

# Docker

Leon Schloemmer

# Inhalt

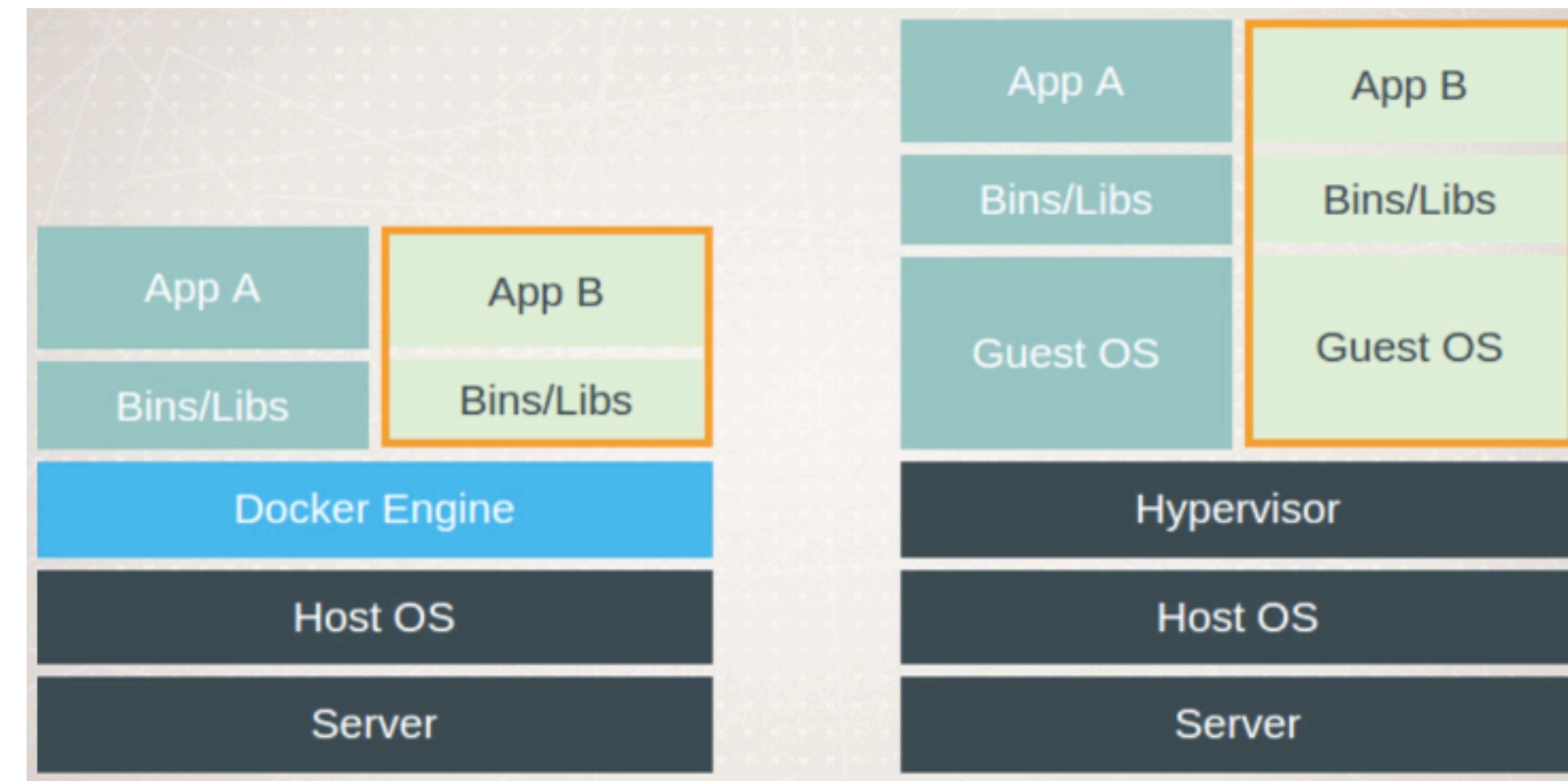
- Warum Docker, was ist ein Container?
- CLI Basics
- Images vs Container
- Containers
- Networks
- Images
- Container lifetime & Persistence
  - Data Volumes
  - Bind Mounts
- Docker Compose

