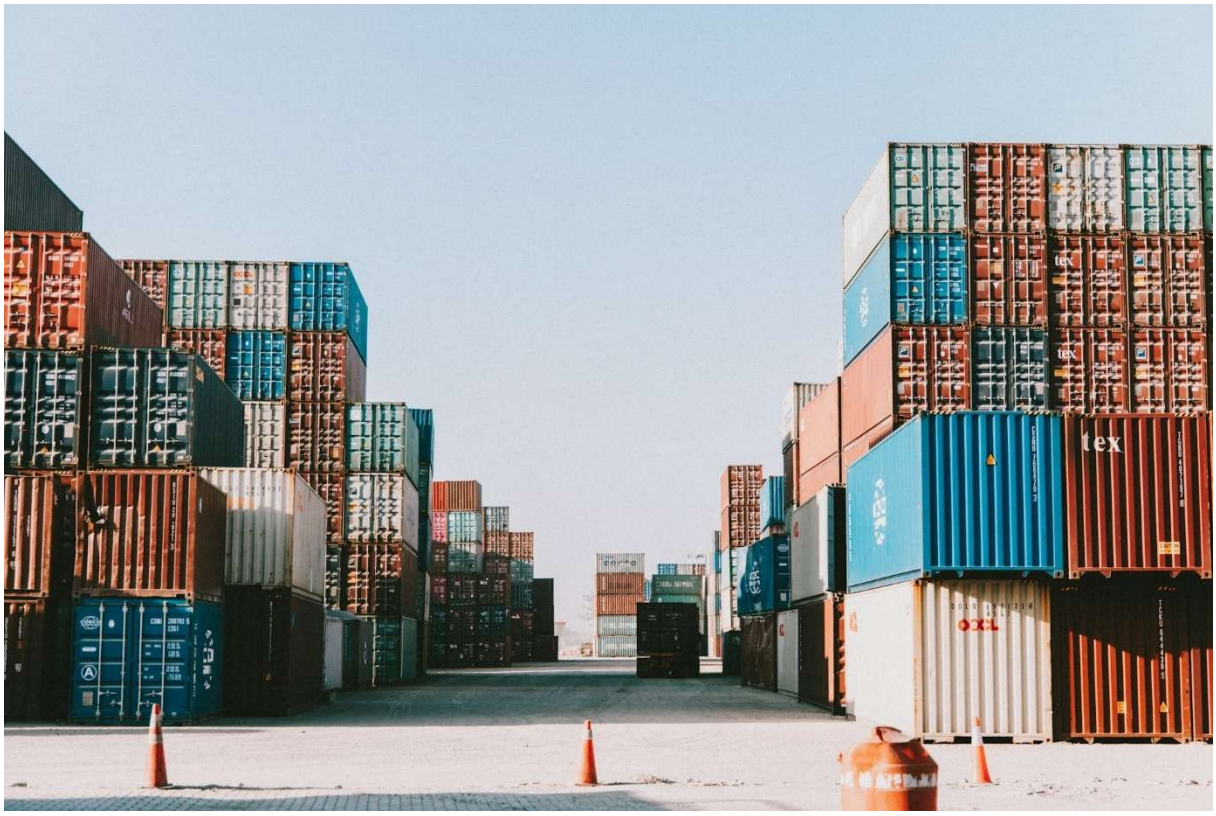


Abbildung 1 Symbolbild Container. Quelle: Foto von Fejuz auf Unsplash

Dienst mit Container anwenden

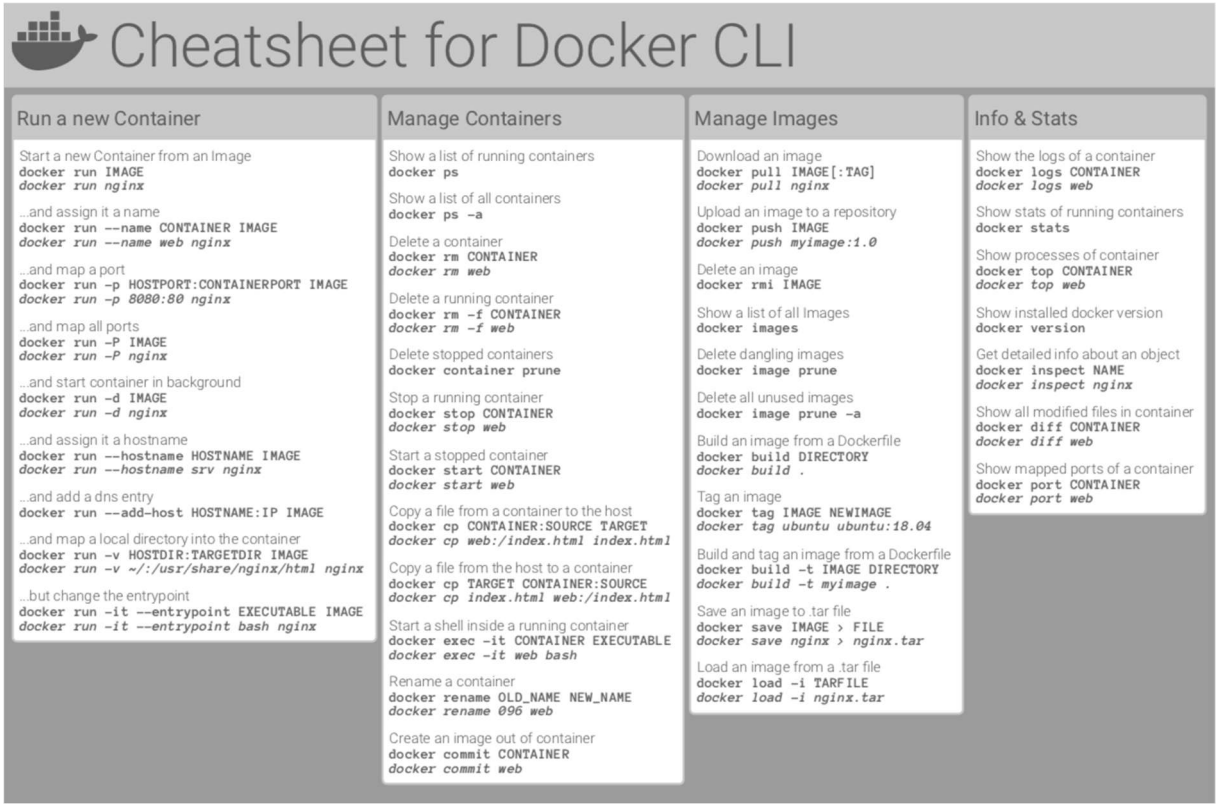
Lerndokumentation / Portfolio

Modul	IET-347 – Dienst mit Container anwenden FS26
Eingereicht von	Jonas Vetsch, INFW2025a
Eingereicht bei	Stefan Marti
Datum	1. April 2026

Inhaltsverzeichnis

1 Befehlsbibliothek

1.1 Grundlagen



Cheatsheet for Docker CLI

Run a new Container

Start a new Container from an Image
```
docker run IMAGE
docker run nginx
```

...and assign it a name
```
docker run --name CONTAINER IMAGE
docker run --name web nginx
```

...and map a port
```
docker run -p HOSTPORT:CONTAINERPORT IMAGE
docker run -p 8080:80 nginx
```

...and map all ports
```
docker run -P IMAGE
docker run -P nginx
```

...and start container in background
```
docker run -d IMAGE
docker run -d nginx
```

...and assign it a hostname
```
docker run --hostname HOSTNAME IMAGE
docker run --hostname srv nginx
```

...and add a dns entry
```
docker run --add-host HOSTNAME:IP IMAGE
```

...and map a local directory into the container
```
docker run -v HOSTDIR:TARGETDIR IMAGE
docker run -v ~/:/usr/share/nginx/html nginx
```

...but change the entrypoint
```
docker run -it --entrypoint EXECUTABLE IMAGE
docker run -it --entrypoint bash nginx
```

Manage Containers

Show a list of running containers
```
docker ps
```

Show a list of all containers
```
docker ps -a
```

Delete a container
```
docker rm CONTAINER
docker rm web
```

Delete a running container
```
docker rm -f CONTAINER
docker rm -f web
```

Delete stopped containers
```
docker container prune
```

Stop a running container
```
docker stop CONTAINER
docker stop web
```

Start a stopped container
```
docker start CONTAINER
docker start web
```

Copy a file from a container to the host
```
docker cp CONTAINER:SOURCE TARGET
docker cp web:/index.html index.html
```

Copy a file from the host to a container
```
docker cp TARGET CONTAINER:SOURCE
docker cp index.html web:/index.html
```

Start a shell inside a running container
```
docker exec -it CONTAINER EXECUTABLE
docker exec -it web bash
```

Rename a container
```
docker rename OLD_NAME NEW_NAME
docker rename 096 web
```

Create an image out of container
```
docker commit CONTAINER
docker commit web
```

Manage Images

Download an image
```
docker pull IMAGE[:TAG]
docker pull nginx
```

Upload an image to a repository
```
docker push IMAGE
docker push myimage:1.0
```

Delete an image
```
docker rmi IMAGE
```

Show a list of all Images
```
docker images
```

Delete dangling images
```
docker image prune
```

Delete all unused images
```
docker image prune -a
```

Build an image from a Dockerfile
```
docker build DIRECTORY
docker build .
```

Tag an image
```
docker tag IMAGE NEWIMAGE
docker tag ubuntu ubuntu:18.04
```

Build and tag an image from a Dockerfile
```
docker build -t IMAGE DIRECTORY
docker build -t myimage .
```

Save an image to .tar file
```
docker save IMAGE > FILE
docker save nginx > nginx.tar
```

Load an image from a .tar file
```
docker load -i TARFILE
docker load -i nginx.tar
```

Info & Stats

Show the logs of a container
```
docker logs CONTAINER
docker logs web
```

Show stats of running containers
```
docker stats
```

Show processes of container
```
docker top CONTAINER
docker top web
```

Show installed docker version
```
docker version
```

Get detailed info about an object
```
docker inspect NAME
docker inspect nginx
```

Show all modified files in container
```
docker diff CONTAINER
docker diff web
```

Show mapped ports of a container
```
docker port CONTAINER
docker port web
```

1.2 Images builden, Container starten

Befehl	Funktion
docker run [image]	Der wichtigste Befehl: Er nimmt ein Image (den Bauplan) und startet daraus einen Container.
docker ps	Liste aller Container, die gerade aktiv sind und laufen.
docker ps -a	Zeigt dir alle Container an – auch die, die bereits gestoppt wurden oder abgestürzt sind.
docker images	Listet alle Baupläne (Images) auf, die aktuell auf dem Rechner gespeichert sind.
docker stop [container-id]	Container sauber herunterfahren, ohne ihn zu löschen.
docker start [container-id]	Gestoppten Container wieder weiterlaufen lassen.
docker rm [container-id]	Container endgültig löschen (er muss dafür vorher gestoppt sein).
docker rmi [image-id]	Löscht einen Bauplan (Image) vom Computer, um Speicherplatz zu schaffen.
docker exec -it [container] bash	Sich auf die Konsole (z.B. bash) eines laufenden Containers einloggen.
docker pull [image]	Lädt einen fertigen Bauplan aus dem Internet (Docker Hub) herunter.

1.3 Dockerfile (Build) und Push auf Online Repositories

docker build	Ein Image bauen
docker network create [networkname]	Dockernetzwerk anlegen, wodurch mehrere Container in einem Netzwerk untereinander kommunizieren können
Docker image build -t [tag name]	Image bauen mit Tagname
`docker image pull <Repository>:<tag>`	Images können von Docker Hub heruntergeladen werden.
`docker image tag [name bestehendes image] [zusätzlicher Name geben]`	Einem Image einen neuen Namen taggen (z.B. für den Upload wird hier der Repositorypfad angegeben).
`docker push [image tag inkl. GitLab-Pfad]`	Neu getaggtes Image kann nun gepusht werden. Der Push-Ort (Repository) ergibt sich direkt aus dem Pfad im Tag.
`Docker ps -a`	Alle Container anzeigen, auch die gestoppten
`docker system prune -a --volumes`	Alle Container aufräumen, auch die gestoppten.

1.4 Docker compose und Docker network

`docker compose up -d`	Mehrere Container zusammen starten (und ggf. Verbinden): Führt die Befehle aus **`docker-compose.yaml`** aus.
`docker compose down`	Zusammen gestartete Container wieder herunterfahren und auch direkt aufräumen (löschen).
`docker network create todoapp_network`	Docker Netzwerk anlegen
`docker network ls`	D Netzwerke anzeigen
`docker build -t todo-app:v1 .`	docker image builden
`docker run --net=todoapp_network --name=frontend -d -p 3000:3000 todo-app:v1`	docker Container starten mit Einbindung eines (vorhandenen!) Netzwerks.
`docker logs -f <container id>`	Logs live anschauen, wobei man bei container id auch nur die ersten paar Zeichen eingeben kann, sofern diese eindeutig sind.
`docker system prune -a --volume`	System bereinigen: Alle Container, Images, und Volumes löschen
`docker stop $(docker ps -q) && docker rm $(docker ps -aq)`	Alle laufenden Container stoppen UND entfernen. (sehr praktisch!)

2 Grundlagen

2.1 Was sind Container?

Ein Container ist wie eine Virtuelle Maschine – allerdings ohne das Betriebssystem. Der Container wird auf dem Docker-System durch namespaces und cgroups in seinen Rechten eingeschränkt: Der Namespace definiert, was der Container sieht, die cgroup definiert, was der Container tun darf.

Doch nun fragst du dich: «Wenn ein Container wie eine VM ist, aber ohne das Betriebssystem, wie kann der Container dann ausgeführt werden?». Das Betriebssystem für einen Container ist Ubuntu. Wenn man Docker Desktop installiert, beispielsweise, so erstellt es eine Ubuntu VM. Auf dieser werden dann die Container ausgeführt. Das Betriebssystem ist bei Docker immer Linux, wird Docker auf anderen Betriebssystemen ausgeführt, so wird Linux im Hintergrund einfach virtualisiert.

2.2 Wozu sind Container nützlich?

Will ich z.B. eine selbst gemachte Software auf einem Server laufenlassen, so müsste ich diese händisch dort Aufsetzen (manuelle Konfiguration). Dank einem Container kann ich meine Software einfach in einem Container aufsetzen: Ich konfiguriere den Container so, dass er meine Software korrekt ausführt. Danach kann jeder meine Software ausführen, indem er einfach den Container ausführt. Dieser ist bereits konfiguriert.

Der Container wird dabei nicht selbst verschickt, sondern nur der Bauplan. Die Container sind isoliert (Sandboxing), was sehr gut für die Sicherheit ist. Damit kann ein Container nur auf sein eigenes Dateisystem zugreifen. Wird er kompromittiert, ist nicht der ganze Server in Gefahr, sondern eben nur der Container. Jeder Applikation sollte dabei ihren eigenen Container bekommen. Braucht eine Webseite auch eine Datenbank, so sollte die Datenbank als separater Container gebaut werden.

2.3 Was ist Devops?

Ein Team von Informatiker*innen, in dem sowohl Applikations- als auch Plattformentwickler*innen arbeiten. Früher waren diese beiden Berufsspezialisierungen oft in getrennten Teams tätig, wodurch die Wall-of-Confusion (Unterschiedliche Ziele und Prioritäten) noch grösser war als heute (Kommunikationsprobleme). Wenn beide ein Team bilden, so spricht man hierbei von einem DevOps-Team. Damit rücken die beiden Parteien näher aneinander, was die Kommunikation vereinfacht, und auch sonst profitiert die ganze Zusammenarbeit davon.

2.4 Unterschied Virtualisierung und Containerisierung

Will man dieselbe Umgebung reproduzierbar auf unterschiedlichen Servern installieren (um eine Applikation zu testen oder auszuliefern) so musste man früher eine VM haben. Eine virtuelle Maschine braucht einen Hypervisor, dieser lässt verschiedene Betriebssysteme laufen.

Virtuelle Maschinen sind sehr ressourcenhungrig. Zudem ist es kompliziert und aufwendig eine VM zu transportieren (sie braucht viel Speicherplatz!). Mit Docker kann eine Applikation als

Container statt als gesamtes Betriebssystem verschickt werden. Das bringt viel weniger Overhead mit sich. Auf dem Zielsystem wird der Container ausgeführt. Dabei wird kein Betriebssystem mitgeliefert.

2.5 Wie unterscheidet sich ein Image von einem Container?

Man kann sich ein Image wie ein Kochrezept oder eine Vorlage vorstellen. In diesem Image steht alles drin, was die Software braucht: der Code, die Bibliotheken und alle Einstellungen. Das Image selbst macht aber erst einmal nichts – es liegt einfach nur als Datei auf der Festplatte. Es ist quasi der „Bauplan“, den man leicht verschicken kann.

Ein Container ist dann die Ausführung dieses Bauplans. Wenn ich das Image starte, wird daraus ein laufender Container. Man kann sich das wie beim Backen vorstellen: Das Image ist das Rezept (die Theorie), und der Container ist der fertige Kuchen, der gerade im Ofen gebacken wird (die Praxis). Das Coole daran ist: Ich kann aus einem einzigen Image so viele Container gleichzeitig starten, wie ich will. Jeder Container läuft für sich allein, aber alle basieren auf demselben unveränderlichen Bauplan.

3 Meine ersten Schritte mit Docker

3.1 Softwareversionierung

2.5.11 - bedeutet Major-Version 2, Minor-Version 5 und Patch-Version 11.

Major-Versionen können bestehende Features verändern (Vorsicht beim Updaten!).

Minor-Versionen fügen neue Features hinzu, verändern die bestehenden aber nicht.

Play-With-Docker tönt sehr cool, würde ich gerne ausprobieren falls ich die Zeit finde.

3.2 Entwicklungsabläufe

Dockerdesktop integriert sich in verschiedene IDEs. Aus der IDE heraus kann ich dann Dockercontainer bauen, ausführen und debuggen.

Wenn eine Anwendung mehrere Container braucht (z.B. Backend und Datenbank), dann kann mit Docker Compose (eine YAML-Datei) definiert werden, welche Container gebraucht werden und wie diese gebaut werden müssen. Damit lässt sich dieses Containercluster an anderen Orten direkt wieder gleich aufbauen und ausführen.

Docker Hub ist eine Bibliothek von vorgefertigten Containern, die man verwenden kann.

3.3 Installation und Registrierung

Dank der Anleitung auf smartlearn war die Installation sehr einfach. Ich habe mich entschieden, Docker Desktop auf einem Ubuntu-Gerät zu installieren (nicht VM). Das hat gut geklappt.

Danach habe ich das hello world image geladen und ausgeführt. Auch das Ubuntu-Image hat funktioniert: Hier habe ich gelernt, dass Ubuntu als Container kein vollwertiges Betriebssystem ist (Container stellen schliesslich keine ganzen Betriebssysteme bereit), sondern einfach den Teil von Ubuntu emuliert, den es emulieren kann.

```
jon@jon-ubuntu-think:~$ docker pull ubuntu
Using default tag: latest
latest: Pulling from library/ubuntu
1baf05536e37: Download complete
a3629ac5b9f4: Pull complete
Digest: sha256:cd1dba651b3080c3686ecf4e3c4220f026b521fb76978881737d24f200828b2b
Status: Downloaded newer image for ubuntu:latest
docker.io/library/ubuntu:latest
jon@jon-ubuntu-think:~$ docker run -it ubuntu bash
root@a07482df3978:/# ls
bin  boot  dev  etc  home  lib  lib64  media  mnt  opt  proc  root  run  sbin  srv  sys  tmp  usr  var
root@a07482df3978:/# uname
Linux
root@a07482df3978:/# touch ich-war-da
root@a07482df3978:/#
```

Ein File, das ich in der Umgebung anlege, ist nicht persistent: Jedes Mal, wenn ich die Umgebung neu starte, ist das File wieder weg: Es wird jedes Mal eine frische Umgebung gestartet – auch der Hostname ist anders, wie man hier sieht:

```
jon@jon-ubuntu-think:~$ docker run -it ubuntu bash
root@a07482df3978:/# ls
bin  boot  dev  etc  home  lib  lib64  media  mnt  opt  proc  root  run  sbin
root@a07482df3978:/# uname
Linux
root@a07482df3978:/# touch ich-war-da
root@a07482df3978:/# exit
exit
jon@jon-ubuntu-think:~$ docker run -it ubuntu bash
root@fb80b2c1cb33:/# ls
bin  boot  dev  etc  home  lib  lib64  media  mnt  opt  proc  root  run  sbin
root@fb80b2c1cb33:/# touch ich-war-da
root@fb80b2c1cb33:/# ls -lah
total 56K
drwxr-xr-x   1 root root 4.0K Jan 27 13:23 .
drwxr-xr-x   1 root root 4.0K Jan 27 13:23 ..
-rwxr-xr-x   1 root root    0 Jan 27 13:23 .dockerenv
lrwxrwxrwx   1 root root    7 Apr 22  2024 bin -> usr/bin
drwxr-xr-x   2 root root 4.0K Apr 22  2024 boot
drwxr-xr-x   5 root root  360 Jan 27 13:23 dev
drwxr-xr-x   1 root root 4.0K Jan 27 13:23 etc
drwxr-xr-x   3 root root 4.0K Jan 13 02:11 home
-rw-r--r--   1 root root    0 Jan 27 13:23 ich-war-da
lrwxrwxrwx   1 root root    7 Apr 22  2024 lib -> usr/lib
lrwxrwxrwx   1 root root    9 Apr 22  2024 lib64 -> usr/lib64
drwxr-xr-x   2 root root 4.0K Jan 13 02:04 media
drwxr-xr-x   2 root root 4.0K Jan 13 02:04 mnt
drwxr-xr-x   2 root root 4.0K Jan 13 02:04 opt
dr-xr-xr-x 325 root root    0 Jan 27 13:23 proc
drwx------   2 root root 4.0K Jan 13 02:11 root
drwxr-xr-x   4 root root 4.0K Jan 13 02:11 run
lrwxrwxrwx   1 root root    8 Apr 22  2024 sbin -> usr/sbin
drwxr-xr-x   2 root root 4.0K Jan 13 02:04 srv
dr-xr-xr-x  11 root root    0 Jan 27 13:23 sys
drwxrwxrwt   2 root root 4.0K Jan 13 02:11 tmp
drwxr-xr-x  12 root root 4.0K Jan 13 02:04 usr
drwxr-xr-x  11 root root 4.0K Jan 13 02:11 var
root@fb80b2c1cb33:/# exit
exit
jon@jon-ubuntu-think:~$ docker run -it ubuntu bash
root@c56e6046ceea:/# ls -a
.  ..  .dockerenv  bin  boot  dev  etc  home  lib  lib64  media  mnt  opt  proc
root@c56e6046ceea:/# ls -lah
total 56K
drwxr-xr-x   1 root root 4.0K Jan 27 13:23 .
drwxr-xr-x   1 root root 4.0K Jan 27 13:23 ..
-rwxr-xr-x   1 root root    0 Jan 27 13:23 .dockerenv
lrwxrwxrwx   1 root root    7 Apr 22  2024 bin -> usr/bin
drwxr-xr-x   2 root root 4.0K Apr 22  2024 boot
drwxr-xr-x   5 root root  360 Jan 27 13:23 dev
drwxr-xr-x   1 root root 4.0K Jan 27 13:23 etc
drwxr-xr-x   3 root root 4.0K Jan 13 02:11 home
lrwxrwxrwx   1 root root    7 Apr 22  2024 lib -> usr/lib
lrwxrwxrwx   1 root root    9 Apr 22  2024 lib64 -> usr/lib64
drwxr-xr-x   2 root root 4.0K Jan 13 02:04 media
drwxr-xr-x   2 root root 4.0K Jan 13 02:04 mnt
drwxr-xr-x   2 root root 4.0K Jan 13 02:04 opt
dr-xr-xr-x 327 root root    0 Jan 27 13:23 proc
drwx------   2 root root 4.0K Jan 13 02:11 root
drwxr-xr-x   4 root root 4.0K Jan 13 02:11 run
lrwxrwxrwx   1 root root    8 Apr 22  2024 sbin -> usr/sbin
drwxr-xr-x   2 root root 4.0K Jan 13 02:04 srv
dr-xr-xr-x  11 root root    0 Jan 27 13:23 sys
drwxrwxrwt   2 root root 4.0K Jan 13 02:11 tmp
drwxr-xr-x  12 root root 4.0K Jan 13 02:04 usr
drwxr-xr-x  11 root root 4.0K Jan 13 02:11 var
root@c56e6046ceea:/# exit
exit
```

3.4 Docker pull

Mit einem «:» kann ich angeben, welche Version eines Images ich herunterladen möchte, so z.B.:

```
jon@jon-ubuntu-think:~$ docker pull ubuntu:21.10
21.10: Pulling from library/ubuntu
54b8fda3c9de: Pull complete
Digest: sha256:ff46b78279f207db3b8e57e20dee7cecef3567d09489369d80591f150f9c8154
Status: Downloaded newer image for ubuntu:21.10
docker.io/library/ubuntu:21.10
jon@jon-ubuntu-think:~$ docker run -it ubuntu:21.10 bash
root@9bb15825a8c8:/# cat /etc/lsb-release
DISTRIB_ID=Ubuntu
DISTRIB_RELEASE=21.10
DISTRIB_CODENAME=impish
DISTRIB_DESCRIPTION="Ubuntu 21.10"
root@9bb15825a8c8:/#
```

Um sicherzustellen, dass die aktuellste Version heruntergeladen wird kann «:latest» verwendet werden, so zum Beispiel:

```
jon@jon-ubuntu-think:~$ docker pull ubuntu:latest
latest: Pulling from library/ubuntu
Digest: sha256:cd1dba651b3080c3686ecf4e3c4220f026b521fb76978881737d24f200828b2b
Status: Image is up to date for ubuntu:latest
docker.io/library/ubuntu:latest
jon@jon-ubuntu-think:~$ docker images

IMAGE                ID             DISK USAGE   CONTENT SIZE   EXTRA
hello-world:latest   05813aedc15f       25.9kB         9.52kB    U
ubuntu:21.10         ff46b78279f2        117MB         30.4MB    U
ubuntu:latest        cd1dba651b30        119MB         31.7MB    U
jon@jon-ubuntu-think:~$
```

3.5 Docker Image

Ein Docker-Image ist ein Objekt, das ein Dateisystem eines Betriebssystems (aber nicht das Betriebssystem selbst! Es wird auch «reduziertes Betriebssystem» genannt), eine Anwendung und die Anwendungsabhängigkeiten enthält.

Das Image ist wie die Klasse (Objekt) und der Container ist die Instanz davon. Das Image ist ein Konstrukt zur Erstellungszeit, der Container ist das Konstrukt zur Laufzeit.

Die Images werden bewusst sehr minimal gehalten, was den Inhalt betrifft. Es wird nur eine Shell ausgeliefert, wenn diese auch wirklich notwendig für die Applikation ist, die wir ausliefern wollen. Damit bleiben die Images sehr klein!

Wenn ich ein Dockerimage ausführe, das noch gar nicht heruntergeladen wurde, wird es automatisch zuerst heruntergeladen. So kann ich z.B. einfach

docker run --name some-nginx -d -p 8080:80 nginx

ausführen, ohne nginx herunterzuladen.

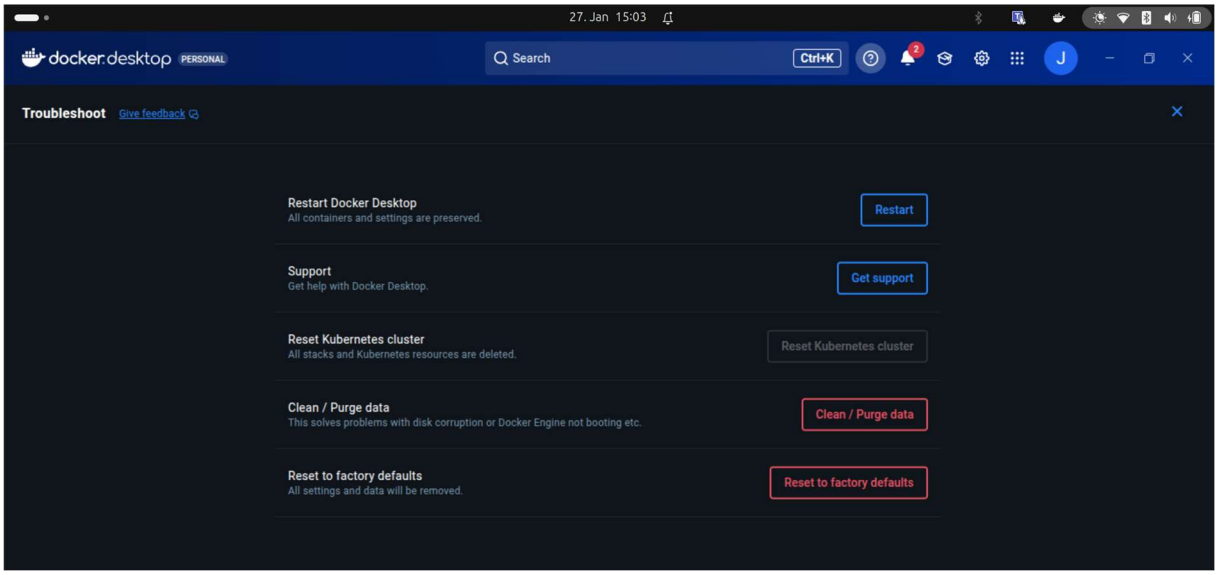
Ein Container erhält einen zufälligen Namen, sofern man ihn nicht angibt beim Starten. Danach kann man keinen zweiten Container mit demselben Namen starten. Dazu müsste man den Container zuerst löschen (auch wenn er schon gestoppt ist, muss man ihn dennoch zuerst löschen!). Wenn man lediglich etwas herumexperimentieren will, kann man den Container mit «--rm» starten, dann löscht er sich beim Stoppen automatisch.

```
jon@jon-ubuntu-think:~$ docker run --name mein-webserver -d --rm -p 8080:80 nginx
d476dc2e72a813d3fb2d869605687ad5e79127974956bec5e3b9d76d9a254b81
jon@jon-ubuntu-think:~$ docker ps -a
CONTAINER ID   IMAGE          COMMAND                  CREATED          STATUS                      PORTS
NAMES
d476dc2e72a8   nginx          "/docker-entrypoint.…"   6 seconds ago    Up 6 seconds                0.0.0.0:8080->80/tcp, [::]:8080->80/tcp
mein-webserver
9bb15825a8c8   ubuntu:21.10   "bash"                   29 minutes ago   Exited (0) 26 minutes ago
sweet_austin
c56e6046ceea   ubuntu         "bash"                   32 minutes ago   Exited (0) 32 minutes ago
exciting_clarke
fb80b2c1cb33   ubuntu         "bash"                   33 minutes ago   Exited (0) 32 minutes ago
agitated_jones
a07482df3978   ubuntu         "bash"                   40 minutes ago   Exited (0) 33 minutes ago
wizardly_knuth
bc94bd878f14   hello-world    "/hello"                 2 hours ago      Exited (0) 2 hours ago
nostalgic_galileo
jon@jon-ubuntu-think:~$
```

Nach dem Experimentieren können alle gestoppten Container gelöscht werden mit:

```
docker system prune -a --volumes
```

Über das GUI ist die Funktion «Clean / Purge data» zu verwenden:



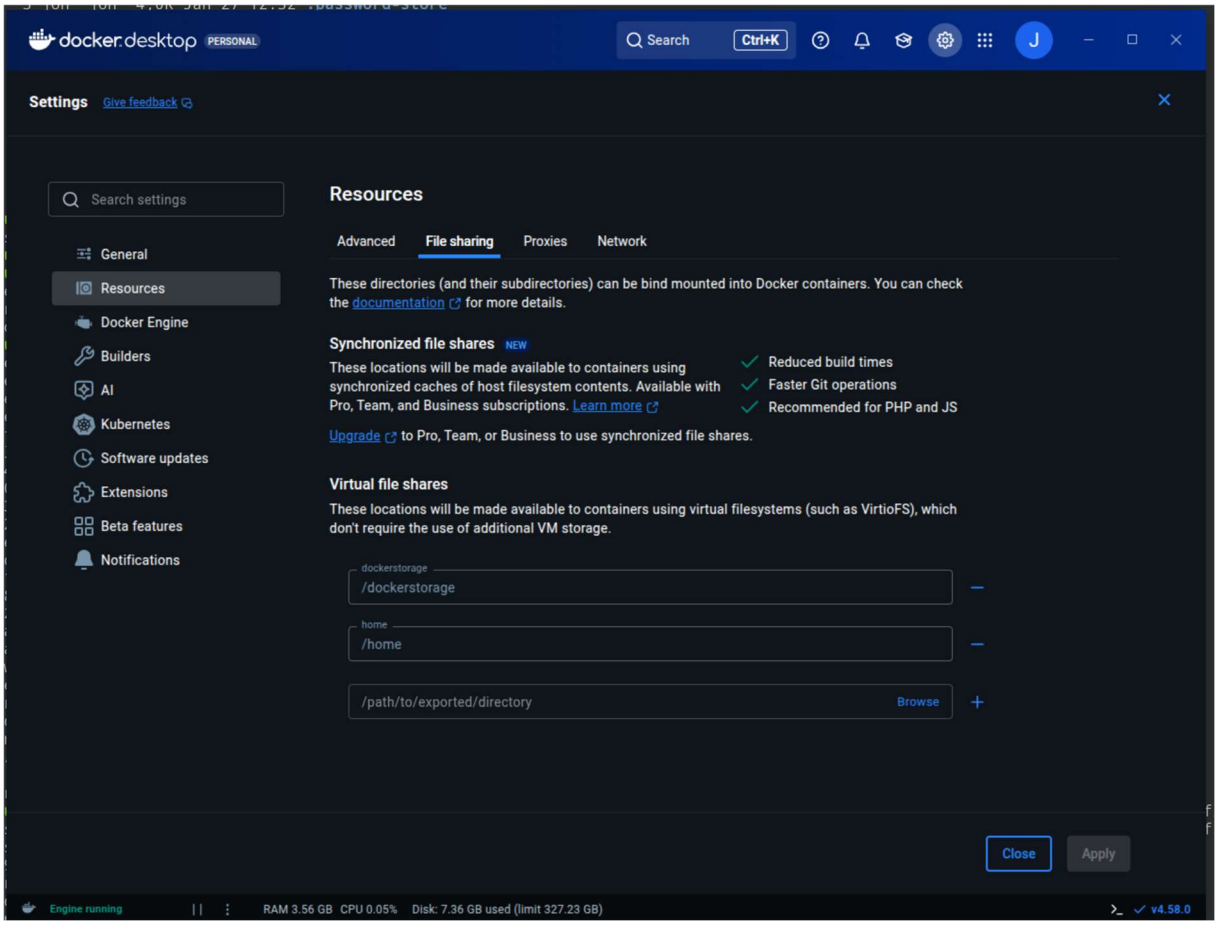
Images werden mit «docker rmi [imagename]» gelöscht.

