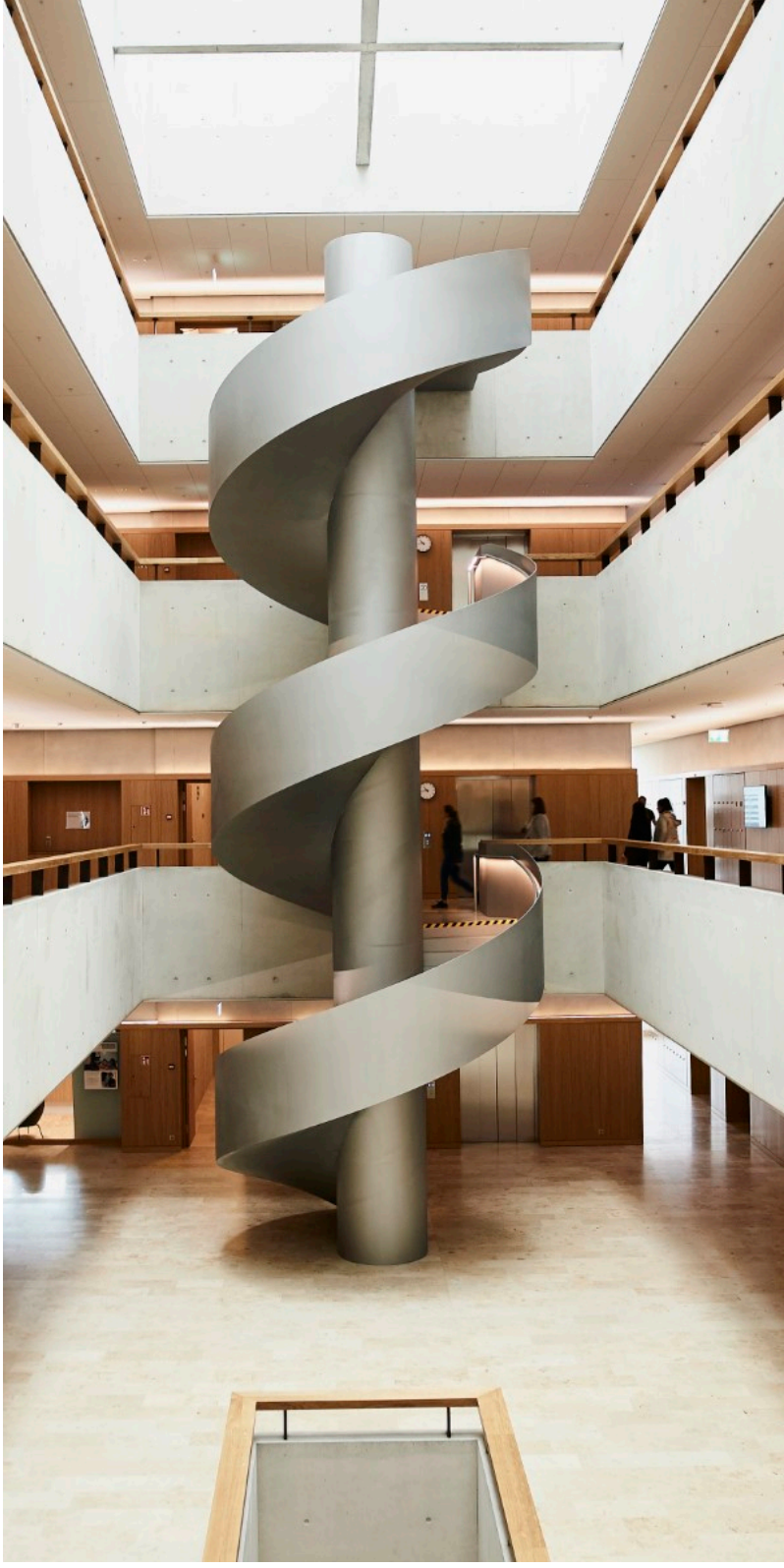
Hochschule Luzern
Modul I.BA_ISLAB



Laborübung

Docker Introduction

HSLU Hochschule Luzern

Dokument Version: #bd77f8f
Template Version: #703559b
Letzte Änderung: 2023-11-13
E. Sturzenegger, N. Richner, T. Jösler

I. Allgemeine Informationen

Name:

Aurel Corti, David Jäggli

Gruppe:

Gruppe 13

Bemerkungen:

Liste der Verfasser

E. Sturzenegger	Release 1.0 Anpassung gemäss Feedback Auf den neusten Stand gebracht Anpassungen Docker Compose Kapitel Docker Registry Mirror hinzugefügt
N. Richner	Docker ENV Syntax geändert
T. Jösler	Deprecated Docker Machine entfernt

Copyright Informationen

Alle Rechte vorbehalten

II. Inhaltsverzeichnis

1. Vorbereitung **4**
1.1. Einleitung . . . 4
1.2. Hausaufgaben . . . 4
1.3. Benötigte Mittel . . . 6
1.4. Vorbereitungsaufgaben . . . 6
1.5. Theorie . . . 6

2. Erstes Container Image **8**

3. Architektur und Dockerfile **10**
3.1. Dockerfile . . . 10
3.2. Docker Images und Containers im Detail . . . 16
3.3. Container Isolation . . . 18
3.4. Aufräumen . . . 21

4. Docker Networks **22**
4.1. Bridge Netzwerktreiber . . . 22
4.2. Host Netzwerktreiber . . . 26

5. Docker Storage **28**
5.1. Erstellen eines Volumes . . . 29
5.2. Bind Mount Host Ordner . . . 29

6. Docker Compose **31**
6.1. Die Compose Datei . . . 31

7. Jetzt sind Sie dran – Ihre eigene Docker App **35**

III. Vorwort

Feedback

Mit Ihrer Mithilfe kann die Qualität des Versuches laufend den Bedürfnissen angepasst und verbessert werden.

Falls in diesem Versuchsablauf etwas nicht so funktioniert wie es beschrieben ist, melden Sie dies bitte direkt dem Laborpersonal oder erwähnen Sie es in Ihrem Laborbericht oder Protokoll. Behandeln Sie die zur Verfügung gestellten Geräte mit der entsprechenden Umsicht.

Bei Problemen wenden Sie sich bitte ebenfalls an das Laborpersonal.

Legende

In den Versuchen gibt es Passagen, die mit den folgenden Boxen markiert sind. Diese sind wie folgt zu verstehen:

Wichtig

Dringend beachten. Was hier steht, unbedingt merken oder ausführen.

Aufgabe III.1

Beantworten und dokumentieren Sie die Antworten im Laborprotokoll.

Hinweis

Ergänzender Hinweis / Notiz / Hilfestellung.

Information

Weiterführende Informationen. Dies sind Informationen, die nicht zur Ausführung der Versuche benötigt werden, aber bekannt sein sollten.

Story

Hierbei wird die Geschichte vermittelt, die in den Versuch einleitet oder den Zweck des Versuches vorstellt.

Zielsetzung

Lernziele, die nach dem Bearbeiten des Kapitels erfüllt sein sollten.

Erkenntnis

Wichtige Erkenntnisse, die aus dem Versuch mitgenommen werden sollten.

1. Vorbereitung

1.1. Einleitung

Diese Laborübung soll den Studierenden den praktischen Umgang mit Containern anhand von Docker näherbringen. Dabei werden die Studierenden das praktisch umgesetzte immer wieder mit den gelernten theoretischen Grundlagen verifizieren.

1.2. Hausaufgaben

Dieses Kapitel beschreibt Vorbereitungsmassnahmen, die vor Beginn der Übung durchzuführen sind.

1.2.1. Theorie

Schauen Sie sich das erste Video an und beantworten Sie die untenstehenden Fragen.

Video 1 (Basics): https://www.youtube.com/watch?v=aLipr7tTuA4

Video 2 ist optional. Falls Sie schon Kenntnisse mitbringen, zeigt dieses Video das Zusammenspiel der verschiedenen Standards anhand von OpenShift (Container Software von Red Hat):

- Was ist ein Container: https://youtu.be/rlj0UZlvGp0
- Welche Standards gibt es: https://youtu.be/rlj0UZlvGp0?t=356
- Wie spielen diese Standards zusammen: https://youtu.be/rlj0UZlvGp0?t=918

1.2.2. Fragen zur Theorie

Aufgabe 1.1

Was sind Docker Images?

Docker Images enthalten die Umgebung mit Betriebssystem, Libraries und so weiter, plus die Applikation die danach auf dem Container laufen soll. Aus ihnen können Container gestartet werden, indem der Docker Daemon das Image laufen lässt und somit die richtige Umgebung bereitstellt. Sie können erweitert und geändert werden, bereits laufende Container können auf neue Versionen gebracht werden und installieren dafür nur die nötigen Änderungen, was es leicht macht.

Aufgabe 1.2

Welche Webseite/Registry bietet Docker Images an?

Docker.com
Dockerhub.com

Aufgabe 1.3

Welches sind die verschiedenen Komponenten einer Dockerumgebung?

Core Components:
Linux: Docker Client, und Docker Daemon
Windows/OS X: Docker Client, Docker Daemon (in einer boot2docker VM instanz)
Workflow Components:
Docker Image, Docker Container, Dockerfile, Docker Registry

Aufgabe 1.4

Docker bringt einige Vorteile mit sich. Zählen Sie 3 davon auf.

Scalable, weil keine Gast OS sind Docker instanzen günstiger als VMs und es können mehr gleichzeitig laufen

Portable, wenn ein Gerät die Docker Engine hat, kann ich meine Container drauf laufen lassen, verschieben

Deployment ist einfach und funktioniert auf jeder Umbegung mit Docker

Aufgabe 1.5

Wie unterscheiden sich Docker Containers zu virtuellen Maschinen? In Bezug auf Speicherplatz, wie werden die beiden gesteuert?

Eine Virtuelle Maschine hat immer ein GastOS, welches läuft. dieses Fällt bei Docker weg. Die Applikationen können direkt auf der Docker Engine (bei virtuellen Maschinen der Hypervisor).
Da ein Gast OS je nach dem recht "schwer" sein kann (viel Speicher brauchen).
Eine Virtuelle Maschine stellt ein komplettes OS zur Verfügung, mit dem wir direkt interagieren. Bei Docker hingegen Kommunizieren wir mit dem Docker Daemon, der die Container verwaltet.

