

MODUL 169

Services mit Containern bereitstellen

Abstract

This is the Document required for the Module 169. In this module we are creating and using services using container / docker.

Class: INF2023n

Start of documentation: 31.01.2025

Simeon Beetschen

Version 1.0

Contents

Tag 1.....	3
Was sind Container und wo sind sie nützlich?	3
Was ist DevOps	3
Unterschied Virtualisierung und Containerisierung.....	3
Unterschied Image und Container	3
Wichtige Befehle und ihre Funktionen.....	4
Basis-Befehle.....	4
Container-Management	4
Images bauen und verwalten.....	5
Netzwerk und Volumes	5
Docker Compose	6
Portainer (GUI-Verwaltung).....	7
Aufräumen.....	8
Printscreen OnlyOffice mit eigenem Namen und Datum im Dokument	8
Tag 2.....	9
App Version 1	9
Printscreen der Version 1 mit eigenem Namen als Todo-Task	9
Printscreen der Images bei GitLab	10
Aufgabe 1: «Ein Image pushen, alle Befehle im Portfolio festhalten	11
App Version 2 (Frontend)	13
Printscreen der Version 2 mit Namen als Todo-Task.....	13
Tag 3.....	14
Kurze Beschreibung: Was ist Docker Compose.....	14
Version 2 der App mit Docker Compose zum Laufen gebracht (Vorgehen, Befehle und Compose-Datei festgehalten)	14
Portainer installiert und mit PrintScreen festgehalten	16
App via Portainer installiert und Vorgehen dokumentiert	17
Das Learning-Beispiel «Shop» via Docker Compose installiert	20
Printscreen vom laufenden Shop, Prometheus und Grafana	20
Vorgehen im Portfolio festgehalten.....	22
Frage aus der Aufgabe:.....	23
Tag 4.....	25
Sie erstellen ein eigenes Projekt und erzeugen ein Image davon.....	25
Das Image pushen Sie in das Repository (Printscreen).....	25
Anleitung für die Installation.....	25



Tag 5.....	26
Was ist Kubernetes?.....	26
Was sind Microservices?	26
Vergleich von lightweight Kubernetes-Anwendungen	26
Kubernetes installiert und Anleitung erstellt.....	27
Verbindung zu Kubernetes-Server (Printscreen) mit Lens	28
Tag 6.....	29
Erläuterung: Was ist der Raft-konsens-Algorithmus und warum eine ungerade Anzahl an Servern in einem Cluster?	29
Was ist Raft?	29
Wie funktioniert Raft?	29
Warum eine ungerade Anzahl an Servern?	29
Dokumentation: Wie läuft die App in Kubernetes?	30
Eintrag zu: Self-Healing, Scale Down, Scale Up	30
Rolling Update funktioniert (V2 in Dashboard)(Screenshot)	31
Eintrag zu Blue-Green Deployment.....	31
Tag 7.....	32
Eintrag zu Cluster IP und Node IP	32
Eintrag zu LoadBalancer	32
PrintScreen Wie Sie auf die App zugreifen. Siehe Kap. Ingress	33
Erklärung warum sie bei "Ingress beim Zugriff auf 127.0.0.1 ein Error 404 erhalten	33
Installation von Portainer auf Kubernetes.....	34
Tag 8.....	35
Eigenes Projekt auf Kubernetes installiert und dokumentiert	35
Eigenes Kubernetes-Projekt: Terraria Server.....	35
Tag 9.....	38
Todo-App auf Podman zum Laufen gebracht und dokumentiert	38
Todo-App auf Podman:	38
Todo-App auf Podman, auf Kubernetes:	39



Tag 1

Was sind Container und wo sind sie nützlich?

Ein **Docker-Container** ist eine laufende Instanz eines Images. Während ein Image unverändert bleibt, kann ein Container Daten zur Laufzeit verändern (z. B. Logs, temporäre Dateien). Container nutzen den Kernel des Host-Systems, sind isoliert und effizient.

Was ist DevOps

DevOps ist ein Ansatz in der Softwareentwicklung, der die Zusammenarbeit zwischen Entwicklern (Development) und IT-Betrieb (Operations) verbessert. Ziel ist es, Software schneller, effizienter und zuverlässiger bereitzustellen. Dies wird durch Automatisierung, kontinuierliche Integration (CI), kontinuierliche Bereitstellung (CD) und enge Zusammenarbeit ermöglicht.

Wichtige Prinzipien von DevOps:

- Kollaboration** zwischen Entwicklern und Betriebsteams
- Automatisierung** von Infrastruktur, Tests und Deployments
- Kontinuierliche Integration & Bereitstellung (CI/CD)**
- Überwachung und Feedback** zur Verbesserung von Anwendungen

Unterschied Virtualisierung und Containerisierung

Virtualisierung nutzt einen Hypervisor, um komplette virtuelle Maschinen mit eigenem Betriebssystem zu erstellen. Das sorgt für hohe Isolation, benötigt aber mehr Ressourcen und eine längere Startzeit.

Containerisierung hingegen läuft auf einer Container-Engine wie Docker, teilt das Host-Betriebssystem und ist dadurch ressourcenschonender sowie schneller. Container sind ideal für moderne Microservices und DevOps, während Virtualisierung für heterogene Umgebungen mit unterschiedlichen Betriebssystemen sinnvoll bleibt.

Unterschied Image und Container

Ein **Docker-Image** ist eine unveränderliche Vorlage, die alle Komponenten (Code, Abhängigkeiten, Konfigurationen) enthält, um einen Container auszuführen. Es besteht aus Schichten (Layers) und wird in einer Registry (z. B. Docker Hub) gespeichert. Ein Image bleibt immer statisch.

Unterschied zwischen Image und Container

Image = Vorlage, unveränderlich



Container = laufende Instanz, kann zur Laufzeit Daten ändern

Löscht man einen Container, sind Änderungen weg, es sei denn, man speichert sie als neues Image

Wichtige Befehle und ihre Funktionen

Basis-Befehle

Images herunterladen (Pull)

```
docker pull <image>
```

Lädt ein bestimmtes Image (z.B. ubuntu, nginx, etc.) vom Docker Hub oder einer anderen Registry herunter.

Container starten (Run)

```
docker run <image>
```

Startet einen Container basierend auf dem angegebenen Image.

Häufige Parameter:

-d: (detached) Im Hintergrund starten

-p <hostPort>:<containerPort>: Portweiterleitung

-it: Interaktiv + TTY (für Bash-Shell)

Laufende bzw. vorhandene Container anzeigen

```
docker ps    # Zeigt laufende Container
```

```
docker ps -a # Zeigt alle Container (auch gestoppte)
```

Images auflisten

```
docker images    # Zeigt alle Basis-Images
```

```
docker images -a # Zeigt alle Images inkl. Zwischenebenen
```

Container-Management

Container stoppen

```
docker stop <containerName_oder_ID>
```

Versucht, den Container „sanft“ herunterzufahren.

Container sofort beenden („Kill“)

```
docker kill <containerName_oder_ID>
```

Beendet den Container abrupt (SIGKILL).



Container entfernen

```
docker rm <containerName_oder_ID>
```

Löscht den Container des Systems (nur möglich, wenn er gestoppt ist).

Logs eines Containers anzeigen

```
docker logs <containerName_oder_ID>
```

```
docker logs -f <containerName_oder_ID> # "live" mitlesen
```

Vorhandenen, gestoppten Container erneut starten

```
docker start <containerName_oder_ID>
```

Images bauen und verwalten

Image aus Dockerfile bauen

```
docker build -t <imagename:tag> .
```

Baut ein Image aus dem Dockerfile im aktuellen Verzeichnis.

-t <name:tag> legt den Namen und Tag fest, z.B. todo-app:v1.

Image mit neuem Tag versehen („Taggen“)

```
docker image tag <altes_image:tag> <neues_image:tag>
```

Macht eine Art „Alias“ für dasselbe Image, z.B. um es woanders hochzuladen.

Image hochladen (Push)

```
docker push <repository>/<imagename>:<tag>
```

Lädt das lokale Image in eine Registry hoch (z.B. Docker Hub oder GitLab).

Image löschen

```
docker rmi <imageName_oder_ID>
```

Netzwerk und Volumes

Netzwerk erstellen

```
docker network create <networkName>
```

Erstellt ein benutzerdefiniertes Docker-Netzwerk (Typ: bridge).

Container in ein Netzwerk starten

```
docker run --net=<networkName> ...
```

Volumes in Container mounten

```
docker run -v /host/pfad:/container/pfad <image>
```



Verbindet ein Host-Verzeichnis mit einem Container-Verzeichnis.

Docker Compose

Docker Compose installieren (Linux)

```
sudo apt install docker-compose
```

Docker Compose: Container starten

```
docker-compose up -d
```

Baut und startet alle in docker-compose.yml definierten Services.

Docker Compose: Container stoppen und entfernen

```
docker-compose down
```

Stoppt und entfernt alle zugehörigen Container (und Netzwerk).

Compose-Datei Beispiel (Ausschnitt)

```
version: "3"

services:
  todoapp:
    build: ./web-frontend
    ports:
      - "3000"
    depends_on:
      - redis-master
      - redis-slave
    networks:
      - todoapp_network

  redis-master:
    build: ./redis-master
    networks:
      - todoapp_network

  redis-slave:
```



```

build: ./redis-slave

depends_on:

  - redis-master

networks:

  - todoapp_network

```

```

networks:

  todoapp_network:

    name: todoapp_network

    driver: bridge

```

Definiert mehrere Services und ein benutzerdefiniertes Netzwerk.

Portainer (GUI-Verwaltung)

Portainer per Docker Compose starten

```

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

```

Aufruf:

```
docker-compose up -d
```

Erreichbar unter <http://localhost:9000>.



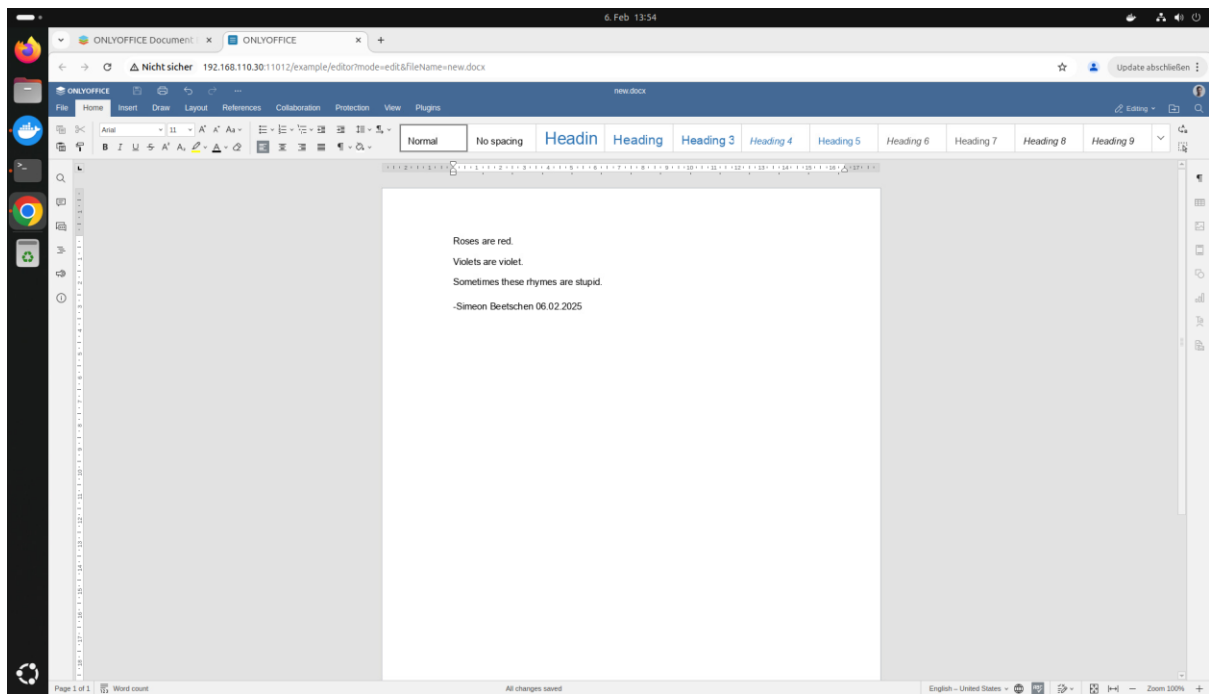
Aufräumen

Systembereinigung

`docker system prune -a --volumes`

Löscht **alle** ungenutzten Container, Images und Volumes (nicht laufende Container).

