



ARS

Member of

 **TIMETOACT GROUP**
SOFTWARE & CONSULTING

Einführung in OCI konforme Containertechnologien

OCI-Konforme-Containertechnologien

Herzlich Willkommen zur Docker Grundlagenschulung!



Container Grundlagen

Historie: Container [1|2]

- Früherer Warenverkehr:
 - Stückgut
 - Zeit- und personalaufwändig



Historie: Container [2|2]

- Moderner Warenverkehr:
 - Standardisierung von Form, Größe, Befestigungspunkten, ...
 - Erlaubt Standardisierung von Schiffen, Wagons, LKWs, Kränen, ...
 - Erhebliche Effizienzsteigerung



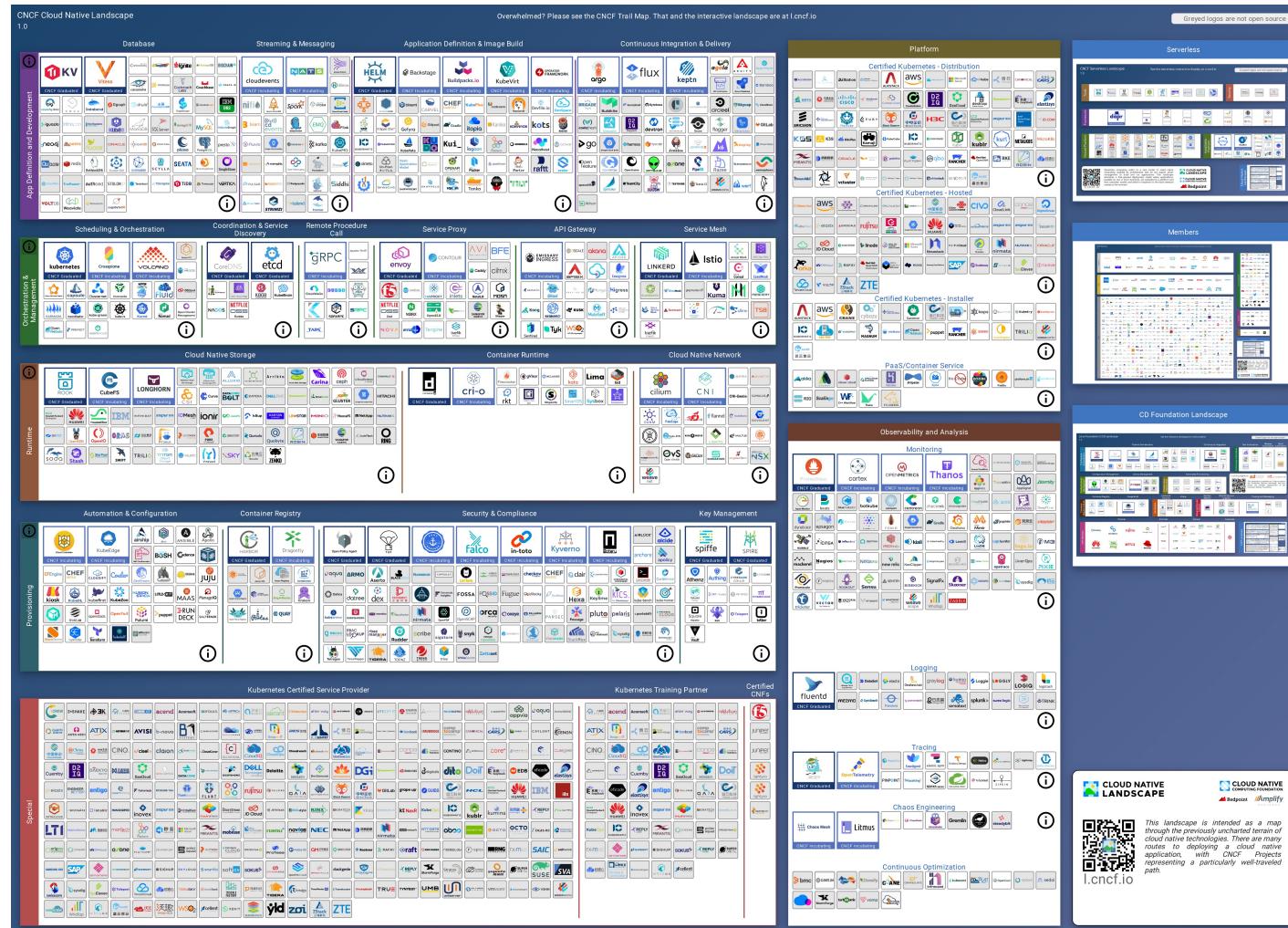
Containerkonzepte sind universell

- Containerkonzepte sind überall im Alltag zu finden
- In der IT: Bezug auf Programme und ihre Laufzeitumgebungen
- **Virtualisierung**
 - z.B. JVM – Java Virtual Maschine
 - Web und EJB Container im Kontext von JavaEE
 - VMWare Images
- **Standardisierung** (mit allen Vor- und Nachteilen)
- **Trennung von Verantwortung**
 - Entwickler erstellt den Inhalt und den Container (z.T. auch nur den „Bauplan“ für den Container)
 - Laufzeitumgebung stellt Systemressourcen bereit
 - Standardisierte Schnittstellen



Landscape der CNCF

- Container und Cloud ergänzen sich perfekt!



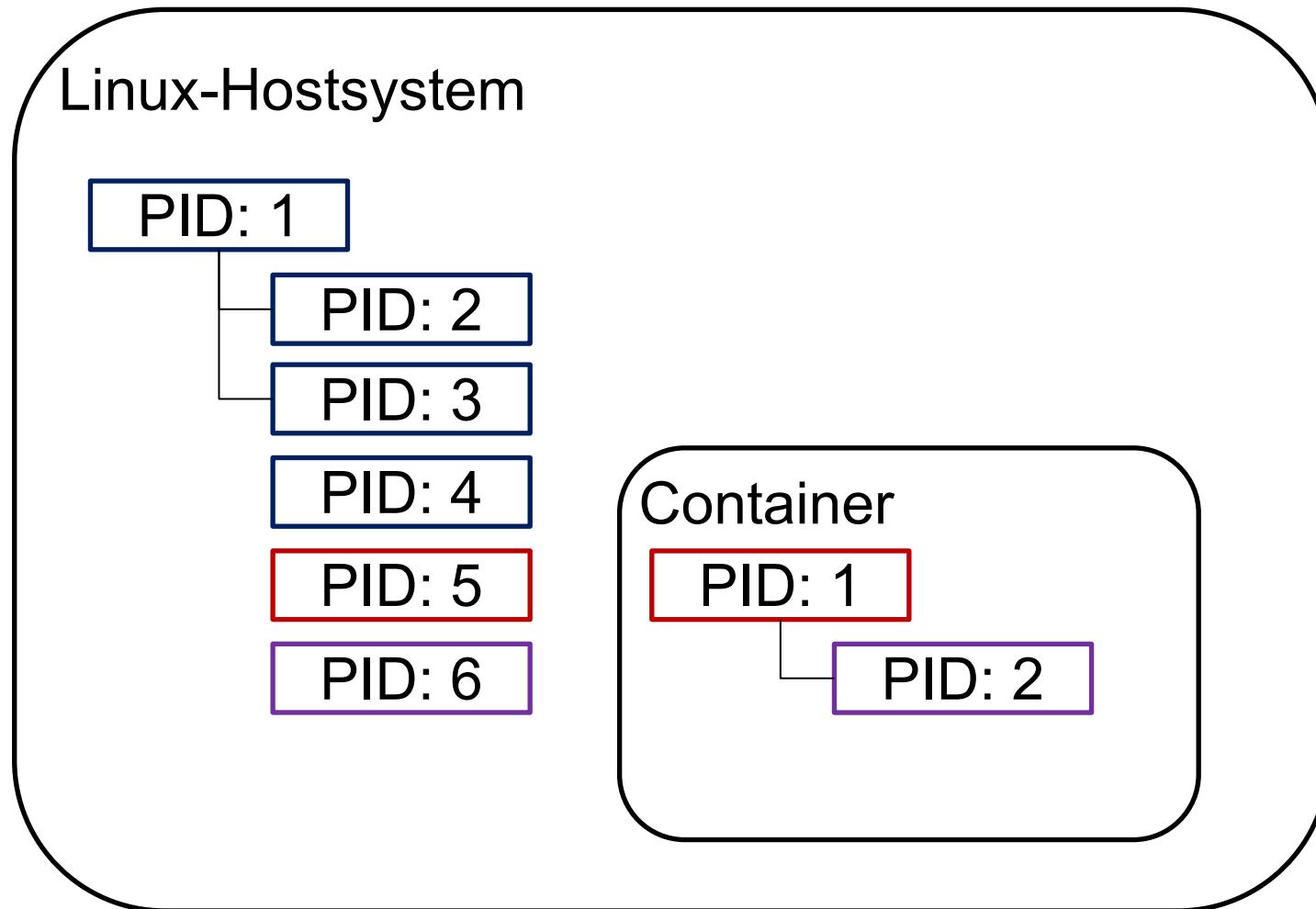
Was ist ein „Docker“-Container?

Container sind nur (Linux-)Prozesse



Process Namespaces

- Containerprozesse werden auf den Linux-Host abgebildet



Process Namespaces – Reales Beispiel

```
root      3077  1579  0 Feb25 ?  00:00:17 /usr/bin/docker-proxy -proto tcp -host-ip 0.0.0.0 -host-port 443 -container-ip 172.17.0.1 -container-port 443
root      3083  1579  0 Feb25 ?  00:00:16 /usr/bin/docker-proxy -proto tcp -host-ip :: -host-port 443 -container-ip 172.17.0.1 -container-port 443
root      3101  1  0 Feb25 ?  01:39:07 /usr/bin/containerd-shim-runc-v2 -namespace moby -id 9b96c066c6487a4f5bbc1c3f4ae7
root      3135  3101  0 Feb25 ?  00:28:11 registry serve /etc/docker/registry/config.yml
root      3174  1579  0 Feb25 ?  00:00:16 /usr/bin/docker-proxy -proto tcp -host-ip 0.0.0.0 -host-port 8000 -container-ip 172.17.0.1 -container-port 8000
root      3180  1579  0 Feb25 ?  00:00:16 /usr/bin/docker-proxy -proto tcp -host-ip :: -host-port 8000 -container-ip 172.17.0.1 -container-port 8000
```



Linux-Hostsystem

PID: 1

⋮

PID: 3083

PID: 3101

PID: 3135

⋮

Container

PID: 1

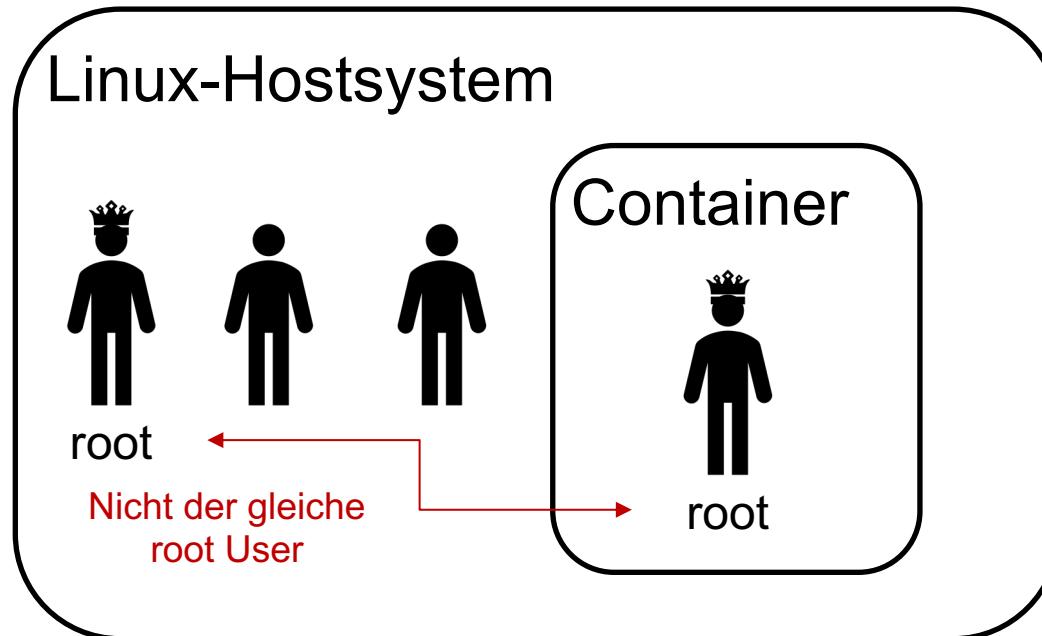
PID: 10

PID: 15

```
[notest@localhost reveal-slides-test]$ docker exec -it registry /bin/sh
/ # ps
PID  USER      TIME  COMMAND
 1 root      28:11  registry serve /etc/docker/registry/config.yml
 10 root      0:00   /bin/sh
 15 root      0:00   ps
```

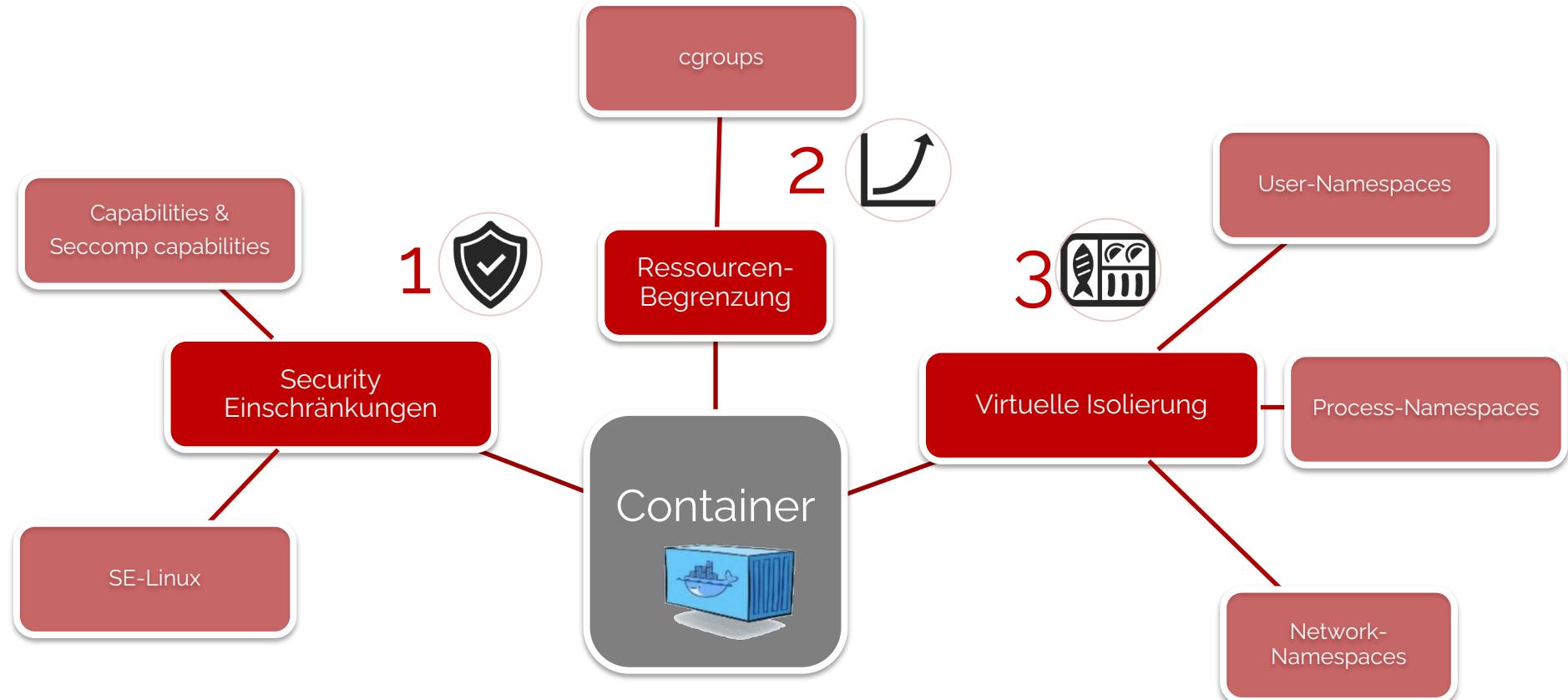
User Namespaces

- Standardmäßig läuft Docker mit dem Root-User und alle Containerprozesse werden als root-User ausgeführt
 - root User hat volle Berechtigungen (Capabilities) /usr/include/linux/capability.h
- Großes Sicherheitsrisiko (Privilege Escalation)
 - Container als unprivilegierten User starten
 - Container root-User auf weniger privilegierten Host-User mappen



