

Modul 347 3.2.2023 - 31.3.2023 v1.0.0

Was sind Container und wofür verwendet man sie überhaupt?

Ein Container ermöglicht es uns, Software in einer isolierten Umgebung laufen zu lassen. Die ist vor allem praktisch, da solche Docker Container standardisiert sind und ich somit auch einen Container eines Kollegen aus Japan ganz einfach auf meinem Rechner laufen lassen kann, ohne dass ich die ganze Umgebung nachbauen muss. Dies ist vor allem sehr praktisch, wenn ich an einer Software arbeite, welche aus mehreren Services besteht. Ich kann z. B. mein Backend und meine DB in einem Container laufen lassen, ohne dass ich dafür die ganze Umgebung einrichten muss. Somit muss ich nur meine Container starten und kann mein Frontend entwickeln.

Was ist überhaupt DevOps?

Wikipedia sagt folgendes über DevOps: "DevOps ist eine Sammlung unterschiedlicher technischer Methoden und eine Kultur zur Zusammenarbeit zwischen Softwareentwicklung und IT-Betrieb". DevOps setzt sich aus **Development** und **Operations** zusammen, was auf Deutsch so viel wie Entwicklung und Betrieb bedeutet. DevOps soll die Qualität und Geschwindigkeit der Softwareentwicklung erhöhen, indem sogenannte DevOps-Teams genutzt werden, welche aus Entwicklern und IT-Operatoren bestehen. Damit können beide Seiten eng zusammenarbeiten und auf die Aspekte der anderen Seite eingehen und wenn nötig Kompromisse finden, welche für beide Seiten akzeptabel sind.

Virtualisierung vs Containerisierung

Virtualisierung und Containerisierung werden manchmal als dasselbe angesehen, da es auf den ersten Blick sehr ähnlich aussieht. Dies ist jedoch nicht der Fall. Bei der Virtualisierung läuft eine App auf Ihrem eigenen Betriebssystem und teilt sich eigentlich nichts Gemeinsames mit dem Rechner, auf dem sie läuft, außer die Hardware. Bei der Containerisierung ist die anders. Eine App läuft zwar in Ihrem eigenen "Bereich", greift trotzdem auf den Kernel des Betriebssystems des Rechners zu, auf dem sie läuft. Die Containerapp ist deshalb nur auf Betriebssystemebene isoliert. Dies macht eine Containerapp ressourcenleichter. Zudem lässt sich ein Container meistens schneller starten als eine VM, da nicht ein komplettes zusätzliches Betriebssystem benötigt wird.

Bei Docker wird für die Containerisierung standardmäßig ein Linux Kernel verwendet, jedoch hat man unter Windows auch die Möglichkeit sich für einen Windows-Container zu entscheiden.

Play with Docker

Unten ein Bild, wie ich bei Play with Docker eingeloggt bin:

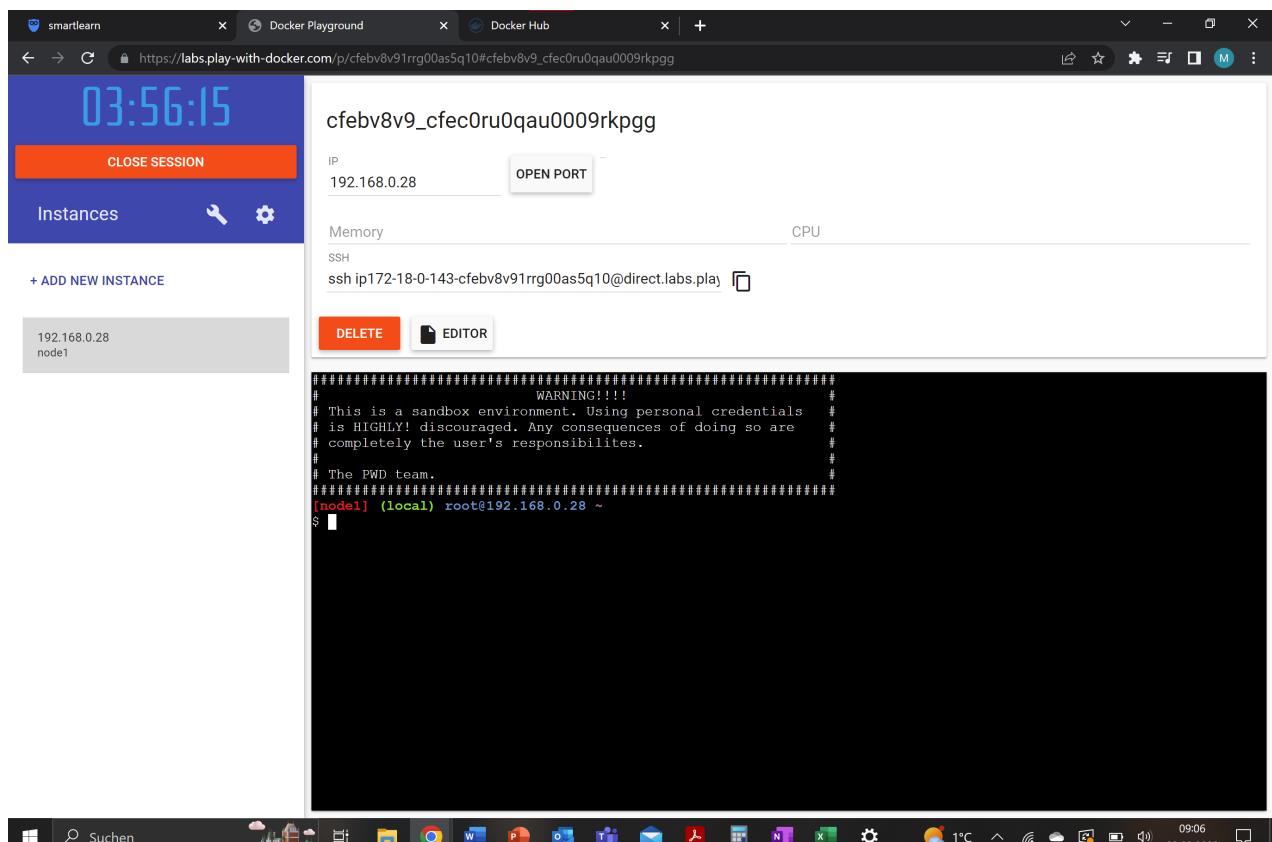


Image vs Container

Ein Image ist quasi der Bauplan für einen Container. Man kann aus einem Image beliebig viele Container starten und jeder Container wird

gleich sein. Deshalb werden auch nur die Images geteilt. Images werden in sogenannten Registries gepusht, wo man sie später wie bei Git pullen kann. Container können beliebig erstellt und zerstört werden. Während der Laufzeit kann man einen Container modifizieren und anhand des modifizierten Containers ein neues Image erstellen.

In Registries wie Dockerhub sind ganz viele öffentliche Images verfügbar, welche man für seine eigenen Images und Container verwenden kann. Natürlich hat man auch die Möglichkeit für sich selbst ein privates Image zu erstellen und dieses zu Dockerhub zu pushen.

Die wichtigsten Befehle und was sie machen

Image Befehle

`docker pull <image_name>` : Mit diesem Befehl kann man sich ein Docker Image von Dockerhub auf sein eigenes Gerät holen. Dies erleichtert das Teilen von Images erheblich. "image_name" ersetzt man durch den Namen des Images, welches man haben möchte. Ohne weitere Spezifikation wird standartmäßig immer die neuste Version gewählt.

`docker build` : Mithilfe dieses Befehls kann man ein Image eines Projektes erstellen. Entweder befindet man sich in im richtigen Verzeichnis oder man kann noch eine URL anhängen, dies kann auch eine Remote URL sein von z.B. GitHub.

`docker images` : Wenn dieser Befehl ausgeführt wird, werden einem die Images angezeigt, welche sich aktuell auf dem Gerät befinden. Dabei werden einem Infos wie das Repository, die Tags, die Id des Images, die Grösse und das Erstellungsdatum angezeigt. Mit der Option -a werden auch noch Zwischenimages angezeigt und mit der Option -q werden nur die Image Ids angezeigt.

`docker history <image>` : Mit diesem Befehl kann die History eines Images einsehen. Für "image" setzt man das Image ein, welches man einsehen möchte.

`docker push <image>` : Mit diesem Befehl kann man ein Image zu einem Registry pushen. Für "image" setzt man das Image ein, welches man pushen möchte.

`docker import <url>` : Damit kann man ein Image auf der Basis eines .tar-Files erstellen. Für "url" setzt man die URL oder den Dateipfad vom .tar-File ein.

`docker rmi <image>` : Mit diesem Befehl kann man ein Image löschen. Für "image" setzt man das Image ein, welches man löschen möchte.

`docker load <file>` : Damit kann man ein Image auf der Basis eines .tar-Files laden. Für "url" setzt man die URL oder den Dateipfad vom .tar-File ein.

Container Befehle

`docker run <image_name>` : Mit diesem Befehl erstellen wir einen Container auf der Basis eines Images und starten diesen. Dieses Image geben wir bei "image_name" mit. Wenn das Image nicht Lokal vorhanden ist versucht Docker dieses von Dockerhub zu pullen. Dieses wird jedoch nur gepulled wenn man die rechte dazu hat und dieses auch existiert. Auch hier gibt es wieder viele Optionen wie zum Beispiel Portmapping mit der Option -p.

`docker create <image_name>` : Mit diesem Befehl erstellen wir einen Container auf der Basis eines Images, dieses Image geben wir bei "image_name" mit. Falls das Image nicht lokal vorhanden ist, versucht Docker dieses von Dockerhub zu pullen. Dieses wird jedoch nur gepulled, wenn man die rechte dazu hat und dieses auch existiert.

`docker start <container>` : Mit diesem Befehl starten wir einen gestoppten Container. Anstelle von "container" geben wir den Container an, welchen wir starten möchten.

`docker ps` : Wenn man diesen Befehl ausführt, werden einem alle laufenden Container und deren Informationen angezeigt. Mit der Option -a kann sich alle Container anzeigen, also auch die gestoppten.

`docker rm <container>` : Mit diesem Befehl kann man einen gestoppten Container löschen. Für "container" setzt man den Container ein, welchen man löschen möchte.

`docker update <container>` : Mit diesem Befehl kann man die Konfiguration eines Containers aktualisieren. Für "container" setzt man den Container ein, welchen man bearbeiten möchte.

`docker stop <container>` : Mithilfe dieses Befehls stoppt man einen laufenden Container. Für "container" setzt man den Container ein, welchen man stoppen möchte.

`docker restart <container>` : Mit diesem Befehl startet man einen Container neu. Das heisst man stoppt ihn und startet ihn wieder. Für "container" setzt man den Container ein, welchen man neu starten möchte.

