allerdings sehr schnell mit hohen Wartungsaufwänden für alte APIs beschäftigt, die keinen Mehrwert für das Produkt bringen.

Häufig wird bei der API-Entwicklung zwischen öffentlichen und nicht-öffentlichen Schnittstellen unterschieden. In der Praxis gibt es aber viel mehr (und feingranularere) Unterscheidungen von APIs. Diese hängen vor allem vom Konsumenten ab und beeinflussen, in welcher Art und Weise man das API weiterentwickeln kann.

Wie bereits oben erwähnt: Wenn der Client ein Web-Frontend im selben Deployment ist, muss man sich bei der Weiterentwicklung keine Gedanken machen, weil mit jedem neuen API auch gleich der neue Client ausgeliefert wird. Er ist somit immer kompatibel. Komplizierter wird es, wenn es externe Konsumenten gibt.

Sofern der Konsument ein anderer (Micro-)Service in derselben Gesamtanwendung ist, möchte man zumindest sicherstellen, dass beide Services unabhängig deployt werden können. Das alte API (bzw. die alte Version des API) muss also noch so lange funktionieren, bis der Konsument auch ein Update durchgeführt hat.

Je nach Art des Konsumenten kann das früher oder später geschehen. Während man in einer Microservices-Architektur ggf. noch davon ausgehen kann, dass ein solches Update innerhalb kürzester Zeit durchgeführt wird, kann das Update bei Clients, die z. B. auf einem mobilen Endgerät installiert sind, deutlich länger auf sich warten lassen, selbst wenn diese nur innerhalb desselben Unternehmenskontexts eingesetzt werden.

Handelt es sich bei dem API um eine B2B-Schnittstelle, müssen vermutlich noch längere Fristen berücksichtigt werden, in denen das alte API weiterhin zur Verfügung stehen muss. Die Fristen sind ggf. in Verträgen festgehalten, an die man gebunden ist. Und bei einem öffentlichen API wird eine alte Version idealerweise niemals abgekündigt, um keine alten Clients zu verlieren.

Bei der Weiterentwicklung eines API hat man also drei konkurrierende Ziele:

- Das API soll immer die aktuellen Businessanforderungen widerspiegeln.
- Alte Clients sollen weiterhin funktionieren.
- Der Wartungsaufwand soll nicht linear mit jeder neuen Version steigen.

Ziel ist es also, APIs so weiterzuentwickeln, dass im Idealfall alle drei Ziele erreicht werden. Um ein API kontinuierlich weiterentwickeln zu können, ohne dass die alten Clients nicht mehr benutzt werden können, werden allerdings nicht nur Anforderungen an das API gestellt, sondern auch an die Clients.

Tolerant Reader Pattern

Ein berühmtes Konzept, das besagt, wie sich der Consumer verhalten soll, damit er bei Änderungen des API robust reagiert und weiterhin funktioniert, ist das Tolerant Reader Pattern. Grundsätzlich geht es dabei darum, dass der Consumer nicht auf ein Schema der Datenstrukturen des Providers festgelegt ist, sondern mit gewissen Änderungen des Schemas umgehen kann.

Wenn es um die konkrete Umsetzung das Tolerant Reader Patterns geht, gibt es im Detail allerdings Unterschiede, mit welchen Änderungen der Consumer umgehen können muss. Sinnvoll ist natürlich, wenn der API-Provider vorher spezifiziert, welche Art von Änderungen der Consumer zu erwarten hat und welche er folglich tolerieren muss. Eine gute Referenz bieten dazu die Zalando-API-Guidelines [2]. Die Kernforderung dort ist, dass der Consumer mit zusätzlichen Attributen klarkommen muss (indem er sie z. B. ignoriert). Tatsächlich handelt es sich hierbei um eine sehr mächtige Forderung, da damit sowohl die Umbenennung von Attributen als auch das Verschieben und bis zu einem gewissen Grad auch das Teilen und Zusammenfügen von Attributen bis hin zur Berechnung komplexer Attribute realisiert werden kann [3].

Consumer-driven Contracts zum Abschalten eines API

Insbesondere, wenn es sich nicht um ein öffentliches API handelt, möchte man eine alte Version irgendwann auch abschalten können. Da stellt sich die Frage, wie der Provider eines API darüber informiert wird, dass alle Konsumenten bereits die neue Version verwenden.

