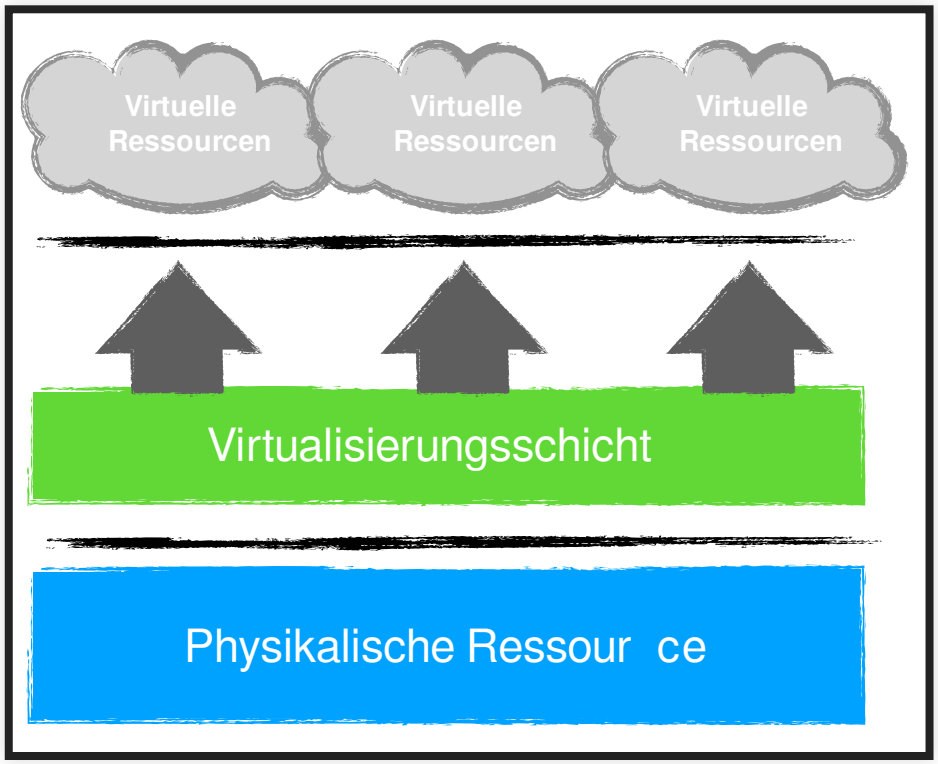
KAPITEL 4: VIRTUALISIERUNG VS. CONTAINER

LERNZIELE

- Virtualisierung und Container vergleichen
- Das Docker Command Line Interface verwenden
- Existierende Dockerfiles verstehen und erklären
- Ein einfaches Dockerfile erstellen und die Grundbestandteile erklären
- Konfigurationen mittels Docker-Compose erklären

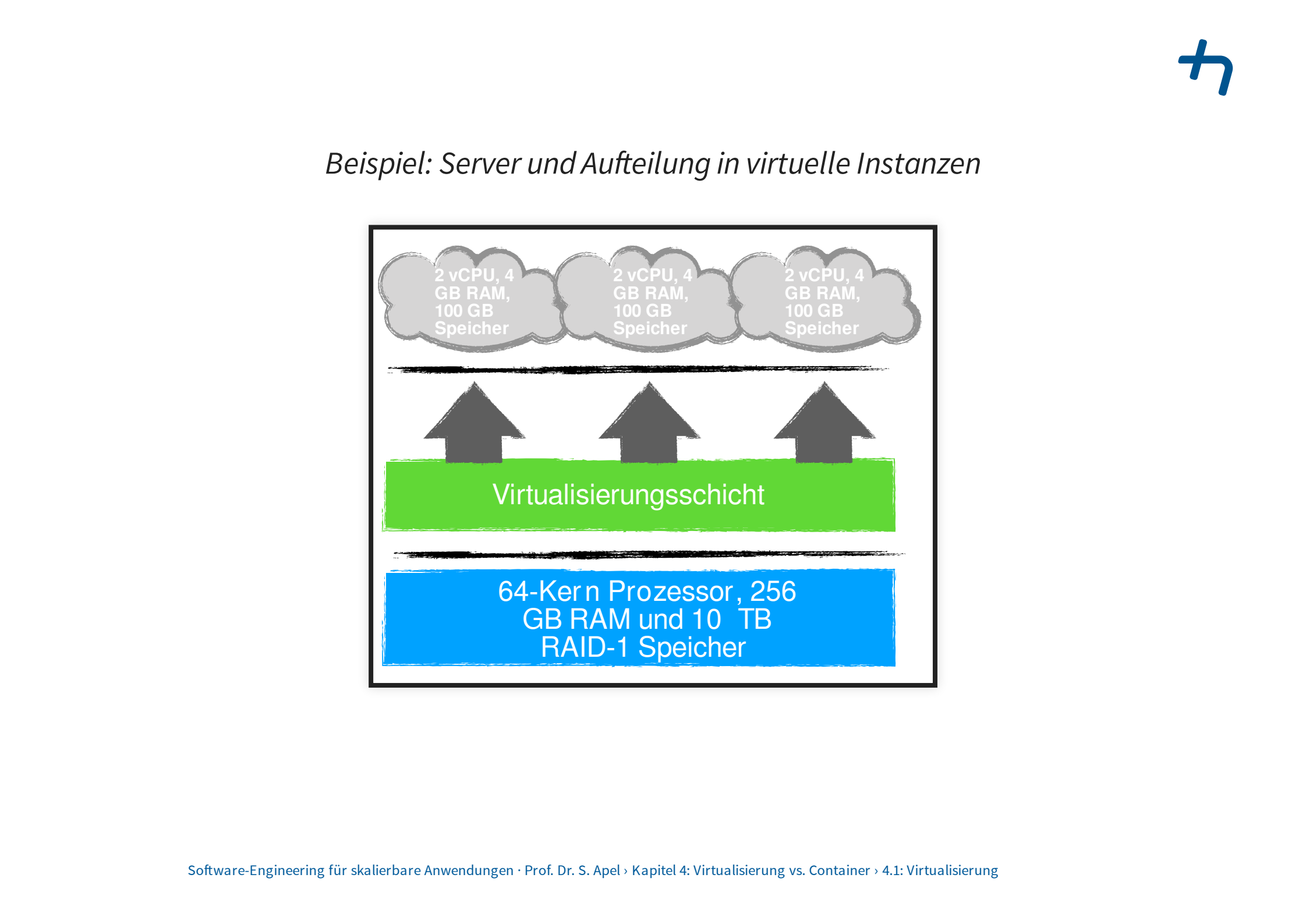
4.1 VIRTUALISIERUNG

Wie können verschiedene Anwendungen auf einer physischen Hardware isoliert betrieben werden?



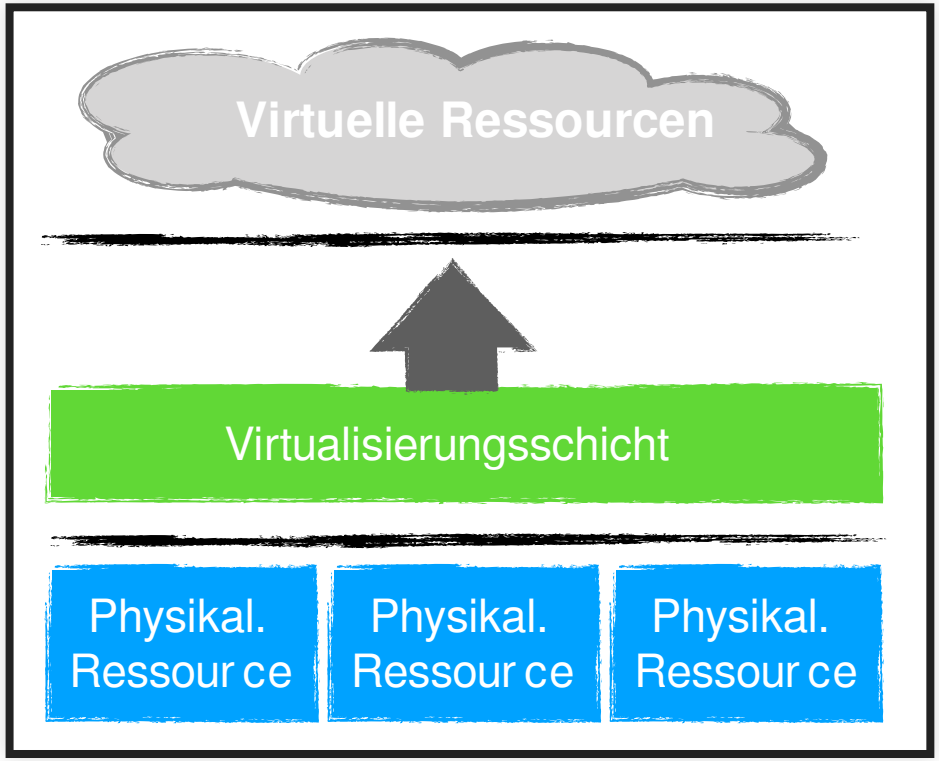
Häufig stehen verfügbare Hardware und auszuführenden Anwendungen nicht im selben Verhältnis, zusätzlich ist es ungünstig, wenn Fehler bei einer Anwendung andere Anwendungen in verteilten Architekturen kompromitieren.

Virtualisierung bietet hier die Möglichkeit verfügbare Hardware aufzuteilen, zu begrenzen und Anwendungen zur Verfügung zu stellen.

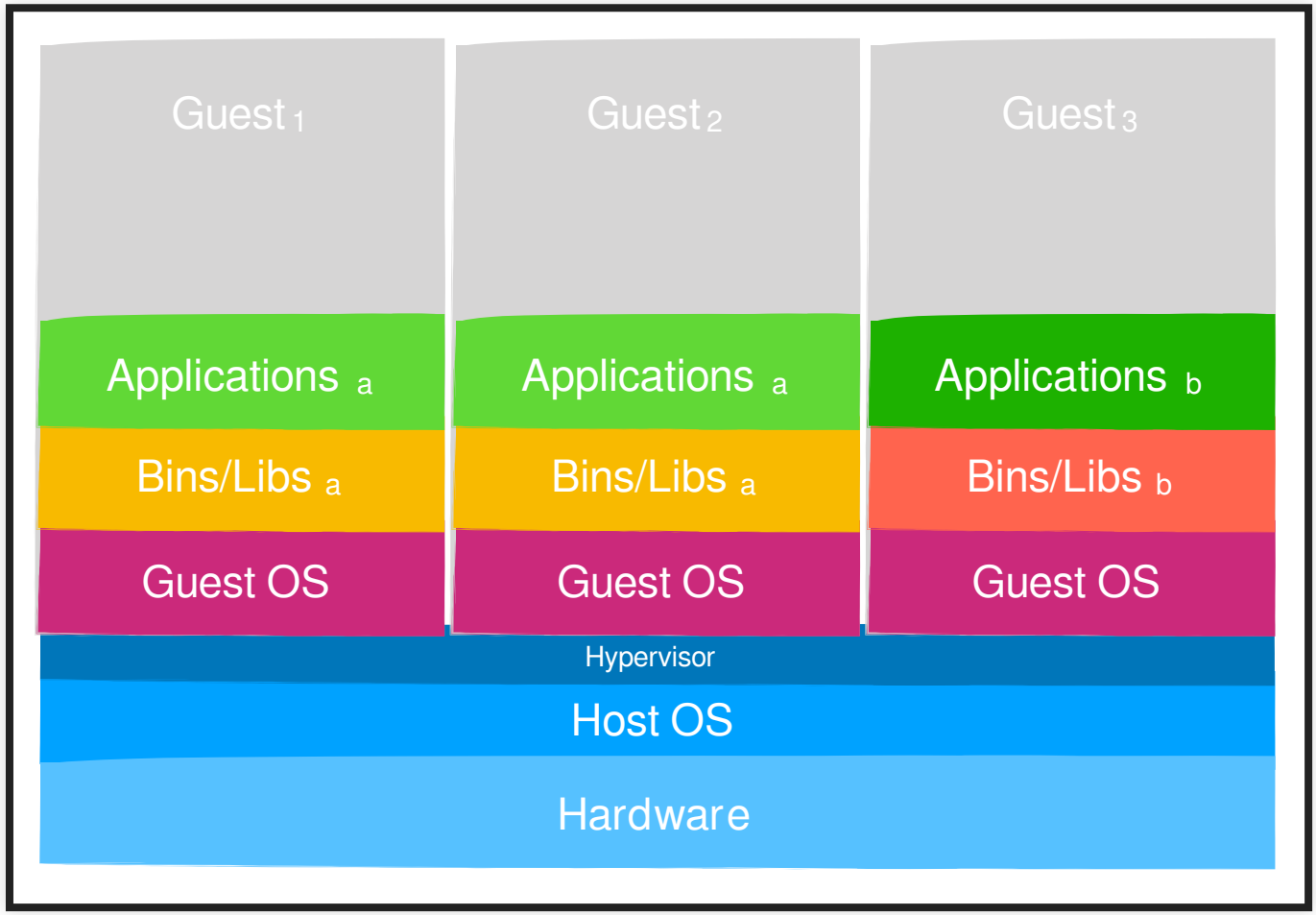


Beispiel: Server und Aufteilung in virtuelle Instanzen

Virtualisierung kann nicht nur für Aufteilung, sondern ebenso für die Schaffung größerer virtueller Ressourcen dienen. Diese wären dann größer als es die physikalischen Ressourcen ermöglichen würden.