Logs gibt es mit «docker logs -f mein-webserver», wobei -f dafür sorgt, dass die Logs immer direkt in der Konsole ausgegeben werden (live)…

3.6 OnlyOffice Übung

3.6.1 Setup

Damit Container auf mein lokales Filesystem zugreifen dürfen, kann ich über Docker Desktop Ordner freigeben. Der Vorteil? Dateien bleiben nach beenden des Containers persistent. :-)



Nicht vergessen danach auf «Apply» zu drücken, das hatte ich nämlich zuerst vergessen ;-)

3.6.2 Setup mit Kommandozeile

Für die OnlyOffice Übung habe ich zunächst persistente Ordner auf dem Hostsystem angelegt:

```
jon@jon-ubuntu-vbox:~$ sudo mkdir -p /app/onlyoffice/DocumentServer/logs
jon@jon-ubuntu-vbox:~$ sudo mkdir -p /app/onlyoffice/DocumentServer/data
jon@jon-ubuntu-vbox:~$ sudo mkdir -p /app/onlyoffice/DocumentServer/lib
jon@jon-ubuntu-vbox:~$ sudo mkdir -p /app/onlyoffice/DocumentServer/db
jon@jon-ubuntu-vbox:~$ sudo chown -R jon:jon /app
jon@jon-ubuntu-vbox:~$ ls -lah /
total 100K
drwxr-xr-x  24 root root 4.0K Feb  3 14:19 .
drwxr-xr-x  24 root root 4.0K Feb  3 14:19 ..
drwxr-xr-x   3 jon  jon  4.0K Feb  3 14:19 app
lrwxrwxrwx   1 root root    7 Apr 22  2024 bin -> usr/bin
drwxr-xr-x   2 root root 4.0K Feb 26  2024 bin.usr-is-merged
drwxr-xr-x   3 root root 4.0K Feb  3 13:06 boot
dr-xr-xr-x   2 root root 4.0K Feb  3 11:08 cdrom
drwxr-xr-x  19 root root 4.0K Feb  3 14:16 dev
drwxr-xr-x 141 root root  12K Feb  3 14:15 etc
drwxr-xr-x   3 root root 4.0K Feb  3 12:05 home
lrwxrwxrwx   1 root root    7 Apr 22  2024 lib -> usr/lib
lrwxrwxrwx   1 root root    9 Apr 22  2024 lib64 -> usr/lib64
drwxr-xr-x   2 root root 4.0K Apr  8  2024 lib.usr-is-merged
drwx------   2 root root  16K Feb  3 11:15 lost+found
drwxr-xr-x   2 root root 4.0K Aug  5 16:48 media
drwxr-xr-x   2 root root 4.0K Aug  5 16:48 mnt
drwxr-xr-x   4 root root 4.0K Feb  3 14:14 opt
dr-xr-xr-x 334 root root    0 Feb  3 14:12 proc
drwx------   5 root root 4.0K Feb  3 12:05 root
drwxr-xr-x  36 root root  960 Feb  3 14:14 run
lrwxrwxrwx   1 root root    8 Apr 22  2024 sbin -> usr/sbin
drwxr-xr-x   2 root root 4.0K Mar 31  2024 sbin.usr-is-merged
drwxr-xr-x  12 root root 4.0K Aug  5 16:54 snap
drwxr-xr-x   2 root root 4.0K Aug  5 16:48 srv
dr-xr-xr-x  13 root root    0 Feb  3 14:12 sys
drwxrwxrwt  16 root root 4.0K Feb  3 14:18 tmp
drwxr-xr-x  12 root root 4.0K Aug  5 16:48 usr
drwxr-xr-x  14 root root 4.0K Feb  3 12:04 var
```

Danach das Image herunterladen und direkt starten mit nur 1 Befehl:

```
sudo docker run -i -t -d -p 11012:80 --restart=always \
-v /app/onlyoffice/DocumentServer/logs:/var/log/onlyoffice \
-v /app/onlyoffice/DocumentServer/data:/var/www/onlyoffice/Data \
-v /app/onlyoffice/DocumentServer/lib:/var/lib/onlyoffice \
-v /app/onlyoffice/DocumentServer/db:/var/lib/postgresql \
-e JWT_SECRET=my_jwt_secret \
onlyoffice/documentserver
```

Das Image wird heruntergeladen…

```
jon@jon-ubuntu-vbox:~$ sudo docker run -i -t -d -p 11012:80 --restart=always     -v /app/onlyoffice/DocumentSer
ver/logs:/var/log/onlyoffice     -v /app/onlyoffice/DocumentServer/data:/var/www/onlyoffice/Data     -v /app/on
lyoffice/DocumentServer/lib:/var/lib/onlyoffice     -v /app/onlyoffice/DocumentServer/db:/var/lib/postgresql
  -e JWT_SECRET=my_jwt_secret     onlyoffice/documentserver
Unable to find image 'onlyoffice/documentserver:latest' locally
latest: Pulling from onlyoffice/documentserver
f34ebba65b2e: Download complete
37e2ee1a21ac: Download complete
46f42c9cc443: Downloading [========>                                          ]  163.6MB/939.4MB
aa1049b78923: Download complete
20043066d3d5: Pull complete
2d44649c5d87: Pull complete
d0e633916035: Downloading [===========>                                       ]  159.4MB/667.2MB
76e7a13aa60a: Download complete
4e9627466e1c: Download complete
299eddeab5e0: Download complete
```

Nach dem Befolgen der Anweisungen auf dem OnlyOffice-Webserver kann ein Test gestartet werden. Dort drin habe ich ein neues Dokument angelegt und kann es bearbeiten.

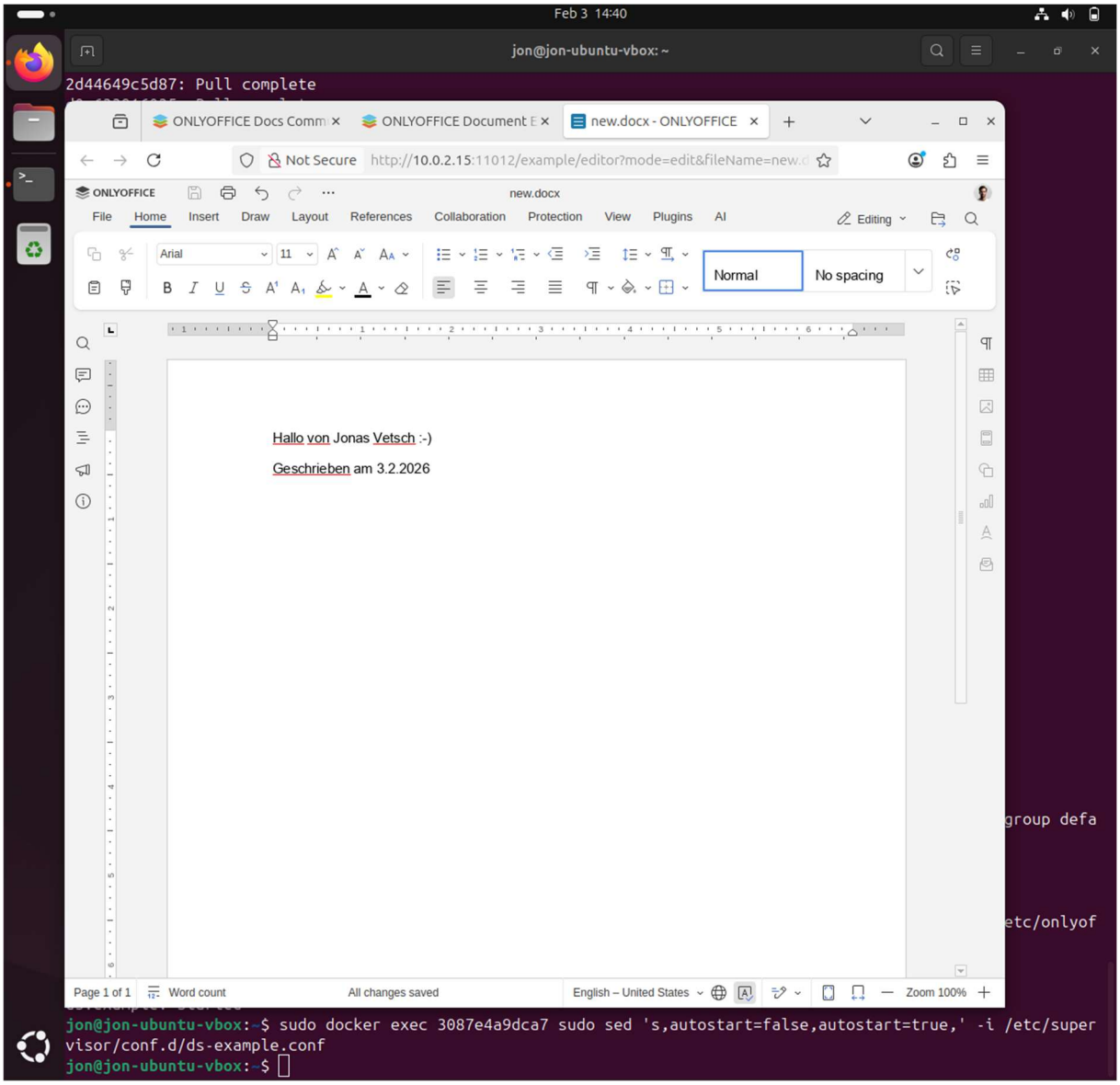
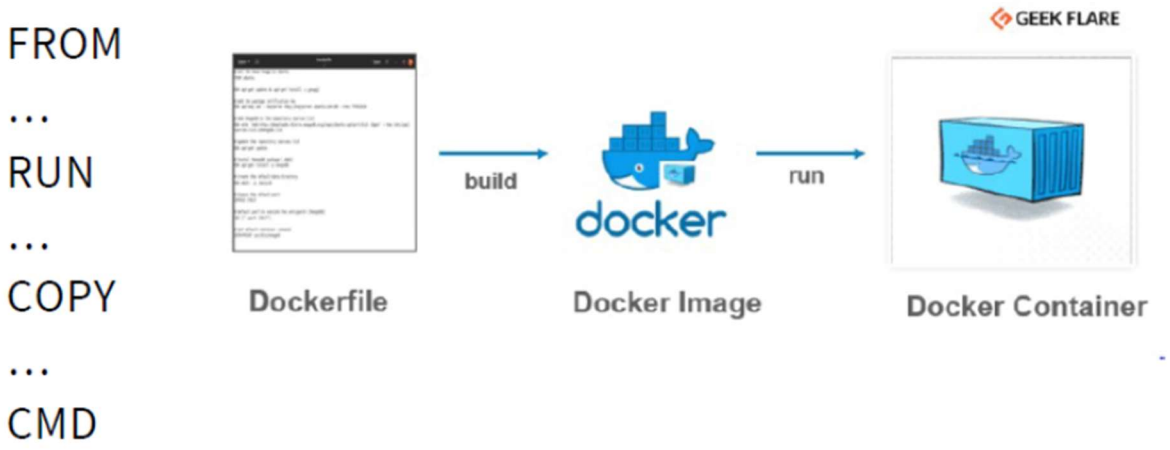


Abbildung 2 PrintScreen OnlyOffice mit eigenem Namen und Datum im Dokument

4 Dockerfile und ToDo-App-Übungsaufgabe



Zuerst den Code herunterladen: https://git.gibb.ch/thomas.staub/cloudmodule/-/tree/main/to-do-appv1

Für die ToDo-App machen wir 3 Container. Da wir erst Sourcecode haben, müssen wir zuerst Images bauen. Diese können wir dann als Container ausführen.

Beim Bauen eines Images werden die Befehle im Dockerfile (ohne Dateiendung!) ausgeführt.

Im Frontend-Ordner das Image bauen:

```
docker image build -t todo-app:v1 .
```

Dann das Image starten:

```
docker run --name frontend -d -p 3000:3000 todo-app:v1
```

In den Unterordnern der Redis-Teile auch diese als Image bauen:

```
docker build -t redis-slave:v1  .
```

```
docker build -t redis-master:v1  .
```

Nun müssen die 3 Container noch miteinander sprechen können. Wir legen ein virtuelles Docker-Netzwerk an:

```
docker network create todoapp_network
```

Nun können wir die drei Container starten und die ToDo-App auf Port 3000 testen.

```
docker run --net=todoapp_network --name=redis-master -d redis-master:v1
```

```
docker run --net=todoapp_network --name=redis-slave -d redis-slave:v1
```

```
docker run --net=todoapp_network --name=frontend -d -p 3000:3000 todo-app:v1
```

Dazu können wir noch schreiben, dass der Container sich selbst neustarten soll, wenn er mal aus Versehen heruntergefahren würde.

```
--restart unless-stopped
```

Das hat bestens funktioniert:

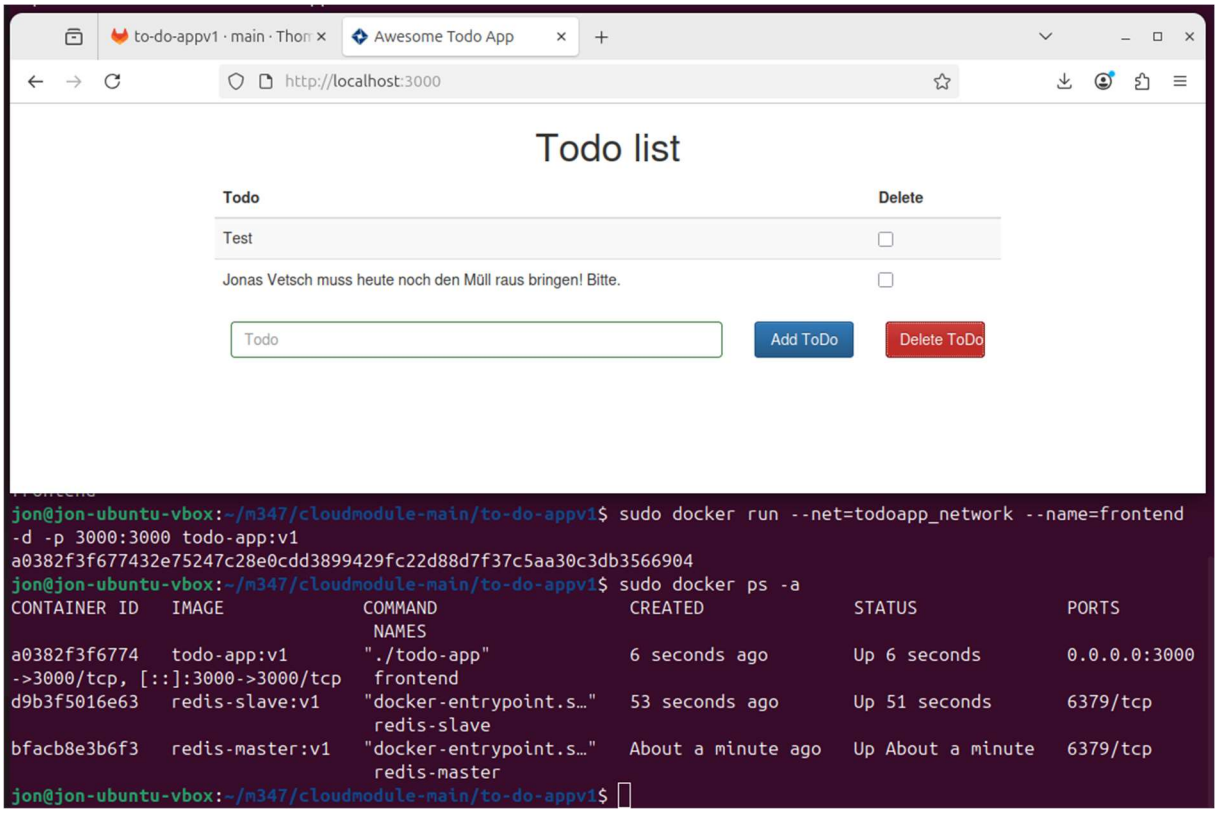


Abbildung 3 PrintScreen ihrer Version 1 mit Ihrem Namen als Todo Task

4.1 Ein Image auf Docker Hub pushen

Images können von Docker Hub heruntergeladen werden:

`docker image pull <Repository>:<tag>`

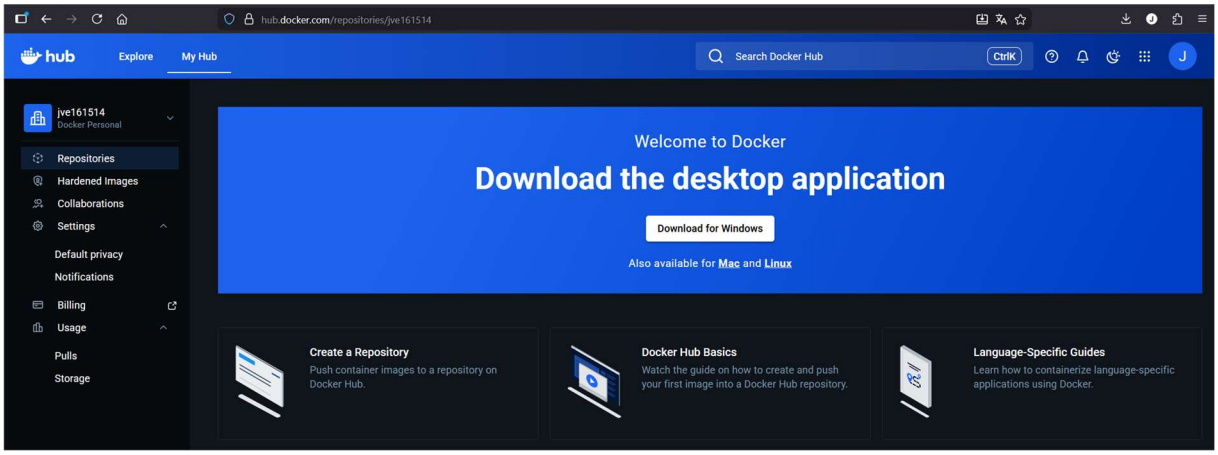


Abbildung 4 Mein persönliches Docker Hub Profil

Ich kann auch Images auf mein eigenes Docker Hub Profil pushen. Dazu muss das Image zuerst nach dem korrekten Pfad meines Nutzers auf Docker Hub benannt werden, inklusive eines Tags (z.B. :v1).

Zuerst aktuelle Namen prüfen mit **`docker image ls`**

Danach umbenennen mit:

```
docker image tag redis-master:v1 jve161514/redis-master:v1
docker image tag redis-slave:v1 jve161514/redis-slave:v1
docker image tag todo-app:v1 jve161514/todo-app:v1
```

Schliesslich pushen mit:

```
docker image push jve161514/redis-master:v1
docker image push jve161514/redis-slave:v1
docker image push jve161514/todo-app:v1
```

4.2 Ein Image auf GitLab pushen

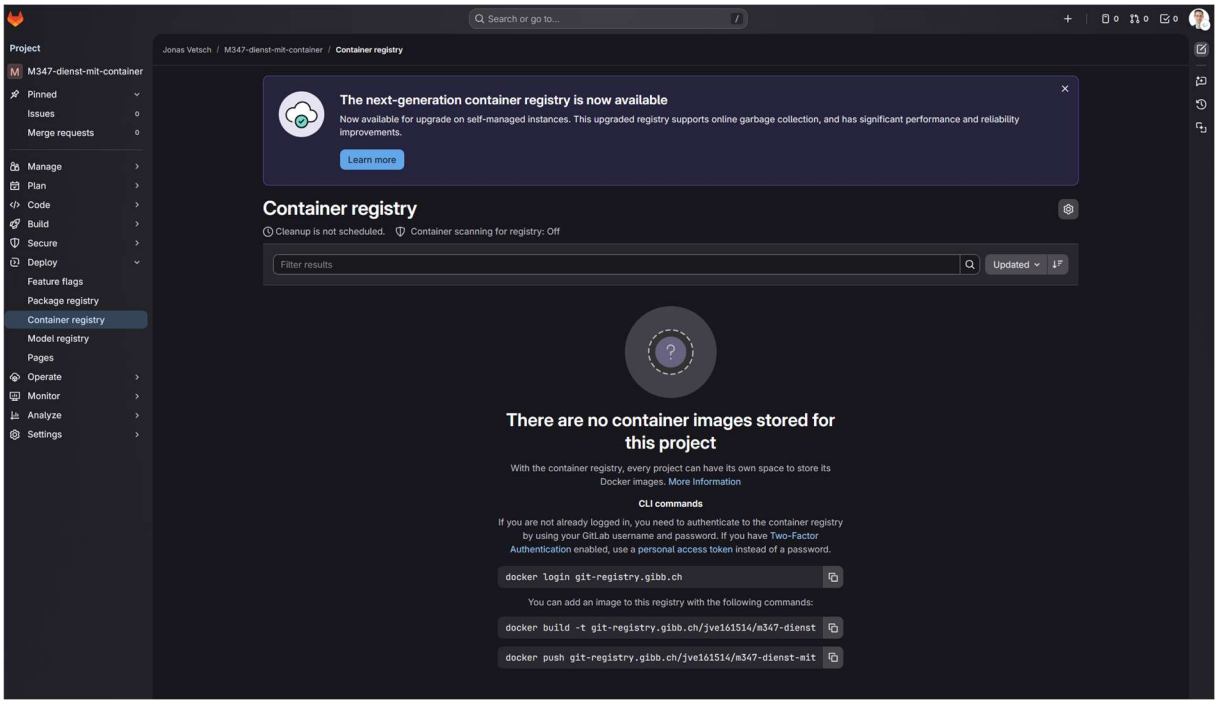
Aus der vorherigen ToDo-App Übung haben wir drei Images, die wir gebaut haben:

Nun wollen wir diese irgendwo hin pushen, damit ein Backup von unserer harten Arbeit gemacht wird!

DockerHub hat rate limits. Das kann man umgehen, indem man ein Image z.B. auf GitLab ablegt.

Zuerst ein Projekt (repo) auf GitLab anlegen.

Vorsicht: Wir sprechen hier nicht den Git-Endpunkt vom gibb GitLab an, sondern den **Container Registry** Endpunkt. Diesen kann man ausfindig machen, indem man im Projekt unter Deploy -> Container Registry geht.



Danach wieder die Images taggen, sodass docker weiss, wohin sie gehören. Der Pfad beim Pushen nimmt Docker nämlich aus den Namen.

docker image tag redis-master:v1 git-registry.gibb.ch/jve161514/m347-dienst-mit-container/to-do-app/redis-master:v1

docker image tag redis-slave:v1 git-registry.gibb.ch/jve161514/m347-dienst-mit-container/to-do-app/redis-slave:v1

docker image tag todo-app:v1 git-registry.gibb.ch/jve161514/m347-dienst-mit-container/to-do-app/todo-app:v1

Es scheint als wurden die Images kopiert. In Wahrheit ist es aber nur ein zusätzlicher Tag.

```
jon@jon-ubuntu-vbox:~$ docker image ls
                                                                                  i Info →  U  In Use
IMAGE                                                                ID             DISK USAGE   CONTENT SIZE   EXTRA
git-registry.gibb.ch/jve161514/m347-dienst-mit-container/to-do-app/redis-master:v1
                                                                     776eba237309       43.2MB         11.9MB    U
git-registry.gibb.ch/jve161514/m347-dienst-mit-container/to-do-app/redis-slave:v1
                                                                     79ebd6acf8e9       43.2MB         11.9MB    U
git-registry.gibb.ch/jve161514/m347-dienst-mit-container/to-do-app/todo-app:v1
                                                                     2683d9b0649a       20.3MB         5.93MB    U
redis-master:v1                                                      776eba237309       43.2MB         11.9MB    U
redis-slave:v1                                                       79ebd6acf8e9       43.2MB         11.9MB    U
todo-app:v1                                                          2683d9b0649a       20.3MB         5.93MB    U
jon@jon-ubuntu-vbox:~$
```

Danach mit Gitlab verbinden:

docker login git-registry.gibb.ch

Dort muss als Passwort ein Personal Access Token eingegeben werden, den kann man auf GitLab unter den User Settings erstellen.

Für Logout: docker logout git-registry.gibb.ch

Nun kann der push gemacht werden:

docker push [image tag inkl. GitLab-Pfad]

Hier noch die Erklärung und Beispiel:

```
docker push pfad-zu-registry/order-für-image/name-des-image:v1
```

Hier ein Beispiel:

```
docker push git-registry.gibb.ch/thomas.staub/cloudmodule/to-do-app/redis-master:v1
```

Hier halte ich noch meine Befehle und die Prozesse detailliert fest:

```
git-registry.gibb.ch/jve161514/m347-dienst-mit-container/to-do-app/redis-master:v1
                                                    776eba237309       43.2MB         11.9MB    U
git-registry.gibb.ch/jve161514/m347-dienst-mit-container/to-do-app/redis-slave:v1
                                                    79ebd6acf8e9       43.2MB         11.9MB    U
git-registry.gibb.ch/jve161514/m347-dienst-mit-container/to-do-app/todo-app:v1
                                                    2683d9b0649a       20.3MB         5.93MB    U
redis-master:v1                                     776eba237309       43.2MB         11.9MB    U
redis-slave:v1                                      79ebd6acf8e9       43.2MB         11.9MB    U
todo-app:v1                                         2683d9b0649a       20.3MB         5.93MB    U
jon@jon-ubuntu-vbox:~$ docker push git-registry.gibb.ch/jve161514/m347-dienst-mit-container/to-do-app/todo-app:
v1
The push refers to repository [git-registry.gibb.ch/jve161514/m347-dienst-mit-container/to-do-app/todo-app]
4f4fb700ef54: Pushed
213ec9aee27d: Pushed
e9062ddda238: Pushed
94dce707e05a: Pushed
9f274cb9924d: Pushed
af600f324a8c: Pushed
v1: digest: sha256:2683d9b0649a4d0df62b07754ff95d24cbba8f9447d9bab17e4306e5af77252c size: 856
jon@jon-ubuntu-vbox:~$ docker push ^[[200~git-registry.gibb.ch/jve161514/m347-dienst-mit-container/to-do-app/re
dis-slave:v1
invalid reference format
jon@jon-ubuntu-vbox:~$ docker push git-registry.gibb.ch/jve161514/m347-dienst-mit-container/to-do-app/redis-sla
ve:v1
The push refers to repository [git-registry.gibb.ch/jve161514/m347-dienst-mit-container/to-do-app/redis-slave]
140a95e714ff: Pushed
ca7dd9ec2225: Pushed
731cc432e6da: Pushed
862de9590cc6: Pushed
83276aa4de36: Pushed
4b937ee5a2e0: Pushed
a26b23e71d57: Pushed
e6e39c545f09: Pushed
e4ba39acee61: Pushed
v1: digest: sha256:79ebd6acf8e9074291933e8d6aa128bc7936d911f130e006a052e19a1d6da3e2 size: 856
jon@jon-ubuntu-vbox:~$ docker push git-registry.gibb.ch/jve161514/m347-dienst-mit-container/to-do-app/redis-mas
ter:v1
The push refers to repository [git-registry.gibb.ch/jve161514/m347-dienst-mit-container/to-do-app/redis-master]
862de9590cc6: Mounted from jve161514/m347-dienst-mit-container/to-do-app/redis-slave
ca7dd9ec2225: Mounted from jve161514/m347-dienst-mit-container/to-do-app/redis-slave
83276aa4de36: Mounted from jve161514/m347-dienst-mit-container/to-do-app/redis-slave
731cc432e6da: Mounted from jve161514/m347-dienst-mit-container/to-do-app/redis-slave
a26b23e71d57: Mounted from jve161514/m347-dienst-mit-container/to-do-app/redis-slave
4b937ee5a2e0: Mounted from jve161514/m347-dienst-mit-container/to-do-app/redis-slave
097b13e9e544: Pushed
799932282273: Pushed
d700073e051c: Pushed
v1: digest: sha256:776eba2373095f524d904e7033a9a2522038d5bc73588bd7d7fe11d9edf94e4f size: 856
jon@jon-ubuntu-vbox:~$
```

Abbildung 5 Meine Befehle für das Pushen auf GitLab

Das hat tiptop funktioniert.

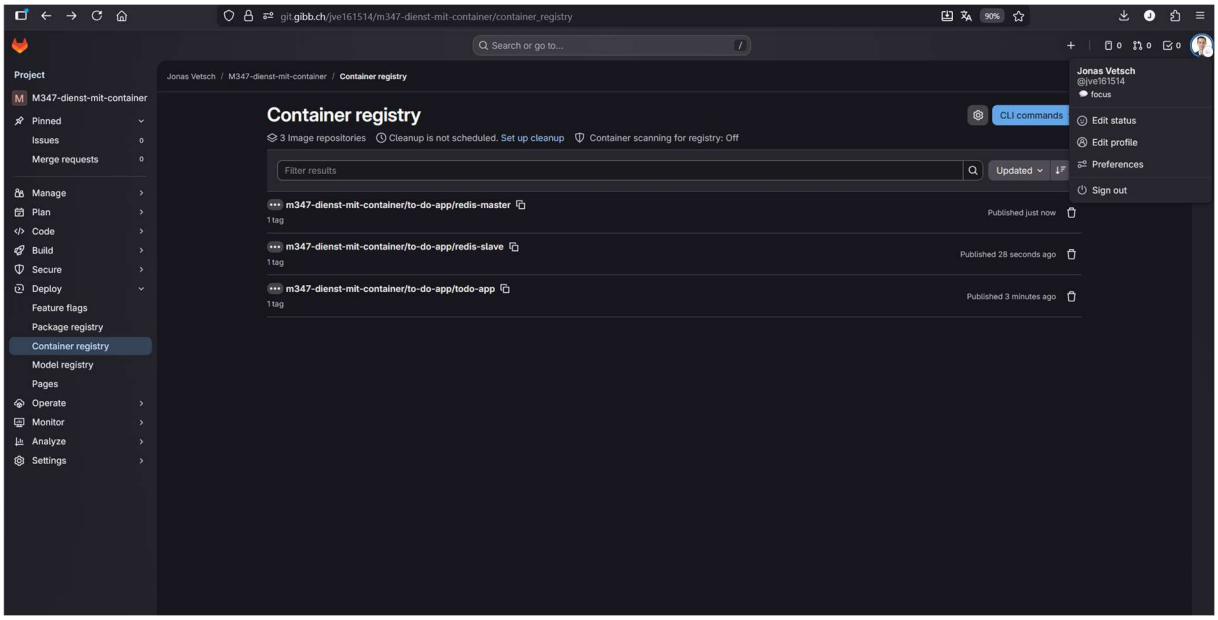


Abbildung 6 PrintScreen der Images bei (gibb) GitLab

4.3 Transferleistung: ToDo-App v2

Nun lernen wir noch, mit verschiedenen Versionen umzugehen. Dabei nutzen wir einfach einen neuen Tag :v2 beim Erstellen des Images.

Danach wieder wie gewohnt im Dockernetzwerk ausführen. Die App läuft wieder 😊

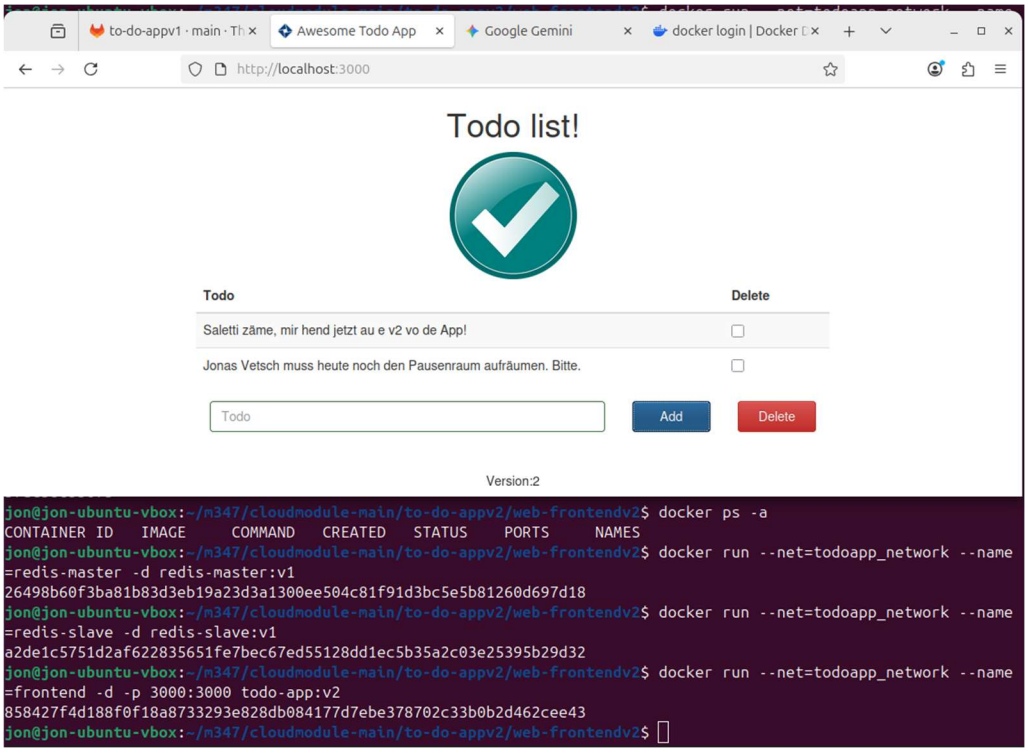


Abbildung 7 PrintScreen Version 2 mit meinem Namen als Todo Task

Das neue Image kann nun wieder als Vorbereitung für den Push korrekt getaggt werden:

```
jon@jon-ubuntu-vbox:~/m347/cloudmodule-main/to-do-appv2/web-frontendv2$ docker image tag todo-app:v2 git-regist
ry.gibb.ch/jve161514/m347-dienst-mit-container/to-do-app/todo-app:v2
jon@jon-ubuntu-vbox:~/m347/cloudmodule-main/to-do-appv2/web-frontendv2$ docker im
image   (Manage images)
images  (List images)
import  (Import the contents from a tarball to create a filesystem image)
jon@jon-ubuntu-vbox:~/m347/cloudmodule-main/to-do-appv2/web-frontendv2$ docker image ls
                                                                                  i Info →  U  In Use
IMAGE                                                                             ID             DISK USAGE   CONTENT SIZE   EXTRA
git-registry.gibb.ch/jve161514/m347-dienst-mit-container/to-do-app/redis-master:v1
                                                                                  776eba237309       43.2MB         11.9MB    U
git-registry.gibb.ch/jve161514/m347-dienst-mit-container/to-do-app/redis-slave:v1
                                                                                  79ebd6acf8e9       43.2MB         11.9MB    U
git-registry.gibb.ch/jve161514/m347-dienst-mit-container/to-do-app/todo-app:v1
                                                                                  2683d9b0649a       20.3MB         5.93MB
git-registry.gibb.ch/jve161514/m347-dienst-mit-container/to-do-app/todo-app:v2
                                                                                  0c4d8b8ec83f       20.5MB         6.03MB    U
redis-master:v1                                                                   776eba237309       43.2MB         11.9MB    U
redis-slave:v1                                                                    79ebd6acf8e9       43.2MB         11.9MB    U
todo-app:v1                                                                       2683d9b0649a       20.3MB         5.93MB
todo-app:v2                                                                       0c4d8b8ec83f       20.5MB         6.03MB    U
```

Dann der Push:

```
jon@jon-ubuntu-vbox:~/m347/cloudmodule-main/to-do-appv2/web-frontendv2$ docker push git-registry.gibb.ch/jve161
514/m347-dienst-mit-container/to-do-app/todo-app:v2
The push refers to repository [git-registry.gibb.ch/jve161514/m347-dienst-mit-container/to-do-app/todo-app]
17d9b665ac39: Pushed
4f4fb700ef54: Layer already exists
213ec9aee27d: Layer already exists
e9062ddda238: Layer already exists
f953a62fa210: Pushed
c0c377f61613: Pushed
v2: digest: sha256:0c4d8b8ec83f07ea7efdfd0c553e1c67f6d1ea9b6e61fc9e53afae54f0a07456 size: 856
```

Und die Kontrolle:

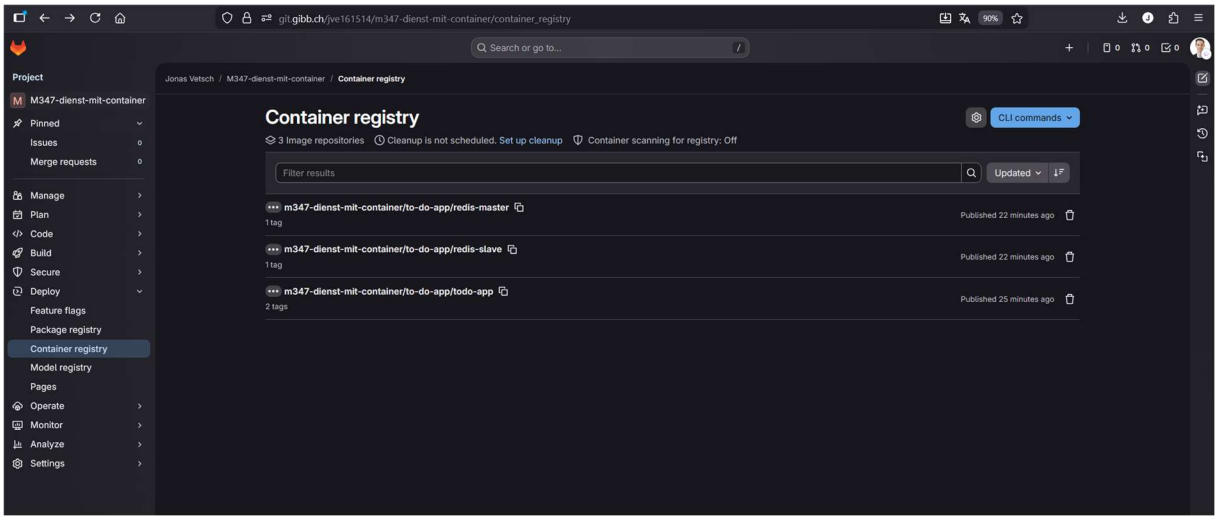


Abbildung 8 GitLab zeigt nun zwei Tags an beim Frontend

5 Docker Compose

5.1 Was ist Docker Compose?

Das manuelle Erstellen, Ausführen, Beenden und Aufräumen von Containern ist mühsam. Das Aufräumen (**`docker system prune -a --volumes`**) geht zudem auch gerne vergessen! Deshalb gibt es eine Möglichkeit, dies zu automatisieren. Docker Compose ist ein Tool von Docker selbst, das es erlaubt, die Containerverwaltung quasi zu skripten. Man erstellt dazu ein .yaml-File und schreibt dort drin die Instruktionen, wie die Container sich verhalten sollen.

Wenn eine Anwendung also mehrere Container braucht, so ist Docker Compose der «way to go», da man damit mehrere Container orchestrieren kann. Unter macOS und Windows kommt docker compose direkt bei Docker mitgeliefert. Unter Linux muss es separat installiert werden:

```
sudo apt install docker compose
```

5.2 Arbeiten mit Docker Compose

Zuerst muss die Datei **`docker-compose.yaml`** angelegt werden. Docker Compose erkennt Abhängigkeiten unter den Containern und weiss, welchen er zuerst starten muss. Docker Compose legt auch automatisch ein Dockernetzwerk für die Container an, damit sie miteinander kommunizieren können.

