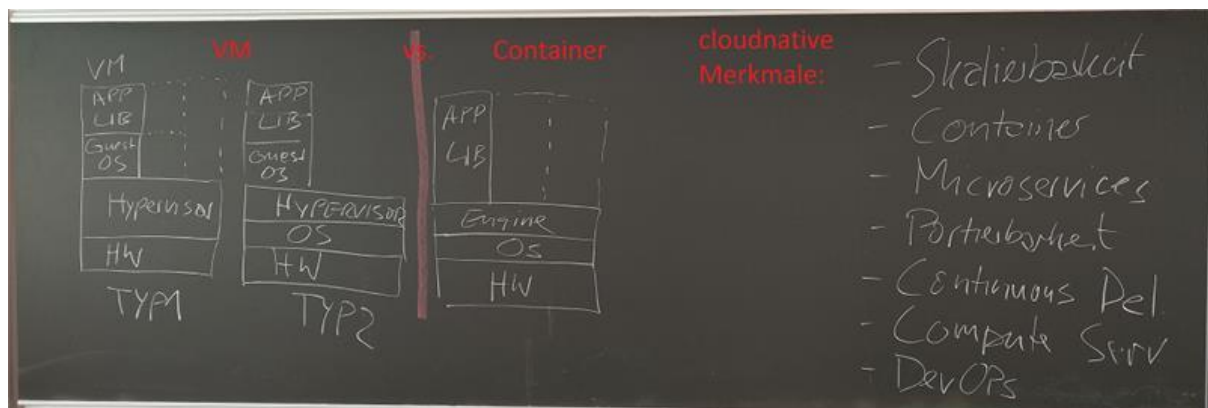
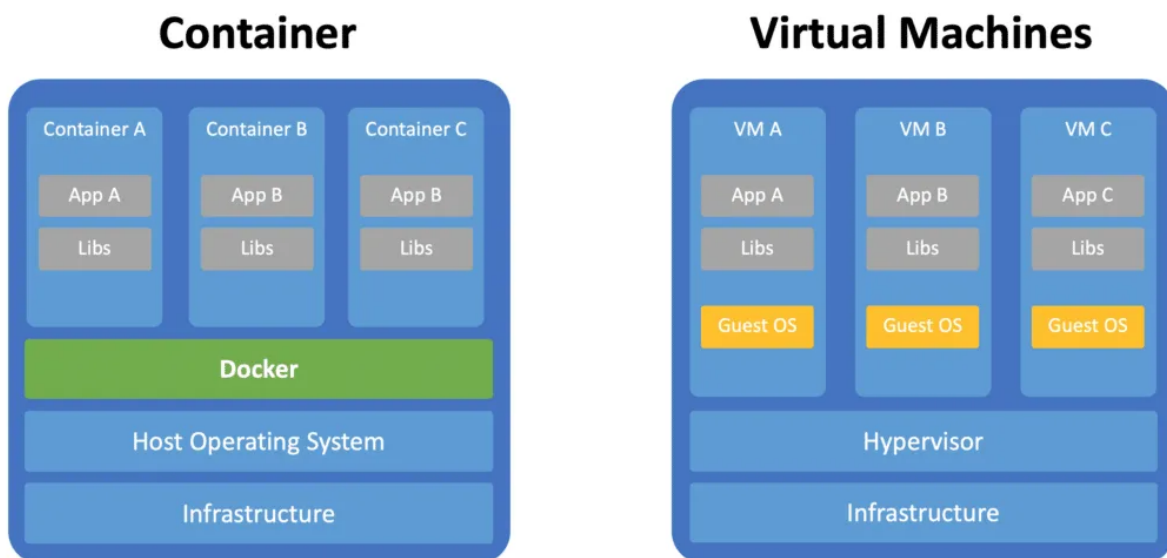