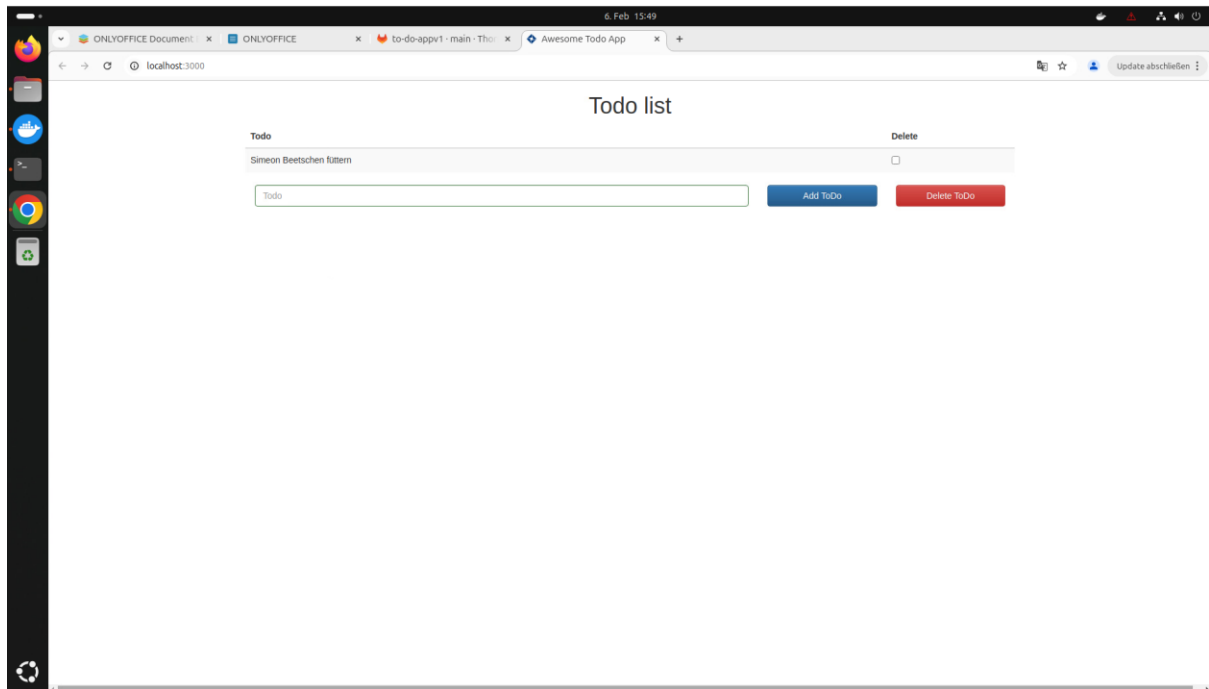
Printscreen OnlyOffice mit eigenem Namen und Datum im Dokument



Tag 2

App Version 1

Printscreen der Version 1 mit eigenem Namen als Todo-Task



Printscreen der Images bei GitLab

The image displays two screenshots of the GitLab Container Registry interface, showing the 'Container Registry' section for a project named 'm169_inf2023n_beetschen_simeon'.

Top Screenshot: todo-app

The interface shows the 'Container Registry' section for the 'todo-app' image. It displays 2 tags, with 'Cleanup disabled' and 'Created Feb 18, 2025 21:13'.

Tag	Size	Published	Digest
v1	5.55 MiB	Published 1 week ago	Digest: 2f0c6dc
v2	5.74 MiB	Published 5 days ago	Digest: 5b690f8

Bottom Screenshot: redis-slave

The interface shows the 'Container Registry' section for the 'redis-slave' image. It displays 2 tags, with 'Cleanup disabled' and 'Created Feb 18, 2025 21:13'.

Tag	Size	Published	Digest
v1	11.29 MiB	Published 1 week ago	Digest: af38205
v2	11.29 MiB	Published 1 week ago	Digest: af38205



The top screenshot shows the 'redis-master' repository page. It displays two tags: 'v1' (11.29 MB, published 1 week ago) and 'v2' (11.29 MB, published 1 week ago). The page includes a search bar, a 'Filter results' dropdown, and a 'Delete selected' button.

The bottom screenshot shows the 'Container Registry' overview page. It lists three repositories: 'm169_inf2023n_beetschen_simeon/todo-app', 'm169_inf2023n_beetschen_simeon/redis-slave', and 'm169_inf2023n_beetschen_simeon/redis-master'. Each repository has two tags and was published 3 minutes ago. The page includes a search bar, a 'Filter results' dropdown, and a 'Delete selected' button.

Aufgabe 1: «Ein Image pushen, alle Befehle im Portfolio festhalten

Access Token in Git erstellen und hier als Passwort angeben:

Als Nutzernamen den Gibb Namen nehmen

`docker login git-registry.gibb.ch`

Images erstellen (Auf Pfade achten):

`docker build -t redis-master:v1 ./redis-master --provenance false`

`docker build -t redis-slave:v1 ./redis-slave --provenance false`

`docker build -t todo-app:v1 ./web-frontend --provenance false`



Images nochmal Taggen:

```
docker image tag redis-master:v1 git-  
registry.gibb.ch/sbe150562/m169_inf2023n_beetschen_simeon/redis-master:v1  
docker image tag redis-slave:v1 git-  
registry.gibb.ch/sbe150562/m169_inf2023n_beetschen_simeon/redis-slave:v1  
docker image tag todo-app:v1 git-  
registry.gibb.ch/sbe150562/m169_inf2023n_beetschen_simeon/todo-app:v1
```

Um die Images zu pushen, diese Commands eingeben:

```
docker push git-registry.gibb.ch/sbe150562/m169_inf2023n_beetschen_simeon/redis-  
master:v1  
docker push git-registry.gibb.ch/sbe150562/m169_inf2023n_beetschen_simeon/redis-  
slave:v1  
docker push git-registry.gibb.ch/sbe150562/m169_inf2023n_beetschen_simeon/todo-app:v1
```

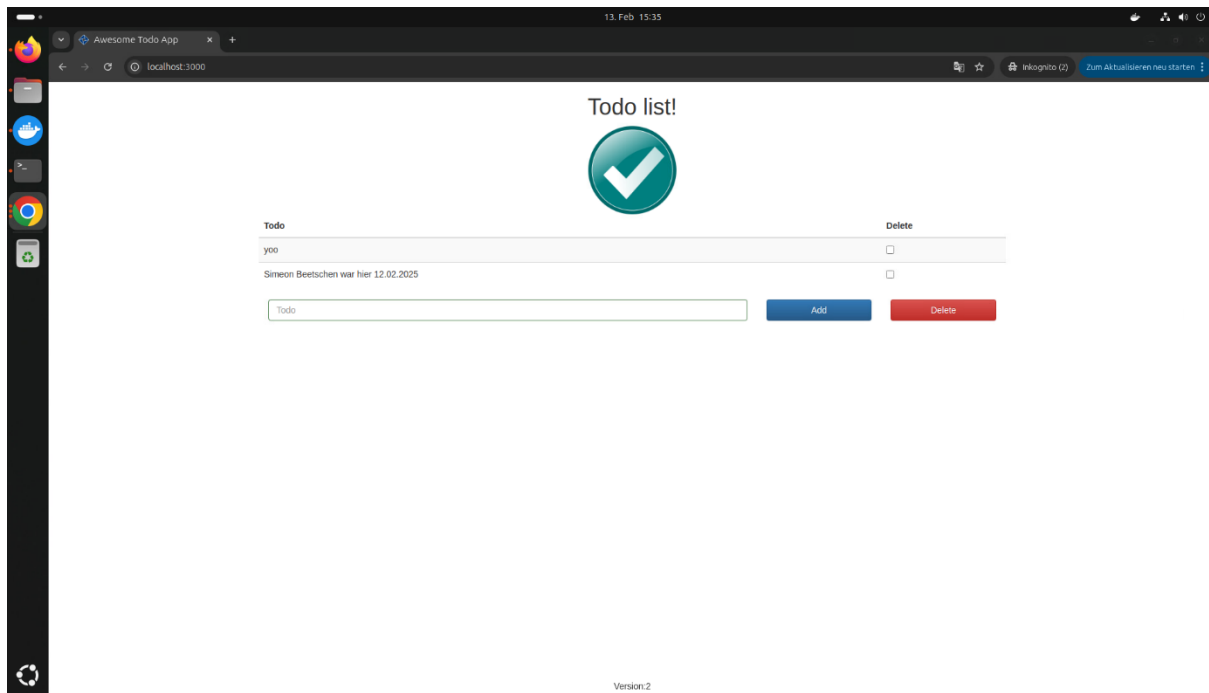
Images wieder pullen:

```
docker pull git-registry.gibb.ch/sbe150562/m169_inf2023n_beetschen_simeon/redis-  
master:v1  
docker pull git-registry.gibb.ch/sbe150562/m169_inf2023n_beetschen_simeon/redis-  
slave:v1  
docker pull git-registry.gibb.ch/sbe150562/m169_inf2023n_beetschen_simeon/todo-app:v1
```



App Version 2 (Frontend)

Printscreen der Version 2 mit Namen als Todo-Task



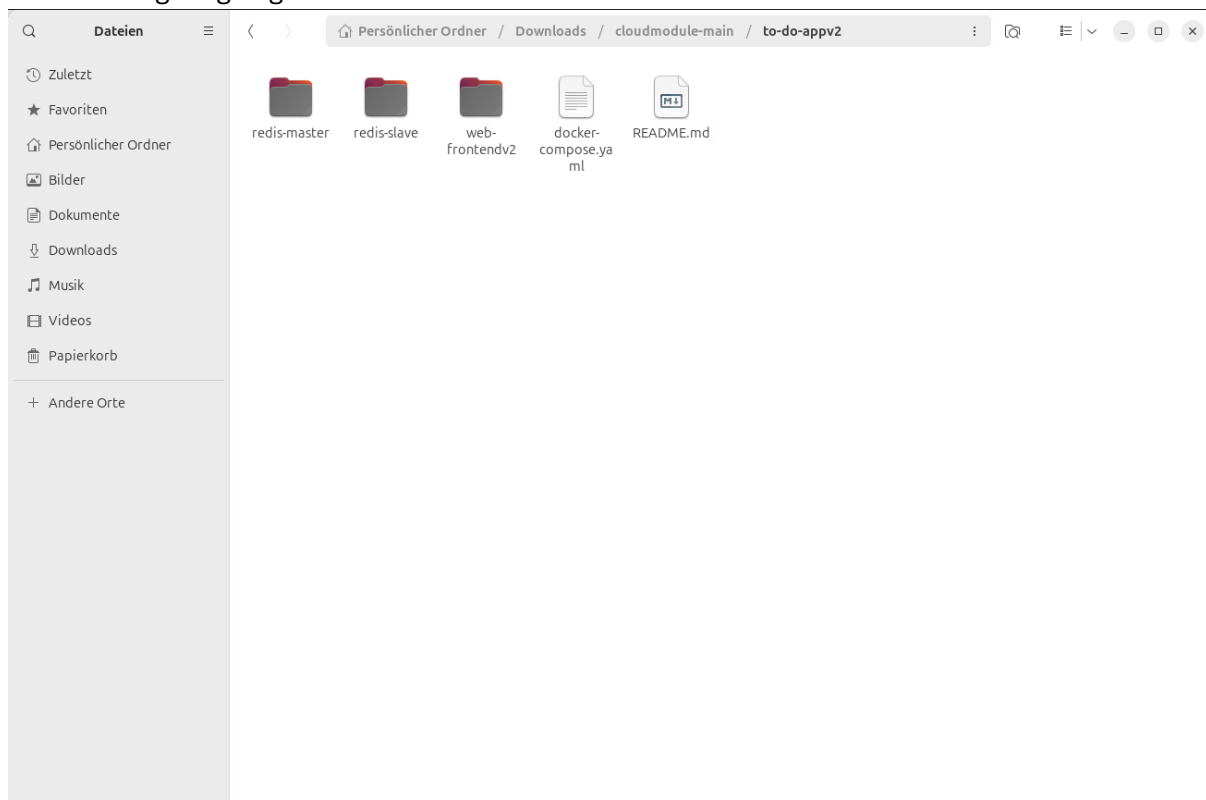
Tag 3

Kurze Beschreibung: Was ist Docker Compose

Docker Compose ist ein Tool, mit dem man mehrere Docker-Container auf einmal starten und verwalten kann. Statt jeden Container einzeln mit langen Befehlen hochzufahren, schreibt man einfach eine docker-compose.yml, in der alles definiert ist – welche Container laufen sollen, welche Ports offen sind und welche Volumes genutzt werden. Mit docker-compose up startet dann das ganze Setup auf einen Schlag. Superpraktisch, wenn man mit mehreren Containern arbeitet und nicht alles manuell machen will.