Eigenschaften von Anwendungscontainern

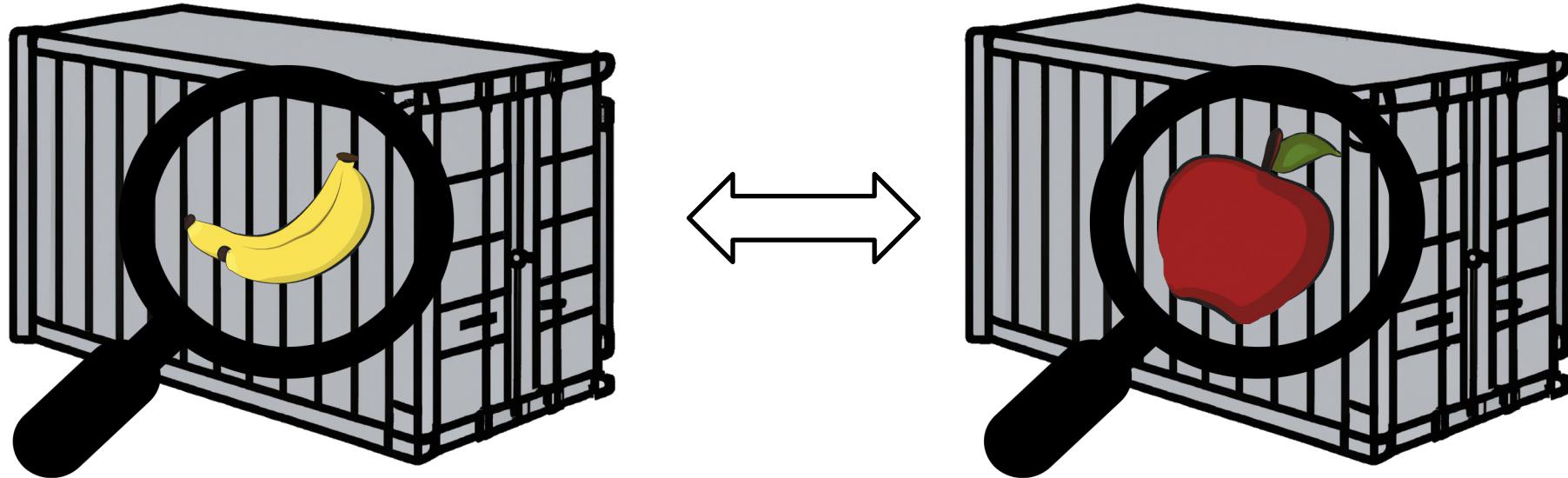
- Container sind lediglich isolierte Linux-Prozesse
- Eigenes Filesystem
- Hieraus ergeben sich einige Vorteile



Vorteile von Containerisierung

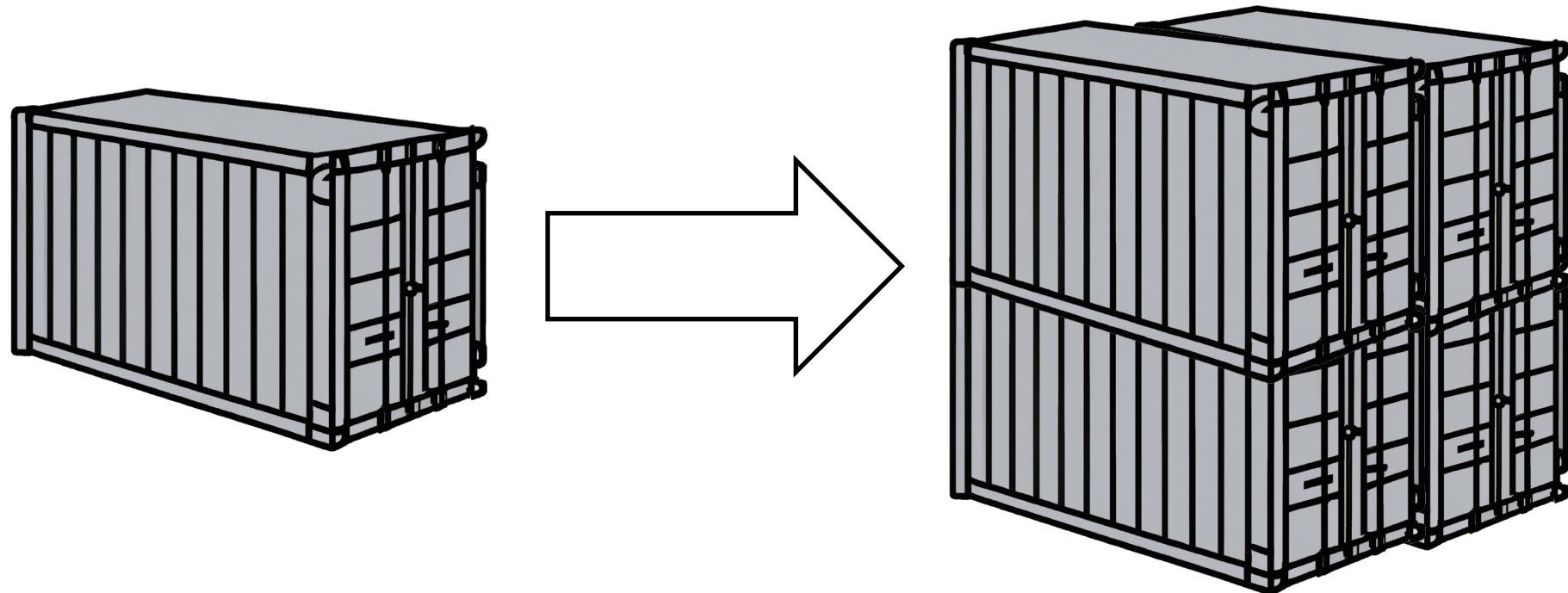
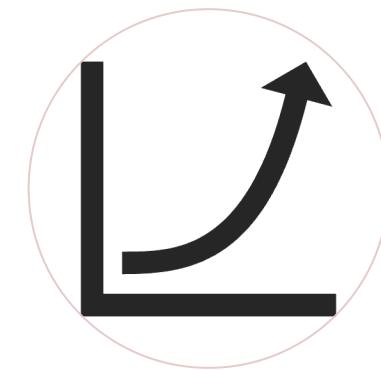
Vorteile von Containerisierung [1|6]

- Virtualisierung und Isolation



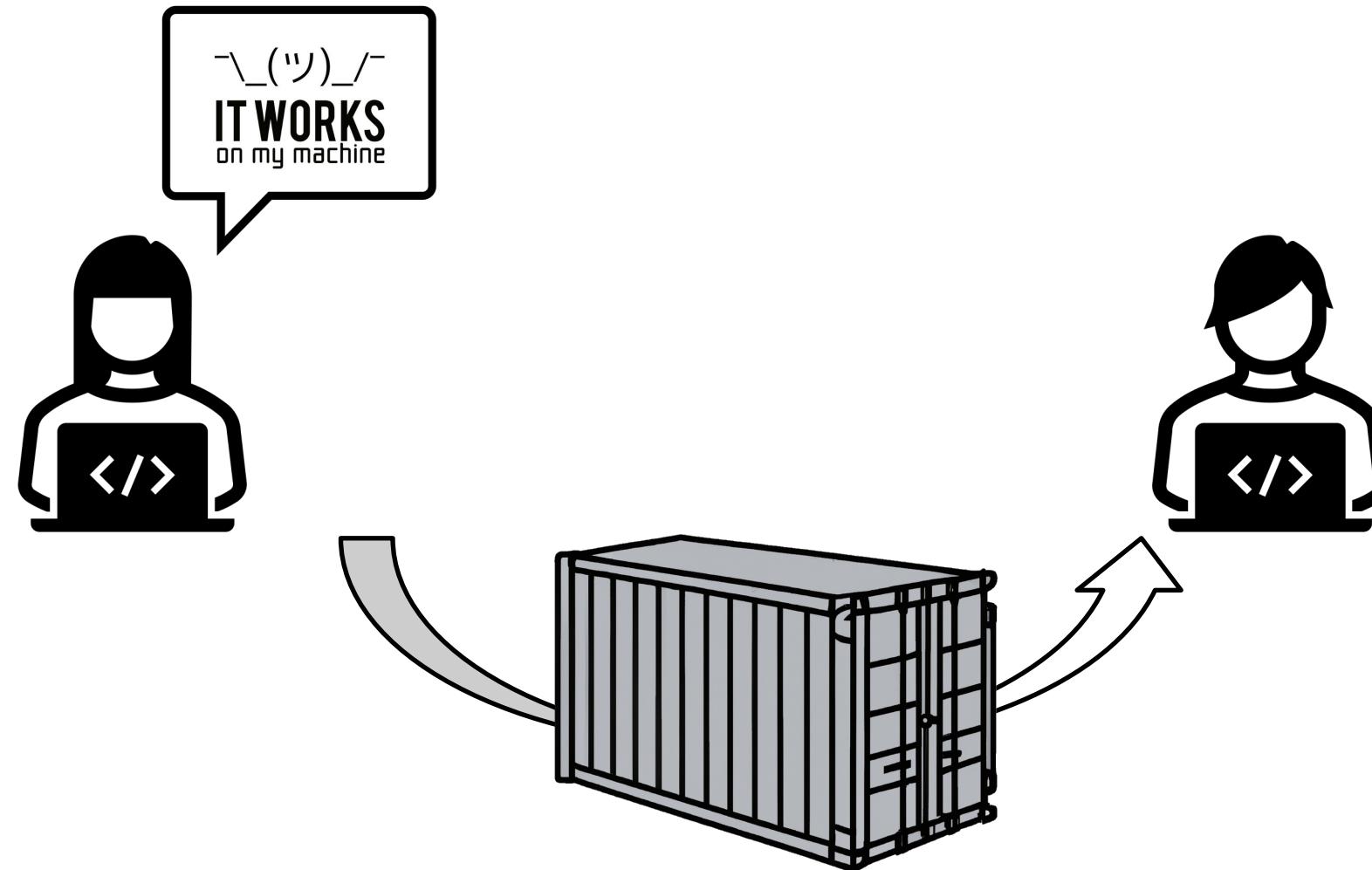
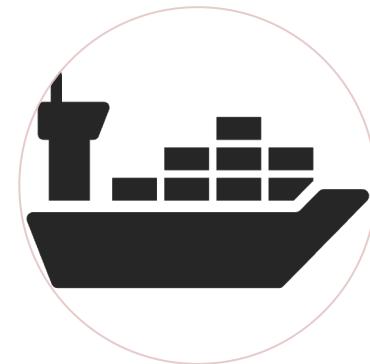
Vorteile von Containerisierung [2|6]

- Ressourcenbegrenzung & Skalierbarkeit



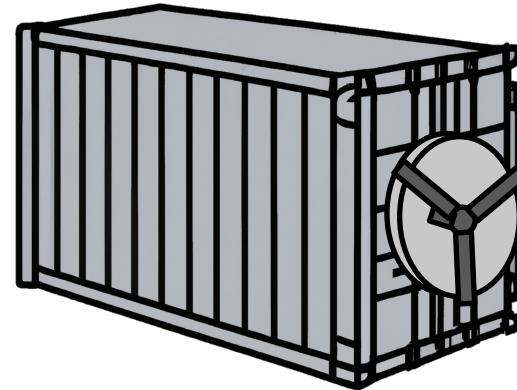
Vorteile von Containerisierung [3|6]

- Portierbarkeit



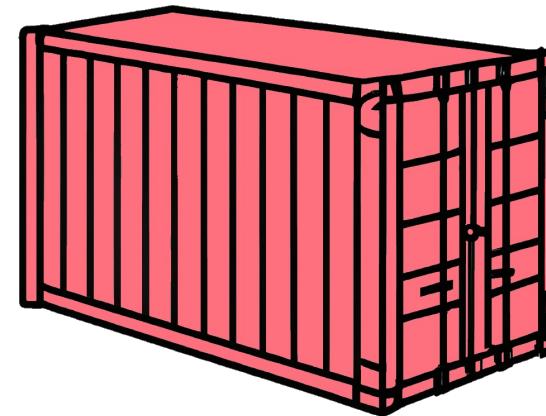
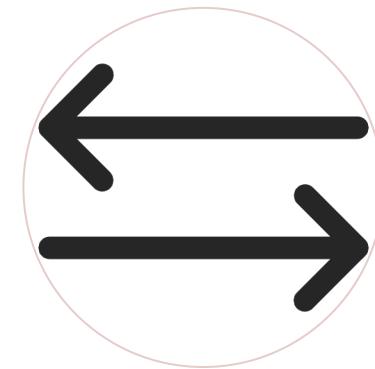
Vorteile von Containerisierung [4|6]

- Sicherheit (?)
 - Wenn alles richtig eingestellt ist



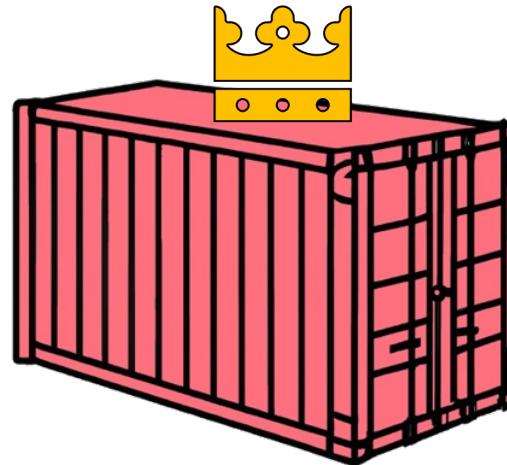
Vorteile von Containerisierung [5|6]

- Austauschbarkeit

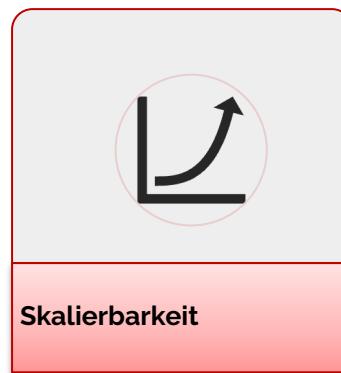


Vorteile von Containerisierung [6|6]

- Alle diese Vorteile machen Container zu einer Schlüsseltechnologie!



Virtualisierung
und Isolation



Skalierbarkeit



Sicherheit



Austauschbarkeit



Portierbarkeit

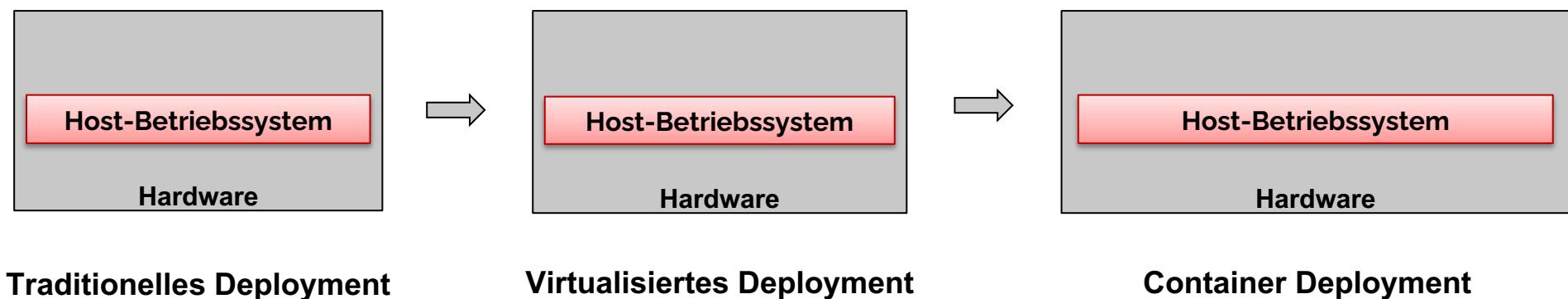
Isolierte Umgebung

- Containerprozesse sehen aus, wie ein eigenes kleines Betriebssystem
 - Jedoch wird der Kernel des Hostbetriebssystems zwischen den Containern geteilt
 - Hierzu gleich mehr...
- Per Terminal kann man sich auf einen Container verbinden
 - Hierdurch kann das Dateisystem des Containers navigiert werden
 - Linux Befehle sind möglich, sofern die entsprechenden Tools Teil des Container-Image sind
- Doch: „Ist das nicht das gleiche, wie eine leichtgewichtige VM?“

Virtualisierung vs. Containerisierung

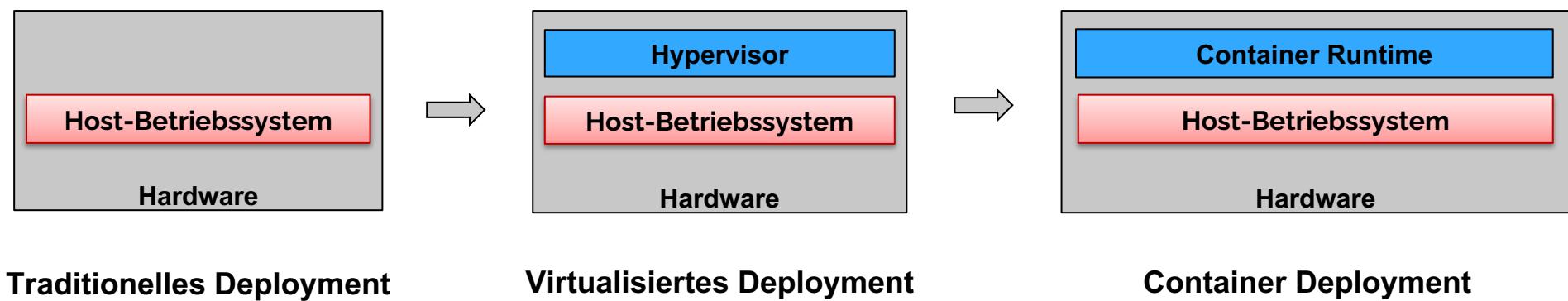
Virtualisierung vs. Containerisierung [1|4]

- Möchte man eine Anwendung betreiben, so braucht man eine „Maschine“ (physikalisch/virtuell)
- Software läuft immer auf einer physikalischen Hardware
- In der Regel benötigt man für Anwendungen auch ein Betriebssystem



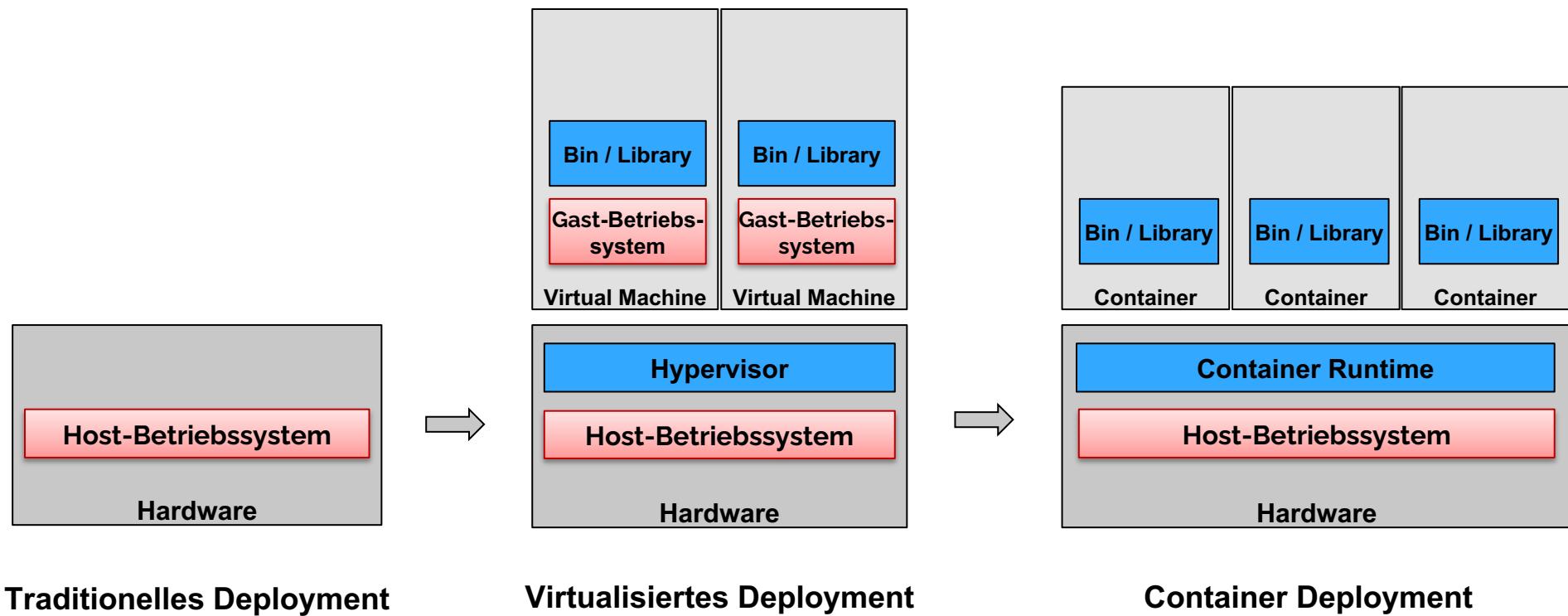
Virtualisierung vs. Containerisierung [2|4]

- VMs brauchen einen Hypervisor (z.B. Hyper-V)
 - Virtualisierungsfrage: Welche Hardware-Ressourcen werden zugewiesen?
- Container benötigen eine Container Runtime (z.B. runc)
 - Containerfrage: Welche Tools und welcher Code gehört zu einem Prozess?