Docker Grundlagen

Basiswissen

1. Container
 - a. Laufende Image Instanz, die es ermöglicht, VM-ähnliche Eigenschaften ohne den Overhead eines kompletten Guest-OS zu haben.
 - b.
2. Container Image
 - a. Abbild eines Containers mit seiner Konfiguration
 - b. Bauplan
 - c. um anzuwenden: Container erstellen
3. Container Runtime
4. Host / VM / Hypervisor
5. Image Repository (Public / Private)
6. Cloud Native

Docker Theorie



Vorteile eines Docker Containers:

- weniger Speicher
- schneller als VMs
- für einen Container nicht komplettes VM Image laden
- lokal mit Docker oder in Cloud
- DockerHub als Ressource verfügbar

docker daemon

- Hintergrundprozess (wie alle daemons)
- verwaltet:
 - Docker Images
 - Container
 - Netzwerke
 - Speichervolumen
 - verarbeitet Docker-API-Requests

Wichtigste Docker CLI Befehle

Grundsätzlich immer "docker COMMAND "help" verwendbar.

"docker run" startet einen neuen Container, unterstützt sehr viele verschiedene Argumente und Konfigurationsmöglichkeiten, die bestimmen, wie ein Container läuft.

```
docker run imagename
```

docker run ...

| | |
|--|---|
| \$ docker run -it ubuntuimage /bin/bash | interactive shell |
| \$ docker run --rm ubuntuimage sleep 20 | startet container, löscht danach |
| \$ docker run -d | detach, Container läuft im Hintergrund |
| \$ docker run -d ubuntu ls -l | ROOT(/) als STDOUT |
| \$ docker ps | Übersicht über aktive container |
| \$ docker ps -a | alle anzeigen (aktive und nicht) |
| \$ docker ps -a -q | -q für quit, gibt von allen nur IDs aus |
| \$ docker images | liste aller images |
| \$ docker image ls | gleich wie oben |
| \$ docker rm containername | entfernt einen container mit name containername |
| \$ docker rm \$(docker ps --filter status=exited -q) | entfernt liste von containern die nicht mehr laufen |
| \$ docker rm -f \$(docker ps -a -q) | alle löschen |

| | |
|--|---|
| \$ docker rmi imagename | löscht image |
| \$ docker rmi docker images -q -f dangling=true | images ohne namen löschen |
| \$ docker start containerid | startet einen bereits beendeten Container mit id containerid |
| \$ docker stop | gracefully stopping, status exited |
| \$ docker kill | löst SIGKILL aus, definitiv nicht graceful, sofortiges stoppen |
| \$ docker logs | enthält STERR und STDOUT mit log messages |
| \$ docker inspect | u.a. konfigurationsoptionen usw, volume mappings, netzwerkeinstellungen zu bestimmten containern oder auch images. änderungen im dateisystem des containers und dem ursprungs image, us dem er gestartet wurde |
| docker top | prozessinformationen für bestimmten container |

Dockerfile vs docker-compose

Dockerfile

- Textdatei mit key value System
- Beinhaltet alle Anweisungen um ein Image zu erstellen
- "docker build"
- build context
 - Dockerfile + Build Context
 - lokale Dateien und Verzeichnisse, die mit ADD und COPY im Dockerfile angesprochen werden können
- Layer / Image Schicht
 - Jede Anweisung führt zu einer neuen Schicht auf dem Image
 - aufbauende Anweisungen

docker-compose:

- ermöglicht Verwendung von mehreren Containern
- einfache, zentrale Konfiguration
- YAML (key value, wichtig: indentation)
- Bestandteile:
 - Version
 - Services
 - build kontext

- ports (liste)
 - volumes (liste)
- Network
- ausführen mittels:
 - docker compose up

Microservices

Das Architekturmuster unterteilt Applikation in verschiedene, nach Funktionalität eingeteilte Subkomponenten, genannt Services.

Diese sind nicht abhängig voneinander und haben ihre eigenen Schnittstellen in Form von APIs.

