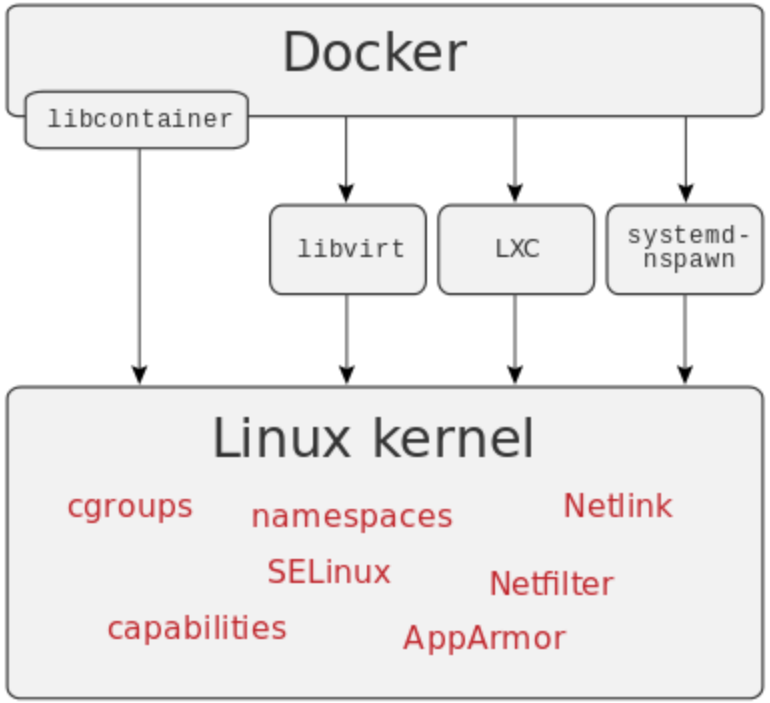
Docker



Quelle: Wikipedia: Docker

Docker

Docker dient zur Isolierung von Anwendungen mit Hilfe von sogenannter *Containervirtualisierung*. *Containervirtualisierung* ist eine Methode um mehrere Instanzen eines Betriebssystems isoliert voneinander den Kernel eines Hostsystems nutzen zu lassen. Im Gegensatz zur Virtualisierung mittels eines Hypervisors hat *Containervirtualisierung* zwar einige Einschränkungen, gilt aber als besonders ressourcenschonend.

Quellen

Wikipedia: Docker

Wikipedia: Containervirtualisierung

Docker: Begrifflichkeiten

Image

Ein Speicherabbild eines Containers. Das Image selbst besteht aus mehreren *Layern*, die schreibgeschützt sind und somit nicht verändert werden können. Ein *Image* ist portabel, kann in Repositories gespeichert und mit anderen Nutzern geteilt werden. Aus einem *Image* können immer mehrere *Container* gestartet werden.

Quelle

Wikipedia: Docker

Container

Als *Container* wird die aktive Instanz eines _Image_s bezeichnet. Der Container wird also gerade ausgeführt und ist beschäftigt. Sobald der *Container* kein Programm ausführt oder mit seinem Auftrag fertig ist, wird der *Container* automatisch beendet.

Quelle

Wikipedia: Docker

Layer

Ein *Layer* ist Teil eines _Image_s und enthält einen Befehl oder eine Datei, die dem *Image* hinzugefügt wurde. Anhand der *Layer* kann die ganze Historie des _Image_s nachvollzogen werden.

Quelle

Wikipedia: Docker

Dockerfile

Eine Textdatei, die mit verschiedenen Befehlen ein *Image* beschreibt. Diese werden bei der Ausführung abgearbeitet und für jeden Befehl ein einzelner *Layer* angelegt.

Quelle

Wikipedia: Docker

Fragen? Kurze Pause?

Als nächstes: **docker-compose**

docker-compose

docker-compose ist ein Werkzeug zur Definition und Ausführung von Multi-Container-Docker-Anwendungen. *docker-compose* verwendet eine YAML-Datei, um die Dienste der Anwendung zu konfigurieren. Mit einem einzigen Befehl können anschließend alle Anwendungen gebaut und gestartet werden.

