Docker & Kubernetes
Teil 2

Teil 2 – Installation und Nutzung

- Installation
  - Docker Desktop
  - Docker Engine
- Bedienung/Management/Kommandos

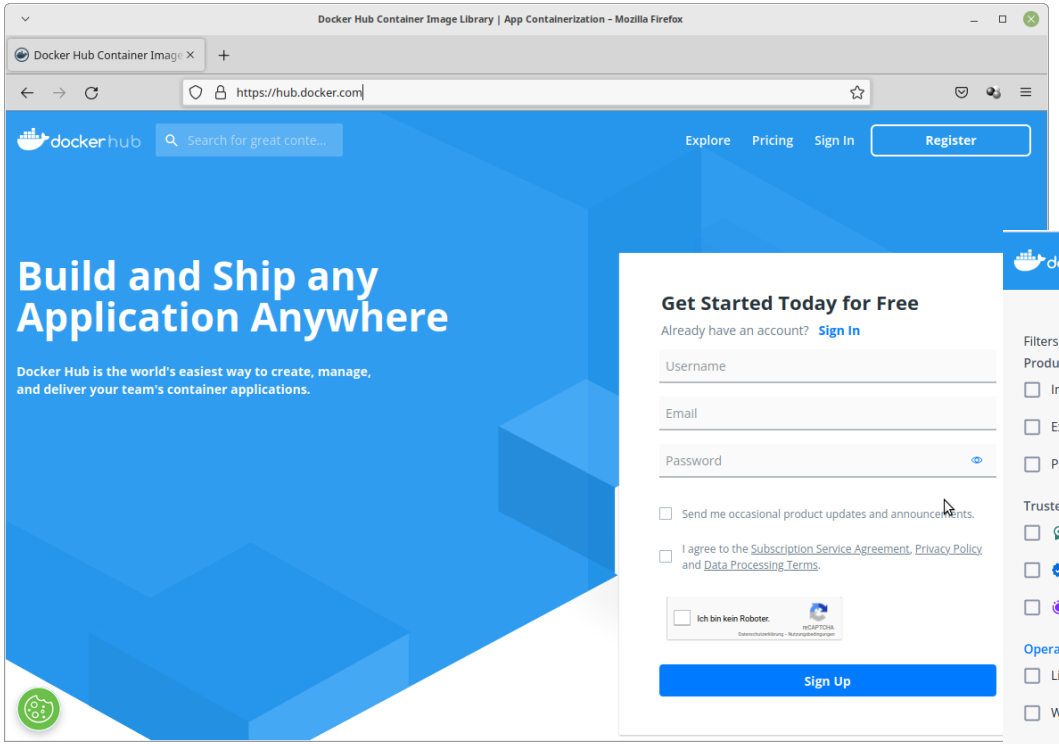
Docker Desktop - Installation

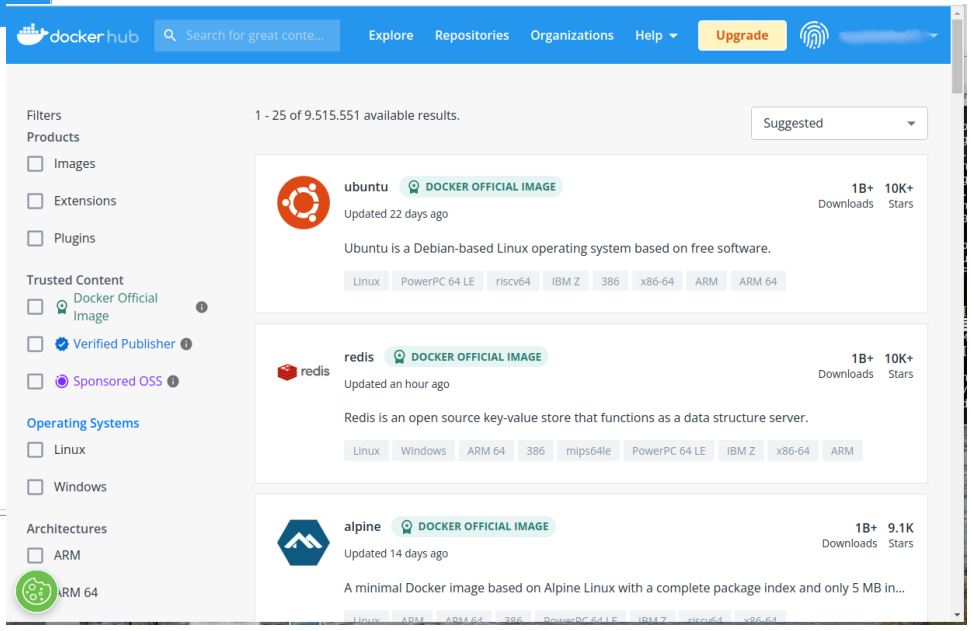
- Windows
  - *WSL 2* installieren … „wsl --install“ in PowerShell
  - *Docker Desktop* laden und installieren
  - https://docs.docker.com/desktop/install/windows-install/
  - Anmeldung bei *Docker Hub*
  - Konfiguration von Docker Desktop

Docker Desktop - Installation

- Linux
  - *Docker Desktop* laden und installieren
  - https://docs.docker.com/desktop/install/linux-install/
  - Anmeldung bei *Docker Hub*
  - Konfiguration von Docker Desktop