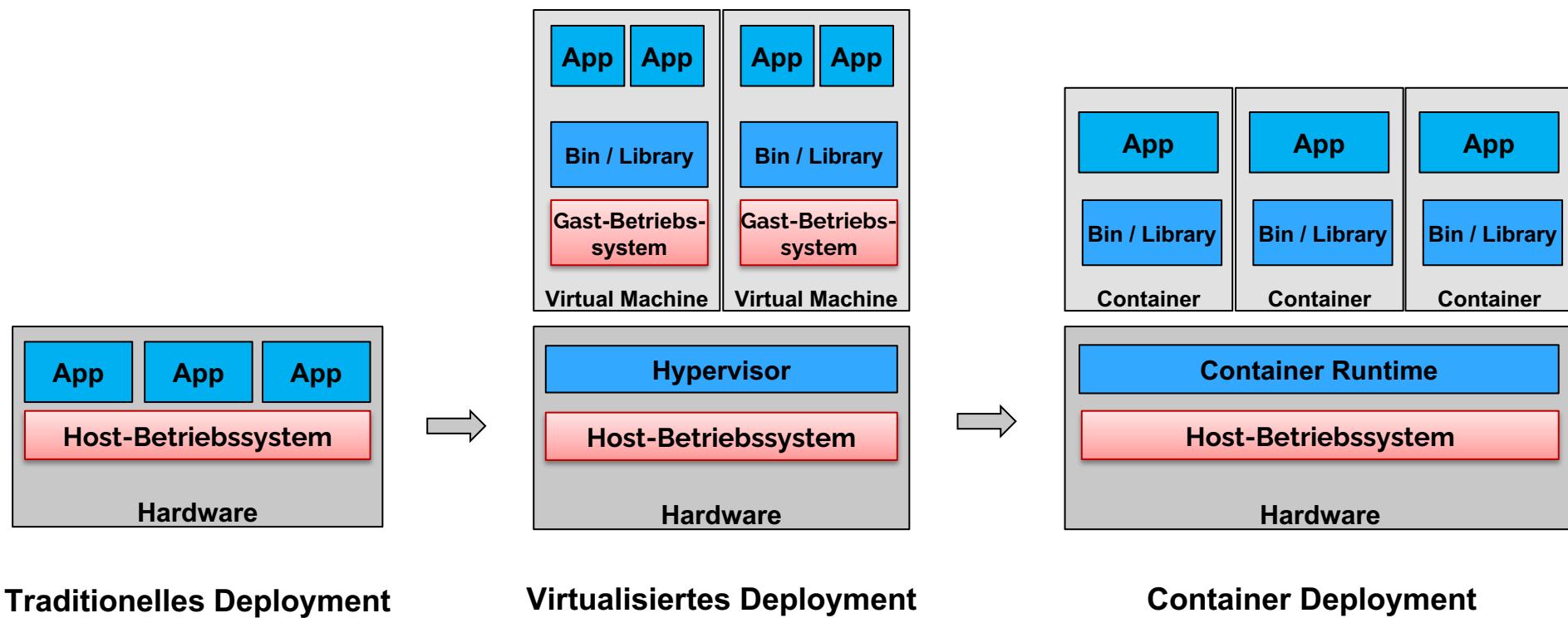
Virtualisierung vs. Containerisierung [3|4]

- VMs bringen ihren eigenen Kernel und eigenes Betriebssystem mit
- Container greifen auf Kernelfunktionen des Gast-Betriebssystems zu



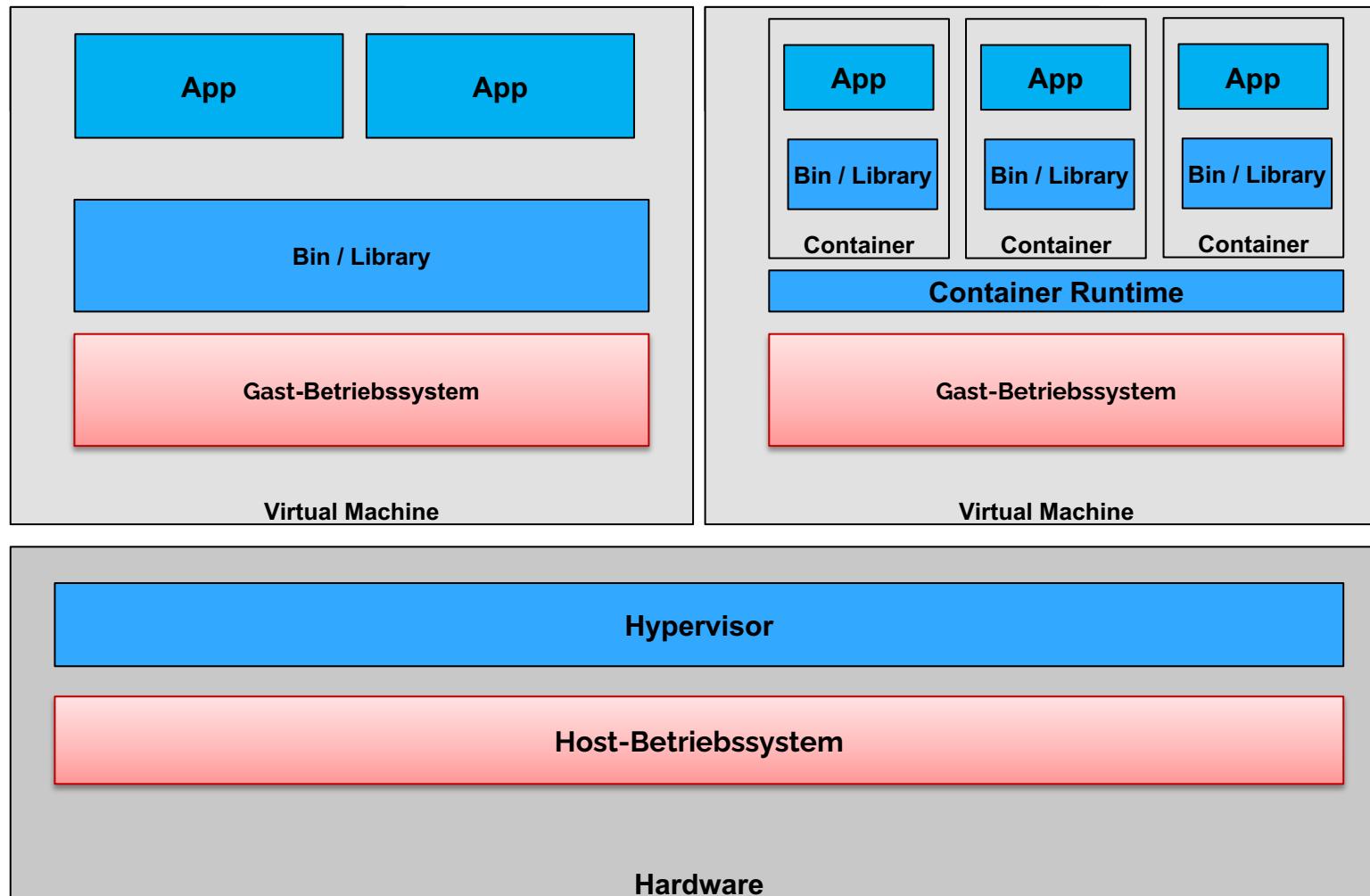
Virtualisierung vs. Containerisierung [4|4]

- „VM-Apps“ haben potentiell Zugriff auf alle Ressourcen der VM
- Container-Apps haben nur Zugriff auf containereigene Ressourcen



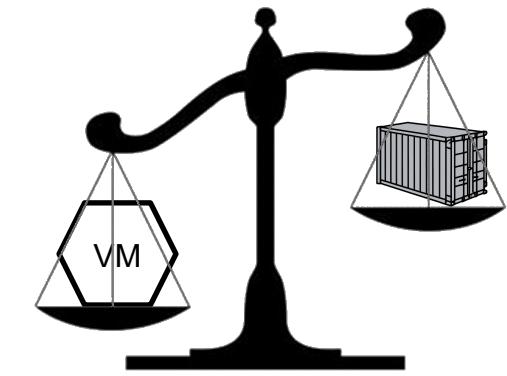
Virtualisierung vs. Containerisierung [1|2]

- Container können wiederum auf einer VM laufen



Virtualisierung vs. Containerisierung [2|2]

- VMs sind komplette Abbilder physikalischer Maschinen
 - Sitzen somit **auf Hardwareebene** auf
 - Verwaltende Instanz ist ein **Hypervisor**
 - Bringen einen eigenen Kernel und **eigenes Betriebssystem** mit
 - Zuweisung von Hardwareressourcen über Hypervisor
 - **Komplettes Filesystem** (X GB) innerhalb der VM ist potentiell für alle Prozesse **sichtbar**
 - Meist **lange Lebenszeiten** und **lange Startzeiten**
- Container sind leichtgewichtige Prozesse
 - Sitzen auf Betriebssystem-Schicht auf
 - Verwaltende Instanz ist eine **Container Runtime/Container Manager**
 - Benutzen Kernelfunktionen des **Gastbetriebssystems**
 - Benutzen CPU und RAM des Gastbetriebssystems
 - Container besitzen ein **eigenes kleines Filesystem (User Space)**
 - Meist **kurze Lebenszeiten** und **kurze Startzeiten**



Docker (& Co.)

Was ist Docker?

- Unternehmen Docker, Inc (ehemals dotCloud)
 - Verkauft an cloudControl (August 2014)
 - Verkauft an Mirantis (2019)
- Software
 - 13. März 2013 erschienen
 - In Go geschrieben
 - Open Source (CE, aber auch EE)
- Virtualisierung von
 - Linux
 - Auch auf Windows 10 (mit Hyper-V oder VirtualBox)
 - MacOS (mit HyperKit oder VirtualBox)
- als Docker Desktop verfügbar
- Ursprünglich in RHEL 7.0 enthalten, jedoch in Version 8 durch podman ersetzt



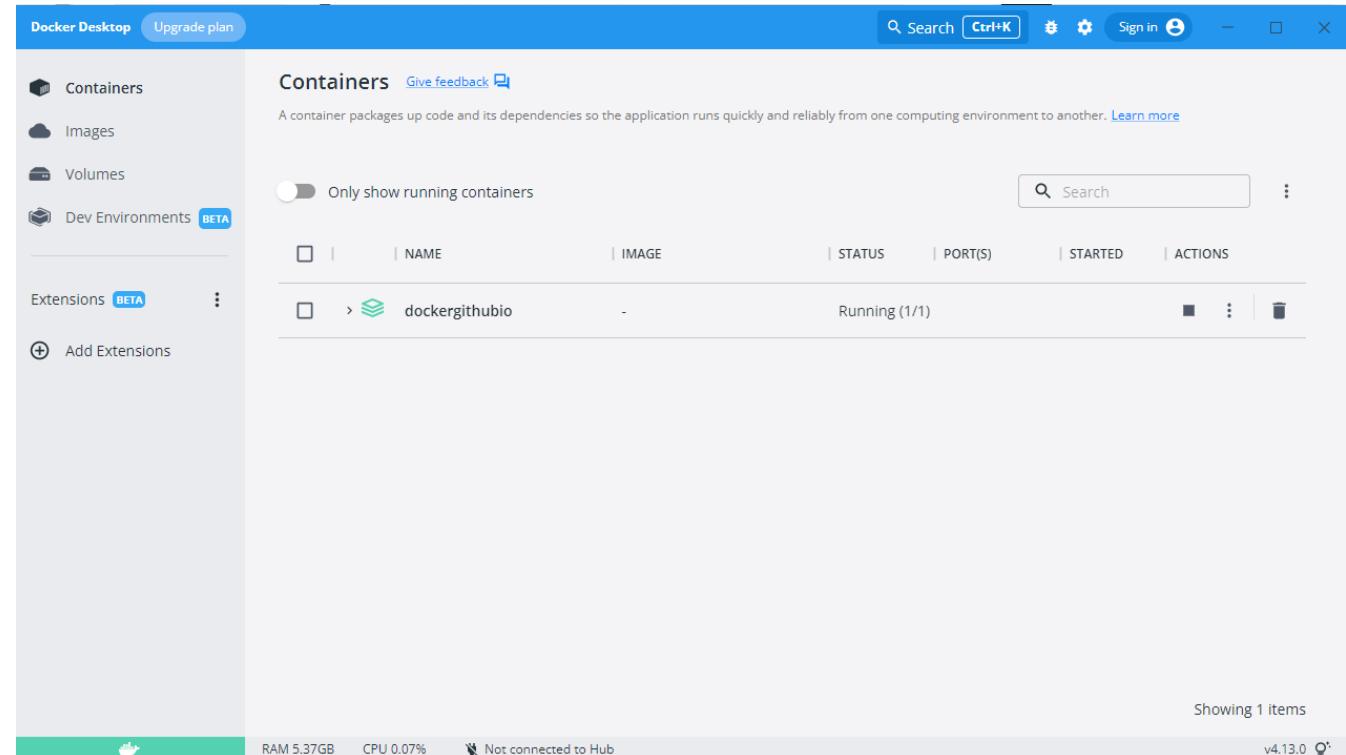
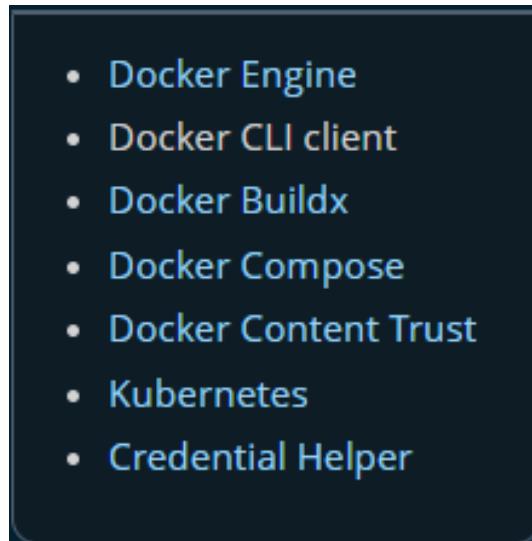
Platform	x86_64 / amd64	arm64 / aarch64	arm (32-bit)	s390x
CentOS	✓	✓		
Debian	✓	✓	✓	
Fedora	✓	✓		
Raspbian			✓	
RHEL				✓
SLES				✓
Ubuntu	✓	✓	✓	✓
Binaries	✓	✓	✓	

Docker Historie

- Veröffentlicht März 2013
- Sep. 2013 Zusammenarbeit mit RedHat
- 2014:
 - Strategische Partnerschaft mit IBM
 - Microsoft kündigt Docker Integration in Windows Server 2016 an
- 2015 „Open Container Initiative“ zu Hersteller- und Betriebssystem-unabhängigen Standards für Containersysteme
 - Zusammen mit IBM, CoreOS, Google, Microsoft, Amazon, Cisco, EMC, Fujitsu Limited, Goldman Sachs, HP, Huawei, Intel, Joyent, Mesosphere, Pivotal, Rancher Labs, Red Hat und Vmware
- 2017 Aufteilung in Community Edition (CE) und Enterprise Edition (EE) sowie Anpassung der Versionsnummern: 1.13 = 17.03 (Aktuell: **24.0.7**, 26.Oktober 2023)