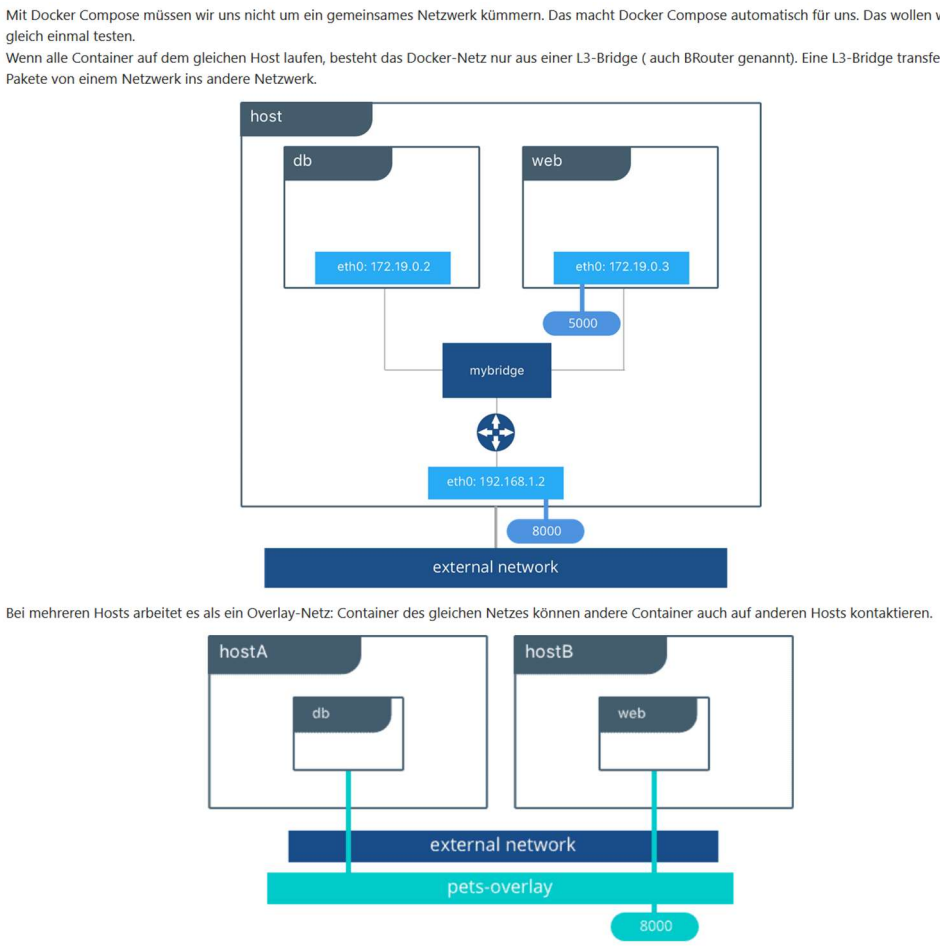


Abbildung 9 Docker-Compose Netzwerk (Quelle: smartlearn.gibb.ch)

5.3 Docker Netzwerke

```yaml
version: "3"
services:
  todoapp:
    build: ./web-frontend
    ports:
      - "3000"

    depends_on:
      - redis-master
      - redis-slave
    networks:
      - todoapp_network
  redis-slave:
    build: ./redis-slave
    depends_on:
      - redis-master
    networks:
      - todoapp_network
  redis-master:
    build: ./redis-master
    networks:
      - todoapp_network
networks:
  todoapp_network:
    name: todoapp_network
    driver: bridge
```

docker-compose.yaml

```yaml
version: "3"
services:
  todoapp:
    build: ./web-frontend
    ports:
      - "3000"
    depends_on:
      - redis-master
      - redis-slave
    networks:
      - todoapp_network
  redis-slave:
    build: ./redis-slave
    depends_on:
      - redis-master
    networks:
      - todoapp_network
  redis-master:
    build: ./redis-master
    networks:
      - todoapp_network
networks:
  todoapp_network:
    name: todoapp_network
    driver: bridge
```

Abbildung 10 Angepasste Datei bei mir auf der VM

Im Beispiel der ToDo-App kann in der docker-compose.yaml ganz unten ein Netzwerk definiert werden. An dieses Netzwerk werden alle zu erstellenden Container angeschlossen. Es gibt drei Arten von Docker-Netzwerken.

5.3.1 Bridge-Netzwerk (Standard)

Hier wird ein Netzwerk innerhalb des Computers (Hosts) angelegt, in dem die Container sich mit ihrem eigenen Servicenamen ansprechen können (brauchen ihre IP-Adresse nicht zu kennen). Z.B. «todoapp» kann einfach «redis-master» aufrufen.

5.3.2 BRouter (L3-Bridge)

Dieser Router ist die Brücke zwischen der Container-Welt und der Aussenwelt. Wenn in der docker-compose.yaml z.B. 3000:3000 gemappt wird, dann wird der Router Anfragen zwischen diesen Ports vermitteln.

5.3.3 Overlay-Netzwerk (mehrere Hosts)

Wird das Container-Konzert zu aufwendig für einen einzelnen Host, so muss man es auf mehrere Maschinen aufsplitten. Mit einem Overlay-Netzwerk wird ein virtueller Switch zwischen zwei oder mehrere Hosts gehängt, auch wenn diese sich in anderen physischen Netzen befinden.

5.4 Was sind Docker Volumes?

Mit Docker Volumes kann man physikalische Verzeichnisse aus dem Gast-System in den Container einbinden. Z.B. will man, dass eine Datenbank persistent bleibt nach dem Neustart eines

Containers. Dazu wird die Datenbank auf einem physikalischen Verzeichnis abgelegt und dieses in den Container eingebunden.

```
docker run -v /home/mount/data:/var/lib/mysql/data  (Hostdir:Containerdir)
```

Zuerst wird das Host-Directory angegeben und danach mit «:» abgetrennt das Container-Directory.

5.5 Was ist YAML für ein Dateiformat?

YAML ist eine vereinfachte Auszeichnungssprache angelehnt an XML. YAML ist ein rekursives Akronym für „YAML Ain't Markup Language" (ursprünglich „Yet Another Markup Language").

Es muss mit Eindrücken gearbeitet werden. Es muss mit Leerzeichen eingerückt werden, nicht mit Tabulator!

5.6 Starten und Stoppen von Containern mit docker compose

Die Container erhalten je einen Unterordner im Projektordner. Im Projektordner selbst wird dann die docker-compose.yaml angelegt. Danach kann das Containerorchester gestartet werden mit:

```
docker compose -f docker-compose.yaml up -d
```

`-d` Container werden im Hintergrund ausgeführt

Und stoppen mit:

```
docker compose -f docker-compose.yaml down
```

➔ Hiermit werden die Container nach dem Herunterfahren auch direkt entfernt, sehr praktisch!

5.6.1 docker-compose.yaml mit lokalen Dateien

Das docker-compose.yaml gitb eine «build:»-Anweisung und verweist auf die Unterordner, die die entsprechenden Containerdaten enthalten.

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

5.6.2 Docker compose mit Dateien auf einem Remote

Wenn die Ordner der Container lokal nicht vorhanden sind, können wir statt mit «build:» einen Unterordner einzubinden auch direkt ein Image auf einem Remote einbinden.

```
version: "3"
services:
  todoapp:
    image: git-registry.gibb.ch/thomas.staub/cloudmodule/todo-app:v1
    ports:
     - "3000"
    depends_on:
     - redis-master
     - redis-slave
  redis-slave:
    image: git-registry.gibb.ch/thomas.staub/cloudmodule/redis-slave:v1
    depends_on:
     - redis-master
  redis-master:
    image: git-registry.gibb.ch/thomas.staub/cloudmodule/redis-master:v1
```

5.7 Logs anzeigen

Danach mit

`docker logs [erste zwei Zeichen der Container ID] -f`

die logs live auf der Konsole ausgeben lassen.

5.8 Übungsaufgabe: V1 der To-Do App mit Docker Compose

Zunächst arbeite ich im v1 Ordner der ToDo-App. Hier muss noch das Netzwerk in der docker-compose.yaml ergänzt werden:

```
version: "3"
services:
  todoapp:
    build: ./web-frontend
    ports:
     - "3000"
    depends_on:
     - redis-master
     - redis-slave
    networks:
     - todoapp_network
  redis-slave:
    build: ./redis-slave
    depends_on:
     - redis-master
    networks:
     - todoapp_network
  redis-master:
    build: ./redis-master
    networks:
      - todoapp_network
networks:
  todoapp_network:
    name: todoapp_network
    driver: bridge
```

Nun kann das Containerorchester gestartet werden mit

`docker compose -f docker-compose.yaml up -d`

Hier hatte ich leider einen Fehler. Offenbar ist docker-compose veraltet und es wird geraten, unter Ubuntu **`docker compose up -d`** zu verwenden. Offenbar liefert das docker Kommando heutzutage das compose bereits mit. Deshalb habe ich nun auch hier in der Dokumentation die Befehle nochmal umgestellt und mit Abstand statt Bindestrich geschrieben.

Mit docker compose up -d wird die Compose-Datei automatisch erkannt, deshalb muss ich kein **`-f [filename of compose file] mitgeben.`**

```
jon@jon-ubuntu-vbox:~/m347_Docker/cloudmodule-main/to-do-appv1$ docker compose up -d
WARN[0000] /home/jon/m347_Docker/cloudmodule-main/to-do-appv1/docker-compose.yaml: the
attribute `version` is obsolete, it will be ignored, please remove it to avoid potentia
l confusion
[+] up 4/4
 ✔ Network todoapp_network              Created                                    0.1s
 ✔ Container to-do-appv1-redis-master-1 Created                                    0.3s
 ✔ Container to-do-appv1-redis-slave-1  Created                                    0.2s
 ✔ Container to-do-appv1-todoapp-1      Created                                    0.2s
```

Abbildung 11 To-Do-App v1 mit Docker Compose gestartet

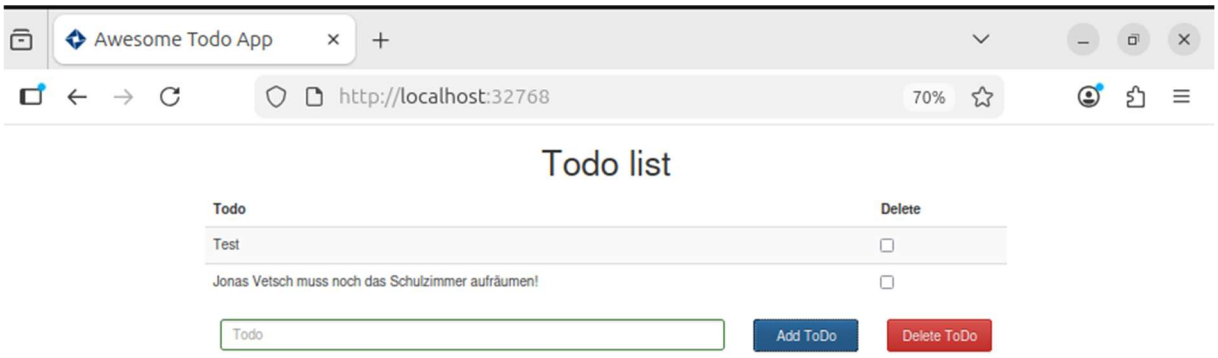


Abbildung 12 Die To-Do-App funktioniert mit Docker Compose.

Nun wollen wir natürlich noch die Logs im Frontendcontainer anschauen:

```
jon@jon-ubuntu-vbox:~/m347_Docker/cloudmodule-main/to-do-appv1$ docker ps
CONTAINER ID   IMAGE                      COMMAND                  CREATED
 STATUS              PORTS                                          NAMES
1954e76f0639   to-do-appv1-todoapp        "./todo-app"             About a minute ago
 Up About a minute   0.0.0.0:32768->3000/tcp, [::]:32768->3000/tcp   to-do-appv1-todoap
p-1
ff7e6176b43f   to-do-appv1-redis-slave    "docker-entrypoint.s…"   About a minute ago
 Up About a minute   6379/tcp                                        to-do-appv1-redis-
slave-1
83751c60d5f3   to-do-appv1-redis-master   "docker-entrypoint.s…"   About a minute ago
 Up About a minute   6379/tcp                                        to-do-appv1-redis-
master-1
jon@jon-ubuntu-vbox:~/m347_Docker/cloudmodule-main/to-do-appv1$ docker logs 19 -f
[negroni] listening on :3000
[negroni] Started GET /
[negroni] Completed 200 OK in 12.143163ms
[negroni] Started GET /script.js
[negroni] Completed 200 OK in 1.79101ms
[negroni] Started GET /read/todo
[negroni] Completed 200 OK in 83.787185ms
[negroni] Started GET /favicon.ico
[negroni] Completed 200 OK in 8.831849ms
[negroni] Started GET /
[negroni] Completed 304 Not Modified in 160.914µs
[negroni] Started GET /read/todo
[negroni] Completed 200 OK in 106.280268ms
[negroni] Started GET /insert/todo/Test
[negroni] Completed 200 OK in 178.389712ms
[negroni] Started GET /read/todo
[negroni] Completed 200 OK in 86.80377ms
[negroni] Started GET /insert/todo/Jonas Vetsch muss noch das Schulzimmer aufräumen!
[negroni] Completed 200 OK in 62.945431ms
[negroni] Started GET /read/todo
[negroni] Completed 200 OK in 57.809582ms
[negroni] Started GET /read/todo
[negroni] Completed 200 OK in 30.344639ms
[negroni] Started GET /read/todo
[negroni] Completed 200 OK in 28.796956ms
[negroni] Started GET /read/todo
```

Abbildung 13 Logs an schauen klappt wunderbar.

5.9 Übung: Version 2 der To-Do App mit Docker Compose

https://smartlearn.gibb.ch/exam-passes/90587/assignments/103780

Gemäss dieser Aufgabe will ich nun die zweite Version der To-Do App mit Docker Compose laufenlassen.

5.9.1 Die Version 1 herunterfahren

Alle laufenden Container herunterfahren und löschen mit

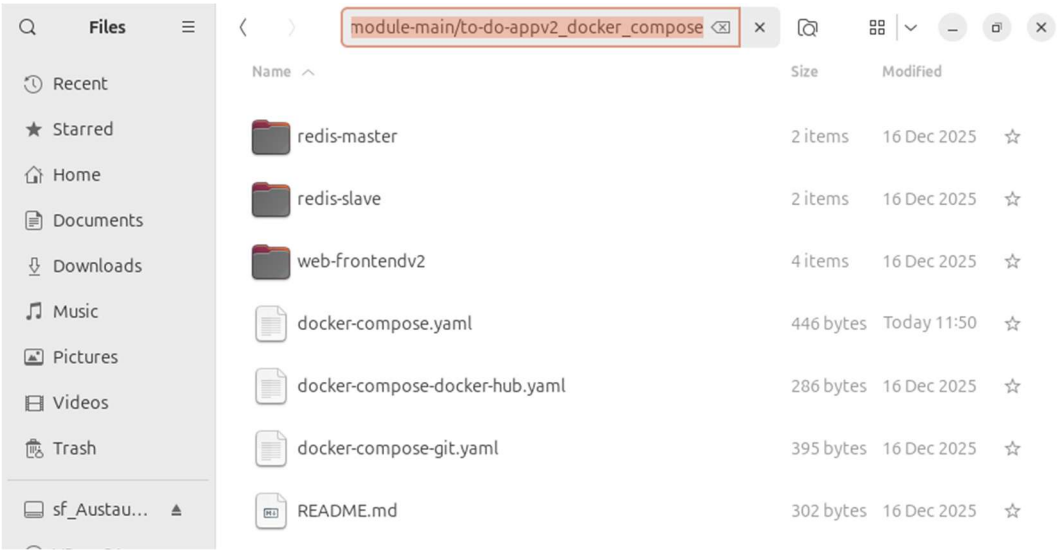
```
docker stop $(docker ps -q) && docker rm $(docker ps -aq)
```

```
^Cjon@jon-ubuntu-vbox:~/m347_Docker/cloudmodule-main/to-do-appv1$ docker ps
CONTAINER ID   IMAGE                      COMMAND                  CREATED          STA
TUS          PORTS                                          NAMES
1954e76f0639   to-do-appv1-todoapp        "./todo-app"             14 minutes ago   Up
14 minutes   0.0.0.0:32768->3000/tcp, [::]:32768->3000/tcp   to-do-appv1-todoapp-1
ff7e6176b43f   to-do-appv1-redis-slave    "docker-entrypoint.s…"   14 minutes ago   Up
14 minutes   6379/tcp                                       to-do-appv1-redis-slave-1
83751c60d5f3   to-do-appv1-redis-master   "docker-entrypoint.s…"   15 minutes ago   Up
14 minutes   6379/tcp                                       to-do-appv1-redis-master-1
jon@jon-ubuntu-vbox:~/m347_Docker/cloudmodule-main/to-do-appv1$ docker stop $(docker ps
 -q) && docker rm $(docker ps -aq)
1954e76f0639
ff7e6176b43f
83751c60d5f3
1954e76f0639
ff7e6176b43f
83751c60d5f3
```

Abbildung 14 Ergebnis.

5.9.2 Ordnerstruktur vorbereiten

Bei der V2 der To-Do-App ändert sich nur das Frontend. Ich lege einen neuen, leeren Ordner an und kopiere die zwei Backend-Ordner von der v1 der App (**`redis-master und redis-slave`**). Zudem übernehme ich ebenfalls die Dateien **`docker-compose*`**



Zudem kopiere ich **`web-frontendv2`** in den Ordner. Damit sind alle Teile nun zusammen. Jeder Container hat seinen eigenen Unterordner.

5.9.3 Docker Compose schreiben

Nun muss die **`docker-compose.yaml`** angepasst werden:

~/m347_Docker/cloudmodule-main/to-do-appv2_docker_compose/docker-compose.yaml

File Edit Selection Find View Goto Tools Project Preferences Help

docker-compose.yaml

```yaml
version: "3"
services:
  todoapp:
    build: ./web-frontendv2
    ports:
      - "3000"
    depends_on:
      - redis-master
      - redis-slave
    networks:
      - todoapp_network
  redis-slave:
    build: ./redis-slave
    depends_on:
      - redis-master
    networks:
      - todoapp_network
  redis-master:
    build: ./redis-master
    networks:
      - todoapp_network
networks:
  todoapp_network:
    name: todoapp_network
    driver: bridge
```

Abbildung 15 Angepasste Docker Compose Datei

5.9.4 Das noch bestehende Netzwerk entfernen

Das Netzwerk vom Auftritt des vorherigen Dockerorchesters ist noch aktiv. Das muss zuerst entfernt werden, da wir sonst einen Fehler bekommen beim Starten des to-do-appv2 Orchesters:

```
jon@jon-ubuntu-vbox:~/m347_Docker/cloudmodule-main/to-do-appv2_docker_compose$ docker n
etwork ls
NETWORK ID     NAME              DRIVER    SCOPE
c04c458d34dd   bridge            bridge    local
9b75265ce516   host              host      local
9d2d3148986c   none              null      local
378d3b4e7e06   todoapp_network   bridge    local
jon@jon-ubuntu-vbox:~/m347_Docker/cloudmodule-main/to-do-appv2_docker_compose$ docker n
etwork rm 378d3b4e7e06
378d3b4e7e06
```

Abbildung 16 Bestehendes Netzwerk entfernen.

5.9.5 Ready to launch?

Bereit zu lift-off? Ich bin’s, ja.

1. Sicherstellen, dass ich im Ordner des v2-Orchesters bin.
2. Das Orchester starten mit: **`docker compose up -d`**

```
jon@jon-ubuntu-vbox:~/m347_Docker/cloudmodule-main/to-do-appv2_docker_compose$ docker c
ompose up -d
WARN[0000] /home/jon/m347_Docker/cloudmodule-main/to-do-appv2_docker_compose/docker-com
pose.yaml: the attribute `version` is obsolete, it will be ignored, please remove it to
 avoid potential confusion
[+] Building 8.3s (28/28) FINISHED
 => [internal] load local bake definitions                                    0.0s
 => => reading from stdin 1.81kB                                              0.0s
 => [redis-slave internal] load build definition from Dockerfile              0.1s
 => => transferring dockerfile: 172B                                          0.0s
```

Abbildung 17 Die Images werden gebaut und danach als Container ausgeführt.

```
 => => unpacking to docker.io/library/to-do-appv2_docker_compose-todoapp:latest   0.2s
 => [todoapp] resolving provenance for metadata file                          0.7s
 => [redis-slave] resolving provenance for metadata file                      0.6s
 => [redis-master] resolving provenance for metadata file                     0.0s
[+] up 7/7
 ✔ Image to-do-appv2_docker_compose-todoapp                Built              8.5s
 ✔ Image to-do-appv2_docker_compose-redis-slave            Built              8.5s
 ✔ Image to-do-appv2_docker_compose-redis-master           Built              8.5s
 ✔ Network todoapp_network                                 Created            0.3s
 ✔ Container to-do-appv2_docker_compose-redis-master-1     Created            0.7s
 ✔ Container to-do-appv2_docker_compose-redis-slave-1      Created            2.0s
 ✔ Container to-do-appv2_docker_compose-todoapp-1          Created            1.4s
```

Abbildung 18 Der Prozess ist Abgeschlossen.

```
jon@jon-ubuntu-vbox:~/m347_Docker/cloudmodule-main/to-do-appv2_docker_compose$ docker p
s
CONTAINER ID   IMAGE                                       COMMAND                  CREAT
ED         STATUS         PORTS                                          NAMES
fa9749c3431a   to-do-appv2_docker_compose-todoapp          "./todo-app"             2 min
utes ago   Up 2 minutes   0.0.0.0:32769->3000/tcp, [::]:32769->3000/tcp  to-do-appv2_d
ocker_compose-todoapp-1
a4f85336f2a2   to-do-appv2_docker_compose-redis-slave      "docker-entrypoint.s…"   2 min
utes ago   Up 2 minutes   6379/tcp                                       to-do-appv2_d
ocker_compose-redis-slave-1
f60ce422a18a   to-do-appv2_docker_compose-redis-master     "docker-entrypoint.s…"   2 min
utes ago   Up 2 minutes   6379/tcp                                       to-do-appv2_d
ocker_compose-redis-master-1
```

Abbildung 19 Die Container laufen.

```
jon@jon-ubuntu-vbox:~/m347_Docker/cloudmodule-main/to-do-appv2_docker_compose$ docker logs
-f fa
[negroni] listening on :3000
[negroni] Started GET /
[negroni] Completed 200 OK in 10.035353ms
[negroni] Started GET /script.js
[negroni] Completed 200 OK in 1.056ms
[negroni] Started GET /checkmark.png
[negroni] Completed 200 OK in 5.52249ms
[negroni] Started GET /favicon.ico
[negroni] Completed 200 OK in 3.594591ms
[negroni] Started GET /read/todo
[negroni] Completed 200 OK in 64.764656ms
[negroni] Started GET /read/todo
[negroni] Completed 200 OK in 42.869974ms
[negroni] Started GET /read/todo
[negroni] Completed 200 OK in 38.066291ms
[negroni] Started GET /read/todo
[negroni] Completed 200 OK in 42.367955ms
[negroni] Started GET /read/todo
[negroni] Completed 200 OK in 61.009709ms
[negroni] Started GET /read/todo
[negroni] Completed 200 OK in 247.616107ms
[negroni] Started GET /insert/todo/This App was brought to you by Container Orchester for T
oDo App v2 !
[negroni] Completed 200 OK in 97.221015ms
[negroni] Started GET /read/todo
[negroni] Completed 200 OK in 36.004017ms
[negroni] Started GET /read/todo
[negroni] Completed 200 OK in 33.180474ms
[negroni] Started GET /insert/todo/Jonas Vetsch thinks that container ochrestras are amazin
g! Even though they're called Docker Compose.
[negroni] Completed 200 OK in 47.024413ms
[negroni] Started GET /read/todo
[negroni] Completed 200 OK in 27.287267ms
```

Abbildung 20 Siehe da, der erste User hat schon ToDos erfasst!

Hier oben sieht man, dass die App die ToDos mit simplem GET-Requests übermittelt: Unverschlüsselt und direkt in der URL. Also bitte keine sensiblen Daten übermitteln 😊

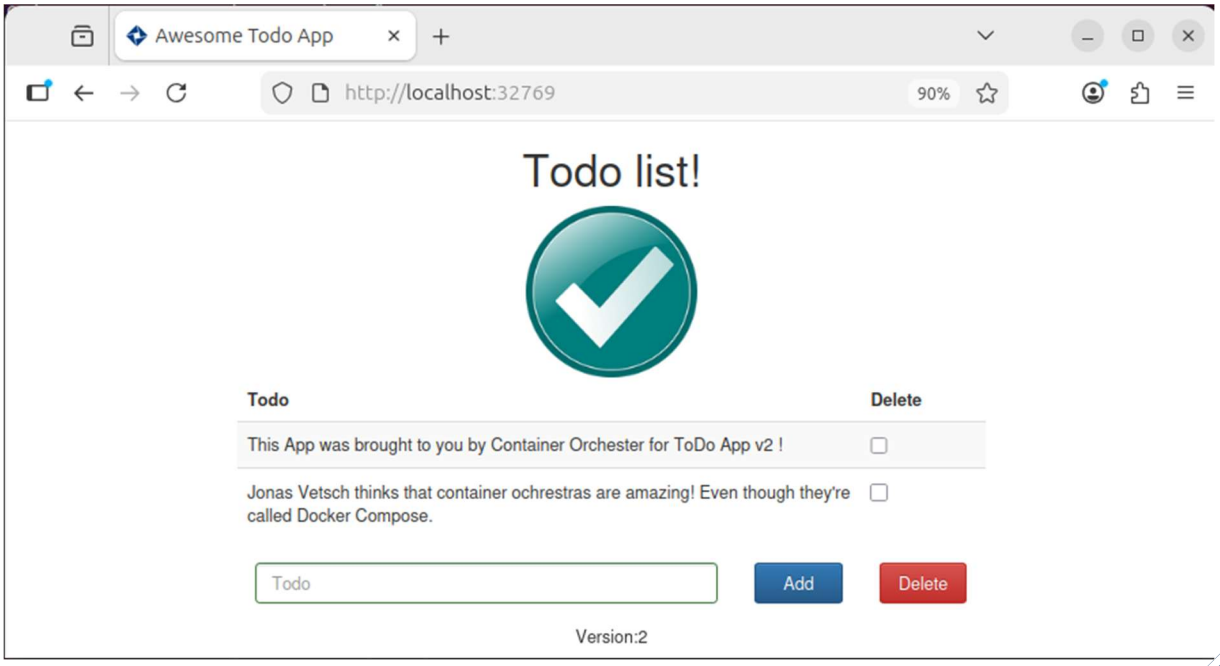


Abbildung 21 Frontend.

6 Portainer

6.1 Wozu Portainer?

Portainer ist eine Weboberfläche zur Verwaltung von Container. Wenn viele Container gleichzeitig laufen und verwaltet werden müssen wird das Terminal oft zu unübersichtlich.

Es ist ein GUI-Interface für Docker. Es zeigt auch Logs und Ressourcenverbrauch der Container an. Zudem gibt es Templates für bekannte Applikationen (Nextcloud, Wordpress, MariaDB etc.).

Und: Portainer ist letztlich selbst auch ein Container 😉

6.2 Portainer installieren und ausführen.

1. Leerer Ordner mit folgender **`docker-compose.yaml`** anlegen:

~/m347_Docker/cloudmodule-main/portainer_docker_compose/docker-compose.yaml

File Edit Selection Find View Goto Tools Project Preferences Help

index.html | docker-compose.yaml

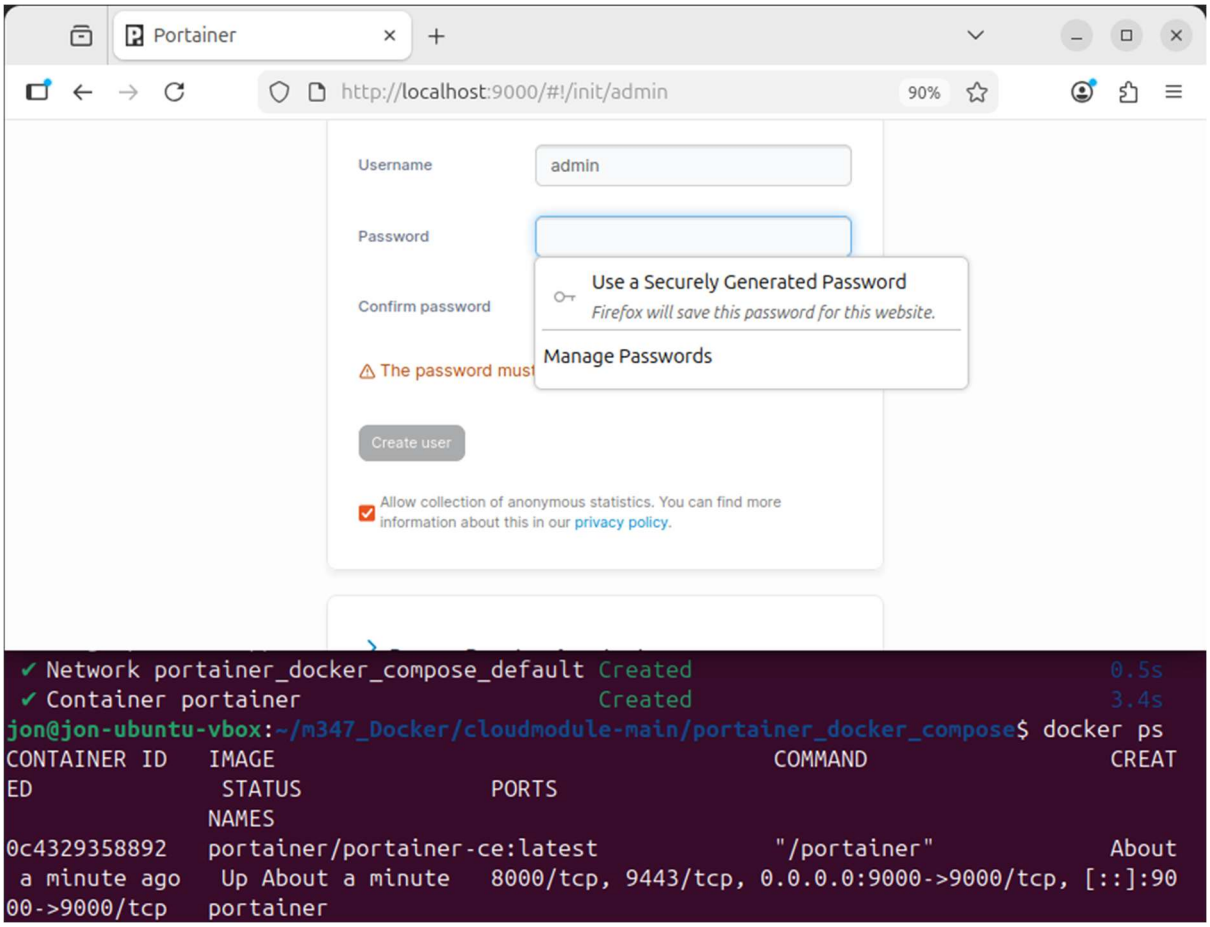
```yaml
version: '3'

services:
  portainer:
    image: portainer/portainer-ce:latest
    container_name: portainer
    restart: unless-stopped
    security_opt:
      - no-new-privileges:true
    volumes:
      - /etc/localtime:/etc/localtime:ro
      - /var/run/docker.sock:/var/run/docker.sock:ro
      - ./portainer-data:/data
    ports:
      - 9000:9000
```

2. **`docker compose up -d`**

```
jon@jon-ubuntu-vbox:~/m347_Docker/cloudmodule-main/portainer_docker_compose$ docker com
pose up -d
WARN[0000] /home/jon/m347_Docker/cloudmodule-main/portainer_docker_compose/docker-compo
se.yaml: the attribute `version` is obsolete, it will be ignored, please remove it to a
void potential confusion
[+] up 13/13
 ✔ Image portainer/portainer-ce:latest      Pulled                          43.3s
 ✔ Network portainer_docker_compose_default Created                         0.5s
 ✔ Container portainer                      Created                         3.4s
```

Abbildung 22 Portainer wird heruntergeladen und ausgeführt.

3. Adminbenutzer anlegen im Web-UI des Containers



Mein Passwort hier nur testweise: sml123456789

4. Portainer Web-UI ist online!

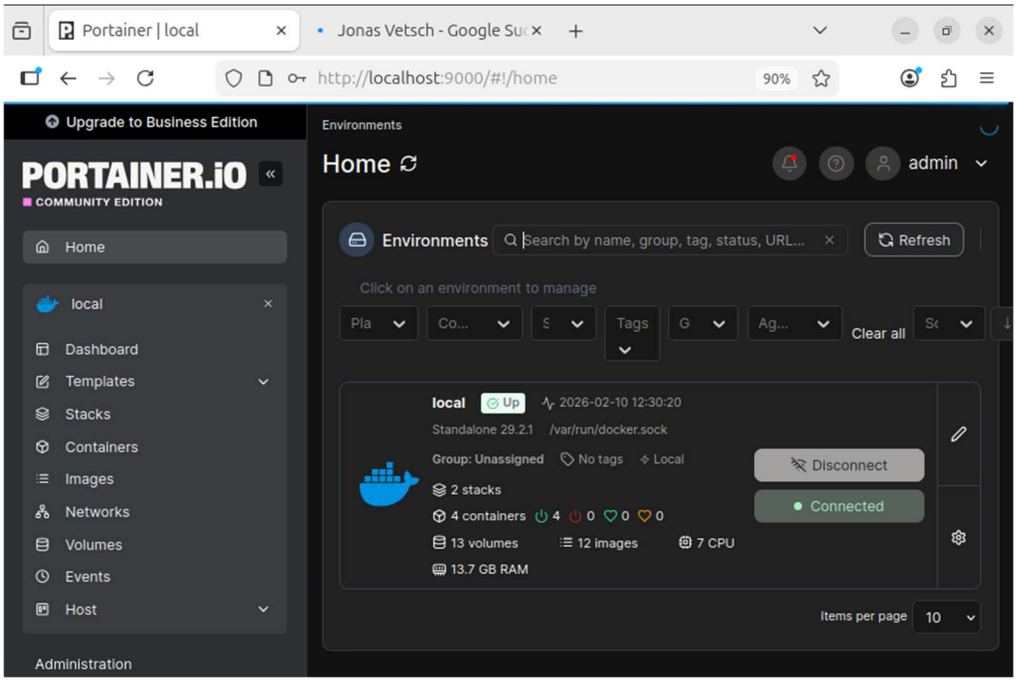


Abbildung 23 Mein Portainer Web-UI.

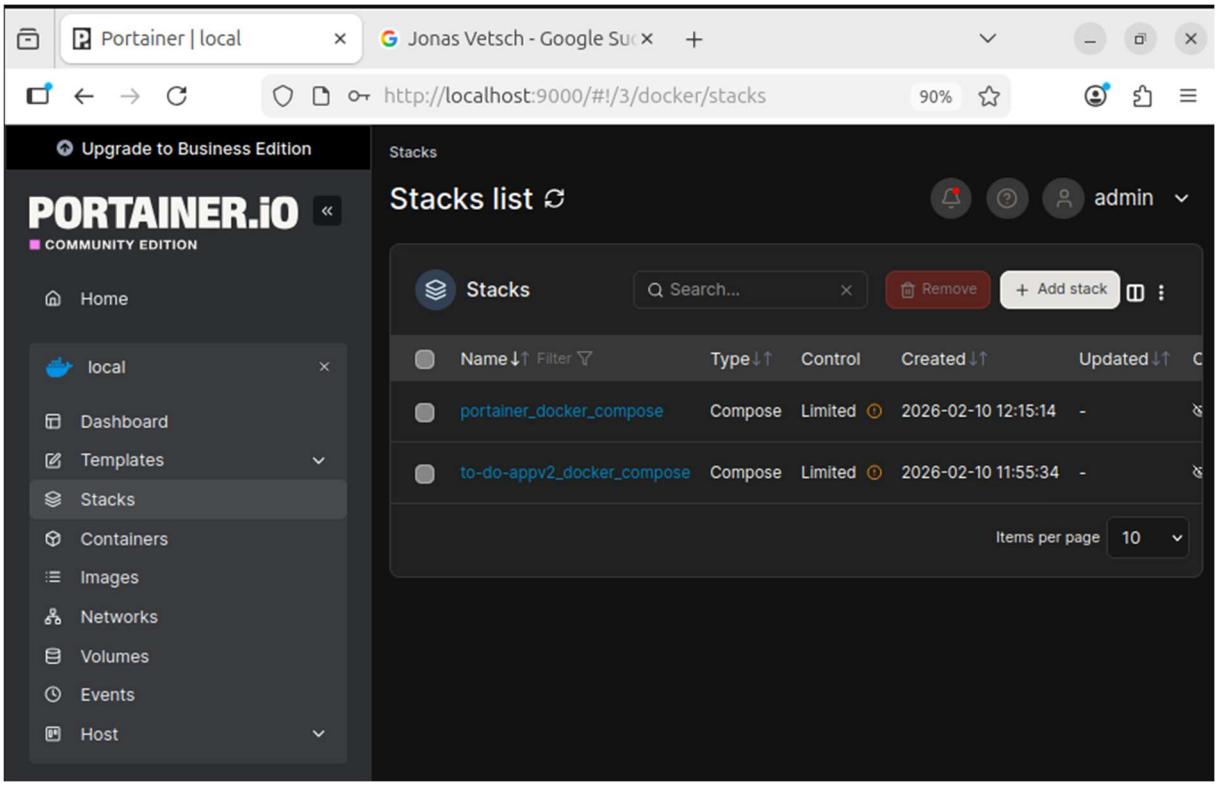
6.3 To-Do-App v2 mit Portainer installieren

Anstelle eines “händischen” Docker Compose machen wir nun ein «Stack» (so heisst es in Portainer). Damit können wir ein Dockerorchester direkt aus dem Web-UI starten.