Eine Möglichkeit, um das zu erreichen, sind Consumer-driven Contracts. Bei diesem Konzept stellen alle Konsumenten eines API dem Provider einen sogenannten Consumer Contract zur Verfügung, der die Information enthält, welchen Teil (und damit auch welche Version) des API sie verwenden.

Um sicherzugehen, dass die Consumer Contracts immer aktuell sind, erfolgt die Erstellung und auch die Bereitstellung automatisiert. Wann immer ein Client ein Deployment auf einer Stage durchführen möchte, führt er automatisierte Tests gegen einen Mock des API durch, um sicherzustellen, dass der neue Code noch mit dem API funktioniert. Dabei passiert automatisch ein Recording der durchgeführten Aufrufe. Aus diesem Recording werden die Consumer Contracts generiert. Beim Deployment werden diese an zentraler Stelle zur Verfügung gestellt und mit der Information versehen, auf welcher Stage der Consumer deployt wurde. Auf diese Weise kann der Provider jederzeit erkennen, welcher Teil des API auf welcher Stage verwendet wird und so ggf. alte Teile des API abschalten, ohne Angst haben zu müssen, dass ein alter Konsument nicht mehr funktioniert.

Was bringt die Zukunft im Bereich API-Design?

Wie APIs der Zukunft aussehen, lässt sich natürlich schwer vorhersagen, aber einige neue Ansätze erfreuen sich aktuelle großer Beliebtheit und es lohnt demnach, auch einen Blick über den Tellerrand zu wagen.

Automatischer API-Layer: Ein aktueller Trend ist die automatische Generierung von APIs, basierend auf vorhandenen Modellen. Ein Modell, das nahezu jede Anwendung besitzt, lebt in der Datenbank. Die Idee ist, auf einen weiteren händisch entwickelten Layer in der Architektur zu verzichten und den bestehenden Layer, wie etwa die Datenbank, automatisch via API zur Verfügung zu stellen. Einige bekannte Vertreter dieser Ansätze sind:

- Hasura – GraphQL API, basierend auf dem Postgres-Datenbankschema
- PostgREST – ähnlich wie Hasura, aber es wird ein REST API bereitgestellt
- Spring Data REST – REST API mit Operationen basierend auf Repositorymethoden

Hier steht vor allem der Pragmatismus im Vordergrund. Da bereits entsprechende Modelle in der Datenbank oder Anwendung existieren und diese in vielen Fällen für Konsumenten bereits in einer brauchbaren Form

vorliegen, spart man sich den Aufwand, eine dedizierte Schnittstelle zu entwerfen. Neue Anforderungen werden nicht im API-Layer umgesetzt, sondern stattdessen in der Datenbank, etwa mit Hilfe von SQL Views, um so die benötigten Daten bereitstellen zu können.

Technologien wie Hasura und Co. können dann sinnvoll sein, wenn die Anwendung nur wenig Businesslogik enthält und eher die Daten statt Prozesse im Fokus stehen.

Die wenigen Businessprozesse, die existieren, lassen sich zusätzlich über einen schmalen API-Layer umsetzen, sodass sichergestellt wird, dass sie die gewünschte Businesslogik durchlaufen. So kann vermieden werden, dass auch ohne größeren API-Layer die eigentliche Datenbank stets in einem konsistenten Zustand bleibt.

Gerade für Prototyping bieten diese Ansätze einen enormen Geschwindigkeitsvorteil, um schnell zu einer lauffähigen Anwendung zu kommen.

Verwendet man diesen Ansatz, sollte man allerdings darauf achten, nicht die alten Fehler aus der SOA-Welt zu wiederholen, indem man zentrale Entity Services einführt, von denen dann erstens alle anderen Services abhängen, und die dann zweitens komplett von der auf ihnen operierenden Businesslogik entkoppelt sind. Beides hat sich aus mehreren Gründen als unpraktikabel erwiesen.

Kein API: Eine ganz andere Richtung, die zunehmend an Beliebtheit gewinnt, und damit gewissermaßen auch eine Art Wiedergeburt darstellt, ist der gänzliche Verzicht auf ein API – insbesondere im Umfeld von browserbasierten Anwendungen. Statt strikter Trennung zwischen Frontend (in Form einer Single Page Application) und Backend, mit ausschließlicher Kommunikation via API, gerät das eher klassische, serverseitige Rendern des UI wieder in den Vordergrund.

