

# Modul CNT

## Container Laufzeitumgebungen

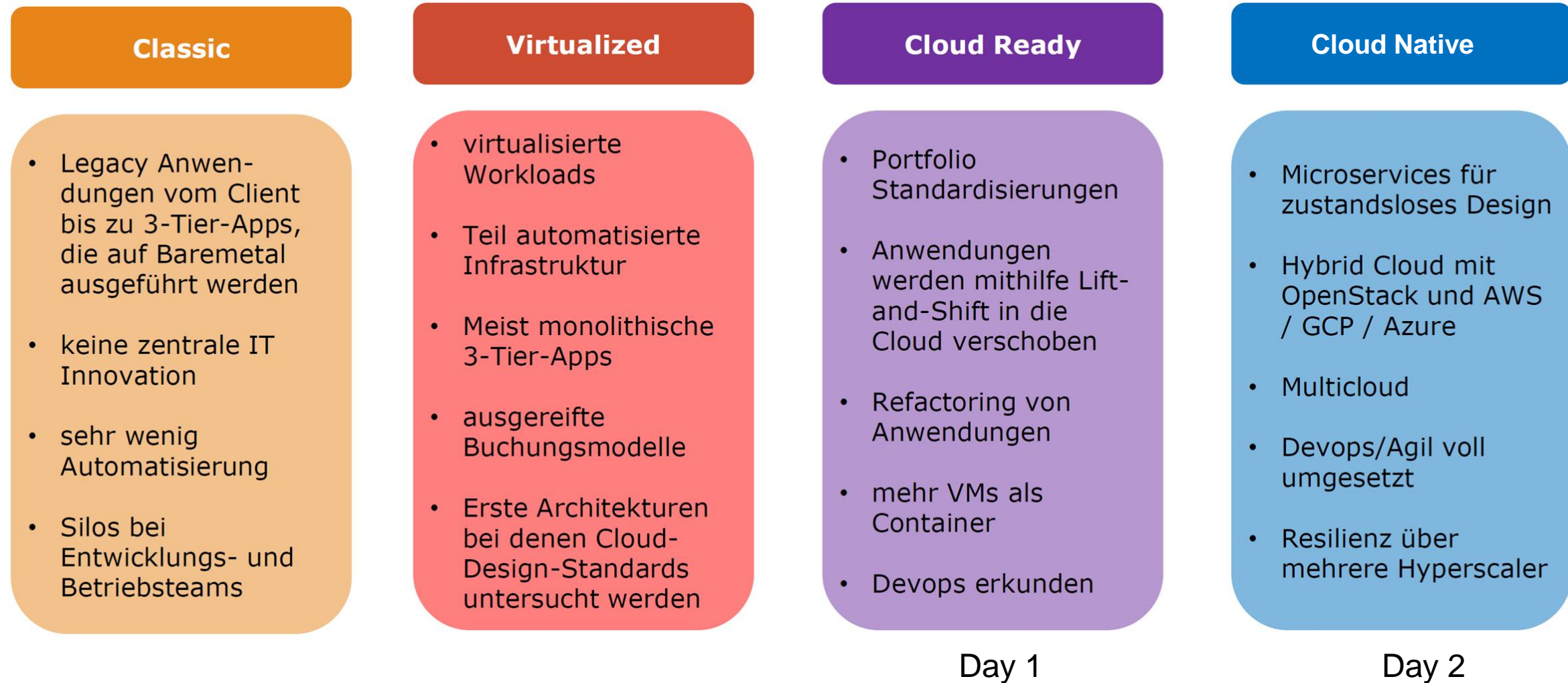
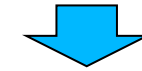
September 2021

Marcel Bernet

Dieses Werk ist lizenziert unter einer  
[Creative Commons Namensnennung - Nicht-kommerziell -  
Weitergabe unter gleichen Bedingungen 3.0 Schweiz Lizenz](https://creativecommons.org/licenses/by-nc-sa/3.0/de/) /



# Maturity Model (Reifegradmodell)



# Lernziele

- ★ Sie haben einen Überblick über Container Laufzeitumgebungen wie Docker, CRI-O, containerd.

# Zeitlicher Ablauf

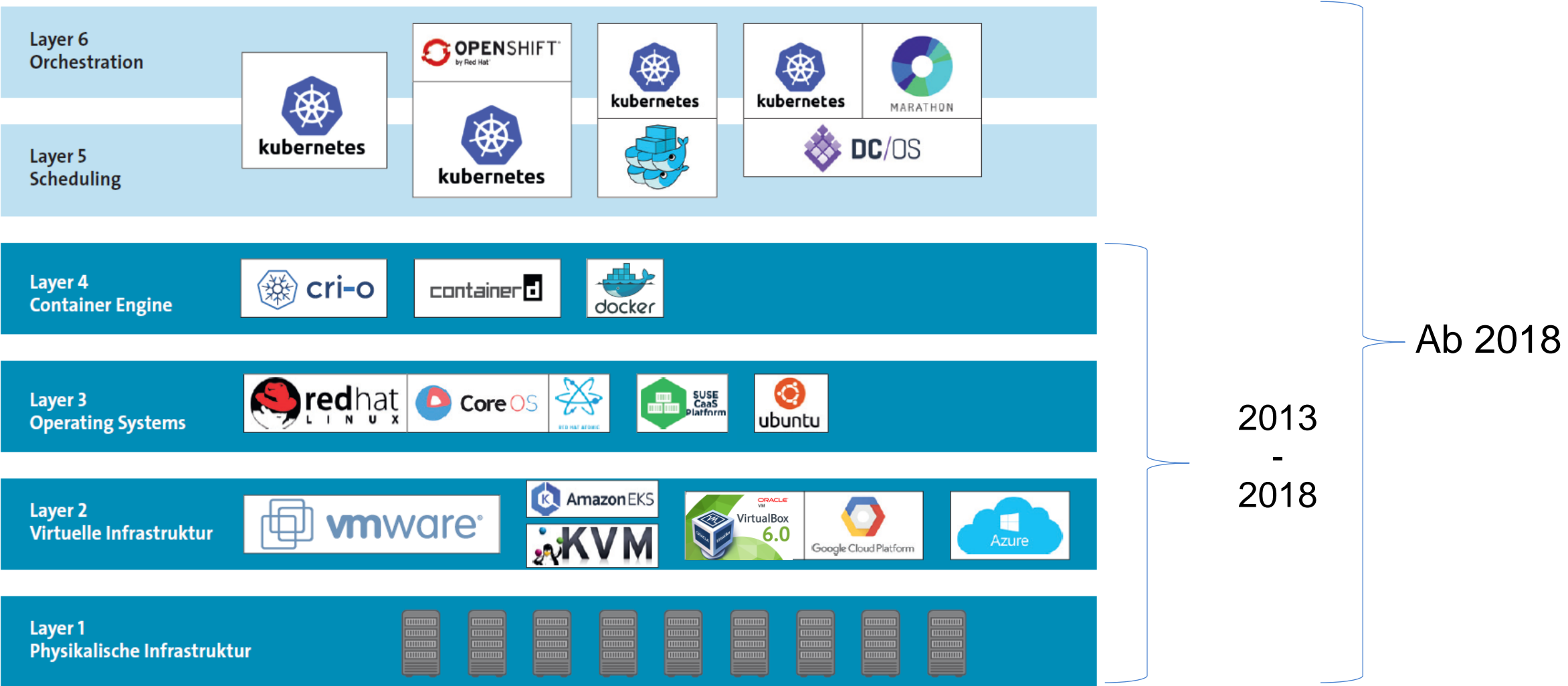
- ★ Container Laufzeitumgebungen (Runtimes)
- ★ Docker Versionen, wichtige Meilensteine und Inkompatibilitäten
- ★ Funktionaler Überblick
- ★ Einfaches Image-Management
- ★ Container Images bauen (u.a. docker build) und verwalten
- ★ Hands-on: Container Images und Registries
- ★ Docker Networking
- ★ Docker Compose
- ★ Docker Volumes
- ★ Reflexion
- ★ Lernzielkontrolle



