Betrieb von Cloud Native Apps bei einem Cloud-Provider

Vorlesung:	Aktuelle Technologien zur Entwicklung verteilter Java-Anwendungen
Dozent:	Theis, Michael
Name:	Engelbrecht, Nils
Matrikelnummer:	98716415
Abgabe:	01.06.2018 (SoSe 2018)

Erklärung zur Ausarbeitung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig verfasst, noch nicht anderweitig für Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen oder Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

München, 01.06.2018 — Nils Engelbrecht

Ort, Datum — *Unterschrift*

Abbildungsverzeichnis

Inhaltsverzeichnis

1 Einleitung

Begriffe wie „Industrie 4.0“ und Digitalisierung sind zurzeit Schlagwörter der Medien die den digitalen Fortschritt im betriebswirtschaftlichen Rahmen prägen. Es gibt keine einheitliche Definition der Begriffe die absolute Gültigkeit besitzt. Dennoch ist klar, dass in der Regel Unternehmen die nicht an der Welle der „Digitalen Transformation“ teilnehmen, bald nicht mehr mit den Wettbewerbern Schritt halten können und vom Markt verschwinden könnten. Anhand der FinTech Firma N26 ist zu sehen, dass die neuen Player/ Startups den Vorteil der grünen Wiese nutzen um mit bestehenden Banken konkurrieren zu können. Es sind unter anderem keine Hardwareanschaffungen mehr nötig. Cloudanbieter wie Amazon Web Services (AWS) vermieten virtualisierte Hardwareressourcen je nach Bedürfnis und rechnen diese je nach Nutzungsdauer ab. Daraus entsteht der Vorteil ohne großes Risiko Applikationen in die Cloud auszulagern.

Im Rahmen dieser Arbeit wird der Betrieb von Cloud Native Applikationen in der Cloud beziehungsweise in einem Kubernetes Cluster thematisiert. In Kapitel 2.1 wird ein Überblick der verschiedenen Cloud-Anbieter geschaffen. Zudem werden Grundlagen der Virtualisierung und Containerisierung aufgegriffen und erläutert welche Rolle diese Konzepte in der Cloud einnehmen. Anschließend wird aufgezeigt was Kubernetes ist und wie das System funktioniert. Danach wird die exemplarische Anwendung von Kubernetes in Kombination mit einer Spring Boot Java Applikation aufgezeigt. Abschließend werden die wichtigsten Informationen in dem Fazit zusammengefasst dargestellt. Zusätzlich wird ein Ausblick darüber gegeben welches weitere Potential der Betrieb von Cloud Native Applikationen in der Cloud zukünftig haben wird.

Ziel dieser Arbeit ist ein grundlegendes Verständnis über den Betrieb von Cloud Native Applikationen in der Cloud zu vermitteln. Insbesondere soll geklärt werden was Kubernetes ist und wie es eingesetzt werden kann, um den Betrieb von Java Applikationen in der Cloud zu ermöglichen.

2 Betrieb von Cloud Native Apps bei einem Cloud-Provider

Zu den thematisierten Grundlagen zählen das Konzept der Containerisierung und Kubernetes. Der exemplarische Betrieb einer Spring Boot Applikation in einem Kubernetes Cluster vermittelt praxisnahen Einblick. Abschließend werden Alternativen zur Nutzung von Kubernetes aufgezeigt.

2.1 Grundlagen

Zunächst ist zu klären welche Cloud-Provider es gibt und wie sich diese voneinander unterscheiden. Zu den bekanntesten Cloud Anbietern zählen Unternehmen wie Amazon (Amazon Web Services), Google (Google Cloud Platform), Microsoft (Azure) und IBM (IBM Cloud). Diese Anbieter sind allerdings so populär, da die Firmen schon vorher bekannt waren. Neben diesen Anbietern gibt es noch Plattformen wie beispielsweise CloudFoundry und DigitalOcean. (“Clutch,” 2018)

Die Fülle an Cloud-Providern ist groß. Die Unterschiede bestehen bei den Kosten und der angebotenen Leistung. Manche Provider bieten nur Hardware-ressourcen an, andere wiederum ganze Plattformen. Die verschiedenen Leistungsarten sind bekannt als (Theis, 2018):

- Infrastructure as a Service (IaaS)
- Container as a Service (CaaS)
- Platform as a Service (PaaS)
- Function as a Service (FaaS)
- Software as a Serice (SaaS)

Die EC2 Instanzen die Amazon Web Services anbieten sind als IaaS zu werten. Im Gegensatz dazu ist Kubernetes als Container as a Service zu sehen. CaaS basiert auf dem Konzept der Containerisierung. Im Zuge dessen, werden zuerst die Grundlagen der Containerisierung erläutert und anschließend wird auf Kubernetes eingegangen.

2.1.1 Containerisierung

Die Funktionsweise von Kubernetes ist ohne dem Konzept der Containerisierung undenkbar. Deshalb ist es wichtig die Grundlagen der Containerisierung zu verstehen.

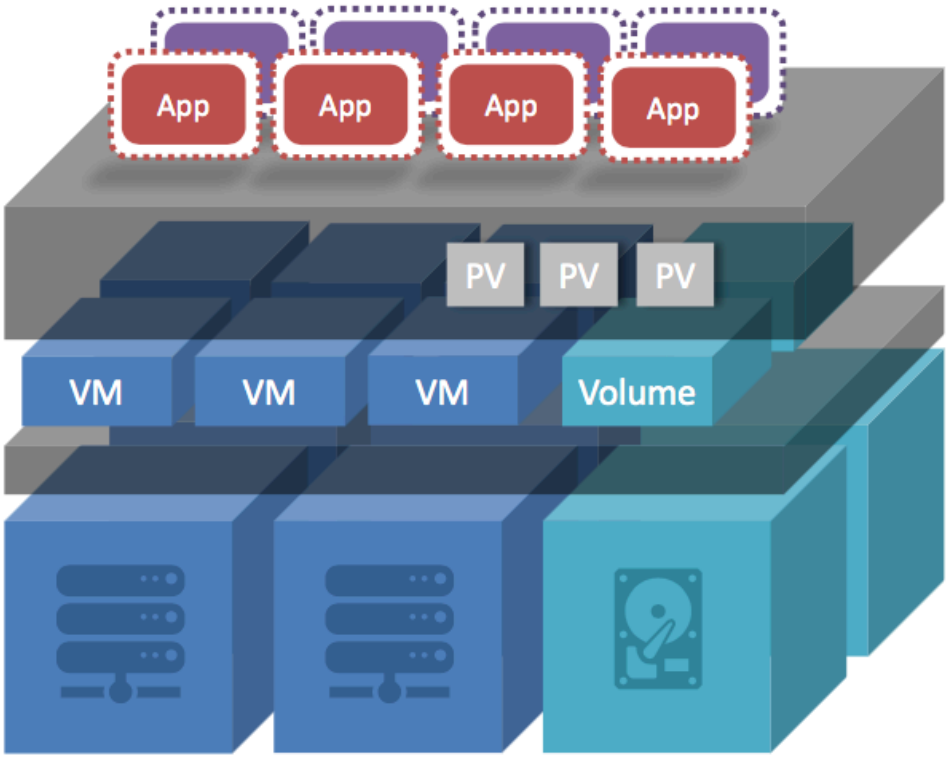


Abbildung 1: Struktur von CaaS in der Cloud (Theis, 2018)

