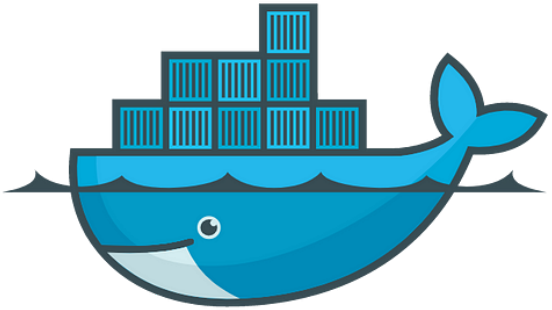


How to docker?

Docker something

Julia Winkler

19.06.2024

Disclaimer

- bei weitem nicht alles zum Thema Docker
- nur Allgemeine Grundlage

commands im Terminal ausführen
Ordner `examples`, wenn nichts da steht

CodeTour (VSCode Plugin)
Code im Repo

Gliederung

Wiso, Weshalb, Warum?

Why use Docker?

Trusted by developers.
Chosen by Fortune 100 companies.

Docker provides a suite of development tools, services, trusted content, and automations, used individually or together, to accelerate the delivery of secure applications.

Wiso, Weshalb, Warum?

Why use Docker?

Trusted by developers.
Chosen by Fortune 100 companies.

Docker provides a suite of development tools, services, trusted content, and automations, used individually or together, to accelerate the delivery of secure applications.

"a sandboxed process on your machine that is isolated from all other processes on the host machine"

Wiso, Weshalb, Warum?

Why use Docker?

Trusted by developers.
Chosen by Fortune 100 companies.

Docker provides a suite of development tools, services, trusted content, and automations, used individually or together, to accelerate the delivery of secure applications.

"a sandboxed process on your machine that is isolated from all other processes on the host machine"

"It works on my computer"

Wiso, Weshalb, Warum?

Why use Docker?

Trusted by developers.
Chosen by Fortune 100 companies.

Docker provides a suite of development tools, services, trusted content, and automations, used individually or together, to accelerate the delivery of secure applications.

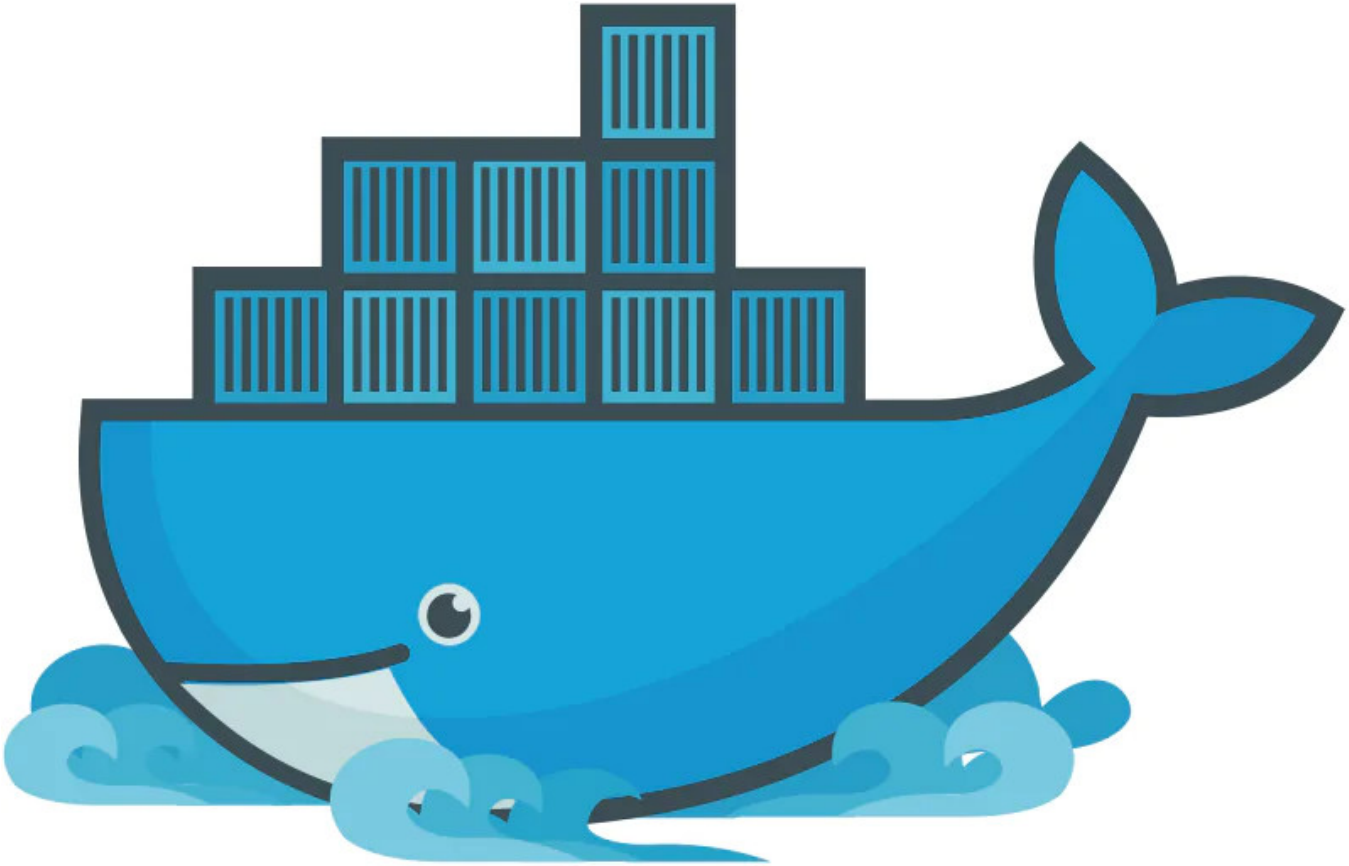
"a sandboxed process on your machine that is isolated from all other processes on the host machine"

"It works on my computer"