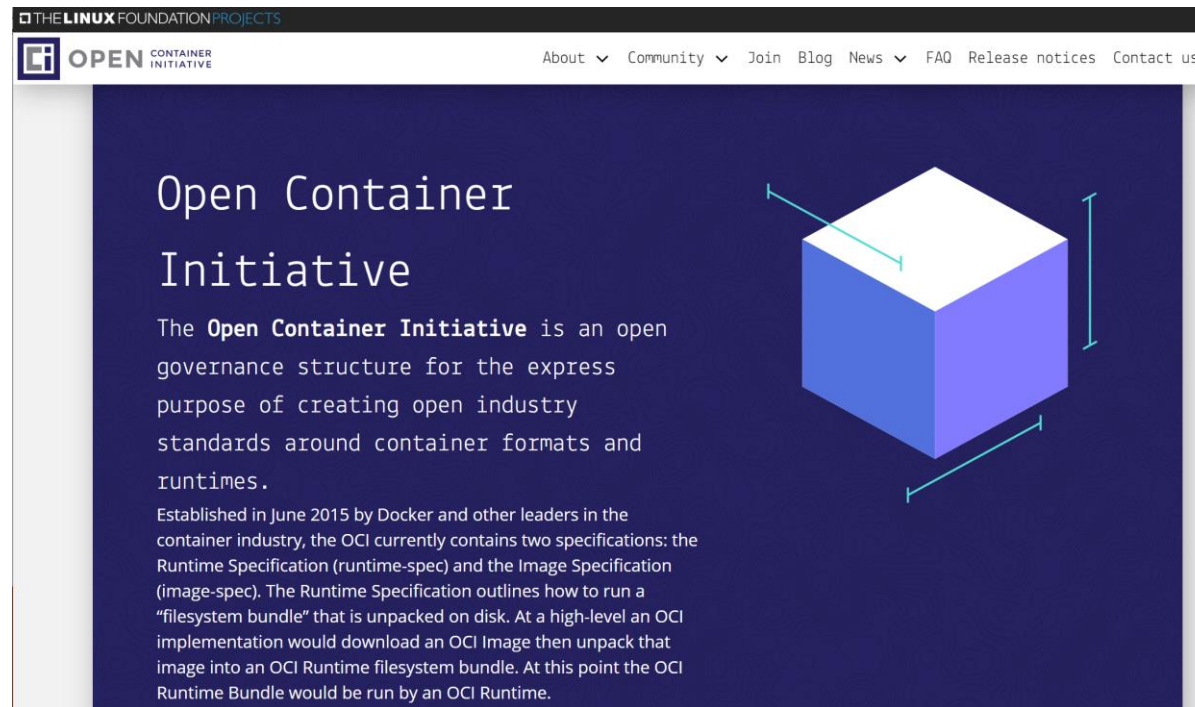
# Container - Geschichte

- ★ Container sind ein altes Konzept. Schon seit Jahrzehnten gibt es in UNIX-Systemen den Befehl chroot, der eine einfache Form der Dateisystem-Isolation bietet.
- ★ Seit 1998 gibt es in FreeBSD das Jail-Tool, welches das chroot-Sandboxing auf Prozesse erweitert.
- ★ **Solaris Zones** boten **2001** eine recht **vollständige Technologie** zum Containerisieren, aber diese war auf Solaris OS beschränkt.
- ★ Ebenfalls 2001 veröffentlichte Parallels Inc. (damals noch SWsoft) die kommerzielle Containertechnologie Virtuozzo für Linux, deren Kern später (im Jahr 2005) als Open Source unter dem Namen OpenVZ bereitgestellt wurde.
- ★ Dann startete **Google** die Entwicklung von CGroups für den Linux-Kernel und begann damit, seine **Infrastruktur in Container zu verlagern**.
- ★ Das Linux Containers Project (LXC) wurde 2008 initiiert, und in ihm wurden (unter anderem) CGroups, Kernel-Namensräume und die chroot-Technologie zusammengeführt, um eine vollständige Containerisierungslösung zu bieten.
- ★ **2013** lieferte **Docker** schließlich die fehlenden Teile für das Containerisierungspuzzle, und die Technologie begann, den Mainstream zu erreichen.
- ★ **Heute:** Kubernetes hat die Lücke zu Orchestrierung (vereinfacht: Management) von Container Umgebungen geschlossen. Reine Container Umgebungen, wie Docker, verlieren an Bedeutung.

# Container Runtimes und Kubernetes



# Open Container Initiative



- ★ Die [Open Container Initiative](#) (OCI) ist ein unter dem Dach der Linux Foundation gestartetes Projekt, um industrieweit gemeinsame Standards für die Container-Virtualisierung zu schaffen.
- ★ Diese Initiative nutzt eine offene Governance-Struktur und wurde als Open Container Project auf der DockerCon im Juni 2015 vorgestellt. Später wurde sie zur Open Container Initiative umbenannt.
- ★ Spezifikationen (weitere in Entwicklung):
  - "[runtime-spec](#)" - Laufzeitverhalten (Runtime)
  - "[image-spec](#)" - Imageformat



# Container Runtimes

- ★ Open Container Initiative (und damit Kubernetes) compatible Container Runtimes sind, u.a.:
  - **Docker (2013), im 2017 Marktanteil von 90 %.**  
-> Kubernetes is deprecating Docker as a container runtime after v1.20 (1.1.2021)!
  - **containerd** ist als Daemon für Linux und Windows verfügbar. Es verwaltet den gesamten Container-Lebenszyklus seines Host-Systems, von der Image-Übertragung und -Speicherung über die Container-Ausführung und -Überwachung bis hin zum Low-Level-Storage und Netzwerk.  
-> **Es ist aus Docker entstanden, ist vereinfacht dessen Container Runtime.**  
-> **Adaptors: IBM Cloud, Google, CloudFoundry, Docker, etc.. -> CNCF graduated project**
  - **CRI-O:** das 2016 gestartete und 2017 vorgestellte CRI-O baut auf dem Kubernetes Container Runtime Interface (CRI) auf. CRI-O erlaubt das direkte Verwenden von OCI-kompatiblen (Open Container Initiative) Containern in Kubernetes, ohne zusätzlichen Code oder weiteres Tooling.  
-> **Contributors sind RedHat, Intel, SUSE, IBM. -> CRI-O is a CNCF incubating project.**  
-> **1.19 K8s Version: Empfehlung CRI-O Container statt Docker EE auf Windows verwenden!**



# Docker: Versionen, wichtige Meilensteine und Inkompatibilitäten

- ★ 1.9: mit komplett überarbeitetem Netzwerk-Modul
- ★ 1.10: neues DB-Format
- ★ 1.11: mit Support für runC und **containerD**
- ★ 1.12: mit integriertem Swarm Mode
- ★ 1.13: unter anderem mit neuem Plugin-Format (z.B. für Erweiterungen wie Volume Manager)

➔ **For full Kubernetes Integration**

- Kubernetes on Docker for Mac is available in 17.12 Edge (mac45) or 17.12 Stable (mac46) and higher.
- Kubernetes on Docker for Windows is available in 18.02 Edge (win50) and higher edge channels only.

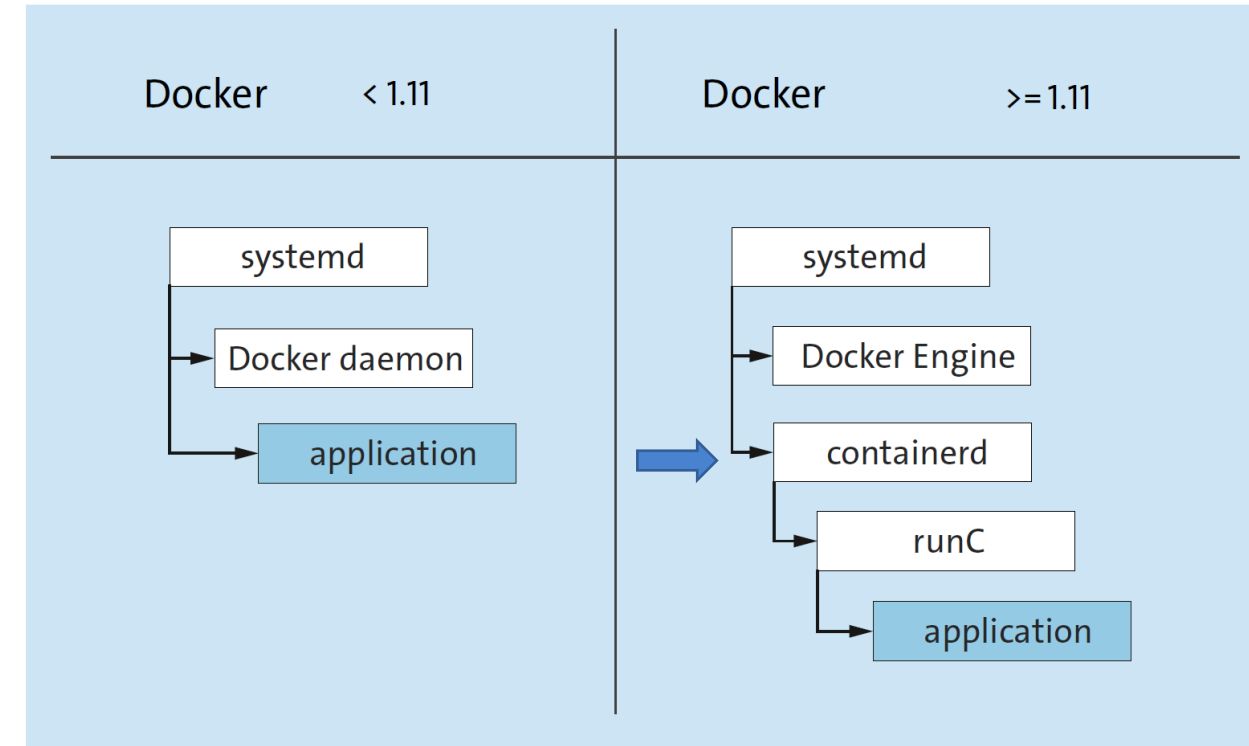


Abbildung 3.4 Funktionaler Docker Stack Pre-/Post 1.11

- ★ **Docker CE (Community Edition):** Support für 4 Monate
- ★ **Docker EE (Enterprise Edition):** Support für ein Jahr ([Mirantis](https://www.mirantis.com/docker-enterprise-edition/))