Zwar lassen sich mit etablierten Technologien wie JSF oder Spring MVC schon seit Ewigkeiten auch UI-basierte Anwendungen entwickeln, bei denen das UI größtenteils im Backend generiert wird, aber für dynamische und vor allem interaktive Inhalte muss mitunter immer noch auf eine Kombination aus clientseitigem JavaScript und einem passenden API im Backend gesetzt werden.

Neue Technologien versuchen, die serverseitige Erstellung des UI mit dynamischen und interaktiven Inhalten zu verbinden – meist so, dass selbst Interaktion, die nahezu in Echtzeit erfolgen sollen, gänzlich im Backend umgesetzt werden.

Während etwa für eine Typeahead-Eingabe üblicherweise ein entsprechendes API zur Suche von passenden Einträgen bereitgestellt wird, das bei jedem Tastenanschlag aufgerufen wird, wird stattdessen der Tastenanschlag als solches ans Backend gesendet. Das Backend würde dann einfach die entsprechende Suche starten und lediglich Veränderung des UI, konkret die angepasste Liste von Einträgen im Typeahead, an den Browser schicken.

Das Backend kontrolliert somit sowohl die eigentlichen Daten als auch das UI, ohne dazwischen einen dedizierten API-Layer zu platzieren. Die notwendige Logik im Browser, um diesen Ansatz zu realisieren, wird dabei durch das entsprechende Framework vor dem Entwickler versteckt.

Als Vorreiter gilt hier das Elixir-basierte Framework Phoenix, das diesen Ansatz mit Hilfe von LiveView implementiert. Auch für etabliertere Frameworks wie Laravel mit Hilfe von LiveView oder Ruby on Rails mit StimulusReflex gibt es entsprechende Umsetzungen.

Insbesondere in Fällen, wo Frontend und Backend sehr stark gekoppelt sind, ist es überlegenswert, ob dieser Ansatz sinnvoll ist, da nicht nur auf ein API verzichtet werden kann, sondern mitunter auch auf ein weiteres Ökosystem, etwa auf die Verwendung von JavaScript im Browser.

Fazit

Um ein passendes API erstellen zu können, muss man sich viele Fragen stellen. Alle diese Fragen richtig zu beantworten, ist selten möglich – gerade bei der initialen Entwicklung des API. Viele Antworten werden erst im Lauf der Nutzung durch die Konsumenten ersichtlich.

Ein API ist daher oft etwas Organisches, das mit der Zeit erst wachsen muss. Niemand kann mal eben so das perfekte API erstellen. Anforderungen von heute decken sich nicht notwendigerweise mit denen von morgen. Daher ist es wichtig, auch die Weiterentwicklung des API von vornherein im Blick zu haben, etwa durch die Forderung des Tolerant Reader Pattern bei den Konsumenten. Unabhängig davon sollten die Konsumenten stets im Blick behalten werden. Im Idealfall werden die oft unausweichlichen Änderungen gemeinsam erarbeitet oder mindestens frühzeitig kommuniziert.

Die vielen technologischen Möglichkeiten sollten nur dann ausgeschöpft werden, wenn die etwaigen Vorteile verstanden werden und die damit oft einhergehende technische Komplexität zu rechtfertigen ist. Im Zweifel sollte die Wahl stets auf die einfachste und verbreitetste Technologie fallen. Um ein bekanntes Zitat zu verfälschen: Nobody ever got fired for building a RESTful API.

Renke Grunwald ist Enterprise-Entwickler bei der OPEN KNOWLEDGE GmbH in Oldenburg. Zu seinen Schwerpunkten gehören moderne Frontend-Technologien (insbesondere im Kontext von SPAs), neuartige Architekturansätze wie Microservices sowie das Thema Continuous Integration/Deployment. Die Mission seiner Arbeit ist immer, den Prozess der Softwareentwicklung zu verbessern und zu vereinfachen.

Arne Limburg ist Softwarearchitekt bei der OPEN KNOWLEDGE GmbH in Oldenburg. Er verfügt über langjährige Erfahrung als Entwickler, Architekt und Consultant im Java-Umfeld und ist auch seit der ersten Stunde im Android-Umfeld aktiv.

