Docker

Doku: https://docs.docker.com/

Basisinstallation

- https://docs.docker.com/engine/installation/linux/suse/
- Installation auf Ubuntuhttps://docs.docker.com/engine/installation/linux/ubuntulinux/
- Docker UI: http://linoxide.com/linux-how-to/setup-dockerui-web-interface-docker/

Docker Daten-Ablage auf btrfs Partition verlagern

```
root@docker3:/etc/docker# less daemon.json

{
   "storage-driver": "btrfs",
   "data-root":  "/mnt/data/docker"
}
```

Images

Images aus Repository runterladen

`docker pull`

Alle Images im lokalen repository zeigen

```
root@develd:~# docker image ls
REPOSITORY                 TAG                 IMAGE ID       CREATED
SIZE
local_discourse/web_only   latest              21e8a905ef5c   4 weeks ago
2.65GB
grundic/jamulus            latest              9e97d3890ca8   5 weeks ago
90.3MB
discourse/base             2.0.20210217-2235   7027ba787aa6   2 months ago
2.22GB
discourse/base             2.0.20201221-2020   c0704d4ce2b4   4 months ago
2.11GB
local_discourse/data       latest              c7524a566464   5 months ago
2.44GB
discourse/base             2.0.20201004-2310   b64c37d7ab06   6 months ago
2.4GB
xbrowsersync/api           latest              a3554c99cc99   12 months ago
119MB
```

Image löschen

```
docker image rm [id]
```

Image Repo aufräumen und ungenutzte Images löschen

```
docker image prune [OPTIONS]
```

z.B: alle images löschen, die nicht von mindestens einem Container genutzt werden

```
docker image prune -a
```

Container Management

Alle Container auf einmal stoppen

```
    docker kill $(docker ps -q)
    docker rm $(docker ps -a -q)
    docker rmi $(docker images -q)
```

Container aus heruntergeladenem Image erzeugen

`docker create` erzeugt den Container und startet ihn **nicht,** `docker run` macht beides.

Alle laufenden und gestoppten Container zeigen:

`docker ps -a`

Filtern auf laufende Container

```
root@develd:~# docker ps -a -f status=running
CONTAINER ID   IMAGE                      COMMAND        CREATED
STATUS        PORTS                  NAMES
31f9a6ffcab8   grundic/jamulus            "Jamulus"      19 hours ago   Up
19 hours                           jolly_beaver
7c3cd1abf744   local_discourse/web_only   "/sbin/boot"   4 weeks ago    Up 5
days     127.0.0.1:84->80/tcp   web_only
05f7f43d0493   local_discourse/data       "/sbin/boot"   5 months ago   Up 5
days                           data
```

Laufende und gestoppte, aber nur die container id ausgeben

`docker ps -aq`

Container zeigen, die nicht laufen

```
root@develd:~# docker ps -aq -f status=exited
b39916cf84e2
e6e7c809ad34
```

Container starten/stoppen

```
docker start [ID] docker stop [ID]
```

Alle Container stoppen oder löschen

```
docker stop $ (docker ps -a -q) docker rm $(docker ps -a -q)
```

Diese können mit docker rm gelöscht werden

```
root@develd:~# docker rm e6e7c809ad34
e6e7c809ad34
```

Alle Container löschen, die nicht laufen

```
docker container prune
```

Restart policy ändern

```
docker update --restart=no matrix_synapse_1
```

Einen Container betreten

```
docker exec -t -i container_ID /bin/bash
```

Alternativ: stdin/out an den Container hängen:

```
docker attach [OPTIONS] CONTAINER
```

Achtung: wieder raus mit **CTRL-p CTRL-q.** Details siehe https://docs.docker.com/engine/reference/commandline/attach/

Docker Build

https://stackify.com/docker-build-a-beginners-guide-to-building-docker-images/

Geht immer von einem Dockerfile aus: https://docs.docker.com/reference/dockerfile/

```
# Use official Alpine release
FROM alpine:3.19.3 as build

# Maintainer
LABEL maintainer="Thomas Rother <thomas.rother@devoteam.com>"
LABEL version="1.2"
LABEL description="radsec proxy server for remote RADIUS operation with TLS
security"
```

```
ENV RADSECVERSION 1.11.0
ENV RADSECURL
https://github.com/radsecproxy/radsecproxy/releases/download/${RADSECVERSION
}/
ENV RADSECFILENAME radsecproxy-${RADSECVERSION}.tar.gz

# Change working dir
WORKDIR /root

# Update apk
RUN apk update

# Install buildtools
RUN apk add --no-cache make g++ openssl-dev nettle-dev musl-dev

# Create output dir
RUN mkdir output

# Download radsecproxy source files
RUN wget ${RADSECURL}${RADSECFILENAME}

# Untar radsecproxy
RUN tar xf ${RADSECFILENAME} --strip-components=1

# Configure
RUN ./configure --prefix=/root/output --sysconfdir=/etc

# Make and install to output dir
RUN make && make install

# Create radsecproxy container
FROM alpine:3.19.3

# Update apk
RUN apk update

# Install openssl, ca-certificates, nettle and tini
RUN apk add --no-cache openssl ca-certificates bash nettle tini

# Copy from 'build' stage
COPY --from=build /root/output/ /
# COPY --from=build /root/radsecproxy.conf-example
/etc/radsecproxy/radsecproxy.conf

# create config directory and add certs
RUN mkdir /etc/radsecproxy/
COPY radsecproxy/ /etc/radsecproxy/
COPY certs/       /etc/radsecproxy/

# Copy start.sh
COPY start.sh /root/start.sh
```

```
# Make start.sh executeable
RUN chmod u+x /root/start.sh

# Create Radsecproxy logging
RUN mkdir /var/log/radsecproxy

# Export volumes
VOLUME /var/log/radsecproxy

# Make Radsecproxy's ports available
EXPOSE 2083

# Set Tini entrypoint
#
https://computingpost.medium.com/how-to-use-tini-init-system-in-docker-conta
iners-69283d0099ed
ENTRYPOINT ["/sbin/tini", "--"]

    # Start Radsecproxy
    CMD ["/root/start.sh"]
```

