52north
exploring horizons

KomMonitor Admin-Schulung

6 – Grundlagen zu Docker

Sebastian Drost, 52°North

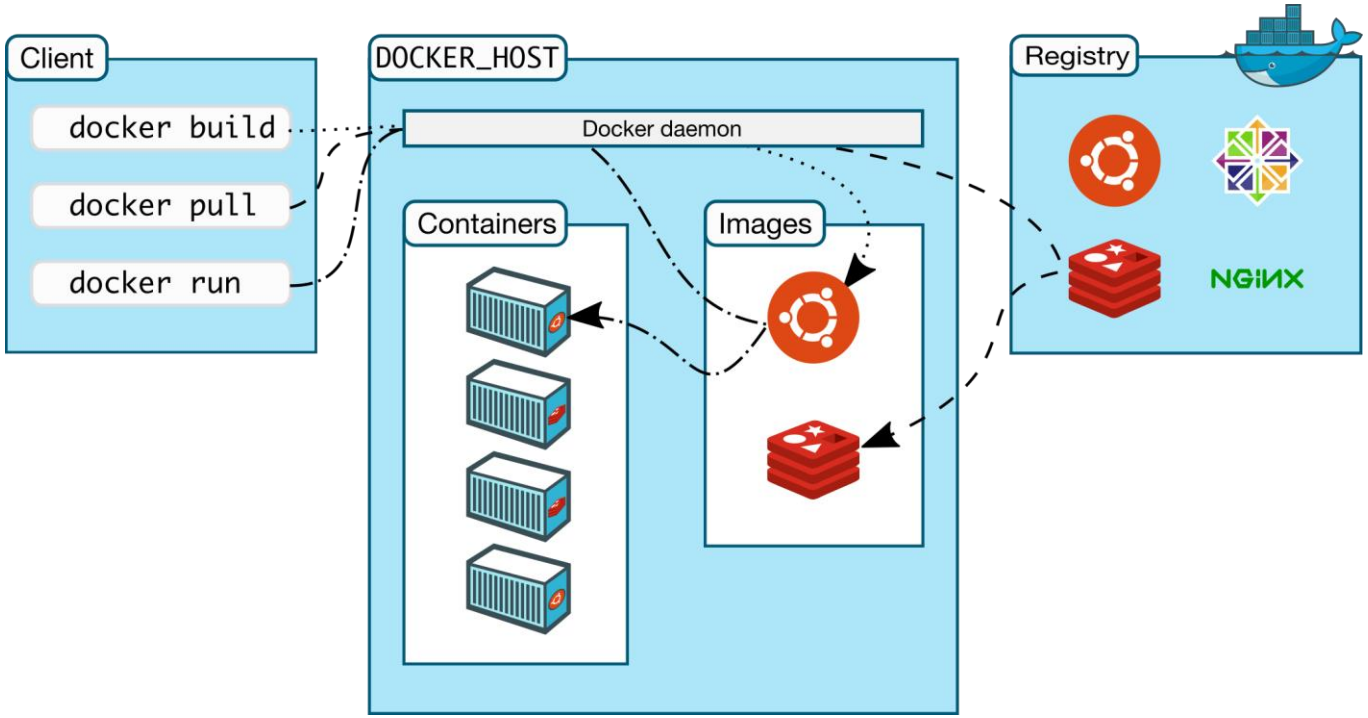
Schulungsinhalte – Docker Grundlagen

1) Docker Architektur und Vorteile von Docker
2) Docker Komponenten
   - Docker Engine und Engine API
   - Docker CLI und Docker Desktop
   - Docker Hub
3) Docker Konzepte
   - Dateimanagement über Volumes
   - Netzwerke in Docker
4) Docker Workflow
   - Dockerfiles schreiben
   - Bauen von Docker Images
   - Starten von Docker Containern

Was ist Docker?

- Tool zur Entwicklung, Verteilung und Ausführung von Anwendungen durch die Nutzung von Containertechnologie
- Eine Applikation wird mit allen notwendigen Bestandteilen für den Betrieb zusammengepackt
  - Bibliotheken, Datenbanken, Treiber, Konfigurationsdateien, Softwarecode
- Docker Container sind systemunabhängig
  - Ausführung von Containern in einer virtuellen Umgebung
- Server-Anwendungen können in Docker Containern fertig installiert und verteilt werden

Docker Architektur



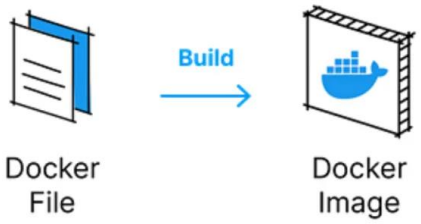
Quelle: Docker Inc. (2023). *Docker Overview*.
Abgerufen von: https://docs.docker.com/get-started/overview/ (Stand: 10.01.2023).

Docker Architektur

- Docker Daemon (Befehl: `dockerd`)
  - Prozess zur Verwaltung von Docker-Objekten (Images, Container, Networks, Volumes, …)
  - Kommunikation via Docker API
- Docker Client (Befehl: `docker`)
  - Command Line Interface (CLI), mit dem Nutzer mit Docker interagieren können
  - Sendet Befehle zur Umsetzung an den Docker daemon
- Docker Desktop
  - Anwendung für Mac, Windows und Linux, um containerisierte Anwendungen zu bauen
  - Beinhaltet u.A. den Docker Daemon und Docker Client
- Docker Registries
  - Repository zum Speichern von Docker Images (z.B. DockerHub)