Aufgabe 1.6

Ist es Best Practice mehrere Applikationen in einem Container laufen zu lassen (Nochmals mit Video verifizieren)?

Nein, es ist Best Practice ein Container für eine Apllikation zu haben. So dass sie isoliert sind und nur das Nötige mitbekommen.

Aufgabe 1.7

In welchen Belangen sind Containers isoliert gegenüber anderen Containern? Zählen Sie 3 Punkte auf.

Binaries / Libraries
Applikation
OS

Zielsetzung

In diesem Lab wird folgendes erarbeitet und gelernt.

- Herunterladen von Images von der Docker Hub Registry und instanziieren von Container
- Aufbau und Verwendung eines Dockerfiles
- Netzwerkaufbau mit Docker Container
- Dateisystemanbindung an einen Container
- Komposition von grösseren Applikationen

1.3. Benötigte Mittel

Verwenden Sie für diesen Versuch die Labor-Arbeitsstationen. Jedes Labor-Team benötigt einen Labor-Doppelarbeitsplatz.

Zudem benötigen Sie die Docker Umgebung. Diese befindet sich auf SWITCHengines und Sie können sich Remote per Remote Desktop damit verbinden.

- Linux Client

Wichtig

Diese Durchführung wird auf neuer Infrastruktur durchgeführt, somit kann es zu Abweichungen kommen. Vor und während dem Bearbeiten der Laborübungen bitte Informationen des Laborpersonals beachten und den Discord-Kanal im Auge behalten. Das Laborpersonal wird mitteilen, sollte trotz der Tests etwas nicht wie in diesem Dokument beschrieben funktionieren.

1.4. Vorbereitungsaufgaben

In diesem Kapitel werden vorbereitende Aktionen durchgeführt, damit die Übung funktioniert.

Bevor Sie mit der Übung fortfahren, sollten Sie den Apache Service stoppen und deaktivieren.

```
1 sudo systemctl disable --now apache2
```

Alle für die Übung benötigten Dateien befinden sich im Ordner **”~/dockerlab”** bzw. **”/home/labadmin/docker-lab”**.

In der Übung werden Sie URLs mit <HOST> antreffen. Sie können diesen Platzhalter mit einem der folgenden Werte ersetzten:

- localhost
- IP des docker0 Interfaces
- Die private IPv4 Adresse des Linux Clients oder islab-lc-<Gruppe>.zh.switchengines.ch

1.5. Theorie

1.5.1. Befehlsstruktur von Docker

Im Verlauf der Übung werden Sie verschiedene Docker Befehle kennenlernen. Diese sind grundsätzlich immer ähnlich aufgebaut:

`docker` <Für was> <Was tun> <Flags> <Image, Name, ID>

Hier einige Beispiele:

Befehl	**<Für was>**	**<Was tun>**	**<Flags>**	**<Image, Name, ID>**
docker	container	run	--name <Container>	nginx
docker	images	pull		nginx
docker	compose		-dit	

2. Erstes Container Image

Ein Container Image ist eine Datei, welche von einem Registry Server (zum Beispiel Docker Hub) geladen oder selbst erstellt werden kann.

Da es mehrere Container Engines nebst Docker gibt, wurde der Open Container Initiative (OCI) Standard für Container Images eingeführt. Dieser Standard definiert, wie Container Image Layers und deren Metadaten auszusehen haben. Die meisten Container Engines (so auch Docker) unterliegen der Container Runtime "runc", welche andere Zuständigkeiten wie die Container Engine besitzt.

Folgendes sind die jeweiligen Aufgaben von Container Engine und Container Runtime.

Container Engine (Docker)	Container Runtime (runc)
• User Input verarbeiten und an eine API (Docker Client) weitergeben zur weiteren Verarbeitung • Container Images von einem Registry Server herunterladen • Packen und entpacken von Container Images auf der Disk • Vorbereiten von Container Mount Points (anhängen von Datenspeicher an einen Container) • Vorbereiten der Metadaten für die Container Runtime (zum Beispiel durch nachträglichen User Input, welcher vordefinierte Instruktionen ersetzt. Sie werden diese Instruktionen im Kapitel 3.1 kennenlernen) • Die Container Runtime starten	• Verwenden des Container Mount Points • Verwenden der Container Metadaten • Kommunikation mit dem Kernel, um den containerized Prozess zu starten • Einrichten von Regeln und zuweisen von Ressourcen

Zu Beginn wollen wir ein erstes nginx Image laden. Da dieses Image zuvor noch nicht heruntergeladen wurde, laden wir es über Docker Hub herunter. Mit dem "latest" Tag geben wir an, dass die neuste Version verwendet werden soll.

Hinweis

Nginx ist eine in C geschriebene Software die meistens Verwendung als Webserver findet. Weitere Anwendungsbereiche sind Reverse Proxy, Load Balancing und vieles mehr.

```
1 docker image pull nginx:latest
```

Vergewissern Sie sich, ob das Image nun lokal vorhanden ist, indem Sie sämtliche lokal vorhandenen Images auflisten.

```
1 docker images
```

Aus dem Image kann nun ein Container gestartet werden. Wir geben dem Container einen Namen und stellen den

Webserver auf unserem Host über Port 8080 zur Verfügung. Dabei wird Port 80 im Container auf Port 8080 auf dem Host gemappt.

- `-i` Interactive. Lässt den Input Kanal des CLI offen (STDIN). In Kombination mit dem -t Flag, ermöglicht dies dem Container Befehle zu senden.
- `-t` TTY. Erstellt ein Terminal Session. Das verwenden Sie auch standardmässig, wenn Sie auf Ihrem Linux Client ein Terminal Tab öffnen. Mit jedem neuen Tab erhalten Sie eine neue Terminal Instanz. Sie können die aktuelle Instanz mit dem Befehl ”tty” ausgeben.
- `-d` Detached oder auch Daemon. Beim Starten eines Containers wechselt Ihre Terminal Ansicht von Host zu Container. Unter Angabe des detached Flag bleiben Sie auf dem Host und der Container wird als Hintergrundprozess gestartet.

```
1 docker container run -itd --name web-server-nginx -p 8080:80 nginx:latest
```

Hinweis

Falls Sie jemand der schnellen Sorte sind, können Sie auch eindeutige Befehle in Kurzschreibweise verwenden. Im letzten Befehl hätte man das Keyword ”container” auch weglassen können.

Vergewissern Sie sich nun, dass der Container läuft, indem Sie sämtliche Container auflisten. Mit dem ”-a” Flag werden alle Container aufgelistet. Ansonsten sehen Sie nur momentan laufende Container. Hier sehen Sie auch die Port Mappings.

```
1 docker container ps -a
```

Öffnen Sie anschliessend mit dem Firefox Webbrowser die Adresse ”localhost:8080”.

Aufgabe 2.1

Was wird auf der Webseite angezeigt?

Welcome to nginx!
If you see this page, the nginx web server is successfully installed and working. Further configuration is required.
For online documentation and support please refer to nginx.org.
Commercial support is available at nginx.com.
Thank you for using nginx.

Hinweis

Container können über ihre ID angesprochen werden. Diese ID sehen Sie bei der Erstellung eines Containers oder auch über den Befehl ”docker container ps -a” in der ersten Spalte. Dabei reicht es, wenn lediglich die ersten paar Zeichen der ID verwendet werden, sodass der Container von den anderen eindeutig unterschieden werden kann. Das wird jedoch schnell unübersichtlich. Deswegen bietet Docker die Option ”--name” für die Erstellung von Images (docker image --name myimagename) und Container (docker container --name mycontainername).

3. Architektur und Dockerfile

Docker basiert auf dem Prinzip Build, Ship, Run. Damit soll ein unkompliziertes, OS unabhängiges Arbeiten ermöglicht werden.

Abbildung 1: Docker Workflow (https://en.wikitolearn.org/Course:Docker_Container_Hands-On/Docker_Foundations/Build,_Ship,_Run)

Damit dies möglich ist, braucht es eine Engine die auf allen Betriebssystemen gleich ist. Docker Engine erfüllt genau diesen Zweck. Die Engine teilt sich auf in 3 Komponenten. Der Docker Client stellt die Befehle, die wir in das Terminal eingeben können, zur Verfügung (zum Beispiel ”docker pull” und ”docker container run” aus dem vorherigen Kapitel). Der Docker Host führt diese Befehle aus und verwaltet sämtliche Images und Container, die lokal vorhanden sind. Über die Docker Registry (zum Beispiel Docker Hub) werden Images zentral verwaltet.

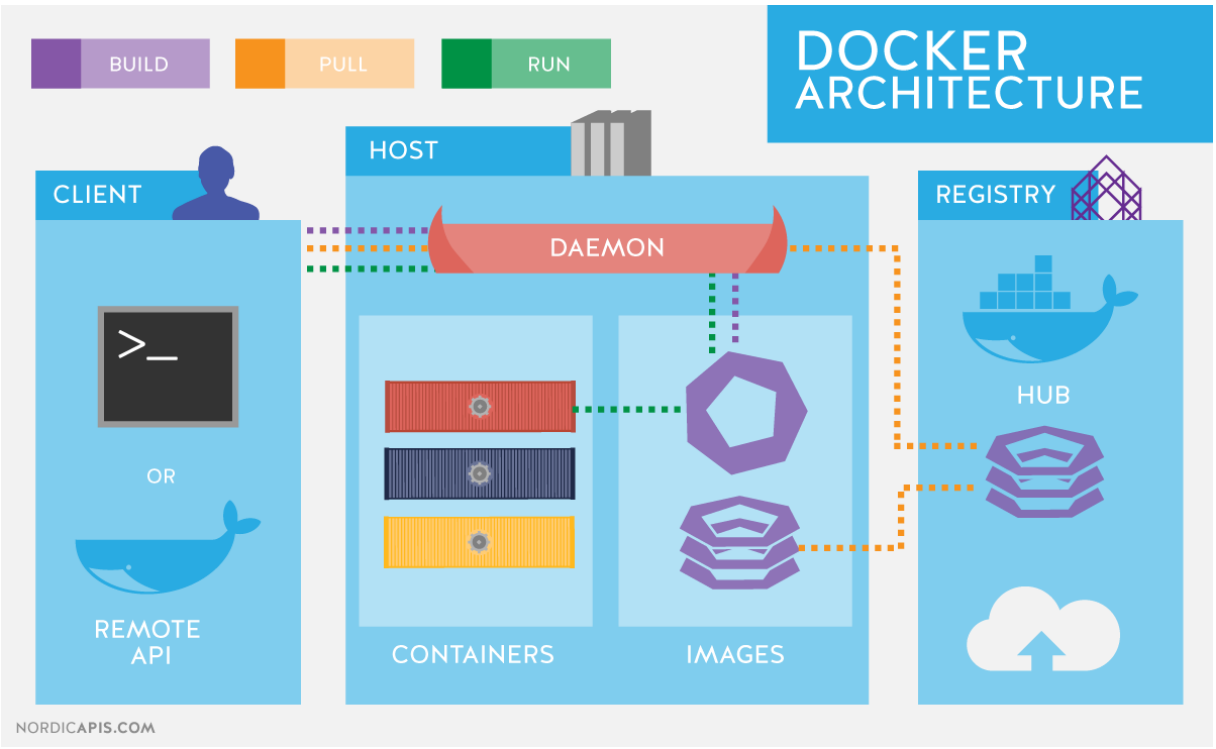


Abbildung 2: Docker Komponenten (https://www.aquasec.com/wiki/display/containers/Docker+Architecture)

3.1. Dockerfile

Das Dockerfile ist die Grundlage (Blueprint) eines Images um daraus Container erstellen zu können.

