# Hinweis: Die Stellen, die sich evtl. nochmal ändern, sind türkisfarben hinterlegt.

# 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. Die CI/CD-Pipeline besteht aus:

* GIT-Repository (auf Github)
* Jenkins-Buildserver (auf EC2-Instanz)
* Sonarqube Quality Gate (auf EC2-Instanz)
* AWS Elastic Container Service Docker-Repository
* KOPS-EC2-Instanz zur Installation des Clusters, Steuerung des Kubernetes-Deployments von Jenkins (und zum initialen Aufsetzen des Kubernetes-Clusters, zur manuellen Administration und als Bastion-Host).

Die Anwendung und das Monitoring wird in Containern in einem Kubernetes-Cluster auf AWS deployed. Wir verwenden nicht die AWS-eigene Kubernetes-Implementation, sondern setzen ihn selbst auf.

Außerdem haben wir ein temporär existierendes Lasttest-Szenario, welches ebenfalls in der AWS-Cloud aufgesetzt wird und per ssh von der KOPS-Instanz aus gesteuert wird.

# Aufsetzen der KOPS-Instanz

Die KOPS-Instanz wird mittels eines CloudFormation-Templates aufgesetzt. Danach ist dort alle benötigte Software installiert. Das erste Aufsetzen des Kubernetes-Clusters führen wir manuell durch; dafür müssen die Webshop-Anwendung und die Kubernetes-YAML-Dateien dort vorhanden sein; diesen Teil haben wir nicht verskriptet, obwohl das möglich gewesen wäre. Weil das Deployment der KOPS-Instanz wegen der Softwareinstallationen und der genannten manuellen Schritte eine Weile dauert, haben wir bereits eine fertige KOPS-Instanz vorbereitet.

# Aufsetzen des Kubernetes-Clusters

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 (maxSize=10):

kops edit ig nodes

kops update cluster --yes

oder auch mit dem folgenden Shellskript:

~/git/368863656547/config/autoscaleingignodes/setmaxminkopsignodes.sh

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

Secrets 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

./metricdeploy/monitoring.sh

Ergebnis prüfen:

kubectl get po

kubectl get po –n kube-system

Das Monitoring wird in den Namespace kube-system und monitoring deployed.

Es dauert eine Weile, bis alles läuft. Manchmal wird der appserver restartet, weil die Datenbank langsamer startet.

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

oder

kubectl config view --minify

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

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

Mit /healthz bekommt man „ok“

!!! Kube-DNS -> Timeout

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

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

# Aufsetzen von Jenkins (CI/CD-Pipeline)

Jenkins läuft auf einer eigenen EC2-Instanz. Ein großer Teil der Installation kann automatisch gemacht werden, danach schließen sich aber einige Schritte an, die manuell (über den Webbrowser) erfolgen müssen, weil sie nicht verskriptet werden können. Das Pipeline-Skript (welches den konkreten Build- bzw. Deploymentvorgang beschreibt, ist von Projekt zu Projekt verschieden und muss ggf. ausgetauscht werden.

Weil das Deployment von Jenkins wegen der Softwareinstallationen und der genannten manuellen Schritte eine Weile dauert, haben wir bereits eine fertige Jenkins-Instanz vorbereitet.

# Aufsetzen von SonarQube (Quality Gate)

Auch Sonarqube läuft auf einer eigenen EC2-Instanz. Die Installation kann automatisch gemacht werden, danach schließen sich aber einige Schritte (Webhook, Jacoco-Pfad) an, die manuell (über den Webbrowser) erfolgen müssen, weil sie nicht verskriptet werden können; zudem muss natürlich das Quality Gate den jeweiligen Erfordernissen angepasst werden, was ziemlich aufwändig ist.

Weil das Deployment von Sonarqube wegen der Softwareinstallationen und der genannten manuellen Schritte eine Weile dauert, haben wir bereits eine fertige Sonarqube -Instanz vorbereitet.

