|  |  |
| --- | --- |
| Informationstechnologie | |
| Dokumentenart | **Arbeitsunterlage** |
| Titel | **OpenShift Grundlagen** |
| Nummer | Nummer eingeben |
| Autor/-in | Urs Frey |
| Kontaktangaben | +41 58 338 58 70  urs.frey@post.ch |
| Ausgabestelle | IT222 |
| Geltungsbereich | Geltungsbereich eingeben |
| Klassifizierung | intern |
| Version | X01.00 |
| Ausgabedatum | 27. März 2017 |
| Ersetzt Ausgabe vom | Datum auswählen |
| Verteiler | Verteiler eingeben |

From: <https://github.com/appuio/techlab>

**Inhalt:**

[1. Lab 1: Quicktour durch OpenShift V3 3](#_Toc478732549)

[1.1 Grundkonzepte 3](#_Toc478732550)

[1.2 Docker 3](#_Toc478732551)

[1.3 Kubernetes 3](#_Toc478732552)

[1.4 Übersicht 3](#_Toc478732553)

[1.5 Container und Docker Images 3](#_Toc478732554)

[1.5.1 Beispiel Tomcat 3](#_Toc478732555)

[1.6 Projekte 4](#_Toc478732556)

[1.7 Pods 4](#_Toc478732557)

[1.8 Services 4](#_Toc478732558)

[1.9 Routen 4](#_Toc478732559)

[1.10 Templates 4](#_Toc478732560)

[2. Lab 2: OpenShift CLI installieren 5](#_Toc478732561)

[2.1 Command Line Interface 5](#_Toc478732562)

[2.2 oc Client herunterladen und installieren 5](#_Toc478732563)

[2.3 Korrekte Berechtigung auf Linux und macOS erteilen 5](#_Toc478732564)

[2.4 den oc Client im PATH registrieren 5](#_Toc478732565)

[2.5 Windows 5](#_Toc478732566)

[2.5.1 Windows Quick Hack 5](#_Toc478732567)

[2.6 Installation verifizieren 5](#_Toc478732568)

[2.7 bash/zsh completion 5](#_Toc478732569)

[3. Lab 3: Erste Schritte auf der Lab Plattform 7](#_Toc478732570)

[3.1 Login 7](#_Toc478732571)

[3.2 Projekt erstellen 7](#_Toc478732572)

[3.3 Aufgabe: LAB3.1 7](#_Toc478732573)

[3.4 Web Console 7](#_Toc478732574)

[3.5 Aufgabe: LAB3.2 7](#_Toc478732575)

[3.6 Lösung: LAB3.1 9](#_Toc478732576)

[4. Lab 4: Ein Docker Image deployen 10](#_Toc478732577)

[4.1 Aufgabe: LAB4.1 10](#_Toc478732578)

[4.2 Betrachten der erstellten Ressourcen 11](#_Toc478732579)

[4.3 Service 11](#_Toc478732580)

[4.4 ImageStream 13](#_Toc478732581)

[4.5 DeploymentConfig 13](#_Toc478732582)

[4.6 Zusatzaufgabe für Schnelle ;-) 13](#_Toc478732583)

[5. Lab 5: Unseren Service mittels Route online verfügbar machen 14](#_Toc478732584)

[5.1 Routen 14](#_Toc478732585)

[5.2 Aufgabe: LAB5.1 14](#_Toc478732586)

[6. Lab 6: Pod Scaling, Readiness Probe und Self Healing 15](#_Toc478732587)

[6.1 Example Applikation hochskalieren 15](#_Toc478732588)

[6.2 Aufgabe: LAB6.1 skalieren unserer Beispiel Applikation 15](#_Toc478732589)

[6.3 Aufgabe: LAB6.2 skalierte App in der Web Console 16](#_Toc478732590)

[6.4 Unterbruchsfreies Skalieren überprüfen 16](#_Toc478732591)

[6.5 Unterbruchsfreies Deployment mittels Readiness Probe und Rolling Update 17](#_Toc478732592)

[6.6 Aufgabe: LAB6.3 17](#_Toc478732593)

[6.7 Self Healing 19](#_Toc478732594)

[7. Lab 7: Troubleshooting, was ist im Pod? 20](#_Toc478732595)

[7.1 In Container einloggen 20](#_Toc478732596)

[7.2 Aufgabe: LAB7.1 20](#_Toc478732597)

[7.3 Aufgabe: LAB7.2 20](#_Toc478732598)

[7.4 Logfiles anschauen 20](#_Toc478732599)

[7.5 Aufgabe: LAB7.3 Port Forwarding 21](#_Toc478732600)

[8. Lab 8: Datenbank anbinden 22](#_Toc478732601)

[8.1 Aufgabe: LAB8.1: MySQL Service anlegen 22](#_Toc478732602)

[8.1.1 CLI 22](#_Toc478732603)

[8.1.2 Web Console 22](#_Toc478732604)

[8.2 Aufgabe: LAB8.2: Applikation an die Datenbank anbinden 23](#_Toc478732605)

[8.3 Aufgabe: LAB8.3: In MySQL Service Pod einloggen und manuell auf DB verbinden 24](#_Toc478732606)

[8.4 Aufgabe: LAB8.4: Dump auf MySQL DB einspielen 25](#_Toc478732607)

[8.5 Lösung: LAB8.4 25](#_Toc478732608)

[9. Lab 9: Code Changes durch Webhook triggern Rebuild auf OpenShift 26](#_Toc478732609)

[9.1 Aufgabe: LAB9.1: Vorbereitung Github Account und Fork 26](#_Toc478732610)

[9.1.1 Github Account 26](#_Toc478732611)

[9.1.2 Beispiel-Projekt forken 26](#_Toc478732612)

[9.1.3 Deployen des eigenen Forks 26](#_Toc478732613)

[9.2 Aufgabe: LAB9.2: Webhook auf GitHub einrichten 27](#_Toc478732614)

[9.3 Aufgabe: LAB9.3: Code anpassen 30](#_Toc478732615)

[9.4 Aufgabe: LAB9.4: Rollback 31](#_Toc478732616)

[10. Lab 10: Persistent Storage anbinden und verwenden für Datenbank 32](#_Toc478732617)

[10.1 Aufgabe: LAB10.1: 32](#_Toc478732618)

[10.1.1 Storage anfordern 32](#_Toc478732619)

[10.1.2 Volume in Pod einbinden 32](#_Toc478732620)

[10.2 Aufgabe: LAB10.2: Persistenz-Test 32](#_Toc478732621)

[10.2.1 Daten wiederherstellen 32](#_Toc478732622)

[10.2.2 Test 32](#_Toc478732623)

[11. Lab 11: Applikationstemplates 34](#_Toc478732624)

[11.1 Templates 34](#_Toc478732625)

[11.2 Aufgabe: LAB11.1: Template instanzieren. 34](#_Toc478732626)

[12. Lab 12: Eigene Templates erstellen 36](#_Toc478732627)

[12.1 Hilfreiche oc client Befehle 36](#_Toc478732628)

[12.2 Generierung 36](#_Toc478732629)

[12.2.1 Generierung vor Erstellung 36](#_Toc478732630)

[12.2.2 Generierung nach Erstellung 36](#_Toc478732631)

[12.2.3 Vorhandene Templates exportieren 37](#_Toc478732632)

[12.3 Parameter 37](#_Toc478732633)

[12.3.1 Parameter von Templates anzeigen 37](#_Toc478732634)

[12.3.2 Parameter von Templates mit Werten ersetzen 37](#_Toc478732635)

[12.4 Templates schreiben 38](#_Toc478732636)

[12.4.1 Generierte Parameter 38](#_Toc478732637)

[12.4.2 Template Merge 38](#_Toc478732638)

[12.5 Anwenden vom Templates 38](#_Toc478732639)

[12.6 Metadata / Labels 38](#_Toc478732640)

[12.7 Ressourcen aus docker-compose.yml erstellen 38](#_Toc478732641)

# Lab 1: Quicktour durch OpenShift V3

In diesem Lab werden die Grundkonzepte von OpenShift vorgestellt. Des Weiteren zeigen wir auf, wie man sich in der Web Console einloggt und stellen die einzelnen Bereiche kurz vor.

Die hier aufgeführten Begriffe und Ressourcen sind ein Auszug aus der offiziellen OpenShift Dokumentation, weiterführende Informationen zu OpenShift können hier entnommen werden:

<https://docs.openshift.com/container-platform/3.3/architecture/index.html>

## Grundkonzepte

OpenShift V3 basiert auf modernen Open Source Konzepten wie Docker und Kubernetes und bietet damit eine Plattform, mit der Software in Containern gebuildet, deployt und betrieben werden kann. OpenShift V3 kann als Container Platform oder Platform as a Service (PaaS) bezeichnet werden.

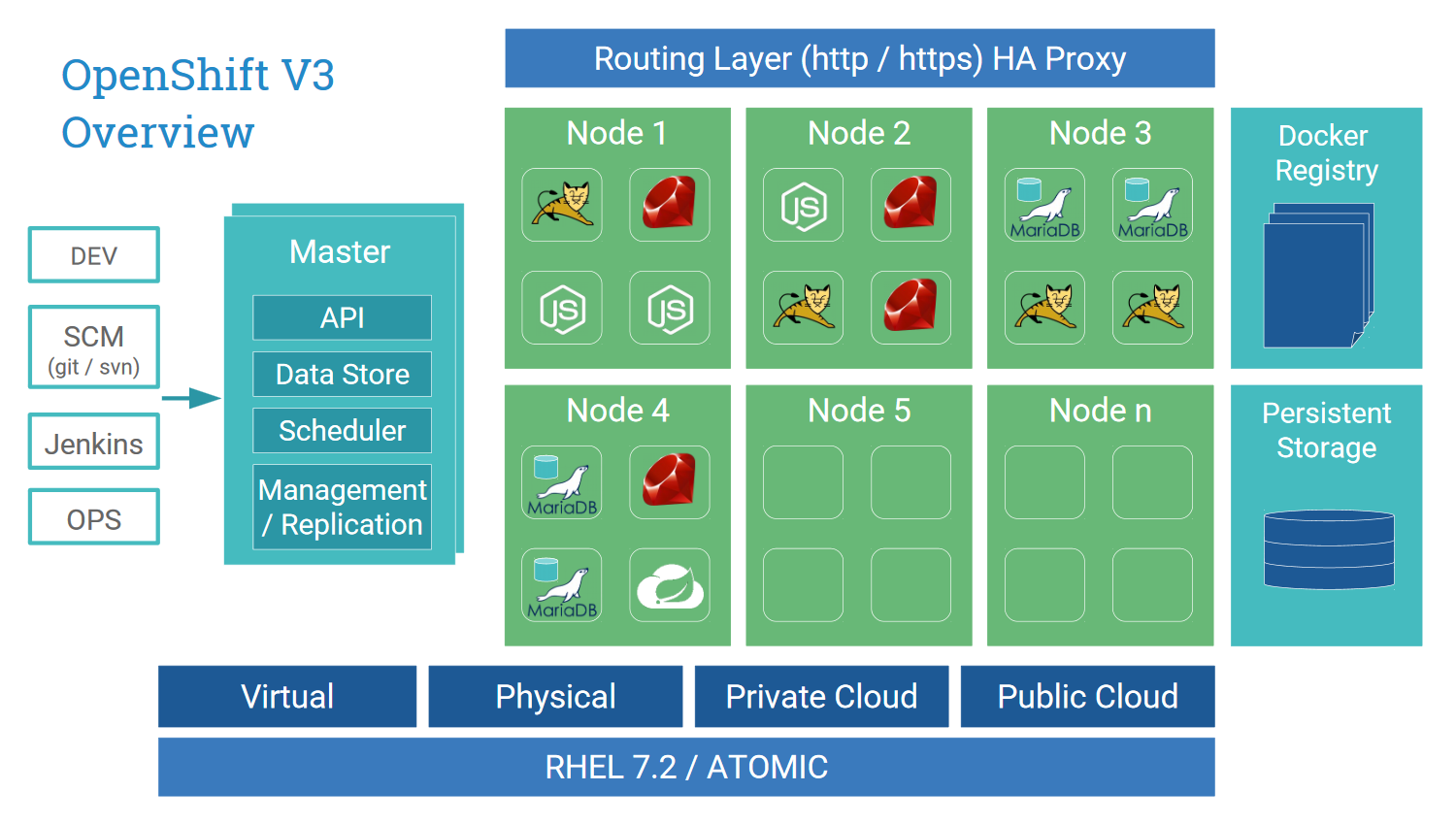
## Docker

[Docker](https://www.docker.com/) ist die offene Plattform für Entwickler und Sysadmins und ihre Applikationen. Wählen Sie das für Ihre Technologie passende Basis-Docker-Images aus, OpenShift baut für Sie nach jedem Build automatisch einen aktualisierten Docker-Container und deployt ihn auf Wunsch auch gleich.

## Kubernetes

Container orchestrieren und managen mit [Kubernetes](http://kubernetes.io/) von Google. Sie definieren, wie viele Instanzen Ihrer Applikation parallel laufen sollen und Kubernetes kümmert sich um die Skalierung, das Loadbalancing und die Stabilität.

## Übersicht

[](https://github.com/appuio/techlab/blob/lab-3.3/images/ose3-overview.png)

## Container und Docker Images

Die Basiselemente von OpenShift Applikationen sind Docker Container. Mit Docker Container können Prozesse auf einem Linuxsystem so isoliert werden, dass sie nur mit den definierten Ressourcen interagieren können. So können viele unterschiedliche Container auf dem gleichen System laufen, ohne dass sie einander "sehen" (Files, Prozesse, Netzwerk). Typischerweise beinhaltet ein Container einen einzelnen Service (Webserver, Datenbank, Mailservice, Cache). Innerhalb eines Docker Containers können beliebige Prozesse ausgeführt werden.

Docker Container basieren auf Docker Images. Ein Docker Image ist eine binary Datei, die alle nötigen Komponenten beinhaltet, damit ein einzelner Container ausgeführt werden kann.

Docker Images werden anhand von DockerFiles (textueller Beschrieb wie das Docker Image Schritt für Schritt aufgebaut ist) gebuildet. Grundsätzlich sind Docker Images hierarchisch angewendete Filesystem Snapshots.