`docker pause <container>` : Mit diesem Befehl kann man den Prozess eines Containers pausieren. Für "container" setzt man den Container ein, welchen man pausieren möchte.

`docker unpause <container>`: Mit diesem Befehl kann man den Prozess in einem pausierten Container wieder starten. Für "container" setzt man den Container ein, welchen man wieder starten möchte.

`docker wait <container>`: Mit diesem Befehl kann man einen Container blockieren, bis ein andere stoppt. Für "container" setzt man den Container ein, welchen man blockieren möchte.

`docker kill <container>`: Killt einen Container, anstatt ihn auf herkömmliche Weise zu stoppen. Für "container" setzt man den Container ein, welchen man killen möchte.

Netzwerk Befehle

`docker network ls`: Zeigt einem alle Netzwerke an.

`docker network create <name>`: Mit diesem Befehl kann man ein neues Netzwerk erstellen. An der Stelle von "name" gibt man den Namen des neuen Netzwerkes ein.

`docker network rm <network>`: Mit diesem Befehl kann man ein Netzwerk löschen. Für "network" setzt man das Netzwerk ein, welches man löschen möchte.

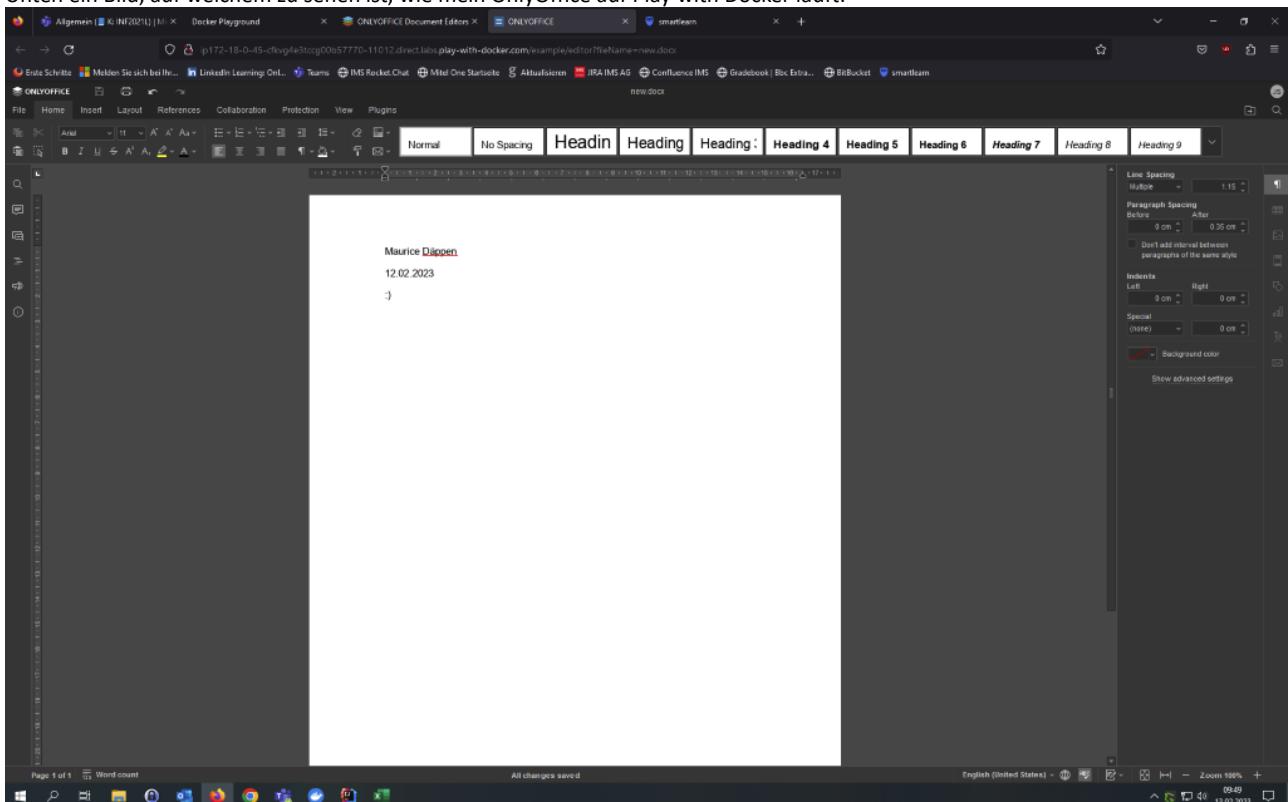
`docker network inspect <network>`: Mit diesem Befehl kann man sich Information zu einem gewünschten Netzwerk anzeigen lassen. Für "network" setzt man das Netzwerk ein, welches man einsehen möchte.

`docker network connect <network> <container>`: Mit diesem Befehl kann man einen Container mit einem Netzwerk verbinden. Für "network" setzt man das Netzwerk ein, welches man verwenden möchte. Für "container" setzt man den Container ein, welchen man verwenden möchte.

`docker network disconnect <network> <container>`: Mit diesem Befehl kann man einen Container, welcher mit einem Netzwerk verbunden ist, davon trennen. Für "network" setzt man das Netzwerk ein, welches man verwenden möchte. Für "container" setzt man den Container ein, welchen man verwenden möchte.

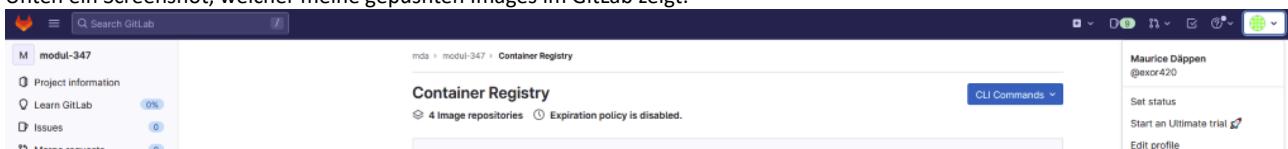
OnlyOffice

Unten ein Bild, auf welchem zu sehen ist, wie mein OnlyOffice auf Play with Docker läuft:



Ein Image pushen

Unten ein Screenshot, welcher meine gepushten Images im GitLab zeigt:



Die wichtigsten Befehle um ein Image zu GitLab zu pushen

`docker login <url>`: Mit diesem Befehl hinterlegen wir unsere Anmeldedaten für eine bestimmte Plattform. Falls keine URL mitgegeben wird, meldet man sich bei Dockerhub an. Da wir zu GitLab pushen möchten, geben wir als URL "registry.gitlab.com" an.

`docker image tag <image> <target_image>`: Mit diesem Befehl kann man ein Image taggen. Dafür setzt man für "image" ein, welches getaggt werden soll und für "target_image" den Namen des Tags. Wichtig ist jedoch, dass nicht das "image" getaggt wird, sondern ein neues mit dem gewählten Tag erstellt wird.

`docker image push <image>`: Mit diesem Befehl kann man ein Image zu einem Registry pushen. Für "image" setzt man das Image ein, welches man pushen möchte.

Ein Image pushen v2

Unten ein Screenshot, welcher meine gepushten Images im GitLab zeigt:

Was ist Docker Compose?

Mit Docker-Compose hat man die Möglichkeit mehrere Container mit nur einem Befehl zu starten, die Container können auch sehr einfach miteinander kommunizieren. Das heisst wir müssen nun nicht mehr immer DB-Container, Backend-Container und Frontend-Container einzeln starten und diese noch miteinander vernetzen. Mit Docker Compose können viel Zeit sparen und auch Fehler vermeiden, da man nur einmal ein Docker-Compose-File schreiben muss. Ein Docker-Compose File wird in YML oder auch YAML geschrieben. Zudem gibt es 3 verschiedene Versionen eines Docker Compose Files. Nämlich 1,2 & 3. Diese unterscheiden sich leicht von der Syntax her. Deshalb gibt man zuoberst im Docker-Compose File die Version an.

Todo-App v2 mit Docker Compose

Als Erstes habe ich mir die erste Version der App mit `git clone https://gitlab.com/thomas-staub/cloudmodules`

`/m169/demobeispiele/to-do-appv1.git` geklont. Danach habe ich noch die zweite Version des Frontends mit `git clone https://gitlab.com/thomas-staub/cloudmodules/m169/demobeispiele/to-do-appv2.git` geklont. Nun habe ich den Ordner "web-frontend" von der Version 1 durch den Ordner "web-frontendv2" von der Version 2 ersetzt, da sich an den Redis Diensten nichts ändert. Zum Schluss habe ich noch "docker-compose.yml" von der Version 1 bearbeitet. Da sich nicht viel verändert hat, musste ich nur auf der Zeile 4 "web-frontend" durch "web-frontendv2" ersetzt.

Nun müssen wir nur in der Kommandozeile nur noch in den Ordner mit der "docker-compose.yml"-Datei navigieren und können das Ganze mit `docker-compose -f docker-compose.yml up -d` starten.

Meine neue "docker-compose.yml"-Datei sieht nun folgendermassen aus:

```
version: "3"
services:
  todoapp:
    build: ./web-frontendv2
    ports:
      - "3000"
    depends_on:
      - redis-master
      - redis-slave
  redis-slave:
    build: ./redis-slave
    depends_on:
      - redis-master
  redis-master:
    build: ./redis-master
```



docker-
compose

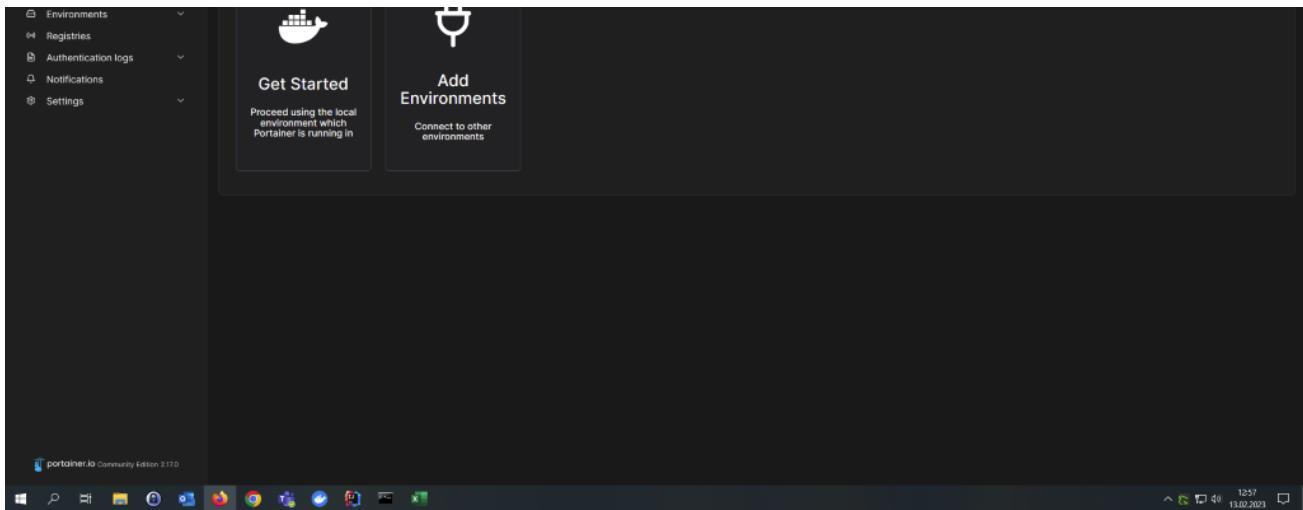
Portainer

Als Erstes zwei Bilder, welche zeigen, dass ich Portainer auf Play with Docker zum Laufen gebracht habe:

03:08:26
CLOSE SESSION
Instances
+ ADD NEW INSTANCE
192.168.0.13 OPEN PORT
Memory CPU
SSH ssh ip172-18-0-100-cfl1ime3tccg00b57d10@direct.labs.play-with-docker.com
DELETE EDITOR
node1
[node1] (Local) root@192.168.0.13 ~
\$ unzip portainer.zip
Archive: portainer.zip
 creating: portainer/
 inflating: portainer/docker-compose.yml
[node1] (Local) root@192.168.0.13 ~
\$ cd portainer/
[node1] (Local) root@192.168.0.13 ~/portainer
\$ docker compose up -d
[+] Running 5/5
 0 Services pulled
 H 722227786281 Pull complete
 H 96ff3b3bfc87 Pull complete
 H 44ea29b97e45 Pull complete
 H c0aecd363176 Pull complete
[+] Running 0/0
 0 Networks portainer_default Created
 0 Containers portainer Started
[node1] (Local) root@192.168.0.13 ~/portainer
\$ docker ps
CONTAINER ID IMAGE CREATED STATUS PORTS NAMES
44ea29b97e45 portainer/portainer-ce:latest "/portainer" 10 seconds ago Up 7 seconds 8000/tcp, 9443/tcp, 0.0.0.0:9000->9000/tcp portainer
ef7fb8b07580 v2_todoapp ./todo-app" 15 minutes ago Up 15 minutes 0.0.0.0:49151->3000/tcp v2-todoapp-1
fah025931973 v2_redis-slave "docker-entrypoint.s..." 15 minutes ago Up 15 minutes 6375/tcp v2-redis-slave-1
bc0ec32dd913 v2_redis-master "docker-entrypoint.s..." 15 minutes ago Up 15 minutes 6375/tcp v2-redis-master-1
[node1] (Local) root@192.168.0.13 ~/portainer
\$

Upgrade to Business Edition Environment Wizard
portainer.io COMMUNITY EDITION Quick Setup
Environment Wizard
Welcome to Portainer
We have connected your local environment of Docker to Portainer.
Get started below with your local portainer or connect more container environments.

Home
Settings
Users



Hier noch ein Bild welches zeigt wie ich die Todo-App über Portainer installiert habe:

This screenshot shows the 'Create stack' page in Portainer. The left sidebar is visible with 'Stacks' selected. The main area shows a 'Create stack' form with the name 'todo-via-portainer'. Under 'Build method', 'Web editor' is selected. The 'Web editor' section contains a code editor with a Docker Compose file:

```

version: "3"
services:
  todooapp:
    image: registry.gitlab.com/thomas-staub/cloudmodules/m169/demobeispiele/to-do-appv2/todo-app:v2
    ports:
      - "3000"
    depends_on:
      - redis-master
      - redis-slave
    redis-slave:
      image: registry.gitlab.com/thomas-staub/cloudmodules/m169/demobeispiele/to-do-appv1/redis-slave:v1
    depends_on:
      - redis-master
    redis-master:
      image: registry.gitlab.com/thomas-staub/cloudmodules/m169/demobeispiele/to-do-appv1/redis-master:v1
  
```

At the bottom, there are buttons for 'Webhooks' and 'Create a Stack webhook'.

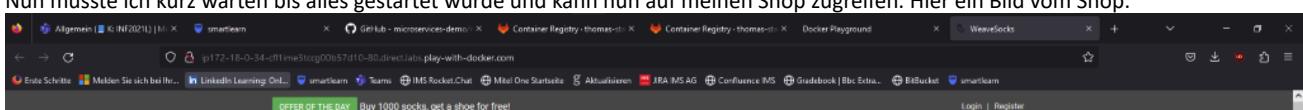
Und hier noch ein Bild, welches die fertige Installation zeigt:

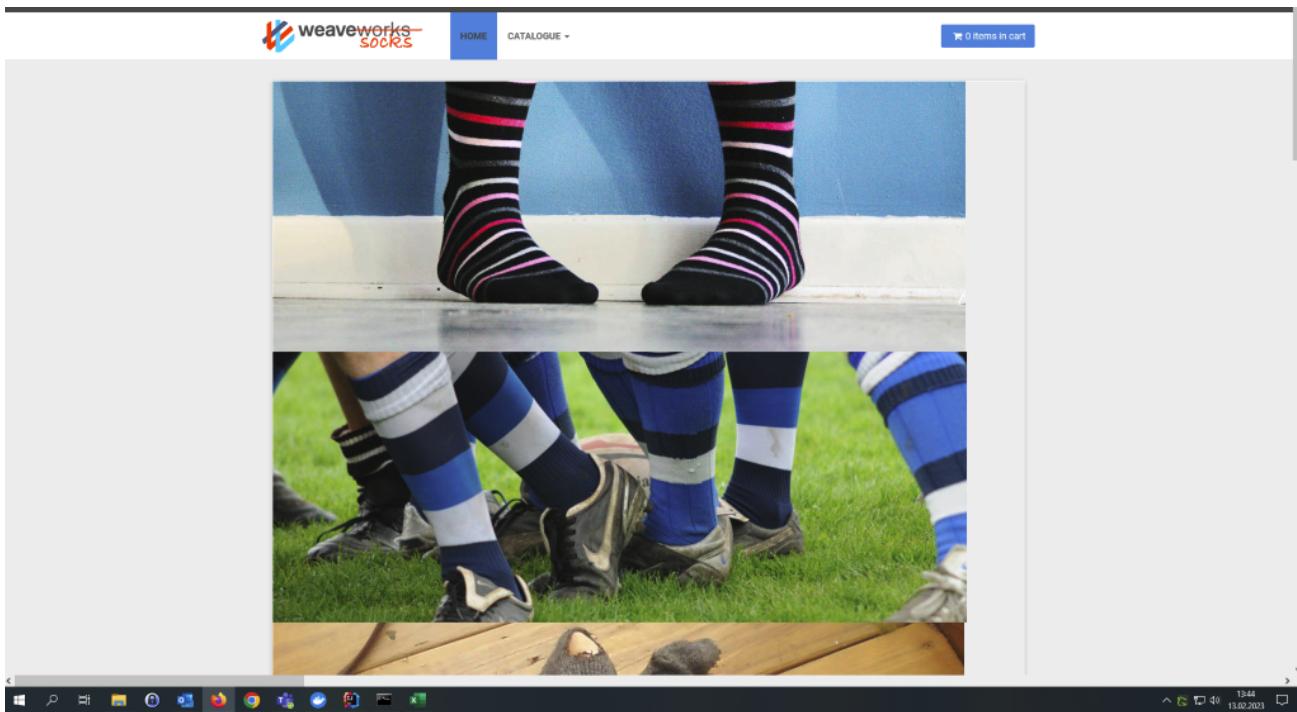
Name	State	Quick Actions	Stack	Image	Created	IP Address	GPUs
todo-via-portainer-redis-mast...	running	Stop Start Restart Pause Resume Remove	todo-via-portainer	registry.gitlab.com/thomas-staub/cloudmodules/m169/demobeispiele/to-do-appv1/redis-master:v1	2023-02-13 13:08:27	172.23.0.2	none
todo-via-portainer-redis-slave-1	running	Stop Start Restart Pause Resume Remove	todo-via-portainer	registry.gitlab.com/thomas-staub/cloudmodules/m169/demobeispiele/to-do-appv1/redis-slave:v1	2023-02-13 13:08:27	172.23.0.3	none
todo-via-portainer-todoapp-1	running	Stop Start Restart Pause Resume Remove	todo-via-portainer	registry.gitlab.com/thomas-staub/cloudmodules/m169/demobeispiele/to-do-appv2/todo-app:v2	2023-02-13 13:08:27	172.23.0.4	none

Sock-Shop

Da Docker Compose es einem sehr einfach macht, ist die Installation des Sock-Shops eine sehr einfache Sache. Als Erstes habe ich mir das Repo des Sock-Shops mit `git clone https://github.com/microservices-demo/microservices-demo.git` geklont. Nun habe ich mich mit `cd microservices-demo` in den geklonten Ordner begeben. Zum Schluss habe ich das Ganze mit `docker compose -f deploy/docker-compose/docker-compose.yml up -d` gestartet.

Nun musste ich kurz warten bis alles gestartet wurde und kann nun auf meinen Shop zugreifen. Hier ein Bild vom Shop:





Eigenes Projekt mit Docker Compose

Nachdem ich einige Zeit damit verschwendet habe einen Wordpress-Blog mit Inhalt zu erstellen, habe ich mich doch dazu entschieden doch meine Webseite vom Modul 293 zu dockerisieren. Als Erstes habe ich eine kleinen Dockerfile erstellt und ein Image erstellt. Dieses Image habe ich zu Gitlab gepusht. Hier ein Screenshot von meinem Image auf GitLab(https://gitlab.com/exor420/modul-347/container_registry):

Zum Schluss habe ich noch ein Docker-Compose-File erstellt. Die ganzen Files der Website inklusive Dockerfile sind hier zu finden: <https://github.com/eXor420/GibbBiVo2021/tree/main/Modul%20347/Docker-Compose%20Project> .

Eigenes Projekt installieren

Um das Ganze nun zu installieren, muss man nur 3 ganz einfach Schritte befolgen.

1. git clone <https://gitlab.com/exor420/modul-347.git>
2. cd modul-347
3. docker-compose up -d

Nun kannst du über den Port 8080 die Website besuchen.

Kubernetes

Was ist Kubernetes?

Kubernetes ist ein Opensource-Orchestrierungstool, welches im Jahr 2014 von Google herausgegeben wurde. Mittlerweile wird Kubernetes von der Cloud Native Computing Foundation verwaltet. Es ist vor allem praktisch, wenn man Microservices verwendet. Denn mit Kubernetes kann man leicht einzelne Komponenten austauschen und skalieren. Falls ich z. B. in meiner Webanwendung die Komponente des Zahlungsdienstes erweitere, kann ich die alten Zahlungsdienste langsam ersetzen und bemerke somit auch, wenn es Fehler geben sollte. In einem Satz zusammengefasst ist Kubernetes ein Tool zur Verwaltung von Containern. Wie bei Linux gibt es auch bei Kubernetes verschiedene "Distributionen" wie z. B. microk8s welches später im Portfolio noch behandelt wird.