Abbildung 1 zeigt den Aufbau der Container as a Service Architektur auf. Die physischen Ressourcen wie Speicher, Netzwerk oder Rechenleistung finden sich in Volumes und Virtual Machines virtualisiert wider. Der Betrieb einer Anwendung auf Ebene der VMs ist durch das verwendete Operating System (OS) beschränkt. Falls beispielsweise kein Java auf diesem OS installiert ist, ist der Betrieb von Spring Boot Applikationen nativ nicht möglich. Die Containerisierung hingegen ermöglicht praktisch alle Applikations- und Diensttypen: die benötigte Laufzeit-umgebung und die Applikation bilden einen Container. Eine Applikation wird skaliert, indem mehrere Container (die dieselbe Anwendung beinhalten) betrieben werden. (Theis, 2018)

2.1.2 Kubernetes

Kubernetes ist ein open-source System um containerisierte Applikationen auf mehreren Rechnern verteilt zu betreiben. Google hat das System, basierend auf 15 Jahren Erfahrung, entwickelt und eingesetzt. (Theis, 2018) Kubernetes beherrscht grundlegende Mechanismen für den Betrieb und die Skalierung der Container. Das System orchestriert Rechner-, Netzwerk- und Speicher-infrastrukturen um die Arbeitslast von Anwendungen zu verteilen. Diese Architektur ist so einfach zu handhaben wie eine Platform as a Service und bietet gleichzeitig die Flexibilität einer Infrastructure as a Service. Bedingt durch diese Vorteile ist Kubernetes kompatibel mit der Infrastruktur der meisten Cloud Provider. Das Wort Kubernetes hat seinen Ursprung aus dem Griechischen und bedeutet Steuermann beziehungsweise Pilot: Kubernetes als Plattform steuert den Betrieb vieler Container. Abgekürzt ist Kubernetes auch als „K8s“ bekannt. Hierbei wurden die acht Buchstaben „ubernete“ durch eine 8 ersetzt. (“What is Kubernetes,” 2018)

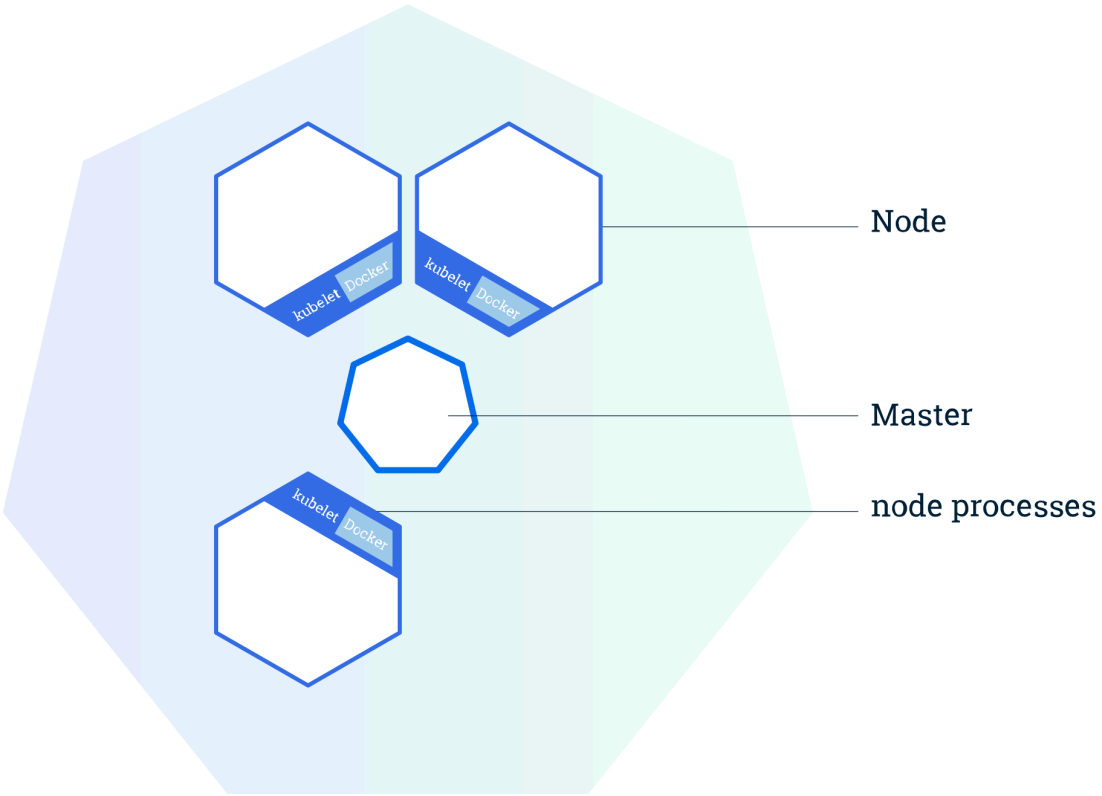


Abbildung 2: Kubernetes Cluster (“Using Minikube to Create a Cluster - Kubernetes,” 2018)

Abbildung 2 bildet die Grundarchitektur eines Kubernetes Clusters ab. Es gibt einen Master-Knoten der den Cluster koordiniert. Zu seinen Koordinationsaufgaben zählen:

- Scheduling,
- Skalierung und das
- Ausspielen von Updates.

Die einzelnen Nodes übernehmen den tatsächlichen Betrieb von Applikationen in Form von Docker Containern und werden als Worker bezeichnet. Sowohl Master als auch Node können BareMetal oder VMs sein. Da solche Knoten für den Betrieb von Docker Container zuständig sind, besitzen sie Tools zur Verwaltung von Container. Dies ist notwendig, da die *kubelets* dafür zuständig sind, das Replikationslevel aufrecht zu erhalten: Sie überwachen die laufenden Container und starten sie im Fehlerfall neu. (“Using Minikube to Create a Cluster - Kubernetes,” 2018)

Für Benutzer ist der Master der einzige Ansprechpartner. Der Master stellt eine REST API zur Verfügung, welche die einzige Kommunikationsmöglichkeit verkörpert. Sowohl kubectl (CLI) als auch die WebUI nutzen diese RESR API. Selbst die einzelnen Nodes des Clusters kommunizieren mit dem Master nur über die soeben genannte API. (“Using Minikube to Create a Cluster - Kubernetes,” 2018)

2.2 Exemplarischer Betrieb einer Spring Boot Anwendung in Kubernetes

Dieses Kapitel beschäftigt sich mit dem exemplarischen Betrieb einer Spring Boot Applikation in einem Kubernetes Cluster. Vorerst wird aufgezeigt wie die Applikation und ein entsprechendes Docker Image erzeugt wird. Anschließend ist das Deployment dieser Anwendung in einem lokalen Kubernetes Cluster praxisnah erklärt. Erst nachfolgend ist das Deployment in der Cloud erläutert.