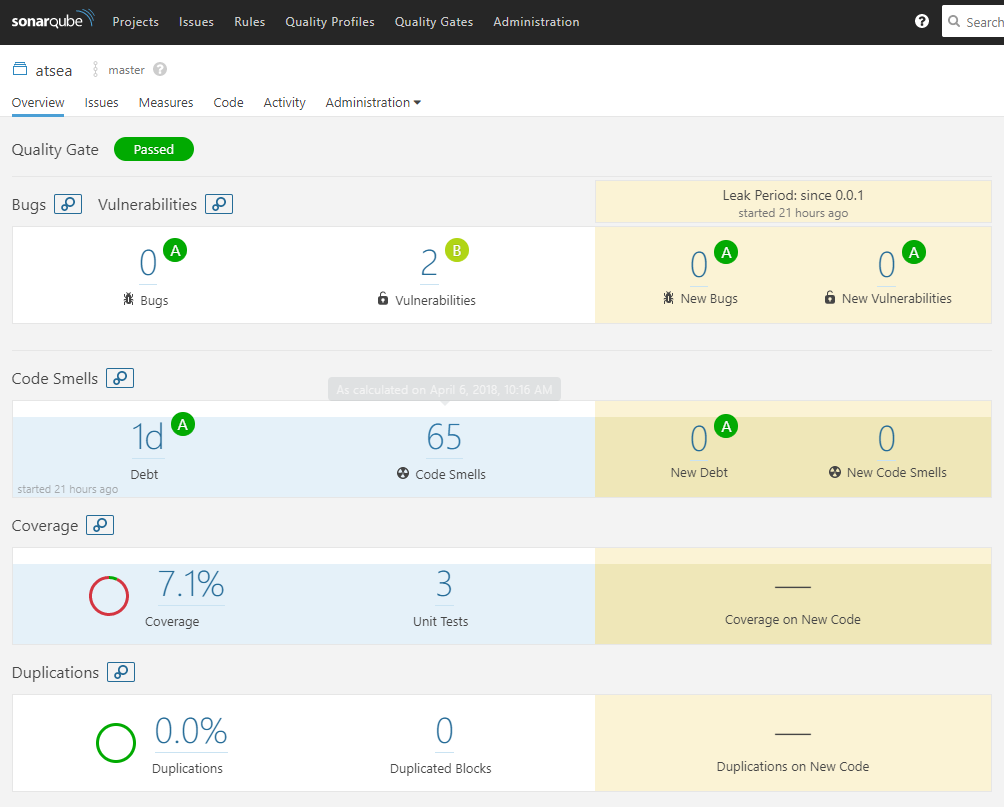
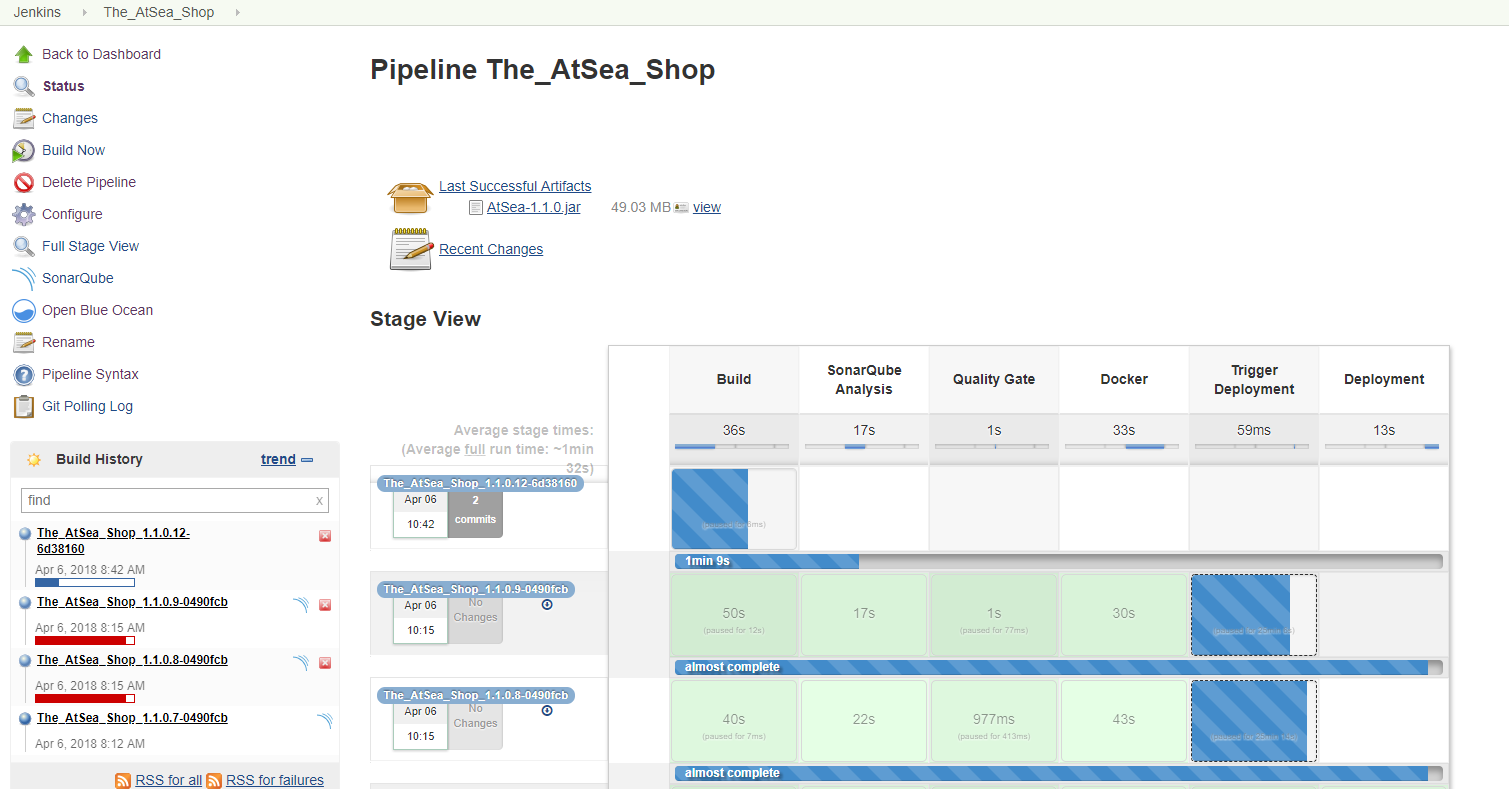