Docker Produkte

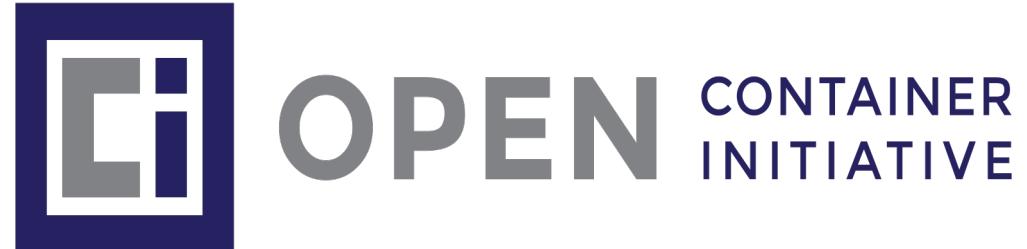
- Docker Engine
- Docker Desktop



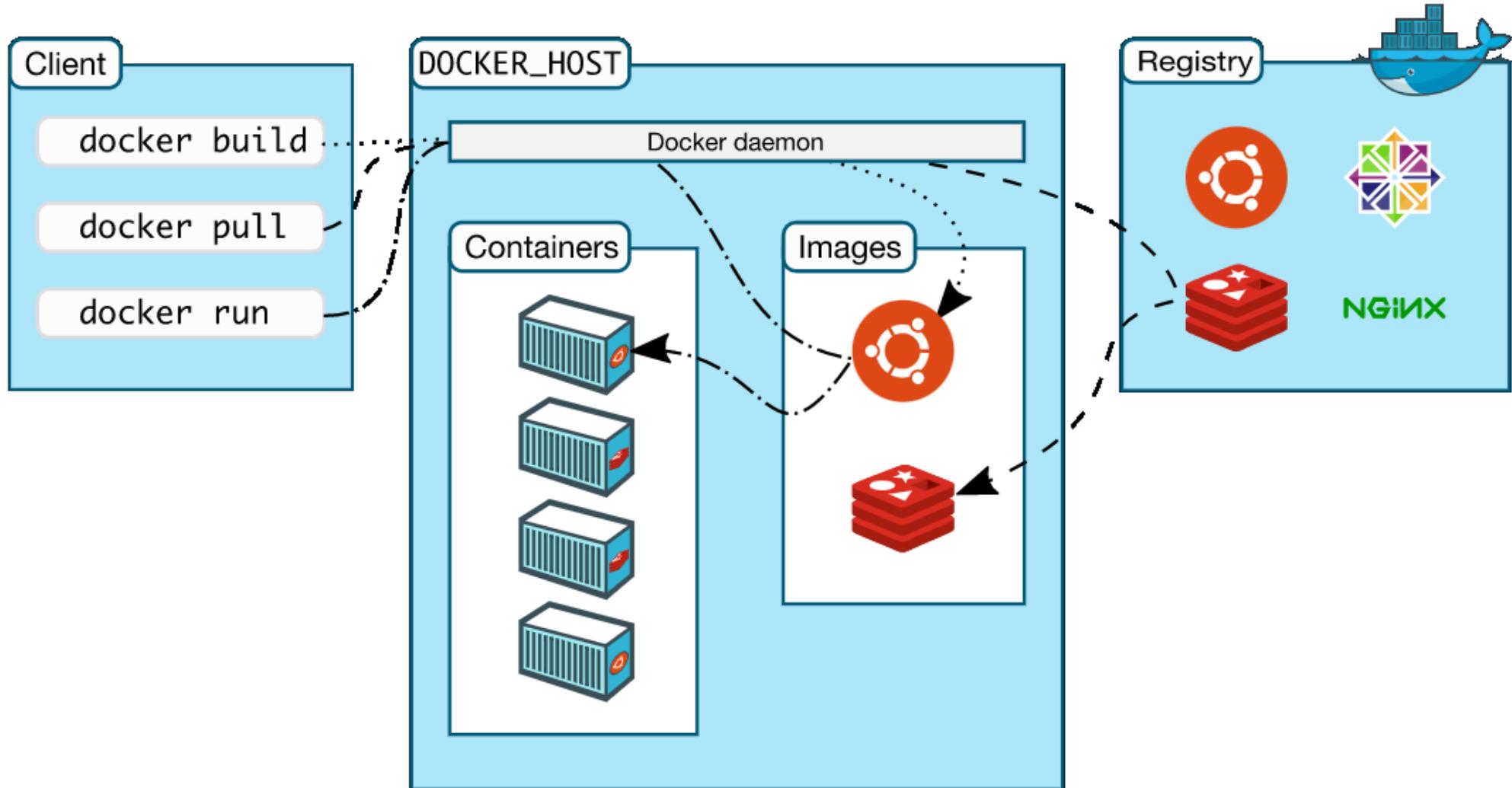
- Rootless
 - Kein Dämon-Prozess, ähnlich wie bei Podman
 - Verwendung etwas umständlicher
- Docker Hub

Open Container Initiative (OCI)

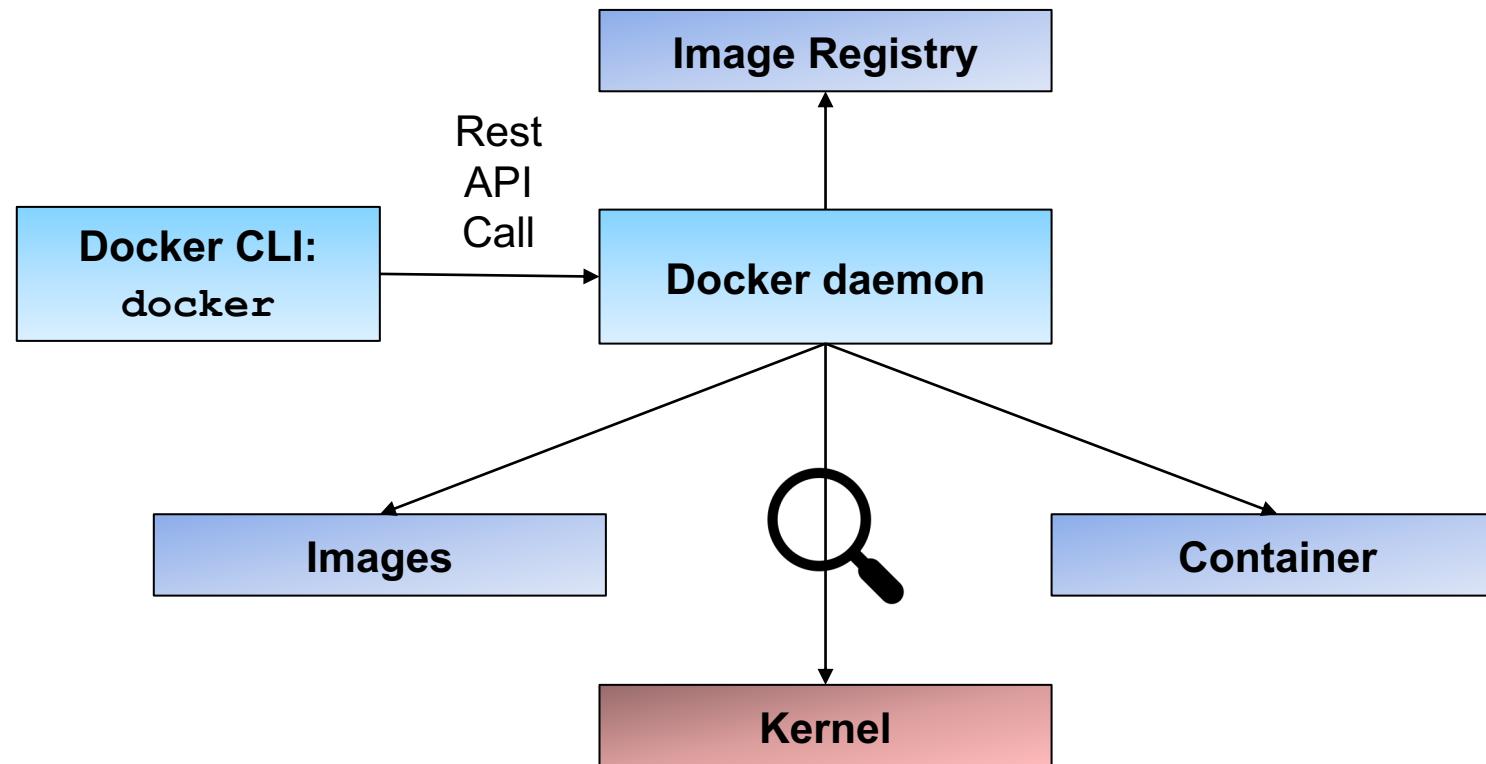
- Projekt der Linux Foundation
 - Juni 2015 u.a. von Docker und CoreOS
- Ziel:
 - Entwicklung offener Standards für Containerformate und deren Laufzeitumgebungen
- Neuigkeiten werden auf der CloudNativeCon vorgestellt
- Aktueller Entwicklungsstand: Drei Spezifikationen
 - image-spec
 - runtime-spec
 - (distribution-spec)



Docker Engine Architektur



Docker Engine Architektur (Vereinfacht)



Container Engines (Container Runtime Engine)

- Docker

- Containerimplementierung seit 2013
- Mittels Hyper-V oder VirtualBox auch auf Windows lauffähig
- Umfasst auch Tools, Container Registry, Docker-daemon



- podman

- Seit 2019 Bestandteil von RHEL 8, sowie Suse und Fedora
- Leichtgewichtige Implementierung ohne Dämon-Prozess
- Kommandokompatibel mit Docker



- Rocket (rkt)

- Kein Dämonprozess
- Customizable Isolation (Isolation Parameters)
- Pod als kleinste Einheit



Container Manager

- containerd
 - Ursprünglich als Teil von Docker entwickelt
 - In Funktionsumfang gewachsen
 - CRI-kompatibel

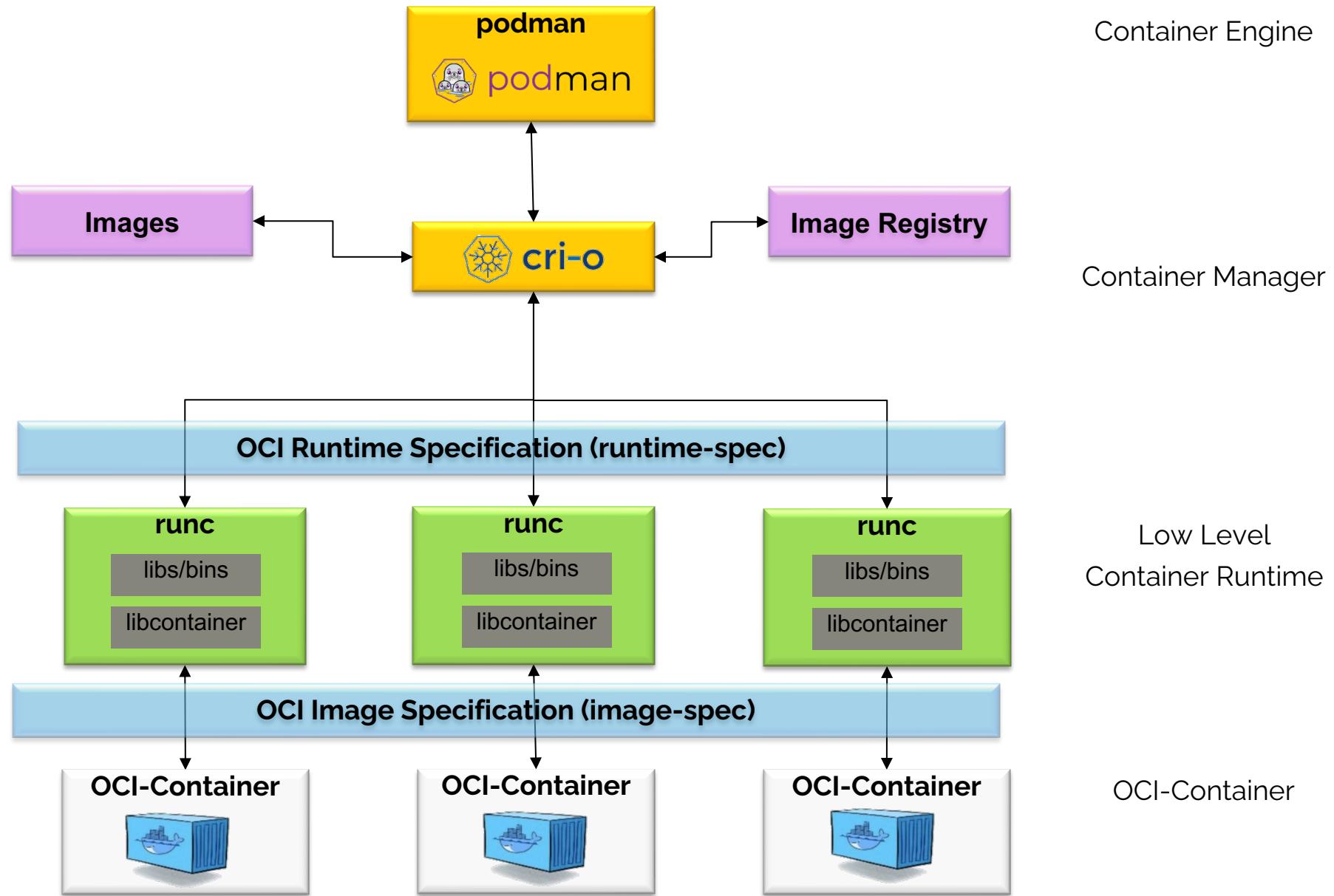


- CRI/CRI-O (RedHat)

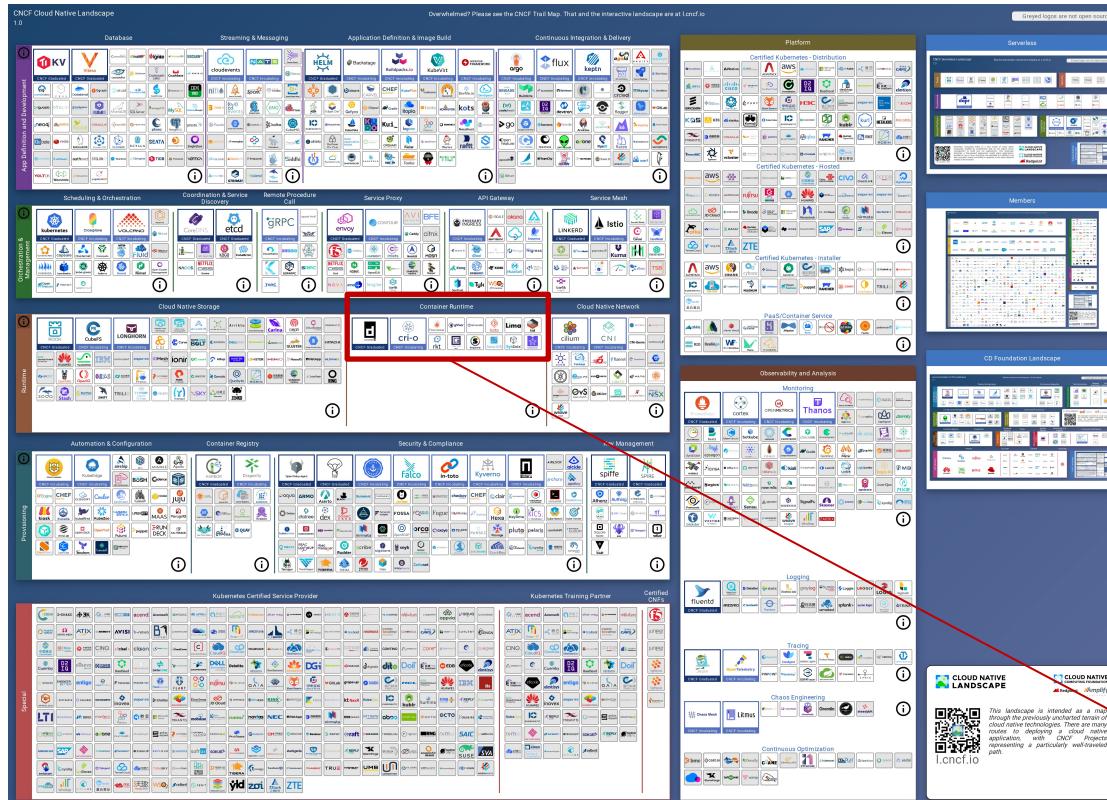
- Container Runtime Interface (OpenShift)
- leichtgewichtig
- Optimiert für Kubernetes
- Funktioniert mit runc, Kata



Docker Alternativen



Container Runtime Implementierungen



Container Runtime



Kontrollfragen: Grundlagen

- Was ist der Unterschied zwischen Virtualisierung und Containerisierung?
- Was ist der Unterschied zwischen Virtual Machines und Containern?
- Welche 3 Eigenschaften müssen Container als Linux-Prozess aufweisen?
- Nennen Sie Vorteile von containerisierten Deployments gegenüber traditionellen Deployments
- Welche Spezifikationen werden durch die OCI definiert?
- Welche Aufgaben übernimmt dockerd im Vergleich zu containerd?
- Was ist ein Container Lifecycle Management?



Kontrollfragen: Grundlagen

- Was ist der Unterschied zwischen Container Engine, Container Managern, Container Runtime?
- Wird Docker benötigt, um Kubernetes zu verwenden?