Quelle

https://docs.docker.com/

docker-compose

Die Verwendung von *docker-compose* ist ein dreistufiger Prozess:

1. Definierung der Anwendung in einem *Dockerfile*
2. Anschließend werden in der Datei `docker-compose.yml` die Dienste, aus denen die Applikation besteht (auch *Services* genannt), definiert
3. `docker compose up` baut (sofern nötig) und startet alle definierten *Services*

Quelle

https://docs.docker.com/

Minimaler Aufbau

```
version: "3.7"

services:
  php:
    image: php
```

Image - Aufbau

Syntax: `image: <what>[:<version>[-<how>[-<kind>]]]`

Image - Version

- `image: php`
- `image: php:latest` (gleichbedeutend mit `image: php` - `:latest` ist implizit)
- `image: php:8`
- `image: php:8.0`
- `image: php:7.1.33`

Image - How & Kind (Tags)

- `image: php:7.1-fpm` (`<how>` ist hier `fpm`. Alternative wäre z.B. `apache` oder `cli`)
- `image: php:7.1-fpm-alpine` (`<kind>` ist `alpine`, ein **minimales** OS image. Alternative wären `buster` und `stretch`)

Image - Other

- `image: ubuntu`
- `image: ubuntu:latest`
- `image: ubuntu:20.04`
- `image: Dgame/php-custom`

Container - Name

```yaml
version: "3.7"

services:
  php:
    image: php:8.1
```

Format: `<service-name>_<image-name>_<service-nummer>`

Hier also `php_php_1`

Container - Name

```
version: "3.7"

services:
  php:
    container_name: php
    image: php:8.1
```

Ports

Short Syntax

- Beide Ports angeben: `<Host>:<Container>`
- Nur den Container Port angeben (ein freier Ports auf dem Host wird zufällig ausgewählt): `:<Container>`
- Host-IP-Adresse und die Ports (default IP ist `0.0.0.0`): `<ip>:<Host>:<Container>`.

Ports

WICHTIG: Ports immer als String angeben

> When mapping ports in the HOST:CONTAINER format, you may experience erroneous results when using a container port lower than 60, because YAML parses numbers in the format xx:yy as a base-60 value. For this reason, we recommend always explicitly specifying your port mappings as strings.

Quelle: https://docs.docker.com/

Ports

Short Syntax - Beispiele

```
ports:
  - "3000"
  - "3000-3005"
  - "8000:8000"
  - "9090-9091:8080-8081"
  - "49100:22"
  - "127.0.0.1:8001:8001"
  - "127.0.0.1:5000-5010:5000-5010"
  - "127.0.0.1::5000
  - "6060:6060/udp"
  - "12400-12500:1240"
```

Ports

Long Syntax

- `target`: Container Port
- `published`: Host Port
- `protocol`: Port Protokol (`tcp` oder `udp`)
- `mode`: *host* für einen Host-Port oder *ingress* für einen Port im Schwarmmodus für's Load-Balancing

```
ports:
  - target: 80
    published: 8080
    protocol: tcp
    mode: host
```

Dependency

```
version: "3.7"

services:
  php:
    container_name: php
    image: php:8.0-fpm-alpine
    depends_on:
      - db
      - redis
  redis:
    image: redis
  db:
    image: postgres
```

Dependency

- `docker-compose up` startet die definierten Services in der Reihenfolge der Abhängigkeiten. Im Beispiel: `db` > `redis` > `web`
- `docker-compose up <service>` schließt automatisch die Abhängigkeiten von `<service>` ein. Daher **startet** `docker-compose up web` auch `db` und `redis`.
- `docker-compose stop` **stoppt** Services in der Reihenfolge der Abhängigkeiten. Daher wird `web` vor `db` und `redis` gestoppt.

Quelle

https://docs.docker.com/

Restart

- `restart: "no"`: Default. Container wird unter keinen Umständen neugestartet
- `restart: always`: Wann immer der Container beendet wird (z.B. durch einen Fehler oder weile eine Verbindung abbricht)
- `restart: on-failure`: Nur wenn der Container durch einen Fehler beendet wurde
- `restart: unless-stopped`: Solange bis der Container erfolgreich beendet wird (z.B. manuell)

Quelle

https://docs.docker.com/

Restart - Beispiel

```
version: "3.7"

services:
  php:
    container_name: php
    image: php:8.0-fpm-alpine
    volumes:
      - .:/var/www/html/
  db:
    image: postgres
    restart: always
```

Volumes

Jeder Container geht bei jedem Start von der Image-Definition aus. Container können Dateien erstellen, aktualisieren und löschen, allerdings gehen diese Änderungen verloren, wenn der Container entfernt wird. Um das zu verhindern, benötigen wir *Volumes*.

Volumes bieten die Möglichkeit, bestimmte Dateisystempfade des Containers mit dem Host-Rechner zu verbinden. Wenn ein Verzeichnis im Container *gemountet* wird, werden Änderungen in diesem Verzeichnis auch auf dem Host-Rechner gesynct.