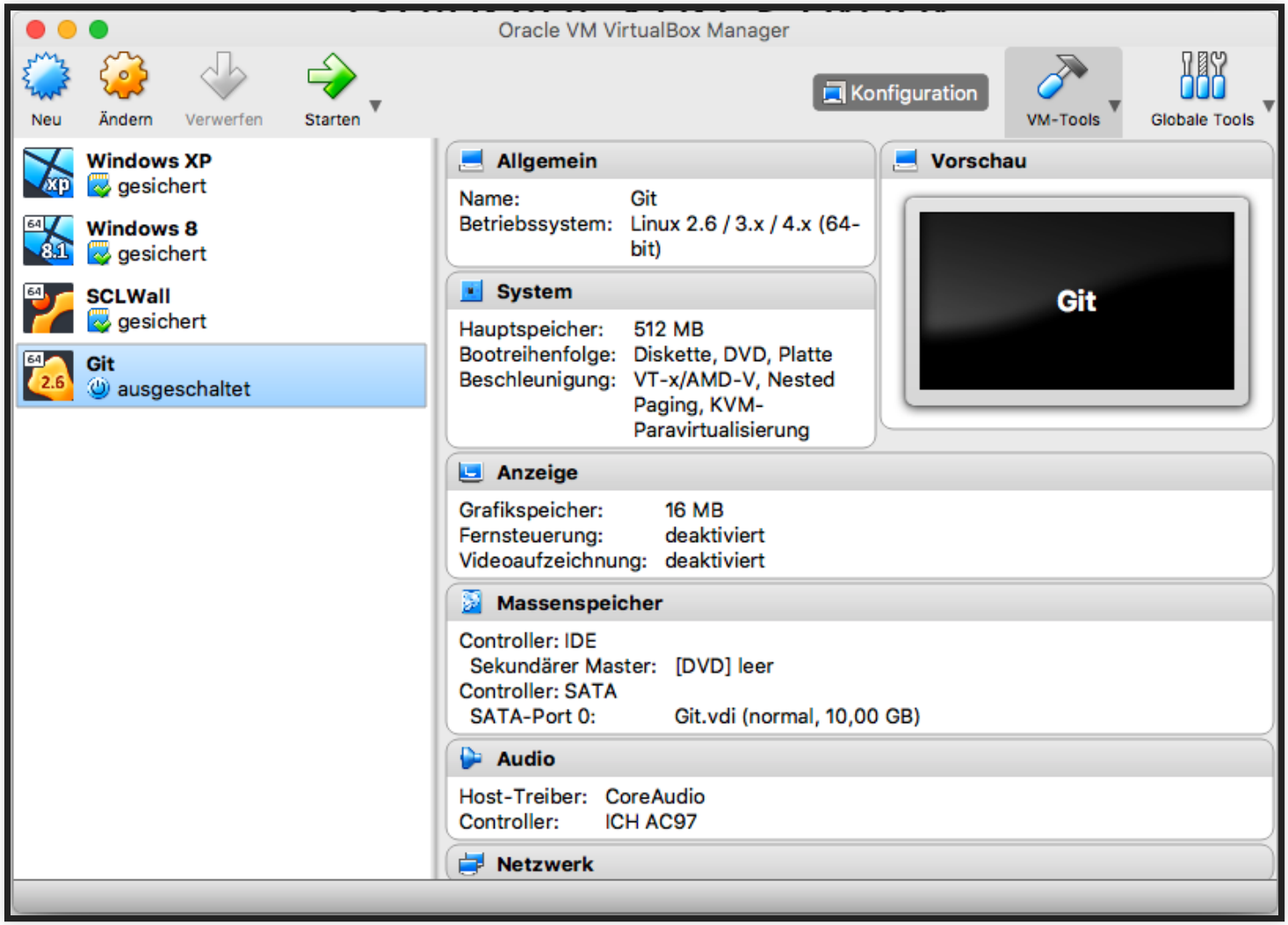


Ressourcenmanagement



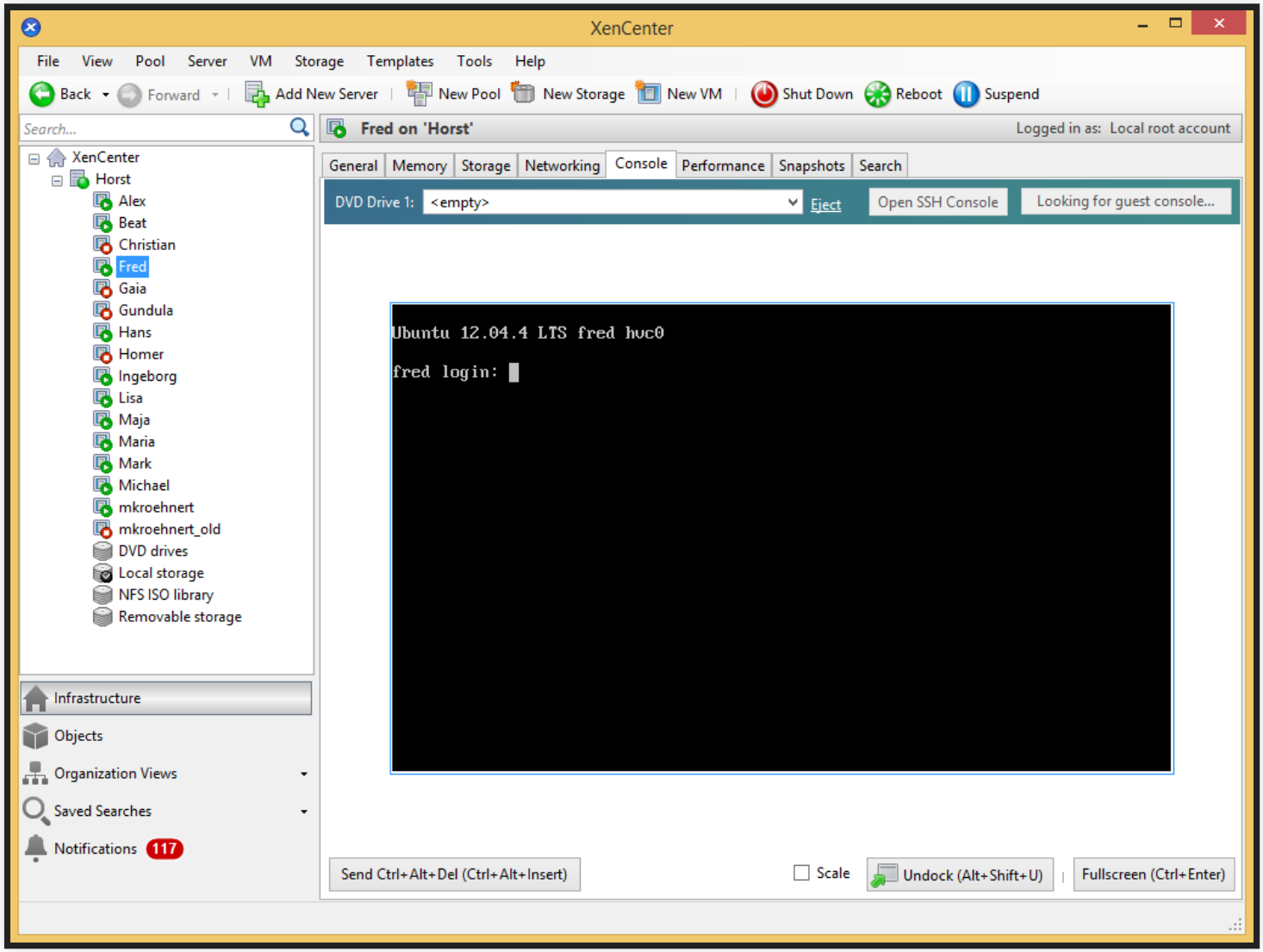
- Auf der physischen Hardware - dem sogenannten Host - wird ein geeignetes Betriebssystem ausgeführt
- Mithilfe eines Hypervisors (eine spezielle Anwendung) werden die Ressourcen verwaltet und virtuellen Instanzen zur Verfügung gestellt
- Im Fall von virtuellen Maschinen - den Guests - entstehen damit Instanzen, die selbst wieder ein eigenes Betriebs installieren und die bereitgestellte virtuelle Hardware nutzen
- Aus der Perspektive der Guests sieht es so aus, als wenn nicht mehr zur Verfügung steht
- Der Host managed im Hintergrund die verfügbaren Ressourcen
- Eine Übernutzung der eigentlichen Kapazitäten ist möglich (Resource Pools)

Beispiel Oracle VirtualBox



- Virtualisierungssoftware von Oracle
- Ausführbar in vielen Betriebssystemen
- Ermöglicht die Erstellung von virtuellen Maschinen mit konfigurierbaren Ressourcen und eigenem Betriebssystem
    - u.a. RAM, vCPU, Festplatte und Netzwerk

Beispiel XenServer / XenCenter



- Virtualisierungssoftware Xen in einem CentOS ausgeliefert als Komplettpaket XenServer
- Wird als Betriebssystem zur Virtualisierung installiert
- Linux-Distribution auf den Betrieb von virtuellen Systemen optimiert
- Verwaltung der Guests mittels Desktop Anwendung auf dem Arbeitsrechner (Remote Zugriff)

BEISPIEL: DEBIAN IN VIRTUALBOX

ÜBUNG: VIRTUELLE MASCHINE INSTALLIEREN

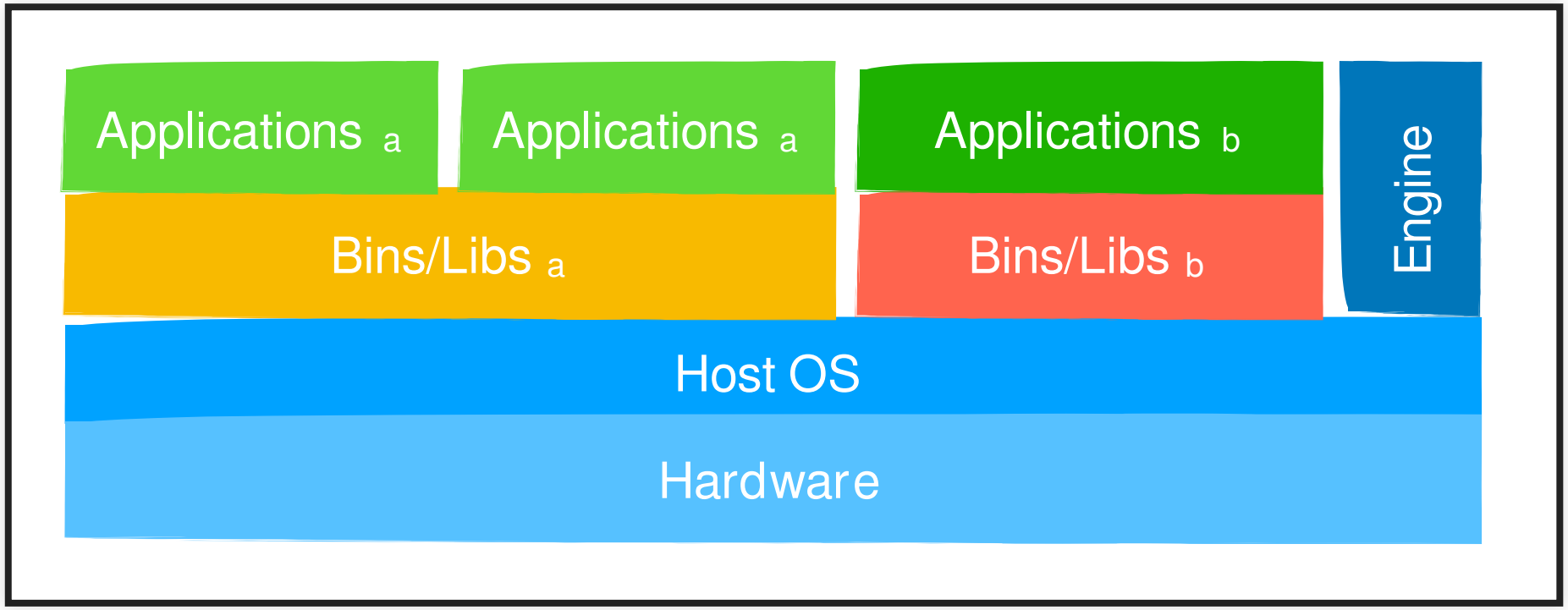
Installieren Sie VirtualBox und folgen Sie anschließend dem Beispiel, um eine eigene virtuelle Maschine zu installieren. Sie können hierfür ein Debian Linux nutzen, aber auch auf ein anderes ausweichen.

Versuchen Sie anschließend in dieser virtuellen Maschine mindestens ein Beispiel aus dem Kapitel Architekturen auszuführen (z.B. REST mit Spring).

Hinweise

- Sie müssen den Quellcode des Beispiels in die VM bekommen
- Sie können einen Ordner im Host-System im Guest als Mount hinzufügen
- Alternativ könnten Sie Git installieren und das Projekt aus einem Remote-Repository auschecken
- Anschließend benötigen Sie geeignete Werkzeuge zur Ausführung der Anwendung, z.B. Java
- In einem Debian kann mittels `sudo apt install default-jdk` Java installiert werden
- Weitere Abhängigkeiten im Beispiel währen Maven: `sudo apt install maven`
- Beispiel-Anwendung starten, z.B. mit `mvn clean compile spring-boot:run`
- Port-Forward im Host konfigurieren, damit Port `8080` verfügbar wird

4.2 CONTAINER



- Beschreibt ein Virtualisierungskonzept auf Betriebssystem-Ebene
- Kernel wird von allen laufenden Instanzen geteilt
- Ressourcen werden den einzelnen Anwendungen zugeordnet

- Container verwenden normale Systemaufrufe und werden nicht emuliert, eine Virtualisierung des gesamten Systems entfällt
- Erstmals vorgestellt von Gaurav Banga
- Verbreitete Implementierung ist **LXC**:
  - userspace-Schnittstelle für die Container-Funktionen des Linux Kernels
  - namespaces für die Prozessisolation
  - Apparmor und SELinux für die Rechteverteilung für den Zugriff auf Ressourcen
  - Verschieben von Wurzelverzeichnissen mittels chroot und Ressourcenbegrenzung mittels cgroups
- Weitere Begriffe: Open Container Initiative, Container Runtime Interface, runC, Docker, Kubernetes

OPEN CONTAINER INITIATIVE

Die Open Container Initiative ist eine offene Governance-Struktur mit dem ausdrücklichen Ziel, offene Industriestandards für Container-Formate und -Laufzeiten zu schaffen.