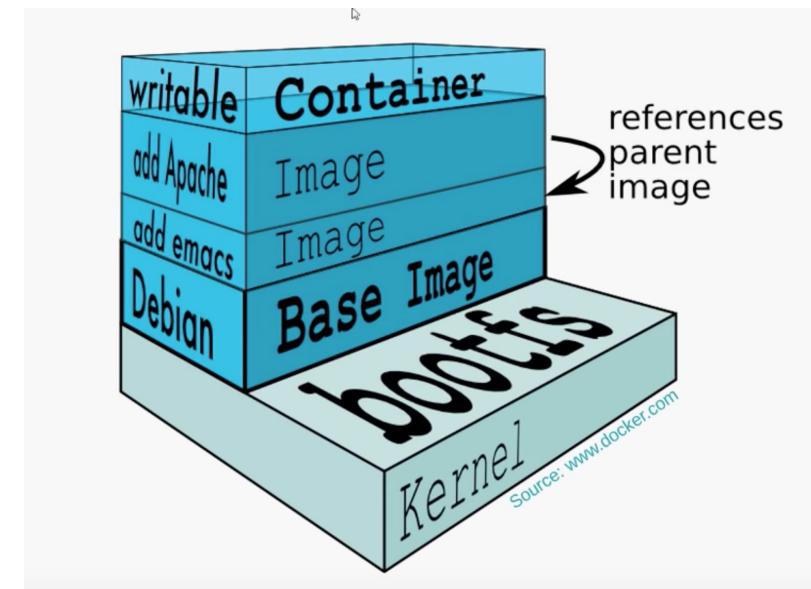
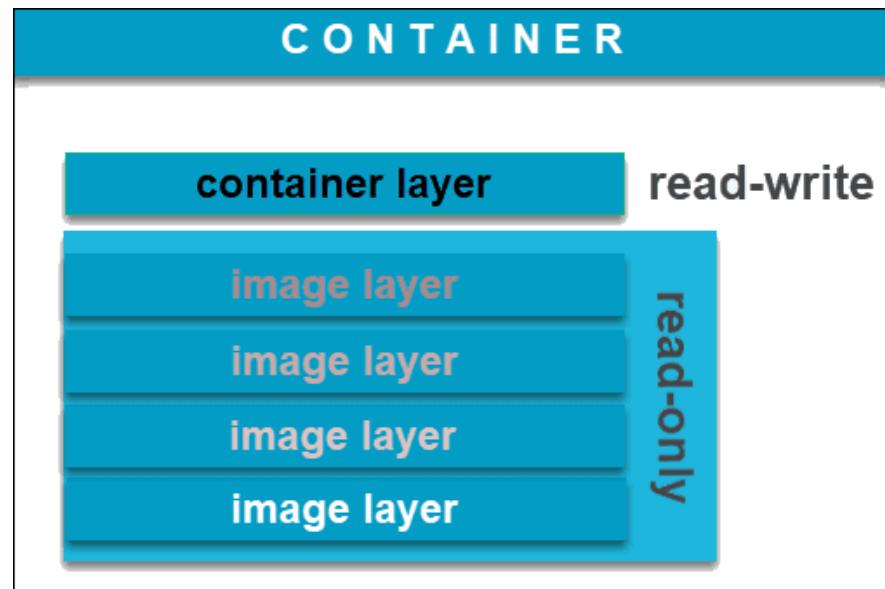
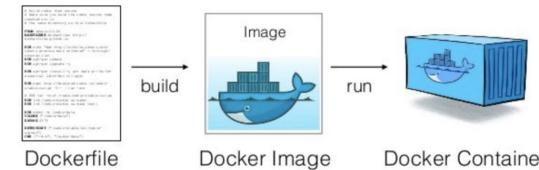


Docker Inbetriebnahme Container

Docker Container

- Ein Docker Container

- ist die ablauffähige Umsetzung eines Docker Images
- wird durch das Kommando „`docker create`“ aus einem Image erzeugt
- erhält beim Anlegen einen zusätzlichen beschreibbaren („read-write“) Layer. Die im zugrundeliegenden Image vorhandenen Layer bleiben unverändert „read-only“.



Docker Container Commands

- docker container <command>

```
[notest@localhost ~]$ docker container --help

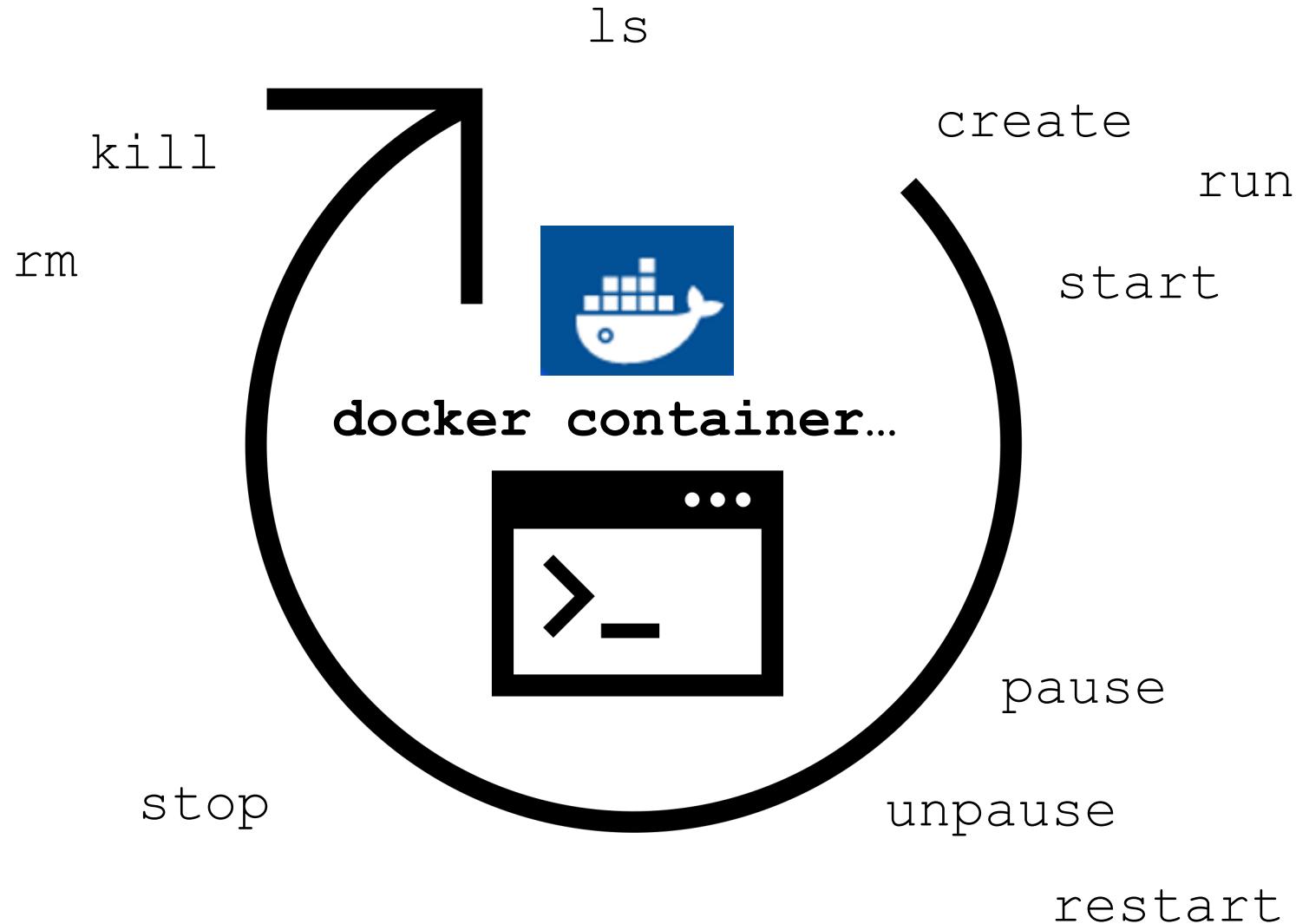
Usage: docker container COMMAND

Manage containers

Commands:
  attach          Attach local standard input, output, and error streams to a running container
  commit          Create a new image from a container's changes
  cp              Copy files/folders between a container and the local filesystem
  create          Create a new container
  diff            Inspect changes to files or directories on a container's filesystem
  exec            Run a command in a running container
  export          Export a container's filesystem as a tar archive
  inspect         Display detailed information on one or more containers
  kill             Kill one or more running containers
  logs            Fetch the logs of a container
  ls              List containers
  pause           Pause all processes within one or more containers
  port            List port mappings or a specific mapping for the container
  prune           Remove all stopped containers
  rename          Rename a container
  restart         Restart one or more containers
  rm              Remove one or more containers
  run              Run a command in a new container
  start           Start one or more stopped containers
  stats           Display a live stream of container(s) resource usage statistics
  stop            Stop one or more running containers
  top             Display the running processes of a container
  unpause         Unpause all processes within one or more containers
  update          Update configuration of one or more containers
  wait            Block until one or more containers stop, then print their exit codes

Run 'docker container COMMAND --help' for more information on a command.
```

CLI: Verwalten von Containern



Container Lebenszyklus: CLI

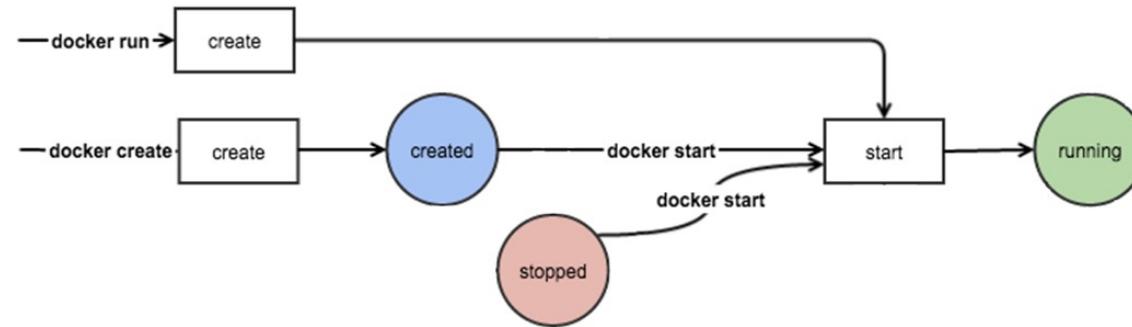
- `docker ps [OPTIONS]`

- Erstellen
 - `docker create [OPTIONS] CONTAINER [CONTAINER...]`
 - `docker start [OPTIONS] CONTAINER [CONTAINER...]`
 - `docker run [OPTIONS] CONTAINER [CONTAINER...]`

- Pausieren und Neustarten
 - `docker pause [OPTIONS] CONTAINER [CONTAINER...]`
 - `docker unpause [OPTIONS] CONTAINER [CONTAINER...]`
 - `docker restart [OPTIONS] CONTAINER [CONTAINER...]`

- Beenden
 - `docker stop [OPTIONS] CONTAINER [CONTAINER...]`
 - `docker kill [OPTIONS] CONTAINER [CONTAINER...]`
 - `docker rm [OPTIONS] CONTAINER [CONTAINER...]`

Phase 1: docker create/start/run



- `docker create [OPTIONS] IMAGE [COMMAND] [ARGS...]`

```
$ docker create --name my_container busybox:latest
```

- `docker start [OPTIONS] CONTAINER [CONTAINER...]`

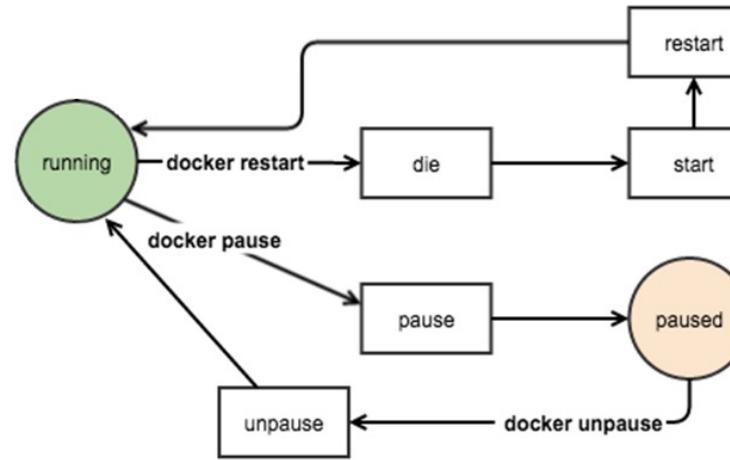
```
$ docker start my_container
```

- `docker run [OPTIONS] IMAGE [COMMAND] [ARGS...]`

- Erstellt den Container und startet ihn

```
$ docker run --name my_container busybox:latest
```

Phase 2: docker pause/unpause/restart



- docker pause CONTAINER [CONTAINER...]

```
$ docker pause my_container
```

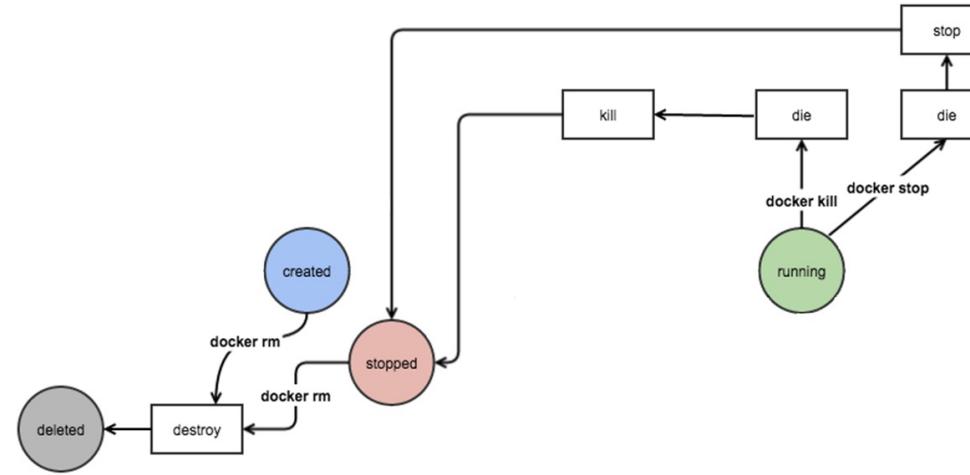
- docker unpause CONTAINER [CONTAINER...]

```
$ docker unpause my_container
```

- docker restart [OPTIONS] CONTAINER [CONTAINER...]

```
$ docker restart my_container
```

Phase 3: docker stop/kill/rm



- `docker stop [OPTIONS] CONTAINER [CONTAINER...]`

```
$ docker stop my_container
```

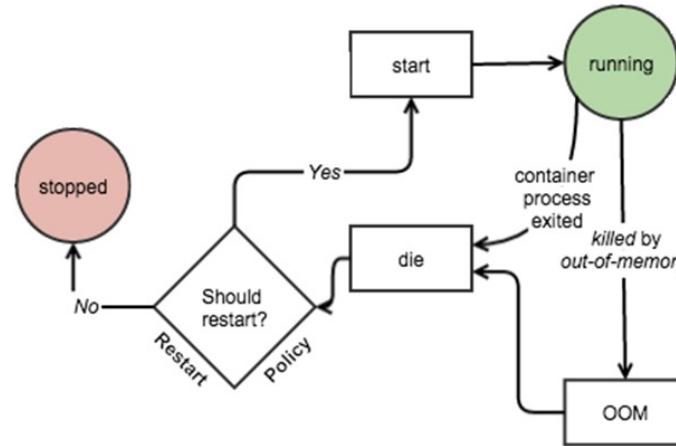
- `docker kill [OPTIONS] CONTAINER [CONTAINER...]`

```
$ docker unpause my_container
```

- `docker rm [OPTIONS] CONTAINER [CONTAINER...]`

```
$ docker rm my_container
```

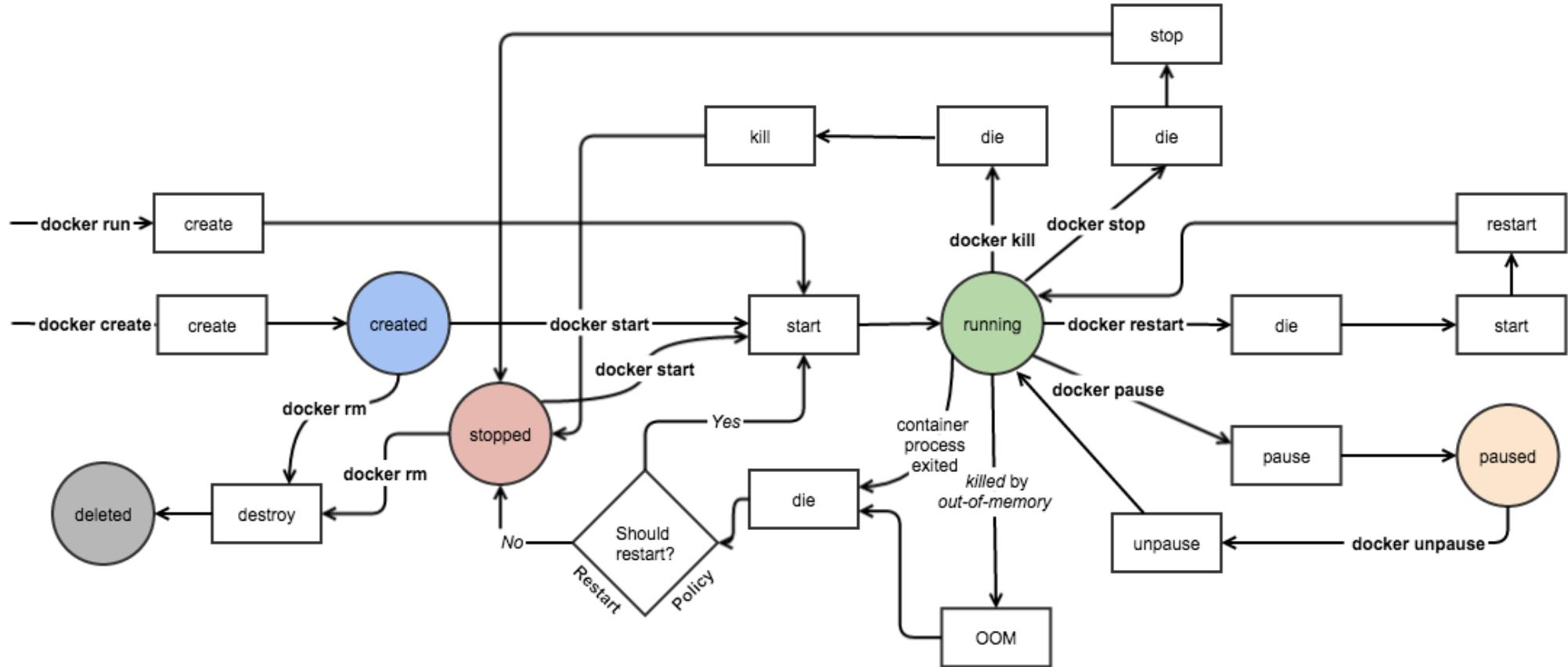
Phase 3: Beendigung ohne Befehl



- Container können auch ohne Befehle gestoppt werden
 - Out-of-memory
 - Hauptprozess wird beendet
- Restart-Policy gibt Verhalten in diesem Fall an
 - `docker run --restart=always CONTAINER [CONTAINER...]`

```
$ docker run --restart=always my_container
```