### Beispiel Tomcat

* Basis Image (CentOs 7)
  + Install Java
  + Install Tomcat
  + Install App

Die gebuildeten Docker Images werden in der OpenShift internen Docker Registry versioniert abgelegt und stehen der Plattform nach dem Build zum Deployment zur Verfügung.

## Projekte

In OpenShift V3 werden Ressourcen (Container und Docker Images, Pods, Services, Routen, Konfiguration, Quotas und Limiten etc.) in Projekten strukturiert. Aus technischer Sicht entspricht ein Projekt einem Kubernetes Namespace und erweitert diesen um gewisse Konzepte.

Innerhalb eines Projekts können berechtigte User ihre Ressourcen selber verwalten und organisieren.

Die Ressourcen innerhalb eines Projektes sind über ein transparentes [SDN](https://de.wikipedia.org/wiki/Software-defined_networking) verbunden. So können die einzelnen Komponenten eines Projektes in einem Multi-Node Setup auf verschiedene Nodes deployed werden. Dabei sind sie über das SDN untereinander sicht- und zugreifbar.

## Pods

OpenShift übernimmt das Konzept der Pods von Kubernetes.

Ein Pod ist ein oder mehrere Container, die zusammen auf den gleichen Host deployed werden. Ein Pod ist die kleinste zu deployende Einheit auf OpenShift.

Ein Pod ist innerhalb eines OpenShift Projektes über den entsprechenden Service verfügbar.

## Services

Ein Service repräsentiert einen internen Loadbalancer auf die dahinterliegenden Pods (Replicas vom gleichen Typ). Der Service dient als Proxy zu den Pods und leitet Anfragen an diese weiter. So können Pods willkürlich einem Service hinzugefügt und entfernt werden, während der Service verfügbar bleibt.

Einem Service ist innerhalb eines Projektes eine IP und ein Port zugewiesen und verteilt Requests entsprechend auf die Pod Replicas.

## Routen

Mit einer Route definiert man in OpenShift, wie ein Service von ausserhalb von OpenShift von externen Clients erreicht werden kann.

Diese Routen werden im integrierten Routing Layer eingetragen und erlauben dann der Plattform über ein Hostname-Mapping die Requests an den entsprechenden Service weiterzuleiten.

Sind mehr als ein Pod für einen Service deployt, verteilt der Routing Layer die Requests auf die deployten Pods

Aktuell werden folgende Protokolle unterstützt:

* HTTP
* HTTPS ([SNI](https://en.wikipedia.org/wiki/Server_Name_Indication))
* WebSockets
* TLS mit [SNI](https://en.wikipedia.org/wiki/Server_Name_Indication)

## Templates

Ein Template beschreibt textuell eine Liste von Ressourcen, die auf OpenShift ausgeführt und entsprechend in OpenShift erstellt werden können.

So hat man die Möglichkeit ganze Infrastrukturen zu beschreiben:

* Java Applikation Service (3 Replicas, rolling Upgrade)
* Datenbank Service
* über Route [https://java.app.appuio-beta.ch](https://java.app.appuio-beta.ch/) im Internet verfügbar

**Ende Lab 1**

# Lab 2: OpenShift CLI installieren

In diesem Lab werden wir gemeinsam den oc Client installieren und konfigurieren, damit wir danach die ersten Schritte auf der OpenShift Techlab Plattform durchführen können.

## Command Line Interface

Der **oc client** stellt ein Interface zu OpenShift V3 bereit.

Der Client ist in Go programmiert und kommt als einzelnes Binary für die folgenden Betriebsysteme daher:

* Microsoft Windows
* macOS
* Linux

## oc Client herunterladen und installieren

Der oc Client wird vom [GitHub-Repository von OpenShift Origin](https://github.com/openshift/origin/releases/tag/v1.3.2) heruntergeladen:

[Windows](https://github.com/openshift/origin/releases/download/v1.3.2/openshift-origin-client-tools-v1.3.2-ac1d579-windows.zip)

[Mac](https://github.com/openshift/origin/releases/download/v1.3.2/openshift-origin-client-tools-v1.3.2-ac1d579-mac.zip)

[Linux 64bit](https://github.com/openshift/origin/releases/download/v1.3.2/openshift-origin-client-tools-v1.3.2-ac1d579-linux-64bit.tar.gz)

[Linux 32bit](https://github.com/openshift/origin/releases/download/v1.3.2/openshift-origin-client-tools-v1.3.2-ac1d579-linux-32bit.tar.gz)

[CHECKSUM](https://github.com/openshift/origin/releases/download/v1.3.2/CHECKSUM)

Sobald der Client heruntergeladen wurde, muss er auf dem System in einem Verzeichnis, das über den **PATH** erreichbar ist, abgelegt werden.

**Linux**

~/bin

**macOS**

~/bin

**Windows**

C:\OpenShift\

## Korrekte Berechtigung auf Linux und macOS erteilen

Der oc Client muss ausgeführt werden können.

cd ~/bin

chmod +x oc

## den oc Client im PATH registrieren

Unter **Linux** und **macOS** ist das Verzeichnis ~/bin bereits im PATH, daher muss hier nichts gemacht werden.

Falls der oc Client in einem anderen Verzeichnis abgelegt wurde, kann der PATH wie folgt gesetzt werden:

$ export PATH=$PATH:[path to oc client]

## Windows

Unter Windows kann der PATH in den erweiterten Systemeinstellungen konfiguriert werden. Dies ist abhängig von der entsprechenden Windows Version:

[Windows 7](http://geekswithblogs.net/renso/archive/2009/10/21/how-to-set-the-windows-path-in-windows-7.aspx): <http://geekswithblogs.net/renso/archive/2009/10/21/how-to-set-the-windows-path-in-windows-7.aspx>

[Windows 8](http://www.itechtics.com/customize-windows-environment-variables/)

[Windows 10](http://techmixx.de/windows-10-umgebungsvariablen-bearbeiten/): <http://techmixx.de/windows-10-umgebungsvariablen-bearbeiten/>

### Windows Quick Hack

Legen sie den oc Client direkt im Verzeichnis *C:\Windows* ab.

## Installation verifizieren

Der oc Client sollte jetzt korrekt installiert sein. Am besten überprüfen wir das, indem wir den folgenden Command ausführen:

$ oc version

Der folgende Output sollte angezeigt werden:

oc v1.3.1

kubernetes v1.3.0+52492b4

Ist dies nicht der Fall, ist möglicherweise die PATH Variable nicht korrekt gesetzt.

## bash/zsh completion

Mit Linux und Mac kann die bash completion mit folgendem Befehl temporär eingerichtet werden:

source <(oc completion bash)

Oder für zsh:

source <(oc completion zsh)

# Lab 3: Erste Schritte auf der Lab Plattform

In diesem Lab werden wir gemeinsam das erste Mal mit der Lab Plattform interagieren, dies sowohl über den oc Client wie auch über die Web Console

## Login

**Note:** Versichern Sie sich, dass Sie [Lab 2](https://github.com/appuio/techlab/blob/lab-3.3/labs/02_cli.md) erfolgreich abgeschlossen haben.

Bitte verwenden Sie für das Login auf dem Webinterface sowie mit oc die durch den Instruktor zur Verfügung gestellten Angaben.

## Projekt erstellen

Ein Projekt in OpenShift ist das Top Level Konzept um ihre Applikationen, Deployments, Builds, Container etc. zu organisieren. Siehe [Lab1](https://github.com/appuio/techlab/blob/lab-3.3/labs/01_quicktour.md).

## Aufgabe: LAB3.1

Erstellen Sie auf der Lab Plattform ein neues Projekt.

**Note**: Verwenden Sie für Ihren Projektnamen am besten Ihren Github Namen oder ihren Nachnamen, bspw. [USER]-example1

Wie kann ein neues Projekt erstellt werden?

**Tipp** 

$ oc help

## Web Console

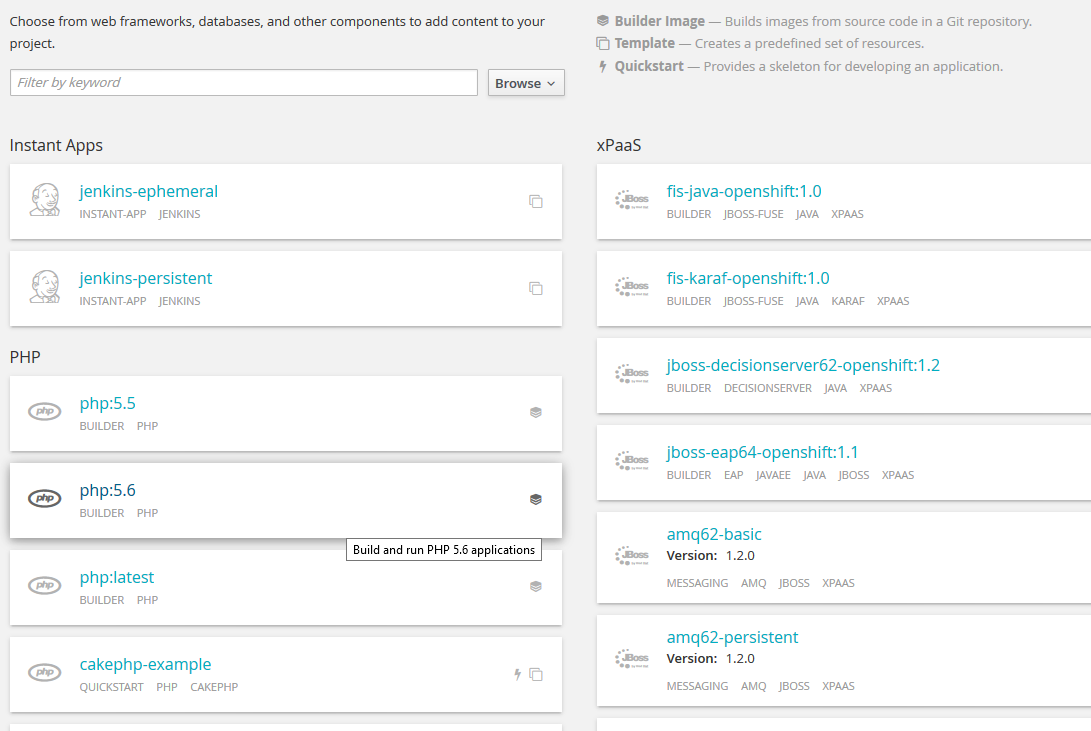
Die OpenShift V3 Web Console erlaubt es den Benutzern gewisse Tasks direkt via Browser vorzunehmen.

## Aufgabe: LAB3.2

1. Loggen Sie sich nun via Web Console auf der Lab Plattform ein.

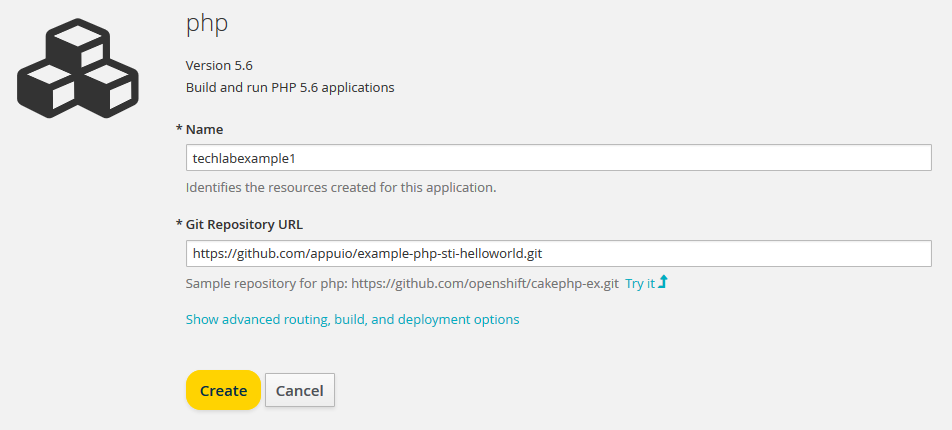
**Note:** Die **URL**, Benutzernamen und Passwort für Ihren Account wird Ihnen entsprechend am Techlab durch den Instruktor zur Verfügung gestellt.

1. Gehen Sie nun in die Übersicht Ihres eben erstellten Projektes. Aktuell ist das Projekt noch leer.
2. Fügen Sie über *Add to Project* Ihre erste Applikation Ihrem Projekt hinzu. Als Beispielprojekt verwenden wir ein APPUiO Example.

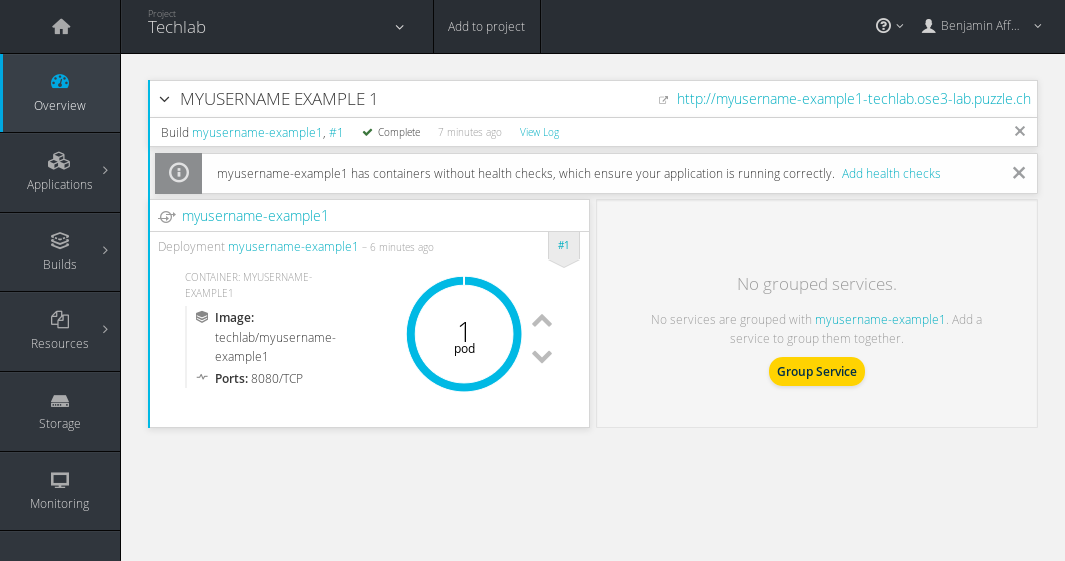
3.1. Wählen Sie dazu das Basis Image php 5.6 aus [](https://github.com/appuio/techlab/blob/lab-3.3/images/lab_3_php5.6.png)

3.2. Geben Sie Ihrem Beispiel einen sprechenden Namen und folgende URL als Repo URL

https://github.com/appuio/example-php-sti-helloworld.git

[](https://github.com/appuio/techlab/blob/lab-3.3/images/lab_3_example1.png)

1. Die Applikation wurde erstellt. Über den Link **Continue to overview** kommt man zur Übersicht.
2. Der Build Ihrer Applikation wird gestartet. Verfolgen Sie den Build und schauen Sie sich nach dem Deployment die Beispiel App an.

