Hochschule Wismar
University of Technology, Business and Design
Fakultät für Ingenieurwissenschaften

Praktikumsbericht

Continuous-Integration und Delivery für E-Government Anwendungen

Eingereicht am: 22. Februar 2019

von: Patrice Matz
geboren am 17. Juli. 1997
in Neubrandenburg

Aufgabenstellung

Ziel ist es, aufbauend auf einer bestehenden Continuous-Integration-Pipeline, eine exemplarische prototypische Implementierung einer Continuous-Delivery-Pipeline für Anwendungen im E-Government Umfeld vorzunehmen.

Im Zuge dessen sollen verschiedene Möglichkeiten zur Umsetzung, hinsichtlich ihrer Anforderungen und Vorteile verglichen werden. Zusätzlich werden auch eine Erweiterung und teilweise Umstellung der bestehenden Continuous-Integration-Pipeline auf Docker-Container untersucht.

Inhalt

1 Einleitung 5
2 Grundlagen 6
2.1 Die Continuous-Integration- / Continuous-Delivery-Pipeline 6
2.1.1 Die einfachste Pipeline 6
2.1.2 Die Continuous-Integration-Pipeline 6
2.1.3 Die Continuous-Delivery und Deployment Pipeline 7
2.2 Docker 8
2.2.1 Unterscheidung zwischen Docker-Container und Virtuelle Maschine 8
2.2.2 Dockerfile 9
2.2.3 Docker Image 10
2.3 Docker Orchestrierung 12
2.3.1 Docker Compose 12
2.3.2 Kubernetes 12
3 Continuous-Integration 15
3.1 Jenkins und Jenkins Pipeline 15
3.2 Zalenium 16
3.2.1 Beispiel 19
4 Continuous-Delivery 20
4.1 VMware Integrated Containers (VIC) 20
4.2 OKD 22
5 Analyse des bestehenden Workflows 24
6 Anforderungsspezifikation 26
6.1 Anforderungen an die CI Pipeline 26
6.2 Anforderungen an die CD Pipeline 27
6.3 Anforderungen an die Orchestrierungsplattform 27
7 CI/CD mit Jenkins und Docker-Containern 28
8 „Dockerisierung“ von ServO.MV und Erstellen der CI/CD-Pipeline 30
8.1 OpenShift als Orchestrierungsplattform 30
8.2 Gewähltes Verfahren 30
8.3 Gewählte Applikation 31
8.4 Vorüberlegung 31
8.5 Implementierung 32
8.5.1 Vorbereitung der Images 32
8.5.2 Komponenten der CI/CD-Pipeline 35

8.5.3 Die CI/CD-Pipeline ... 36
8.6 Prüfung der Implementierung ... 37
8.7 Verbesserung der Implementierung ... 39

9 Zusammenfassung und Ausblick ... 41
9.1 Bewertung der vorgestellten Lösung ... 41
9.2 Verbesserungen in der Zukunft ... 42

10 Literaturverzeichnis ... 44

11 Bilderverzeichnis ... 46

12 Tabellenverzeichnis ... 47

13 Verzeichnis der Abkürzungen ... 48

14 Selbstständigkeitserklärung ... 49

1 Einleitung

Google, Amazon, Microsoft, VMware, all diese Unternehmen bieten Plattformen, für Unternehmen, als Antwort auf eine Variation der folgenden Frage: „Wie kann ich mein Produkt, so effizient und schnell wie möglich ausliefern?“. Diese Plattformen versuchen den Software Delivery Prozess so effizient und zuverlässig wie möglich zu gestalten.

In Kombination mit weiteren modernen Technologien wie Docker-Containern kann nicht nur der Software Delivery Prozess vollständig automatisiert werden. Es ermöglicht auch flexible Architekturen, skalierbare Performance und hohe Ausfallsicherheit. Während der Entwicklung lassen sich einfach Infrastrukturkomponenten nachmodellieren und lokal deployen.

In dieser Arbeit werden verschiedene Systeme und deren Möglichkeiten, zum Einsatz im Datenverarbeitungszentrum Schwerin für n-Tier E-Government-Anwendungen untersucht. Ziel der Arbeit ist es, eine auf Docker-Containern basierende CI/CD-Pipeline für mindestens ein Projekt zu implementieren, das hierfür notwendige Grundlagenwissen zu schaffen und die getroffenen Entscheidungen zu erläutern und zu dokumentieren. Zusätzlich wird das Ersetzen von bestehenden Services durch Docker-basierte Lösungen untersucht.

2 Grundlagen

Im folgenden Kapitel werden die Grundlagen der Technologien, auf denen diese Arbeit basiert, erläutert. Continuous-Integration und Continuous-Delivery sind komplexe Themen, zu deren Umsetzung meist mehrere Technologien notwendig sind. Dieses Kapitel soll dazu dienen das nötige Vorwissen zu schaffen und Begriffe zu definieren.

2.1 Die Continuous-Integration- / Continuous-Delivery-Pipeline

Die Vorteile einer Continuous-Integration / Continuous-Delivery (CI/CD) Pipeline werden bei einer agilen Arbeitsweise besonders deutlich. Denn „Continuous", also kontinuierliches, Testen funktioniert nur wenn es testbare Artefakte gibt. Die Tests sollten so früh wie möglich beginnen und bei einer agilen Arbeitsweise entstehen testbare Artefakte deutlich früher als beim Wasserfall- oder V-Modell. Im Folgenden wird daher davon ausgegangen, das mit SCRUM gearbeitet wird.

Die CI/CD-Pipeline enthält alle Schritte der Auslieferung eines Produktes. Sie beginnt mit der Sprintplanung und endet mit der Bereitstellung, von auslieferbaren, Artefakten. Diese Schrittfolge wird als Pipeline bezeichnet. Die CI/CD-Pipeline sollte mit jedem Sprint mindestens einmal komplett durchlaufen werden. Im Folgenden werden die üblichen Stadien dieser Pipeline erläutert.

2.1.1 Die einfachste Pipeline

Einer der einfachsten Softwareentwicklungsprozesse ist der Folgende. Ein Programmierer arbeitet in bestehendem Source Code, testet seine Änderungen lokal und führt einen Commit dieser Änderungen, in das Source-Code-Management-System (SCM), aus. Für viele Hobbyisten und bei kleinen Projekten endet der Workflow hier.

Es kann keine Aussage über die Zuverlässigkeit, der entstehenden Software, gemacht werden. Daher ist dieser Workflow nicht für den Einsatz in einem professionellen Umfeld geeignet.

2.1.2 Die Continuous-Integration-Pipeline

In einem professionellen Umfeld oder bei größeren Open-Source Projekten ist dies nur der Anfang. Nach dem Commit beginnt die sogenannte Continuous-Integration-Pipeline (CI-Pipeline). Die Software durchläuft in einem standardisierten Umfeld mindestens die Stationen Build, Unittests und Integrationstests. Jede dieser Stationen, oft auch Stages

genannt, kann wiederum aus kleineren sogenannten Schritten (Steps) bestehen. Die hier entstehenden Artefakte können in ein Artefakt Repository geladen werden und sollten damit bereit für den Einsatz beim Kunden sein. Als Artefakt wird das Ergebnis der Pipeline bezeichnet, dies sind im Java-Umfeld oft .WAR- oder .JAR-Dateien.

2.1.3 Die Continuous-Delivery und Deployment Pipeline

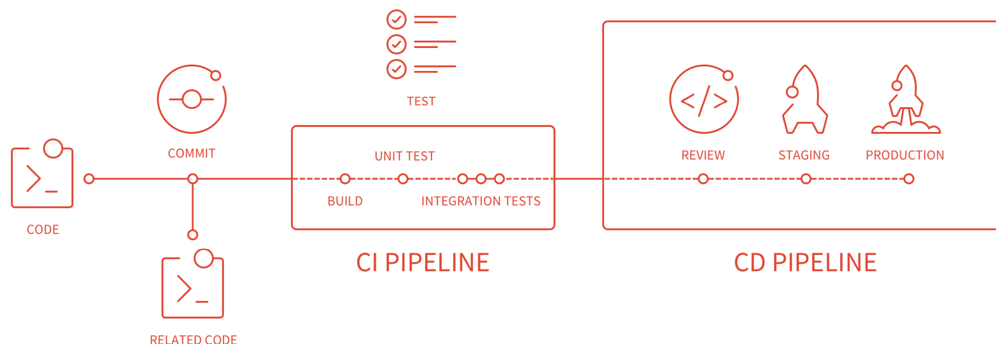


Bild 1: GitLab CI/CD Pipeline von [1]

Der CI Pipeline folgt die CD Pipeline. CD kann sowohl für Continuous-Delivery als auch für Continuous-Deployment stehen, im Folgenden ist CD als Continuous-Delivery definiert. Continuous-Delivery ist ein nicht konkret definierter Begriff, bedingt dadurch dass CD ein relativ neues Feld, und selbst die ältere Fachliteratur frühestens 2014 veröffentlicht wurde. Im Rahmen dieser Arbeit wird Continuous-Delivery als Stage nach der Continuous-Integration Stage definiert. Die Continuous-Delivery Stage stellt, eine ständig aktualisierende, semi-persistente, Umgebung für nicht-technische Tests bereit. Continuous-Deployment ergänzt die Delivery Pipeline um das Deployen in der Produktivumgebung, wird aber vorerst nicht weiter untersucht.

Eine abgesehen von der „Production“ Stage vollständige CI/CD ist in Bild 1 zu sehen. Das „Staging“ ermöglicht es Fehler, frühzeitig zu erkennen und zu beheben. Des Weiteren bietet die Staging Area eine permanente Umgebung, mit der neusten getesteten Version der Software, welche demonstriert werden kann und gegen welche weitere Tests durchgeführt werden können. Dies wird in Kapitel **Fehler! Verweisquelle konnte nicht gefunden werden.** weiter ausgeführt.

Im Idealfall besteht die nötige Infrastruktur, um Docker-Container produktiv einzusetzen. Hier wäre eine Anpassung der Umgebung nicht nötig, da die Container und nicht die Artefakte deployt werden. Das bedeutet, dass die Entwicklungsumgebung immer auch der produktiven Umgebung entspricht.

2.2 Docker

„Container sind eine schlanke und portable Möglichkeit, beliebige Anwendungen und ihre Abhängigkeiten zu verpacken und transportabel zu machen“ [2]. Die häufig verwendete Redewendung „Runs on my machine“ ist hiermit nicht mehr gültig. Es gibt keine Abhängigkeiten, die fehlen können oder keine fehlerhafte Konfigurationen. Im Folgenden ist mit Container immer ein Docker-Container gemeint.

2.2.1 Unterscheidung zwischen Docker-Container und Virtuelle Maschine

Es gibt wenige Szenarien, in denen ein Docker Container eine VM nicht ersetzen kann. Zu diesen Szenarien gehören das Deployment von .Net Applikationen oder anderen Applikationen, die proprietäre Frameworks oder Abhängigkeiten benötigen.

Die Verwendung von Docker Containern setzt aber eine andere Arbeitsweise voraus. Mit der Geschwindigkeit in der Docker-Container aufgesetzt, getestet und verändert werden können, ist es nicht mehr nötig, sich wie bei VMs, auf langjährige Fehlersuche zu begeben und das System wieder gesund zu pflegen. Wenn ein Container nicht funktioniert wird er gelöscht, die Konfiguration angepasst und erneut getestet, bei VMs ist dieses Vorgehen zu zeitaufwendig. Der Unterschied in der Mentalität, im Umgang mit Containern und VMs, wird oft als „Pet vs Cattle“ bezeichnet. Aufgrund dieser Eigenschaften eignet sich Docker sehr gut für CI/CD Anliegen [3].

Das schnelle Start-Up von Containern wird ermöglicht durch das AuFS, ein Layered File System. Dieses ermöglicht unter anderem das Nutzen von Ressourcen des Hostsystems und das Teilen von Layern zwischen verschiedenen Containern.

VMs benötigen dedizierte Kernel, Treiber und Applikationen. Dadurch ist jede VM eines Hypervisors von anderen isoliert und verhält sich wie ein dedizierter Server. Diese Trennung bedeutet aber auch, dass für jeden deployten Service zusätzlicher Overhead entsteht.

Aufgrund des AuFS, welches Docker Container nutzten, kann Speicher im read only Modus zwischen Containern geteilt werden. Theoretisch ist es damit möglich hunderte Instanzen, eines Service, auf einem Host zu betreiben und nur unwesentlich mehr Hauptspeicher zu verbrauchen, als eine Instanz es tun würde.

2.2.2 Dockerfile

Die Dockerfile ist die Grundlage für schnelles, iteratives Arbeiten. Die Dockerfile ist menschenlesbar und beschreibt das gesamte Deployment einer Applikation. Sie enthält alle Befehle, die vor dem Starten des Containers ausgeführt werden. Dockerfiles werden üblicherweise in Repositorien gespeichert und versioniert.

Eine Dockerfile kann mehr als 17 verschiedene Instruktionen enthalten. Die wichtigsten hiervon sind: FROM, LABEL, RUN, CMD, EXPOSE, ENV, COPY, ENTRYPOINT, VOLUME, USER.

LABEL ist eine optionale Anweisung. Es gehört aber zu einem guten Stil diese zu verwenden. Hier können Metadaten als Key-Value Paare angegeben werden.

ENTRYPOINT gibt ein Programm an, welches beim Starten des Containers ausgeführt wird. Oft ist dies /bin/bash oder ein Webserver.