Docker Desktop – Docker Hub

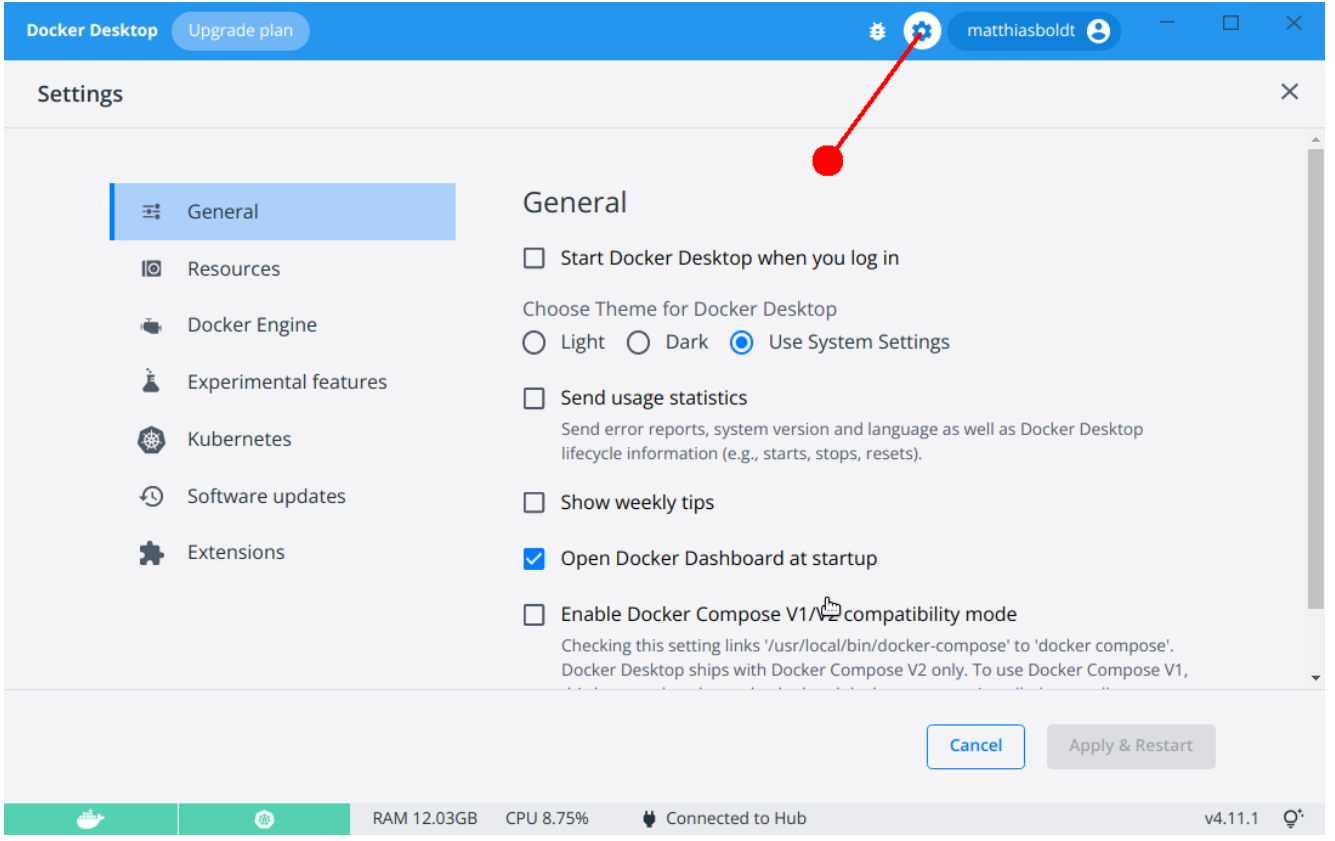




- https://hub.docker.com
- optional

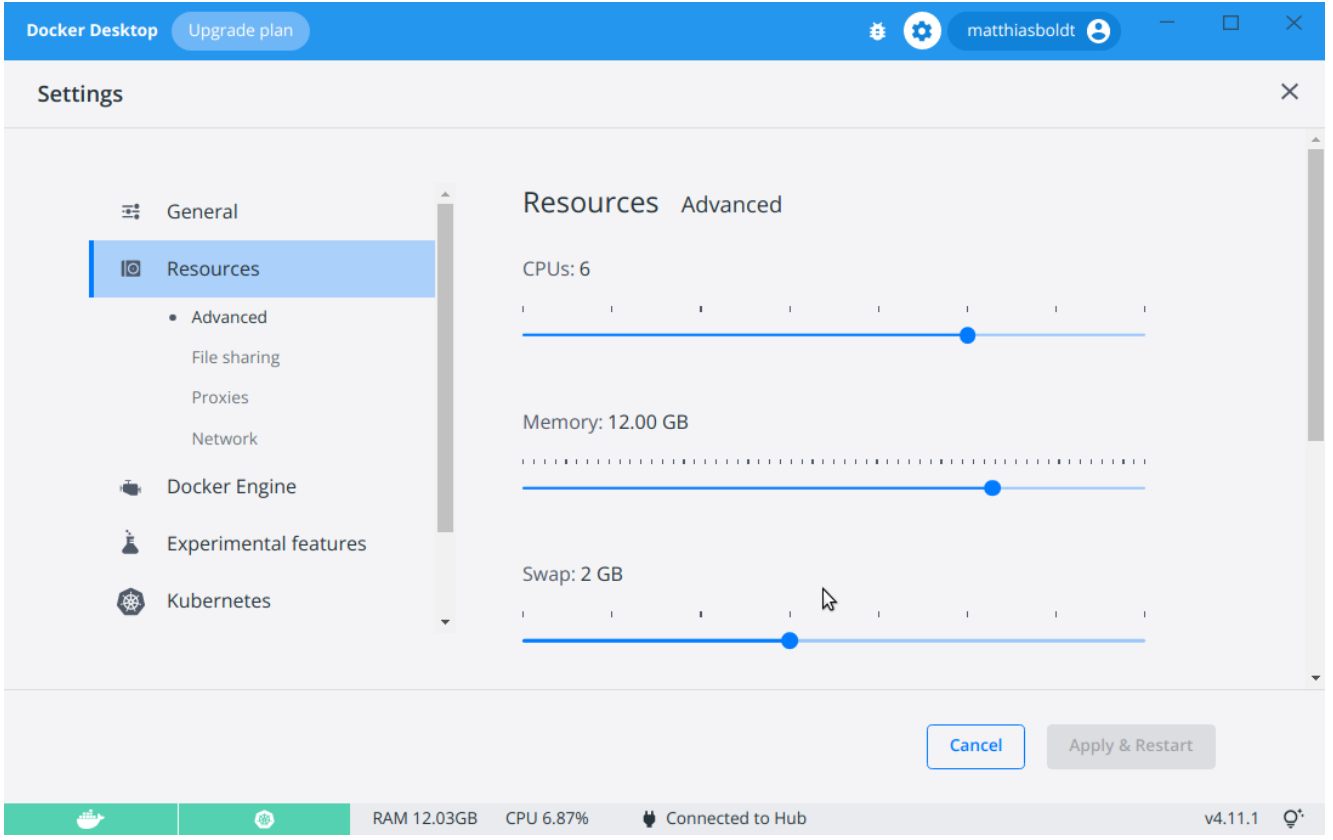
Docker Desktop - Konfiguration

- Allgemeine Einstellungen für die Komponenten von Docker Desktop



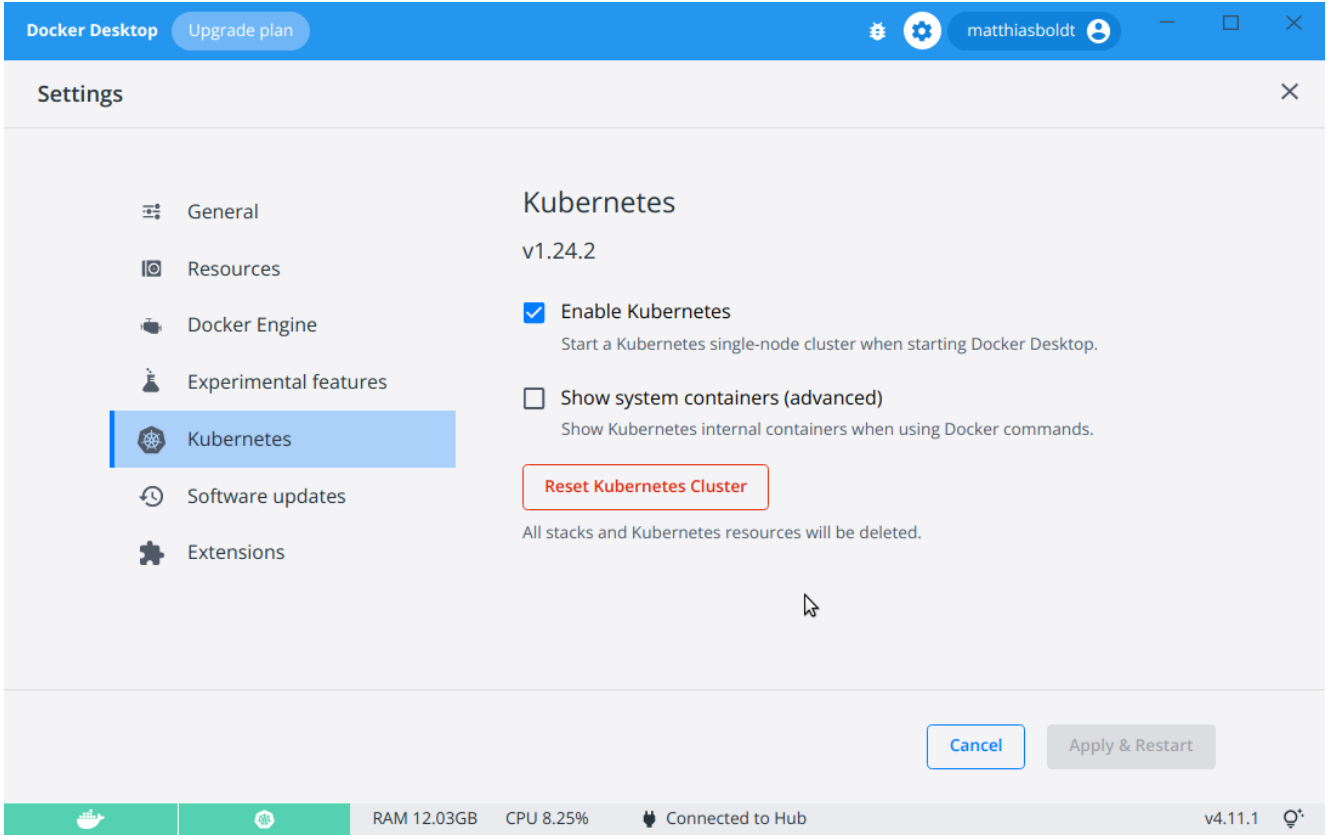
Docker Desktop - Konfiguration

- abhängig von den Ressourcen der genutzten Umgebung
- nicht zu gering wählen
- parallel genutzte Software (z.B. Dev-Umgebung) beachten



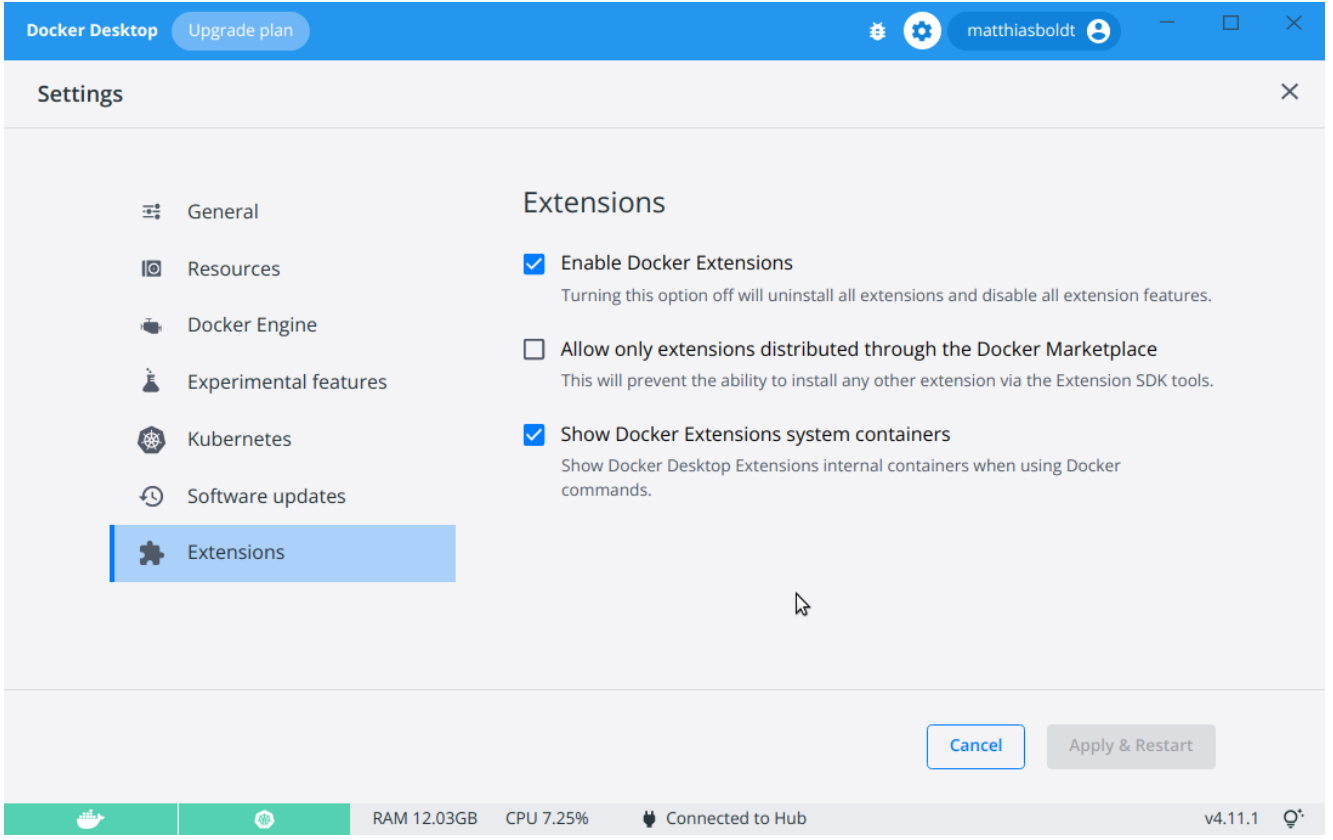
Docker Desktop - Konfiguration

- Ist die Nutzung von Kubernetes nicht vorgesehen … das auch nicht auswählen
- Startet viele Container mit Kubernetes-Basis-Diensten