[](https://github.com/appuio/techlab/blob/lab-3.3/images/lab_3_example1-deployed.png)

Sie haben nun ihre erste Applikation mittels sogenanntem [**Source to Image**](https://docs.openshift.com/container-platform/3.3/architecture/core_concepts/builds_and_image_streams.html#source-build) Build auf OpenShift deployed.

**Tipp:** mit dem folgenden Command können Sie in ein anderes Projekt wechseln:

$ oc project [projectname]

**Tipp:** Mit den folgenden Befehlen kann das obere Beispiel aus der Kommandozeile erstellt werden:

$ oc new-app https://github.com/appuio/example-php-sti-helloworld.git --name=appuio-php-sti-example

$ oc expose svc appuio-php-sti-example

**Tipp:** eine ganze App kann mit dem folgenden Befehl gelöscht werden:

$ oc delete all -l app=appname

bspw.

$ oc delete all -l app=appuio-php-sti-example

## Lösung: LAB3.1

$ oc new-project [USER]-example1

**Ende Lab 3**

# Lab 4: Ein Docker Image deployen

In diesem Lab werden wir gemeinsam das erste "pre-built" Docker Image deployen und die OpenShift-Konzepte Pod, Service, DeploymentConfig und ImageStream etwas genauer anschauen.

## Aufgabe: LAB4.1

Nachdem wir im [Lab 3](https://github.com/appuio/techlab/blob/lab-3.3/labs/03_first_steps.md) den Source-to-Image Workflow verwendet haben, um eine Applikation auf OpenShift zu deployen, wenden wir uns nun dem Deployen eines pre-built Docker Images von Docker Hub oder einer anderen Docker-Registry zu.

[Weiterführende Dokumentation](https://docs.openshift.com/container-platform/3.3/dev_guide/new_app.html#specifying-an-image)

Als ersten Schritt erstellen wir dafür ein neues Projekt. Ein Projekt ist eine Gruppierung von Ressourcen (Container und Docker Images, Pods, Services, Routen, Konfiguration, Quotas, Limiten und weiteres). Für das Projekt berechtigte User können diese Ressourcen verwalten. Innerhalb eines OpenShift V3 Clusters muss der Name eines Projektes eindeutig sein.

Erstellen Sie daher ein neues Projekt mit dem Namen [USER]-dockerimage:

$ oc new-project [USER]-dockerimage

oc new-project wechselt automatisch in das eben neu angelegte Projekt. Mit dem oc get Command können Ressourcen von einem bestimmten Typ angezeigt werden.

Verwenden Sie

$ oc get project

um alle Projekte anzuzeigen, auf die Sie berechtigt sind.

Sobald das neue Projekt erstellt wurde, können wir in OpenShift mit dem folgenden Befehl das Docker Image deployen:

$ oc new-app appuio/example-spring-boot

Output:

--> Found Docker image d790313 (3 weeks old) from Docker Hub for "appuio/example-spring-boot"

APPUiO Spring Boot App

----------------------

Example Spring Boot App

Tags: builder, springboot

\* An image stream will be created as "example-spring-boot:latest" that will track this image

\* This image will be deployed in deployment config "example-spring-boot"

\* Port 8080/tcp will be load balanced by service "example-spring-boot"

\* Other containers can access this service through the hostname "example-spring-boot"

--> Creating resources with label app=example-spring-boot ...

imagestream "example-spring-boot" created

deploymentconfig "example-spring-boot" created

service "example-spring-boot" created

--> Success

Run 'oc status' to view your app.

Für unser Lab verwenden wir ein APPUiO-Beispiel (Java Spring Boot Applikation):

* Docker Hub: <https://hub.docker.com/r/appuio/example-spring-boot/>
* GitHub (Source): <https://github.com/appuio/example-spring-boot-helloworld>

OpenShift legt die nötigen Ressourcen an, lädt das Docker Image in diesem Fall von Docker Hub herunter und deployt anschliessend den ensprechenden Pod.

**Tipp:** Verwenden Sie oc status um sich einen Überblick über das Projekt zu verschaffen.

Oder verwenden Sie den oc get Befehl mit dem -w Parameter, um fortlaufend Änderungen an den Ressourcen des Typs Pod anzuzeigen (abbrechen mit ctrl+c):

$ oc get pods -w

Je nach Internetverbindung oder abhängig davon, ob das Image auf Ihrem OpenShift Node bereits heruntergeladen wurde, kann das eine Weile dauern. Schauen Sie sich doch in der Web Console den aktuellen Status des Deployments an:

1. Loggen Sie sich in der Web Console ein
2. Wählen Sie Ihr Projekt [USER]-dockerimage aus
3. Klicken Sie auf Applications
4. Wählen Sie Pods aus

**Tipp** Um Ihre eigenen Docker Images für OpenShift zu erstellen, sollten Sie die folgenden Best Practices befolgen: <https://docs.openshift.com/container-platform/3.3/creating_images/guidelines.html>

## Betrachten der erstellten Ressourcen

Als wir oc new-app appuio/example-spring-boot vorhin ausführten, hat OpenShift im Hintergrund einige Ressourcen für uns angelegt. Die werden dafür benötigt, dieses Docker Image zu deployen:

* [Service](https://docs.openshift.com/container-platform/3.3/architecture/core_concepts/pods_and_services.html#services)
* [ImageStream](https://docs.openshift.com/container-platform/3.3/architecture/core_concepts/builds_and_image_streams.html#image-streams)
* [DeploymentConfig](https://docs.openshift.com/container-platform/3.3/dev_guide/deployments/how_deployments_work.html)

## Service

[Services](https://docs.openshift.com/container-platform/3.3/architecture/core_concepts/pods_and_services.html#services) dienen innerhalb OpenShift als Abstraktionslayer, Einstiegspunkt und Proxy/Loadbalancer auf die dahinterliegenden Pods. Der Service ermöglicht es, innerhalb OpenShift eine Gruppe von Pods des gleichen Typs zu finden und anzusprechen.

Als Beispiel: Wenn eine Applikationsinstanz unseres Beispiels die Last nicht mehr alleine verarbeiten kann, können wir die Applikation bspw. auf drei Pods hochskalieren. OpenShift mapt diese als Endpoints automatisch zum Service. Sobald die Pods bereit sind, werden Requests automatisch auf alle drei Pods verteilt.

**Note:** Die Applikation kann aktuell von aussen noch nicht erreicht werden, der Service ist ein OpenShift-internes Konzept. Im folgenden Lab werden wir die Applikation öffentlich verfügbar machen.

Nun schauen wir uns unseren Service mal etwas genauer an:

$ oc get services

NAME CLUSTER-IP EXTERNAL-IP PORT(S) AGE

example-spring-boot 172.30.124.20 <none> 8080/TCP 2m

Wie Sie am Output sehen, ist unser Service (example-spring-boot) über eine IP und Port erreichbar (172.30.124.20:8080) **Note:** Ihre IP kann unterschiedlich sein.

**Note:** Service IPs bleiben während ihrer Lebensdauer immer gleich.

Mit dem folgenden Befehl können Sie zusätzliche Informationen über den Service auslesen:

$ oc get service example-spring-boot -o json

{

"kind": "Service",

"apiVersion": "v1",

"metadata": {

"name": "example-spring-boot",

"namespace": "techlab",

"selfLink": "/api/v1/namespaces/techlab/services/example-spring-boot",

"uid": "b32d0197-347e-11e6-a2cd-525400f6ccbc",

"resourceVersion": "17247237",

"creationTimestamp": "2016-06-17T11:29:05Z",

"labels": {

"app": "example-spring-boot"

},

"annotations": {

"openshift.io/generated-by": "OpenShiftNewApp"

}

},

"spec": {

"ports": [

{

"name": "8080-tcp",

"protocol": "TCP",

"port": 8080,

"targetPort": 8080

}

],

"selector": {

"app": "example-spring-boot",

"deploymentconfig": "example-spring-boot"

},

"portalIP": "172.30.124.20",

"clusterIP": "172.30.124.20",

"type": "ClusterIP",

"sessionAffinity": "None"

},

"status": {

"loadBalancer": {}

}

}

Mit dem entsprechenden Befehl können Sie auch die Details zu einem Pod anzeigen:

$ oc get pod example-spring-boot-3-nwzku -o json

**Note:** Zuerst den pod Namen aus Ihrem Projekt abfragen (oc get pods) und im oberen Befehl ersetzen.

Über den selector Bereich im Service wird definiert, welche Pods (labels) als Endpoints dienen. Dazu die entsprechenden Konfigurationen vom Service und Pod zusammen betrachten.

Service:

--------

...

"selector": {

"app": "example-spring-boot",

"deploymentconfig": "example-spring-boot"

},

...

Pod:

----

...

"labels": {

"app": "example-spring-boot",

"deployment": "example-spring-boot-1",

"deploymentconfig": "example-spring-boot"

},

...

Diese Verknüpfung ist mittels dem oc describe Befehl zu sehen:

$ oc describe service example-spring-boot

Name: example-spring-boot

Namespace: techlab

Labels: app=example-spring-boot

Selector: app=example-spring-boot,deploymentconfig=example-spring-boot

Type: ClusterIP

IP: 172.30.124.20

Port: 8080-tcp 8080/TCP

Endpoints: 10.1.3.20:8080

Session Affinity: None

No events.

Unter Endpoints finden Sie nun den aktuell laufenden Pod.

## ImageStream

[ImageStreams](https://docs.openshift.com/container-platform/3.3/architecture/core_concepts/builds_and_image_streams.html#image-streams) werden dafür verwendet, automatische Tasks auszuführen wie bspw. ein Deployment zu aktualisieren, wenn eine neue Version des Images oder des Basisimages verfügbar ist.

Builds und Deployments können Image Streams beobachten und auf Änderungen entsprechend reagieren. In unserem Beispiel wird der Image Stream dafür verwendet, ein Deployment zu triggern, sobald etwas am Image geändert hat.

Mit dem folgenden Befehl können Sie zusätzliche Informationen über den Image Stream auslesen:

$ oc get imagestream example-spring-boot -o json

## DeploymentConfig

In der [DeploymentConfig](https://docs.openshift.com/container-platform/3.3/dev_guide/deployments/how_deployments_work.html) werden folgende Punkte definiert:

* Update Strategy: wie werden Applikationsupdates ausgeführt, wie erfolgt das Austauschen der Container?
* Triggers: Welche Triggers führen zu einem Deployment? In unserem Beispiel ImageChange
* Container
  + Welches Image soll deployed werden?
  + Environment Configuration für die Pods
  + ImagePullPolicy
* Replicas, Anzahl der Pods, die deployt werden sollen

Mit dem folgenden Befehl können zusätzliche Informationen zur DeploymentConfig ausgelesen werden:

$ oc get deploymentConfig example-spring-boot -o json

Im Gegensatz zur DeploymentConfig, mit welcher man OpenShift sagt, wie eine Applikation deployt werden soll, definiert man mit dem ReplicationController, wie die Applikation während der Laufzeit aussehen soll (bspw. dass immer 3 Replicas laufen sollen).

**Tipp:** für jeden Resource Type gibt es auch eine Kurzform. So können Sie bspw. oc get deploymentconfig auch einfach als oc get dc schreiben.

## Zusatzaufgabe für Schnelle ;-)

Schauen Sie sich die erstellten Ressourcen mit oc get [ResourceType] [Name] -o json und oc describe [ResourceType] [Name] aus dem ersten Projekt [USER]-example1 an.

**Ende Lab 4**

# Lab 5: Unseren Service mittels Route online verfügbar machen

In diesem Lab werden wir die Applikation aus [Lab 4](https://github.com/appuio/techlab/blob/lab-3.3/labs/04_deploy_dockerimage.md) über **http** vom Internet her erreichbar machen.

## Routen

Der oc new-app Befehl aus dem vorherigen [Lab](https://github.com/appuio/techlab/blob/lab-3.3/labs/04_deploy_dockerimage.md) erstellt keine Route. Somit ist unser Service von *aussen* her gar nicht erreichbar. Will man einen Service verfügbar machen, muss dafür eine Route eingerichtet werden. Der OpenShift Router erkennt aufgrund des Host Headers auf welchen Service ein Request geleitet werden muss.

Aktuell werden folgende Protokolle unterstützt:

* HTTP
* HTTPS ([SNI](https://en.wikipedia.org/wiki/Server_Name_Indication))
* WebSockets
* TLS mit [SNI](https://en.wikipedia.org/wiki/Server_Name_Indication)

## Aufgabe: LAB5.1

Vergewissern Sie sich, dass Sie sich im Projekt [USER]-dockerimage befinden. **Tipp:** oc project [USER]-dockerimage

Erstellen Sie für den Service example-spring-boot eine Route und machen Sie ihn darüber öffentlich verfügbar.

**Tipp:** Mittels oc get routes können Sie sich die Routen eines Projekts anzeigen lassen.

$ oc get routes

Aktuell gibt es noch keine Route. Jetzt brauchen wir den Servicenamen:

$ oc get services

NAME CLUSTER-IP EXTERNAL-IP PORT(S) AGE

example-spring-boot 172.30.124.20 <none> 8080/TCP 11m

Und nun wollen wir diesen Service veröffentlichen / exposen:

$ oc expose service example-spring-boot

Per default wird eine http Route erstellt.