Version 2 der App mit Docker Compose zum Laufen gebracht (Vorgehen, Befehle und Compose-Datei festgehalten)

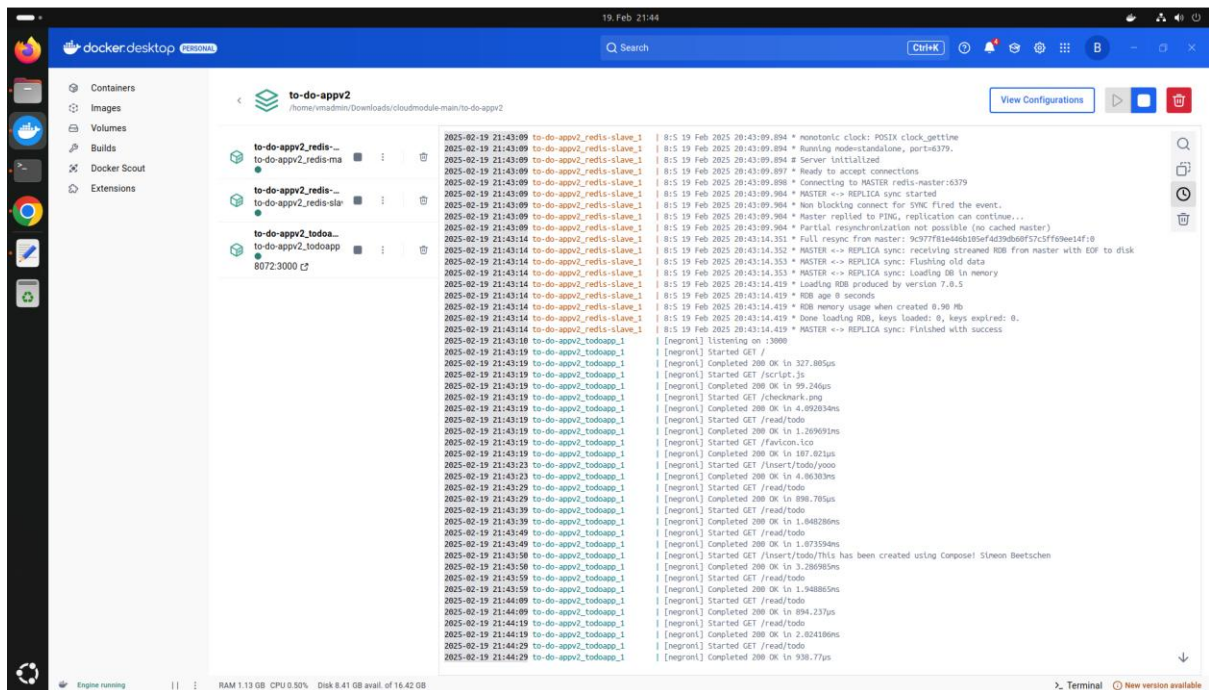
Files wie folgt angelegt:



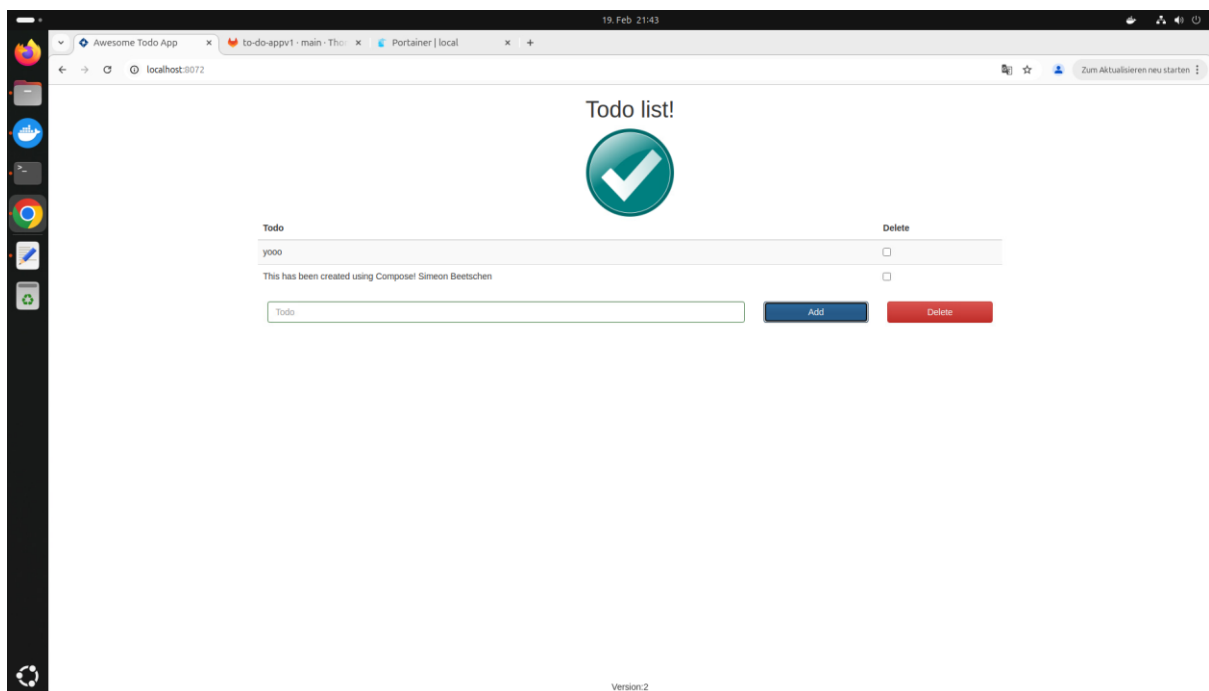
cmd
docker-compose up -d



Intern



The screenshot shows the Docker Desktop interface with the 'to-do-appv2' container selected. The logs on the right show the container's startup sequence, including Redis initialization, Redis replication, and the application's startup. The application logs show a series of GET requests to the /read/todo endpoint, all of which are completed successfully. The status bar at the bottom indicates that the engine is running and shows system resources: RAM 1.13 GB, CPU 0.50%, and Disk 8.41 GB available of 16.42 GB.



The screenshot shows a web browser displaying the 'Awesome Todo App' running on localhost:8072. The page has a title 'Todo list!' and a large green checkmark icon. Below the icon, there is a table with two columns: 'Todo' and 'Delete'. The table contains two entries: 'yooo' and 'This has been created using Compose! Simeon Beetschen'. At the bottom of the page, there is a text input field labeled 'Todo' and two buttons: 'Add' and 'Delete'. The version number 'Version:2' is displayed at the bottom right of the page.

Compose Datei:

```
docker-compose.yaml
```

```
version: "3"
```

```
services:
```

```
  todoapp:
```

```
    build: ./web-frontendv2
```

```
    ports:
```

```
      - "8072:3000"
```

```
    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

```

Portainer installiert und mit PrintScreen festgehalten

Mit «cd» in das Verzeichnis «portainer_docker_compose» wechseln und folgendes File ablegen:

```

docker-compose.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

```

Danach folgenden Befehl ausführen:

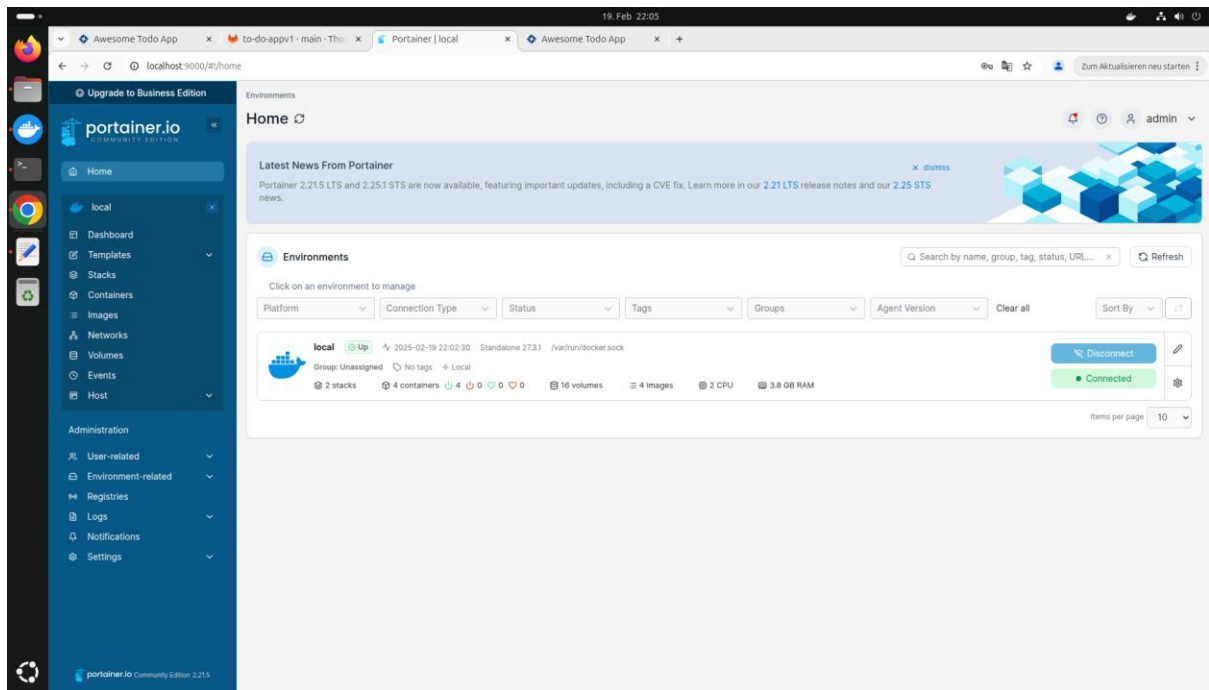
```

console
docker-compose up -d

```

Website auf localhost:9000 aufrufen



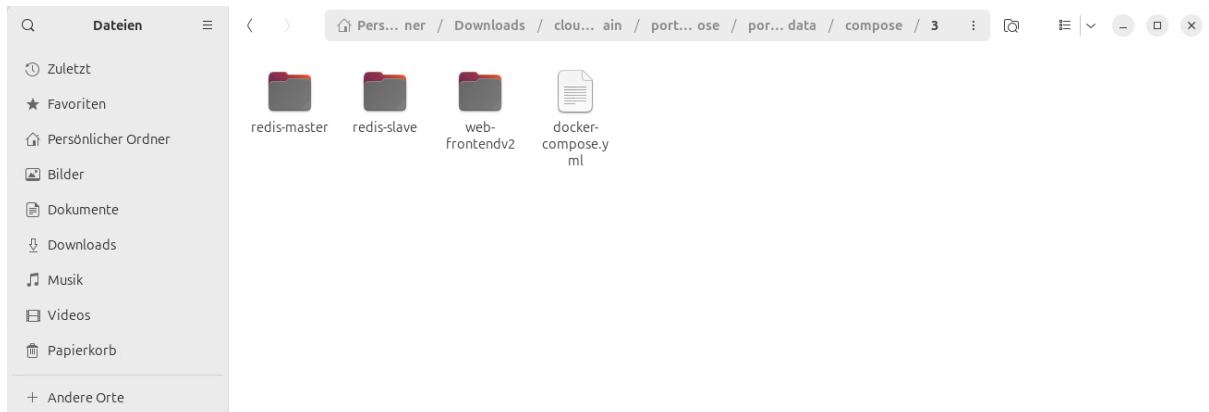


App via Portainer installiert und Vorgehen dokumentiert

Ordner der v2 App an folgendem Ort ablegen:

/home/vmadmin/Downloads/cloudmodule-main/portainer_docker_compose/portainer-data/compose/3

Sollte dann so aussehen:



Home -> local



The screenshot shows the Portainer.io Dashboard interface. The left sidebar contains navigation links for Home, local, Dashboard, Templates, Stacks, Containers, Images, Networks, Volumes, Events, Host, Administration, User-related, Environment-related, Registries, Logs, Notifications, and Settings. The main content area displays the 'Environment summary' and 'Dashboard' with the following resource counts:

- 2 Stacks
- 4 Containers (0 running, 0 healthy, 0 stopped, 0 unhealthy)
- 4 Images (512.7 MB)
- 16 Volumes
- 6 Networks

The 'Environment info' section shows details for the local environment:

- Environment: local (2, 3.8 GB - Standalone 27.3.1)
- URL: /var/run/docker.sock
- GPU: none
- Tags: -