Der Vorteil: Aus Portainer gestartete Orchester können danach im Portainer-Web-UI verwaltet werden.

6.3.1 Stack vorbereiten in Portainer

Im Portainer zur lokalen Umgebung navigieren (wir arbeiten ja schliesslich nicht auf einem Server, sondern auf der VM). Dann zu «Stacks»:



Nun «Add Stack».

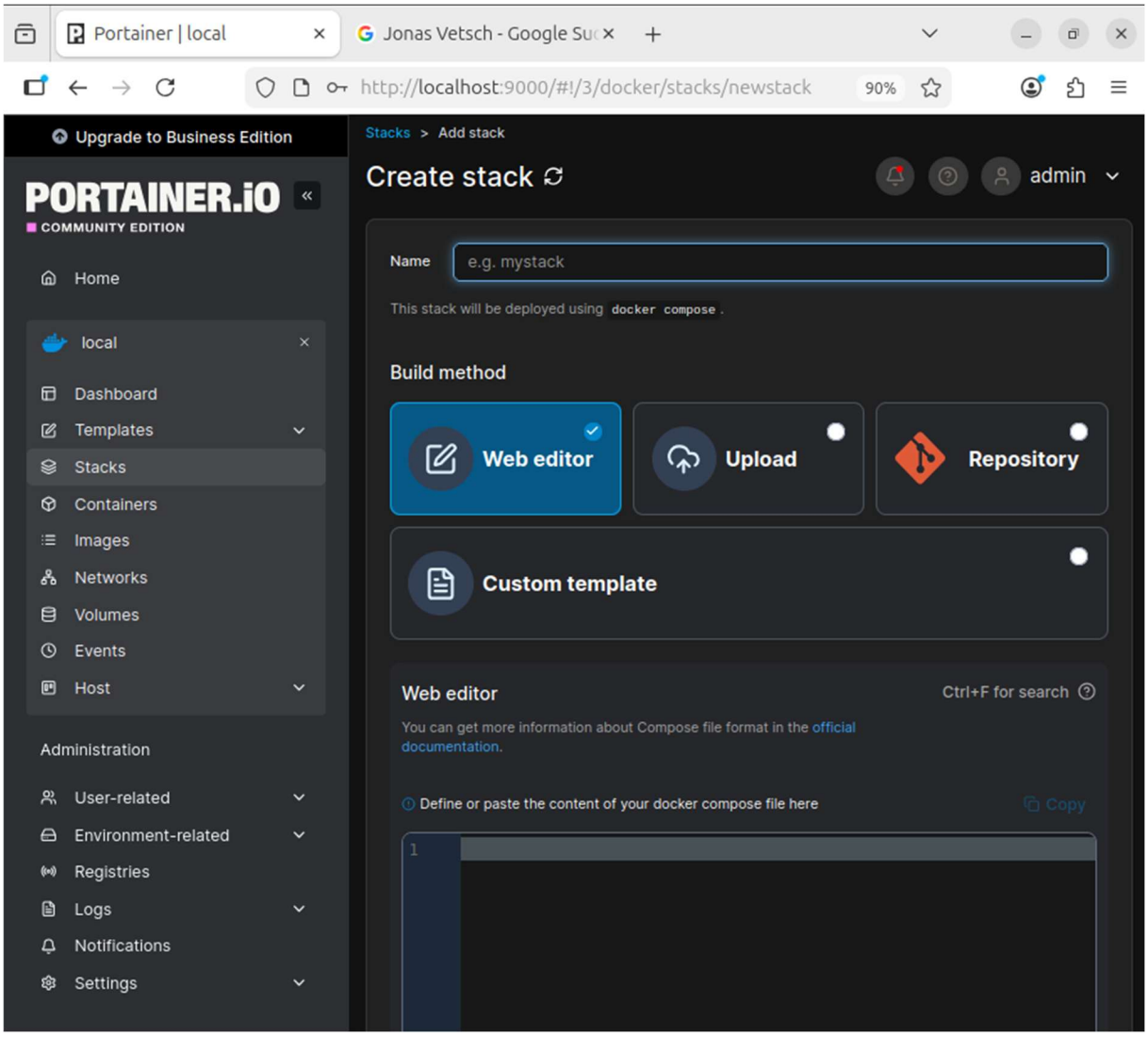


Abbildung 24 Stack erstellen: Es muss die Docker Compose angegeben werden.

Nun müssen wir die Docker Compose angeben, damit Portainer weiss, wie er das Orchester bauen soll.

Das Problem: Meine Dateien, die ich als Container bauen will, liegen aktuell nur lokal. Sie müssten mit absoluten Pfaden eingebunden werden:

```yaml
services:
  web:
    image: nginx
    volumes:
      - /home/jon/mein-projekt/html:/usr/share/nginx/html # Absoluter Pfad!
```

Abbildung 25 Beispiel von Einbindung mit absolutem Pfad

Eleganter: Die Container zuerst auf ein Online Registry pushen und dann per Link einbinden.

6.3.2 Container auf einem Remote vorbereiten

https://git.gibb.ch/jve161514/m347-dienst-mit-container.git

Ich habe mich dafür entschieden, auf dem bereits bestehenden GitLab Repo (GitLab von der gibb) die Files der Container hochzuladen inkl. docker-compose.yaml:

```
jon@jon-ubuntu-vbox:~/m347_Docker/cloudmodule-main/to-do-appv2_docker_compose$ git init
hint: Using 'master' as the name for the initial branch. This default branch name
hint: is subject to change. To configure the initial branch name to use in all
hint: of your new repositories, which will suppress this warning, call:
hint:
hint:   git config --global init.defaultBranch <name>
hint:
hint: Names commonly chosen instead of 'master' are 'main', 'trunk' and
hint: 'development'. The just-created branch can be renamed via this command:
hint:
hint:   git branch -m <name>
Initialized empty Git repository in /home/jon/m347_Docker/cloudmodule-main/to-do-appv2_docker_compose/.git/
jon@jon-ubuntu-vbox:~/m347_Docker/cloudmodule-main/to-do-appv2_docker_compose$ git config --global init.defaultB
ranch main
jon@jon-ubuntu-vbox:~/m347_Docker/cloudmodule-main/to-do-appv2_docker_compose$ git branch -m main
jon@jon-ubuntu-vbox:~/m347_Docker/cloudmodule-main/to-do-appv2_docker_compose$ git remote rename origin old-orig
in
error: No such remote: 'origin'
jon@jon-ubuntu-vbox:~/m347_Docker/cloudmodule-main/to-do-appv2_docker_compose$ git remote add origin git@git.gib
b.ch:jve161514/m347-dienst-mit-container.git
```

Abbildung 26 Remote verknüpfen.

```
jon@jon-ubuntu-vbox:~/m347_Docker/cloudmodule-main/to-do-appv2_docker_compose$ git commit -m "Add files to build
 and run todo app v2 with portainer."
[main (root-commit) 66d9d2c] Add files to build and run todo app v2 with portainer.
 15 files changed, 287 insertions(+)
 create mode 100644 to-do-app-portainer/README.md
 create mode 100644 to-do-app-portainer/docker-compose-docker-hub.yaml
 create mode 100644 to-do-app-portainer/docker-compose-git.yaml
 create mode 100644 to-do-app-portainer/docker-compose.yaml
 create mode 100644 to-do-app-portainer/redis-master/Dockerfile
 create mode 100644 to-do-app-portainer/redis-master/start-redis-master.sh
 create mode 100644 to-do-app-portainer/redis-slave/Dockerfile
 create mode 100644 to-do-app-portainer/redis-slave/start-redis-slave.sh
 create mode 100644 to-do-app-portainer/web-frontendv2/Dockerfile
 create mode 100644 to-do-app-portainer/web-frontendv2/bin/todo-app
 create mode 100644 to-do-app-portainer/web-frontendv2/main.go
 create mode 100644 to-do-app-portainer/web-frontendv2/public/checkmark.png
 create mode 100644 to-do-app-portainer/web-frontendv2/public/favicon.ico
 create mode 100644 to-do-app-portainer/web-frontendv2/public/index.html
 create mode 100644 to-do-app-portainer/web-frontendv2/public/script.js
jon@jon-ubuntu-vbox:~/m347_Docker/cloudmodule-main/to-do-appv2_docker_compose$ git push --set-upstream origin --
all
Enumerating objects: 23, done.
Counting objects: 100% (23/23), done.
Delta compression using up to 7 threads
Compressing objects: 100% (20/20), done.
Writing objects: 100% (23/23), 1.56 MiB | 15.77 MiB/s, done.
Total 23 (delta 1), reused 0 (delta 0), pack-reused 0
To git.gibb.ch:jve161514/m347-dienst-mit-container.git
 * [new branch]      main -> main
branch 'main' set up to track 'origin/main'.
jon@jon-ubuntu-vbox:~/m347_Docker/cloudmodule-main/to-do-appv2_docker_compose$ git push --set-upstream origin --
tags
Everything up-to-date
```

Abbildung 27 Commit und Push

Nun liegen die Files hier bereit: https://git.gibb.ch/jve161514/m347-dienst-mit-container.git

6.3.3 Stack erstellen in Portainer

Nun kann das Stack im Portainer erstellt werden. Weil das Repo privat ist, musste ich noch einen Personal Access Token generieren im GitLab.

Zudem ist es wichtig, den Pfad zum Compose (Compose path) anzugeben. Bei meinem Repo ist es nämlich in einem Unterordner und liegt nicht im Hauptordner.

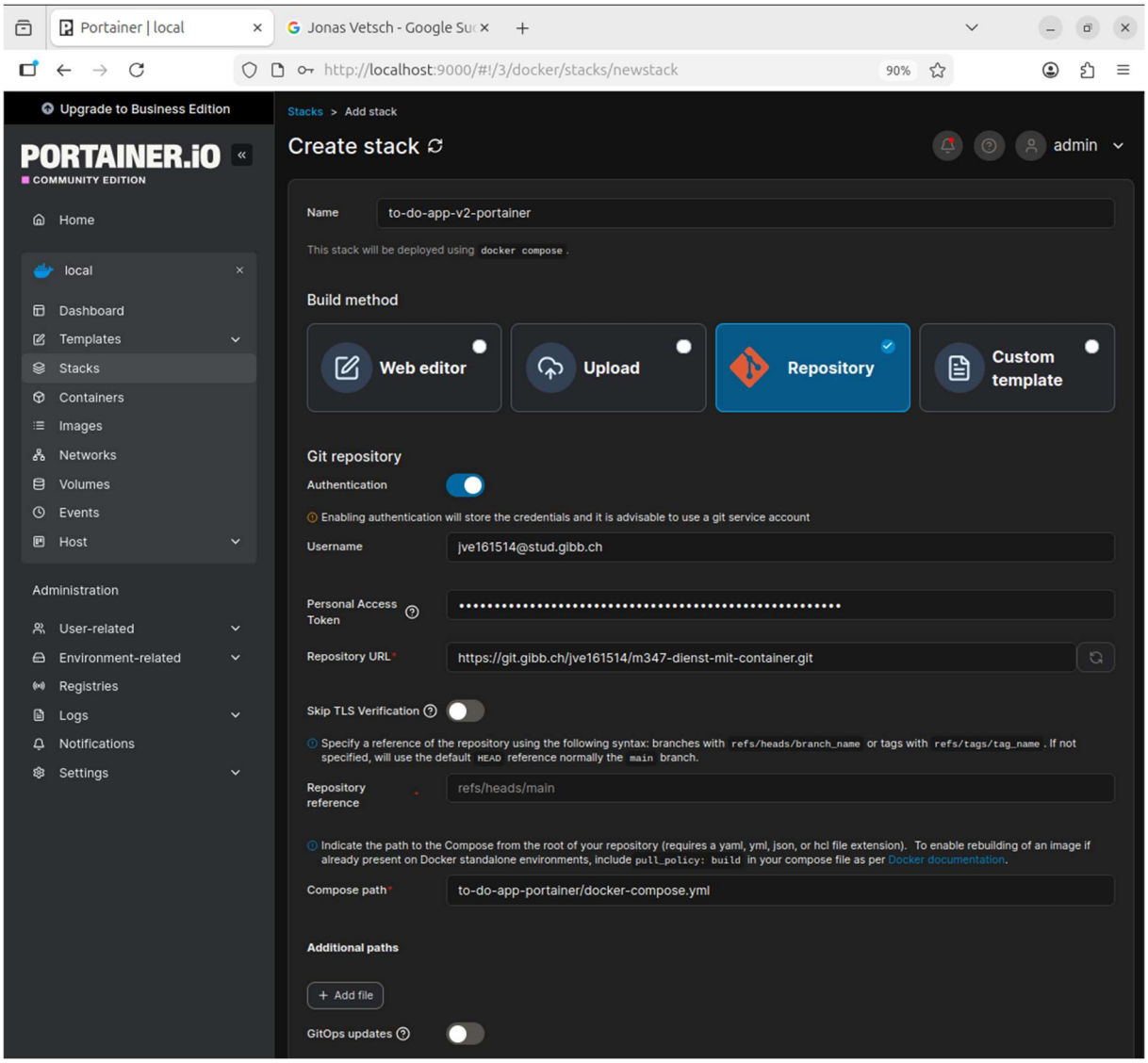


Abbildung 28 Stack anlegen in Portainer

Abbildung 29 Stack erstellen.

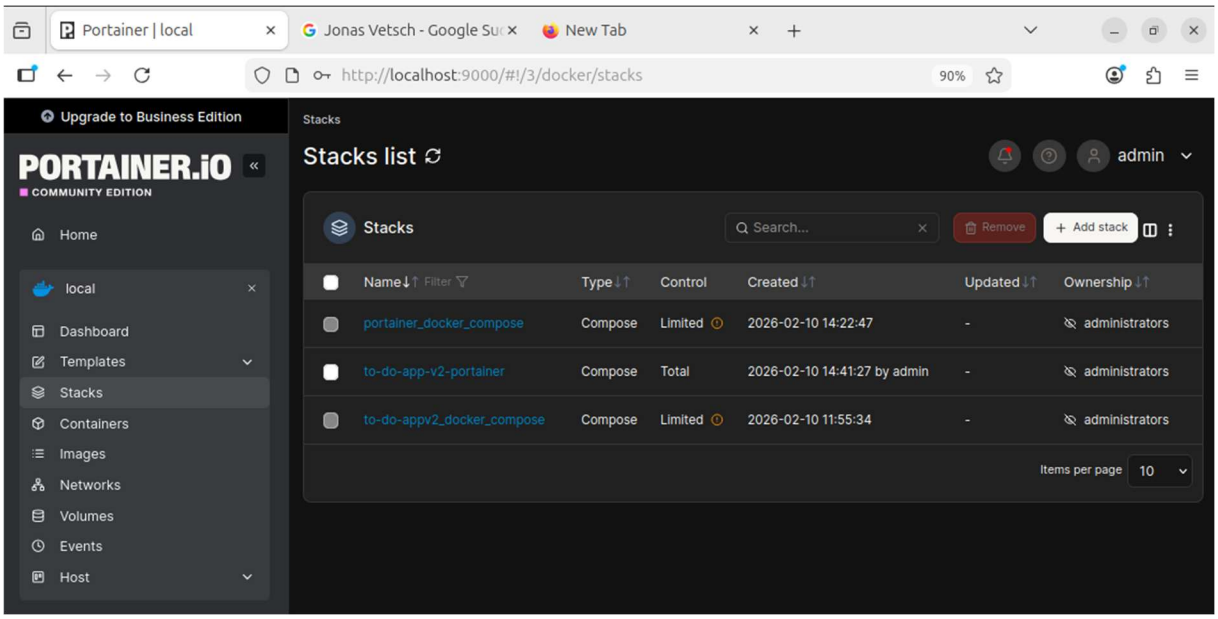
Nun kann das Stack mit dem Drücken auf «Deploy the stack» erstellt werden.

Containerlogs zeigen, was im Hintergrund nun passiert.

```
2026/02/10 02:41PM INF github.com/portainer/portainer/pkg/libstack/compose/logwriter.go:55 > level=warning msg="
Docker Compose is configured to build using Bake, but buildx isn't installed" |
#0 building with "default" instance using docker driver

#1 [redis-master internal] load build definition from Dockerfile
#1 transferring dockerfile:
#1 transferring dockerfile: 176B 0.2s done
#1 DONE 1.1s

#2 [redis-master internal] load metadata for docker.io/library/redis:alpine3.16
#2 DONE 5.7s

#3 [redis-master internal] load .dockerignore
#3 transferring context:
#3 transferring context: 2B 0.1s done
#3 DONE 1.1s

#4 [redis-master internal] load build context
#4 DONE 0.0s

#5 [redis-master 1/3] FROM docker.io/library/redis:alpine3.16@sha256:2700d5097763fda285c463f4eefc3d0730a2df2a9d4
8e66707b19d5a5e5f23d4
#5 resolve docker.io/library/redis:alpine3.16@sha256:2700d5097763fda285c463f4eefc3d0730a2df2a9d48e66707b19d5a5e5
f23d4
#5 resolve docker.io/library/redis:alpine3.16@sha256:2700d5097763fda285c463f4eefc3d0730a2df2a9d48e66707b19d5a5e5
f23d4 1.3s done
#5 DONE 1.4s

#4 [redis-master internal] load build context
#4 transferring context: 92B 0.1s done
```

Und schliesslich:

```
2026/02/10 02:42PM INF github.com/portainer/portainer/pkg/libstack/compose/composeplugin.go:156 > Stack deployme
nt successful |
```



Das Stack wurde erstellt.

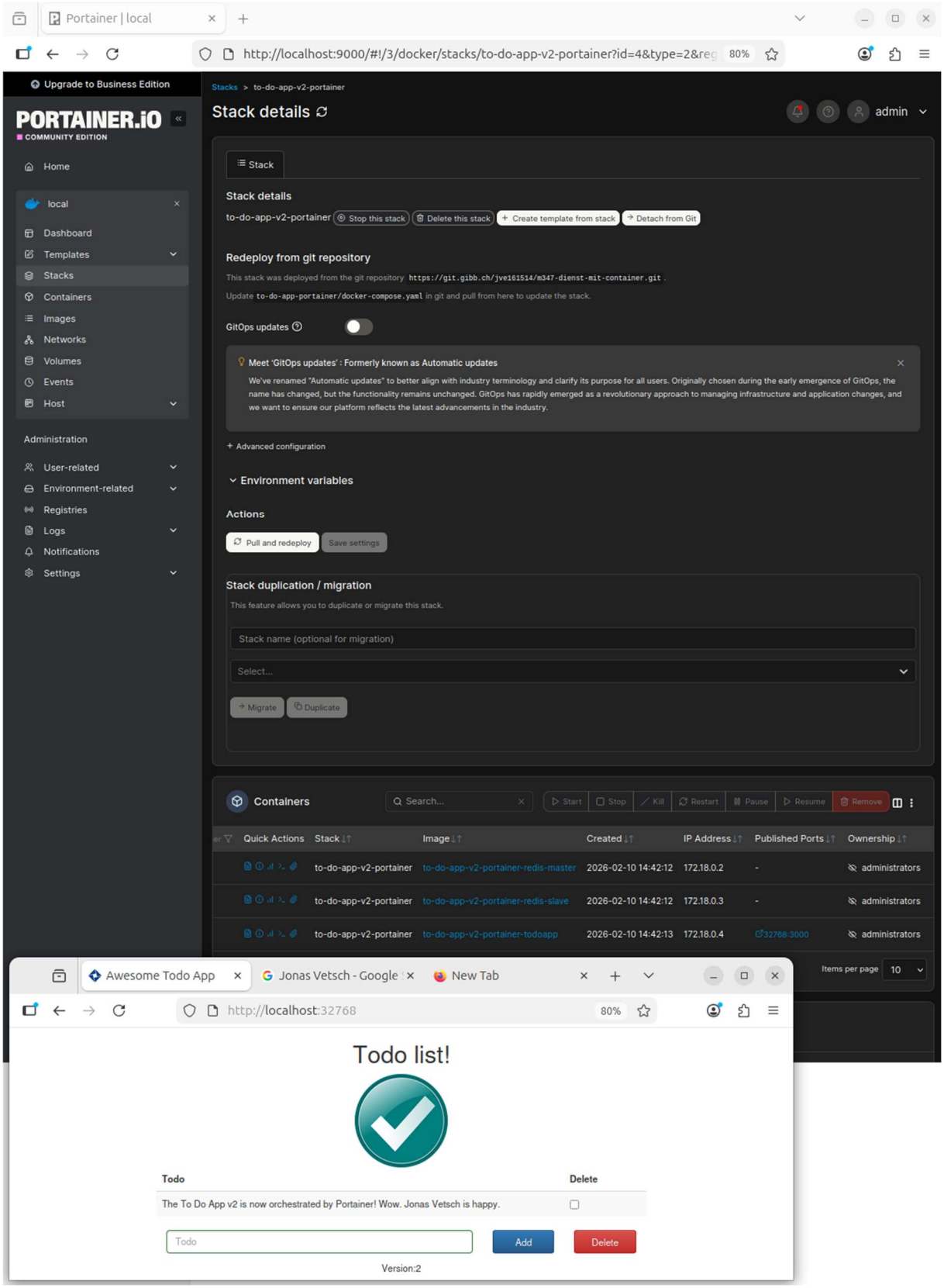


Abbildung 30 Stack verwalten und Web App

Das Stack kann nun in Portainer überwacht und verwaltet werden. Zudem läuft natürlich die To-Do-App v2 wieder im Browser, so wie es soll 😊

7 Lernbeispiel «Shop»

https://smartlearn.gibb.ch/exam-passes/90587/assignments/103782

Optional: Nicht bearbeitet, da Fokus auf Übungsprojekt.

8 Übungsprojekt

Als Freizeitprojekt entwickle ich aktuell eine Web-App, die den händisch geführten Ämtliplan in unserem Betrieb ersetzen soll.

Die Web-App ist noch in Entwicklung. Allerdings ist dieses Übungsprojekt hier perfekt dafür geeignet, das Deployment der App zu testen: Damit kann ich bereits jetzt prüfen, wie ich die Web-App in Zukunft dann containerisieren könnte. Ggf. ergeben sich daraus sogar wichtige Erkenntnisse zur Architektur der App, die ich bereits jetzt im weiteren Entwicklungsprozess beachten sollte.

8.1 Ziel

Bei der Web-App besteht aktuell erst die MySQL-Datenbank und eine erste, sehr simple Version des Backends als Java Springboot Applikation. Das Ziel ist es, dieses Backend inkl. Datenbank in einem Container zu deployen. Es soll in Swagger UI (API-Test-Konsole als Web-Interface, die von OpenAPI zur Verfügung gestellt wird) eine API-Anfrage zur Erstellung eines Teams an das Backend geschickt werden und das Backend soll das Team in der Datenbank anlegen.

8.2 Aufbau mit Docker

Es werden zwei Container benötigt: Ein MySQL-Container für die Datenbank und ein Container für die Java-Applikation. Wichtig ist, dass der My-SQL Container die Datenbank selbst und deren Tabellen anlegt beim ersten Start, da die Java-Applikation bereits angelegte Tabellen erwartet. Zudem darf sie erst starten, sobald dieser Prozess abgeschlossen ist, da sonst Spring Boot einen Fehler werfen würde, wenn es die Datenbank und die Tabellen nicht findet beim Start.

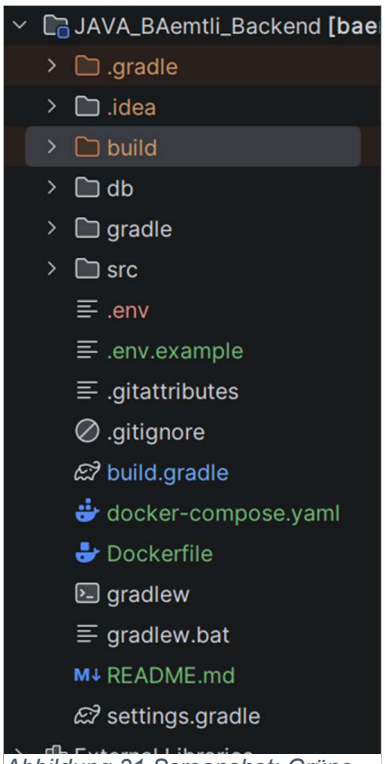


Abbildung 31 Screenshot: Grüne Dateien wurden hinzugefügt

Eines meiner wichtigsten Learnings während diesem kleinen Projekt war, dass ich die Dateien für Docker direkt im bestehenden Backenprojekt anlegen kann. Zunächst dachte ich nämlich, dass ich wohl einen separaten Projektordner anlegen sollte, in den ich dann mein Java-Projekt als .jar-Datei kopiere. In diesem separaten Projektordner wären dann die docker-compose und alles andere, was Docker braucht.

Nach einem Fachgespräch mit Gemini habe ich dann herausgefunden, dass es Best Practice ist, die Dockerdateien direkt im bestehenden Projekt anzulegen. So muss auch kein sepa-

rates Git Repository gemacht werden, denn in meinem Javaprojekt existiert ja bereits eine Versionskontrolle. Der Screenshot zeigt sehr schön, welche Dateien ich angelegt habe, um meinem bestehenden Projekt die Container-Fähigkeit als quasi «Add-on» zu geben.

8.3 Dockerfile

Als nächsten Schritt habe ich das Dockerfile geschrieben für den Build-Prozess der Backendapplikation.

```
# Instructions how to build the Java backend application of BAemtli
# (MySQL Database is built separately, see docker-compose.yaml)
# Java 21 Image
FROM eclipse-temurin:21-jre-slim

# the working directory
WORKDIR /app

# Copy the gradle build file
COPY build/libs/baemtli-app.jar app.jar

# Open port to Spring Boot Application
EXPOSE 8080

# start the app
ENTRYPOINT ["java", "-jar", "app.jar"]
```

Damit die Build-Datei von Gradle immer gleich heisst («baemtli-app.jar»), musste ich dies noch in build.gradle anpassen:

```
// Fixate name for build file so that docker-compose finds it
bootJar {
    archiveFileName = 'baemtli-app.jar'
}
```

8.4 docker-compose.yaml

Nun war ich bereit für die docker-compose.yaml. Hier habe binde ich das MySQL 8 Image von Docker Hub ein. Das ist der erste Container. Als zweiter Container wird die Java Applikation gebaut und eingebunden.

8.4.1 Datenbankcontainer

Eine besondere Herausforderung, beim Erstellen der docker-compose.yaml, war, zu verstehen, wie das Datenbankpasswort gesetzt wird. Ich habe herausgefunden, dass man dem

MySQL Image, wenn es gebaut wird, die Properties mitgeben kann über das docker-compose.yaml. Das Problem: Wenn ich hier ein Passwort festlege, dann wird dies in Git sichtbar sein. Deshalb habe ich eine .env Datei angelegt im Rootverzeichnis des Projekts. Die docker-compose.yaml nimmt nun die Properties für die Datenbank aus dem .env-File. In Git werde ich eine Beispiel-.env anlegen, damit Menschen, die mein docker-compose ausführen wollen, sehen, wie sie die .env-Datei aufbauen müssen, damit es funktioniert.

```
services:
  db:
    image: mysql:8.0
    environment: # uses variables from .env file
      - MYSQL_ROOT_PASSWORD=${DB_ROOT_PASSWORD} # root user password
      - MYSQL_DATABASE=Baemtli                  # Fallback in case init script fails to create the db
      - MYSQL_USER=baemtli_user                 # create a user for the backend
      - MYSQL_PASSWORD=${DB_USER_PASSWORD}      # set password for backend user
```

Abbildung 32 Datenbankproperties in docker-compose.yaml aus .env-File lesen

Dem DB-Container gebe ich danach noch die Initialisierungsskripte mit. In meinem Fall ist das nur ein .sql-File mit den Create Database und Create Table Befehlen, damit die Datenbank bereit ist. Zudem erstelle ich ein Laufwerk «mysql_data» auf dem Hostbetriebssystem, in dem die Datenbank ihre Daten persistent speichern kann.

Zudem habe ich einen Healthcheck eingebaut, der prüft, ob die Datenbank korrekt läuft. Bei diesem musste ich die Zeitangaben etwas tweaken, um die Startup Zeit des DB-Containers von ursprünglich 30 Sekunden auf 6 Sekunden zu senken.

```
services:
  db:
    image: mysql:8.0
    environment: # uses variables from .env file
      - MYSQL_ROOT_PASSWORD=${DB_ROOT_PASSWORD} # root user password
      - MYSQL_DATABASE=Baemtli                  # Fallback in case init script fails to create the db
      - MYSQL_USER=baemtli_user                 # create a user for the backend
      - MYSQL_PASSWORD=${DB_USER_PASSWORD}      # set password for backend user

    volumes:
      - ./db:/docker-entrypoint-initdb.d # executes .sql scripts to setup db and tables
      - mysql_data:/var/lib/mysql # make sure db data is persistent when docker container restarts
    ports:
      - "3306:3306" # Lets developer access the db from outside the container for debugging
    networks:
      - backend-net
    healthcheck:
      test: ["CMD", "mysqladmin" ,"ping", "-h", "localhost", "-u", "root", "-p${DB_ROOT_PASSWORD}"]
      interval: 2s # Check every 2 seconds
      timeout: 3s
      retries: 10
      start_period: 5s # give db some time to boot, before counting errors

```

8.4.2 Backendcontainer

Auf der Seite der Java-Applikation habe ich im docker-compose.yaml den Container-Port 8080 auf den Host-Port 8080 gemappt, damit man die API über das Swagger UI Web-Interface testen kann. Zudem habe ich spezifiziert, dass der Java-Container auf den Datenbank-Container warten soll, bis dieser «Healthy» ausgibt. Denn sonst würde Spring Boot die Datenbank nicht finden, wenn sie noch nicht bereit ist.

Danach übergebe ich noch Werte für die application.properties, wobei ich DB_PASSWORD wiederum aus .env ziehe.

Docker compose erstellt automatisch ein Dockernetzwerk für die Container. Ich wollte jedoch explizit eines anlegen, auch wenn es eigentlich redundant ist, einfach zu Übungszwecken.

```yaml
services:
  db:
    image: mysql:8.0
    environment: # uses variables from .env file
      - MYSQL_ROOT_PASSWORD=${DB_ROOT_PASSWORD} # root user password
      - MYSQL_DATABASE=Baemtli                  # Fallback in case init script fails to create the db
      - MYSQL_USER=baemtli_user                 # create a user for the backend
      - MYSQL_PASSWORD=${DB_USER_PASSWORD}      # set password for backend user

    volumes:
      - ./db:/docker-entrypoint-initdb.d # executes .sql scripts to setup db and tables
      - mysql_data:/var/lib/mysql # make sure db data is persistent when docker container restarts
    ports:
      - "3306:3306" # Lets developer access the db from outside the container for debugging
    networks:
      - backend-net
    healthcheck:
      test: ["CMD", "mysqladmin" ,"ping", "-h", "localhost", "-u", "root", "-p${DB_ROOT_PASSWORD}"]
      interval: 2s # Check every 2 seconds
      timeout: 3s
      retries: 10
      start_period: 5s # give db some time to boot, before counting errors


  backend:
    build: . # expects file "Dockerfile" in project root with build instructions
    ports:
      - "8080:8080" # Important to access Swagger UI API-Documentation for testing the API
    depends_on:
      db:
        condition: service_healthy # waits until db is up and running
    networks:
      - backend-net
    environment: # tell the spring app where to find the db, use variable from .env file
      - DB_HOST=db
      - DB_USER=baemtli_user
      - DB_PASSWORD=${DB_USER_PASSWORD}


volumes:
  mysql_data: # creates a persistent folder on the host system, gets reused when container restarts
    # no parameters needed, default settings are good enough for this project


networks:
  backend-net:
    driver: bridge
```

Abbildung 33 Die vollständige docker-compose.yaml

8.5 Ein Image bauen und auf GitHub veröffentlichen

Ein GitHub Repository bestand bereits für das Java-Projekt, ich musste lediglich meinen Berufsbildner noch bitten, das Repository öffentlich zu machen. Er hat zugestimmt.

Danach musste ich das Image bauen und pushen. Ich habe gemerkt, dass man den Tag-Namen für das Image auch einfach in der docker-compose.yaml mitgeben kann.

```
25  backend:
26    build: . # expects file "Dockerfile" in project root with build instructions
27    image: ghcr.io/ba-2025-2026/java_baemtli_backend:latest # Give a name for the image
```

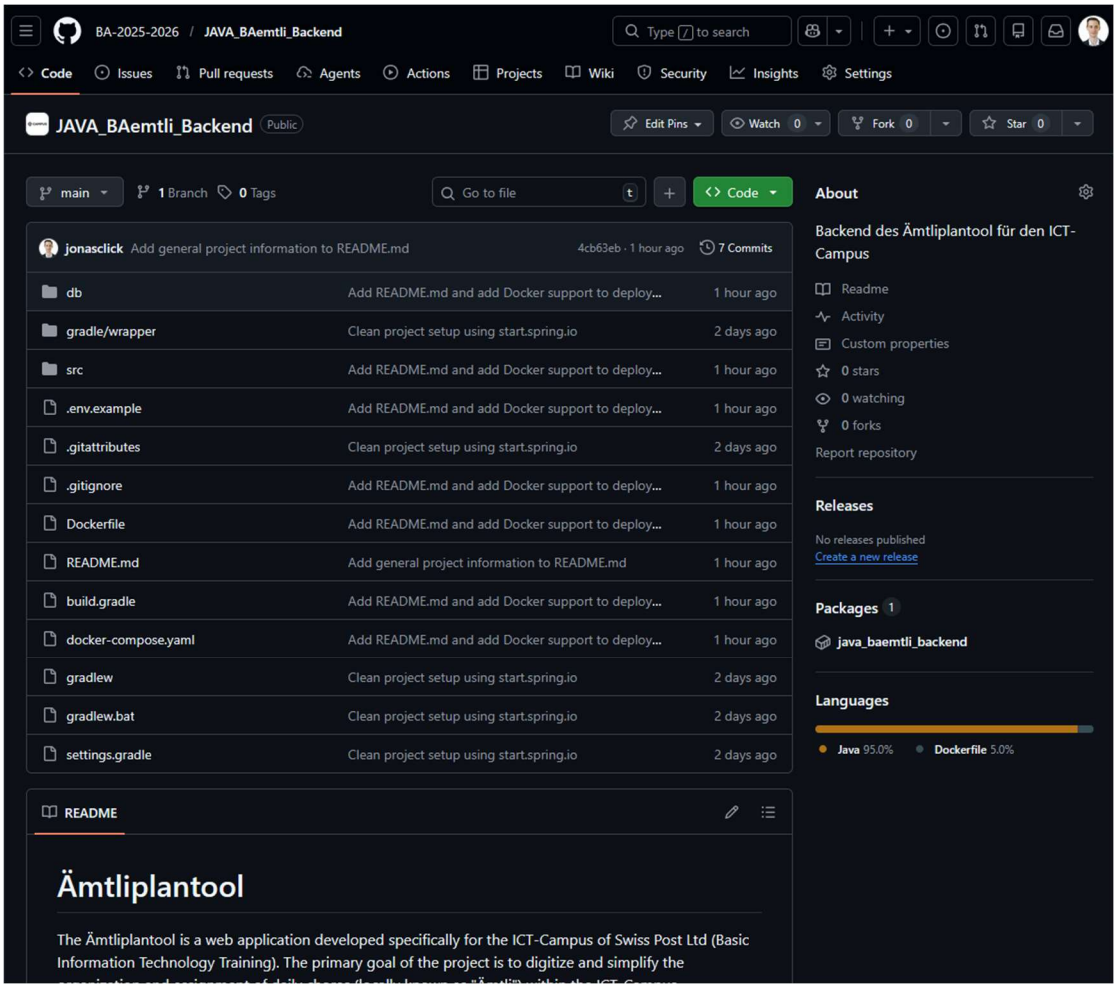
Danach einfach **`docker compose build`** und **`docker compose push`**

Falls Docker Desktop noch nicht mit meinem GitHub Profil eingeloggt ist:

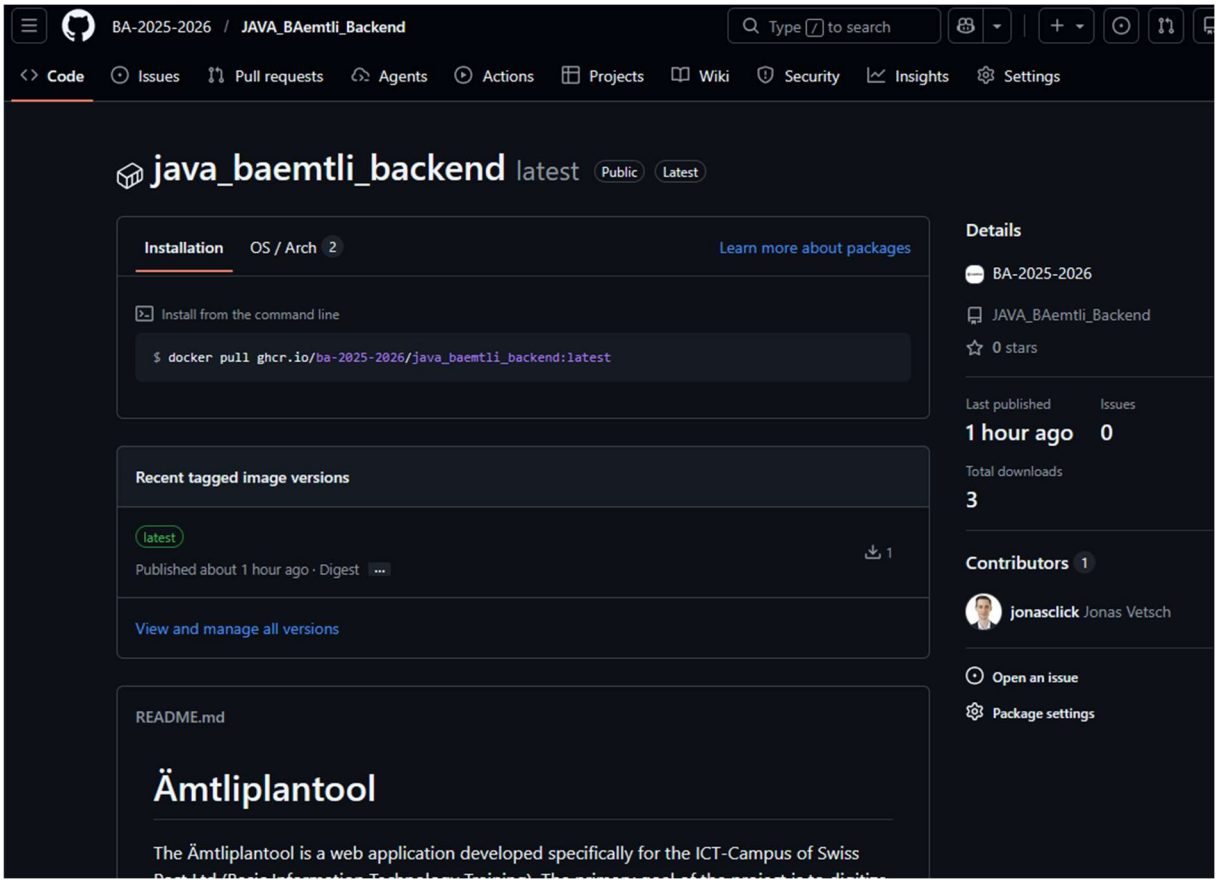
`echo "YOUR_GITHUB_PERSONAL_ACCESS_TOKEN" | docker login ghcr.io -u YOUR_GITHUB_USERNAME --password-stdin`

Auf GitHub muss das Image dann noch mit dem Repository verknüpft werden, auch wenn das eigentlich durch den Pfad des Image Tags schon klar ist. Es muss dennoch nochmal händisch verknüpft und auf öffentlich gestellt werden.