Der Build wird normalerweise gecacht, das kann man abschalten:

```
docker build .
```

Dockerfle Syntax check

```
docker build --check .
```

Docker build mit tag:

```
docker build . -t yourusername/example-node-app
```

```
thommie@odysseus3:~/git/gitea/dockerbuilds/myradius> docker build -f
myradius.docker .
Sending build context to Docker daemon  5.632kB
Step 1/3 : FROM freeradius/freeradius-server:latest
 ---> 0a093ead10b6
Step 2/3 : COPY raddb/ /etc/raddb/
 ---> 096f00c66db0
Step 3/3 : EXPOSE 1812-1813/udp
 ---> Running in 6d5f572b0a8b
Removing intermediate container 6d5f572b0a8b
 ---> c6c0d41f944a
Successfully built c6c0d41f944a
```

Docker Compose

Docker Compose erzeugt Docker Applikationen, die aus mehreren Containern bestehen. docker

compose up startet alles im Verbund.

1. Das Dockerfile definiert die Laufzeit-Umgebung
2. docker-compose.yml beschreibt die Services, die in Containern zusammen arbeiten
3. "docker compose up" erzeugt und startet die gesamte Applikation

Achtung: Die python basierten Docker Versionen (V1) in manchen Distro-Repos sind veraltet. Es empfiehlt sich, die V2 aus https://github.com/docker/compose zu benutzen, die in GOLANG neu geschrieben wurde. Installation siehe https://github.com/docker/compose/

Hinweis: Für Raspis wird das Binary für ...-linux-armv7 benutzt (ARM Architektur)

compose yaml validieren

```
docker compose -f docker-compose-pro.yaml config
```

Update per docker compose

Update der Images, die im compose file referenziert sind

```
docker compose -f docker-compose-pro.yaml pull
```

Daraus die Container neu bauen und startenb

```
docker compose -f docker-compose-pro.yaml up --build
```

```
docker compose -f docker-compose-pro.yaml up --force-recreate --build -d
```

Docker Compose startet die Container, aber nicht daemonisiert. Dazu -d hinzufügen:

```
docker compose up -d
```

Üblicherweise nimmt man dafür systemd.

Docker Netzwerke

Standardmässig werden drei Netze bridge, host, none angelegt. Alle anderen sind custom Networks, die z.B: über compose angelegt wurden:

```
root@docker1:~# docker network ls
NETWORK ID     NAME           DRIVER    SCOPE
54e670dd998d   bridge         bridge    local
5b1cd745d5a3   docker_back    bridge    local
b56f108784b3   docker_dbnet   bridge    local
```

```
f3619965eb7b   docker_front   bridge    local
c91196bf89eb   host           host      local
b4f69adafbb3   none           null      local
```

Container werden an die bridge "docker0" auf dem Host gebunden, solange nicht beim docker create ein anderes Netzwerk gewählt wurde (`docker create –network=<NETWORK>`). Mit

```
docker network inspect bridge
```

sieht man den Zustand eines Docker networks

Custom networks

docker network create erzeugt ein eigenes Netzwerk:

```
docker network create –subnet 192.168.82.0/24 –driver bridge bridge2
```

```
locutus:/home/thommie # docker network inspect bridge2 [ { "Name":
"bridge2", "Id":
"9c353bcf0c2c6ccee0b821e1ff4d1740a074bdea94e93959c522d46a4e6fde8e", "Scope":
"local", "Driver": "bridge", "EnableIPv6": false, "IPAM": { "Driver":
"default", "Options": {}, "Config": [ { "Subnet": "192.168.82.0/24" } ] },
"Internal": false, "Containers": {}, "Options": {}, "Labels": {} } ]
```

Mit

```
docker attach container1
```

sieht man das Netzwerk von innen

Docker logs

Analog zu tail -f:

```
docker logs --follow [containerid]
```

Docker volumes

Volumes sind Verzeichnisse/Dtane, die vom lokalen Docker Host in den Container gemappt werden. Volumes werden vom Docker Dämon gemanagt, die Daten liegen innerhalb des vom Docker Dämon verwalteten Speicherbereichs.

Doku: https://docs.docker.com/engine/admin/volumes/

In compose:

```
services:
  frontend:
    image: node:lts
    volumes:
      - myapp:/home/node/app
volumes:
  myapp:
    external: true
```

Syntax: **[lokales volume Verzeichnis Host]:[Verzeichnis im Container]**

Immer relativ zum compose file gesehen!

Begrifflichkeiten

- Master = koordiniert den Cluster über die Kubernetes API - auf dem Master laufen keine Pods
- Node = Maschine, auf der Cluster (Pod) laufen (kann eine oder mehrere phys. Maschine oder VMs sein)
- Pod = einer oder mehrere Container, die gemeinsam Ressourcen nutzen (z.B. gemeinsamer Speicherplatz, gemeinsame IP Adresse, Informationen, wie der Container zu betreiben ist). Pod = Container + gemeinsame Ressourcen (Speicher, RAM, CPU, Netzwerk usw.)
- Service: Funktion, die von einem oder mehreren Pods bereitgestellt wird

Portainer

```
docker run -d -p 8000:8000 -p 9000:9000 --name portainer --restart=always -v
/var/run/docker.sock:/var/run/docker.sock -v portainer_data:/data
portainer/portainer-ce:latest
```