Somit gibt es keinen SPOF und die einzelnen Services sind besser spezialisiert. Allerdings kann die Wartung eines solchen Systems viel aufwändiger sein.

- Eigenschaften Microservices
 - Eigenständig
 - spezialisiert

REST:

- Representational State Transfer
- HTTP
- Client / Server Architektur
- Zustandslos, keine Sessions benötigt
- ressourcenorientiert

Notizen zu Auftrag Flask_server

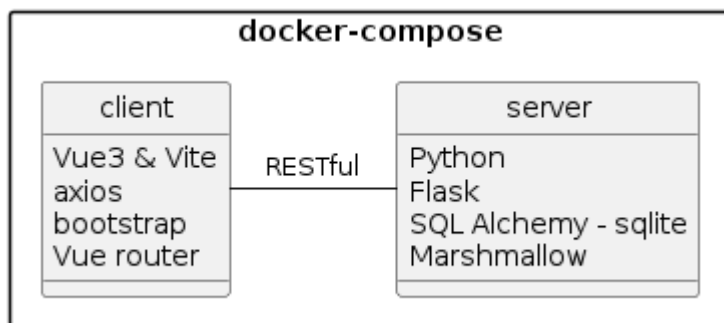
- docker build -t flask_server .
 - befinden uns im directory server
 - erstellen einen container mit namen flask_server
- docker run -d -p 5000:5000 -e FLASK_HOST='0.0.0.0' -e FLASK_PORT='5000' -e FLASK_APP='app.py' -e FLASK_DEBUG='1' flask_server
 - docker startet lässt im hintergrund den container flask_server
 - port, outside:inside 5000 auf host zu 5000 im container gemappt
 - argumente für flask mitgegeben
 - gleicher zweck wie environment variablen in docker compose
 - laufen lassen mit docker compose up

```

services:
  student-server:
    build: ./server
    ports:
      - 5000:5000
    volumes:
      - ./server:/app
    environment:
      # flask vars
      - FLASK_HOST=0.0.0.0
      - FLASK_PORT=5000
      - FLASK_APP=app.py
      - FLASK_DEBUG=1
      - CHOKIDAR_USEPOLLING=true
  student-clinet:
    build: ./client
    ports:
      - 3000:3000
    volumes:
      - ./client:/app
      - /app/node_modules/
    expose:
      - "3000"
    environment:
      - CHOKIDAR_USEPOLLING=true

```

-
- unterstes argument ermöglicht hot reload (nützlich bei entwicklung)\$



Volumes

- Form der Datenpersistierung
- Mount bei Linux
 - Datenverzeichnis unter anderem System oder Pfad einbinden
- Verschiedene Mount Arten:
 - bind mount
 - Festplatte des Hostrechners
 - nur sinnvoll wenn Folder von Code am gleichen Ort wie Container
 - volume
 - named oder unnamed
 - von docker verwaltet
 - tmpfs mounts
 - in Memory
 - nur kurzfristig sinnvoll, schnelle read / writes

- Erstellen von Volumes
 - --mount (mit allen Arten)
 - -v
 - --tmpfs
 - Beispiel mit run
 - `docker run -v /inhalt/auf/ihrem/rechner:/mount/pfad/in/container -d nginx`
 - Geht auch mit docker volume create
 - Geht auch in docker compose
 - Beispiel:
 - version: "3.9"
 - services:
 - web:
 - image: nginx:alpine
 - volumes:
 - - type: volume
 - source: mydata
 - target: /data
 - volume:
 - nocopy: true
 - - type: bind
 - source: ./static
 - target: /opt/app/static
 - erstes volume; klassische volume, zweites volume: bind mount volume
- Inkrafttreten eins Volumes
 - CLI
 - Geschieht mit docker run
 - `docker run -v /inhaltRechner:/mount/pfadcontainer -d`
 - YAML
 - volumes:
 - - type: volume
 - source: mydata
 - targe: /dockerpath
 - volume:
 - nocopy: true