2.2.1 Spring Boot Applikation

Es handelt sich bei der exemplarisch zu betreibenden Applikation um eine typische CRUD Java Anwendung die nach außen eine RESTful API anbietet.

Eine simple Spring Boot Applikation kann über den „Spring Initializr“ (siehe https://start.spring.io) erzeugt werden:

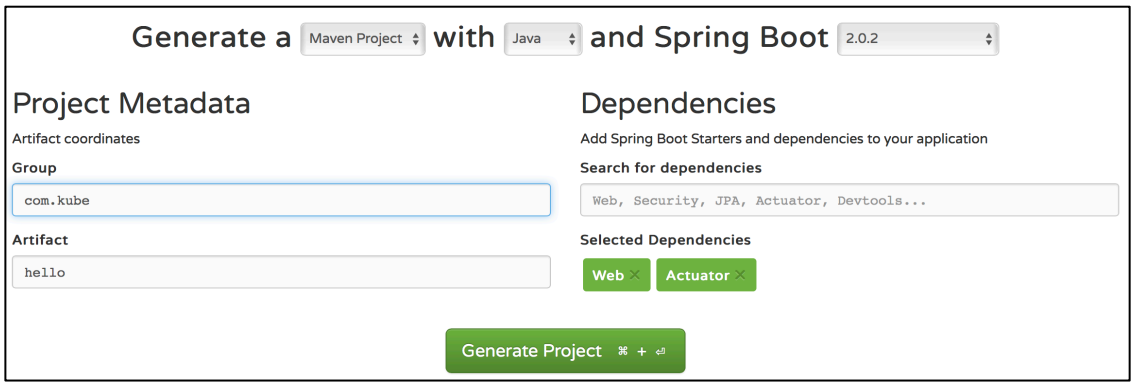
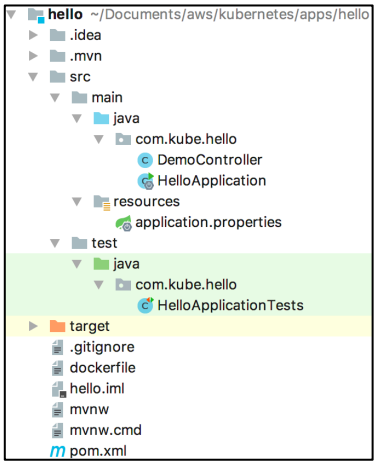


Abbildung 3: Spring Initializr (“Spring Initializr,” 2018)

Wie in Abbildung 3 zu sehen, wird die exemplarische Spring Boot Applikation mit Maven generiert. Im vorhinein bekannte Dependencies können an dieser Stelle schon eingebunden werden. Da nur eine simple Java Anwendung gebaut werden soll, werden die Web Dependency und der Actuator benötigt, sodass nach Inbetriebnahme der Applikation in Kubernetes die Erreichbarkeit testbar ist. Der /health Endpoint dient Kubernetes als Anhaltspunkt, um zu bestimmen wann die Applikation bereit ist Anfragen entgegen zu nehmen (Liveness und Readiness Probes). Die Actuator Endpoints bieten die Möglichkeit gewisse Metriken von Spring Boot Apps zu überwachen. (Schane, 2017)



Der nebenstehenden Abbildung ist die generierte Projektstruktur zu entnehmen. Um einen eigenen Rest-Endpoint anzubieten ist die Klasse *DemoController* dem Projekt hinzugefügt. Die implementierte Methode wird im weiteren Verlauf bei einem Aufruf von http://localhost:8080/hello aufgerufen und gibt das Objekt Message zurück.

Abbildung 4: Spring Boot Projektstruktur

Die *DemoController* Java Klasse ist wie folgt aufgebaut:

```
package com.kube.hello;

import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
public class DemoController {

    @RequestMapping("/hello")
    public ResponseEntity<Message> simpleHello() {
        return new ResponseEntity<>(new Message("Hello K8s"), HttpStatus.OK);
    }

    private class Message {
        private String message;

        public Message() {
        }

        public Message(String message) {
            this.message = message;
        }

        public String getMessage() {
            return message;
        }

        public void setMessage(String message) {
            this.message = message;
        }
    }
}
```

Da die Actuator Endpoints standardmäßig nicht freigeschalten sind, sind entweder alle oder nur die benötigten Endpoints in der *application.properties* Datei zu aktivieren:

```
management.endpoints.web.exposure.include=health,shutdown
management.endpoint.health.enabled=true
management.endpoint.shutdown.enabled=true
```

Wie dem Properties File zu entnehmen ist, wird die Applikation nur den /health und den /shutdown Endpoint zur Verfügung stellen. Im weiteren Verlauf (siehe Abbildung 10) wird deutlich, weshalb der /shutdown Endpoint eingebunden ist.

Die JAR ist mit nachfolgendem Maven Befehl zu erstellen:

```
$ mvn clean package
```

Anschließend kann die Applikation lokal hochgefahren werden um sicherzustellen, dass die Anwendung ohne Probleme startet:

```
➜ hello java -jar target/hello-0.0.1-SNAPSHOT.jar


  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::        (v2.0.2.RELEASE)

2018-05-31 15:58:28.506  INFO 6196 --- [           main] com.ku
be.hello.HelloApplication          : Starting HelloApplication
```

Abbildung 5: Spring Boot Applikationsstart

In einem weiteren Terminal kann via CURL nun der Health-Endpoint aufgerufen werden um den Status der Applikation zu ermitteln:

```
$ curl http://localhost:8080/actuator/health
{ “status“ : “UP“ }
```

Alternativ kann dies auch über einen Browser abgefragt werden:

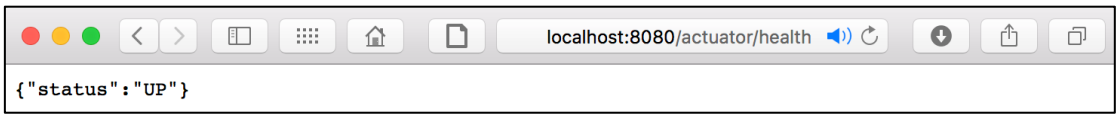


Abbildung 6: Health-Endpoint Abfrage via Browser

2.2.2 Docker Image

Da nun die grundsätzliche Funktionalität der Applikation bestätigt ist, kann ein Docker Image erstellt werden. Dieses Image wird dann im weiteren Verlauf benötigt um die Anwendung in Kubernetes betreiben zu können. Einerseits kann das Docker Image manuell erstellt werden, andererseits gibt es Maven Plugins welche beim Maven Build automatisch ein Docker Image erstellen. Unabhängig davon ist ein Docker File notwendig:

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

Der Name der Java JAR Datei ist als ${JAR_FILE} Variable angegeben. Diese Variable wird bei dem Build als Parameter übergeben. Das *openjdk* Docker Image wird als Basis benutzt. Die Version *8-jdk-alpine* beinhaltet Java 8, sodass die Spring Boot Applikation betrieben werden kann. Für diese Arbeit wird ein Maven Plugin von spotify benutzt, welche wie folgt in der *pom.xml* eingebunden wird (“Spring Boot with Docker,” 2018):

```
<properties>
   <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
   <docker.image.prefix>hellokube</docker.image.prefix>
</properties>
...
<plugins>
    ...
    <plugin>
        <groupId>com.spotify</groupId>
        <artifactId>dockerfile-maven-plugin</artifactId>
        <version>1.3.6</version>
        <configuration>
           <repository>${docker.image.prefix}/${project.artifactId}</repository>
           <buildArgs>
              <JAR_FILE>target/${project.build.finalName}.jar</JAR_FILE>
           </buildArgs>
        </configuration>
        <executions>
            <execution>
                <id>default</id>
                <phase>package</phase>
                <goals>
                    <goal>build</goal>
                </goals>
            </execution>
        </executions>
    </plugin>
</plugins>
```

Das Docker File hat auf oberster Projektebene zu liegen (siehe Abbildung 4). Diesmal dauert der Maven Build etwas länger, da vorerst das verwendete Basis Docker Image (openjdk) heruntergeladen wird. Nachdem der Maven Build abgeschlossen ist, taucht sowohl das neugebaute als auch das Basis Image in dem lokalen Docker Repository auf:

```
 $ docker images
REPOSITORY       TAG           IMAGE ID       CREATED              SIZE
hellokube/hello  latest        9490daf949fa   About a minute ago   119MB
openjdk          8-jdk-alpine  224765a6bdbe   4 months ago         102MB
```

2.2.3 One Node Kubernetes Cluster (lokal)

Bevor das neu erstellte Docker Image mit Kubernetes in der Cloud betrieben wird, sollte zuerst lokal getestet werden ob dies reibungslos abläuft. Dazu ist ein lokaler Kubernetes Cluster notwendig. Es gibt mehrere Möglichkeiten diesen auf dem eigenen Rechner aufzusetzen. Erwähnt sei, dass neuere Docker Edge Versionen das Aufsetzen eines Kubernetes Clusters unterstützten. Die Errichtung des Clusters wird an dieser Stelle jedoch nicht weiter thematisiert. Unabhängig davon ob der Kubernetes Cluster lokal oder in der Cloud betrieben wird, wird der Kubernetes Client benötigt. Dieser kann wie folgt installiert werden:

```
$ brew install kubectl
```

Kubectl ermöglicht die Kommunikation mit dem Kubernetes Master Knoten. Sobald ein Kubernetes Cluster erstellt ist und kubectl installiert ist, kann ein Deployment erstellt werden. Die Datei *deployment.yaml* ist zu erzeugen und enthält die Konfiguration um das *hellokube/hello* Docker Image in Kubernetes zu betreiben.

Der Inhalt der Konfigurationsdatei sieht wie folgt aus:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello-service-deployment
spec:
  selector:
    matchLabels:
      app: hello-service
  replicas: 2
  template:
    metadata:
      labels:
        app: hello-service
    spec:
      containers:
      - name: hello
        image: hellokube/hello:latest
        imagePullPolicy: Never
        ports:
        - containerPort: 8080
```

Wichtig hierbei ist *spec.template.spec.containers.imagePullPolicy* auf Never zu setzten, da sonst in der öffentlichen Docker-Hub Registry nach diesem Image gesucht wird. Das erzeugte Image liegt allerdings in dem lokalen Docker Repository. Diese *deployment.yaml* Datei wird verwendet um das Kubernetes API Objekt vom Typ Deployment anzulegen (“Run a Stateless Application Using a Deployment - Kubernetes,” 2018):

```
$ kubectl apply –f deployment.yaml
deployment “hello-service-deployment“ created
```

Da in der Praxis mehrere Benutzer auf den Kubernetes Cluster zugreifen und Anwendungen betreiben, ist zu erwähnen, dass nachfolgender Befehl die aktuelle Beschreibung des *hello-service-deployments* ermöglicht.

```
➜ hello kubectl describe deployment hello-service-deployment
Name:                   hello-service-deployment
Namespace:              default
CreationTimestamp:      Sat, 26 May 2018 16:46:55 +0200
Labels:                 app=hello-service
Annotations:            deployment.kubernetes.io/revision=12
                        kubectl.kubernetes.io/last-applied-configuration={"apiVersion
":"apps/v1","kind":"Deployment","metadata":{"annotations":{},"name":"hello-service-de
ployment","namespace":"default"},"spec":{"replicas":2,...
Selector:               app=hello-service
Replicas:               2 desired | 2 updated | 2 total | 2 available | 0 unavailable
StrategyType:           RollingUpdate
MinReadySeconds:        0
RollingUpdateStrategy:  25% max unavailable, 25% max surge
Pod Template:
  Labels:  app=hello-service
  Containers:
   hello:
    Image:        hellokube/hello:latest
    Port:         8080/TCP
    Environment:  <none>
    Mounts:       <none>
  Volumes:        <none>
```

Abbildung 7: Beschreibung eines Deployments

Die Applikation ist bisher noch nicht von außen erreichbar. Um die Applikation von außen erreichen zu können, ist ein Service in Kubernetes anzulegen. Die dazu benötigte Datei *service.yaml* sieht wie folgt aus (“Services - Kubernetes,” 2018):

```yaml
apiVersion: v1
kind: Service
metadata:
  name: hello-service
spec:
  ports:
    - port: 9090
      targetPort: 8080
      nodePort: 32100
  selector:
    app: hello-service
  type: NodePort
```

Der Service-Typ (*spec.type*) bestimmt inwiefern der Service erreichbar ist. Beispielsweise kann der Service-Typ „ClusterIP“ gewählt werden, wenn der Service nur innerhalb des Clusters erreichbar sein soll. Sinnvoll ist dies, wenn mehrere Micro-Services in einem Cluster betrieben werden, welche miteinander kommunizieren und nur ein paar wenige Services von außen erreichbar sein sollen. Weitere Service-Typen sind „LoadBalance“ und „NodePort“. Auf den Typ „LoadBalance“ wird in dem nachfolgenden Kapitel noch eingegangen, da die genutzte Umgebung keinen Loadbalancer bereitstellt. Für die lokale Variante von Kubernetes benutzen wird den Service-Typ „NodePort“, sodass der Service von außerhalb des Clusters erreichbar ist. Anzumerken ist, dass das Feld *spec.port.nodePort* Werte zwischen 30000 und 32767 annehmen kann. Um den Service in Kubernetes anzulegen wird nachfolgender Befehl ausgeführt (“Services - Kubernetes,” 2018):

```
$ kubectl create –f service.yaml
service „hello-service“ created
```

Der Service ist somit erstellt und nun über http://localhost:32100/ erreichbar. Weitere Detailinformationen zu dem Service können wie folgt abgerufen werden:

```
$ kubectl describe service hello-service
Name:                     hello-service
Namespace:                default
Labels:                   <none>
Annotations:              <none>
Selector:                 app=hello-service
Type:                     NodePort
IP:                       10.105.185.134
LoadBalancer Ingress:     localhost
Port:                     <unset>  9090/TCP
TargetPort:               8080/TCP
NodePort:                 <unset>  32100/TCP
Endpoints:                10.1.0.25:8080,10.1.0.26:8080
Session Affinity:         None
External Traffic Policy:  Cluster
Events:                   <none>
```

Falls der Source-Code der Applikation geändert beziehungsweise erweitert wurde, kann durch einen neuen Maven Build automatisiert ein neues Docker Image erzeugt werden. Dieses Image trägt dann den Tag „latest“ und nimmt dadurch dem Vorgänger den Tag. Validiert werden kann dies über die Spalte „Created“ mittels Docker Befehl „docker images“. Durch nachfolgenden Kubernetes Befehl kann das benutze Docker Image eines Deployments angepasst werden:

```
$ kubectl set image deployment/hello-service-deployment hello=hellokube/hello
Deployment „hello-service-deployment“ image updated
```

Der Rollout Status des Deployments gibt Auskunft über den Fortschritt des Image-Updates. Da das Deployment nur zwei Pods (ein Pod = n Container) umfasst ist das Update zügig abgeschlossen. Das Update veranlasst, dass die alten Pods gelöscht werden und neue Pods (in gleicher Anzahl wie gelöschte Pods) mit aktualisiertem Docker Image hochgefahren werden. (“Run a Stateless Application Using a Deployment - Kubernetes,” 2018)

```
$ kubectl rollout status deployment/hello-service-deployment
Waiting for rollout to finish: 1 old replicas are pending termination...
Waiting for rollout to finish: 1 old replicas are pending termination...
Deployment “hello-service-deployment“ successfully rolled out
$ kubectl get pods
NAME                                        READY     STATUS        RESTARTS    AGE
hello-service-deployment-5b4845c577-4jnq2   1/1       Running       0           16s
hello-service-deployment-5b4845c577-c2k6t   1/1       Running       0           14s
```

Sehr nützlich hierbei ist zu verstehen nach welcher Logik die Pods gelöscht und neu erzeugt werden. Die Konfiguration und Events des Deployments geben Aufschluss darüber:

```
➜ hello kubectl describe deployment hello-service-deployment
Name:                   hello-service-deployment
Namespace:              default
CreationTimestamp:      Sat, 26 May 2018 16:46:55 +0200
Labels:                 app=hello-service
Annotations:            deployment.kubernetes.io/revision=13
                        kubectl.kubernetes.io/last-applied-configuration={"apiVersion":"apps/v1","kind":"Deployment","metada
ta":{"annotations":{},"name":"hello-service-deployment","namespace":"default"},"spec":{"replicas":2,...
Selector:               app=hello-service
Replicas:               2 desired | 2 updated | 2 total | 2 available | 0 unavailable
StrategyType:           RollingUpdate
MinReadySeconds:        0
RollingUpdateStrategy:  25% max unavailable, 25% max surge
Pod Template:
  Labels:  app=hello-service
  Containers:
   hello:
    Image:        hellokube/hello
    Port:         8080/TCP
    Environment:  <none>
    Mounts:       <none>
  Volumes:        <none>
Conditions:
  Type           Status  Reason
  ----           ------  ------
  Available      True    MinimumReplicasAvailable
  Progressing    True    NewReplicaSetAvailable
OldReplicaSets:  <none>
NewReplicaSet:   hello-service-deployment-5b4845c577 (2/2 replicas created)
Events:
  Type    Reason             Age   From                   Message
  ----    ------             ----  ----                   -------
  Normal  ScalingReplicaSet  7m    deployment-controller  Scaled down replica set hello-service-deployment-646c8c47f6 to 2
  Normal  ScalingReplicaSet  35s   deployment-controller  Scaled up replica set hello-service-deployment-5b4845c577 to 1
  Normal  ScalingReplicaSet  33s   deployment-controller  Scaled down replica set hello-service-deployment-646c8c47f6 to 1
  Normal  ScalingReplicaSet  33s   deployment-controller  Scaled up replica set hello-service-deployment-5b4845c577 to 2
  Normal  ScalingReplicaSet  31s   deployment-controller  Scaled down replica set hello-service-deployment-646c8c47f6 to 0
```

Abbildung 8: Deployment UpdateStrategie und Events

Wie in obiger Abbildung ersichtlich, besitzt das Deployment *„hello-service-deployment“* die RollingUpdate Strategie. Ferner ist unter dem Punkt „RollingUpdateStrategy“ beschrieben, dass maximal 25% der Pods nicht erreichbar sein dürfen. Das Image Update des Kubernetes Deployments ist verantwortlich für die in Abbildung 8 zu sehenden Events: Es ist vorerst ein neues Replica Set (hello-service-deployment-5b4845c577) angelegt und hochskaliert

worden. Anschließend wurde das alte Replica Set (hello-service-deployment-646c8c47f6) herunterskaliert. Dieses Vorgehen ermöglicht Deployments mit keinerlei Downtime.

Neben dem Konzept des Rolling Updates bietet Kubernetes die Möglichkeit Applikationen durch Skalierung hochverfügbar zu machen. Die Anzahl an Pods kann sowohl in der *deployment.yaml* Konfiguration manuell über den Wert *spec.replicas* angegeben werden, als auch im nachhinein angepasst werden.

```
➜ hello kubectl get deployments
NAME                       DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
hello-service-deployment   2         2         2            2           5d
➜ hello kubectl scale deployment hello-service-deployment --replicas=5
deployment "hello-service-deployment" scaled
➜ hello kubectl get pods
NAME                                        READY     STATUS    RESTARTS   AGE
hello-service-deployment-5b4845c577-6bqzd   1/1       Running   0          9s
hello-service-deployment-5b4845c577-dt8fc   1/1       Running   0          5m
hello-service-deployment-5b4845c577-gvv2z   1/1       Running   0          9s
hello-service-deployment-5b4845c577-p6wkn   1/1       Running   0          5m
hello-service-deployment-5b4845c577-wdxw8   1/1       Running   0          9s
```

Abbildung 9: Skalierung von Kubernetes Deployment

Abbildung 9 zeigt, dass vor der Hochskalierung des Deployments von *„hello-service-deployment“* zwei Pods gewünscht und erreichbar waren. Der neu gesetzte Wert replicas (auf 5) veranlasst den Start von drei weiteren Pods. Dies ist am Alter der Pods erkennbar. Die Skalierung muss nicht unbedingt manuell angegeben werden. Vor allem um eine Anwendung hochverfügbar anbieten zu können, empfiehlt sich der Einsatz eines „Horizontal Pod Autoscaler“ (“Horizontal Pod Autoscaler Walkthrough - Kubernetes,” 2018):

