Ausgabe 02/2019

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 02

Java aktuell

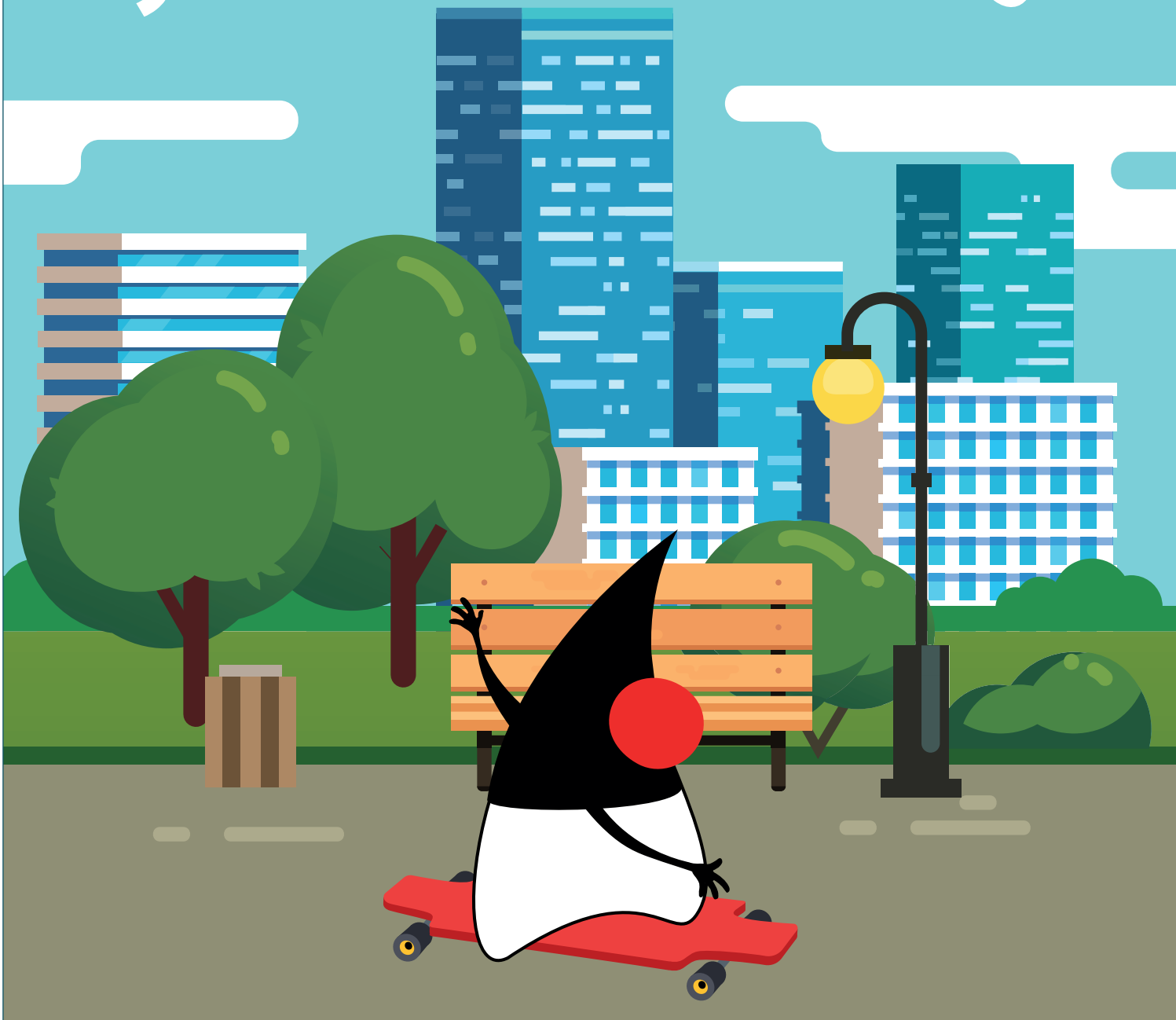
iJUG Verbund
www.ijug.eu

Open Source
Die Apache Software Foundation

Kubernetes Basics
Wie die App in die Cloud kommt

Bessere Usability
Design Thinking in der Entwicklung

Java macht Spaß



QAWARE
SOFTWARE ENGINEERING

Kommt vorbei:
JavaLand
Stand 618

IT-Probleme lösen.
Digitale Zukunft gestalten.