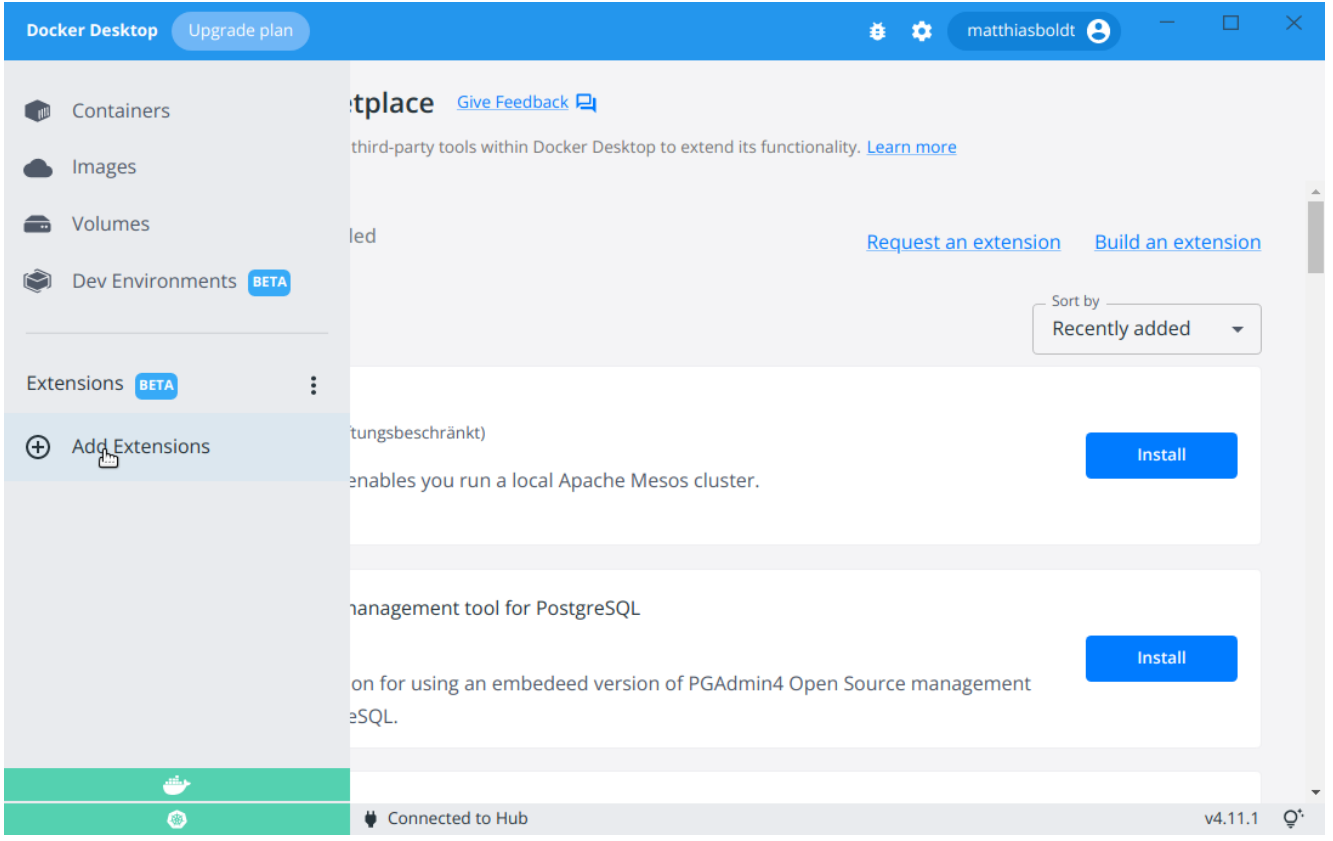
Docker Desktop - Konfiguration

- Nützlich aber optional



Docker Desktop - Konfiguration

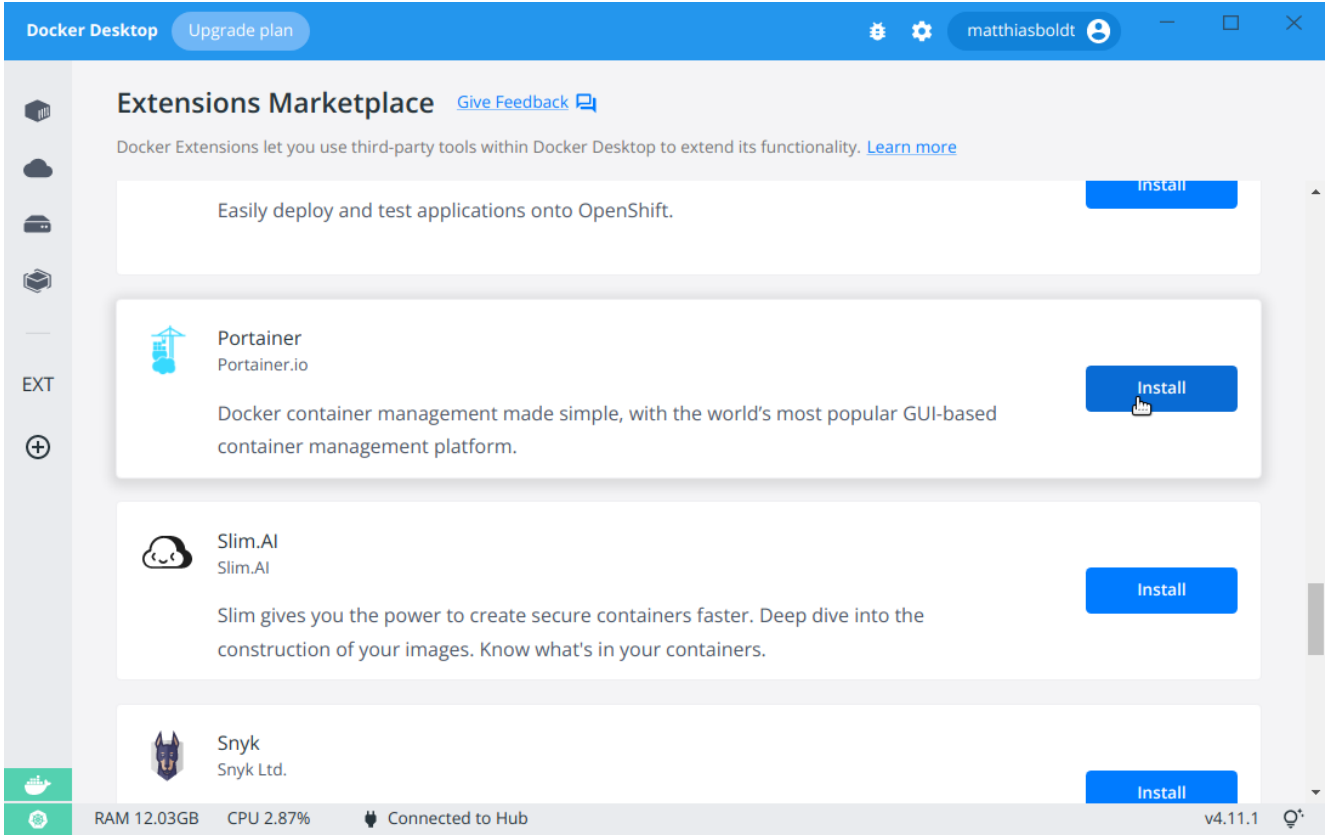
- Dashboard zeigt, ob Docker und Kubernetes lafen
- Auswahl von Extensions



Docker Desktop - Konfiguration

- Portainer.io
- Sinnvolle Verwaltung von Images, Containern, Volumes, … aus dem Browser

- Alle Funktionen … zur Ergänzung von Docker Desktop

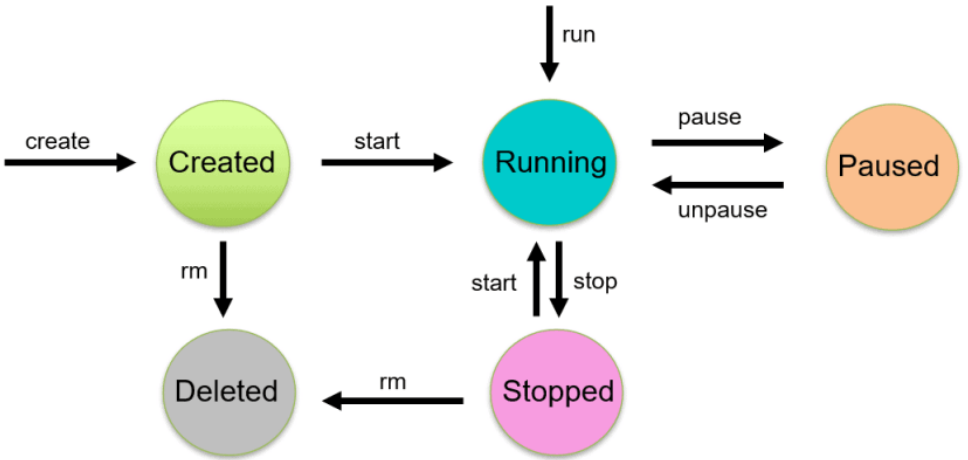


Docker Desktop - Nutzung

- enthaltene Funktionen zur Image Verwaltung
  - Pull
  - Push (zu Docker Hub)
  - Remove
  - Inspect

Docker Desktop - Nutzung

- enthaltene Funktionen zur Container Verwaltung
  - Pause
  - Resume
  - Stop
  - Start
  - Delete

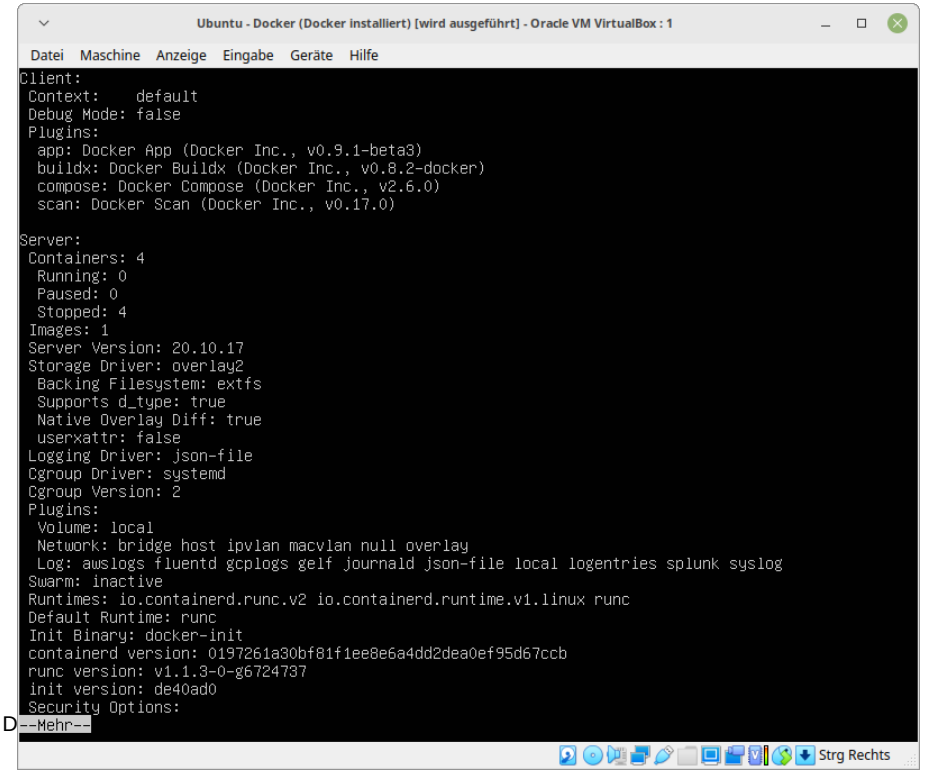


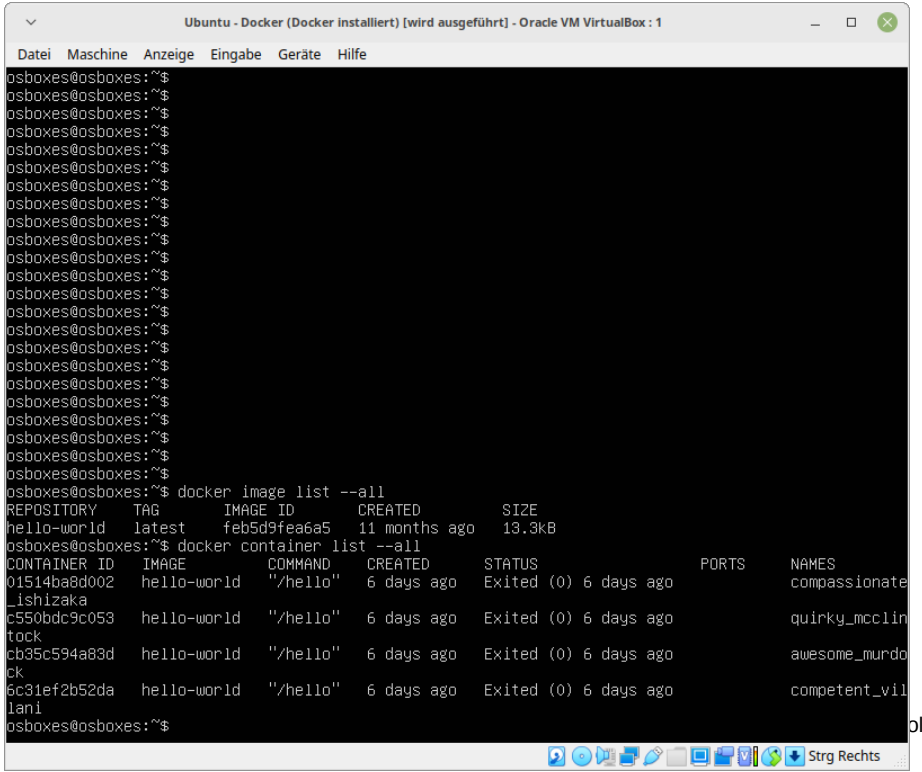
Docker Desktop - Nutzung

- enthaltene Funktionen zur Volume Verwaltung
  - Delete
  - Create

Docker Engine - Installation

- Linux-Server (ubuntu 22.04) … Vorbereitete VM für VirtualBox „Ubuntu-Docker.ova“
- WSL auf Windows 10/11 mit ubuntu-Basis
- Sammlung der Kommandos in „CMD-Docker-auf-Linux-Server.txt“





Docker Engine - Installation

- Grundlegende Pakte

```
$ sudo apt-get update
$ sudo apt-get install ca-certificates curl gnupg lsb-release
```

- GPG key

```
$ sudo mkdir -p /etc/apt/keyrings
$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o /etc/apt/keyrings/docker.gpg
```

- ubuntu Paketliste (eine Zeile)

$ echo "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.gpg] https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.list > /dev/null

Docker Engine - Installation

- Docker Engine & Docker Compose installieren

```
$ sudo apt-get update
$ sudo apt-get install docker-ce docker-ce-cli containerd.io docker-compose-plugin
```

- Funktionstest (einen Container ausführen)

```
$ sudo docker container run hello-world
```

- Start mit Boot konfigurieren

```
$ sudo systemctl enable docker.service
$ sudo systemctl enable containerd.service
```

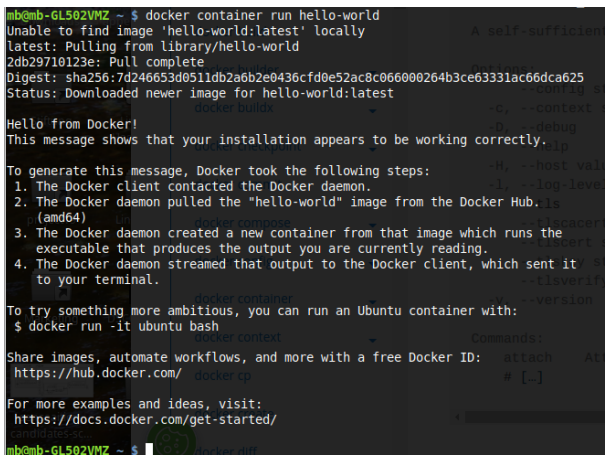
Docker Engine - Installation

- Als Nicht-Root-User starten

```
$ sudo groupadd docker
$ sudo usermod -aG docker $USER
$ newgrp docker
$ mkdir /home/"$USER"/.docker
$ sudo chown "$USER":"$USER" /home/"$USER"/.docker -R
$ sudo chmod g+rwx "$HOME/.docker" -R
```

- Test als Nicht-Root-User
  - Funktioniert auch unter Windows

```
$ docker container run hello-world
```



Docker Engine – Portainer

- Data Volume erzeugen

```
$ docker volume create portainer_data
```

- Portainer laden & starten als Container (eine Zeile)