-> Stack

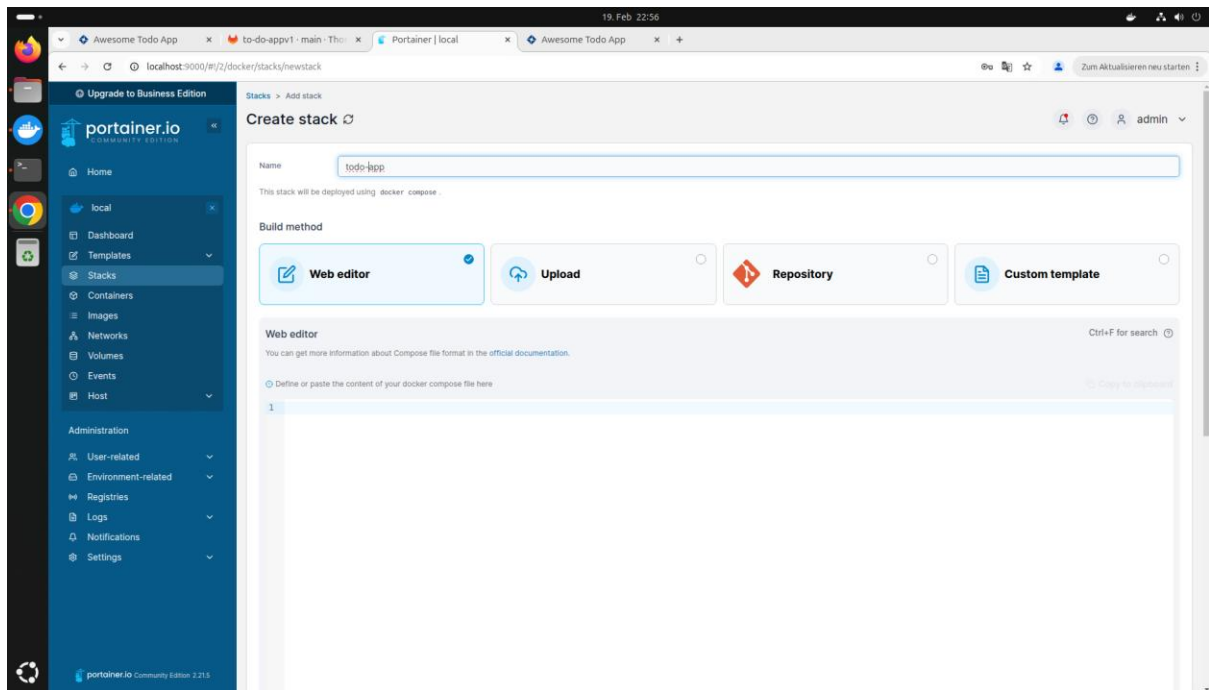
The screenshot shows the 'Stacks list' page in Portainer.io. It displays a table of stacks with the following columns: Name, Type, Control, Created, Updated, and Ownership. The table contains two entries:

Name	Type	Control	Created	Updated	Ownership
bla	Compose	Total	2025-02-19 21:46:47 by admin	-	administrators
portainer_docker_compose	Compose	Limited	2025-02-19 21:57:27	-	administrators

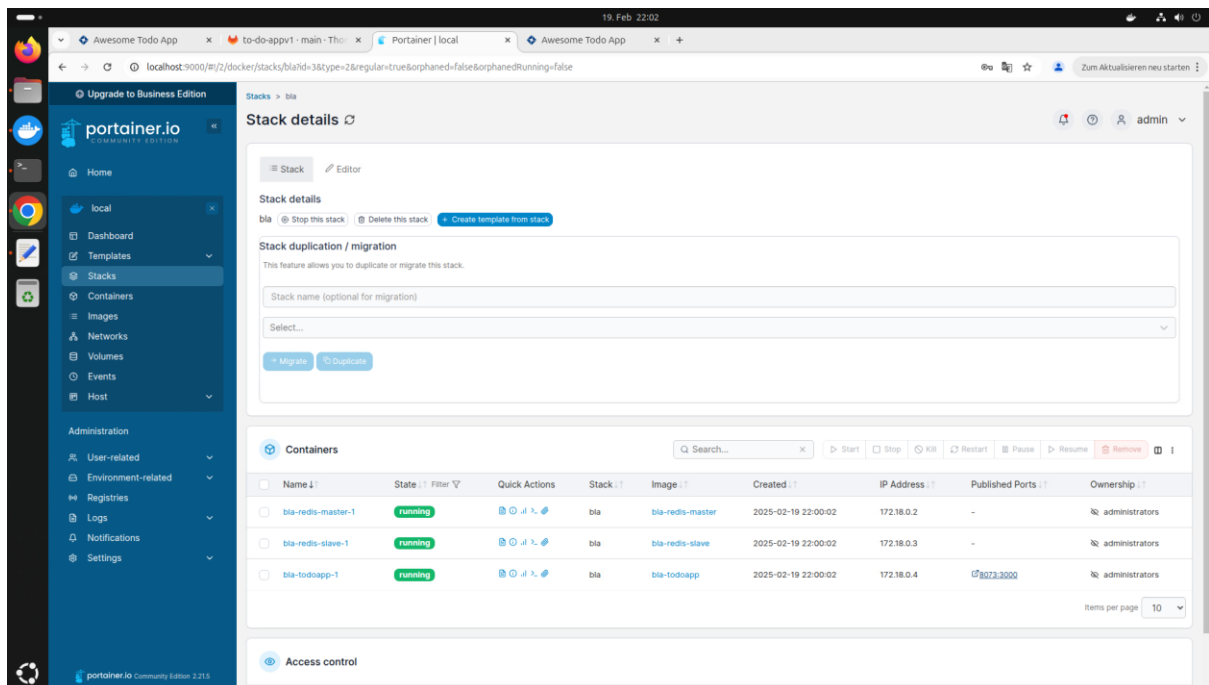
The page also includes a search bar, a 'Remove' button, an 'Add stack' button, and a 'Items per page' dropdown set to 10.

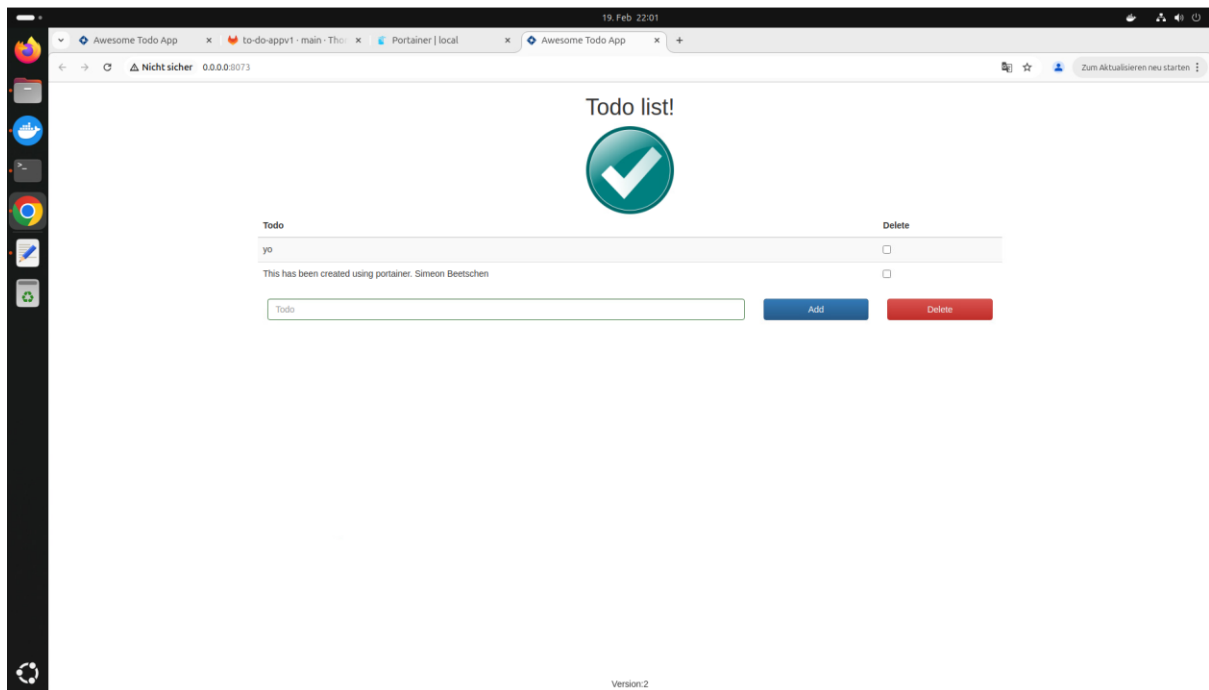
-> Add Stack





Das Docker-compose der vorherigen Aufgabe hier einfügen und danach -> Stack erstellen