Mittels oc get routes können wir überprüfen, ob die Route angelegt wurde.

$ oc get routes

NAME HOST/PORT PATH SERVICE TERMINATION LABELS

example-spring-boot example-spring-boot-techlab.app.appuio.ch example-spring-boot:8080-tcp app=example-spring-boot

Die Applikation ist nun vom Internet her über den angegebenen Hostnamen erreichbar, Sie können also nun auf die Applikation zugreifen.

**Tipp:** Wird kein Hostname angegeben wird der Standardname verwendet: *servicename-project.osecluster*

In der Overview der Web Console ist diese Route mit dem Hostnamen jetzt auch sichtbar.

**Ende Lab 5**

# Lab 6: Pod Scaling, Readiness Probe und Self Healing

In diesem Lab zeigen wir auf, wie man Applikationen in OpenShift skaliert. Des Weiteren zeigen wir, wie OpenShift dafür sorgt, dass jeweils die Anzahl erwarteter Pods gestartet wird und wie eine Applikation der Plattform zurückmelden kann, dass sie bereit für Requests ist.

## Example Applikation hochskalieren

Dafür erstellen wir ein neues Projekt

$ oc new-project [USER]-scale

und fügen dem Projekt eine Applikation hinzu

$ oc new-app appuio/example-php-docker-helloworld --name=appuio-php-docker

und stellen den Service zur Verfügung (expose)

$ oc expose service appuio-php-docker

Wenn wir unsere Example Applikation skalieren wollen, müssen wir unserem ReplicationController (rc) mitteilen, dass wir bspw. stets 3 Replicas des Images am Laufen haben wollen.

Schauen wir uns mal den ReplicationController (rc) etwas genauer an:

$ oc get rc

NAME DESIRED CURRENT AGE

appuio-php-docker-1 1 1 33s

Für mehr Details:

$ oc get rc appuio-php-docker-1 -o json

Der rc sagt uns, wieviele Pods wir erwarten (spec) und wieviele aktuell deployt sind (status).

## Aufgabe: LAB6.1 skalieren unserer Beispiel Applikation

Nun skalieren wir unsere Example Applikation auf 3 Replicas:

$ oc scale --replicas=3 rc appuio-php-docker-1

Überprüfen wir die Anzahl Replicas auf dem ReplicationController:

$ oc get rc

NAME DESIRED CURRENT AGE

appuio-php-docker-1 3 3 1m

und zeigen entsprechend die Pods an:

$ oc get pods

NAME READY STATUS RESTARTS AGE

appuio-php-docker-1-2uc89 1/1 Running 0 21s

appuio-php-docker-1-evcre 1/1 Running 0 21s

appuio-php-docker-1-tolpx 1/1 Running 0 2m

Zum Schluss schauen wir uns den Service an. Der sollte jetzt alle drei Endpoints referenzieren:

$ oc describe svc appuio-php-docker

Name: appuio-php-docker

Namespace: techlab-scale

Labels: app=appuio-php-docker

Selector: app=appuio-php-docker,deploymentconfig=appuio-php-docker

Type: ClusterIP

IP: 172.30.166.88

Port: 8080-tcp 8080/TCP

Endpoints: 10.1.3.23:8080,10.1.4.13:8080,10.1.5.15:8080

Session Affinity: None

No events.

Skalieren von Pods innerhalb eines Services ist sehr schnell, da OpenShift einfach eine neue Instanz des Docker Images als Container startet.

**Tipp:** OpenShift V3 unterstützt auch Autoscaling, die Dokumentation dazu ist unter dem folgenden Link zu finden: <https://docs.openshift.com/container-platform/3.3/dev_guide/pod_autoscaling.html>

## Aufgabe: LAB6.2 skalierte App in der Web Console

Schauen Sie sich die skalierte Applikation auch in der Web Console an.

## Unterbruchsfreies Skalieren überprüfen

Mit dem folgenden Befehl können Sie nun überprüfen, ob Ihr Service verfügbar ist, während Sie hoch und runter skalieren. Ersetzen Sie dafür [route] mit Ihrer definierten Route:

**Tipp:** oc get route

while true; do sleep 1; curl -s http://[route]/pod/; date "+ TIME: %H:%M:%S,%3N"; done

oder in PowerShell (Achtung: erst ab PowerShell-Version 3.0!):

while(1) {

Start-Sleep -s 1

Invoke-RestMethod http://[route]/pod/

Get-Date -Uformat "+ TIME: %H:%M:%S,%3N"

}

und skalieren Sie von **3** Replicas auf **1**. Der Output zeigt jeweils den Pod an, der den Request verarbeitete:

POD: appuio-php-docker-6-9w9t4 TIME: 16:40:04,991

POD: appuio-php-docker-6-9w9t4 TIME: 16:40:06,053

POD: appuio-php-docker-6-6xg2b TIME: 16:40:07,091

POD: appuio-php-docker-6-6xg2b TIME: 16:40:08,128

POD: appuio-php-docker-6-ctbrs TIME: 16:40:09,175

POD: appuio-php-docker-6-ctbrs TIME: 16:40:10,212

POD: appuio-php-docker-6-9w9t4 TIME: 16:40:11,279

POD: appuio-php-docker-6-9w9t4 TIME: 16:40:12,332

POD: appuio-php-docker-6-6xg2b TIME: 16:40:13,369

POD: appuio-php-docker-6-6xg2b TIME: 16:40:14,407

POD: appuio-php-docker-6-6xg2b TIME: 16:40:15,441

POD: appuio-php-docker-6-6xg2b TIME: 16:40:16,493

POD: appuio-php-docker-6-6xg2b TIME: 16:40:17,543

POD: appuio-php-docker-6-6xg2b TIME: 16:40:18,591

Die Requests werden an die unterschiedlichen Pods geleitet, sobald man runterskaliert auf einen Pod, gibt dann nur noch einer Antwort

Was passiert nun, wenn wir nun während dem der While Befehl oben läuft, ein neues Deployment starten:

$ oc deploy appuio-php-docker --latest

Währen einer kurzen Zeit gibt die öffentliche Route keine Antwort

POD: appuio-php-docker-6-6xg2b TIME: 16:42:17,743

POD: appuio-php-docker-6-6xg2b TIME: 16:42:18,776

POD: appuio-php-docker-6-6xg2b TIME: 16:42:19,813

POD: appuio-php-docker-6-6xg2b TIME: 16:42:20,853

POD: appuio-php-docker-6-6xg2b TIME: 16:42:21,891

POD: appuio-php-docker-6-6xg2b TIME: 16:42:22,943

POD: appuio-php-docker-6-6xg2b TIME: 16:42:23,980

# keine Antwort

POD: appuio-php-docker-7-pxnr3 TIME: 16:42:42,134

POD: appuio-php-docker-7-pxnr3 TIME: 16:42:43,181

POD: appuio-php-docker-7-pxnr3 TIME: 16:42:44,226

POD: appuio-php-docker-7-pxnr3 TIME: 16:42:45,259

POD: appuio-php-docker-7-pxnr3 TIME: 16:42:46,297

POD: appuio-php-docker-7-pxnr3 TIME: 16:42:47,571

POD: appuio-php-docker-7-pxnr3 TIME: 16:42:48,606

POD: appuio-php-docker-7-pxnr3 TIME: 16:42:49,645

POD: appuio-php-docker-7-pxnr3 TIME: 16:42:50,684

In unserem Beispiel verwenden wir einen sehr leichtgewichtigen Pod. Das Verhalten ist ausgeprägter, wenn der Container länger braucht bis er Requests abarbeiten kann. Bspw. Java Applikation von LAB 4: **Startup: 30 Sekunden**

Pod: example-spring-boot-2-73aln TIME: 16:48:25,251

Pod: example-spring-boot-2-73aln TIME: 16:48:26,305

Pod: example-spring-boot-2-73aln TIME: 16:48:27,400

Pod: example-spring-boot-2-73aln TIME: 16:48:28,463

Pod: example-spring-boot-2-73aln TIME: 16:48:29,507

<html><body><h1>503 Service Unavailable</h1>

No server is available to handle this request.

</body></html>

TIME: 16:48:33,562

<html><body><h1>503 Service Unavailable</h1>

No server is available to handle this request.

</body></html>

TIME: 16:48:34,601

...

Pod: example-spring-boot-3-tjdkj TIME: 16:49:20,114

Pod: example-spring-boot-3-tjdkj TIME: 16:49:21,181

Pod: example-spring-boot-3-tjdkj TIME: 16:49:22,231

Es kann dann sogar sein, dass der Service gar nicht mehr online ist und der Routing Layer ein **503 Error** zurück gibt.

Im Folgenden Kapitel wird beschrieben, wie Sie ihre Services konfigurieren können, dass unterbruchsfreie Deployments möglich werden.

## Unterbruchsfreies Deployment mittels Readiness Probe und Rolling Update

Die Update Strategie [Rolling](https://docs.openshift.com/container-platform/3.3/dev_guide/deployments/deployment_strategies.html#rolling-strategy) ermöglicht unterbruchsfreie Deployments. Damit wird die neue Version der Applikation gestartet, sobald die Applikation bereit ist, werden Request auf den neuen Pod geleitet und die alte Version undeployed.

Zusätzlich kann mittels [Container Health Checks](https://docs.openshift.com/container-platform/3.3/dev_guide/application_health.html) die deployte Applikation der Plattform detailliertes Feedback über ihr aktuelles Befinden geben.

Grundsätzlich gibt es zwei Checks, die implementiert werden können:

* Liveness Probe, sagt aus, ob ein laufender Container immer noch sauber läuft.
* Readiness Probe, gibt Feedback darüber, ob eine Applikation bereit ist, um Requests zu empfangen. Ist v.a. im Rolling Update relevant.

Diese beiden Checks können als HTTP Check, Container Execution Check (Shell Script im Container) oder als TCP Socket Check implementiert werden.

In unserem Beispiel soll die Applikation der Plattform sagen, ob sie bereit für Requests ist. Dafür verwenden wir die Readiness Probe. Unsere Beispielapplikation gibt auf der folgenden URL auf Port 9000 (Management-Port der Spring Applikation) ein Status Code 200 zurück, sobald die Applikation bereit ist.

<http://[route]/health/>

## Aufgabe: LAB6.3

In der Deployment Config (dc) definieren im Abschnitt der Rolling Update Strategie, dass bei einem Update die App immer verfügbar sein soll: maxUnavailable: 0%

Dies kann in der Deployment Config (dc) konfiguriert werden:

**YAML:**

...

spec:

strategy:

type: Rolling

rollingParams:

updatePeriodSeconds: 1

intervalSeconds: 1

timeoutSeconds: 600

maxUnavailable: 0%

maxSurge: 25%

resources: { }

...

Die Deployment Config kann via Web Console (Applications → Deployments → example-php-docker-helloworld, edit) oder direkt über oc editiert werden.

$ oc edit dc appuio-php-docker

Oder im JSON-Format editieren:

$ oc edit dc appuio-php-docker -o json

**json**

"strategy": {

"type": "Rolling",

"rollingParams": {

"updatePeriodSeconds": 1,

"intervalSeconds": 1,

"timeoutSeconds": 600,

"maxUnavailable": "0%",

"maxSurge": "25%"

},

"resources": {}

}

Die Readiness Probe muss in der Deployment Config (dc) hinzugefügt werden, und zwar unter:

spec --> template --> spec --> containers unter halb von resources: { }

**YAML:**

...

resources: { }

readinessProbe:

httpGet:

path: /health/

port: 8080

scheme: HTTP

initialDelaySeconds: 10

timeoutSeconds: 1

...

**json:**

...

"resources": {},

"readinessProbe": {

"httpGet": {

"path": "/health/",

"port": 8080,

"scheme": "HTTP"

},

"initialDelaySeconds": 10,

"timeoutSeconds": 1

},

...

Passen Sie das entsprechend analog oben an.

Die Konfiguration unter Container muss dann wie folgt aussehen: **YAML:**

containers:

-

name: example-php-docker-helloworld

image: 'appuio/example-php-docker-helloworld@sha256:6a19d4a1d868163a402709c02af548c80635797f77f25c0c391b9ce8cf9a56cf'

ports:

-

containerPort: 8080

protocol: TCP

resources: { }

readinessProbe:

httpGet:

path: /health/

port: 8080

scheme: HTTP

initialDelaySeconds: 10

timeoutSeconds: 1

terminationMessagePath: /dev/termination-log

imagePullPolicy: IfNotPresent

**json:**

"containers": [

{

"name": "appuio-php-docker",

"image": "appuio/example-php-docker-helloworld@sha256:9e927f9d6b453f6c58292cbe79f08f5e3db06ac8f0420e22bfd50c750898c455",

"ports": [

{

"containerPort": 8080,

"protocol": "TCP"

}

],

"resources": {},

"readinessProbe": {

"httpGet": {

"path": "/health/",

"port": 8080,

"scheme": "HTTP"

},

"initialDelaySeconds": 10,

"timeoutSeconds": 1

},

"terminationMessagePath": "/dev/termination-log",

"imagePullPolicy": "Always"

}

],

Verifizieren Sie während eines Deployment der Applikation, ob nun auch ein Update der Applikation unterbruchsfrei verläuft:

Einmal pro Sekunde ein Request:

while true; do sleep 1; curl -s http://[route]/pod/; date "+ TIME: %H:%M:%S,%3N"; done

Starten des Deployment:

$ oc deploy appuio-php-docker –latest

## Self Healing

Über den Replication Controller haben wir nun der Plattform mitgeteilt, dass jeweils **n** Replicas laufen sollen. Was passiert nun, wenn wir einen Pod löschen?

Suchen Sie mittels oc get pods einen Pod im Status "running" aus, den Sie *killen* können.

Starten sie in einem eigenen Terminal den folgenden Befehl (anzeige der Änderungen an Pods)

oc get pods -w

Löschen Sie im anderen Terminal einen Pod mit folgendem Befehl

oc delete pod appuio-php-docker-3-788j5

OpenShift sorgt dafür, dass wieder **n** Replicas des genannten Pods laufen.