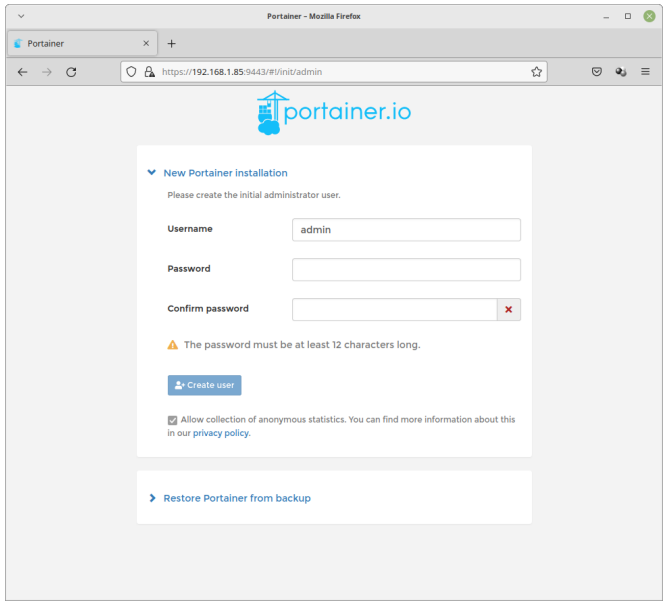
```
$ docker run -d -p 8000:8000 -p 9443:9443 --name portainer --restart=always -v
/var/run/docker.sock:/var/run/docker.sock -v portainer_data:/data portainer/portainer-ce:2.14.2
```

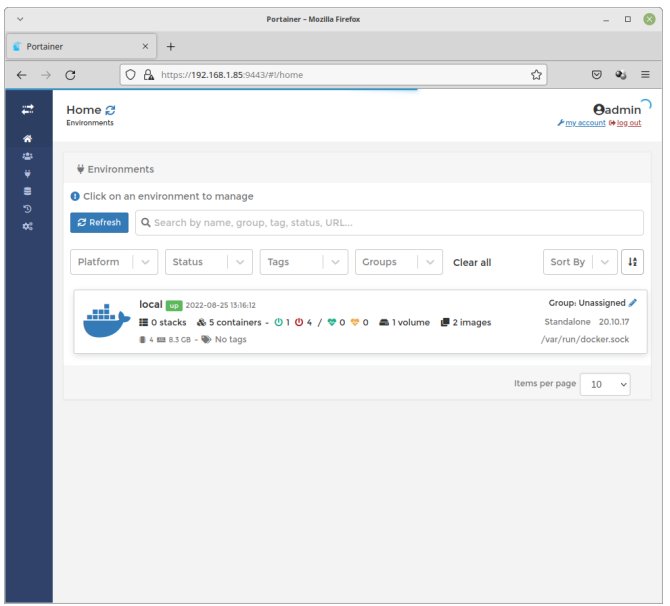
```
$ docker container list
```

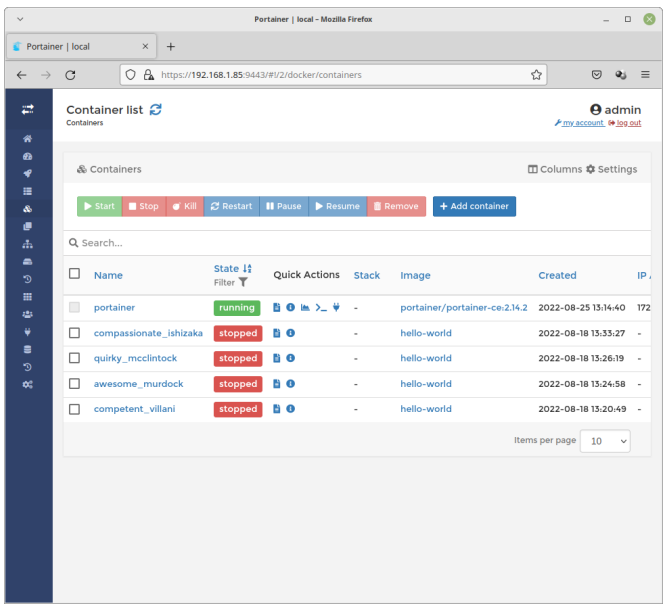
```
osboxes@osboxes:~$
osboxes@osboxes:~$
osboxes@osboxes:~$ docker container list
CONTAINER ID   IMAGE                           COMMAND        CREATED         STATUS         PORTS
                                                                                             NAMES
92dc20f1de87   portainer/portainer-ce:2.14.2   "/portainer"   9 minutes ago   Up 8 minutes   0.0.0.0
:8000->8000/tcp, :::8000->8000/tcp, 0.0.0.0:9443->9443/tcp, :::9443->9443/tcp, 9000/tcp   portainer
osboxes@osboxes:~$
```

Docker Engine – Portainer

- Start im Browser: https://localhost:9443
- Domain/IP-Nummer passend tauschen :-)







Docker - CLI-Kommandos

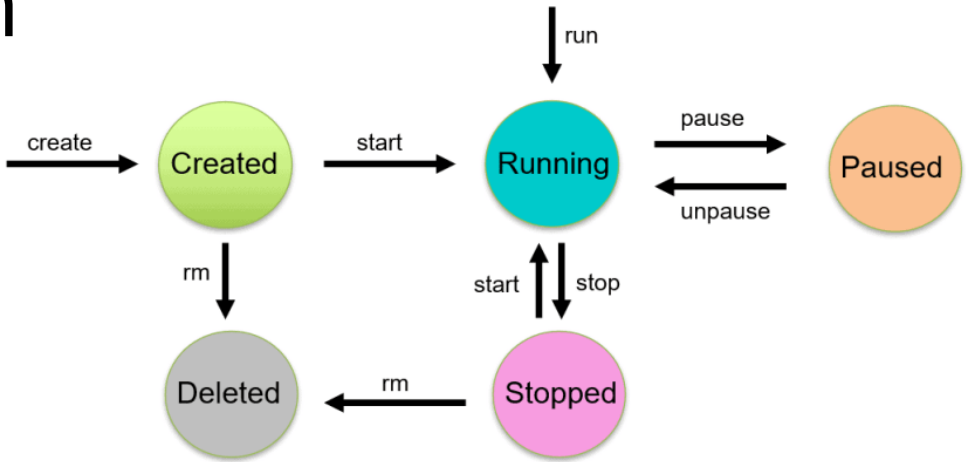
- Wesentliche Kommandos des CLI
- Übersicht im CLI selbst

```
$ docker --help
```

- Aufruf immer in der Form

```
$ docker [OPTIONS] COMMAND [SUBCOMMAND] [ARG ...]
```

- Steuerung des Zyklus



Docker CLI

- Dokumentation

  https://docs.docker.com/engine/reference/commandline/cli/

- Management-Kommandos & Subkommandos (oft ohne Management-Kommandos aufrufbar => viele Dopplungen) => allgemeine Syntax:

```
$ docker [OPTIONS] COMMAND [SUBCOMMAND] [ARG ...]
```

- Eine Auswahl der wichtigsten Kommandos ...

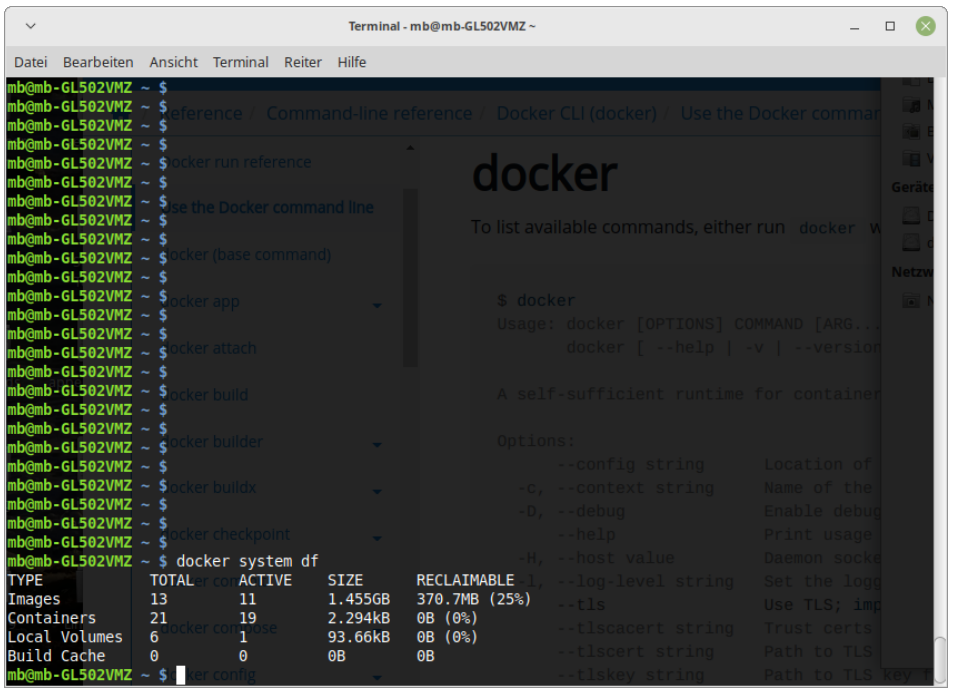
Docker CLI – wichtige Kommandos

- system Manage Docker
- image Manage images
- container Manage containers
- network Manage networks
- volume Manage volumes