Das Dockerfile enthält sämtliche Instruktionen, welche für das Erstellen eines Images notwendig sind. Die Instruktionen werden durch Docker automatisch in Command Line Befehle interpretiert.

Die Instruktionen in einem Dockerfile werden immer von oben nach unten gelesen und ausgeführt. Das Dockerfile wird in drei Bereiche aufgeteilt.

Fundamental	Fundamentale Angaben wie zum Beispiel: Image Basis aus eigener oder aus externer Registry (FROM), benötigte CLI Argumente (ARG).
Configuration	Dieser Bereich des Dockerfiles dient der Konfiguration des Containers. Dazu gehören zum Beispiel: Software Dependencies (RUN), Dateistrukturen (ADD | COPY) sowie Umgebungsvariabeln (ENV).
Execution	Instruktionen im Bereich der Execution werden nach der Configuration ausgeführt. Die CMD Instruktion wird ausgeführt, wenn dem Container bei der Erstellung kein Befehl mitgegeben wird. Im Gegensatz zur CMD wird die ENTRYPOINT Instruktion auch dann ausgeführt, wenn ein Befehl dem Container mitgeben wird. Die EXPOSE Instruktion ist nur eine Dokumentation. Sie gibt an, dass der spezifische Port bei der Erstellung eines Containers explizit freigegeben werden sollte.

Hinweis

Die oben aufgeführten Instruktionen werden im Anschluss behandelt. Andernfalls finden Sie vertiefende Informationen hier: https://docs.docker.com/engine/reference/builder

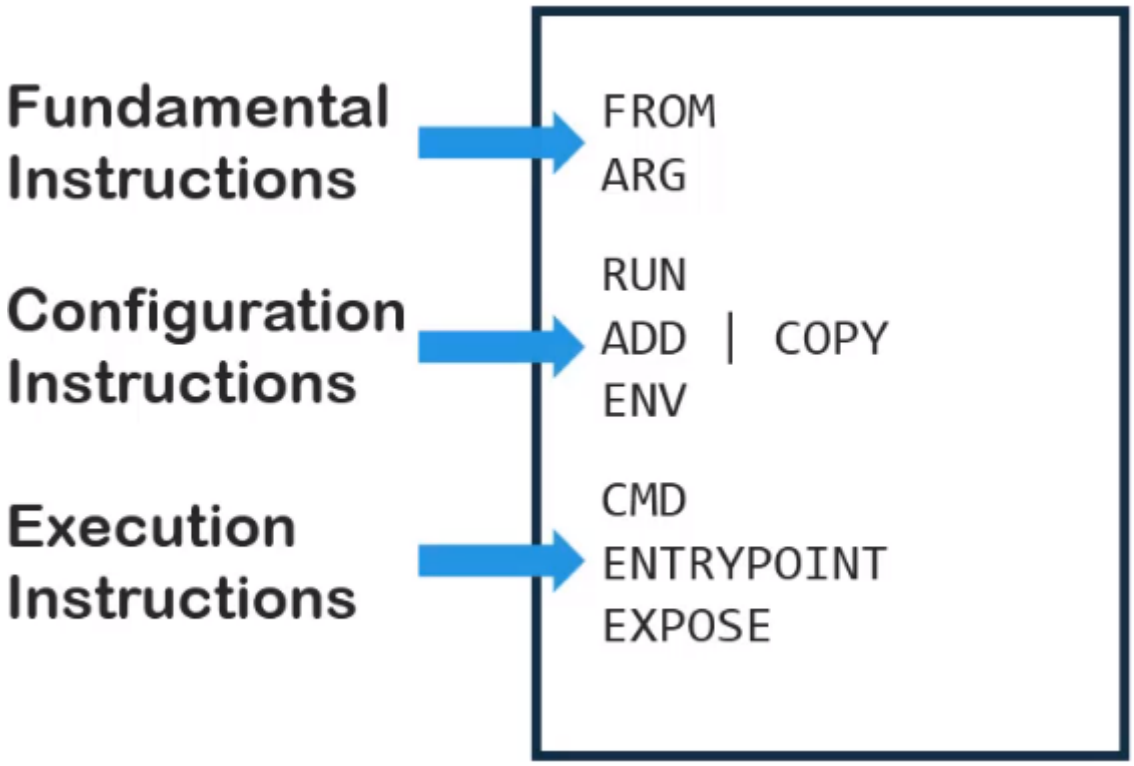


Abbildung 3: Dockerfile Struktur

Öffnen Sie nun den Ordner **"~/dockerlab/dockerfile"** und erstellen Sie ein neues Dockerfile.

```
1 cd ~/dockerlab/dockerfile
```

```
nano Dockerfile
```

Wir werden als Einstieg die Instruktionen ARG, FROM und RUN verwenden. Wichtig hierbei sind vor allem die zwei letzteren.

ARG	Mit der ARG Instruktion erstellen wir eine Dockerfile Variable und verwenden Sie in der nächsten Instruktion.
FROM	Definiert, welches Image aus der Registry geladen werden soll. Standardmässig wird die aktuellste Version vom Repository heruntergeladen. Soll eine spezifische Version verwendet werden, kann dies mit einem Doppelpunkt und dem entsprechenden Tag (Versionsnummer, Name) genauer spezifiziert werden.
RUN	Mit RUN können Befehle ausgeführt werden. Dabei sind alle Befehle erlaubt, die auch sonst in einem Terminal verwendet werden können. Standardmässig werden sämtliche Befehle über die ”/bin/sh -c” Shell ausgeführt. Wollen Sie eine andere Shell verwenden, muss diese explizit angegeben werden: ”/bin/bash” echo $HOME.

Hinweis

Achten Sie darauf, dass Befehle, wie zum Beispiel ”apt-get update” und ”apt-get install”, immer in einer einzelnen RUN Instruktion verknüpft werden:

```
RUN apt-get update && apt-get install -y curl ...
```

Sind die beiden Befehle oben in separaten RUN Instruktionen, wird der Layer mit der RUN Instruktion ”apt-get update” als komplett separiert angesehen. Wird nun ein Container geupdatet, sprich das Dockerfile modifiziert, werden nur die geänderten Zeilen im Dockerfile neu gebuildet. Die nicht geänderten Zeilen werden aus dem Cache gelesen. Somit kann es passieren, dass Pakete mit ”apt-get install” veraltet sind oder nicht installiert werden können. Dieses Problem tritt natürlich nur dann auf, wenn es bereits Layers im Cache hat – dies ist dann der Fall, wenn ein Rebuild des Images stattfindet.

Sie schreiben nun ein Dockerfile, welches auf dem aktuellen Alpine Image basiert. Mit der RUN Instruktion soll die Bibliothek aktualisiert werden.

Hinweis

Alpine ist ein auf Linux basierendes Betriebssystem. Durch die sichere und leichte Bauweise wird es gerne für Container verwendet. Zu beachten ist, dass nicht wie bei Debian und Ubuntu über ”apt-get install” Programme installiert werden, sondern über ”apk add”.

Folgenden Inhalt gilt es ins Dockerfile einzufüllen.

```
ARG CODE_VERSION=latest
FROM alpine:${CODE_VERSION}
RUN apk update
```

Aufgabe 3.1

Welche Sicherheitsproblematik kann beim Verwenden des folgenden Dockerfile entstehen und wie könnte man es besser machen ?

```
1 FROM ubuntu:18.04
2 RUN apt-get update
3 RUN apt-get install -y curl
```

update und install sollten in einem RUN command ausgeführt werden.
Ansonsten kann es vorkommen, dass bei einer neuen Version ein veränderter install command nochmal ausgeführt wird, der update command jedoch nicht. Somit kann es sein, dass alte Software Versionen geladen werden, oder das Paket nicht verfügbar ist.
-> RUN apt-get update && apt-get install -y curl

Speichern Sie die Datei ab und erstellen Sie aus dem Dockerfile ein Image. Da wir nicht wollen, dass das Image als ”alpine:latest” angezeigt wird, geben wir dem Image einen neuen Tag ”alpineimage”. Vergessen Sie bei dem Befehl den Punkt am Ende nicht. Dieser gibt den Ordner an, in welchem das Dockerfile enthalten ist (in diesem Fall **”~/dockerlab/dockerfile”**)

```
1 docker image build -t alpineimage .
```

Aufgabe 3.2

Von welcher Registry wurde das ”alpine:latest” Image heruntergeladen? Hint: Schauen Sie sich den Output des Befehls an.

docker.io/library/alpine:latest

Nach dem Absetzten des ”docker image build” Befehls sehen Sie die einzelnen Schritte (Step 1/3 etc.). Für dieses Dockerfile wurden drei Schritte ausgeführt, also genau so viele wie es Instruktionen im Dockerfile gibt.