# Docker: CLI

- ★ **Befehlsaufbau:**
  - `docker <sub-command>`
- ★ **attach:** an eine laufende Container Sitzung ankoppeln
- ★ **build:** neues Image erzeugen
- ★ **tag:** Image weitere Bezeichnung zuweisen, z.B. um Registry zu wechseln.
- ★ **container:** (ab Version 1.13) neues Subkommando für die Container-spezifischen Verwaltungsbefehle: `[container] attach cp diff export kill ls port rename rm start stop unpause wait commit create exec inspect logs pause prune restart run stats top update`
- ★ **cp:** Dateien zwischen Host und Container kopieren
- ★ **diff:** zeige Änderungen im Filesystem des laufenden Containers
- ★ **exec:** Kommandos innerhalb des Containers ausführen
- ★ **ps:** Container-Instanzen auflisten
- ★ **container prune:** (ab 1.13) Container-Instanzen löschen
- ★ **run:** startet den Container und erzeugt eine randomisiert oder explizit benannte neue Instanz des Images
- ★ **start:** startet eine vorhandene, gestoppte Container-Instanz
- ★ **restart:** restartet einen laufenden Container. Per `-t` kann ein maximaler Timeout für Stopp angegeben werden, bevor der Container per kill beendet und neu gestartet wird.
- ★ **stop:** stoppt eine laufende Container-Instanz
- ★ **kill:** Killen eines Containers (gegebenenfalls mit explizitem Signal `[-s]`, Default ist **SIGKILL**)
- ★ **rm:** löscht eine gestoppte Container-Instanz und alle damit verbundenen Dateien des Read/Write Layers im unterliegenden Dateisystem
- ★ Dokumentation:  
<https://docs.docker.com/engine/reference/run/>

# Container: Funktionaler Überblick

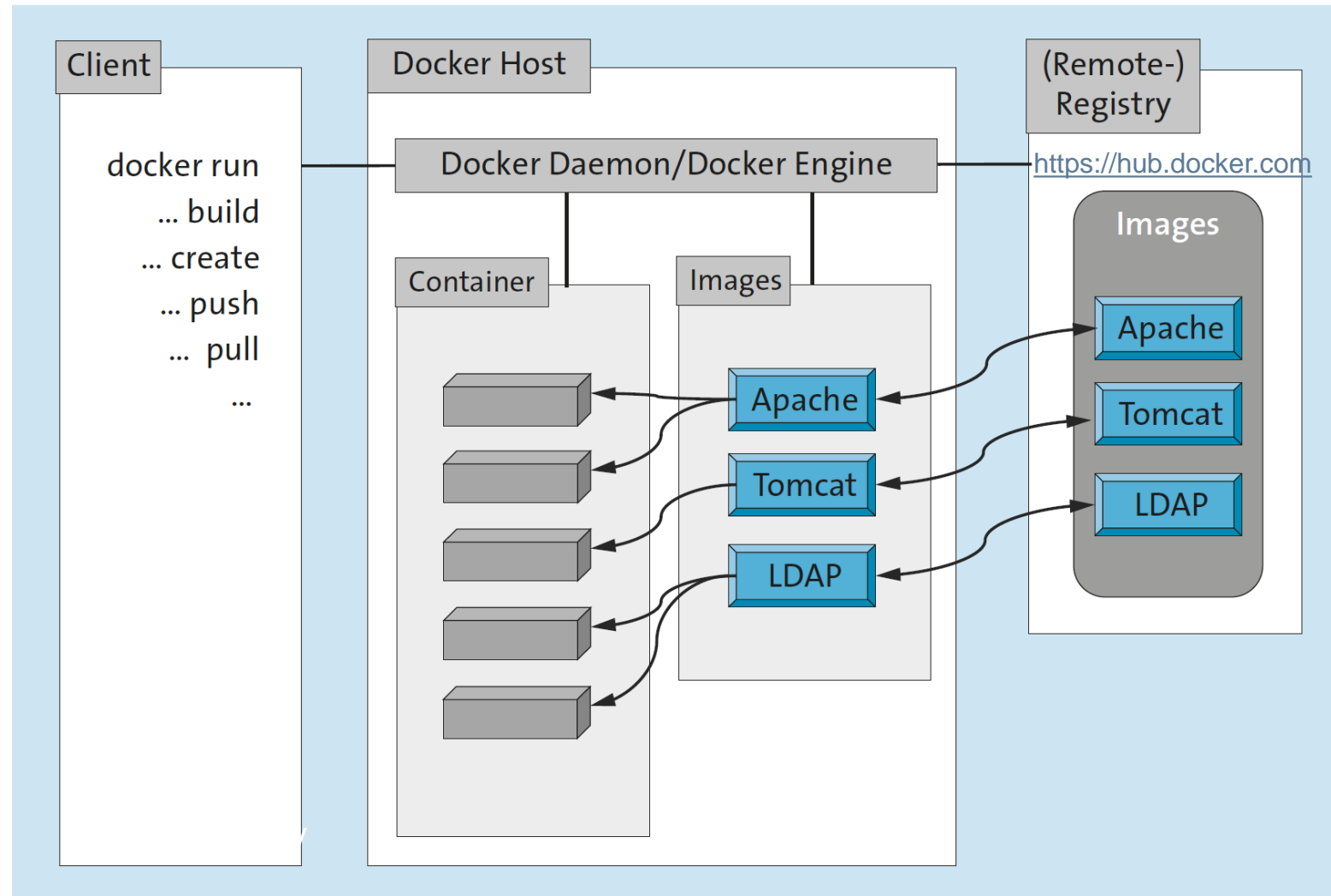


Abbildung 4.2 Docker Client, Docker Host und (Remote-)Registry

# Einfaches Image-Management

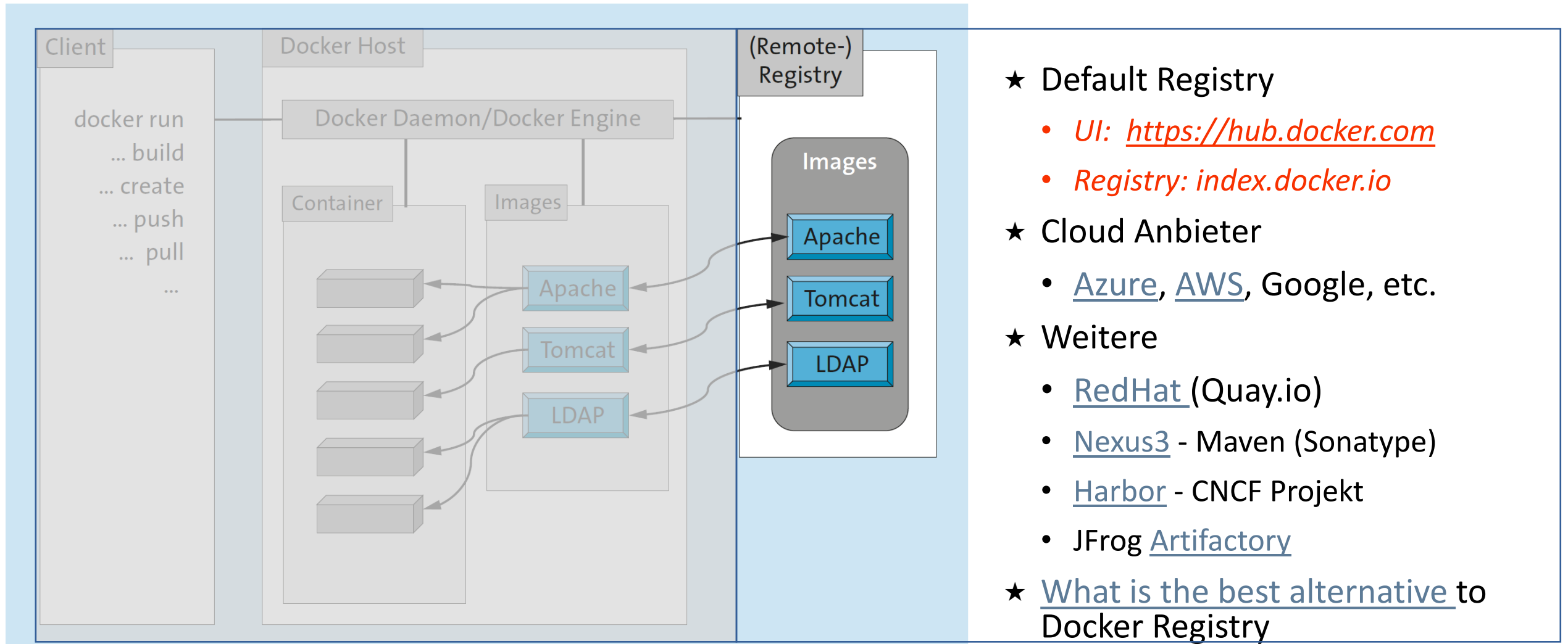
## ★ Wie kommen wir zu Images?

- Von vertrauenswürdigen Registry herunterladen (docker pull/run).
- Selber Erstellen

### Hinweis

Für Produktivumgebungen sind öffentlich zugängliche Registries keine wirkliche Alternative. Es sollten ausschließlich selbst erzeugte und verwaltete Images, welche in einer vertrauenswürdigen, firmenspezifischen Registry abgelegt sind, Verwendung finden.