```
$ kubectl autoscale deployment hello-service-deployment --cpu-percent=50 --min=2 --max=10
deployment "hello-service-deployment" autoscaled
```

Die Auto-Skalierung übernimmt die Aufgabe für genügend Pods zu sorgen. Im vorangegangenem Beispiel sind mindestens zwei aber maximal zehn Pods erwünscht. Steigt die CPU-Auslastung der Pods auf über 50 Prozent, so fährt Kubernetes automatisch weitere Pods hoch, bis die erwünschte Auslastung oder die obere replicas Grenze erreicht ist. Der Betrieb mehrerer Deployments kann

dadurch effizient gestaltet werden, da Ressourcen individuell je nach Bedarf vergeben werden. Wenn beispielsweise die RESTful API von dem *hello-service-deployment* plötzlich extrem beansprucht wird, dann steigt die CPU Auslastung der Pods. Sobald die vorgegebene Grenze überschritten ist, werden weitere Pods hochgefahren. Die Nutzung des *hello-service-deployments* geht zurück: nun sinkt die Anzahl der Pods automatisch und die Ressourcen sind für andere wieder frei verfügbar.

Anzumerken ist, dass Kubernetes Deployments standardmäßig „Liveness und Readiness Probes“ der Pods durchführt. Diese Aufgabe übernimmt das kubelet (Abbildung 2). Die Liveness Probe gibt Auskunft darüber, ob der Container neugestartet werden muss oder nicht. Die Readiness Probe hingegen bestimmt, wann ein Container bereit ist, Anfragen entgegenzunehmen. Zurzeit sind 5 Pods bereit (“Configure Liveness and Readiness Probes - Kubernetes,” 2018):

```
➜  ~ kubectl get pods
NAME                                     READY     STATUS    RESTARTS   AGE
hello-service-deployment-5b4845c577-6bqzd   1/1       Running   0          11h
hello-service-deployment-5b4845c577-dt8fc   1/1       Running   0          11h
hello-service-deployment-5b4845c577-gvv2z   1/1       Running   0          11h
hello-service-deployment-5b4845c577-p6wkn   1/1       Running   0          11h
hello-service-deployment-5b4845c577-wdxw8   1/1       Running   0          11h
➜  ~ curl -X POST http://localhost:32100/actuator/shutdown
{"message":"Shutting down, bye..."}%
➜  ~ kubectl get pods
NAME                                     READY     STATUS    RESTARTS   AGE
hello-service-deployment-5b4845c577-6bqzd   1/1       Running   0          11h
hello-service-deployment-5b4845c577-dt8fc   1/1       Running   1          11h
hello-service-deployment-5b4845c577-gvv2z   1/1       Running   0          11h
hello-service-deployment-5b4845c577-p6wkn   1/1       Running   0          11h
hello-service-deployment-5b4845c577-wdxw8   1/1       Running   0          11h
```

Abbildung 10: Automatischer Neustart eines Pods

Der in Abbildung 10 zu sehende CURL Befehl, fährt eine Instanz herunter. Kubernetes bemerkt das und startet diesen Pod automatisch neu.

2.2.4 Kubernetes WebUI

Bisher sind alle Änderungen an Deployments und Services über die Kommandozeile via *kubectl* Befehl an den Kubernetes Cluster kommuniziert worden. Der Vorteil der Nutzung des Kubernetes Clients besteht darin, dass Benutzer beziehungsweise Administratoren des Kubernetes Clusters Operationen skripten können. Diese kommandozeilenbasierte Nutzung ermöglicht die Errichtung von Continuous Deployment Pipelines.

Nicht alle Benutzer eines Kubernetes Clusters sind terminalaffin. Eine graphische Oberfläche bietet die komfortable Möglichkeit den Cluster zu überwachen und zu steuern. Das WebUI ist standardmäßig nicht mit ausgeliefert. Die Oberfläche ist ein eigenständiges Deployment und kann wie folgt installiert werden:

```
$ kubectl create –f kubernetes–dashboard.yaml
secret “kubernetes–dashboard–certs“ created
serviceaccount “kubernetes–dashboard“ created
role “kubernetes–dashboard–minimal“ created
rolebinding “kubernetes–dashboard–minimal“ created
service “kubernetes–dashboard“ created
```

Die kubernetes-dashboard.yaml Datei kann durch diesen Link ersetzt werden: https://raw.githubusercontent.com/kubernetes/dashboard/master/src/deploy/recommended/kubernetes-dashboard.yaml Optional kann diese auch heruntergeladen werden. Das Dashboard ist erst unter http://localhost:8001/ui/ erreichbar, sobald nachfolgender Befehl abgesetzt worden ist:

```
$ kubectl proxy
Starting to serve on 127.0.0.1:8001
```

Das webbasierte User Interface kann zusätzlich um weitere Komponenten erweitert werden. Hierzu zählen beispielsweise, wie in Abbildung 11 zu sehen, die CPU und Memory Auslastungsgrafik. Gerade für Administratoren des Clusters ist eine solche visuelle Aufbereitung nützlich.

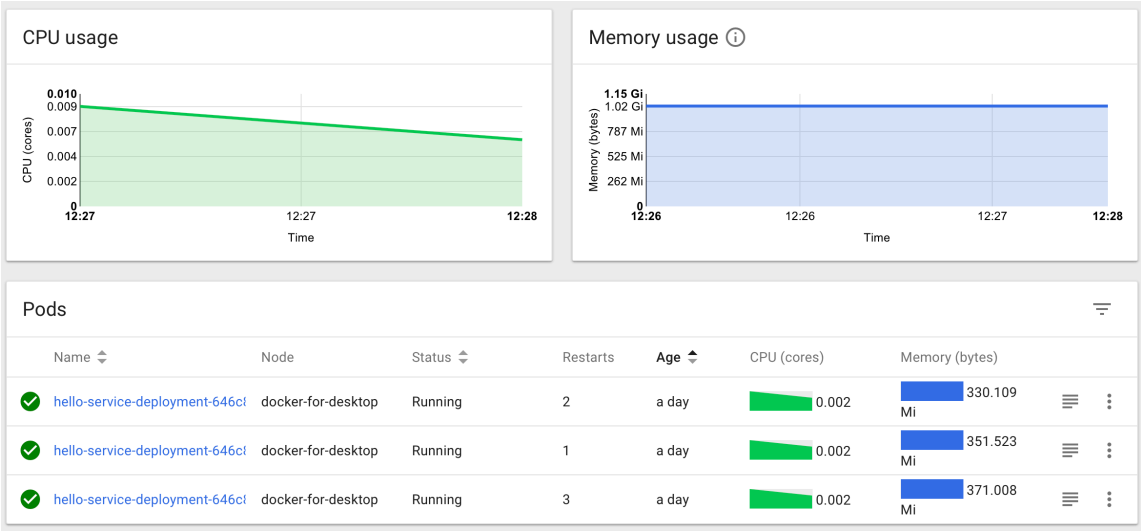


Abbildung 11: Kubernetes WebUI mit CPU und Memory Graphen

Standardmäßig sind diese Graphen nicht im Dashboard enthalten. Falls erwünscht sind diese Komponenten wie folgt dem Kubernetes Cluster hinzuzufügen:

```
$ kubectl create -f
  https://raw.githubusercontent.com/kubernetes/heapster/master/deploy/kube-
  config/influxdb/heapster.yaml
$ kubectl create -f
  https://raw.githubusercontent.com/kubernetes/heapster/master/deploy/kube-
  config/rbac/heapster-rbac.yaml
$ kubectl create -f
  https://raw.githubusercontent.com/kubernetes/heapster/master/deploy/kube-
  config/influxdb/influxdb.yaml
$ kubectl create -f
  https://raw.githubusercontent.com/kubernetes/heapster/master/deploy/kube-
  config/influxdb/grafana.yaml
```

Die Konfigurationsdateien der einzelnen Services können über die angegebenen Links eingesehen werden. Nach Ausführung der oben aufgeführten *kubectl* create Statements dauert es ein wenig bis die enthaltenen Deployments verfügbar und die Graphen im WebUI enthalten sind.

Die Log Dateien von Cloud Native Apps werden nirgends abgelegt, sondern sind als Streams anzusehen. Diese Streams können komfortabel in der graphischen Oberfläche eingesehen werden.

2.2.5 Kubernetes Cluster (cloud)

Nachdem eine Spring Boot Applikation lokal in Kubernetes betrieben wurde, wird diese Applikation beziehungsweise dieses Docker Image nun exemplarisch in der Cloud deployed. Voraussetzungen hierzu ist, Zugriff auf einen Kubernetes Cluster bei einem Cloud-Provider zu haben. Der Kontext des Cloud Clusters ist der lokalen *~/.kube/config* Datei hinzuzufügen. Anschließend kann der kubectl Kontext gewechselt werden:

```
$ kubectl config use-context private.k8s.local
Switched to context "private.k8s.local".
```

Einer der großen Vorteile von Kubernetes besteht darin, dass die containerisierte Java Applikation bis auf wenige Konfigurationen, genauso in der Cloud betrieben wird wie es lokal der Fall ist. Noch vor Aufsetzen des Deployments und der Services, ist die IP-Adresse des Clusters zu erfragen:

```
➜ hello kubectl get nodes
NAME                                         STATUS    ROLES     AGE       VERSION
ip-172-20-32-194.eu-central-1.compute.internal   Ready     master    17m       v1.9.6
ip-172-20-52-94.eu-central-1.compute.internal    Ready     node      16m       v1.9.6
➜ hello kubectl cluster-info
Kubernetes master is running at https://api-private-k8s-local-v4nklj-853246186.eu-central-1.elb.amazonaws.com
```

Abbildung 12: Information zu Nodes und Cluster

Um die deployment.yaml in der Cloud benutzten zu können, ist das *hello* Docker Image der öffentlich zugänglichen Docker Registry bereitzustellen. Falls dies nicht gewünscht ist muss die entsprechende Zugriffsberechtigung eines privaten Repositories der Deployment Konfiguration hinterlegt werden. Die *imagePullPolicy* ist nun auf IfNotPresent gesetzte.

Des weiteren wird nun auf den ServiceTypen „LoadBalancer“ zurückgegriffen. Dieser kann verwendet werden, falls der genutzte Cloud Provider einen zur Verfügung stellt. So kann Kubernetes beispielsweise den LoadBalancer von AWS nutzen, um den Traffic eines angefragten Deployments auf die einzelnen Pods zu verteilen. Die Beschreibung (siehe Abbildung 13; „LoadBalancer Ingress“) des angepassten Services gibt die URL, über die das Deployment erreichbar ist, preis.