Docker CLI – *system* Kommando

- Festplattenbelegung anzeigen

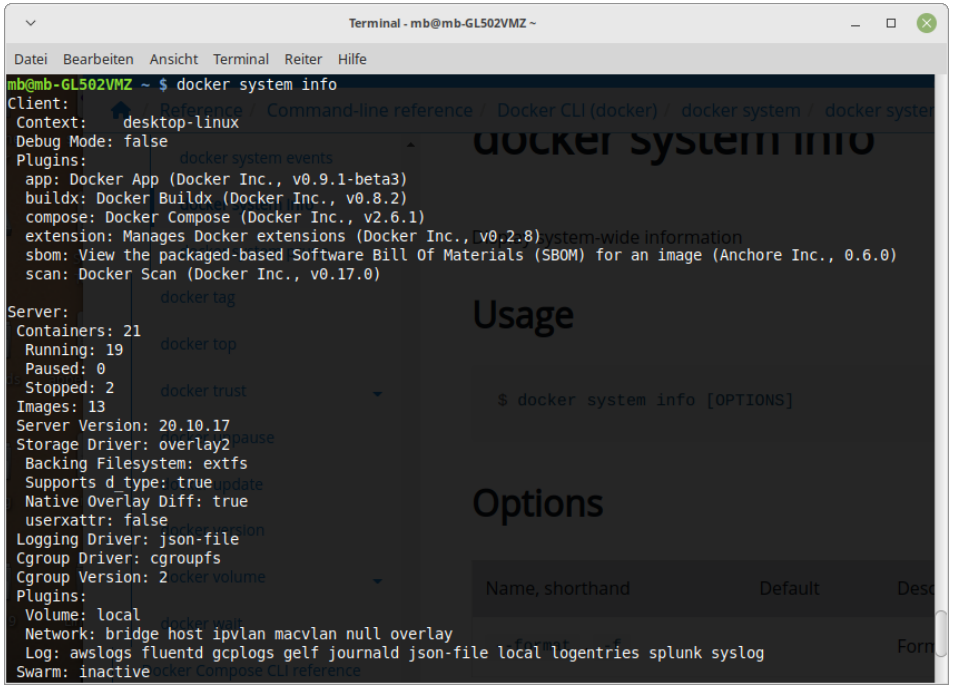
```
$ docker system df
```



Docker CLI – *system* Kommando

- Systeminformationen anzeigen

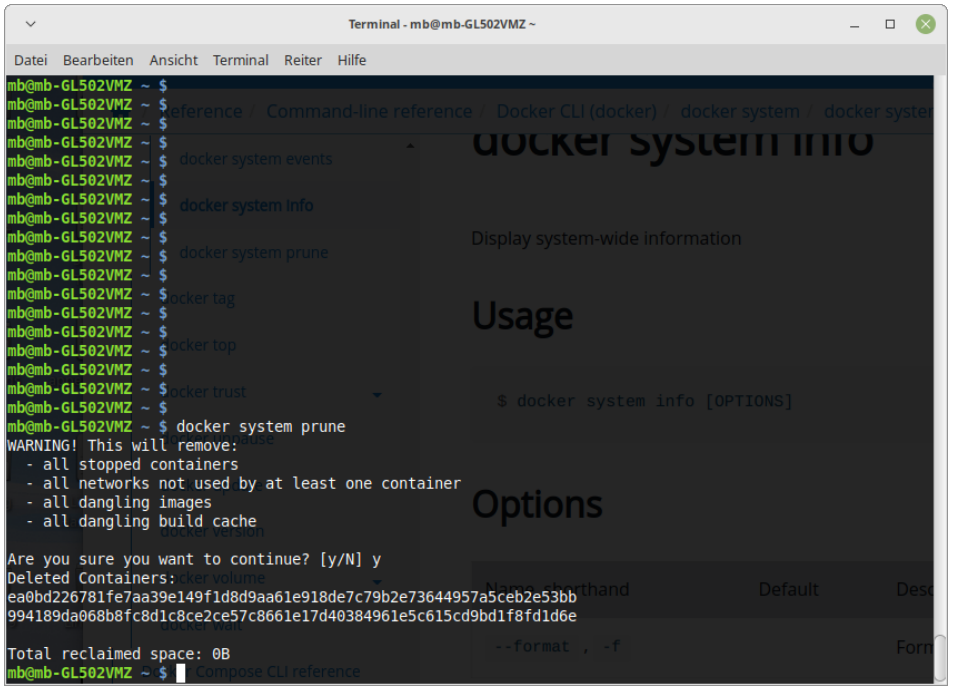
```
$ docker system info
```



Docker CLI – *system* Kommando

- Speicherplatz freigeben / aufräumen

```
$ docker system prune
```

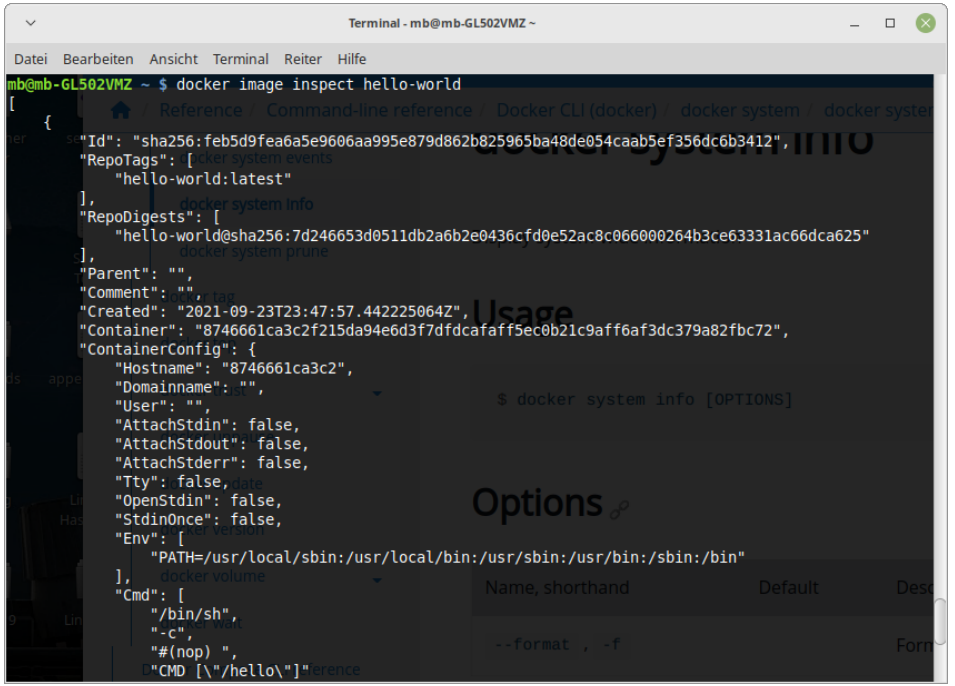


Docker CLI – *image* Kommando

- Ein Image untersuchen

```
$ docker image inspect <image name>
```

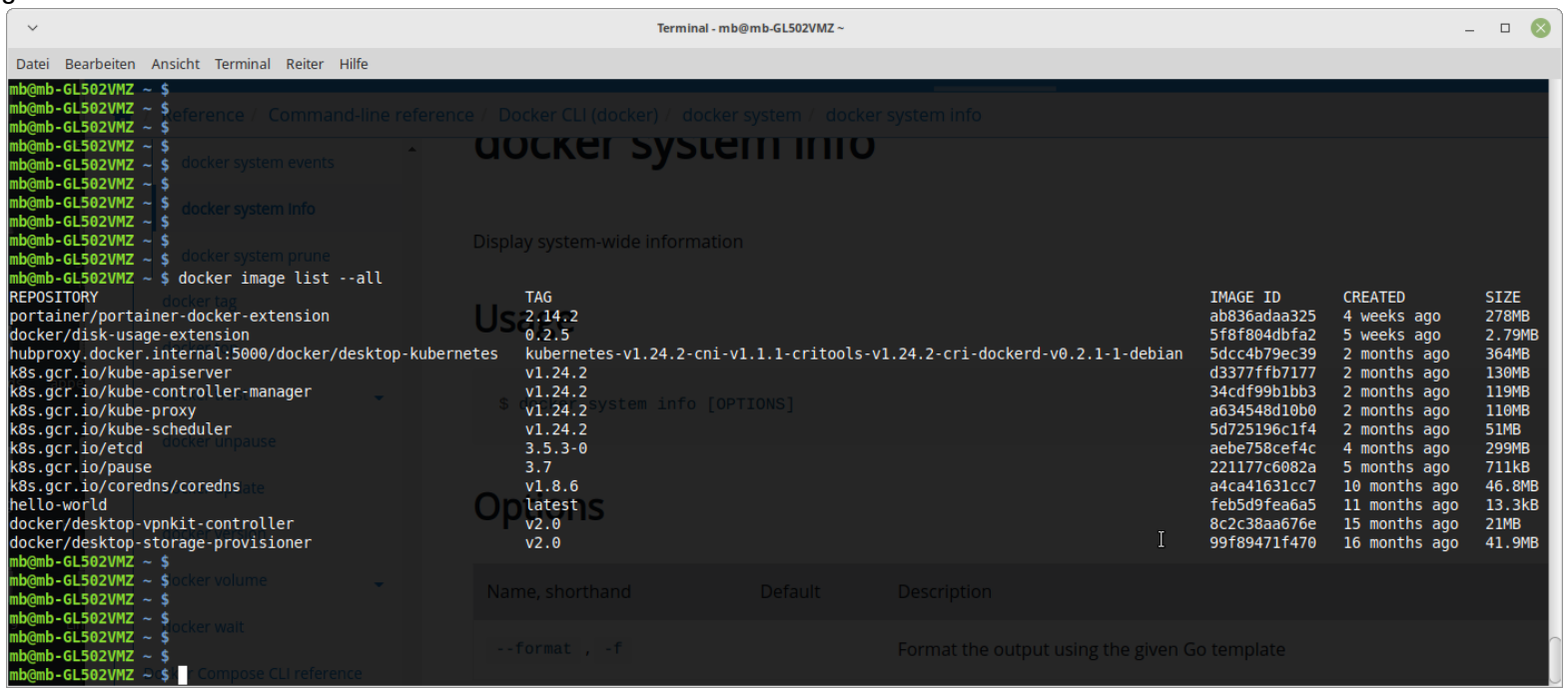
```
$ docker image inspect hello-world
```



Docker CLI – *image* Kommando

- Images auflisten

```
$ docker image list
$ docker image list --all
```

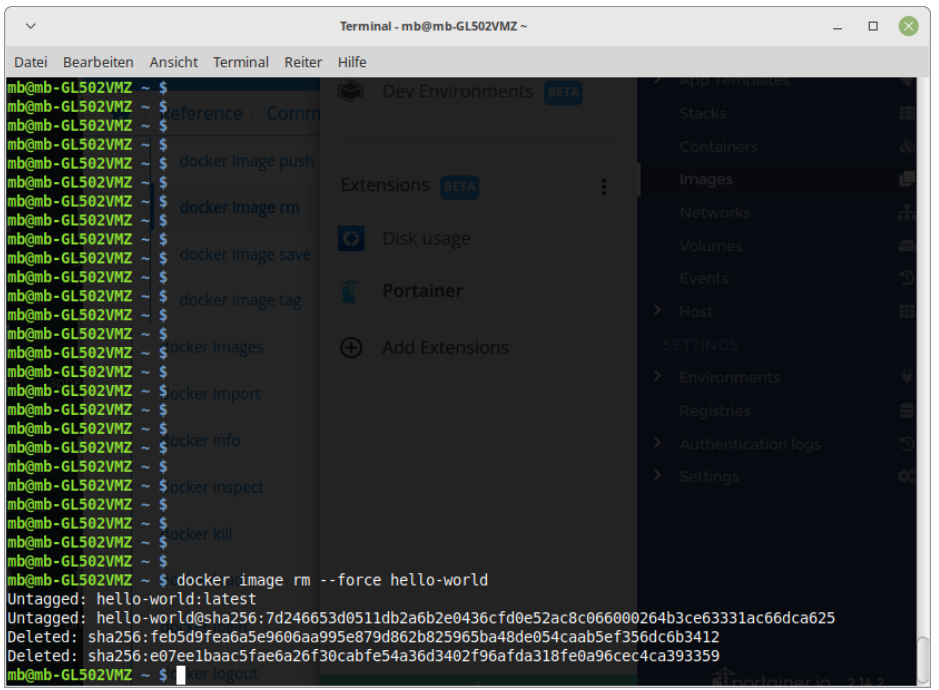


Docker CLI – *image* Kommando

- Ein Image entfernen

```
$ docker image rm --force <image name>
```

```
$ docker image rm --force hello-world
```



Docker CLI – *image* Kommando

- Ein Image von Registry laden

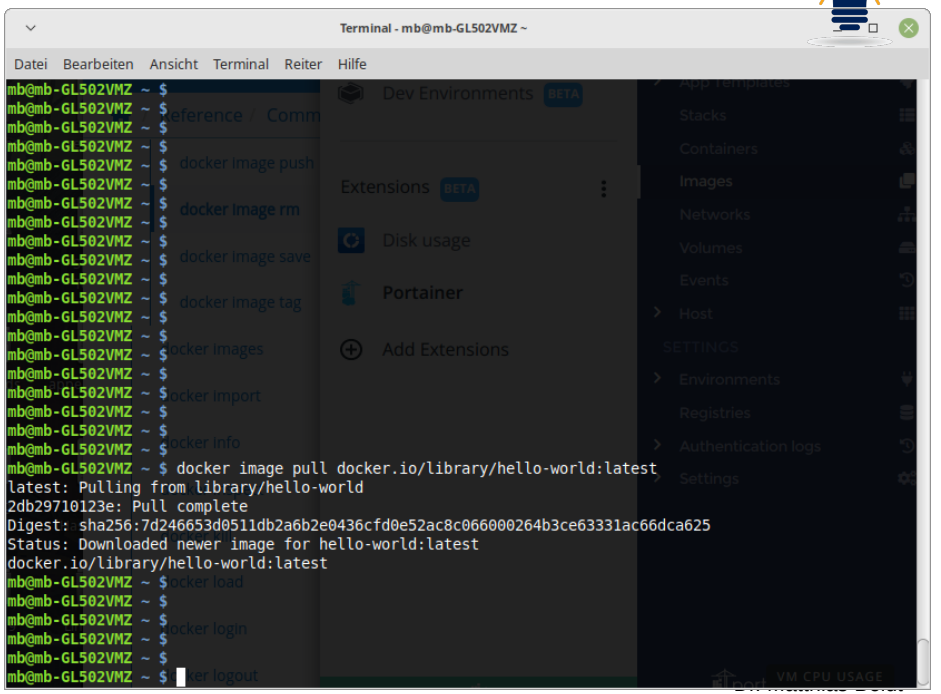
```
$ docker image pull <image name>
```

```
$ docker image pull docker.io/library/hello-world:latest
```

<image name>

- Registry-URL
- Name des Images
- „:“
- Tag

bei Nutzung von Docker Hub kann die Registry-URL entfallen



Docker CLI – *image* Kommando

- Ein Image in Registry speichern

```
$ docker image push <image name>

$ docker image push registry.gitlab.com/xxx/test:latest
```

- Zuvor Tag anlegen

```
$ docker image SOURCE_IMAGE[:TAG] TARGET_IMAGE[:TAG]

$ docker -l fatal image tag registry.gitlab.com/xxx/test:v123 registry.gitlab.com/xxx/test:latest"
```

Docker CLI – *image* Kommando

- Ein Image als Datei speichern

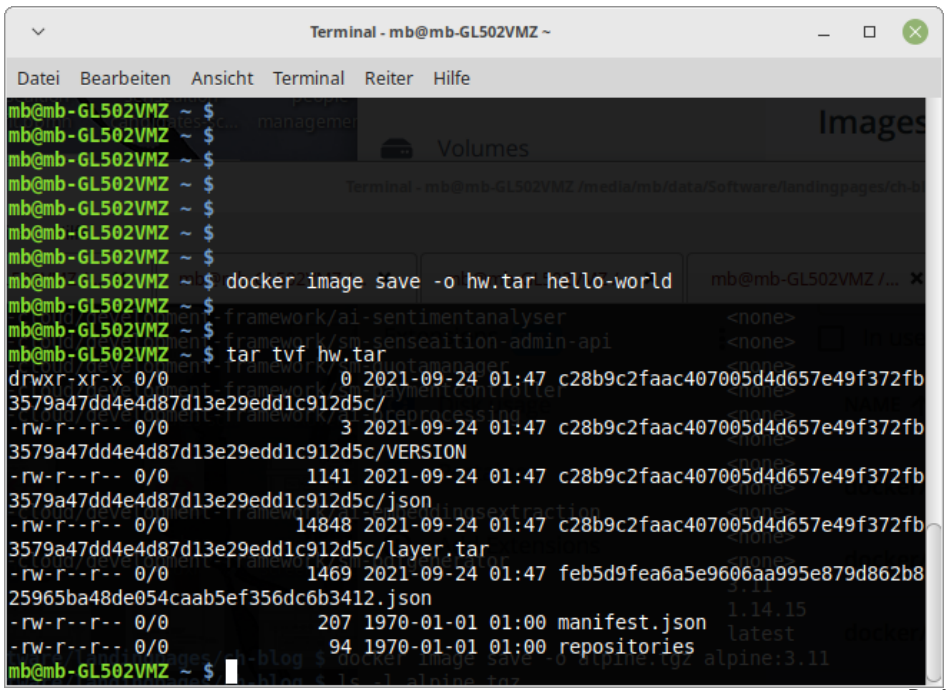
```
$ docker image save <image name>
$ docker image save -o <tar-file name> <image name>
```

```
$ docker image save -o hw.tar hello-world
```

```
$ docker image save hello-world > hw.tar
```

Docker schreibt das Abbild des Images in ein TAR-Archiv.