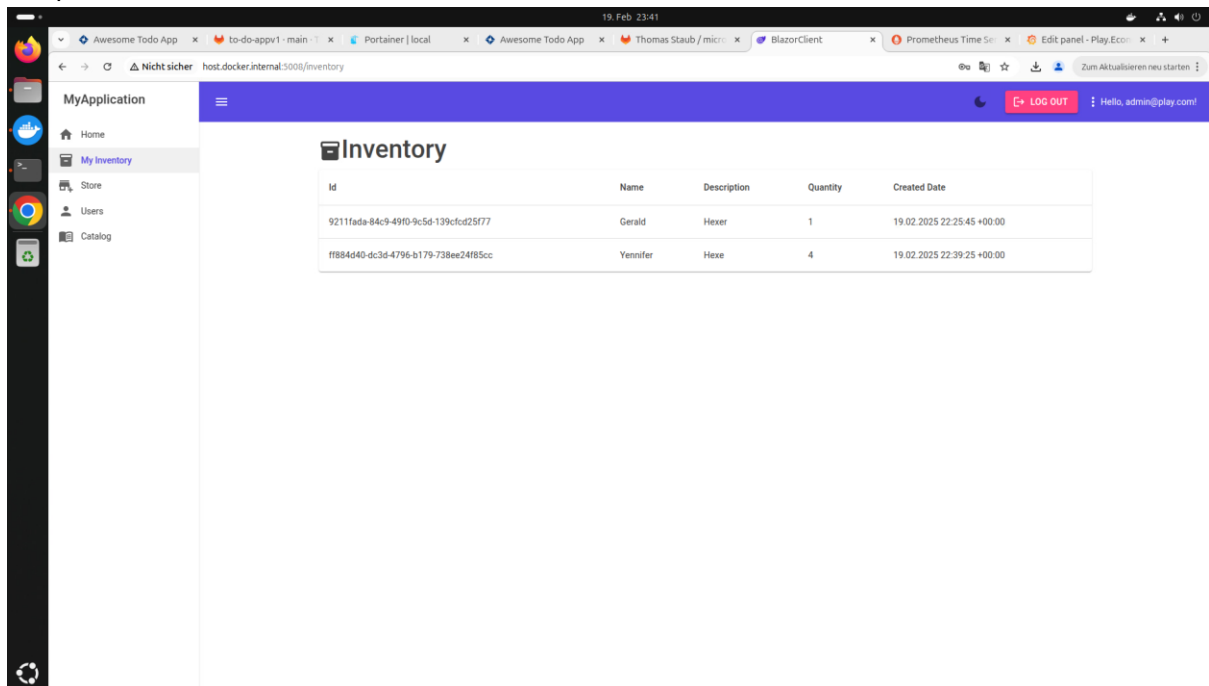




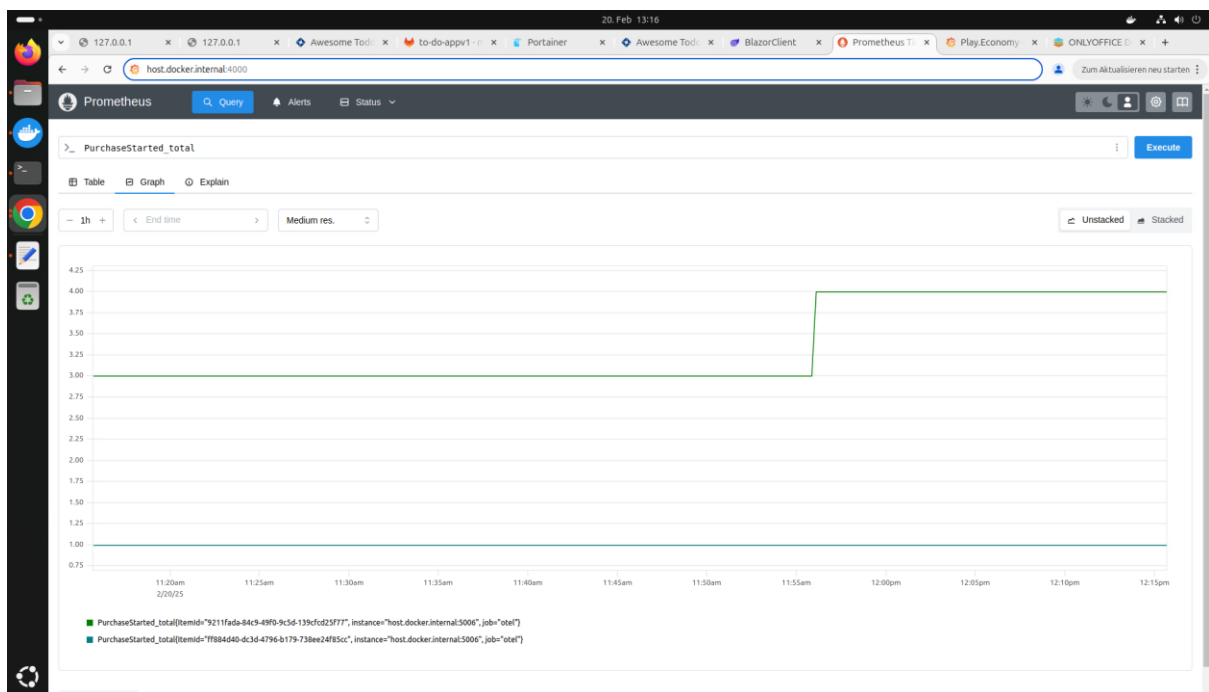
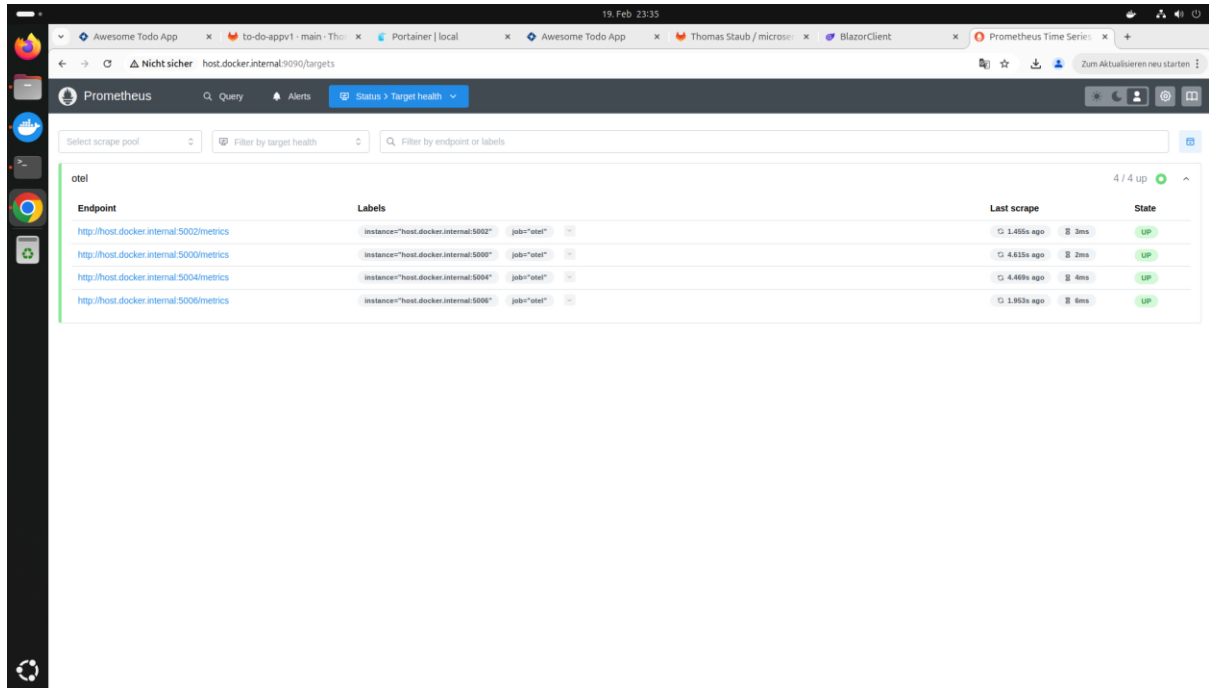
Das Learning-Beispiel «Shop» via Docker Compose installiert

Printscreen vom laufenden Shop, Prometheus und Grafana

Shop:

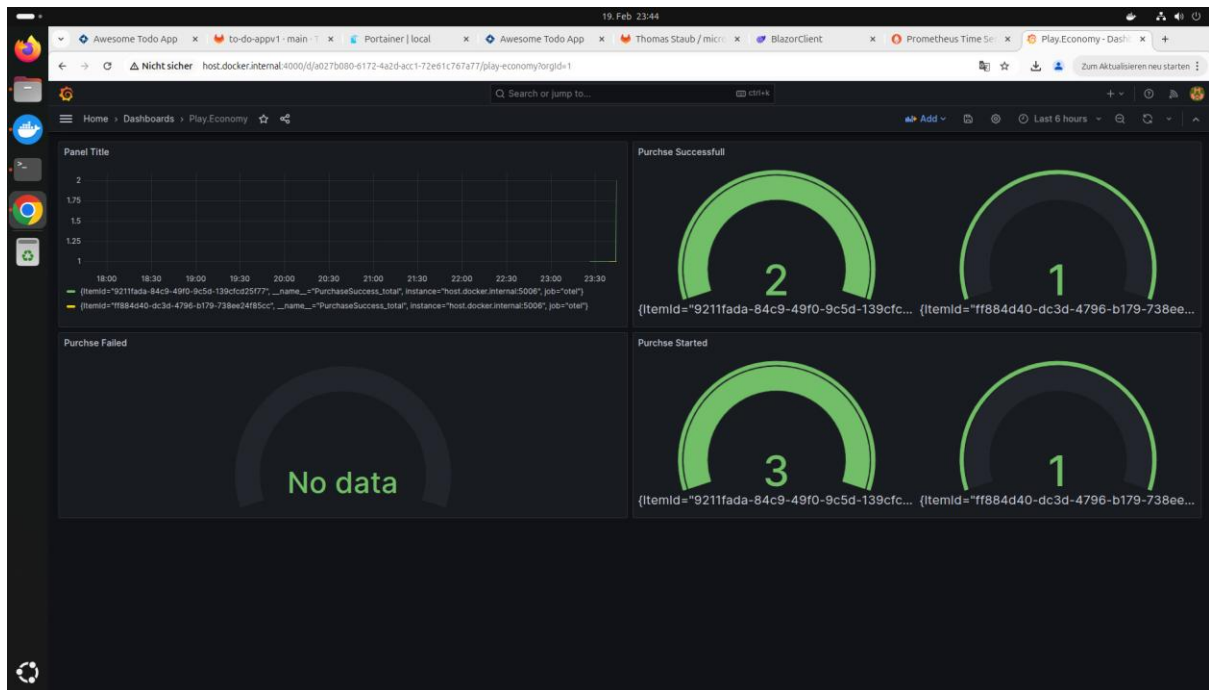


Prometheus:



Grafana:





Vorgehen im Portfolio festgehalten

Hosts-Eintrag hinzufügen: 127.0.0.1 host.docker.internal.

Microservices-Repo (Play.Infra/docker) laden und per “docker-compose up” starten.

Shop testen:

-Browser: <http://host.docker.internal:5008>, einloggen (admin@play.com / Pass@word1).

-Produkte im Katalog erfassen und via „Users“ Gil hinzufügen.

-Produkte kaufen und Inventar prüfen.

Daten persistieren:

-Volumes (z.B. mongodbdata:/data/db) im docker-compose.yaml definiert und kontrolliert.

Umgebungsvariablen nutzen:

-Beispiel mongo-express mit ME_CONFIG_*-Variablen, depends_on für Reihenfolge.

Troubleshooting:

-Seq (Logs): <http://host.docker.internal:8085> -> Probleme mit RabbitMQ erkannt (Startverzögerung).

-RabbitMQ (Message Broker): <http://host.docker.internal:15672> -> Funktionskontrolle beim Einkaufen.

-Jaeger (Traces): <http://host.docker.internal:16686> -> TraceID aus Seq kopieren, Dauer und beteiligte Services einsehen.

-Prometheus (Metrics): <http://host.docker.internal:9090> -> Targets, Graphen und Status (Services up/down).



-Grafana (Dashboards): <http://host.docker.internal:4000> -> Datenquelle Prometheus, Visualisierung z.B. Play-Economy-Dashboard.

Frage aus der Aufgabe:

Was ist RabbitMQ?

RabbitMQ ist eine Open-Source-Software, die als sogenannter Message Broker fungiert. Stell dir das wie eine „Poststelle“ vor, über die Nachrichten ausgetauscht werden: Anwendungen (Producer) können Nachrichten dort ablegen, und andere Anwendungen (Consumer) können diese Nachrichten abholen und verarbeiten. RabbitMQ unterstützt verschiedene Kommunikationsmuster, beispielsweise Warteschlangen (Queue), Publish-Subscribe und mehr. Seine Hauptaufgabe ist also, Nachrichten zuverlässig zu vermitteln und dabei sicherzustellen, dass kein Datenverlust stattfindet und Produzenten sowie Konsumenten entkoppelt bleiben.

Was macht ein Message Broker?

Ein Message Broker übernimmt die Rolle eines Vermittlers zwischen verschiedenen Systemen oder Diensten. Er empfängt Nachrichten von einem Sender und leitet sie an den richtigen Empfänger weiter. Damit löst man mehrere Probleme:

- Entkopplung:** Systeme müssen nicht direkt miteinander verbunden sein und können unabhängig voneinander agieren.
- Zuverlässigkeit:** Der Broker stellt sicher, dass Nachrichten zwischengespeichert werden und nicht verloren gehen.
- Skalierbarkeit:** Dadurch, dass Prozesse asynchron ablaufen können, lässt sich die Anzahl der Produzenten und Konsumenten unabhängig voneinander skalieren.
- Flexibilität:** Verschiedene Protokolle oder Formate können genutzt werden, ohne dass Produzenten und Konsumenten sich gegenseitig kennen müssen.

Warum braucht es einen Message Broker bei Microservices?

In einer Microservices-Architektur gibt es viele kleine, lose gekoppelte Services, die bestimmte Aufgaben erfüllen. Wenn jeder Service direkt mit jedem anderen Service kommunizieren müsste, entstünde schnell ein komplexes Netz an Abhängigkeiten. Ein Message Broker löst dieses Problem, indem er als zentrale Vermittlungsstelle fungiert:

- Lose Kopplung:** Services können unabhängig entwickelt, deployed und aktualisiert werden.
- Asynchrone Kommunikation:** Wenn ein Service eine Nachricht schickt, muss er nicht auf eine sofortige Antwort warten; das steigert die Effizienz und Robustheit des Gesamtsystems.
- Fehlertoleranz:** Fällt ein Service kurz aus, kann die Nachricht zwischengespeichert werden und geht nicht verloren. Sobald der Service wieder verfügbar ist, werden die aufgelaufenen Nachrichten verarbeitet.



Warum ist ein Message Broker bei der Skalierung von Microservices wichtig?

Bei steigender Last oder mehr Nutzern muss ein System mitwachsen können. Mit einem Message Broker lässt sich das besonders effektiv umsetzen:

-Lastverteilung: Kommen viele Nachrichten gleichzeitig rein, können weitere Konsumenten (Microservices-Instanzen) hinzugefügt werden, ohne den Rest des Systems zu beeinflussen.

-Asynchrone Verarbeitung: Lange oder aufwendige Prozesse blockieren nicht andere Services, da sie im Hintergrund weiterlaufen können, während neue Anfragen bereits angenommen werden.

-Erhöhte Zuverlässigkeit: Selbst wenn einzelne Services ausfallen oder neu gestartet werden müssen, bleiben die Nachrichten im Broker erhalten und werden später zugestellt.

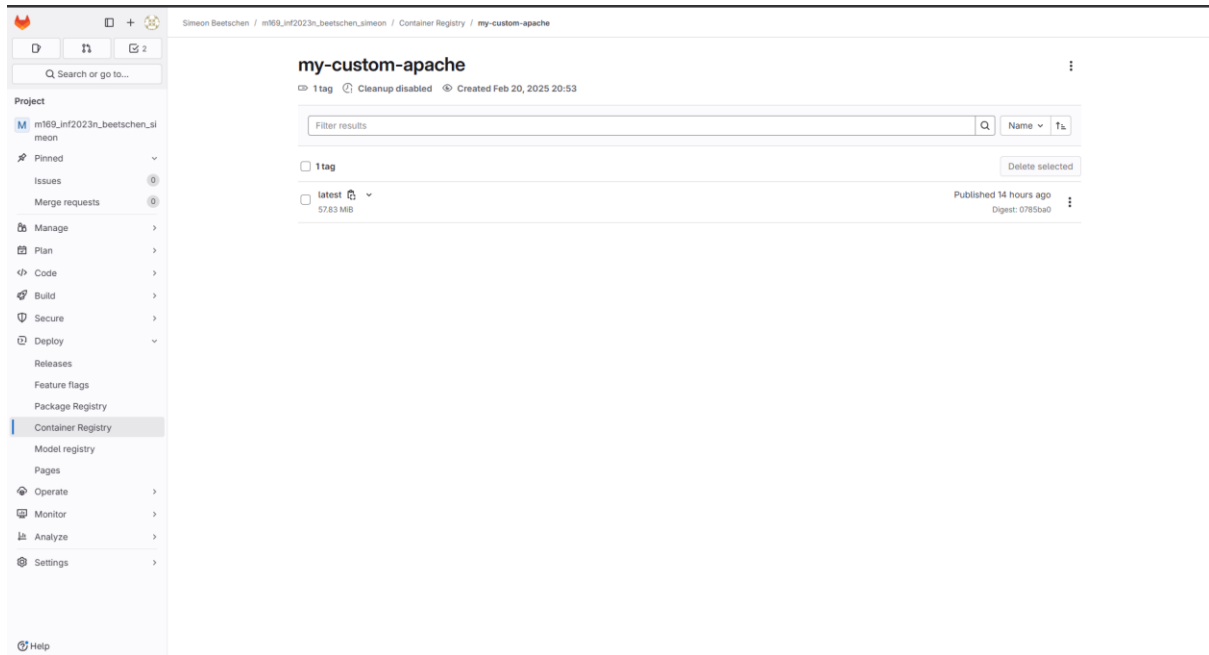
-Zentrale Steuerung: Man erhält einen guten Überblick, wie viele Nachrichten gerade im System zirkulieren, und kann darauf basierend skalieren.