RUN und CMD haben eine ähnliche Funktionalität. Beide führen Befehle im Container aus, allerdings dient RUN zur Herstellung der Umgebung und CMD zum Ausführen von Programmen in dieser Umgebung. RUN wird also immer vor CMD ausgeführt.

CMD darf nur einmal, pro Dockerfile, vorkommen. CMD kann entweder Programm und Argumente verarbeiten oder nur Argumente entgegennehmen, welche dem ENTRYPOINT weitergeleitet werden.

EXPOSE dient dem Öffnen von Ports, ENV dem Erstellen von Umgebungsvariablen, COPY dem Kopieren von Dateien aus dem Build-Kontext in den Container und USER dem Festlegen von Container-Rechten.

Mit VOLUME können Verzeichnisse des Hosts dem Container bereitgestellt werden, so können Daten einfach persistiert werden, oder z. B. Web Apps schneller entwickelt werden, indem HTML Dateien zur Laufzeit des Containers geändert werden und nicht mit einem Neustart per COPY oder SCM (Git) geladen werden müssen.

Jede Dockerfile muss immer mit einem „FROM“ beginnen, hier wird das Base Image angegeben. Danach können die Anweisung in beliebiger Reihenfolge vorkommen und werden in dieser ausgeführt. Jedes Docker Image kann als ein Base Image verwendet werden. Der Docker Hub ist ein öffentliches Registry bei dem verschiedenen Image, für spezielle Anwendungsfälle, bereits vordefiniert sind und nur noch angepasst werden müssen[2].

So gibt es beispielsweise ein Apache-HTTP-Server Image, das einen simplen HTML Web Server bereitstellt. Dieser kann mit folgendem Befehl gestartet werden:

```
docker run -d --name my-apache-app -p 80:80 \
          -v ./:/usr/local/apache2/htdocs/ httpd:2.4
```

Mit diesem Befehl wird ein Container mit dem Image „httpd“ mit dem Tag „2.4“ im detached-Modus gestartet, bekommt den Namen „my-apache-app“, stellt den Port 80 bereit und vorbereitete HTML Dateien, des aktuellen Ordners, werden dem Container zur Verfügung gestellt. Unter http://localhost erscheint nach dem Start die eingebundene Website.

2.2.3 Docker Image

Mit dem „docker build“ Befehl kann aus einer Dockerfile ein ausführbares Image gebaut werden. Dabei muss der „build context“ übergeben werden. Der „build context“ enthält alle Dateien, die von einem Container benötigt werden. Es empfiehlt sich, einen Ordner, pro Dockerfile, anzulegen und in diesem auch den „build context“ abzulegen.

Eine Registry ist ein Image Repository, in dem Images gespeichert und versioniert werden. Wie in Kapitel 2.2.2 bereits erwähnt gibt es eine offizielle Registry den Docker Hub aus dem Images heruntergeladen werden können. Mit „docker pull *Nutzer/Image:tag*“ wird das angeforderte Image, mit dem angegebenen „tag“, in ein lokales Registry geladen. Dies verhindert, dass jedes Image pro Build erneut heruntergeladen wird. Ist ein, in der Dockerfile angegebenes, Image noch nicht in der lokalen Registry vorhanden so wird es zu Beginn des Build Prozesses heruntergeladen. Im Enterprise Umfeld kommen, aus Sicherheitsgründen, meist „self hosted“ Registries zum Einsatz. Diese können entweder als Default oder nach folgendem Muster beim Pull angegeben werden „Docker pull *my.registry.address:port/name*“.

Docker fordert ein TLS Zertifikat für die angegebene Registry, solange nicht die „—insecure-registry“ Flag gesetzt wurde. Self-Signed-Certificates reichen aus, wenn die entsprechenden Zertifikate im Host hinterlegt wurden. Um ungesicherte Registries nutzen zu können, muss die Domain der entsprechenden Registry in der /etc/docker.config eingetragen werden. In dieser Datei sind alle nutzbaren, nutzbaren, aber ungesicherten und gesperrten Registries eingetragen. Beim Entwickeln mit Docker kann es Sinn ergeben, hier alle Registries, mit Ausnahme des eigenen (On-Premise), zu sperren. Das stellt sicher das nur geprüfte Images verwendet werden können. Die genutzten Registries sollten auch im eignen Netzwerk HTTPS verwenden, dies macht das Arbeiten sowohl einfacher als auch sicherer. Alle Images in der On-Premise

Registry müssen regelmäßig auf Sicherheit und Aktualität geprüft und gegebenenfalls aktualisiert werden, um Fehlern, in Produktion und Entwicklung, vorzubeugen und Sicherheitsrisiken zu minimieren.

Mit „docker run [OPTIONS] *image_name:tag* –t *container_name*“ wird aus einem image ein Container gebaut und benannt.

Mit „docker start [OPTIONS] *container_name*“ können bereits erstellte Container ausgeführt werden. Beide Befehle unterstützen verschiedene Flags, um z. B. einen Container interaktiv oder „detached“ zu starten, um Ports umzuleiten oder vieles anderes, was hier den inhaltlichen Rahmen übersteigen würde.

Wie bereits in vorangegangenen Kapiteln erwähnt ist ein Docker Image aus Schichten, auch Layer genannt, aufgebaut. Jede Schicht des Images ist Read-Only und somit unveränderlich. Ein Layer wird auch „intermediate“ Container genannt. Diese temporären Container werden pro ausgeführten Befehl währen des Build Prozesses erzeugt. Dies ist auch in der Konsolenausgabe beim Durchführen des Builds zu sehen. Jedes Image, mit Ausnahme des Basis Images, referenziert das vorangegangene, sogenannte Parent Image. Mit dem Ausführen eines Images, in anderen Worten dem Erzeugen eines Containers aus einem Image, wird das sogenannte Container Layer als Letze Schicht hinzugefügt. Anders als die restlichen Schichten ist das Container Layer Read-And-Write fähig [12].

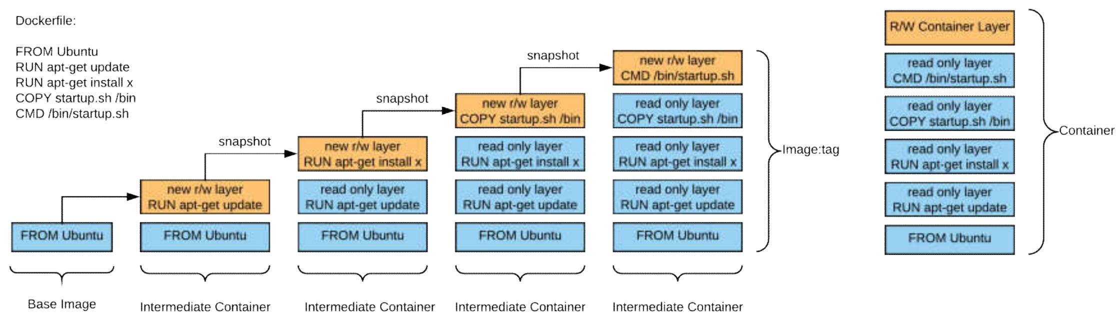


Bild 2: Docker-Container Layer Filesystem

Weil Änderungen nur im Container Layer stattfinden kann ein Image von mehreren Containern gleichzeitig genutzt werden. Dabei werden nicht mehrere Instanzen des gesamten Images erzeugt, sondern jeweils nur ein weiteres Container Layer. Mehrere Container mit Webservices basierend auf demselben Base Image sind somit deutlich effizienter als, dieselbe Anzahl von, dedizierten VMs mit denselben Webservices [12].

2.3 Docker Orchestrierung

Wie bereits in der Einleitung erwähnt werden Mircroservice-Architekturen stetig relevanter. Container kommen hier oft zum Einsatz, weil das Setup unkompliziert und schnell ist und dadurch häufiges updaten der einzelnen Services ermöglicht. Zusätzlich kann komplexe Infrastruktur schnell nachgebaut werden.

Der in vorangegangenen Kapiteln beschriebene Ablauf ist hierfür aber zu umständlich und aufwendig, weil es zu viele manuelle Schritte pro Container erfordert. Zur Automatisierung dieser Schritte gibt es Lösungen wie Docker Compose, Swarm oder Kubernetes. Die vorgestellten Lösungen konkurrieren nicht, sondern sind für verschiedene Szenarien gedacht. Grundlegend sorgt die Orchestrierung dafür, dass alle Teile eines Systems zusammenarbeiten und kommunizieren können [2].

2.3.1 Docker Compose

Compose unterscheidet sich von den anderen Orchestrierungsmethoden darin, dass alle verwalteten Container zwingend auf demselben Host ausgeführt werden. Es ist die einfachste Möglichkeit mehrere Container gleichzeitig zu verwalten.

Ein Docker Compose File enthält die Definition mehrerer Services und wird im YAML Syntax geschrieben. Sie beginnt mit der Version der verwendeten Syntax, aktuell ist zurzeit „version: ‘3‘ “, darauf folgt immer „services:“. Anschließend werden die gewünschten Container beschrieben.

Es können Build Vorschriften für Dockerfiles festgelegt werden oder ein vorbereitetes Image verwendet werden. In [5] sind beide Varianten aufgezeigt. „Product-Service“ stellt eine REST API zur Verfügung und liefert über diese ein JSON Array. Da „website“ dieses Array verwendet wird der „depends_on“ Befehl verwendet. Der Container „website“ liefert jetzt eine PHP Seite auf der eine Liste mit Produkten ausgegeben wird. Für eine simple portable Website, die eine Datenbank oder eine REST API benötigt, ist dies meist ausreichend. Wenn aber vertikales Skalieren nicht mehr ausreicht und auch horizontal skaliert werden muss, oder Features wie automatischen Load-Balancing, gewollt sind, wird eine komplexere Lösung benötigt [2].

2.3.2 Kubernetes

Kubernetes ist ein von Google angestoßenes Open Source Projekt zum Verwalten von „containerisierten“ Applikationen auf verteilten Hosts und bietet dabei Mechanismen zum Deployen, Warten und Skalieren von Applikationen[4]. Die grundlegende Architektur ist in Bild 3 dargestellt. Es gibt ein Master Node an dem sich Worker Nodes

registrieren. Der Master Node ist zuständig für die Verwaltung des Kubernetes Cluster, die Verteilung der Container auf die Worker Nodes und die Verwaltung dieser.

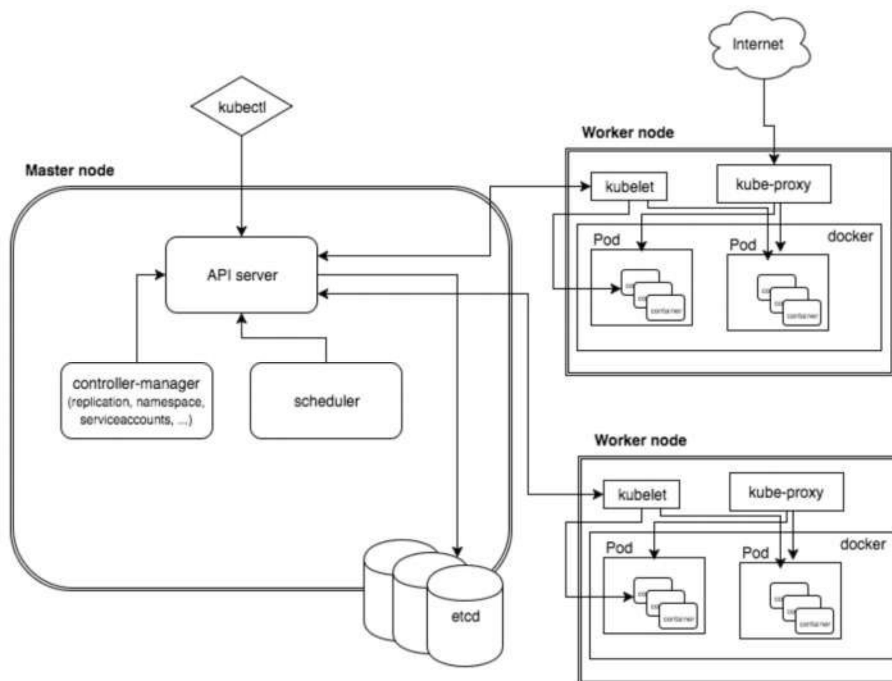


Bild 4: "High level Kubernetes architecture showing a cluster with a master and two worker aus [6]

Der Master Node selbst besteht aus 4 Microservices, dem API Server, etcd storage, dem Scheduler und dem controller-manager.

Der API Server ist die Schnittstelle nach außen und stellt eine REST API zur Verfügung. Über diese API wird der Cluster administriert.

Etcd ist ein simpler, verteilter, konsistenter Key-Value Store. Die Hauptaufgabe besteht darin die Konfigurationen und Service Discovery zu speichern und über eine REST API dem Cluster bereit zu stellen.

Das Deployment konfigurierter Pods und Services auf die Pods wird vom Scheduler vorgenommen. Dem Scheduler stehen Informationen über Ressourcenverfügbarkeit und Anforderungen der Pods und Services zur Verfügung. Auf Basis dessen erfolgt das Deployment.

Zusätzlich zu den bereits erwähnten Microservices können weitere Controller im Master Node ausgeführt werden, der Controller-Manager dient der Verwaltung dieser. Ein

Beispiel für einen weiteren Controller wäre ein Replication Controller, welcher dafür sorgt das immer eine bestimmte Menge an Pods parallel laufen.