Aus jedem dieser Schritte wird ein Layer erstellt und in einem Cache gespeichert. Wird das Dockerfile nochmals gebildet, nimmt Docker die nicht veränderten Teile aus dem Cache. Das beschleunigt den Build Prozess.

Speziell für die FROM und RUN Instruktion wird ein sogenannter ”Intermediate Container” erstellt. In diesem Container wird die Instruktion ausgeführt. Anschliessend entsteht eine Momentaufnahme, welche wiederum im Cache gespeichert wird. Nun sehen Sie, wieso es wichtig ist Update und Install immer zusammen durchzuführen.

```
Sending build context to Docker daemon  3.584kB
Step 1/3 : ARG CODE_VERSION=latest
Step 2/3 : FROM alpine:${CODE_VERSION}
latest: Pulling from library/alpine
a0d0a0d46f8b: Pull complete
Digest: sha256:e1c082e3d3c45cccac829840a25941e679c25d438cc8412c2fa221cf1a824e6a
Status: Downloaded newer image for alpine:latest
 ---> 14119a10abf4
Step 3/3 : RUN apk update
 ---> Running in 5c5912878259
fetch https://dl-cdn.alpinelinux.org/alpine/v3.14/main/x86_64/APKINDEX.tar.gz
fetch https://dl-cdn.alpinelinux.org/alpine/v3.14/community/x86_64/APKINDEX.tar.gz
v3.14.2-38-g27e4ada230 [https://dl-cdn.alpinelinux.org/alpine/v3.14/main]
v3.14.2-36-g70ff2140e8 [https://dl-cdn.alpinelinux.org/alpine/v3.14/community]
OK: 14938 distinct packages available
Removing intermediate container 5c5912878259
 ---> c3b318207ce6
Successfully built c3b318207ce6
Successfully tagged alpineimage:latest
```

Abbildung 4: Build Image Output

Öffnen Sie nun das Dockerfile im Ordner **"~/dockerlab/dockerfile/exercise"**. Nebst den RUN Instruktionen werden hier zwei neue Instruktionen eingeführt.

ENV	Diese Instruktion setzt Umgebungsvariablen im Image. Die Applikation im Container kann diese auslesen.
CMD	Wenn beim Starten eines Containers kein Befehl mitgegeben wird, verwendet der Container den im CMD angegeben Befehl. Wird beim Starten des Containers ein Befehl mitgegeben, wird der CMD-Befehl jedoch ignoriert.

Hinweis

Ein Container sollte grundsätzlich eine Anwendung ausführen. Somit erhöhen Sie die Wartbarkeit und reduzieren Komplexität eines Containers.

```
1  FROM alpine:latest
2
3  LABEL Creator: "HSLU"
4
5  RUN apk update && apk add curl bash
6  RUN mkdir /codes
7  RUN curl -o - https://jsonplaceholder.typicode.com/todos/1 | grep title > /
       codes/output.txt
8
9  ENV APACHE_RUN_USER="www-data"
10 ENV APACHE_RUN_GROUP="www-data"
11 ENV APACHE_LOG_DIR="/var/log/apache2"
12
13 CMD ["/bin/bash"]
```

Aufgabe 3.3

Erklären Sie, was das oben aufgeführte Dockerfile macht. Tipp: Mit curl kann eine URL aufgerufen werden. Der vertikale Strich "|" piped bzw. gibt weiter was der vorhergehende Befehl ausgegeben hat.

Updated die Repos, installiert curl und bash
Macht ein neues dir namens codes und legt dort ein output.txt file an, in welches das titel JSON - Feld von der Webseite reingeschrieben wird.
Erstellt einen Apache user und gruppe und stellt das Log directory für Apache auf /var/log/apache2 ein.
Danach wird eine bash shell gestartet.

Aufgabe 3.4

Worin unterscheiden sich die Befehle CMD, RUN, ENTRYPOINT? Wann würden Sie CMD und wann ENTRYPOINT verwenden? Machen Sie jeweils ein Beispiel.

RUN führt Befehle beim Erstellen des Images aus. -> RUN apk update
ENTRYPOINT legt die Hauptfunktion beim Starten fest. ENTRYPOINT["python", "app.py"]
CMD gibt optionale Argumente an. CMD["--help"]

Wichtig

Erstellen Sie wiederum aus diesem Dockerfile ein Image mit dem Tag "alpineimagemitenv" und vergewissern Sie sich, dass es unter den Images aufgeführt wird.
Achtung: Stellen Sie sicher, dass Sie das richtige Dockerfile builden!

Wir starten nun aus dem erstellten Image "alpineimagemitenv" einen Container.

```
1 docker container run -itd --name alpinecontainermitenv alpineimagemitenv
```

Der Container sollte in der Docker Prozessliste ersichtlich sein. Mit dem Parameter "-a" erhalten Sie nebst den aktiven Containern (Status Up) auch die gestoppten (Status STOPPED).

```
1 docker container ps -a
```

Sie werden feststellen, dass der durch das Dockerfile erstellte "/codes" Ordner nicht direkt von aussen ansprechbar ist.

```
1 ls /codes
```

Da wir uns im Container umsehen wollen, führen wir ein **neues** "bash" Terminal im Container aus und hängen uns an die CLI des Containers. Hierfür verwenden wir das Interactive und TTY Flag.

```
1 docker container exec -it alpinecontainermitenv bash
```

Verifizieren Sie, dass die Instruktionen des Dockerfiles im Container ausgeführt wurden. Verwenden Sie hierfür die untenstehenden Befehle.

```
1 echo $APACHE_RUN_USER
2 ls /codes
```

Aufgabe 3.5

Wie lautet der Inhalt der Datei output.txt im Ordner /codes ?

delectus aut autem

Führen Sie den ”tty” Befehl im Container aus. Dieser gibt die momentane Terminal Instanz in der Form ”/dev/pts/<Instanz Nr.>” aus.

```
tty
```

Aufgabe 3.6

Wieso wird nach der Eingabe des tty Befehls im Container nicht die erste Instanz ausgegeben? Hint: Die erste TTY Instanz ist immer 0 (/dev/pts/0). Jede weitere TTY Instanz wird entsprechend inkrementell nummeriert.

Die erste Instanz wird von Docker selber benutzt, als Interface zum Container. Diese wird benötigt, dass Docker mit dem Container interagieren kann.

Verlassen Sie nun den Container wieder.

```
exit
```

3.2. Docker Images und Containers im Detail

In diesem Kapitel untersuchen wir Images genauer. Auf Docker Hub können registrierte User selbstgemachte Images hochladen. Die Vorteile liegen in der zentralen Ablage und Verwaltung der Images. Hat man als Entwickler sich entschieden ein fremdes Image von dieser Registry herunterzuladen und daraus einen Container zu starten, kann das auch Probleme bereiten.

Aufgabe 3.7

Was für Probleme entstehen?

Wenn man Container installiert, weiss man auch nicht unbedingt, was darauf läuft. Eben so wie bei Software. Das kann zu Problemen führen.

Die Verhaltensweise dieser Images ist dem User nicht immer bekannt. Startet der Container immer wieder neu? Verwendet der Container Root Rechte? Werden Verbindungen zu externen Hosts gemacht?

Hinweis

Ende 2018 wurden bösartige Images vom Docker Hub entfernt, da diese zum Mining der Kryptowährung Monero verwendet wurden. Dabei wurde ein Script ausgeführt, sobald der Container mit dem ”run” Befehl gestartet wurde. Weitere Informationen: https://techcrunch.com/2018/06/15/tainted-crypto-mining-containers-pulled-from-docker-hub/

Wir wollen nun das alpine Image genauer betrachten. Folgender Befehl liefert ein JSON Objekt als Output. Der Output zeigt den Inhalt des Dockerfile sowie alle verwendeten Abhängigkeiten.

```
1 docker pull alpine:latest
2 docker image inspect alpine:latest
```

Mit dem ”--format” Flag lässt sich der Output weiter filtern.

```
1 docker image inspect --format "{{.RepoTags}} : {{.RepoDigests}} " alpine:
    latest
```

Aufgabe 3.8

Auf welcher Architektur und welchem OS basiert das Image?

Das Image basiert auf linux und auf der amd64 Architektur.

Information

Das alpine Image ist eine abgespeckte Version einer Linux Umgebung und wird vor allem in Container Technologien verwendet. Sie enthält die nötigsten Bibliotheken und Komponenten, um eine Applikation zu betreiben.

Docker Images können nach Belieben über die FROM Instruktion des Dockerfile mit bereits vorhandenen Images erweitert werden. Jedes dieser Images hat ein eigenes Dockerfile mit Instruktionen darin. Wir wollen nun einen Apache Webserver im Container ausführen. Passen Sie das Dockerfile gemäss folgendem Inhalt an.

```
FROM alpine:latest

LABEL Creator: "HSLU"

RUN apk update && apk add curl bash apache2
RUN mkdir /codes
RUN curl -o - https://jsonplaceholder.typicode.com/todos/1 | grep title > /
   codes/output.txt

ENV APACHE_RUN_USER="www-data"
ENV APACHE_RUN_GROUP="www-data"
ENV APACHE_LOG_DIR="/var/log/apache2"

EXPOSE 80

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

Anschliessend builden Sie dieses Image mit dem Tag ”apacheimage”.

```
docker build -t apacheimage .
```

Alle Images können auf ihre History untersucht werden. Die Ausgabe des history Befehls gibt dabei an, welche Instruktionen das Dockerfile des Images enthält. Führen Sie den history Befehl aus.

```
docker image history apacheimage
```

Aufgabe 3.9

Welche Einträge gehören zum Image apacheimage und nicht zum Base Image (FROM)?

RUN apk update && apk add curl bash apache2

EXPOSE 80

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

Stoppen Sie alle Container bevor Sie mit dem nächsten Kapitel weiterfahren.