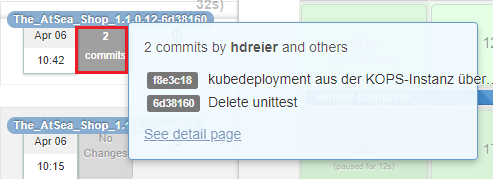
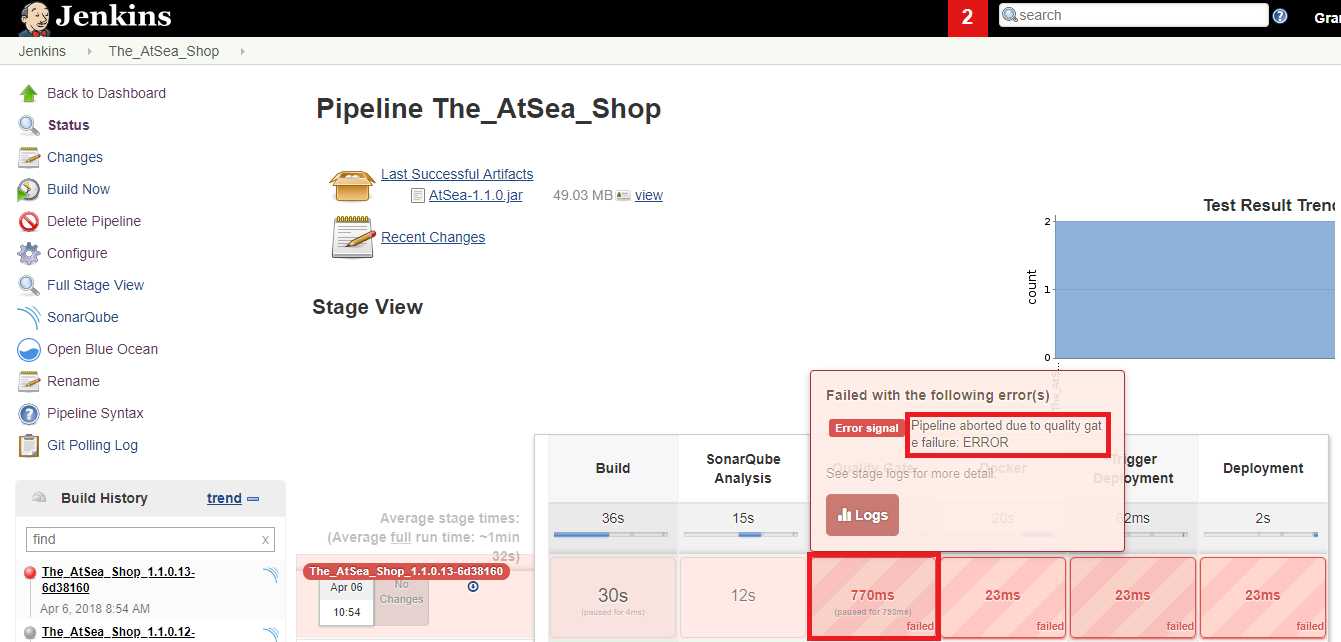