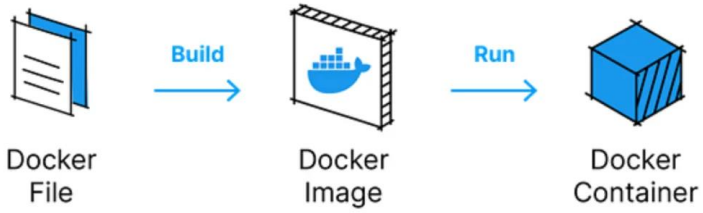
Docker Images

- Template, das Anweisungen enthält, um einen Docker Container zu bauen
- Docker-Images bestehen aus einem oder mehreren Dateisystem-Layern, die in einzelnen Build-Schritten aufeinander aufgebaut werden
- Images basieren i.d.R. auf einem Base-Layer (z.B. Ubuntu-Image)
- Images können aus einer Container Registry bezogen oder selber aufgebaut werden
- Dockerfiles enthalten Anweisungen zum Bauen von Images

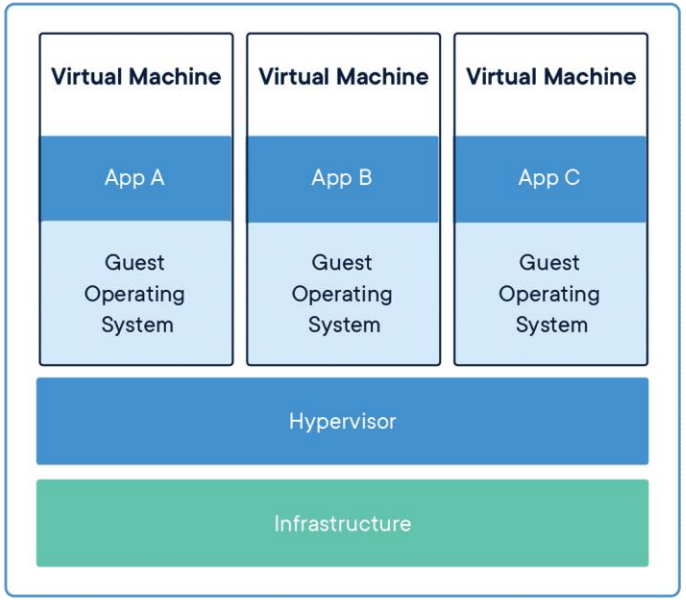


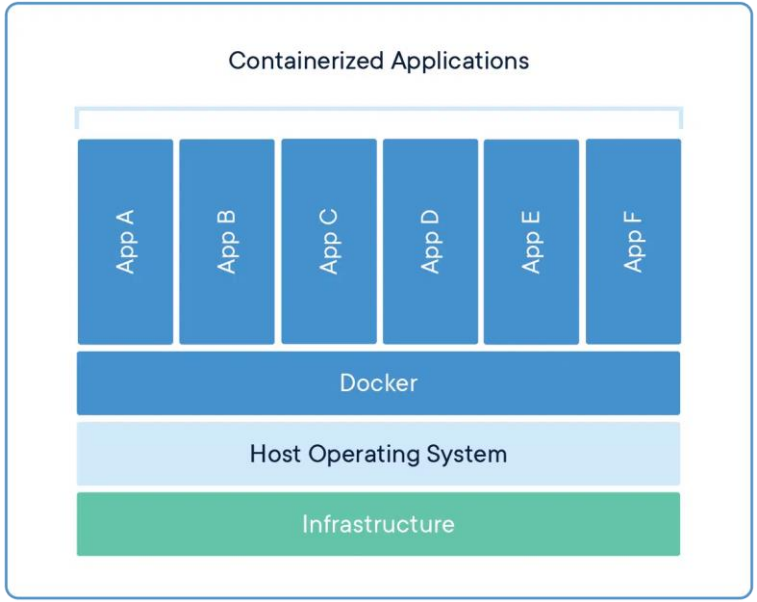
Docker Container

- Ausführbare Instanz eines Images, das alles beinhaltet, um eine Anwendung auszuführen
  - Software Code, Programmbibliotheken, Laufzeitumgebung, Systemtools, Konfigurationen
- Prozess, der von allen anderen Prozessen der Betriebsumgebung isoliert ist
  - Container laufen voneinander entkoppelt, können aber über ein gemeinsames Netzwerk miteinander kommunizieren
  - Container teilen sich die Systemressourcen
- Container können mit Hilfe der Docker CLI verwaltet werden (starten, stoppen, löschen, …)