```
docker stop alpinecontainermitenv web-server-nginx
```

3.3. Container Isolation

In diesem Kapitel lernen wir Container Isolation kennen. Container Isolation ist ein wichtiges Konzept der Container Architektur. Eine klare Trennung der Container ermöglicht es konfliktfrei neue Container zu generieren oder alte Container ohne weiteren ”footprint” abzuräumen.

Information

Docker verwendet viele Kernel Features, um fast optimale Isolation zu erreichen. Einige davon werden wir im Docker Security Versuch besser kennenlernen. Weitere Informationen: https://resources.infosecinstitute.com/topic/securing-containers-using-docker-isolation/

Anhand einer vom Labor-Team erstellten Web Applikation wollen wir die Isolation erforschen. Die Web Applikation verfügt über drei Funktionen, die mit dem passenden Aufruf ausgelöst werden können:

- ”/create” erstellt eine Datei in der Container Dateistruktur,
- ”/read” gibt den Inhalt der Datei auf der Webseite aus
- ”/delete” löscht die Datei wieder.

Wechseln Sie in den Ordner **”~/dockerlab/dockerweb”** und schauen Sie die Dateien an.

app.js	Ist unser Server. Die Datei instanziiert einen Express Webserver, welcher Verbindungen auf Port 3000 akzeptiert. Die auf dem Webserver laufende App ermöglicht uns, via HTTP Aufruf Dateien in der Linux Dateistruktur zu erstellen.
package.json	In dieser Datei werden alle Abhängigkeiten unserer Web Applikation (Imports) definiert. Achtung: hier geht es um Abhängigkeiten der Web Applikation! Das hat mit Docker direkt nichts zu tun.
Dockerfile	Schon bekannt aus den letzten Kapiteln.

Öffnen Sie das Dockerfile. Wir lernen nun zwei neue Instruktionen kennen:

- WORKDIR ist eine relative Pfadangabe im Container selbst. Ist der angegebene Pfad nicht vorhanden wird dieser erstellt. Der Pfad des WORKDIR wird für die darauffolgenden Dockerfile Instruktion verwendet.
- COPY. Kopiert Dateien und Ordner von einem Ort zum anderen. Falls WORKDIR nicht angegeben wurde, wird das oberste Verzeichnis ”/” als Ausgangspunkt verwendet.

Erstellen Sie das Image und starten Sie den Container.

```
docker image build -t dockerisolation .
docker container run -d --name intro1 dockerisolation
```

Wir verwenden ”curl”, um eine HTTP GET Anfrage zu senden.

```
curl <HOST>:9000
```

Aufgabe 3.10

Was ist das Resultat und wieso?

Wir haben kein Zugriff, da beim Starten kein Port geöffnet wurde mit -p.
Somit ist der Container isoliert und nimmt die Anfragen auf den Port 9000 nicht entgegen.

Container sind mittels eines Host-Only-Netzwerks mit dem Host verbunden. Innerhalb dieses Netzwerks ist der Port erreichbar, sofern der Parameter ”-p <port>” angegeben wurde. Beenden und löschen Sie den Container bevor Sie ihn anschliessend mit den Portangaben erneut starten.

```
docker container kill intro1
docker container rm intro1
docker container run -d -p 9000:3000 --name intro1 dockerisolation
```

Welcher der beiden Ports kann nun in welchem Kontext angesprochen werden? Ordnen Sie unten zu! Hint: Konsultieren Sie nochmals Kapitel 2.

Aufgabe 3.11

Hostsystem:

Das Hostsystem kann Port 9000 ansprechen.

Aufgabe 3.12

Container:

Der Container hat intern Port 3000 offen.

Wir starten einen zweiten Container mit dem gleichen Image.

```
1 docker container run -d -p 9001:3000 --name intro2 dockerisolation
```

Kontrollieren Sie mit dem Browser, dass beide Webseiten auf Port 9000 und Port 9001 erreichbar sind.

Öffnen Sie die Webseite <HOST>:9000/create im Browser. Das erstellt eine Datei namens ”page.html” mit dem Inhalt ”Datei Inhalt”.

Aufgabe 3.13

Was passiert, wenn Sie <HOST>:9001/read aufrufen? Sehen Sie den Inhalt der zuvor erstellten Datei? **Begründen Sie!**

Nein, der Container, der 9001 offen hat hat ein isoliertes Dateisystem, welches vom Container auf 9000 nicht abgeändert wird. Somit wird die Datei die auf 9000 erstellt ist nicht auf 9001 existieren und wir können sie nicht finden.

Hinweis

Mit ”docker container exec <id | name> **<BEFEHL>**” kann man Befehle direkt im Container absetzen.
Zum Beispiel ”docker container exec intro2 **ls**”

Natürlich können auch die besten Docker Isolationsmechanismen mit falscher Konfiguration oder veralteten Versionen von Angreifern ausser Kraft gesetzt werden. Durch falsche Konfigurationen können Container sich gegenseitig aushungern oder Host Ressourcen schlecht ausnutzen. Aber auch zu viele Berechtigungen, nicht gebrauchte offene Ports und vieles mehr kann die Container Isolierung schwächen und das Host System potenziell gefährden. Im zweiten Versuch zu Docker Security werden Sie mehr über diese Thematik erfahren.

3.4. Aufräumen

Container können im Status STOPPED, EXITED oder UP sein. Damit ein Container abgeräumt werden kann muss er zuerst STOPPED oder EXITED sein. Wir wollen kurz auf die einzelnen Schritte des nachfolgenden Befehls eingehen.

Mit "$(docker container ls -aq)" führen wir alle Container IDs auf. Das Konstrukt "$(…)" bewirkt, dass der Befehl in den Klammern zuerst ausgeführt wird. Damit kann das Resultat der Container Auflistung an "docker container stop" als Parameter übergeben werden. Über "docker container stop" werden alle Container in den STOPPED oder EXITED Status versetzt.

```
1 docker container stop $(docker container ls -aq)
```

Nun können Sie alles abräumen. In der Entwicklung/Test Phase kann es vorkommen, dass viele Container erstellt werden. Diese einzeln abzuräumen ist aufwendig deshalb bietet Docker hierfür den Befehl "prune".

--all	Es werden alle nicht verwendeten Images entfernt (da alle Container gestoppt sind, sind auch keine Images mehr in Verwendung).
--force	Wir wollen nicht warten bis die laufenden Container herunterfahren.
--volumes	Alle erstellten Dateianbindungen sollen abgeräumt werden. Was Volumes sind, wird später in Kapitel 5 behandelt.

```
1 docker system prune --all --force --volumes
```

4. Docker Networks

Es macht wenig Sinn, mehrere Applikationen in einem Container laufen zu lassen. Das wäre völlig entgegen der Vorteile von Container Technologien und man könnte direkt VMs verwenden.

Wenn mehrere Applikationen/Container miteinander kommunizieren müssen (zum Beispiel Webserver und Datenbank), braucht es zusätzliche Konfigurationen.

Docker bietet Netzwerktreiber an, um Container untereinander oder Container mit dem Host zu verknüpfen. Da wir nie einen Treiber angegeben haben, wurden bislang alle unsere Docker Container mit dem Netzwerktreiber ”bridge” gestartet. Sie werden in diesem Kapitel diesen Treiber kennenlernen.

Container können mehrere Endpunkte besitzen. Ein Endpunkt ist ein virtuelles Interface des Containers. Endpunkte werden über Container Networks miteinander verbunden. Container Networks werden von der Docker Engine verwaltet.

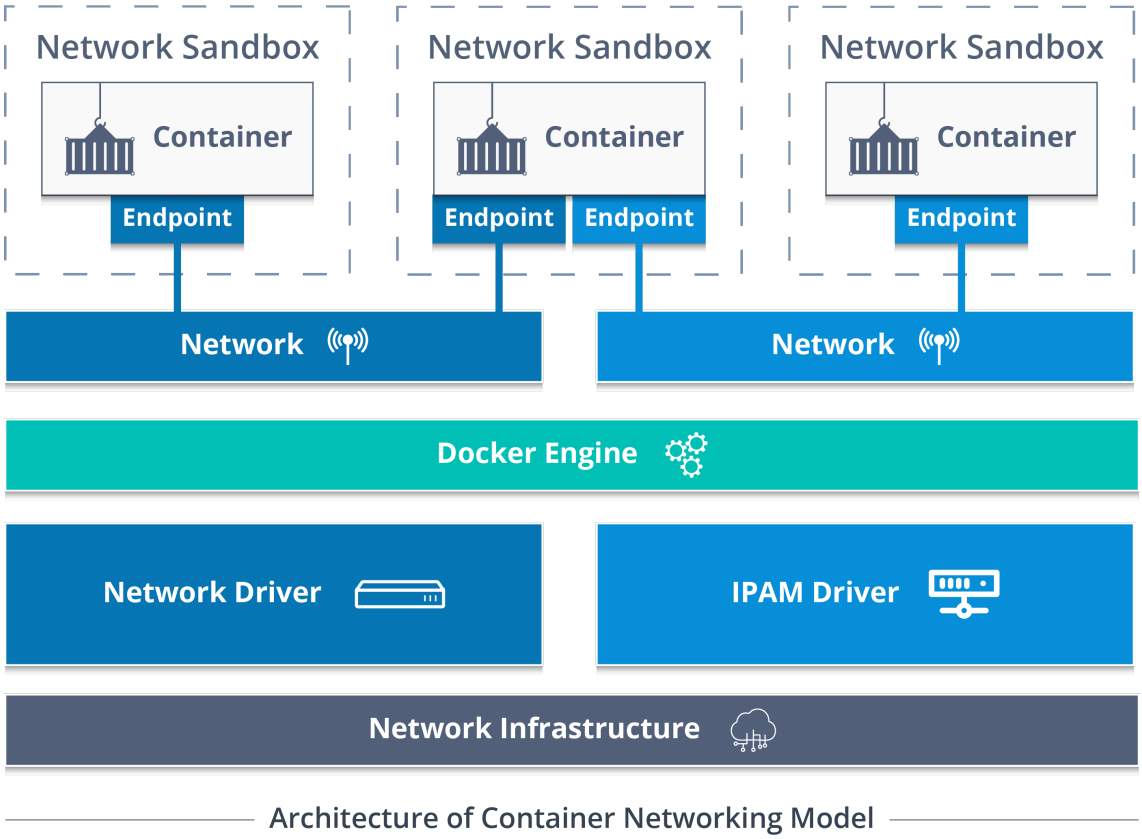


Abbildung 5: Container Networking Model Architektur (https://www.edureka.co/blog/docker-networking/)

In diesem Kapitel werden Sie die unterschiedlichen Docker Netzwerk Treiber untersuchen.

4.1. Bridge Netzwerktreiber

Docker erstellt auf dem Host ein virtuelles Interface ”docker0”. Ein Container kann durch sein veth Interface (ein virtueller Ethernet Link) über ”docker0” mit anderen Containern oder dem Host kommunizieren.

Information

Kommunikation vom Container ins Internet wird über die iptables NAT geregelt. Genaueres zum Thema der iptables NAT von Docker erfahren Sie hier: https://medium.com/@havloujian.joachim/advanced-docker-networking-outgoing-ip-921fc3090b09

Wenn Sie die Netzwerkinterfaces des Hosts auflisten, sehen Sie ”docker0” aufgeführt. Da noch keine Container gestartet wurden sind keine ”veth” Interfaces ersichtlich.