Links & Literatur

[1] https://github.com/eclipse/microprofile-open-api

[2] https://opensource.zalando.com/restful-api-guidelines/

[3] https://entwickler.de/api/kolumne-enterprisetales-039

Ein praxistaugliches Qualitätsmodell

Quality-driven Software Architecture

Kurze Zusammenfassung der ersten Folgen: Bei Entwurf und Implementierung von Systemen sollten wir auf ein vernünftiges, angemessenes Verständnis der von Stakeholdern gewünschten oder geforderten Qualität zurückgreifen können. Dafür müssen wir zwei Dimensionen klären: Welche Eigenschaften (Dimension-1) sind in jeweils welcher Ausprägung (Dimension-2) für unsere Systeme relevant? Klingt kompliziert, doch ist praktisch zum Glück recht einfach machbar.

von Dr. Gernot Starke

Nachdem wir in der ersten Folge ein paar Herausforderungen mit dem Begriff „Qualität“ kennengelernt haben und uns in der zweiten Folge (sehr) kritisch mit dem aktuellen ISO-Standard 25010 zur Softwarequalität auseinandergesetzt haben, gehen wir jetzt in den konstruktiven Modus über. Wir geben einen kompakten Überblick über Qualitätsmodelle. Dann finden Sie ein pragmatisches und praxistaugliches Modell, das den Weg für konkrete Qualitätsanforderungen mit Hilfe der so genannten Qualitätsszenarien bereitet.

Trauriger Stand der Praxis

Nach meiner Erfahrung aus mehr als 30 (dreißig!) Jahren praktischer Softwareentwicklung liegen Qualitätsanforderungen im Projektalltag aus unterschiedlichen Gründen nur selten vor. Stakeholder

- wissen selbst (noch) nicht, welche Eigenschaften das System haben soll – weil es beispielsweise um eine Produktinnovation geht und niemand heute wissen kann, wie das System vom Zielpublikum aufgenommen oder verwendet werden wird,
- möchten sich (noch) nicht festlegen, aus Angst, diese Entscheidung wäre irreversibel,
- halten gewisse Eigenschaften für völlig selbstverständlich und sprechen diese Themen daher nicht an
- übersehen, dass Entwicklungsteams manche Anforderungen kennen müssen

Als Entwicklungsteam können Sie jetzt entweder die Produktverantwortlichen (aka product owner) erziehen oder selbst Hand anlegen und die Qualitätsanforderungen

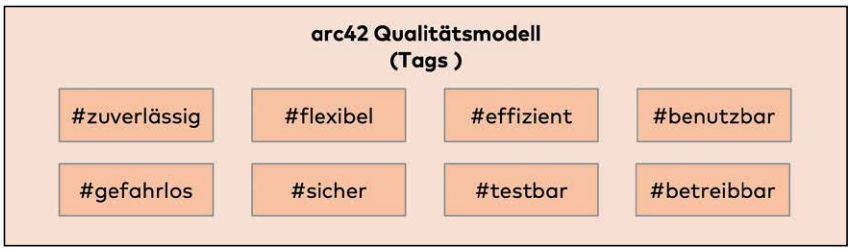

Abb. 1: Das pragmatische Qualitätsmodell von arc42

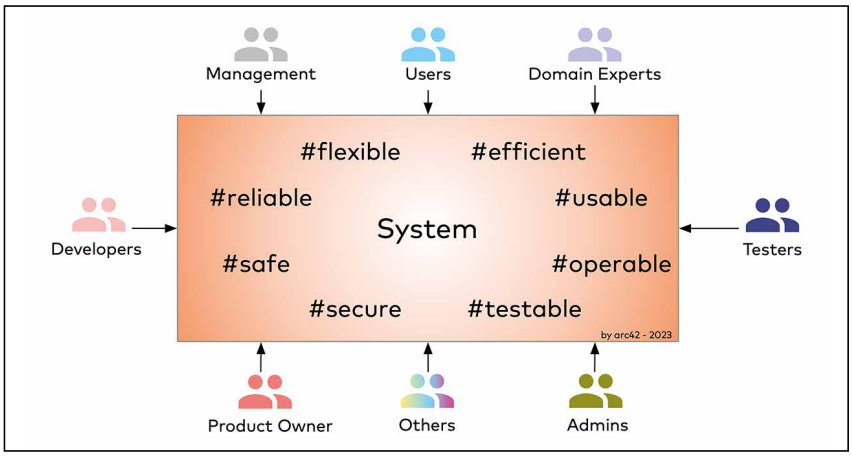

Abb. 2: Englische Version der Qualitätsbegriffe

aktiv erarbeiten. Das fällt Ihnen erheblich leichter, wenn Sie eine praxistaugliche Checkliste einsetzen können, und genau da kommt unsere „Dimension-1“ ins Spiel, eine Sammlung häufig geforderter Qualitätseigenschaften von Software (sprich: ein vernünftiges Qualitätsmodell).