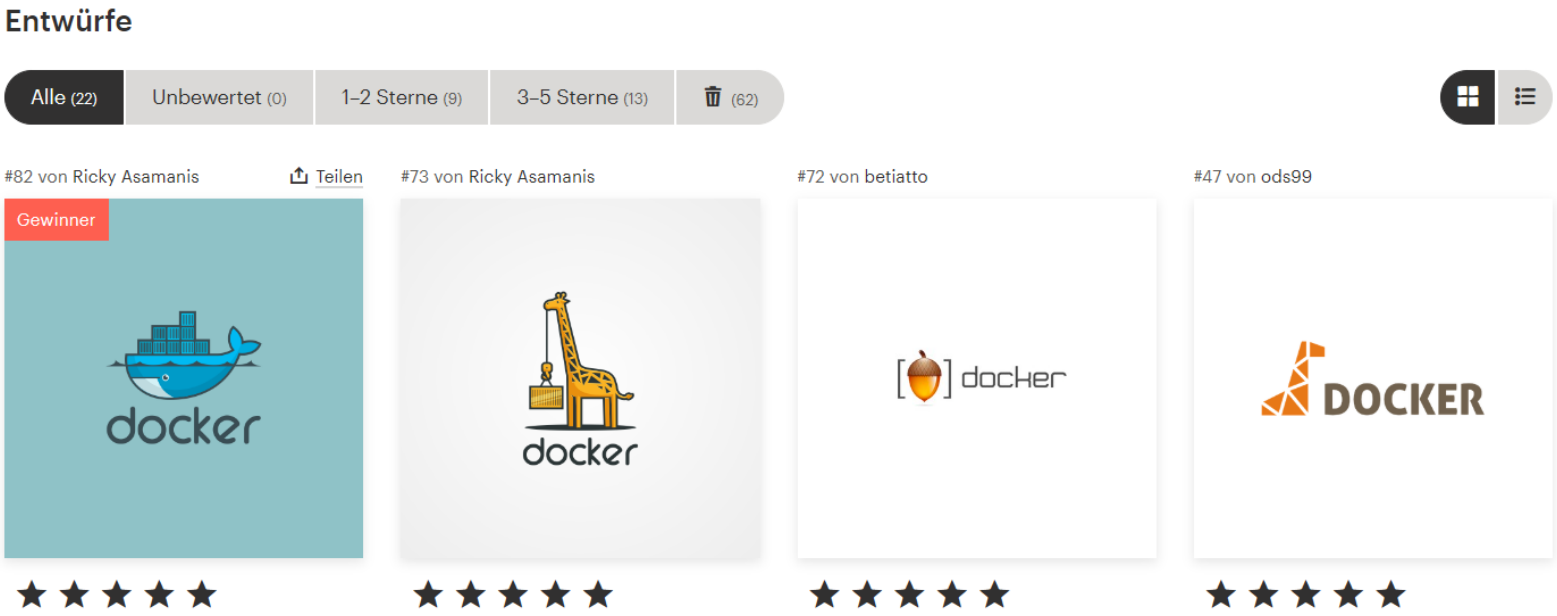
"faster onboarding and testing while also simplifying the deployment of services"

Wer ist Moby Dock?

Wer ist Moby Dock?



Wer ist Moby Dock?

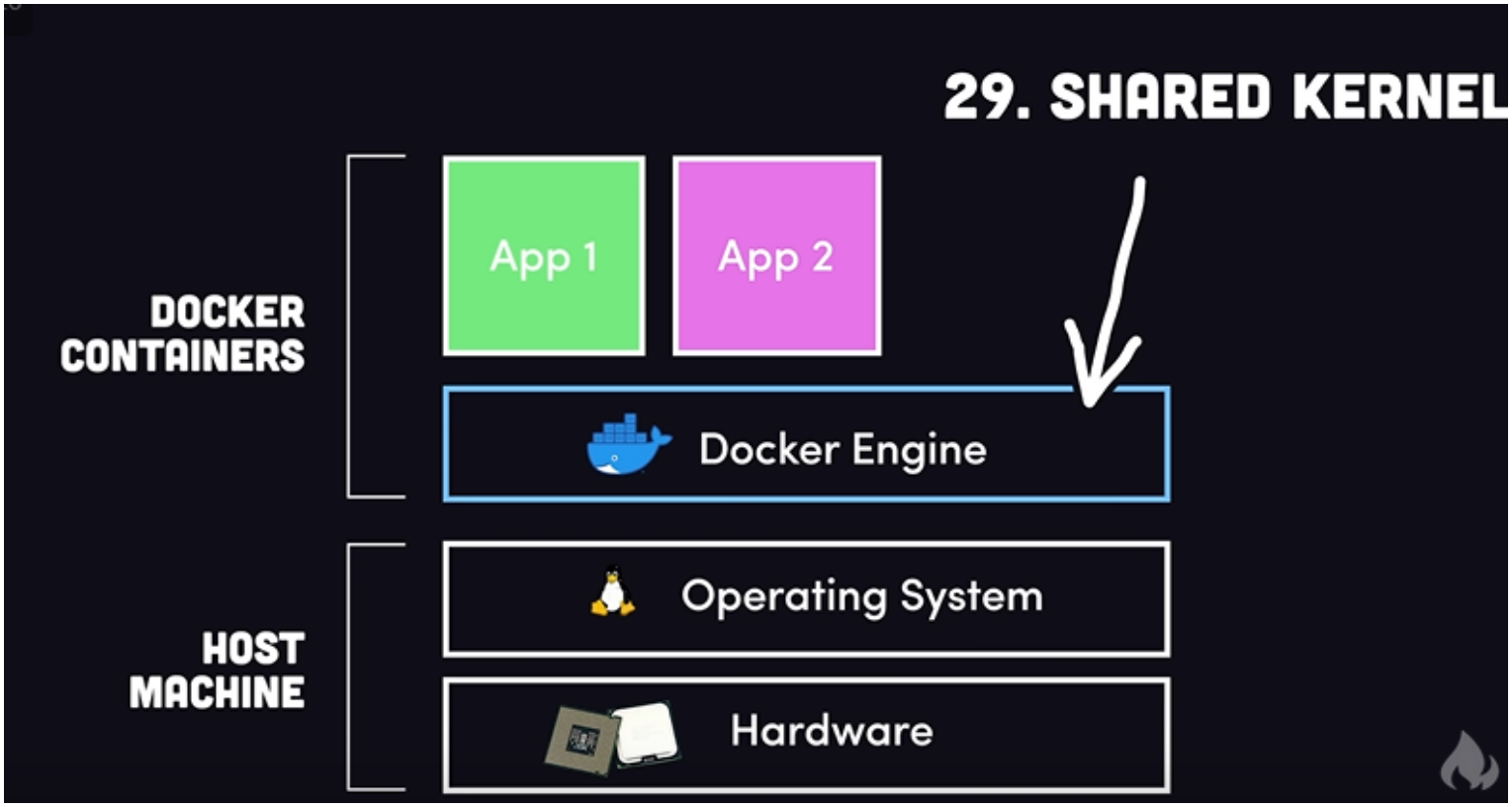


Wettbewerb zum Icon für Docker

Was ist Docker?