Danach ist es im Repository sichtbar im Aside unter «Packages»:



Wenn man das Package (Docker Image) dort im Repository anklickt sieht man noch Detailinformationen:



8.6 Dem Projekt ein README.md geben

Da nun einiges an Komplexität zum ehemals simplen Spring Boot Projekt dazugekommen ist, habe ich ein README.md geschrieben:

Ämtliplantool Backend

The Ämtliplantool is a web application developed specifically for the ICT-Campus of Swiss Post Ltd (Basic Information Technology Training). The primary goal of the project is to digitize and simplify the organization and assignment of daily chores (locally known as "Ämtli") within the ICT-Campus.

As of 2025, planning is often done manually or via decentralized lists. This tool provides a central platform for coaches and trainees to manage data and plan the monthly schedules.

For Users: Run this application without building

If you're just here to run this application, without building it.

1. Make sure you have Docker Desktop installed and running
2. In your terminal: Clone this repo with `git clone https://github.com/BA-2025-2026/JAVA_BAemtli_Backend.git`
3. Navigate to the project folder with `cd JAVA_BAemtli_Backend`
4. Create .env-file from template `cp ./.env.example ./.env`
5. Edit .env and choose a root password and a user password for your database
6. Run `docker compose pull` to pull the prebuilt docker images from the repo
7. Run `docker compose up -d` to start the application

For Developers: How to build and run?

You can build and run this project

- manually in your IDE
- or as a docker compose setup.

Build and run this project manually

1. install MySQL 8 server on your local machine
2. make a copy of `./src/main/resources/application.properties.example` and call it `application.properties`
3. in there: change the default values "pleasechange" to your database user and password
4. setup database and tables by running everything in `./db/create-database.sql`
5. run Spring Boot application
6. connect to the Swagger UI API-Interface by opening http://localhost:8080/swagger-ui/index.html
7. have fun!

Build and run this project with docker compose

1. Create .env-file from template `cp ./.env.example ./.env`
2. Edit .env and choose a root password and a user password for your database
3. to build the gradle project run in your terminal `.\gradlew clean bootJar`
4. start Docker desktop on your local machine
5. start the containers with `docker compose up --build -d`
6. connect to the Swagger UI API-Interface by opening http://localhost:8080/swagger-ui/index.html
7. have fun!

Re-Build image and push

If you make changes to this application that you would like to see reflected in the container image on GitHub: Rebuild and push as follows:

1. `docker compose build`
2. `docker compose push`

If Docker can't push to your GitHub account, you're maybe not logged in yet. To log in:

`echo "YOUR_GITHUB_PERSONAL_ACCESS_TOKEN" | docker login ghcr.io -u YOUR_GITHUB_USERNAME --password-stdin`

8.7 Installationsanleitung für die BAemtli-API

Nachfolgend finden sich die Instruktionen für die Lehrperson zum Ausführen des Projekts.

Diese Anleitung beschreibt die Schritte, um die Anwendung direkt über Docker zu starten, ohne den Quellcode manuell kompilieren zu müssen. Dabei wird mein vorkompiliertes Image von GitHub heruntergeladen.

8.7.1 Voraussetzungen

WICHTIG: Stellen Sie sicher, dass Docker Desktop auf Ihrem System installiert ist und bereits im Hintergrund ausgeführt wird. Stellen Sie sicher, dass keine Instanz von MySQL auf ihrem System läuft, da der MSQL-Port TCP 3306 für diese App frei sein muss.

8.7.2 Schritt-für-Schritt-Anleitung

1. Repository klonen: Öffnen Sie Ihr Terminal (Eingabeaufforderung oder PowerShell) und laden Sie das Projekt mit folgendem Befehl herunter:

   ```
   git clone https://github.com/BA-2025-2026/JAVA_BAemtli_Backend.git
   ```

2. In das Projektverzeichnis wechseln: Navigieren Sie mit dem cd-Befehl in den neu erstellten Ordner:

   ```
   cd JAVA_BAemtli_Backend
   ```

3. Konfigurationsdatei erstellen: Erstellen Sie eine Kopie der Datei **`./.env.example`** und benennen Sie diese in **`.env`** um.

4. Passwörter festlegen: Öffnen Sie die neu erstellte .env-Datei mit einem Texteditor. Legen Sie dort ein Root-Passwort und ein Benutzer-Passwort für die MySQL-Datenbank fest, die nachfolgend im Hintergrund erstellt wird.

5. Docker-Images herunterladen: Laden Sie die vorkonfigurierten Images direkt aus dem Repository herunter:

   ```
   docker compose pull
   ```

6. Anwendung starten: Starten Sie die Container im Hintergrund (Detached Mode) mit folgendem Befehl:

   ```
   docker compose up -d
   ```

7. Anwendung benutzen:

   Öffnen Sie im Browser: http://localhost:8080/swagger-ui/index.html

   Hier können Sie die API testen. Legen Sie ein neues Team an (POST /teams -> Try it out! -> Execute) und schauen Sie, ob sie den Status Code 201 Created zurückbekommen.

9 Kubernetes Grundlagen

9.1 Was ist Kubernetes?

Eigentlich wie ein Docker Compose, doch mit einem wichtigen Unterschied. Mit Docker Compose kann ich mehrere Container starten. Zudem kann ich ein Netzwerk definieren, in dem diese Container dann auch miteinander sprechen können. Wenn ich restart: unless stopped definiere in der docker-compose.yaml, dann sorgt Docker Compose auch direkt dafür, dass der Container stets wieder gestartet wird, sollte er einmal abstürzen oder der Host neu gestartet werden.

Docker Compose läuft auf einem Host. Wenn der Host abstützt oder ein Netzwerkproblem hat, dann ist meine Applikation dennoch unverfügbar. Hier kommt Kubernetes ins Spiel. Kubernetes verwaltet Container auf mehreren Hosts. Dabei werden immer so viel Instanzen von einem Container gestartet, wie gerade notwendig sind. Hat die Web-App mehr Traffic, können weitere Instanzen hochgefahren werden. Auch kontrolliert Kubernetes die Auslastung der an dieser wunderbaren Kubernetes-Symphonie beteiligten Server. Ein Server ist bereits voll ausgelastet? Dann startet Kubernetes weitere Container auf einem anderen Server. Fällt ein Server ganz aus, dann startet Kubernetes die Container, die darauf liefen, einfach auf einem anderen Server. Dieses System führt zu sehr hoher Verfügbarkeit, da eine Applikation damit sehr redundant betrieben wird.

9.2 Was sind Microservices?

Früher hat man Applikationen als eine grosse Code-Basis geschrieben. Frontend, Backend, Datenbank etc. waren eng miteinander verknüpft. Hat eine Web-App z.B. mehrere Teile (Authentifizierung, Datenbank, News-Feed, Warenkorb, Bestellungen etc.), so hat früher eine Applikation alles gesteuert. Problem: Applikationen wurden immer komplexer und dadurch entstehen sehr viele Abhängigkeiten. Wenn ein Teil der Applikation ein neues Framework einbauen will, geht das vielleicht nicht, weil ein anderer Teil der Applikation noch auf einer alten Technologie basiert, die nicht mit dem neuen gewünschten Framework kompatibel ist. Zudem: Wenn an einem Ende ein neues Feature gebaut wird (z.B. beim News-Feed) kann es sein, dass z.B. der Warenkorb plötzlich ein Bug hat, weil im Hintergrund die selbe Infrastruktur steht.

All diese Probleme löst das Microservice-Konzept: Jeder Teil einer Applikation läuft als eigenständiges Programm «ein Microservice» (eigene, abgeschottete Codebase, eigenes Team). Dafür müssen die Bereiche zunächst klar voneinander abgegrenzt werden. Danach ist aber jedes Team unabhängig von den anderen und kann ihre eigenen Frameworks etc. einsetzen, ohne dass in anderen Teams Bugs auftreten. Die Microservices kommunizieren untereinander z.B. über http (POST, GET, etc.) oder über andere Kommunikationsstandards wie MQTT (IoT-Bereich).

9.3 Wo kann ich Kubernetes herunterladen?

Mit Kubernetes verhaltet es sich ähnlich wie mit dem Linux-Kernel: Es ist Open Source und an sich noch nicht wirklich lauffähig. Es braucht also eine Kubernets-Distribution. Die Wahl der Distribution ist eine Wissenschaft für sich, da es sehr viele Distributionen gibt.

9.4 Lightweight Kubernetes Anwendungen

Die drei Cloud-Riesen haben je ihre eigene Kubernetes-Distribution

- GKE (Google Kubeneetes Engine)
- AKS (Azure Kubernetes Service)
- EKS (Amazon Elastic Kubernetes Service)

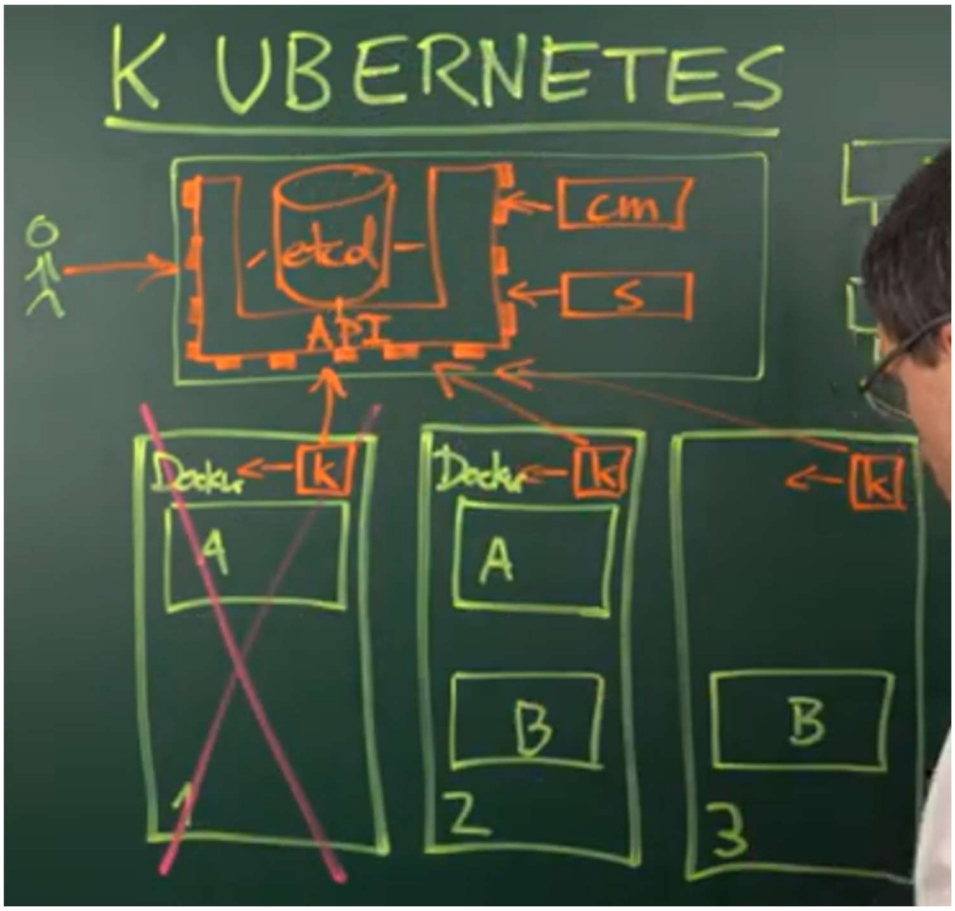
Diese sind aber für die Entwicklungs- und Testphase oft zu komplex und zu teuer. Deshalb wird fürs Entwickeln und Testen nutzt man deshalb gerne lightweight Kubernetes Tools:

- minikube, https://github.com/kubernetes/minikube): zum schnell K8s Luft Schnuppern. Nur Single Node
- microk8s, https://microk8s.io/docs (Ubuntu) ist das kleinste, schnellste und vollständig konforme Kubernetes System.
- kind https://kind.sigs.k8s.io/
- k3s https://k3s.io/ Lightweight Kubernetes
- kubeadm https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/create-cluster-kubeadm/ : Single Node bis HA Cluster
- Docker for Windows https://docs.docker.com/desktop/install/windows-install/ (direkt in Docker eingebaut)

Vergleich

Tool	Fokus	Besonderheit
Docker Desktop	Bequemlichkeit	Ideal für Entwickler, die eh schon Docker nutzen. Ein Klick und K8s läuft.
minikube	Lokales Lernen	Der Klassiker. Simuliert einen Cluster oft in einer VM. Sehr stabil, aber ressourcenhungrig.
kind (Kubernetes in Docker)	CI/CD & Testing	Startet Kubernetes-Nodes als Docker-Container. Extrem schnell und super für automatisierte Tests.
k3s	Edge & IoT	Von Rancher entwickelt. Alles Unnötige wurde entfernt. Läuft sogar auf einem Raspberry Pi perfekt.
microk8s	Ubuntu/Snap	Von Canonical. Sehr einfach zu installieren unter Linux (snap install ...) und sehr modular.
kubeadm	«Der harte Weg»	Das offizielle Tool, um «echte» Cluster aufzusetzen. Weniger eine «Lightweight-App», eher das Standard-Werkzeug für Profis.

9.5 Kubernetes Deep Dive



https://www.youtube.com/watch?v=QLIxqx-S4h4

Kubernetes hat eine Datenbank (**etcd**) (Key-Value-Store). Diese hat eine API. Wenn ich eine Anwendung auf meinem Kubernetessystem laufen lassen will, schriebe ich eine .yaml (ähnlich wie docker compose) mit den Details und schicke diese über die API auf die Datenbank (etcd) von Kubernetes.

Der Controllermanager (**cm**) beobachtet die Datenbank und sieht: Ein neues Deployment ist da. Er macht ein Replika aus der Instruktionsdate (yaml) und aus diesem dann einen Pod. Ein Pod ist eine Gruppe von Containern, die auf einer Node ausgeführt werden kann.

Der Scheduler (**s**) beobachtet die Pods und sieht: Da ist ein neuer Pod! Er sucht eine Node aus, die noch Platz hat und appendet in die Pod-Datei, auf welcher Node dieser Pod ausgeführt werden soll.

Wenn ein Knoten ausfällt, dann reagiert dessen Kubelet (**k**) nicht mehr auf Anfragen des cm. Der cm legt darauf hin einen neuen Pod an, der dann vom s geschedulet wird und somit wieder ausgeführt wird auf einer anderen Node.

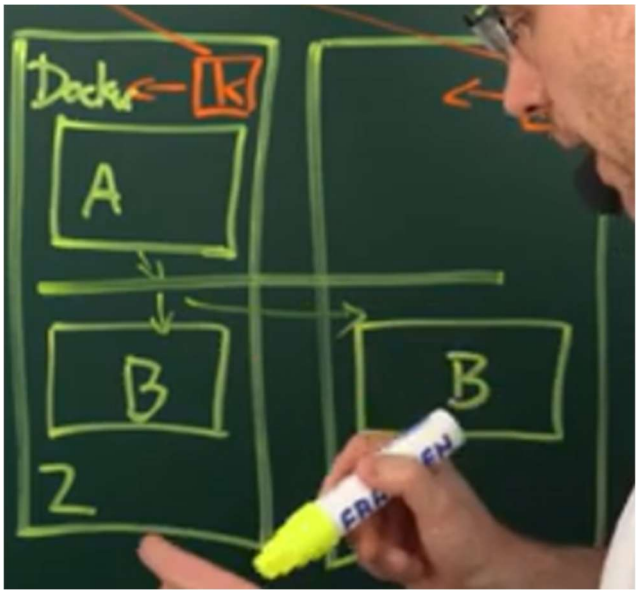
9.5.1 Container Runtime Interface CRI

Schnittstelle für die Nodes. Damit kann quasi jede Art von Hardware und Software seine Rechenpower für Kubernetes zur Verfügung stellen. Klassischerweise sind es aber natürlich Linuxmaschinen auf denen Docker läuft.

9.5.2 Container Network Interface CNI

Netzwerk zwischen den Pods. Damit sie kommunizieren können, auch über mehrere Nodes hinweg! Jeder Pod bekommt eine IP-Adresse. Man kann dann auch Firewalls definieren.

Load Balancers



Wenn Container A mit Container B kommunizieren muss, kann Kubernetes statt der Container B-IP einfach ein Load Balancer anlegen und dessen IP an A geben. Dieser schickt dann die Anfragen an den Load Balancer und dieser teilt diese auf mehrere Instanzen (Pods auf denen B-Instanzen laufen) auf.

Dafür gibt es innerhalb von Kubernetes ein eigenes DNS-System: Sky DNS

9.5.3 Container Storage Interface CSI

Persistenter Speicher für Containerdaten (bzw. Daten der Nodes...). Das nennen sie **persistent Volumes**. Wo dieses Volume liegt im Cluster, kann man selbst definieren.

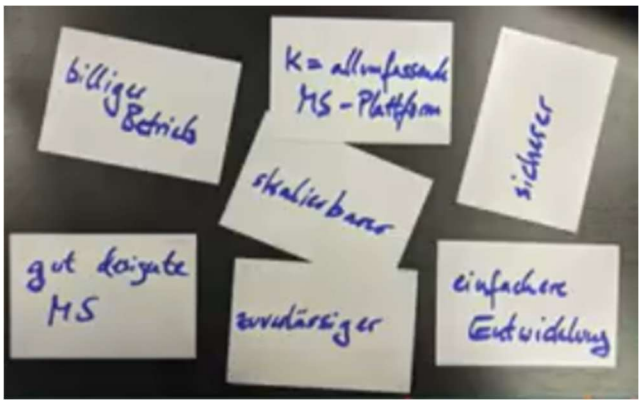
9.5.4 Cloud Controller

erlaubt es Kubernetes, sich mit Cloud Services wie zum Beispiel Storage und Load Balancern zu integrieren.

9.5.5 Redundanz

Sowohl DB, API, cm als auch s können auf mehreren Servern laufen (mehrere Instanzen). Zudem kann das ganze Cluster auch über mehrere Datencenter verteilt laufen.

9.5.6 Falsche Annahmen



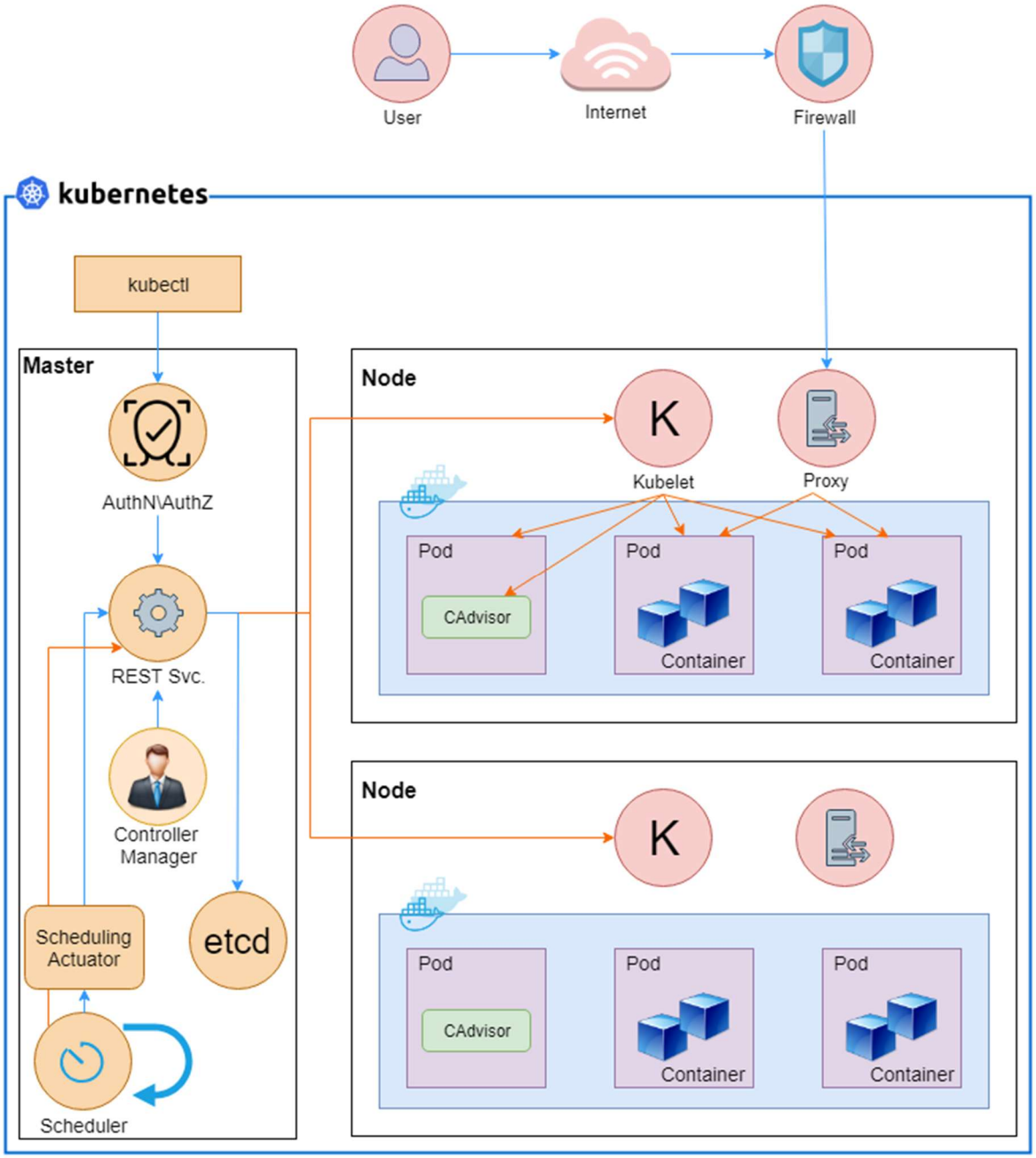
Alle diese Annahmen stimmen nicht direkt, es braucht immer einen Mehraufwand! Nur Kubernetes alleine aufsetzen bringt noch keine Vorteile, man muss es auch richtig konfigurieren und die in Kubernetes gestarteten Anwendungen müssen wir auch so coden, damit sie dasunterstützen!

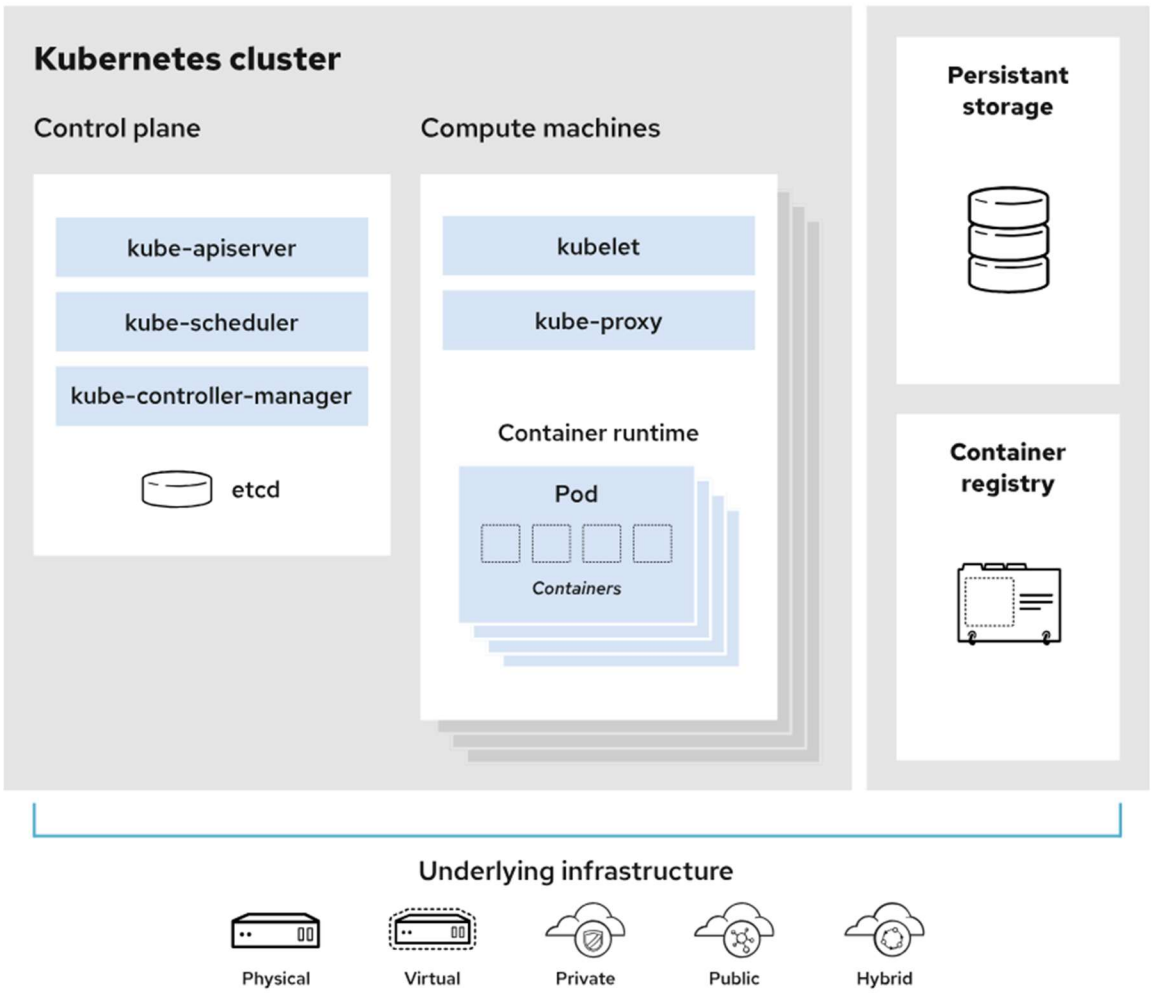
Macht nicht die ganze Microservice-Architektur: Kubernetes hat z.B. keine Verschlüsselung.

9.5.7 Vorteile

Vermindert Vendor-Lockin: Sobald meine Kubernetes-Architektur steht, kann ich sie portabel deployen: Heute on-prem im eigenen Rechenzentrum, morgen bei AWS, übermorgen bei Azure!

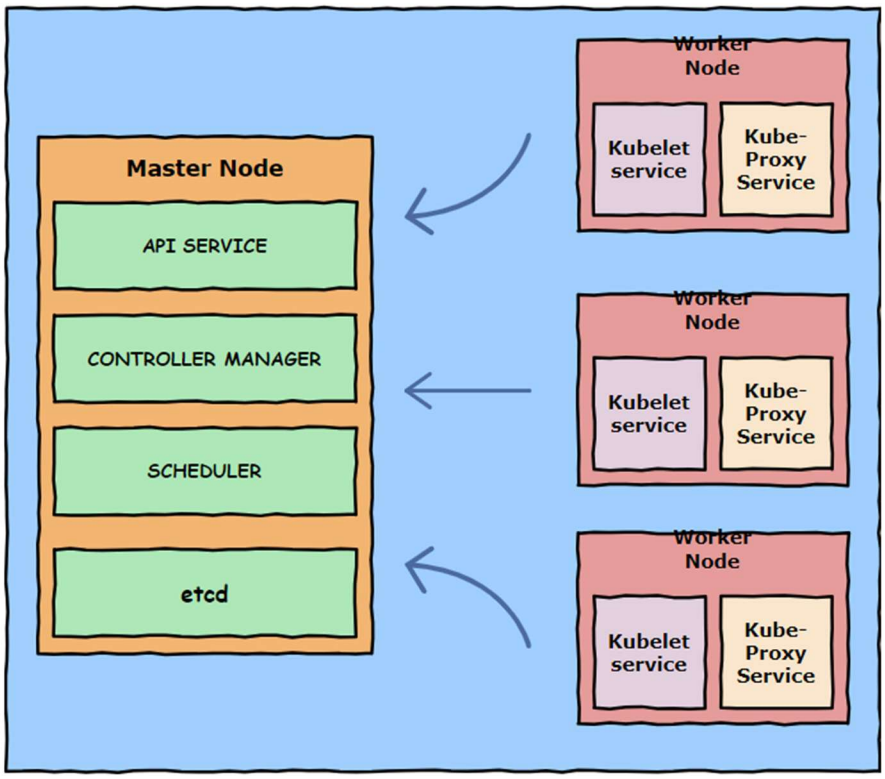
9.6 Kubernetes-Architektur im Detail verstehen





Es wird (noch) zwischen Master- («master») und Worker-Nodes («node») unterschieden. Dies wird in Zukunft wohl wegfallen, so dass alle alles machen können.

Master machen die Steuerung. Worker-Nodes führen unsere Anwendungen aus.



9.6.1 Masternodes

Normalerweise 4 Services:

Master führen die folgenden Services aus, die die Control-plane bilden (oder eben das Gehirn des Clusters):

- API Server
- Scheduler
- Key/Value-Store
- Cloud Controller
- Weitere ...
-

Ausfallsicherheit: Man hat 3 bis 5 Masternodes, die in unterschiedlichen Datencentern liegen

9.6.2 Workernodes

- Linux Nodes führen Linux Apps aus, Windows Nodes führen Windows Apps aus.
- Worker Nodes sind grösser als Masternodes und brauchen mehr Ressourcen!

Alle Nodes führen ein paar erwähnenswerte Services aus:

- Kubelet
- Container Runtime

Ein **Kubelet** ist ein Kubernetes Agent. Er folgt dem Node zum Cluster und kommuniziert mit der Control-plane – Dinge wie das Empfangen von Aufgaben und das Reporting über den Aufgabenstatus.

Die **Container Runtime** startet Container und führt sie aus.

9.7 Raft-Konsens-Algorithmus

Der Raft-Algorithmus wurde entwickelt, um in einem Verbund von Computern (Cluster) eine gemeinsame Wahrheit herzustellen und ist dabei bewusst verständlicher gestaltet als ältere Alternativen wie Paxos.

Jeder Server im Cluster nimmt einen von drei möglichen Zuständen ein: Anführer (Leader), Untertan (Follower) oder Kandidat (Candidate).

Im fehlerfreien Normalbetrieb gibt es immer genau einen Leader, während alle anderen Server als Followers agieren.

Nur der Leader nimmt Lese- und Schreibaufgaben von Clients entgegen.

Um zu signalisieren, dass er noch aktiv ist, sendet der Leader kontinuierlich sogenannte Heartbeat-Nachrichten an die Followers.

Fällt der Leader aus und die Heartbeats bleiben aus, greift ein Zufallsmechanismus: Jeder Server hat eine individuell zufällige Wartezeit (Election Timeout).

Dem Server, dessen Wartezeit zuerst abläuft, «reisst der Geduldsfaden»: Er wechselt in den Candidate-Zustand, stimmt für sich selbst und bittet die anderen Server um ihre Stimme. * Sobald dieser Kandidat die absolute Mehrheit der Stimmen erhält, ernennt er sich zum neuen Leader und sendet wieder Heartbeats.

Bei neuen Schreibaufträgen schickt der Leader die Änderung an alle Followers.

Erst wenn mehr als die Hälfte der Server den Empfang bestätigt haben, wendet der Leader die Änderung endgültig an (Commit) und antwortet dem Client.

9.8 Warum ungerade Anzahl Server im Cluster?

Raft basiert auf dem Prinzip der absoluten Mehrheitswahl, um Konflikte und das gefürchtete Split-Brain-Szenario zu verhindern (das ist dasselbe Prinzip beim Bundesrat, der aus 7 Personen besteht!).

- Eine ungerade Anzahl von Servern wird in allen Dokumentationen empfohlen, da sie klare Mehrheiten ermöglicht.
- In einem Cluster mit drei Servern liegt die absolute Mehrheit bei zwei Stimmen, weshalb ein Server problemlos ausfallen darf.
- Fügt man einen vierten Server hinzu, steigt die benötigte absolute Mehrheit auf drei Stimmen an.
- Daher darf auch bei vier Servern weiterhin nur ein einziger Server ausfallen, wodurch die Stabilität nicht gewonnen hat, das System aber komplexer wird.

9.9 Kubernetes Pods

Kubernetes führt Container innerhalb von Pods aus. Es ist quasi eine weitere Schicht um einen Container herum. Normalerweise ist pro Pod nur ein Container enthalten.

Ähnlich wie bei docker-compose.yaml werden Pods auch in .yaml geschrieben:

In der Abbildung ist im orangen Rahmen die **Bereitstellung** zu sehen. Sie ist sozusagen die gesamte "Schachtel" welche weitere Schachteln beinhaltet.

Die **apiVersion** und kind Zeilen sagen Kubernetes den Typ und die Version des Objektes, das deployt werden soll. In diesem Fall ein Pod Objekt, wie es in der v1 API definiert ist.

Der **metadata** Block listet den Pod Namen und ein einzelnes Label auf. Der Name hilft dabei, den Pod zu identifizieren und zu verwalten, während er ausgeführt wird. Das Label (app = todo-app) ist nützlich, um Pods zu organisieren und sie mit anderen Objekten zu assoziieren, zum Beispiel Load Balancern. Wir sehen Labels später noch in Aktion.

Der **spec** Abschnitt gibt weitere Angaben, z.B. wie viele Replikationen (**Replikationsfaktor**) dieser Anwendung verfügbar sein sollen, wie ein Update ausgeführt werden soll, usw.

Die weisse Umrandung zeigt, wie der Pod (weiss) den Container verpackt. Diese Pod Verpackung ist notwendig, damit ein Container auf Kubernetes ausgeführt werden kann, und sie ist ziemlich leichtgewichtig, weil sie nur Metadaten hinzufügt.

