Ausgabe 04/2019

Java aktuell

Praxis. Wissen. Networking. Das Magazin für Entwickler

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

4 191978 304903 04

Java aktuell

iJUG Verbund
www.ijug.eu

Java 12
Die neuen Features im Überblick

Jakarta EE
Neuigkeiten und Ergebnisse der Verhandlungen

Web-APIs
Diese Programmierschnittstellen erleichtern Ihnen die Arbeit

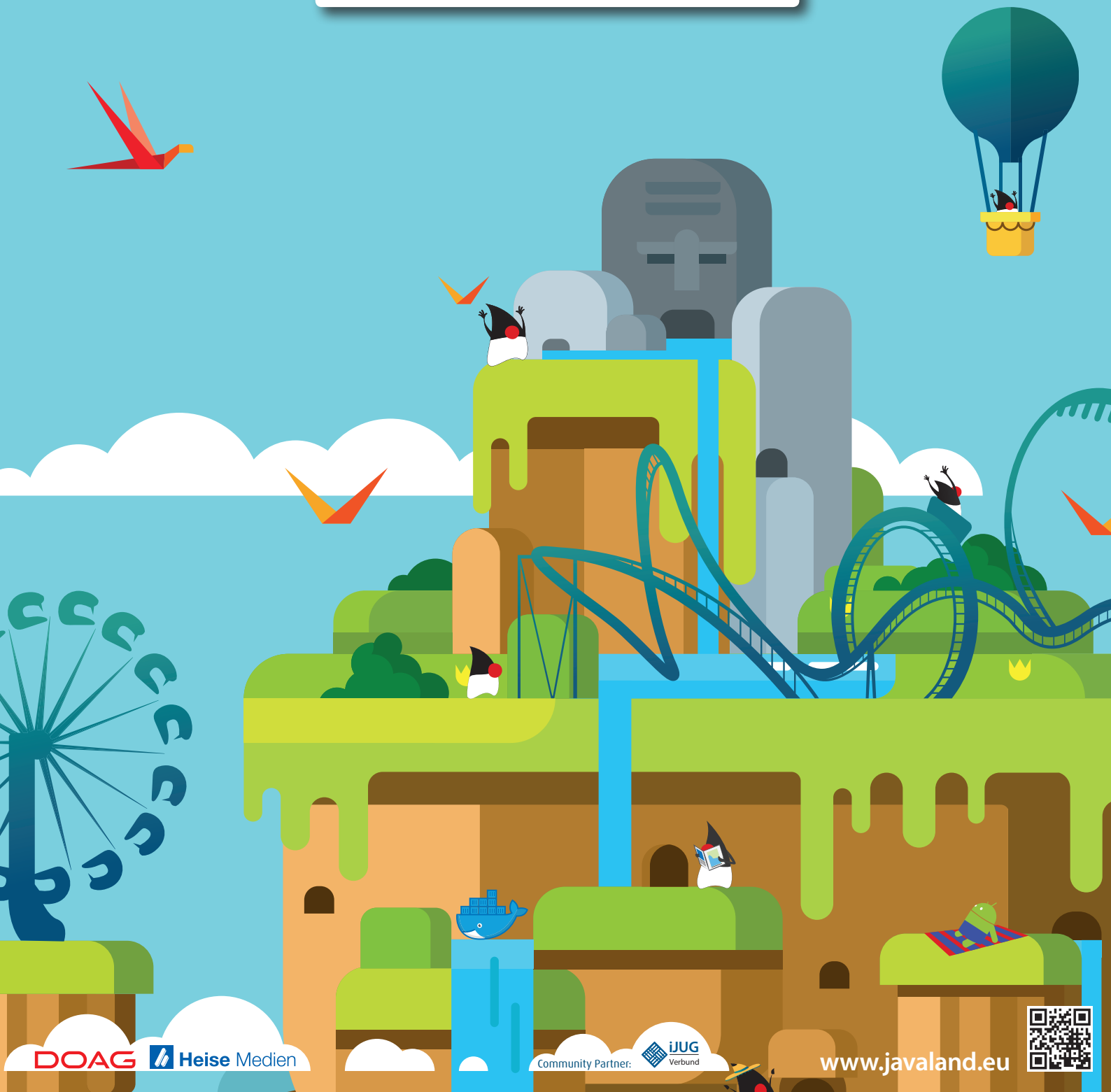
Java im Wandel

JavaLand

Save the Date

16. - 19. März 2020

in Brühl bei Köln



DOAG Heise Medien

Community Partner: iJUG Verbund

www.javaland.eu

Mit Spring, Docker und Kubernetes nach Produktion schippern

Timm Hirsens, Atlas Dienstleistungen für Vermögensberatung

Beispiele, um eine Spring-Boot-Anwendung in einem Docker-Container zu deployen, gibt es wie Sand am Meer. Aber reicht das auch für einen entspannten Produktionsbetrieb in Kubernetes? Lassen sich Docker und Java so ohne Weiteres kombinieren? Welche Fallen lauern hier? Wie bauen wir verschiedene Umgebungen mit Kubernetes auf? Wie konfigurieren wir unsere Anwendung am besten für diese?

Gründe dafür, Spring-Boot-Anwendungen auf einem Kubernetes-Cluster zu betreiben, gibt es reichlich. Damit es bei diesem Vorhaben keine bösen Überraschungen gibt, sollten einige Dinge jedoch unbedingt beachtet werden.

Das richtige Docker-Image

Damit wir mit unserer Spring-Boot-Anwendung mithilfe von Kubernetes nach Produktion schippern können, müssen wir sie zunächst in einen Container verpacken. Nach einer kurzen Google-Suche ist die vermeintliche Lösung auch schnell gefunden.

Das Ergebnis dieses Dockerfiles *(siehe Listing 1)* ist durchaus ein funktionierendes und korrektes Docker-Image. Allerdings gibt es hier auch noch einiges an Verbesserungspotenzial, wenn unser Ziel ein möglichst reibungsloser Betrieb in Produktion ist. Beginnen wir mit der ersten Zeile des Dockerfiles. Im Beispiel erben wir von dem Docker-Image `openjdk:8-jdk-alpine`. Bei den Images mit dem Namen „openjdk" handelt es sich um offizielle Images aus dem Docker Hub. Grundsätzlich macht man hier also nicht allzu viel falsch. Das Basis-Image aus dem Beispiel verwendet „Alpine Linux" als Unterbau, hierbei handelt es sich um eine Linux Distribution, die gerne für den Betrieb als Container