Quelle: https://opencontainers.org/

- OCI wurde im Juni 2015 von Docker und anderen Unternehmen der Container-Branche gegründet
- enthält derzeit drei Spezifikationen:
    - die Runtime Specification (Runtime-spec, beschreibt, wie ein auf der Festplatte entpacktes "Dateisystembündel" ausgeführt werden kann)
    - die Image Specification (Image-spec, beschreibt, wie ein OCI-Image heruntergeladen und dieses Image dann in ein OCI-Runtime-Dateisystembündel entpackt werden kann)
    - die Distribution Specification (Distribution-spec, beschreibt, wie ein OCI-Runtime-Dateisystembündel ausgeführt werden kann).
- runC ist ein OCI-konformes Werkzeug zum Erzeugen und Ausführen von Containern entsprechend der OCI-Spezifikation

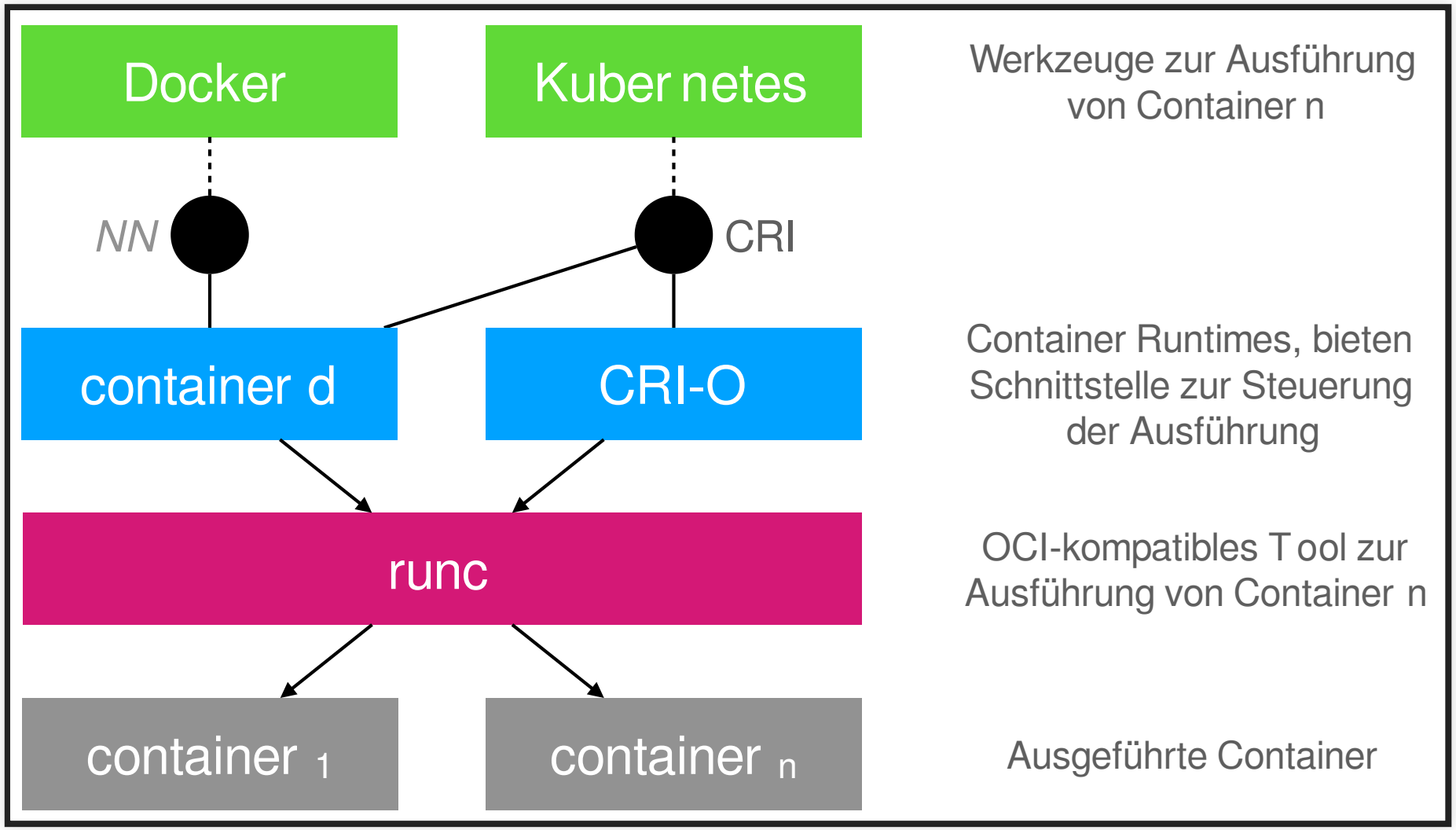
Quelle: https://opencontainers.org/

CONTAINER RUNTIME INTERFACE

- Die CRI ist eine Plugin-Schnittstelle, die es Kubernetes ermöglicht, eine Vielzahl von Container-Laufzeiten zu verwenden, ohne Kubernetes modifizieren zu müssen.
- Notwendig ist eine CRI kompatible Laufzeitumgebung, damit ein Orchestrierungswerkzeug wie Kubernetes auf jedem Knoten im Cluster Container starten kann.
- Definiert das mittels gRPC-Protokoll die notwendigen Funktionen für die Kommunikation zwischen dem Orchestrierungswerkzeug und Container Runtime (erzeugen, starten, stoppen, entfernen, usw.)

Quelle: https://kubernetes.io/docs/concepts/architecture/cri/

ÜBERSICHT


Docker
Kuber netes
NN
CRI
container d
CRI-O
runc
container 1
container n
Werkzeuge zur Ausführung von Container n
Container Runtimes, bieten Schnittstelle zur Steuerung der Ausführung
OCI-kompatibles T ool zur Ausführung von Container n
Ausgeführte Container

4.3 LINUX CONTAINERS

Virtualisierung auf Betriebssystemebene kann über die sogenannten **L**inu**x** **C**ontainer (LXC) realisiert werden.

Hierbei kommt keine Virtualisierungsschicht zum Einsatz, es werden in einer virtuellen Umgebung Anwendungen in Prozessen im Host-System ausgeführt und gemeinschaftlich der Kernel des Host-Systems verwendet.

```
Local ~$ lxc-create -t download -n my-container
Downloading the image index
---
DIST    RELEASE    ARCH    VARIANT    BUILD
---
alpine    3.6    amd64    default    20190515_13:00
alpine    3.6    arm64    default    20190515_13:01
alpine    3.6    armhf    default    20190515_13:00
alpine    3.6    i386    default    20190515_13:00
---

Distribution:
alpine
...
```

- Startet einen neuen Container
- Auswahl der zu verwendenden Basis-Distribution für den Container

```
...
Release:
3.9
Architecture:
i386

Downloading the image index
Downloading the rootfs
Downloading the metadata
The image cache is now ready
Unpacking the rootfs

---
You just created an Alpinelinux 3.9 x86 (20190515_13:07) container.
```

- Auswahl des Releases der Basis-Distribution für den Container
- Auswahl der Architektur der Basis-Distribution für den Container
- Download und Vorbereitung
- Container vorbereitet

```
Local ~$ lxc-start -n my-container -d # start as daemon
```

Start des Containers

```
Local ~$ lxc-info -n my-container
Name:           my-container
State:          RUNNING
PID:            1607
IP:             10.0.3.98
Memory use:     1.61 MiB
KMem use:       1.19 MiB
Link:           vethKTI7DN
 TX bytes:      1.69 KiB
 RX bytes:      2.43 KiB
 Total bytes:   4.12 KiB
Local ~$ lxc-attach -n my-container
/ # apk add python2 # install something
# ...
```

Prüfen, dass dieser läuft

Zusammenfassung

- Erstellen von Containern auf Basis von Templates mit `lxc-create -t download -n my-container`
- Starten mit `lxc-start -n my-container -d`
- Stoppen mit `lxc-stop -n my-container`
- Attach Shell um im Container zu arbeiten mit `lxc-attach -n my-container`

- Container ermöglichen die Ergänzung von zusätzlichen Abhängigkeiten / Bibliotheken - z. B. lassen sich somit Ausführungsumgebung in unterschiedlichen Versionen unkompliziert parallel betreiben
- Erstellungsprozess von Containern kann standardisiert werden und somit im Idealfall der gesamte Entwicklungs-, Test- und Produktionsprozess auf einer einheitlichen Basis ausgeführt werden
- Sämtliche Inhalte des Dateisystems des Containers finden Sie unter `/var/lib/lxc/my-container/rootfs/` für `my-container`

ÜBUNG: CONTAINER MIT LXC STARTEN

Starten Sie einen einfachen Container mit LXC

Je nach Betriebssystem gibt es verschiedene Anleitungen, um einen Container mittels LXC auszuführen. Wenn Sie aus dem Abschnitt zur virtuellen Maschine einen Debian Guest (10 oder 11) installiert haben, können Sie die LXC-Abhängigkeiten nachinstallieren und direkt nutzen:

```
Local ~$ apt-get install lxc
```

Folgen Sie anschließend den vorangegangenen Schritten, um einen Container mit Debian oder Alpine (amd64) als Basis zu installieren. Abschließend können Sie erneut versuchen, eine Anwendung in dem Container zu Ausführung zu bringen, z.B. das Beispiel REST mit Spring.

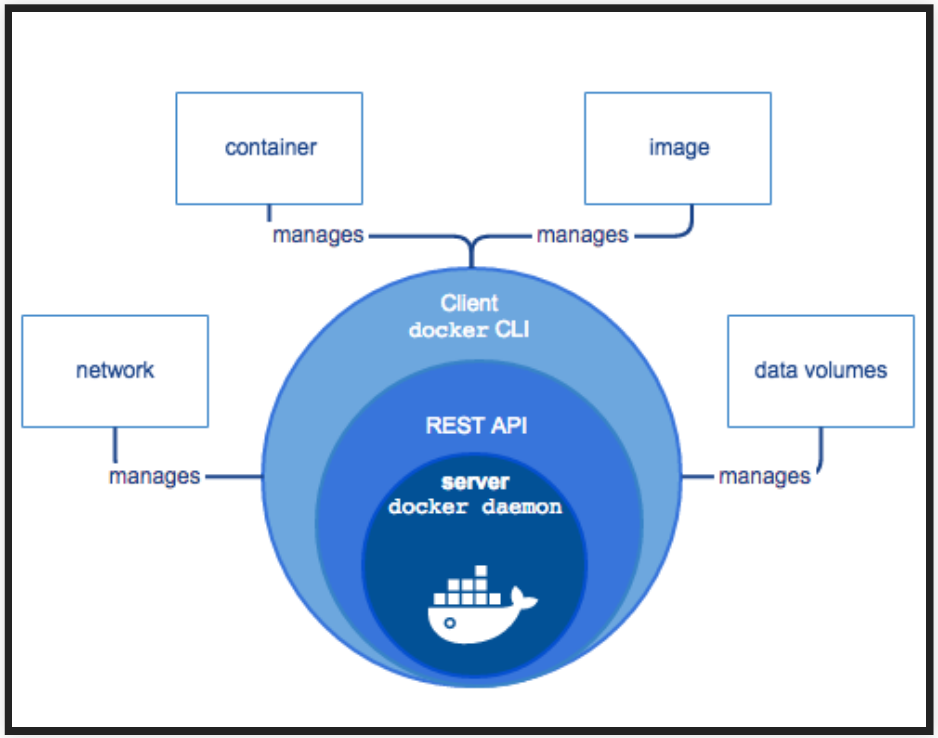
Hinweis: Sie müssen hier vom Grundkonzept genauso vorgehen, wie im Fall der virtuellen Maschine.

Was hat sich geändert? Welchen Einfluss haben die Installationen der Abhängigkeiten auf den Host?

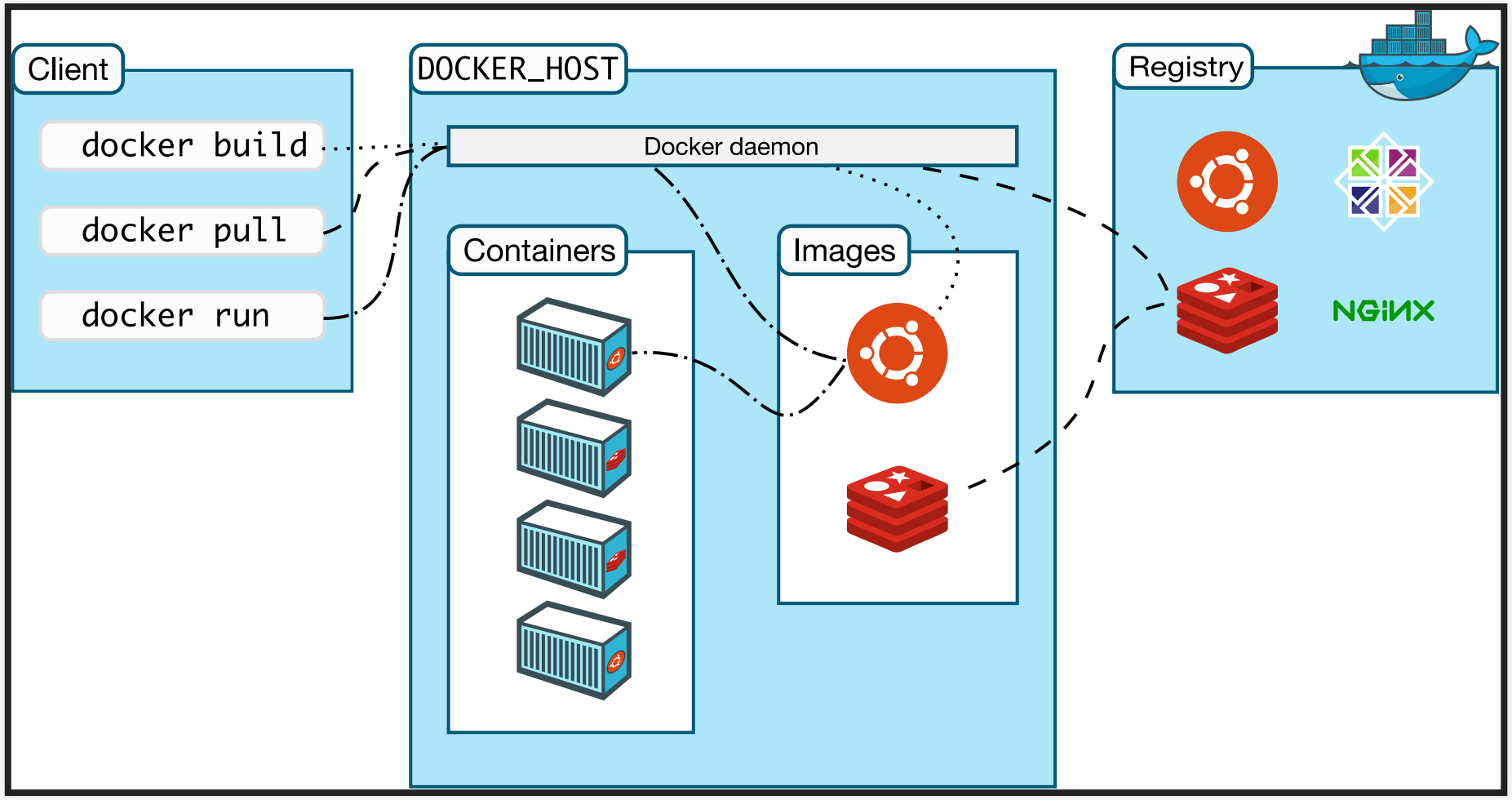
4.4 DOCKER

- Docker basierte auf Linux Containern und erlaubt die Bündelung von Software und ihren Abhängigkeiten sowie deren Ausführung auf beliebigen (Linux-) Systemen
- Docker verwendet heute eine eigene Implementierung (libcontainer), die im Kern aber ebenso auf cgroups und namespaces des Kernels setzt
- Entwickelt von der Firma dotCloud (seit Oktober 2013 Docker Inc.), seit März 2013 unter der Apache Lizenz 2.0 publiziert
- Docker bietet eine API und zentrale Verwaltung (Registry) der Container-Instanzen, ein Containerformat (Images) sowie ein Beschreibungsformat (Dockerfile) zur Erstellung von Images