Auf den Worker Nodes laufen die Container, daher haben sie auch alle nötigen Services zum Verwalten der Kommunikation der Container unter einander und mit dem / den Master Node/s und den Ressourcen, die den Containern bereitgestellt werden. Die hierfür notwendigen Services sind kubelet, kube-proxy und kubectl.

Kubelet kommuniziert mit dem Master Node, bezieht die Konfigurationen und sorgt dafür, dass alle Container in der richtigen Anzahl und mit der richtigen Konfiguration ausgeführt werden.

Kube-proxy ist ein Netzwerk Proxy, agiert zusätzlich als Load Balancer und routet TCP und UDP Pakete.

Kubectl ist das Comand-Line-Tool, mit dem Befehle an den API Server des Master Nodes gesendet werden und welches vom Kubernetes Dashboard verwendet wird [2].

Zentrale Konzepte, für die Benutzung, von Kubernetes sind Pods, Services, Volumes und Deployments.

Ein Pod ist die kleinste von Kubernetes verwaltbare Einheit. Ein Pod kann mehrere Container enthalten. Ein Pod kann über die Deployment-Configuration beschrieben und konfiguriert werden. Hier können verwendete Images, Ressourcenlimitationen, die Art des Deployment, Anzahl der Repliken, Namespaces, Volume-Mounts und Hooks angegeben werden. Innerhalb eines Pods können Container frei miteinander kommunizieren. Für die Kommunikation von Containern verschiedener Pods werden Services empfohlen.

Services, dienen als Load-Balancer und ermöglichen es einen Pod von außerhalb des Clusters anzusprechen

Volumes dienen der Zuweisung von persistentem Speicher zu Containern. Nach Erstellung können sie in Container gemountet werden. Ein Container kann mehrere Volumes nutzen und ein Volume kann von mehreren Containern genutzt werden. Den Vorgang der Zuweisung wird als „claim“ bezeichnet [12].

3 Continuous-Integration

Continuous-Integration ist heutzutage selbst in jedem Start-Up ein Muss. Es ist eine Grundlegende Voraussetzung, um zuverlässige und getestete Software entwickeln zu können. Das Durchführen, Protokollieren und Tracken von Unit und Integrationstests ist manuell schwer möglich, daher gibt es Produkte wie Jenkins oder GitLab CI. Beide sind Lösungen für den professionellen Einsatz, Jenkins ist aber schon deutlich länger am Markt und hat einen höheren Marktanteil als konkurrierende Produkte.

Jenkins wird im DVZ Schwerin produktiv eingesetzt und wird daher im Folgenden betrachtet.

3.1 Jenkins und Jenkins Pipeline

Jenkins ist ein Java-basierter Automatisierungsserver, der zur Automatisierung von Builds, Tests und Deployments genutzt werden kann. Jenkins kann aus einem einzigen Master und Slave auf einer VM bestehen, in einem Docker-Container laufen, oder über einen Cluster skaliert werden. Durch die Vielzahl von angebotenen Plug-Ins kann die Funktionalität von Jenkins an die eigenen Bedürfnisse angepasst werden. So ermöglicht Jenkins es sogar, Tests in verschiedenen Clouds oder auf selbst gehosteten Clustern auszuführen.

Es gibt verschiedene Möglichkeiten um Tests in Jenkins zu automatisieren. Die für diese Arbeit relevante, ist die „Pipeline“ Methode. Pipeline ist für komplexe Jobs mit langer Laufzeit gedacht, die sich über verschiedene Slaves erstrecken können. Pipeline Definitionen können im Webinterface vorgenommen werden, es ist aber zu empfehlen eine Jenkinsfile in einem SCM abzulegen. Auf diese Art steht selbst die Pipeline unter Versionskontrolle. Eine Pipeline ist eine von mehreren Möglichkeiten einen Job auszuführen. Ein Beispiel für eine simple Pipeline ist in im Folgenden zu sehen [11].

```
1 pipeline {
2     agent any
3     stages {
4           stage (‘Test‘) {
5                 steps{
6                       sh ‘‘‘
7                       git clone http://10.4.48.203/t/zalenium_demo.git
8                       cd ./zalenium_demo/
9                       mvn package‘‘‘
10                }
11          }
12    }
13 }
```

Die Idee der Continuous-Integration basiert darauf, dass eine Test Suite in regelmäßigen Abständen in einer definierten und unveränderlichen Umgebung durchgeführt wird. Dies ermöglicht es, eine Aussage über die Korrektheit der Implementierung zu treffen. Die auszeichnende Eigenschaft von Docker ist das die Laufzeitumgebung von Applikationen immer wohldefiniert und unveränderlichen ist. Es ist daher sinnvoll Docker-Container in der CI Pipeline einzusetzen.

Jenkins bietet die Möglichkeit Tests innerhalb eines Containers auszuführen. Hierfür muss innerhalb der Pipeline-Definition Docker als Agent angegeben werden. Das verwendete Docker Image muss über die nötigen Fähigkeiten, der jeweiligen Tests, verfügen. Soll eine Python 3 Applikation ausgeführt werden muss Python 3 auch innerhalb des Containers installiert sein. Wenn Integrationstests gegen eine Applikation auf dem Jenkins Server durchgeführt werden sollen kann nicht wie vorher „localhost“ in der URL angegeben werden. Es muss die IP-Adresse des Hosts (Jenkins Server) angegeben werden, da sich „localhost“ innerhalb des Containers nur auf den Container bezieht. Abgesehen hiervon benötigen Tests meist keine Anpassungen.

3.2 Zalenium

Selenium ist ein Framework zum automatisierten Testen von Webapplikationen. Um diese Tests zu parallelisieren kann ein Selenium Grid zum Einsatz kommen. Selenium Grid verwendet eine Master / Slave Architektur. Die Nodes werden üblicherweise auf VMs deployt und stellen dort eine Reihe von Browsern und Browserinstanzen bereit. Der Hub (Master) verteilt die auszuführenden Tests als JSON per HTTP an die Nodes (Slaves). Die Kommunikation mit dem Hub erfolgt ebenfalls per HTTP.

Ein häufiges Problem ist das Einfrieren der Browser, eines Nodes, das nur über einen manuellen Neustart behoben werden kann. Dies sorgt für Zeitverlust und Kosten. Zalenium wurde von Zalando entwickelt, um dieses Problem zu beheben. Die Lösung heißt Zalenium und wurde unter der Apache 2.0 Lizenz freigegeben. Zalenium ermöglicht es, die benötigten Nodes bei Bedarf zu erstellen und nach ausführen des Tests automatisch zu zerstören. Dies wird als „dynamisches Grid mit Selbstheilungsfähigkeit“ bezeichnet und macht manuelle Neustarts unnötig.

Das wird möglich indem Hub und Nodes jeweils in Docker-Container laufen. Dies ergibt bei den Nodes besonderen Sinn, da diese ständig erzeugt und zerstört werden und deshalb vom geschwindigkeitsgewinn durch Docker-Container profitieren. Beim Hub sind diese Vorteile weniger gewichtig, aber es ermöglicht die sehr einfache Handhabung des Grids. Das Grid kann wie folgt gestartet werden:

```
docker run --rm -ti --name zalenium \
-p 4444:4444 -p 80:80 -p 443:443 \
-v /var/run/docker.sock:/var/run/docker.sock \
-v /tmp/videos:/home/seluser/videos \
-v /home/patrice/docker_test/resolv.conf:/tmp/node/etc/resolv.conf\
--privileged dosel/zalenium start
```

Die verwendeten Flags wurden bereits in Kapitel 2.2.2 besprochen, mit Ausnahme des --priveleged. Dieses Tag ermöglicht es dem gestarteten Container weitere Container, auf dem Host zu starten und das nicht in einer „Container in Container" Konfiguration, sondern „Container neben Container". Dazu ist es nötig dem Container den Docker-Daemon des Hosts bereitzustellen, was über den ersten Volume Mount geschieht. Des Weiteren muss auch eine Namensauflösung innerhalb des Containers stattfinden, dazu nötig ist in diesem Fall eine angepasste resolv.conf notwenig, die in den Container eingehängt werden muss. Da Zalenium ein Feature bietet um vergangene Testdurchläufe anzuzeigen, der Speicher eines Containers aber nicht persistent ist, müssen die Videos mit dem 2. Volume-Mount im Host gespeichert werden. Nachdem das Grid gestartet wurde hat der Nutzer Zugriff auf die 3 Dashboards aus Tabelle 1.

URL Endung	Bereitgestellt von	Beschreibung
:4444/grid/admin/live	Selenium	Zeigt einen live video feed jedes laufenden Containers, welche lokal ausgeführt werden.
:4444/grid/console	Selenium	Zeigt eine Übersicht aller registrierter Nodes, mit Konfigurationsdetails
:4444/dashboard	Zalenium	Zeigt die Videos und Logs aller vergangener Tests

Tabelle 1: Übersicht der Selenium/Zalenium Dashboards

Wie in Bild 5 zu sehen ist, ist Zalenium eine Erweiterung für Selenium und kein ein Ersatz. Wenn der Hub mit einem Test beauftragt wird, spricht dieser einen Docker Client an, der den Docker Daemon des Host-Systems anspricht. Dieser startet daraufhin die gewünschten Docker-Container, die sich automatisch beim Hub anmelden. Der Hub beauftragt daraufhin die gestarteten Nodes mit den Tests. Den Ablauf dieser Tests zeichnet Zalenium auf und stellt die Aufzeichnungen unter „/dasbboard" bereit.

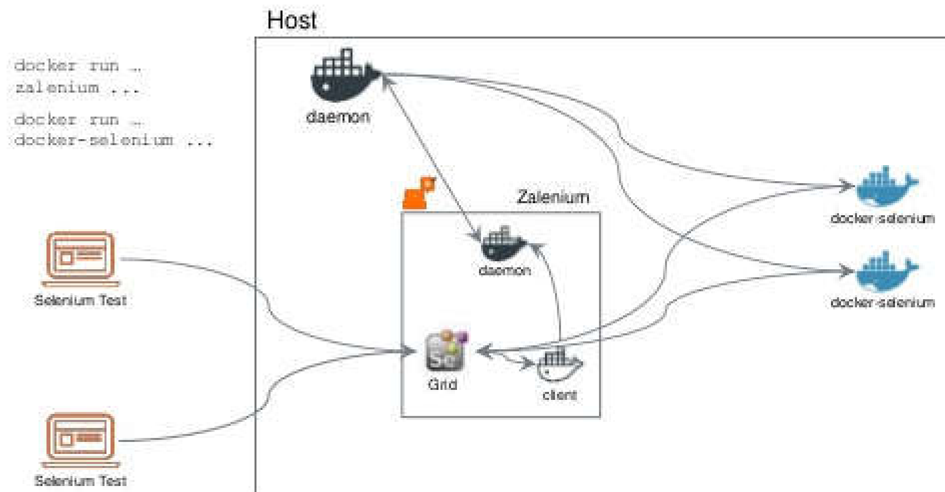


Bild 5: Aufbau eines Zalenium Grids von [7]

Diese Architektur ermöglicht es, das bestehende Selenium Grids ohne Probleme sofort von Zalenium Grids ersetzt werden können und alle bestehenden Tests die vorher das Selenium Grid genutzt haben werden auch mit dem Zalenium Grid fehlerfrei funktionieren.

Im DVZ wurde diese Umstellung bereits durchgeführt, indem die, VMs auf denen Hub und Nodes liefen, abgeschaltet wurden. Die IP Adresse des alten Hubs wurde einer neuen VM zugewiesen, auf der das Zalenium Hub bereits deployt war. Bestehende Windows Nodes wurden am neuen Hub angemeldet. Dies sorgte dafür, dass bestehende Tests nicht angepasst werden mussten. Die Entwickler waren sehr erfreut darüber, die nächtlichen Tests jetzt nicht nur anhand von Logs sondern auch mit Videos nachvollziehen zu können.

3.2.1 Beispiel

Aufgrund der bisher aufgeführten Vorteile wurde für den Ersatzt des bestehenden Selenium Grids durch Zalenium entschieden. Im folgenden Kapitel ist die Verwendung des Zalenium Grids beschrieben.

Nachdem eine VM konfiguriert und Zalenium gestartet wurde kann das Zalenium Grid entweder direkt aus der IDE eines Entwicklers oder von einem CI Server wie Jenkins genutzt werden.

Die Erstellung des Testcodes kann über ein Plug-In für Google Chrome oder Firefox erfolgen und wurde für das folgende Beispiel auch getan. Es werden 2 Browser mit unterschiedlichen Eigenschaften definiert, von denen einer zufällig für diesen Test ausgewählt wird.

```
@Before
public void setUp() throws Exception {
      DesiredCapabilities capabilities;
      capabilities = (DesiredCapabilities)DesiredCapabilities.chrome();
      capabilities.setVersion("69.0.3497.81");
      int randomNum = ThreadLocalRandom.current().nextInt(0, 2);
      if(randomNum == 0){
            capabilities = (DesiredCapabilities)
            DesiredCapabilities.firefox();
            capabilities.setVersion("61.0.2");
      }
      driver = new RemoteWebDriver(
      newURL("http://10.4.48.200:4444/wd/hub"), capabilities);
 }
```

Diese Anforderungen werden, zusammen mit der URL des Zalenium Hub, an eine Funktion übergeben, welche einen WebDriver zurückliefert mit dem die Tests ausgeführt werden.

```
@Test
public void testUntitledTestCase() throws Exception {
    driver.get("https://www.w3.org/");
    driver.findElement(By.linkText("Automotive")).click();
    driver.findElement(By.linkText("Participate")).click();
    driver.findElement(By.linkText("Membership")).click();
    driver.findElement(By.linkText("About W3C")).click();
    driver.findElement(By.linkText("Contact")).click();
}
```

Aus Entwicklersicht macht es kaum einen Unterschied ob ein RemoteWebDriver oder ein WebDriver verwendet wird. Es werden die Entwicklerrechner entlastet, sämtliche Protokollierung erfolgt an einem zentralen Punkt und der Testverlauf kann dank der Videos auch visuell nachvollzogen werden. Diese Infrastruktur kann auch von Jenkins genutzt werden.