Docker

freie Software zur Isolierung von Anwendungen
Containervirtualisierung
"light weight" Virtual Maschine



Wichtige Begriffe

Container

Umgebung in der die tatsächliche Anwendung läuft

Image

Blaupausen, um einen Container zu erstellen

Dockerfile

Anleitung, um ein Image zu erstellen

Wichtige Begriffe

Container

Umgebung in der die tatsächliche Anwendung läuft

Image

Blaupausen, um einen Container zu erstellen

Dockerfile

Anleitung, um ein Image zu erstellen

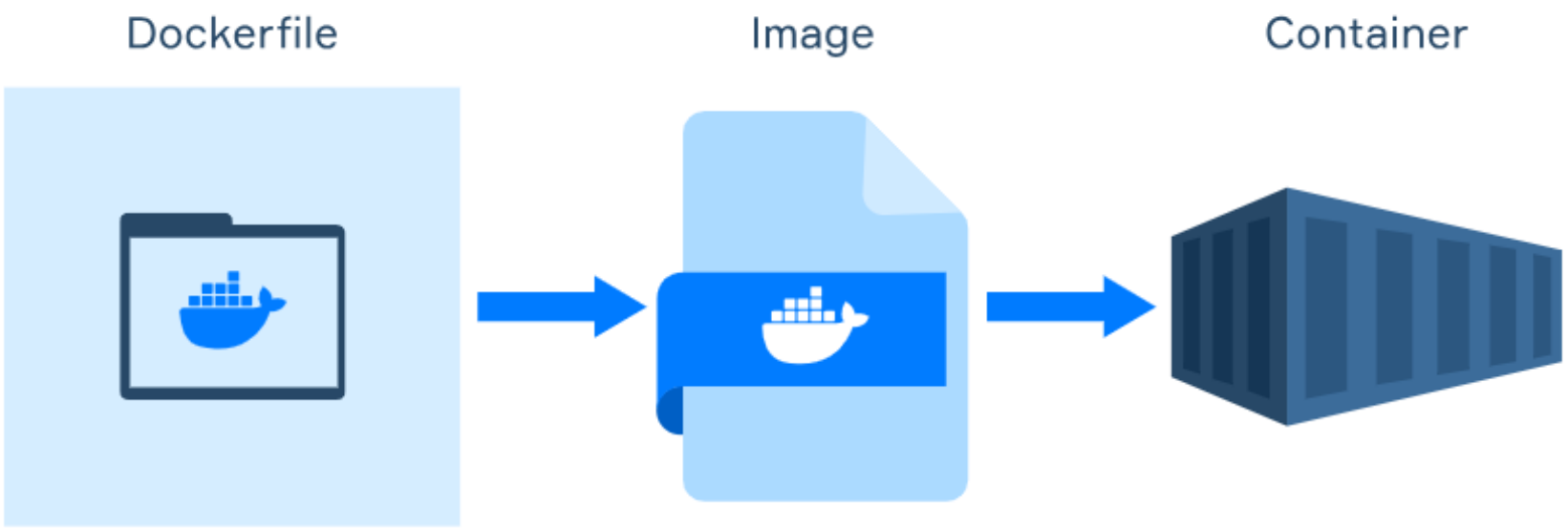
Registry

z.B. Docker Hub, EAC.... Ort an dem viele verschindene Images gespeichert und geteilt werden können