Tag 4

Sie erstellen ein eigenes Projekt und erzeugen ein Image davon.

Das Image pushen Sie in das Repository (Printscreen)



Anleitung für die Installation

Sicherstellen, dass Git und Docker installiert ist. Oder das Repo als Zip herunterladen und entzippen. Danach Konsole öffnen und folgende Befehle eingeben:

- ➔ git clone https://git.gibb.ch/sbe150562/m169_inf2023n_beetschen_simeon.git
- ➔ cd m169_inf2023n_beetschen_simeon
- ➔ cd uebungsprojekt
- ➔ docker-compose up -d

Webseite sollte nun auf (localhost:80 oder 127.0.0.1:80 oder (IP-Adresse des Gerätes):80) verfügbar sein.



Tag 5

Was ist Kubernetes?

Kubernetes ist ein Open-Source-Tool zur Verwaltung von Containern. Es automatisiert Bereitstellung, Skalierung und Betrieb von Anwendungen, verteilt die Last, stellt Ausfallsicherheit sicher und erleichtert die Verwaltung grosser Systeme. Ideal für DevOps und Cloud-native Anwendungen.

Was sind Microservices?

Die Microservices-Architektur ist ein Ansatz, bei dem eine Anwendung aus mehreren unabhängigen Diensten besteht. Jeder Service übernimmt eine klar definierte Aufgabe und kommuniziert über APIs mit anderen. Dadurch sind Anwendungen flexibler, skalierbarer und leichter zu warten als monolithische Systeme.

Vorteile:

- **Skalierbarkeit:** Jeder Service kann unabhängig skaliert werden.
- **Flexibilität:** Verschiedene Technologien können kombiniert werden.
- **Schnellere Entwicklung:** Teams können parallel an einzelnen Services arbeiten.
- **Stabilität:** Ein Fehler in einem Service legt nicht das gesamte System lahm.

Typische Einsatzgebiete sind z. B. Website-Migrationen, Medien-Streaming, Zahlungsabwicklung und Datenverarbeitung. Microservices laufen oft in **Containern**, wobei Kubernetes für die Verwaltung genutzt wird.

Vergleich von lightweight Kubernetes-Anwendungen

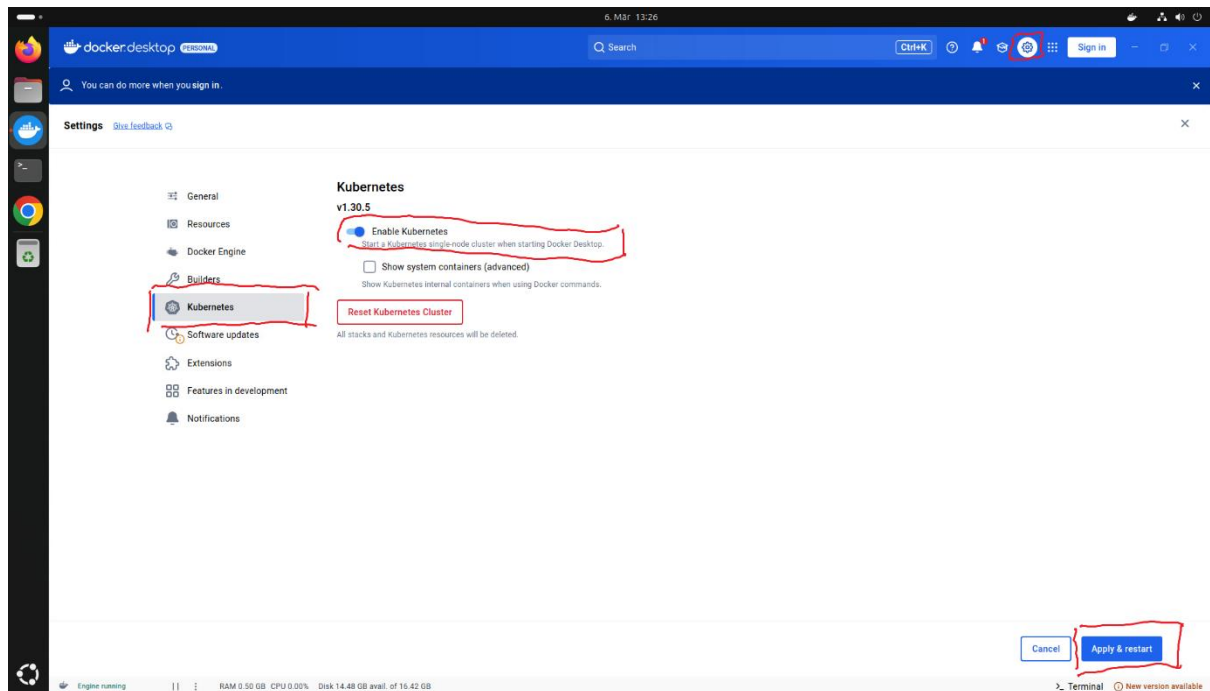
- **minikube** – Einfaches Tool für lokale Tests, schnell eingerichtet, aber nur Single Node.
- **microk8s** – Vollständiges Kubernetes für Entwicklung und Edge, minimaler Ressourcenverbrauch, Multi-Node möglich.
- **kubeadm** – Für produktionsnahe Setups, erfordert manuelle Einrichtung, skaliert bis zu HA-Clustern.
- **Docker for Windows** – Integriert in Docker Desktop, praktisch für lokale Tests auf Windows, aber eingeschränkt.
- **kind** – Kubernetes in Docker-Containern, leichtgewichtig, ideal für CI/CD-Tests.
- **k3s** – Extrem schlank, optimiert für Edge und IoT, läuft auch mit schwacher Hardware.

Für schnelles Testen eignen sich minikube oder Docker for Windows. CI/CD-Workflows profitieren von kind. Wer Kubernetes effizient auf kleiner Hardware nutzen will, greift zu k3s oder microk8s. kubeadm ist für produktionsnahe Tests geeignet.

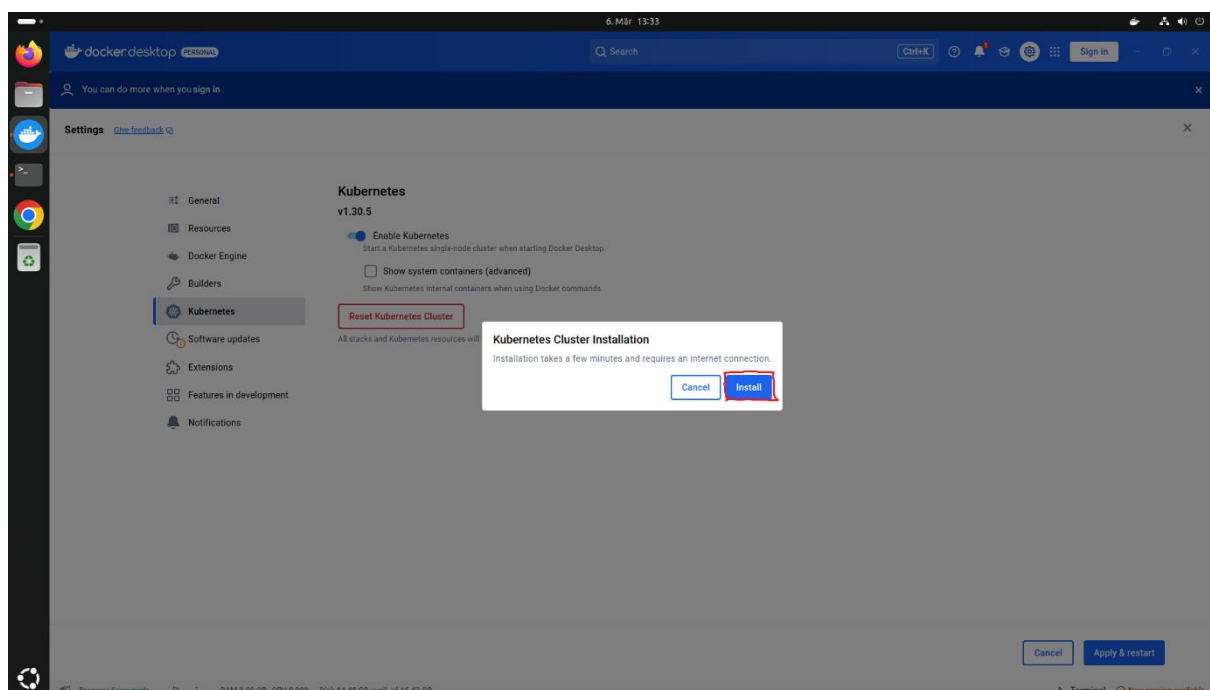


Kubernetes installiert und Anleitung erstellt

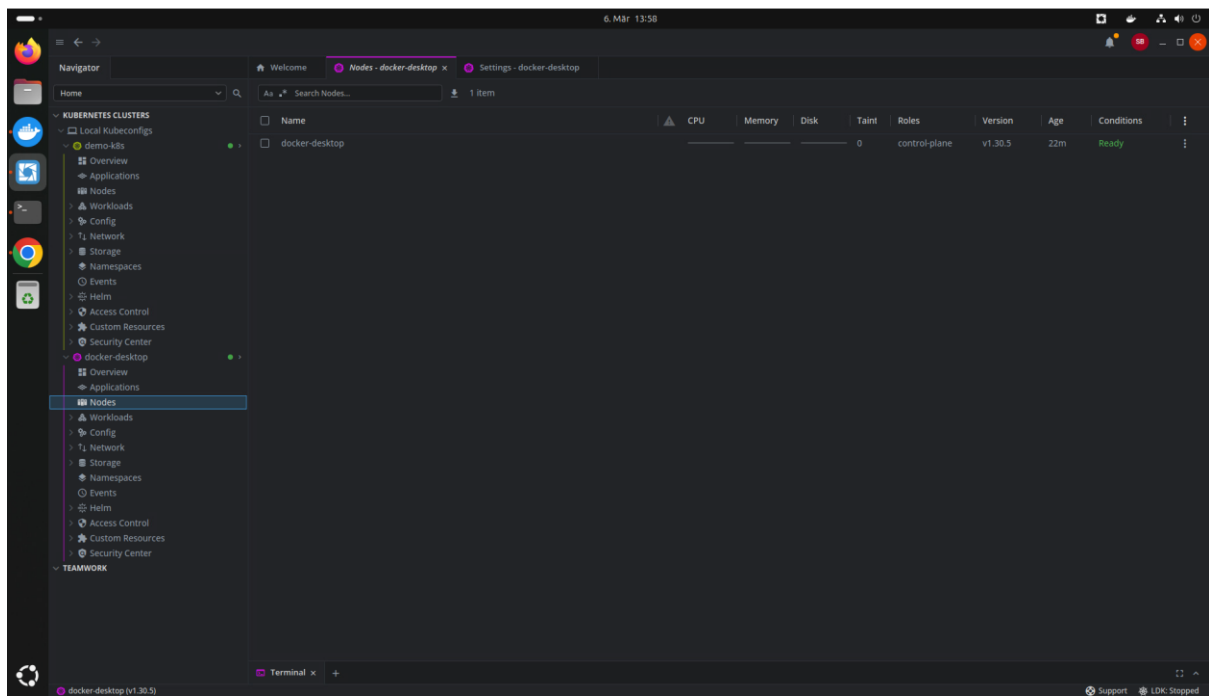
Öffne Docker -> Einstellungen -> Kubernetes -> «Enable Kubernetes» -> «Apply & restart»



-> «Install»



Verbindung zu Kubernetes-Server (Printscreen) mit Lens



Tag 6

Erläuterung: Was ist der Raft-konsens-Algorithmus und warum eine ungerade Anzahl an Servern in einem Cluster?

Was ist Raft?

Der **Raft-Konsens-Algorithmus** sorgt dafür, dass mehrere Server in einem verteilten System sich auf einen gemeinsamen Zustand einigen. Er wird z. B. in etcd (das Herzstück von Kubernetes) genutzt, um Daten konsistent zu halten.

Wie funktioniert Raft?

- **Leader-Wahl:** Einer der Server wird zum Anführer (Leader), die anderen folgen (Follower).
- **Daten-Synchronisation:** Der Leader verteilt alle Änderungen an die Follower. Erst wenn die Mehrheit zustimmt, gilt die Änderung als bestätigt.
- **Fehlertoleranz:** Fällt der Leader aus, wählen die restlichen Server automatisch einen neuen.

Warum eine ungerade Anzahl an Servern?

Damit Raft funktioniert, braucht es eine **Mehrheit (Quorum)**. Nur wenn die Mehrheit der Server erreichbar ist, kann das System Entscheidungen treffen.