- **Images**: Vergleichbar mit einem unveränderlichem Snapshot eines Containers. Images sind können als Container beliebig oft instanziiert, transportiert und zentral veröffentlicht werden. Identifiziert sich über einen Nutzer, der es bereitstellt, einen Namen und einen Tag: `someone/something:v1.0`
- **Container**: Eine Instanz eines Images, verändert das Image selbst nicht und hat eine eindeutige ID in der Ausführungsumgebung, sowie einen Namen (ggf. zufällig)



- Isolation von Dateisystem, Ressourcen und Netzwerk
- Copy-On-Write
- Protokollierung
- Container Schichten
- Interaktive Shell und Verwaltung für Container


Client
docker build
docker pull
docker run
DOCKER_HOST
Docker daemon
Containers
Images
Registry
NGINX

- Der **Daemon** verwaltet Images und Container auf dem Host, damit ein Container gestartet werden kann, muss das Image lokal verfügbar sein - existiert es nicht, muss es vorher geladen werden
- Die **Registry** stellt Images bereit, bzw. erlaubt das Bereitstellen von neuen Images - die Default Registry ist der sogenannte Docker Hub
- Im Client läuft die **Docker CLI**, diese kommuniziert mit dem Daemon und erlaubt das Starten, Stoppen und Verwalten von Containern und Images im Host - dieser muss nicht zwingend auf dem Host ausgeführt werden

IMAGE NAMING

`registry-1.docker.io/myuser/myimage:v1.0`

- **Namen von Images sind Slash-separierte Komponenten**, zum Beispiel `myuser/myimage`
- Optional kann der Name einen Hostname der bereitstellenden Registry voran haben, der Default vom sogenannten Docker Hub ist `registry-1.docker.io`
- Optional kann der Domain ein Port folgen
- Nach den Namen-Komponenten kann optional ein Doppelpunkt, gefolgt von einem Tag angegeben werden, zum Beispiel `:v1.0`
- In der Praxis wird auf Dockerhub die erste Namen-Komponente für den Account oder die Organisation verwendet, gefolgt vom Namen des Projektes

Hinweis: Einige verifizierte Container benötigen keinen Account- oder Organisationsprefix

Image ID

- Unabhängig vom Namen hat jedes Image eine eindeutige ID.
- Diese ergibt sich aus dem Bauprozess des Containers und ist ein Hash

Architectures

- Docker Images werden für eine bestimmte System Architekturen bereitgestellt (z.B. arm64v8, amd64, i386)
- Docker verwendet Images passend für die aktuelle System Architektur, was häufig `amd64` Architektur entspricht
- Zu berücksichtigen, wenn z.B. Docker Images in `amd64`-Architekturen gebaut werden und später auf einem Raspberry Pi genutzt werden soll, der eine `arm`-Architektur verwendet

DOCKER CLI

Run

```
Local ~$ docker run alpine echo "Hello World"
Unable to find image 'alpine:latest' locally
latest: Pulling from library/alpine
ff3a5c916c92: Pull complete
Digest: sha256:7df6db5aa61ae9480f52f0b3a06a140ab98d427f...
Status: Downloaded newer image for alpine:latest
Hello World
```

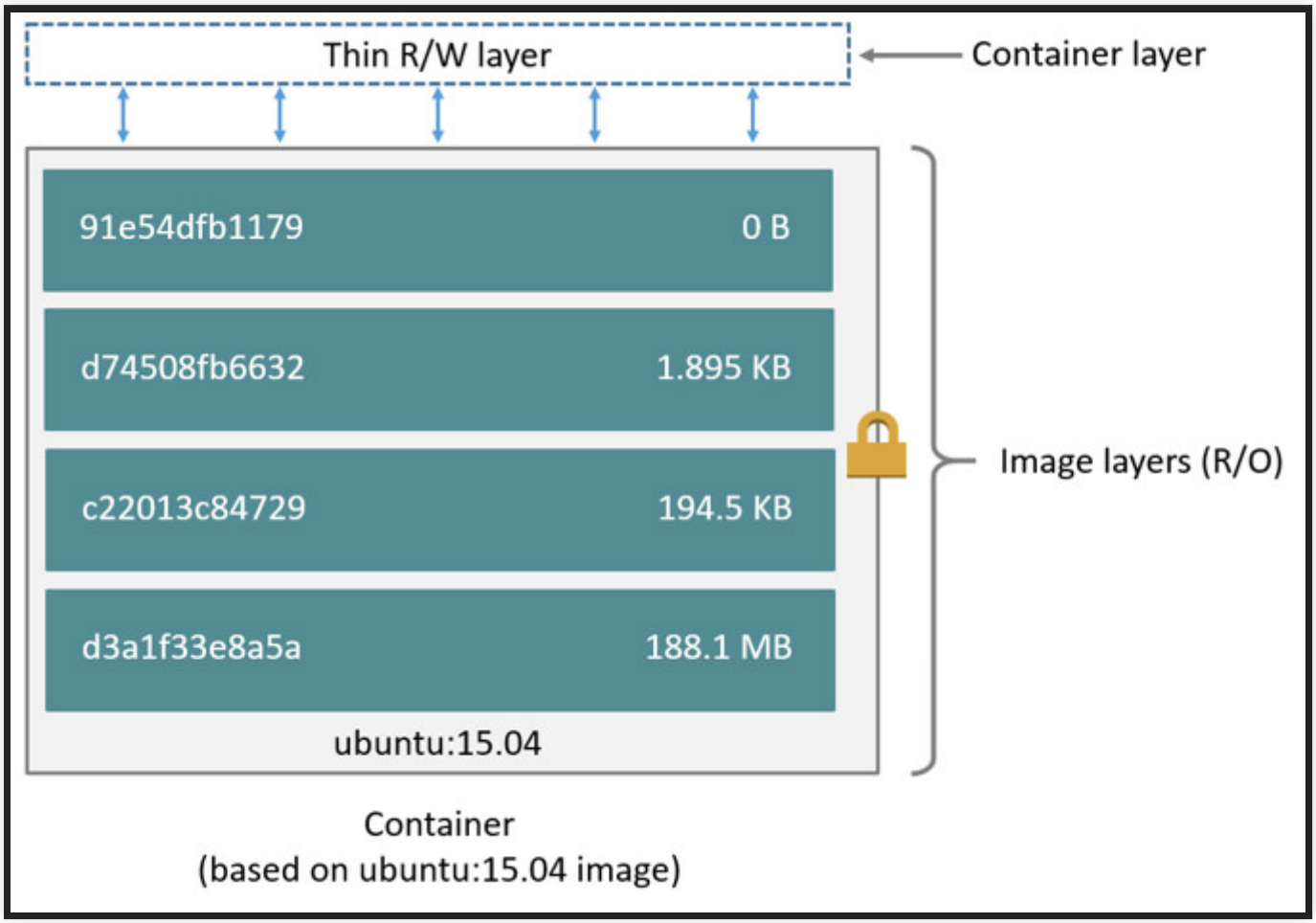
- Einfaches Beispiel mithilfe des Docker CLI zum starten eines Alpine basierten Containers
- Dabei wird echo "Hello World" im Container ausgeführt ...
- ... und in der Konsole zu sehen
- Da das Image auf dem Host nicht existierte, wird dieser hier heruntergeladen
- Mit dem Start erhält der Container eine eindeutige ID und (wenn nicht angegeben) einen Namen

Anstatt `echo` kann jede andere Anwendung, die in dem Container installiert ist, ausgeführt werden

```
Local ~$ docker run -it alpine /bin/sh
/ # echo "Hello World"
Hello World
/ # exit
Local ~$ ...
```

Zum Beispiel die Shell `/bin/sh`, damit Docker die interaktiven Eingaben weiterreicht, muss der Parameter `-it` ergänzt werden.

- Der Start eines Containers stellt das root-Dateisystem nicht wie gewöhnlichen in den Lese-/Schreibmodus um
- Es wird eine neue, beschreibbare Schicht mittels Union Mount hinzugefügt
- Diese Layer liegen auf dem Basisabbild eines Containers oder auf einer vorherigen Schichten
- Docker nutzte bis 18.06 hierfür die Copy-On-Write Funktion von `aufs` (advanced multi layered unification filesystem), aktuell ist `overlay2` der bevorzugte *Storage Driver*
- Ein Docker Image besteht somit aus den Änderungen am Dateisystem zum Eltern-Image und einer spezifischen Konfiguration mit Referenz auf das übergeordnete Abbild.



Quelle: https://docs.docker.com/storage/storagedriver/

Diff

```
Local ~$ docker run -it alpine /bin/sh
/ # echo "Hello World" > test
/ # exit
Local ~$ docker diff focused_shaw
C /root
A /root/.ash_history
A /test
```

- Neuer Container mit Shell im interaktiven Modus
- Eine Veränderung im Dateisystem
- Container beenden
- Anzeige der Änderungen mittels `docker diff` (name kann mittels `docker ps -a` gefunden werden)
- Durch die Schichten entstehen wie in einem Source Control Management Systemen eine Historie über die Dateisystem-Schichten

Inspect

```
Local ~$ docker inspect fervent_khayyam
[
    {
        "Id": "6bd467674315508cab5044e3a5f4bc68f21970913bb6ed3842628...",
        "Created": "2021-09-06T08:22:13.569066037Z",
        "Path": "/bin/sh",
        "Args": [],
        "State": {
            "Status": "exited",
            "Running": false,
            "Paused": false,
...
    }
]
```

Erstellte Container können inspiziert werden mit `docker inspect <name|id>`

Auflisten

```
Local ~$ docker ps --format 'table {{.Names}}\t{{.Image}}'
NAMES               IMAGE
focused_shaw         alpine
```

- Alle aktuell in Ausführung befindlichen Container können mittels `docker ps` angezeigt werden.
- Die Ausgabe kann angepasst werden, hier wurde auf die Ausgabe des Container-Namen und des Images reduziert.
- Wird `-a` ergänzt, können auch alle nicht mehr laufenden Container eingesehen werden

Commit

```
Local ~$ docker commit focused_shaw
sha256:c7df0d64f2a0499c5eb78ec31345599ceacaeede3dab181f666d6a605ef25444
```

Mittels `docker commit <name|id>` kann der Zustand eines Containers in einem Image eingefrohren werden.

Es entsteht ein neues Image mit einer eindeutigen ID.

```
Local ~$ docker run -it c7df0d64f2a0499c5eb78ec31345599ceacab... /bin/sh
/ # cat test
Hello World
/ # exit
Local ~$ docker ps -a --format 'table {{.Names}}\t{{.Image}}'
NAMES                IMAGE
vigorous_hermann     c7df0d64f2a0
focused_shaw         alpine
```

Die ID des neuen Images aus dem Commit kann wieder über den Run-Befehl in einem Container instanziert werden.

Es ergeben sich damit zwei unabhängige Container.