Image Repository

Container Registry:

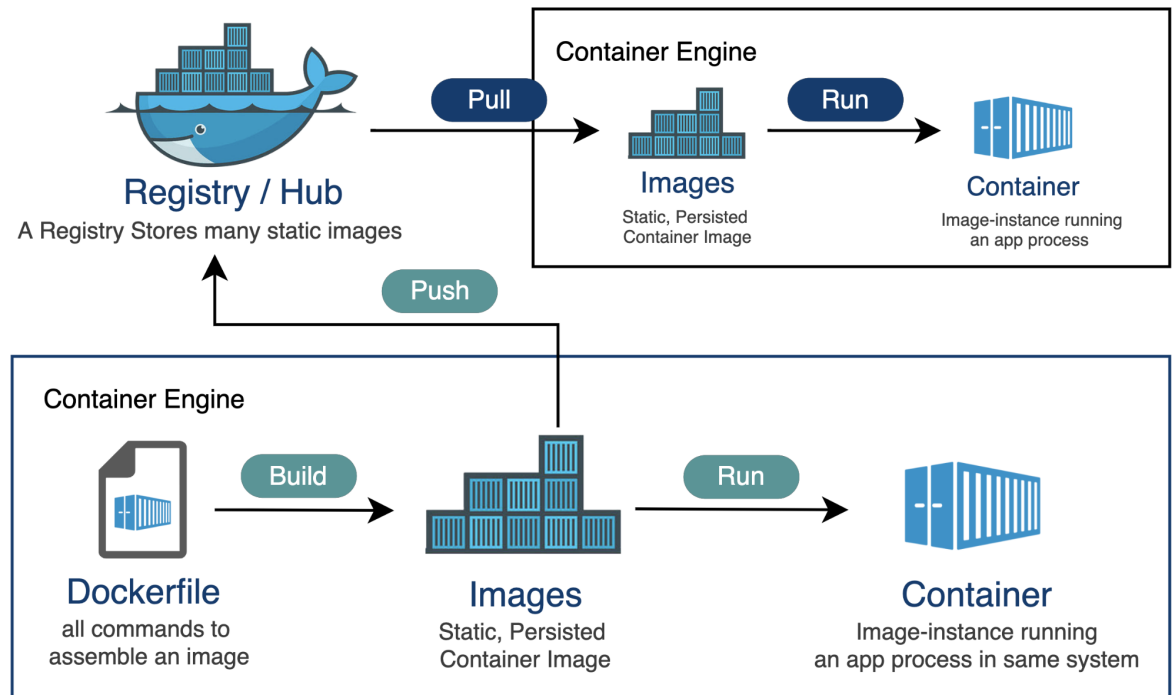
- zentraler Speicherplatz con Images
- versioniert
- enthalten alle notwendigen Abhängigkeiten

Accessibility von Registries

- öffentlich
 - jeder auf DockerHub kann pulls von dieser Registry machen
- privat
 - nur Leute mit explitem Zugang können auf Ressourcen zugreifen

Vorteile von Registries:

- Zentrales Speichern & Verwalten von Images
- Herstellbarkeit verschiedener Versionszustände
- DevOps bestandteil




- Bild: Ablauf Workflow mit Registries

Versionisierung von registries

- mittels tags
- wie id, macht uniqueely identifyable
- verschiedene Tags können einem image angewendet werden
- oft verwendet versionisierungsnummer und latest
- gewährleistung von kompatibilität verschiedener dienste



ubuntu

 DOCKER OFFICIAL IMAGE

• 1B+ • 10K+

Ubuntu is a Debian-based Linux operating system based on free software.

```
docker pull ubuntu
```

[Overview](#)

[Tags](#)

Quick reference

- Maintained by:
[Canonical](#) and [Tianon \(Debian Developer\)](#)
- Where to get help:

Recent Tags

[rolling](#) [lunar-20230301](#) [lunar](#) [latest](#)
[kinetic-20230217](#) [kinetic](#) [jammy-20230301](#) [jammy](#)
[focal-20230301](#) [focal](#)

Sonstige Notizen aus den folders als README

Tag 1

Docker CLI

1. Docker Version prüfen

```
docker -v
```

2. Nach offiziellem Docker-Image (nginx, apache, mysql, ubuntu) auf dockerhub suchen

```
docker search ubuntu
```

3. Ausgewähltes image herunterladen

```
docker pull ubuntu:latest
```

4. Neuen Container starten im Hintergrund

```
docker run -d ubuntu
```

5. Status des Containers überprüfen

```
docker ps -a
```

6. Interaktive shell in ubuntu starten

```
docker run -it /bin/bash
```

7. Einige Befehle austesten

```
touch textfile.txt
```

8. Interactive shell stoppen

```
exit
```

9. Container stoppen

```
docker stop [name]
```

10. Alle gestoppten Container auflisten

```
docker container ls --filter "status=exited"
```

11. Gestoppten Container entfernen

```
docker rm -f [name]
```

Tag 2

Dockerfile

```
# Pull the minimal Ubuntu image
FROM nginx:alpine
COPY . /usr/share/nginx/html

# Copy the Nginx config
COPY default /etc/nginx/sites-available/default

# Expose the port for access
EXPOSE 80/tcp

# Run the Nginx server
CMD ["/usr/sbin/nginx", "-g", "daemon off;"]
```

default config

```
server {
    listen 80 default_server;
    listen [::]:80 default_server;

    root /usr/share/nginx/html;
    index index.html index.htm;

    server_name _;
    location / {
        try_files $uri $uri/ =404;
    }
}
```

dockerfile python server

```
FROM ubuntu:latest
```

```
RUN apt-get update \  
    && apt-get install -y software-properties-common vim \  
    && apt-get update -y \  
    && apt-get install -y python3.6 \  
    && pip3 install --upgrade pip  
  
WORKDIR /app  
  
COPY . /app  
  
EXPOSE 5000:5000  
  
ENTRYPOINT [ "python3" ]  
  
CMD [ "python3", "-m" , "flask", "run", "--host=0.0.0.0"]
```

tag 3

Microservice Architectural Pattern

- Subkomponenten haben eigene unabhängige API
- Jeder Service kann unabhängig der anderen funktionieren
- Spezialisiert nach Funktionalität

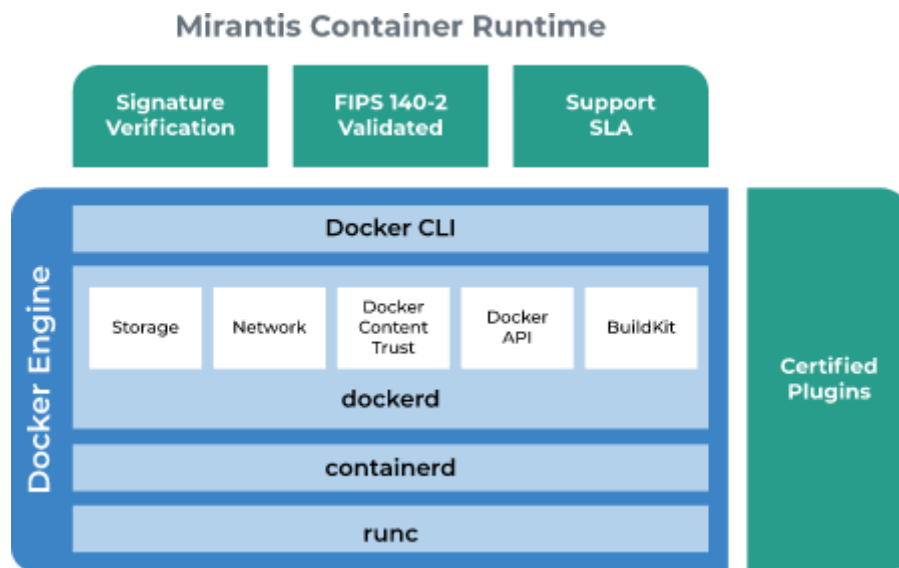
REST API

- Representational state transfer
- Client-Server über Http
- Keine Sessions, Request an sich kann isoliert verstanden werden