Ohne „-o“ wird auf die Standardausgabe STDOUT geschrieben und mit „-o“ wird die spezifizierte Datei erzeugt.



Docker CLI – *image* Kommando

- Ein Image aus Datei laden

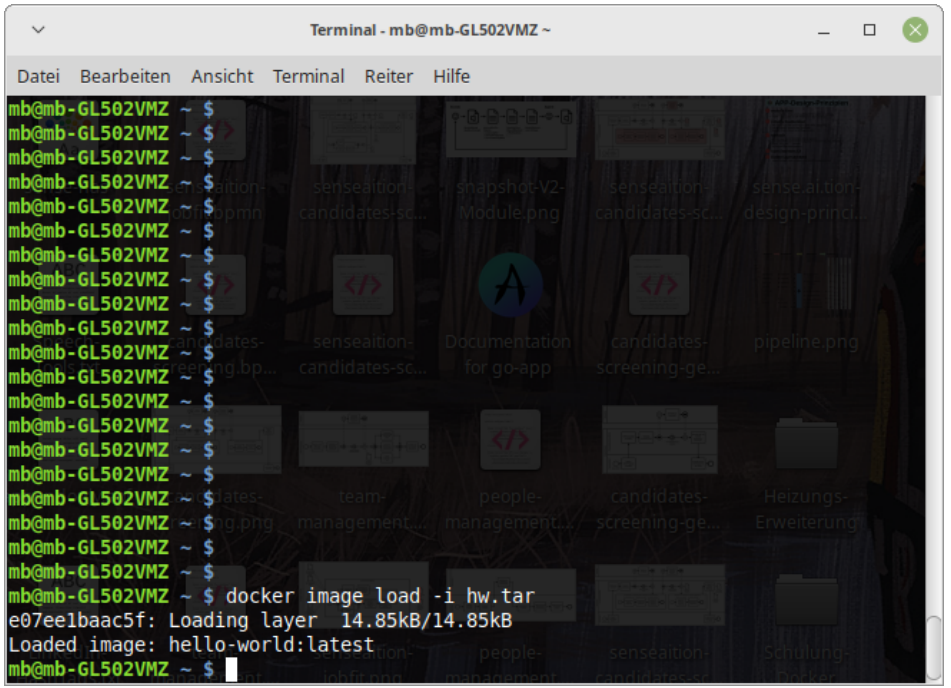
```
$ docker image load
$ docker image load -i <tar-file name>
```

```
$ docker image rm hello-world
$ docker image load -i hw.tar
```

```
$ docker image rm hello-world
$ cat hw.tar | docker image load
```

Docker liest ein Abbild eines Images aus einem TAR-Archiv.

Ohne „-i“ wird von der Standardeingabe STDIN gelesen.



Docker CLI – *image* Kommando

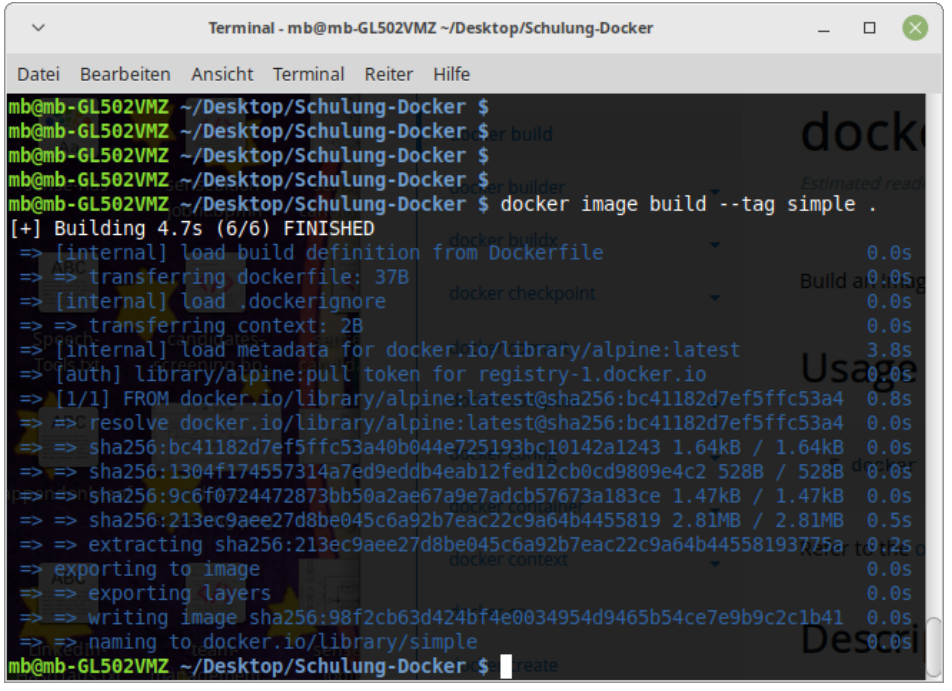
- Ein Image erzeugen/bauen

```
$ docker image build [OPTIONS] PATH | URL | -
$ docker image build --tag <image name>:<image tag> .
$ docker image build –file <Dockerfile-name> --tag <image name>:<image tag> .
```

```
$ docker image build --tag simple .
$ cat Dockerfile | docker image build --tag simple -
```

Docker erstellt anhand eines „Dockerfile“ ein Image. Die Datei „Dockerfile“ wird im Verzeichnis PATH erwartet. Mit Option „--file“ kann ein anderer Dateiname angegeben werden.

```
# sehr einfaches Dockerfile
FROM alpine:latest
CMD ["echo", "Das ist ein sehr einfacher Container..."]
```



Docker CLI – *Dockerfile*

- Dockerfile (Wiederholung :-)
  - Text-Datei
  - Folge aus Anweisungen zum Aufbau des Images
  - Ist extrem vielseitig und flexibel nutzbar
  - Jede Anweisung erzeugt ein Layer im Image
  - Anweisung in Form: `INSTRUCTION arguments`
  - Kommentare mit „#“ am Anfang der Zeile
  - https://docs.docker.com/engine/reference/builder/
  - Wesentliche Anweisungen ...

Docker CLI – *Dockerfile* - Beispiel

```
# ai-audioanalyser
FROM alpine:latest

# installation of necessary certificates for ssl communication
RUN apk --no-cache add ca-certificates

# installation of service module for the API within the home directory of the image
ADD ai-audioanalyser /home

# only necessary for running container within docker, kubernetes manages its own
environment
ENV SMLOGLEVEL 100
ENV SMHOST 0.0.0.0
ENV SMPORT 4242
ADD cluster.json /home
ADD queue.json /home
ADD keywords.csv /home
ADD classifier.csv /home

# start program/process
STOPSIGNAL SIGTERM
EXPOSE ${SMPORT}
ENTRYPOINT /home/ai-audioanalyser
```

Docker CLI – *Dockerfile*

`FROM <image name>:<tag>`

- Setzt das „base image“
- Alle weiteren Build-Schritte beziehen sich darauf
- Festlegung der Plattform möglich
- Alpine Linux, Ubuntu, … Windows

```
FROM alpine:latest
```

Docker CLI – *Dockerfile*

`RUN <command>`

- Führt ein Kommando in einem neuen Layer aus
- Das Ergebnis wird integriert in das Image

```
RUN /bin/bash -c 'source $HOME/.bashrc; echo $HOME'
RUN apk --no-cache add ca-certificates
RUN cd /src && CGO_ENABLED=0 go build -o /root/client ./main.go
```

Docker CLI – *Dockerfile*

`WORKDIR <directory>`

- Arbeitsverzeichnis (im Container) setzen
- Für Kommandos
  - RUN
  - CMD
  - ENTRYPOINT
  - COPY
  - ADD

```
WORKDIR /home
```

Docker CLI – *Dockerfile*

`ADD <src> <desc>`

- Kopiert von <src> nach <desc>
- Kopiert werden Dateien, Verzeichnisse (auch von URLs) von <src>
- Kopiert wird nach <desc> im Image
- Endet <desc> mit „/“, wird in das Verzeichnis kopiert, sonst wird eine Datei geschrieben (falls <desc> kein Verzeichnis ist)
- Kann TAR-Archive automatisch entpacken

```
ADD build/mybinary /home/
ADD http://domain.com/config.txt /home/config/
ADD data/* /home/data/
```

Docker CLI – *Dockerfile*

`COPY <src> <desc>`

- Wie ADD, aber
  - Kein Abruf von Dateien von URL‘s
  - Kein automatisches Entpacken von TAR-files
- *best practice* ist Nutzung von COPY

```
COPY build/mybinary /home/
COPY data/* /home/data/
```

Docker CLI – *Dockerfile*

`VOLUME <directory>`

- Benennt einen Mount-Point für externe Volumes

```
VOLUME /external-data
```

Docker CLI – *Dockerfile*

`ENV <key>=<value>`

- Setzen von Umgebungsvariablen
- 12 Faktoren => Parametrisierung von Programmen über Umgebungsvariablen
- Umgebungsvariablen können beim Start eines Containers überschrieben werden

```
$ docker container run --env <key>=<value> <image>
```

```
ENV LOGLEVEL=100
ENV HOST=0.0.0.0
ENV PORT=4242
```

Docker CLI – *Dockerfile*

`EXPOSE <port>`

`EXPOSE <port>/<protocol>`

- Teilt Docker mit, welche Ports der Container nach außen nutzt
- Können bei Start automatisch publiziert werden

```
$ docker container run <image> --publish-all
```

- Überschreiben beim Start

```
$ docker container run <image> --publish <host-port>:<container-port>
$ docker container run <image> --publish 80:4242
$ docker container run <image> --publish 443:443
```

```
EXPOSE 4242
EXPOSE 80/udp
```

Docker CLI – *Dockerfile*

```
CMD <command> <para-1> … <para-n>
CMD [“<command>“],[“<para-1>“],…, [“<para-n>“]
CMD [“<para-1>“],…, [“<para-n>“]
```

- Standardprogramm des Containers
- Kommando wird bei Start des Containers ausgeführt
- Erste Form wird in shell gestartet (von „/bin/sh -c“ aus)
- Dritte Form dient der Parameterübergabe an ENTRYPOINT
- Dockerfile darf nur ein einziges CMD enthalten

```
CMD ["echo", "Das ist ein sehr einfacher Container…"]
CMD echo "Das ist ein sehr einfacher Container…"
```

Docker CLI – *Dockerfile*

```
ENTRYPOINT <command> <para-1> … <para-n>
ENTRYPOINT [“<command>“],[“<para-1>“],…, [“<para-n>“]
```

- Standardprogramm, das bei Start des Containers ausgeführt wird
- Mit ENTRYPOINT kann ein Container als Executable ausgeführt werden
- Erste Form wird in shell gestartet (von „/bin/sh -c“ aus)
- Dockerfile darf nur ein einziges ENTRYPOINT enthalten