# Warum Docker?

- Docker ist eine Container-Technologie
- Schnell
- Speichereffizient

# Was sind Container?

- Vergleichbar mit VMs
- Haben aber nicht viel mit ihnen gemeinsam
- Containers sind nur Prozesse
- Sie sind limitiert auf Ressourcen, auf die sie zugreifen können / müssen
- Containers stoppen mit ihrem Prozess



Quelle: Bret Fisher; Docker Mastery

# Docker CLI

- `$ docker`
  - liefert eine liste von verfügbaren commands
- Früher: Alle subcommands wurden gezeigt
- Heute: Commands sind unterteilt
  - `docker container`
  - `docker image`
  - `docker network, etc`

- Früher: docker run
- Heute: docker container run
- Früher: docker ps
- Heute: docker container ls
- Warum geht noch beides?  
Für Backwards-Kompatibilität

# Image vs Container

- **Image:** Die Applikation die wir ausführen möchten
- **Container:** Eine Instanz eines Images, welche als Prozess läuft
- Images werden von Registries geladen, das Default Registry ist das **Docker Hub**

# Containers

- Debriefing % docker container run —publish 8080:80 nginx
  - Was ist passiert?
    1. nginx wurde vom Docker Hub gedownloaded
    2. Ein Container des Image wurde gestartet
    3. 8080 wurde auf localhost geöffnet
    4. Traffic wird auf den container gerouted