REST API mit microservices

- geschieht über JSON
- Docker Compose um nicht jeden Service einzeln starten zu müssen
- Hot Reload
 - bei jeder Änderung nicht container neu erstellen und starten

Container Runtime

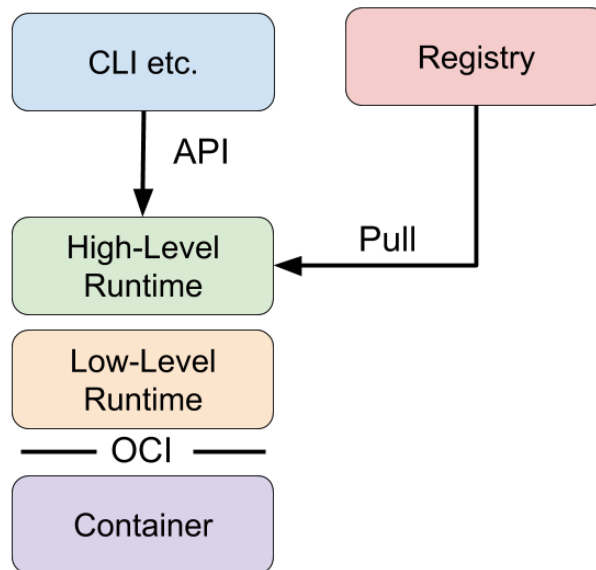


<https://www.mirantis.com/software/mirantis-container-runtime/>

Die Container-Runtime übernimmt die Aufgabe, sicherzustellen, dass die Containerumgebung optimal funktioniert. Sie kümmert sich um die richtige Konfiguration und Verwaltung aller erforderlichen Komponenten, die für das reibungslose Ausführen der Container notwendig sind. Dazu gehört die Bereitstellung des erforderlichen Betriebssystems, die Zuweisung von Ressourcen wie Speicher und CPU, das Netzwerkmanagement und die Isolierung der Container voneinander.

Die Container-Runtime stellt sicher, dass die Containerumgebung stabil, sicher und effizient läuft, indem sie die erforderlichen Einstellungen und Abhängigkeiten verwaltet und eine geeignete Laufzeitumgebung für die Container bereitstellt. Dadurch können Anwendungen in Containern isoliert und portabel ausgeführt werden, unabhängig von der zugrunde liegenden Infrastruktur.

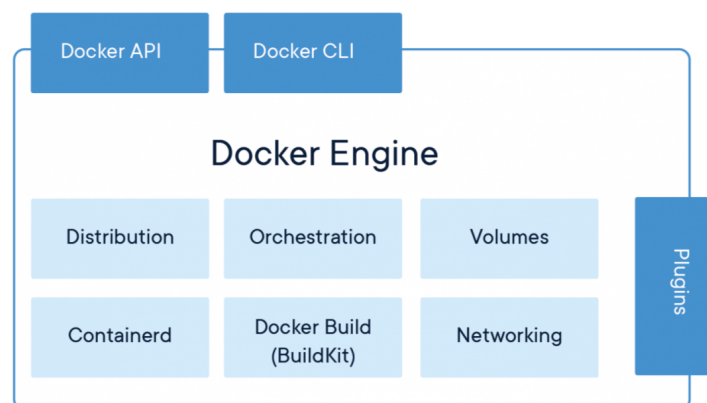
Ausserdem ist sie verantwortlich für das effiziente Laden von Container-Images aus einem Repository, die isolierte Zuteilung von Systemressourcen für den Container und die zuverlässige Verwaltung der Container-Lebensdauer. Durch diese Funktionen ermöglicht sie die sichere und zuverlässige Ausführung von Anwendungen in Containern, gewährleistet die Ressourcenisolierung und stellt sicher, dass die Containerumgebung optimal genutzt wird.