Dimension-1: Welche Eigenschaften

Lange Zeit war es in der Branche sehr umstritten, welche Eigenschaften von Software (oder Systemen im Allgemeinen) für die Entwicklung relevant sein könnten. Als Ergebnis entstanden in der Vergangenheit unterschiedliche Qualitätsmodelle (Kasten: „Qualitätsmodelle: Der Blick in die Geschichte“) von denen der ISO-Standard 25010 (die letzte Folge dieser Kolumne hat das ausführlich vorgestellt) die wohl größte Verbreitung erreichen konnte. ISO-25010 ist jedoch einerseits ziemlich in die Jahre gekommen (der Standard wurde 2011 nach jahrelanger Gremienarbeit veröffentlicht) und wirkt andererseits mit über 40 Begriffen auf zwei Ebenen überladen. Essenzielle Merkmale (wie Time-to-market, Energieeffizienz oder Deploybarkeit) fehlen, und die Begriffsdefinitionen sind teilweise sehr sperrig.

Da möchte ich konstruktiv nachlegen und das arc42-Qualitätsmodell vorstellen, das methodisch neue Wege geht (**Abb. 1**, englische Version in **Abb. 2**). Der erste Unterschied zu bekannten Modellen ist die Wahl von Adjektiven (statt Substantiven wie alle übrigen Q-Modelle) auf der obersten Ebene. Mir war seit Jahren unklar, warum die Verantwortlichen von ISO und anderen Qualitätsmodellen zwar über „Qualitätseigenschaften“ schreiben, dafür aber stets Substantive verwenden.

Ein zweiter grundlegender Unterschied besteht im Verzicht einer festen Hierarchie: In bestehenden Model-

Qualitätseigenschaft	Erklärung	Q42-Tags
Administrierbarkeit	Notwendiger Aufwand zur Administration (Verwaltung, Überwachung) einer Software im laufenden Betrieb	#betreibbar
Änderbarkeit	Aufwand, der zur Durchführung vorgegebener Änderungen notwendig ist. Änderungen sind Korrekturen, Anpassungen der Umgebung und Infrastruktur, der Anforderungen, der internen Struktur, der Implementierung o. ä.	#flexibel, #effizient
Autonomie	Fähigkeit eines Systems, sein Leistungsniveau unabhängig von anderen Systemen zu erbringen	#betreibbar, #zuverlässig
Durchsatz	Menge an Daten oder Ereignissen, die das System in gegebener Zeit verarbeiten kann	#effizient, #benutzbar
Latenz	Synonym: Verzögerungszeit; Zeit vom Ende eines Ereignisses bis zum Beginn der Reaktion auf dieses Ereignis	#effizient, #benutzbar
Modularität	Zerlegung eines Systems in Einzelteile mit definierten Schnittstellen und Aufgaben	#flexibel, #effizient
Portabilität	Plattformunabhängigkeit, auch: Übertragbarkeit	#flexibel
Wiederherstellbarkeit	Fähigkeit, bei einem Versagen das Leistungsniveau wiederherzustellen und die direkt betroffenen Daten wiederzugewinnen	#zuverlässig

Tabelle 1: Qualitätseigenschaften und ihre Zuordnung zu arc42-Tags

len (z. B. ISO-25010) besteht häufig die Schwierigkeit, zu welchem der Top-Level-Merkmale eine bestimmte Qualitätsanforderung gehört. Nehmen Sie beispielsweise die Anforderung „Suche nach Sonderangeboten muss während der Kernzeit 9-17 Uhr ständig verfügbar sein". Gehört diese zu Verfügbarkeit oder Benutzbarkeit? Meiner Ansicht nach zu beidem – das ist im starren ISO-Modell aber nicht vorgesehen.

Das arc42-Modell Q42 (gesprochen „Kju-Fortytwo" oder „Kju-Four-Two") geht hier einen anderen Weg: Statt sich auf eine feste Hierarchie von Begriffen zu fixieren, setzt arc42 auf (zurzeit) acht „Tags" (Etiketten). Das ermöglicht es, eine konkrete Qualitätsanforderung mit mehreren dieser Tags auszuzeichnen.