Quelle

https://docs.docker.com/

Volumes

Syntax:

```
  volumes:
    - <host>:<container>
```

Volumes - Beispiel

```
version: "3.7"

services:
  php:
    container_name: php
    image: php:8.0-fpm-alpine
    volumes:
      - .:/var/www/html/
```

Fragen? Kurze Pause?

Als nächstes: **Dockerfile**

Dockerfile

- `bash` zur Auto-Completion
- `git` for obvious reasons wie composer
- `shadow` um Benutzer zu verwalten (Docker startet alles als Root - BAD!)
- `ssh` & `ssl`

Dockerfile - docker-compose

```
version: "3.7"

services:
  php:
    container_name: php
    build:
      dockerfile: ./.docker/php/Dockerfile
      context: .
      args:
        USER_ID: $USER_ID
    volumes:
      - .:/var/www/html/:delegated
```

Context

Durch den Verweist `dockerfile: ./.docker/php/Dockerfile` wird als *context* `./.docker/php/` verwendet. Das heißt, dass das kopieren von Dateien von `./.docker/php/` ausgeht. Um einen anderen Ordner als Context zu verwenden, kann die Option `context <folder>` benutzt werden. Um z.B. im derzeitigen Ordner zu bleiben, wird `context: .` definiert.

Dockerfile

```
FROM php:8.0-fpm-alpine

ARG USER_ID=1000

RUN apk update --quiet && \
    apk add --quiet --no-cache bash git shadow openssh openssl-dev

WORKDIR .
COPY . .

COPY --chown=www-data:www-data --from=composer:2 /usr/bin/composer /usr/local/bin/composer

RUN usermod -u $USER_ID www-data && chown -R www-data:www-data /var/www/ .
USER www-data

CMD ["php-fpm"]
```

FROM

Syntax: `FROM <image> [AS <name>]`

- `FROM php:8`
- `FROM php:latest`
- `FROM php:7.1`
- `FROM php:8.0-fpm`
- `FROM php:8.0-fpm-alpine`

ARG

Syntax: `ARG <name>[=<default value>]`

- `ARG USER_ID`
- `ARG USER_ID=1000`

RUN

Syntax: `RUN <command>`

```
RUN apk update --quiet && \
    apk add --quiet --no-cache bash git shadow openssh openssl-dev
```

WORKDIR

Syntax: `WORKDIR <folder>`

```
WORKDIR .
```

COPY

Syntax: `COPY <src> <dest>`

`COPY . .`

WICHTIG: Wenn `WORKDIR` spezifiziert ist, geht der Pfad im Container von diesem aus.

COPY

Es ist auch möglich, von anderen *images* etwas zu kopieren.

Syntax: `COPY [--chown=<user>:<group>] [--from=<image>] <src> <dest>`

```
COPY --chown=www-data:www-data --from=composer:2 /usr/bin/composer
/usr/local/bin/composer
```

USER

Syntax: `USER <user-on-os>`

```
USER www-data
```

CMD

Syntax: `CMD [<cmd>, <arg1>, <arg2>, ...]`

Achtung: `[` ist hier **kein** Anzeichen für ein optionales Argument

- `CMD ["php-fpm"]`
- `CMD ["php", "-S", "0.0.0.0:9000"]`

WICHTIG: Bei letzterem ist der *Port* **im** Container. Um diesen Port vom Host aus anzusprechen, muss im *docker-compose.yml* ein entsprechendes Port-Mapping stattfinden.

Good to know

ENV

`ENV` kann verwendet werden, um die Umgebungsvariable *PATH* zu aktualisieren.

Syntax: `ENV <name>=<value>`

ENV - Beispiel

```
ENV PG_MAJOR=9.3
ENV PG_VERSION=9.3.4
RUN curl -SL https://example.com/postgres-${PG_VERSION}.tar.xz | tar -xJC /usr/src/postgres
ENV PATH=/usr/local/postgres-${PG_MAJOR}/bin:${PATH}
```

Fragen? Kurze Pause?

Als nächstes: **Best practise** & **Multi-Staged**

Best practise

Eine der größten Herausforderungen beim Erstellen von Images ist es, die Größe des Images gering zu halten. Jede Anweisung im Dockerfile fügt dem Image eine Schicht hinzu.

