Docker

Container sind neu, Container sind in sich abgeschlossene Entitäten, Container ersetzen Virtualisierung, Container sind universell portabel, Container sind prinzipiell sicher, Container heissen Docker

(Einige der grössten Fehlannahmen im Hinblick auf das Thema Container und Docker)

Lernziele

★ Sie haben einen Überblick über Docker.

Zeitlicher Ablauf

- ★ Versionen, wichtige Meilensteine und Inkompatibilitäten
- ★ Funktionaler Überblick
- ★ Einfaches Image-Management
- ★ Betrieb und Management von Docker-Containern, Prozesse, Logging
- ★ Dedizierte Docker Image-Stände bauen (docker build) und verwalten
- ★ Docker Networking
- ★ Docker Compose
- ★ Docker Storage Driver und Volumes
- ★ Reflexion
- ★ Lernzielkontrolle

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.

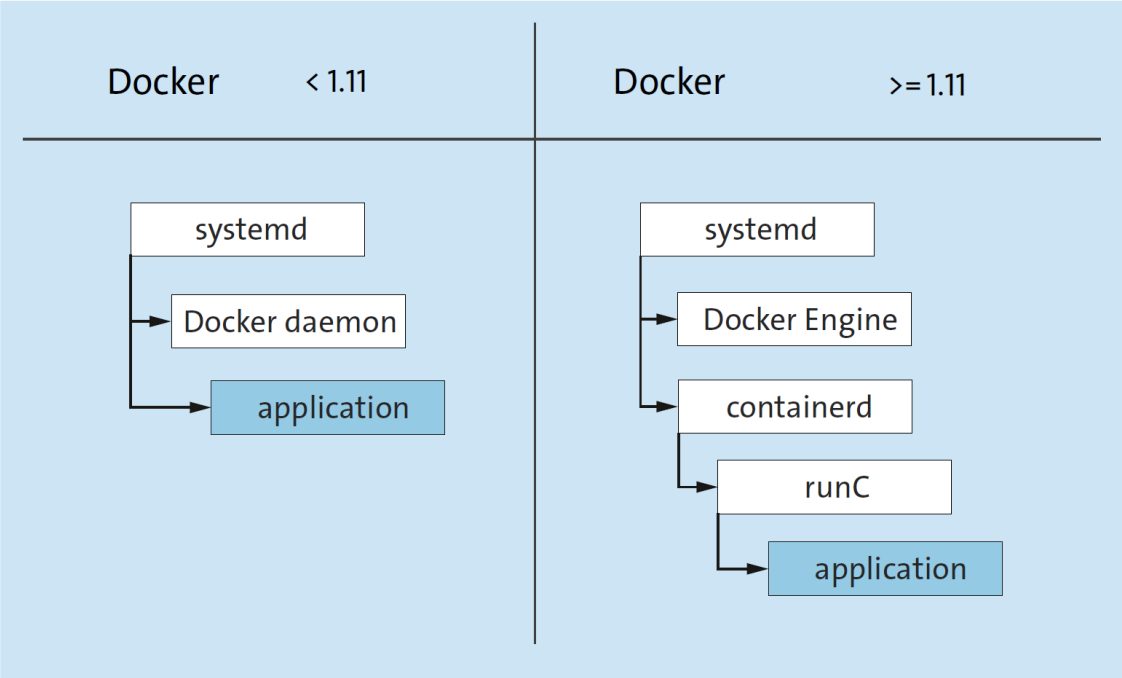


Abbildung 3.4 Funktionaler Docker Stack Pre-/Post 1.11

- ★ **Docker-LTS-Versionen?**
  - "All Docker Releases ›should‹ be backward compatible."

Quelle: Buch Skalierbare Container-Infrastrukturen, Kapitel 4.1

Funktionaler Überblick (1)

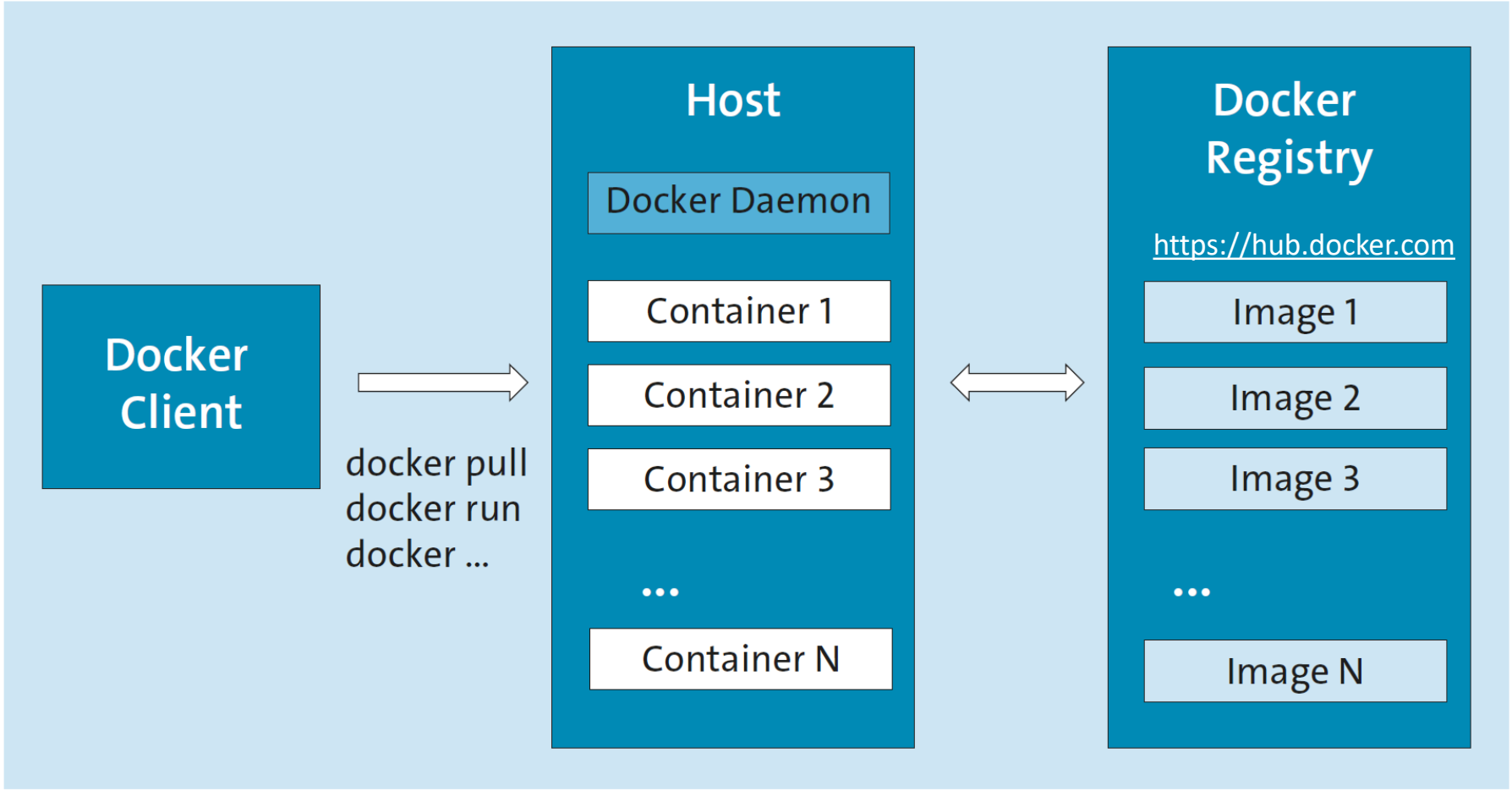


Abbildung 4.1 Docker Client, Daemon und Registry

Quelle: Buch Skalierbare Container-Infrastrukturen, Kapitel 4.1.5

Betrieb und Management von Docker-Containern

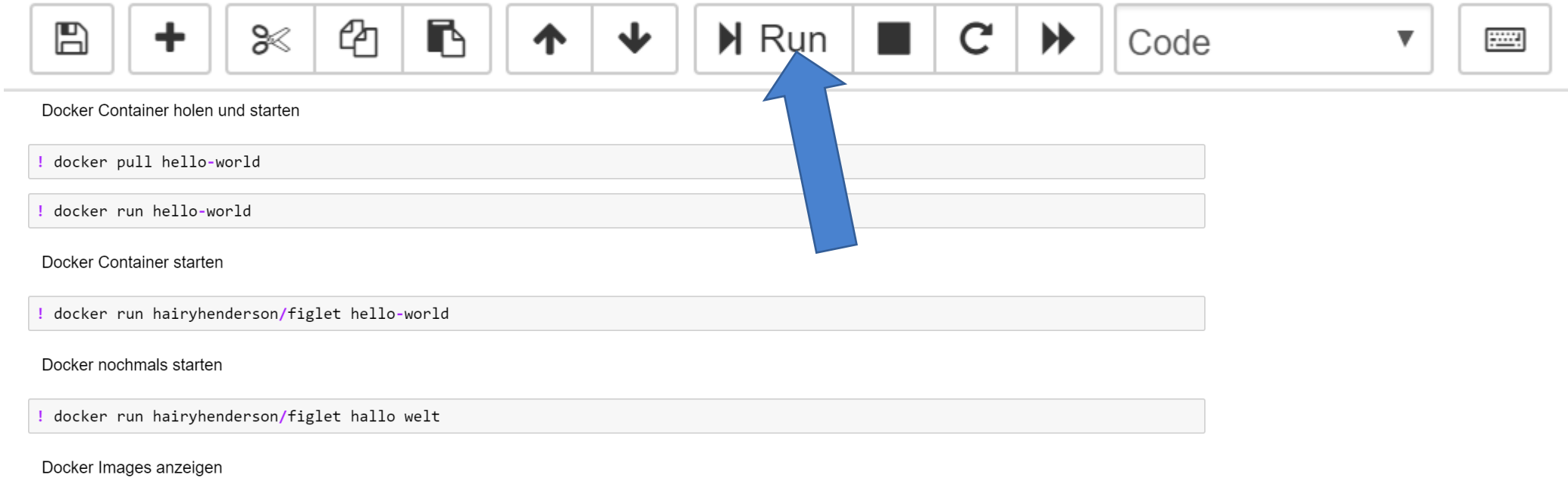
- **Befehlsaufbau:**
  - `docker <sub-command>`

- **attach**: an eine laufende Container Sitzung ankoppeln
- **build**: neues Image erzeugen
- **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
- Dokumenation: https://docs.docker.com/engine/reference/run/

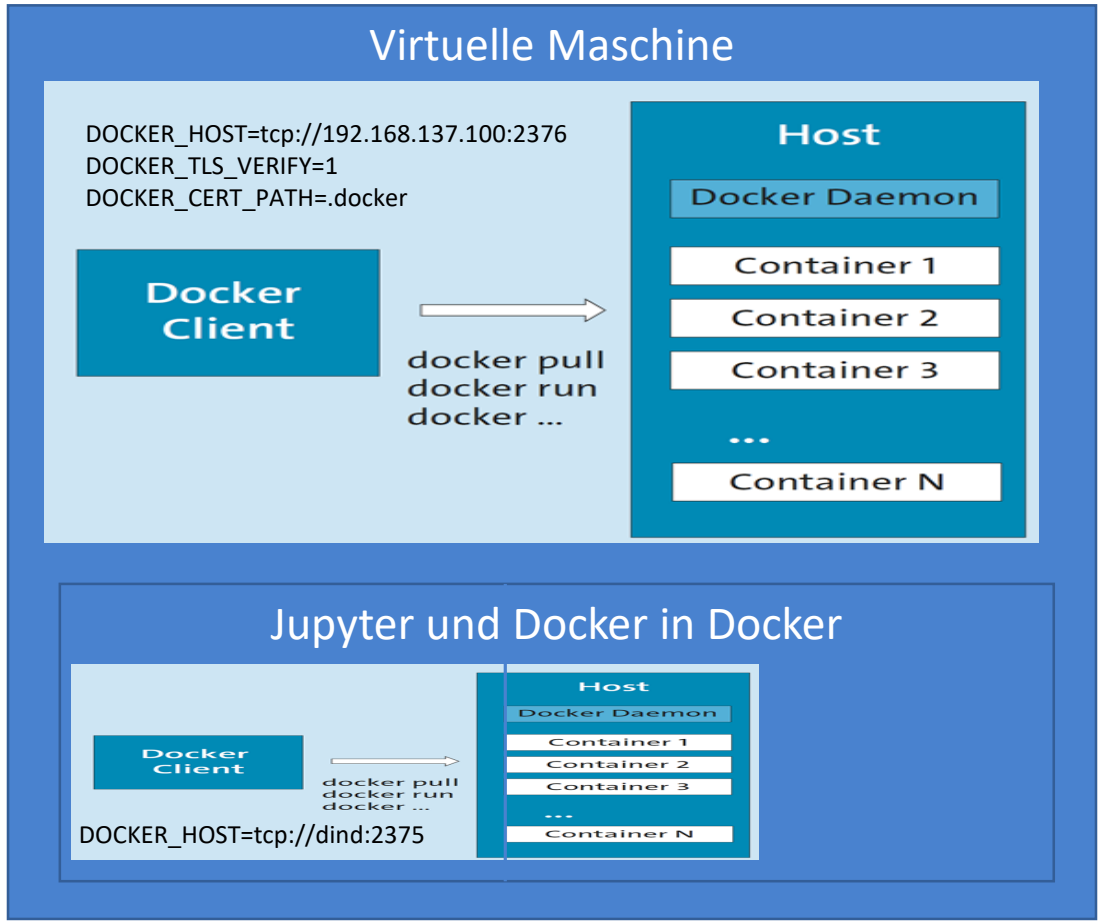
Quelle: Buch Skalierbare Container-Infrastrukturen, Kapitel 4.9

Übung: Funktionaler Überblick (03-1-Docker)

- ★ Startet die Jupyter (Lernumgebung) mit der «Docker in Docker» Umgebung
  - `kubectl apply -f duk/jupyter/jupyter-base.yaml`
  - `kubectl apply -f duk/jupyter/dind.yaml`
- ★ Wechselt auf den Link (rechts oben) und spielt die Übung durch.



Übungsumgebung: Docker in Docker



- ★ Die Virtuelle Umgebung (VM) basiert auf einer Linux Installation mit Docker und Kubernetes.
- ★ Innerhalb der VM laufen zwei Container:
  - Jupyter – eine Interaktive Lernumgebung mit dem Docker Client
  - Docker in Docker (DinD) mit einem Docker Daemon.
- ★ Der Zugriff auf den Docker Daemon unterscheidet sich von ausserhalb der VM und von Jupyter -> DinD

Einfaches Image-Management

★ Wie kommen wir zu Images?

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

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.

Quelle: Buch Skalierbare Container-Infrastrukturen, Kapitel 4.6

Zusammenspiel zwischen Docker Host, Client und Registry

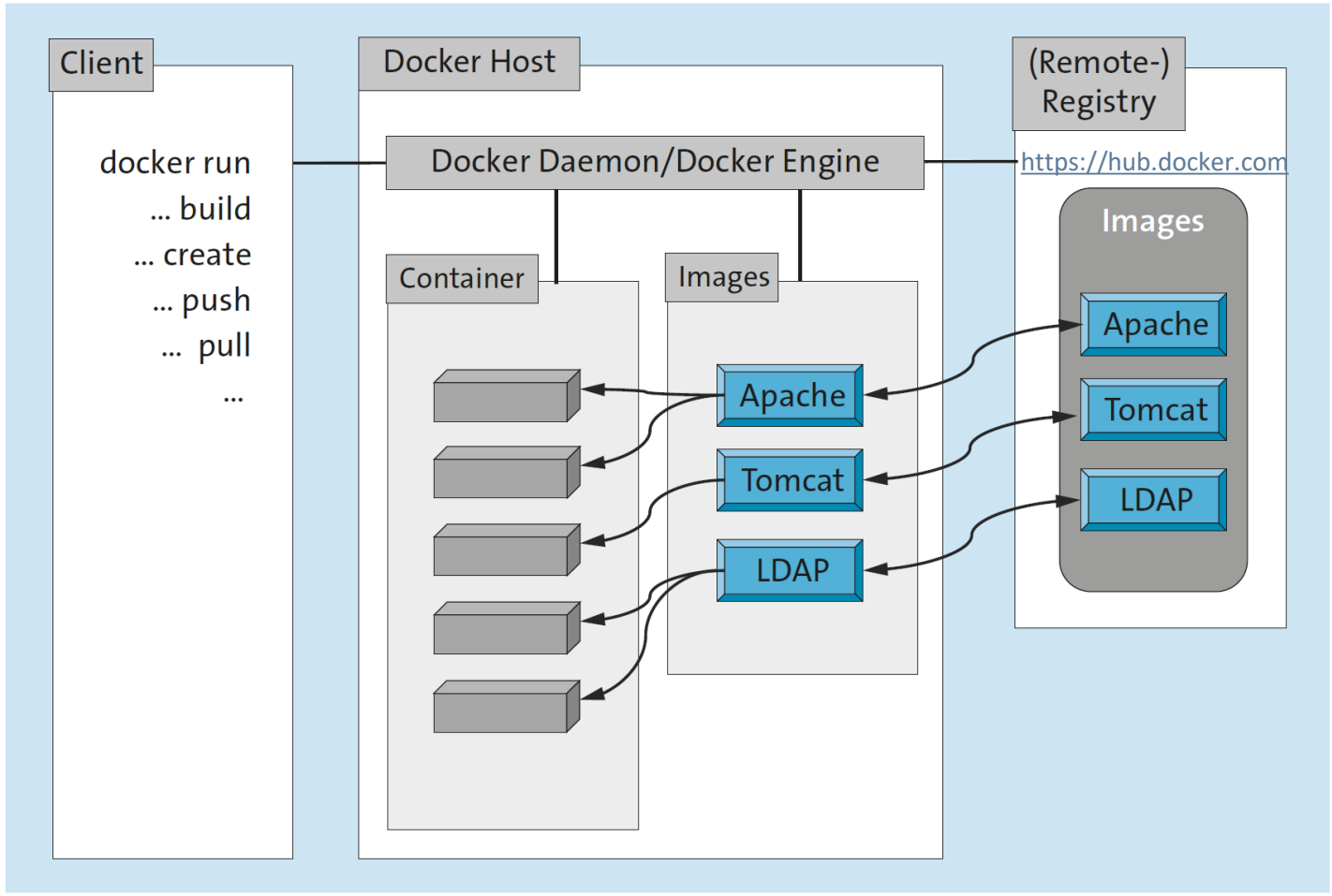


Abbildung 4.2 Docker Client, Docker Host und (Remote-)Registry

Docker-Namensräume und das Default Registry-Problem

★ **Default Registry**: Jeder search- oder pull-Befehl des docker-Kommandos läuft Default gegen die docker.io-Registry:

- `docker info | grep -i registry`

★ **Namensräume**: [REGISTRY_HOST[:REGISTRY_PORT]/]REPOSITORY[:TAG]

- **REGISTRY_HOST/PORT**: optionale Angabe einer eigenen Registry
- **REPOSITORY (user/name):** bestehend aus Username und Name des Images. Der Username entspricht login auf https://hub.docker.com/. Offizielle Images haben keinen Usernamen.
- **TAG:** optionaler Tag, entspricht einer Versionnummer, teilweise mit Hinweisen auf Originalimage z.B. :3-alpine.

Quelle: Buch Skalierbare Container-Infrastrukturen, Kapitel 4.6.6 und https://docs.docker.com/engine/security/trust/content_trust/

Beispiele: Docker Namensräume

- ★ https://hub.docker.com
  - `docker run hello-world`
  - `docker pull misegr/bpmn-frontend`
- ★ Lokale Registry (https://github.com/mc-b/duk/tree/master/registry#docker-registry)
  - `docker pull localhost:5000/ubuntu`
- ★ Google Container Registry (https://cloud.google.com/container-registry/docs/overview?hl=de)
  - `docker pull k8s.gcr.io/kubernetes-dashboard-amd64`
- ★ Microsoft Azure Registry (https://azure.microsoft.com/en-in/services/container-registry/)
  - `docker run --rm mcr.microsoft.com/dotnet/core/samples`
- ★ Docker Registry Alternativen:
  - https://www.g2.com/products/google-container-registry/competitors/alternatives
  - https://www.slant.co/topics/2436/~best-docker-image-private-registries

Aufbau eines Images

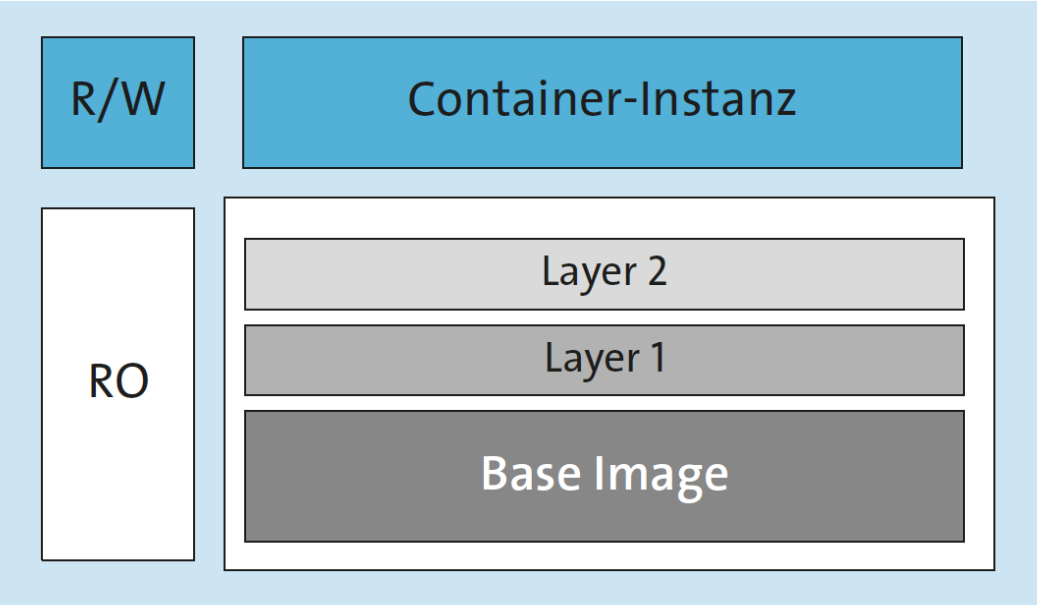
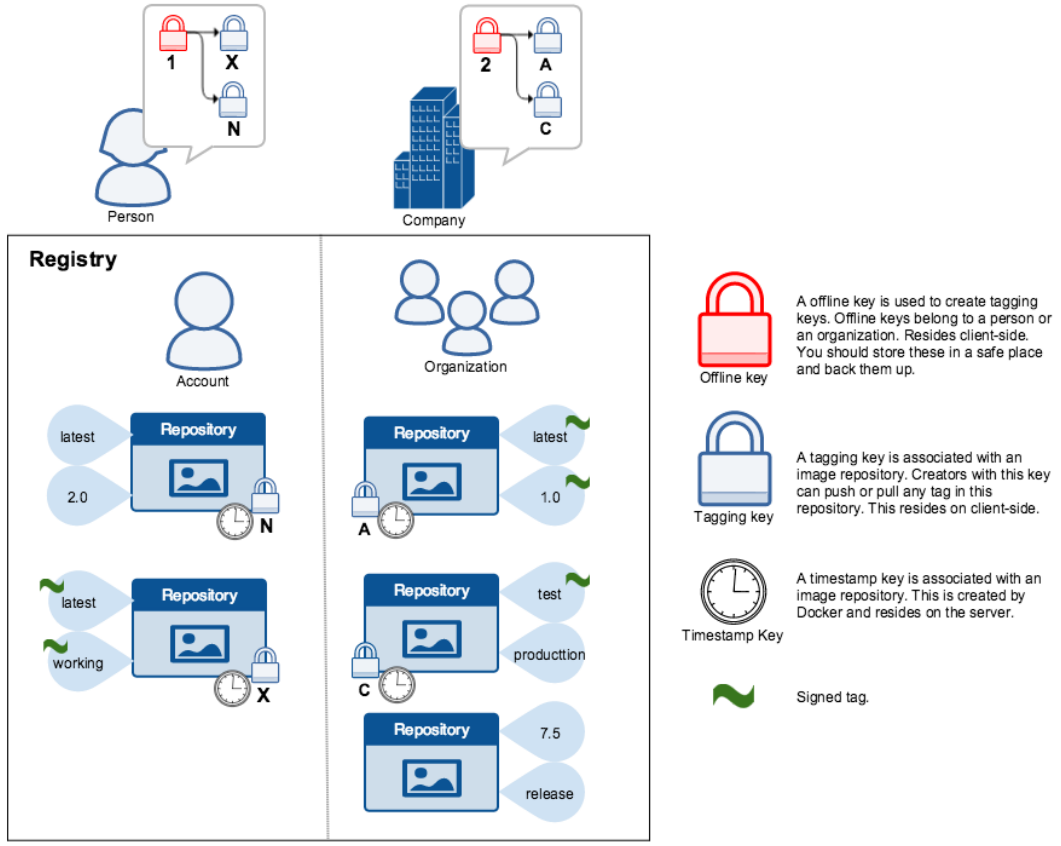


Abbildung 3.9 Schematischer Aufbau eines Images

- ★ Container Instanz: Read / Write
- ★ Container Image: Read Only
  - Besteht aus Layers (Snapshots), max. 127.
  - Jede Zeile in Konfigurationsdatei (Dockerfile) erzeugt einen neuen Layer.
  - **V2:** Multi-Architektur-Images -Plattformspezifische Versionen
  - **V2:** Images mit eindeutigem Hash
- ★ Ausgabe der Layers
  - `docker image history hello-world`
- ★ Docker image Befehl
  - https://docs.docker.com/engine/reference/commandline/images/
  - Siehe auch: https://www.opencontainers.org/

Quelle: Buch Skalierbare Container-Infrastrukturen, Kapitel 3.7.2

Trusted Images



★ Buch Skalierbare Container-Infrastrukturen, Kapitel 4.7 mit SUSE/SLES mit sle2docker.

★ Docker Handbuch

- Use Trusted Images: https://docs.docker.com/engine/security/trust/

```
export DOCKER_CONTENT_TRUST=1
```

```
$ docker push <username>/trusttest:testing
The push refers to a repository [docker.io/<username>/trusttest] (len: 1)
9a61b6b1315e: Image already exists
902b87aaaec9: Image already exists
latest: digest: sha256:d02adacee0ac7a5be140adb94fa1dae64f4e71a68696e7f8e7cbf9db8dd49418 size: 3220
Signing and pushing trust metadata
You are about to create a new root signing key passphrase. This passphrase
will be used to protect the most sensitive key in your signing system. Please
choose a long, complex passphrase and be careful to keep the password and the
key file itself secure and backed up. It is highly recommended that you use a
password manager to generate the passphrase and keep it safe. There will be no
way to recover this key. You can find the key in your config directory.
Enter passphrase for new root key with id a1d96fb:
Repeat passphrase for new root key with id a1d96fb:
Enter passphrase for new repository key with id docker.io/<username>/trusttest (3a932f1):
Repeat passphrase for new repository key with id docker.io/<username>/trusttest (3a932f1):
Finished initializing "docker.io/<username>/trusttest"
```

Übung : Einfaches Image-Management (03-2-Docker)

★ Erstellt einen Account auf https://hub.docker.com/.

Erstellt einen Account auf https://hub.docker.com/.

Erstellt, auf http://localhost:32188/ ein neues `Terminal` rechts oben mit `New` und logt Euch auf Docker Hub ein:

```
docker login
```

Tagt die zwei vorhanden Images mit Eurem Namen und pusht diese in auf Docker Hub

z.B.:

```
docker tag hello-world <username>/hello-world
docker push <username>hello-world
```

```
! docker ...
```

Docker Images als im «tar»-Format speichern und laden

```
! docker save hello-world -o hello-world.tar
! docker rmi hello-world
! docker image ls
```

```
! ls -ls *.tar
```

```
! docker load -i hello-world.tar
! docker image ls
```

Prozessverwaltung im Container

★ Der erste Prozess **im Container** bekommt die **Id: 1**.

- `docker exec c1 ps ax`

```
! docker container exec c1 ps ax
```

```
  PID TTY      STAT   TIME COMMAND
    1 ?        Ss     0:00 /bin/bash -c while true; do sleep 30000; done
    5 ?        S      0:00 sleep 30000
    6 ?        Rs     0:00 ps ax
```

★ Ausserhalb des Containers ist er mit der effektiven Prozess Id sichtbar.

- `docker container top c1`
  - ★ `root` **`6471`** `/bin/bash -c while true; do sleep 30000; done`

★ **Die Sichtweise auf die Prozesse des Containers ist von innen (innerhalb des Containers) und von aussen (Host), völlig unterschiedlich.**

Quelle: Buch Skalierbare Container-Infrastrukturen, Kapitel 4.10

Docker logging

- ★ Seit Docker 1.6 können unterschiedliche Log Driver für Container-Instanzen zum Einsatz kommen, einstellbar entweder global über die Startoption --log-driver des Docker Daemons dockerd oder über **docker [container] run** beim Start einer Container-Instanz.
- ★ Zudem können wir per --log-level die Verbosität des Loggings innerhalb der Container-Instanzen festlegen, ebenfalls global oder pro Instanz. Als Log-Level stehen zur Verfügung: debug, **info** [Default], warn, error, fatal.

- ★ Weitere Informationen
  - docker logs - https://docs.docker.com/engine/reference/commandline/logs/
  - Configure logging drivers - https://docs.docker.com/config/containers/logging/configure/

Quelle: Buch Skalierbare Container-Infrastrukturen, Kapitel 4.12

Übung: Betrieb und Management von Docker-Containern (03-3-Docker)

Docker Container, im Hintergrund, mit Namen c1 starten

```
! docker container run --detach --name c1 ubuntu /bin/bash -c "while true; do sleep 30000; done"
```

Laufende Container ausgeben

```
! docker container ps
```

Container stoppen und Kontrolle ob nicht mehr vorhanden

```
! docker container stop c1
! docker container ps
```

Gestoppten Container wieder starten und Kontrolle ob wieder vorhanden

```
! docker container start c1
! docker container ps
```

Änderungen im Dateisystem des Containers

Erstellen einer Datei im Container

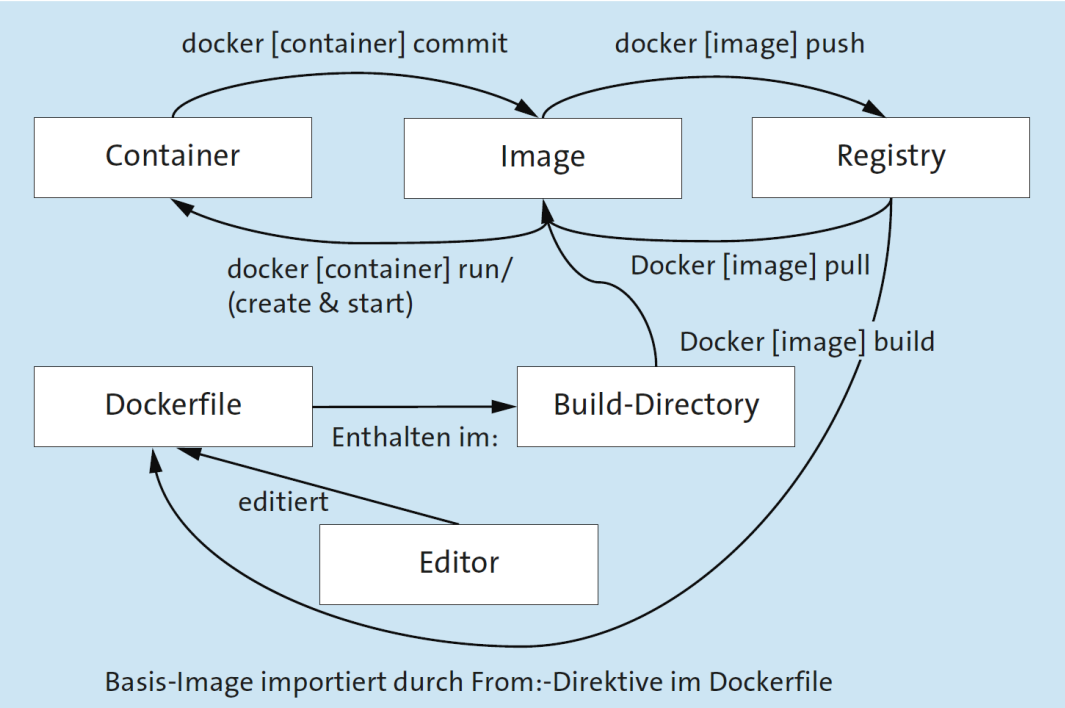
```
! docker container exec c1 /bin/bash -c "echo 'Ein Test um `date`' >/test.txt"
```

Dedizierte Docker Image-Stände bauen (docker build) 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) commited.
- ★ 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.

Quelle: Buch Skalierbare Container-Infrastrukturen, Kapitel 4.17

Dedizierte Docker Image-Stände bauen (docker build) 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 docker/4.15.5_co7httpd

```
FROM centos

RUN yum install -y httpd httpd-
tools net-tools bind-utils

ADD ["index.html",
"/var/www/html" ]

HEALTHCHECK  --interval=1m --
timeout=10s --retries=2 CMD curl -
f http://localhost/ || exit 1

EXPOSE 80

CMD ["/usr/sbin/httpd","-D",
"FOREGROUND"]
```

Quelle: Buch Skalierbare Container-Infrastrukturen, Kapitel 4.17

Best-Practice/File-Hierarchie

```
[root@jake1 ~]# ls -la {co*,sle*}
co7-httpd:
insgesamt 12
drwxr-xr-x   2 root root   37  4. Mär 12:06 .
dr-xr-x---. 20 root root 4096  4. Mär 12:06 ..
-rw-r--r--   1 root root  385  4. Mär 12:06 .bashrc
-rw-r--r--   1 root root  413  4. Mär 12:06 Dockerfile
sles12-redis:
insgesamt 12
drwxr-xr-x   2 root root   43  4. Mär 12:06 .
dr-xr-x---. 20 root root 4096  4. Mär 12:06 ..
-rw-r--r--   1 root root  413  4. Mär 12:06 Dockerfile
-rw-r--r--   1 root root 1728  4. Mär 12:06 nsswitch.conf
```

- ★ Da sich das Dockerfile jeweils auf ein bestimmtes Image bezieht (z. B. centos7-httpd oder sles12-redis), sollte auf dem Build-Host eine entsprechende Datei- / Verzeichnisstruktur für jedes zu bauende Image erzeugt werden.
- ★ Dies ist umso wichtiger, da die ADD/COPY-Kommandos (diese werden zum Kopieren von Dateien, Verzeichnissen und Tarballs in den Containern verwendet) die zu kopierenden Dateien in dem jeweiligen Ordner erwarten, in welchem sie vom Dockerfile beim Build aufgerufen werden.

Quelle: Buch Skalierbare Container-Infrastrukturen, Kapitel 4.18

Dockerfile-Direktiven/-Instruktionen

- ★ **FROM** – von welchem Basisimage abgeleitet
- ★ **MAINTAINER** – Autor (Achtung: In neueren Docker-Versionen deprecated. Stattdessen sollte Label <maintainer=<>> verwendet werden.)
- ★ **RUN** – im Read/Write Layer des Intermediate Container auszuführende Aktionen
- ★ **ENV** – Umgebungsvariablen
- ★ **EXPOSE** – Portmappings
- ★ **ENTRYPOINT** – »fixes« Startkommando für primäre Applikation
- ★ **CMD** – überschreibbares Startkommando für primäre Applikation
- ★ **LABEL** – eindeutige Bezeichnungen für spätere Selektion
- ★ **ADD** – Files, Directories, remote URLs in Image kopieren
- ★ **COPY** – Files in Image kopieren
- ★ **VOLUME** – (persistente) Volumes für Image definieren
- ★ **USER** – User für Kommandoausführung festlegen
- ★ **WORKDIR** – Setze das Working Directory für CMD, RUN, COPY, ADD, ENTRYPOINT
- ★ **ARG** – Build Arguments
- ★ **ONBUILD** – Postbuild Trigger für spätere, auf diesem Image basierende Builds

Quelle: Buch Skalierbare Container-Infrastrukturen, Kapitel 4.17.3 und https://docs.docker.com/engine/reference/builder/

Übung: Dedizierte Docker Image-Stände bauen und verwalten (03-4-Docker)

Nun wollen wir ein Docker Image erstellen, dazu brauchen wir ein Dockerfile.

Im Verzeichnis ist schon eines Vorhanden und wir können es einfach mal Ausgeben:

```
! cat Dockerfile
```

Um das Anschliessend das Docker Image zu bilden

```
! docker build -t mysql .
```

Das Ergebnis können wir uns mit `docker image ls` anschauen

```
! docker image ls
```

Ebenfalls können die einzelnen Buildschritte sichtbar gemacht werden

```
! docker history mysql
```

- ★ Weitere Beispiele siehe Buch Skalierbare Container-Infrastrukturen, Kapitel 4.15
- ★ Offizielle Docker Images: https://hub.docker.com/search/?isAutomated=0&isOfficial=1&page=1&pullCount=0&q=%2A&starCount=0
- ★ https://github.com/mc-b/devops/tree/master/docker

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-Ordners** 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
- ★ Multi-Line-Argumente in **Befehlen (alphanumerisch)** sortieren
- ★ **Build-Cache** (Gilt der Cache während des Build-Vorgangs einmal als invalidiert, so werden alle folgenden Instruktionen ohne Berücksichtigung des Caches durchgeführt.)
- ★ Siehe auch: Best practices for writing Dockerfiles

Quelle: Buch Skalierbare Container-Infrastrukturen, Kapitel 4.18

Beispiele: Dockerfile

- ★ hello-world - minimales Dockerfile
  - 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
- ★ 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
- ★ .NET Core Docker Samples
  - https://github.com/dotnet/dotnet-docker/blob/master/samples/README.md
  - Tip: Visual Studio zur Erzeugung eines Dockerfiles verwenden. Nächste Folie beachten!

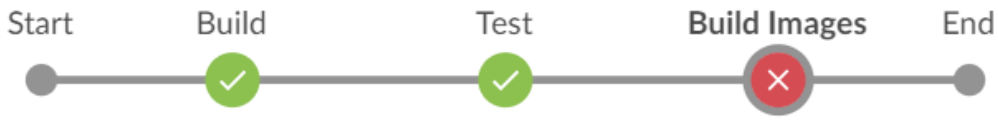
Build Best Pratices: multi-stage builds vs. CI

- ★ multi-stage builds
- ★ es fehlen
  - Unit-Tests
  - Build Historie …

```
FROM golang:1.7.3
WORKDIR /go/src/github.com/alexellis/
RUN go get -d -v golang.org/x/net/htm
COPY app.go .
RUN CGO_ENABLED=0 GOOS=linux go build

FROM alpine:latest
RUN apk --no-cache add ca-certificate
WORKDIR /root/
COPY --from=0 /go/src/github.com/alex
CMD ["./app"]
```

```
FROM microsoft/dotnet:2.2-sdk AS build
WORKDIR /app

# copy csproj and restore as distinct layers
COPY dotnetapp/*.csproj ./dotnetapp/
COPY utils/*.csproj ./utils/
WORKDIR /app/dotnetapp
RUN dotnet restore

# copy and publish app and libraries
WORKDIR /app/
COPY dotnetapp/. ./dotnetapp/
COPY utils/. ./utils/
WORKDIR /app/dotnetapp
RUN dotnet publish -c Release -o out


# test application -- see: dotnet-docker-unit-testing.md
FROM build AS testrunner
WORKDIR /app/tests
COPY tests/. .
ENTRYPOINT ["dotnet", "test", "--logger:trx"]


FROM microsoft/dotnet:2.2-runtime AS runtime
WORKDIR /app
COPY --from=build /app/dotnetapp/out ./
ENTRYPOINT ["dotnet", "dotnetapp.dll"]
```

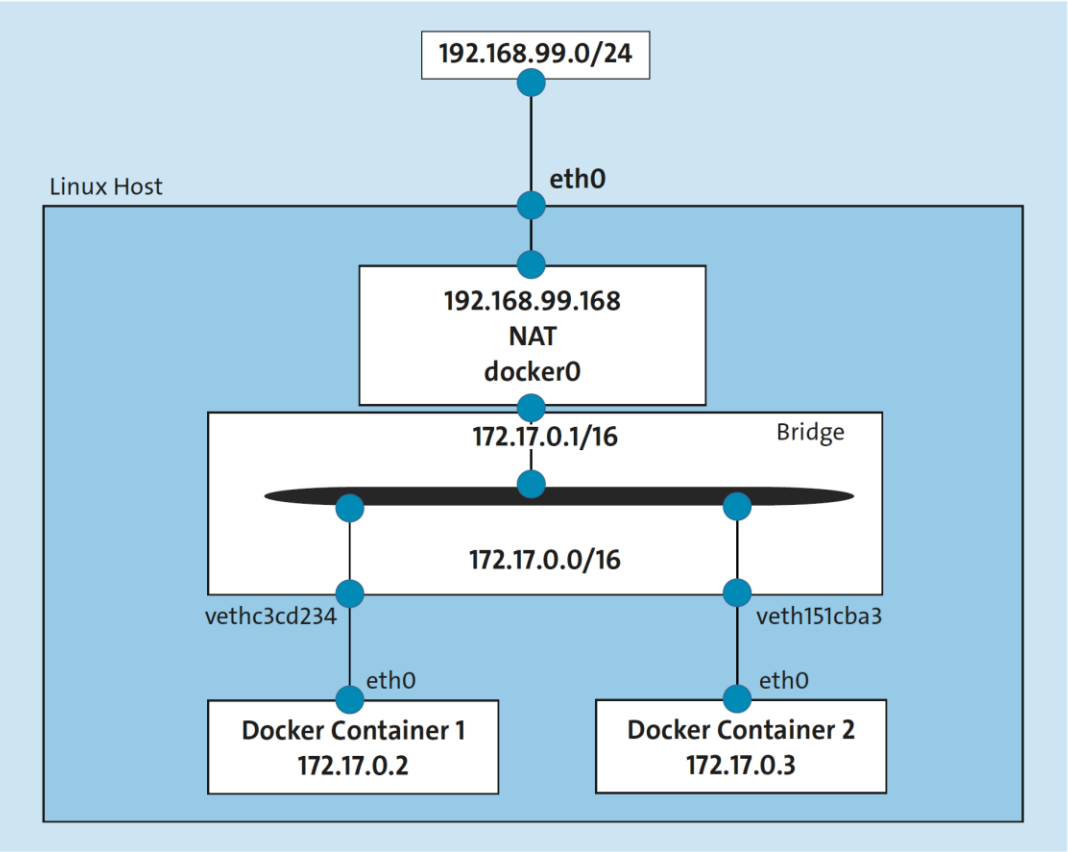
- ★ CI Pipeline




```
pipeline {
    agent none
    stages {
        stage('Build') {
            agent {
                docker {
                    image 'maven:3-alpine'
                    args '-v /root/.m2:/root/.m2'
                }
            }
            steps {
                sh 'cd scs-demo-esi-order/ && mvn -B -D
                archiveArtifacts 'scs-demo-esi-order/ta
            }
        }
```

```
stage('Build Images') {
    agent any
    steps {
            sh 'ls -l'
            sh 'cd docker/varnish     && /usr/bin/docker build -t scsesi_varnish .'
            sh 'cd scs-demo-esi-common && /usr/bin/docker build -t scsesi_common .'
            sh 'cd scs-demo-esi-order  && /usr/bin/docker build -t scsesi_order .'
    }
}
```

Docker Networking



- ★ Wie alles im Container-Universum sind auch die Dinge im Bereich des Networkings für den Container-technisch Unbedarften etwas speziell, zumindest aber gewöhnungs-bedürftig.
- ★ 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.**

Quelle: Buch Skalierbare Container-Infrastrukturen, Kapitel 4.19

Docker networks: bridge, host, none und mehr

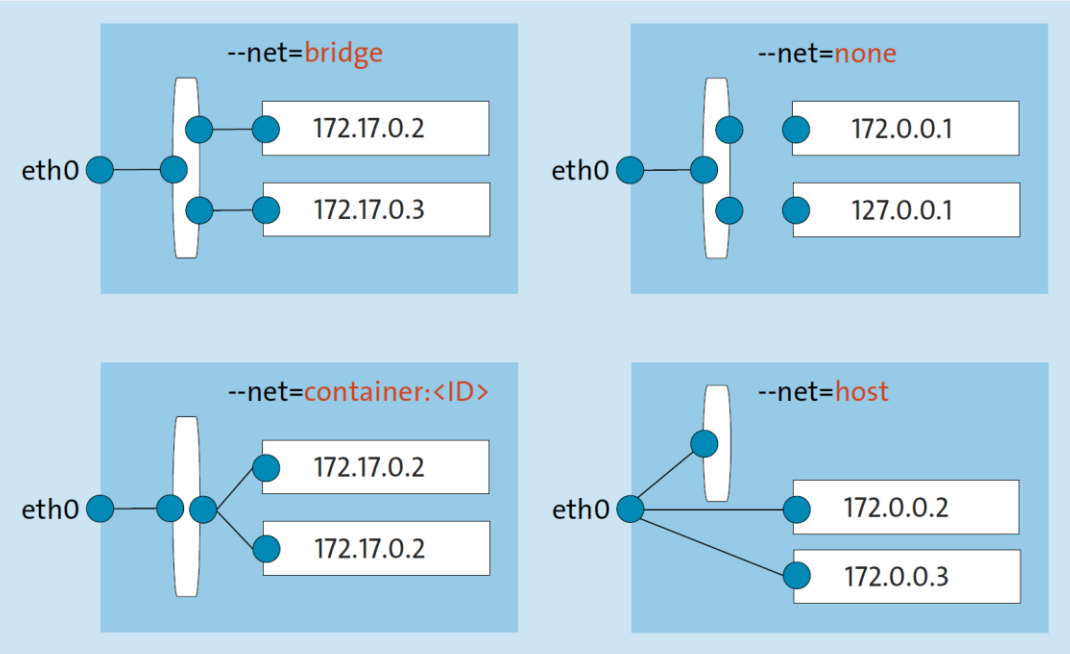


Abbildung 4.5 docker [container] run --network: Konnektivitätstypen

- **--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 wird Container angewiesen, direkt auf den Netzwerkstack des Hosts unter Umgehung der Bridge zuzugreifen. Sicherheitstechnisch bedenklich!

Quelle: Buch Skalierbare Container-Infrastrukturen, Kapitel 4.19

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/

Quelle: Buch Skalierbare Container-Infrastrukturen, Kapitel 4.20

Docker Compose

- **Docker-Compose** ein Tool aus dem Docker-Universum, mit dem die Erzeugung, das Ausrollen und die Verwaltung vom 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.

Quelle: Buch Skalierbare Container-Infrastrukturen, Kapitel 4.21

Übung: Docker-Compose

- ★ Startet das CLI und wechselt in die VM:
  - `kubeps.bat` oder `kubesh.bat`
  - `vagrant ssh`
- ★ In der VM, Installiert docker-compose, clont das SCS-ESI Git-Repository und führt docker-compose aus:
  - `apt install docker-compose`
  - `git clone https://github.com/mc-b/SCS-ESI.git`
  - `cd SCS-ESI/docker`
  - `docker-compose up`
- ★ **Hinweis**: zum Testen des Microservices ist statt http://localhost:8080, http://192.168.137.100:8080 einzugeben.

Reflexion

- ★ Container sind **nicht** neu.
- ~~★ Container sind in sich abgeschlossene Entitäten.~~
- ★ 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 priviligiert, …),** sicher.
- ~~★ Container heissen Docker.~~

- ★ **Anmerkung**: Erfahrungen des Kursleiters.

Lernzielkontrolle

★ Sie haben einen Überblick über Docker.