4 Continuous-Delivery

Wie bereits in Kapitel 2.1.3 wurde Continuous-Delivery als Stage nach der Continuous-Integration Stage, welche eine ständig aktualisierte, semi-persistente Umgebung für nicht technische Test bereitstellt, definiert. Zur Bereitstellung der benötigten Deployment Umgebungen werden 2 Möglichkeiten untersucht: unter Zuhilfenahme von VMware Integrated Containers und OpenShift Origin / OKD. Beides sind Orchestrierungsplattformen, die auch für den produktiven Einsatz gedacht sind und hohen Unternehmensansprüchen, in Bezug auf Überwachung und Sicherheit, gerecht werden. Um den inhaltlichen Rahmen einzuhalten werden nur die wichtigsten Konzepte beider Plattformen erwähnt.

4.1 VMware Integrated Containers (VIC)

VMware bietet VSphere, vormals VMware Infrastructure, was es ermöglicht, auf einem vCenter Cluster VMs zu erstellen und zu verwalten. VMware VIC ist ein Produkt für die VSphere Plattform um Docker-Container auf bestehender VMware Infrastruktur auszuführen. Es besteht aus den 3 Komponenten Admiral, Harbor und VIC Engine. Die Komponenten stehen unter einer Open Source Lizenz und sind kostenfrei, auf bestehender VMware Infrastruktur, nutzbar.

Admiral ist die eigentliche Orchestrierungsplattform, da es die Verwaltung der Container und das Lifecycle Management übernimmt. Des Weiteren ist die Überwachung des Clusters und der Container hiermit möglich, genauso wie das Erstellen und Speichern von HELM Charts für n-Tier Anwendungen.

Harbor ist die integrierte Registry, welche nicht nur Images speichert, sondern sie auch auf eventuelle Sicherheitsrisiken scannt und den Administrator benachrichtigt, wenn ein Risiko erkannt wird.

VIC Engine ist die Container Runtime für VSphere Cluster und ist die Komponente, die es ermöglicht Container auszuführen. Mit Ausnahme der VIC Engine, diedirekt auf dem Cluster installiert werden muss, können die erwähnten Komponenten als Docker-Container auf einem beliebigen Host im Netzwerk deployt werden.

Um Container auszuführen gibt es 2 Möglichkeiten. Entweder es werden „Virtual Container Hosts“ (VCH) verwendet oder „Docker-Container Hosts“ (DCH). VCH sind für den Einsatz in der Produktion gedacht. Sie haben eine langsamere Deployment Geschwindigkeit, bieten aber eine höhere Geschwindigkeit zur Laufzeit und eine Netzwerktrennung der Container eines Hosts. DCH sind für die Entwicklung gedacht.

Sie haben eine deutlich höhere Deployment Geschwindigkeit und sind voll kompatibel mit den Docker Client Tools. Bei Nutzung des DHC erfolgt keine Netzwerktrennung der Container eines Hosts.

VIC stellt Sicherheit als einen der wichtigsten Aspekte dar. So können Container einfach auf andere Hardware verlagert werden, um eine Applikation von anderen zu trennen und besondere Sicherheitsvorgaben einzuhalten. Die Authentifizierung und Autorisierung kann über LDAP oder andere „activ directory“ Dienste erfolgen. Über Role-based access control (RBAC) können Rechte fein granular festgelegt werden. Die Registry „Harbour“ ist zwar nicht VIC exklusiv, arbeitet aber sehr gut damit zusammen und scannt automatisch alle dort befindlichen Images auf bekannte Schwachstellen [9].

Zu den nicht technischen Vorteilen gehören der gute Support durch einen Marktführer, das Weiternutzen bestehender Infrastruktur, geringe Weiterbildungskosten, einfach Mandantenverwaltung und der Fakt das VIC kostenlos und Open Source ist [9].

Trotz dieser Vorteile hat sich VIC bisher nicht gegen Kubernetes und Kubernetes Distributionen durchsetzen können. Nachdem VMware selbst im November 2018 sagte das die Zukunft der Container Orchestrierung Kubernetes ist, ist die Zukunft von VIC unklar [10]. Daher ist VIC, für den Aufbau einer Container-unterstützenden Infrastruktur, ungeeignet.

4.2 OKD

OKD steht für „The Origin community distribution of Kubernetes that powers OpenShift“. Genau wie VIC ist OKD kostenlos und Open Source und wurde daher zum Vergleich herangezogen.

OKD ist das Open Source Äquivalent zur RedHat OpenShift Container Plattform. OKD ist ein Kubernetes Distribution von RedHat / IBM. Kubernetes wurden um einige Konzepte ergänzt, der Fokus liegt stark auf Sicherheit aber auch einfacherer Benutzung.

Zu den neu hinzugefügten Konzepten gehört „Pipeline“. Dies erinnert nicht ohne Grund an Jenkins Pipeline, denn es ermöglicht das Builden und deployen von Containern und Anwendungen, wofür eine Kombination aus Jenkins Build Strategie, Jenkinsfile und Jenkins Plug-In zum Einsatz kommt. Hierdurch wird es möglich aus Jenkins heraus Container zu builden und zu deployen. Dies sorgt für eine sehr starke Integration von Jenkins in den Container basierten Workflow.

Beim Builden und Deployen von Images auf dem OKD Cluster gibt es Fallstricke zu beachten. Unter anderem wurde in Kapitel 2.2.2 der USER Befehl in der Dockerfile erwähnt. Viele Container erfordern standardmäßig Root Berechtigungen, OKD verbietet es aber Container mit Root Rechten auszuführen. Container die Root Rechte verlangen können nicht erfolgreich deployt werden, es sei denn die Sicherheitsrichtlinien des Clusters werden verändert. Dies würde es auch ermöglichen Container aus Containern heraus auf dem Cluster zu starten. Auch das ist nicht erwünscht, da hierfür Zugriff auf den Docker Daemon des Hosts benötigt und so die Ausführung von ungeprüftem Code ermöglicht wird. Der USER Befehl lässt sich einfach überschreiben, erfordert aber das anpassen des Images.

Ein weiterer Unterschied liegt in der Art auf die Images deployt und konfiguriert werden. Kubernetes verwendet die beliebten HELM Charts, OpenShift kann HELM Charts ausführen, allerdings werden dafür erhöhte Berechtigungen benötigt, was, wie oben beschrieben, als Sicherheitsrisiko von OpenShift verstanden wird. Üblicherweise verwendet OKD Templates.

Templates können mit JSON oder YAML Syntax geschrieben werden und sind eine Aggregation von allen Konfigurationen, die ein Projekt haben kann, einschließlich Build und Deployment Konfiguration, genauso wie Services, Routes und mehr. In Kapitel 8 wird ein Beispiel eines solchen Templates erläutert. Templates ermöglichen es aus der CaaS Plattform, welche OKD eigentlich ist, eine PaaS Plattform zu machen. Mit wenigen Schritten kann der Nutzer / Entwickler so sogar komplexe N-Tier Anwendungen in weniger als einer Minute deployen, vorausgesetzt die verwendeten Images wurden vorher richtig konfiguriert.

CaaS und PaaS beschreiben den Abstraktionsgrad zwischen Entwickler und Hardware, diese Einteilung geht von IaaS über CaaS und PaaS zu SaaS, wobei IaaS nur einer VM und SaaS einer Webapp entsprechen kann.

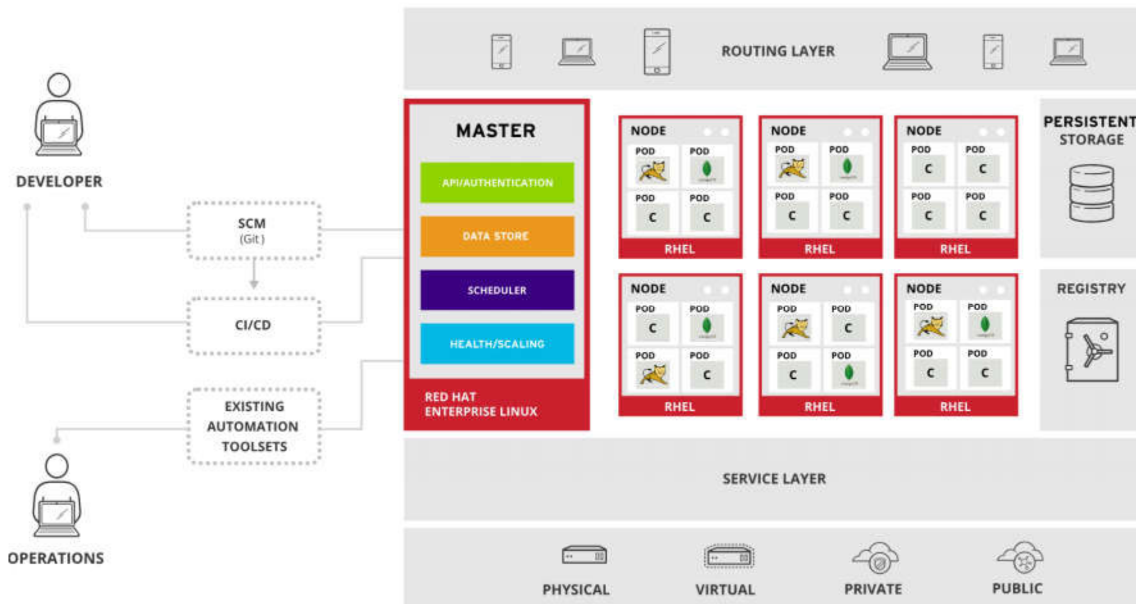


Bild 6: OpenShift Architektur aus [13]

In Bild 6 ist eine Abstrakte Darstellung eines OpenShift Clusters zu sehen. Dargestellt ist ein Cluster auf einer beliebigen Infrastruktur, mit einem Master und 6 Nodes. Service und Routing Layer entsprechen den Services und Routes aus Kapitel 2.3.2 und werden von Infrastruktur Nodes verwaltet. Auf den Nodes sind wiederum je 4 Container in ihren jeweiligen Pods deployt. Dies ist für den produktiven Einsatz auch empfohlen. Für Tests oder temporäre Deployments ist es auch akzeptabel mehrere Container in einem Pod zu deployen.

Services und Routes werden vom Router verwaltet. Der Router ist kein Teil von Kubernetes. Innerhalb von Kubernetes muss ein Router manuell deployt werden, andernfalls erfolgt die Kommunikation mit deployten Containern unverschlüsselt per HTTP. Hierdurch wird auch ein Mechanismus zum automatischen erzeugen von Zertifikaten nötig. Dies entfällt bei OKD, da alle Routen automatisch mit einem Wildcard Zertifikat versehen werden. Wird bei der Installation des Clusters kein Wildcard Zertifikat angegeben, so erzeugt OKD eines.

OKD unterstützt, wie VIC auch, LDAP und bietet ein fein granular einstellbares RBAC. Im Gegensatz zu VIC kann OKD aber auch auf VMs installiert werden. Empfohlen werden, für einen produktiven Einsatz, mindesten Drei. Zur Verwendung in kleinen Teams oder zum Erforschen der Möglichkeiten ist es auch möglich Master, Node und Infrastruktur Node auf einer VM zu installieren.

5 Analyse des bestehenden Workflows

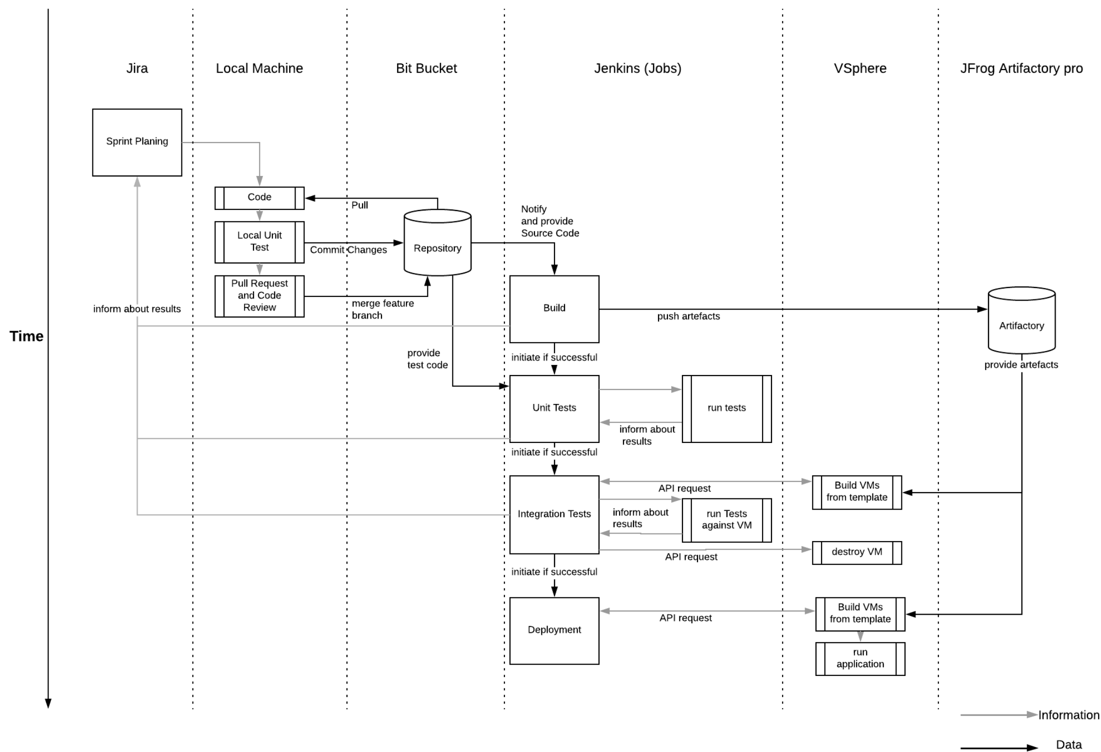


Bild 7: CI/CD Pipeline „Nationales Waffenregister"

Im Gespräch mit Entwicklern und Administratoren wurde der Ist-Stand des Projektes „Nationales Waffenregister" (NWR) wie folgt aufgenommen. Die CI/CD-Pipeline dieses Projektes dient als Muster für zukünftige Projekte.