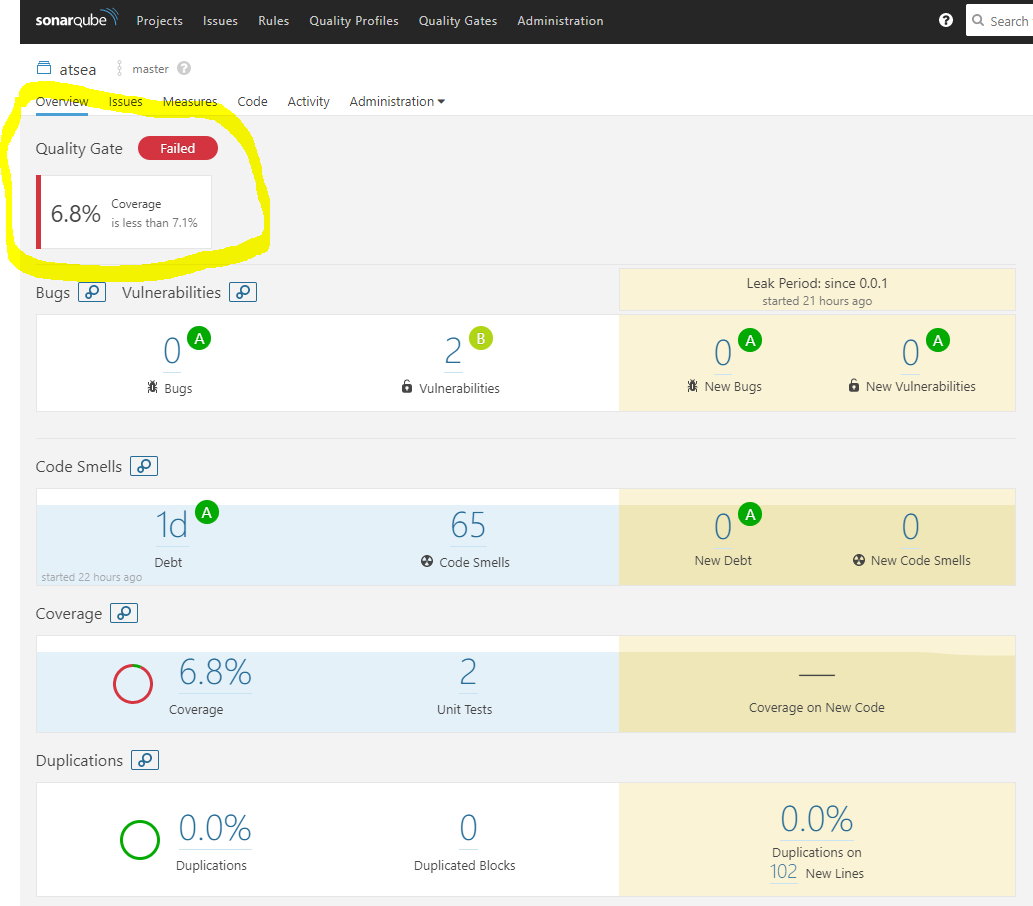
# CI/CD-Pipeline im Betrieb