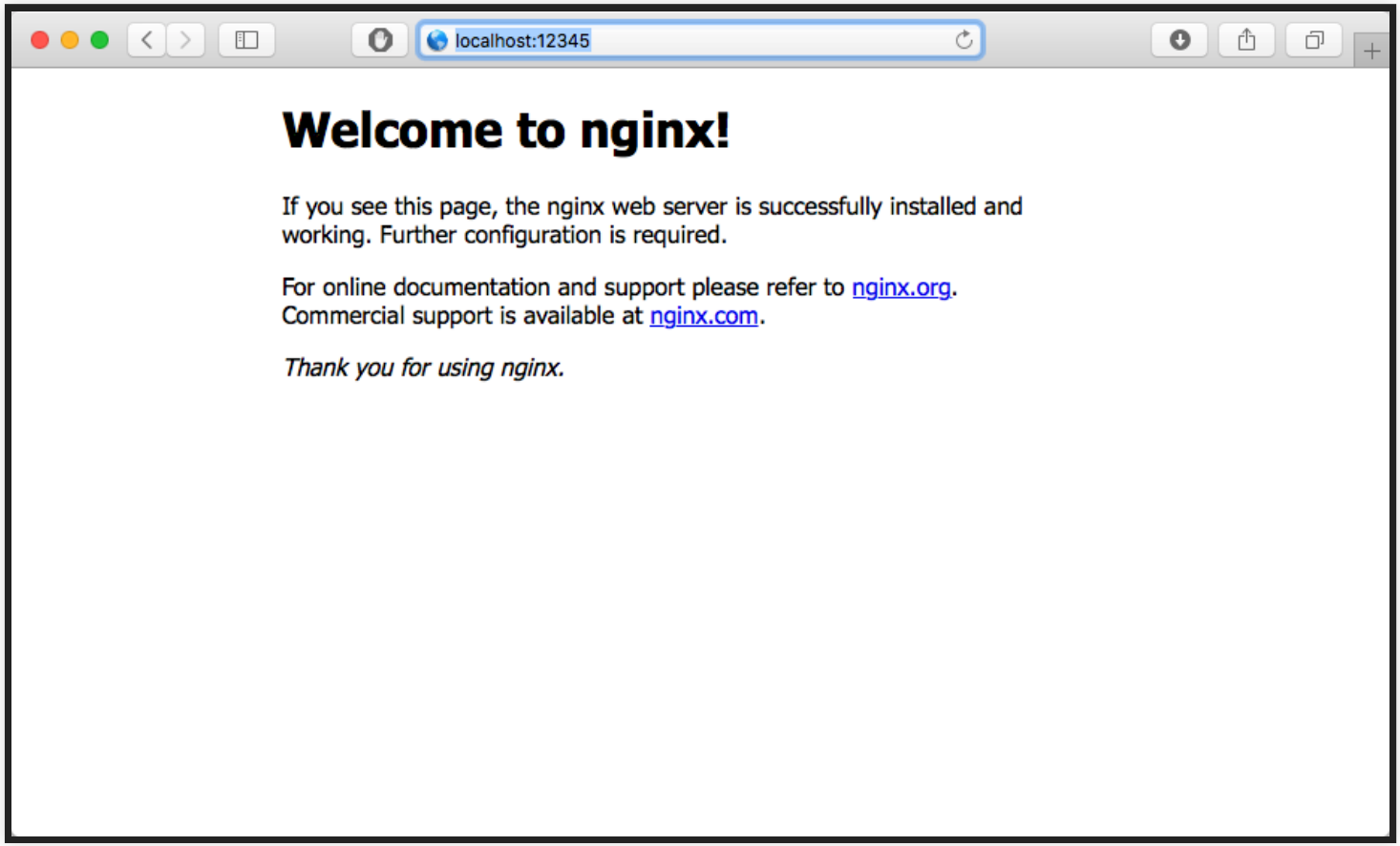
Port-Binding

Starten eines Webservers (nginx), bereitgestellt als Contanier und weiterleiten des Ports 80 innerhalb des Containers auf den Port 12345 des Hosts

```
Local ~$ docker run -d -p 12345:80 nginx # einfacher http server
Unable to find image 'nginx:latest' locally
latest: Pulling from library/nginx
f2aa67a397c4: Pull complete
3c091c23e29d: Pull complete
4a99993b8636: Pull complete
Digest: sha256:dd34e67e3371dc2d1328790c3157ee42dfcae74afffd86b2...
Status: Downloaded newer image for nginx:latest
cd84f215e897714486a7f5c400ea98fb2a930bed36750eb725b76b491863b532
```

Mittels `-p [host]:[container]` werden Port-Bindings erzeugt, hier zeigt der Port 12345 vom Host auf Port 80 im Container.

Ergänzend wurde der Container im Hintegrund gestartet mittels `-d`, vgl. Daemon.



Auf dem Host kann dann über http://localhost:12345 der Webserver aufgerufen werden.

Links

Container haben keine feste IP und ggf. nicht einmal feste publizierte Ports

```
docker run --link c560edccdae9 -it alpine /bin/sh
/ # printenv
C560EDCCDAE9_ENV_NGINX_VERSION=1.13.12-1~stretch
C560EDCCDAE9_ENV_NJS_VERSION=1.13.12.0.2.0-1~stretch
C560EDCCDAE9_NAME=/determined_mirzakhani/c560edccdae9
C560EDCCDAE9_PORT_80_TCP_ADDR=172.17.0.3
C560EDCCDAE9_PORT_80_TCP_PORT=80
C560EDCCDAE9_PORT_80_TCP_PROTO=tcp
C560EDCCDAE9_PORT_80_TCP=tcp://172.17.0.3:80
C560EDCCDAE9_PORT=tcp://172.17.0.3:80
```

Hierfür besteht die Möglichkeit Container miteinander mittels `--link <name|id>` zu verknüpfen.

Innerhalb des Containers stehen anschließend *Environment Variables* bereit, die Informationen für den Zugriff bieten.

Diese Links können benannt werden, womit die ENV-Variablen einen erwartbaren Namen erhalten:

```
Local ~$ docker run --link c560edccdae9:srv -it alpine /bin/sh
/ # printenv
SRV_ENV_NGINX_VERSION=1.13.12-1~stretch
SRV_ENV_NJS_VERSION=1.13.12.0.2.0-1~stretch
SRV_NAME=/elegant_jackson/srv
SRV_PORT_80_TCP_PORT=80
SRV_PORT_80_TCP_PROTO=tcp
SRV_PORT_80_TCP_ADDR=172.17.0.3
SRV_PORT_80_TCP=tcp://172.17.0.3:80
SRV_PORT=tcp://172.17.0.3:80
# ...
```

Mount

Daten in den Container Schichten sind nicht persistent! Hierfür steht die Möglichkeit zum Einbinden von Daten aus externen Quellen zur Verfügung, z.B. einem Ordner des Hosts oder verwaltete Volumes.

```
Local ~$ mkdir foo && echo "Hello World" > foo/bar
Local ~$ docker run --mount type=bind,source=`pwd`/foo,target=/tmp/stuff \
  -it alpine /bin/sh
/ # cat /tmp/stuff/bar
Hello World!
```

- Einen Ordner im Host erzeugen
- Den Ordner im Host auf den Ordner `/tmp/stuff` im Container abbilden, sogenannte **Mounts**
- Im Container kann man dann auf den Inhalt zugreifen
- Es sind nur absolute Pfade erlaubt, `` `pwd` `` hilft hier um den aktuellen Pfad als Basis zu verwenden

Mit dem Argument `--mount` werden die Mounts explizit konfiguriert und werden empfohlen. Über das Argument -v lässt sich eine Kurzform nutzen:

```
Local ~$ mkdir foo && echo "Hello World" > foo/bar
Local ~$ docker run -v `pwd`/foo:/tmp/stuff -it alpine /bin/sh
/ # cat /tmp/stuff/bar
Hello World!
```

Setzt die selbe Art eines Mounts um. Vergleichbar mit der vorangegangenen Art und Weise.

Volumes

Sollen keine Ordner vom Dateisystem verwendet werden, sondern extern verwaltete Ressourcen, müssen diese erstellt werden.

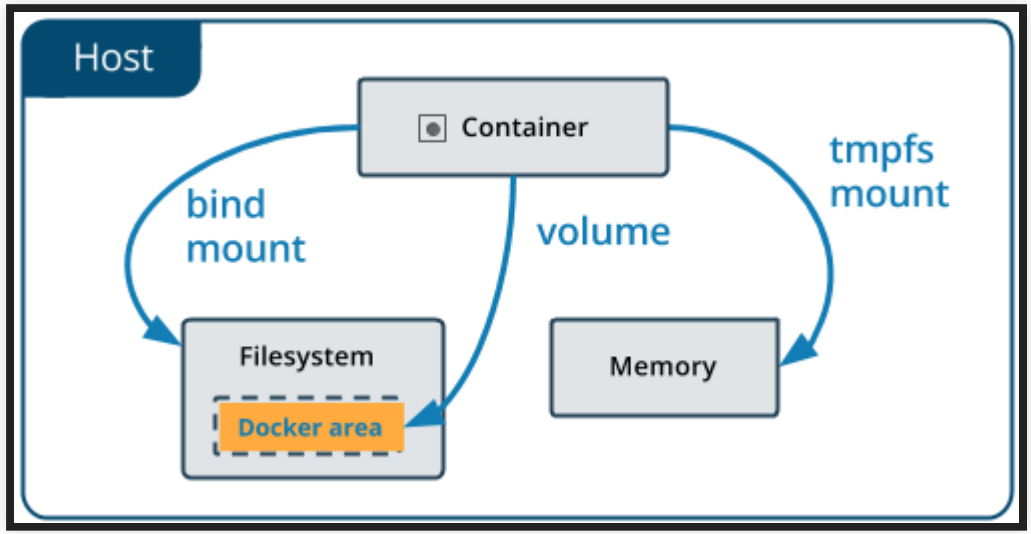
```
Local ~$ docker volume create my-vol
Local ~$ docker run --rm -it --mount source=my-vol,target=/tmp/stuff \
  alpine /bin/sh
/ # echo "Hello?" > /tmp/stuff/bar
/ # exit
Local ~$ docker run --rm -it --mount source=my-vol,target=/tmp/stuff \
  alpine /bin/sh
/ # cat /tmp/stuff/bar
```

- Erzeugt ein neues Volume
- Startet einen Container (der nach dem Beenden gelöscht wird) und verwendet das Volume
- Erstellt eine Datei in dem Ordner
- Startet einen weiteren Container und gibt den Inhalt aus , der Inhalt hat über die Existenz des Containers bestand.

Alternativ besteht die Möglichkeit zur Nutzung von Weiterverwendung von Volumes

```
Local ~$ docker run -v /var/volume1 --name DATA busybox true
Unable to find image 'busybox:latest' locally
latest: Pulling from library/busybox
f70adabe43c0: Pull complete
Digest: sha256:58ac43b2cc92c687a32c8be6278e50a063579655fe309...
Status: Downloaded newer image for busybox:latest
Local ~$ docker run --volumes-from DATA -it alpine /bin/sh
/ # echo "Hello World" > /var/volume1/bar
/ # exit
Local ~$ docker run --volumes-from DATA -it alpine /bin/sh
/ # cat /var/volume1/bar
Hello World
```

- Startet einen Container mit einem verwalteten Volume
- Startet einen weiteren Container, welcher die Volumes vom ersten Container übernimmt
- Startet einen weiteren Container, welcher ebenso die Volumes vom ersten Container übernimmt



- Volumes sind einfacher zu sichern oder zu migrieren als Mounts
- Volumes lassen sich mit Docker CLI-Befehlen oder der Docker-API verwalten
- Volumes können auf mehrere Container verteilt werden
- Volume-Treibern lassen sich Volumes auf entfernten Hosts oder Cloud-Providern speichern, um den Inhalt von Volumes zu verschlüsseln oder andere Funktionen hinzuzufügen
- Volumes können durch einen Container vorinitialisiert werden

Weitere Befehle

- Stoppen eines Containers: `docker stop <name|id>`
- Entfernen eines Containers: `docker rm <name|id>`
- Auflisten von lokal verfügbaren Images: `docker image ls`
- Entfernen eines Images: `docker image rm <name|id>`
- ... alles weitere in der Hilfe: `docker help`

Prune

Bereinigung von Resourcen

- Alle gestoppten Container entfernen: `docker container prune`
- Alle Volumes ohne Bezug zu Containern entfernen: `docker volume prune`
- Nicht getaggte Images entfernen: `docker image prune`
- Images die nicht min. als ein Container laufen: `docker image prune -a`
- Alle Netzwerke die nicht mit min. einem Container in Verbindung sind: `docker network prune`
- Komplette Bereinigung: `docker system prune`

ÜBUNG: NGINX-DOCKER-CONTAINER

Starten Sie einen Container unter Verwendung von des nginx-Docker-Images

Ergänzen Sie notwendige Port-Bindings und nutzen Sie einen Mount, um einen Ordner im Container einzubinden. Platzieren Sie in dem Ordner eine einfache HTML-Datei und versuchen Sie diese im Browser zu laden, wenn der Container läuft.

ÜBUNG: DOCKER-CONTAINER MIT EIGENER ANWENDUNG

Starten Sie einen Docker-Container und führen Sie ein Beispiel aus

Verwenden Sie zum Beispiel das REST-Beispiel aus dem Vorangegangenen Kapitel. Achten Sie darauf, den Container direkt mit einem korrekten Port-Binding auszustatten, dies lässt sich später nicht mehr korrigieren.