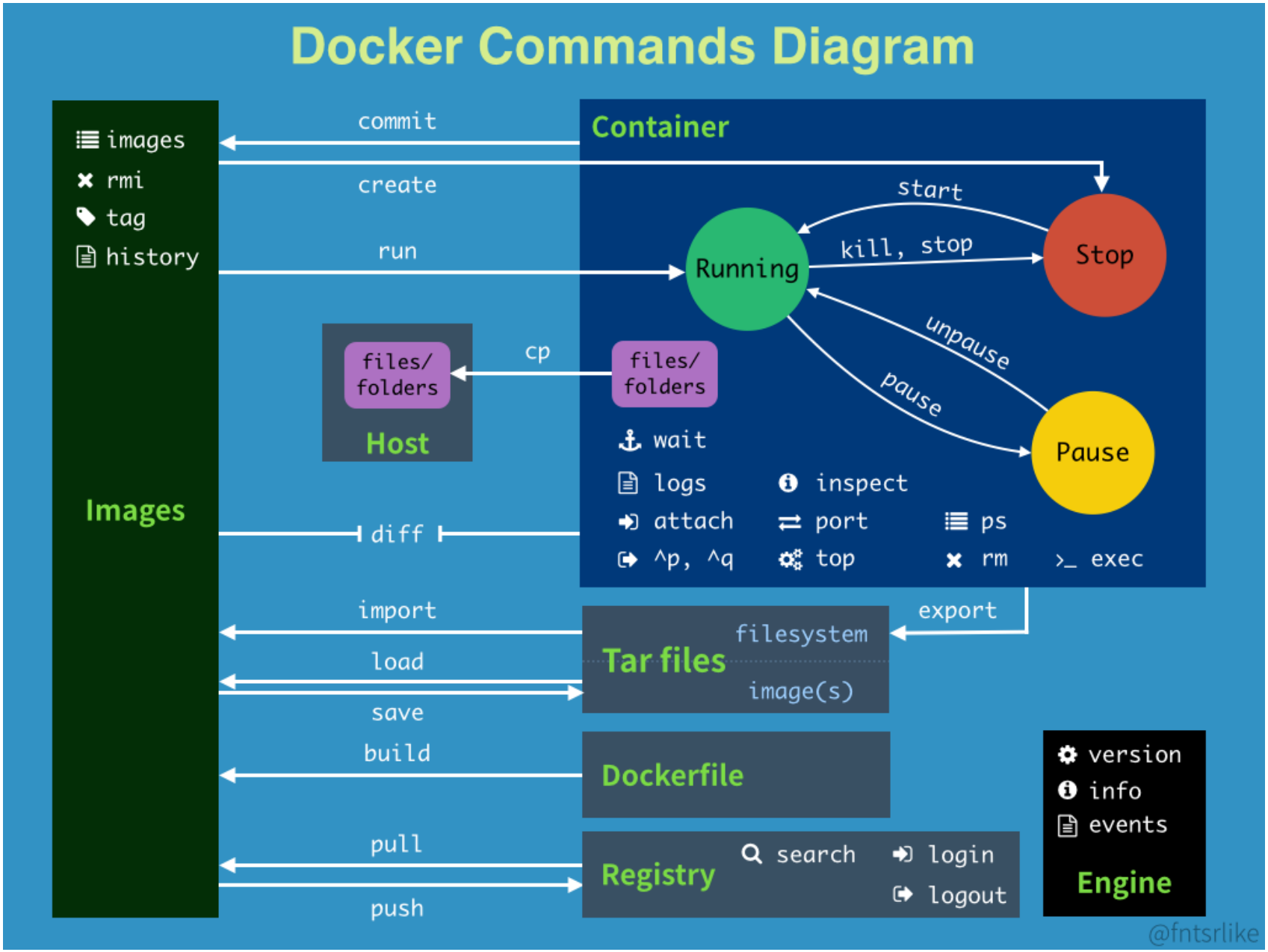
Docker Compose

Orchestrierungstool für Dockerfile
Wrapper für einen oder mehrere Container

Zusammenhang der Docker Komponenten



Zusammenhang der Docker Komponenten



Wie kreige ich dieses "Docker"?

docker.docs Guides Manuals Reference

- Overview
- Get Docker
- Docker Desktop
- Docker Extensions
- Docker Scout
- Docker Engine
  - Overview
  - Install
    - Overview
    - CentOS
    - Debian
    - Fedora
    - RHEL (s390x)
    - SLES
    - Ubuntu
    - Raspberry Pi OS (32-bit)
    - Binaries
    - Post-installation steps
    - Troubleshoot installation
  - Storage
  - Networking

Manuals / Docker Engine / Install / Overview

Install Docker Engine

This section describes how to install Docker Engine on Linux, also known as Docker CE. Docker Engine is also available for Windows, macOS, and Linux, through Docker Desktop. For instructions on how to install Docker Desktop, see:

- Docker Desktop for Linux
- Docker Desktop for Mac (macOS)
- Docker Desktop for Windows

Supported platforms

Platform	x86_64 / amd64	arm64 / aarch64	arm (32-bit)	ppc64le	s390x
CentOS	✓	✓		✓	
Debian	✓	✓	✓	✓	
Fedora	✓	✓		✓	
Raspberry Pi OS (32-bit)			✓		
RHEL (s390x)					✓
SLES					✓
Ubuntu	✓	✓	✓	✓	✓
Binaries	✓	✓	✓		

Edit this page

Request changes

Table of contents

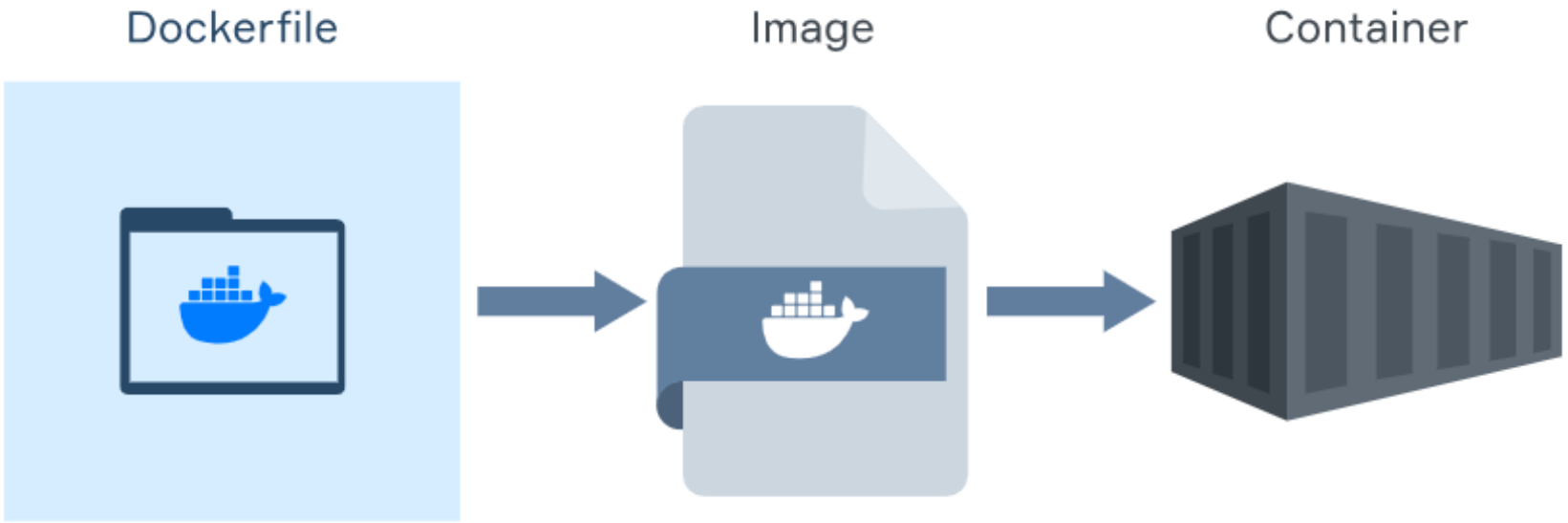
- Supported platforms
  - Other Linux distros
- Release channels
- Support
  - Upgrade path
  - Licensing
- Reporting security issues
- Get started

Doku

Hello World

```
> docker -v
> docker --help
> docker run hello-world
```

Zusammenhang der Docker Komponenten



Dockerfile

- Anleitung um ein Image zu erstellen
- hießt standardmäßig 'Dockerfile'

ein beispielhaftes `Dockerfile`:

```
FROM alpine:latest

CMD [ "echo", "Hello World" ]
```