Im laufenden Betrieb finden alle Deployments neuer Anwendungsversionen über Jenkins statt; sie können manuell angestoßen werden, oder (Standard) indem Jenkins in Abständen (hier: 1 Min.) prüft, ob ein Commit vorliegt und dann die Build-Pipeline automatisch startet. Zudem können weitere, manuell zu startende Jobs eingerichtet werden (hier z.B. der Rollback-Job). Wir zeigen hier beispielhaft typische Abläufe.

# Java Code Anpassung mit SonarQube Rule Verletzung inkl. Git commit (Start der Pipeline)

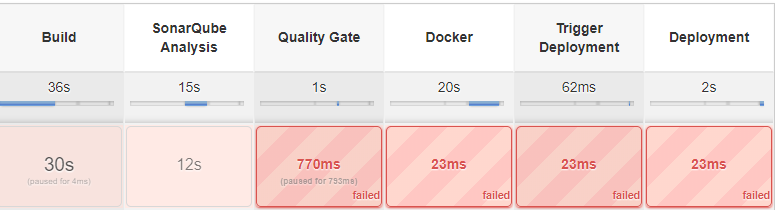
Grundlagen:

* Jenkins
  + admin:admin
  + ec2-18-184-24-58.eu-central-1.compute.amazonaws.com:8080
* GitHub
  + 368863656547:Materna.123
  + https://github.com/368863656547/368863656547.git
* Sonarqube
  + admin:admin
  + ec2-18-196-192-76.eu-central-1.compute.amazonaws.com:9000

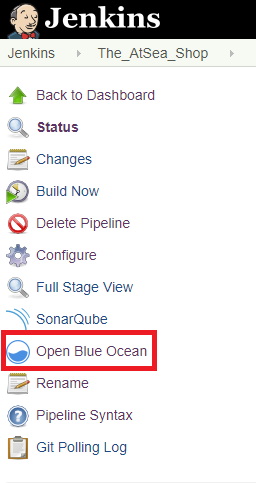
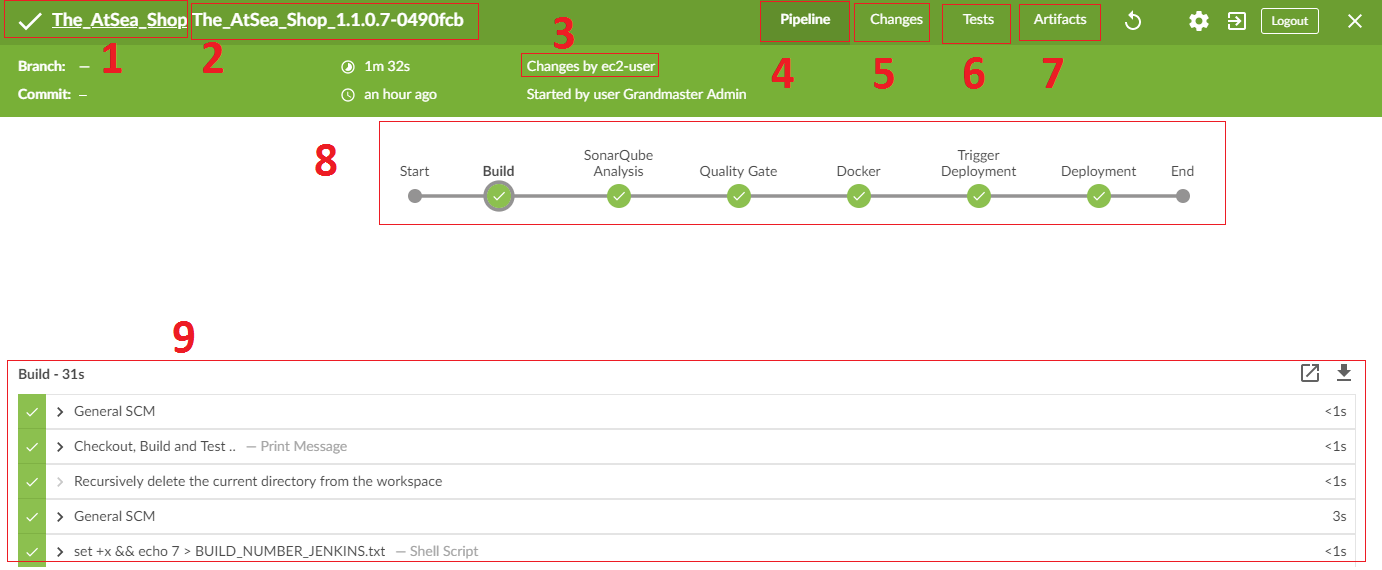
1. Aufruf Sonarqube
   1. Sonarqube aufrufen: ec2-35-158-78-192.eu-central-1.compute.amazonaws.com:9000
   2. Project „atsea“ aufrufen.
   3. Zeigen, dass dort keine Regeln verletzt sind.  
      