**Ende Lab 6**

# Lab 7: Troubleshooting, was ist im Pod?

In diesem Lab wird aufgezeigt, wie man im Fehlerfall und Troubleshooting vorgehen kann und welche Tools einem dabei zur Verfügung stehen.

## In Container einloggen

Wir verwenden dafür wieder das Projekt aus [Lab 4](https://github.com/appuio/techlab/blob/lab-3.3/labs/04_deploy_dockerimage.md) [USER]-dockerimage

Laufende Container werden als unveränderbare Infrastruktur behandelt und sollen generell nicht modifiziert werden. Dennoch gibt es Usecases, bei denen man sich in die Container einloggen muss. Zum Beispiel für Debugging und Analysen.

## Aufgabe: LAB7.1

Mit OpenShift können Remote Shells in die Pods geöffnet werden ohne dass man darin vorgängig SSH installieren müsste. Dafür steht einem der Befehl oc rsh zur Verfügung.

Wählen Sie mittels oc get pods einen Pod aus und führen Sie den folgenden Befehl aus:

$ oc rsh [POD]

Sie können nun über diese Shell Analysen im Container ausführen:

bash-4.2$ ls -la

total 16

drwxr-xr-x. 7 default root 99 May 16 13:35 .

drwxr-xr-x. 4 default root 54 May 16 13:36 ..

drwxr-xr-x. 6 default root 57 May 16 13:35 .gradle

drwxr-xr-x. 3 default root 18 May 16 12:26 .pki

drwxr-xr-x. 9 default root 4096 May 16 13:35 build

-rw-r--r--. 1 root root 1145 May 16 13:33 build.gradle

drwxr-xr-x. 3 root root 20 May 16 13:34 gradle

-rwxr-xr-x. 1 root root 4971 May 16 13:33 gradlew

drwxr-xr-x. 4 root root 28 May 16 13:34 src

## Aufgabe: LAB7.2

Einzelne Befehle innerhalb des Containers können über oc exec ausgeführt werden:

$ oc exec [POD] env

$ oc exec example-spring-boot-4-8mbwe env

PATH=/opt/app-root/src/bin:/opt/app-root/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin

HOSTNAME=example-spring-boot-4-8mbwe

KUBERNETES\_SERVICE\_PORT\_DNS\_TCP=53

KUBERNETES\_PORT\_443\_TCP\_PROTO=tcp

KUBERNETES\_PORT\_443\_TCP\_ADDR=172.30.0.1

KUBERNETES\_PORT\_53\_UDP\_PROTO=udp

KUBERNETES\_PORT\_53\_TCP=tcp://172.30.0.1:53

...

## Logfiles anschauen

Die Logfiles zu einem Pod können sowohl in der Web Console als auch auch im CLI angezeigt werden.

$ oc logs [POD]

Der Parameter -f bewirkt analoges Verhalten wie tail -f

Befindet sich ein Pod im Status **CrashLoopBackOff** bedeutet dies, dass er auch nach wiederholtem Restarten nicht erfolgreich gestartet werden konnte. Die Logfiles können auch wenn der Pod nicht läuft mit dem folgenden Befehl angezeigt werden.

$ oc logs -p [POD]

Mit OpenShift wird ein EFK (Elasticsearch, Fluentd, Kibana) Stack mitgeliefert, der sämtliche Logfiles sammelt, rotiert und aggregiert. Kibana erlaubt es Logs zu durchsuchen, zu filtern und grafisch aufzubereiten. Weitere Informationen und ein optionales LAB finden sie [hier](https://github.com/appuio/techlab/blob/lab-3.3/additional-labs/logging_efk_stack.md).

## Aufgabe: LAB7.3 Port Forwarding

OpenShift 3 erlaubt es, beliebige Ports von der Entwicklungs-Workstation auf einen Pod weiterzuleiten. Dies ist z.B. nützlich, um auf Administrationskonsolen, Datenbanken, usw. zuzugreifen, die nicht gegen das Internet exponiert werden und auch sonst nicht erreichbar sind. Im Gegensatz zu OpenShift 2 werden die Portweiterleitungen über dieselbe HTTPS-Verbindung getunnelt, die der OpenShift Client (oc) auch sonst benutzt. Dies erlaubt es auch dann auf OpenShift 3 Platformen zuzugreifen, wenn sich restriktive Firewalls und/oder Proxies zwischen Workstation und OpenShift befinden.

Übung: Auf die Spring Boot Metrics aus [Lab 4](https://github.com/appuio/techlab/blob/lab-3.3/labs/04_deploy_dockerimage.md) zugreifen.

oc get po --namespace="[USER]-dockerimage"

oc port-forward example-spring-boot-1-xj1df 9000:9000 --namespace="[USER]-dockerimage"

Nicht vergessen den Pod Namen an die eigene Installation anzupassen. Falls installiert kann dafür Autocompletion verwendet werden.

Die Metrics können nun unter folgendem Link abgerufen werden: <http://localhost:9000/metrics/> Die Metrics werden Ihnen als Json angezeigt. Mit dem selben Konzept können Sie nun beispielsweise mit ihrem localen SQL Client auf eine Datenbank verbinden.

Unter folgendem Link sind weiterführende Informationen zu Port Forwarding zu finden: <https://docs.openshift.com/container-platform/3.3/dev_guide/port_forwarding.html>

**Ende Lab 7**

# Lab 8: Datenbank anbinden

Die meisten Applikationen sind in irgend einer Art stateful und speichern Daten persistent ab. Sei dies in einer Datenbank oder als Files auf einem Filesystem oder Objectstore. In diesem Lab werden wir in unserem Projekt einen MySQL Service anlegen und an unsere Applikation anbinden, sodass mehrere Applikationspods auf die gleiche Datenbank zugreifen können.

Für dieses Beispiel verwenden wird das Spring Boot Beispiel aus [LAB 4](https://github.com/appuio/techlab/blob/lab-3.3/labs/04_deploy_dockerimage.md), [USER]-dockerimage

## Aufgabe: LAB8.1: MySQL Service anlegen

Für unser Beispiel verwenden wir in diesem Lab ein OpenShift Template, welches eine MySQL Datenbank mit EmptyDir Data Storage anlegt. Dies ist nur für Testumgebungen zu verwenden, da beim Restart des MySQL Pods alle Daten verloren gehen. In einem späteren Lab werden wir aufzeigen, wie wir ein Persistent Volume (mysql-persistent) an die MySQL Datenbank anhängen. Damit bleiben die Daten auch bei Restarts bestehen und ist so für den produktiven Betrieb geeignet.

Den MySQL Service können wir sowohl über die Web Console als auch über das CLI anlegen.

Um dasselbe Ergebnis zu erhalten müssen lediglich Datenbankname, Username, Password und DatabaseServiceName gleich gesetzt werden, egal welche Variante verwendet wird:

* MYSQL\_USER appuio
* MYSQL\_PASSWORD appuio
* MYSQL\_DATABASE appuio
* DATABASE\_SERVICE\_NAME mysql

### CLI

Über das CLI kann der MySQL Service wie folgt angelegt werden:

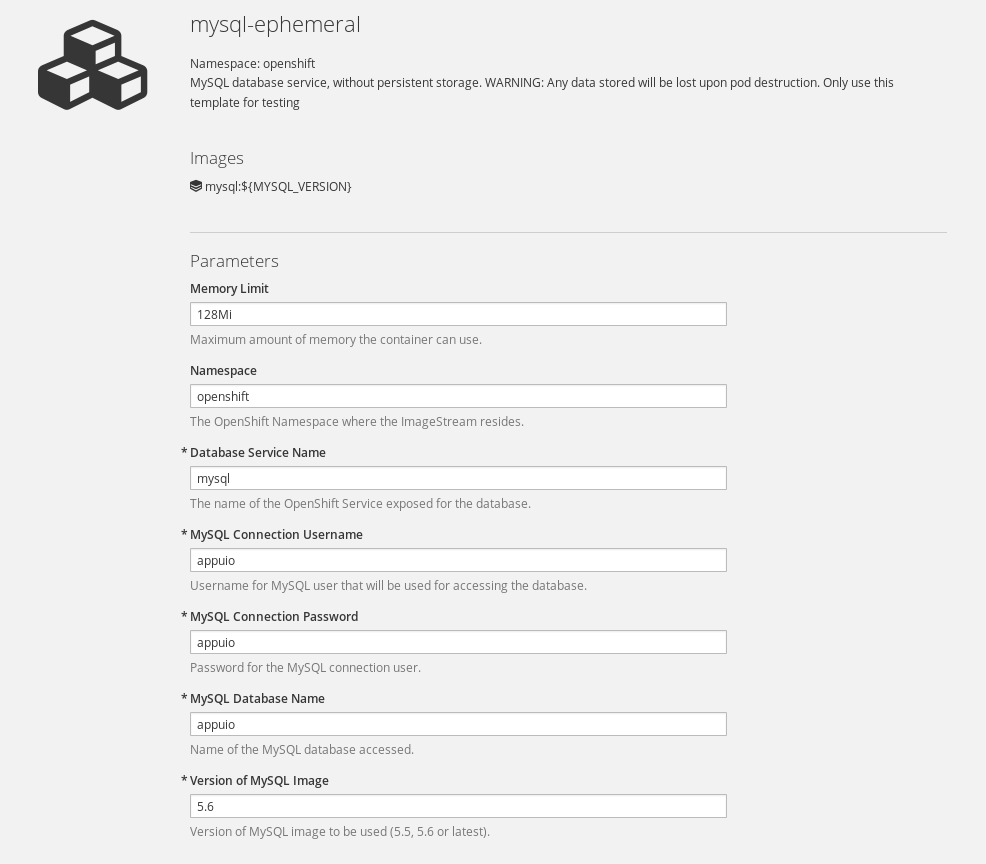
$ oc new-app mysql-ephemeral \

-pMEMORY\_LIMIT=128Mi \

-pMYSQL\_USER=appuio -pMYSQL\_PASSWORD=appuio \

-pMYSQL\_DATABASE=appuio -pDATABASE\_SERVICE\_NAME=mysql

### Web Console

In der Web Console kann der MySQL Service dem Projekt über "Add to Project" hinzugefügt werden. [](https://github.com/appuio/techlab/blob/lab-3.3/images/lab_8_addmysql_service.png)

## Aufgabe: LAB8.2: Applikation an die Datenbank anbinden

Standardmässig wird bei unserer example-spring-boot Applikation eine H2 Memory Datenbank verwendet. Dies kann über das Setzen der folgenden Umgebungsvariablen entsprechend auf unseren neuen MySQL Service umgestellt werden:

* SPRING\_DATASOURCE\_USERNAME appuio
* SPRING\_DATASOURCE\_PASSWORD appuio
* SPRING\_DATASOURCE\_DRIVER\_CLASS\_NAME com.mysql.jdbc.Driver
* SPRING\_DATASOURCE\_URL jdbc:mysql://[Adresse des MySQL Service]/appuio?autoReconnect=true

Für die Adresse des MySQL Service können wir entweder dessen Cluster IP (oc get service) oder aber dessen DNS-Namen (<service>) verwenden. Alle Services und Pods innerhalb eines Projektes können über DNS aufgelöst werden.

So lautet der Wert für die Variable SPRING\_DATASOURCE\_URL bspw.:

Name des Services: mysql

jdbc:mysql://mysql/appuio?autoReconnect=true

Diese Umgebungsvariablen können wir nun in der DeploymentConfig example-spring-boot setzen. Nach dem **ConfigChange** (ConfigChange ist in der DeploymentConfig als Trigger registriert) wird die Applikation automatisch neu deployed. Aufgrund der neuen Umgebungsvariablen verbindet die Applikation an die MySQL DB und [Liquibase](http://www.liquibase.org/) kreiert das Schema und importiert die Testdaten.

**Note:** Liquibase ist Open Source. Es ist eine Datenbank unabhängige Library um Datenbank Änderungen zu verwalten und auf der Datenbank anzuwenden. Liquibase erkennt beim Startup der Applikation, ob DB Changes auf der Datenbank angewendet werden müssen oder nicht. Siehe Logs.

SPRING\_DATASOURCE\_URL=jdbc:mysql://mysql/appuio?autoReconnect=true

**Note:** mysql löst innerhalb ihres Projektes via DNS Abfrage auf die Cluster IP des MySQL Service auf. Die MySQL Datenbank ist nur innerhalb des Projektes erreichbar. Der Service ist ebenfalls über den folgenden Namen erreichbar:

Projektname = techlab-dockerimage

mysql.techlab-dockerimage.svc.cluster.local

Befehl für das Setzen der Umgebungsvariablen:

$ oc env dc example-spring-boot \

-e SPRING\_DATASOURCE\_URL=jdbc:mysql://mysql/appuio?autoReconnect=true \

-e SPRING\_DATASOURCE\_USERNAME=appuio -e SPRING\_DATASOURCE\_PASSWORD=appuio \

-e SPRING\_DATASOURCE\_DRIVER\_CLASS\_NAME=com.mysql.jdbc.Driver

Über den folgenden Befehl können Sie sich die DeploymentConfig als JSON anschauen. Neu enthält die Config auch die gesetzten Umgebungsvariablen:

$ oc get dc example-spring-boot -o json

...

"env": [

{

"name": "SPRING\_DATASOURCE\_USERNAME",

"value": "appuio"

},

{

"name": "SPRING\_DATASOURCE\_PASSWORD",

"value": "appuio"

},

{

"name": "SPRING\_DATASOURCE\_DRIVER\_CLASS\_NAME",

"value": "com.mysql.jdbc.Driver"

},

{

"name": "SPRING\_DATASOURCE\_URL",

"value": "jdbc:mysql://mysql/appuio"

}

],

...