Einstieg: Das bedeuten diese „Tags"

Die Tags des arc42-Qualitätsmodells sollten nahezu selbsterklärend sein, dennoch versuche ich in Kurzform mal eine Klärung durch Beispiele:

- *#zuverlässig:* verfügbar, robust, ausfallsicher, fehlerfrei, wiederherstellbar, konsistent
- *#flexibel:* Zur Entwicklungs- und Laufzeit leicht änderbar, anpassbar, skalierbar, wiederverwendbar, modular

Qualitätsmodelle: Der Blick in die Geschichte

Seit circa 1977 streiten sich die (wissenschaftlichen) Geister um eine Begriffsordnung rund um das Thema Softwarequalität [2]. McCall schlug 1977 eine Hierarchie vor, deren erste Ebene die drei Elemente *Operation*, *Revision* und *Transition* umfasst. Zu Revision zählte er z. B. Maintainbility, Flexibility und Testability.
Danach gab's eine Reihe alternative Vorschläge, von denen aus meiner Sicht der Ansatz FURPS [4] und dessen Erweiterung FURPS+ von IBM [5] in der Praxis ordentlich Anklang fand. Die ISO-Organisation hat sich dann des Themas angenommen, und veröffentlicht seit 1991 hersteller- und produktneutrale Standards, beginnend mit ISO-9126. Dieser hat 20 Jahre lang „gehalten" und wurde 2011 durch den noch heute gültigen ISO-25010 abgelöst. Beide galten jeweils als begriffliche Referenz für Softwarequalität und haben erhebliche Verbreitung in der Praxis gefunden. Meiner bescheidenen Meinung nach liegt das in erheblichem Maße daran, dass die ISO-Gremien die verwendeten Begriffe allesamt recht ordentlich definieren. Deswegen sei ihnen auch verziehen, dass sie einige wirklich relevante Qualitätsmerkmale schlichtweg ignoriert (oder vergessen?) haben.
Gut gefallen haben mir auch Qualitätseigenschaften des VOLERE Requirements Template [6], die jedoch im deutschsprachigen Raum weniger Akzeptanz gefunden haben als die ISO-Standards.
Mein Fazit: Die Erklärung von 30-40 Untereigenschaften der ISO- oder FURPS+-Modelle braucht mehr Zeit, als die meisten Entwicklungsteams dafür aufbringen möchten. Einfachere Modelle sind gefragt, am liebsten Open Source (das bietet arc42 ja schon lange).

Autor:innen,	Jahr	Art	Anzahl der Eigenschaften	Eigenschaften
McCall	1977	Hierarchisches Modell, zwei Ebenen Einstiegspunkte Operation, Revision, Transition	11	Kaum Unterschied zwischen Revision und Transition
Barry Boehm	1978	Hierarchisches Modell, drei Ebenen Einstiegspunkte Utility, Maintainability, Portability	23	
ISO 9126	1991	Zweistufiges Modell, sechs Top-Level-Eigenschaften	27	Security nur nachgelagert
R. Grady, „FURPS"	1992	Einstufiges Modell, Functionality, Usability, Reliability, Performance, Supportability	5	
IBM FURPS+	1999	Erweitert das FURPS-Modell um zahlreiche Untermerkmale	ca. 30	Teil des (zu) umfangreichen Rational-Unified Process
VOLERE	1999	In umfassendes Requirements Template integriert	8	Kombiniert Q-Anforderungen mit Constraints
ISO 25010	2011	Hierarchie mit zwei Ebenen, acht Top-Level-Merkmale	32	Teils unpraktikable Begriffsdefinitionen, nicht mehr zeitgemäß
SEI	2022	Eine Ebene, siehe [1]	10	Manche Details zu hoch priorisiert
ISO 25010-2022 (Draft-Version)	2022	Erweitert auf neun Top-Level-Merkmale	9	39 Sub-Merkmale, teils überschneidend

Qualitätsmodelle im Quick-Check

- *#effizient:* schnell, energie-, speicher- und überhaupt ressourceneffizient, mit angemessener Kapazität
- *#benutzbar:* leicht erlernbar, hilfreich, selbsterklärend, attraktiv, fehlervermeidend
- *#gefahrlos:* Erfüllt Safety-Anforderungen (Kasten: „Das Übersetzungsproblem mit Safety und Security"), erkennt und warnt vor Risiken und Gefahren
- *#sicher:* Erfüllt Anforderungen an Datenschutz und -sicherheit, vertraulich, authentisch, integer, verantwortlich
- *#testbar:* prüfbar, analysierbar, automatisiert und manuell angemessen einfach zu testen
- *#betreibbar:* installierbar, überwachbar, administrierbar

Funktioniert das in der Praxis?