Container Lebenszyklus



Übung



Inbetriebnahme

Wichtigster Befehl: docker run

- `docker run [OPTIONS] IMAGE [COMMAND] [ARG...]`
 - Startet eine Instanz eines Images, genauer: einen Container zu einem Image
 - Fügt dem Container eine weitere Schicht hinzu, die jedoch schreibbar ist
 - Zahlreiche Optionen zum Setzen von Laufzeiteigenschaften

```
$ docker run -t -i --rm ubuntu bash
root@bc338942ef20:/# mount -t tmpfs none /mnt
mount: permission denied
```

```
$ docker run -t -i --privileged ubuntu bash
root@50e3f57e16e6:/# mount -t tmpfs none /mnt
root@50e3f57e16e6:/# df -h
Filesystem      Size  Used Avail Use% Mounted on
none            1.9G    0   1.9G   0% /mnt
```

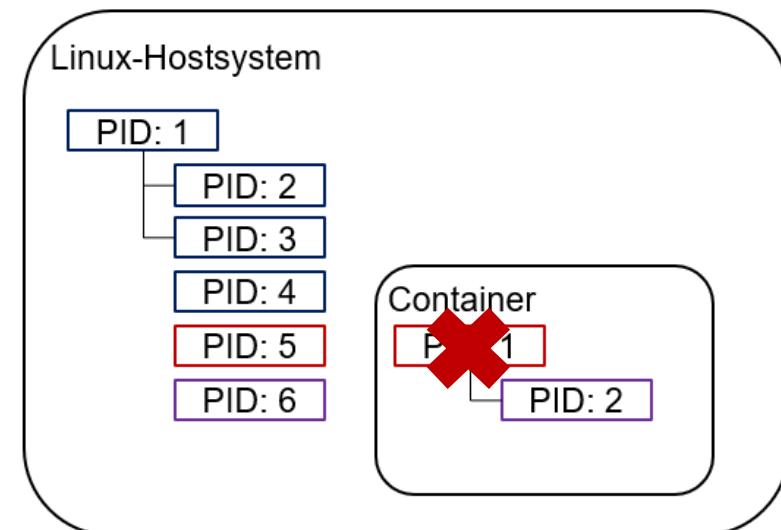
```
$ docker run -t -i --privileged ubuntu bash
```

Docker Image Commands

- Beim Ausführen von `docker run` oder `docker create` wird geprüft, ob das Image lokal vorliegt.
- Falls ja, wird eine Instanz des Images angelegt
- Falls nein, wird das Image gepullt
 - Es werden nur die Image Layer gepullt die lokal nicht vorhanden sind

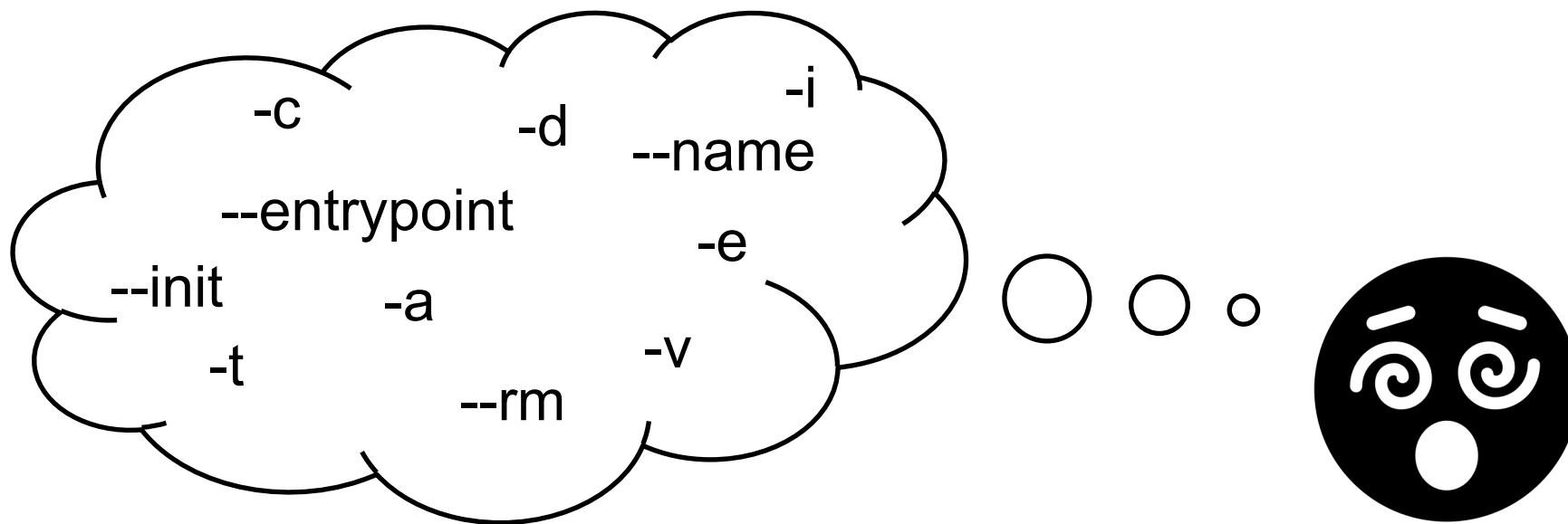
Hauptprozess/Einstiegspunkt

- 2 Möglichkeiten einen neuen Container zu erstellen:
 - docker create [OPTIONS] IMAGE [COMMAND] [ARGS...]
 - docker run [OPTIONS] IMAGE [COMMAND] [ARGS...]
- Jeder Container benötigt für die Instanziierung:
 - Ein Image
 - Einen Hauptprozess/Einstiegspunkt
- Der Hauptprozess hat die interne Process ID PID 1
- Wenn der Hauptprozess beendet wird, stoppt der Container



docker create/run Optionen

- Es gibt sehr viele Optionen (fast 100)
 - Diese Optionen zu kennen und verstehen ist die Quintessenz von Docker
 - Innerhalb einer Schulung nicht möglich, alle Optionen vorzustellen
 - Die wichtigsten Optionen werden sukzessive in den Übungen erklärt und praktisch angewendet
 - Cheatsheets helfen, dabei Befehle schnell wiederzufinden



Kurzer Exkurs: Port Mapping -p, -P [1|2]

- Jeder Container kann Ports zur Netzwerkkommunikation öffnen
 - Interne Ports sind nur innerhalb des Containers sichtbar
- Mit -p können interne Container Ports explizit auf die Ports des Hostsystems gemappt werden, z.B.:
 - `docker run -p <Host-Port>:<ContainerPort> nginx`
- Mit -P werden interne Ports auf zufällige Host Ports gemappt
 - `docker run -P nginx`



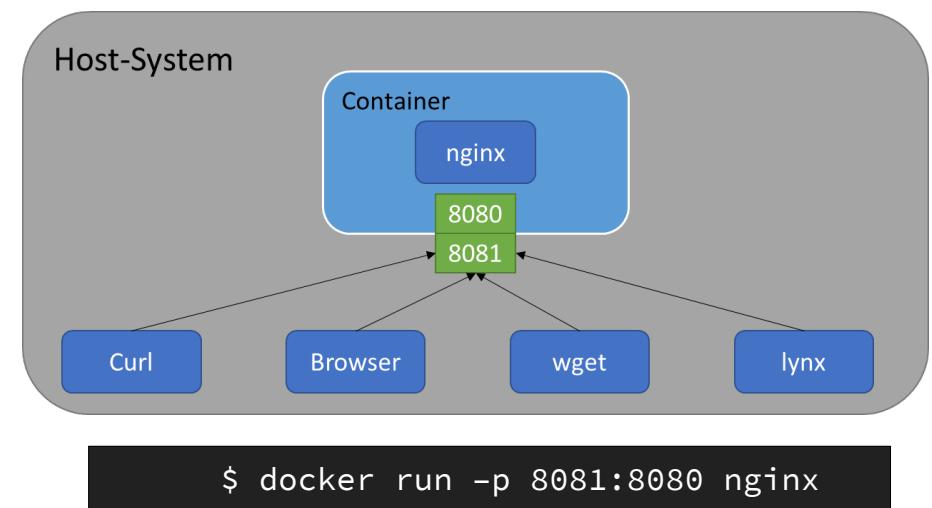
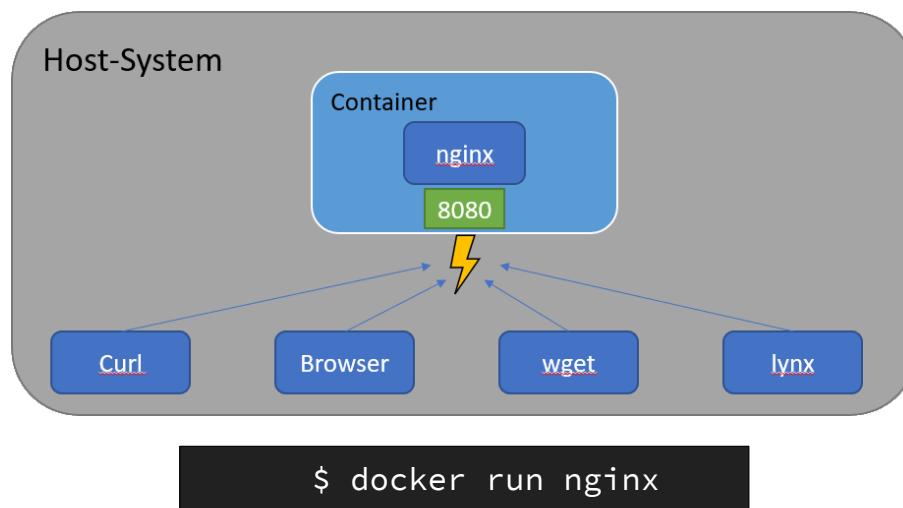
```
$ docker run nginx
```



```
$ docker run -p 8081:8080 nginx
```

Kurzer Exkurs: Port Mapping -p, -P [2|2]

- Ohne die Option -p können externe Dienste nicht auf den Containerprozess zugreifen.
 - Es fehlt die externe Schnittstelle



Kurzer Exkurs: Weitere Optionen

- -d
 - Container wird als Hintergrundprozess (Daemon) gestartet
- --rm
 - Container wird nach Beendigung entfernt
- --name
 - Zuweisung eines selbstgewählten Namens
- -it
 - Interaktiver Modus mit TTY
- -a
 - attach: um STDOUT zu empfangen

Kontrollfragen: Inbetriebnahme Container

- Was sind Container Layer? Welcher Layer sind schreibbar?
- Nennen Sie ein paar der wichtigsten Lifecycle-Befehle für Container.
- Was ist der Einstiegspunkt eines Containers?
- Was ist Port Mapping?
- Was ist der Unterschied zwischen docker run und docker start?



Übung



Container

Docker Inbetriebnahme

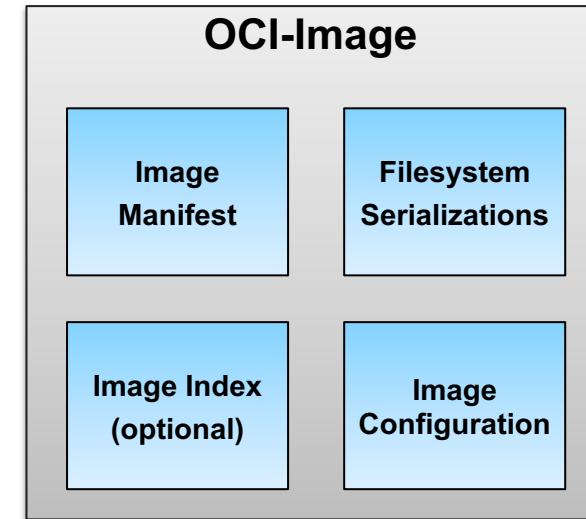
Images

Image Spec

- Ein OCI-Image besteht aus den folgenden Dateien:

- Image Manifest
- Image Configuration
- Filesystem Serialization (Layer Information)
- Image Index (optional)

- wird heruntergeladen
- auf dem Filesystem entpackt
- von der Laufzeitumgebung ausgeführt.

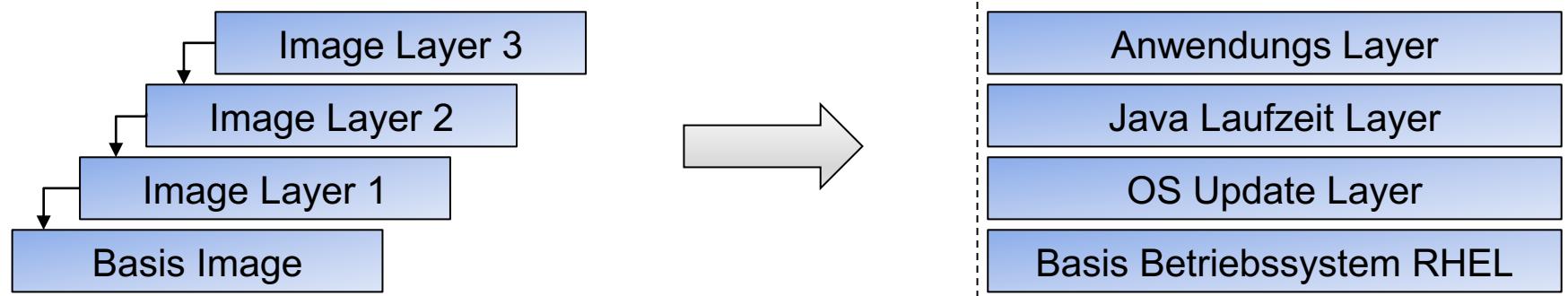
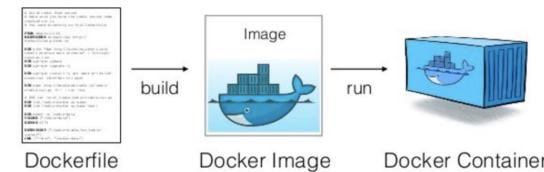


- Jede Schicht besitzt eine gehashte Kennung (z.B. 502c67ff11...)
- Eindeutige Zuordnung von Images
- Wichtig für Schichtweisen Aufbau von Images

Docker Images

■ Ein Docker Image

- ist eine **nicht veränderbare** Datei, die alle Komponenten enthält, welche ein Programm zum Ablauf benötigt (Code, Bibliotheken, Werkzeuge, ...)
- kann selbst **nicht ausgeführt** werden
- Aufbau wird beschrieben durch eine Datei „Dockerfile“
- ist in Schichten (Layern) aufgebaut, so dass komplexere Images von einfacheren abgeleitet werden



- Funktioniert wie ein Template zur Generierung von Containern

Docker Image Commands

- docker image <command>

```
[notest@localhost ~]$ docker image --help

Usage: docker image COMMAND

Manage images

Commands:
  build      Build an image from a Dockerfile
  history    Show the history of an image
  import     Import the contents from a tarball to create a filesystem image
  inspect    Display detailed information on one or more images
  load       Load an image from a tar archive or STDIN
  ls         List images
  prune     Remove unused images
  pull       Pull an image or a repository from a registry
  push       Push an image or a repository to a registry
  rm        Remove one or more images
  save      Save one or more images to a tar archive (streamed to STDOUT by default)
  tag       Create a tag TARGET_IMAGE that refers to SOURCE_IMAGE

Run 'docker image COMMAND --help' for more information on a command.
```

CLI: Verwalten von Images – docker images

- docker image ls [OPTIONS] [REPOSITORY[:TAG]] oder
- docker images [OPTIONS] [REPOSITORY[:TAG]]
 - Zeigt Liste aller Images (unter Berücksichtigung der optionalen Parameter)

```
$ docker images java
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
java	8	308e519aac60	6 days ago	824.5 MB
java	7	493d82594c15	3 months ago	656.3 MB
java	latest	2711b1d6f3aa	5 months ago	603.9 MB

```
$ docker images java:8
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
java	8	308e519aac60	6 days ago	824.5 MB

Docker Image Registry

- Die **Docker Image Registry**
 - ist eine Anwendung, zur Ablage von Docker Images
 - wird über das Docker CLI bedient
- `docker pull myregistrydomain:<port>/<xxx/yyy>`
 - sucht in Registry <myregistrydomain> auf Port <port> nach dem Image <xxx/yyy>
- **Docker Hub** ist die online verfügbare Docker Image Registry
- Alternativen: z.B. **quay.io**
- Eigene Docker Image Registries



CLI: Verwalten von Images – docker pull

- docker image pull [OPTIONS] NAME [:TAG | @DIGEST] oder
- docker pull [OPTIONS] NAME [:TAG | @DIGEST]
 - Lädt ein Image von einer Registry auf das lokale Filesystem herunter

```
$ docker pull debian

Using default tag: latest
latest: Pulling from library/debian
fdd5d7827f33: Pull complete
a3ed95caeb02: Pull complete
Digest: sha256:e7d38b3517548a1c71e41bffe9c8ae6d6d29546ce46bf62159837aad072c90aa
Status: Downloaded newer image for debian:latest
```

```
$ docker pull myregistry.local:5000/testing/test-image
```