Mit Erfindergeist
und Handwerksstolz.

GREAT PLACE TO WORK®
2018
Beste Arbeitgeber™
Information & Telekommunikation

Von Mitarbeitenden empfohlen:
kununu.com/qaware
qaware.de/karriere

kubernetes

Kubernetes Basics – wie die App in die Cloud kommt

Christian Kühn, synyx GmbH

Kubernetes ist eines der bekanntesten und am stärksten wachsenden Frameworks zur Container-Orchestrierung. Einige der meistbeworbenen Features sind die einfache Automatisierbarkeit, Selbstheilung, Load Balancing, Skalierbarkeit und die Möglichkeit, Infrastruktur als Code zu beschreiben und zu versionieren. Viele Manager fordern daraufhin die Migration auf eine Kubernetes-Plattform, viele Entwickler stellen sich allerdings die Fragen: Brauchen wir das überhaupt? Wie kommt unsere App da jetzt rein? Wie stelle ich sicher, dass die nötigen Abhängigkeiten und Ressourcen für meine Applikation bereitstehen?

Kubernetes ist im Jahr 2015 aus einem Google-Projekt hervorgegangen und wird als Open-Source-Projekt unter der Cloud Native Computing Foundation (CNCF) entwickelt. Als Plattform ist Kubernetes modular aufgebaut und über ein Plug-in-System einfach erweiterbar. Sie erlaubt das Deployment containerbasierter Applikationen und kann deren kompletten Lifecycle abbilden und steuern.

Die Plattform bietet ein einfach konfigurierbares Monitoring, mit dem sich sämtliche Komponenten und Apps überwachen und bei gewissen Kriterien automatisch neu starten lassen. Zusätzlich können Cluster-Komponenten und die angebotenen Dienste automatisch skalieren. Anwendungen und Ressourcen lassen sich innerhalb eines Clusters in gekapselten Namespaces logisch voneinander trennen und gegen unbefugte Zugriffe absichern. Ein rollenbasiertes Autorisierungssystem steuert entsprechende Berechtigungen für die verschiedenen Namespaces und Komponenten. Applikationen werden mithilfe einer Container-Runtime betrieben (die nachstehend beschriebenen Beispiele laufen auf Basis von Docker).

Steuerung

Kubernetes lässt sich über einen API-Server steuern. Dieser bietet eine REST-Schnittstelle an, über die man den Cluster und alle zu betreibenden Dienste steuern und konfigurieren kann. Der API-Server bietet eine OpenAPI-kompatible Beschreibung mit einem Swagger-Endpoint *(siehe „https://swagger.io")* an. In der Regel wird über eine Definition im YAML- oder JSON-Format der gewünschte Zustand beschrieben.

Kubernetes folgt hier einem komplett deklarativen Ansatz. Der Cluster beziehungsweise die Steuer-Komponenten kümmern sich darum, dass dieser definierte Zustand erreicht und erhalten wird. Als Best Practice hat sich hier etabliert, entsprechende Definitionen zu versionieren und innerhalb von CI/CD-Pipelines zu verwenden, um sicherzustellen, dass man jederzeit mit wenig Aufwand jeden gewünschten Zustand in der Historie (wieder-)herstellen kann.

Beispiel-Ressourcen und Übungsumfeld

Die nachfolgend verwendeten Beispiel-Ressourcen kann man auch im GitHub-Repository unter *„https://github.com/cy4n/java-aktuell-kubernetes"* finden und nachstellen. Zum ersten Kennenlernen von Kubernetes empfiehlt es sich, ein lokales Kubernetes-Demosystem aufzusetzen. Die folgenden Beispiele wurden mithilfe von Minikube *(siehe „https://github.com/kubernetes/minikube")*, das als virtuelle Maschine auf dem eigenen Entwickler-Rechner läuft, aufgebaut. Zur Kommunikation mit dem API-Server wird kubectl, das offizielle Kubernetes-Command-Line-Tool, verwendet. Hier eine kurze Übersicht über die wichtigsten kubectl-Befehle:

- **kubectl get <typ>**
  Ressource(n) anzeigen
- **kubectl apply -f <Dateiname>**
  Definition aus „yaml"-/„json"-Datei anwenden
- **kubectl delete <typ> <name>**
  Ressource entfernen
- **kubectl logs <name>**
  Logs („stdout") eines Pod anzeigen

- **kubectl describe <typ> <name>**
  Details einer Ressource anzeigen
- **kubectl exec <name>**
  Befehle innerhalb eines Pod ausführen

Kubernetes-Basis-Ressource für die Applikation

Die kleinste deploybare Einheit in Kubernetes ist ein „pod". Sie enthält in der Regel einen Applikations-Container und repräsentiert einen Prozess beziehungsweise eine Instanz einer Anwendung. Die Definition eines Pod enthält unter anderem Informationen über das zu verwendende „container"-Image, den Port, den dieser Container anbietet und (Speicher-) Volumes, die der Applikation zur Verfügung gestellt werden sollen. Zusätzlich können für diesen Container „compute"-Ressourcen definiert werden, etwa wie viel CPU und RAM mindestens zur Verfügung stehen sollen, aber auch wieviel maximal genutzt werden darf. Eine Definition eines Pod könnte wie in *Abbildung 1* aussehen und mithilfe von kubectl angewendet werden *(siehe Abbildung 2)*.

```yaml
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: hello
5  spec:
6    containers:
7    - name: hello
8      image: cy4n/hello:0.0.4
9      ports:
10     - containerPort: 8080
```

Abbildung 1: Ressourcen-Definition „pod.yml"

```
~/demo> kubectl apply -f pod.yml
pod/hello created
~/demo> kubectl get pods -o wide
NAME      READY     STATUS    RESTARTS   AGE       IP           NODE
hello     1/1       Running   0          8s        172.17.0.6   minikube
```

Abbildung 2: Anwenden der Definition und Anzeige der laufenden Pods

```
~/demo> kubectl exec hello -- curl -s localhost:8080/host
hello
```

Abbildung 3: Ausführung eines Befehls auf der Shell des Pod

Kubectl dient hier nicht nur zum Schreiben beziehungsweise Senden neuer Definitionen an Kubernetes, sondern auch zum Anzeigen des aktuellen Zustands. In diesem Fall werden die laufenden Pods mit einigen Zusatz-Informationen angezeigt, so zum Beispiel auf welchem Cluster-Node der Container mit der Applikation gestartet und welche IP-Adresse zugewiesen wurde.

Die Anwendung in diesem Beispiel ist eine Spring-Boot-App, die ein REST-API mit einem Endpoint zur Anzeige des Server-Hosts anbietet. Da innerhalb von Kubernetes jeder Pod seinen eigenen Namen als Hostname erkennt, wird entsprechend hier immer der Pod-Name angezeigt. Da ein einfacher Pod noch nicht von außerhalb von Kubernetes angesprochen werden kann, kann man mithilfe von „kubectl exec" einen Test mit „curl", einem „commandLine"-HTTP-Client, durchführen *(siehe Abbildung 3)*.

Deployment-Applikation mit mehr Zustands-Informationen

Um die geplanten Anwendungen mit etwas mehr Zusammenhang zu versorgen, empfiehlt es sich, anstelle einzelner Pods ein „deployment" anzulegen *(siehe Abbildung 4)*. Es enthält unter anderem ein „pod"-Template (Zeile 12), das die Basis-Einstellungen eines Pod definiert. Dieses Template erweitert den Pod um ein „label", über das

```yaml
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: hello
5    labels:
6      app: hello
7  spec:
8    replicas: 2
9    selector:
10     matchLabels:
11       app: hello
12   template:
13     metadata:
14       labels:
15         app: hello
16     spec:
17       containers:
18       - name: hello
19         image: cy4n/hello:0.0.4
20         ports:
21         - containerPort: 8080
```