Was sind Microservices?

Auf der einen Seite gibt es monolithische Apps und auf der anderen Seite Microservices. Monolithische Apps sind einfach ein riesiger Codehaufen, welcher alles enthält. Manchmal sind nicht einmal Front- und Backend ein unterschiedliches Projekt. So wie ich jedoch monolithische Apps schon angetroffen habe, sind einfach nur Frontend und Backend mit DB voneinander getrennt. Das heisst im Frontend ist das UI und im Backend alles wie z. B. Zahlungsservice, Archivierungsservices und verschiedenste Endpoints.

Mit Microservices lässt sich das Ganze vereinfachen. Das heisst man erstellt verschiedene Projekte, welche alle eher klein sind und sobald sie zu gross werden, spaltet man diese wieder auf. Z. B. wären im obigen Beispiel der Zahlungsservice oder der Archivierungsservice ein eigenes Projekt. Jeder dieser Services ist selbstständig und kommuniziert mithilfe von API (Programmierschnittstellen) miteinander. Die erleichtert das Deployment und die Entwicklung in Teams sehr. Falls jetzt ein Team etwas am Zahlungsservice ändert, kann diese Änderung sehr einfach released werden, da man nur einen kleinen Teil ersetzen muss und die App weiterlaufen kann.

Minikube vs Microk8s vs Kubeadm vs K3s

Minikube ermöglicht es einem, eine einzelne Kubernetes-Instanz auf dem eigenen Rechner auszuführen. Dies ist vor allem für Entwickler sehr praktisch, wenn man zuerst eine Applikation lokal testen möchte. Das hilft einem schon früh Fehler zu erkennen. Auch wenn nur die Int-Umgebung durch den Fehler unbrauchbar gemacht wird, sind Personen (die Entwickler im Team, welche dieselbe Int-Umgebung nutzen) davon betroffen. In solchen Fällen kann Minikube einem das Leben sehr einfach machen.

Microk8s ist dagegen vor allem auf Einfachheit und Komplexitätsreduktion ausgelegt. Es lässt einem auch wie Minikube eine einzelne Kubernetes-Instanz auf dem eigenen Rechner ausführen. Leider kann es nicht in eine Cluster-Umgebung integriert werden, da es eine lokale Installation ist. Was dafür ein grosser Vorteil ist, ist jedoch, dass Microk8s als Snap-Paket auf jeder Linux-Distribution verfügbar ist.

Kubeadm ist ein von der Kubernetes-Community entwickeltes Tool, um die Verwaltung von Clustern zu vereinfachen. Es automatisiert viele Bereiche der Einrichtung eines Clusters. Dazu gehört zum Beispiel die Generierung von Zertifikaten. Seit dem Juni 2016 gehört Kubeadm zum offiziellen Kubernetes-Projekt. Mittlerweile wird es auch von den meisten Cloudanbietern unterstützt.

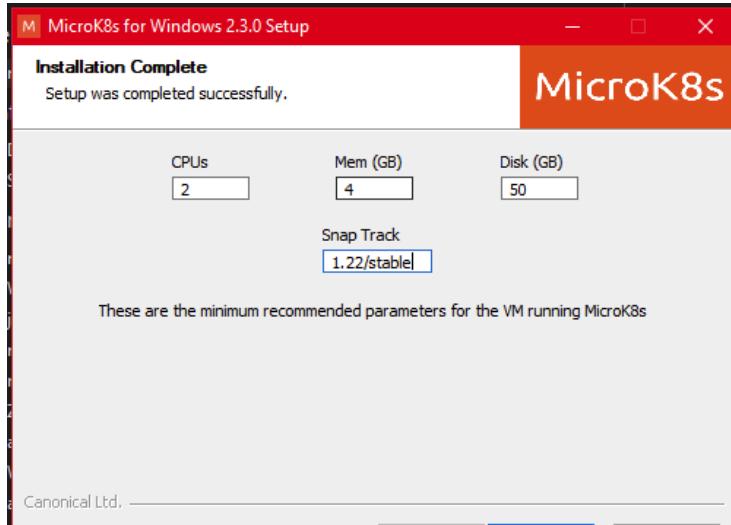
K3s steht für "Kubernetes Lightweight, Lean and Low-Resource Service". K3s ist darauf ausgelegt, sehr ressourcenschonend zu agieren. Das heisst es ist perfekt für Umgebungen, wo die Ressourcen sehr begrenzt oder ausgesprochen teuer sind. Zudem kann es in wenigen Minuten bereitgestellt werden und wird von vielen Cloudanbietern unterstützt.

Installation von Microk8s auf meinem Rechner

Hier eine kleine Anleitung, wie ich Microk8s auf meinem Windows-Rechner installiert habe:

Achtung: Auf einem Gerät welches Windows 10/11 Home verwendet müssen noch zusätzliche Schritte wie die Installation von VirtualBox vorgenommen werden!

Als Erstes laden wir uns hier <https://microk8s.io/docs/install-windows> den Installer runter. Nachdem wir den Nutzungsbedingungen zugestimmt haben, wählen wir "Add 'kubectl' to path" aus. Falls wir gefragt werden, ob wir microk8s konfigurieren wollen, wählen wir ja aus und geben folgende Werte an:



Da ich jedoch im Verlauf des Moduls auf eine Linux-VM umgestiegen bin, erstelle ich auch noch eine Installationsanleitung für die Installation von Microk8s unter Linux:

Als erstes installieren wir mit snap Microk8s:

```
sudo snap install microk8s --classic --channel=1.22/stable
```

Danach fügen wir unseren User der Microk8s-Gruppe hinzu:

```
sudo usermod -a -G microk8s exor
```

Nun erstellen wir in unserem Home-Verzeichniss einen Ordner welcher den Name ".kube" trägt und bewegen uns hinein:

```
cd $HOME  
mkdir .kube  
cd .kube
```

Nun schreiben wir die Konfiguration von Microk8s in eine Config-File:

```
microk8s config > config
```

Nun machen wir noch unseren User zum Besitzer des ".kube"-Ordners:

```
sudo chown -f -R exor ~/.kube
```

Um die Konfiguration zu beenden bearbeiten wir das Config-File und ersetzen die unter dem Punkt "server" stehende IP mit "127.0.0.1":

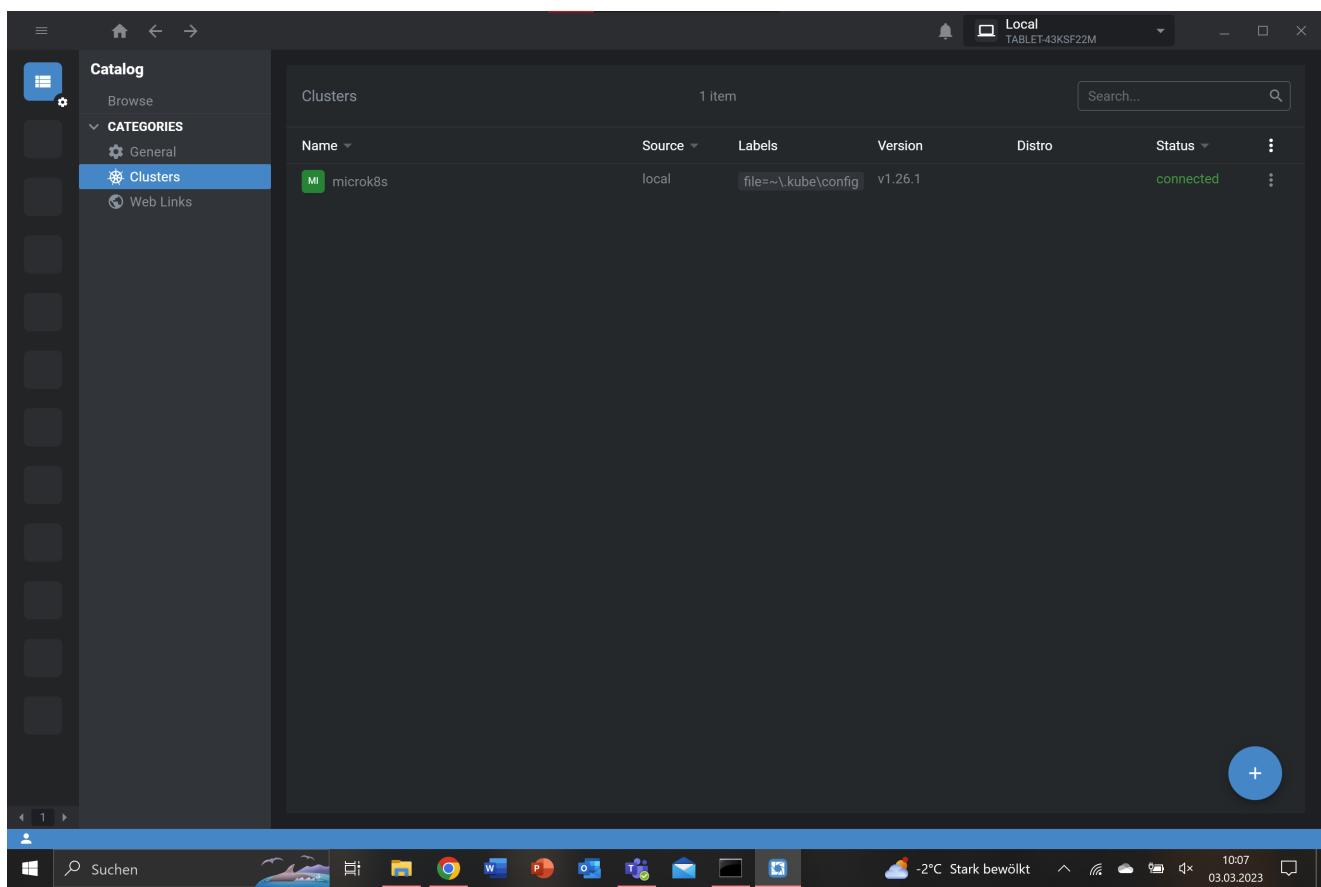
```
sudo nano config
```

Zum Schluss starten wir noch unsere VM/Rechner neu:

```
sudo reboot
```

Lens

Unten ein Bild, auf welchem zu sehen ist, wie ich mich auf mein oben installiertes microk8s verbunden habe:



Was ist der Raft-Konsens-Algorithmus?