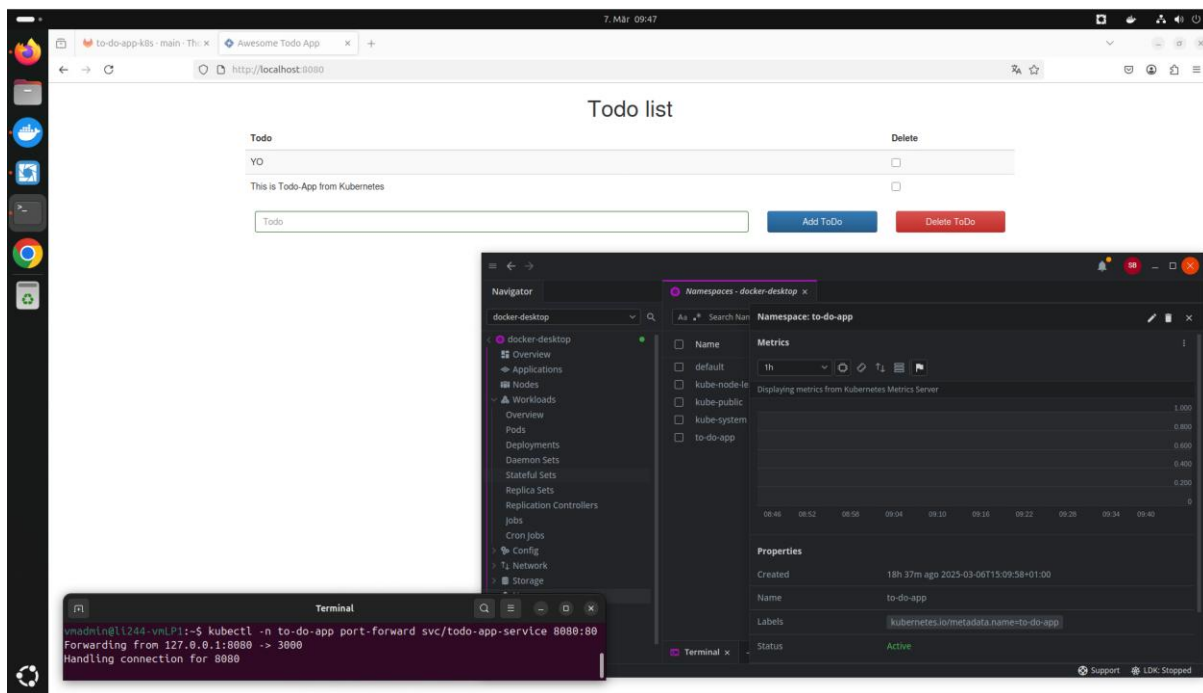
Beispiel:

- **3 Server** -> Quorum = 2 -> 1 Server kann ausfallen, ohne dass das System stehen bleibt.
- **4 Server** -> Quorum = 3 -> Bringt nichts, weil nur 1 Server ausfallen darf, genau wie bei 3.
- **5 Server** -> Quorum = 3 -> Jetzt können sogar 2 Server ausfallen.

Eine gerade Anzahl an Servern bringt also keinen Vorteil, frisst aber unnötig Ressourcen. Deshalb setzt man immer auf eine ungerade Anzahl.



Dokumentation: Wie läuft die App in Kubernetes?



Eintrag zu: Self-Healing, Scale Down, Scale Up

Self-Healing: Kubernetes erkennt ausgefallene Pods und ersetzt sie automatisch. Der Deployment Controller überprüft kontinuierlich den Cluster-Zustand und startet fehlende Pods neu, um die gewünschte Anzahl an replicas aufrechtzuerhalten.

Scale Down: Reduziert die Anzahl der Pods, wenn weniger Last anliegt, um Ressourcen zu sparen.

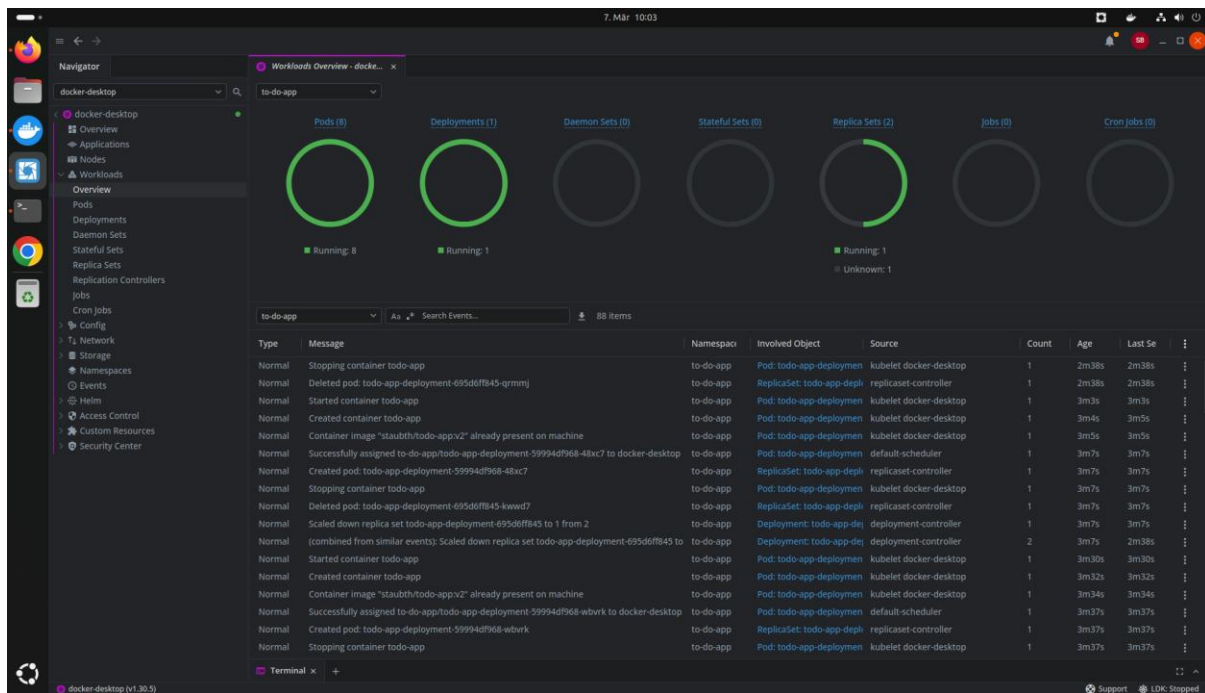
-> `kubectl scale --replicas=1 deployment/app-name`

Scale Up: Erhöht die Anzahl der Pods, wenn die Last steigt, um Performance-Einbussen zu vermeiden.

-> `kubectl scale --replicas=5 deployment/app-name`



Rolling Update funktioniert (V2 in Dashboard)(Screenshot)



Eintrag zu Blue-Green Deployment

Green-Blue Deployment ist eine **Deployment-Strategie**, die **Zero Downtime** ermöglicht, indem **zwei Umgebungen (Green & Blue)** parallel betrieben werden.

Ablauf:

1. **"Blue"** ist die aktuelle **Produktionsumgebung** (Live).
2. Eine neue Version der Anwendung wird in **"Green"** (Staging) bereitgestellt.
3. Tests werden auf **Green** durchgeführt.
4. Der **Traffic wird umgeschaltet** von **Blue → Green**.
5. **Blue bleibt erhalten** als Fallback, falls Fehler auftreten.
6. Falls die neue Version fehlerhaft ist, kann **sofort auf Blue zurückgeschaltet** werden.



Tag 7

Eintrag zu Cluster IP und Node IP

Cluster IP -> Interne IP für die Kommunikation zwischen Kubernetes-Services, nicht von aussen erreichbar.

Node IP -> IP-Adresse eines Kubernetes-Nodes, kann für externen Zugriff auf NodePort-Services genutzt werden.

Eintrag zu LoadBalancer

Ein LoadBalancer in Kubernetes macht deine Anwendung nach aussen hin verfügbar und verteilt alle eingehenden Anfragen automatisch auf mehrere Pods. Das bedeutet konkret:

-> Externe IP: Du bekommst eine IP-Adresse, über die Nutzer von ausserhalb auf deinen Service zugreifen können.

-> Lastverteilung: Wenn mehrere Pods laufen, schickt der LoadBalancer den Traffic immer an die, die gerade Ressourcen frei haben, damit keiner überlastet wird.

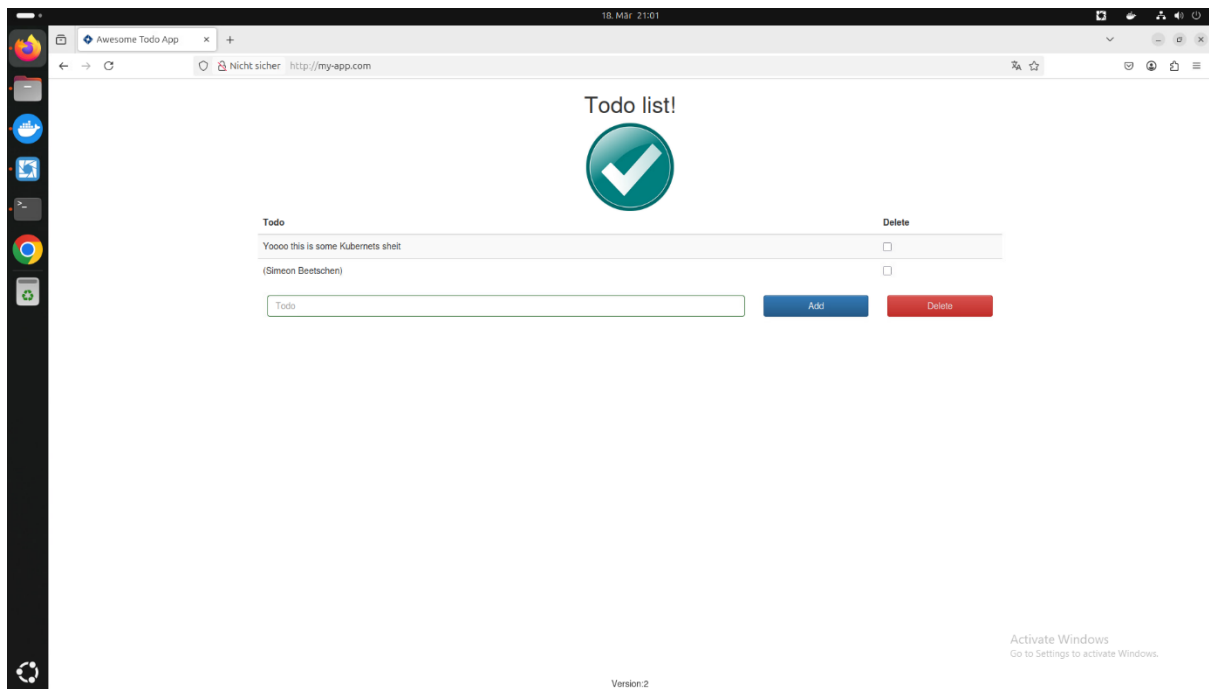
-> Hochverfügbarkeit: Fällt ein Pod aus, übernimmt ein anderer – für den Endnutzer bleibt alles reibungslos.

In der Cloud (AWS, GCP, Azure) regelt Kubernetes das vollautomatisch: Du sagst einfach type: LoadBalancer, und schon richtet der Cloud-Provider alles für dich ein. Wenn du On-Premise arbeitest, nutze MetalLB oder andere Methoden wie NodePort oder Ingress, um ähnliche Funktionen umzusetzen.

Tldr: Mit einem LoadBalancer-Service erreichst du eine stabile, skalierbare und von aussen zugängliche Anwendung – ohne grossen Konfigurationsaufwand.



PrintScreen Wie Sie auf die App zugreifen. Siehe Kap. Ingress



Erklärung warum sie bei "Ingress beim Zugriff auf 127.0.0.1 ein Error 404 erhalten

Ingress ist so eingestellt, dass er nur Anfragen für my-app.com weiterleitet. Wird stattdessen 127.0.0.1 im Browser aufgerufen, gibt es keine passende Regel – Kubernetes weiss nicht, wohin mit der Anfrage, also kommt ein 404-Fehler.

Die /etc/hosts-Datei sorgt dafür, dass my-app.com auf 127.0.0.1 zeigt. Wird dann my-app.com aufgerufen, passt der Hostname zur Ingress-Regel und alles funktioniert.

Soll 127.0.0.1 direkt laufen, muss die Ingress-Konfiguration angepasst werden, entweder indem der Hostname entfernt oder 127.0.0.1 explizit als Host erlaubt wird.



Installation von Portainer auf Kubernetes

The screenshot shows the Docker Desktop interface with the 'Namespaces - docker-desktop' tab selected. The left sidebar shows the 'Namespaces' section expanded. The main panel displays a table of namespaces with columns for Name, Labels, Age, and Status.