Containers vs. Virtual Machines



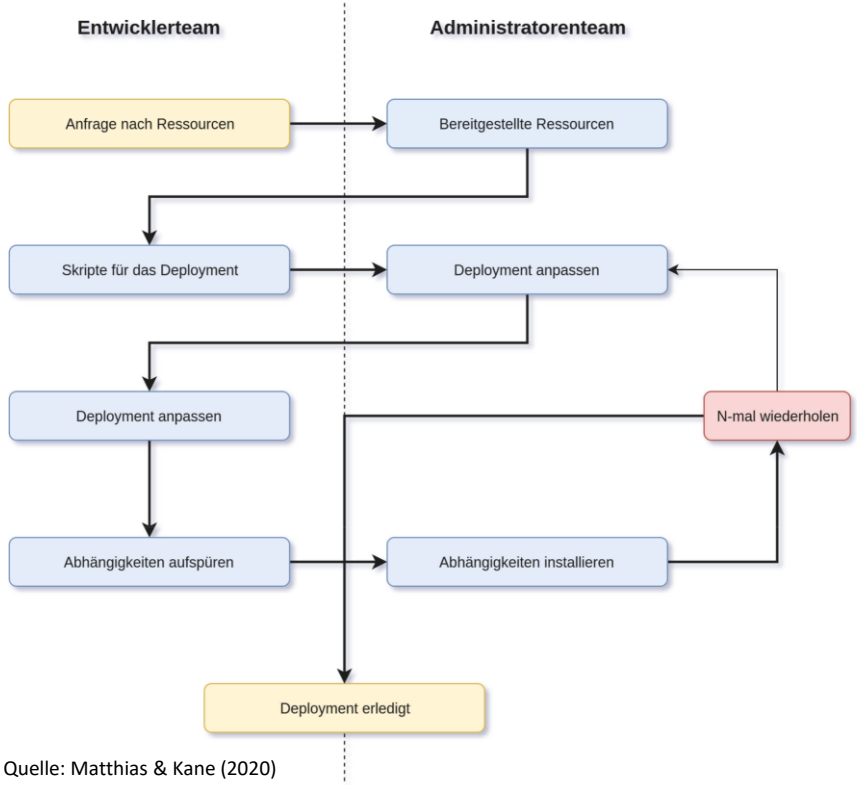


Quelle: Docker Inc. (2022). *Use containers to Build, Share and Run your applications.* Abgerufen von: https://www.docker.com/resources/what-container/ (Stand: 10.01.2023).

Vorteile von Docker

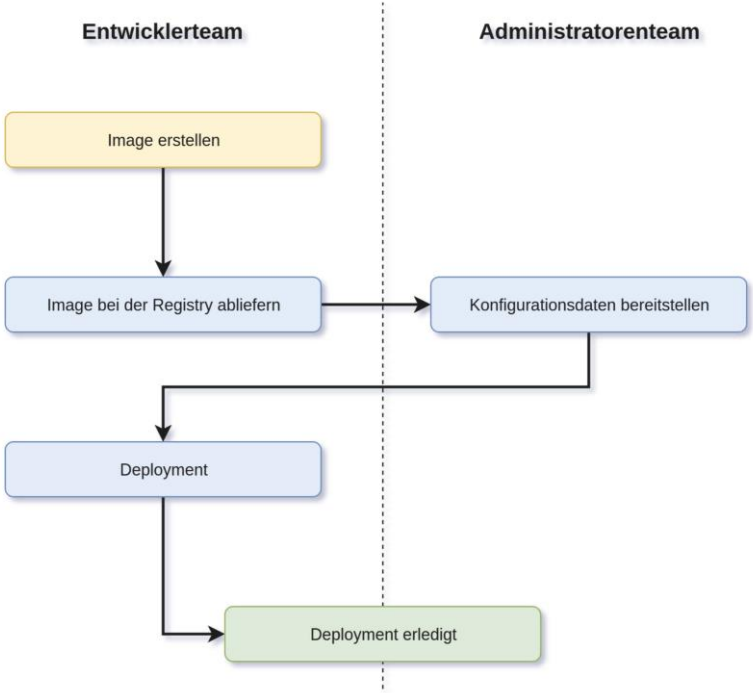
- Anwendungssoftware und Betriebssystemdateien werden in einem einheitlichen Image Format gebündelt
  - Keine spezifische Paketierung für unterschiedliche Zielsysteme
  - Docker bietet eine einheitliche Ausführungsumgebung
- Container eignen sich hervorragend für CI/CD Workflows
  - Ausliefern und Testen sowie Redeployments lassen sich automatisieren
- Leichtgewichtige und portable Container eignen sich für eine Vielzahl von Zielsystem
  - Laptop eines Entwicklers, physische oder virtuelle Maschinen, Cloud Plattformen
- Dynamische Skalierbarkeit bei starken oder niedrigen Auslastungen
- Hardware Ressourcen eines Servers werden optimal ausgenutzt, durch möglichst minimale Container

Deployment Workflow - Klassisch



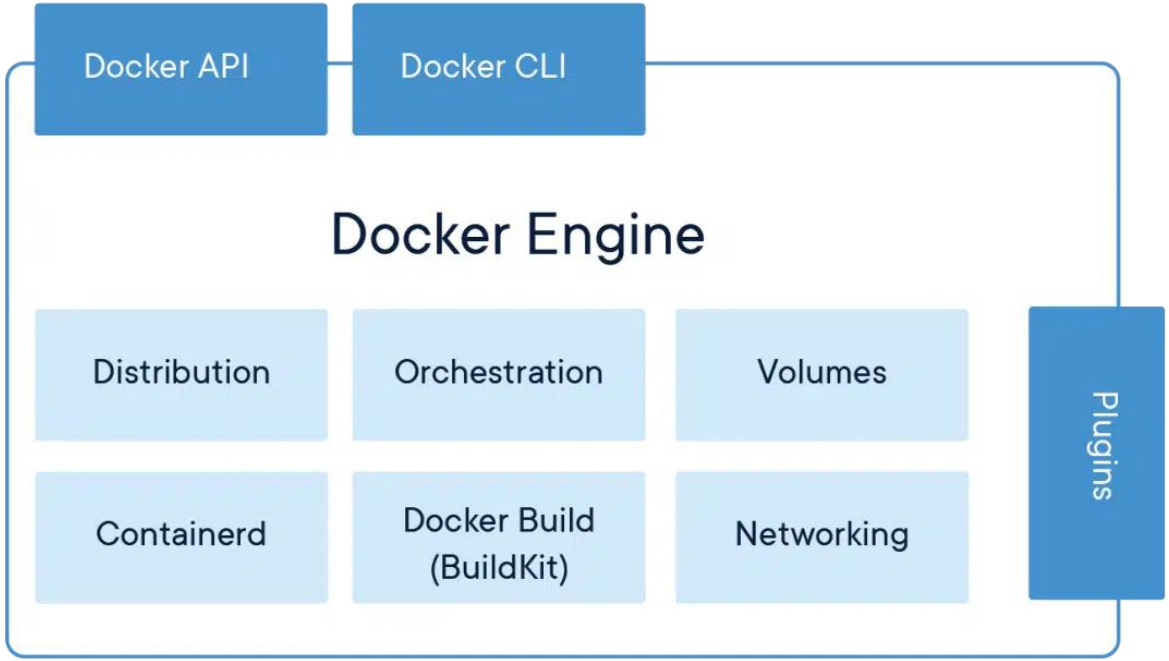
Quelle: Matthias & Kane (2020)