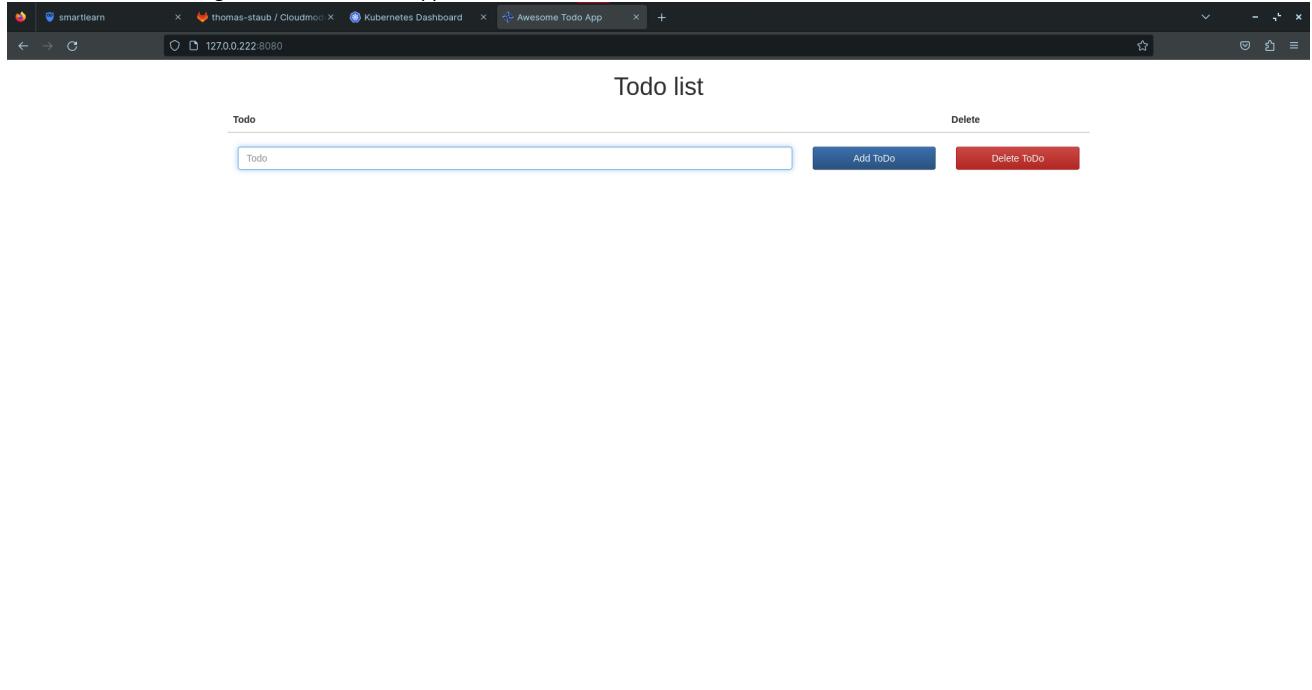
Der Raft-Konsens-Algorithmus hilft dabei, dass mehrere Nodes auf verschiedenen Rechnern miteinander arbeiten können und einen konsistenten Zustand haben. Dafür wird ein sogenannter Leader (Führer) definiert, welcher den anderen Nodes mitteilt, was sie zu tun haben und dafür zuständig ist, wie die Ressourcen verteilt sind. In Bezug auf Kubernetes sind dies die Masternodes. Die Worker-Nodes verfügen über Pods in denen Container laufen. Falls ein Worker-Node ausfällt, bemerkt der Masternode dies und versucht die verlorenen Pods auf anderen Nodes zu starten. Der Algorithmus wird dafür verwendet, dass alle Masternodes den gleichen Stand haben, damit der Cluster auch weiterlaufen kann, wenn ein Masternode ausfällt.

Warum sollte es in einem Cluster immer eine ungerade Anzahl Server geben?

Es sollte immer eine ungerade Anzahl an Nodes in einem Server haben, falls ein Masternode ausfällt und ein neuer Leader gewählt werden muss. Denn irgendwann bemerkt ein Worker-Node, dass ein Masternode fehlt. Somit stellt sich dieser zur Wahl zum Leader. Nun braucht es ein absolutes Mehr, um zum Leader zu werden. Bei 3 Node würde einer ausfallen, einer stellt sich zur Wahl und der letzte stimmt dafür. Bei 5 Nodes würde eine ausfallen, einer stellt sich zur Wahl und mindestens 2 müssten dafür stimmen. Somit ergibt es keinen Sinn, wenn es eine gerade Anzahl an Nodes hat, außer wenn mehr als einer gleichzeitig ausfällt, aber dies ist so selten der Fall, dass es sich nicht lohnt.

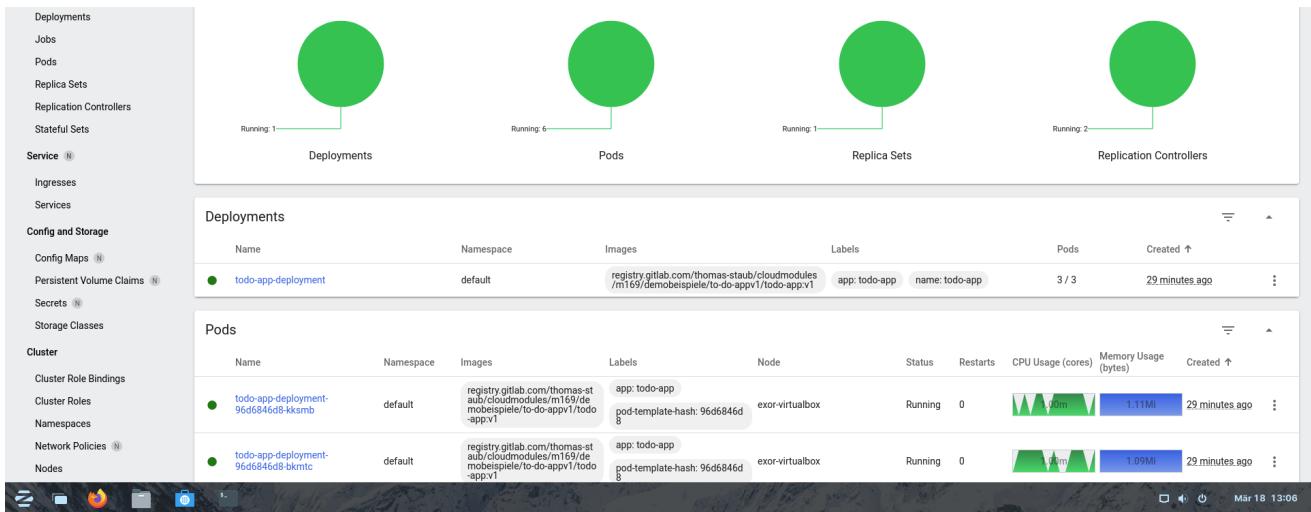
Todo-App V1

Unten Bilder, welche zeigen, dass meine Todo-App in Kubernetes läuft:



Mein Kubernetes Dashboard

Unten ein Bild, welches mein Kubernetes-Dashboard zeigt:



Was ist Self-Healing?

Self-Healing lässt sich sehr einfach erklären. Ich gebe meinem Kubernetes eine Konfiguration. Z. B. "Ich möchte 10 Pods". Wenn jetzt einer dieser Pods gelöscht wird oder sonstige Probleme hat und abstürzt, bekommt Kubernetes das mit und startet einen neuen Pod, damit meine Konfiguration weiterhin erfüllt bleibt. Das ist sehr praktisch, da ich nicht immer selbst schauen will, ob alle Pods laufen und evtl. selbst einen neuen starten muss.

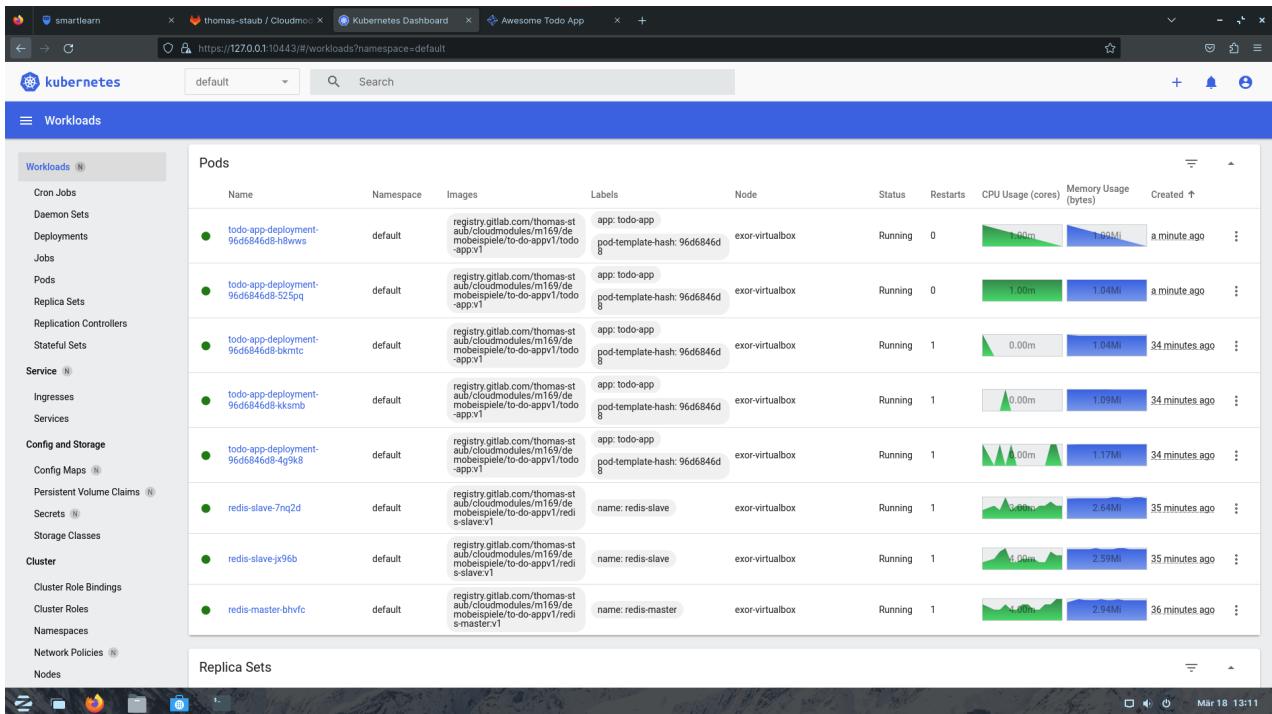
Was ist Down- und Upscaling?