Name	Labels	Age	Status
default	kubernetes.io/metadata.name=default	12d	Active
ingress-nginx	app.kubernetes.io/instance=ingress-nginx, app.kubernetes.io/name=ingress-nginx	54m	Active
kube-node-lease	kubernetes.io/metadata.name=kube-node-lease	12d	Active
kube-public	kubernetes.io/metadata.name=kube-public	12d	Active
kube-system	kubernetes.io/metadata.name=kube-system	12d	Active
portainer	kubernetes.io/metadata.name=portainer	117s	Active
to-do-app	kubernetes.io/metadata.name=to-do-app	12d	Active

Below the table, a terminal window shows the command `kubectl get namespaces` being executed, resulting in the same list of namespaces.

Overlaid on the right is a web browser window titled 'Awesome Todo App' showing a 'Todo list!' with a green checkmark icon. The list contains two items: '(Simeon Beetschen)' and 'new Kubernetes Thing', each with a 'Delete' button. At the bottom, there is an 'Add' button and a 'Delete' button.



Tag 8

Eigenes Projekt auf Kubernetes installiert und dokumentiert

Eigenes Kubernetes-Projekt: Terraria Server

Was ist das Projekt?

Ich habe einen eigenen Terraria-Server in einem Kubernetes-Cluster deployed. Ziel war es, den Server als Container bereitzustellen, persistenten Speicher für die Welt zu nutzen und alles über YAML zu konfigurieren – also komplett ohne manuelles Setup.

Setup: Was steckt dahinter?

- Ich verwende ein lokales Kubernetes-Cluster (Docker Desktop)
- Das Terraria-Image kommt ursprünglich von hexlo/terraria-server-docker
- Ich habe es gepullt, neu getaggt und in meine private GitLab-Registry gepusht
- Die Weltdateien liegen in einem Persistent Volume, damit sie beim Neustart nicht verloren gehen
- Externer Zugriff läuft über einen NodePort-Service

Container-Image in das eigene Registry pushen

CLI
docker pull hexlo/terraria-server-docker:latest
docker tag hexlo/terraria-server-docker:latest git-registry.gibb.ch/sbe150562/m169_inf2023n_beetschen_simeon/terraria-server:v1
docker push git-registry.gibb.ch/sbe150562/m169_inf2023n_beetschen_simeon/terraria-server:v1

Speicher für die Welt

Die Welt wird in einem PVC gespeichert – so bleiben die Daten erhalten, auch wenn der Pod neu gestartet wird.

Deployment des Servers

Das Deployment benutzt mein eigenes Image aus der Registry. Anfangs hatte ich ein Problem, weil ich versucht habe, eine Welt zu laden, die noch gar nicht existierte. Lösung: einfach autocreate nutzen – dann erstellt Terraria die Welt automatisch beim ersten Start.

Service: Zugriff von aussen

Damit man von aussen auf den Server kommt (z. B. für den Terraria-Client), habe ich einen NodePort-Service erstellt:



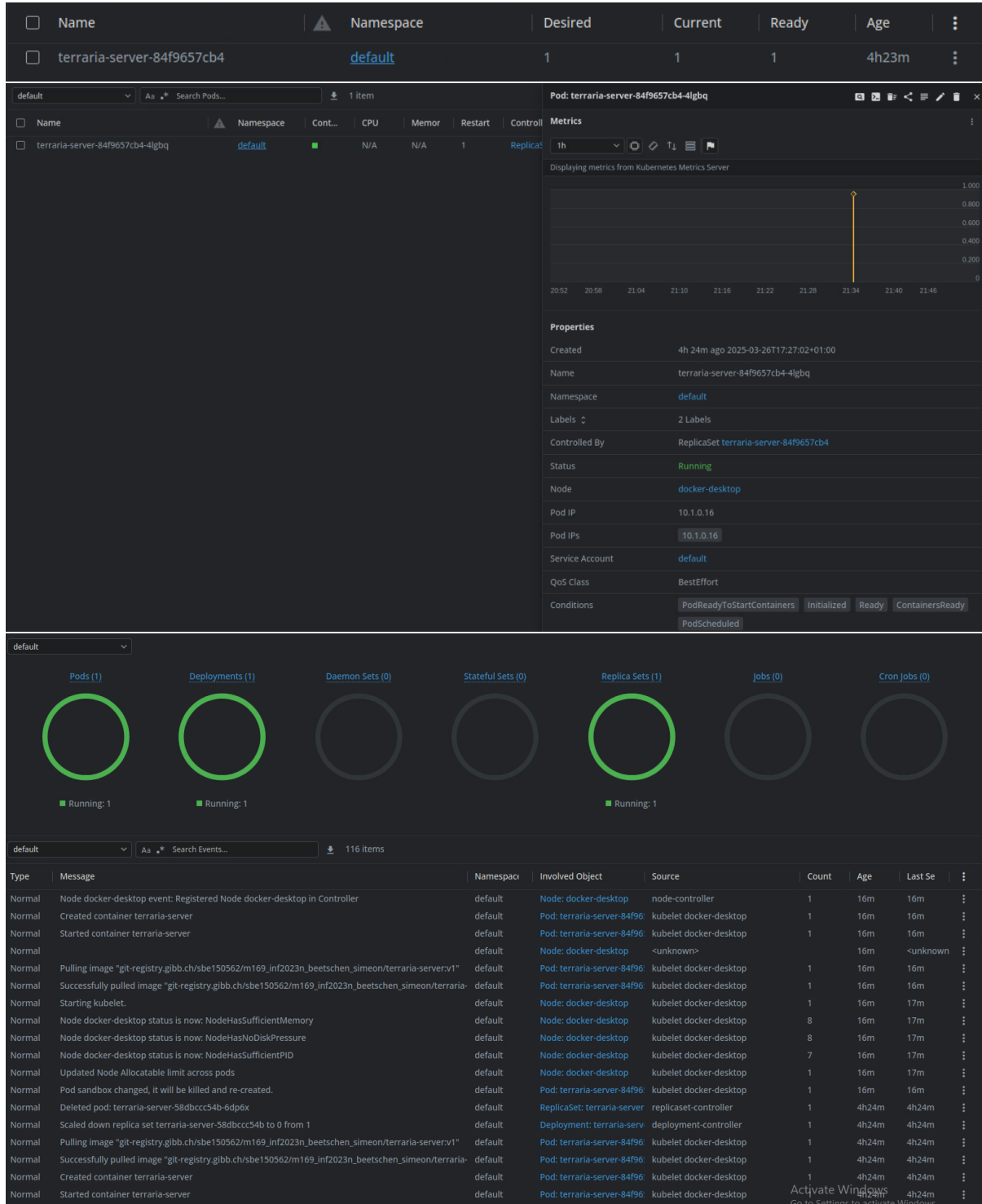
Und alles anwenden mit:

```
bash
```

```
kubectl apply -f terraria-all.yaml
```

Läuft das Ding?

Leider lässt sich das nur schwer in der Umgebung der GIBB testen. Aber ich habe hier ein paar Screenshots:



Fazit

- Der Server läuft im Kubernetes-Cluster
- Die Welt bleibt bei Neustarts erhalten
- Zugriff von aussen ist möglich
- Alles ist mit YAML-Dateien definiert
- Image liegt in der privaten Registry

Dateien im Projekt

- terraria-deployment.yaml
- terraria-pvc.yaml
- terraria-service.yaml
- Container-Image:
`git-registry.gibb.ch/sbe150562/m169_inf2023n_beetschen_simeon/terraria-server:v1`



Tag 9

Todo-App auf Podman zum Laufen gebracht und dokumentiert

Todo-App auf Podman:

Podman installieren:

bash
sudo apt install podman podman-compose

Nach der Installation ist Podman direkt einsatzbereit. Es wird kein Daemon benötigt, und Podman läuft standardmäßig im Rootless-Modus.

App mit Podman starten

Es wird eine docker-compose.yaml erstellt, welche todo-app, redis-master und redis-slave definiert.

(Diese sind im Repo hinterlegt.)

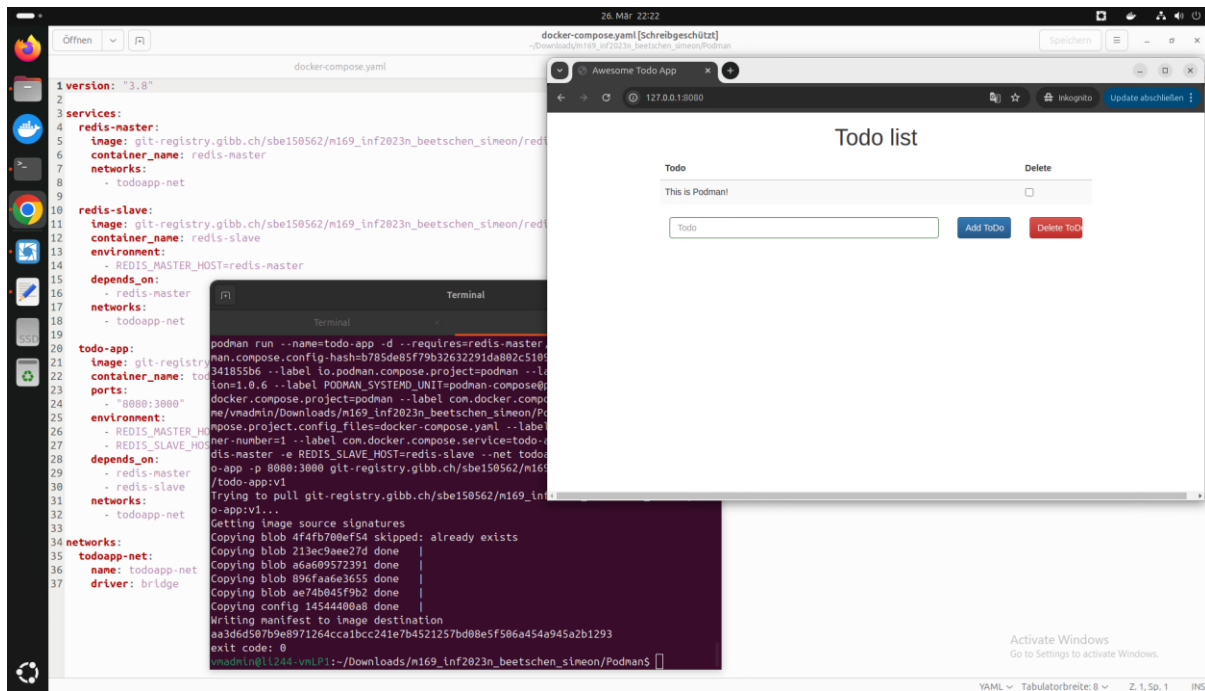
Die Container werden mit folgendem Befehl gestartet:

bash
podman-compose up -d

Damit der Zugriff auf das Web-Frontend im Rootless-Modus funktioniert, wird der Port auf **8080** gesetzt. Die Anwendung ist anschliessend unter <http://localhost:8080> erreichbar. Die Container laufen im selben Netzwerk, die Verbindung zu Redis erfolgt über Umgebungsvariablen. So wie beim Original auch.



Screenshot:



Todo-App auf Podman, auf Kubernetes:

Mit “podman generate kube” werden aus den laufenden Containern YAML-Dateien erstellt. Diese werden überarbeitet, in Deployment- und Service-Ressourcen umgewandelt und mit kubectl apply im Kubernetes-Cluster bereitgestellt. (Diese sind im Repo hinterlegt.)

```
bash
```

```
kubectl apply -f todo-kube.yaml
```

Die App-Komponenten kommunizieren über DNS (z. B. redis-master). Der Zugriff auf das Web-Frontend erfolgt über einen NodePort auf Port **30500**.



Screenshot:

The screenshot displays a Kubernetes dashboard interface on the left and a web browser window on the right.

Kubernetes Dashboard:

- Navigator:** Shows a sidebar with various Kubernetes resources like Local Kubernetes, Overview, Nodes, Workloads, Pods, Deployments, etc.
- Pods - docker-desktop:** A table listing pods in the default namespace.

Name	Namespace	Cont...	CPU	Memor	Restart	Controlled B	Node	QoS	Age	Status
redis-master-695bc769b4-laxh7	default		N/A	N/A	0	ReplicaSet	docker-desktop	BestEffort	89s	Running
redis-slave-77bd44b86-5vbrc	default		N/A	N/A	0	ReplicaSet	docker-desktop	BestEffort	89s	Running
terraria-server-84f9657cb4-4lgbq	default		N/A	N/A	1	ReplicaSet	docker-desktop	BestEffort	5h22m	Running
todo-app-6cc08779fc-rjnmj	default		N/A	N/A	0	ReplicaSet	docker-desktop	BestEffort	89s	Running

Awesome Todo App:

The browser window shows a web application titled "Todo list" at the address 127.0.0.1:30500. It features a form to add and delete todos.

Todo list

Todo	Delete
Yooo this is PODMAN KUBERNETES	<input type="checkbox"/>
By Simeon Beechen	<input type="checkbox"/>

Below the table is a form with a text input labeled "Todo" and two buttons: "Add To Do" (blue) and "Delete To Do" (red).