Deployment Workflow - Docker



Quelle: Matthias & Kane (2020)

Docker Engine



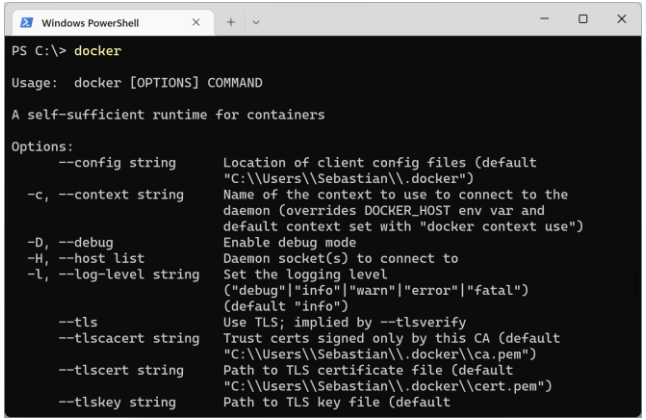
Quelle: Docker Inc. (2023). *The Industry-Leading Container Runtime.*
Abgerufen von: https://www.docker.com/products/container-runtime/ (Stand: 12.01.2023).

Docker Engine

- Client-Server Technologie, um containerisierte Anwendungen zu bauen und zu betreiben
- Bestandteil
  - Server mit ständig laufendem Daemon Prozess *dockerd*
  - API als Schnittstelle zur Kommunikation mit dem Docker Daemon
  - Command Line Interface *docker*
- Verfügbar für zahlreiche Desktop- und Server-Systeme: https://docs.docker.com/engine/install/#supported-platforms
- Docker Engine implementiert Konzepte für Container-basierte Workloads
  - Verschiedene Speicherkonzepte, um Daten eines Containers zu persistieren
  - Netzwerksubsystem, über das Container miteinander kommunizieren
  - Docker Build, zum Bauen von Docker Images

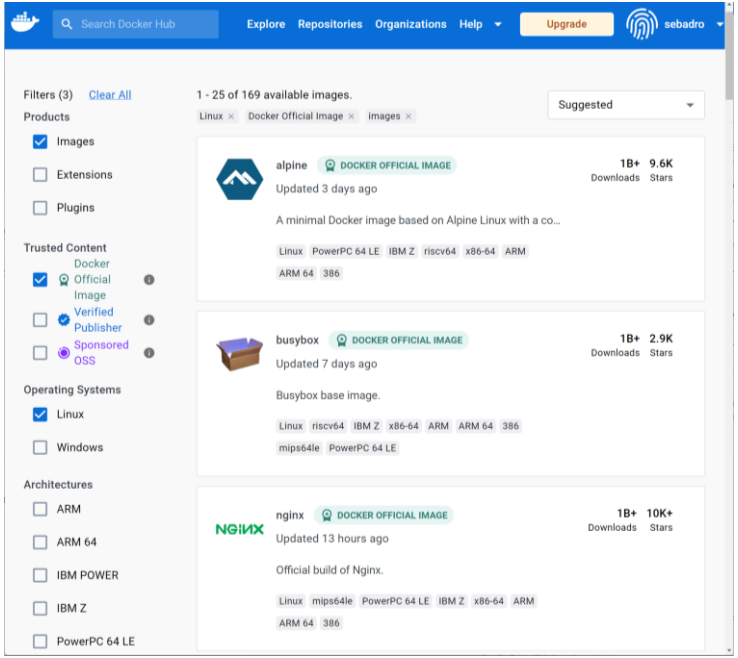
Docker CLI und Docker Engine API

- Docker stellt ein API bereit, um mit dem Docker daemon zu interagieren
  - RESTful API, mit der Clients über HTTP kommunizieren zu können
  - Referenzdokumentation: https://docs.docker.com/engine/api/v1.41/
  - Docker stellt SDKs (für Go oder Python) bereit, um eigene Docker Anwendungen zu bauen
- Das Docker Command Line Interface (CLI) ist ein Kommandozeilentool
  - Kommuniziert über die Docker Engine API, über Skripte oder direkt mit dem Docker daemon
  - Primäre Möglichkeit für Nutzer, um mit Docker zu arbeiten
  - Basis Kommando: `docker`