```
1 ip addr show
```

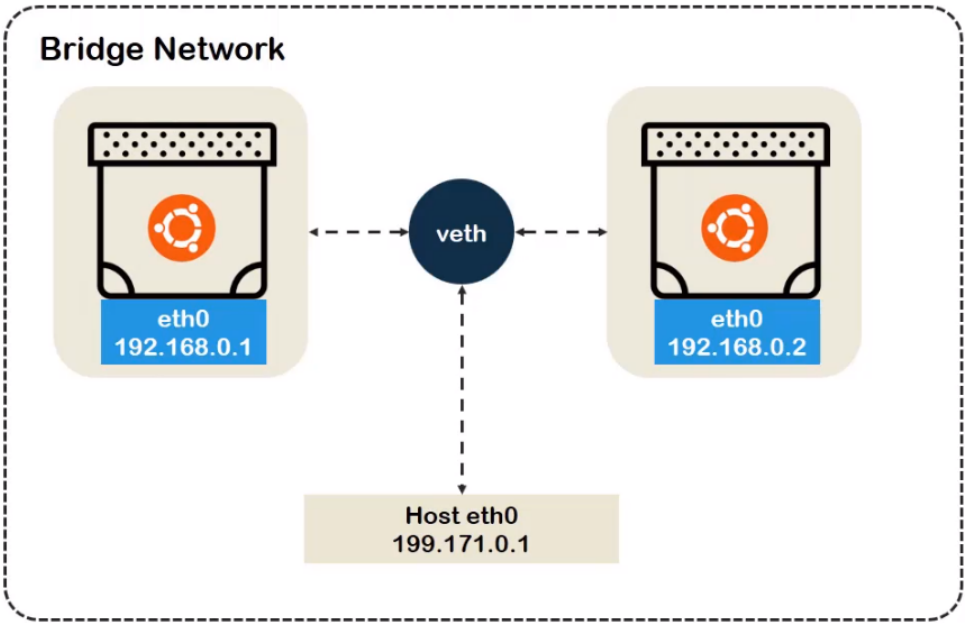


Abbildung 6: Bridge Netzwerk

Der für diese Verbindung verwendete Treiber heisst ”bridge”. Diese Einstellung ist gut während der Entwicklungsphase aber sollte vermieden werden, wenn die Applikation produktiv genutzt wird. Einer der Gründe ist, dass über das Standard ”bridge” Netzwerk keine Namensauflösung stattfindet. Das heisst Container können andere Container nur über Ihre IP ansprechen. Um dem entgegenzuwirken, bietet Docker an, eigene bridge Netzwerke zu erstellen.

Führen Sie alle Docker Netzwerke auf. Das Standard ”bridge” Netzwerk sollte nebst den Netzwerken ”host” und ”none” ersichtlich sein.

```
1 docker network ls
```

Starten Sie nun zwei Container, welche die alpine Shell ausführen. Da wir kein Netzwerk definieren, wird standardmässig das ”bridge” Netzwerk verwendet. Vergewissern Sie sich, dass beide Container gestartet sind.

```
1 docker container run -itd --name alpine1 alpine sh
2 docker container run -itd --name alpine2 alpine sh
```

Inspizieren Sie nun das ”bridge” Netzwerk.

```
1 docker network inspect bridge
```

Aufgabe 4.1

Sind die zuvor erstellten Container aufgeführt? Falls ja, wie lauten die IPv4 Adressen? Was vermuten Sie wird nun bei den Host Netzwerk Interfaces angezeigt?
Hint: ”ip addr show”

Ja: alpine1: 172.17.0.2; alpine2: 172.17.0.3

docker0, und die beiden virtuellen Interfaces der Docker Container, die gestartet sind.

Öffnen Sie nun das CLI von ”alpine1” und listen Sie die Netzwerk Interfaces im Container auf. Mit dem Befehl ”docker attach” können Sie auf den mit CMD oder ENTRYPOINT angegebenen Prozess zugreifen.

```
1 docker attach alpine1
2 ip addr show
```

Aufgabe 4.2

Welche Interfaces sind aufgeführt? Geben Sie die IPv4 Adresse und den Namen an.

link/loopback: 127.0.0.1/8
eth0@if27: 172.17.0.2/16

Versuchen Sie einen Ping im Container auf folgende Adressen auszuführen. Was ist das Resultat? Funktionieren diese, begründen Sie!

Hint: Die IP-Adresse konnten Sie über das docker network inspect herausfinden.

Aufgabe 4.3

Google.ch:

Google ist erreichbar.

Aufgabe 4.4

alpine2:

alpine2 ist nicht erreichbar: bad address

Aufgabe 4.5

IP von alpine2:

die IP von alpine 2 ist erreichbar

Falls Sie noch in der Container CLI von ”alpine1” sind, verlassen Sie diese. Da wir uns in den Container über attach angehängt haben, wird dieser nun in den EXITED Status gesetzt. Der EXITED Status heisst in diesem Fall, dass der laufende Prozess (also ”/bin/sh”) beendet wurde und somit kein Prozess mehr im Container läuft.

```
exit
```

Bevor wir weiterfahren, sollten die erstellten Container abgeräumt werden.

```
docker container stop alpine1 alpine2
docker container rm alpine1 alpine2
```

Wir wollen nun unser eigenes bridge Netzwerk erstellen.

```
docker network create --driver=bridge meinebridge
```

Es ist ein neuer Eintrag mit dem erstellten Netzwerk ersichtlich.

```
docker network ls
```

Starten Sie nun zwei Container im ”meinebridge” Netzwerk, einen im normalen ”bridge” und einen in beiden (alpine4).

```
docker container run -itd --name alpine1 --network meinebridge alpine sh
docker container run -itd --name alpine2 --network meinebridge alpine sh
docker container run -itd --name alpine3 alpine sh
docker container run -itd --name alpine4 --network meinebridge alpine sh
docker network connect bridge alpine4
```

Bevor Sie fortfahren, geben Sie die IP-Adressen der Container aus. Sie werden diese im Anschluss benötigten.

```
docker inspect -f '{{.Name}} = {{range .NetworkSettings.Networks}} {{.
    IPAddress}}{{end}}' $(docker container ps -aq)
```

Öffnen Sie das CLI von ”alpine2” und pingen Sie die anderen Container an.

Aufgabe 4.6

Sind alpine1, alpine3 und alpine4 erreichbar über den Namen? Wie sieht es aus mit pingen über die IP-Adresse? Was können Sie vom Container ”alpine3” pingen?

alpine1 und 4 sind über die namen erreichbar, alpine3 hingegen nicht.

mit der IP ist 1 erreichbar, 3 und 4 nicht

vom container alpine 3 können wir die anderen Container nicht pingen

Wie nun klar ersichtlich war, können Container durch manuell erstellte Netzwerke separiert werden. Zudem können innerhalb eines solchen Netzwerks Verbindungen über den Container Namen aufgebaut werden.

Räumen Sie alle Container und Netzwerke ab.

```
1 docker container stop alpine1 alpine2 alpine3 alpine4
2 docker container rm alpine1 alpine2 alpine3 alpine4
3 docker network rm meinebridge
```

4.2. Host Netzwerktreiber

Beim Host Netzwerk wird der Container direkt an den Host gebunden. Somit läuft die Applikation, als ob Sie direkt auf dem Host gestartet wurde. Der Container ist jedoch in den restlichen belangen (Storage, Prozess Namespaces, User Namespaces) vom Host isoliert.

Hinweis

Prozess und User Namespaces werden wir in der nächsten Laborübung betrachten.

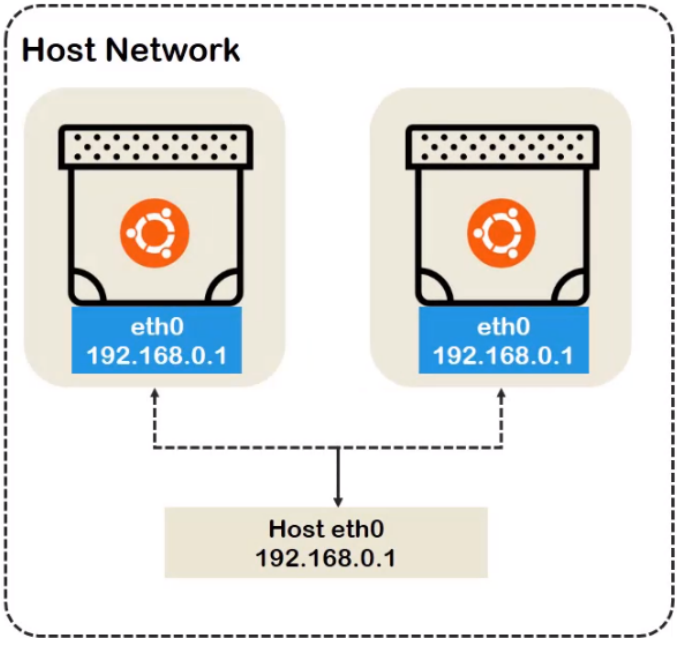


Abbildung 7: Host Netzwerk

Ein nginx Web Server Container mit Host Netzwerk kann mit folgendem Befehl erstellt werden. Es kann anschliessend

über `http://localhost` die Willkommens Seite von nginx erreicht werden.

```
1 docker container run -d --network host --name hostnginx nginx
```

Beim Aufführen aller Netzwerk Interfaces ist lediglich das docker0 bridge Netzwerk ersichtlich. Im Gegensatz zum bridge Netzwerk, wird beim Host Netzwerk kein neues ”veth” Interface erstellt.