<https://www.ianlewis.org/en/container-runtimes-part-3-high-level-runtimes>

Es gibt drei verschiedene Arten von Container Runtimes: die Low-Level Container Runtime, die High-Level Container Runtime, die Sandboxed und Virtualized Container Runtimes. Die Low-Level Container Runtime fokussiert sich auf das Lifecycle Managements der Container. Die High-Level Container Runtime hat verschiedene Features wie ein CLI oder ein Container Image-Build-Service. Ein Beispiel für eine High-Level Container Runtime ist Docker. Zudem gibt es Sandboxed und Virtualized Container Runtimes, bei diesen hat der Containerized Process und der Host je einen anderen Kernel, wodurch sie sicherer, aber auch langsamer sind.

Container Engine

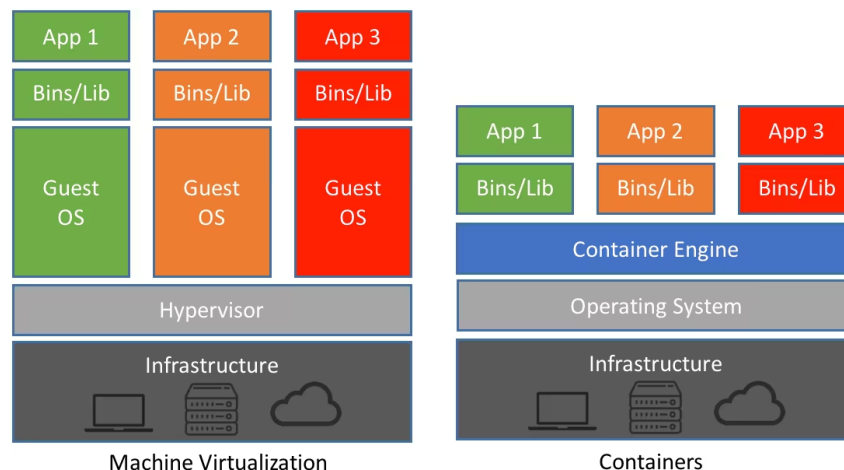


<https://www.docker.com/products/container-runtime/>

Container Engine ist nicht das gleiche wie Container Runtime, sind aber stark von der Container Runtime abhängig. Container Engine ist das, was der

Endbenutzer verwendet, um die Images und Containers zu verwalten (High-Level), wobei der Container Runtime die Hintergrundaufgaben, wie die Verwaltung der Komponente und Systemressourcen, übernimmt (Low-Level).