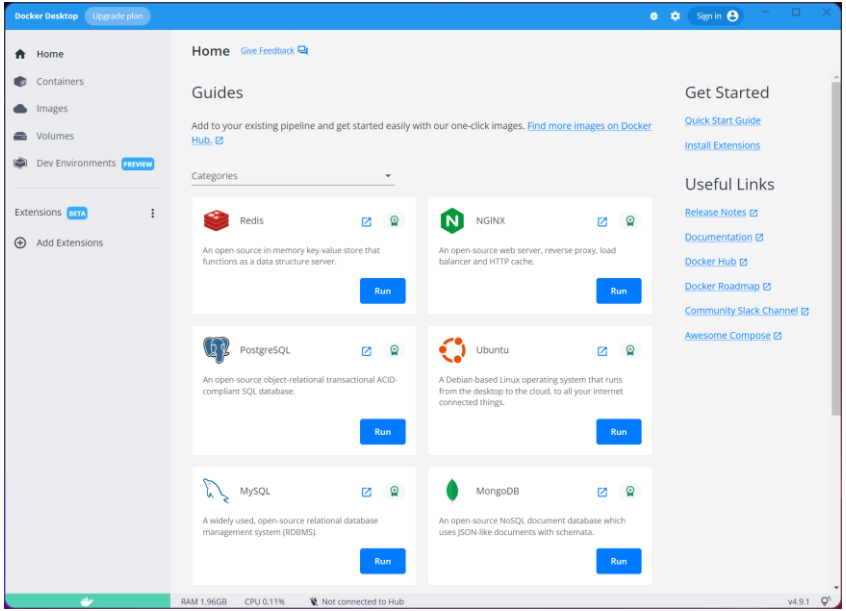
Docker Hub

- Öffentliche Plattform, um Docker Images mit der Community oder dem eigenen Team zu teilen
- Standardmäßig sucht Docker zunächst auf Docker Hub nach Images, wenn diese nicht auf dem lokalen System vorliegen
- Docker Hub unterstützt automatisch durchführbare Builds
  - Automated Builds für Business User
  - Externe Builds, z.B. von GitHub oder GitLab



Docker Desktop

- Installierbare Anwendung für Mac, Linux und Windows, um Docker Container zu bauen und zu teilen
- Bestandteile
  - Docker Engine
  - Docker CLI Client
  - Docker Build
  - Docker Compose
  - Kubernetes
- Docker Container können per CLI oder über eine einfach grafische Oberfläche verwaltet werden



Docker CLI – Nützliche Commands

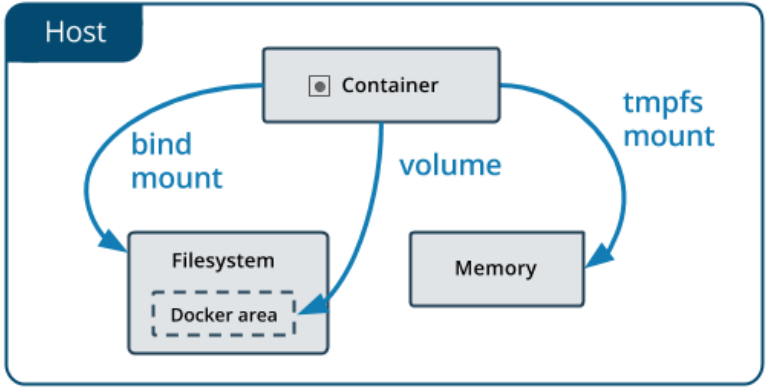
Command	Beschreibung
`docker build`	Baut ein Docker Image
`docker run`	Startet einen Container auf Basis eines Image
`docker stop`	Stoppt laufende Container
`docker rm`	Entfernt vorhandene Container
`docker rmi`	Entfernt ein vorhandenes Image
`docker restart`	Startet einen Container neu
`docker exec`	Führt einen Befehl innerhalb eines Containers aus
`docker logs`	Logs eines Containers abfragen

Docker CLI – Nützliche Commands

Command	Beschreibung
`docker images`	Auflistung aller Images auf dem Host
`docker ps`	Auflistung aller laufender Container
`docker ps -a`	Auflistung aller Container (auch beendete)
`docker pull`	Lädt ein Image einer Docker Registry (z.B. von Docker Hub) herunter
`docker push`	Lädt ein Image zu einer Docker Registry (z.B. Docker Hub) hoch
`docker volume create`	Erstellt ein neues Volume
`docker volume ls`	Listet alle Volumes auf

Dateimanagement in Docker

- Dateien innerhalb eines laufenden Containers überdauern nicht dessen Laufzeit
  - Dateien werden lediglich auf dem obersten, schreibbaren Layer geschrieben und sind an diesen, sowie den Host gebunden
  - Existiert ein Container nicht mehr, werden Daten, die zur Laufzeit angelegt wurden, nicht persistiert
- Optionen, zum Speichern von Daten
  - **Volumes:** Werden von Docker verwaltet
  - **Bind mounts:** Ordner oder Dateien des Host-Systems, die in einen Container gemounted werden
  - **tmpfs mounts:** Werden im Arbeitsspeicher des Host-Systems zur Container-Laufzeit vorgehalten (nur für Docker for Linux)



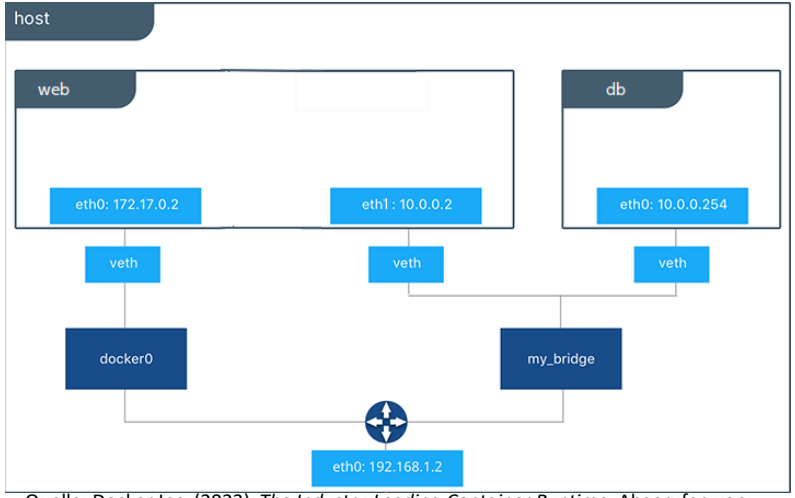
Quelle: Docker Inc. (2022). *Manage data in Docker.* Abgerufen von: https://docs.docker.com/storage/ (Stand: 12.01.2023).