   4. Stand jetzt, es sind alle Sonar Anforderungen erreicht und das Quality Gate ist passend.
2. Jenkins aufmachen.
   1. Dort die Pipeline aufmachen und zeigen, dass alles gut ist  
      
3. Nun Commit nach GitHub.
   1. Unter „\app\app\src\test\java\com\docker\atsea\test\UnitTest.java“ einen Unittest auskommentieren.
   2. Änderung nach GitHub comitten.
   3. Nach dem Commit rennt der Jenkins in der Pipeline los und die App wird gebaut.
   4. Die Pipeline läuft los und mit Mouseover auf Commits, siehst du dann auch deinen Commit:  
      
   5. Da wir nun keine 7.1% CodeCoverage haben, ist das Quality Gate nicht erreicht und der Build knallt:
   6.   
      Wenn du mit der Maus auf die QualityGate Stage gehst dann öffnetsich das Fenster mit der Fehlermeldung und wir sehen, Quality Gate failure.
4. Wir wechseln zurück nach Sonarqube
   1. Das atsea Project sollte noch offen sein.
   2. Hier sehen wir nun auch, dass das QualityGate nicht erreicht wurde und dass die CodeCoverage Schuld ist:  
      
5. Showcase ist damit durchgelaufen.

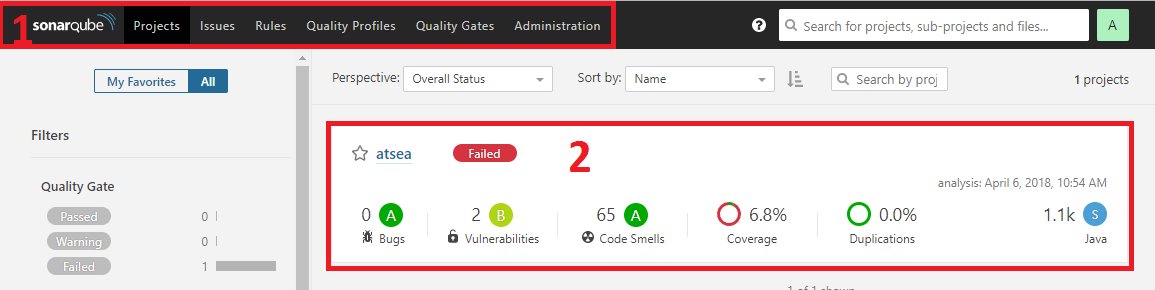
CI/CD Pipeline zeigen (Quality Gates, Tests,

1. Jenkins Pipeline Classic
   1. Gehe auf den Jenkins



* + 1. Build = Build und Unittests
    2. SonarQube Analysis = Sonaranalyse
    3. Quality Gate = Check ob Quality Gate erreicht.
    4. Docker = Erstelle Docker Container, Tag ihn und Push ihn.
    5. Trigger Deployment= Willst du deployen ja oder nein? Mouseover und dann Promote!
    6. Deployment = Deployment gegen den Cluster.

1. Jenkins Pipeline Blue Ocean
   1. Erstmal auf den Jenkins gehen.
   2. Dann ins Projekt gehen.
   3. Im Projekt gibt es einen Button BlueOcean:  
      
   4. Nun landest du bei Blue Ocean:  
        
      1 = Projektname  
      2 = Buildname  
      3 = Changes from User  
      4 = Zeig mir die Pipeline  
      5 = Zeig mir die Changes  
      6 = Zeig mir die Tests  
      7 = Zeig mir die Artefakte des Builds (ich lasse die Jar hochladen)  
      8 = Auflisten der Pipeline Stages! Wenn du auf einen der Kreise/Stages klickst, wird dir …  
      9 = der Ablauf der Aufgaben der Stage angezeigt.
2. SonarQube
   1. Gehe nach Sonarqube



1 = Menü  
2 = Projekte

Rule Verletzung beheben im Java Code, anschl. neuer Pipeline Lauf

* Einfach hier den Unittest wieder reinnehmen und einchecken.

Manueller Trigger für PROD Deployment)

… zeigen wir jetzt hier nicht; man würde einfach einen weiteren Triggfer und ein weiteres Deployment an die Pipeline anhängen. Man kann auch eine Liste von Deployments angeben (die werden dann in der Pipeline optisch untereinander angezeigt), von denen manche ausgewählt werden sollen und andere z.B. nicht, sobald der Build an der entsprechenden Stelle angekommen ist.

Prüfen des neuen Deployments (neue Applikation)

* Wenn du eine Änderung eincheckst dann wird diese deployed und du siehst dann die neue App nach dem Deployment.