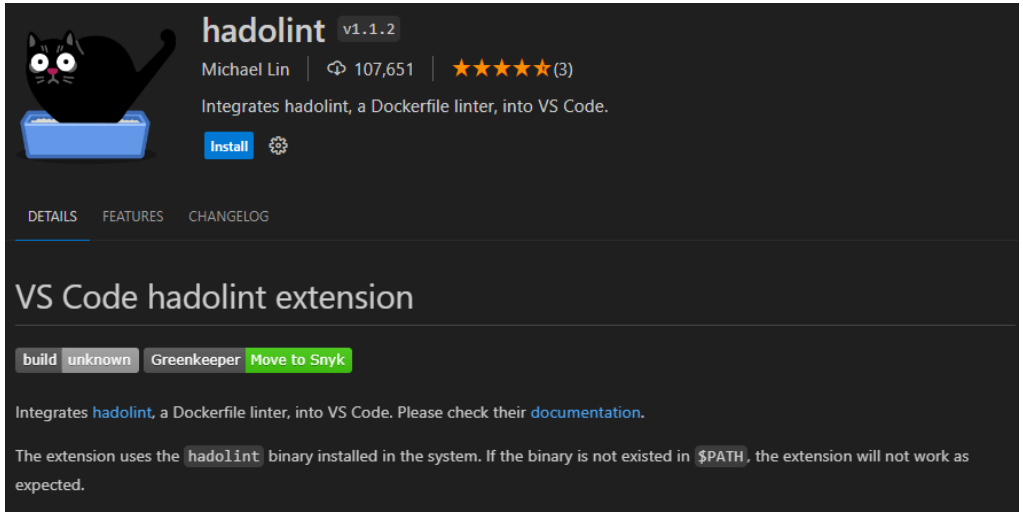
Dockerfile

- Anleitung um ein Image zu erstellen
- hießt standardmäßig 'Dockerfile'

ein beispielhaftes `Dockerfile`:

```
FROM alpine:latest

CMD [ "echo", "Hello World" ]
```



Plugin für die Arbeit mit Docker

Dockerfile

- Anleitung um ein Image zu erstellen
- hießt standardmäßig 'Dockerfile'

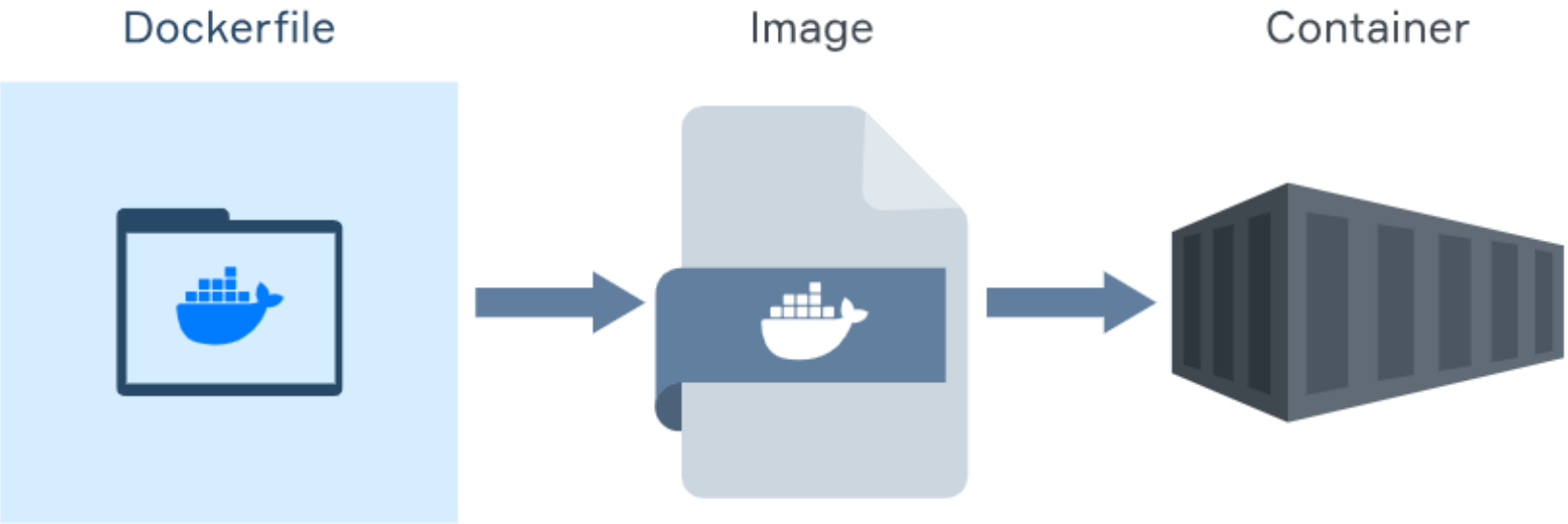
ein beispielhaftes `Dockerfile`:

```
FROM alpine:latest

CMD [ "echo", "Hello World" ]
```

Weitere Informationen und Instruction
https://docs.docker.com/reference/dockerfile/

Zusammenhang der Docker Komponenten



docker build command

`docker build [OPTIONS] PATH | URL | -`

Build an image from a Dockerfile

`[OPTIONS]`

-t, –tag stringArray Name and optionally a tag (format: "name:tag")

-f, –file string Name of the Dockerfile (default: "PATH/Dockerfile")

...

`PATH` Pfad zum Build Kontext (Ordner), meistens `.`

```
docker build . # 'Dockerfile' im aktuellen Ordner
docker build -t myimage:v1 .
docker build -f Docker.cmd .
docker build FastAPI
```

weitere Optionen mit `docker buildx build`

RUN

```dockerfile
FROM ubuntu:22.04

LABEL author=HyperUser

RUN apt-get update -y \
  && apt-get upgrade -y \
  && apt-get install iputils-ping -y \
  && apt-get install net-tools -y

ENTRYPOINT ["/bin/bash"]
```

```dockerfile
FROM ubuntu:22.04

LABEL author=HyperUser

RUN apt-get update -y
RUN apt-get upgrade -y
RUN apt-get install iputils-ping -y
RUN apt-get install net-tools -y

ENTRYPOINT ["/bin/bash"]
```

```
> docker build -t example:single -f Dockerfile.single .
> docker build -t example:multi -f Dockerfile.multi .
# Vergleicht die Build-time

# Vergleicht die Größe - Wie?
> docker images # Entstandene Images anschauen
```

RUN

```
FROM ubuntu:22.04

LABEL author=HyperUser

RUN apt-get update -y \
  && apt-get upgrade -y \
  && apt-get install iputils-ping -y \
  && apt-get install net-tools -y

ENTRYPOINT ["/bin/bash"]
```

```
FROM ubuntu:22.04

LABEL author=HyperUser

RUN apt-get update -y
RUN apt-get upgrade -y
RUN apt-get install iputils-ping -y
RUN apt-get install net-tools -y

ENTRYPOINT ["/bin/bash"]
```