Netzwerke in Docker

- Das Netzwerk-Konzept von Docker ermöglicht die Verknüpfung von Containern untereinander, unabhängig von der Systemumgebung
- Verschiedene Netzwerktreiber erlauben Flexibilität
  - **bridge**: Standard Netzwerktreiber, der verwendet wird, wenn Docker Container untereinander kommunizieren. Container innerhalb eines bridge Netzwerk sind von anderen Netzwerken isoliert.
  - **host:** Netzwerktreiber, der das Netzwerk des Hosts direkt benutzt. Container sind nicht voneinander isoliert.
  - **overlay:** Verbindet mehrere Docker daemons miteinander.
  - **ipvlan:** Ipvlan Netzwerke geben Nutzern die volle Kontrolle über IPv4 und IPv6 Adressierungen
  - **macvlan:** Ermöglicht die Zuweisung von MAC Adressen zu Containern, um über diese den Traffic zu routen.
  - **none:** Deaktiviert Netzwerke.
  - **Network plugin:** 3rd-Party Plugins für zusätzliche Netzwerkfunktionen

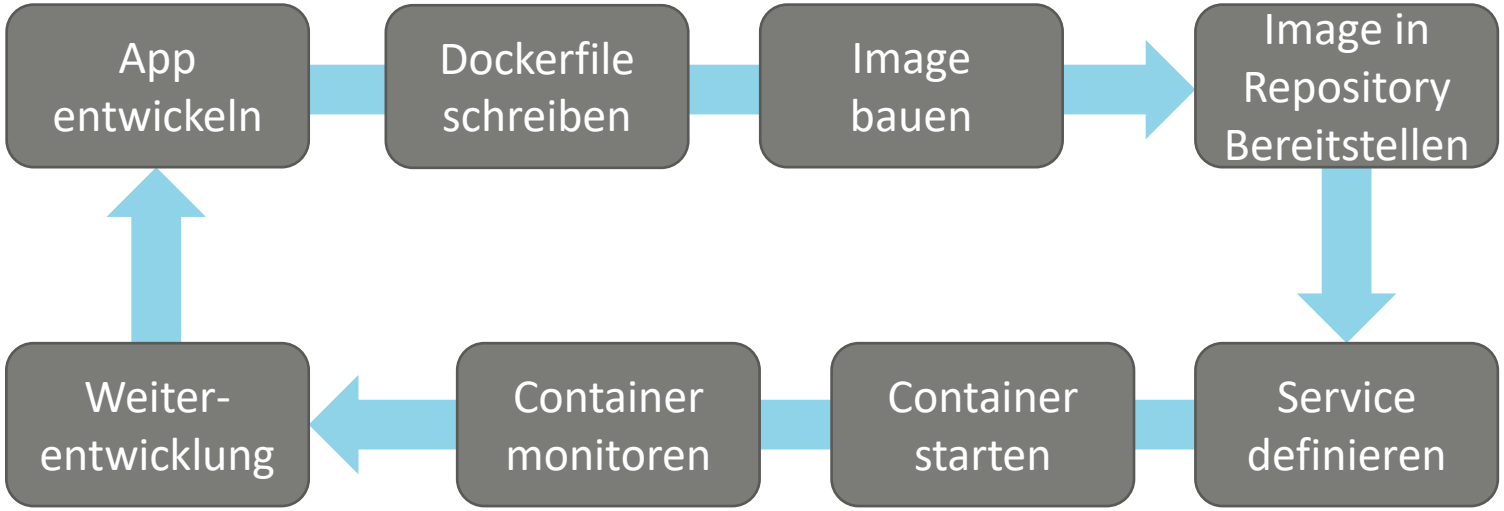
Netzwerke in Docker

- Docker Container lassen sich beliebig vielen Netzwerken zuweisen
- Standardmäßig werden Container dem Netzwerk „bridge“ zugewiesen, wenn keine andere Zuweisung erfolgt ist
- **Beispiel:**
  - Der Container *web* ist den Netzwerken *docker0* (alias für *bridge*) und *my_bridge* zugewiesen
  - Der Container *db* ist dem Netzwerk *my_bridge* zugewiesen
  - *web* kann ausschließlich über das Netzwerk *my_bridge* mit *db* kommunizieren
  - *db* ist für *web* unter der IP im *my_bridge* Netzwerk erreichbar (10.0.0.254) oder über den Container Namen (*db*)



Quelle: Docker Inc. (2023). *The Industry-Leading Container Runtime.* Abgerufen von: https://docs.docker.com/engine/tutorials/networkingcontainers/ (Stand: 13.01.2023).

Docker Workflow



52north

Dockerfiles – Grundlagen

- Dockerfiles sind Textdokumente, die Anweisungen beinhalten, um ein Docker Image zu bauen
- Anweisungen können alle Befehle sein, die auf einer Kommandozeile eingegeben werden können, um ein Docker Image aufzubauen
- Dockerfiles müssen zwingend mit einem **FROM** Statement beginnen
  - Definition eines Parent-Image, das als Basis zum Aufbau des eigenen Image verwendet wird
  - Als Parent-Image können z.B. Images aus öffentlichen Repositories (z.B. Docker Hub) verwendet werden
- Format eines Dockerfiles →

```
# Comment
INSTRUCTION arguments
```