verwendet wird. Sie besticht besonders durch ihre geringe Größe von lediglich rund 7 MB. Diese Größe gibt es allerdings nicht umsonst; in einem Alpine-Image ist statt einer vollständigen GNU/Linux-Umgebung nur „busybox" enthalten und statt der C-Bibliothek „glibc" enthält Alpine Linux die Bibliothek „muslc". Besonders bei der Arbeit mit JNI (Java Native Interface) kann dieser Unterschied an Bedeutung gewinnen.

Wenn ein Container dann doch mal durch einen Debugger betrachtet werden soll, ist es durchaus nützlich, wenn man sich in ihm auch zurechtfindet und die nötigen Tools schon installiert sind. Daher ergibt es durchaus Sinn, das Container-Image etwas größer werden zu lassen. Als Basis eignet sich beispielsweise ein CentOS. CentOS ist im Vergleich zu Alpine Linux mit rund 75 MB zwar wesentlich größer, allerdings wiegt ja selbst die JRE/JDK 100 bis 250 MB. Daher sind die Einsparungen auf Betriebssystem-Ebene in den meisten Fällen eher zu vernachlässigen. Ein einfaches Basis-Image für unsere Java-Anwendungen könnte daher so aussehen *(siehe Listing 2)*.

Wir erben von einem aktuellen CentOS-Image und installieren dort die aktuellste Java-Version. Wir verwenden hier das JDK, zum reinen Betrieb reicht jedoch auch eine JRE. Anschließend räumen wir noch ein wenig auf, damit das Image nicht unnötig groß wird. Die Angabe der Versionsnummer beziehungsweise des Tags in der `FROM`-Klausel

```
FROM openjdk:8-jdk-alpine
VOLUME /tmp
ARG JAR_FILE
COPY ${JAR_FILE} app.jar
ENTRYPOINT ["java","-Djava.security.egd=file:/dev/./urandom","-jar","/app.jar"]
```

Listing 1

```
FROM centos:7.6.1810

LABEL maintainer="Timm Hirsens <mail@timmhirsens.de>"

RUN yum install java-11-openjdk-headless-11.0.2.7 java-11-openjdk-devel-11.0.2.7 --setopt=tsflags=nodocs -y \
    && rm -rf /var/cache/yum \
    && yum clean all
```

Listing 2

kann bei Dockerfiles weggelassen werden. Um fiese Überraschungen zu vermeiden, ist es sinnvoll, bei der Arbeit mit Docker-Images immer eine konkrete Versionsnummer anzugeben. Selbst bei der Verwendung eines Image als Build-Node sollte die Versionsnummer angegeben werden. Das bedeutet natürlich auch, dass hier regelmäßig manuelle Updates stattfinden müssen, um eventuelle Sicherheitslücken zu schließen.