Der bisherige Workflow läuft wie in Bild 7 gezeigt ab. Im Issue Tracking Tool „Jira" werden die Aufgaben aus der Sprintplanung an die Entwickler vergeben. Diese arbeiten auf Ihren lokalen Maschinen und führen lokal Integrationstests durch. Wenn diese ohne Fehler enden wird eine Pull-Request in „Bit Bucket" erstellt. Nach einem Code Review wird diese entweder abgelehnt oder es wird ein Merge des Feature Branch durchgeführt. Daraufhin wird von einem Jenkins Server ein Build mit dem neuen Code durchgeführt. Die entstandenen Artefakte werden in ein zentrales Artifactory geladen. Jenkins führt in regelmäßigen Abständen Unit- und Integrationstests mit diesen Artefakten durch. Für die Integrationstests erzeugt Jenkins VMs im VSphere Cluster. Die VMs werden aus Templates erstellt und nach Beenden der Tests wieder zerstört.

Beenden diese ohne Fehler, erstellt Jenkins nach demselben Schema die VMs der Staging Area. Diese dienen der Durchführung nichtfunktionaler Tests und eventueller weiterer Integrationstests. Nähere Informationen zu Jenkins gibt es in Kapitel 3.1. Nach erfolgreichem Durchlauf der Pipeline ist die Geschäftsentscheidung, ob der aktuelle

Stand in die Produktion übergeben wird, zu treffen. Dies geschieht über ein Jenkins Plug-In, welches es ermöglicht auf Knopfdruck die aktuellen Artefakte aus der „staging area“ in ein „production Artifactory“ zu „befördern“. Von dort aus stehen sie zum Deployment bereit.

6 Anforderungsspezifikation

Im Folgenden werden die Anforderungen an die CI/CD-Pipeline und Umgebungsbereitstellung formuliert.

6.1 Anforderungen an die CI Pipeline

Die Build- und Unit-Tests wurden bereits entsprechend der geltenden Vorgaben implementiert. Auf die Anforderungen an diese wird hier daher nicht weiter eingegangen.

Ein großer Kritikpunkt an den bisherigen Integrationstests ist die lange Start-Up Zeit der verwendeten VMs und die nichtvorhandene Rückmeldung des Status der VMs. Daher wird bisher mit Timeouts auf Erfahrungsbasis gearbeitet. Dies ist fehleranfällig und umständlich. Es gilt, mit Hilfe von Containern, die Wartezeiten zu verringern und den Status der Container zu kommunizieren.

Zu den Integrationstests gehören UI-Tests, welche von dem in Kapitel 3.2 aufgesetzten Zalenium Grid ausgeführt werden. Daraus ergibt sich die Notwenigkeit von Freigaben zwischen Zalenium Grid und Deployment-Umgebung. Dafür muss eine eindeutige Adressierung der deployten Applikation innerhalb der Umgebung möglich sein.

Die Integrationsumgebung sollte nach Beendigung der Tests zerstört werden, um für jeden Testdurchlauf dieselbe Umgebung zu garantieren.

Die entstehenden Images sollten, entsprechend dem existierenden Muster für das markieren (tagging) von Artefakten, getaggt werden.

Es ist in der Vergangenheit zu Fehlern aufgrund von Unterschieden in der Konfiguration der Entwicklungs- und Produktivumgebung gekommen. Mit Docker ist es möglich, Regressionstest der Deployment-Umgebung durchzuführen. So könnten zu Konfigurationen, die zu Fehlern führen, schnell gefunden werden. Wenn zeitlich möglich, kann ein Mechanismus zum automatischen Testen aller Container eines Tags implementiert werden.

6.2 Anforderungen an die CD Pipeline

Um den gewünschten Automatisierungsgrad zu erreichen, muss der Übergang von Integration Stage zu Delivery Stage automatisiert sein.

Die hier deployte Applikation muss immer dem aktuellsten, erfolgreich getesteten Stand entsprechen.

Die deployte Umgebung sollte eine möglichst hohe Erreichbarkeit und geringe Downtime haben.

Die Tests gegen die deployte Umgebung dürfen erst nach erfolgreichem Deployment der aktuellsten Artefakte beginnen. Dies setzt eine Kommunikation des aktuellen Status der Container in der Deployment Umgebung voraus.

6.3 Anforderungen an die Orchestrierungsplattform

Der amerikanische Patriot Act und ähnliche Gesetze, die Unternehmen in den USA dazu zwingen, Geheimdiensten Zugriff auf ihre Daten zu gewähren, sorgen dafür, dass Cloud basierte Orchestrierungsplattformen nicht von Unternehmen wie dem DVZ genutzt werden können. Dies gilt vor allem für Dienste von Unternehmen mit einem Firmensitz oder Servern in den USA. Welche Plattform auch genutzt werden wird, sollte im eigenen Rechenzentrum deployt werden können.

Fachverfahren (Applikationen) von Polizei und Gerichten erfordern oft den Schutzbedarf „hoch“, teilweise sogar „sehr hoch“. Dieser Schutzbedarf kann auf die 3 Grundwerte Vertraulichkeit, Integrität und Verfügbarkeit bezogen sein. Der Schutzbedarf kann unterschiedlich hoch pro Grundwert sein.[15] Die gewählte Plattform sollte diese vom BSI definierten Werte möglichst gut erfüllen können.

Von Vorteil wäre die Möglichkeit, die Nutzerverwaltung über ein bestehendes Active Directory vorzunehmen. Je detaillierter die Rechtevergabe desto besser.

Im Idealfall kann die gewählte Orchestrierungsplattform auch in der Produktion eingesetzt werden. Um die Kosten für den Betrieb zu verringern, muss der Overhead so gering wie möglich sein.

7 CI/CD mit Jenkins und Docker-Containern

Genauso wie eine Continuous-Integration-Pipeline eine Testumgebung benötigt, benötigt eine Continuous-Delivery-Pipeline eine Umgebung, in der Applikationen längerfristig deployt werden können. Jenkins wird bereits als Testumgebung verwendet, ist aber als Staging Area ungeeignet, da sich Staging Area und Testumgebung sonst gegenseitig beeinflussen könnten. Dies ist nicht gewünscht.

Mögliche Umgebung	Unittest	Integrationstest	Staging geeignet
In OKD	Nein	Ja	Ja
In VM	Nein	Ja	Ja
In Jenkins	Ja	Ja	Nein

Tabelle 2: Überblick über Kombinationsmöglichkeiten von Jenkins und Staging Area

Wie aus Tabelle 2 ersichtlich wird, ist eine Kombination von Tools nötig, was verschiedene Abläufe ermöglicht. Im Folgenden werden zwei Kombinationsmöglichkeiten erläutert

Möglichkeit 1 bestehen darin Jenkins so weit wie möglich zu nutzen. In diesem Szenario werden Unit- und Integrationstests in Jenkins durchgeführt. Als Staging Area dienent eine Orchestrierungsplattform wie OKD oder eine dedizierte VM.

Möglichkeit 2 sieht vor, die Integrationstests gegen ein temporäres Deployment auf einer VM oder Orchestrierungsplattform durchzuführen. Dies hat die Vorteile der besseren Überwachung und das Einsparen von Ressourcen bei den Jenkins Slaves.

Bei Möglichkeit 3 Jenkins in einem OpenShift Cluster ausgeführt. Dies sorgt, dafür dass weniger dedizierte VMs genutzt werden müssen. Jenkins nutzt damit alle Vorteile die Container mit sich bringen. Bei Bedarf können Slaves dynamisch erzeugt und zerstört werden. Das Builden von Applikationen könnte mit Source-to-Image-Containern (S2I-Container) ablaufen, die von Jenkins gestartet werden können und Quellcode direkt aus einem Git Repository beziehen. Somit kann die gesamte CI/CD-Pipeline inklusive Teile der Entwicklung und das Deployment innerhalb des OpenShift Clusters ablaufen. Dies setzt ein sehr gutes Verständnis der OpenShift Plattform, angepasste Workflows und entsprechende Infrastruktur voraus.

Grundsätzlich ist Möglichkeit 1 simpler und plattformunabhängiger. N-Tier Systeme benötigen Docker Compose oder proprietäre Formate wie OKDs Template. Diese sind für das Deployment in die Staging Area ohnehin anzufertigen, allerdings benötigt Jenkins die Rechte, Portfreigaben und Service Accounts, um mit der gewählten Plattform über Plug-Ins zu kommunizieren und um dieses temporäre Deployment vorzunehmen. Dies ist Aufwand, der für das permanente Deployment in die Staging Area nicht nötig ist, da die Schnittstelle zur Staging Area durch das Artifactory gegeben ist. Es sind daher keine API Aufrufe zum hoch- bzw. herunterskalieren der Applikation nötig, die zu diesen Anforderungen führen.

Möglichkeit 2 benötigt zwar mehr Vorbereitung, bietet aber die bereits erwähnten Vorteile der Überwachung und Ressourceneinsparung. Sollten Tests mehr Ressourcen benötigen als geplant, können diese so automatisch zugewiesen werden. Alternativ wird der Administrator informiert, wodurch verhindert werden kann, dass der Test aus unbekannten Gründen abbricht.

Existiert bereits eine Orchestrierungsplattform, so ist Möglichkeit 2 zu bevorzugen, da die nötigen Vorbereitungen vermutlich bereits getroffen wurden und somit kein Nachteil gegenüber Möglichkeit 1 mehr besteht. Bestehen weder CD Pipeline, noch Orchestrierungsplattform, so könnte Möglichkeit 1 angemessen sein, weil mit bereits bestehender Infrastruktur weit fortgeschritten werden kann.

Möglichkeit 3 ist eine Weiterentwicklung von Möglichkeit 2 und kann eine natürliche Weiterentwicklung darstellen. Sie bietet die Vorteile der hohen Integration und einfachen Skalierung, setzt aber Erfahrungen aus Möglichkeit 2 voraus.

8 „Dockerisierung“ von ServO.MV und Erstellen der CI/CD-Pipeline

In den vorangegangenen Kapiteln wurde ein umfangreiches Vorwissen zu Technologien wie Docker Containern und Jenkins geschaffen. Ebenfalls untersucht wurden Möglichkeiten diese zu kombinieren. In dem folgenden Kapitel wird die Anwendung der beschriebenen Technologien auf eine Applikation dokumentiert.

8.1 OpenShift als Orchestrierungsplattform

Im DVZ existiert bisher keine Orchestrierungsplattform. VIC setzt auf einem VSphere Cluster auf, was bedeutet, dass die Abteilung für Betrieb (SC) diesen bereitstellen müsste. RedHat bietet OKD, eine kostenlose und der OpenShift Container Plattform sehr ähnliche, Plattform an. OKD ist Hardware-agnostisch und kann auf einer VM ausgeführt werden. Dies ermöglicht ein unabhängiges und somit schnelleres Vorgehen, weshalb OKD gegenüber VIC vorgezogen wurde. Zusätzlich wird die OpenShift Container Plattform unter anderem vom ITDZ Berlin, einem Rechenzentrum des Landes Berlin, eingesetzt. Dies ermöglicht einen Wissensaustausch.

Der erste Schritt der Implementierung war die Installation eines OKD Clusters. Hierfür wurden die Ansible Playbooks genutzt, die RedHat über GitHub zur Verfügung stellt. Die verwendete VM wird im VMware Ressourcenpool des Sachgebietes GEE gehostet und verfügt über 4 CPUs mit 24 Gb RAM.

8.2 Gewähltes Verfahren

Der gewählte Workflow setzt sich aus den folgenden Schritten zusammen. Der bestehende Jenkins Server bleibt erhalten und führt Unit- und Integrationstests lokal durch. Die Images hierfür werden von JFrog Artifactory Pro bezogen. Nach erfolgreichem Durchlauf der Tests, werden die Artefakte in die Staging Area deployt. Als Staging Area wird OKD genutzt.