Hinweise

- Sie benötigen ein geeignetes Basis-Image
- Sie können mit einem Alpine-Linux starten oder recherchieren nach JDK- oder Maven-Containern
- Nutzen Sie Mounts, um Quellcode aus dem Host-System in dem Container verfügbar zu machen
- Starten Sie die Anwendung

NETWORKING

One of the reasons Docker containers and services are so powerful is that you can connect them together[...]

https://docs.docker.com/network/

- Der Docker Daemon startet auf dem Host ein virtuelles Default-Netzwerk, in dem (ohne weitere Konfiguration) die erstellten Container gestartet werden und eine IP erhalten
- Docker erlaubt die Verwaltung verschiedener virtueller Netzwerke und die Zuordnung von Containern zu ein oder mehreren dieser Netzwerke
- Netzwerke können unterschiedliche Treiber verwenden, die entsprechende Netzwerkfunktionalitäten mitsichbringen

Driver

- `bridge` ist der Default-Driver, es wird ein virtuelles Netzwerk mit eigenem IP-Bereich erstellt, der Docker Daemon arbeitet als Bridge / Gateway und erlaubt die Kommunikation über das Interface des Hosts nach außen, Isoliert die Container von der Außenwelt
- `host` nutzt das Netzwerk des Hosts
- `overlay` erzeugt ein virtuelles Netzwerk, welches sich über mehrere Docker Daemons erstrecken kann
- `none` deaktiviert alle Netzwerkfunktionalitäten

Beispiel

```
Local ~$ docker network create --driver bridge alpine-net
bae93458142a5e9b7a3e594a6080156f8929f4645dc29968c690df7a35ccb643
Local ~$ docker network ls
NETWORK ID     NAME                  DRIVER    SCOPE
bae93458142a   alpine-net            bridge    local
# ...
Local ~$ docker run --network alpine-net alpine ping google.de
PING google.de (216.58.212.131): 56 data bytes
64 bytes from 216.58.212.131: seq=0 ttl=37 time=221.089 ms
64 bytes from 216.58.212.131: seq=1 ttl=37 time=176.945 ms
# ...
```

- Erzeuge ein Netzwerk mit Driver `bridge` und Namen `alpine-net`
- Liste Netzwerke auf
- Starte einen Container in dem Netzwerk

DNS

In der Standardkonfiguration übernehmen Container die DNS Einstellungen vom Host. Zusätzlich können die Namen der Container innerhalb eines (Docker-)Netzwerkes aufgelöst werden.

```
Local ~$ docker run -it --name test1 --network alpine-net alpine /bin/sh
```

Starte einen Container mit dem Namen `test1` in einem **eigenen** Terminal.

```
Local ~$ docker run -it --name test2 --network alpine-net alpine /bin/sh
```

Starte einen weiteren Container mit dem Namen `test2` in einem **anderen** Terminal.

```
Local ~$ docker run -it --name test2 --network alpine-net alpine /bin/sh
/ # ping test1
PING test1 (172.21.0.2): 56 data bytes
64 bytes from 172.21.0.2: seq=0 ttl=64 time=0.083 ms
64 bytes from 172.21.0.2: seq=1 ttl=64 time=0.176 ms
64 bytes from 172.21.0.2: seq=2 ttl=64 time=0.344 ms
# ...
```

- Beide Container-IPs können über den jeweiligen Namen aufgelöst werden
- Hier mittels `ping test1` ein Kommunikationsversuch aus Container `test2`

DOCKERFILE

- Docker kann Images automatisch erstellen, indem es die Anweisungen aus einem Dockerfile liest
- Ein Dockerfile ist ein Textdokument, welches alle Befehle enthält, die auf der Kommandozeile aufrufen werden, um ein Image zusammenzustellen.
- Docker-Build kann genutzt werden um den automatisierten Build zu erstellen

```
FROM alpine
RUN echo "Hello World" > bar
CMD ["cat","bar"]
```

- Mittels `FROM` setzt das Basis-Image auf dem das zu erzeugende aufbaut
- Der `RUN` führt einen beliebigen Befehl aus
- ... und `CMD` wird ausgeführt, wenn eine Instanz von diesem Image gestartet wird

```
Local ~$ docker build -t test .
Sending build context to Docker daemon  38.91kB
Step 1/3 : FROM alpine
 ---> 3fd9065eaf02
Step 2/3 : RUN echo "Hello World" > bar
 ---> Running in 375fecbab867
Removing intermediate container 375fecbab867
 ---> 912c07939bf6
Step 3/3 : CMD cat bar
 ---> Running in 42e6c8c1d115
Removing intermediate container 42e6c8c1d115
 ---> 3ae5ef41f9e5
Successfully built 3ae5ef41f9e5
Successfully tagged test:latest
Local ~$ docker run test
Hello World
```

Gebaut wird das Dockerfile mit `docker build -t test .` (-t benennt den Build mit einem Tag).

Anschließend werden alle Befehle im Dockerfile verarbeitet ...

... und lokal das Image bereitgestellt (über ID oder Name)

```
FROM alpine
ENV PLACE=World
WORKDIR /stuff
RUN echo "Hello $PLACE" > ./bar
COPY foo ./
ADD https://en.wikipedia.org/wiki/File:Example.jpg ./
CMD ls -la
```

- Mittels `ENV` werden Umgebungsvariabeln definiert, verfügbar ab der Definition
- Das `WORKDIR` setzt das aktuelle Arbeitsverzeichnis
- Über `COPY <src> ... <dst>` werden Dateien in das Dateisystem des Containers an der Stelle hinzugefügt
- Über `ADD <src> ... <dst>` werden URL(s) in das Dateisystem des Containers an der Stelle hinzugefügt

Erneutes Bauen zeigt anschließend mehr Bau-Schritte

```
Local ~$ docker build -t test .
Sending build context to Docker daemon  39.94kB
Step 1/7 : FROM alpine
 ---> 3fd9065eaf02
Step 2/7 : ENV PLACE=World
 ---> Running in c6f01c0198db
Removing intermediate container c6f01c0198db
 ---> 40fd18ef931f
Step 3/7 : WORKDIR /stuff
Removing intermediate container fcf9088dc774
 ---> a299a7fce331
Step 4/7 : RUN echo "Hello $PLACE" > ./bar
 ---> Running in 6337dbf80995
Removing intermediate container 6337dbf80995
 ---> bc950e2c590b
Step 5/7 : COPY foo ./
 ---> b5782a5d458d
```

```
Step 6/7 : ADD https://en.wikipedia.org/wiki/File:Example.jpg ./
Downloading  53.34kB
 ---> b51100bef37a
Step 7/7 : CMD ls -la
 ---> Running in 1a0240cb8139
Removing intermediate container 1a0240cb8139
 ---> a930ac44318f
Successfully built a930ac44318f
Successfully tagged test:latest
Local ~$ docker run test
total 72
drwxr-xr-x    1 root     root          4096 May 24 10:01 .
drwxr-xr-x    1 root     root          4096 May 24 10:01 ..
-rw-------    1 root     root         53345 May 24 08:00 File:Example.jpg
-rw-r--r--    1 root     root            12 May 24 10:01 bar
-rw-r--r--    1 root     root             5 May 23 14:39 foo
```

Als CMD wurde `ls -la` hinterlegt, es ist beim Start der Inhalt des Working-Directories zu sehen.

Hinweise

- Möglichst `COPY` zur Übertragung von Dateien aus dem Host bevorzugen (um die Magic von ADD nur bewusst zu verwenden)
- Die `RUN` Befehle sparsam einsetzen, lieber lange Befehlsketten (erzeugt jeweils eine Schicht)
- Zur Übersichtlichkeit kann man Skripte auslagern, hinein kopieren und im Dockerfile zur Ausführung bringen

Achtung

- Ein Dockerfile definiert in seiner Abfolge Befehle die im Bauprozess genutzt werden, um das Image zu erzeugen
- Nicht jeder Befehl wird deswegen bereits beim Bauprozess ausgeführt
- Zum Beispiel wird mittels `CMD` nur hinterlegt, was zukünftig mit `docker run ...` gestartet werden soll
- Es gibt diverse Informationen, die sich über den Docker-Run-Aufruf wieder überschreiben lassen (z.B. `CMD`, `ENTRYPOINT`, `ENV`, usw.)

Ports

EXPOSE 12345 informiert Docker, dass der Container auf diesen Port lauscht

```
FROM alpine
ENV PLACE=World
WORKDIR /stuff
RUN echo "Hello $PLACE" > ./bar
COPY foo ./
ADD https://en.wikipedia.org/wiki/File:Example.jpg ./
EXPOSE 80
CMD ls -la
```

Dies bedeutet nicht automatisch, dass der Port publiziert wird!

Tags

Tags können verwendet werden um Versionen von gebauten Containern zu benennen.

```
Local ~$ docker build -t test .
# ...
Successfully built a930ac44318f
Successfully tagged test:latest
Local ~$ docker tag test test:variante1
```

Hierfür kann z. B. `docker tag test test:variante1` verwendet werden

Entrypoint

```
FROM alpine
ENV PLACE=World
WORKDIR /stuff
RUN echo "Hello $PLACE" > ./bar
COPY foo ./
ADD https://en.wikipedia.org/wiki/File:Example.jpg ./
EXPOSE 80
ENTRYPOINT ["ls","-la"] # vs. CMD ["ls", "-la"] od CMD ls -la
```

Mittels `ENTRYPOINT ["executable", "param1", "param2", ...]`
kann ein CMD-Prefix konfiguriert werden.

Dieser wird beim Start vor dem eigentlichen Kommando platziert.

```
Local ~$ docker build -t test .
# ...
Local ~$ docker run test
total 72
drwxr-xr-x    1 root    root     4096 May 24 10:01 .
drwxr-xr-x    1 root    root     4096 May 24 10:01 ..
-rw-------    1 root    root    53345 May 24 08:00 File:Example.jpg
-rw-r--r--    1 root    root       12 May 24 10:01 bar
-rw-r--r--    1 root    root        5 May 23 14:39 foo
Local ~$ docker run test:variante2 /bin/sh
lrwxrwxrwx    1 root    root       12 Jan  9 19:37 /bin/sh -> /bin/busybox
```

- Führt `ls -la` aus und gibt damit den Inhalt des Ordners in dem der Container arbeitet (vgl. `WORKDIR`)
- Führt `ls -la /bin/sh` aus und zeigt damit Informationen zur Anwendung `/bin/sh`

Volumes

```
FROM alpine
ENV PLACE=World
WORKDIR /stuff
RUN echo "Hello $PLACE" > ./bar
COPY foo ./
ADD https://en.wikipedia.org/wiki/File:Example.jpg ./
EXPOSE 80
VOLUME ["/var/log"]
ENTRYPOINT ["ls","-la"]
```

Der Befehl `VOLUME` erlaubt es Mount-Points zu definieren, welche später von extern zur Verfügung gestellt werden

Reihenfolge bei der Verwendung von VOLUME beachten

```
FROM alpine
ENV PLACE=World
WORKDIR /stuff
RUN echo "Hello $PLACE" > ./bar
COPY foo ./
ADD https://en.wikipedia.org/wiki/File:Example.jpg ./
EXPOSE 80
RUN echo "Hello $PLACE" > /var/log/test
VOLUME ["/var/log"]
ENTRYPOINT ["ls","-la"]
```

Neue `VOLUME` Ordner werden aus dem vorherigen Zustand automatisch initialisiert, nicht jedoch danach!

Beispiel RUN vor VOLUME ausführen

```
Local ~$ docker run test
# ...
Local ~$ docker ps -a --format 'table {{.Names}}\t{{.Image}}'
NAMES                       IMAGE
serene_yonath               test
Local ~$ docker inspect serene_yonath --format \
  '{{range .Mounts}}{{.Source}}{{end}}'
/var/lib/docker/volumes/bfd0aa8c2da4076...bc1d10c3ac/_data
Local ~$ ls -la /var/lib/docker/volumes/bfd0aa8c2da4076...bc1d10c3ac/_data
total 12
drwxr-xr-x    2 root     root          4096 May 24 11:18 .
drwxr-xr-x    3 root     root          4096 May 24 11:18 ..
-rw-r--r--    1 root     root            12 May 24 11:18 test
```

- Ausführen um den Container und das Volume zu erzeugen
- Name herausfinden
- Speicherort für das Volume auf dem Host identifizieren
- Inhalt des Ordners anzeigen

Alternatives Beispiel mit VOLUME vor RUN

```
FROM alpine
ENV PLACE=World
WORKDIR /stuff
RUN echo "Hello $PLACE" > ./bar
COPY foo ./
ADD https://en.wikipedia.org/wiki/File:Example.jpg ./
EXPOSE 80
VOLUME ["/var/log"]
RUN echo "Hello $PLACE" > /var/log/test
ENTRYPOINT ["ls","-la"]
```

Es wird erst das Volume definiert und dann in dem Ordner eine Datei erzeugt.

Beispiel VOLUME vor RUN ausführen

```
Local ~$ docker run test
# ...
Local ~$ docker ps -a --format 'table {{.Names}}\t{{.Image}}'
NAMES                       IMAGE
romantic_hamilton           test
Local ~$ docker inspect romantic_hamilton --format \
  '{{range .Mounts}}{{.Source}}{{end}}'
/var/lib/docker/volumes/1b2d46fd32488...2998888b2688/_data
Local ~$ ls -la /var/lib/docker/volumes/1b2d46fd32488...2998888b2688/_data
total 8
drwxr-xr-x    2 root     root          4096 Jan  9 19:37 .
drwxr-xr-x    3 root     root          4096 May 24 11:25 ..
```

Der Ordner welcher als Mount angegeben wurde verbleibt leer.

- Die selbe Vorsicht ist bei der Instanzierung von Containern notwendig
- Wenn im `docker run` mittels `-v` ein neuer Mount-Point entsteht, wird innerhalb des Containers das Dateisystem überlagert
- Dateien die ggf. bei der Erstellung des Images durch das Dockerfile vorgesehen sind, könnten durch ein Mount-Point an selbige Stelle nicht verfügbar sein

ÜBUNG: DOCKERFILE

Erstellen Sie ein Dockerfile, welches eine Beispiel-Anwendung ausführen kann

Erstellen Sie, z.B. für das REST-Beispiel in Spring, ein Dockerfile, welches die Anwendung enthält, alle Abhängigkeiten bereitstellt und starten kann.