# Registries



- ★ Default Registry
  - UI: <https://hub.docker.com>
  - Registry: [index.docker.io](https://index.docker.io)
- ★ Cloud Anbieter
  - [Azure](#), [AWS](#), Google, etc.
- ★ Weitere
  - [RedHat](#) (Quay.io)
  - [Nexus3](#) - Maven (Sonatype)
  - [Harbor](#) - CNCF Projekt
  - JFrog [Artifactory](#)
- ★ [What is the best alternative to Docker Registry](#)

Abbildung 4.2 Docker Client, Docker Host und (Remote-)Registry

Quelle: free eBook Containerized Application Lifecycle with Microsoft Platform and Tools<sup>13</sup>

# Registries, Repositories, Tags

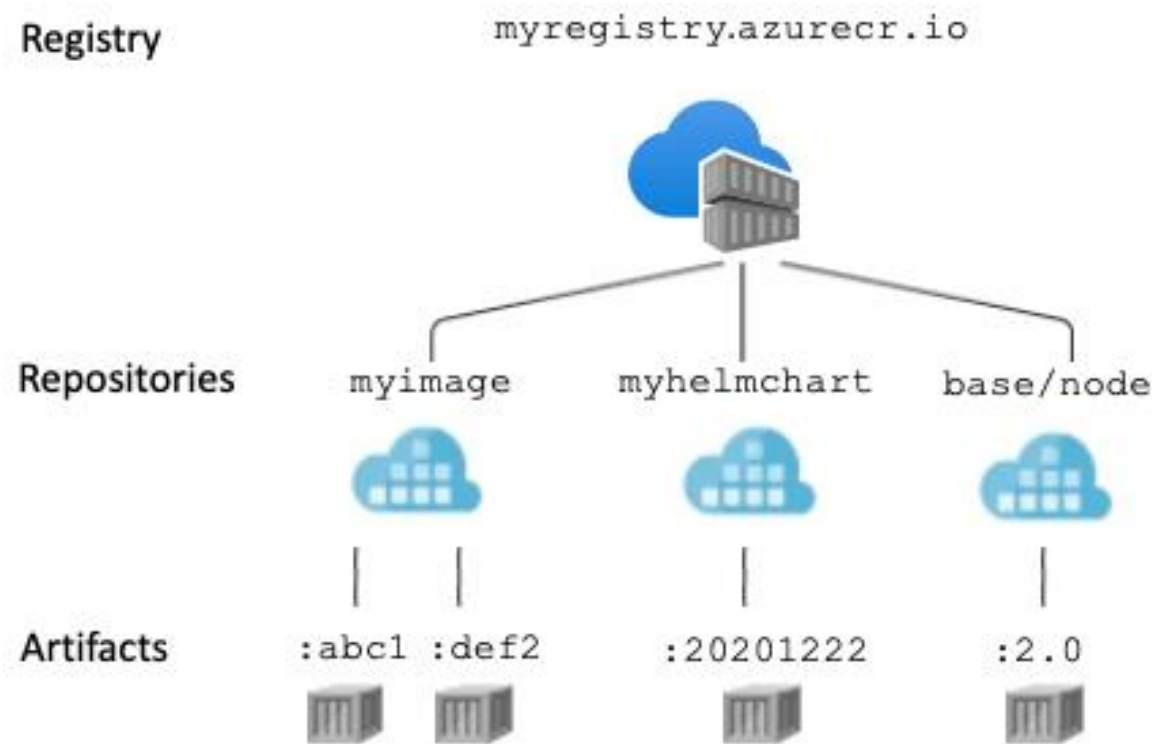
★ **Container Image Namensraum:** `[REGISTRY_HOST[:REGISTRY_PORT]/]REPOSITORY[:TAG]`

- **REGISTRY\_HOST/PORT:** optionale Angabe einer (**FQDN**) Registry.
- **REPOSITORY:** Eigentlicher Speicherplatz von Images. Ein Repository kann mehrere Unterschiedliche Versionen eines Images beinhalten.  
Bei <https://hub.docker.com> muss ein Repositoryname zwingend aus User/Image bestehen. Der Username entspricht dem login auf <https://hub.docker.com/>. Offizielle Docker Images haben keinen Usernamen.
- **TAG:** Optionaler Tag, entspricht einer Versionsnummer, teilweise mit Hinweisen auf Originalimage z.B. :3-alpine. Wird kein Tag angegeben, wird automatisch latest angefügt.

Tag Beispiele anhand von [maven](#):

- `3.6.3-jdk-11` , `3.6-jdk-11` , `3-jdk-11` , `3.6.3-openjdk-11` , `3.6-openjdk-11` , `3-openjdk-11`
- `3.6.3-ibmjava-8-alpine` , `3.6.3-ibmjava-alpine` , `3.6-ibmjava-8-alpine` , `3.6-ibmjava-alpine` , `3-ibmjava-8-alpine` , `ibmjava-alpine`

# Registry vs. Repository vs. Artifact



- ★ Eine Registry hat einen eindeutigen FQDN (Fully Qualified Domain Name) Namen und beinhaltet 1:n Repositories
- ★ Ein Repository hat einen eindeutigen Namen, je nach Registry mit vorangestelltem Username und beinhaltet mehrere Artifacts (Versionen) eines Container Images
- ★ Ein Artifact ist ein mit einem Tag versehenes Container Image

<https://docs.microsoft.com/de-de/azure/container-registry/container-registry-concepts>



# Aufbau eines Images

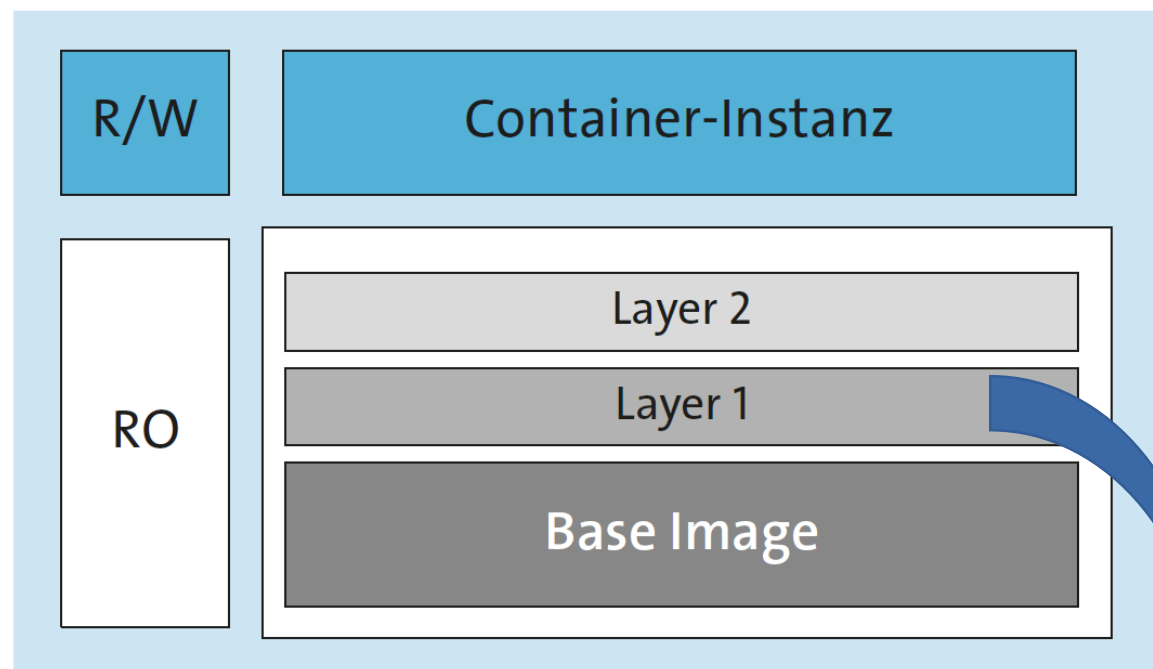
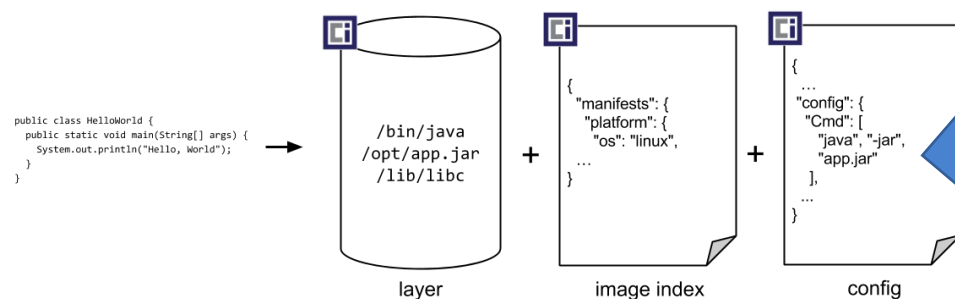