Die Konfiguration kann auch in der Web Console angeschaut und verändert werden:

(Applications → Deployments → example-spring-boot, Actions, Edit YAML)

## Aufgabe: LAB8.3: In MySQL Service Pod einloggen und manuell auf DB verbinden

Wie im Lab [07](https://github.com/appuio/techlab/blob/lab-3.3/labs/07_troubleshooting_ops.md) beschrieben kann mittels oc rsh [POD] in einen Pod eingeloggt werden:

$ oc get pods

NAME READY STATUS RESTARTS AGE

example-spring-boot-8-wkros 1/1 Running 0 10m

mysql-1-diccy 1/1 Running 0 50m

Danach in den MySQL Pod einloggen:

$ oc rsh mysql-1-diccy

Nun können Sie mittels mysql Tool auf die Datenbank verbinden und die Tabellen anzeigen:

$ mysql -u$MYSQL\_USER -p$MYSQL\_PASSWORD -h$MYSQL\_SERVICE\_HOST appuio

Welcome to the MySQL monitor. Commands end with ; or \g.

Your MySQL connection id is 54

Server version: 5.6.26 MySQL Community Server (GPL)

Copyright (c) 2000, 2015, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its

affiliates. Other names may be trademarks of their respective

owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql>

Anschliessend können Sie mit

show tables;

alle Tabellen anzeigen.

## Aufgabe: LAB8.4: Dump auf MySQL DB einspielen

Die Aufgabe ist es, in den MySQL Pod den [Dump](https://raw.githubusercontent.com/appuio/techlab/lab-3.3/labs/data/08_dump/dump.sql) einzuspielen.

**Tipp:** Mit oc rsync können Sie lokale Dateien in einen Pod kopieren. **Achtung:** Beachten Sie, dass dabei der rsync-Befehl des Betriebssystems verwendet wird. Auf UNIX-Systemen kann rsync mit dem Paketmanager, auf Windows kann bspw. [cwRsync](https://www.itefix.net/cwrsync) installiert werden. Ist eine Installation von rsync nicht möglich, kann stattdessen bspw. in den Pod eingeloggt und via curl -O <URL> der Dump heruntergeladen werden.

**Tipp:** Verwenden Sie das Tool mysql um den Dump einzuspielen.

**Tipp:** Die bestehende Datenbank muss vorgängig leer sein. Sie kann auch gelöscht und neu angelegt werden.

## Lösung: LAB8.4

Ein ganzes Verzeichnis (dump) syncen. Darin enthalten ist das File dump.sql. Beachten Sie zum rsync-Befehl auch obenstehenden Tipp.

oc rsync ./labs/data/08\_dump mysql-1-diccy:/tmp/

In den MySQL Pod einloggen:

$ oc rsh mysql-1-diccy

Bestehende Datenbank löschen:

$ mysql -u$MYSQL\_USER -p$MYSQL\_PASSWORD -h$MYSQL\_SERVICE\_HOST appuio

...

mysql> drop database appuio;

mysql> create database appuio;

Dump einspielen:

$ mysql -u$MYSQL\_USER -p$MYSQL\_PASSWORD -h$MYSQL\_SERVICE\_HOST appuio < /tmp/08\_dump/dump.sql

**Note:** Den Dump kann man wie folgt erstellen:

mysqldump --user=$MYSQL\_USER --password=$MYSQL\_PASSWORD --host=$MYSQL\_SERVICE\_HOST appuio > /tmp/dump.sql

**Ende Lab 8**

# Lab 9: Code Changes durch Webhook triggern Rebuild auf OpenShift

In diesem Lab zeigen wir den Docker Build Workflow anhand eines Beispiels auf und Sie lernen, wie Sie mit einem Push in das Git Repository einen Build und ein Deployment der Applikation auf OpenShift starten.

## Aufgabe: LAB9.1: Vorbereitung Github Account und Fork

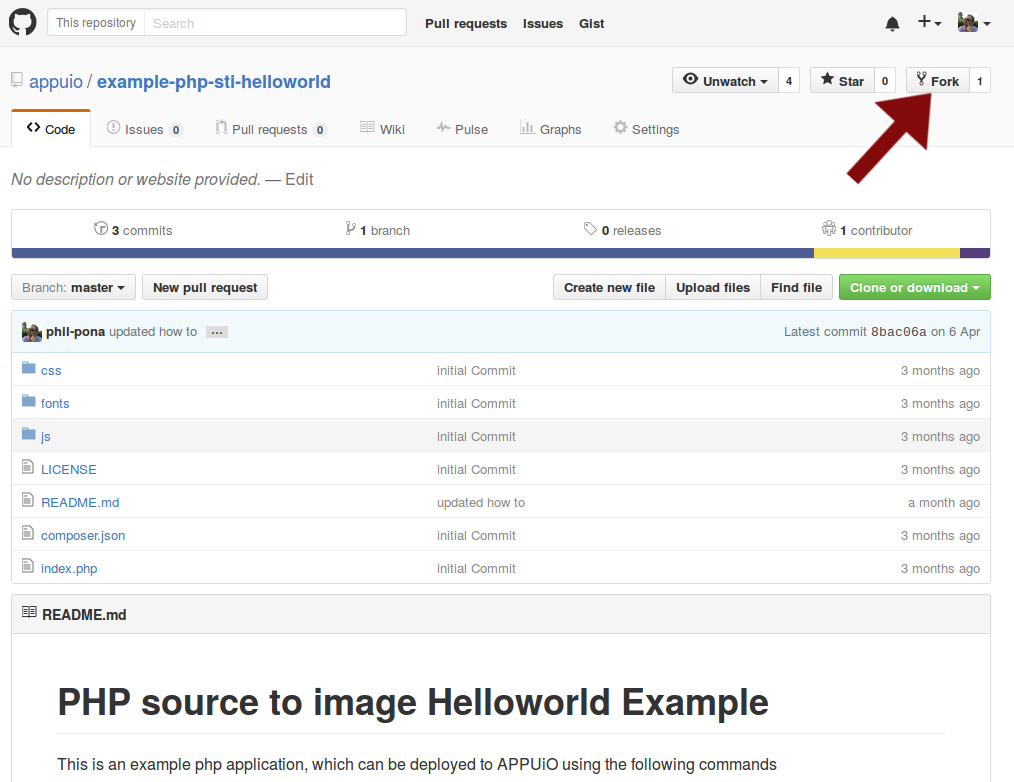
### Github Account

Damit Sie Änderungen am Source Code unserer Beispielapplikation vornehmen können, benötigen Sie einen eigenen GitHub Account. Richten Sie sich einen Account unter <https://github.com/> ein, falls Sie nicht bereits über einen verfügen.

### Beispiel-Projekt forken

**Beispiel-Projekt:** <https://github.com/appuio/example-php-docker-helloworld>

Gehen Sie auf die [GitHub Projekt-Seite](https://github.com/appuio/example-php-docker-helloworld) und [forken](https://help.github.com/articles/fork-a-repo/) Sie das Projekt.

[](https://github.com/appuio/techlab/blob/lab-3.3/images/lab_9_fork_example.png)

Sie haben nun unter

https://github.com/[YourGitHubUser]/example-php-docker-helloworld

einen Fork des Example Projektes, den Sie so erweitern können wie Sie wollen.

### Deployen des eigenen Forks

Erstellen Sie ein neues Projekt:

$ oc new-project [USER]-example4

Erstellen Sie für Ihren Fork eine neue App. **Note:** Ersetzen Sie [YourGithubUser] mit dem Namen Ihres GitHub Accounts:

$ oc new-app https://github.com/[YourGithubUser]/example-php-docker-helloworld.git --strategy=docker --name=appuio-php-docker-ex

Mittels Parameter --strategy=docker sagen wir dem oc new-app Befehl nun explizit, er soll im angegebenen Git Repository nach einem Dockerfile suchen und dieses für den Build verwenden.

Nun exponieren Sie den Service mit:

$ oc expose service appuio-php-docker-ex

## Aufgabe: LAB9.2: Webhook auf GitHub einrichten

Beim Erstellen der App wurden in der BuildConfig (bc) direkt Webhooks definiert. Diese können Sie über den folgenden Befehl anzeigen:

$ oc describe bc appuio-php-docker-ex

Name: appuio-php-docker-ex

Created: 57 seconds ago

Labels: app=appuio-php-docker-ex

Annotations: openshift.io/generated-by=OpenShiftNewApp

Latest Version: 1

Strategy: Docker

URL: https://github.com/appuio/example-php-docker-helloworld.git

From Image: ImageStreamTag php-56-centos7:latest

Output to: ImageStreamTag appuio-php-docker-ex:latest

Triggered by: Config, ImageChange

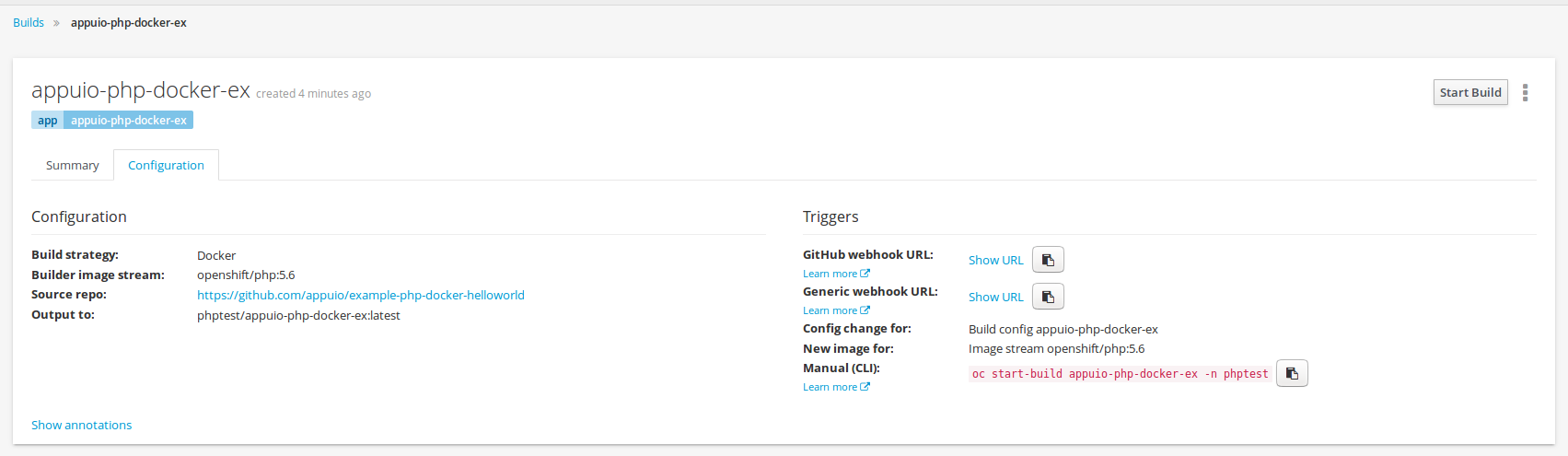
Webhook Generic: https://master.appuio-beta.ch:443/oapi/v1/namespaces/techlab-example4/buildconfigs/appuio-php-docker-ex/webhooks/EqEq18JtxaY3vG2zvPSU/generic

Webhook GitHub: https://master.appuio-beta.ch:443/oapi/v1/namespaces/techlab-example4/buildconfigs/appuio-php-docker-ex/webhooks/hqQ3h1CzUGIXvWqjiV-G/github

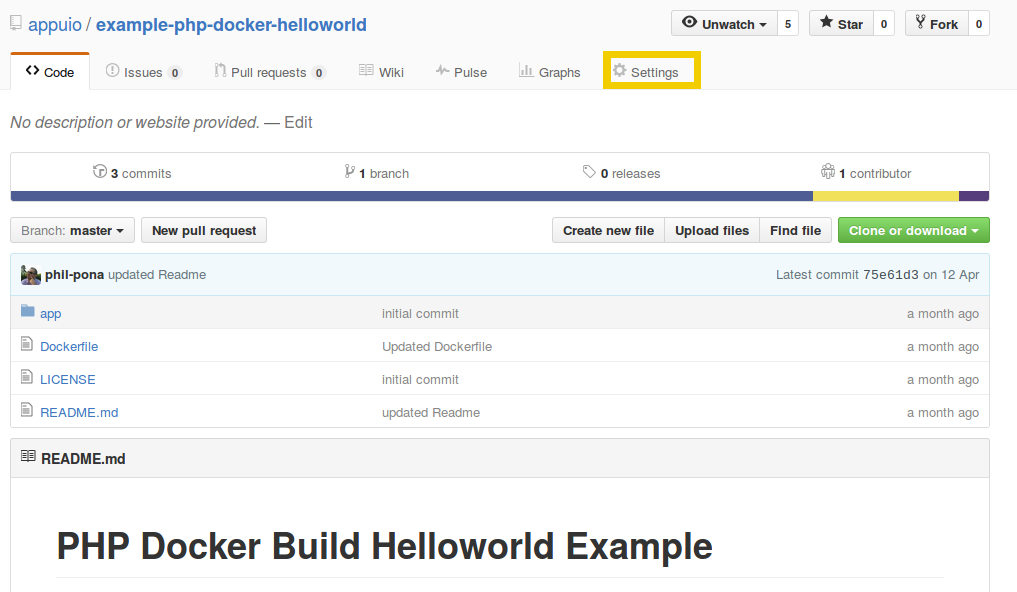
Build Status Duration Creation Time

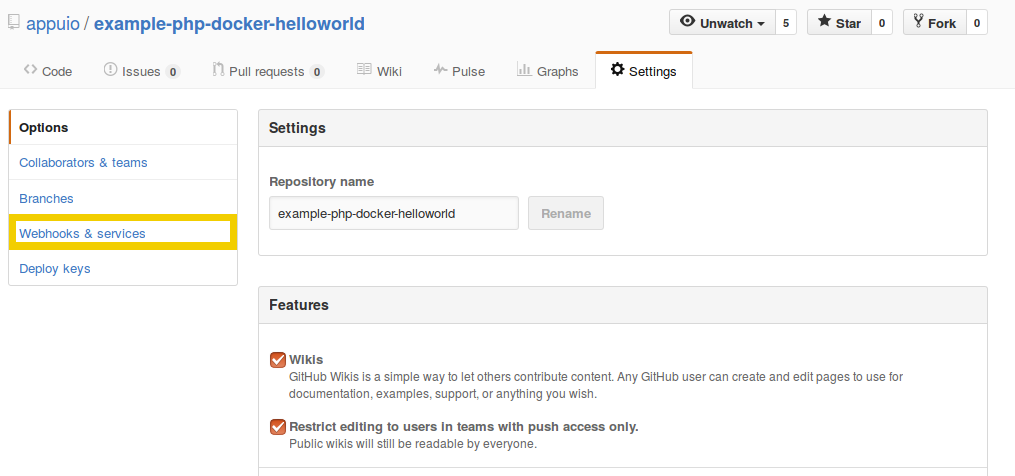
appuio-php-docker-ex-1 running running for 56s 2016-06-17 16:56:34 +0200 CEST