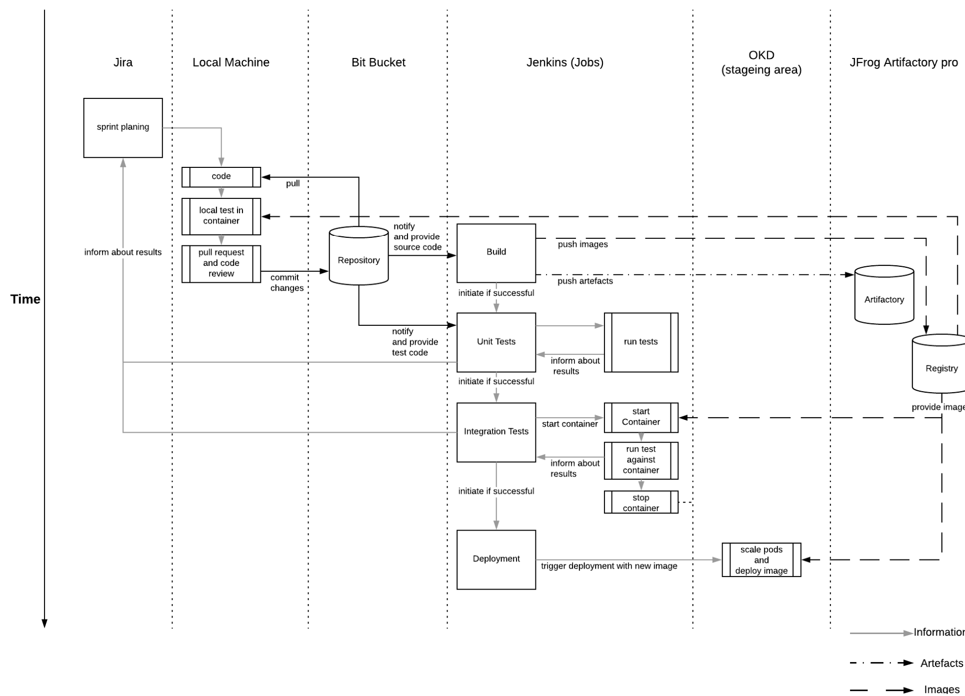


Bild 8: CI/CD Pipeline Entwurf (vereinfacht)

8.3 Gewählte Applikation

Bei der Applikation, deren CI/CD-Pipeline erweitert und „dockerisiert“ werden soll, handelt es sich um „ServO MV“ (Projektname), kurz „ServO“, eine in der Entwicklung befindliche Dienstleistungsplattform des Landes Mecklenburg-Vorpommern. Das Ziel dieser Plattform ist es, dem Bürger Zeit einzusparen, indem Anträge von zu Hause aus gestellt und digital übertragen werden. ServO ist eine Java-basierte Web-Anwendung mit MariaDB als Back-End.

8.4 Vorüberlegung

Um OpenShift als Staging Area nutzen zu können, muss es zuerst aufgesetzt werden und die bestehende Infrastruktur den neuen Bedürfnissen angepasst werden. Hierfür das Upgrade der bestehenden JFrog Artifactory, auf eine JFrog Artifactory Pro, nötig um diese als Docker Registry nutzen zu können.

Da Jenkins die Umgebung bereitstellen wird, gegen die Integrationstests durchgeführt werden, ist eine doppelte Beschreibung des Deployments nötig. Jenkins und OpenShift benötigen jeweils eine Beschreibung. Die Deployment-Beschreibung für Jenkins erfolgt mit Docker Compose, die Beschreibung für OpenShift mit einem Template. Um einen hohen Grad an Kontrolle zu erzielen, werden die Docker Compose File und das OpenShift Template manuell erstellt. Das entstehende Docker Compose File kann für

lokale Tests verwendet werden. Wenn das lokale Testen mit Docker COmpose nicht gewünscht ist, kann OpenShift für die Integrationstests genutzt werden, in diesem Fall entfällt die Erstellung der Docker-Compose File.

Zuerst sollten die Dockerfiles und dass Docker Compose File erstellt werden, um die erfolgreiche „Dockerisierung“ der Applikation zu überprüfen. Die dabei entstandenen Docker Images können anschließend in die bereits existierende JFrog Artifactory Registry, dass Docker-Compose File und das OpenShift Template in das Bit Bucket Repository gepusht werden. Von dort können sie von Jenkins und OpenShift bezogen werden. Nachdem sichergestellt wurde das ServO ordnungsgemäß in den Docker-Containern ausgeführt wird, kann die Jenkins Pipeline erweitert werden.

Die Vorteile Dockers sind nicht nur auf die Applikation anwendbar, sondern auch auf die Testumgebung. Jenkins ist, wie in Kapitel 3.1 besprochen, in der Lage, einen Docker-Container als Testumgebung zu nutzen. Dies setzt ein Docker Image mit den nötigen Fähigkeiten voraus. Da es sich bei ServO um eine aktiv entwickelte Applikation handelt, sollten so schnell wie möglich Tests mit der Jenkins Pipeline möglich sein. Um dies zu erreichen, kann das Testen in einem Docker-Container aufgeschoben werden, bis gezeigt wurde, dass die Pipeline in einer traditionellen Umgebung funktioniert.

8.5 Implementierung

Grundlage für eine CI/CD-Pipeline mit Docker sind die Images. Diese müssen korrekt vorbereitet werden und für N-Tier-Applikationen so konfiguriert werden, dass korrekt kommunizieren. Bevor mit der Implementierung der Pipeline begonnen werden kann, muss dies sichergestellt werden. Die Erstellung der Images geschieht mit Dockerfiles und Docker Compose.

8.5.1 Vorbereitung der Images

Es wurde mit dem Erstellen der Dockerfiles und der Docker-Compose File begonnen. Das ServO-Image basierte auf einem Tomcat 9 Basis-Image, das ein OpenJDK 8 Image als Basis nutzt. Die MariaDB 10.2 Datenbank wird ebenfalls in einem Container bereit gestellt.

Für erleichtertes Debugging wurde ServO vorerst mit der In-Memory Datenbank HSQLDB containerisiert. Daraufhin folgte die Anbindung einer produktiv einsetzbaren Datenbank mit Docker Compose. Sobald dies funktionierte, fand die Einbindung in Jenkins und anschließend das Deployment in OKD statt.

Hierbei gab es Probleme mit Tomcat-Versionen höher als 8.0.20. Grund hierfür war die unerwartete Systemsprache des Docker Containers. Es wurde Deutsch erwartet, was bei Docker Images standardmäßig nicht konfiguriert ist. Nach Änderungen im Quellcode der Applikation wurde die Standardsprache auf Deutsch und nicht die Systemsprache gesetzt, wodurch der Fehler behoben wurde.

Um Unit-Tests durchführen zu können, ohne eine DB zu benötigen, wurde auf HSQLDB als eine In-Memory-Datenbank, gesetzt. Nach den hierfür notwenigen Anpassungen steht ein statisches (nicht aktualisierendes) ServO Image zur Verfügung, das für Unit- und Integrationstests genutzt werden kann.

Nach dem der PoC gelungen ist und das entstandene Image in Jenkins und auf dem OpenShift Cluster deployt wurde, wurde mit dem erstellen der komplexeren Images fortgefahren werden.

Es wurde bereits gezeigt, das ServO in einem Docker-Container deployt werden kann. Im Folgenden geht es darum, den Webserver mit dem Datenbankserver kommunizieren zu lassen. Für Integrationstest, Demonstrationen und weitere Tests ist es nötig, dass die Dastenbank mit Daten gefüllt wird. Hierfür gibt es 2 Möglichkeiten.

Möglichkeit 1, die Datenbank wird gestartet und nachdem der Container bereit ist Anfragen entgegen zu nehmen, wird die Datenbank von einem 2. Container befüllt.

Möglichkeit 2, SQL Dateien können in dem Basis-Container in das Verzeichnis „/docker-entrypoint-initdb.d“ angefügt werden. Die dort befindlichen Dateien werden bei Start des Containers ausgeführt. Dies funktioniert mit dem ADD Befehl. ADD und COPY sind sich funktional ähnlich, mit ihnen können Dateien von einem Ort an einen anderen kopiert werden. ADD bietet unter anderem Funktionen wie das Entpacken von Archiven oder das Herunterladen aus externen Quellen, die COPY nicht bietet. Es wurde Möglichkeit 2 gewählt. Wichtig ist es, nach jedem Durchlauf ein „docker-compose down“ auszuführen, um alle von docker-compose persistierten Daten zu löschen, weil diese sonst zu unerwartetem Verhalten führen können.

Üblicherweise sollte eine Applikation robust genug sein, um eine fehlende Datenbank zu ignorieren und zu warten, bis eine Datenbank bereitgestellt wird. Da dies nicht immer der Fall ist, ist eine Start-Up-Reihenfolge umzusetzen. Die Datenbank mit Testdaten startet zuerst, danach folgt der Tomcat Container. Obwohl dies einer der häufigsten Use-Cases ist, wird er von Docker Compose nicht unterstützt. Ein möglicher Workaround ist der Folgende. Der DB-Container wird mit einem Health-Check versehen, der den Status des Containers ermittelt. Über ein „depends_on“ und „restart: always“ Annotation in der Docker Compose File wird dafür gesorgt, dass der Tomcat Container wartet bis die Datenbank bereit ist Anfragen zu verarbeiten. Da ein hoher

Automatisierungsgrad eine Anforderung an die CI/CD-Pipeline ist, muss das Beziehen der aktuellsten Artefakte automatisch passieren. Zu diesem Zweck wurde ein Shell-Skript angefertigt, welches zuerst feststellt, welche Artefakte die aktuellsten sind, und diese anschließen herunterlädt. Dasselbe gilt für SQL Dateien, die der Datenbank bereitgestellt werden sollen. Beide Schritte werden innerhalb der Docker-Container ausgeführt. Um die Datenbankadresse auflösen zu können, kann eine Eigenschaft von Docker, die Host-Name-Resolution, genutzt werden. Sie agiert wie ein DNS Server für die Container eines Hosts. Alle Container eines Hosts sind in einem Netzwerk. Die angegebene Adresse „url="jdbc:mariadb://db:3306"“ kann aufgelöst werden, da der Container, der die Datenbank enthält, in der Docker-Compose File “db“ genannt wurde.

Nachdem hiermit gezeigt wurde das die Applikation auch mit einer tatsächlichen Datenbankanbindung ausgeführt werden kann, kann mit dem vorbereiten der Images, der CI/CD Pipeline begonnen werden.

Aufgrund der Einschränkungen der deklarativen Pipeline in Jenkins wurde die Pipeline auf zwei Jobs aufgeteilt. Es ist nicht möglich, zwei sequentielle Stages mit verschiedenen Agents in einer Pipeline zu definieren und in der zweiten dieser Stages mehrere parallele Stages auszuführen. Daher sind Build und Tests mit unterschiedlichen Agents nur möglich, wenn alle Stages sequentiell Ablaufen. Dies benötigt deutlich mehr Zeit als zwei Pipelines mit parallelen Stages.

Die erste Pipeline verwendet ein mit Zertifikaten und Maven Konfiguration ausgestattetes Maven 3.5 Image. Als Basis-Image für die Testumgebung dient das offizielle Docker-Image (library/Docker), das auf Alpine Linux basiert. Da dieses Image als Testumgebung dienen soll, müssen noch Open JDK und Maven installiert werden. Durch Dockers Schichtenmodell kann der Inhalt der Dockerfiles für das OpenJDK und das Maven Image übernommen werden. Maven benötigt zwei weitere Dateien, die aus der GitHub Repository bezogen werden können. Da die Pipeline Docker Compose verwendet, muss auch dies noch in dem Image installiert werden. Dies erfolgt mit dem Package-Manager „py-pip“. Wie bei Zalenium in Kapitel 3.2 muss der Docker Socket in den Container gemountet werden, damit weitere Container auf dem Host erstellt werden können. Dies kann nur mit einer Anpassung des „run“ Befehls geschehen und wurde daher in der Agent Definition der Dockerfile umgesetzt.

8.5.2 Komponenten der CI/CD-Pipeline

Die Komponenten der Pipeline sind die Git Repositories, das JFrog Artifactory, die Jenkins Pipelines und OKD. Im Folgenden werden kurz die Rollen dieser Komponenten in der CI/CD-Pipeline reiterriert.

Es werden drei Git Repositories genutzt: eines zur Versionierung der Pipeline selbst und der benötigten Dateien, eines für die eigentliche Applikation und ein weiteres für die Integrationstests. Jedes dieser Repositories kann die Jenkins Pipelines auslösen, somit wird sichergestellt, dass weder die Applikation, noch die Tests oder die Konfiguration zu unerwartetem Verhalten führen.

Die Build Pipeline nutzt ein vorbereitetes und im Artifactory abgelegtes Build Image, welches als Build-Environment dient. Dies ermöglicht es, das Verhalten von Software zu untersuchen, wenn sie mit verschiedenen Maven oder Java Versionen gebaut wird. Die Test Pipeline löst die Abhängigkeiten auf, führt ein „mvn install“ aus, führt Unittests durch und pusht das entstehende Artefakt in das Artifactory. Nach erfolgreichem Durchlaufen der Pipeline wird die Integrationstest-Pipeline angestoßen. Diese Pipeline nutzt ebenfalls ein im Artifactory abgelegtes Image als Testumgebung. Sie besteht aus vier Stages, die jeweils weitere parallel ablaufende Stages enthalten. Die erste Stage dient der Vorbereitung der Umgebung. Es werden die Images der Anwendung gebaut, die Tests werden aus dem Repository abgerufen und Xray wird auf die Testdurchführung vorbereitet. Xray ist ein Jira Plug-In, welches das Testmanagement erleichtert. In der zweiten Stage werden die Tests vorbereitet. Die Container werden gestartet, die Abhängigkeiten der Tests werden aufgelöst und es wird sichergestellt, dass die Services ausgeführt werden bevor die Tests beginnen. In Stage drei werden Cucumber-Tests (Selenium-Tests) parallel mit Chrome und Firefox auf einem Linux- und einem Windows-System ausgeführt. Diese Tests gehen gegen die auf dem Jenkins gestarteten Container. Anschließend werden alle Container beendet, die erstellten Images werden in das Artifactory Registry gepusht und die Testergebnisse werden zu Xray exportiert.