Abbildung 3.9 Schematischer Aufbau eines Images

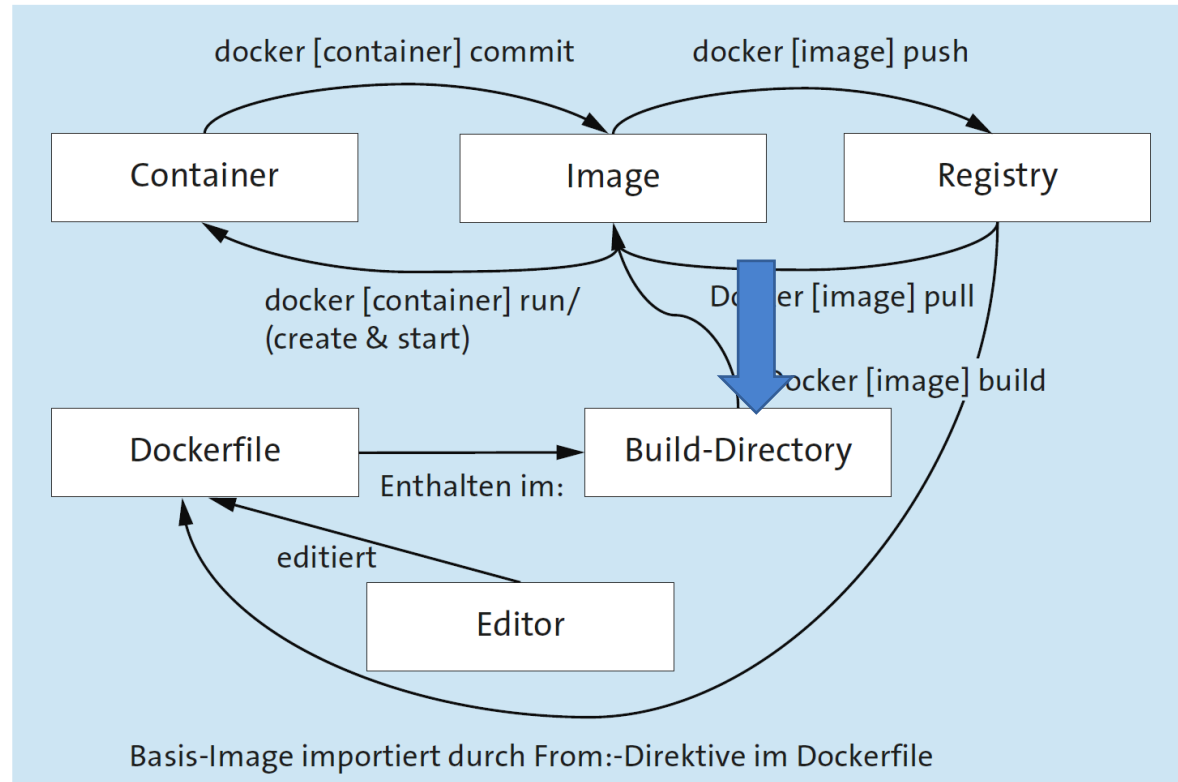


- ★ Container Instanz: Read / Write
- ★ Container Image: Read Only
  - Besteht aus Layers (Snapshots), max. 127 (Begrenzung [Overlay2](#))
  - Jede **Zeile** in Konfigurationsdatei (Dockerfile) erzeugt einen **neuen Layer**.
  - Als **Base Image** wird der **unterste Layer** bezeichnet. Dieser kann von einem anderen Image abgeleitet oder leer (scratch) sein.
  - **V2**: Multi-Architektur-Images -Plattformspezifische Versionen
  - **V2**: Images mit eindeutigem Hash
- ★ Image Standardisierung
  - <https://www.opencontainers.org/>
  - <https://github.com/opencontainers>
  - <https://github.com/opencontainers/image-spec/blob/master/spec.md>

# Container Images bauen und verwalten (1)

- ★ Der effizienteste Weg, um einen bestimmten vordefinierten, funktional validierten und vor allem jederzeit reproduzierbaren (Base-)Image-Stand zu erschaffen, läuft über ein Buildfile, das sogenannte **Dockerfile**.
- ★ Das Dockerfile dient dazu: einen vordefinierten **Build-/Installations-Prozess** zu definieren/starten welcher vollautomatisiert Kommandos durchführt und **an dessen Ende ein fertiges Image** zur Verfügung steht.
- ★ Beim Build-Vorgang selbst wird das per Definition zugrunde liegende Readonly-Basis-Image geklont und in den Speicher geladen, dann wird (pro Build-Aktion) ein neuer Read/Write Layer darauf aufgesetzt. In diesen Read/Write Layern werden die im Dockerfile gelisteten Aktionen abgearbeitet und bei Erfolg (als intermediate Build) committed.
- ★ Schaut man während eines Builds auf die Docker-Prozesse (docker ps), sieht man, dass der Intermediate Build als Container-Prozess auf dem Host aktiv ist.
- ★ **Alternativen:** Google Cloud [buildpacks](#), Getting started with [Buildah](#), [Docker without Docker](#)

# Container Images bauen und verwalten (2)



## Hinweis

Nicht der `docker [image] build`-Befehl selbst erstellt das Image – die via Build-Befehl und Dockerfile übergebenen Instruktionen werden an den Docker Daemon weitergeleitet, und dieser führt den eigentlichen Build-Prozess aus.

## ★ Beispiel Dockerfile für (GO) Applikation:

- `FROM scratch` # leeres Dateisystem
- `COPY myapp /`
- `CMD ["/myapp"]`

## ★ Beispiel Dockerfile für System Service:

- `FROM ubuntu:20.04`
- `RUN apt-get update`
- `RUN apt-get -q -y install apache2`
- `CMD "/usr/sbin/apache2 -DFOREGROUND"`

# Dockerfile-Direktiven/-Instruktionen

- ★ **FROM** – von welchem Basisimage abgeleitet
- ★ ~~**MAINTAINER** – Autor (Achtung: In neueren Docker-Versionen (deprecated).~~
- ★ **RUN** – im Read/Write Layer des Intermediate Container auszuführende Aktionen
- ★ **ENV** – Umgebungsvariablen
- ★ ~~**EXPOSE** – Portmappings (K8s: Services)~~
- ★ **ENTRYPOINT** – »fixes« Startkommando für primäre Applikation
- ★ **CMD** – überschreibbares Startkommando für primäre Applikation
- ★ ~~**HEALTHCHECK** – Überwachung des Prozesses im Container auf Funktionstüchtigkeit. (K8s Health Probe Patterns)~~
- ★ ~~**LABEL** – eindeutige Bezeichnungen für spätere Selektion (K8s Labels).~~
- ★ **ADD** – Files, Directories, remote URLs in Image kopieren
- ★ **COPY** – Files in Image kopieren
- ★ ~~**VOLUME** – (persistente) Volumes für Image definieren (K8s PersistentVolumesClaim)~~
- ★ **USER** – User für Kommandoausführung festlegen
- ★ **WORKDIR** – Setze das Working Directory für CMD, RUN, COPY, ADD, ENTRYPOINT
- ★ **STOPSIGNAL**: legt das Systemaufrufsignal fest, das zum Beenden an den Container gesendet wird. Z. B. 9, oder ein Signalname im Format SIGNAME, z. B. SIGKILL.
- ★ Etc.

# Beispiele (vereinfacht): Dockerfile (1)

## ★ Hello-world Linux

- FROM scratch # leeres Dateisystem
- COPY hello /
- CMD ["/hello"]

## ★ Hello-world Windows

- FROM mcr.microsoft.com/**windows**/nanoserver
- COPY hello.txt C:
- CMD ["cmd", "/C", "type C:\\hello.txt"]

## ★ Apache Web Server Ubuntu

- FROM **ubuntu:<version>**
- RUN apt-get update
- RUN **apt-get** -q -y install apache2
- CMD "/usr/sbin/apache2 -DFOREGROUND"

## ★ Apache Web Server Fedora

- FROM **fedora:<version>**
- RUN **yum** install httpd
- CMD "/usr/sbin/apache2 -DFOREGROUND"

# Beispiele: Dockerfile (2)

- ★ hello-world - minimales Dockerfile (von vorheriger Folie)
  - <https://github.com/docker-library/hello-world/blob/master/Dockerfile-linux.template>
- ★ Alpine Linux - ADD mit gleichzeitigem Entpacken
  - [https://github.com/alpinelinux/docker-alpine/blob/da78fcf5c5da55092aa82a97095274b3df648866/x86\\_64/Dockerfile](https://github.com/alpinelinux/docker-alpine/blob/da78fcf5c5da55092aa82a97095274b3df648866/x86_64/Dockerfile)
- ★ nginx - Problematik Umleitung stdout, stderr (letzte paar Zeilen im Dockerfile)
  - <https://github.com/nginxinc/docker-nginx/blob/master/stable/alpine/Dockerfile>
- ★ OpenJDK - mit Anleitung wie das Base Image zu verwenden ist:
  - [https://hub.docker.com/\\_/openjdk](https://hub.docker.com/_/openjdk)
- ★ .NET Core Docker Samples
  - <https://github.com/dotnet/dotnet-docker/blob/master/samples/README.md>

# Best Build Practices

- ★ Container sind **kurzlebig** und jederzeit **reproduzierbar**
- ★ **Wer** hat's gemacht? (MAINTAINER bzw. LABEL-Direktive)
- ★ Wie ist es **bezeichnet**? (--tag)
- ★ Verwenden eines eigenen **Build-Ordnerns** pro Template
- ★ Verwendung eines **.dockerignore-Files**
- ★ **Schlanke Images** (Ein Image sollte klein, simpel und so schlank/kompakt wie möglich sein.)
- ★ Nur **ein Prozess** pro Container
- ★ Anzahl der **Layer minimieren**/niedrig halten
- ★ Siehe auch: [Best practices for writing Dockerfiles](#) und [Not every container has an operating system inside](#)



# Dockerfile Empfehlungen (des Kursleiters)

- ★ Baut für jede eingesetzte Programmiersprachen ein Base Image: Java, .NET, NodeJS etc.
- ★ Leitet die Microservices von diesem Base Image ab.
- ★ Haltet Euch an die Microservice Regeln - <https://12factor.net>
- ★ Erstellt Richtlinien für
  - Container Image Namensräume
  - User und Group-IDs in den Containern
  - Verwendet keine fixen Verbindungen, z.B. zu Logging Systemen
- ★ Beispiel DOCKERFILE
  - `FROM <unser Base Image>`
  - `USER 1000:1000`
  - `COPY <MicroService> .`
  - `CMD` oder `ENTRYPOINT`