- pro `RUN` baut Docker einen Layer

RUN

```
FROM ubuntu:22.04

LABEL author=HyperUser

RUN apt-get update -y \
  && apt-get upgrade -y \
  && apt-get install iputils-ping -y \
  && apt-get install net-tools -y

ENTRYPOINT ["/bin/bash"]
```

```
FROM ubuntu:22.04

LABEL author=HyperUser

RUN apt-get update -y
RUN apt-get upgrade -y
RUN apt-get install iputils-ping -y
RUN apt-get install net-tools -y

ENTRYPOINT ["/bin/bash"]
```

- pro `RUN` baut Docker einen Layer
- mehr Layer vergrößern das Image
- Layer werden gecached und nach Möglichkeit wiederverwendet

RUN

```
FROM ubuntu:22.04

LABEL author=HyperUser

RUN apt-get update -y \
  && apt-get upgrade -y \
  && apt-get install iputils-ping -y \
  && apt-get install net-tools -y

ENTRYPOINT ["/bin/bash"]
```

```
FROM ubuntu:22.04

LABEL author=HyperUser

RUN apt-get update -y
RUN apt-get upgrade -y
RUN apt-get install iputils-ping -y
RUN apt-get install net-tools -y

ENTRYPOINT ["/bin/bash"]
```

- pro `RUN` baut Docker einen Layer
- mehr Layer vergrößern das Image
- Layer werden gecached und nach Möglichkeit wiederverwendet
- verbinden von `RUN` instructions verbessert built time und Image Größe

CMD vs. ENTRYPOINT

```
FROM alpine


# Exec form
CMD ["echo", "Hello World."]

#shell form
CMD echo Hello Students
```

```
FROM alpine

# ENTRYPOINT ["echo"]
# CMD ["Hello","Students."]

ENTRYPOINT ["echo", "Hello World"]
```

```
> docker build -t example:cmd -f Dockerfile.cmd .
> docker build -t example:entry -f Dockerfile.entry .
```

CMD vs. ENTRYPOINT

```
FROM alpine


# Exec form
CMD ["echo", "Hello World."]

#shell form
CMD echo Hello Students
```

```
FROM alpine

# ENTRYPOINT ["echo"]
# CMD ["Hello","Students."]

ENTRYPOINT ["echo", "Hello World"]
```

```
> docker build -t example:cmd -f Dockerfile.cmd .
> docker build -t example:entry -f Dockerfile.entry .

> docker run example:cmd
> docker run example:cmd echo hello
> docker run example:entry hello
```

CMD vs. ENTRYPOINT

```
FROM alpine


# Exec form
CMD ["echo", "Hello World."]

#shell form
CMD echo Hello Students
```

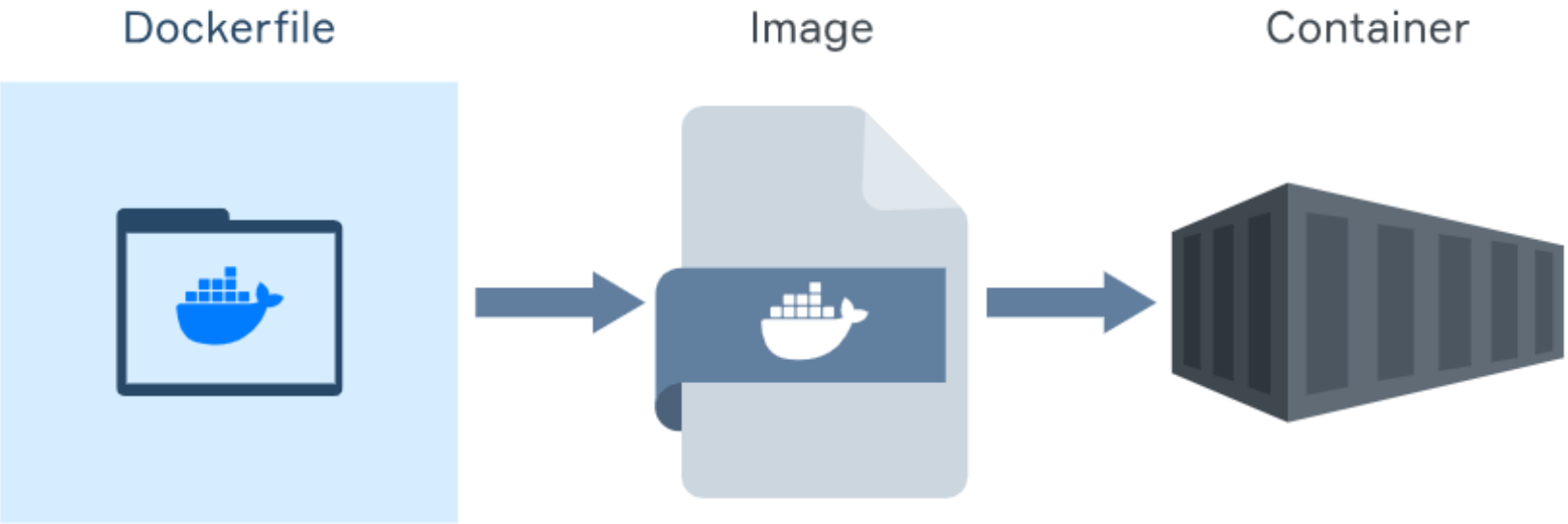
```
FROM alpine

# ENTRYPOINT ["echo"]
# CMD ["Hello","Students."]

ENTRYPOINT ["echo", "Hello World"]
```

- beide definieren den, was nach Container start ausgeführt wird
- `CMD` kann überschrieben werden
- `ENTRYPOINT` bestimmt den command, neue Parameter werden angehangen

Zusammenhang der Docker Komponenten



docker run command

`docker run [OPTIONS] IMAGE [COMMAND] [ARG...]`

Create and run a new container from image

`[OPTIONS]`

-d	Detach from terminal — run in background
-e	Set environment variables
-it	Interactive terminal, enter container terminal
–mount mount	Attach a filesystem mount to the container
-p [host]:[port]	Publish a container's port(s) to the host
-P	Publish all exposed ports
–rm	Auto remove the container when it exits
-v, –volume list	Bind mount a volume
-w	Provides an execution directory inside the container

...

