# Einleitung

Wir wollen hier zeigen, wie eine containerisierte Anwendung in der Cloud aussehen kann, und wie wir uns die Entwicklungspipeline, das Monitoring und den Lasttest vorstellen.

Grundlage ist eine rudimentäre Webapplikation aus 3 Teilen, die einen Webshop simuliert:

* Artikel sind in einer Datenbank gespeichert,
* werden von Appservern ausgelesen und präsentiert,
* die die Einkäufe an eine Zahlungsdienstleister-Schnittstelle weiterreicht

Es ist eine Beispielapplikation für Docker, die wir von Docker Swarm auf Kubernetes migriert haben.

Außerdem haben dafür eine CI/CD-Pipeline und ein Monitoring implementiert, sowie ein Lasttest-Szenario geschaffen.

Alle Bestandteile laufen in der AWS-Cloud. Das sind:

* KOPS-Instanz. Wird komplett mit Cloudformation aufgesetzt. Hiermit setzen wir per bash-Befehl den Cluster und ggf. den Lasttest auf und administrieren den Cluster manuell, sollte das nötig werden (die Fälle, die nicht durch die CI/CD-Pipeline abgedeckt werden); dient auch als Bastion-Host.
* Kubernetes-Cluster. Hierin läuft die Anwenung und das Monitoring. Wird mit bash-Befehl von der KOPS-Instanz aus aufgesetzt.
* Jenkins. Buildserver. Wird mit Cloudformation aufgesetzt und muss aktuell danach manuell weiter bearbeitet werden.
* SonarQube-Instanz. Wird mit Cloudformation aufgesetzt und muss aktuell danach manuell weiter bearbeitet werden.
* Lasttest-Szenario. Temporär. Wird mit bash-Befehl von der KOPS-Instanz aus aufgesetzt.

KOPS-Instanz haben wir bereits vorbereitet, ebenso Jenkins und Sonarqube. Wir zeigen Ihnen nun, wie wir den Kubernetes-Cluster aufsetzen:

(ssh gegen die KOPS-Instanz)

kops create cluster ${AWS\_CLUSTER\_NAME} --image=${AWS\_CLUSTER\_IMAGE} --zones=${AWS\_AVAILABILITY\_ZONES} --master-size=${AWS\_MASTER\_SIZE} --node-size=${AWS\_NODE\_SIZE} --networking=flannel --ssh-access="139.2.0.0/16" --ssh-public-key="~/.ssh/authorized\_keys" --state=${KOPS\_STATE\_STORE} --cloud aws --topology private --yes

Es dauert eine Weile, bis der Cluster betriebsbereit ist. Man kann das überprüfen mit:

kops validate cluster

Danach wird die AWS-Policy angewandt, die für den Cluster-Autoscaler benötigt wird:

createpolicy.sh

Und die Instance-Group für die Nodes wird dazu passend geändert:

Dies ändert die max. Anzahl an Nodes für den Cluster-Autoscaler auf 10.

Screts für die Anwendung erzeugen (dieser Schritt könnte auch automatisch deployed werden, aber dann stünden die Secrets in einer .yaml-Datei):

kubectl create secret generic secrets --from-file=revprox-cert=certs/domain.crt --from-file=revprox-key=certs/domain.key --from-literal=postgres-password="gordonpass" --from-literal=payment-token="staging"

Der Cluster-Autoscaler läuft in einem Container und wird parallel zu der Anwendung und dem Monitoring deployed:

kubectl create -f ./kubedeployment

kubectl create -f ./metricdeploy

Ergebnis prüfen:

kubectl get po

kubectl get po –n kube-system

Das Monitoring wird in den Namespace kube-system deployed.

Nun kann man sich mit den einzelnen Services verbinden:

Webshop-URL ergibt sich aus:

aws elb describe-load-balancers --query 'LoadBalancerDescriptions[?ListenerDescriptions[?Listener.LoadBalancerPort==`5005`]]|[0].DNSName' | sed s/\"//g | sed s#^#https://# | sed 's#$#/index.html#'

Die URLs der anderen Dienste erhält man mit

kubectl cluster-info

Um sich mit dem Kubernetes-Master zu verbinden: /ui an die URL des Masters anhängen. Sonst bekommt man nur eine JSON-Übersicht.

(Benutzername &) Kennwort bekommt man mit:

cat ~/.kube/config

Das Kennwort dient auch als „Token“. Auf der /ui-Seite braucht man keinen Benutzernamen.

!!! Heapster -> 404 Page not found (hat keine Seite)

!!! Kube-DNS -> Timeout

!!! Grafana -> Leeres Dashboard !!! hier muss noch was getan werden

!!! Influx-DB -> 404 Page not found (hat keine Seite)

1. Automatisches Aufsetzen einer neuen Cloud-basierten Umgebung (nebenläufig)
2. Automatisierung
   1. Schnellüberblick WebShop
   2. Java Code Anpassung mit SonarQube Rule Verletzung inkl. Git commit (Start der Pipeline)
   3. CI/CD Pipeline zeigen (Quality Gates, Tests,
   4. Rule Verletzung beheben im Java Code, anschl. neuer Pipeline Lauf
   5. Manueller Trigger für PROD Deployment)
   6. Prüfen des neuen Deployments (neue Applikation)
3. Infrastruktur zeigen
   1. AWS Konsole:
      1. EC2 Instanzen
      2. Load Balancer
      3. Security Groups, Access Control Lists, etc.
      4. Evtl. Cloud Formation Template bez. Security zeigen
   2. AWS Cloudwatch: Auslastung, Metriken
   3. Kubernetes Cluster
      1. Heapster/ kubectl (Master Node, Worker Nodes, Pods)
   4. Prometheus
4. Monitoring
   1. AWS Problem erzeugen: EC2 Instanz abschießen, DB runterfahren
      1. Cloudwatch Monitoring
   2. Kubernetes Problem erzeugen: Pods (?) abschießen
      1. Heapster / kubectl
   3. Applikationsmonitoring
      1. Prometheus
5. Rollback
   1. Fehlerhaften Java Code einchecken und Deployment durchführen
   2. Applikationsfehler über Web UI zeigen
   3. Daten für Post Mortem Analyse sichern
   4. Automatischen Rollback antriggern
   5. Webshop funktioniert wieder mit alter Version
6. Lasterzeugung, Autoscaling
   1. JMeter VM in AWS verwenden, um System unter Last zu setzen
   2. Monitoring
      1. AWS Cloudwatch: Lasterhöhung
7. „Zähigkeit“ Web UI zeigen
8. Autoscaling beobachten (EC2 Instanzen, Load Balancer)
9. Nach Scaling sollte Web UI wieder flüssig laufen