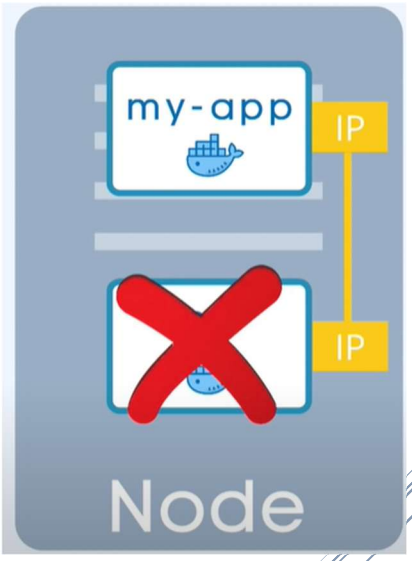
todo-app-deploy-v2.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: todo-app-deployment
  labels:
    app: todo-app
spec:
  replicas: 3
  selector:
    matchLabels:
      app: todo-app

  minReadySeconds: 20
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 0
  template:
    metadata:
      labels:
        app: todo-app
    spec:
      containers:
      - name: todo-app
        image: staubth/todo-app:v2
        ports:
        - containerPort: 3000
          protocol: TCP
```

Der **template** Abschnitt definiert den eigentlichen Pod wiederum mit Metadaten wie labels usw.

Im **spec** Abschnitt des Pods ist dann der eigentliche Container definiert.

9.10 Netzwerk



In Kubernetes erhät jeder Pod (bzw. dessen Service) (nicht aber der Container!) eine eigene kubernetes-interne IP-Adresse.

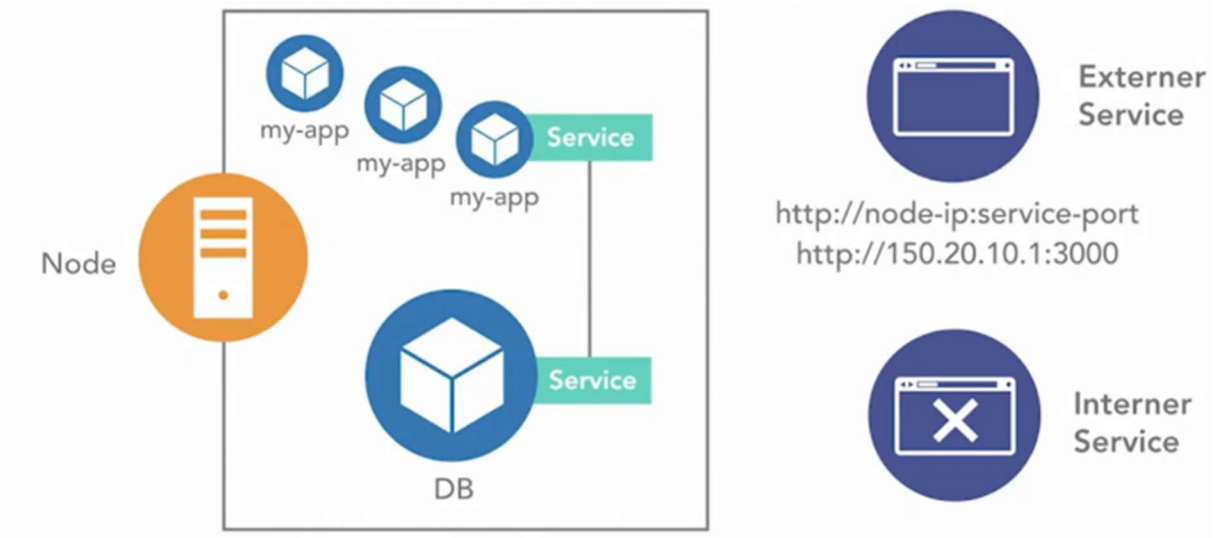
Fällt ein Pod aus, so wird er wieder neu erstellt. Dadurch erhält er aber eine **neue** IP! Das ist sehr unpraktisch: Man müsste z.B. beim Backend jedes Mal die IP der Datenbank im Code ändern!

Deshalb hat jeder Pod einen Service. Die IP-Adresse gehört dem Service, nicht dem Pod. Dadurch kann der Pod crashen und neu starten, ohne dass die IP-Adresse ändert. Denn der Service eines Pods bleibt persistent 😊

Services werden im kubernetes-internen DNS-Server eingetragen.

Ein Pod kann sowohl einen internen als auch einen externen Service haben. Damit kann man Anwendungen von Aussen (z.B. Webseite) erreichbar machen.

Zudem kann der Service auch als Load-Balancer agieren: Er ist mit mehreren Pods verknüpft und verteilt die Anfragen:



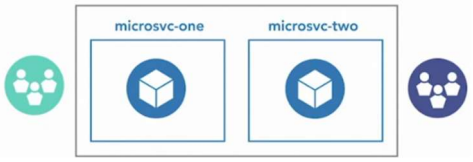
9.11 Namespaces / Namensräume

Ein Kubernetescluster wird schnell mal sehr komplex. Dadurch ergibt sich die Notwendigkeit, Pods in Gruppen zu organisieren. Namespaces sind eine logische Gruppierung der Komponenten des Clusters.

Ein Kubernetes-Cluster kann also in verschiedene (virtuelle) Cluster aufgeteilt werden, die sogenannten Namespaces. Pods, Services laufen, ohne Konfiguration, einfach mal im «default namespace».

Beispiel A: Wir brauchen zusätzliche Datenbanken, nur um die Logs der unterschiedlichen Pods zu sammeln und aufzubereiten! Dann brauchen wir Monitoring-Tools, um das Cluster zu überwachen, diese starten wir ebenfalls als Pods. Etc.

Beispiel B: Microservice-Architektur: Unterschiedliche Teams arbeiten je an ihren Pods. Da kann es zu Konflikten kommen, wenn all diese Pods im selben Namespace laufen. Namespaces helfen hier bei der Trennung.



Beispiel C: Ressourcen-Quotas: Ressourcen können den Namespraces zugeteilt werden. Namespace A soll maximal so und so viel RAM bekommen.

10 Kubernetes installieren

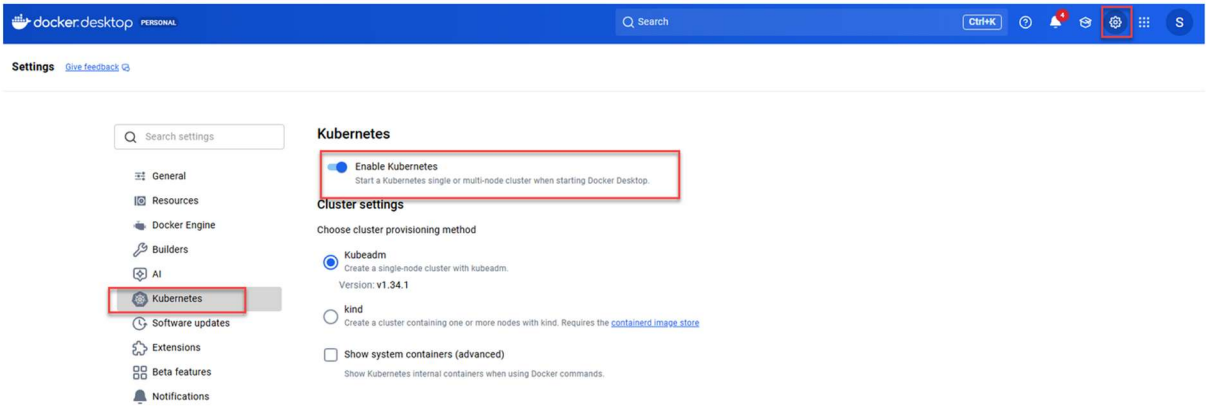
10.1 Das Cluster steuern: Was ist kubectl?

Um ein Cluster zu verwalten, gibt es drei Möglichkeiten:

- UI (ein Web-UI, welches bereitgestellt wird und dann über den Browser erreichbar ist)

- API (Befehle direkt an das Cluster via dessen API schicken, das ist praktisch für Skripts!)
- kubectl (Kommandozeilen-Tool)

10.2 Kubernetes in Docker Desktop aktivieren



Und verknüpfen dies mit unserer Docker Desktop Instanz.

```
kubectl config get-contexts
kubectl config use-context docker-desktop
```

```
vmadmin@li224-vmLM1:~/Downloads/microservices-main$ sudo snap install kubectl --classic
kubectl 1.28.2 from Canonical✓ installed
vmadmin@li224-vmLM1:~/Downloads/microservices-main$ kubectl config get-contexts

CURRENT   NAME             CLUSTER          AUTHINFO         NAMESPACE
*         docker-desktop   docker-desktop   docker-desktop
vmadmin@li224-vmLM1:~/Downloads/microservices-main$
vmadmin@li224-vmLM1:~/Downloads/microservices-main$ kubectl config use-context docker-desktop
Switched to context "docker-desktop".
vmadmin@li224-vmLM1:~/Downloads/microservices-main$
```

10.3 Kubernetes mit microk8s installieren

Da ich auf einer lokalen VirtualBox-VM arbeite (Ubuntu) und diese die Virtualisierung nicht unterstützt…

```
jon@jon-ubuntu-vbox:~$ lsmod | grep kvm
```

… installiere ich Kubernetes über microk8s:

```
sudo snap install microk8s --classic

sudo usermod -aG microk8s $USER

sudo chown -f -R $USER ~/.kube
```

```
jon@jon-ubuntu-vbox:~$ lsmod | grep kvm
jon@jon-ubuntu-vbox:~$ sudo snap install microk8s --classic
[sudo] password for jon:
microk8s (1.33/stable) v1.33.7 from Canonical✓ installed
jon@jon-ubuntu-vbox:~$ sudo usermod -aG microk8s $USER
jon@jon-ubuntu-vbox:~$ sudo chown -f -R $USER ~/.kube
```

10.3.1 kubectl installieren

```
sudo snap install kubectl --classic
```

```
jon@jon-ubuntu-vbox:~$ sudo snap install kubectl --classic
kubectl 1.34.4 from Canonical✓ installed
```

10.3.2 Context verbinden

```
# Erstellt den Ordner, falls er fehlt
mkdir -p ~/.kube
# Schreibt die microk8s Konfiguration in die Standard-Datei für kubectl
sudo microk8s config > ~/.kube/config
```

10.3.3 kubectl mit microk8s verknüpfen

```
kubectl config get-contexts
```

```
jon@jon-ubuntu-vbox:~$ kubectl config get-contexts
CURRENT   NAME       CLUSTER            AUTHINFO   NAMESPACE
*         microk8s   microk8s-cluster   admin
```

```
kubectl config use-context microk8s
```

```
jon@jon-ubuntu-vbox:~$ kubectl config use-context microk8s
Switched to context "microk8s".
```

10.3.4 Testen

```
kubectl get nodes
```

Nun sollte eine Node angezeigt werden:

```
jon@jon-ubuntu-vbox:~$ kubectl get nodes
NAME              STATUS   ROLES    AGE   VERSION
jon-ubuntu-vbox   Ready    <none>   12m   v1.33.7
```

10.4 Mit Lens IDE auf den Server zugreifen

Die Lens IDE ist eine IDE für Kubernetes.

10.4.1 Lens-Account erstellen: https://lenshq.io/

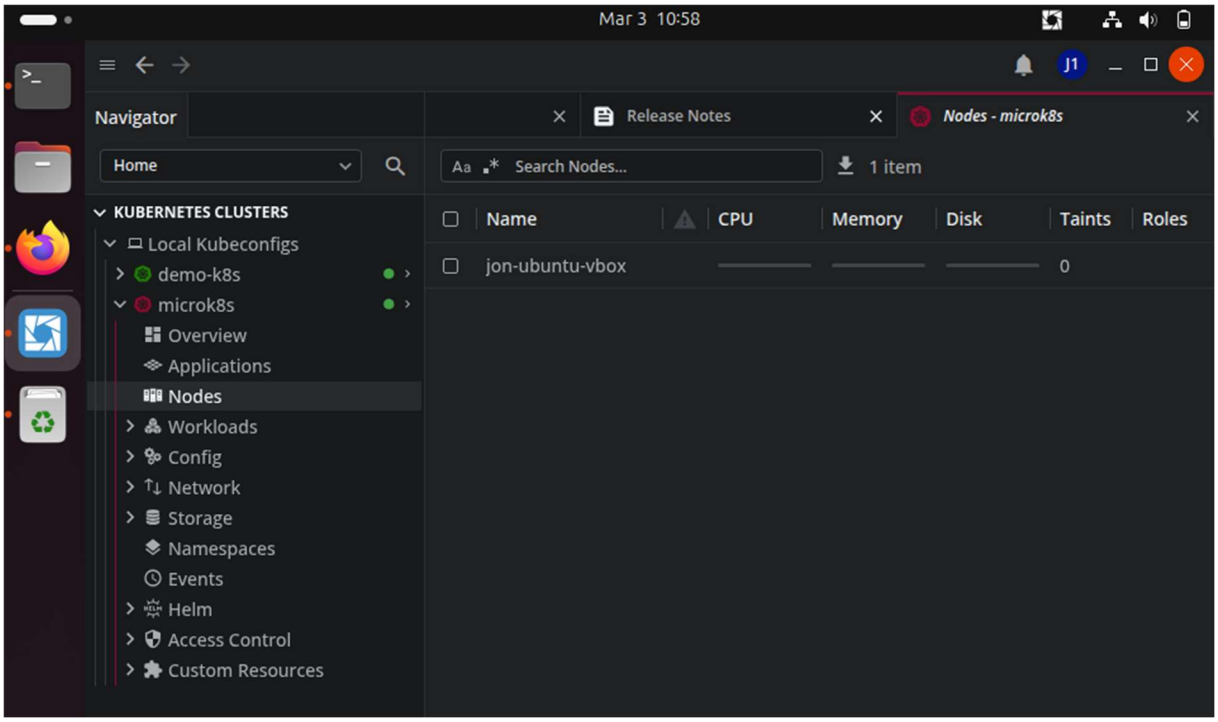
Ich habe hierfür einfach mein bestehendes GitHub-Konto verbunden.

10.4.2 Lens installieren

```
sudo snap install kontena-lens -classic
```

Dann Lens öffnen und einloggen.

Ich konnte dann mit meinem Kubernetesserver verbinden und das microk8s-Cluster anzeigen lassen:



11 To-Do App in Kubernetes

Das Projekt gibt bereits eine **`todo-app-deployment.yaml`** vor. Mit dieser kann ich mit

`kubectl create -f todo-app-deploy.yaml -n to-do-app`

die Pods starten.

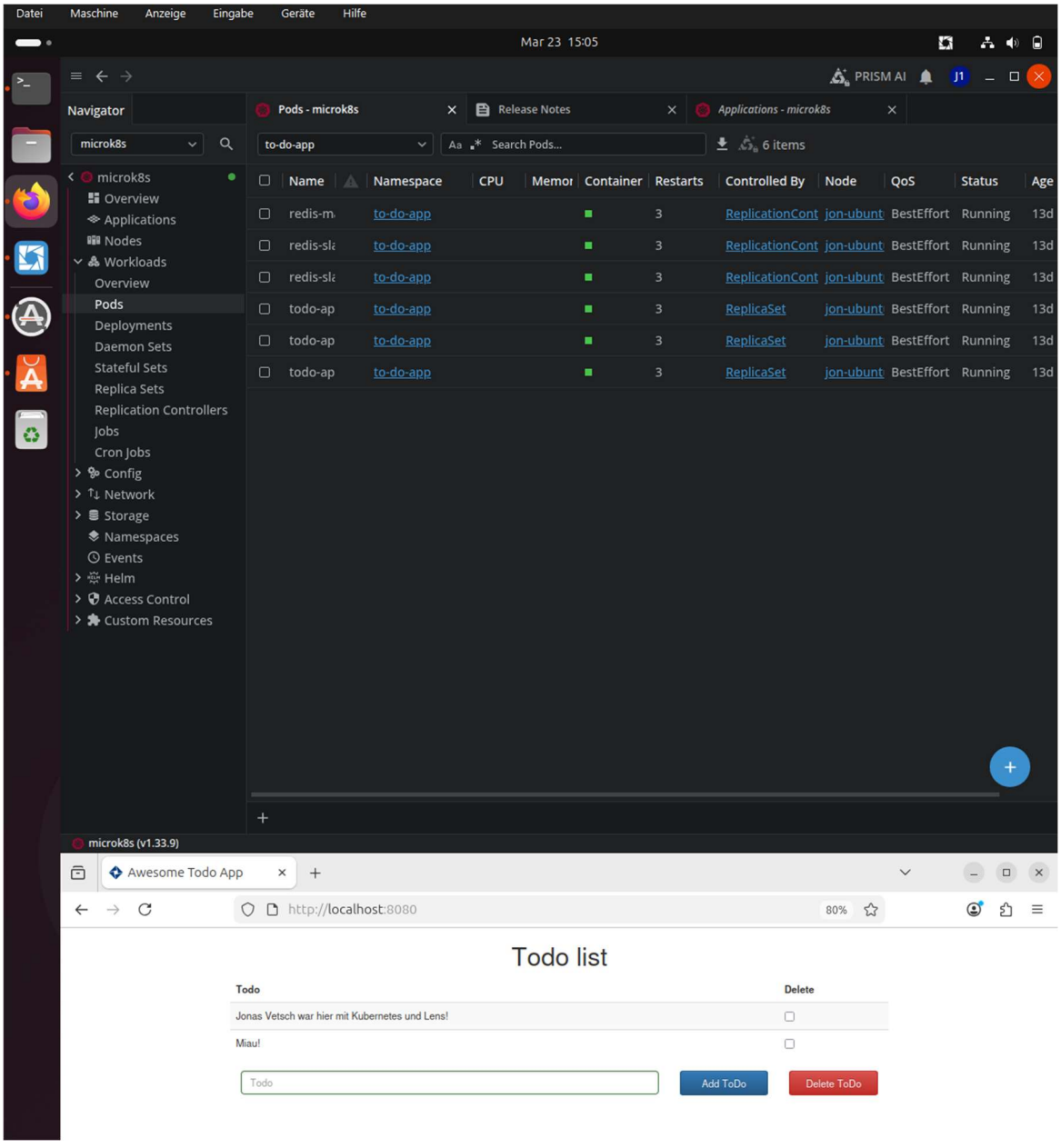
Danach die Services einrichten:

`kubectl create -f todo-app-service-deploy.yaml -n to-do-app`

Und dann noch den Port forwarden:

`kubectl -n to-do-app port-forward svc/todo-app-service 8080:80`

Nun läuft die App auf K8s!



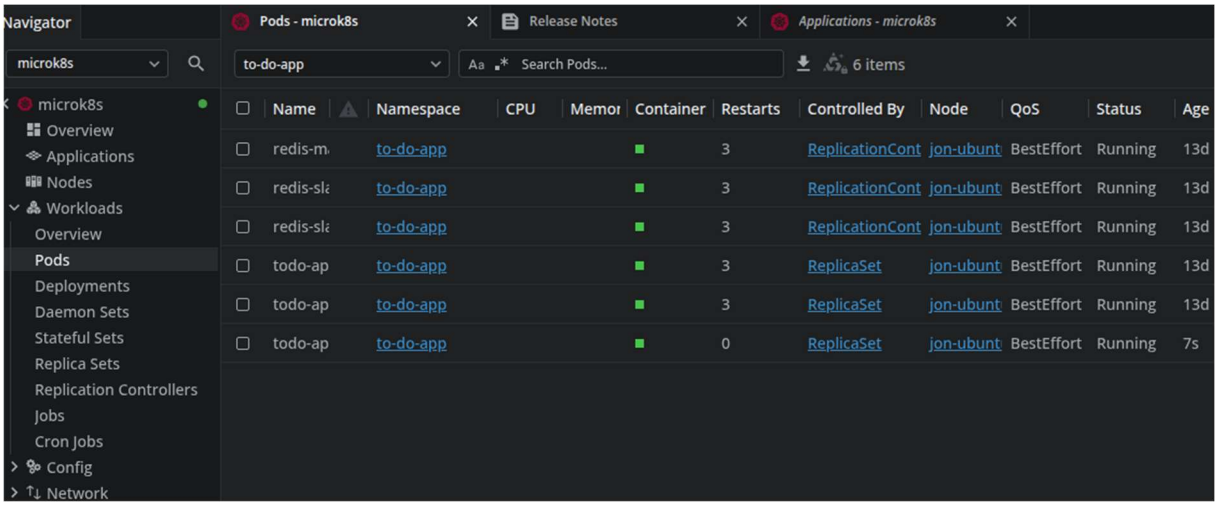
11.1 Self Healing

todo-app-deploy.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: todo-app-deployment
  labels:
    app: todo-app
spec:
  replicas: 3
  selector:
    matchLabels:
      app: todo-app
  template:
    metadata:
      labels:
        app: todo-app
    spec:
      containers:
      - name: todo-app
        image: staubth/todo-app:v1
        ports:
        - containerPort: 3000
          protocol: TCP

```

Weil wir in der **`todo-app-deployment.yaml`** angeben, dass wir 3 Replicas möchten, erstellt Kubernetes ein ReplicaSet mit 3 Pods. Nun stellt Kubernetes sicher, dass auch immer diese drei Pods existieren. Der Controller Manager in Kubernetes hat einen Sub-Dienst, der sich Replica Controller nennt. Dieser prüft, ob die Vorgaben aus unserer yaml-Datei auch zu jeder Zeit eingehalten werden. Wenn ich nun also einen Pod lösche, erstellt Kubernetes direkt wieder einen neuen. Konkret prüft der Replica Controller, ob die Anzahl bestehenden Pods mit der im etcd hinterlegten Anzahl übereinstimmt. Ist dies nicht der Fall, macht er ins etcd einen Eintrag, dass ein neuer Pod gestartet werden muss. Der Scheduler sieht diesen Eintrag und prüft, auf welchem Node es Ressourcen gibt, um diesen Pod zu starten. Danach sendet er einen Befehl an den Kubelet Service auf diesem Node, der dann den Pod startet.

Hier teste ich das konkret: Ich lösche einen Pod manuell in der Lens IDE und K8s startet ihn direkt neu. Wir sehen: Ein Pod ïst erst 7 Sekunden alt:

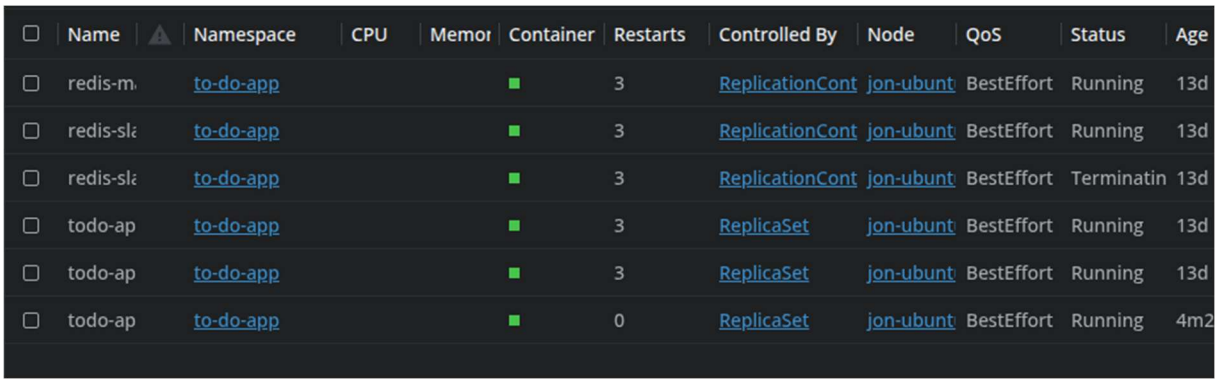


11.2 Scale Down

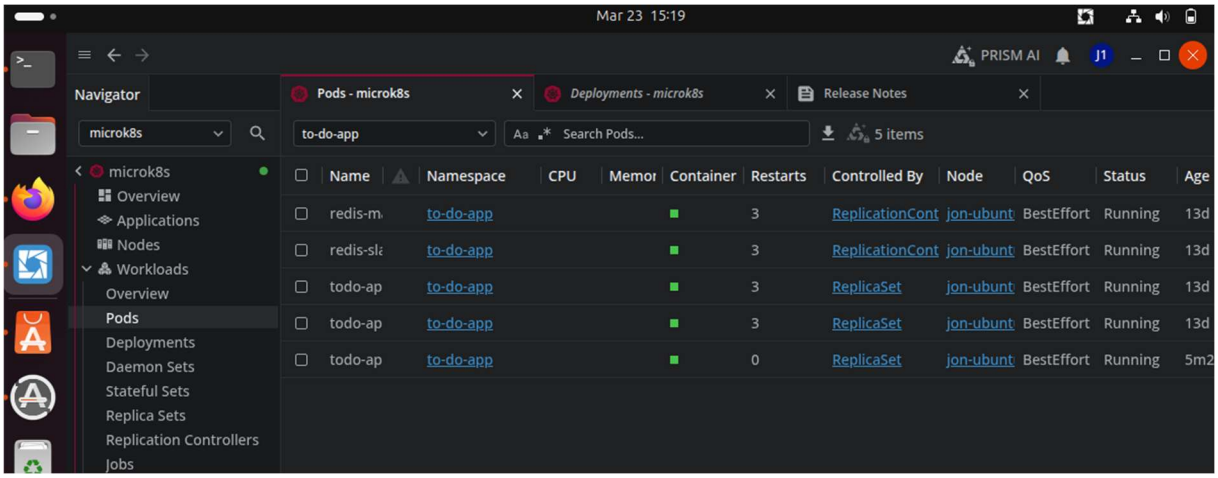
Wenn wir weniger Traffic auf der App haben als zunächst geplant, können wir die Replica Anzahl auch verringern. Das geht einerseits manuell über

```
kubectl scale --replicas=1 rc redis-slave -n to-do-app
```

Nun sieht man wie ein Pod `Terminating` anzeigt:



und wenig später ist er ganz verschwunden:



Alternativ können wir die Anzahl auch in der todo-app-deploy.yaml anpassen und dieses yaml-File auf unser Deployment erneut anwenden mit

```
kubectl apply -f todo-app-deploy.yaml -n to-do-app
```

11.3 Scale Up

Hochskalierung in Kubernetes ermöglicht es Entwicklern, schnell auf steigende Nutzerzahlen zu reagieren, sei es durch den Befehl **`kubectl scale`** oder (besser) durch die Anpassung der **`replicas`** in der YAML-Datei. Während die Kommandozeile sofortige Ergebnisse liefert, sichert der deklarative Weg via **`kubectl apply`** die Konsistenz der Konfiguration und verhindert, dass spätere Updates die Anzahl der Pods ungewollt zurücksetzen.

Man kann es auch mit Horizontal Pod Autoscaling automatisieren. Das passt die Ressourcen dynamisch an die tatsächliche Last an, wie man es von der SBB-App bei Pendlerströmen oder Online-Shops am Black Friday kennt. Diese Flexibilität ist das Herzstück des Cloud-Computings und erlaubt eine präzise, nutzungsbasierte Abrechnung («Pay-per-Use»). So werden Ressourcen geschont, wenn sie nachts kaum benötigt werden, während bei Lastspitzen die Performance stabil bleibt.

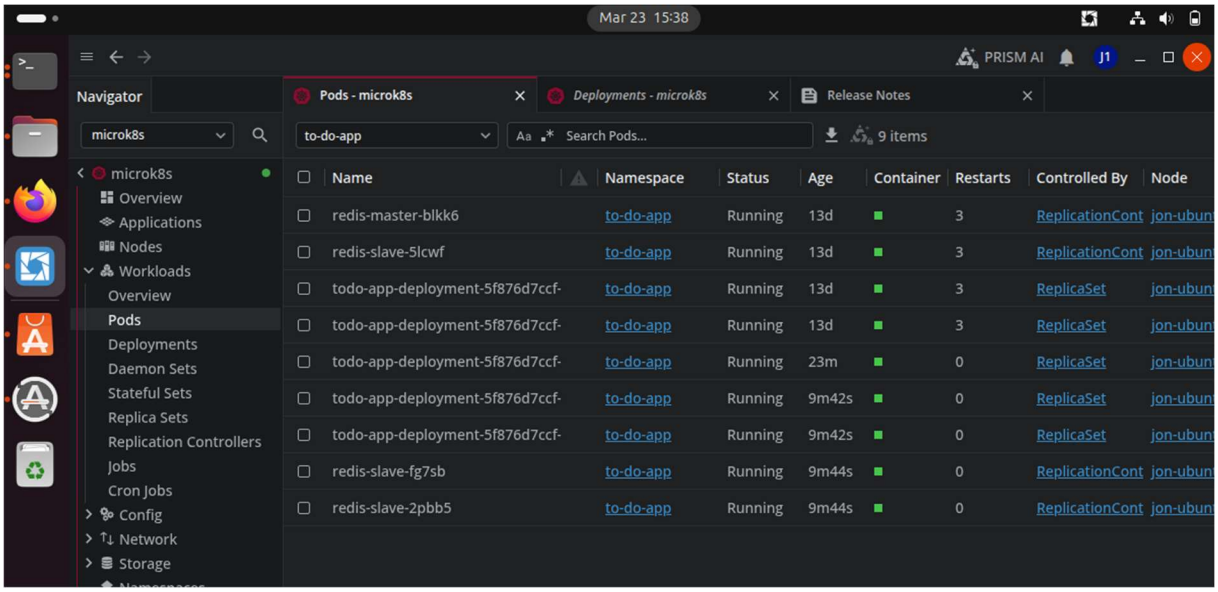
Doch man muss auch vorsichtig sein: Wenn man vergisst, hochskalierte Instanzen nach einem Test wieder zu reduzieren, wird man schnell von hohen Cloud-Kosten überrascht. Ein disziplinierter Umgang mit den Konfigurationsdateien ist daher im professionellen Umfeld der Goldstandard.

11.4 Rolling Update

Das Rolling Update ist das normale Vorgehen in Kubernetes, um Anwendungen ohne Ausfallzeiten auf eine neue Version zu heben. Dabei werden die alten Pods nicht alle gleichzeitig abgeschaltet, sondern sukzessive durch neue Instanzen ersetzt, was eine kontinuierliche Verfügbarkeit der App sicherstellt.

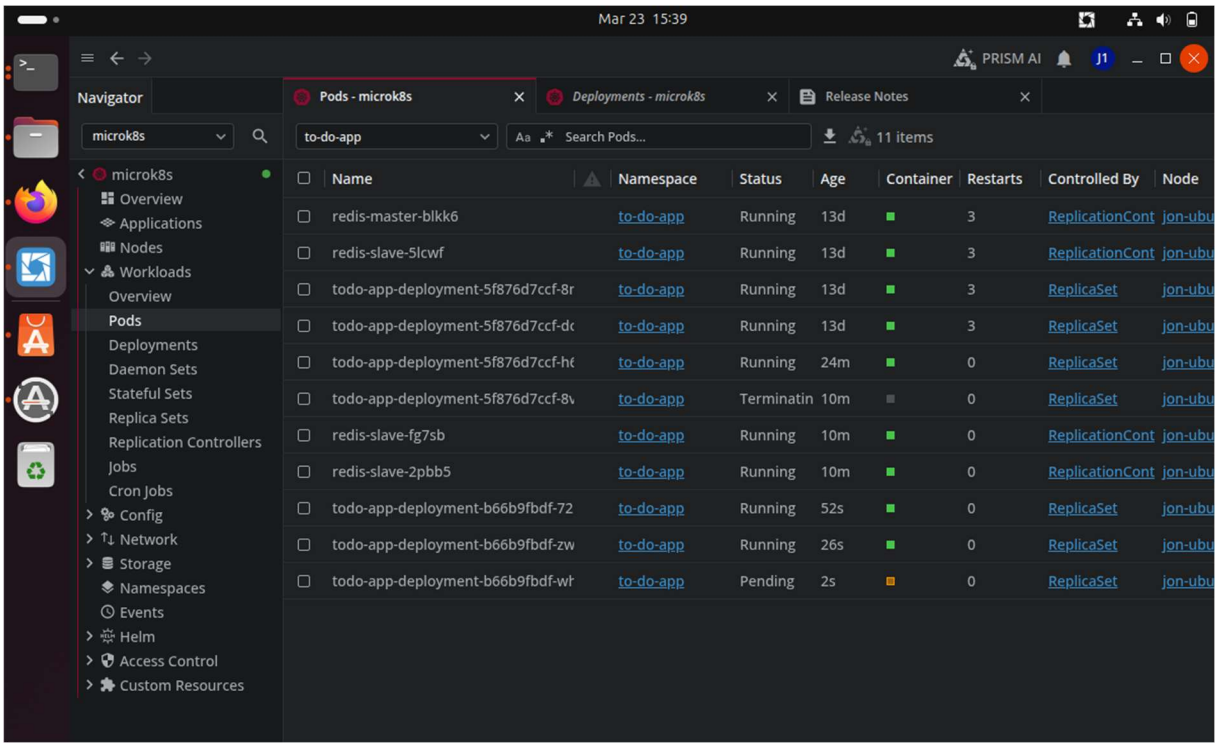
Durch Parameter wie **`maxSurge`** und **`maxUnavailable`** lässt sich präzise steuern, wie viele zusätzliche Ressourcen während des Prozesses kurzzeitig genutzt werden dürfen. Die Einstellung **`minReadySeconds`** bietet zudem ein wertvolles Zeitfenster, um die Stabilität der neuen Replikate zu beobachten, bevor der nächste Schritt erfolgt. Schlägt ein neuer Pod fehl, stoppt Kubernetes den Rollout automatisch, wodurch das Risiko für das Gesamtsystem minimiert wird. Diese Methode ist ideal für kleine, regelmässige Updates, da sie Ressourcen schont und den Datenverkehr fliessend von der alten auf die neue Version lenkt.

Um das Rolling Update zu testen habe ich zunächst die App etwas hochskaliert:



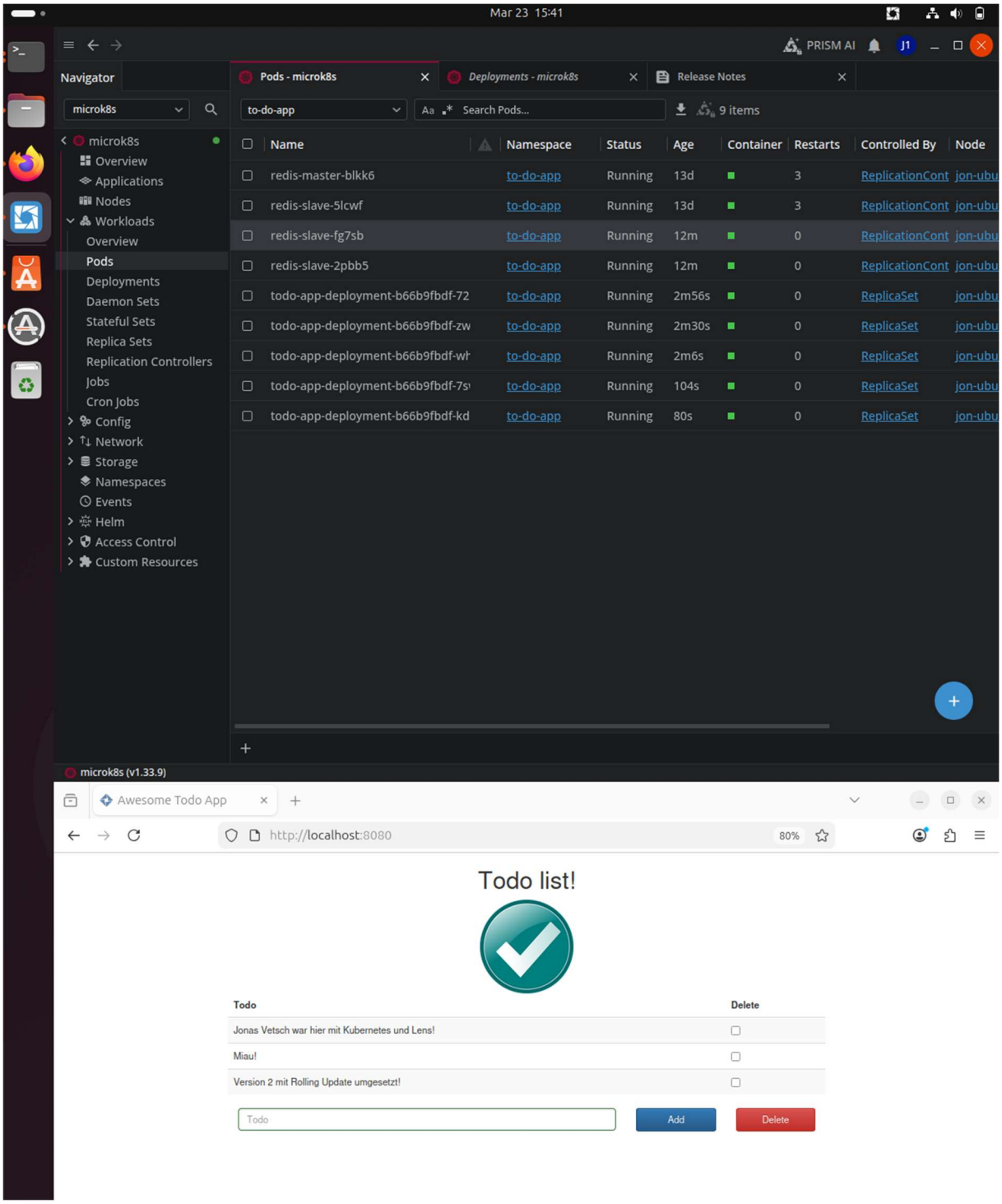
Nun laden wir die neue yaml-Datei:

```
kubectl apply -f todo-app-deploy-v2.yaml -n to-do-app
```



Man sieht nun schön, wie zuerst ein zusätzlicher Pod gestartet wird und dann ein bestehender gelöscht wird. Das geht so lange, bis die ganze App umgestellt hat.