CLI: Verwalten von Images – docker push

- docker image push [OPTIONS] NAME [:TAG] oder
- docker push [OPTIONS] NAME [:TAG]
 - Lädt ein Image in eine Registry hoch

```
$ docker image tag rhel-httpd:latest registry-host:5000/myadmin/rhel-httpd:latest  
$ docker image push registry-host:5000/myadmin/rhel-httpd:latest
```

CLI: Verwalten von Images – docker commit

- docker commit [OPTIONS] CONTAINER [REPOSITORY[:TAG]]
 - Erzeugt ein neues Image aus einem Container

```
$ docker ps

CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS
NAMES
c3f279d17e0a        ubuntu:12.04      /bin/bash          7 days ago        Up 25
hours              desperate_dubinsky
197387f1b436        ubuntu:12.04      /bin/bash          7 days ago        Up 25
hours              focused_hamilton

$ docker commit c3f279d17e0a  svendowideit/testimage:version3
f5283438590d

$ docker images

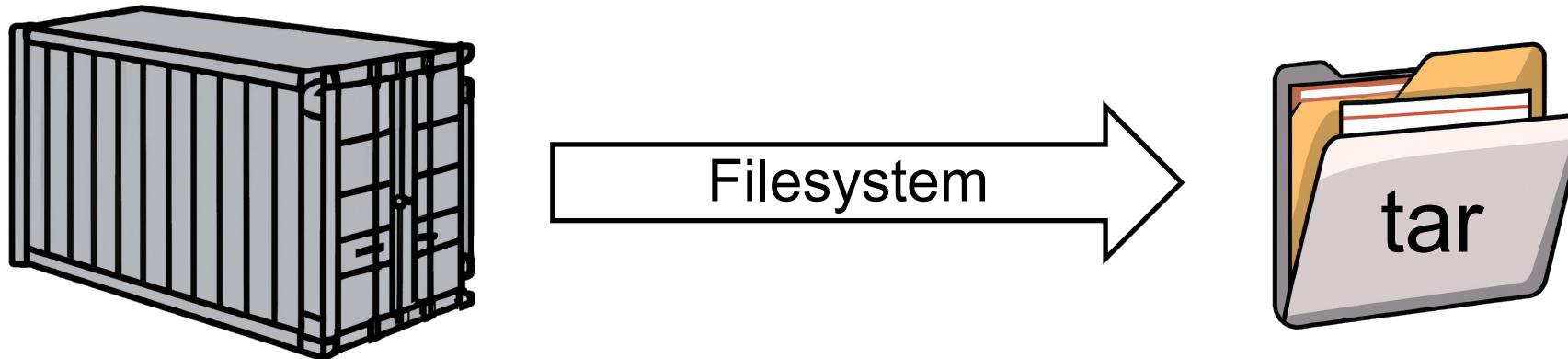
REPOSITORY          TAG                 ID                  CREATED            SIZE
svendowideit/testimage    version3        f5283438590d    16 seconds ago
335.7 MB
```

CLI: Verwalten von Images – docker export

- docker export [OPTIONS] CONTAINER
 - Exportiert das Dateisystem eines Containers in eine tar-Datei

```
$ docker export nginx > latest.tar
```

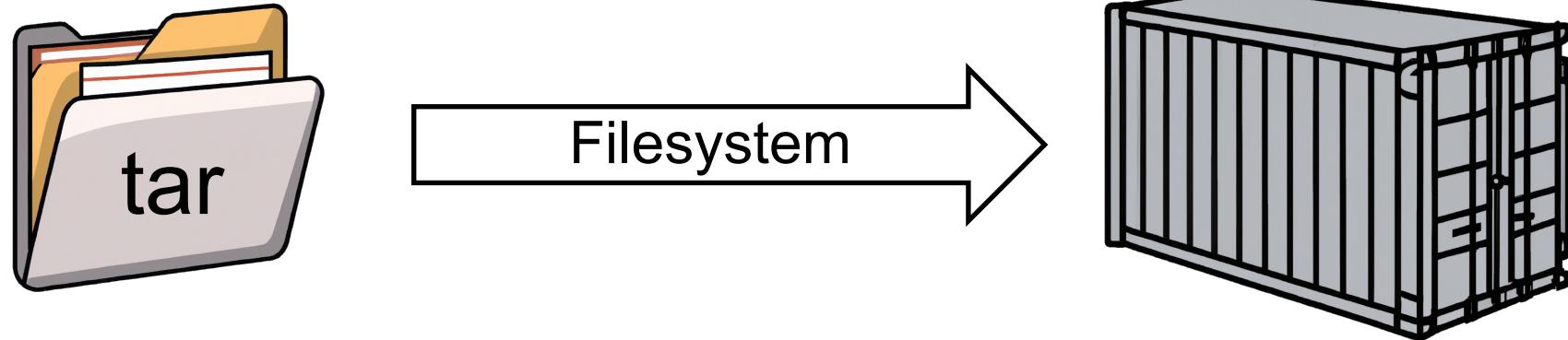
- **Wichtig:** Es wird ein Container gesichert, kein Image.
- Die History des Containers geht dabei verloren



CLI: Verwalten von Images – docker import

- docker import [OPTIONS] file|URL|- [REPOSITORY[:TAG]]
 - Importiert den Inhalt eines tar-Files in das Filesystem eines Images
 - Import durch Eingabe einer URL auch für „remote“-Datei möglich

```
$ docker import http://example.com/exampleimage.tgz
$ sudo tar -c . | docker import - exampleimagedir
$ sudo tar -c . | docker import --change "ENV DEBUG=true" - exampleimagedir
```

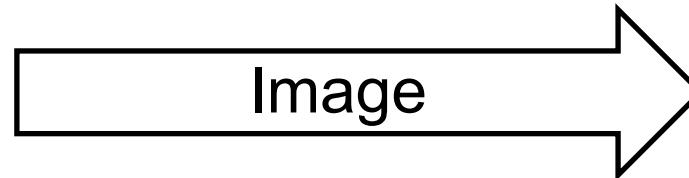
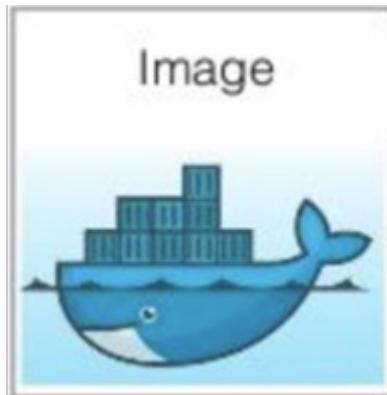


CLI: Verwalten von Images – docker save

- docker save [OPTIONS] IMAGE [IMAGE ...]

- Sichert ein oder mehrere Images in ein tar-File
- Ausgabe ohne Optionen nach stdout

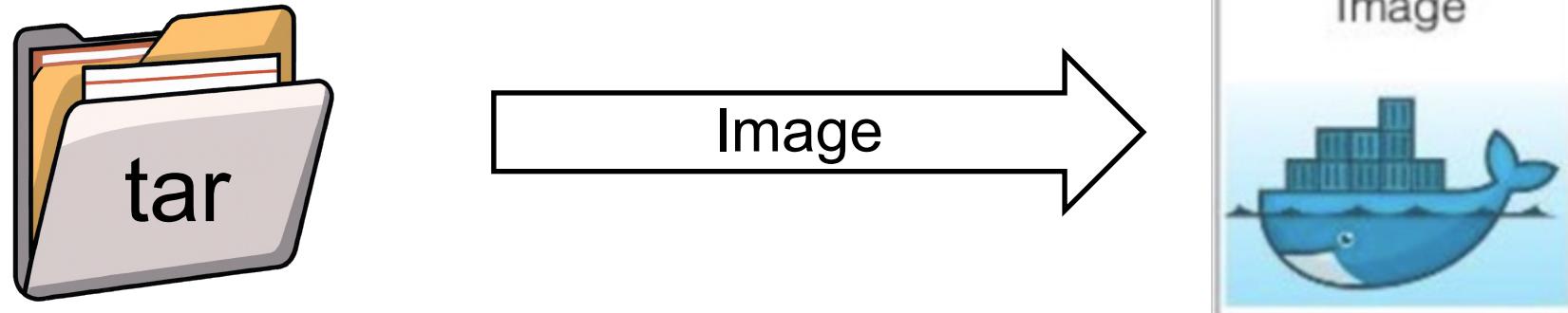
```
$ docker save busybox > busybox.tar
```



CLI: Verwalten von Images – docker load

- docker load [OPTIONS]
 - Lädt ein Image im tar-Format (auch komprimiert)
 - Standardmäßig von stdin

```
$ docker load busybox < busybox.tar
```



Kontrollfragen: Inbetriebnahme Images

- Was sind Image Layer?
- Was ist eine Image Registry? Welche Public Registries gibt es?
- Was macht „docker commit“?
- Können Images mit docker export und docker import gesichert werden?
- Was macht docker save/load im Vergleich zu docker push/pull?



Übung



Images

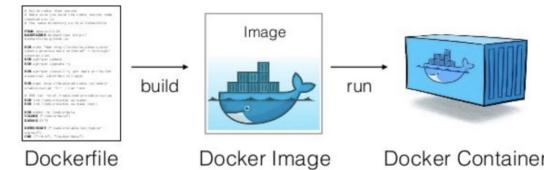
Docker Inbetriebnahme

Dockerfile

Dockerfile [1|2]

■ Ein Dockerfile

- beschreibt den Aufbau eines Docker Images
- wird durch das Kommando „docker build“ zu einem Image umgesetzt



■ Beispiel für ein Dockerfile

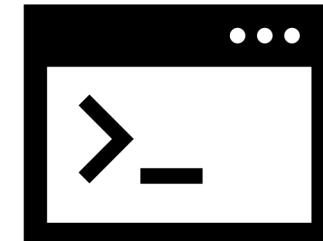
```
FROM registry.access.redhat.com/ubi8/ubi  
ENV foo=text  
RUN dnf install -y java-11-openjdk  
ADD myapp.jar /home/myapp.jar  
EXPOSE 8080  
CMD java -jar /home/myapp.jar
```

- 1 Ableiten von Basis Image
- 2 Umgebungsvariablen setzen
- 3 Installieren der Java Laufzeit
- 4 Eigene App als neuen Layer hinzufügen
- 5 Port der Anwendung bekanntgeben
- 6 Starten der Anwendung

Dockerfile [2|2]

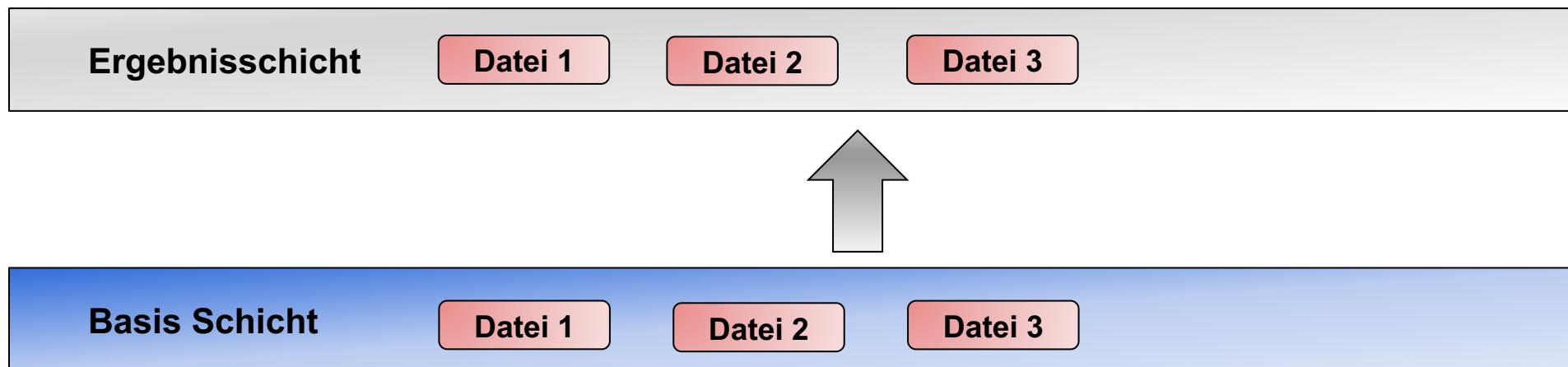
- Beschreibt den Bauprozess eines Images
 - Zeilen werden der Reihe nach ausgeführt
- Zeilen beginnend mit „#“ sind Kommentare oder Parser Direktiven
- Eine der ersten Anweisung bei abgeleiteten Images ist stets „FROM“
 - Beschreibt das Basis-Image
- Für jede Anweisungszeile im Dockerfile wird ein Image-Schicht (Layer) erzeugt
 - Die Größe der Schicht reflektiert die Größe der Änderung am Dateisystem

```
FROM ubuntu:latest
LABEL maintainer="myname@somecompany.com"
RUN apt-get update && apt-get upgrade -y
RUN apt-get install nginx -y
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```



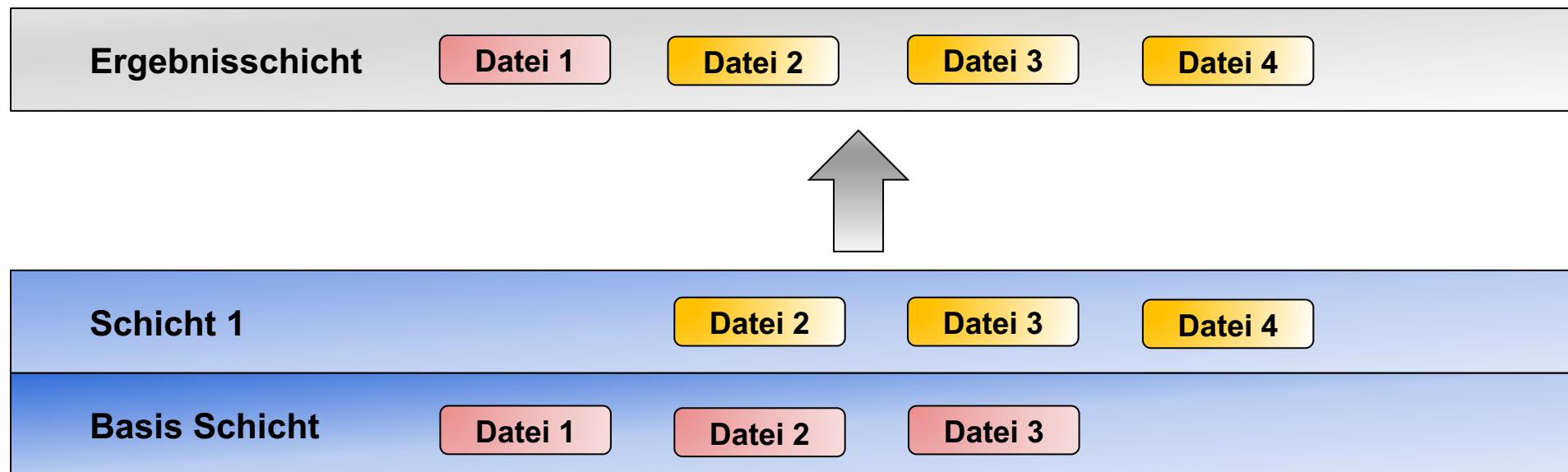
Schichtweise Aufbau des Dateisystems [1|3]

Der schichtweise Aufbau des Docker Images, zeigt sich auch auf Dateisystem:



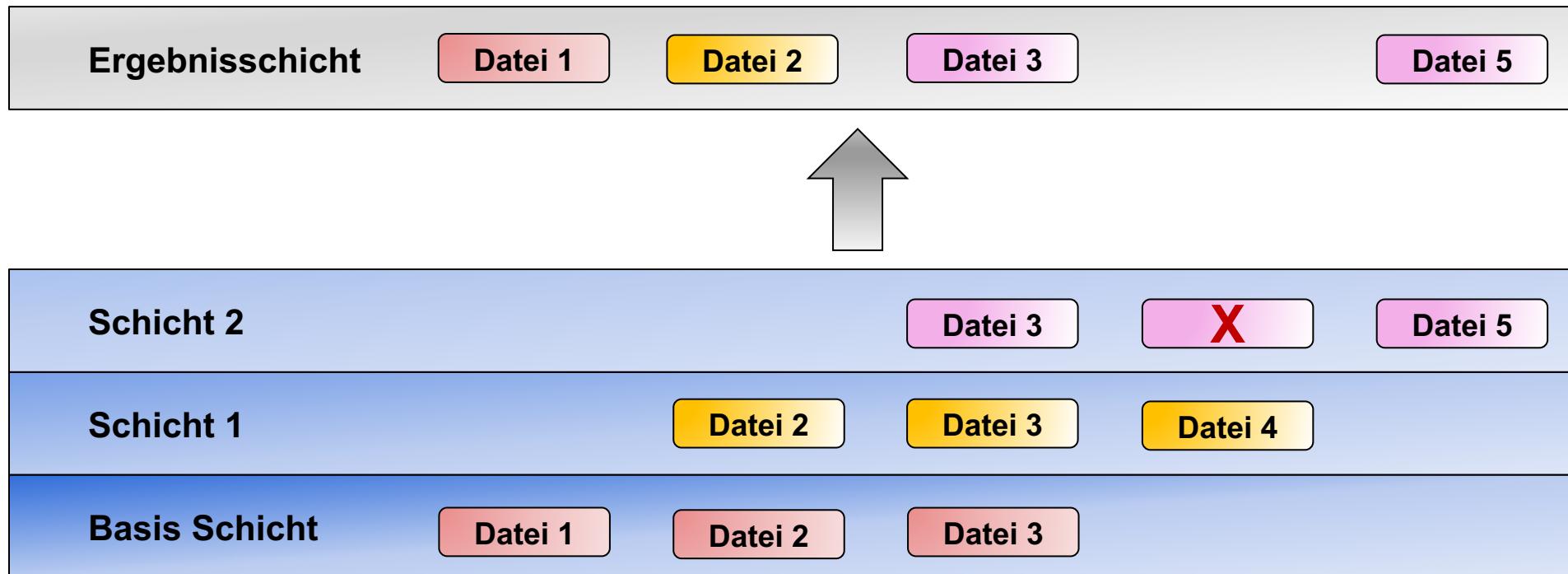
Schichtweise Aufbau des Dateisystems [2|3]

Der schichtweise Aufbau des Docker Images, zeigt sich auch auf Dateisystem:



Schichtweise Aufbau des Dateisystems [3|3]

Der schichtweise Aufbau des Docker Images, zeigt sich auch auf Dateisystem:



Neuer Container

Wenn ein neuer Container angelegt wird, wird eine schreibbare Schicht zum Image hinzugefügt.