```
1 ip addr show
```

Aufgabe 4.7

Welcher Prozess ist angebunden an den Port 80?
Hint: Mit dem Befehl `sudo netstat -lpn | grep :80` können Sie nach aktiven Verbindungen auf Port 80 suchen.

nginx ist erreichbar, über IPv4 und IPv6.

Stoppen Sie den Container und entfernen Sie ihn.

```
1 docker container stop hostnginx && docker container rm hostnginx
```

Hinweis

Es gibt 2 weitere Netzwerktypen, die hier nicht genauer betrachtet werden.

Overlay	Wird bei verteilten Systemen verwendet, in denen die Kommunikation Host-übergreifend stattfindet. Das Overlay Netzwerk findet bei Docker Swarm Verwendung.
None	Wenn der Container hinsichtlich des Netzwerks vollkommen abgekapselt werden soll.

5. Docker Storage

Docker erstellt standardmässig für jeden gestarteten Container ein temporäres Dateisystem, welches nur im entsprechenden Container selbst besteht. Dieses wird abgeräumt, sobald der Container zerstört wird. Wenn zum Beispiel ein Container Daten wiederverwenden will, sollten diese Daten auf dem Host persistent gespeichert werden. Somit gehen zum Beispiel bei einem Absturz die Daten nicht verloren.

Docker bietet drei Arten von Datenspeicherungslösungen an.

Volumes	Auf Volumes kann nur durch Docker Prozesse zugegriffen werden. Somit haben User ausserhalb der Gruppen docker und root keinen Zugriff. Sie befinden sich im Docker Ordner **"/var/lib/docker/volumes"**.
Bind Mount	Das Anbinden eines beliebigen Ordners an den Container. Diese Ordner sind über normale Dateisystempfade auf dem Host zugänglich. Die darin enthaltenen Dateien können, auch nachdem der Container beendet wurde, eingesehen werden.
tmpfs Mount	Dieser Mount ist während der Laufzeit eines Containers über den Host zugänglich wird jedoch nach dem Beenden des Containers wieder abgeräumt. Somit sind alle Daten danach verloren. tmpfs gehört somit nicht zu den persistenten Optionen. Wenn jedoch Daten schnell geschrieben werden müssen, sind Schreiboperationen in Dateien zu langsam. tmpfs schreibt direkt in den temporären Zwischenspeicher (Memory) und ist so schneller.

Information

Einen weiteren Anwendungsbereich findet tmpfs Mount in der Speicherung von sensitiven Daten: https://docs.docker.com/engine/swarm/secrets/

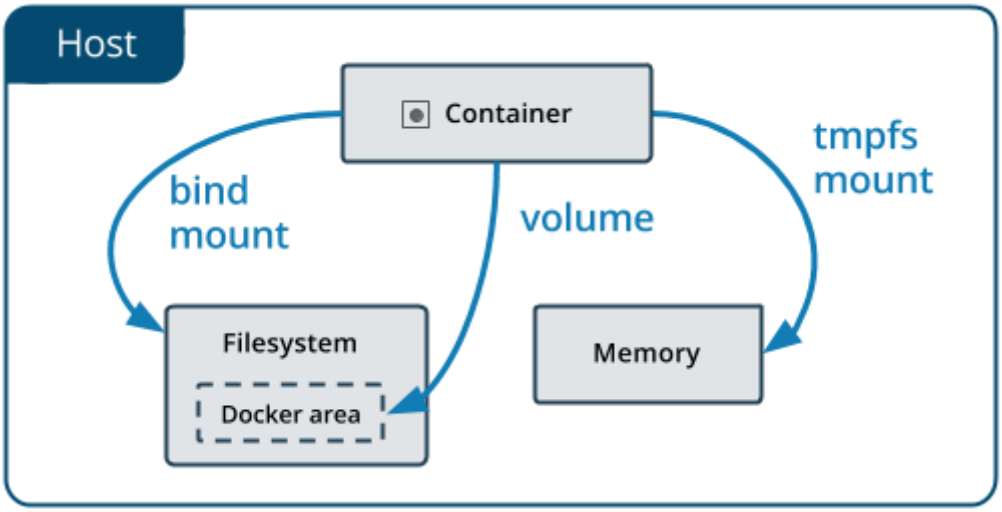


Abbildung 8: Datenspeicherung (https://docs.docker.com/storage/)

Im nächsten Kapitel lernen Sie, wie ein Volume erstellt und eingebunden werden kann.

5.1. Erstellen eines Volumes

Mit dem Flag ”--mount” kann durch Angabe des Typs ”type=volume” ein Volume eingebunden werden. Mit ”src” können Sie einen expliziten Namen für das Volume und mit ”dst” den Ordner im Container mitgeben.

```
1 docker container run -d --mount type=volume,src=myvolume,dst=/tmp alpine
```

Listen Sie anschliessend alle Volumes auf. Sie sollten das erstellte Volume sehen.

```
1 docker volume ls
```

Aufgabe 5.1

Wo befindet sich das erstellte Volume ”myvolume” auf dem host System?
Hint: Wir haben gelernt, dass es ein inspect Keyword für Images gibt. Das gleiche Keyword existiert auch für Volumes.

/var/lib/docker/volumes

Starten Sie erneut einen Container. Der Container soll nur kurz starten und direkt wieder abgeräumt werden. Dabei bleibt das Volume mit dem Namen ”danglingvolume” übrig.

```
1 docker container run --rm --mount type=volume,src=danglingvolume,dst=/tmp
    alpine
```

Mit dem filter Flag ”dangling=true” könne Sie alle Volumes anzeigen, die nicht mehr an einen Container angebunden sind (eben dangling). Das Volume können Sie auf Ihrem System belassen. In der nächsten Übung werden wir die Konsequenzen dieser Dangling Volumes kennenlernen.

```
1 docker volume ls --filter "dangling=true"
```

5.2. Bind Mount Host Ordner

Bind Mounts sind für Entwicklungsphasen eine gute Alternative zu Volumes. Sie lassen sich mit dem ”docker container run” Befehl als Parameter mitgeben. Bind Mounts werden nicht von Docker verwaltet. Das heisst Sie können zum Beispiel nicht sehen, welche Container denselben bind Mount verwenden. Sie ermöglichen jedoch Container Ordner mit Host Ordner zu überschreiben. Eine Möglichkeit wäre das Überschreiben einer Standardkonfiguration, um verschiedene Verhalten des Containers zu erreichen.

Im folgenden Beispiel erstellen Sie eine Datei und binden diese an den Container. Erstellen Sie im Ordner **”~/dockerlab”** eine Textdatei.

```
1 echo "hello world" > ~/dockerlab/bindmount.txt
```

Mit dem folgenden Befehl erstellen Sie einen Container und binden den Ordner **”~/dockerlab”** ein. Im Container wird dieser Ordner als **”/app”** zu finden sein.

```
1 docker container run -dit --name bindmount --mount type=bind,source=/home/
    labadmin/dockerlab,target=/app nginx:latest
```

Wenn Sie den Ordner im Container auflisten, sehen Sie den Inhalt von **"~/dockerlab"**.

```
1 docker container exec bindmount ls -la /app
```

Löschen Sie nun die Textdatei.

```
1 docker container exec bindmount rm /app/bindmount.txt
```

Wenn Sie auf dem Host nochmals ein Datei Listing machen, werden Sie sehen, dass die Textdatei nicht mehr vorhanden ist.

Stoppen Sie alle Container und Räumen Sie mit dem prune Befehl alles auf, bevor Sie mit dem nächsten Kapitel weiterfahren.

```
1 docker system prune --all --volumes
```

6. Docker Compose

Compose ist ein Tool, um komplexe Applikationen zu erstellen/verwalten. Applikationen, die in einem Stück geschrieben sind (Monolithen), können schnell unübersichtlich und schwer wartbar werden. Um dem entgegenzuwirken, wird schon in der Architektur-Phase in kleinere Services aufgeteilt. Dieses Konzept wird Microservices Architecture genannt. Im Rahmen von Microservices werden Docker Container Services genannt.

Die Services können einzeln gestartet oder sogar im laufenden Betrieb ausgetauscht werden. Das ermöglicht den Applikationen ihren Anforderungen entsprechend zu skalieren, ohne ein gesamtes System herunterzufahren.

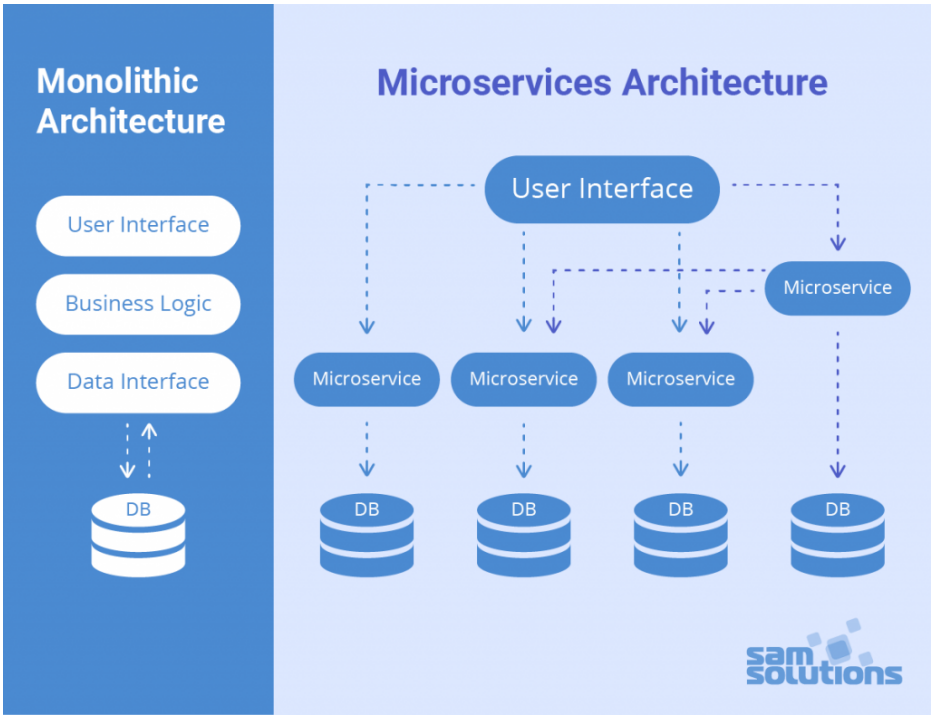


Abbildung 9: Monolith vs. Microservices
(https://www.sam-solutions.com/blog/microservices-vs-monolithic-real-business-examples/)

6.1. Die Compose Datei

Die Compose Datei kann mehrere Container mit einem Befehl starten. Damit muss nicht jeder Container von Hand einzeln erstellt werden. Wir betrachten die einzelnen Services anhand des Beispiels im Ordner **"~/dockerlab/dockercompose"**. Die Compose Datei ist im YAML Format verfasst.