Hinweise

- Wählen Sie ein geeignetes Basis-Image aus
- Was sollte idealerweise im Container platziert werden? Java-Dateien oder Class-Dateien?
- Überlegen Sie sich, welche Schritte Sie manuell in der virtuellen Maschine, im LXC Container oder beim Nutzen der Docker CLI verwendet haben. Bilden Sie diese Schritte im Dockerfile ab

REGISTRY

- Images werden auf sogenannten Registries bereitgestellt
- Die öffentliche Registry von Docker ist der Docker Hub
- Es lassen sich eigene Registries betreiben:

```
Local ~$ docker run -d -p 5000:5000 --restart=always --name registry registry:2
```

- GitLab bietet z.B. Docker Registries als Teil eines Projektes
- Über Cloud-Anbieter lassen sich eigene Registries betreiben, die für Deployments genutzt werden (vgl. AWS)

Beispiel Dockerfile

Für die folgenden Beispiele kommt ein einfaches Dockerfile als Beispiel zum Einsatz.

```
FROM alpine
CMD ["echo", "Hello, World!"]
```

Welches wie bekannt gebaut wird:

```
Local ~$ docker build -t reg-test .
```

Local Registry Push

```
Local ~$ docker tag reg-test localhost:5000/reg-test:v1.0
Local ~$ docker push localhost:5000/reg-test:v1.0
Local ~$ docker image rm reg-test
Local ~$ docker image rm localhost:5000/reg-test:v1.0
Local ~$ docker run --rm localhost:5000/reg-test:v1.0
```

- Erzeugt einen Tag mit Verweis auf die Registry-Infrastruktur
- Schiebt das Image auf das Zielsystem
- Zum Test kann man Lokal alles löschen
- Starten des Containers basierend auf dem Image

Login with Registry

Docker muss sich bei einer Registry anmelden

```
Local ~$ docker login registry-1.docker.io
Username: YOUR_USER_ID
Password:
Login Succeeded
```

Tag Image

Zuerst muss ein gebautes Image einen Namen in der Registry erhalten, im Fall des Docker Hubs muss der Nutzername als Prefix verwendet werden (oder ein Projekt- / Organisationsname).

```
Local ~$ docker image tag reg-test YOUR_USER_ID/registry-test
```

Hinweis: Im Fall von GitLab dagegen gibt es eine andere Namensstruktur

Push Image

Anschließend kann das Image auf die Registry geschoben werden.

```
Local ~$ docker image push YOUR_USER_ID/registry-test
Using default tag: latest
The push refers to repository [docker.io/YOUR_USER_ID/registry-test]
5216338b40a7: Mounted from trafex/alpine-nginx-php7
latest: digest: sha256:4e30094a7754c39e184...7fd797925c003618 size: 527
```

Image laden und ausführen

Anschließend kann das Image auf anderen Maschinen geladen und ausgeführt werden:

```
Local ~$ docker run YOUR_USER_ID/registry-test
```

ÜBUNG: DOCKER-IMAGE COMMIT

(Optional) Erstellen Sie sich einen Docker-Hub-Account und laden Sie einen Beispiel-Container, welcher funktioniert, auf dem Docker-Hub hoch. Sie können ebenfalls das folgende Beispiel nutzen.

Nutzen Sie das bereitgestellte Projekt-Archiv docker-registry.zip, entpacken Sie dieses und importieren Sie den resultierenden Ordner.

Wichtig: Nutzen Sie keine kritische Anwendung, der Container ist öffentlich verfügbar.

ÜBUNG: DOCKER-IMAGE PULL

(Optional) Entfernen Sie mittels `docker image rm` alle verbleibenden Images auf Ihrem Host und versuchen Sie Ihren Container vom Docker-Hub zu laden.

BEISPIEL: NGINX

```
FROM dockerfile/ubuntu

RUN \
  add-apt-repository -y ppa:nginx/stable && \
  apt-get update && \
  apt-get install -y nginx && \
  rm -rf /var/lib/apt/lists/* && \
  echo "\ndaemon off;" >> /etc/nginx/nginx.conf && \
  chown -R www-data:www-data /var/lib/nginx

VOLUME ["/etc/nginx/sites-enabled", "/etc/nginx/certs",
  "/etc/nginx/conf.d", "/var/log/nginx", "/var/www/html"]

WORKDIR /etc/nginx
CMD ["nginx"]
EXPOSE 80
EXPOSE 443
```

https://github.com/dockerfile/nginx

BEISPIEL: GITLAB

```
FROM ubuntu:14.04
MAINTAINER Sytse Sijbrandij

# Install required packages
RUN apt-get update -q \
    && DEBIAN_FRONTEND=noninteractive apt-get install -yq --no-install-recom
      ca-certificates \
      openssh-server \
      wget \
      apt-transport-https \
      vim \
      nano
#...
```

https://hub.docker.com/r/gitlab/gitlab-ce/~/dockerfile/

```
# Download & Install GitLab
# If you run GitLab Enterprise Edition point it to a location where you have
RUN echo "deb https://packages.gitlab.com/gitlab/gitlab-ce/ubuntu/ `lsb_rele
RUN wget -q -O - https://packages.gitlab.com/gpg.key | apt-key add -
RUN apt-get update && apt-get install -yq --no-install-recommends gitlab-ce

# Manage SSHD through runit
RUN mkdir -p /opt/gitlab/sv/sshd/supervise \
    && mkfifo /opt/gitlab/sv/sshd/supervise/ok \
    &&
printf "#!/bin/sh\nexec 2>&1\numask 077\nexec /usr/sbin/sshd -D" \
       > /opt/gitlab/sv/sshd/run \
    && chmod a+x /opt/gitlab/sv/sshd/run \
    && ln -s /opt/gitlab/sv/sshd /opt/gitlab/service \
    && mkdir -p /var/run/sshd
#...
```

https://hub.docker.com/r/gitlab/gitlab-ce/~/dockerfile/

```
# Disabling use DNS in ssh since it tends to slow connecting
RUN echo "UseDNS no" >> /etc/ssh/sshd_config

# Prepare default configuration
RUN ( \

echo "" && \

echo "# Docker options" && \

echo "# Prevent Postgres from trying to allocate 25% of total memory" && \

echo "postgresql['shared_buffers'] = '1MB'" ) >> /etc/gitlab/gitlab.rb && \
  mkdir -p /assets/ && \
  cp /etc/gitlab/gitlab.rb /assets/gitlab.rb
#...
```

https://hub.docker.com/r/gitlab/gitlab-ce/~/dockerfile/

```
# Expose web & ssh
EXPOSE 443 80 22

# Define data volumes
VOLUME ["/etc/gitlab", "/var/opt/gitlab", "/var/log/gitlab"]

# Copy assets
COPY assets/wrapper /usr/local/bin/

# Wrapper to handle signal, trigger runit and reconfigure GitLab
CMD ["/usr/local/bin/wrapper"]
```

https://hub.docker.com/r/gitlab/gitlab-ce/~/dockerfile/

ÜBUNG: DOCKERFILE (1)

Erstellen Sie ein Dockerfile, welches alle notwendigen Bauwerkzeuge enthält, damit eine Beispiel-Anwendung compiliert werden kann.

Erstellen Sie zum Beispiel ein einfaches Projekt mit einer main.cpp und einer CMakeLists.txt. Das Dockerfile sollte dann notwendige Bauwerkzeuge enthalten.

ÜBUNG: DOCKERFILE (2)

Erstellen Sie ein Dockerfile, welches den Bauprozess abbildet und das Ergebnis (die Executable) startet, wenn der Container gestartet wird.

Erstellen Sie zum Beispiel ein einfaches Projekt mit einer main.cpp und einer CMakeLists.txt. Das Dockerfile sollte dann notwendige Bauwerkzeuge enthalten. Entsprechende Anweisungen, die beim Start auszuführen sind, wären mitzugeben.

MULTI-STAGE-DOCKERFILE

Wie lassen sich Bauprozesse und resultierende Artefakte mit Dockerfiles beschreiben?

Es ist nicht zwingend sinnvoll im resultierenden Artefakt Werkzeuge zu transportieren, die für den Bauprozess notwendig waren.

Der Inhalt eines Images, also alle installierten Abhängigkeiten und hinzugefügten Dateien, sollten sich auf einen konkreten Anwendungszweck konzentrieren. Je mehr Sie einem Image hinzufügen, desto Größer und damit aufwendiger wird der Bereitstellungsprozess aber auch desto mehr eventuelle Sicherheitsrisiken sind mit der Ausführung verbunden. Sollte es möglich sein aus Ihrer Anwendung auszubrechen, und Zugriff auf die Kommandozeile zu gelangen, dann entscheidet sich über die Zugriffsrechte und Anwendungen im Container, welcher Schaden angerichtet werden könnte.

Ziel: Bauprozesse von publizierbaren Images trennen

Eine Trennung von Bauprozess und resultierenden Artefakt wäre mehrere Dockerfiles möglich.

Beispiel `build.Dockerfile`:

```
FROM alpine
RUN apk add --update-cache cmake build-base \
  && rm -rf /var/cache/apk/*
WORKDIR /app
COPY . ./
RUN mkdir build && cd build && cmake .. && make all
```

Beispiel `Dockerfile` für die Ausführung der Anwendung:

```
FROM alpine
RUN apk add --update-cache libstdc++ \
  && rm -rf /var/cache/apk/*
WORKDIR /root
COPY app ./
CMD ["./app"]
```

Damit beide Dockerfiles genutzt werden können, bietet sich ein Shell-Script an, welcher den Bauprozess beschreibt:

```
#!/bin/sh
echo Building thi/hello-world:build
docker build -t thi/hello-world:build . -f build.Dockerfile
docker container create --name extract thi/hello-world:build
docker container cp extract:/app/build/cmake_hello ./app
docker container rm -f extract

echo Building thi/hello-world:latest
docker build --no-cache -t thi/hello-world:latest .
```

Docker liefert hierfür einen eigenen Syntax

Grundlage hierfür ist die Verwendung von mehreren FROM-Angaben.

```
FROM alpine
RUN apk add --update-cache cmake build-base \
  && rm -rf /var/cache/apk/*
WORKDIR /app
COPY . ./
RUN mkdir build && cd build && cmake .. && make all

FROM alpine
RUN apk add --update-cache libstdc++ \
  && rm -rf /var/cache/apk/*
WORKDIR /root
COPY --from=0 /app/cmake_hello ./app
CMD ["./app"]
```

Beim Kopieren wird dann angegeben, aus welcher Stage Inhalte kopiert werden sollen.

Es ist Möglich die Stages zu benennen

```
FROM alpine AS builder
RUN apk add --update-cache cmake build-base \
  && rm -rf /var/cache/apk/*
WORKDIR /app
COPY . ./
RUN mkdir build && cd build && cmake .. && make all

FROM alpine
RUN apk add --update-cache libstdc++ \
  && rm -rf /var/cache/apk/*
WORKDIR /root
COPY --from=builder /app/cmake_hello ./app
CMD ["./app"]
```

REFERENZEN

- Christoph Schröder: Container-Virtualisierung mit Docker. Internet https://media.itm.uni-luebeck.de/teaching/ws2013/sem-cloud-computing/Container-Virtualisierung_mit_Docker.pdf
- Dockerfile References. https://docs.docker.com/engine/reference/builder/

4.5 INFRASTRUCTURE AS CODE

Infrastructure-as-Code (IaC) ist die Verwaltung von Infrastruktur (Netzwerken, virtuellen Computern, Lastenausgleichsmodulen und der Verbindungstopologie) in einem beschreibenden Modell.

Quelle: https://docs.microsoft.com/de-de/devops/deliver/what-is-infrastructure-as-code

Dabei wird die gleiche Versionsverwaltung verwendet, wie sie das DevOps-Team für Quellcode nutzt. Wie bei dem Prinzip, dass mit dem gleichen Quellcode auch die gleichen Binärdaten generiert werden, wird mit einem IaC-Modell bei jeder Anwendung die gleiche Umgebung generiert. IaC ist eine wichtige DevOps-Methode und wird in Verbindung mit Continuous Deliveryverwendet.

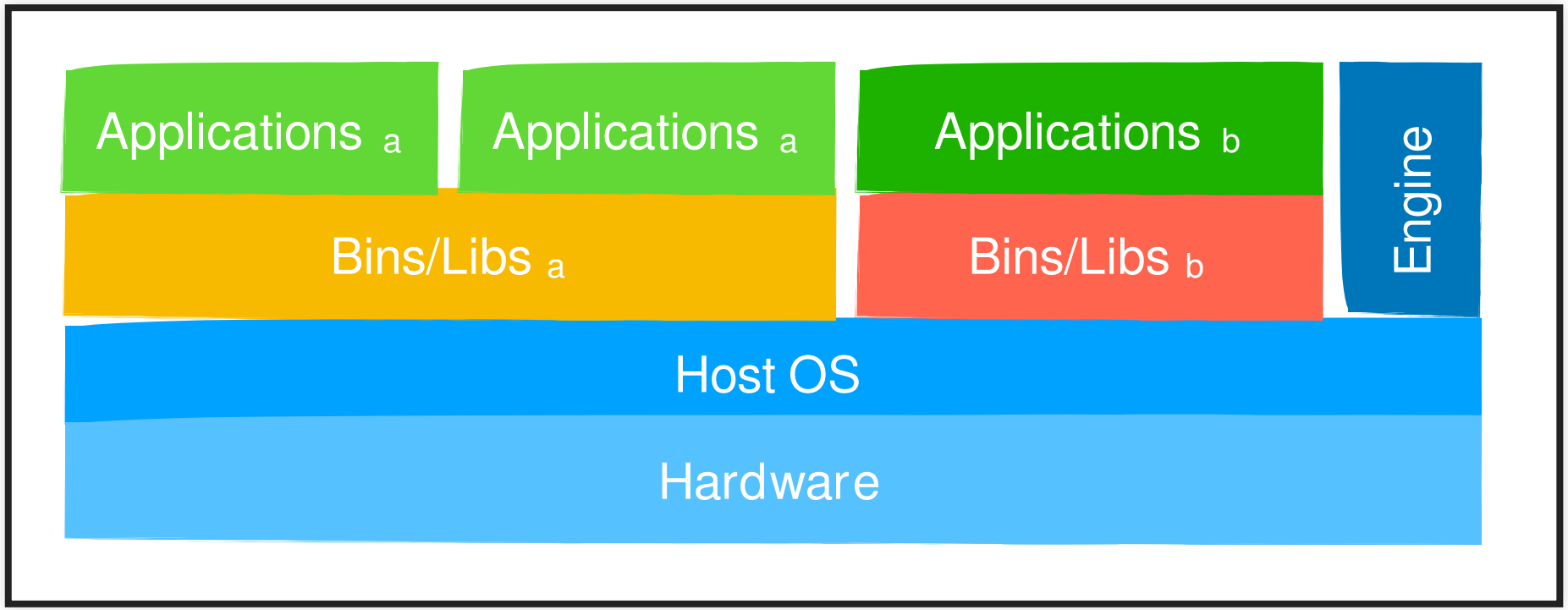
Quelle: https://docs.microsoft.com/de-de/devops/deliver/what-is-infrastructure-as-code

DOCKER-COMPOSE

- `Dockerfile` beschreibt die Erstellung eines Images, nicht wie ein Container entsteht
- Jeder Container erfordert eine bestimmte Menge an Konfigurationen zur Ausführung
- Konfiguration ist abhängig vom Anwendungsfall und des jeweiligen Deployments

Infrastrukturbeschreibung mit Docker

- Docker-Compose ist ein Werkzeug zur Definition und Ausführung von (Multi-) Container Anwendungen
- Zur Beschreibung der Container Konfiguration wird eine YAML Datei genutzt
- Anschließend kann die Konfiguration mittels Docker-ähnlichen Befehlen verwaltet werden: **up**, **down**, **ps**, **restart**...