Unterschied Container vs. VM



<https://www.netapp.com/blog/containers-vs-vms/>

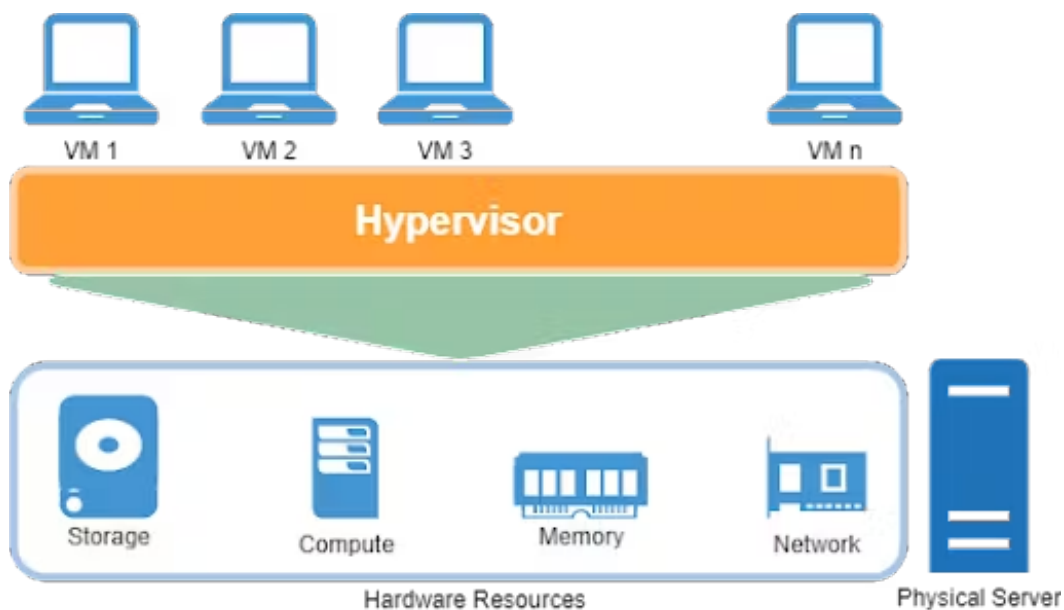
Virtuelle Maschinen führen stets ein vollständiges Betriebssystem samt Kernel aus. Das fordert mehr Systemressourcen, Arbeitsspeicher und Speicherplatz. Docker als ein Container-Typ führt hingegen nur die notwendigen Komponenten eines Betriebssystems aus.

Container lassen sich konfigurieren, sodass nur die Dienste im Container enthalten sind, die wirklich zum Ausführen, zum Beispiel einer App, notwendig sind. Auf einen Server passen mehr Container als virtuelle Maschinen.

Host / VM / Hypervisor

Ein Hypervisor ist eine Software, die virtuelle Maschinen erstellt und verwaltet. Er isoliert das Betriebssystem und die Ressourcen der virtuellen Maschinen und ermöglicht die Erstellung und Verwaltung dieser VMs. Die Ressourcen werden als gemeinsamer Pool betrachtet und können zwischen den VMs problemlos neu zugewiesen werden. Der Hypervisor stellt allen VMs die zugewiesenen Ressourcen zur Verfügung und verwaltet deren Zeitplanung in Abhängigkeit von den physischen Ressourcen. Mit einem Hypervisor können mehrere unterschiedliche Betriebssysteme parallel ausgeführt werden und die gleichen virtualisierten Hardwareressourcen nutzen. Es gibt verschiedene Hypervisoren wie VMware, KVM, Xen und Hyper-V.

[Informations-Quelle](#)



[Bild-Quelle](#)

Eine virtuelle Maschine (VM) ist ein virtuelles Betriebssystem, das auf einem physischen Hardwaresystem erstellt wird. Durch eine Software namens Hypervisor werden die Rechnerressourcen von der Hardware getrennt und der VM zur Verfügung gestellt. VMs haben normalerweise ihr eigenes Betriebssystem und können mehrere ressourcenintensive Funktionen ausführen. Sie können ganze Server, Betriebssysteme, Desktops, Datenbanken und Netzwerke abstrahieren, aufteilen, duplizieren und emulieren. Der Hypervisor behandelt die Computing-Ressourcen als einen Pool von Ressourcen, die problemlos auf Guests oder neue VMs verteilt werden können. Mit VMs können Sie mehrere verschiedene Betriebssysteme gleichzeitig auf einem einzelnen Computer ausführen. Die in der VM emulierte Endbenutzererfahrung ist nahezu identisch mit einer Echtzeit-Betriebssystemerfahrung, die auf einer physischen Maschine ausgeführt wird.

Ein Host ist ein physisches Hardwaresystem welches Hardwareressourcen zur Verfügung stellt. Auf den Hosts können dann mehrere Guests laufen, welche alle auf die Ressourcen des Hosts zurückgreifen.