Down- und Upscaling ist einfach gesagt, einfach die Veränderung der Konfiguration. Denn sobald sich die Konfiguration ändert, versucht Kubernetes sofort den Stand der Konfiguration nachzubilden. Dies können wir entweder erreichen, indem wir unser Config-File anpassen und diese Anpassung anwenden oder indem wir Kubernetes mithilfe von kubectl direkt den Befehl geben, die Pods zu reduzieren/erhöhen. Jedoch müssen wir dabei aufpassen, da sich unser Config-File nicht ändert, wenn wir die Anweisung direkt mit kubectl übergeben. Das heisst, wenn ich das Config-File in einer anderen Umgebung nutze, werde ich immer noch die alte Anzahl an Pods haben. Dies kann vor allem in Produktivumgebungen zu grossen Problemen führen.

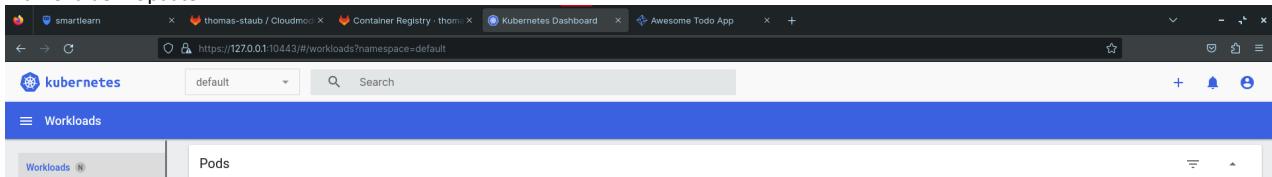
Todo-App V2 (Rolling Updates)

Unten Bilder, welche die funktionierende Todo-App V2 zeigen und wie sie im Dashboard aussieht:

Vor dem Update:



Während dem Update:



	Name	Namespace	Images	Labels	Node	Status	Restarts	CPU Usage (cores)	Memory Usage (bytes)	Created
Cron Jobs										
Daemon Sets										
Deployments	todo-app-deployment-96d6846d8-mc4xs	default	registry.gitlab.com/thomas-staub/cloudmodules/m169/de/mobespiel/to-do-appv1/todo-appv1	app: todo-app pod-template-hash: 96d6846d8	exor-virtualbox	Running	0	-	-	23 seconds ago
Jobs										
Pods	todo-app-deployment-96d6846d8-gmzx2	default	registry.gitlab.com/thomas-staub/cloudmodules/m169/de/mobespiel/to-do-appv1/todo-appv1	app: todo-app pod-template-hash: 96d6846d8	exor-virtualbox	Running	0	-	-	23 seconds ago
Replica Sets										
Replication Controllers										
Stateful Sets										
Service										
Ingresses	todo-app-deployment-96d6846d8-tgjfr	default	registry.gitlab.com/thomas-staub/cloudmodules/m169/de/mobespiel/to-do-appv1/todo-appv1	app: todo-app pod-template-hash: 96d6846d8	exor-virtualbox	Running	0	0.00m	1.04Mi	2.minutes ago
Services										
Config and Storage										
Config Maps										
Persistent Volume Claims										
Secrets										
Storage Classes										
Cluster										
Cluster Role Bindings										
Cluster Roles										
Namespaces										
Network Policies										
Nodes	redis-slave-7nq2d	default	registry.gitlab.com/thomas-staub/cloudmodules/m169/de/mobespiel/to-do-appv1/redis-slave:v1	name: redis-slave	exor-virtualbox	Running	1	9.00m	2.64Mi	an.hour.ago
	redis-slave-jx96b	default	registry.gitlab.com/thomas-staub/cloudmodules/m169/de/mobespiel/to-do-appv1/redis-slave:v1	name: redis-slave	exor-virtualbox	Running	1	4.00m	2.60Mi	an.hour.ago
	redis-master-bhvfc	default	registry.gitlab.com/thomas-staub/cloudmodules/m169/de/mobespiel/to-do-appv1/redis-master:v1	name: redis-master	exor-virtualbox	Running	1	4.00m	2.98Mi	an.hour.ago

Nach dem Update:

Kubernetes Dashboard

Workloads

	Name	Namespace	Images	Labels	Node	Status	Restarts	CPU Usage (cores)	Memory Usage (bytes)	Created
Cron Jobs										
Daemon Sets										
Deployments	todo-app-deployment-5c87c4f97e-2jq7k	default	registry.gitlab.com/thomas-staub/cloudmodules/m169/de/mobespiel/to-do-appv2/todo-appv2	app: todo-app pod-template-hash: 5c87c4f976	exor-virtualbox	Running	0	-	-	31.seconds.ago
Jobs										
Pods	todo-app-deployment-5c87c4f97e-pcqzp	default	registry.gitlab.com/thomas-staub/cloudmodules/m169/de/mobespiel/to-do-appv2/todo-appv2	app: todo-app pod-template-hash: 5c87c4f976	exor-virtualbox	Running	0	7.00m	1.82Mi	58.seconds.ago
Replica Sets										
Replication Controllers										
Stateful Sets										
Service										
Ingresses	todo-app-deployment-5c87c4f97e-tg78s	default	registry.gitlab.com/thomas-staub/cloudmodules/m169/de/mobespiel/to-do-appv2/todo-appv2	app: todo-app pod-template-hash: 5c87c4f976	exor-virtualbox	Running	0	0.00m	1.12Mi	a.minute.ago
Services										
Config and Storage										
Config Maps										
Persistent Volume Claims										
Secrets										
Storage Classes										
Cluster										
Cluster Role Bindings										
Cluster Roles										
Namespaces										
Network Policies										
Nodes	redis-slave-7nq2d	default	registry.gitlab.com/thomas-staub/cloudmodules/m169/de/mobespiel/to-do-appv1/redis-slave:v1	name: redis-slave	exor-virtualbox	Running	1	3.00m	2.60Mi	an.hour.ago
	redis-slave-jx96b	default	registry.gitlab.com/thomas-staub/cloudmodules/m169/de/mobespiel/to-do-appv1/redis-slave:v1	name: redis-slave	exor-virtualbox	Running	1	3.00m	2.76Mi	an.hour.ago
	redis-master-bhvfc	default	registry.gitlab.com/thomas-staub/cloudmodules/m169/de/mobespiel/to-do-appv1/redis-master:v1	name: redis-master	exor-virtualbox	Running	1	3.00m	2.92Mi	an.hour.ago

Todo list!



Todo

Delete

Add

Delete

Was ist Blue/Green Deployment

Blue/Green Deployment ist eine Deployment-Strategie, bei welcher es 2 Umgebungen gibt. Die beiden Umgebungen sind in den optimalen Fällen genau gleich. Sagen wir mal, dass die grüne (Green) Umgebung aktuell unsere Produktivumgebung ist, auf welche alle Kundenanfragen landen, während die blaue (Blue) Umgebung die Stagingumgebung ist. Auf der blauen Umgebung kann ich oder jemand aus meinem Team ohne Sorgen unsere App updaten und sofort testen, während die grüne Umgebung normal weiterläuft. Funktioniert nun alles wie geplant, können wir die beiden Umgebungen austauschen. Das heisst unsere grüne Umgebung wird jetzt zur Produktivumgebung, die blaue wird zur Stagingumgebung und das Ganze beginnt wieder von vorn.

Was ist Cluster IP und was ist Node IP?

In Kubernetes gibt es verschiedene Servicetypen. Der standardmässige Servicetype ist Cluster-IP. Dabei ist die Cluster-IP eine virtuelle IP-Adresse, welche einen Service in einem Cluster verfügbar macht. Damit ich mit meinem Rechner auf eine Cluster-IP zugreifen kann, muss ein Kubernetes-Proxy dazwischengeschaltet werden.

Im Gegensatz zur Cluster-IP ist Node-IP kein Kubernetes-Servicetype. Die Node-IP ist lediglich die IP-Adresse von einem Node und wird zur clusterinternen Kommunikation verwendet.

Was ist NodePort?

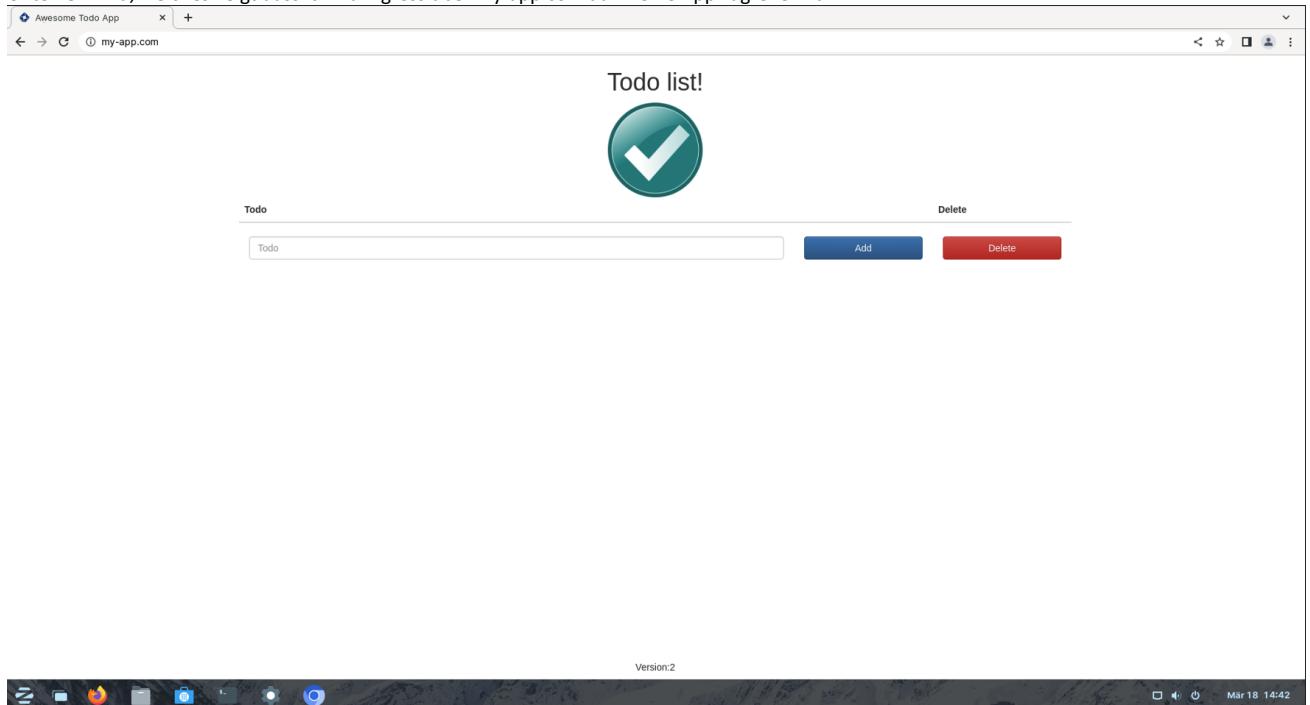
NodePort ist ein weiterer Kubernetes-Servicetyp. Dieser Servicetyp ermöglicht externen Zugriff auf einen Service. Dabei erhält werden den Node-IP's je einen statischen Port in einer Range von 30000-3276 zugewiesen.