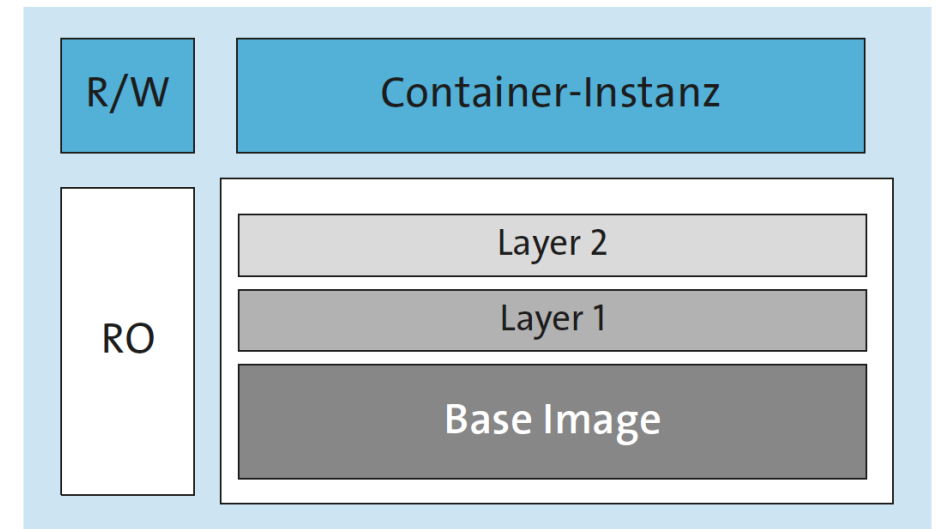
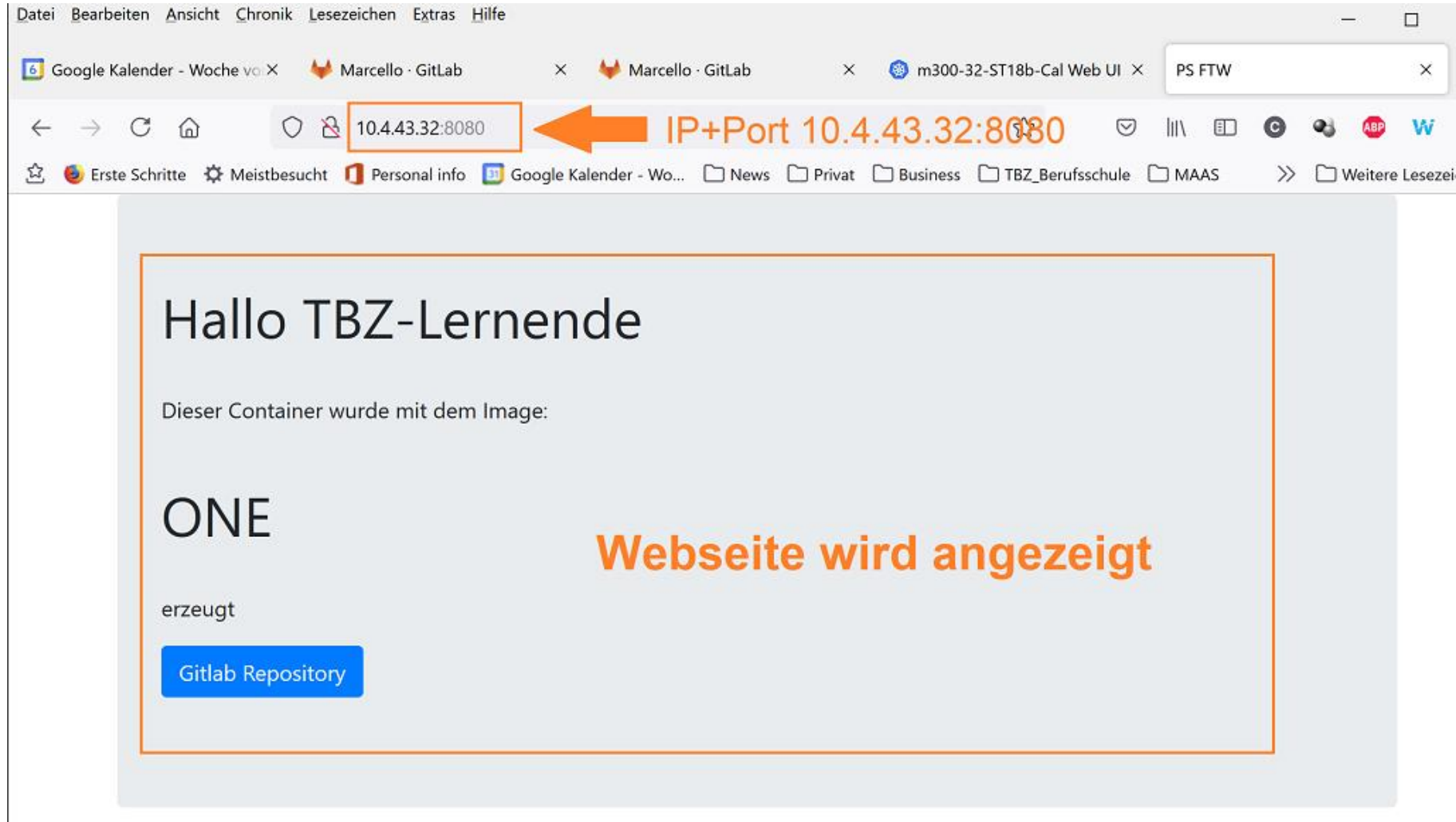


Abbildung 3.9 Schematischer Aufbau eines Images

# Selbstcheck

- ★ Ich kann mich mit einer Linux Maschine via SSH verbinden, mich auf dem Betriebssystem (Linux) zurechtfinden und Dateien editieren (z.B. mittels nano, vi oder Remote via Bitvise).
  - **NEIN:** Installiert [Docker Desktop](#), auf Eurem Notebook und fährt weiter Hands-on
  
- ★ **JA:** Hands-on in der MAAS Cloud und Cloud-init Scripten

# Hands-on: Container Images und Registries




[https://gitlab.com/ch-tbz-hf/Stud/cnt/-/tree/main/2\\_Unterrichtsressourcen/I#hands-on](https://gitlab.com/ch-tbz-hf/Stud/cnt/-/tree/main/2_Unterrichtsressourcen/I#hands-on)

# Wer verwendet was? (Trend)

## Entwicklung (Dev)

### 1. CONTAINERIZATION


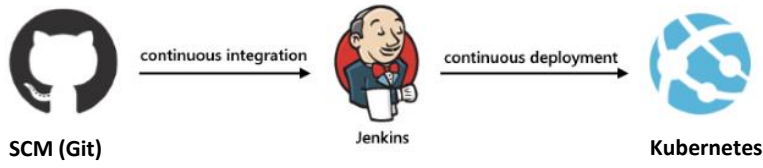
- Commonly done with Docker containers
- Any size application and dependencies (even PDP-11 code running on an emulator) can be containerized
- Over time, you should aspire towards splitting suitable applications and writing future functionality as microservices




## Tests/Integration/Paketierung

### 2. CI/CD




- Setup Continuous Integration/Continuous Delivery (CI/CD) so that changes to your source code automatically result in a new container being built, tested, and deployed to staging and eventually, perhaps, to production
- Setup automated rollouts, roll backs and testing
- Argo is a set of Kubernetes-native tools for deploying and running jobs, applications, workflows, and events using GitOps paradigms such as continuous and progressive delivery and MLOps


## Betrieb (Ops)

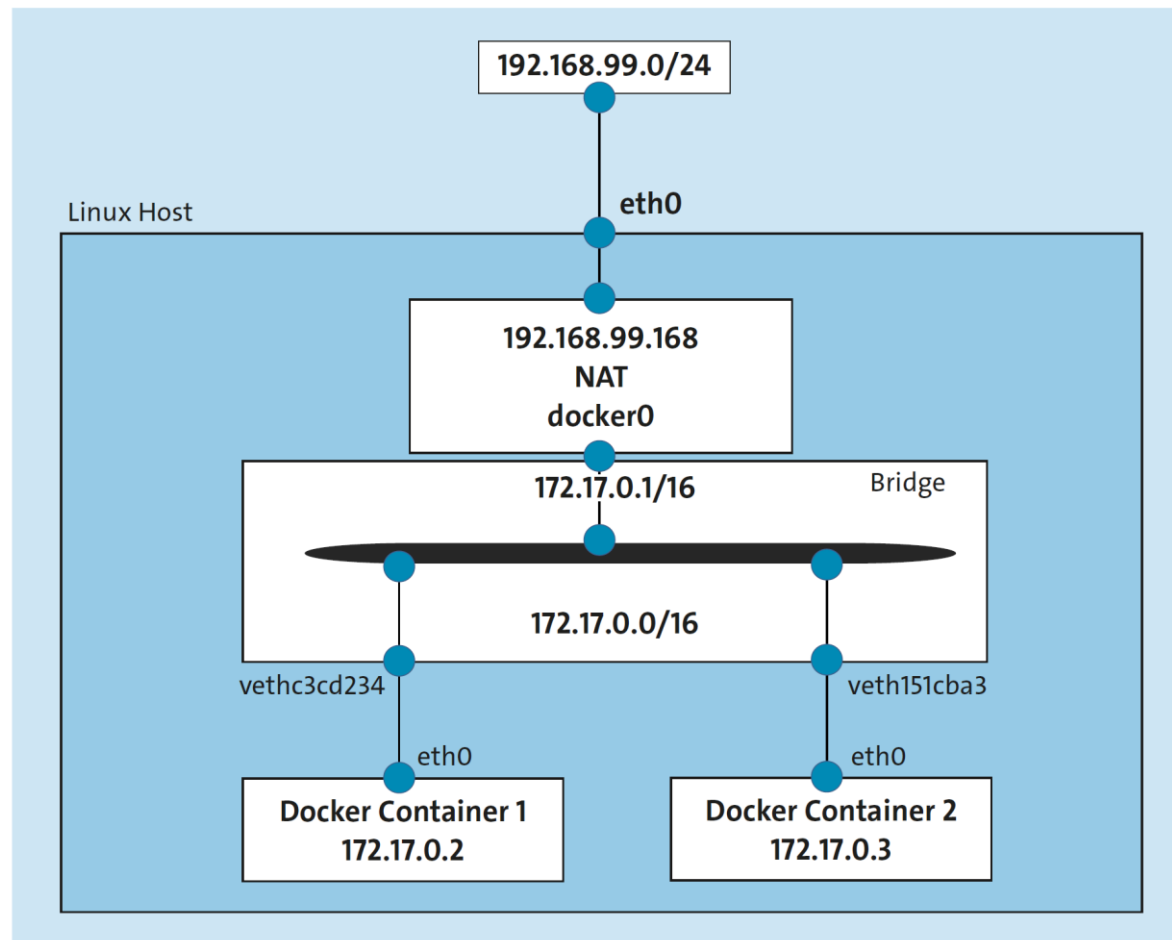
### 3. ORCHESTRATION & APPLICATION DEFINITION