Das Docker-Overlay-Dateisystem nutzen

Da wir jetzt ein gutes Basis-Image erstellt haben, wollen wir nun natürlich auch unsere Anwendung in einen Container verfrachten. In unserem Beispiel wurde dafür die einfachste Variante gewählt, es wurde unser Spring Boot Fat-JAR mithilfe des `COPY`-Befehls in das Docker-Image kopiert. Grundsätzlich spricht auch gegen dieses Vorgehen nichts, trotzdem können wir hier noch optimieren.

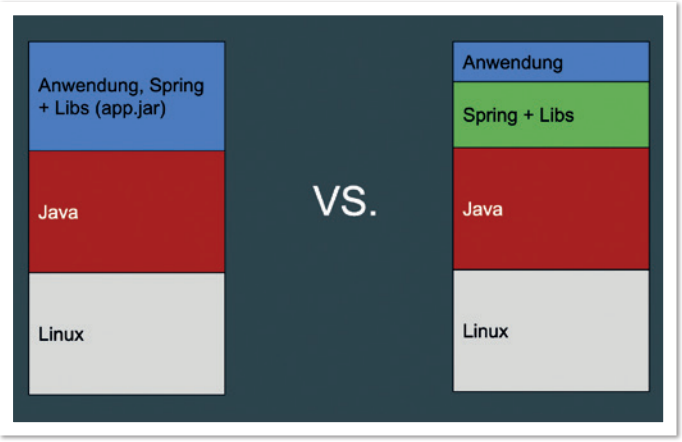


Abbildung 1: Vereinfachte Aufteilung des Container-Image in Schichten (Quelle: Timm Hirsens)

```
FROM mycompany.com/openjdk:11.0.2.7

VOLUME /tmp

EXPOSE 8080
EXPOSE 1099

ARG DEPENDENCY=build/dependency
COPY ${DEPENDENCY}/BOOT-INF/lib /app/lib
COPY ${DEPENDENCY}/org /app/org
COPY ${DEPENDENCY}/META-INF /app/META-INF
COPY ${DEPENDENCY}/BOOT-INF/classes /app
```

Listing 3

Docker-Images werden in Schichten aufgebaut, wobei jede Zeile in unserem Dockerfile eine neue Schicht bildet. Hierbei handelt es sich vor allem um eine Optimierungsmaßnahme für den Transport von Docker-Images. Wird ein Docker-Image aus einer Docker-Registry heruntergeladen, müssen nur die Schichten geladen werden, die der Anfrager noch nicht lokal kennt. Starten wir einen Docker-Container in unserem Kubernetes-Cluster, muss hier auch das Image aus der Registry geladen werden, aber eben nur die Schichten, die auf dem Kubernetes-Node noch nicht bekannt sind. Teilen wir unser Image also so ein, dass die Schicht mit den meisten Änderungen möglichst klein ist, können wir sowohl unsere Build- als auch unsere Deployment-Zeiten verkürzen. Das können wir erreichen, indem wir das Spring Boot Fat-JAR wieder entpacken und folgende Schichten bilden *(siehe Abbildung 1)*:

- Die Spring-Boot-Bibliotheken und ihre Abhängigkeiten
- Unsere Metainformationen („META-INF“)
- Unsere statischen Ressourcen
- Unsere eigenen Klassen

Somit bildet der Part unseres Fat-JAR, der sich am häufigsten ändert, unsere eigenen Klassen, die letzte Schicht im Docker-Image *(siehe Listing 3)*.

Kubernetes Deployment konfigurieren

Wenn wir unseren Container gebaut haben, kommt der nächste Schritt in Richtung Produktion, unser Container wird „verschifft“. Damit ein Container innerhalb eines Pod in einem Kubernetes-Cluster gestartet wird, gibt es verschiedene Möglichkeiten, diesen zu konfigurieren. In den meisten Fällen ist die Verwendung eines „Deployments“ die sinnvollste Variante. In einem solchen wird unter anderem angegeben, wie viele Instanzen („Replicas“) eines Containers gewünscht sind, wie diese gestartet werden sollen, welche Ressourcen zur Verfügung stehen und vieles mehr *(siehe Listing 4)*.