Abbildung 4: Ressourcen-Definition „deployment.yml"

```
~/demo> kubectl get deployment,replicaset,pod
NAME                           DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
deployment.extensions/hello    2         2         2            2           1m

NAME                                      DESIRED   CURRENT   READY   AGE
replicaset.extensions/hello-746844fd6c    2         2         2       1m

NAME                           READY     STATUS    RESTARTS   AGE
pod/hello-746844fd6c-452pr     1/1       Running   0          1m
pod/hello-746844fd6c-vdglk     1/1       Running   0          1m
```

Abbildung 5: Anzeigen des laufenden Deployments, der „replica"-Sets und der Pods

```
1   kind: Service
2   apiVersion: v1
3   metadata:
4     name: hello-service
5   spec:
6     selector:
7       app: hello
8     type: NodePort
9     ports:
10    - port: 8080
```

Abbildung 6: Ressourcen-Definition „service.yml"

```
1   apiVersion: extensions/v1beta1
2   kind: Ingress
3   metadata:
4     name: hello-ingress
5   spec:
6     rules:
7     - host: example.com
8       http:
9         paths:
10        - path: /
11          backend:
12            serviceName: hello-service
13            servicePort: 8080
```

Abbildung 7: Ressourcen-Definition „ingress.yml"

man später alle Instanzen dieser Anwendung selektieren kann. Zusätzlich sind weitere Zustands-Informationen enthalten, etwa die Anzahl der Instanzen, mit denen diese Anwendung betrieben werden soll (in Zeile 8 als „replicas" bezeichnet.

Kubernetes generiert bei einem „apply" mehrere Ressourcen, um diesen Zustand zu beschreiben *(siehe Abbildung 5)*. Zusätzlich zum erwarteten Deployment wird ein „replica"-Set generiert, das für den Lifecycle der Pods verantwortlich ist, die zu diesem Stand des Deployments gehören. Um die Zugehörigkeit darzustellen, wird der Name des „replica"-Sets aus dem Namen des Deployments (blau) und einer generierten Id (grün) aufgebaut. Die Pods wiederum beziehen ihren Namen aus dem Namen des „replica"-Sets mit einer weiteren Id (rot), die innerhalb der Pods eines „replica"-Sets eindeutig ist. Bei jeder Änderung des Pod-Templates wird ein neues „replica"-Set erstellt, das den neuen Stand abbildet und neue Pods startet, die der jeweils gültigen Template-Definition genügen.

Fester Applikations-Endpunkt

Da Pods beim Start dynamisch eine IP-Adresse zugewiesen bekommen, die auch nach deren Beenden wieder neu vergeben wird, ist es nicht sinnvoll möglich, eine Anwendung rein auf Pod-Ebene bereitzustellen. Zu diesem Zweck bietet Kubernetes die Möglichkeit, einen „service" zu definieren.

Ein Service stellt einen Endpunkt als Abstraktion der Anwendung dar, der mit einer festen IP-Adresse versehen ist und per DNS auch von anderen Anwendungen innerhalb von Kubernetes gefunden und angebunden werden kann. In einem Service wird mit einem „selector" (siehe *Abbildung 6*, Zeilen 6 und 7) zugewiesen, für welche Pods dieser Dienst angeboten wird.

Im gezeigten Beispiel in *Abbildung 6* werden alle Pods selektiert, denen das Label „app:hello" zugewiesen wurde (siehe *Abbildung 4*, Zeile 15). Kubernetes registriert im Service allerdings nur Pods, die im Zustand „READY" sind, und leitet Requests für diesen Service nur an diese Pods weiter. Diese aktuell eingebundenen und bereiten Pods kann man sich mit dem Befehl „kubectl get endpoints <service-name>" anzeigen lassen.

Mithilfe der abgebildeten „service"-Definition ist es bereits möglich, den Dienst auch schon von außerhalb des Kubernetes-Clusters zu erreichen. Der Typ „NodePort" richtet auf dem jeweiligen Worker im Cluster eine (TCP-)Port-Weiterleitung ein.