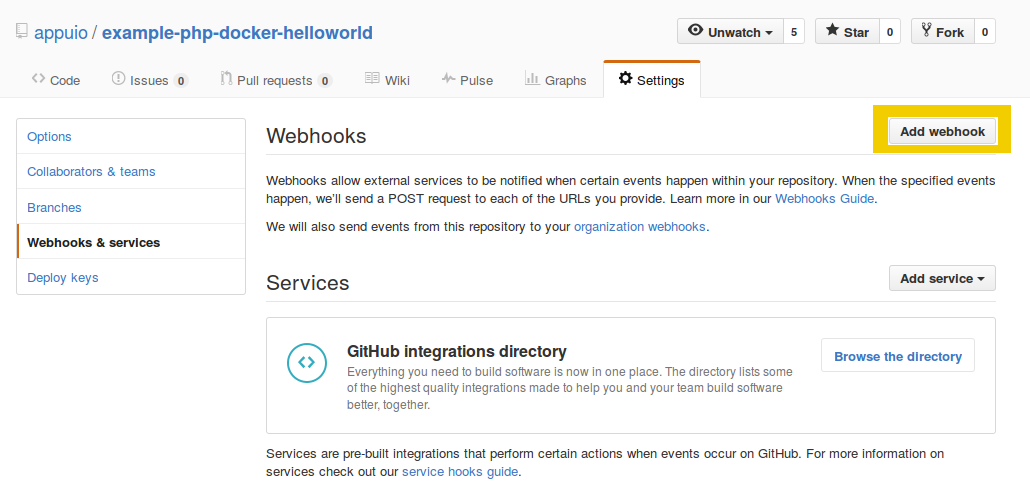
Den GitHub Webhook können Sie auch von der Web Console kopieren. Gehen Sie dafür via Builds → Builds auf den entsprechenden Build und wählen Sie das Tab Configuration aus:

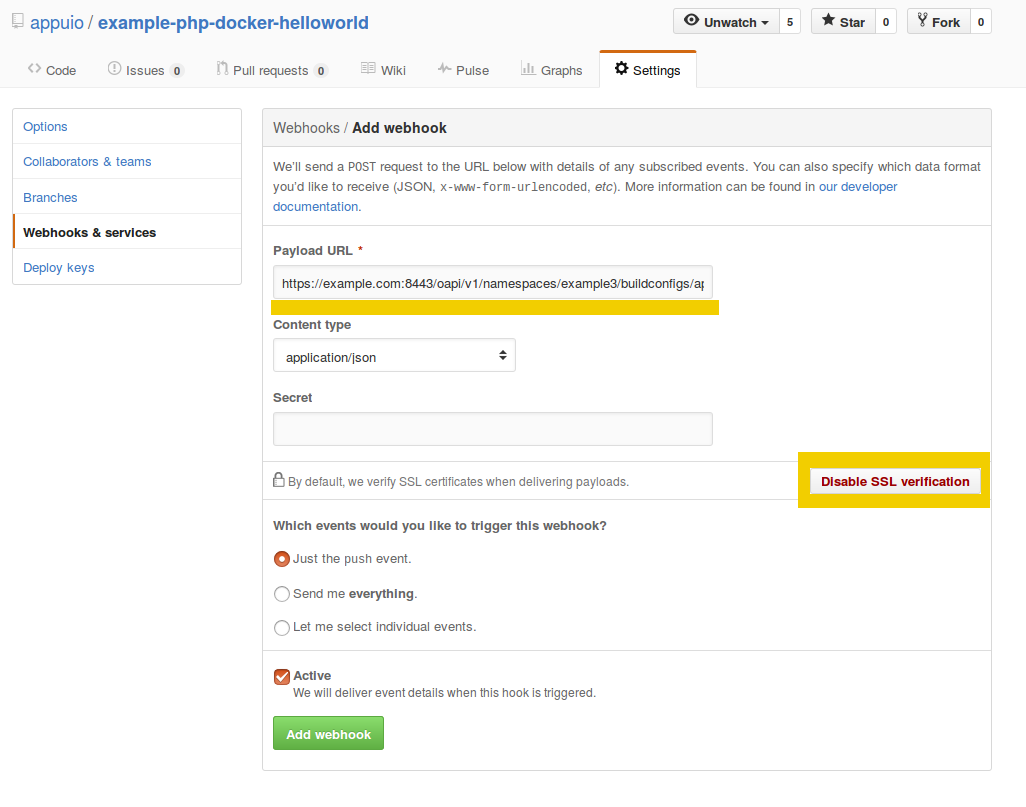
[](https://github.com/appuio/techlab/blob/lab-3.3/images/lab_9_webhook_ose3.png)

Kopieren Sie die GitHub [Webhook](https://developer.github.com/webhooks/) URL und fügen Sie sie auf GitHub entsprechend ein.

Klicken Sie in Ihrem Projekt auf Settings: [](https://github.com/appuio/techlab/blob/lab-3.3/images/lab_09_webhook_github1.png)

Klicken Sie auf Webhooks & services: [](https://github.com/appuio/techlab/blob/lab-3.3/images/lab_09_webhook_github2.png)

Fügen Sie einen Webhook hinzu: [](https://github.com/appuio/techlab/blob/lab-3.3/images/lab_09_webhook_github3.png)

Fügen Sie die entsprechende GitHub Webhook URL aus Ihrem OpenShift Projekt ein und "disablen" Sie die SSL verification. Auf der Lab Plattform verfügen wir nur über self-signed Zertifikate. [](https://github.com/appuio/techlab/blob/lab-3.3/images/lab_09_webhook_github4.png)

Ab jetzt triggern alle Pushes auf Ihrem GitHub Repository einen Build und deployen anschliessend die Code-Änderungen direkt auf die OpenShift-Plattform.

## Aufgabe: LAB9.3: Code anpassen

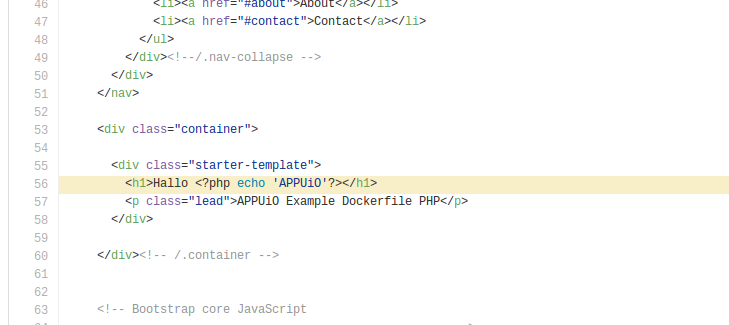
Klonen Sie Ihr Git Repository und wechseln Sie in das Code Verzeichnis:

$ git clone https://github.com/[YourGithubUser]/example-php-docker-helloworld.git

$ cd example-php-docker-helloworld

Passen Sie das File bspw. auf Zeile 56 ./app/index.php an:

$ vim app/index.php

[](https://github.com/appuio/techlab/blob/lab-3.3/images/lab_9_codechange1.png)

<div class="container">

<div class="starter-template">

<h1>Hallo <?php echo 'OpenShift Techlab'?></h1>

<p class="lead">APPUiO Example Dockerfile PHP</p>

</div>

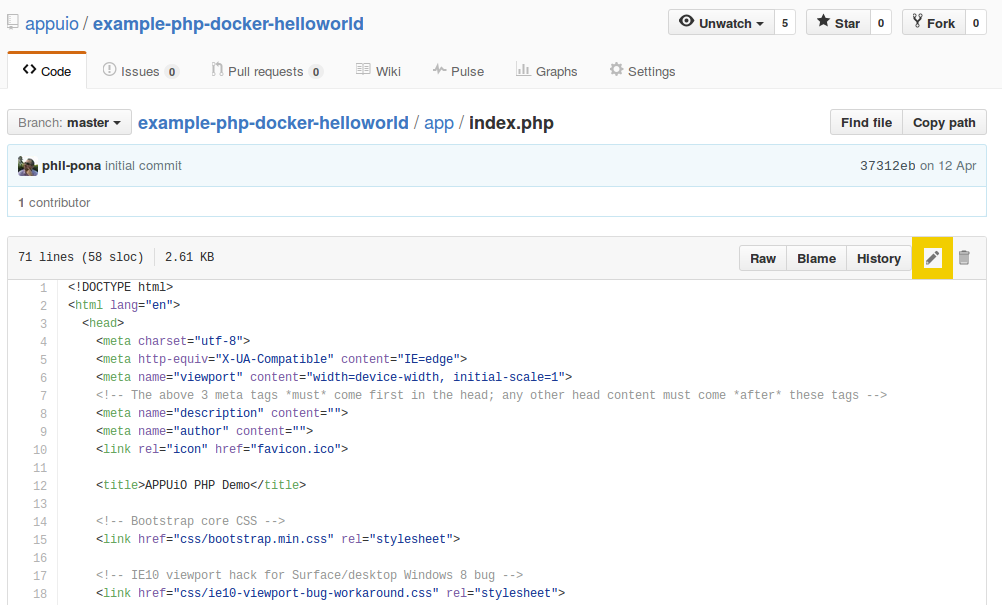
</div>

Pushen Sie Ihren Change:

$ git add .

$ git commit -m "updated Hello"

$ git push

Als Alternative können Sie das File auch direkt auf GitHub editieren: [](https://github.com/appuio/techlab/blob/lab-3.3/images/lab_9_edit_on_github.png)

Sobald Sie die Änderungen gepushed haben, startet OpenShift einen Build des neuen Source Code

$ oc get builds

und deployed anschliessend die Änderung.

## Aufgabe: LAB9.4: Rollback

Mit OpenShift lassen sich unterschiedliche Software-Stände aktivieren und deaktivieren, indem einfach eine andere Version des Images gestartet wird.

Dafür werden die Befehle oc rollback und oc deploy verwendet.

Um ein Rollback auszuführen, brauchen Sie den Namen der DeploymentConfig:

$ oc get dc

NAME TRIGGERS LATEST

appuio-php-docker-ex ConfigChange, ImageChange 2

Mit dem folgenden Befehl können Sie nun ein Rollback auf die Vorgänger-Version ausführen:

$ oc rollback appuio-php-docker-ex

#3 rolled back to appuio-php-docker-ex-1

Warning: the following images triggers were disabled: appuio-php-docker-ex

You can re-enable them with: oc deploy appuio-php-docker-ex --enable-triggers -n phptest

Sobald das Deployment der alten Version erfolgt ist, können Sie über ihren Browser überprüfen, ob wieder die ursprüngliche Überschrift **Hello APPUiO** angezeigt wird.

**Tipp:** Die automatischen Deployments neuer Versionen ist nun für diese Applikation ausgeschaltet um ungewollte Änderungen nach dem Rollback zu verhindern. Um das automatische Deployment wieder einzuschalten führen Sie den folgenden Befehl aus:

$ oc deploy appuio-php-docker-ex --enable-triggers

**Ende Lab 9**

# Lab 10: Persistent Storage anbinden und verwenden für Datenbank

Per se sind Daten in einem Pod nicht persistent, was u.a. auch in unserem Beispiel der Fall ist. Verschwindet also unser MySQL-Pod bspw. aufgrund einer Änderung des Images, sind die bis zuvor noch vorhandenen Daten im neuen Pod nicht mehr vorhanden. Um genau dies zu verhindern hängen wir nun Persistent Storage an unseren MySQL-Pod an.

## Aufgabe: LAB10.1:

### Storage anfordern

Das Anhängen von Persistent Storage geschieht eigentlich in zwei Schritten. Der erste Schritt beinhaltet als erstes das Erstellen eines sog. PersistentVolumeClaim für unser Projekt. Im Claim definieren wir u.a. dessen Namen sowie Grösse, also wie viel persistenten Speicher wir überhaupt haben wollen.

Der PersistentVolumeClaim stellt allerdings erst den Request dar, nicht aber die Ressource selbst. Er wird deshalb automatisch durch OpenShift mit einem zur Verfügung stehenden Persistent Volume verbunden, und zwar mit einem mit mindestens der angeforderten Grösse. Sind nur noch grössere Persistent Volumes vorhanden, wird eines dieser Volumes verwendet und die Grösse des Claims angepasst. Sind nur noch kleinere Persistent Volumes vorhanden, kann der Claim nicht verbunden werden und bleibt solange offen, bis ein Volume der passenden Grösse (oder eben grösser) auftaucht.

### Volume in Pod einbinden

Im zweiten Schritt wird der zuvor erstellte PVC im richtigen Pod eingebunden. In [LAB 6](https://github.com/appuio/techlab/blob/lab-3.3/labs/06_scale.md) bearbeiteten wir die Deployment Config, um die Readiness Probe einzufügen. Dasselbe tun wir nun für das Persistent Volume. Im Unterschied zu [LAB 6](https://github.com/appuio/techlab/blob/lab-3.3/labs/06_scale.md) können wir aber mit oc volume die Deployment Config automatisch erweitern.

Der folgende Befehl führt beide beschriebenen Schritte zugleich aus, er erstellt also zuerst den Claim und bindet ihn anschliessend auch als Volume im Pod ein:

$ oc volume dc/mysql --add --name=mysql-data --type persistentVolumeClaim \

--claim-name=mysqlpvc --claim-size=256Mi --overwrite

**Note:** Durch die veränderte Deployment Config deployt OpenShift automatisch einen neuen Pod. D.h. leider auch, dass das vorher erstellte DB-Schema und bereits eingefügte Daten verloren gegangen sind.

Unsere Applikation erstellt beim Starten das DB Schema eigenständig.