Nun wird mir die Version 2 der App ausgeliefert, wenn ich sie im Webbrowser öffne:



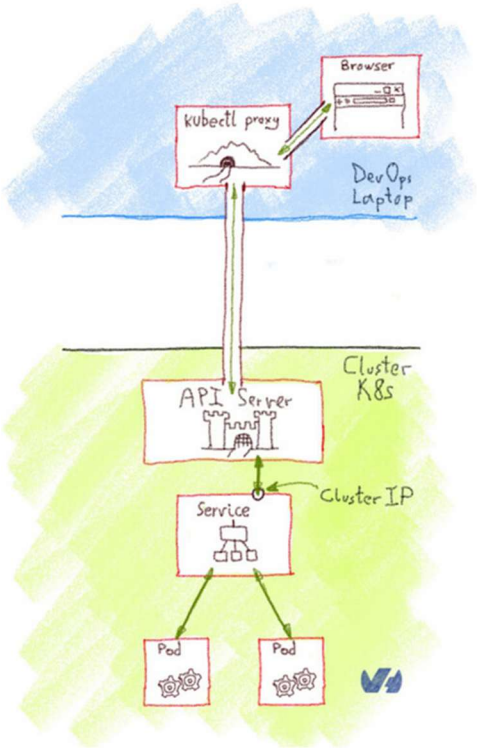
11.5 Blue/Green Deployment

Im Gegensatz zum schrittweisen Vorgehen basiert das Blue/Green Deployment auf der Bereitstellung zweier identischer, aber getrennter Umgebungen. Während die "blaue" Umgebung die aktuelle Live-Version bedient, wird die neue Version in der "grünen" Umgebung vollständig hochgefahren und isoliert getestet. Sobald die neue Version final abgenommen wurde, wird der gesamte Datenverkehr durch eine einfache Änderung am Service oder Loadbalancer schlagartig von Blau auf Grün umgeleitet. Dies ermöglicht ein nahezu sofortiges Umschalten und bietet den entscheidenden Vorteil eines extrem schnellen Rollbacks: Sollten Probleme

auftreten, leitet man den Traffic einfach wieder zurück auf die noch bestehende blaue Umgebung. Da jedoch für kurze Zeit die doppelte Menge an Ressourcen benötigt wird, ist diese Strategie zwar kostspieliger, bietet aber maximale Sicherheit bei kritischen, grossen Releases.

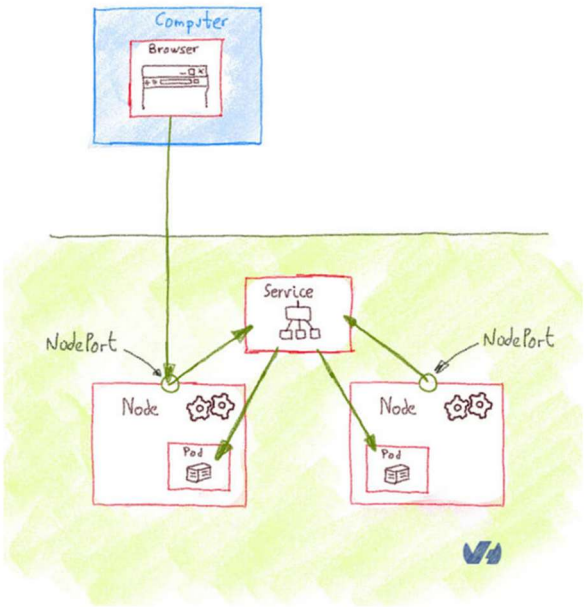
12 Kubernetes Ingress

12.1 ClusterIP



Der Service-Typ **ClusterIP** fungiert als der interne Standard in Kubernetes und weist dem Dienst eine IP-Adresse zu, die ausschliesslich innerhalb des Clusters erreichbar ist. Dies dient primär der Absicherung, da Microservices so untereinander kommunizieren können, ohne direkt dem öffentlichen Internet ausgesetzt zu sein. Ein externer Zugriff ist hierbei nur über Umwege wie einen Proxy möglich.

12.2 NodePort



Möchte man eine Anwendung aber direkter nach aussen öffnen, nutzt man den **NodePort**, der einen spezifischen Port im Bereich von 30000 bis 32767 auf jedem Cluster-Knoten (Node) reserviert. Über die Kombination aus der IP-Adresse des Nodes und diesem statischen Port kann der Dienst im Browser aufgerufen werden, was besonders in lokalen Testumgebungen oder VMs nützlich ist. Während ClusterIP für die interne Struktur sorgt, bildet NodePort eine der einfachsten Brücken in die Aussenwelt, hat jedoch Einschränkungen bei der Skalierbarkeit und Port-Verwaltung. In der Praxis werden beide Typen oft als Basis für komplexere Routing-Lösungen wie Ingress verwendet.

12.3 LoadBalancer

Der Service LoadBalancer ist die bevorzugte Lösung für professionelle Produktionsumgebungen, da er die Brücke zwischen dem Kubernetes-Cluster und der Infrastruktur eines Cloud-Anbieters schlägt. Sobald dieser Typ deklariert wird, stellt der Provider automatisch eine externe, öffentlich erreichbare IP-Adresse bereit, die den eingehenden Datenverkehr effizient auf die dahinterliegenden Pods verteilt. In lokalen Umgebungen wie MicroK8s übernimmt ein Add-on wie MetalLB diese Aufgabe, um das Verhalten einer echten Cloud zu simulieren. Der grosse Vorteil liegt in der hohen Verfügbarkeit und der einfachen Handhabung für Endnutzer, die lediglich eine einzige IP oder Domain ansteuern müssen. Allerdings ist diese Methode oft mit höheren Kosten verbunden, da Cloud-Anbieter pro instanziiertem LoadBalancer abrechnen und jeder Dienst eine eigene IP beansprucht. Um diese Kosten zu optimieren, wird in grösseren Setups häufig nur ein einziger LoadBalancer geschaltet, der den Traffic an einen Ingress-Controller weiterleitet. So lassen sich viele verschiedene Dienste über eine einzige öffentliche Schnittstelle und kosteneffiziente ClusterIPs verwalten.

12.4 Erklärung, warum bei Ingress bei Zugriff auf 127.0.0.1 der Statuscode 404 zurückgegeben wird

Der Ingress-Controller fungiert als Türsteher (Reverse-Proxy), der den eingehenden Datenverkehr basierend auf Host-Headern sortiert. In unserer **`ingress.yaml`** haben wir festgelegt, dass der Dienst nur auf Anfragen reagiert, die den Host im HTTP-Header mitsenden. Wenn man die IP 127.0.0.1 direkt in den Browser eingibt, fehlt dies.

Da der Ingress-Controller für die IP-Adresse keine passende Regel (Routing-Rule) findet, weiss er nicht, an welchen internen Service er die Anfrage weiterleiten soll. Er liefert daher den Standard-Fehler «404 Not Found», da für dieses Ziel kein Inhalt definiert wurde.

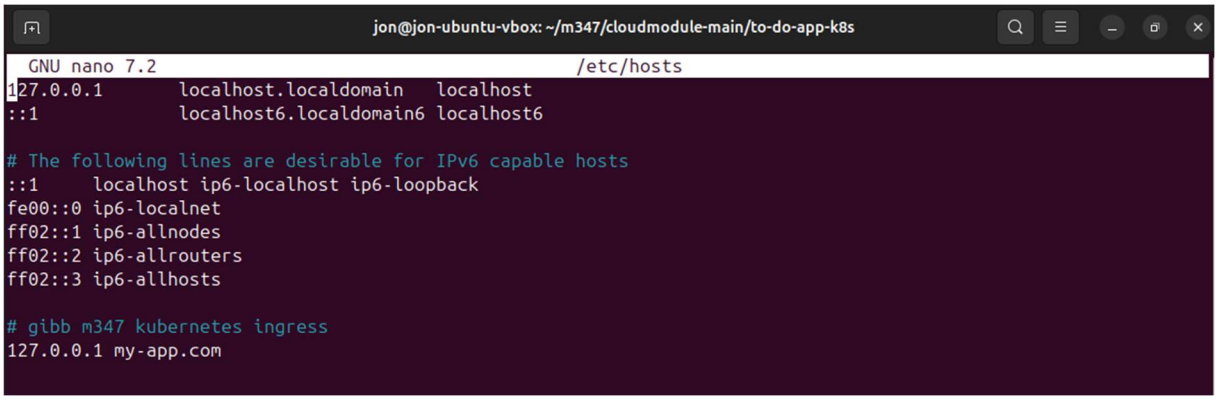


Abbildung 34 Eintrag in die /etc/hosts für my-app.com

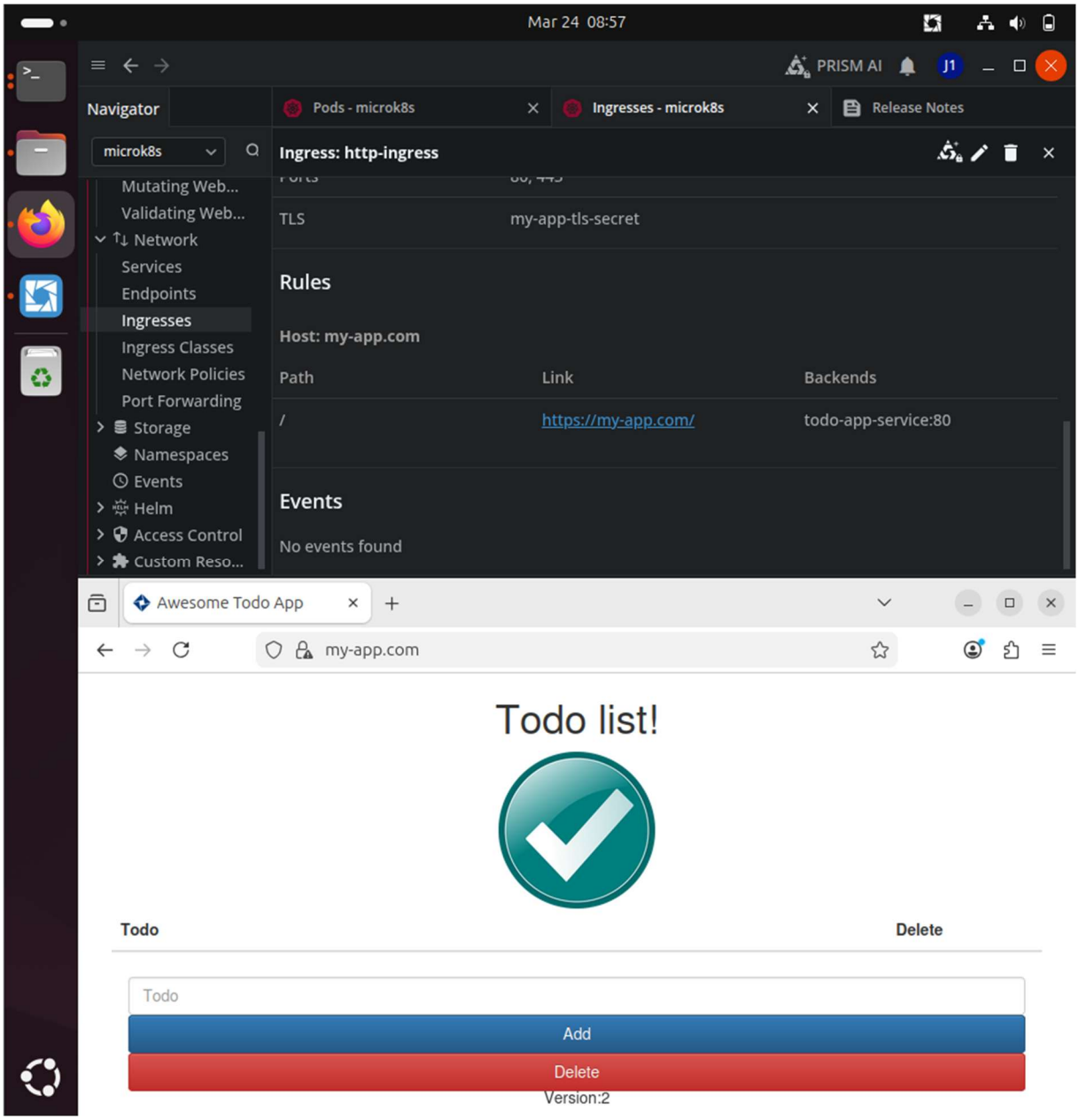
Durch den Eintrag in unserer Hosts-Datei wird sichergestellt, dass der Browser beim Tippen von my-app.com zwar die IP unserer VM kontaktiert, aber gleichzeitig den Namen my-app.com als «Host» mitschickt. So erkennt der Ingress-Controller die Übereinstimmung mit der Konfiguration und leitet uns erfolgreich an unsere App weiter.

12.5 ToDo-App mit Ingress erreichbar machen

Nach etwas Debugging konnte ich die ToDo App v2 unter https://my-app.com/ erreichen. Da ich mit mikrok8s arbeite, musste ich, zusätzlich zu den Schritten auf der Smartlearnanleitung, noch ein Port-Forwarding einrichten, da der Ingress-Controller nicht standardmässig auf die Ports 80 und 443 meines Ubuntu-Systems hörte:

```
sudo microk8s kubectl port-forward -n ingress-nginx service/ingress-nginx-controller 443:443 --address 127.0.0.1
```

Damit hat es dann geklappt:



13 Helm und Portainer auf Kubernetes

13.1 Helm

Helm ist der Paketmanager von Kubernetes. Damit kann man Kubernetes yaml.-Dateien, die ein Deployment beschreiben, organisieren und verteilen.

In Helm werden yaml.-Dateien als sogenannte **Charts** zusammengefasst. Ein Chart ist eine Sammlung von allen Kubernetes-Manifesten, die eine Anwendung braucht, um deployed zu werden.

Ein **Release** ist eine Version des Charts, da dieser natürlich auch versioniert sein will (wie bei git).

Helm ist installiert und erkennt den aktuellen Kubernetes-Kontext.

```
(base) jonas@jon-2 ~ % helm version

version.BuildInfo{Version:"v4.1.3", GitCommit:"c94d381b03be117e7e57908edbf642104e00eb8f", GitTreeState:"clean"
, GoVersion:"go1.26.1", KubeClientVersion:"v1.35"}
(base) jonas@jon-2 ~ % kubectl config current-context
docker-desktop
(base) jonas@jon-2 ~ % helm list
NAME    NAMESPACE       REVISION        UPDATED STATUS  CHART   APP VERSION
(base) jonas@jon-2 ~ %
```

13.2 Portainer mit Helm installieren

Um Helm zu testen, installieren wir Portainer über Helm (Portainer ist eigentlich gar nicht so wichtig, da wir ja Lens haben, aber es ist einfach ein Helm-Beispiel. Es geht nur ums Testen von Helm). Wir fügen also zunächst das Portainer-Repository hinzu und installieren dann Portainer:

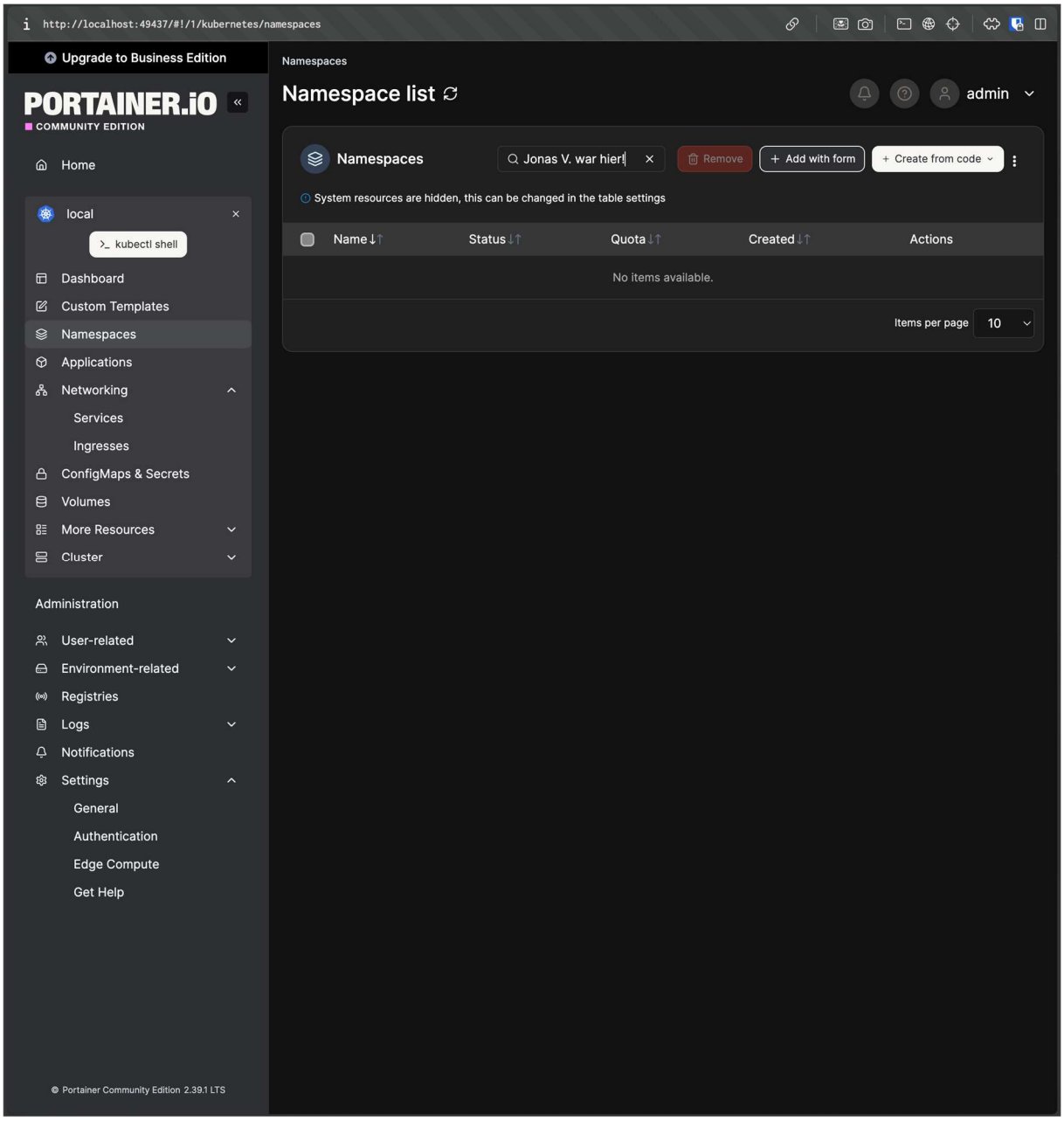
```
(base) jonas@jon-2 ~ % helm repo add portainer https://portainer.github.io/k8s/
"portainer" has been added to your repositories
(base) jonas@jon-2 ~ % helm list
NAME    NAMESPACE       REVISION        UPDATED STATUS  CHART   APP VERSION
(base) jonas@jon-2 ~ % helm repo update


Hang tight while we grab the latest from your chart repositories...
...Successfully got an update from the "portainer" chart repository
Update Complete. ⎈Happy Helming!⎈
(base) jonas@jon-2 ~ % helm install --create-namespace -n portainer portainer portainer/portainer


NAME: portainer
LAST DEPLOYED: Tue Mar 31 20:26:36 2026
NAMESPACE: portainer
STATUS: deployed
REVISION: 1
DESCRIPTION: Install complete
NOTES:
Get the application URL by running these commands:
    export NODE_PORT=$(kubectl get --namespace portainer -o jsonpath="{.spec.ports[1].nodePort}" services port
ainer)
  export NODE_IP=$(kubectl get nodes --namespace portainer -o jsonpath="{.items[0].status.addresses[0].address
}")
  echo https://$NODE_IP:$NODE_PORT
```

Portainer läuft nun, aber ohne Load-Balancer.

Ich fahre das Deployment nochmal runter, indem ich den Namespace lösche. Dann erstelle ich den Namespace neu und deploye Portainer erneut, diesmal mit Load Balancer:

```
(base) jonas@jon-2 ~ % kubectl delete namespace portainer


namespace "portainer" deleted
(base) jonas@jon-2 ~ % kubectl create namespace portainer


namespace/portainer created
(base) jonas@jon-2 ~ % helm install --create-namespace -n portainer portainer portainer/portainer --set servic
e.type=LoadBalancer


NAME: portainer
LAST DEPLOYED: Tue Mar 31 20:32:45 2026
NAMESPACE: portainer
STATUS: deployed
REVISION: 1
DESCRIPTION: Install complete
NOTES:
Get the application URL by running these commands:
     NOTE: It may take a few minutes for the LoadBalancer IP to be available.
           You can watch the status of by running 'kubectl get --namespace portainer svc -w portainer'
  export SERVICE_IP=$(kubectl get svc --namespace portainer portainer --template "{{ range (index .status.load
Balancer.ingress 0) }}{{.}}{{ end }}")
  echo https://$SERVICE_IP:9443
(base) jonas@jon-2 ~ % kubectl get --namespace portainer svc -w portainer
NAME        TYPE           CLUSTER-IP     EXTERNAL-IP   PORT(S)                                         AGE
portainer   LoadBalancer   10.96.87.232   172.18.0.5    9000:30334/TCP,9443:30923/TCP,8000:32473/TCP   16s
^C%
(base) jonas@jon-2 ~ % kubectl get --namespace portainer svc -w portainer
NAME        TYPE           CLUSTER-IP     EXTERNAL-IP   PORT(S)                                         AGE
portainer   LoadBalancer   10.96.87.232   172.18.0.5    9000:30334/TCP,9443:30923/TCP,8000:32473/TCP   70s
(base) jonas@jon-2 ~ %
```

Portainer kann nun auf localhost:9000 erreicht werden:



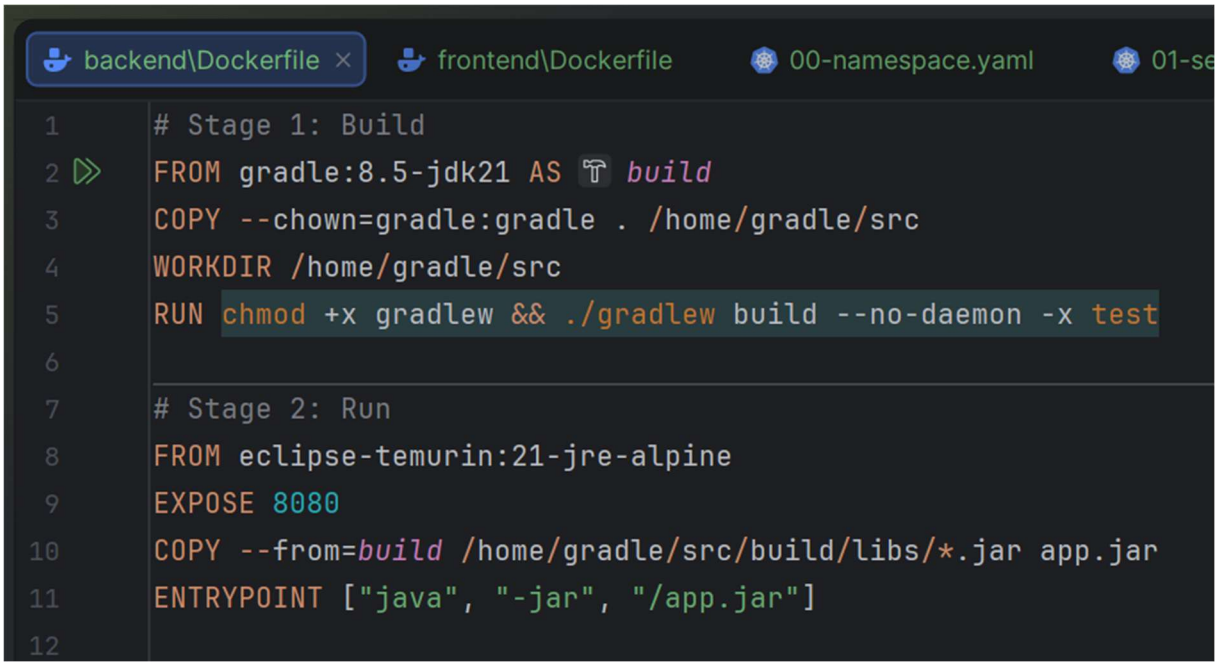
Damit ist Portainer auf meinem Kubernetes Cluster mit Helm installiert und ausgeführt worden.

14 Eigenes Projekt auf Kubernetes umsetzen

Das Projekt **`BAemtli`** ist eine Fullstack-Applikation zur Verwaltung von Ämtli im ICT-Campus der Schweizerischen Post AG (mein Lehrbetrieb). Sie besteht aus einem Spring Boot Backend, einem Next.js Frontend und einer MySQL-Datenbank. Die Applikation ist aktuell noch in Entwicklung, das Backend ist aber bereits fast komplett umgesetzt. Das Frontend besteht zurzeit erst aus einer Test-Seite, die eine CRUD-Operation (create) auf den Endpunkt /teams unterstützt. Ziel des hier vorgelegten Kubernetesprojekts war, die Orchestrierung der gesamten Infrastruktur mittels Kubernetes (K8s), wobei ich einen Fokus auf Skalierbarkeit und Datenpersistenz legte.

14.1 Docker-Images

Zuerst habe ich mich um die Containerisierung meiner Komponenten gekümmert. Für das Backend erstellte ich ein **`Dockerfile`** mit einem Multistage Build, um die Build-Umgebung sauber von der Laufzeitumgebung zu trennen. Ich nutzte ein Gradle-Image, um die Applikation zu bauen, und kopierte das fertige JAR-File anschliessend in ein schlankes JRE-Image auf Basis von Java 21.



Danach bereitete ich das **`Dockerfile`** für mein Frontend vor. Hierbei habe ich, den Standalone-Modus von Next.js genutzt, damit das finale Image so klein wie möglich bleibt.

backend\Dockerfile | frontend\Dockerfile | 00-namespace.yaml | 01-secrets.yaml

```dockerfile
# Stage 1: Dependencies
FROM node:20-alpine AS deps
RUN apk add --no-cache libc6-compat
WORKDIR /app
COPY package.json package-lock.json ./
RUN npm ci

# Stage 2: Build
FROM node:20-alpine AS builder
WORKDIR /app
COPY --from=deps /app/node_modules ./node_modules
COPY . .
# TailwindCSS und TypeScript will be processed during bild
RUN npm run build

# Stage 3: Runner (optimized for standalone)
FROM node:20-alpine AS runner
WORKDIR /app
ENV NODE_ENV=production

# Add a user to improve safety
RUN addgroup --system --gid 1001 nodejs
RUN adduser --system --uid 1001 nextjs

# Copy only necessary files from standalone output
COPY --from=builder /app/public ./public
COPY --from=builder --chown=nextjs:nodejs /app/.next/standalone ./
COPY --from=builder --chown=nextjs:nodejs /app/.next/static ./.next/static

USER nextjs
EXPOSE 3000
ENV PORT 3000

CMD ["node", "server.js"]
```

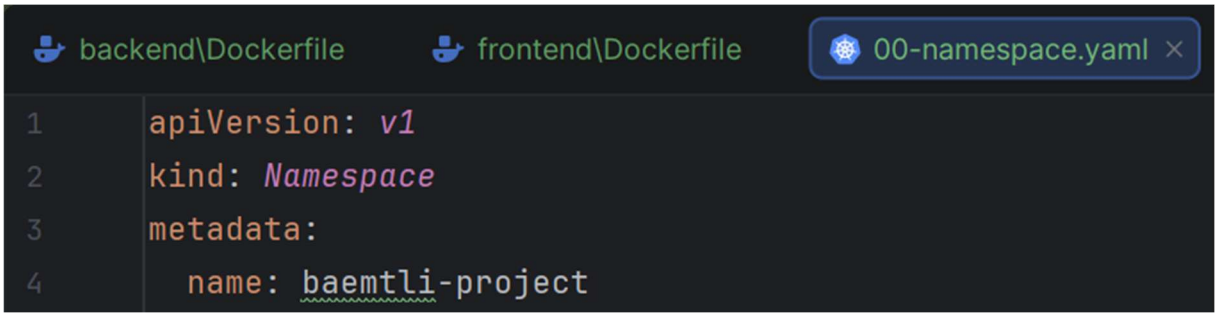
Für die Datenbank (MySQL) musste ich kein Dockerfile (Image) erstellen, da es hierfür bereits Images gibt.

14.2 Kubernetes vorbereiten

Für die lokale Kubernetes-Umgebung verwendete ich Docker Desktop und startete dort den integrierten Cluster. Zudem installierte ich die Lens IDE, um mein Cluster und Deployment zu überwachen. Als Entwicklungsumgebung nutze ich IntelliJ. IntelliJ bietet sehr gutes Syntax-Highlighting die .yaml-Dateien von Kubernetes.

14.2.1 Namespace erstellen

Zur besseren Übersicht im Cluster erstellte ich ein Namespace-File und wendete es mit dem Befehl **`kubectl apply -f k8s/00-namespace.yaml`** an, um den Bereich **`baemtli-project`** zu definieren.



14.2.2 Secrets auslagern

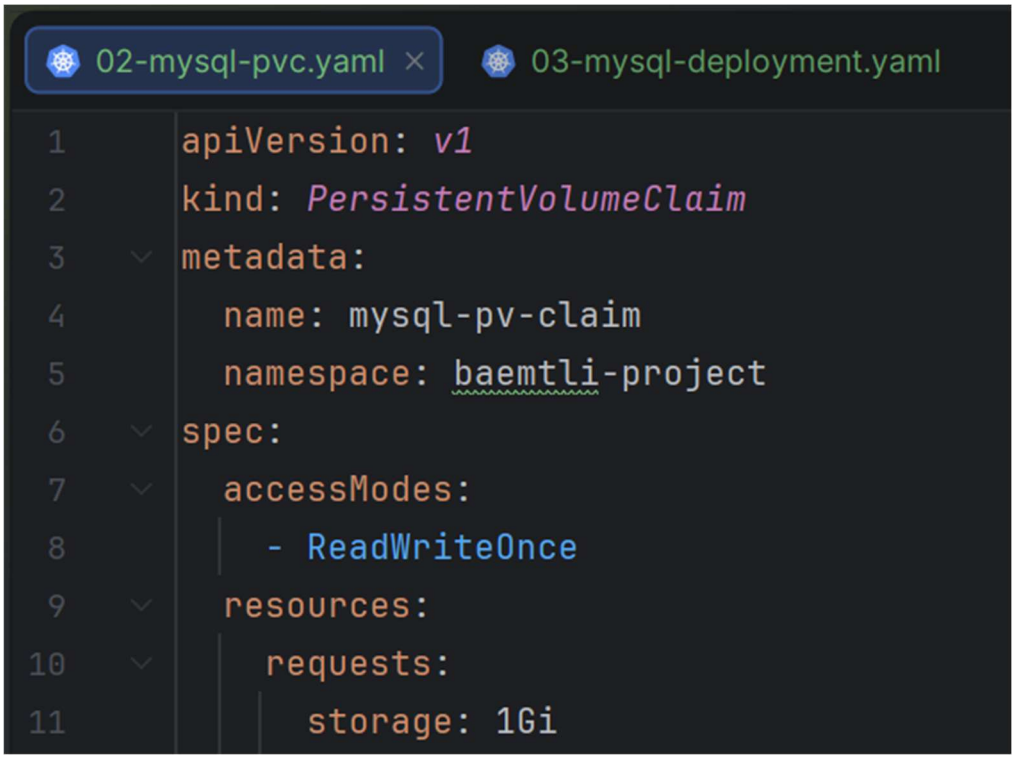
Danach legte ich meine Secrets an, in denen ich sensible Informationen wie die Datenbank-Passwörter und mein JWT-Secret speicherte. Dies geschah über die Datei **`01-secrets.yaml`**, damit diese Daten nicht im Klartext in den restlichen Konfigurationen auftauchen.

Angewendet habe ich die Datei mit **`kubectl apply -f k8s/01-secrets.yaml`**

14.3 Datenbank: MySQL Deployment

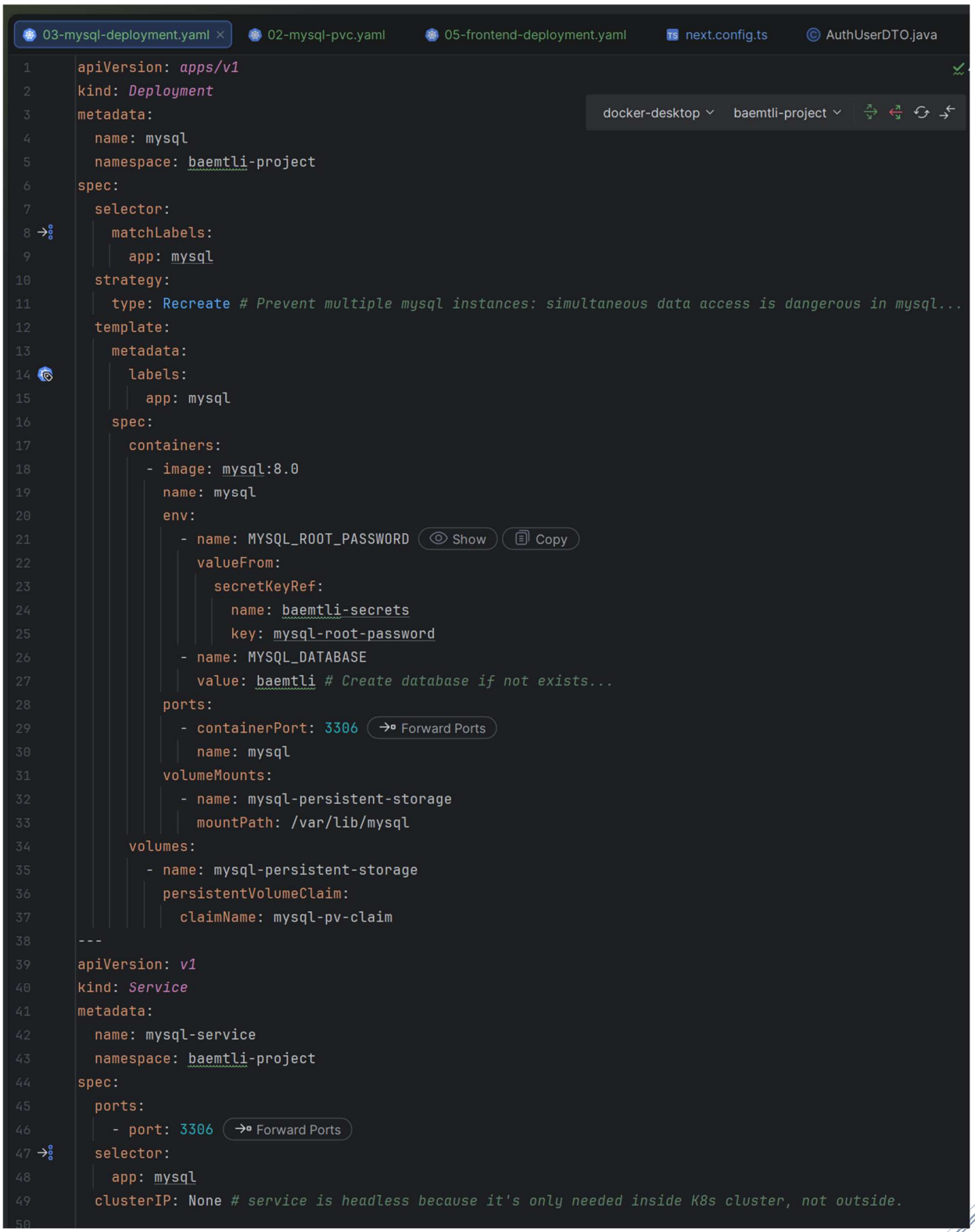
14.3.1 Datenpersistenz mit Persistent Volume Claim

Beim Deployment der Datenbank war mir wichtig, dass die Daten einen Neustart der Pods überleben. Ich definierte daher einen **`PersistentVolumeClaim`**, um festen Speicherplatz für MySQL zu reservieren. Ich entschied mich 1 GB Speicher zu reservieren.