Die getesteten Images können nun aus der Registry auf dem OKD Cluster deployt werden. Die größte Herausforderung und wichtigste Aspekt einer CI/CD-Pipeline ist der Automatisierungsgrad. Der letzte wichtige Aspekt hierbei sind die Schnittstellen zwischen diesen Komponenten, die sicherstellen dass die Übergänge reibungslos erfolgen.

8.5.3 Die CI/CD-Pipeline

Um die in den vorherigen Kapiteln beschriebenen Komponenten zu einer Pipeline zusammen zufügen sind 2 Schnittstellen nötigt. Die erste Schnittstelle zwischen Entwicklern und Jenkins stellt das Repository da. Die Schnittstelle zwischen Jenkins und Staging Area ist die Registry. Ein Commit in das Repository soll einen Jenkins Job (Jenkins Pipeline) auslösen. Nach erfolgreichem Durchlauf der Jenkins Pipelines wird das erstellte Image in eine Registry gepusht. Dadurch verändert sich das Image in der Registry, wodurch ein Deployment des aktuellen Images in der Staging Area getriggert wird.

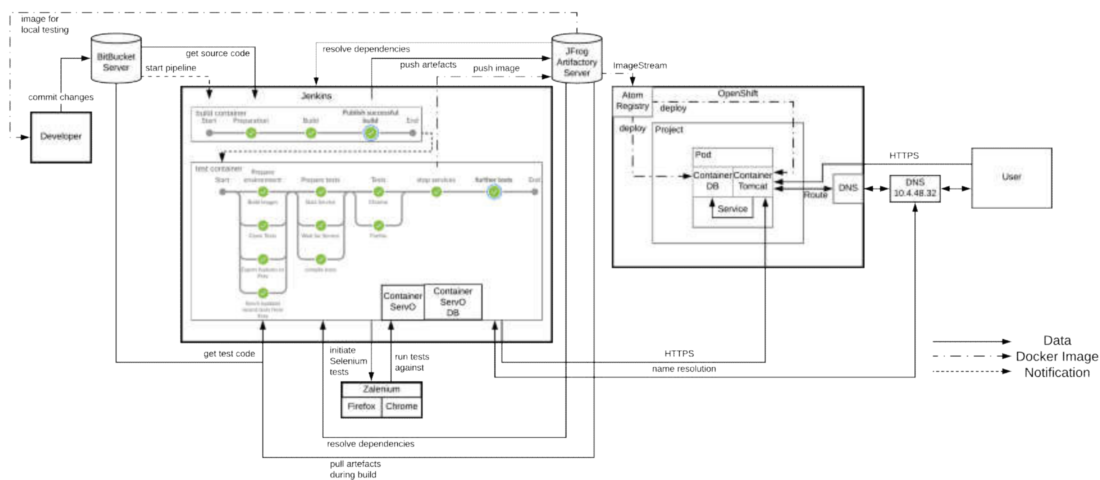


Bild 9: CI/CD Pipeline mit Jenkins und OpenShift Cluster

Damit das Triggern des erneuten Deployments funktioniert, muss das zu deployende Image bereits in der internen Registry vorliegen. Die Interne Registry ist aber nicht die primär genutzte. Die primär genutzte Registry ist eine Artifactory Pro Registry welche diese Trigger nicht unterstützt. Um dieses Problem zu umgehen, kann dem Template ein ImageStream hinzugefügt werden. Dieser importiert ein Image aus einer externen Registry und übernimmt alle gewünschten Tags die zum Zeitpunkt des Imports existieren. Wenn die „--scheduled“ Flag gesetzt wird, wird das externe Image regelmäßig auf Updates geprüft. Standardmäßig erfolgt dieses Update alle 15 Minuten und kann nur für den gesamten Cluster verändert werden. Diese Einstellung kann in folgender Datei angepasst werden: /etc/origin/master/master-config.yaml.

Der empfohlene Weg einen ImageStream zu erzeugen ist der „oc import-image“ Befehl [14]. Für eine ungesicherte Registry mit zu überwachenden Images kann der Befehl wie folgt aussehen:

```
oc import-image Registry:Port/Project/Image --confirm --insecure --scheduled
```

Anschließend kann mit dem “oc get” Befehl eine fertige ImageStream Definition exportiert werden:

```
oc get -o json is > tmp.json
```

Dieser Befehl exportiert alle ImageStreams eines Projektes im JSON Format in eine Datei mit dem Namen „tmp.json“. Aus dieser Datei kann die gewünschte Definition kopiert und als „Object“ in das bestehende Template eingefügt werden.

Nicht zwingend notwendig, aber stabilitätserhöhend, sind die sogenannten „Readiness“ und „Liveness“ Probes. Readiness-Probes überprüfen, ob ein Service nach der Initialisierung bereit ist. Dies kann per HTTP/HTTPS, TCP oder Docker-Commands erfolgen. Liveness-Probes testen die Verfügbarkeit auf dieselbe Art, aber in regelmäßigen Abständen. Sollte eine „Probe“ fehlschlagen, wird der entsprechende Container neu gestartet. Diese Probes entsprechen den bekannten „Health Checks“ aus Docker Compose.

Das Repository mit der Jenkinsfile, kann über Jenkins Blue Ocean verknüpft werden, wodurch automatisch eine Multi Branch Pipeline mit WebHooks erstellt wird. Zusätzlich wurde in den Pipeline-Einstellungen ein Cron Job eingerichtet, der die gesamte Pipeline einmal, abhängig von der Serverauslastung, täglich zwischen ein und zwei Uhr morgens ausführt.

Mit diesen beiden Erweiterungen (dem Setzen der WebHooks und dem Erstellen eines ImageStreams) sind die Komponenten der CI/CD-Pipeline verknüpft. In Bild 9 ist die vollständige Pipeline inklusive des netzwerkinternen DNS Server mit der IP-Adresse 10.4.48.32 dargestellt.

8.6 Prüfung der Implementierung

Es wird angenommen das, wenn die Pipeline erfolgreich durchlaufen werden kann und somit alle Tests erfolgreich durchlaufen worden sind, so ist die Implementierung korrekt.

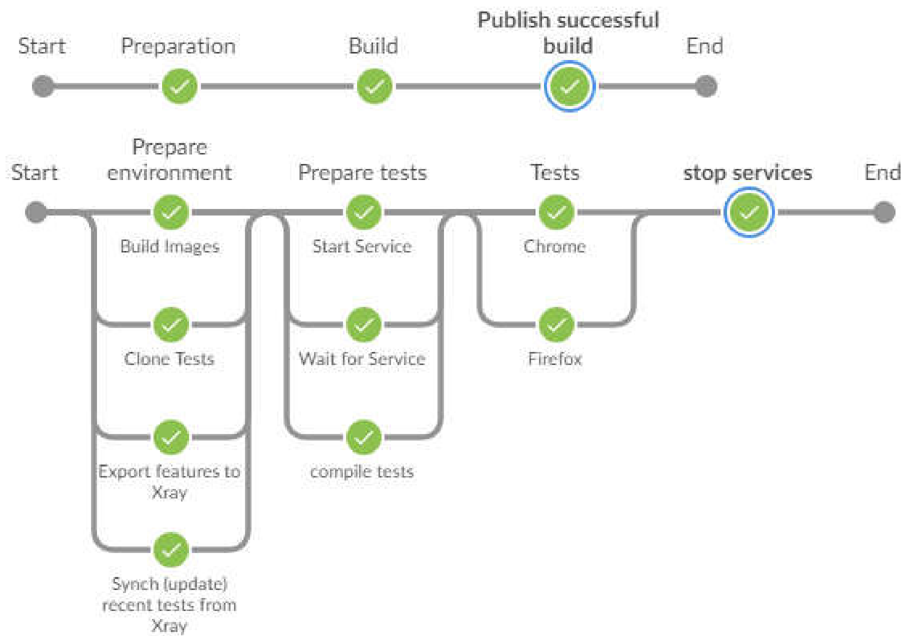


Bild 10: Ausschnitte der Jenkins Pipelines für ServO.MV

Um die Pipeline anzustoßen, wurde ein Commit in den Master Branch des ServO Repository vorgenommen. Dies löste die Pipeline aus.

In Bild 10 sind die beiden Jenkins Pipelines abgebildet. Die grünen Stages sind erfolgreich durchlaufen worden. Unter der Build Pipeline ist ebenfalls zu sehen, dass die Testpipeline erfolgreich angestoßen und durchlaufen worden ist.

Nun gilt es zu prüfen, ob die neuen Images auch auf dem OKD Cluster deployt wurden. Nach der Integration Stage werden die Images in die Artifactory Pro Registry gepusht. Über den im OpenShift Cluster bereits deployten ImageStream werden die, Cluster-interne Registry und die Artifactory Pro Registry synchronisiert.

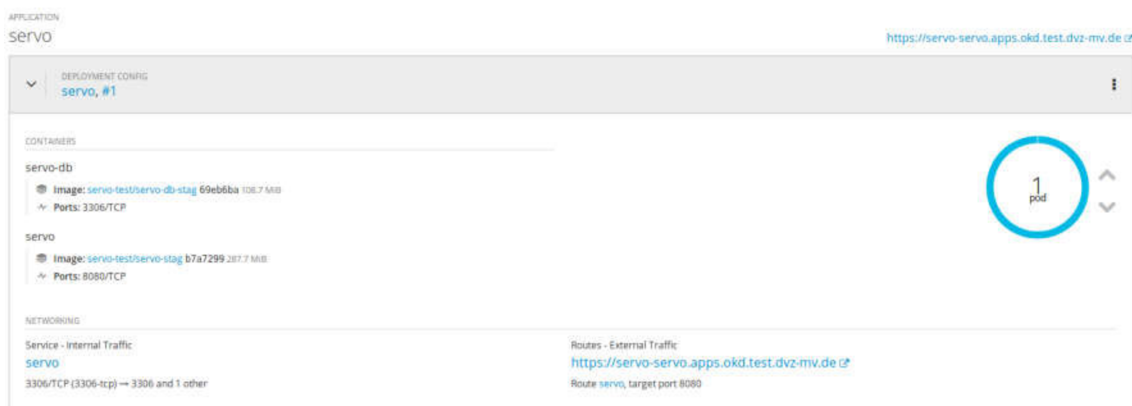


Bild 11: OpenShift Übersicht der deployten Applikation

Die Synchronisation erfolgt alle zehn Sekunden und löst ein „recreate“ Deployment mit den neuen Image aus. Über die Route der Applikation, zu sehen in Bild 11, kann manuell überprüft werden, ob das Image erfolgreich deployt wurde. Dies ist der Fall.

Hiermit wurde gezeigt, dass die Pipeline automatisch nach einem Commit durchlaufen werden kann und die entstanden Images korrekt deployt wurden.

Dieser Arbeit ist eine Animation dieses Ablaufes beigefügt.

8.7 Verbesserung der Implementierung

Die bisher vorgestellte Implementierung erfüllt alle in Kapitel 6 definierten Anforderungen. Das gewünschte Ergebnis wurde schneller als geplant erreicht, daher ist es möglich den vorgestellten Ansatz weiter zu optimieren. Im Verlauf der Umsetzung wurde das Thema „Docker“ im DVZ immer beliebter. Sodas, weitere Sachgebiete Interesse an den in Vorträgen vorgestellten Lösungen anmeldeten. Im Zuge dessen wurde ein weiterer OKD Cluster aufgesetzt, der ausreichende Ressourcen zum Betreiben von Projekten mehrerer Sachgebiete besitzt.

Die doppelte Umgebungsbeschreibung durch Docker Compose und ein OKD Template ist nicht ideal. Docker Compose ist weiterhin eine gute Möglichkeit, um mehrere Container gleichzeitig zu bauen. Das temporäre Deployment der Applikation für Integrationstests wurde auf den OKD Cluster ausgelagert. Dadurch entfällt die Beschreibung der Laufzeitumgebung durch Docker Compose. Ein positiver Nebeneffekt dessen ist die Reduzierung der benötigten Ressourcen pro Test auf dem Jenkins Server.

Um das Beziehen der Images für die Testumgebungen einfacher und effizienter zu gestalten, wurde eine Image Build Pipeline erstellt. Dadurch müssen die genutzten Images nicht für jeden Testdurchlauf erneut gebaut werden. Dies ermöglicht es ebenfalls, Anpassung an einem Basis-Image durch alle darauf aufbauenden Images kaskadieren zu lassen. Alle Images dieser Pipeline wurden in einem Zentralen Artifactory abgelegt, wodurch alle Teams Zugriff auf diese haben und sie in ihren Projekten verwenden können.

Nachdem die aufgeführten Verbesserungen durchgeführt wurden, wurde die Pipeline für ServO entsprechend angepasst. Die Build- und Testumgebung basieren auf Images aus dem Artifactory und die Integrationstests werden gegen den OKD Cluster durchgeführt.

Nach demselben Muster wurde die CI/CD Pipeline des XTA, eine von verschiedensten E-Government Projekten genutzte Schnittstelle zu weiteren Schnittstellen, angepasst. Die Applikation wurde zuerst in einen Docker Container verlegt, anschließend wurde das OKD Template erstellt und parametrisiert, sodass für Integrationstest und

Deployment dieselbe Datei genutzt werden kann. Mit den bei ServO gesammelten Erfahrungen konnten diese Prozesse in wenigen Tagen beendet werden.