Beschränkung von Ressourcen

Es gibt zwei wichtige Optionen bei Kubernetes-Deployments, die für einen produktiven Betrieb auf jeden Fall angegeben werden sollten.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: spring-auf-kubernetes
  labels:
    app: spring-auf-kubernetes
spec:
  replicas: 2
  selector:
    matchLabels:
      app: spring-auf-kubernetes
  template:
    metadata:
      labels:
        app: spring-auf-kubernetes
    spec:
      containers:
        - name: spring-auf-kubernetes
          image: fr1zle/spring-auf-kubernetes:2cf65dbe
          ports:
            - containerPort: 8080
          args: [
            "/usr/bin/java",
            "-Djava.security.egd=file:/dev/./urandom",
            "-cp",
            "app:app/lib/*",
            "org.springframework.boot.loader.JarLauncher"
          ]
          readinessProbe:
            httpGet:
              path: /actuator/health
              port: 8080
            initialDelaySeconds: 30
            periodSeconds: 10
            timeoutSeconds: 3
          livenessProbe:
            httpGet:
              path: /actuator/info
              port: 8080
            initialDelaySeconds: 30
            periodSeconds: 10
            timeoutSeconds: 3
          resources:
            requests:
              cpu: 100m
              memory: 800Mi
            limits:
              cpu: 1
              memory: 800Mi
```

Listing 4

Dazu gehört als Erstes die Beschränkung der Ressourcen. Kubernetes bietet uns die Möglichkeit, im Gegensatz zu einem klassischen Application-Server, Ressourcen für einzelne Anwendungen sehr genau zu beschränken. Im Deployment geben wir an, wie viele CPU- und Arbeitsspeicherressourcen unserem Pod zur Verfügung stehen sollen. Hierzu stehen zwei Kategorien zu Verfügung: `Requests` und `Limits`. Für Ressourcen, die in `Requests` definiert werden, gibt der Kubernetes-Cluster eine Garantie, dass diese zum Start des Pod und zu seiner gesamten Lebensdauer auch zur Verfügung stehen.

Die Ressource-CPU wird in Tausenderschritten angegeben, „1000m" entspricht dabei einer CPU. Ein Pod, der „100m" CPU anfordert, erhält also ein Zehntel der CPU-Zeit eines Pod, der „1000m" oder „1" CPU anfordert. Die Anzahl der CPUs, die bei den `Requests` angegeben wird, berücksichtigt die JVM bei der Berechnung der verfügbaren CPU-Kerne. Hierbei wird auf den nächsten ganzen CPU-Kern gerundet. Die so errechneten CPU-Kerne werden von der JVM für die Berechnung der internen Threads (also für Just-In-Time Compilation, Garbage Collection usw.) herangezogen. Mit den `Limits` definieren wir, wie viel CPU unser Pod maximal verbrauchen darf. Wird mehr CPU verwendet, drosselt Kubernetes den Pod entsprechend. So ist auch im Fall eines Fehlers sichergestellt, dass eine Anwendung einer anderen nicht die CPU-Zeit „klauen" kann.

Für den Betrieb einer kleineren Spring-Boot-Anwendung reichen für gewöhnlich „100m" CPU. Der Start einer Spring-Boot-Anwendung kann allerdings, je nach verwendeten Bibliotheken, einiges an CPU benötigen. Um nicht unnötig viele Ressourcen zu verbrauchen, aber dennoch einen schnellen Start der Anwendung zu ermöglichen, geben wir unserem Pod ein `Request` von „100m" CPU und ein `Limit` von „1" CPU.

Die Berechnung der Arbeitsspeicherressourcen ist deutlich einfacher, sie werden einfach in Megabyte angegeben. Auch hier schaut die JVM seit neueren Versionen genauer hin; die Anzahl von Megabyte, die wir in den „Limits" definieren, werden für die Berechnung der einzelnen Speicherbereiche im Heap der JVM verwendet. Eine einfache Spring-Boot-Anwendung kommt mit etwa 800 MB gut aus. Die 800 MB werden sowohl bei `Requests` als auch bei den `Limits` eingetragen. Da die JVM die `Limits` zur Berechnung verwendet, müssen wir mit den `Requests` sicherstellen, dass der gewünschte Speicher auch zur Verfügung steht.