14.3.2 MySQL Deployment

Danach erstellte ich das eigentliche Deployment für MySQL 8.0. Hier war es wichtig, **`Recreate`** zu setzen, da es nicht mehre MySQL Pods geben sollte, die gleichzeitig auf dieselbe Datenbank zugreifen. Zudem konnte ich hier bequem die für meine Applikation notwendige Datenbank **`baemtli`** anlegen lassen, ohne selbst DDL schreiben oder ausführen zu müssen.




```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mysql
  namespace: baemtli-project
spec:
  selector:
    matchLabels:
      app: mysql
  strategy:
    type: Recreate # Prevent multiple mysql instances: simultaneous data access is dangerous in mysql...
  template:
    metadata:
      labels:
        app: mysql
    spec:
      containers:
        - image: mysql:8.0
          name: mysql
          env:
            - name: MYSQL_ROOT_PASSWORD
              valueFrom:
                secretKeyRef:
                  name: baemtli-secrets
                  key: mysql-root-password
            - name: MYSQL_DATABASE
              value: baemtli # Create database if not exists...
          ports:
            - containerPort: 3306
              name: mysql
          volumeMounts:
            - name: mysql-persistent-storage
              mountPath: /var/lib/mysql
      volumes:
        - name: mysql-persistent-storage
          persistentVolumeClaim:
            claimName: mysql-pv-claim
---
apiVersion: v1
kind: Service
metadata:
  name: mysql-service
  namespace: baemtli-project
spec:
  ports:
    - port: 3306
  selector:
    app: mysql
  clusterIP: None # service is headless because it's only needed inside K8s cluster, not outside.
```

Auch die **`CREATE TABLE`** Befehle musste ich nicht manuell in einem sql-Skript vorbereiten, da ich im Backend mit Spring Boot und Hibernate arbeite. Wenn man in **`application.properties`** folgende Variable setzt **`spring.jpa.hibernate.ddl-auto=update`** erstellt Hibernate die benötigten Tabellen selbst.

```
# HIBERNATE
# Auto add tables if not existing
spring.jpa.hibernate.ddl-auto=update
spring.jpa.hibernate.naming.physical-strategy=org.hibernate.boot.model.naming.PhysicalNamingStrategyStandardImp

# Logging to help with debugging on K8s
spring.jpa.show-sql=true
```

Das Deployment habe ich dann gestartet mit **`kubectl apply -f k8s/03-mysql-deployment.yaml`**

```
PS C:\proj\Java\JAVA_BAemtli> kubectl apply -f k8s/03-mysql-deployment.yaml
deployment.apps/mysql created
Warning: spec.SessionAffinity is ignored for headless services
service/mysql-service created
```

Um sicherzugehen, dass alles korrekt initialisiert wurde, prüfte ich die Logs mit dem Befehl **`kubectl logs -f mysql-784554974d-n6t7b -n baemtli-project`**. Es zeigte sich, dass die Datenbank baemtli erfolgreich angelegt wurde:

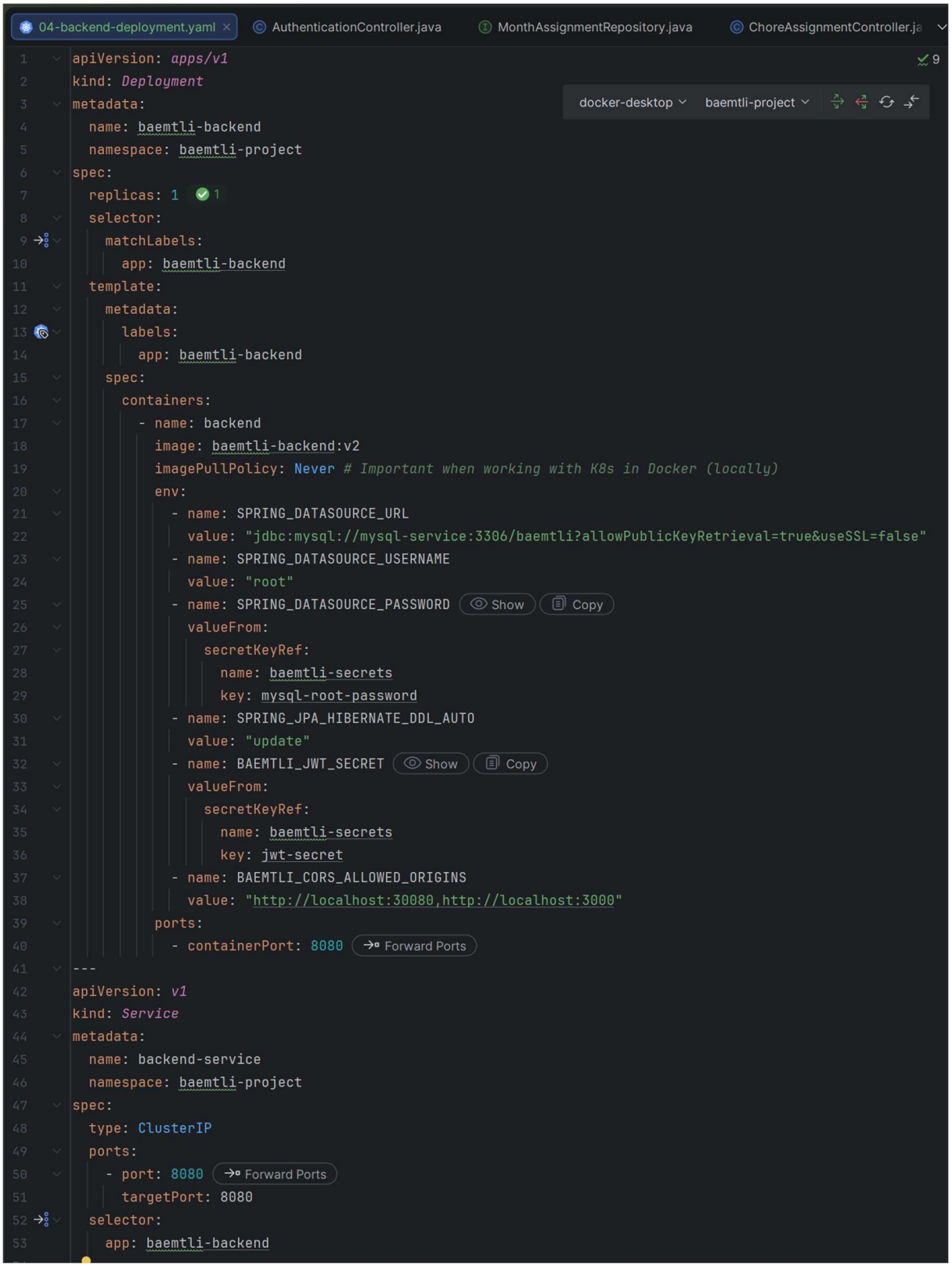
```
PS C:\proj\Java\JAVA_BAemtli> kubectl logs -f mysql-784554974d-n6t7b -n baemtli-project
2026-03-24 14:36:10+00:00 [Note] [Entrypoint]: Entrypoint script for MySQL Server 8.0.45-1.el9 started.
2026-03-24 14:36:11+00:00 [Note] [Entrypoint]: Switching to dedicated user 'mysql'
2026-03-24 14:36:11+00:00 [Note] [Entrypoint]: Entrypoint script for MySQL Server 8.0.45-1.el9 started.
2026-03-24 14:36:11+00:00 [Note] [Entrypoint]: Initializing database files
2026-03-24T14:36:11.145114Z 0 [Warning] [MY-011068] [Server] The syntax '--skip-host-cache' is deprecated and will be removed in a future release. Please use SET GLOB
 host_cache_size=0 instead.
2026-03-24T14:36:11.145241Z 0 [System] [MY-013169] [Server] /usr/sbin/mysqld (mysqld 8.0.45) initializing of server in progress as process 82
2026-03-24T14:36:11.157735Z 1 [System] [MY-013576] [InnoDB] InnoDB initialization has started.
2026-03-24T14:36:12.284078Z 1 [System] [MY-013577] [InnoDB] InnoDB initialization has ended.
2026-03-24T14:36:14.669655Z 6 [Warning] [MY-010453] [Server] root@localhost is created with an empty password ! Please consider switching off the --initialize-insecur
option.
2026-03-24 14:36:20+00:00 [Note] [Entrypoint]: Database files initialized
2026-03-24 14:36:20+00:00 [Note] [Entrypoint]: Starting temporary server
2026-03-24T14:36:21.018362Z 0 [Warning] [MY-011068] [Server] The syntax '--skip-host-cache' is deprecated and will be removed in a future release. Please use SET GLOB
 host_cache_size=0 instead.
2026-03-24T14:36:21.020109Z 0 [System] [MY-010116] [Server] /usr/sbin/mysqld (mysqld 8.0.45) starting as process 126
2026-03-24T14:36:21.035457Z 1 [System] [MY-013576] [InnoDB] InnoDB initialization has started.
2026-03-24T14:36:21.427614Z 1 [System] [MY-013577] [InnoDB] InnoDB initialization has ended.
2026-03-24T14:36:21.864939Z 0 [Warning] [MY-010068] [Server] CA certificate ca.pem is self signed.
2026-03-24T14:36:21.865015Z 0 [System] [MY-013602] [Server] Channel mysql_main configured to support TLS. Encrypted connections are now supported for this channel.
2026-03-24T14:36:21.882878Z 0 [Warning] [MY-011810] [Server] Insecure configuration for --pid-file: Location '/var/run/mysqld' in the path is accessible to all OS use
. Consider choosing a different directory.
2026-03-24T14:36:21.928201Z 0 [System] [MY-011323] [Server] X Plugin ready for connections. Socket: /var/run/mysqld/mysqlx.sock
2026-03-24T14:36:21.928290Z 0 [System] [MY-010931] [Server] /usr/sbin/mysqld: ready for connections. Version: '8.0.45'  socket: '/var/run/mysqld/mysqld.sock'  port: 0
MySQL Community Server - GPL.
2026-03-24 14:36:21+00:00 [Note] [Entrypoint]: Temporary server started.
'/var/lib/mysql/mysql.sock' -> '/var/run/mysqld/mysqld.sock'
Warning: Unable to load '/usr/share/zoneinfo/iso3166.tab' as time zone. Skipping it.
Warning: Unable to load '/usr/share/zoneinfo/leap-seconds.list' as time zone. Skipping it.
Warning: Unable to load '/usr/share/zoneinfo/leapseconds' as time zone. Skipping it.
Warning: Unable to load '/usr/share/zoneinfo/tzdata.zi' as time zone. Skipping it.
Warning: Unable to load '/usr/share/zoneinfo/zone.tab' as time zone. Skipping it.
Warning: Unable to load '/usr/share/zoneinfo/zone1970.tab' as time zone. Skipping it.
2026-03-24 14:36:24+00:00 [Note] [Entrypoint]: Creating database baemtli

2026-03-24 14:36:24+00:00 [Note] [Entrypoint]: Stopping temporary server
2026-03-24T14:36:24.034243Z 11 [System] [MY-013172] [Server] Received SHUTDOWN from user root. Shutting down mysqld (Version: 8.0.45).
2026-03-24T14:36:26.520262Z 0 [System] [MY-010910] [Server] /usr/sbin/mysqld: Shutdown complete (mysqld 8.0.45)  MySQL Community Server - GPL.
2026-03-24 14:36:27+00:00 [Note] [Entrypoint]: Temporary server stopped

2026-03-24 14:36:27+00:00 [Note] [Entrypoint]: MySQL init process done. Ready for start up.

2026-03-24T14:36:27.250511Z 0 [Warning] [MY-011068] [Server] The syntax '--skip-host-cache' is deprecated and will be removed in a future release. Please use SET GLOB
 host_cache_size=0 instead.
2026-03-24T14:36:27.251633Z 0 [System] [MY-010116] [Server] /usr/sbin/mysqld (mysqld 8.0.45) starting as process 1
2026-03-24T14:36:27.261884Z 1 [System] [MY-013576] [InnoDB] InnoDB initialization has started.
2026-03-24T14:36:27.669642Z 1 [System] [MY-013577] [InnoDB] InnoDB initialization has ended.
2026-03-24T14:36:27.969886Z 0 [Warning] [MY-010068] [Server] CA certificate ca.pem is self signed.
2026-03-24T14:36:27.969925Z 0 [System] [MY-013602] [Server] Channel mysql_main configured to support TLS. Encrypted connections are now supported for this channel.
2026-03-24T14:36:27.977319Z 0 [Warning] [MY-011810] [Server] Insecure configuration for --pid-file: Location '/var/run/mysqld' in the path is accessible to all OS use
. Consider choosing a different directory.
2026-03-24T14:36:27.991126Z 0 [System] [MY-011323] [Server] X Plugin ready for connections. Bind-address: '::' port: 33060, socket: /var/run/mysqld/mysqlx.sock
2026-03-24T14:36:27.991185Z 0 [System] [MY-010931] [Server] /usr/sbin/mysqld: ready for connections. Version: '8.0.45'  socket: '/var/run/mysqld/mysqld.sock'  port: 3
6  MySQL Community Server - GPL.
```

14.4 Backend: Java Spring Boot API Deployment

Für das Projekt BAemtli entwickle ich eine API mit Java und Spring Boot. Die API verfügt bereits über einen Grossteil der für die Applikation notwendigen Endpunkte. Im Rahmen dieses Kubernetes-Projekts wollte ich den Endpunkt **`/api/v1/teams`** für **`GET`**- und **`POST`**-Anfragen öffnen, und zwar ohne Authentifizierung (dies würde den Rahmen dieses Projekts sprengen). Deshalb habe ich, nur im Rahmen dieses Kuberenetes-Projekts, in der **`SecurityConfig.java`** für diese zwei Methoden auf genau diesem Endpoint gelockert, so dass man sie auch unangemeldet aufrufen darf:

```
// --- TEAMS ---
// Temporarily open for gibb K8s Project
.requestMatchers(HttpMethod.GET, apiPrefix + "/teams/**").permitAll()
.requestMatchers(HttpMethod.POST, apiPrefix + "/teams/**").permitAll()
.requestMatchers(HttpMethod.GET, apiPrefix + "/teams/**").hasAuthority(Permission.TEAM_READ_ALL)
  .requestMatchers(HttpMethod.POST, apiPrefix + "/teams/**").hasAuthority(Permission.TEAM_WRITE_ALL)
.requestMatchers(HttpMethod.PATCH, apiPrefix + "/teams/**").hasAuthority(Permission.TEAM_WRITE_ALL)
.requestMatchers(HttpMethod.DELETE, apiPrefix + "/teams/**").hasAuthority(Permission.TEAM_WRITE_ALL)
```

Danach erstellte ich die YAML-Datei für das Backend. Ich konfigurierte die Umgebungsvariablen so, dass das Backend den internen Service-Namen der Datenbank nutzt, um die Verbindung aufzubauen.



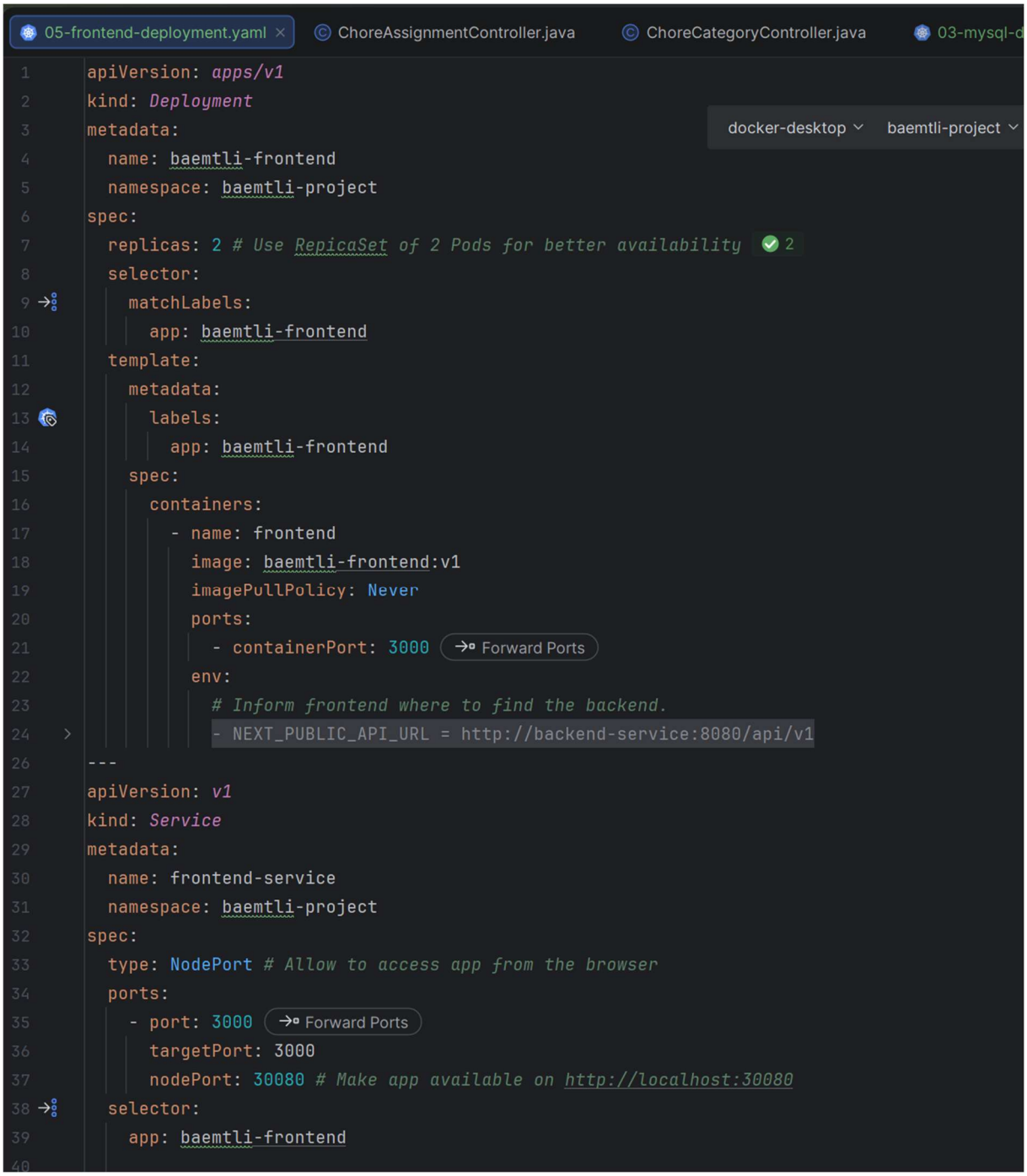

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: baemtli-backend
  namespace: baemtli-project
spec:
  replicas: 1
  selector:
    matchLabels:
      app: baemtli-backend
  template:
    metadata:
      labels:
        app: baemtli-backend
    spec:
      containers:
        - name: backend
          image: baemtli-backend:v2
          imagePullPolicy: Never # Important when working with K8s in Docker (locally)
          env:
            - name: SPRING_DATASOURCE_URL
              value: "jdbc:mysql://mysql-service:3306/baemtli?allowPublicKeyRetrieval=true&useSSL=false"
            - name: SPRING_DATASOURCE_USERNAME
              value: "root"
            - name: SPRING_DATASOURCE_PASSWORD
              valueFrom:
                secretKeyRef:
                  name: baemtli-secrets
                  key: mysql-root-password
            - name: SPRING_JPA_HIBERNATE_DDL_AUTO
              value: "update"
            - name: BAEMTLI_JWT_SECRET
              valueFrom:
                secretKeyRef:
                  name: baemtli-secrets
                  key: jwt-secret
            - name: BAEMTLI_CORS_ALLOWED_ORIGINS
              value: "http://localhost:30080,http://localhost:3000"
          ports:
            - containerPort: 8080
---
apiVersion: v1
kind: Service
metadata:
  name: backend-service
  namespace: baemtli-project
spec:
  type: ClusterIP
  ports:
    - port: 8080
      targetPort: 8080
  selector:
    app: baemtli-backend
```

Beim Starten der Applikation beobachtete ich die Logs mit **`kubectl logs -f baemtli-backend-5864797fff-6kl4s -n baemtli-project`** und stellte fest, dass Hibernate wie gewünscht alle Tabellen in der Datenbank automatisch generierte:

```
PS C:\proj\Java\JAVA_BAemtli> kubectl apply -f k8s/04-backend-deployment.yaml
deployment.apps/baemtli-backend created
service/backend-service created

PS C:\proj\Java\JAVA_BAemtli> kubectl logs -f baemtli-backend-5864797fff-6kl4s -n baemtli-project

  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/

 :: Spring Boot ::                (v4.0.4)

2026-03-24T14:42:29.205Z  INFO 1 --- [baemtli] [           main] n.ictcampus.baemtli.BaemtliApplication   : Starting BaemtliApplication v0.0.1-SNAPSHOT using Java 21.0.
10 with PID 1 (/app.jar started by root in /)
2026-03-24T14:42:29.215Z  INFO 1 --- [baemtli] [           main] n.ictcampus.baemtli.BaemtliApplication   : No active profile set, falling back to 1 default profile: "d
efault"
2026-03-24T14:42:30.485Z  INFO 1 --- [baemtli] [           main] .s.d.r.c.RepositoryConfigurationDelegate : Bootstrapping Spring Data JPA repositories in DEFAULT mode.
2026-03-24T14:42:30.562Z  INFO 1 --- [baemtli] [           main] .s.d.r.c.RepositoryConfigurationDelegate : Finished Spring Data repository scanning in 66 ms. Found 8 J
PA repository interfaces.
2026-03-24T14:42:31.567Z  INFO 1 --- [baemtli] [           main] o.s.boot.tomcat.TomcatWebServer          : Tomcat initialized with port 8080 (http)
2026-03-24T14:42:31.590Z  INFO 1 --- [baemtli] [           main] o.apache.catalina.core.StandardService   : Starting service [Tomcat]
2026-03-24T14:42:31.590Z  INFO 1 --- [baemtli] [           main] o.apache.catalina.core.StandardEngine    : Starting Servlet engine: [Apache Tomcat/11.0.18]
2026-03-24T14:42:31.630Z  INFO 1 --- [baemtli] [           main] b.w.c.s.WebApplicationContextInitializer : Root WebApplicationContext: initialization completed in 2335
 ms
2026-03-24T14:42:31.783Z  WARN 1 --- [baemtli] [           main] JpaBaseConfiguration$JpaWebConfiguration : spring.jpa.open-in-view is enabled by default. Therefore, da
tabase queries may be performed during view rendering. Explicitly configure spring.jpa.open-in-view to disable this warning
2026-03-24T14:42:31.995Z  INFO 1 --- [baemtli] [           main] org.hibernate.orm.jpa                    : HHH008540: Processing PersistenceUnitInfo [name: default]
2026-03-24T14:42:32.093Z  INFO 1 --- [baemtli] [           main] org.hibernate.orm.core                   : HHH000001: Hibernate ORM core version 7.2.7.Final
2026-03-24T14:42:32.631Z  INFO 1 --- [baemtli] [           main] o.s.o.j.p.SpringPersistenceUnitInfo      : No LoadTimeWeaver setup: ignoring JPA class transformer
2026-03-24T14:42:32.670Z  INFO 1 --- [baemtli] [           main] com.zaxxer.hikari.HikariDataSource       : HikariPool-1 - Starting...
2026-03-24T14:42:32.907Z  INFO 1 --- [baemtli] [           main] com.zaxxer.hikari.pool.HikariPool        : HikariPool-1 - Added connection com.mysql.cj.jdbc.Connection
Impl@5b4d9bda
2026-03-24T14:42:32.910Z  INFO 1 --- [baemtli] [           main] com.zaxxer.hikari.HikariDataSource       : HikariPool-1 - Start completed.
2026-03-24T14:42:33.017Z  INFO 1 --- [baemtli] [           main] org.hibernate.orm.connections.pooling    : HHH10001005: Database info:
        Database JDBC URL [jdbc:mysql://mysql-service:3306/baemtli?allowPublicKeyRetrieval=true&useSSL=false]
        Database driver: MySQL Connector/J
        Database dialect: MySQLDialect
        Database version: 8.0.45
        Default catalog/schema: baemtli/undefined
        Autocommit mode: undefined/unknown
        Isolation level: REPEATABLE_READ [default REPEATABLE_READ]
        JDBC fetch size: none
        Pool: DataSourceConnectionProvider
        Minimum pool size: undefined/unknown
        Maximum pool size: undefined/unknown
2026-03-24T14:42:34.052Z  INFO 1 --- [baemtli] [           main] org.hibernate.orm.core                   : HHH000489: No JTA platform available (set 'hibernate.transac
tion.jta.platform' to enable JTA platform integration)
2026-03-24T14:42:36.714Z  INFO 1 --- [baemtli] [           main] j.LocalContainerEntityManagerFactoryBean : Initialized JPA EntityManagerFactory for persistence unit 'd
efault'
2026-03-24T14:42:37.008Z  INFO 1 --- [baemtli] [           main] o.s.d.j.r.query.QueryEnhancerFactories   : Hibernate is in classpath; If applicable, HQL parser will be
 used.
2026-03-24T14:42:37.350Z  INFO 1 --- [baemtli] [           main] eAuthenticationProviderManagerConfigurer : Global AuthenticationManager configured with AuthenticationP
rovider bean with name authenticationProvider
2026-03-24T14:42:37.351Z  WARN 1 --- [baemtli] [           main] r$InitializeUserDetailsManagerConfigurer : Global AuthenticationManager configured with an Authenticati
onProvider bean. UserDetailsService beans will not be used by Spring Security for automatically configuring username/password login. Consider removing the Authenticatio
nProvider bean. Alternatively, consider using the UserDetailsService in a manually instantiated DaoAuthenticationProvider. If the current configuration is intentional,
to turn off this warning, increase the logging level of 'org.springframework.security.config.annotation.authentication.configuration.InitializeUserDetailsBeanManagerCon
figurer' to ERROR
2026-03-24T14:42:38.304Z  INFO 1 --- [baemtli] [           main] o.s.boot.tomcat.TomcatWebServer          : Tomcat started on port 8080 (http) with context path '/'
2026-03-24T14:42:38.332Z  INFO 1 --- [baemtli] [           main] n.ictcampus.baemtli.BaemtliApplication   : Started BaemtliApplication in 9.795 seconds (process running
 for 11.019)
```

14.5 Frontend Deployment erstellen

Anschliessend kümmerte ich mich um das Frontend-Deployment. Hier entschied ich mich für ein Replikationsfaktor 2, damit zwei Frontend-Instanzen erstellt werden. Mit **`NEXT_PUBLIC_API_URL`** setzte ich die URL für das Backend, damit das Frontend weiss, wo es das Backend finden kann. Hier konnte ich bereits mit dem Servicenamen **`backend-service`** arbeiten, den ich im Backend Deployment definiert hatte. Zudem definierte ich einen **`NodePort`**-Service, damit die Web-Applikation von meinem Browser aus über den **`Port 30080`** erreichbar ist.

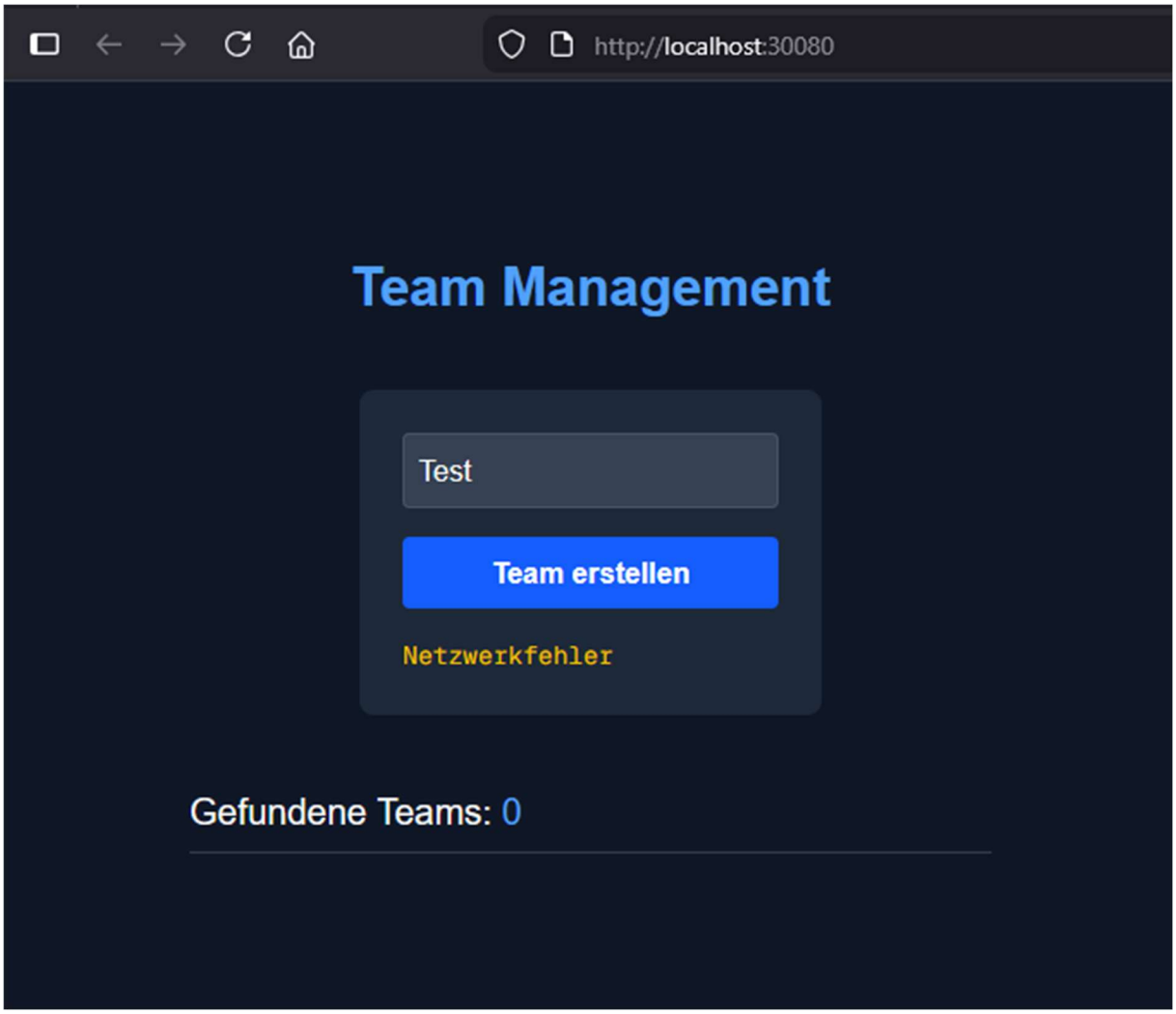


Das Deployment habe ich dann mit **`kubectl apply -f k8s/05-frontend-deployment.yaml`** angewendet.

Mit **`kubectl logs -f baemtli-frontend-7c48986595-kvrsl -n baemtli-project`** habe ich auch hier die Logs geprüft:

```
PS C:\proj\Java\JAVA_BAemtli> kubectl logs -f baemtli-frontend-7c48986595-kvrsl -n baemtli-project
▲ Next.js 16.2.1
- Local:         http://baemtli-frontend-7c48986595-kvrsl:3000
- Network:       http://baemtli-frontend-7c48986595-kvrsl:3000
✓ Ready in 0ms
```

Nun war die App zum ersten Mal im Browser erreichbar:



Die Verbindung zum Backend wurde allerdings noch durch CORS blockiert.

14.6 CORS Konfiguration anpassen

Obwohl die Container liefen, stiess ich beim ersten Aufruf auf CORS-Fehler, da mein Browser die Anfragen vom Frontend zum Backend blockierte. Ich musste also noch meine CorsConfig.java in der Spring Boot Applikation anpassen.

```java
@Configuration  Jonas Vetsch *
public class CorsConfig implements WebMvcConfigurer {

    // Load global API prefix from application.properties
    @Value("${baemtli.api.prefix}")
    private String apiPrefix;

    // CORS Configuration for K8s deployment
    @Value("${baemtli.cors.allowed-origins:http://localhost:3000}")
    private String allowedOrigins;

    @Override  no usages  Jonas Vetsch
    public void configurePathMatch(PathMatchConfigurer configurer) {
        // Fügt /api/v1 vor alle Klassen, die mit @RestController annotiert sind
        // dies sorgt für eine saubere Trennung zwischen Frontend und Backend auf URL-Ebene
        configurer.addPathPrefix(apiPrefix,
                Class<?> c -> c.isAnnotationPresent(RestController.class));
    }

    @Bean  Jonas Vetsch *
    public WebMvcConfigurer corsConfigurer() {
        return new WebMvcConfigurer() {  Jonas Vetsch *
            @Override  no usages  Jonas Vetsch *
            public void addCorsMappings(CorsRegistry registry) {
                registry.addMapping( pathPattern: apiPrefix + "/**")
                        // Schutz vor CSRF-Angriffen
                        // Welche URL darf auf das Backend zugreifen?
                        .allowedOrigins(allowedOrigins.split( regex: ","))
                        .allowedMethods("GET", "POST", "PUT", "DELETE", "OPTIONS")
                        // Welche Header werden akzeptiert? Bspw. Token kommen in Authorization. * ist eine W
                        .allowedHeaders("*")
                        .allowCredentials(true);
            }
        };
    }
}
```

Ich wollte die erlaubten Origins über eine Variable steuerbar machen. Die hier gezeigte **`@Value`** Annotation erlaubt es mir, die Origins über die K8s-YAML-Dateien flexibel in den Pod reinzugeben, ohne jedes Mal das Backend-Image neu bauen zu müssen, sobald ich die Origins anpassen will. Das wäre unpraktisch.

Die Variable setzte ich dann in **`04-backend-deployment.yaml`** wie folgend:

```yaml
- name: BAEMTLI_CORS_ALLOWED_ORIGINS
  value: "http://localhost:30080,http://localhost:3000"
```

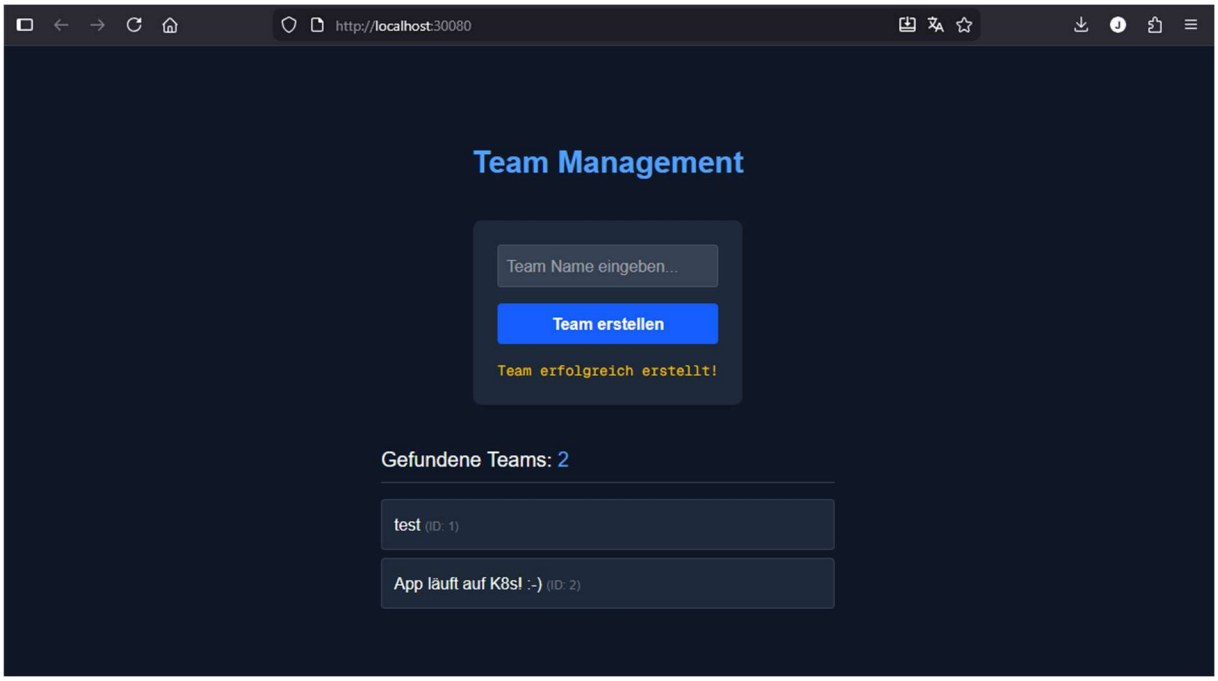
Danach baute das Backend-Image erneut, diesmal als Version v2 und hob im Deployment auf meinem Cluster in der Datei **`04-backend-deployment.yaml`** ebenfalls auf :**`v2`** an.

```
containers:
  - name: backend
    image: baemtli-backend:v2
```

Trotz dieser Anpassungen blieb das CORS-Problem zunächst bestehen. Dieses Problem blockierte mich eine Weile lange und ich musste viel recherchieren. Schliesslich fand ich folgende Erklärung: Mein Browser versuchte, das Backend unter localhost:8080 zu erreichen, was im isolierten Cluster ohne Weiteres nicht möglich ist. Zudem fehlten dadurch die nötigen CORS-Header in der Antwort. Da ich in der **`SecurityConfig.java`** bereits die korrekten CORS-Einstellungen hinterlegt hatte, dachte ich, dass die Konfiguration eigentlich stimmen musste. Und das war auch so, denn die Lösung war schlussendlich ein fehlendes Port-Forwarding, mit dem ich einen Tunnel von meinem Rechner zum Backend-Service im Cluster schuf.

```
kubectl port-forward service/backend-service 8080:8080 -n baemtli-project
```

Sobald dieser Tunnel stand, fand mein Browser das Backend unter **`localhost:8080`**. Da die Anfrage nun von **`localhost:30080`** kam und dieser Origin in meiner neuen Konfiguration erlaubt war, liess Spring Security den Zugriff auf die Teams-Endpunkte zu. Ich konnte GET und POST-Anfragen erfolgreich testen:



14.7 Datenpersistenz testen

Die spannende Frage ist nun: Persistieren die soeben erstellten Daten?

Zum Schluss wollte ich natürlich testen, ob meine Persistenz-Konfiguration wirklich funktioniert. Ich fuhr alle Deployments herunter, löschte die Pods und startete danach alles wieder neu.

```
PS C:\proj\Java\JAVA_BAemtli> kubectl delete deployment baemtli-backend baemtli-frontend mysql -n baemtli-project
deployment.apps "baemtli-backend" deleted from baemtli-project namespace
deployment.apps "baemtli-frontend" deleted from baemtli-project namespace
deployment.apps "mysql" deleted from baemtli-project namespace

PS C:\proj\Java\JAVA_BAemtli> kubectl apply -f k8s/03-mysql-deployment.yaml
deployment.apps/mysql created
service/mysql-service unchanged

PS C:\proj\Java\JAVA_BAemtli> kubectl apply -f k8s/04-backend-deployment.yaml
deployment.apps/baemtli-backend created
service/backend-service unchanged

PS C:\proj\Java\JAVA_BAemtli> kubectl apply -f k8s/05-frontend-deployment.yaml
deployment.apps/baemtli-frontend created
service/frontend-service unchanged
```

Dank des Persistent Volume Claims waren alle zuvor angelegten Daten in der Datenbank wieder verfügbar. Das Setup ist stabil und die Daten sind vom Lebenszyklus des Datenbank-Pods entkoppelt.