9 Zusammenfassung und Ausblick

In diesem Kapitel soll der vorgestellte Lösungsansatz nach den in Kapitel 6 definierten Anforderungen bewertet werden und ein Ausblick auf Möglichkeiten zur Weiterführung, der vorgestellten Konzepte geboten werden.

In dieser Arbeit wurden verschiedene Möglichkeiten aufgezeigt, eine Container-basierte CI/CD Pipeline für diverse Applikationen zu implementieren. Es wurde mit simplen Versuchen begonnen, verschiedene Applikationen in Docker Container zu packen. Nachdem dies gelungen war, wurde mit dem Erstellen der Jenkins Pipeline und parallel dazu dem Aufsetzen des OKD Clusters begonnen. Anfangs gab es keine Festlegung bezüglich der Staging Area, weshalb Jenkins als Deployment Umgebung, für Integrationstests genutzt wurde. Nachdem diese Pipeline zufriedenstellend implementiert wurde, wuchs das Interesse an Docker-basierten Workflows. Daher wurde zuerst die Pipeline für ServO entsprechend angepasst, sodass der OKD Cluster auch für Integrationstest genutzt werden konnte. Das hierbei erlangte Know-How konnte anschließend auf weitere Projekte wie den XTA und das Nationale Waffenregister angewandt werden.

Im Rahmen dieser Arbeit wurden nicht nur Infrastruktur und Vorlagen für Jenkins und OKD, sondern auch Prozesse für Docker-basierte CI/CD Pipelines für verschiedenste Projekte geschaffen. Die erstellten Pipelines haben sich als flexibel erwiesen, sodass eine Jenkinsfile in ein beliebiges Maven Projekt eingefügt werden kann. Projektabhängig sind nur die verwendeten Applikations-Images und eine Umgebungsvariable für den Namen der Applikation.

In Kooperation mit weiteren Sachgebieten und Abteilungen wurde ein weiterer OKD Cluster im Rechenzentrum des DVZ aufgesetzt. Dieser ist von den beteiligten Sachgebieten für jeweils 2 Projekte nutzbar. Dies ermöglichte es, das erlangte Know-How anwendungsspezifisch an interessierte Personen zu verteilen. Der hierfür aufgesetzte OKD Cluster dient als Übergangslösung und Testumgebung, bis ein OpenShift Container Platform Cluster im Rahmen des Projektes „Container Technologien im DVZ“ verfügbar wird.

9.1 Bewertung der vorgestellten Lösung

Die vorgestellte Lösung erfüllt alle in Kapitel 6 definierten Muss-Anforderungen.

Zu den in Kapitel 6 definierten Anforderungen an die CI/CD-Pipeline gehörten: Die Start-Up Zeit zu verringern, den Status der Applikation zu kommunizieren, die

eindeutige Adressierung der Applikation innerhalb des Netzwerkes und das Tagging der Images nach dem bestehenden Muster. Die deployten Artefakte müssen dem aktuellsten Stand entsprechen, der Übergang zwischen Artefakten sollte möglichst ohne Ausfallzeit stattfinden und die gesamte Pipeline muss komplett automatisiert sein.

Es galt, die Start-Up Zeit zu verringern. Die Start-Up Zeit gegenüber dem VM-basierten Ansatz wurde um ca. 4 Minuten und 20 Sekunden verringert. Dies entspricht einer Zeitersparnis von ca. 60%. Diese Tests wurden aber unter unterschiedlichen Bedingungen durchgeführt. Die Docker-basierte Pipeline verfügte über weniger Ressourcen, führte aber auch weniger Tests aus. Ein direkter Vergleich ist aufgrund der zugrundeliegenden Infrastruktur nicht möglich. Diese Tests haben nur eine qualitative, keine quantitative Aussagekraft nicht aber eine quantitative.

Es galt den Status der Applikation zu kommunizieren. Die Applikation wird erst deployt, nachdem die genutzte Datenbank bereit ist, um Anfragen entgegen zu nehmen. Sobald die Applikation erreichbar ist, ist davon auszugehen, dass sie erfolgreich deployt wurde. Sollte die Applikation nach 5 min noch immer nicht erreichbar sein, wird dies als fehlgeschlagener Test gewertet.

Es musste die eindeutige Adressierung der Applikation innerhalb des Netzwerks ermöglicht werden. Die Adressierung, des Deployments in der Staging Area, erfolgt über den von OKD zugewiesenen FQDM.

Das Tagging der Images sollte nach dem bestehenden Muster erfolgen. Es wurde ein neues Muster geschaffen, nachdem Artefakte und Images einheitlich getaggt werden.

Durch den verwendeten ImageStream erfolgt das Deployment eines neuen Images nach spätestens zehn Sekunden. Durch die gewählt „recreate" Deployment-Strategie und die Implementierung von Readiness-Probes, entsteht keine Downtime während des Deployments. Der Status der Container wird indirekt kommuniziert. Ist das neue Image nicht nach 200 Sekunden deployt, wird dies als fehlgeschlagenes Deployment gewertet und das vorherige Image erneut deployt.

9.2 Verbesserungen in der Zukunft

Der in dieser Arbeit vorgestellte Workflow ermöglicht Weiterentwicklungen in viele Richtungen. Regressionstests des Quellcodes sind bekannt und üblich, mit Docker ist es möglich Regressionstests gegen die Umgebung durchzuführen. Nachdem ein Fehler reproduziert wurde, könnte ein automatischer Test erstellt werden, welcher gegen alle in einem bestimmten Zeitraum erstellten Images durchgeführt wird. Somit sind Konfigurationsfehler genauso rückverfolgbar wie es auch bei Quellcode üblich ist.

Das DVZ Schwerin ist nach der ISO-Norm 27001 zertifiziert. Docker-Container werden zurzeit nicht in dieser Norm betrachtet. Momentan ist es daher nicht möglich, Docker-Container in der Produktion zu betreiben. Da Docker-Container in den letzten Jahren aber immer mehr an Bedeutung gewonnen haben und andere Rechenzentren der Länder wie das ITDZ in Berlin bereits Docker-Container betreiben besteht die Möglichkeit, das diese in der Zukunft in die ISO-Norm aufgenommen werden. Sobald die rechtliche Grundlage besteht, könnten zertifizierungskonforme Images von der Abteilung für Betrieb (SC) erstellt werden, die in Entwicklung und Produktion verwendet werden könnten. Dies würde Entwicklung und Betrieb vereinfachen, da Entwicklungsumgebung und Produktivumgebung dieselbe wären. Somit ließe sich Zeit einsparen, indem Fehler durch Konfigurationsunterschiede vermieden würden. Dies ermöglicht es nicht nur Continuous-Delivery sondern auch, Continuous-Deployment zu praktizieren. Continuous-Deployment ist bei einer nicht Container-basierten Produktivumgebung nur schwer möglich.

Wie in Kapitel 7 erwähnt, könnte bei einer weiteren Verbreitung von Docker-Containern weitere existierende Services in z. B. einem OpenShift Cluster betrieben werden. Dies hätte die dort genannten Vorteile der leichten, und bei Bedarf automatischen Skalierung und weitere, wie effiziente Backups von laufenden Systemen zu gewissen Zeitpunkten, erleichterte Wartbarkeit und höhere Zuverlässigkeit.

Die in dieser Arbeit erlangten Kenntnisse und erarbeiteten Prozessen sollen in das Projekt „Container Technologien im DVZ“ einfließen. Geleitet wird dieses vom Enterprise Architekten des DVZ. Dieses Projekt zielt darauf ab, den Einsatz von Container Technologien in jedem Schritt von der Entwicklung bis zur Produktion zu ermöglichen. Das Projekt wurde im Januar 2019 begonnen.

10 Literaturverzeichnis

[1] HTTPS://DOCS.GITLAB.COM: GitLab Continuous-Integration (GitLab CI/CD). https://docs.gitlab.com/ee/ci/README.html#gitlab-continuous-integration-gitlab-ci-cd (23.09.2018, 14:56 Uhr)

[2] MOUAT, ADRIAN: Docker, Software entwickeln und deployen mit Containern. 1. Auflage, dpunkt Verlag, 2016

[3] Qi Zhang , Ling Liu , Calton Pu , Qiwei Dou , Liren Wu , and Wei Zhou A Comparative Study of Containers and Virtual Machines in Big Data Environment. arXiv:1807.01842v1 [cs.DC] 5. Juli 2018

[4] HTTPS://KUBERNETES.IO: Kubernetes Dokumentation https://kubernetes.io/docs/home/?path=users&persona=app-developer&level=foundational (2.10.2018, 09:08 Uhr)

[5] HTTPS://GITHUBG.COM: Docker Compose in 12 Minutes https://github.com/jakewright/tutorials/tree/master/docker/02-docker-compose (Latest commit 6232487 on 13 Mar 2017)

[6] HTTPS://WWW.AQUASEC.COM: Kubernetes Components and Architecture https://www.aquasec.com/wiki/display/containers/Kubernetes+Architecture+101 (3.10.2018, 13:33 Uhr)

[7] HTTPS://WWW.SLIDESHARE.NET: Docker Meetup - ZALENIUM https://www.slideshare.net/Zalando-adtech-lab/10052017-docker-meetup-zalenium (9.10.2018, 12:55 Uhr)

[8] HTTPS://VMWARE.GITHUB.IO/: Building Open Source for Everyone https://vmware.github.io/ (21.10.2018, 15:19 Uhr)

[9] HTTPS://WWW.VMWARE.COM/: VMWARE CLOUD-NATIVE ENTERPRISE INFRASTRUCTURE: Build and Run Traditional and Modern Apps with vSphere Integrated Containers https://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/solutionbrief/vmware-vic-solutions-brief.pdf (14.11.18 9:00 Uhr)

[10] HTTPS://BLOGS.VMWARE.COM: VMware Completes Heptio Acquisition https://blogs.vmware.com/cloudnative/2018/12/11/heptio-close/ (20.02.2019)

[11] Leszko, Rafał: Continuous-Delivery with Docker and Jenkins. 1st Edition, Packt Verlag, August 2017

[12] Denis Zuev, Artemii Kropachev, Aleksey Usov: Learn OpenShift, Deploy build manage and migrate applications with OpenShift Origin 3.9.1st Edition, Packt Verlag, Juli 2018

[13] HTTPS://WWW.SLIDESHARE.NET/: Red Hat OpenShift Containers Platform on Azure https://www.slideshare.net/KhaledElbedri/red-hat-openshift-containers-platform-on-azure (12.11.2018 13:37)

[14] HTTPS://DOCS.OPENSHIFT.COM: Managing Images
https://docs.openshift.com/container-platform/3.11/dev_guide/managing_images.html (21.02.2019, 23:20 Uhr)

[15] HTTPS://WWW.BSI.BUND.DE/: Schutzbedarf und Schutzziele
https://www.bsi.bund.de/DE/Themen/ITGrundschutz/ITGrundschutzAbout/ITGrundschutzSchulung/WebkursITGrundschutz/Schutzbedarfsfeststellung/Schutzbedarfskategorien/Schutzziele/schutzziele_node.html (13.11.2018, 10:30 Uhr)

11 Bilderverzeichnis

1. Bild 1: GitLab CI/CD Pipeline von [1] 7
2. Bild 2: Docker-Container Layer Filesystem 11
3. Bild 3: "High level Kubernetes architecture showing a cluster with a master and two worker nodes" von [6] 12
4. Bild 4: "High level Kubernetes architecture showing a cluster with a master and two worker aus [6] 13
5. Bild 5: Aufbau eines Zalenium Grids von [7] 18
6. Bild 6: OpenShift Architektur aus [13] 23
7. Bild 7: CI/CD Pipeline „Nationales Waffenregister“ 24
8. Bild 8: CI/CD Pipeline Entwurf (vereinfacht) 31
9. Bild 9: CI/CD Pipeline mit Jenkins und OpenShift Cluster 36
10. Bild 10: Ausschnitte der Jenkins Pipelines für ServO.MV 38
11. Bild 11: OpenShift Übersicht der deployten Applikation 38

12 Tabellenverzeichnis

1. Tabelle 1: Übersicht der Selenium/Zalenium Dashboards 17

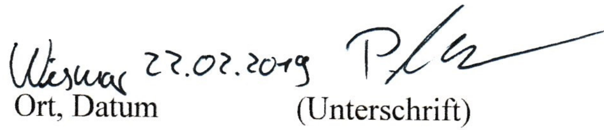
2. Tabelle 2: Überblick über Kombinationsmöglichkeiten von Jenkins und Staging Area 28

13 Verzeichnis der Abkürzungen

CI	Continuous-Integration
CD	Continuous-Delivery
SCM	Source Code Managementsystem
VM	Virtuelle Maschine
VIC	VMware Integrated Containers
OKD	The Origin community distribution of Kubernetes that powers OpenShift
IaaS	Infrastructure as a Service
CaaS	Container as a Service
PaaS	Platform as a Service
SaaS	Software as a Service
DNS	Domain Name System
DVZ	DVZ Datenverarbeitungszentrum Mecklenburg-Vorpommern GmbH
DVZ Schwerin	DVZ Datenverarbeitungszentrum Mecklenburg-Vorpommern GmbH

14 Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die hier vorliegende Arbeit selbstständig, ohne unerlaubte fremde Hilfe und nur unter Verwendung der in der Arbeit aufgeführten Hilfsmittel angefertigt habe.



Ort, Datum (Unterschrift)