- Docker-Compose ist ein in Python geschriebener Proxy zum Zugriff auf die Docker-Engine
- Docker-Compose arbeitet eigenständig und nutzt die Docker-CLI um die Konfigurationen auf Docker-Befehle abzubilden

GRUNDLAGEN

Eine Text-Datei mit dem Namen docker-compose.yml

```
version: "3"

services:
  app:
    image: nginx
```

- Jedes Compose-File benötigt eine Version, aktuell ist 3.X
- Anschließend folgt die Definition der verschiedenen Infrastrukturbausteine: Service, Volumes, Netzwerke

```
Local ~$ docker-compose up
Creating network "example_default" with the default driver
Creating example_app_1 ...
Creating example_app_1 ... done
Attaching to example_app_1
```

- Docker-Compose bringt ein eigenes CLI zum Verwalten der aus dem Compose-File entstehenden Infrastruktur
- Mittels `docker-compose up` werden alle Dienste der Konfiguration erzeugt, mittels `-d` im Hintergrund als Daemon
- Dabei dient der aktuelle Ordner Name als Basis für die Benennung der Container
- Mit dem Beispiel startet ein neues Default-Netzwerk, sowie ein Container mit dem Namen `example_app_1`

- Mittels `docker-compose stop` werden alle Dienste der Konfiguration gestoppt
- Mittels `docker-compose rm` werden alle Dienste der Konfiguration entfernt
- Mittels `docker-compose down` werden alle Dienste der Konfiguration gestoppt **und** entfernt

```
Local ~$ docker-compose down
Stopping example_app_1 ... done
Removing example_app_1 ... done
Removing network example_default
```

```
Local ~$ docker-compose pull
Pulling app (library/nginx:latest)...
latest: Pulling from library/nginx
Digest: sha256:55cc0c7d24ccc8fb7af7120ee309ba1ee3409d20448887da58df1a94552d0
Status: Image is up to date for library/nginx:latest
```

- Mittels `docker-compose pull` werden die lokal verfügbaren Images bei Bedarf aktualisiert
- Mittels `docker-compose restart` werden alle Dienste der Konfiguration neugestartet
- Mittels `docker-compose ps` werden alle Dienste der Konfiguration angezeigt

```
Local ~$ docker-compose up app
Creating network "example_default" with the default driver
Creating example_app_1 ...
Creating example_app_1 ... done
Attaching to example_app_1
```

- Es lassen sich einzelne Dienste der Konfiguration gezielt ansprechen
- Hierfür ist der Dienstname hinter dem entsprechenden Kommando zu ergänzen

Kombination mit Dockerfiles

```
version: "3"

services:
  app:
    build: .
```

- Anstatt die Angabe eines Image kann build verwendet werden um auf ein Dockefile zu verweisen
- Dieses wird beim Start der Konfiguration gebaut
- Neubau muss ggf. manuell Ausgelöst werden mit `docker-compose up --build`

Port-Forwarding

```yaml
version: "3"

services:
  app:
    image: nginx
    ports:
      - 12345:80
```

- Mittels Attribut `ports` können die zu publizierenden Ports festgelegt werden
- Hier gilt der selbe Syntax wie mit der Docker CLI: `<host>:<container>[/<protocol>]`, z. B. `80:80` oder `1194:1194/udp`

ENV-Variablen

```
version: "3"

services:
  app:
    image: nginx
    ports:
      - 12345:80
    environment:
      VAR1: KEY1
      VAR2: KEY2
```

Mittels Attribut environment können Umgebungsvariabeln gesetzt werden.

Abhängigkeiten

```yaml
version: "3"

services:
  db:
    image: mariadb
  app:
    image: nginx
    ports:
      - 12345:80
    depends_on:
      - db
```

- Mittels Attribut `depends_on` können Äbhängigkeiten zwischen Services gesetzt werden
- Abhängigkeiten garantieren kein Warten, bis der andere Dienst gestartet wurde
- Es wird sichergestellt das abhängige Container zusammen starten. Zum Beispiel startet der DB-Container, wenn mittels `docker-compose up app` die App startet

Start, Initialisierung, Ready

- Der Zeitpunkt, wann eine Anwendung vollständig gestartet ist, ist für die Docker Engine unbekannt
- Ein Trick ist hier z. B. auf die Verfügbarkeit von Endpunkten zu warten
- Das warten wird mittels Bash- oder Shell-Skripten realisiert, welche jedoch vorher ausführbar im Docker Image liegen müssen
- Siehe: https://docs.docker.com/compose/startup-order/

Neustart-Verhalten

```
version: "3"

services:
  app:
    image: nginx
    ports:
      - 12345:80
    restart: unless-stopped
```

- Mittels Attribut restart kann beeinflusst werden, was mit dem Dienst passiert, wenn er nicht läuft
- Möglich sind `no`, `always`, `on-failure` oder `unless-stopped`

Alternativer CMD und/oder ENTRYPOINT

```yaml
version: "3"

services:
  app:
    image: nginx
    ports:
      - 12345:80
    entrypoint: ["service", "nginx"]
    command: ["start"]
```

Mittels Attribut command und entrypoint kann beeinflusst werden, wie der Container startet.

Volumes (1)

```
version: "3"

services:
  app:
    image: nginx
    ports:
      - 12345:80
    entrypoint: ["service", "nginx"]
    command: ["start"]
    volumes:
      - /var/www
```

Automatisch verwaltete Volumes, Docker-Compose vergibt einen Namen und bindet den Ordner an das Volume.

Volumes (2)

```
version: "3"

services:
  app:
    image: nginx
    ports:
      - 12345:80
    entrypoint: ["service", "nginx"]
    command: ["start"]
    volumes:
      - ./www:/var/www
```

Bekannte Volumes von Docker:

Host Mapping, bzw. Binding, in Docker-Compose kann relativ adressiert werden, in der Docker CLI nicht.

Volumes (3)

```
version: "3"

services:
  app:
    image: nginx
    ports:
      - 12345:80
    entrypoint: ["service", "nginx"]
    command: ["start"]
    volumes:
      - www:/var/www

volumes:
  www:
```

Verwendung von Named Volumes

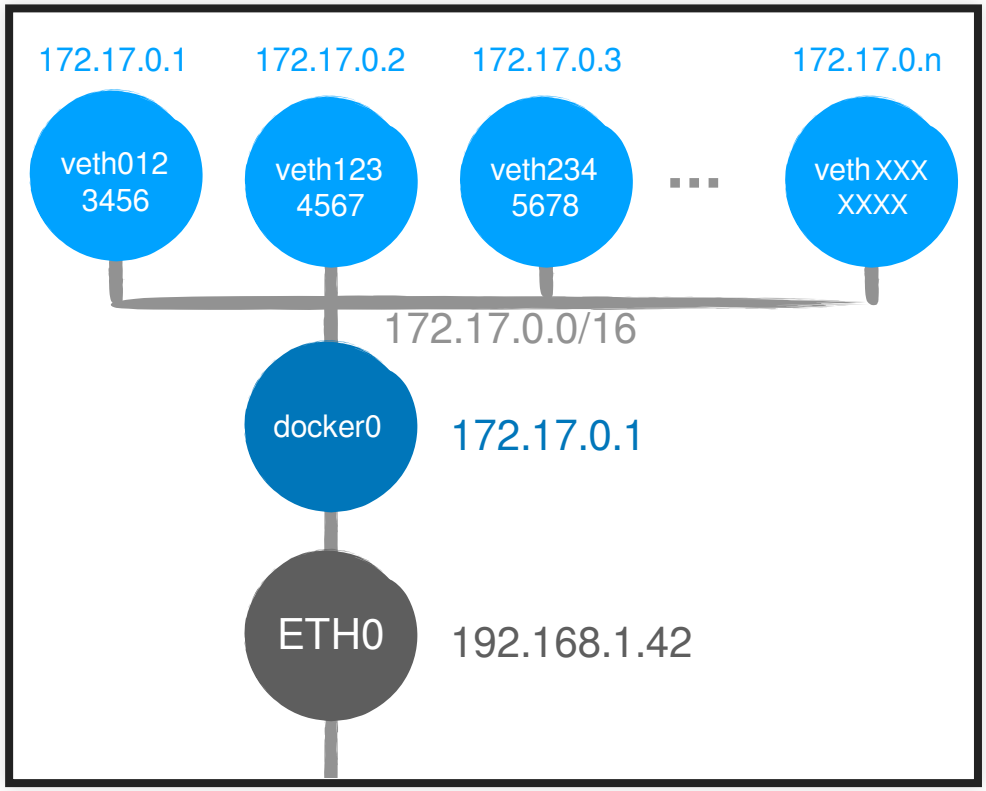
Hierfür muss ein neben den Services ein Konfigurationsblock für Volumes erstellt werden, wo alle benannten Volumes definiert werden. Werden keine weiteren Parameter ergänzt, ist dies ein Standard Docker-Volume

Hinweise zu Volumes

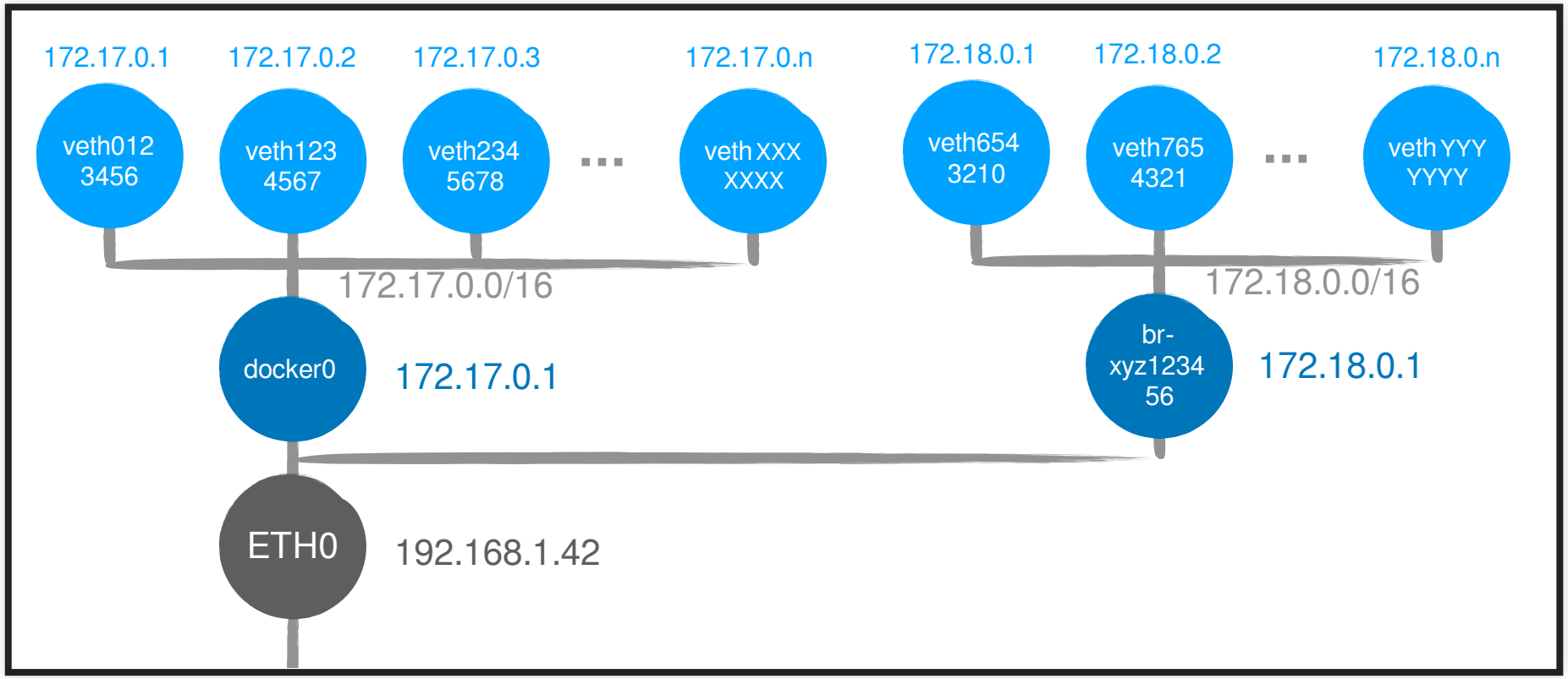
- Automatisches Docker Volume, Host Mapping und Named Volume sind einfache Varianten, die sich schnell in kleinen Setups umsetzen lassen
- In der Praxis werden Konfigurationen häufig mittels Mapping vom Host in den Container gereicht
- Für Laufzeitdaten (z.B. für Datenbanken) empfehlen sich dagegen Named Volumes
- *Wichtig:* Zum Skalieren von Systemen ist dies jedoch nicht der richtige Weg

Netzwerke

Ohne Angabe landen manuell erstellte Container im virtuellen Default Netzwerk.



Docker-Compose startet automatisch jede Konfiguration in einem separaten Netzwerk.



Diese Netzwerkstruktur kann über die Konfiguration beeinflusst und erweitert werden.

```yaml
version: "3"

services:
  app:
    image: nginx
    ports:
      - 12345:80
    networks:
      - apps

networks:
  apps:
    driver: bridge
```

Mittels `networks` in einem Dienst werden alle Netzwerke angegeben, in denen der Container aktiv sein soll.

Mittels separaten `networks`-Block werden Netzwerke die mit diesem Compose-File genutzt werden sollen definiert.

Auf die selbe Art und Weise ist es möglich sich in bereits existierenden (`external`), durch Docker verwalteten, Netzwerken zu begeben

```yaml
version: "3"

services:
  app:
    image: nginx
    ports:
      - 12345:80
    networks:
      - apps

networks:
  apps:
    external: true
```

Konfiguration kann erst gestartet werden, wenn eine andere Konfiguration das Netzwerk anlegt oder mittels: `docker network create --driver=bridge apps`