Das arc42-Qualitätsmodell habe ich einer umfangreichen Validierung unterzogen: Möglichst viele mögliche Qualitätseigenschaften müssen sich mit den Tags des arc42-Qualitätsmodells zuordnen lassen. Dazu habe ich circa 100 Namen und Definitionen von Qualitätseigenschaften gesammelt (Basis dafür war [7]) – danke dem Wortreichtum der deutschen Sprache. Die Zuordnung funktioniert nahezu perfekt – die Details finde Sie online bei [8]. Tabelle 1 zeigt einen kurzen Auszug davon. Sie können [8] als Checkliste für eigene Qualitätsanforderungen verwenden.

Fazit

Sie haben ein pragmatisches Qualitätsmodell gesehen, das mit nur neun Qualitätseigenschaften („Tags") auskommt. Das ist für Architektur und Entwicklung relevant großer IT-Systeme meiner Meinung die minimale Menge. Statt auf große und schwer verständliche Hierarchien von Begriffen zu setzen (wie ISO-25010), geht das arc42-Qualitätsmodell den Weg über „Tags", die als echte Eigenschaften formuliert sind. Damit haben wir (endlich!) ein praktisches Modell für die Frage nach Dimension-1 („Welche Eigenschaften?"). Beim nächsten Mal werden wir dann noch konkreter – und beschreiben ein paar reale Qualitätsanforderungen im Detail.

Das Übersetzungsproblem mit Safety und Security

Die englische Sprache differenziert zwischen den Begriffen Safety und Security – die leider im Deutschen beide mit „Sicherheit" übersetzt werden:

- *Safety:* Vermeidung von Schäden, Verletzungen oder Gefahren für Menschen und Systeme
- *Security:* Vertraulichkeit, Integrität und Konsistenz von Daten oder Prozessen

Die einfache Übersetzung „Sicherheit" verliert damit einen Teil der möglichen Bedeutung. Eine treffendere Übersetzung wäre „Gefahrlosigkeit" (oder „Harmlosigkeit"). Erstere Version habe ich auch im arc42-Qualitätsmodell gewählt.

Bis dahin alles Gute – möge die Macht hoher Qualität mit Ihnen sein!

Dr. Gernot Starke (INNOQ Fellow, (Mit-)Gründer von arc42.org und aim42.org, Gründungsmitglied des iSAQB) arbeitet als Coach für Softwarearchitektur. Systematischer Umgang mit Qualität gehört seit Jahren zu seinen beruflichen Schwerpunkten.

Links & Literatur

[1] Bass et. Al: „Software Architecture in Practice", Fourth Edition, Addison-Wesley 2022 (hier ist wirklich die vierte Auflage wichtig, weil darin das Thema Qualität signifikant gegenüber vorigen Auflagen erweitert wurde).

[2] Jamwal et al.: „Comparative Analysis of Different Software Quality Models. Proceedings of the 3rd National Conference"; INDIACom-2009; https://enos.itcollege.ee/~nafurs/suvi2019/huvitavat/alternatiivid_FURPSile.pdf

[3] Grady, L. R. B.: „Practical Software Metrics for Project Management and Process Improvement", enthält das FURPS-Modell Prentice Hall, 1992

[4] ISO-25010: https://www.iso.org/standard/35733.html

[5] Eeles, Peter: „Non-functional Requirements": https://pdfs.semanticscholar.org/f3bb/91080c4573f6f78f30bc5b48bda3ef252bf2.pdf

[6] VOLERE Requirements Template: https://www.volere.org/

[7] arc42 Quality Requirements: Beispiele konkreter Qualitätsanforderungen. Wird mittelfristig durch [8] ersetzt: https://github.com/arc42/quality-requirements.

[8] arc42 Quality Model: https://quality.arc42.org (Draft) Definitionen wesentlicher Qualitätseigenschaften, Mapping zu den arc42-Quality-Tags