Load Balancer und Reverse-Proxy

In der Regel wird man keine Anwendung direkt bereitstellen wollen, sondern noch einen Load Balancer bzw. Proxy vorschalten. Im Falle der Web-Anwendung in der vorliegenden Demo wird ein HTTP-Proxy verwendet. Minikube bringt hier die Möglichkeit mit, einen Ingress-Controller auf Basis des bekannten Web-Servers NGINX *(siehe „https://github.com/kubernetes/ingress-nginx")* zu installieren. Dieser Server kann von allen Anwendungen in Minikube benutzt werden. Hierzu definiert man eine „ingress"-Ressource *(siehe Abbildung 7)*.

```
17        containers:
18        - name: hello
19          image: cy4n/hello:0.0.4
20          ports:
21          - containerPort: 8080
22          livenessProbe:
23            httpGet:
24              path: /actuator/health
25              port: 8080
26            initialDelaySeconds: 20
27          readinessProbe:
28            httpGet:
29              path: /actuator/health
30              port: 8080
31            initialDelaySeconds: 20
```

Abbildung 8: Ressourcen-Definition „deployment-health.yml" (Ausschnitt)

```
1   apiVersion: v1
2   kind: ConfigMap
3   metadata:
4     name: applicationyaml
5   data:
6     application.yml: |
7       management:
8         endpoints:
9           web:
10            exposure:
11              include: health,env
12      spring:
13        security:
14          user:
15            roles: ACTUATOR
```

Abbildung 9: Ressourcen-Definition „config.yml"

Die vorliegende Definition leitet HTTP-Anfragen, die die Request-URL *„example.com"* enthalten, an den Service „hello-service" auf Port 8080 weiter. Dieser Mechanismus dient dazu, Anfragen von außen weiterzuleiten. Im Ingress ist es zum Beispiel möglich, ein SSL-Zertifikat zu referenzieren, um den Dienst per HTTPS anzubieten. Wie in herkömmlichen „ReverseProxy"-Deployments ist es meist einfacher, ein Zertifikat und einen Key zur Verschlüsselung im Proxy zu definieren als in der Anwendung selbst, wo man zunächst (im Falle von Java) einen Keystore erstellen und diesen dann einbinden müsste.

Um das gezeigte Beispiel im Browser auszuprobieren, muss zunächst noch ein DNS-Eintrag angelegt werden, der die URL *„example.com"* auf die IP der Minikube-VM verweist. Über „curl" kann man die Anfrage auch ohne DNS-Mapping verschicken: „curl -H 'Host: example.com' http://192.168.99.100/host", wobei als IP-Adresse die IP der Minikube-VM einzutragen ist, hier der Standard-Wert. Bis hierhin hat man ein einfaches Deployment einer Web-Applikation erstellt, die verschiedenen Instanzen zu einer Abstraktion – einem Service – zusammengefasst und mit einem Ingress zur Verfügung gestellt.

Erweiterung 1: Readiness- und Liveness-Checks

Im Beispiel eines Pod hat *Abbildung 2* gezeigt, dass ein Pod sofort nach seinem Start in den Zustand „READY" geht. Der Pod ist laut Anzeige acht Sekunden alt, es sind bereits 1/1 Container bereit. In einer Spring-Boot-Anwendung ist das so nicht immer möglich. Hier sind Startzeiten von fünfzehn Sekunden und länger keine Seltenheit. An dieser Stelle kommen zwei neue Checks ins Spiel, ein Readiness- und ein Liveness-Check *(siehe Abbildung 8)*.

Der Liveness-Check prüft, ob ein neuer Pod beziehungsweise die enthaltene Anwendung initial gestartet wurde. Falls dies bis zu einer definierbaren Zeit nicht erfolgt, wird dieser Pod erneut gestartet. Ab Zeile 22 wurde dazu die bestehende „deployment.yml" um zwei Probes erweitert.

Sobald sich der Pod einmal im Zustand „RUNNING" befindet, wird infolge eines fehlerhaften Liveness-Checks der Pod als ungesund angesehen, abgeschaltet und komplett gelöscht. An seiner Stelle wird dann gleichzeitig ein neuer Pod gestartet.