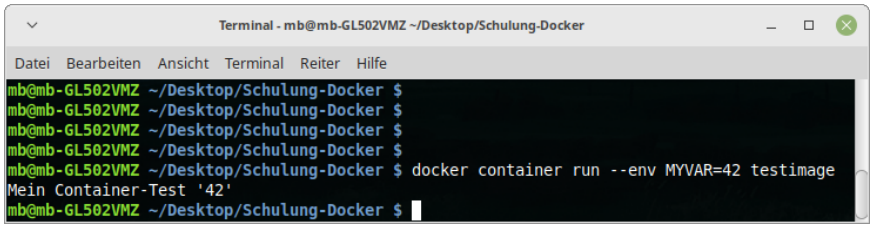
```
ENTRYPOINT ["echo", "Das ist ein sehr einfacher Container…"]
ENTRYPOINT echo "Das ist ein sehr einfacher Container…"
```

Docker CLI – *Dockerfile*

- Entwurf eines eigenen Dockerfiles
  - „Dockerfile-1.test“
  - Umgebungsvariable MYVAR auf „1“ setzen
  - Container beim Start/Lauf gibt kurzen Text plus Wert von MYVAR aus
- … editieren … ausprobieren

```
$ docker image build --tag testimage --file Dockerfile-1.test .
$ docker image build -t testimage -f Dockerfile-1.test .

$ docker container run testimage
$ docker container run --env MYVAR=42 testimage
```



- … nach 15 Minuten Vergleich

Docker CLI – *container* Kommando

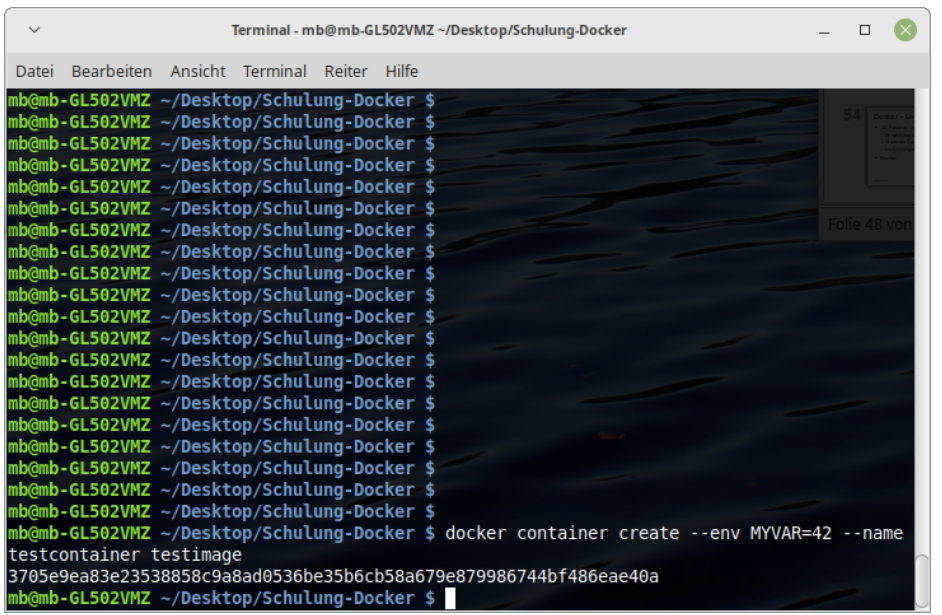
- Einen container erzeugen

```
$ docker container create [OPTIONS] IMAGE [COMMAND] [ARG…]

$ docker container create --env MYVAR=42 --name testcontainer testimage
```

Wichtigste Optionen

- --name
- --env, -e
- --interactive, -i
- --network
- --publish
- --volumn, -v
- --tmpfs
- --tty, -t



Docker CLI – *container* Kommando

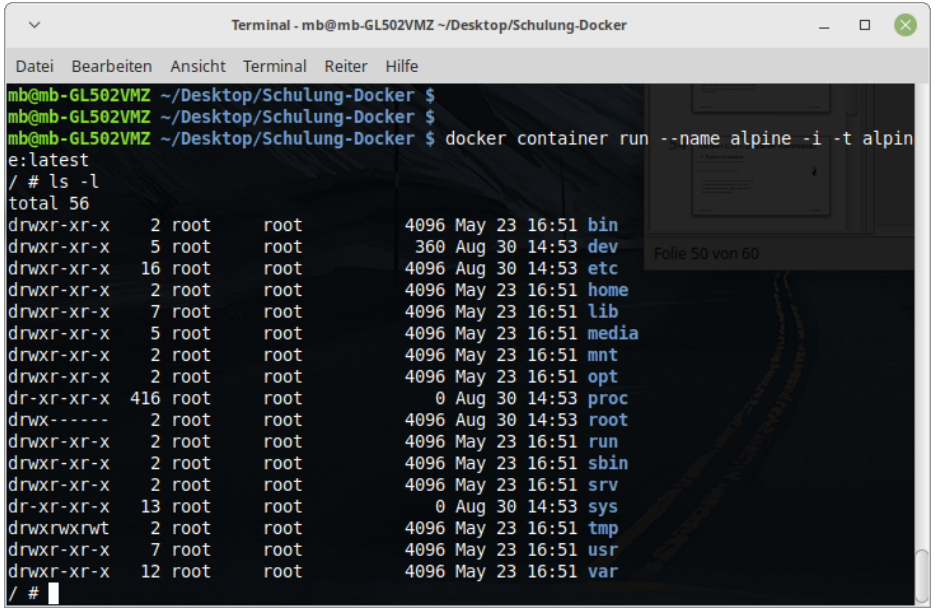
- Option von „create“

--name <container-name>
- Erzeugt einen Container unter dem angegebenen Namen
- In weiter Kommandos kann der Namen genutzt werden

--env <key>=<value>
-e <key>=<value>
- Setzt/überschreibt die Umgebungsvariable beim Start des Containers
- Kann auch Liste sein

--interactive, -i
- Verbindung zu STDIN des Containers wird von Console erzeugt

--tty, -t
- Ein pseudo-tty nutzen (Console)



Docker CLI – *container* Kommando

- Option von „create“

--network=<network name>
- Verbindung des Containers mit dem angegebenen Netzwerk

--publish [<host-ip>:]<host-port>:<container-port>[/<protocol>]
-p [<host-ip>:]<host-port>:<container-port>[/<protocol>]
- Publizieren/Binden eines Ports

```
$ docker container run -p 127.0.0.1:80:8080/tcp testcontainer
$ docker container run -p 80:8080 testcontainer
```

--volumn [<volume name>:]<host-directory>:<container-directory>
-v [<volume name>:]<host-directory>:<container-directory>
- Bindet ein Verzeichnis des Hosts in den Container ein

--tmpfs <container-directory>
- Nutzung eines temporären Verzeichnisses im RAM des Hosts
- Nur bei Linux-Hosts
- Werden nicht zwischen Containern geteilt

Docker CLI – *container* Kommando

- Einen container starten

```
$ docker container start [OPTIONS] <container name>
```

```
$ docker container start testcontainer
```

Startet einen erzeugten und/oder gestoppten Container.

Docker CLI – *container* Kommando

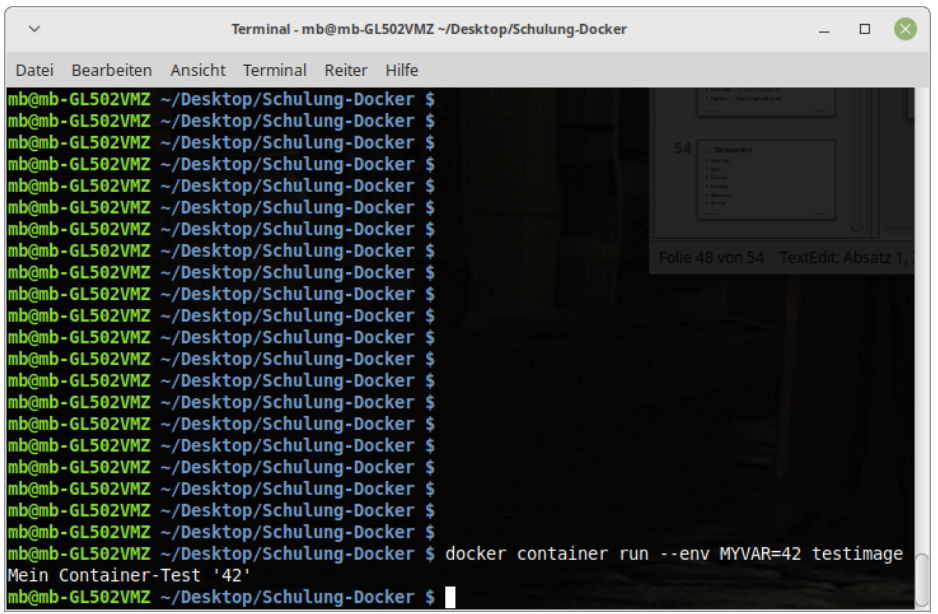
- Einen container erzeugen & starten

```
$ docker container run [OPTIONS] IMAGE [COMMAND] [ARG…]
```

```
$ docker container run --env MYVAR=42 testimage
```

„run“ führt nacheinander „create“ und „start“ aus

Die Parameter entsprechen denen von „create“.



Docker CLI – *container* Kommando

- Einen container anhalten

```
$ docker container stop [OPTIONS] CONTAINER
$ docker container kill [OPTIONS] CONTAINER
```

Stoppt einen laufenden Container.

„stop“ sendet ein SIGTERM und „kill“ ein SIGKILL an den laufenden Prozess im Container.

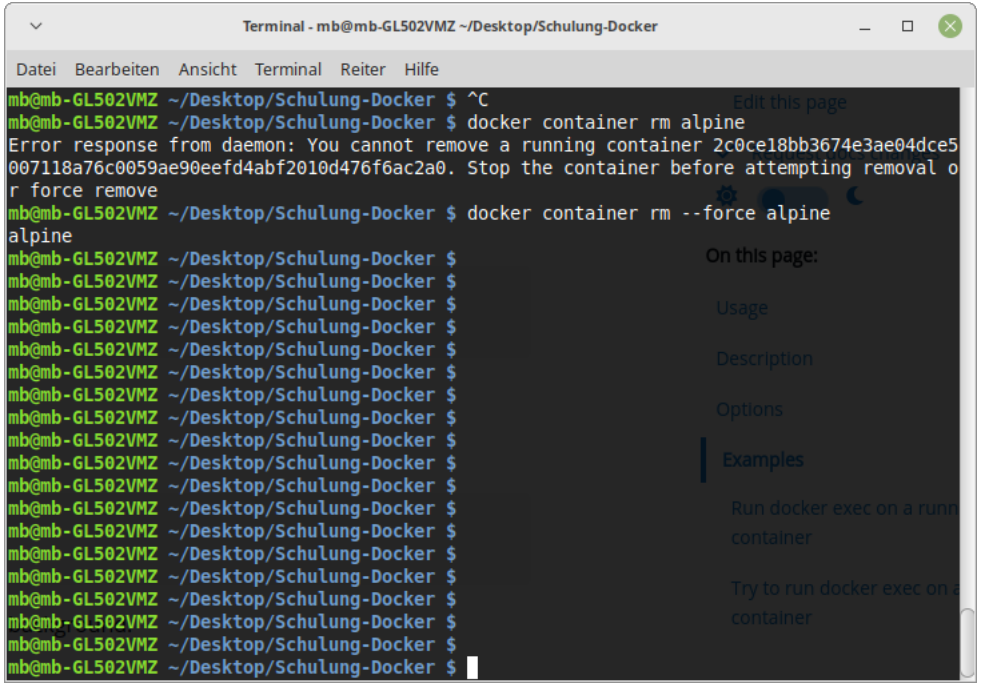
Docker CLI – *container* Kommando

- Einen container löschen

```
$ docker container rm [OPTIONS] CONTAINER

$ docker container run --name alpine --rm -i -t -d alpine top
$ docker container rm --force testcontainer
```

Mit --force können auch laufende Container gelöscht werden.