```
$ docker create -n my_container my_image
```

Read
Write

Containerschicht

Read
Only

Ergebnisschicht

Datei 1

Datei 2

Datei 3

Datei 5

Neue Dateien

Es können zum Beispiel neue Dateien angelegt werden

```
$ docker exec my_container touch temp.txt
```

Read
Write

Containerschicht

temp.txt

Read
Only

Ergebnisschicht

Datei 1

Datei 2

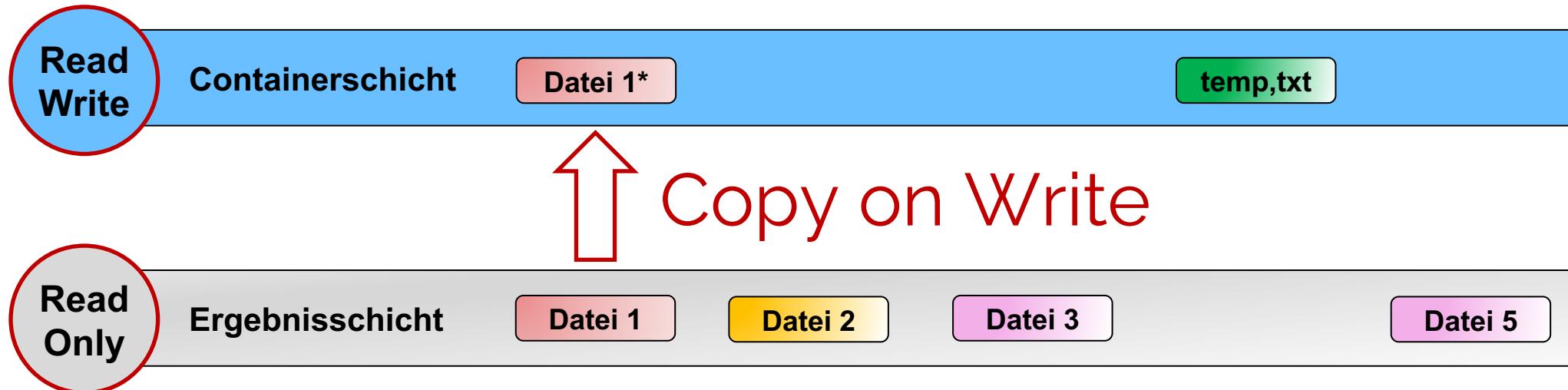
Datei 3

Datei 5

Copy on Write

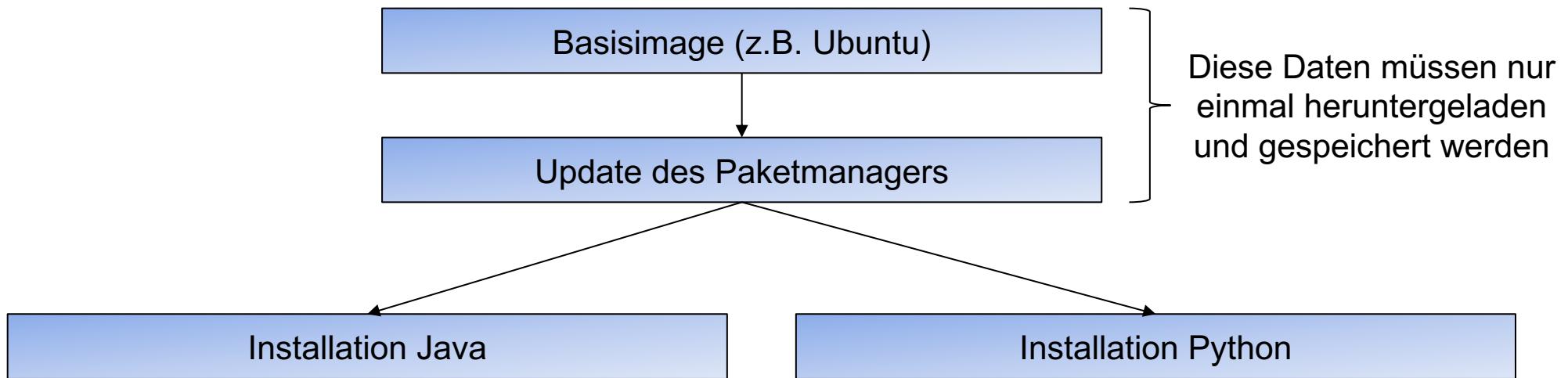
Daten aus dem Image werden nicht überschrieben, sondern werden vor dem Beschreiben in die Containerschicht kopiert

```
$ docker exec my_container echo "Neuer Eintrag" >> "Datei 1"
```



Warum dieser schichtweise Aufbau?

- Effizienz hinsichtlich Speicherung und Netzwerklast!
 - Beispiel: 2 Dockerfiles mit ähnlichen Anweisungen



Wichtige Befehle im Dockerfile - FROM

- FROM [--platform=<platform>] <image> [AS <name>]
 - Beschreibt die Ableitung vom Basis-Image
 - In der Regel die erste Anweisung im Dockerfile
- Nur das Kommando ARG darf noch davor stehen
 - Optionale Parameter, um die Laufzeitplattform zu spezifizieren

```
ARG CODE_VERSION=latest
FROM ubuntu:${CODE_VERSION}
...
```

Wichtige Befehle im Dockerfile - RUN

- RUN <command>

```
RUN ["executable", "param1", "param2"]
```

- Teil des Bauprozess für ein Image
- Die obere Ausführungsvariante verwendete eine Shell, so dass Variablenauflösung wie \$HOME oder Verkettungen mit „&&“ möglich sind.
- Führt den angegebenen Befehl in einer Shell (/bin/sh) aus und sichert den Zustand nach dessen Ausführung in einem neuen Layer des Images.

```
RUN /bin/bash -c 'source $HOME/.bashrc; echo $HOME'  
RUN ["/bin/bash", "-c", "echo hallo"]  
...
```

Wichtige Befehle im Dockerfile - COPY

- COPY [--chown=<user>:<group>] <src>... <dest>
COPY [--chown=<user>:<group>] ["<src>", ... "<dest>"]
 - Kopiert Dateien oder Verzeichnisse von <src> in das Verzeichnis <dest> des Containers
 - <src> ist ein relative Pfad zum Kontext des Images
 - <dest> kann absoluter Pfad sein oder relativer Pfad zum <WORKDIR>
 - Die Option “—chown” funktioniert nur unter Linux

```
COPY hom* /targetDir/  
COPY dummy.txt /absolutePath/  
COPY dummy2.txt relativePath/  
COPY --chown=55:mygroup files* /targetDir/
```

Wichtige Befehle im Dockerfile - EXPOSE

- EXPOSE <port> [<port>/<protocol>...]
 - Zeigt Docker an, dass ein Container auf einen bestimmten Port hören soll
 - Default für Protokoll ist "tcp"

```
EXPOSE 80/tcp  
EXPOSE 80/udp
```

- Das eigentliche Mapping von Ports des Containers zu den tatsächlich aufrufbaren Ports geschieht erst mit dem CLI Befehl "docker run"

```
docker run -p 80:80/tcp -p 80:80/udp ...
```

Wichtige Befehle im Dockerfile - CMD

- `CMD ["executable", "param1", "param2"]`

- `CMD ["param1", "param2"]`

- `CMD command param1 param2`

- Beschreibt den Hauptprozess, welcher standardmäßig beim Start des Images ausgeführt wird.
- Sollte maximal einmal in einem Dockerfile stehen. Wenn es mehrfach vorkommt, wird nur das letzte ausgeführt.
- Die oberste Aufrufvariante ist die empfohlene

```
CMD [ "echo", "Hallo" ]
```

- Kann jedoch beim Starten des Images überschrieben werden

```
docker run $image $other_command
```

Hauptprozess/Einstiegspunkt [1|2]

- Wird im Dockerfile mit dem Anweisung **CMD** definiert
 - `CMD ["executable", "param1", "param2"...]`
 - `CMD ["nginx", "-g", "daemon off;"]`
- Kann auch über die Option **--entrypoint** mitgegeben werden
 - `docker run --entrypoint [COMMAND] [ARGS...] IMAGE`
- Bei Auslassen der Option wird der Default-Befehl ausgeführt

Hauptprozess/Einstiegspunkt [2|2]

- Es gibt drei Möglichkeiten den Hauptprozess (PID1) anzupassen

- Im Dockerfile: ENTRYPOINT & CMD
- Über das CLI: Command
- Über das CLI: --entrypoint

- docker run IMAGE

- docker run my_image
- Hauptprozess: ENTRYPOINT + CMD → sleep 5

- docker run IMAGE [COMMAND] [ARGS...]

- docker run my_image 7
- Hauptprozess: ENTRYPOINT + [COMMAND] [ARGS] → z.B. sleep 7

- docker run IMAGE --entrypoint [COMMAND] [ARGS...]

- docker run my_image --entrypoint whoami
- Hauptprozess: [COMMAND] [ARGS...] → z.B. whoami



```
FROM ubuntu  
  
ENTRYPOINT ["sleep"]  
  
CMD ["5"]
```

CLI: Erstellen von Images – docker build (1/2)

- `docker build [OPTIONS] PATH | URL | -`
 - Erstellt ein Image auf Basis der Anweisungen in einem sog. Dockerfile
 - Das Dockerfile beschreibt den Bauprozess eines Images
 - Zeilen werden der Reihe nach ausgeführt
 - Der Pfad bzw. URL beschreibt den Kontext, d.h. alle anderen Dateien im Verzeichnis
- Mit Hilfe der „build“-Optionen können weitere Systemeigenschaften gesetzt werden
 - Netzwerkparameter, CPU- und Speichergrenzen
 - ulimit

CLI: Erstellen von Images – docker build (2/2)

- `docker build .`
 - sucht im lokalen Verzeichnis nach einer Datei „Dockerfile“ und baut daraus das Image.
 - Kontext ist das lokale Verzeichnis, d.h. auf alle Dateien in diesem Verzeichnis hat der Dämon beim Bauen des Images Zugriff
- `docker build -f <path/filename>`
 - verwendet die Datei `<path/filename>` anstatt „Dockerfile“
- `docker build - < <filename>`
 - liest die Datei `<filename>` vom STDIN, hat aber keinen Kontext
- `docker build -f ctx/Dockerfile http://server/ctx.tar.gz`
 - lädt das Archivfile von der URL
 - hat als Kontext das komplett entpackte Archivfile
 - sucht darin die Datei „ctx/Dockerfile“ zum Bauen des Images
- `curl example.com/remote/Dockerfile | docker build -f - .`

Kontrollfragen: Inbetriebnahme Dockerfile

- Nennen Sie die wichtigsten Befehle im Dockerfile und was diese machen.
- Was ist der Unterschied zwischen der Ergebnisschicht und dem Container Layer?
- Welche Dockerfile-Quellen können für den Bauprozess verwendet werden?
- Was ist der Kontext beim Bauprozess?



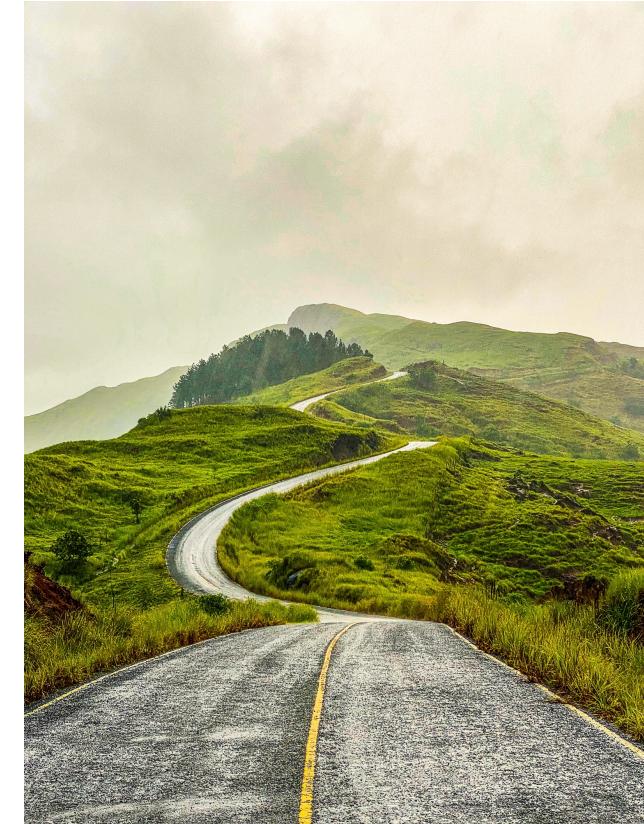
Übung



Dockerfiles

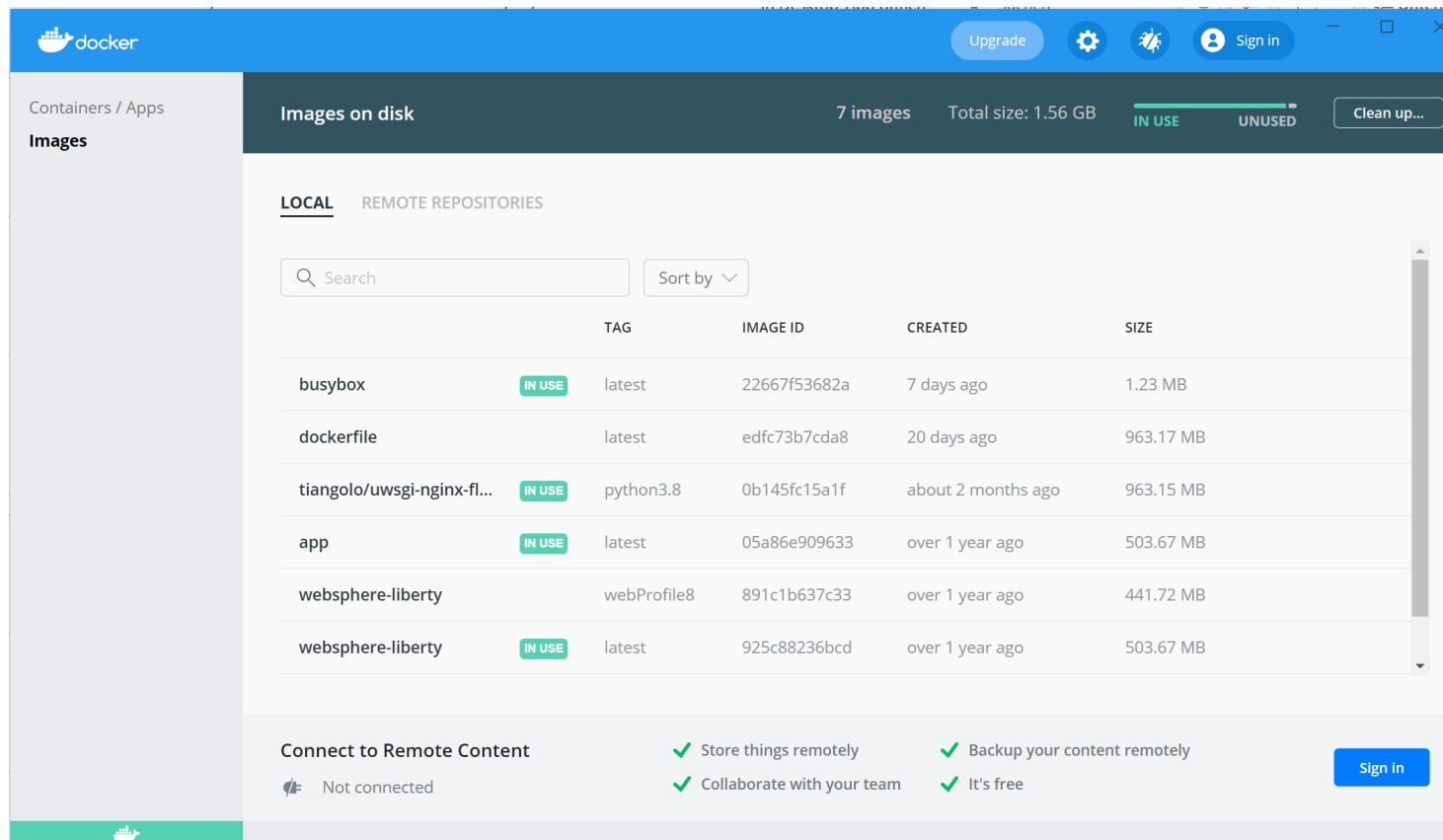
Was kommt als Nächstes?

- Kubernetes!
 - Orchestrierung auch mit Docker Swarm möglich
- Container Security
- Cloud-Native Application Design
 - 12-Factor App
 - Message Queuing
- CI/CD mit Docker
 - Continuous Integration/Continuous Delivery
- Bash Scripting
 - Hilft in jederlei Hinsicht
- Grafische Verwaltung von Containern mit GUI
 - z.B. mit Docker Desktop oder Portainer
 - Wahre Automatisierung ist jedoch nur mit CLI-Tools gut möglich



Alternative im Docker Desktop: Docker Dashboard

- Ein GUI auf Windows und Mac zum Verwalten von Images und Containern
- Im Vergleich zur CLI nur sehr eingeschränkter Funktionsumfang



Demo: Alternative Portainer

- Ein Container zum Verwalten von Images und Containern
- Als OCI-Container mit jeder Architektur verwendbar