Anhand des Readiness-Checks wird entschieden, ob ein Pod als Endpunkt für einen Service registriert wird. Im Unterschied zum Liveness-Check kann es sein, dass ein Pod mehrfach den Status

```
29              path: /actuator/health
30              port: 8080
31            initialDelaySeconds: 20
32          volumeMounts:
33          - name: applicationyaml-volume
34            mountPath: /app/config
35            readOnly: true
36        volumes:
37        - name: applicationyaml-volume
38          configMap:
39            name: applicationyaml
```

Abbildung 10: Ressourcen-Definition „deployment-volume.yml"

```
32          volumeMounts:
33          - name: applicationyaml-volume
34            mountPath: /app/config
35            readOnly: true
36          env:
37          - name: SPRING_SECURITY_USER_NAME
38            valueFrom:
39              secretKeyRef:
40                name: actuator-user
41                key: user
42          - name: SPRING_SECURITY_USER_PASSWORD
43            valueFrom:
44              secretKeyRef:
45                name: actuator-user
46                key: password
47        volumes:
```

Abbildung 11: Ressourcen-Definition „deployment-env.yaml"

```
1   apiVersion: v1
2   kind: Secret
3   metadata:
4     name: actuator-user
5   data:
6     user: amF2YQ==
7     password: YWt0dWVsbA==
```

Abbildung 12: Ressourcen-Definition „secret.yml"

wechselt. Die sogenannten „Probes" werten hier den HTTP-Status aus; als „gut" wird hier ein Status ab 200 und unter 400 angenommen, als „schlecht" beziehungsweise „ungesund" ein Status von 400 und höher.

In Spring Boot könnte man für den Readiness-Check etwa den „health"-Actuator *(siehe „https://spring.io/guides/gs/actuator-service")* benutzen. Hier wird der Service als „DOWN" (Status 503) angezeigt, wenn beispielsweise eine Abhängigkeit (wie eine Datenbank) nicht verfügbar ist. In diesem Fall könnte man sich entscheiden, diesem Pod keine Requests zuzuführen. Durch die Nutzung dieses Actuator für den Readiness-Check würde dann im Status „DOWN" der Pod als Endpunkt im Service entfernt, der Service würde also keine Requests an diesen Pod leiten.

Für den Liveness-Check eignet sich der „health"-Actuator nur bedingt. Im genannten Fall könnte ein Ausfall einer Abhängigkeit des Pod etwa in einem Drittsystem dazu führen, dass ein Pod komplett gelöscht und ein neuer als Ersatz gestartet wird, was möglicherweise gar nicht zum Erfolg führt. Der Entscheidung, wie man die Checks verwendet, liegen hier natürlich die individuelle System-Architektur der Anwendung und deren Abhängigkeiten zugrunde.

Erweiterung 2: Konfigurationsdateien

In realistischen Deployment-Szenarien lässt es sich nicht umgehen, die gleiche Anwendung mit verschiedenen Konfigurationen zu starten, etwa um zwischen Entwicklungs- und Produktions-Systemen zu unterscheiden und je nach Gebrauch unterschiedliche Parameter zu hinterlegen. Am Beispiel der Spring Boot Externalized Configuration *(siehe „https://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-external-config.html")* lässt sich anschaulich die Verwendung von Konfigurationsdateien in Kubernetes lernen. Im Beispiel wird in Kubernetes eine „configMap" angelegt *(siehe Abbildung 9)*. Der Inhalt dieser Konfiguration (Zeilen 8 bis 16) wird beim Start jedes Pod als Datei mit dem Namen „application.yml" (Zeile 7) auf ein virtuelles Dateisystem innerhalb des Pod geschrieben.

Dazu wird innerhalb des Pod-Templates im Deployment ein „volume" angelegt und im Container-Dateisystem auf Höhe der „.jar"-Datei im Unterordner „config" gemountet (siehe *Abbildung 10*, Zeile 32 als Erweiterung des bekannten Deployments). Dieser Mountpoint wurdest hier nach Spring-Boot-Spezifikation *(siehe „https://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-external-config.html")* ausgesucht. Die Beispiel-Konfiguration würde in Spring Boot den Actuator „/env" aktivieren. Zusätzlich wird mithilfe von Spring Security einem eingeloggten Benutzer die Rolle „ACTUATOR" zugewiesen (mehr dazu in Erweiterung 3).