`IMAGE` Referenz zum Image (Tag oder Id/Hash)

Python / FastAPI

FastAPI 0.1.0 OAS 3.1

/openapi.json

root

GET / Read Root

songs

GET /song/get Get Songs

POST /song/add Add Song

DELETE /song/{id} Delete Song

Schemas

HTTPValidationError > Expand all object

Song > Expand all object

ValidationError > Expand all object

Python FastAPI im Conatiner

```
FROM python:3.10.11

WORKDIR /code

COPY ./requirements.txt /code/requirements.txt

RUN pip install --no-cache-dir --upgrade -r /code/requirements.txt

COPY ./app /code/app

CMD ["uvicorn", "app.api:app", "--host", "0.0.0.0", "--port", "80"]
```

Python FastAPI im Conatiner

```
> cd examples/FastAPI
> docker build -t fastapiapp:v1 .
> docker run -d --name backend -p 8000:80 fastapiapp:v1
# Open http://localhost:8000/docs
> docker exec -it backend bash
# 'exit' um den Container zu verlassen
> docker stop backend
> docker start backend
> docker rm backend
> docker run -it --name backend \
    -p 8000:80 fastapiapp:v1 bash
```

docker exec command

`docker exec [OPTIONS] CONTAINER COMMAND [ARG...]`

Command in einem laufenden Container ausführen

`[OPTIONS]`

- **-d** im Hintergund ausführen
- **-e** env Variablen setzen
- **-it** Interaktives Terminal öffnen
- **-w, –workdir string** Aktuelles Verzeichnis im Container ändern
- ...

docker start & stop command

`docker build [OPTIONS] PATH | URL | -`

DESC

`[OPTIONS]`

-bla BLA

Wo ist mein Song?

Problem reproduzieren:

```
> cd examples/FastAPI
> docker build -t fastapiapp:v1 .
> docker run -d --name backend -p 8000:80 --rm fastapiapp:v1
# Öffne http://localhost:8000/docs + add_song() ausführen
> docker stop backend # Container automatisch gelöscht
> docker run -d --name backend -p 8000:80 --rm fastapiapp:v1
# get_songs() ausführen -> Song fehlt :/
```

Wo ist mein Song?

Problem reproduzieren:

```
> cd examples/FastAPI
> docker build -t fastapiapp:v1 .
> docker run -d --name backend -p 8000:80 --rm fastapiapp:v1
# Öffne http://localhost:8000/docs + add_song() ausführen
> docker stop backend # Container automatisch gelöscht
> docker run -d --name backend -p 8000:80 --rm fastapiapp:v1
# get_songs() ausführen -> Song fehlt :/
```

- der Song ist im Container gespeichert, nicht im Image
- `--rm` löscht den Container nach Beendigung

Wie bekomme ich den Song permantent gespeichert?

Python FastAPI im Conatiner

- Option 1: json anpassen, Image neu erstellen

Python FastAPI im Conatiner

- Option 1: json anpassen, Image neu erstellen
- Option 2: Änderungen commiten

```
> docker commit backend fastapiapp:v2
> docker run -d --name backend2 --rm \
    -p 8080:80 fastapiapp:v2
# Öffne localhost:8080/docs -> get_songs() hat neue Songs

> docker run -d --name backend --rm \
    -p 8000:80 fastapiapp:v1
# Öffne localhost:8000/docs -> get_songs() hat keine
```

Python FastAPI im Conatiner

- Option 1: json anpassen, Image neu erstellen
- Option 2: Änderungen commiten
- Option 3: Volumes und Mounts verwenden

```
> docker commit backend fastapiapp:v2
> docker run -d --name backend2 --rm \
    -p 8080:80 fastapiapp:v2
# Öffne localhost:8080/docs -> get_songs() hat neue Songs

> docker run -d --name backend --rm \
    -p 8000:80 fastapiapp:v1
# Öffne localhost:8000/docs -> get_songs() hat keine
```

Volumes und Mounts

- Docker containers are stateless by default, data inside is lost after shutdown.
- both map data/storage from the host machine to data/storage in the Container for persistent storage.

Volumes und Mounts

- Docker containers are stateless by default, data inside is lost after shutdown.
- both map data/storage from the host machine to data/storage in the Container for persistent storage.

Volume

- are managed by Docker and stored default at `var/lib/docker/volumes/VOLUMENAME`
- don't increase the size of the containers
- simplify and allow sharing data between containers

Volumes und Mounts

- Docker containers are stateless by default, data inside is lost after shutdown.
- both map data/storage from the host machine to data/storage in the Container for persistent storage.

Volume

- are managed by Docker and stored default at `var/lib/docker/volumes/VOLUMENAME`
- don't increase the size of the containers
- simplify and allow sharing data between containers

Mount

- a file or directory on the host machine is attached to the containers filesystem
- dependant on the host machine while volumes are managed by docker

Python FastAPI im Conatiner mit Volume

- Option 1: json anpassen, Image neu erstellen
- Option 2: changes commiten
- Option 3: Volume, wenn man die json changes behalten möchte, aber den container per se nicht

```
# Quiz: why is this not working
> docker run -d --rm --name backend \
    -v ${PWD}/app/songs.json:/app/songs.json \
    -p 8000:80 fastapiapp:v1
```

Python FastAPI im Conatiner mit Volume

- Option 1: json anpassen, Image neu erstellen
- Option 2: changes commiten
- Option 3: Volume, wenn man die json changes behalten möchte, aber den container per se nicht

```
# Quiz: why is this not working
> docker run -d --rm --name backend \
    -v ${PWD}/app/songs.json:/app/songs.json \
    -p 8000:80 fastapiapp:v1
# Note: be aware of the working directory of your app
# in this case code 'WORKDIR /code'
> docker run -d --rm --name backend \
    -v ${PWD}/app/songs.json:/code/app/songs.json \
    -p 8000:80 fastapiapp:v1
```

React

My Favorit Songs

Title: A Change Is Gonna Come | **Artist:** Sam Cook | **Year:** 1964
Delete Song

Title: Like a Rolling Stone | **Artist:** Bob Dylan | **Year:** 1965
Delete Song

Title: string | **Artist:** string | **Year:** 2020
Delete Song

Add new Song

React - Dockerfile

```
# pull official base image
FROM node:18.16.0-alpine

# set working directory
WORKDIR /app

# add `/app/node_modules/.bin` to £PATH
ENV PATH /app/node_modules/.bin:$PATH

# install app dependencies
COPY package.json ./
COPY package-lock.json ./
RUN npm install --silent
RUN npm install react-scripts@3.4.1 -g --silent

# add app
COPY . ./

# start app
CMD ["npm", "start"]
```