Dockerfiles – Beispiel

- **Ausgangslage**: Python-Anwendung mit folgender Ordnerstruktur
  - *libs*: Enthält Python Module
  - *app.py*: Ausführbares Python-Programm
  - *requirements.txt*: Dependencies des Projekts
  - *Dockerfile*: Dockerfile der Anwendung
- **Ziel**: Definition des Dockerfiles, um für die Python-Anwendung ein Docker-Image zu erstellen

```
.
├── docs
├── libs
│   ├── __init__.py
│   └── module.py
├── tests
│   ├── __init__.py
│   └── test.py
├── myapp.py
├── requirements.txt
├── Dockerfile
└── README.md
```

Dockerfiles – Beispiel

Zeile	Erläuterung
1	Definition des Parent Image
2	Definition des Working-Directory
3	Kopieren aller Dateien des lokalen Ordners nach */opt/app*
4	Ausführung des Befehls zum Installieren von pip-Dependencies
5	Port-Zuweisung für den Container
6	Definition des Befehls, der beim Starten des Containers ausgeführt werden soll

```
1 FROM python:3.8

2 WORKDIR /opt

3 COPY . ./app

4 RUN pip install -r ./app/requirements.txt

5 EXPOSE 8000

6 CMD ["python", "/opt/app/myapp.py"]
```


52north

Dockerfiles – Nützliche Commands

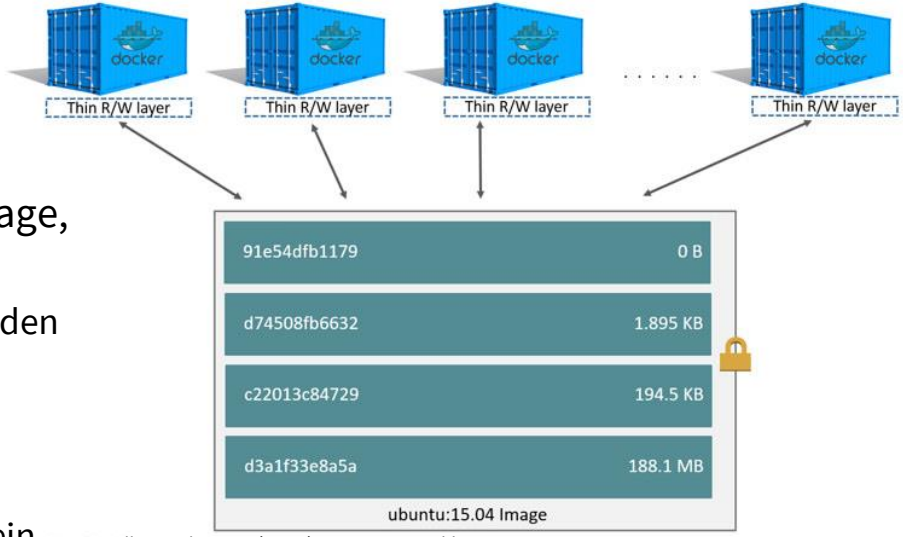
Command	Beschreibung	Beispiel
FROM	Definition des Basis Image	
RUN	Ausführung eines Befehls	
ADD	Kopiert Dateien des lokalen Systems oder einer URL in das Image	
COPY	Kopiert Dateien des lokalen Systems in das Image	
CMD	Definiert einen Standard-Befehl, der ausgeführt wird, wenn ein Container gestartet wird	
ENTRYPOINT	Definiert den Hauptbefehl des Image, sodass das Image, wie eine ausführbares Programm gestartet werden kann	

Dockerfiles – Nützliche Commands

Command	Beschreibung	Beispiel
WORKDIR	Definiert ein Arbeitsverzeichnis als Basis für alle weiteren Commands	
VOLUME	Erstellt ein Volume als Mount Point auf dem Host	
LABEL	Fügt beliebige Metadaten dem Image hinzu	
ENV	Environment Variable, die von nachfolgenden Commands verwendet werden kann	
ARG	Argument, das Nutzer beim Erzeugen eines Image Builds angeben können	
EXPOSE	Definiert den Port auf dem der Container Anfragen entgegennehmen kann	

Docker Images und Layer

- Docker Images bestehen aus einer Serie aufeinander aufbauender Layer
- Alle Anweisungen in Dockerfiles, die das Dateisystem verändern, fügen einen neuen Image-Layer hinzu:
  - Z.B.: FROM, COPY, RUN, CMD
- Layer eines Docker-Image sind read-only
- Beim Erstellen eines Containers für ein Image, wird ein schreibbarer Layer hinzugefügt
  - Änderungen am laufenden Container, werden in diesen obersten Layer geschrieben
  - Jede Container-Instanz hat einen eigenen schreibbaren Layer
  - So sind mehrere Container-Instanzen für ein einzelnes Image möglich



Quelle: Docker Inc. (2023). *Container and layers.* Abgerufen von: https://docs.docker.com/storage/storagedriver/#container-and-layers (Stand: 11.01.2023).

Docker Build

- Docker Build ist Bestandteil der Docker Engine und wird verwendet, um Docker Images zu erstellen
  - CLI Command: `docker build` → sendet Build-Befehl an die Docker Engine
- Docker Build verwendet Dockerfiles, um neue Container Images zu bauen
- I.d.R. werden neue Docker Images auf Grundlage eines vorhandenen Parent Image gebaut
  - Docker Build bezieht die notwendigen Image Layer von einem Repository (z.B. Docker Hub), wenn diese noch nicht im lokalen System vorliegen
  - Ausnahme: Base Images (z.B. essentielle OS Images) besitzen kein Parent Image
- Docker Images werden mit einem eindeutigen Tag versehen
  - Format: *<org>/<image-name>:<version>*
  - Beispiel: *n52/kommonitor-web-client:1.1.3*
- Docker Images können in öffentlichen oder privaten Repositories bereitgestellt werden