- Kubernetes is the market-leading orchestration solution
- You should select a Certified Kubernetes Distribution, Hosted Platform, or Installer: [cncf.io/ck](https://cncf.io/ck)
- Helm Charts help you define, install, and upgrade even the most complex Kubernetes application


Anmerkung: Erfahrung des Kursleiters

# Docker Networking (K8s Networking, ohne NAT)



- ★ Wie alles im Container-Universum sind auch die Dinge im Bereich des Networkings für den Container-technisch Unbedarften etwas speziell, zumindest aber gewöhnungsbedürftig.
- ★ Die DevOps-Teams müssen sich hier etwas von der klassischen Sichtweise »*Meine VM und die darin residierenden Services sind immer via IP/Port XYZ zugänglich*« verabschieden.
- ★ **Das Docker-Netzwerkmodell spielt primär nur für Docker-Standalone-Nodes und in Swarm-Clustern eine Rolle und ist daher unter Kubernetes weitestgehend zu vernachlässigen.**

# Docker Networking: bridge, host, none und mehr

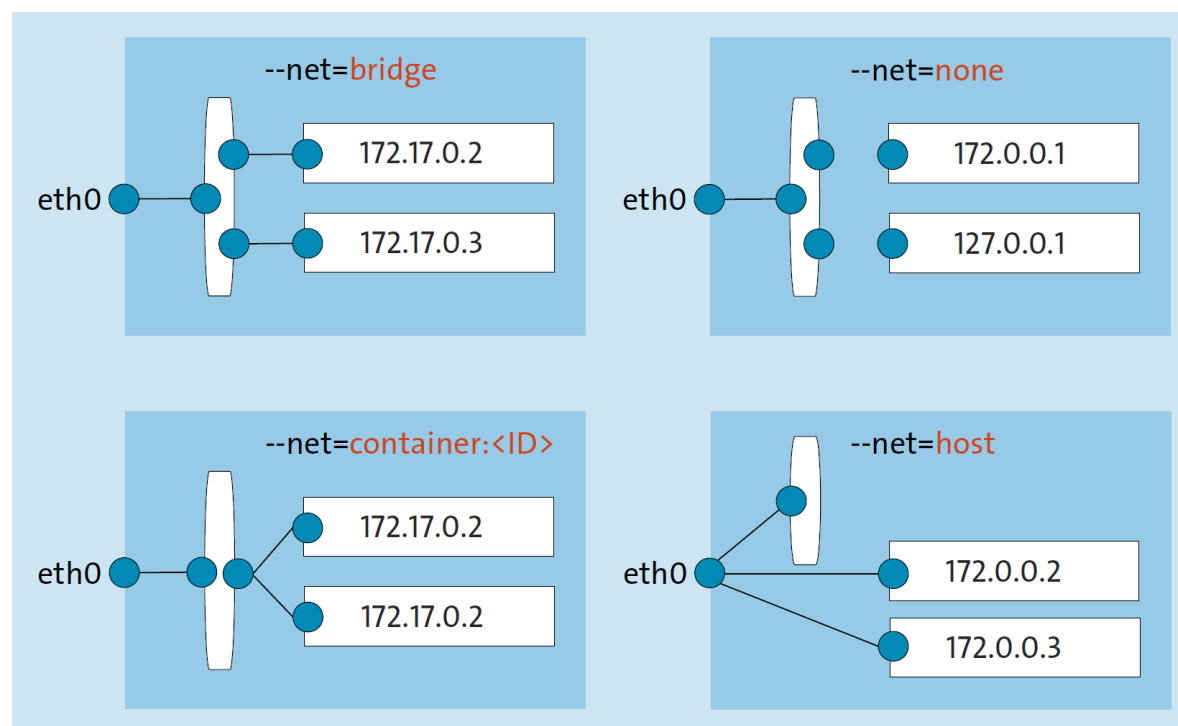


Abbildung 4.5 docker [container] run --network: Konnektivitätstypen

- ★ **--net=bridge** - Verwende die Docker Bridge. Der Container kommuniziert mit docker0
- ★ **--net=none** - Kein Netzwerk. Jeder Container ist netzwerktechnisch isoliert.
- ★ **--net=container:<ID>** - Verwende den Netzwerkstack/Netzwerk-Namespace eines anderen Containers. Ähnlich Kubernetes.
- ★ **--net=host** - Verwende das Host-Netzwerk. Der Container wird angewiesen, direkt auf den Netzwerkstack des Hosts unter Umgehung der Bridge zuzugreifen. Sicherheitstechnisch bedenklich!
- ★ **Im Gegensatz zu Kubernetes fehlt ein DNS Server.**

# Docker Networking: Legacy Links



Abbildung 4.6 Legacy Link Apache <-> OpenLDAP

## ★ Container miteinander verknüpfen (alt)

- <https://docs.docker.com/network/links/>

## ★ Beispiele:

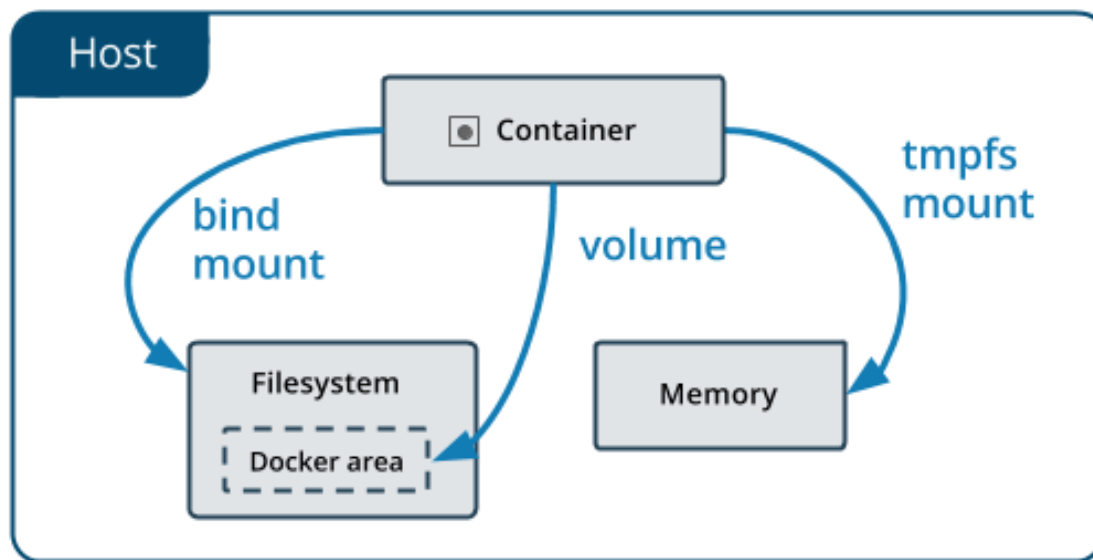
- SCS-ESI - <https://github.com/mc-b/SCS-ESI/tree/master/docker>  
Container varnish linkt order und common.  
Problem: varnish muss am Schluss gestartet werden.
- OSTicket - <https://hub.docker.com/r/osticket/osticket/>
- Adminer - [https://hub.docker.com/\\_/adminer/](https://hub.docker.com/_/adminer/)



# Docker Compose (K8s: YAML Dateien, HELM etc.)

- ★ **Docker-Compose** ein Tool aus dem Docker-Universum, mit dem die Erzeugung, das Ausrollen und die Verwaltung von multiplen Images/Container-Instanzen, die miteinander interagieren sollen, automatisiert werden kann.
- ★ Über **Compose** und entsprechende, **Yaml-basierte Template-Dateien** können wir unsere containerisierten Applikationen mit einem einzigen Kommando ausrollen.
- ★ Hinweis: [kompose](#) ist ein Werkzeug, um Benutzern die mit Docker Compose vertraut sind, beim Umzug nach Kubernetes zu helfen. Kompose nimmt eine Docker-Compose-Datei und übersetzt sie in Kubernetes-Ressourcen.

# Docker Persistenz (K8s: PersistentVolume)



- ★ Die Daten bleiben nicht erhalten, wenn ein Container nicht mehr vorhanden ist.
- ★ Es ist schwierig wenn mehrere Container auf die gleichen Daten zugreifen müssen.

## ★ Lösungen

- Daten auf den Container Host schreiben
- Ein Container Volume einrichten

## ★ Weitere Informationen

- <https://docs.docker.com/storage/>

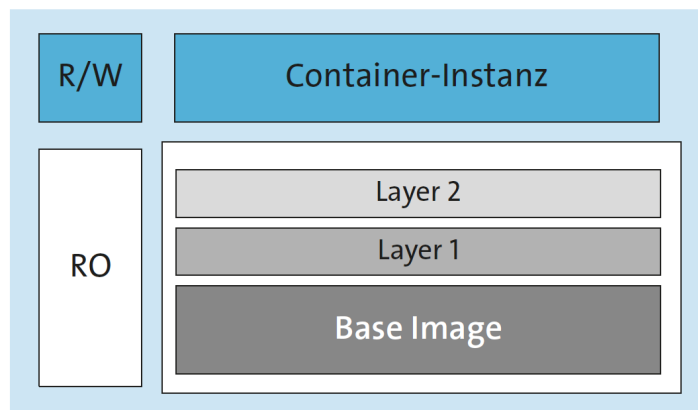


Abbildung 3.9 Schematischer Aufbau eines Images

# Reflexion

- ★ Container sind **nicht** neu.
- ★ Container ~~ersetzen~~ **ergänzen** Virtualisierung.
- ★ Container, **welcher der «Open Container Initiative Runtime Specification» entsprechen**, sind universell portabel.
- ★ Container sind ~~prinzipiell~~, **bei entsprechendem Handling (keine unnötige SW, nicht privilegiert, ...)**, sicher.
- ★ ~~Container heissen Docker.~~
  
- ★ **Anmerkung:** Erfahrungen des Kursleiters.

# Lernzielkontrolle

- ★ Sie haben einen Überblick über Container Laufzeitumgebungen wie Docker, CRI-O, containerd.

# Selbstcheck

- ★ Ich kann mich mit einer Linux Maschine via SSH verbinden, mich auf dem Betriebssystem (Linux) zurechtfinden und Dateien editieren (z.B. mittels nano, vi oder Remote via Bitvise).
  - **NEIN:** Installiert [Docker Desktop](#), auf Eurem Notebook und fährt weiter mit der Praktischen Arbeit
  
- ★ **JA:** Praktische Arbeit

# Auftrag für Praktische Arbeit



## ★ Problemstellung

- Die Entwickler haben x Web-Apps produziert und möchten diese als Container betreiben.

## ★ Aufgabe

- Verpackt 2 - 3 Services in Container Images und legt diese auf verschiedene Registries ab. Startet die Services als Container.
- Präsentiert, am Schluss, die Lösung (10') und die erworbenen Kompetenzen (H - I).

## ★ Zeit

- 4 Lektionen

# Präsentation der Lösungen



- ★ Team Vorstellung
- ★ Eure Lösung
  - Idee
  - Umsetzung
  - Probleme
- ★ Erworbene Kompetenzen (H - I)
- ★ Fazit



# Wo finde ich Container Runtimes?

## CNCF Cloud Native Landscape

Cloud Native Landscape v20180525

See the interactive landscape at [l.cncf.io](https://l.cncf.io)

Greyed logos are not open source

Database and Data Warehouse Streaming Source Code Management Application Definition and Image Build Continuous Integration / Continuous Delivery (CI/CD)

App Definition and Development

Scheduling & Orchestration Coordination & Service Discovery Service Management

Runtime - Container Runtime (12)

Host Management / Tooling Provisioning Public

Cloud

containerd  
Cloud Native Computing Foundation (CNCF)  
★ 4,590

cri-o  
Cloud Native Computing Foundation (CNCF)  
★ 1,951

Firecracker  
Amazon Web Services  
★ 9,101  
MCap: \$879.83B

gVisor  
Google  
★ 8,959  
MCap: \$834.36B

kata  
OpenStack  
★ 1,415

lxd  
Canonical  
★ 2,368  
Funding: \$12.8M

Nabla Containers  
IBM  
★ 170  
MCap: \$120.9B

Pouch  
Alibaba Cloud  
★ 4,018  
MCap: \$458.57B

Open Container Initiative (OCI)  
★ 6,240

Singularity  
Sylabs  
★ 1,319

SmartOS  
Joyent  
★ 1,326  
Funding: \$131M

Unik  
Solo.io  
★ 2,011  
Funding: \$13.5M

<https://landscape.cncf.io/>

# Installation Runtime und Start Container

## Installation einer beliebigen Linux Distribution, z.B. Ubuntu, Fedora etc.

### ★ Docker (Ubuntu)

```
$ sudo apt-get update
$ sudo apt-get install docker-ce docker-ce-cli containerd.io
```

- Start Container mittels Docker oder Kubernetes.
- Build Images mittels Docker.

### ★ Cri-o (Fedora)

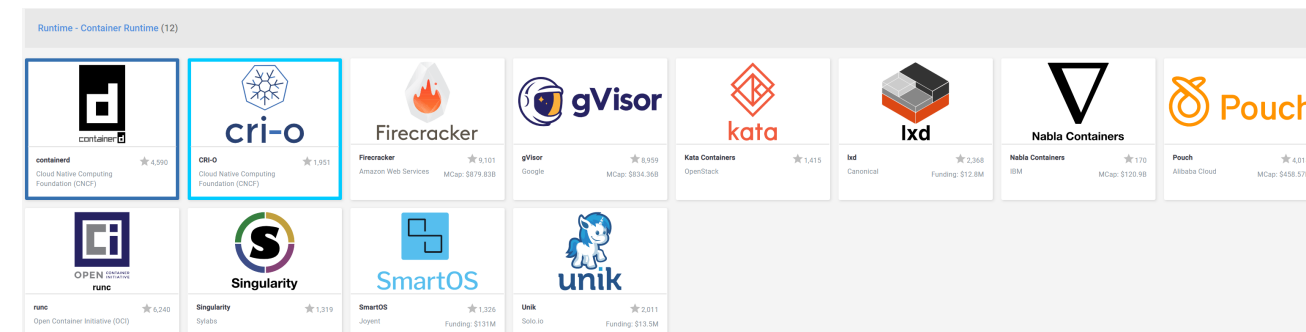
```
sudo dnf module enable cri-o:$VERSION
sudo dnf install cri-o
```

- Start Container mittels Kubernetes
- Build Images mittels [kaniko](#), [img](#), und [buildah](#)

### ★ containerd (Ubuntu)

```
## Install containerd
sudo apt-get update && sudo apt-get install -y containerd.io
```

- Start Container mittels Kubernetes
- Build Images mittels [kaniko](#), [img](#), und [buildah](#)



★ <https://landscape.cncf.io/>

<https://kubernetes.io/docs/setup/production-environment/container-runtimes/>  
[Migrating Kubernetes from Docker to containerd](#)

# Container-Datei- und Layer-Operationen mit Overlay(FS)

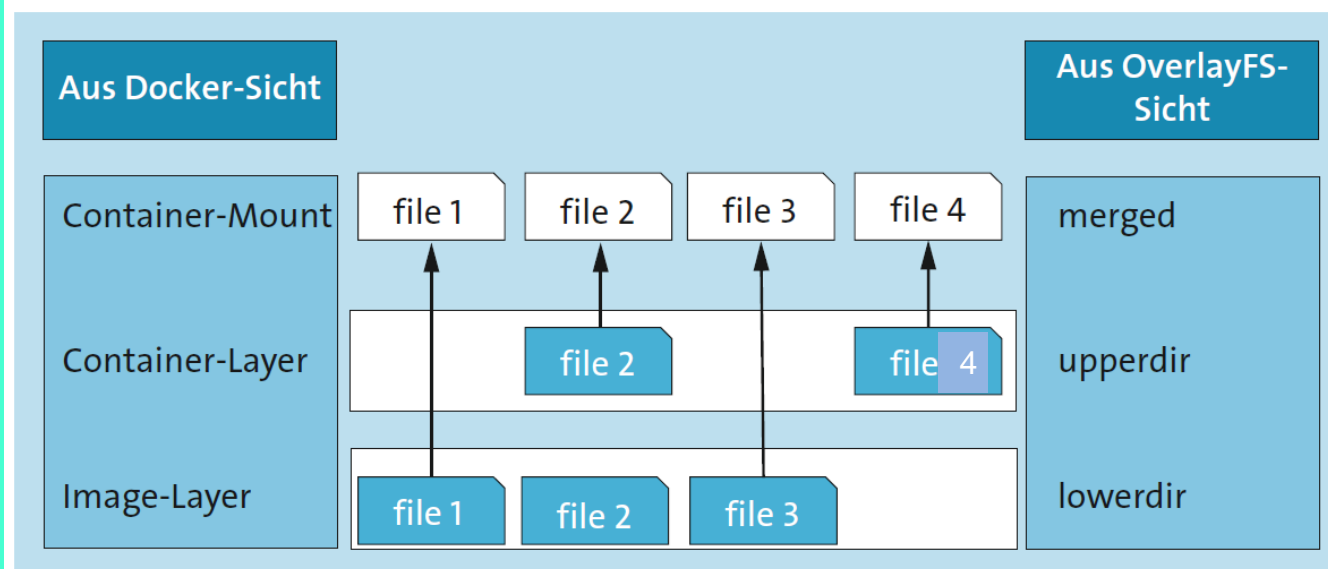


Abbildung 4.11 OverlayFS(2)-Modell

- ★ Die Datei (file1/3) existiert noch nicht im Read/Write-Layer des Containers (upperdir):
  - Die Datei wird der Image-Ebene (lowerdir) gelesen.
- ★ Die Datei (file4) existiert nur in der Container- (Read/Write-)Ebene (upperdir):
  - Die Datei wird aus dem Read/Write-Layer des Containers gelesen.
- ★ Die Datei (file2) existiert im Read/Write-Layer des Containers (upperdir) und in der Image-Datei-(Readonly-Layer-)Ebene (lowerdir):
  - Die Datei wird aus dem Container-Layer gelesen.
- ★ **merged:** Verzeichnis /var/lib/docker/overlay2