React im Container

```
> cd examples/React
> docker build -t reactapp:dev .
> docker run -it --rm --name frontenddev \
    -v ${PWD}:/app -v /app/node_modules \
    -e CHOKIDAR_USEPOLLING=true \
    -p 3000:3000  reactapp:dev
# Öffne localhost:3000
```

(Der Container `backend` sollte laufen, damit die Webseite richtig funktioniert)

Multistage builds

Idee: Image aufeinanderaufbauende Teile teilen, zwischen den Teilen nur die nötigen Dinge kopieren
z.B. Stage 1: App compile, Stage 2: Compilierte App ausführen (kein Build context) Vorteile

- Smaller image size
- faster build times
- improved security (only runtime artifacts and dependencies)
- code isolation and reusability
- Easier debugging and troubleshooting

React - Multistage

```dockerfile
# build environment
FROM node:18.16.0-alpine as build
WORKDIR /app
ENV PATH /app/node_modules/.bin:$PATH
COPY package.json ./
COPY package-lock.json ./
RUN npm ci --silent
RUN npm install react-scripts@3.4.1 -g --silent
COPY . ./
RUN npm run build

# production environment
FROM nginx:stable-alpine
COPY --from=build /app/build /usr/share/nginx/html
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

React - Multistage

```
> docker build -f Dockerfile.prod -t reactapp:prod .
> docker run -it --rm --name frontend \
    -p 1337:80 reactapp:prod
```

Vergleiche die Größe der Images:

React - Multistage

```
> docker build -f Dockerfile.prod -t reactapp:prod .
> docker run -it --rm --name frontend \
    -p 1337:80 reactapp:prod
```

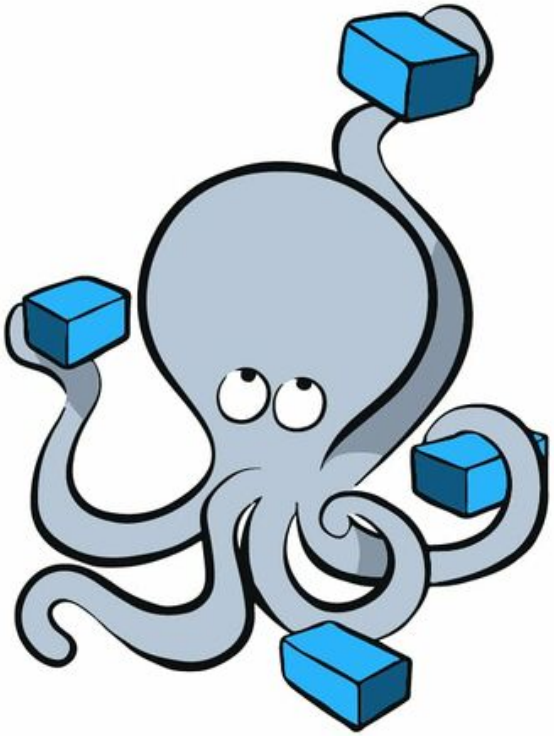
Vergleiche die Größe der Images:
frontenddev: 832 MB
frontend: 50.9 MB

Dockerfile Best Practices

- `RUN` instructions mit && zusammenfassen
- `COPY` sinnvoll platzieren, damit Cache best möglich genutzt werden kann
- `ADD` nur für `ADD` specifische Funktionen
- Volumes und Mounts für persistententen Speicher nutzen
- Multistage builds verwenden

title

```
> docker push
> docker pull
> docker create
> docker ps
..
%% TODO:
```



docker
Compose

Docker Compose

Was?

?

Warum?

-
- Vorteile
- UseCases

Anmerkung: Python in `examples/FastAPI`, React in `examples/React` und Full App in `examples` ausführen

Docker Compose zu Python

```yaml
version: '3.7'

services:
  fastapi:
    container_name: backend
    build:
      context: .
      dockerfile: Dockerfile
    # image: fastapiapp:v2
    ports:
      - '8000:80'
    volumes:
      - ./app/songs.json:/code/app/songs.json
```

```
docker compose up
```

VS.

Docker Compose zu Python

```yaml
version: '3.7'

services:
  fastapi:
    container_name: backend
    build:
      context: .
      dockerfile: Dockerfile
    # image: fastapiapp:v2
    ports:
      - '8000:80'
    volumes:
      - ./app/songs.json:/code/app/songs.json
```

```
docker compose up
```

VS.

```
> docker build -t fastapiapp:v1 .
> docker run -d --rm --name backend \
    -v ${PWD}/app/songs.json:/code/app/songs.json \
    -p 8000:80 fastapiapp:v1
```

Docker Compose Webapp

```yaml
version: '3.7'

services:

  frontend:
    container_name: frontend
    build:
      context: .
      dockerfile: Dockerfile.prod
    ports:
      - '1337:80'
```

```
> docker-compose -d -f docker-compose.prod.yml \
    up
```

vs.

```
> docker run -it --rm -d --name frontend \
    -p 1337:80 frontend:prod
```

Docker Compose Full App

```yaml
version: '3.7'

services:
  frontend:
    container_name: frontend
    build:
      context: ./React/
      dockerfile: Dockerfile.prod
    ports:
      - '3000:80'
  fastapi:
    container_name: backend
    build:
      context: ./FastAPI/
      dockerfile: Dockerfile
    ports:
      - '8000:80'
    volumes:
      - ./FastAPI/app/songs.json:/code/app/songs.json
```

```
> docker-compose up
```

docker compose command

`docker build [OPTIONS] PATH | URL | -`

DESC

`[OPTIONS]`

-bla BLA

Andere UseCases - Docker

- OpenDrone Map
- Nathalies Kubernetes Arbeit
- Felix Hiwi arbeit
- deply your app on a cloud hosted frame work

Andere UseCases - Docker Compose

- OpenDrone Map
- Nathalies Kubernetes Arbeit
- Felix Hiwi arbeit
- deply your app on a cloud hosted frame work

Cheatsheet

- docker run
- docker build
- docker push, pull
- docker ps -a
- docker rm / rmi
- ...

Coole Quellen und so weiter

- https://www.docker.com/
- Offizielle Dokumentation:
  https://docs.docker.com/get-started/
-