Rollback

Fehlerhaften Java Code einchecken und Deployment durchführen

* Einchecken usw. hast du ja bereits gemacht. Das Deployment läuft durch und die kaputte App wird deployed wenn:
  + Unittests noch durchlaufen
  + Quality Gate nicht failed

Beispiel (auf der KOPS-Instanz):

cd ~/git/368863656547

vi app/app/react-app/src/components/Header/index.js

„Welcome to the atsea shop“ ändern in „Welcom to the at-sea shop“

git status

git commit -a

Den commit-Kommentar editieren & speichern

git push

Benutzername/Passwort: 368863656547:Materna.123

Jenkins aufrufen: <http://ec2-18-184-24-58.eu-central-1.compute.amazonaws.com:8080>

admin/admin: Build läuft automatisch los.

Deployment-Trigger auslösen. Nach ~15s ist das Deployment durch.

* Dann kannst du die kaputte App zeigen und fertig.

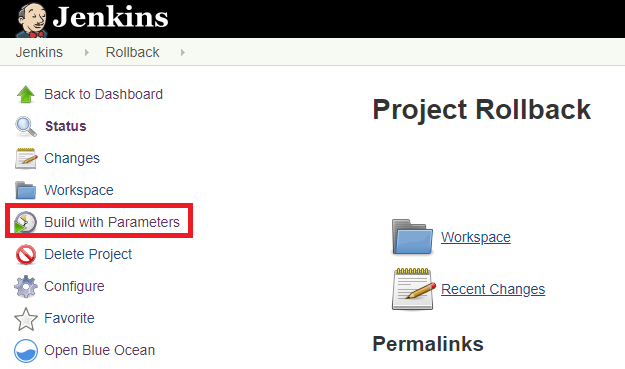
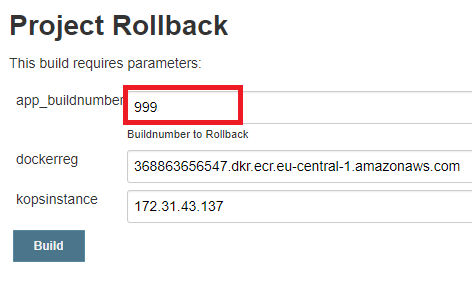
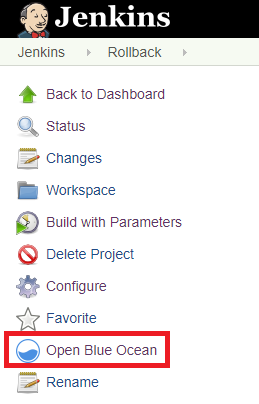
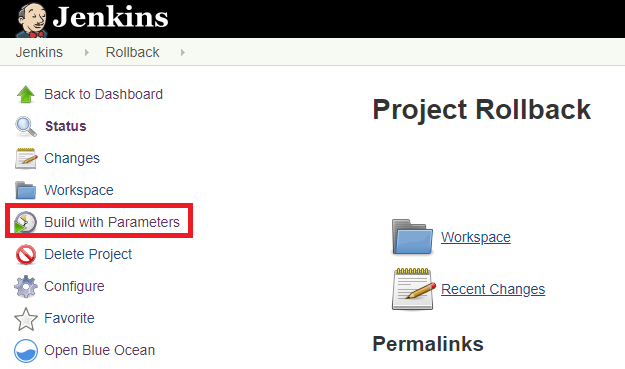
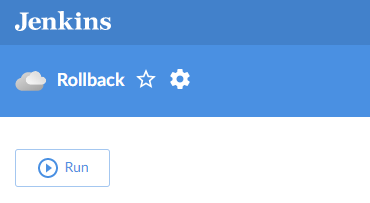
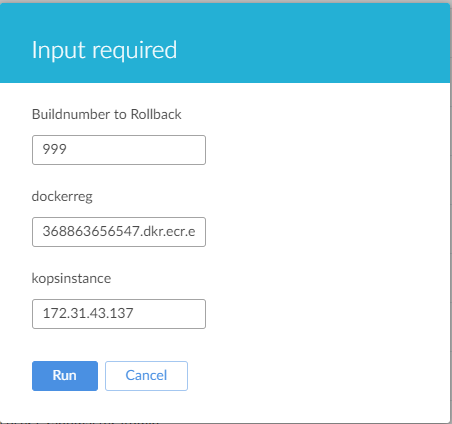
Applikationsfehler über Web UI zeigen

Atseashop zeigen: Schwerer Fehler.