Java und cgroups

Die verfügbaren Ressourcen eines Docker-Containers werden über sogenannte „cgroups" (Control-Groups) gesteuert. In älteren Java-Versionen beachtet die JVM allerdings nicht die Ressourcen, die ihr durch eine „cgroup" zugeteilt werden, sondern die des Host-Computers. Hat also ein Kubernetes-Node 32 CPU-Kerne, berechnet die JVM anhand dieser Anzahl unter anderem, wie viele Kerne sie für die Garbage Collection verwenden kann. Damit das nicht passiert, sollte eine aktuelle Java-Version verwendet werden. Ab Java 11 oder Java 8 Update 191 verwendet die JVM standardmäßig die Einträge aus den „cgroups" für die Berechnung der CPU-Kerne und des verfügbaren Heap. Lässt sich der Einsatz einer älteren Java-Version nicht vermeiden, sollte unbedingt mindestens die Option `-XX:+UseCGroupMemoryLimitForHeap` gesetzt werden. Außerdem sollten die Threads des Garbage Collector entsprechend den verfügbaren Ressourcen konfiguriert werden.

Kubernetes-Probes konfigurieren

Die Kubernetes-Probes sind das zweite Feature, das in Produktion auf jeden Fall aktiviert werden sollte. Mit ihrer Hilfe wird sichergestellt, dass Anwendungen auch nach dem Deployment möglichst unterbrechungsfrei weiterlaufen. Die Aufgabe der `Liveness Probe` ist es, sicherzustellen, dass der in einem Container gestartete Prozess noch läuft und auf Anfragen reagiert. Ist dies nicht so, wird der entsprechende Pod gelöscht und ein neuer, identischer Pod aufgebaut. Die `Readiness Probe` hingegen stellt sicher, dass die Anwendung auch bereit ist, Anfragen entgegenzunehmen. An dieser Stelle wird sichergestellt, dass auch die externen Abhängigkeiten, wie eine Datenbank, verfügbar sind und erreicht werden können. Ist das nicht der Fall, wird der Pod aus dem Loadbalancing des Service genommen und erhält so lange keine Anfragen mehr, bis es wieder positive Antworten von der `Readiness Probe` gibt.

Beim Betrieb einer Spring-Boot-Anwendung in Kubernetes eignen sich besonders die `Actuator`-Endpunkte für die genannten Probes. Viele Beispiele verwenden hier den `Health`-Endpunkt für

beide Probes, was im produktiven Betrieb für die eine oder andere negative Überraschung sorgen kann. Da die beiden Probes ganz unterschiedliche Aufgaben erfüllen, sollte auch bei den `Actuator`-Endpunkten unterschieden werden. Für die `Liveness Probe` empfiehlt es sich, einen Endpunkt zu verwenden, der möglichst statisch eine positive Antwort (HTTP 200) liefert. Von den Spring-Boot-`Actuator`-Endpunkten eignet sich hier vor allem der Info-Endpunkt (`/actuator/info`); dieser liefert, sobald die Anwendung läuft, immer eine Antwort.

Für die Kubernetes „Readiness Probe" eignet sich der „Health"-Endpunkt. Dieser liefert für gewöhnlich einen Fehler, sobald eine Komponente, auf der eine Applikation eine Abhängigkeit hat, nicht mehr zur Verfügung steht. Üblicherweise zählen dazu Datenbank, Kafka-Cluster und externe Services, die für den Betrieb der Anwendung vonnöten sind. Steht eine der Komponenten nicht zur Verfügung oder ist für die Anwendung nicht erreichbar, erhält sie keinen weiteren Traffic mehr durch den Service. In den meisten Fällen ist dieses Verhalten auch so gewünscht, bei einigen Bibliotheken sollte allerdings darauf geachtet werden, ob diese einen eigenen Health Check registrieren. Wird beispielsweise ein Circuit Breaker aus der Bibliothek „resilience4j" eingesetzt, so wird standardmäßig auch ein Health Check hinzugefügt – mit der Folge, dass sobald ein Circuit geöffnet wird, die Applikation aufgrund des fehlerhaften Health Checks nicht mehr erreichbar wäre. Wenn hier ein Fallback-Szenario implementiert wurde, ist das natürlich nicht das gewünschte Verhalten. Daher sollten die angemeldeten Health Checks immer im Auge behalten werden.

Verschiedene Umgebungen mit Kubernetes