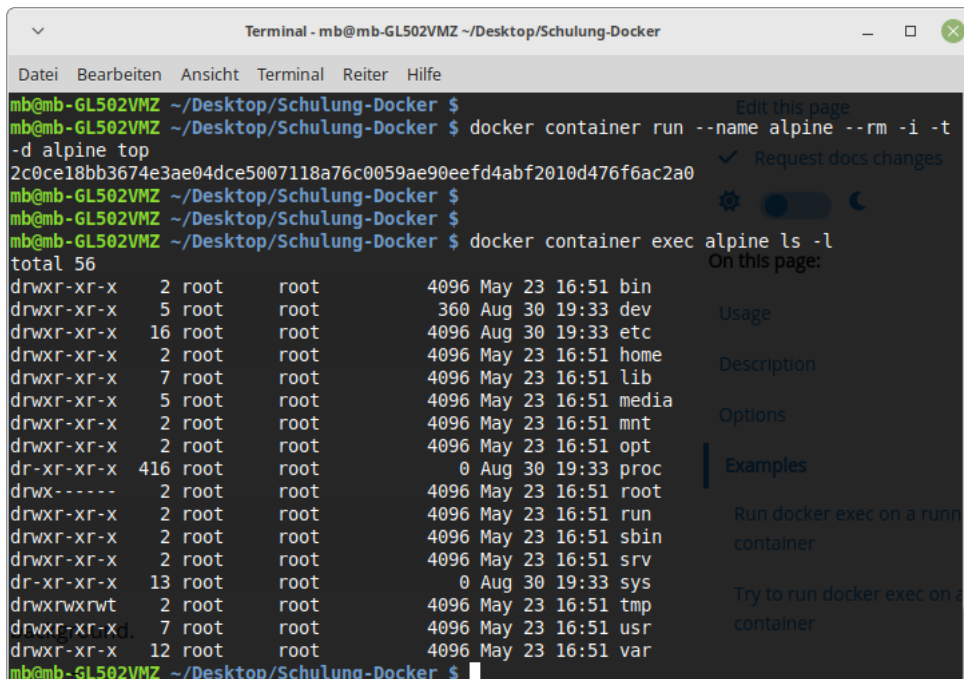
Docker CLI – *container* Kommando

- Ein zus. Programm in einem container starten

```
$ docker container exec [OPTIONS] CONTAINER COMMAND [ARG...]
```

```
$ docker container run --name alpine --rm -i -t -d alpine top
$ docker container exec alpine ls -l
```

Das Programm/Kommando wird nur in einem laufenden Container ausgeführt.



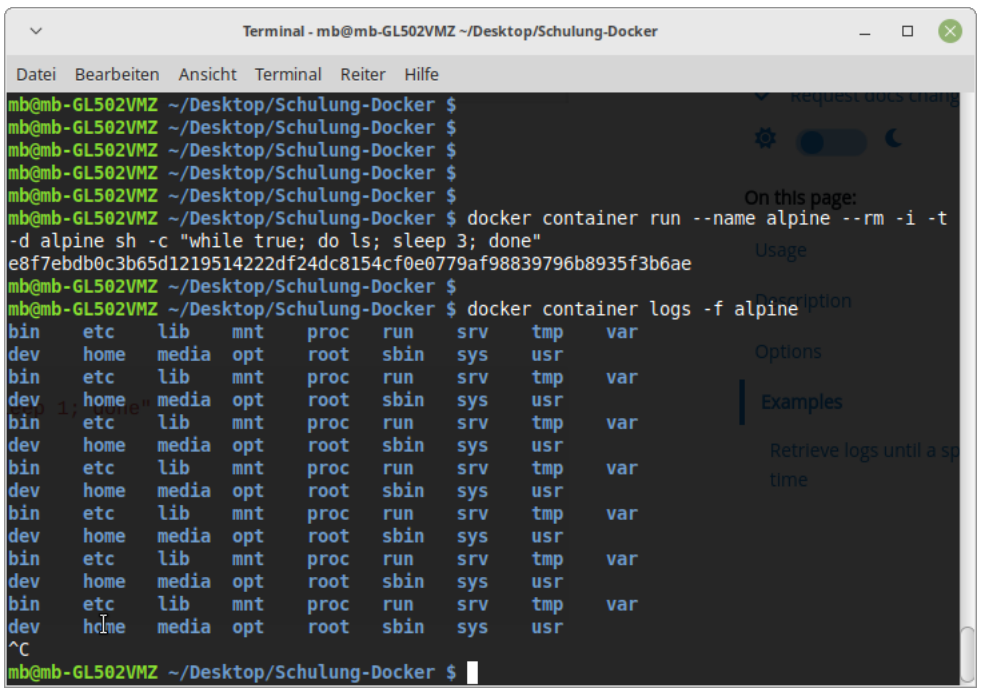
Docker CLI – *container* Kommando

- Logs eines Containers zeigen

```
$ docker container logs [OPTIONS] CONTAINER

$ docker container run --name alpine --rm -i -t -d alpine sh -c "while true; do ls; sleep 3; done"
$ docker container logs -f alpine
```

Die Option „-f“ folgt dem Ende der Logs … STDOUT.



Docker CLI – *container* Kommando

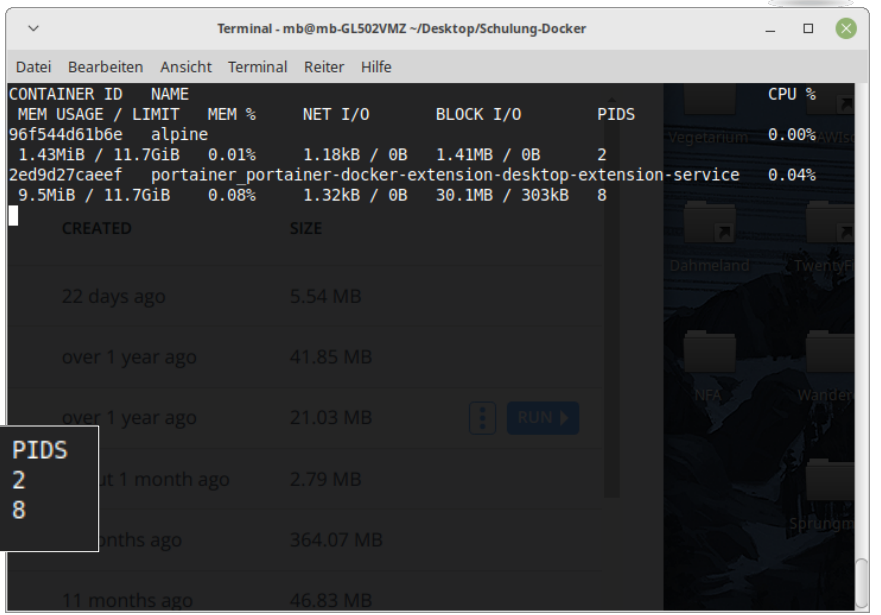
- Ressourcenverbrauch der Container zeigen

```
$ docker container stats [OPTIONS] [CONTAINER...]

$ docker container run --name alpine --rm -i -t -d alpine sh -c "while true; do ls; sleep 3; done"
$ docker container stats
$ docker container rm --force alpine
```

Zeigt Verbrauch der Ressourcen des Hosts und von eingestellten Limits.

Beenden mit String+C




```
CPU %     MEM USAGE / LIMIT    MEM %     NET I/O       BLOCK I/O        PIDS
0.00%     1.418MiB / 11.7GiB   0.01%     1.39kB / 0B   1.41MB / 0B      2
0.00%     9.703MiB / 11.7GiB   0.08%     1.46kB / 0B   30.2MB / 344kB   8
```

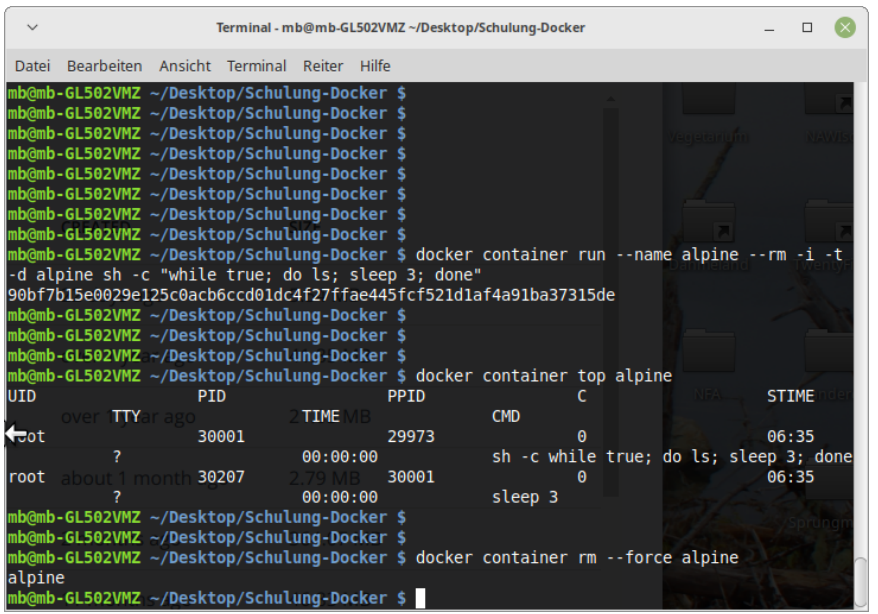
Docker CLI – *container* Kommando

- Prozesse eines Containers zeigen

```
$ docker container top CONTAINER

$ docker container run --name alpine --rm -i -t -d alpine sh -c "while true; do ls; sleep 3; done"
$ docker container top alpine
$ docker container rm --force alpine
```

Zeigt die in einem Container laufenden Prozesse an.



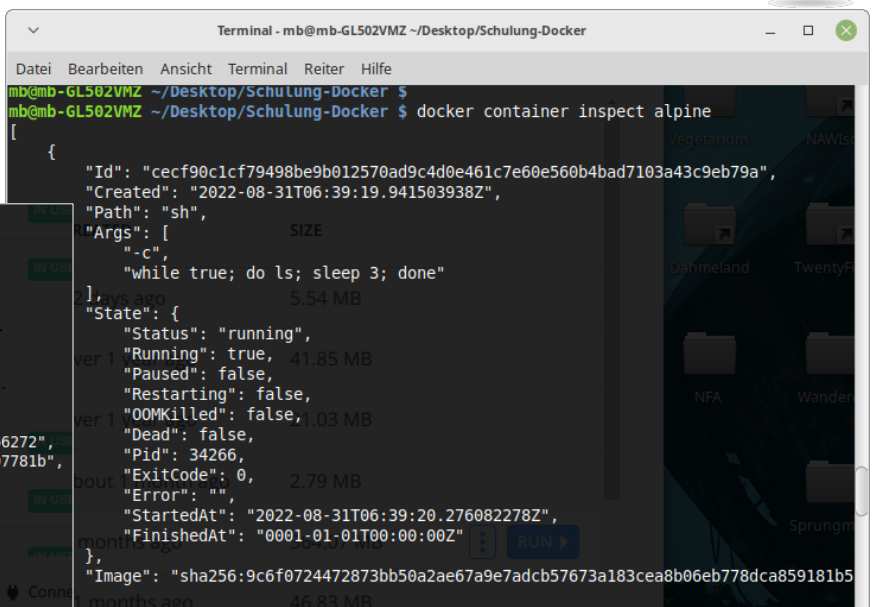
Docker CLI – *container* Kommando

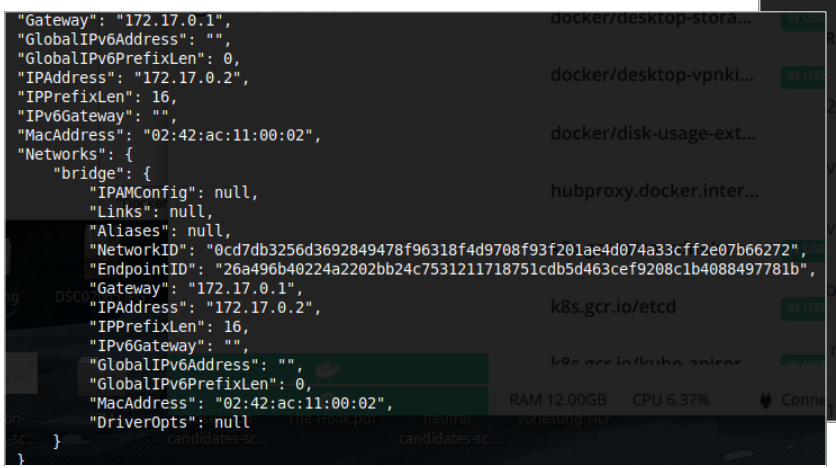
- Details eines Containers zeigen

```
$ docker container inspect CONTAINER

$ docker container run --name alpine --rm -i -t -d alpine sh -c "while true; do ls; sleep 3; done"
$ docker container inspect alpine
$ docker container rm --force alpine
```

Zeigt detaillierte Informationen über einen Container an … z.B. zum Netzwerk.





… Bildquellen

- RedHat
- IDG
- Docker
- Earthly
- Aquasec
- dev.to