Daten für Post Mortem Analyse sichern

* Logs des Appserver-Pods des vorangehenden Starts abrufen:
  + kubectl logs appserver-fcbdd794f-qtkf7 –p
  + Im konkreten Fall sind da natürlich keine Fehler zu sehen.

Automatischen Rollback antriggern

* Man könnte voll automatisieren, will man aber meist nicht (weil nicht immer klar ist, welche Version die richtige ist – nicht jeder Fehler fällt im Build auf).
* Zunächst musst du die letzte funktionierende Versionsnummer identifizieren
  + Wechsel nach AWS Amazon ECS
  + Dort appserver öffnen  
    [https://eu-central-1.console.aws.amazon.com/ecs/home?region=eu-central-1#/repositories/appserver#images;tagStatus=ALL](https://eu-central-1.console.aws.amazon.com/ecs/home?region=eu-central-1#/repositories/appserver)
  + Hier die letzte gesunde Version auslesen z.B. 1.1.0.7
* Auf den Jenkins wechseln
  + Rollbackprojekt auswählen
  + Jenkins Classic
    - Build with Parameters auswählen  
      
    - Hier die Buildnummer eintragen  
      
    - Auf Build hauen und der Rollback läuft.
  + Rollbackprojekt auswählen
  + Jenkins BlueOcean aufrufen  
    
    - Build with Parameters auswählen  
      
    - Auf Run Klicken  
      
    - Ein Fenster öffnet sich. Hier kann man dann die Versionsnummer auf die zurück gerollt werden soll, eingetragen werden. Dann auf Run und Rollback läuft:
    - 

Webshop funktioniert wieder mit alter Version

* Ab auf die Seite und es sollte wieder gehen.

# Monitoring

# Lasttest

Für den Lasttest haben wir einen CloudFormation-Stack. Die Lasttest-Konfiguration besteht aus einem jMeter-Master und 4 Nodes, die die Last erzeugen.

Deployment des jMeter-Clusters:

aws cloudformation create-stack --capabilities CAPABILITY\_NAMED\_IAM --stack-name Lasttest --template-body <file://jMeterCluster.json>

Schritte, um den Lasttest auszuführen:

Im Atseashop einen Benutzer "hdreier" mit Passwort "xyz anlegen",

Atseashop.jmx bearbeiten und die URL gegen die aktuelle des Loadbalancers austauschen,

lokal (Windows) mit jmeter öffnen und die Parameter anpassen (Ramp-up time etc.)

in das /tmp-Verzeichnis des jMeterMasters kopieren,

Lasttest starten:

# mit logs

/apache-jmeter-4.0/bin/./jmeter -n -t /tmp/Atseashop.jmx -r -e -o /tmp/output -l /tmp/atsea.log

#ohne logs

/apache-jmeter-4.0/bin/./jmeter -n -t /tmp/Atseashop.jmx -r

Nachdem man die Ergebnisse gesichert hat, Verzeichnis / Logdatei löschen, bevor man neu startet.

Der Webshop tut nicht viel; er ist viel schneller als die jMeter-Nodes. Trotzdem sollte sich ein Scale-Up des Clusters beobachten lassen. Notfalls kann man, um das zu zeigen, die greeter-App in einem Kubernetes-Container deployen; die erzeugt Last direkt auf den Kubernetes-Nodes.

# Probleme, mit denen man zu kämpfen hat

* Kubernetes/Kops ist in ständiger schneller Entwicklung; Dokumentation ist teilweise dürftig, wenn es um spezielle Details geht. Stackoverflow hilft einem hier meist auch nicht weiter, weil noch kein großer Erfahrungsschatz vorliegt.
* AWS ist zuverlässig, aber v.a. die t2-Instanzen zeigen manchmal ein Throttling-Verhalten, obwohl noch CPU-Credits übrig sind. Das fällt beim Lasttest auf.
* Wir würden am Liebsten alles verskripten, denn nur so kann man etwas wirklich reproduzierbar halten (und spart auf die Dauer auch noch jede Menge Zeit und Doku-Aufwand). Leider hat sich das noch nicht bis zu allen Toolentwicklern durchgesprochen, bzw. die Vorgehensweisen sind schlecht beschrieben.

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