Erweiterung 3: Umgebungsvariablen

Umgebungsvariablen bieten eine weitere Möglichkeit, die Anwendung zu erweitern. Im Beispiel sollen so ein Benutzername und ein Passwort für einen Benutzer abgelegt werden, auch bei der Spring Boot Externalized Configuration, die es ermöglicht, der genannten „application.yml" auch einzelne Properties als Environment-Variable zu definieren, in diesem Falle ist das der Spring-Security-Parameter „SPRING_SECURITY_USER_NAME und SPRING_SECURITY_USER_NAME".

Die Beispiel-Anwendung ist mithilfe von Spring Security so konfiguriert, dass der oben definierte Actuator-Endpunkt „/env" abgesichert wird und nur von einem eingeloggten Benutzer abgerufen werden darf, der die Rolle „ACTUATOR" besitzt. Diese Rolle ist ja bereits in der Konfigurationsdatei in Erweiterung 2 definiert. Um für einen Pod neue Umgebungsvariablen zu definieren, wird auch hier das Pod-Template im Deployment erweitert (siehe *Abbildung 11*, Zeilen 36 bis 46).

Die hier verwendeten Zugangsdaten müssen in Kubernetes als „secret" definiert sein. Hier wird die jeweilige Umgebungsvariable, etwa „SPRING_SECURITY_USER_NAME", die im „deployment" definiert wurde, mit dem entsprechenden Wert versehen *(siehe Abbildung 12)*.

Secrets werden in Kubernetes in „base64" codiert. Dies ist natürlich kein Sicherheits-Feature, sondern dient lediglich dazu, Probleme mit Sonder- und Steuerzeichen zu umgehen. Mit diesen Erweiterungen wurde die Anwendung mit einem Health-Check gestartet, sodass Kubernetes automatisch erkennt, ob ein Problem mit einer Instanz der Anwendung vorliegt und mit dem Löschen der defekten Instanz und dem Neustart weiterer Instanzen reagieren kann. Außerdem wurde die Anwendung erweitert, sodass laufzeitspezifische Konfigurationen übergeben werden können, die nicht mit im Container oder der Anwendung paketiert werden müssen.

Christian Kühn
kuehn@synyx.de

Christian Kühn ist System-Entwickler bei synyx. Zu seinen Aufgabenbereichen zählen Software-Entwicklung, Beratung und System-Administration in agilen, cross-funktionalen Teams. Seine Erfahrung beinhaltet Einflüsse aus mehr als zehn Jahren im Bereich „Operations" und mehreren Jahren in der Software-Entwicklung. Er ist Mitorganisator des DevOps-Meetups Karlsruhe und hält Vorträge zu Software-Entwicklung, Cloud und DevOps.

JavaLand

19. - 21. März 2019 in Brühl bei Köln

Ab sofort Ticket & Hotel buchen!

www.javaland.eu

Präsentiert von: DOAG Heise Medien

Community Partner: iJUG Verbund

www.javaland.eu

BESUCHE UNS AUF DER JAVALAND AM STAND 302.

arvato
BERTELSMANN

Du weißt es vielleicht nicht, aber wir stehen hinter zahlreichen Produkten und Leistungen, die Du nutzt – im Schnitt hat jeder Verbraucher in Deutschland achtmal täglich Kontakt mit uns. Arvato ist ein führender internationaler Dienstleister, der von und mit digitaler Technologie lebt. Mehr als 70.000 Mitarbeiter in über 40 Ländern unterstützen jeden Tag unsere Kunden dabei, erfolgreich am Markt zu agieren. Dazu konzipieren und realisieren wir maßgeschneiderte Lösungen für unterschielichste Geschäftsprozesse entlang integrierter Dienstleistungsketten. Daraus ergeben sich für Dich spannende Perspektiven. Es wird Zeit, dass wir uns persönlich kennenlernen!

Du findest uns ...

... zum Beispiel auf der JavaLand:
Vom 19. – 20. März 2019 am Stand 302
im Phantasialand in Brühl.
arva.to/career