**Tipp:** redeployen Sie den Applikations-Pod:

$ oc deploy example-spring-boot --latest

Mit dem Befehl oc get persistentvolumeclaim, oder etwas einfacher oc get pvc, können wir uns nun den im Projekt frisch erstellten PersistentVolumeClaim anzeigen lassen:

$ oc get pvc

NAME STATUS VOLUME CAPACITY ACCESSMODES AGE

mysqlpvc Bound pv34 256Mi RWO,RWX 14s

Die beiden Attribute Status und Volume zeigen uns an, dass unser Claim mit dem Persistent Volume pv34 verbunden wurde.

Mit dem folgenden Befehl können wir auch noch überprüfen, ob das Einbinden des Volumes in die Deployment Config geklappt hat:

$ oc volume dc/mysql

deploymentconfigs/mysql

pvc/mysqlpvc (allocated 256MiB) as mysql-data

## Aufgabe: LAB10.2: Persistenz-Test

### Daten wiederherstellen

Wiederholen Sie [Lab-Aufgabe 8.4](https://github.com/appuio/techlab/blob/lab-3.3/labs/08_database.md#l%C3%B6sung-lab84).

### Test

Skalieren Sie nun den mysql Pod auf 0 und anschliessend wieder auf 1. Beobachten Sie, dass der neue Pod die Daten nicht mehr verliert.

**Ende Lab 10**

# Lab 11: Applikationstemplates

In diesem Lab zeigen wir auf, wie Templates ganze Infrastrukturen beschreiben und entsprechend mit einem Befehl instanziert werden können.

## Templates

Wie Sie in den vorangegangenen Labs gesehen haben, können einfach über die Eingabe unterschiedlicher Befehle Applikationen, Datenbanken, Services und deren Konfiguration erstellt und deployt werden.

Dies ist fehleranfällig und eignet sich schlecht zum Automatisieren.

OpenShift bietet dafür das Konzept von Templates, in welchen man eine Liste von Ressourcen beschreiben kann, die parametrisiert werden können. Sie sind also quasi ein Rezept für eine ganze Infrastruktur (bspw. 3 ApplikationsContainer, eine Datenbank mit Persistent Storage)

**Note:** der Clusteradmin kann globale Templates erstellen, welche allen Usern zur Verfügung stehen.

Alle vorhandenen Templates anzeigen

$ oc get template -n openshift

Über die Web Console kann dies via "Add to Project" gemacht werden, über diese Funktionalität können Templates direkt instanziert werden.

Diese Templates können im Json Format sowohl im Git Repositors neben ihrem Source Code abgelegt werden wie auch über eine URL aufgerufen oder gar lokal im Filesystem abgelegt sein.

## Aufgabe: LAB11.1: Template instanzieren.

Die einzelnen Schritte die wir in den vorherigen Labs manuell vorgenommen haben, können nun mittels Template in einem "Rutsch" durchgeführt werden.

$ oc new-project [USER]-template

Template erstellen

$ oc create -f https://raw.githubusercontent.com/appuio/example-spring-boot-helloworld/master/example-spring-boot-template.json

Template instanzieren (Ersetzen Sie [project] mit [USER]-template)

$ oc new-app example-spring-boot

--> Deploying template example-spring-boot for "example-spring-boot"

With parameters:

APPLICATION\_DOMAIN=

MYSQL\_DATABASE\_NAME=appuio

MYSQL\_USER=appuio

MYSQL\_PASSWORD=appuio

MYSQL\_DATASOURCE=jdbc:mysql://mysql/appuio?autoReconnect=true

MYSQL\_DRIVER=com.mysql.jdbc.Driver

--> Creating resources ...

imagestream "example-spring-boot" created

deploymentconfig "example-spring-boot" created

deploymentconfig "mysql" created

route "example-spring-boot" created

service "example-spring-boot" created

service "mysql" created

--> Success

Run 'oc status' to view your app.

Mittels:

oc deploy example-spring-boot --latest

startet OpenShift danach einen Build und deployt die Container wie im Template spezifiziert.

**Tipp:** Sie könnten Templates auch direkt verarbeiten in dem Sie ein Template direkt $ oc new-app -f template.json -p Param = value aufrufen

Als Abschluss dieses Labs können Sie sich noch das Template anschauen

https://github.com/appuio/example-spring-boot-helloworld/blob/master/example-spring-boot-template.json

**Note:** Bestehende Ressourcen können als Template exportiert werden, verwenden Sie dafür den oc export [ResourceType] --as-myapptemplate Command. Bspw.

oc export is,bc,dc,route,service --as-template=example-spring-boot -o json > example-spring-boot-template.json

Wichtig ist, dass die Imagestreams zuoberst im Template File definiert sind. Ansonsten wird der erste Build nicht funktionieren.

**Ende Lab 11**

# Lab 12: Eigene Templates erstellen

Im Unterschied zum [Lab 11](https://github.com/appuio/techlab/blob/lab-3.3/labs/11_template.md) schreiben/definieren wir hier unsere eigenen Templates bevor wir damit Applikationen erstellen.

## Hilfreiche oc client Befehle

Auflisten aller Befehle:

$ oc help

Konzepte und Typen:

$ oc types

Übersicht aller Ressourcen:

$ oc get all

Infos zu einer Ressource:

$ oc get <RESOURCE\_TYPE> <RESOURCE\_NAME>

$ oc describe <RESOURCE\_TYPE> <RESOURCE\_NAME>

## Generierung

Über "oc new-app" oder "Add to project" in der Web Konsole werden die Ressourcen automatisch angelegt. In der Web Konsole kann die Erstellung einfach konfiguriert werden.

Für den produktiven Einsatz reicht das meistens nicht aus. Da braucht es mehr Kontrolle über die Konfiguration. Eigene Templates sind hierfür die Lösung. Sie müssen jedoch nicht von Hand geschrieben sondern können als Vorlage generiert werden.

### Generierung vor Erstellung

Mit **oc new-app** parst OpenShift die gegebenen Images, Templates, Source Code Repositories usw. und erstellt die Definition der verschiedenen Ressourcen. Mit der Option **-o** erhält man die Definition ohne dass die Ressourcen angelegt werden.

So sieht die Definition vom hello-world Image aus.

$ oc new-app hello-world -o json

Spannend ist auch zu beobachten, was OpenShift aus einem eigenen Projekt macht. Hierfür kann ein Git Repository oder ein lokaler Pfad des Rechners angeben werden.

Beispiel-Befehl, wenn man sich im Root-Verzeichnis des Projekts befindet:

$ oc new-app . -o json

Wenn verschiede ImageStreams in Frage kommen oder keiner gefunden wurde, muss er spezifiziert werden:

$ oc new-app . --image-stream=wildfly:latest -o json

oc new-app erstellt immer eine Liste von Ressourcen. Bei Bedarf kann eine solche mit [jq](https://stedolan.github.io/jq/) in ein Template konvertiert werden:

$ oc new-app . --image-stream=wildfly:latest -o json | \

jq '{ kind: "Template", apiVersion: .apiVersion, metadata: {name: "mytemplate" }, objects: .items }'

### Generierung nach Erstellung

Bestehende Ressourcen werden mit **oc export** exportiert.

$ oc export route my-route

Welche Ressourcen braucht es?

Für ein vollständiges Template sind folgende Ressourcen notwendig:

* Image Streams
* Build Configurations
* Deployment Configurations
* Persistent Volume Claims
* Routes
* Services

Beispiel-Befehl um einen Export der wichtigsten Ressourcen als Template zu generieren:

$ oc export is,bc,pvc,dc,route,service --as-template=my-template -o json > my-template.json

Ohne die Option *--as-template* würde eine Liste von Items anstatt eines Templates mit Objects exportiert.

Momentan gibt es einen offenen [Issue](https://github.com/openshift/origin/issues/8327) welcher zur Folge hat, dass ImageStreams nach dem re-importieren nicht mehr korrekt funktionieren. Als Workaround kann das Attribut .spec.dockerImageRepository, falls vorhanden, mit dem Wert des Attributs .tags[0].annotations["openshift.io/imported-from"] ersetzt werden. Mit [jq](https://stedolan.github.io/jq/) kann dies gleich automatisch erledigt werden:

$ oc export is,bc,pvc,dc,route,service --as-template=my-template -o json |

jq '(.objects[] | select(.kind == "ImageStream") | .spec) |= \

(.dockerImageRepository = .tags[0].annotations["openshift.io/imported-from"])' > my-template.json

Attribute mit Wert null sowie die Annotation openshift.io/generated-by dürfen aus dem Template entfernt werden.

### Vorhandene Templates exportieren

Es können auch bestehende Templates der Plattform abgeholt werden um eigene Templates zu erstellen.

Verfügbare Templates sind im OpenShift Namespace hinterlegt. So werden alle Templates aufgelistet:

$ oc get templates -n openshift

So erhalten wir eine Kopie vom eap70-mysql-persistent-s2i Template:

$ oc export template eap70-mysql-persistent-s2i -o json -n openshift > eap70-mysql-persistent-s2i.json

## Parameter

Damit die Applikationen für die eigenen Bedürfnisse angepasst werden können, gibt es Parameter. Generierte oder exportierte Templates sollten fixe Werte, wie Hostnamen oder Passwörter, durch Parameter ersetzen.

### Parameter von Templates anzeigen

Mit **oc process --parameters** werden die Parameter eines Templates angezeigt. Hier wollen wir sehen, welche Paramter im CakePHP MySQL Template definiert sind:

$ oc process --parameters cakephp-mysql-example -n openshift

NAME DESCRIPTION GENERATOR VALUE

NAME The name assigned to all of the frontend objects defined in this template. cakephp-mysql-example

NAMESPACE The OpenShift Namespace where the ImageStream resides. openshift

MEMORY\_LIMIT Maximum amount of memory the CakePHP container can use. 512Mi

MEMORY\_MYSQL\_LIMIT Maximum amount of memory the MySQL container can use. 512Mi

...

### Parameter von Templates mit Werten ersetzen

Für die Erzeugung der Applikationen können gewünschte Parameter mit Werten ersetzt werden. Dazu **oc process** verwenden:

oc process -f eap70-mysql-persistent-s2i.json \

-v PARAM1=value1,PARAM2=value2 > processed-template.json

So werden Parameter vom Template mit den gegebenen Werten ersetzt und in eine neue Datei geschrieben. Diese Datei wird eine Liste von Resources/Items sein, welche mit **oc create** erstellt werden können:

oc create -f processed-template.json

Dies kann auch in einem Schritt erledigt werden:

oc process -f eap70-mysql-persistent-s2i.json \

-v PARAM1=value1,PARAM2=value2 \

| oc create -f –

## Templates schreiben

OpenShift Dokumentation:

* [Template Konzept](https://docs.openshift.com/container-platform/3.3/architecture/core_concepts/templates.html)
* [Templates schreiben](https://docs.openshift.com/container-platform/3.3/dev_guide/templates.html)

Applikationen sollten so gebaut werden, dass sich pro Umgebung nur ein paar Konfigurationen unterscheiden. Diese Werte werden im Template als Parameter definiert. Somit ist der erste Schritt nach dem Generieren einer Template-Definition das Definieren von Parametern. Das Template wird mit Variablen erweitert, welche dann mit den Parameterwerten ersetzt werden. So wird die Variable ${DB\_PASSWORD} durch den Parameter mit Namen DB\_PASSWORD ersetzt.

### Generierte Parameter

Oft werden Passwörter automatisch generiert, da der Wert nur im OpenShift Projekt verwendet wird. Dies kann mit einer Generate Definition erreicht werden.

parameters:

- name: DB\_PASSWORD

description: "DB connection password"

generate: expression

from: "[a-zA-Z0-9]{13}"

Diese Definition würde ein zufälliges, 13 Zeichen langes Passwort mit Klein- und Grossbuchstaben sowie Zahlen generieren.

Auch wenn ein Parameter mit Generate Definition konfiguriert ist, kann er bei der Erzeugung überschrieben werden.

### Template Merge

Wenn z.B eine App mit einer Datenbank zusammen verwendet wird, können die zwei Templates zusammengelegt werden. Dabei ist es wichtig, die Template Parameter zu konsolidieren. Dies sind meistens Werte für die Anbindung der Datenbank. Dabei einfach in beiden Templates die gleiche Variable vom gemeinsamen Parameter verwenden.

## Anwenden vom Templates

Templates können mit oc new-app -f <FILE>|<URL> -p <PARAM1>=<VALUE1>,<PARAM2>=<VALUE2>... instanziert werden. Wenn die Parameter des Templates bereits mit oc process gesetzt wurden, braucht es die Angabe der Parameter nicht mehr.

## Metadata / Labels

oc new-app fügt standardmässig das Label app=<TEMPLATE NAME> in alle instanzierten Ressourcen ein. Bei einigen OpenShift-Versionen kann dies zu [ungültigen](https://github.com/openshift/origin/issues/10782) Ressourcendefinitionen führen. Als Workaround kann mit oc new-app -l <LABEL NAME>=<LABEL VALUE> ... ein alternatives Label konfiguriert werden.

## Ressourcen aus docker-compose.yml erstellen

Seit Version 3.3 bietet die OpenShift Container Platform die Möglichkeit, Ressourcen aus der Docker Compose Konfigurationdatei docker-compose.yml zu erstellen. Diese Funktionalität ist noch als experimentell eingestuft. Beispiel:

git clone -b techlab https://github.com/appuio/weblate-docker#techlab

oc import docker-compose -f docker-compose.yml -o json

Die Möglichkeit eine Datei direkt via URL zu importieren ist vorgesehen aber noch nicht implementiert. Durch Weglassen der Option -o json werden die Ressourcen direkt angelegt statt ausgegeben. Momentan werden Services für bereits vorhandene Docker Images nur angelegt, falls eine explizite Portkonfiguration in docker-compose.yml vorhanden ist. Diese können in der Zwischenzeit mithilfe von oc new-app angelegt werden:

oc new-app --name=database postgres:9.4 -o json|jq '.items[] | select(.kind == "Service")' | oc create -f -

oc new-app --name=cache memcached:1.4 -o json|jq '.items[] | select(.kind == "Service")'|oc create -f -

**Ende Lab 12**