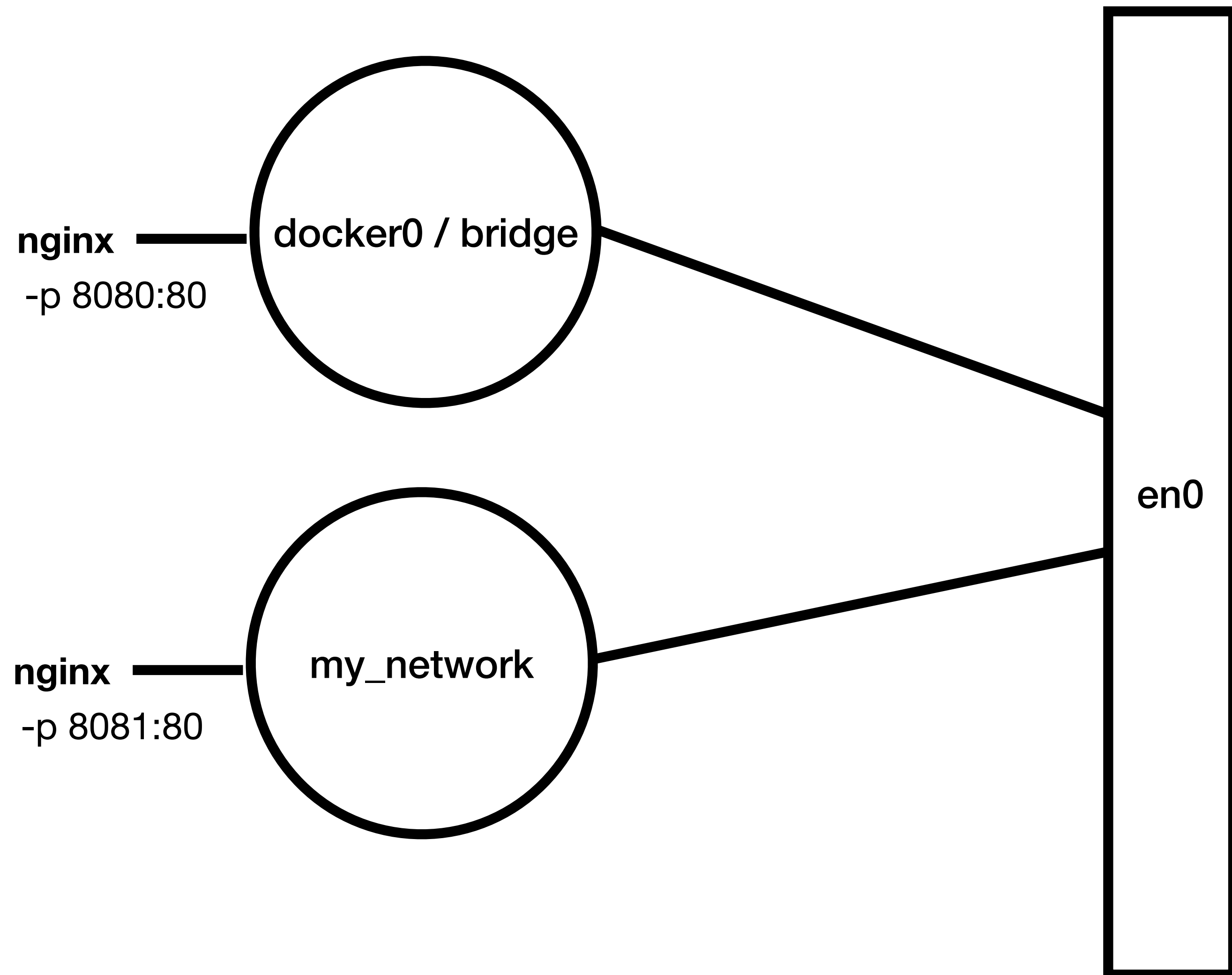


# Terminal in Container

- Beim Starten: ... run -it ... bash
- Bei Laufenden: docker container exec -it [name] bash

# Networks

- isolierte Network-Umgebung für Container
- Können nach außen freigegeben werden
- Default: docker0 (legacy), bzw aktuell: bridge



# Networks

- Container in einem Network können miteinander kommunizieren

# Docker Network DNS

- Wie kann man Container miteinander kommunizieren lassen?
- Docker hat ein eingebautes DNS
- Auf KEINEN Fall statische IPs verwenden!!!
- DNS ist im bridge Netzwerk allerdings deaktiviert

# Images

- Besteht aus den App binaries und den Dependencies
- Enthält Metadata über das Image, und darüber wie das Image auszuführen ist
- Ist kein vollständiges Betriebssystem, der Host stellt den Kernel, Kernel Module, etc bereit
- Ist meist eine sehr kleine Datei, kann aber auch eine sehr große sein

# Docker Hub

- <https://hub.docker.com>
- Beispiel nginx
- Versionen von Images haben **Tags**
  - Jeder Unterpunkt ist eine Version, mit den verschiedenen Tags, die zu dieser Version gehören
  - `docker pull nginx:mainline == docker pull nginx:latest`
- Official Images, sind die einzigen, deren Namen nicht den Username des Veröfentlichers enthalten müssen

# Alle Official Images und deren Dockerfiles

- [https://hub.docker.com/search?q=&type=image&image\\_filter=official](https://hub.docker.com/search?q=&type=image&image_filter=official)
- <https://github.com/docker-library/official-images/tree/master/library>

# Image Layers

- Images sind in layers aufgebaut, Images bauen auf Images
- Manche Images haben gemeinsame base-Images
  - Der Vorteil von Docker ist: 1 Image wird im System nur 1 mal gespeichert
- Verändert ein Container sein Image, werden die Veränderungen in den Container kopiert == **Copy on Write**



# Images referenzieren

- % docker pull [image]
- image kann sein
  - Repository (username/repo)
  - Tag (Zeiger auf einen speziellen Commit)
  - ID
- Wird das gleiche image 2 mal unter verschiedenen tags gepulled, scheint es unter % *docker image ls* auch 2 mal auf, aber mit der selben Image ID

# Dockerfiles

- Sind Dateien, die beschreiben wie Images zu bauen sind
- Jedes Command ist ein **Layer** im Dockerfile
- Beim Builden geht Docker Schrittweise vor, jeder Layer ist ein Step
- Jeder Step erhält ein Hash
- Hat sich ein Step, und die Steps darüber beim erneuten BUILD nicht verändert, baut Docker diesen Step nicht noch einmal, sondern verwendet den alten Step aus dem Cache