```
➜ hello kubectl describe service hello-service
Name:                     hello-service
Namespace:                default
Labels:                   <none>
Annotations:              kubectl.kubernetes.io/last-applied-configuration={"apiVersion":"v1","kind":"Servic
e","metadata":{"annotations":{},"name":"hello-service","namespace":"default"},"spec":{"ports":[{"nodePort":3
2100,"port...
Selector:                 app=hello-service
Type:                     LoadBalancer
IP:                       100.70.30.27
LoadBalancer Ingress:     a63e87b07658011e894fc02403ce0308-2137235333.eu-central-1.elb.amazonaws.com
Port:                     <unset>  9090/TCP
TargetPort:               8080/TCP
NodePort:                 <unset>  32100/TCP
Endpoints:                100.96.1.7:8080,100.96.1.8:8080
Session Affinity:         None
External Traffic Policy:  Cluster
Events:
  Type    Reason                Age   From                Message
  ----    ------                ----  ----                -------
  Normal  Type                  4m    service-controller  NodePort -> LoadBalancer
  Normal  EnsuringLoadBalancer  4m    service-controller  Ensuring load balancer
  Normal  EnsuredLoadBalancer   4m    service-controller  Ensured load balancer
➜ hello curl http://a63e87b07658011e894fc02403ce0308-2137235333.eu-central-1.elb.amazonaws.com:9090/hello
{"message":"Spring Boot Application is running and says Hello K8s"}%
```

Abbildung 13: Curl on LoadBalancer external Endpoint

Gerade in diesem Zusammenhang ist es wichtig, dass Anwendungen den Anforderungen von Cloud Native Applikationen gerecht werden, da der Traffic auf viele verschiedene Instanzen geroutet wird. (“Using a Service to Expose Your App,” 2018)

2.3 Alternativen

Es herrscht die allgemeine Meinung, dass der Betrieb und die Installation der Kubernetes Plattform komplex sei. (Wright, 2017) Daher wird in diesem Kapitel auf Alternativen zu Kubernetes eingegangen, um einen Überblick über konkurrenzfähige Plattformen zu verschaffen.

Neben Plattformen die Docker Container betreiben gibt es die Möglichkeit Cloud Native Applikationen beispielsweise direkt in einer EC2 Instanz auf AWS zu starten. Dies ist allerdings viel aufwändiger zu betreiben. Zudem fallen Vorteile wie Skalierbarkeit und Hochverfügbarkeit weg. Diese Nachteile versucht Amazon auszugleichen, indem sie zum Beispiel „Elastic Beanstalk“ anbieten. Notwendig ist nur das Hochladen des Codes und der Service von Amazon kümmert sich um die Skalierung und Hochverfügbarkeit dieser Anwendung. Kosten für den Service selber fallen keine an. Sie beschränken sich auf die AWS-Ressourcen welche für

die Ausführung und Speicherung der Anwendung benötigt werden. Ein Nachteil besteht jedoch darin, dass das Aufsetzen mehrerer Micro-Services schnell aufwendig wird. (“AWS,” 2018)

Vorteilhaft sind daher Plattformen die den Betrieb mehrerer Anwendungen beziehungsweise Services managen. Zu den namenhaften Konkurrenten von Kubernetes zählen unteranderem folgende Produkte (“Slant,” 2018):

- Docker Swarm
- OpenShift
- Apache Mesos

Um den Rahmen der Arbeit nicht zu sprengen, beschränkt sich der Vergleich von K8s auf Docker Swarm. Eine Auswertung von „Platform9“ aus dem Jahr 2017, in der Kubernetes und Docker Swarm in fünf Kategorien verglichen werden, ist nachfolgend abgebildet:

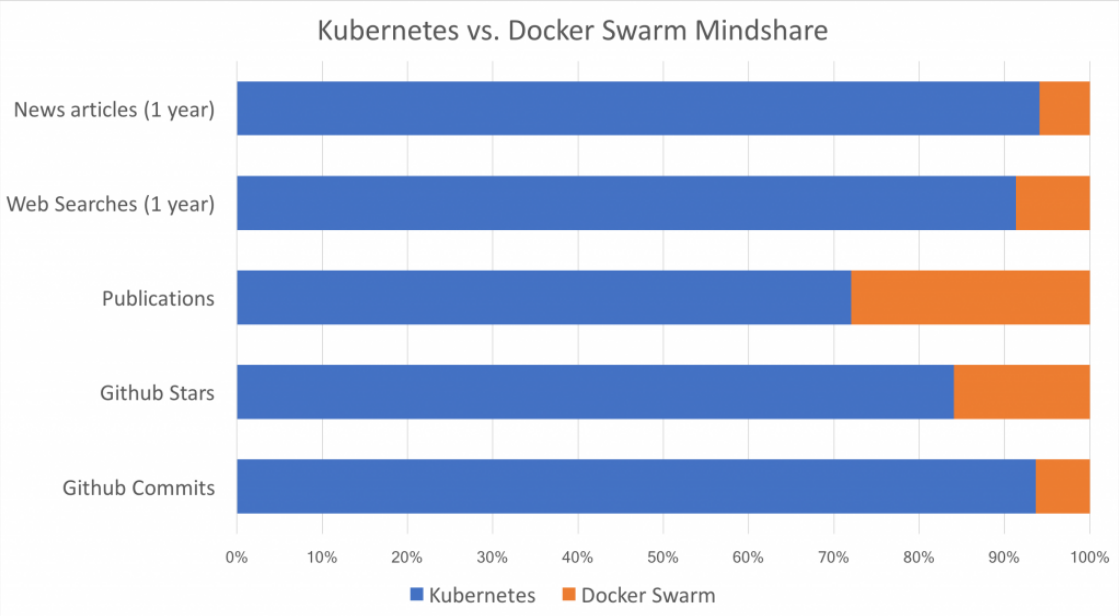


Abbildung 14: Kubernetes versus Docker Swarm (Wright, 2017)

Aus Abbildung 14 geht hervor, dass Kubernetes im Gegensatz zu Docker Swarm beliebter ist. Außerdem kann gedeutet werden, dass Kubernetes eine größere Community hat, da Kubernetes in der Kategorie „Github Commits“ und „Publications“ eindeutig dominiert.

3 Fazit & Ausblick

Der Betrieb von Cloud Native Apps in der Cloud kann kompliziert sein. Die Open-Source Plattform Kubernetes schafft an dieser Stelle Abhilfe. K8s kann sowohl in der Cloud als auch auf eigenen Servern betrieben werden. Kubernetes ist nicht nur praktisch um Spring Boot Java Applikationen zu betreiben, sondern für jegliche Art von Docker Images geeignet. Kubernetes verwaltet somit eine Vielzahl verschiedener Docker Container und bietet Vorteile wie Skalierung, RollingUpdates und Hochverfügbarkeit von Anwendungen und Diensten. Diese Universalität macht Kubernetes so einzigartig, mächtig und beliebt.

Ein Nachteil an Kubernetes jedoch ist die Komplexität der immer größer werdenden Plattform. Eine Alternative zu Kubernetes ist beispielsweise Docker Swarm. Es ist allerdings eindeutig, dass Kubernetes durch den jahrelangen Einsatz bei Google seine Berechtigung gefestigt hat. Die Kubernetes Plattform wächst dank der großen Community und entwickelt sich dadurch bedingt laufend weiter. Um an aktuelle und detaillierte Informationen zu gelangen ist die offizielle Dokumentation unter http://www.kubernetes.io/ zu empfehlen. Des weiteren ist unter https://kubernetes.io/docs/tutorials/online-training/overview/ eine Übersicht von Online Trainings aufgelistet.

Schlagwörter wie 4.0, Internet of Things und viele weiter prägen zurzeit die Medienlandschaft. Diese neuartigen Technologien werden höchstwahrscheinlich die Bewegung Richtung Cloud vorantreiben. Der Betrieb von Cloud Native Applikationen in der Cloud wird somit immer mehr an Wichtigkeit gewinnen und einen Beitrag zur Wirtschaft leisten.

4 Literatur- und Quellenverzeichnis

AWS [WWW Document], 2018. . AWS Elastic Beanstalk. URL https://aws.amazon.com/de/elasticbeanstalk/?nc2=h_m1 (accessed 5.28.18).

Clutch [WWW Document], 2018. . Best Cloud Serv. Provid. URL https://clutch.co/cloud (accessed 5.24.18).

Configure Liveness and Readiness Probes - Kubernetes [WWW Document], 2018. URL https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-probes/ (accessed 6.1.18).

Horizontal Pod Autoscaler Walkthrough - Kubernetes [WWW Document], 2018. . Kubernetes. URL https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale-walkthrough/ (accessed 5.31.18).

Run a Stateless Application Using a Deployment - Kubernetes [WWW Document], 2018. URL https://kubernetes.io/docs/tasks/run-application/run-stateless-application-deployment/ (accessed 6.1.18).

Schane, L., 2017. JVM Microservice with spring boot, docker and kubernetes [WWW Document]. Creat. Microservice Proj. Using Spring Boot. URL https://blog.shanelee.name/2017/07/15/jvm-microservice-with-spring-boot-docker-and-kubernetes/ (accessed 5.24.18).

Services - Kubernetes [WWW Document], 2018. URL https://kubernetes.io/docs/concepts/services-networking/service/ (accessed 6.1.18).

Slant [WWW Document], 2018. . What Best Altern. Kubernetes. URL https://www.slant.co/options/11649/alternatives/~kubernetes-alternatives (accessed 5.28.18).

Spring Boot with Docker [WWW Document], 2018. URL https://spring.io/guides/gs/spring-boot-docker/ (accessed 5.26.18).

Spring Initializr [WWW Document], 2018. . Spring Initial. URL https://start.spring.io (accessed 5.24.18).

Theis, M., 2018. Cloud Native Plattformen [WWW Document]. URL https://tschutschu.de/resources/tschutschu/docs/SS2018/SS2018_08_CNA_Plattformen.pdf (accessed 5.31.18).

Using a Service to Expose Your App [WWW Document], 2018. . Kubernetes. URL https://kubernetes.io/docs/tutorials/kubernetes-basics/expose/expose-intro/ (accessed 5.30.18).

Using Minikube to Create a Cluster - Kubernetes [WWW Document], 2018. . Kubernetes. URL https://kubernetes.io/docs/tutorials/kubernetes-basics/create-cluster/cluster-intro/ (accessed 5.28.18).

What is Kubernetes [WWW Document], 2018. . Kubernetes. URL https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/ (accessed 5.27.18).

Wright, C., 2017. Platform9 [WWW Document]. Kubernetes Docker Swarm. URL https://platform9.com/blog/kubernetes-docker-swarm-compared/ (accessed 5.28.18).