```
1 cd ~/dockerlab/dockercompose
2 cat docker-compose.yaml
```

Wichtig

Die Einrückungen sind in einem yaml wichtig! Weitere Informationen (Schreibweisen, Regeln) finden Sie hier: https://vsupalov.com/yaml-docker-compose-kubernetes/

Sie haben das **"docker-compose.yaml"** unten nochmals komplett aufgeführt. Sie werden die darin enthaltenen Elemente in diesem Kapitel kennenlernen.

```yaml
version: '3.3'
services:
  db:
    image: mysql:5.7
    container_name: mysql_database
    volumes:
      - db_data:/var/lib/mysql
    restart: always
    networks:
      - custom_net
    environment:
      MYSQL_ROOT_PASSWORD: word@press
      MYSQL_DATABASE: wordpress
      MYSQL_USER: wordpress
      MYSQL_PASSWORD: abc@123

  wordpress:
    depends_on:
      - db
    image: wordpress:latest
    container_name: wp_frontend
    volumes:
      - wordpress_files:/var/www/html
    networks:
      - custom_net
    ports:
      - "8000:80"
    restart: always
    environment:
      WORDPRESS_DB_HOST: db:3306
      WORDPRESS_DB_USER: wordpress
      WORDPRESS_DB_PASSWORD: abc@123
      WORDPRESS_DB_NAME: wordpress
volumes:
  wordpress_files:
  db_data:
networks:
  custom_net:
    driver: bridge
```

Information

Der erste Eintrag ”version” weist darauf hin, welche Docker Compose Version verwendet werden soll. Die gewählte Version ist mit entsprechenden Docker Engine Versionen kompatibel. Damit hat die Compose Version auch Einfluss, welche Compose Befehle und Keywords verfügbar sind. Weitere Informationen hier: https://docs.docker.com/compose/compose-file/compose-versioning/

In unserer Docker Compose Datei sind zwei Services definiert: ”db” und ”wordpress”.

```
db:
 image: mysql:5.7
 container_name: mysql_database
 restart: always
```

Der ”db” Service ist unsere Datenbank und basiert auf einem MySQL Image. Der ”db” Service wird immer neu gestartet, sobald der Service unerwartet beendet wurde.

```
volumes:
 - db_data:/var/lib/mysql
```

Damit MySQL die Daten speichern kann, wird das Volume ”db_data” für den Service definiert. Das Volume soll auf ”/var/lib/mysql” im Container zeigen. Weitere Einträge können einfach mit einem neuen Bindestrich unten aufgeführt werden.

```
networks:
 - custom_net
```

Der Service soll nicht im Standard Netzwerk sondern in einem User-definierten sein.

```
environment:
 MYSQL_ROOT_PASSWORD: word@press
 MYSQL_DATABASE: wordpress
 MYSQL_USER: wordpress
 MYSQL_PASSWORD: abc@123
```

Da wir MySQL nicht mit den Standardwerten starten wollen, definieren wir die notwendigen Umgebungsvariabeln mit den drei ”environment” Einträgen.

Aufgabe 6.1

In welchem Ordner des Host Systems wird sich das Volume db_data befinden?
Hint: Machen Sie eine Annahme mit Hilfe von Kapitel 5.

/var/lib/docker/volumes

```
wordpress:
 networks:
  - custom_net
```

Der ”wordpress” Service ist die eigentliche Webseite.

```
ports:
 - "8000:80"
```

Die Webseite soll vom Host über den Port 8000 erreichbar sein.

```
depends_on:
 - db
```

Mit ”depends_on” wartet der ”wordpress” Service, bis der ”db” Container gestartet ist.

Networks und Volumes müssen zusätzlich ausserhalb der Services definiert werden. Da es sich dabei um User-

defined Networks und Volumes handelt, bedarf es dieser Einträge. Sie können das mit den in Kapitel 4.1 und Kapitel 5.1 verwendeten Befehlen vergleichen. Da muss zuvor ebenfalls das Network/Volume erstellt werden, bevor es verwendet werden kann.

```
volumes:
  wordpress_files:
  db_data:
networks:
  custom_net:
    driver: bridge
```

Starten Sie nun alle Services mit dem ”docker compose up” Befehl. Dieser buildet alle Services und startet diese anschliessend. Wie auch beim ”docker” Befehl, startet der Parameter -d die Container im Hintergrund.

```
docker compose up -d
```

Sehen Sie sich die laufenden Container Liste an und kontrollieren Sie mittels Webbrowser, ob die Wordpress Seite verfügbar ist.

Wichtig

Richten Sie die Wordpress Seite ein und erstellen Sie einen kurzen Beitrag. Das Zusammenspiel zwischen Datenbank und Wordpress sollte somit ersichtlich sein, denn das Wordpress Framework braucht eine Datenbank, um zu funktionieren. Verlieren Sie hierbei nicht zu viel Zeit!

Docker Compose verwendet fast die gleichen Befehle, wie zum Beispiel ”docker container run”. Speziell von Interesse kann jedoch das Logging sein (sofern das Compose als Daemon/Hintergrund Prozess mit -d läuft, andernfalls ist der Log direkt ersichtlich). Mit dem ”--tail” Flag kann der Inhalt auf die letzten paar Einträge beschränkt werden.

```
docker compose logs --tail=10
```

Prozesse eines Compose können ebenfalls im Detail betrachtet werden.

```
docker compose top
```

Wir sind nun fertig mit unserem Compose und wollen sämtliche laufenden Services des Compose herunterfahren.

```
docker compose down
```

7. Jetzt sind Sie dran – Ihre eigene Docker App

Ihr Vorgesetzter hat Ihnen den Auftrag erteilt, alle Mitarbeiter auf einer Webseite mit Alter und Name aufzuführen. Den Inhalt der Webseite hat er selbst erstellt und Ihnen im Ordner **”~/dockerlab/agechecker”** abgelegt. Dabei ist es ihm wichtig, dass die Webseite in 3 Applikationen (Services) aufgeteilt wird. Ihr Vorgesetzter ist schon recht weit gekommen und die Basis für die Webseite steht. Leider hat er keinen Plan bezüglich Docker. Sie sollen hier der Experte sein.

Der Service frontend hält alle notwendigen Dateien im Ordner **”~/dockerlab/agechecker/frontend”**. Nebst den Webseiten Dateien ist auch ein Dockerfile enthalten. Dieses scheint aber nicht vollständig zu sein. Ergänzen Sie die fehlenden Stellen und bedenken Sie, dass Ihr Vorgesetzter die Webseite über Port 4200 erreichen will.

Der Service backend hält alle notwendigen Dateien im Ordner **”~/dockerlab/agechecker/backend”**. Auch hier ist wieder ein unvollständiges Dockerfile das ergänzt werden muss. **Ihr Vorgesetzter hat Sie darauf hingewiesen, dass das Backend die Datenbank über den Namen ”database” anspricht**. Ein anderer Name für die Datenbank wird nicht funktionieren. Das Frontend soll später das Backend über Port 3000 ansprechen können. Beim Backend ist zu beachten, dass es erst starten soll, wenn die Datenbank läuft, um Fehler zu vermeiden.

Nun bleibt noch der letzte Service **database**. Ihr Vorgesetzer hat diesbezüglich nichts vorbereitet. Da Sie sich nicht die Mühe machen wollen ein Dockerfile zu schreiben, verwenden Sie das Image mongo im Docker Compose. Das von Ihrem Vorgesetzten vorbereitete Backend ist nicht mit dem aktuellen Major Release von MongoDB kompatibel. Verwenden Sie deshalb eine Version von MongoDB aus dem 5.x Major Release. Die zu verwendende Version kann als Tag angegeben werden. Der Tag wird nach einem Doppelpunkt an den Namen des Images angehängt (z.B. mongo:5.0.13). Wurde kein Tag festgelegt, verwendet Docker Engine den Tag «:latest». Es bedarf bei diesem Service keiner Port Angabe, da wir auf die Datenbank nicht von aussen zugreifen müssen. Diesen müssen Sie also später im Docker Compose definieren.

Wichtig

Sie haben im Kapitel 6 über das Keyword ”image” definiert, was für die Services verwendet werden soll. Das setzt voraus, dass dieses Image aus einem Dockerfile erstellt oder von Docker Hub heruntergeladen worden ist. Sie können jedoch auch anstelle von ”image” das Keyword ”build: <Ordner wo das Dockerfile ist>” angeben (zum Beispiel ”build: backend”).
Das Docker Compose muss dann folgendermassen gestartet werden, um die Dockerfiles zu builden. Der Build Prozess kann etwas dauern:

```
1 docker compose up --build -d
```

Sie können also nun mit dem **”~/dockerlab/agechecker/docker-compose.yml”** beginnen. Definieren Sie darin die drei Services. Die Ports sollen immer auf den gleichen Port gemappt werden wie auch im Container verwendet werden.

Aufgabe 7.1

Was ist notwendig damit die Datenbank mit dem Backend und das Backend mit dem Frontend kommunizieren kann? Passen Sie entsprechend Ihr ”docker-compose.yml” an.

Wir brauchen ein Netzwerk, auf welchem alle 3 services sind, damit sie miteinander kommunizieren können.

Bevor Sie das Resultat ihrem Vorgesetzten vorzeigen, wollen Sie alles einmal starten und im **Webbrowser** testen.

Aufgabe 7.2

Welche Mitarbeitenden sind bereits eingetragen und wie alt sind sie?

Lina Morris, Age 30

Wenn Sie fertig mit der Übung sind, entfernen Sie das Compose.

```
1 docker compose down
```

Notizen