Was ist ein LoadBalancer?

Ein Loadbalancer teilt den Netzwerkverkehr auf verschiedene Server auf so dass möglichst jeder Server gleich stark ausgelastet ist. Dabei wird ein Service auch extern verfügbar gemacht. Wenn man eine Cloud wie Azure oder AWS nutzt stellt der Cloud-Anbieter einen Loadbalancer zur Verfügung. Bei unserem Microk8s müssen wir selbst einen LoadBalancer enablen. Bei Microk8s ist dass Metallb.

Zugriff auf App via Ingress

Unten ein Bild, welches Zeigt dass ich via Ingress über my-app.com auf meine App zugreifen kann:

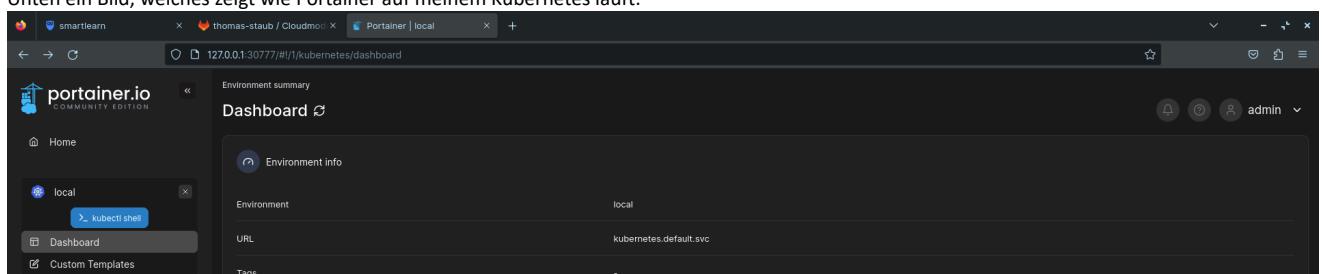


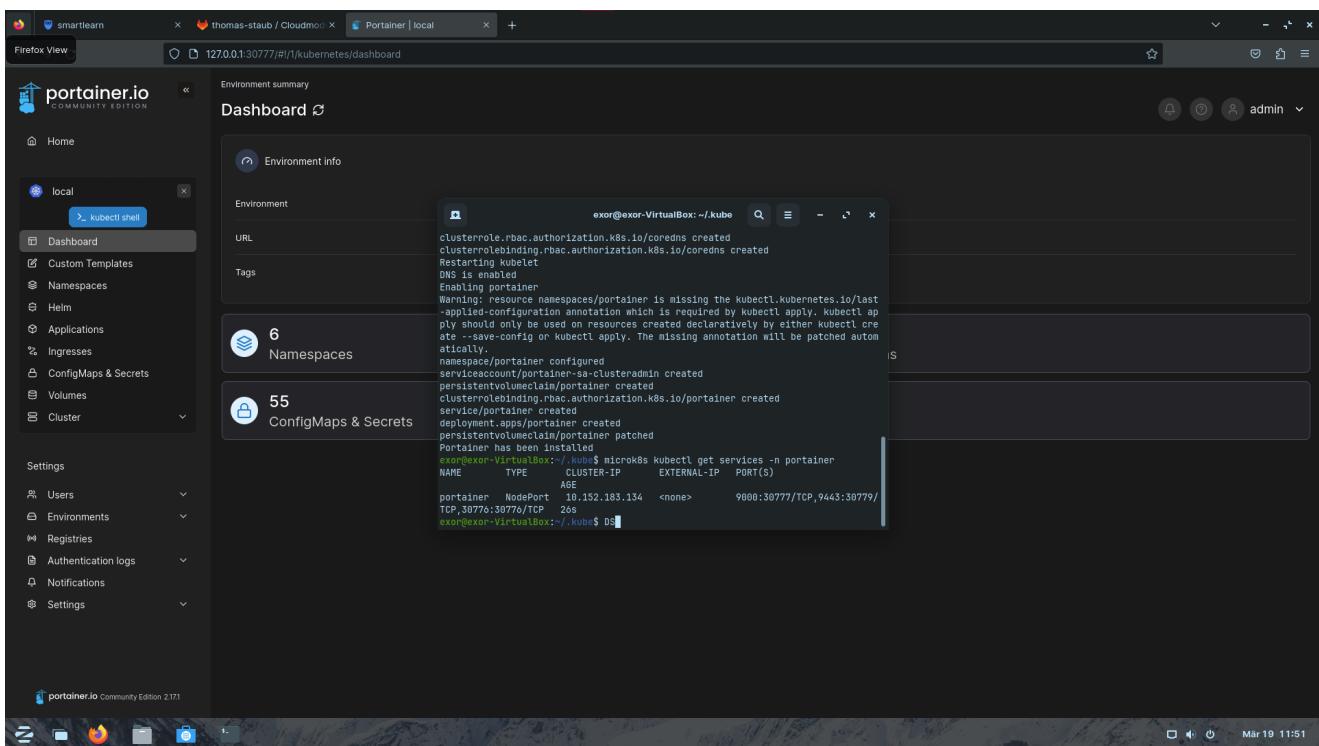
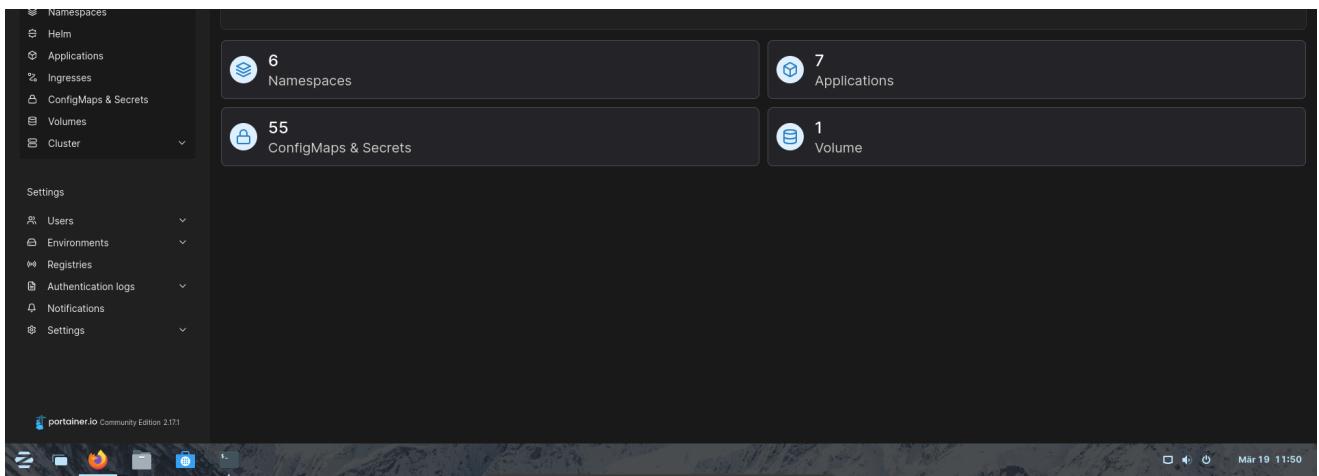
Was ist ein Namespace?

Ein Namespace ist eigentlich nur ein virtueller Bereich in welcher Gruppierungen innerhalb eines Clusters ermöglicht. Mit der Hilfe eines Namespaces ist einfacher Ressourcen und Zugriffsrechte zu verwalten. Zudem hat man insgesamt eine besser Kontrolle über die Umgebung. Dabei ist es wichtig zu sagen dass es nicht möglich ist einen Namespace innerhalb eines Namespaces zu erstellen.

Portainer auf Kubernetes

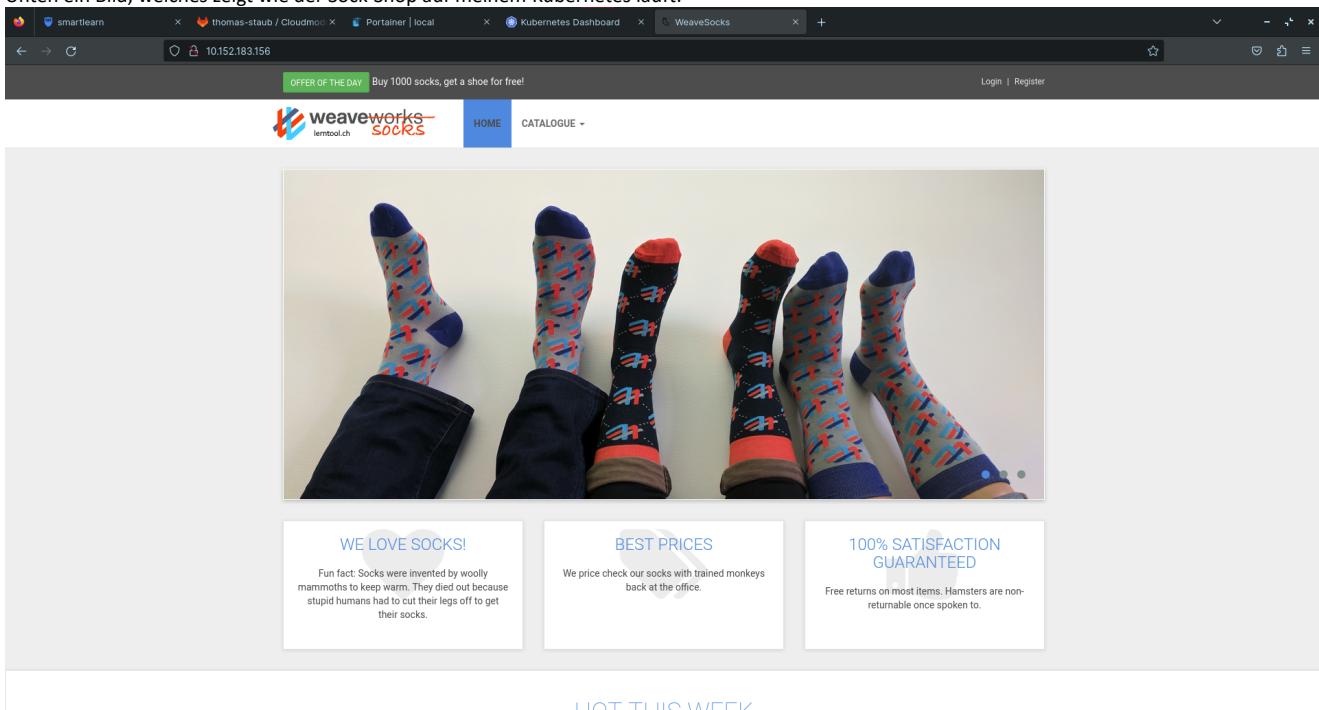
Unten ein Bild, welches zeigt wie Portainer auf meinem Kubernetes läuft:

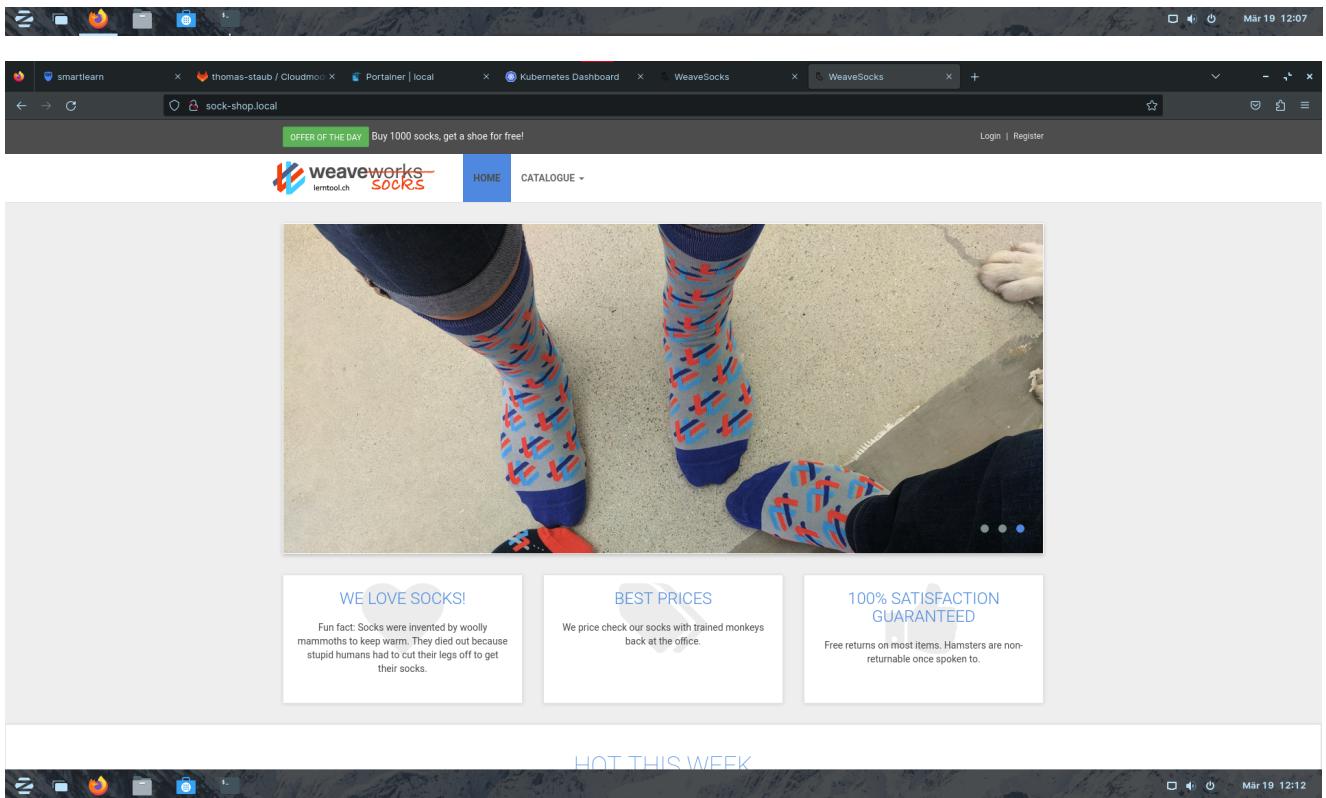




Sock-Shop

Unten ein Bild, welches zeigt wie der Sock-Shop auf meinem Kubernetes läuft:





Kiali

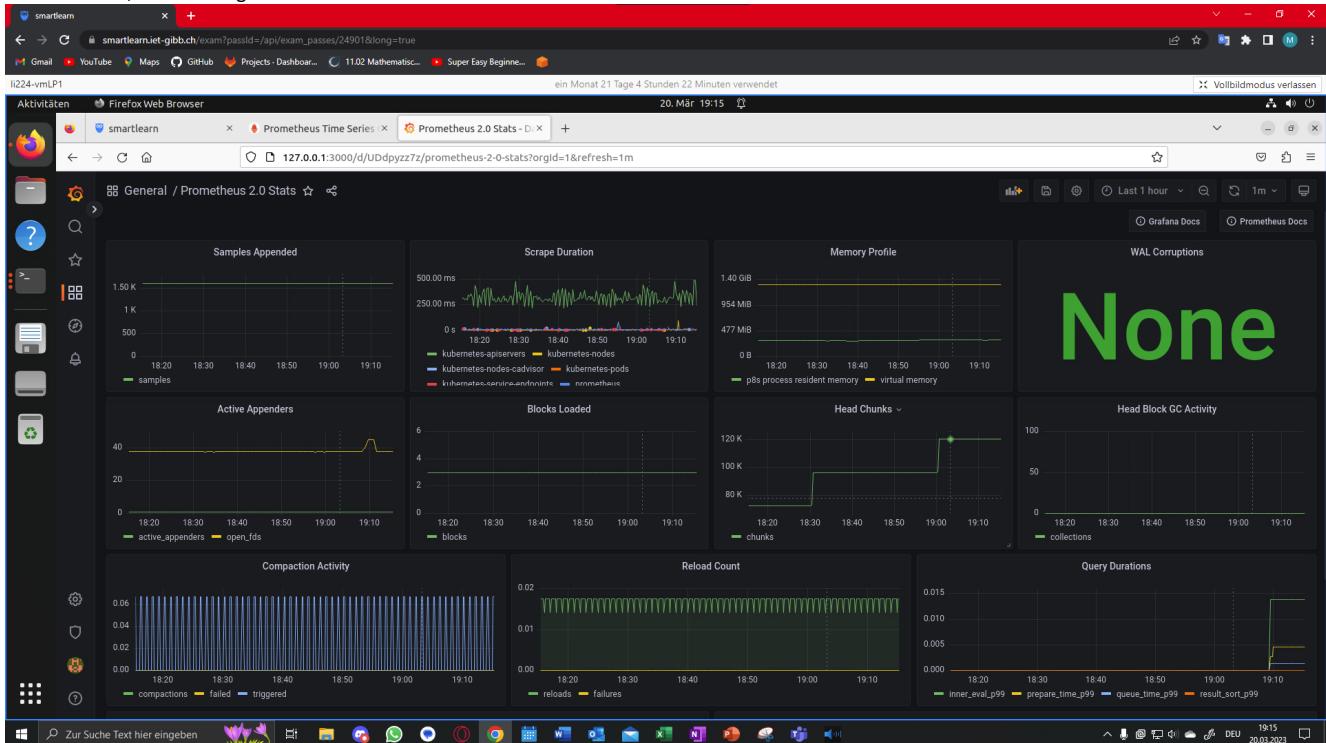
Unten ein Bild, welches zeigt wie Kiali läuft:

The Kiali interface displays traffic monitoring and configuration for various Kubernetes namespaces. In the top screenshot, the 'Overview' tab is selected, showing a grid of cards for each namespace. The 'sock-shop' card is highlighted. The bottom screenshot shows the 'Applications' section for the 'sock-shop' namespace, listing 14 healthy applications.

Namespace	Labels	Istio Config	Applications	Status
container-registry	1 Label	N/A	1 Application	Healthy
default	1 Label	N/A	0 Applications	N/A
ingress	1 Label	N/A	0 Applications	N/A
istio-system	1 Label	N/A	6 Applications	Healthy
metalLB-system	2 Labels	N/A	1 Application	Unhealthy
sock-shop	2 Labels	N/A	14 Applications	Healthy

Grafana

Unten ein Bild, welches zeigt wie Grafana meinen Cluster überwacht:



Eigenes Projekt auf Kubernetes

Ich wollte auch schon wie beim Docker-Compose Projekt meine Website aus dem Modul 293 nutzen. Das Docker-Image ist daher schon bestehend und hier zu finden: https://gitlab.com/exor420/modul-347/container_registry/. Dieses Image habe ich für dieses Projekt verwendet. Der Source-Code für die Website inklusive Dockerfile ist hier zu finden: <https://github.com/eXor420/GibbBiVo2021/tree/main/Modul%20347/Docker-Compose%20Project>.

Nun musste ich nur noch ein Kubernetes-Deployment erstellen. Dafür habe ich zum einen ein Deployment, wie auch einen Service erstellt. Für den Service habe ich als Servicetyp Nodeport gewählt, da ich dies schon aus meiner Firma kenne und mir somit nicht unnötig Schwierigkeiten machte. Ansonsten habe ich nicht wirklich etwas Spezielles angepasst. Die Replicas habe zum Beginn mal auf 1 gesetzt, da meine persönliche VM nicht so viel Leistung hat, aber dies lässt sich mit einer einfachen Anpassung skalieren. Mein Deploymentfile in welchem mein Deployment wie auch mein Service definiert ist, ist hier zu finden: <https://gitlab.com/exor420/modul-347/-/blob/main/my-portfolio-deployment.yaml>.

Anleitung:

Um das Ganze zum laufen zu bringen benötigen wir eine/n VM/Rechner auf welchem/welcher Microk8s installiert ist. Die folgende Anleitung funktioniert für eine frisch zurück gesetzte Smartlearn-VM vom Modul 347.

Als erstes holen wir unser Deploymentfile von GitLab mit folgendem Befehl:

```
git clone https://gitlab.com/exor420/modul-347.git
```

Mit dem nächsten Befehl navigieren wir in unser geklontes Verzeichnis:

```
cd modul-347/
```

Um das Deployment und den Service zu erstellen führen wir folgenden Befehl aus:

```
microk8s kubectl apply -f my-portfolio-deployment.yaml
```

Damit wir nun auf unsere Website zugreifen können führen wir folgende Befehle aus:

```
microk8s kubectl get service my-portfolio-service  
microk8s kubectl get nodes -o wide
```

Nachdem wir diese beiden Befehle ausgeführt haben sollten wir eine ähnliche Ausgabe wie die Folgende erhalten:

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
my-portfolio-service	NodePort	10.152.183.154	<none>	80 31002/TCP	91s
li224-vm1p1.smartlearn.lan	Ready	<none>	108d	v1.22.17-3+710dbf24ae2171 192.168.110.30:30	Up 22.04.1 LTS 5.15.0-58-generic containerd://1.5.13

Die beiden rot eingekreisten Werte sind wichtig. Den aus diesen können wir nun eine URL bilden, über welche wir auf unsere Website zugreifen können. In diesem Fall wäre es 192.168.110.30:31002 wie auch im folgenden Bild zu sehen ist:

