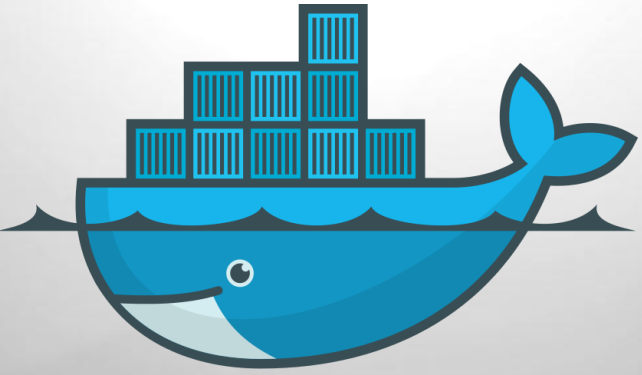
HTL LEONDING

How To



docker

Informatik

Medientechnik

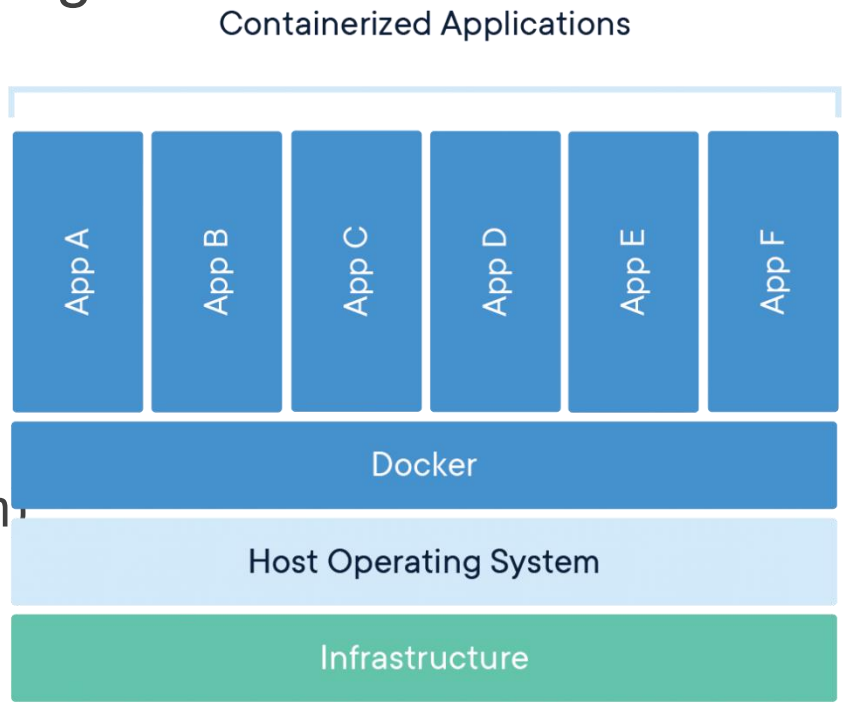
Elektronik - Technische Informatik

Medizintechnik

Docker Container

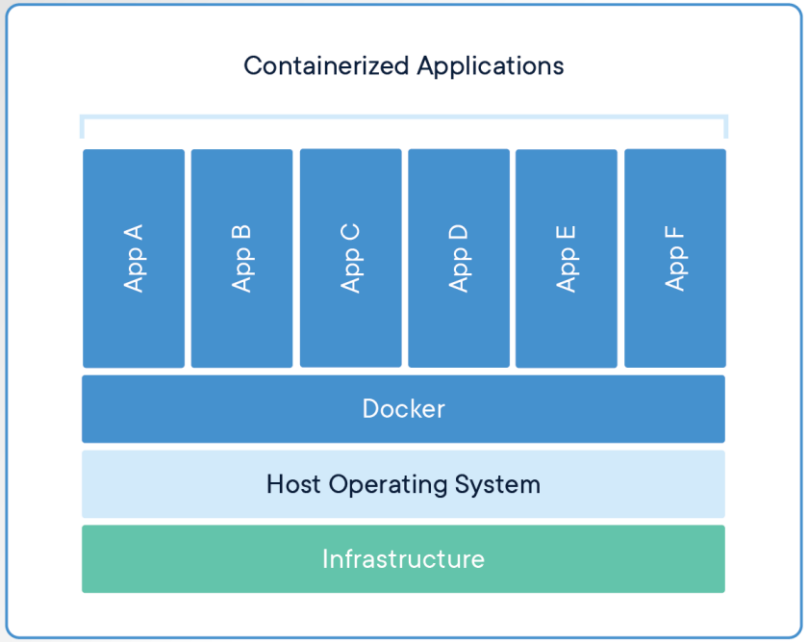
„A docker container is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings”

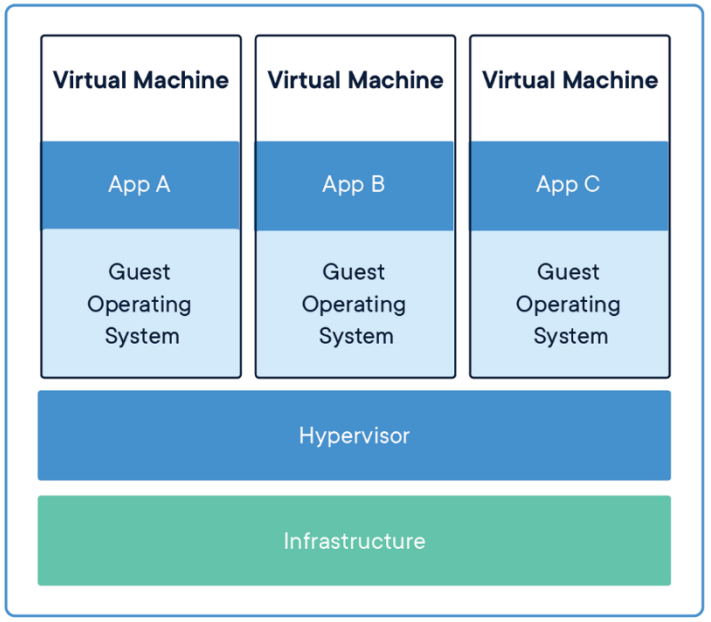
- Läuft out-of-the-box auf jedem System, das Docker installiert hat.
- Kann mit anderen Containern kommunizieren.
- Basiert auf einem Image, das über die Docker Registry (dockerhub.com) bezogen wird oder lokal selbst erstellt wurde



HTL LEONDING

Docker versus Virtual Machine





- Mehrere Container teilen sich dasselbe OS und laufen als unabhängige Prozesse im User Space
- ***Tens of MBs***

- Jede VM enthält ein vollständiges Betriebssystem
- ***Tens of GBs***

HTL LEONDING

Docker: Begriffe

Docker Image	Datei, die alles enthält, was zum Ausführen einer Applikation als Docker-Container notwendig ist. Setzt sich aus mehreren Schichten (Layers) zusammen. Schreibgeschützt.
Docker Container	Die Laufzeit-Instanz eines Docker Image. Kann gestartet und gestoppt werden und enthält Ausführungsparameter zu einem konkreten Docker-Image (z.B. Port, unter dem der Container erreichbar ist).
Docker Registry	Dockerhub.com: Enthält alle öffentlich verfügbaren Docker Images (offizielle und inoffizielle) inklusive Gebrauchsanweisungen und Versionshistorie. Kann auch für die Verwaltung von privaten Docker Images verwendet werden. Commands: *docker pull, docker push*
Dockerfile	Textdatei, die Befehle zum Erstellen eines docker image enthält (eigene Syntax, z.B. FROM, RUN, COPY, WORKDIR, CMD). Jeder Befehl resultiert in einem eigenen Image Layer.
Docker Compose	Tool zum Konfigurieren und Administrieren einer Gruppe von Containern
Docker-Host	System, auf dem der Docker-Daemon und der Docker-Client läuft
Docker-Client	Setzt auf Docker-Daemon auf, der auf dem Docker-Host läuft. Wird durch die Binary *docker* repräsentiert.

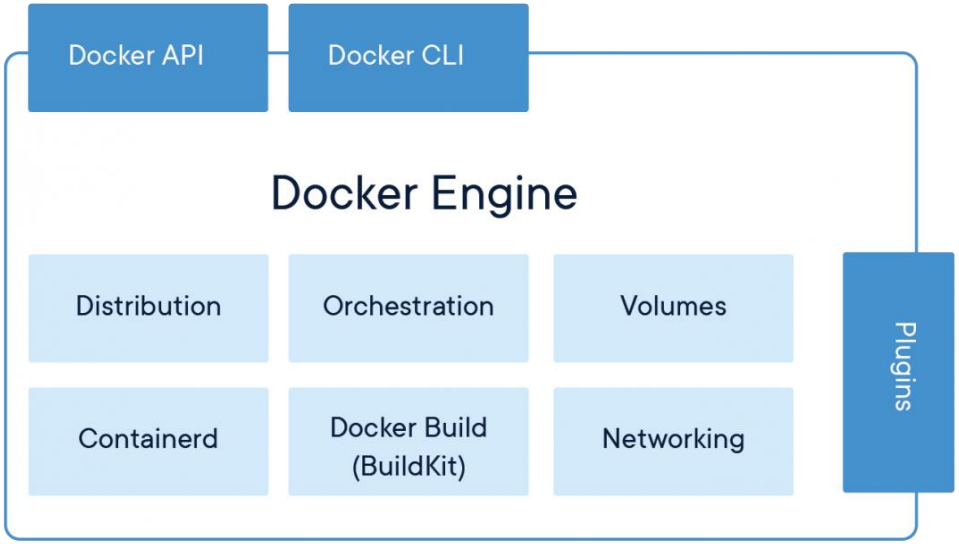
Docker Installation: Linux

- Docker CE (Community Edition)
  - Enthält Docker Engine und Docker Client (CLI)
  - Installation mit get.docker.com-Script, z.B. für Ubuntu: https://docs.docker.com/install/linux/docker-ce/ubuntu/#install-using-the-convenience-script

```
$ curl -fsSL https://get.docker.com -o get-docker.sh
$ sudo sh get-docker.sh
  sudo usermod -aG docker your-user
```

  - Test:

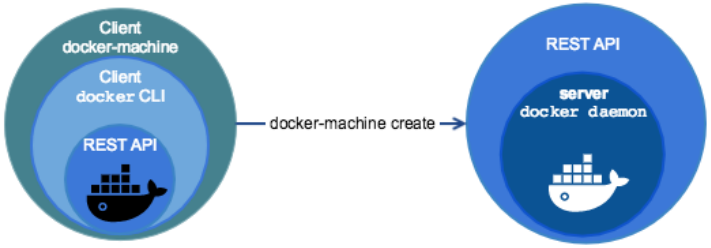
```
docker run hello-world
```



Linux: Docker-Machine

- Docker-Machine:
  - Für die Verwaltung mehrerer Docker-Hosts
  - Letzte Version suchen auf https://github.com/docker/machine/releases/
  - Beispiel für v0.16.1:

```
curl -L
https://github.com/docker/machine/releases/download/v0.16.1/docke
r-machine-`uname -s`-`uname -m` > /tmp/docker-machine
&& chmod +x /tmp/docker-machine
&& sudo cp /tmp/docker-machine /usr/local/bin/docker-machine
```



HTL LEONDING

Linux: Docker-Compose

- Letzte Version suchen auf https://github.com/docker/compose/releases
- Download und Install (Bsp: V1.23.2):

```
sudo curl -L
"https://github.com/docker/compose/releases/download
/1.23.2/docker-compose-$(uname -s)-$(uname -m)" -o
/usr/local/bin/docker-compose

sudo chmod +x /usr/local/bin/docker-compose
```

- Test:

```
$ docker-compose --version
docker-compose version 1.23.2, build 1110ad01
```

Docker-Installation Windows

- Docker Desktop for Windows
  - Nur für Microsoft Windows 10 Professional oder Enterprise 64 bit
  - Enthält Docker-Engine, Docker-Machine, Docker-Compose
- Docker Toolbox for Windows
  - Für Windows 10 Home Edition oder ältere Versionen
  - Verwendet Virtual Box als Docker-Host
  - Enthält Docker-Engine, Docker-Machine, Docker-Compose und Oracle VM VirtualBox

Docker-Installation Mac

- Docker Desktop for Mac:
  - https://docs.docker.com/docker-for-mac/install/

Die wichtigsten Befehle

Versionsinformationen	`docker –version`
	`docker-machine –version`
	`docker-compose –version`

Pull latest Hello-World-Image, starte Container	`docker run hello-world`
Pull und Start eines nginx-Webserver-Containers auf Port 80	`docker run --detach --publish=80:80 --name=webserver nginx`

Verwaltung aller lokalen Docker Images	`docker image ls`
	`docker image rm ..`
	`docker image pull …`
	`docker image inspect …`

Verwaltung aller Docker Container	`docker container ls`
	`docker container run …`
	`docker container start …`
	`docker container stop …`
	`docker container ps [-a]`
	`docker container rm ..`

Nginx Webserver

```
docker container run --detach --publish
80:80 --name webserver nginx
```

- Sucht Image im lokalen Image-Cache, andernfalls Download von Docker Hub (default: latest)
- Erzeugt einen neuen Container und macht ihn startbereit
- Weist virtuelle IP-Adresse im privaten Docker-Netzwerk zu
- Öffnet Port 80 auf dem Host und leitet auf Port 80 im Docker-Netzwerk weiter
- Startet den Container mithilfe des CMD aus dem Dockerfile

```
docker container logs webserver
docker container top webserver
docker container ls –a
docker container stop/start <container_id>
docker container rm <container_id>
```

`docker`-Optionen (Auszug)

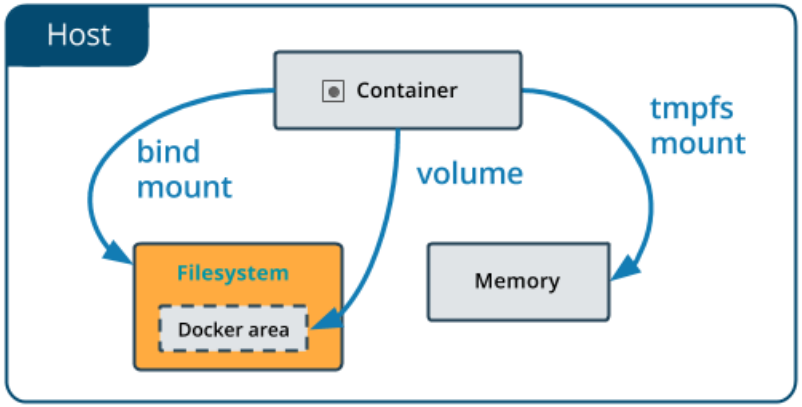
-i	Interaktiv – Man kann mit dem Container interagieren
-t	Bindet Ein- und Ausgaben an den interaktiven Container (tty)
-d	Daemonmode – Sinnvoll für Container, die Server enthalten
-p	Port-Binding in der Form <Hostport>:<Containerport> Somit können zB mehrere Container vom gleichen Image laufen, und jeweils auf einen anderen Port am Host gebunden werden
-v	Volume-Link in der Form <Host-Dir>:<Container-Dir> Ein Verzeichnis vom Container wird auf ein Verzeichnis vom Host-System gelinkt
-name	Benennt einen Container. Ansonsten ist nur Zugriff über die generierte ContainerID möglich (so viele Stellen davon, bis diese eindeutig ist)

```
> docker container run -it alpine /bin/sh
/ # echo "Hello from Container"
Hello from Container
/ # exit
```

Startet Container und führt `/bin/sh` aus

Container & Persistente Daten

- Container ist kurzlebig (start – stop – restart - …), aber Daten sollten persistent sein



- **Named Volumes**
  - Ordnerstruktur im Container wird als persistentes Volume definiert => unter Docker-Verwaltung
- **Bind Mounts**
  - Ordner/Datei-Struktur im Host-Filesystem wird in den Container gemounted => unter Host-Verwaltung

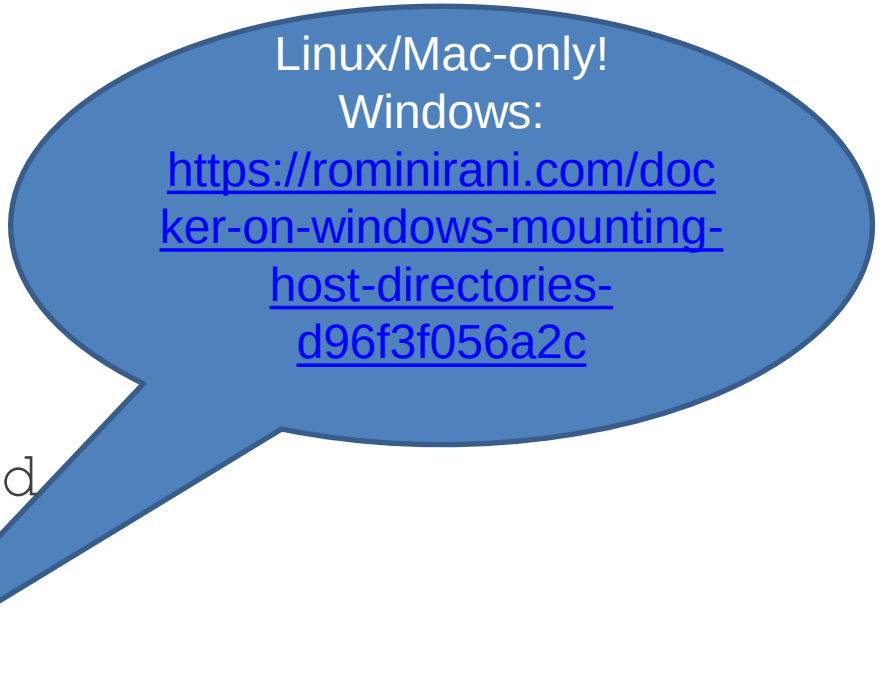
Named Volumes : Bind Mounts

- Named Volumes
  - Usecase: DBs von DBMS
  - `docker volume ls`
  - `docker volume inspect <id>`
  - `docker volume create …`
  - Container mit named volume starten:
    - `docker container run … -v <name>:<path>`
- Bind Mounts
  - Usecase: Editieren von Web-Dateien am Host
  - Mapping Host-Location auf Container-Location
  - `… run –v /users/name/dir:/path/container`

Übung: Apache 2.4 Webserver

- Verwenden Sie folgendes Image vom Docker-Hub:
  - `httpd:2.4`
- Ordner-Struktur im Host-Filesystem:
  - Erstellen Sie einen neuen lokalen Ordner `webroot`
  - und legen Sie darin eine HTML-Datei an mit dem Namen `index.html`.
- Starten Sie nun einen Container basierend auf dem httpd-Image.
  - Mappen Sie den Container-Port `80` auf einen freien Port Ihres Rechners.
  - Weiters mounten Sie den Container-Pfad `/usr/local/apache2/htdocs/` auf den von Ihnen erstellten Ordner `webroot`.
  - Der Container soll als Daemon laufen
  - und den Namen `my-apache-app` erhalten.
- Versuchen Sie, die Webseite im Browser zu laden
  - (IP ermitteln mittels `docker-machine ip`)
- Stoppen und löschen Sie den Container wieder

Übung – Webserver Bind Mount



- `docker container run -d`
  `--name my-apache-app`
  `-p 99:80`
  `-v $PWD/webroot/`
  `:/usr/local/apache2/htdocs/`
  `httpd:2.4`
- `docker container stop my-apache-app`
- `docker container rm my-apache-app`

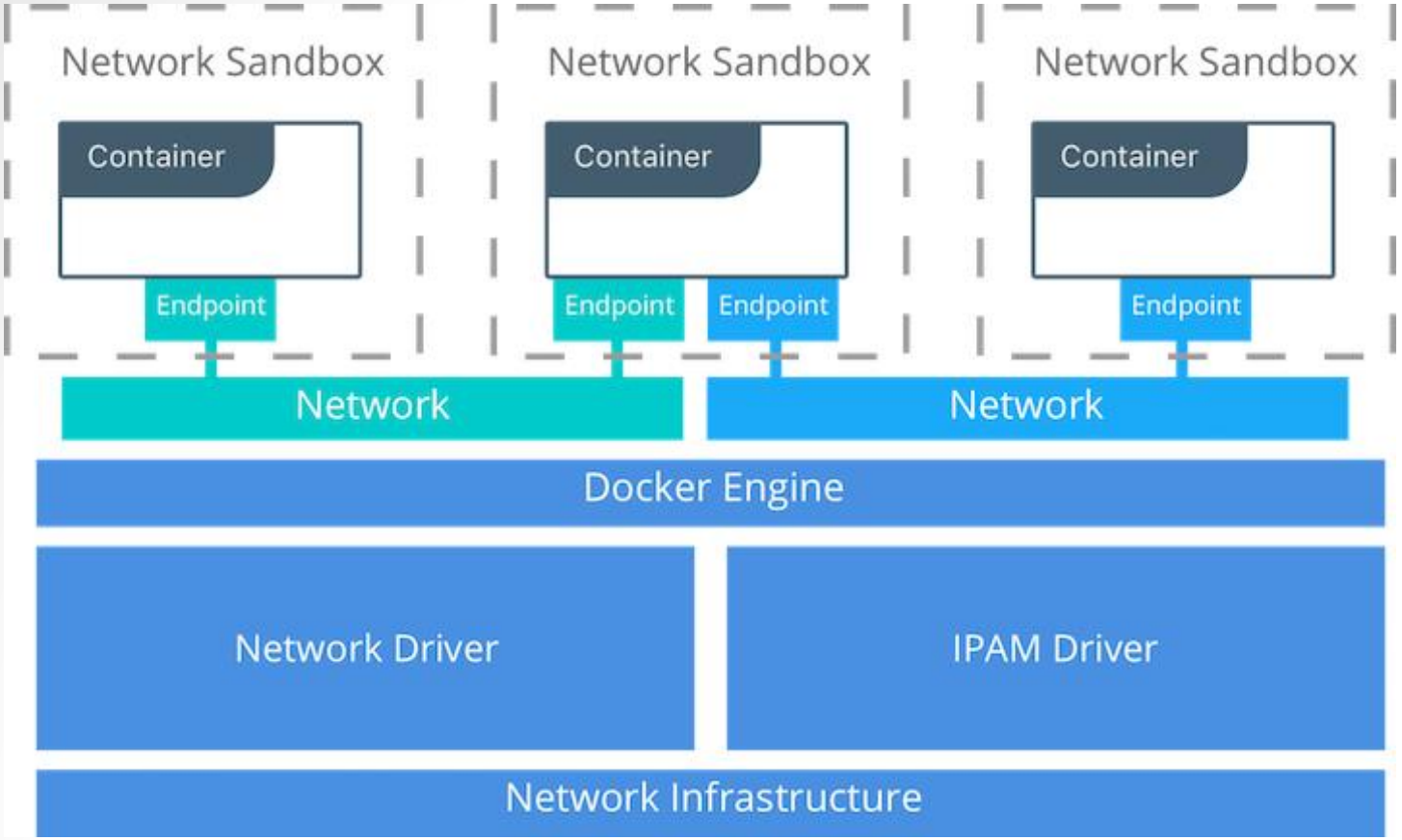
Übung: nginx, mysql, Apache2 httpd

- Nginx auf Port 80, httpd auf 8080, mysql auf 3306
- Mysql:
  - Image: mysql
  - Root-Passwort:
    - Verwende --env für die Umgebungsvariable MYSQL_RANDOM_ROOT_PASSWORD=yes
    - Verwende logs um das generierte Root-Passwort herauszufinden
    - Alternative: Verwende MYSQL_ROOT_PASSWORD=my_password
- Testen
- Zusammenräumen der Container mit stop / rm

In Container Verbinden

- docker container stats <id>
- docker container inspect <id>
- docker container exec –it <id> bash

Docker Networking



HTL LEONDING

Docker Network (--network)

- Default: Privates virtuelles Netzwerk „bridge“
  - Skip mit --`net=host`
  - DNS eingebaut (über Container-Names)

```
docker container port <name>
docker container inspect –format ´{{
.NetworkSettings.IPAddress }}` <name>
```

- Wird über eine NAT-Firewall auf die host-IP geroutet
- Container innerhalb des privaten Netzwerks erreichbar, von außen aber nur über „`--publish`“ (oder `-p`)
- Mehrere private Docker-Netzwerke konfigurierbar

```
docker network create my-network
docker network ls
docker network inspect my-network
```

Was sind Images?

- Jeder Container basiert auf einem Image
- Ein Image entspricht in etwa einem eingefrorenen Zustand eines Containers
- Images bauen aufeinander auf. Bei der Konfiguration wird jeweils ein Basis-Image angegeben.

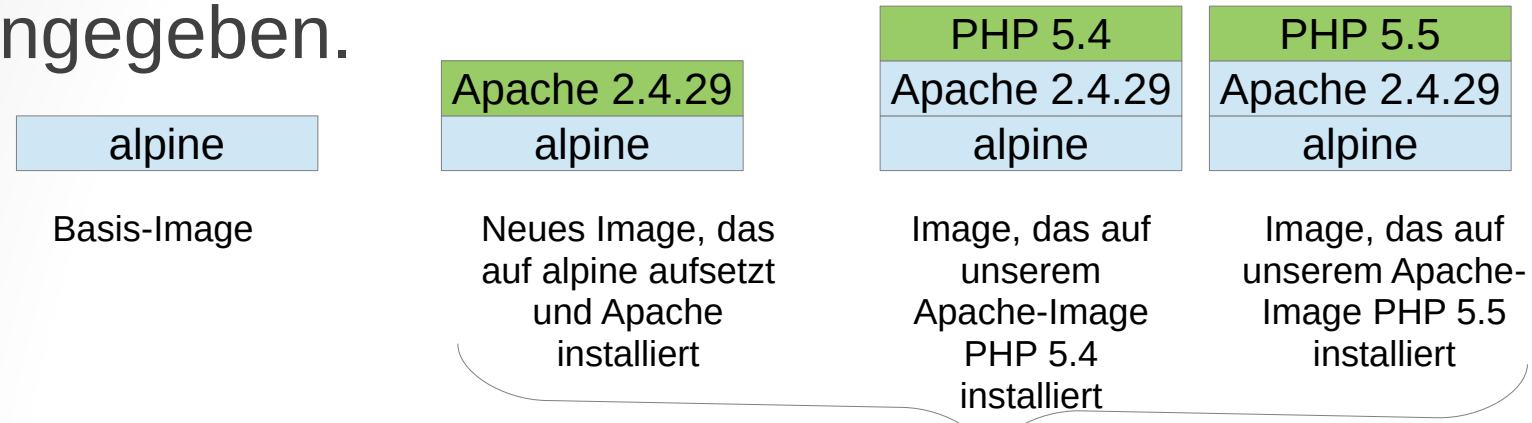
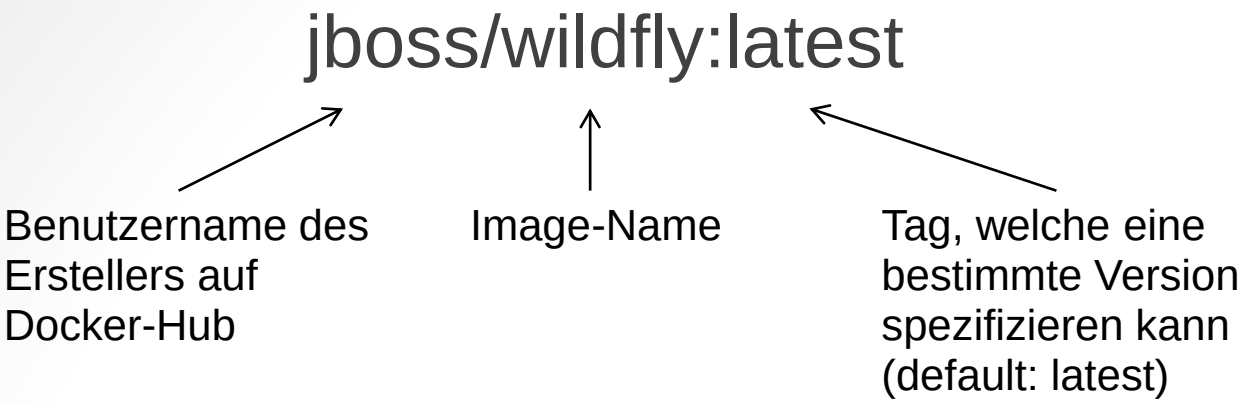


Image-Name




```
> docker pull jboss/wildfly:latest
latest: Pulling from jboss/wildfly
…

> docker pull jboss/wildfly
Using default tag: latest
…
Status: Image is up to date for jboss/wildfly:latest
```

Image-Befehle

`docker images`	Listet auf dem Docker-Host installierte Images auf
`docker pull NAME`	Lädt das spezifizierte Image vom Docker-Hub. Wird mit `docker run` versucht ein nicht verfügbares Imagse zu starten, wird es automatisch geladen.
`docker image rm NAME`	Löscht ein **Image** vom Docker-Host

Image aktualisieren

- Soll nachträglich ein Image aktualisiert werden, kann dies wie folgt durchgeführt werden:
  - Starten eines Containers des entsprechenden Images, wobei die ContainerID zu merken ist
  - Änderungen innerhalb des Containers durchführen
  - Container verlassen
  - ```
    docker commit -m "ÄNDERUNGSBESCHREIBUNG"
                  -a "USER" CONTAINERID
                  NAME
    ```
  - Taggen eines Images (ansonsten immer latest):
    ```
    docker tag IMAGEID USER/NAME:TAG
    ```
    ```

Was ist ein Dockerfile?

- Textfile, das den **Aufbau eines Docker-Images** beschreibt
- Ersetzt das manuelle Absetzen von diversen Kommandos, um ein Image zu erstellen
- Wird von `docker build` verwendet, um ein neues Image zu generieren
- Beispiel-Konfiguration eines Images, die in einem Dockerfile festgelegt werden könnten:

  - Verwende Basisimage ALPINE
  - Installiere Apache2
  - Ändere Config-File
  - Speichere neue Version des Images

- Nach jeder Zeile wird ein Zwischenimage erzeugt!

Aufbau / Befehle

- **FROM <Basis>**

  Der erste Befehl im Dockerfile gibt das Basis-Image für unser neu zu erstellendes Image an.
  Dies können Betriebssystem-Basisimages wie ubuntu, alpine, etc. sein oder auch eigene Images bzw. Images von anderen Benutzern.

  ```
  FROM ubuntu:bionic
  ```

- **LABEL <Key>=<Value>**

  Fügt Meta-Daten zu einem Image hinzu. ZB verwendet um den Ersteller zu dokumentieren (früher MAINTAINER)

  ```
  LABEL maintainer="..."
  ```

- **ENV <Key>=<Value> ...**

  Setzt Environment-Variablen

Aufbau / Befehle

- **RUN <Befehl>**

  Führt den angegebenen Befehl in der Shell aus, was somit mehr oder weniger alle Kommandos ermöglicht. Auch hier gilt wieder, dass nach jeder Zeile ein Zwischenimage erzeugt wird → Zusammenfassen sinnvoll!

  ```
  RUN sudo apt-get install -y apache2 npm
  ```

- **CMD <Befehl>**

  Start-Befehl für alle von dem Image erzeugten Container. Pro Image nur eines möglich (bei Mehrfachnennung wird nur das letzte ausgeführt), kann vom Container überschrieben werden.

  ```
  CMD echo "This is a test."
  ```

Aufbau / Befehle

- **`EXPOSE <Port> [<Port>/<Protocol>...]`**
  Teilt Docker mit, dass der Container zur Laufzeit an diesem Port/Protokoll horcht. Dient eher zu Dokumentationszwecken, um den Port nach aussen freizugeben muss die Option `-p` bei `docker run` angegeben werden.
  ```
  EXPOSE 80
  ```
- **`COPY [--chown=<user>:group>] <src>.. <dest>`**
  `COPY [--chown=..] ["<src>",.. "<dest>"]`
  Kopiert Datei/Verzeichnis vom Kontextordner in das Image. Ohne chown ist user=group=0.
  2. Variante ist für Dateinamen mit Leerzeichen.
  Alternative wäre **`ADD`**, erlaubt URLs als Quelle.
  ```
  COPY hom* /mydir/
  ```

Aufbau / Befehle

- **`WORKDIR <Directory>`**

  Setzt das Referenzverzeichnis für nachfolgende Befehle.

  `WORKDIR /var/www`

- **`VOLUME ["/data"]`**

  `VOLUME <Dir1> <Dir2> ...`

  Erstellt einen Mount-Point für den/die angegebenen Ordner. Beim Starten des Containers kann ein lokaler Ordner an den Mount-Point gebunden werden und somit der Container auf Daten des Docker-Hosts zugreifen.

  `VOLUME /var/www/htdocs`

- Link zur Doku: https://docs.docker.com/engine/reference/builder/

Dockerfile: HelloWorld

1. Dockerfile
   - basierend auf dem Image Alpine
   - Start-Befehl: echo „HelloWorld“

```
FROM alpine
CMD echo "HelloWorld"
```

2. Image Bauen
   - -t <image_name>
   - . (Dockerfile im aktuellen Verzeichnis)

```
docker build -t my-hello-world .
```

3. Container Starten

```
docker run my-hello-world
```

Beispiel – Webserver

- **FROM ubuntu:latest**
- # Aktualisieren der Pakete
  RUN apt-get update && apt-get -yy upgrade
- # Installation von Apache2
  RUN apt-get install -yy apache2
- # Kopieren von Webseite
  COPY webroot/index.html /var/www/html/
- # Port 80 öffnen
  EXPOSE 80/TCP
- # Apache Starten und Logs ausgeben
  CMD service apache2 start && tail -f /var/log/apache2/access.log

Node-Express-Server: Hello World

- app.js

```
var express = require('express');
var app = express();

app.get('/', function (req, res) {
  res.send('Hello World!');
});

app.listen(3000, function () {
  console.log('Example app listening on port 3000!');
});
```

- package.json

```
{
  "name": "docker_web_app",
  "version": "1.0.0",
  "description": "Node.js on Docker",
  "author": "First Last <first.last@example.com>",
  "main": "app.js",
  "scripts": {
    "start": "node app.js"
  },
  "dependencies": {
    "express": "^4.16.1"
  }
}
```

Dockerfile: NodeJS im Container

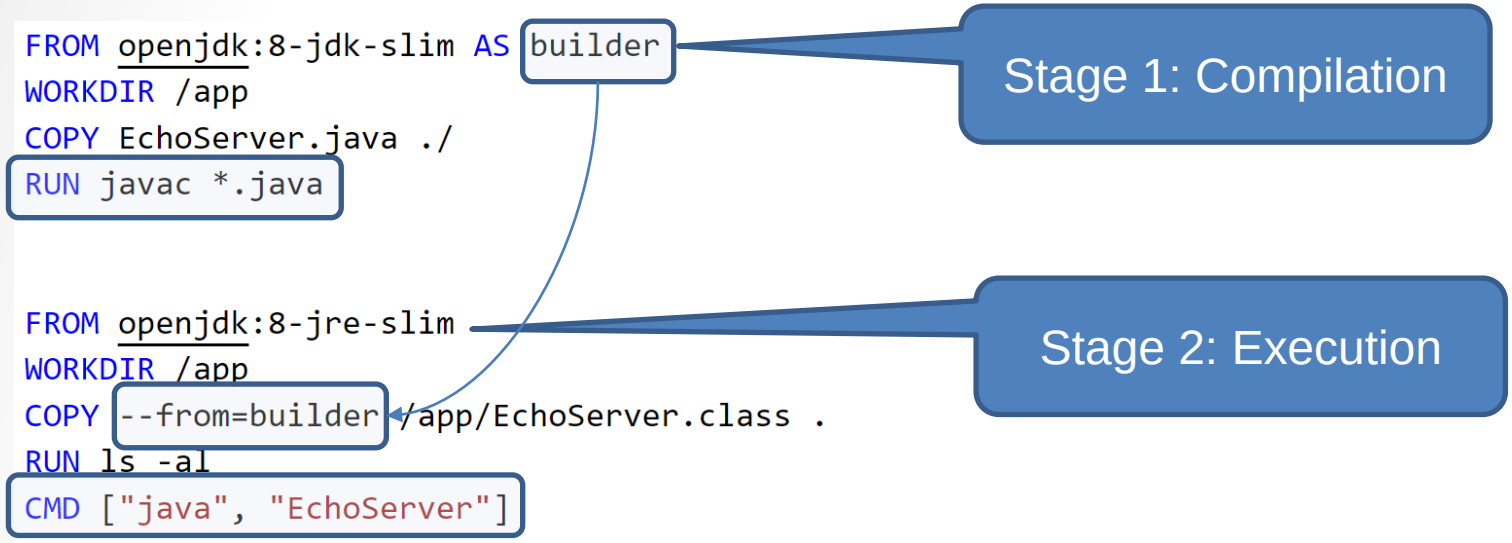
FROM	Basisimage von Docker-Hub	`FROM node`
EXPOSE	Port, unter dem der Container erreichbar sein soll	`EXPOSE 3000`
WORDKIR	Setze Arbeitsverzeichnis im Container	`WORKDIR /app`
COPY	Kopiere Dateien aus dem aktuellen Host-Verzeichnis in das Arbeitsverzeichnis des Containers	`COPY . .`
RUN	Beliebige Shell-Kommandos	`npm install`
CMD	Start-Kommando für den Container	`CMD [„node“, „app.js“]`

1. Image Bauen
```
docker build -t my-node-app .
```
2. Container Starten
```
docker run -p 8080:3000 my-node-app
```
3. Aufruf im Browser

http://<docker-machine-ip>:8080

Multi-Stage Dockerfiles

- Trennung Kompilieren und Ausführen
  - Zum Bauen ist gesamter Source Code und Compiler-Umgebung nötig (z.B. JDK)
  - Zum Ausführen jedoch nur das Kompilat und die Runtime (z.B. JRE)



HTL LEONDING

Asp.Net-Server MultiStage-Dockerfile

```
FROM mcr.microsoft.com/dotnet/core/sdk:3.0 AS builder

WORKDIR /app

# Copy csproj and restore as distinct layers
COPY *.sln .
COPY MyAspNetServer.Core/*.csproj ./MyAspNetServer.Core/
COPY MyAspNetServer.Persistence/*.csproj ./MyAspNetServer.Persistence/
COPY MyAspNetServer.ImportConsole/*.csproj ./MyAspNetServer.ImportConsole/
COPY MyAspNetServer.Web/*.csproj ./MyAspNetServer.Web/
COPY Utils/*.csproj ./Utils/
RUN dotnet restore

COPY . ./
WORKDIR /app/MyAspNetServer.Web
RUN dotnet publish -c Release -o out
```

Stage 1: Compilation (nuget Packages, build & publish)

```
# Build runtime image
FROM mcr.microsoft.com/dotnet/core/aspnet:3.0
WORKDIR /app
COPY --from=builder /app/MyAspNetServer.Web/out .
COPY --from=builder /app/*.csv ./
ENTRYPOINT ["dotnet", "MyAspNetServer.Web.dll"]
```

Stage 2: Execution (copy from output directory of builder stage and run)

Docker Compose

- YAML-Files
  - Default: `docker-compose.yml`
  - Definition von Abhängigkeiten zwischen mehreren Containern
  - Definition von gemeinsamen Ressourcen für mehrere Container (volumes, network)
  - Definition von Ausführungsparametern für Container (anstelle von command-line-Argumenten)
- CLI-Tool docker-compose
  - One-Liner zum Starten und Stoppen von mehreren Containern gleichzeitig

```
docker-compose up
docker-compose down
```

docker-compose.yml

- **`image: <Name>`**

  Definiert, auf welchem Image der jeweilige Container basieren soll. Ist dieses noch nicht lokalen Image Cache vorhanden wird es heruntergeladen (pull)

  ```
  image: httpd
  ```

- **`build: <Pfad>`**

  Gegenstück zu image. Es wird kein fertiges Image geladen, sondern ein relativer Pfad zu einem Ordner mit Dockerfile angegeben und das Image dann erstellt

  ```
  build: mywebserver/
  ```

docker-compose.yml

- **`ports:`**
  Öffnet die angegebenen Ports in dem Container und verbindet sie mit Ports auf dem Host.
  Schreibweisen:
  - einzelne Zahl: Am Host wird zufälliger Port vergeben
  - Doppelpunkt: Mapping (Host:Container)

```
ports:
  - "80:80"
  - "443:443"
```

- **`volumes:`**
  Bind Mounts oder Named Volumes
  Mit Doppelpunkt Mapping (Host:Container)

```
volumes:
  - /tmp/webroot:/var/www/htdocs
```

docker-compose.yml

- **`volumes_from:`**
  Übernimmt alle definierten Volumes von einem anderen Container

```
volumes_from:
  - mywebserver
```

Docker-Compose: MySql + PHPMyAdmin

```yaml
version : '3'

services:
  mysql:
    image: mysql:5.6
    container_name: dev_mysql
    environment:
      MYSQL_USER: user
      MYSQL_PASSWORD: user
      MYSQL_ROOT_PASSWORD: root
      MYSQL_DATABASE: default_schema
    ports:
      - 3306:3306
  phpmyadmin:
    image: phpmyadmin/phpmyadmin
    container_name: dev_pma
    links:
      - mysql
    environment:
      PMA_HOST: mysql
      PMA_PORT: 3306
      PMA_ARBITRARY: 1
    restart: always
    ports:
      - 8183:80
    volumes:
      - my-datavolume:/var/lib/mysql
volumes:
  my-datavolume:
```


```
docker volume ls
docker volume inspect phpmyadmin_my-datavolume
```


HTL LENDING

Docker-Compose: Asp.Net + MSSQL

```yaml
version: '3.4'

services:
  dbserver:
    image: microsoft/mssql-server-linux:2017-latest
    hostname: 'dbserver'
    environment:
      ACCEPT_EULA: Y
      SA_PASSWORD: "Admin4Data"
    volumes:
      - data:/var/opt/mssql3
    ports:
      - '1433:1433'
    expose:
      - 1433

  asp-api-server:
    restart: on-failure
    depends_on:
      - dbserver
    build:
      context: ./MyAspNetServer
    ports:
      - 9999:80
volumes:
  data:
```

Docker-Compose: Network

Docker-Compose erstellt automatisch eigenes Subnetz für alle Services im yml-File

```
docker network ls

docker network inspect 77
```

```
"IPAM": {
    "Driver": "default",
    "Options": null,
    "Config": [
        {
            "Subnet": "172.19.0.0/16",
            "Gateway": "172.19.0.1"
        }
    ]
},
"Containers": {
    "5541e678e19fb6b90006870882e4dc0e189e81e9068a463262a728149b312b1e": {
        "Name": "dev_mysql",
        "EndpointID": "0a47dae04729b96daa32af71b622ff9fa290191ee07690d054d32f30f6a0090f",
        "MacAddress": "02:42:ac:13:00:02",
        "IPv4Address": "172.19.0.2/16",
        "IPv6Address": ""
    },
    "f276a10747badf4e929fa2eb95dc86cde3c6d9b4b5e30b4926f90290ca73bbd2": {
        "Name": "dev_pma",
        "EndpointID": "68bc0ab47a22ebe7fbdeb984a3132ffd8a2cedcd05130bd1d35357c96dc6008a",
        "MacAddress": "02:42:ac:13:00:03",
        "IPv4Address": "172.19.0.3/16",
        "IPv6Address": ""
    }
```

Docker-Compose: Subnetz Definieren

1. Networks definieren
2. Network zuweisen

```yaml
version : '3.5'

services:
  mysql: ···
  phpmyadmin: ···
volumes: ···
networks:
  backend:
    name: backend
    driver: bridge
    ipam:
      config:
        - subnet: 10.2.0.0/16
  frontend:
    name: frontend
    driver: bridge
    ipam:
      config:
        - subnet: 10.3.0.0/16
```

frontend

```yaml
ports:
  - 8183:80
```

```yaml
phpmyadmin:
  image: phpmyadmin/phpmyadmin
  container_name: dev_pma
  links: ···
  environment: ···
  restart: always
  ports: ···
  networks:
    - backend
    - frontend
  volumes: ···
```

backend

```yaml
mysql:
  image: mysql:5.6
  container_name: dev_mysql
  environment: ···
  ports: ···
  networks:
    - backend
```


HTL LEONDING

Docker-Compose: MySql + PHPMyAdmin

- docker-compose.yml

```yaml
version: '2.0'

services:
  mysql:
    build: docker-mysql
    ports:
      - 3308:3306
    volumes:
      - mysql_data:/var/lib/mysql
    environment:
      - MYSQL_ROOT_PASSWORD=root
  phpmyadmin:
    image: phpmyadmin/phpmyadmin
    ports:
      - 5050:80
    depends_on:
      - mysql
    environment:
      - PMA_HOST=mysql
      - MYSQL_ROOT_PASSWORD=root
volumes:
  mysql_data:
```

- Dockerfile für mysql:

```
FROM mysql:5

COPY setup-db.sql /docker-entrypoint-initdb.d/
```

- Setup-db.sql:

```
create database mydb;

use mydb;
DELIMITER $$
CREATE FUNCTION to_date(s varchar(20), fmt varchar(20)) RETURNS DATE
BEGIN
    DECLARE dt DATE;
    select str_to_date(s, '%d-%m-%Y') into dt;
    RETURN dt;
END;
$$
DELIMITER ;

-- insert historical dept/empt data :)

CREATE TABLE `DEPT` (
  `DEPTNO` SMALLINT NOT NULL AUTO_INCREMENT,
  `DNAME` VARCHAR(14) NULL DEFAULT NULL,
  `LOC` VARCHAR(13) NULL DEFAULT NULL,
  PRIMARY KEY (`DEPTNO`))
ENGINE = InnoDB
DEFAULT CHARACTER SET = latin1;


CREATE TABLE `EMP` (
```

Docker-Compose: Drupal CMS

```yaml
version: "2"

services:
  drupal:
    image: drupal
    ports:
      - "8080:80"
    volumes:
      - drupal-modules:/var/www/html/modules
      - drupal-profiles:/var/www/html/profiles
      - drupal-sites:/var/www/html/sites
      - drupal-themes:/var/www/html/themes
  postgres:
    image: postgres
    environment:
      - POSTGRES_PASSWORD=my_passwd

volumes:
    drupal-modules:
    drupal-profiles:
    drupal-sites:
    drupal-themes:
```

Docker-Compose: Wordpress + MariaDB

```yaml
version: '2'

services:

  wordpress:
    image: wordpress
    ports:
      - 8080:80
    environment:
      WORDPRESS_DB_HOST: mysql
      WORDPRESS_DB_NAME: wordpress
      WORDPRESS_DB_USER: example
      WORDPRESS_DB_PASSWORD: examplePW
    volumes:
      - ./wordpress-data:/var/www/html

  mysql:
    image: mariadb
    environment:
      MYSQL_ROOT_PASSWORD: examplerootPW
      MYSQL_DATABASE: wordpress
      MYSQL_USER: example
      MYSQL_PASSWORD: examplePW
    volumes:
      - mysql-data:/var/lib/mysql

volumes:
  mysql-data:
```

Nginx als Reverse Proxy für httpd

- docker-compose.yml:

```yaml
version: '3'

services:
  proxy:
    image: nginx:1.13 # this will use the latest version of 1.13.x
    ports:
      - '80:80' # expose 80 on host and sent to 80 in container
    volumes:
      - ./nginx.conf:/etc/nginx/conf.d/default.conf:ro
  web:
    image: httpd  # this will use httpd:latest
```

- Nginx.conf (Proxy für Apache2 httpd)

```
server {

    listen 80;

    location / {

        proxy_pass          http://web;
        proxy_redirect      off;
        proxy_set_header    Host $host;
        proxy_set_header    X-Real-IP $remote_addr;
        proxy_set_header    X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header    X-Forwarded-Host $server_name;

    }
}
```

Variante: Build Nginx mit Dockerfile

- Docker-compose.yml: Build Image with Dockerfile

```
services:
  proxy:
    build:
      context: .
      dockerfile: nginx.Dockerfile
    ports:
      - '80:80'
  web:
    image: httpd
    volumes:
      - ./html:/usr/local/apache2/htdocs/
```

- Dockerfile:

```
FROM nginx:1.13

COPY nginx.conf /etc/nginx/conf.d/default.conf
```

Docker-Config auf HTL Linux VMs

- Vermeiden, dass die Standard-Bridge IP-Adressen belegt, die mit den schulinternen Subnetzen kollidieren:
  - Docker-Daemon-Konfiguration editieren:

```
sudo nano /etc/docker/daemon.json
```

  - IP-Range für default-Bridge ändern:

```
{
        "bip": "10.0.1.1/24"
}
```

  - Docker-Service neu starten:

```
sudo systemctl restart docker
```