52north

Docker Build

- Befehl zum Bauen eines Image: **`docker build`**
  - Baut ein Image anhand eines Dockerfiles und eines Build Kontexts
  - Build Kontext: Dateien, die unter einem lokalen Pfad oder einer URL liegen
- Beispiel:
  `docker build -t n52/demo-app:0.1.0 .`
  - **-t**: Tag, der für das Image verwendet wird
  - **.**: Aktueller Pfad, wird als Kontext für den Build verwendet
    - Die gesamte Ordnerstruktur wird für den Build an den Docker Daemon gesendet
- Dokumentation:
  https://docs.docker.com/engine/reference/commandline/build/

```
.
├── docs
├── libs
│   ├── __init__.py
│   └── module.py
├── tests
│   ├── __init__.py
│   └── test.py
├── myapp.py
├── requirements.txt
├── Dockerfile
└── README.md
```


52north

Docker Container ausführen

- Ausführung per **`docker run`** Command
  - Startet einen neuen Container auf Basis eines Image, indem ein neuer schreibbarer Layer dem Image on-top hinzugefügt wird
  - Container, die gestoppt wurden, können wieder gestartet werden, ohne Verlust von der Änderungen am Dateisystem (lediglich das Löschen eines Containers bewirkt Dateiverluste)
  - Beispiel: **`docker run -it -p 8000:8080 n52/demo-app:0.1.0`**
    → Startet den Container demo-app mit einem Port-Mapping von 8000 auf 8080
  - Dokumentation: https://docs.docker.com/engine/reference/commandline/run/
- Ausführung per **`docker compose`** Command
  - Docker Compose ist ein Tool, um Multi-Container Anwendungen zu bauen
  - Container-basierte Dienste werden in einer YAML-Datei definiert
  - Beispiel: **`docker compose up`**
    → Startet einen oder mehrere Container gemäß der Definition einer docker-compose.yml

Docker Compose – docker-compose.yml

Zeile	Erläuterung
1	Docker Compose Version
2-13	Definition aller Anwendungen als Service
3-11	Demo App Services
4	Container-Image, das verwendet werden soll
5-6	Port Mapping von 8000 (Host-Port) auf 8080 (Container-Port
7-9	Volumes für den Container
8	Mount Host Path
9	Named Volume
10-11	Abhängigkeiten zu anderen Services (relevant für das Starten eines Containern)
12	App DB Service
13	Verwendung des neuesten (latest) Container Image
14	Konfiguration eines Names Volumes

```
1  version: "3.9"
2  services:
3    demo-app:
4      image: n52/demo-app:0.1.0
5      ports:
6        - "8000:8080"
7      volumes:
8        - .:/data
9        - logvolume:/var/log
10     depends_on:
11       - app-db
12   app-db:
13     image: redis/redis:latest
14 volumes:
15   logvolume: {}
```


52north

Docker Workflow

	Schritt	Befehl
1.	Erstellen eines Dockerfiles	
2.	Bauen eines Docker Image	**docker build –t n52/demo-app:0.1.0 .**
3.	Starten eines Containers	**docker run –it –p 8000:8080 n52/demo-app:0.1.0** (oder per Docker Compose)
4.	Container monitoren	**docker logs <CONTAINER-ID>** (oder per Portainer)
5.	Anwendung ausführen	z.B. Service Request via Web Browser
6.	Container beenden	**docker stop <CONTAINER-ID>**
7.	Container entfernen	**docker rm <CONTAINER-ID>**

52north

Quellen

Docker Inc. (2023a). *Get started*. https://docs.docker.com/get-started/, letzter Zugriff 16.01.2023.

Docker Inc. (2023b). *Docker Dekstop*. https://docs.docker.com/desktop/, letzter Zugriff 16.01.2023.

Docker Inc. (2023c). *Docker Engine*. https://docs.docker.com/engine/, letzter Zugriff 16.01.2023.

Docker Inc. (2023d). *Docker Build*. https://docs.docker.com/build/, letzter Zugriff 16.01.2023.

Docker Inc. (2023e). *Docker Compose*. https://docs.docker.com/compose/, letzter Zugriff 16.01.2023.

Docker Inc. (2023f). *Docker run reference*. https://docs.docker.com/engine/reference/run/, letzter Zugriff 16.01.2023.

Docker Inc. (2023g). *Dockerfile reference*. https://docs.docker.com/engine/reference/builder/, letzter Zugriff 16.01.2023.

Docker Inc. (2023h). *Compose specification*. https://docs.docker.com/compose/compose-file/, letzter Zugriff 16.01.2023.

Hopp, H.-M. (2021). *Docker Handbuch für Einsteiger: Der leichte Weg zum Docker-Experten* (Ausgabe 2021). BMU Media GmbH.

Matthias, K., & Kane, S. P. (2020). *Docker: Praxiseinstieg* (S. Vijayakumaran & K. Lorenzen, Übers.; 2. Auflage). mitp.

Übung

Der gesamte Docker Workflow kann in einer praktischen Übung nachvollzogen und verstätigt werden

- Dockerfiles schreiben, Docker Images bauen, Docker Container starten, Interaktion mit Docker Services, Umgang mit Docker Compose

1) Stellen Sie sicher, dass Sie Docker Desktop installiert haben!
   - https://www.docker.com/products/docker-desktop/
2) Klonen Sie das nachfolgende GitHub Repository mit git oder laden Sie das *.zip-Paket herunter und entpacken es!
   - https://github.com/SebaDro/python-docker-demo
3) Öffnen Sie ein beliebiges Kommandozeilenprogramm innerhalb entpackten Projektordners!
4) Folgen Sie den Anweisungen der README innerhalb des Repositories!

52north
exploring horizons

Kontakt bei Rückfragen

s.drost@52north.org