Für einen entspannten Produktionsbetrieb braucht es jedoch nicht nur eine stabile Produktionsumgebung, sondern auch noch mindestens eine zusätzliche Testumgebung, auf der Änderungen an den Applikationen getestet werden können. Bei der Verwendung von Kubernetes gibt es grundsätzlich zwei verschiedene Varianten, das Konstrukt „Umgebung" abzubilden. Die erste Variante ist die Lösung über sogenannte „Namespaces". In einem Kubernetes-Cluster können über Namespaces Ressourcen und Rechte eingeschränkt werden, außerdem können Services aus einem Namespace, je nach Konfiguration, nicht ohne Weiteres auf den Service eines anderen Namespace zugreifen. Eine Testumgebung ließe sich also durch einen zusätzlichen Namespace abbilden, hiervon ist allerdings eher abzuraten.

Sinnvoller ist es, für jede Umgebung einen eigenen Kubernetes-Cluster zu erstellen. Bei dieser Variante ist es quasi unmöglich, dass ein Fehler in der Testumgebung ernsthafte Auswirkungen auf die Produktionsumgebung hat. Nebenbei eignet sich die Testumgebung so auch für den Test eines Kubernetes-Updates oder dazu, um den Austausch einer Kubernetes-Komponente zu testen.

Staging und Konfiguration mit Spring Boot

Wenn unsere Anwendung nun auf mehreren Umgebungen laufen soll, muss sie sich natürlich auch entsprechend konfigurieren lassen. Glücklicherweise bringt auch hier Spring Boot schon einiges mit. Standardmäßig erfolgt die Konfiguration von Spring-Boot-Anwendungen über Konfigurationsdateien („application.properties" oder „application.yaml"). Über Profile können wir steuern, welche Konfigurationsdateien angezogen werden sollen. Die Datei „application.properties" wird immer angezogen, wenn wir das Spring-Profil `test` aktivieren. Indem wir beispielsweise die Umgebungsvariable `SPRING_PROFILES_ACTIVE` auf den Wert `test` setzen, wird zusätzlich auch die Datei „application-test.properties" angezogen. So können wir über das Setzen einer Umgebungsvariablen steuern, auf welcher Umgebung sich die Applikation aktuell befindet.

Wenn sich nur wenige Konfigurationen umgebungsabhängig ändern, reicht es unter Umständen völlig aus, diese einfach als Umgebungsvariable zu definieren. So wird aus der Property „spring.datasource.url" die Umgebungsvariable `SPRING_DATASOURCE_URL`. Die Daten aus Umgebungsvariablen haben dabei immer Vorrang vor den Daten aus der Konfigurationsdatei.

Die dritte Möglichkeit ist, dass die Konfigurationsdateien nicht in der Applikation enthalten sind, sondern über eine Kubernetes ConfigMap bereitgestellt werden. Diese ConfigMap kann dann als Volume in den Container als Mount geladen werden *(siehe „https://kubernetes.io/docs/tasks/configure-pod-container/configure-pod-configmap/#add-configmap-data-to-a-volume")*. Über die Umgebungsvariable `SPRING_CONFIG_LOCATION` kann dann der Pfad zu der ConfigMap konfiguriert werden.

Fazit

Wir sehen also, mit dem reinen Befolgen einiger Tutorials erhalten unsere Spring-Boot- und Docker-Anwendungen nicht unbedingt Produktionsreife. Mit der Berücksichtigung einiger Tücken und dem Einhalten von Best Practices erreicht man diese jedoch recht schnell. Sind im Unternehmen einmal sinnvolle Standards gefunden worden, sollten diese in eine Dokumentation oder in ein Template, beispielsweise mit Helm *(siehe „https://helm.sh/docs/helm/#helm-template")*, gegossen werden, damit nicht jedes Team diese Erfahrungen erneut machen muss. Dann steht einem entspannten Segeln in Produktion nichts mehr im Wege. In meinem GitHub-Account habe ich ein Projekt, in dem einige dieser Standards in Form eines Beispiels festgehalten sind *(siehe „https://github.com/fr1zle/spring-auf-kubernetes")*.

Timm Hirsens
hi@timmhirsens.de

Jetzt Anmelden!

Sei dabei

Go DevOps 2019

2.+3. Sept. 2019 in Berlin

DOAG

godevops.doag.org

esentri

IT'S IN ALL OF US *TO CREATE*

Jetzt bewerben unter
career@esentri.com

#inallofus
www.esentri.com