Best practise - Nicht so gut

```
FROM golang:1.7.3
WORKDIR /go/src/github.com/alexellis/href-counter/
COPY app.go .
RUN go get -d -v golang.org/x/net/html
RUN CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o app .
```

Best practise - Nicht so gut

```
FROM golang:1.7.3
WORKDIR /go/src/github.com/alexellis/href-counter/
COPY app.go .
RUN go get -d -v golang.org/x/net/html
RUN CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o app .
```

Zwei `RUN` - Zwei Layer

Best practise - Besser

```
FROM golang:1.7.3
WORKDIR /go/src/github.com/alexellis/href-counter/
COPY app.go .
RUN go get -d -v golang.org/x/net/html \
    && CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o app .
```

Quelle: https://docs.docker.com/

Multi-Staged

Multi-Staged - Nicht so gut

```
FROM golang:1.7.3
WORKDIR /go/src/github.com/alexellis/href-counter/
RUN go get -d -v golang.org/x/net/html
COPY app.go .
CMD ["./app"]
```

Multi-Staged - Besser

```
FROM golang:1.7.3 AS builder
WORKDIR /go/src/github.com/alexellis/href-counter/
RUN go get -d -v golang.org/x/net/html
COPY app.go .

FROM alpine:latest
WORKDIR /root/
COPY --from=builder /go/src/github.com/alexellis/href-counter/app .
CMD ["./app"]
```

Quelle: https://docs.docker.com/

Makefile

`make` ist ein build Tool um komplexe Befehle (und ggf. deren Abhängigkeiten) zusammenzufassen. Ein *Makefile* besteht aus sogenannten *Targets*, optional Abhängigkeiten (separiert durch mind. einem Leerzeichen) und ausführbare Befehle. Letztere werden per Tab eingerückt unterhalb der *Targets* und deren optionalen Abhängigkeiten. Ein Beispiel:

```
A: C B
	@echo "World"
B:
	@echo "my"
C:
	@echo "Hello"
```

Was kommt bei der Ausführung von `make A` raus?

Makefile

Der Befehl `make A` würde im *Target* `A` die Abhängigkeit zu `C` und `B` sehen und somit zunächst `C`, dann `B` und danach erst `A` ausführen. Die Ausgabe wäre:

```
Hello
my
World
```

Makefile: Benennung

Um direkt per `make A` das *Target* `A` auszuführen, muss die Datei `Makefile` heißen (so wie die Standard Datei für Docker `Dockerfile` heißt).

Wenn die Datei nicht `Makefile` heißt oder heißen soll/kann, dann muss die Datei auf die Endung `*.mk` enden und kann mit dem `-f` Argument eingelesen werden. Würde die Datei also z.B. `test.mk` heißen, würde das *Target* `A` wie folgt ausgeführt werden:

```
make -f test.mk A
```

Makefile: Wozu?

Um z.B. per *docker-compose* einen neuen Container von Grund auf zu bauen und ältere Artefakte loszuwerden, muss man den folgenden Befehl ausführen:

```
docker-compose up -d --build --remove-orphans
```

Das ist lang und umständlich. Stattdessen kann ein *Makefile* verwendet werden mit dem folgenden Inhalt:

```
build:
	docker-compose up -d --build --remove-orphans
```

Was dann per `make build` ausgeführt werden kann.

Makefile: Warum kein Shell-Script?

- Shell Scripte sind nicht unbedingt kompatibel zwischen unterschiedlichen Shells (`sh`, `dash`, `bash`, `fish`, `zsh`, etc.)
- Die Syntax ist komplexer
- `make` ist auf jedem Linux-artigen Betriebssystem (und oft auch bei Windows) vorinstalliert

Fragen? Kurze Pause?

Als nächstes: **Exkurs 1 - 3**

Danach: **Ende**

Exkurs #1

Erstelle mit *docker/docker-compose* einen PHP Service, der eine PHP Datei einliest mit dem folgenden Inhalt:

```
<?php

phpinfo();
```

Als **PHP Version** soll **7.4** verwendet werden.

- Unter http://localhost:8080 soll die *phpinfo* ausgegeben werden.
- Änderungen an der Datei auf dem Host sollen im Container ohne Neustart sichtbar sein (Stichwort *Volumes*).

Tipp: Verwende den builtin php Server (`php -S ...`)
Zeit: 30 - 60 Minuten

Exkurs #2

Erweitere das Ergebnis aus *Exkurs #1* insofern, dass anstatt des builtin php Servers *php-fpm* verwendet wird.

Zeit: 2 Stunden

Exkurs #3

Erstelle einen PHP sowie einen MySQL Service per *docker*/*docker-compose*. Als **PHP Version** soll **7.4** verwendet werden.

- Änderungen an der Datei auf dem Host sollen im Container ohne Neustart sichtbar sein (Stichwort *Volumes*)
- *composer* steht als Befehl im Container zur Verfügung
- Der PHP-Service soll bei jedem Aufruf einen neuen (random) Eintrag in eine Tabelle in der MySQL aufnehmen und die ersten 5 Ergebnisse ausgeben.

Zeit: 2 - 4 Stunden

Fragen?