# Typische Kommandos

- FROM - *required*
- ENV
- RUN
- EXPOSE
- WORKDIR
- COPY
- CMD - *required*

# RUN

- Shell commands im Image ausführen
- wenn mehrer commands hintereinander ausgeführt werden, hängt man sie mit \ und && zusammen um:
  - Anzahl der Layer niedrig zu halten -> Zeit und Platz sparen

```
15  RUN apt-get update \
16      && apt-get install --no-install-recommends --no-install-suggests -y gnupg1 \
17      && \
18      NGINX_GPGKEY=573BFD6B3D8FBC641079A6ABABF5BD827BD9BF62; \
19      found=''; \
20      for server in \
21          ha.pool.sks-keyservers.net \
22          hkp://keyserver.ubuntu.com:80 \
23          hkp://p80.pool.sks-keyservers.net:80 \
24          pgp.mit.edu \
25      ; do \
26          echo "Fetching GPG key $NGINX_GPGKEY from $server"; \
27          apt-key adv --keyserver "$server" --keyserver-options timeout=10 --recv-keys "$NGINX_GPGKEY" && found=yes && br
28      done; \
29      test -z "$found" && echo >&2 "error: failed to fetch GPG key $NGINX_GPGKEY" && exit 1; \
30      apt-get remove --purge -y gnupg1 && apt-get -y --purge autoremove && rm -rf /var/lib/apt/lists/* \
31      && echo "deb http://nginx.org/packages/mainline/debian/ stretch nginx" >> /etc/apt/sources.list \
32      && apt-get update \
33      && apt-get install --no-install-recommends --no-install-suggests -y \
34          nginx=${NGINX_VERSION} \
35          nginx-module-xslt=${NGINX_VERSION} \
36          nginx-module-geoip=${NGINX_VERSION} \
37          nginx-module-image-filter=${NGINX_VERSION} \
38          nginx-module-njs=${NJS_VERSION} \
39          gettext-base \
40      && rm -rf /var/lib/apt/lists/*
```

# Tipp!

- Zeilen die sich im Dockerfile wenig verändern, sollten eher oben sein,  
Zeilen die sich viel verändern, weiter unten

# Container Lifetime & Persistence

- Key Concepts:
  - **immutable**
  - **ephemeral**
- = “unverändernd” und temporär, verworfbar
- Man spricht von einer ***immutable infrastructure***
  - = bestehende Container nicht verändern, sondern immer wieder redeployen

# Was ist mit persistenten Daten?

- Idealerweise sollten persistente Daten von der Applikation getrennt werden, auch bei Datenbanken!
  - = **Separation of concerns**
- Persistente Daten = ***Unique Data***
- So kann man die Applikation updaten (den Container recyceln), und *unique Data* ist noch vorhanden



# Data Volumes

- Dockerfile Command: VOLUME /path/in/container
- Vom Container aus kann man auf dieses Directory normal zugreifen
- Daten darin werden allerdings nicht im Container gespeichert
- Volumes haben einen **längeren Lebenszyklus** als Container

# Named Volumes

- `docker volume ls`
  - unleserlich!
- Wenn man volumes einen Namen geben möchte
- `% docker container run ... -v [volume_name]:/path/to/dir`

# Bind Mounts

- Sind nicht für Production gedacht
- Mappen ein directory am Host zu einem Directory im Container

# Docker Compose

- Ist eine Kombination aus CLI und Konfigurationsfile
- Verwalten von Beziehungen zwischen Containern
- Speichern von *% docker run ... -d...* in einem File
- Ganze Development Environments mit nur einem Befehl starten

# docker-compose.yml

- **YAML** files werden gemeinsam mit dem **docker-compose** command benutzt
- Sind für **lokale Automation** gedacht - **nicht für Production Environments!!!**
- In Docker Swarm kann man sie allerdings sogar für production environments benutzen
- YAML files ersetzen shell scripts welche *docker run* commands automatisieren

# YAML Versionen

- 3.1
  - 2
  - 1
- 
- Wir verwenden 3.1
  - Wird im File ganz oben angegeben: ***version '3.1'***

# docker-compose CLI

- Auf Linux muss es separat installiert werden
- Kein Tool für Produktionsumgebungen!
- Ideal für lokale Entwicklung
- Commands:
  - **docker-compose up**
  - **docker-compose down**
- docker-compose erstellt sogar networks von selbst!

**Vielen Dank für Ihre Aufmerksamkeit**