Lernzielzusammenfassung Modul 165

Ich kann die Begriffe Containerisierung, Image, Layer, Container, Repository, Registry und Dockerfile erklären:

Containerisierung: Containerisierung ist eine Technologie, die es ermöglicht, Anwendungen und deren Abhängigkeiten isoliert in Containern zu verpacken.

Image: Ein Docker Image ist eine schreibgeschützte Vorlage, die die Anwendung und alle notwendigen Abhängigkeiten enthält. Es ist die Basis, um Container zu erstellen.

Layer: Images werden oft in Schichten (Layers) aufgebaut, um effizienter wiederverwendet zu werden.

Container: Ein Container ist eine laufende Instanz eines Docker Images. Er enthält alles, was die Anwendung benötigt, um zu laufen, und ist isoliert vom restlichen System.

Repository: Eine Sammlung von Images, oft für eine bestimmte Anwendung oder Version.

Registry: Eine Registry ist ein Speicherort für Docker Images. Die bekannteste Registry ist Docker Hub, es gibt aber auch private Registries. Aus der Registry werden Images heruntergeladen (pulled) oder hochgeladen (pushed).

Dockerfile: Eine Textdatei, die die Befehle enthält, um ein Docker Image zu erstellen. Das Dockerfile beschreibt Schritt für Schritt, wie die Umgebung des Containers aufgebaut werden soll.

Ich kenne die Architektur der Docker-Engine:

Docker-Engine ist eine Open-Source-Containerisierungstechnologie zum Erstellen und Containerisieren von Anwendungen. Docker Engine ist als Client-Server-Anwendung aufgebaut:

- Server mit einem langlaufenden Daemon-Prozess dockerd.
- Docker-CLient (docker) als Befehlszeilenschnittstelle (CLI).
- APIs, die Schnittstellen definieren, über die Programme mit dem Docker-Daemon kommunizieren und ihn anweisen können.

Ich kenne die verschiedenen Stati eines Containers:

Status	Beschreibung
`created`	Der Container wurde erstellt, aber noch nicht gestartet.
`running`	Der Container läuft gerade und führt Prozesse aus.
`paused`	Der Container läuft noch, aber alle Prozesse sind angehalten (z. B. `docker pause`).
`exited`	Der Container ist beendet (entweder erfolgreich oder mit Fehler).
`dead`	Der Container ist in einem fehlerhaften Zustand (z. B. nicht korrekt beendet).

Diese Zustände kannst du z. B. mit docker ps -a anzeigen lassen.

Ich kenne die elementaren Commands, um Container zu erzeugen, starten, beenden und löschen:

Sie kennen die elementaren Commands, um Container zu erzeugen, starten, beenden und löschen.

- docker run: Erstellt und startet Container.
- docker start / stop: Startet oder stoppt vorhandene Container.
- docker rm: Löscht gestoppte Container.
- Docker rmi <name>: Löscht Docker-Image von deinem System.
- docker ps: Zeigt laufende Container.
- docker ps -a: Zeigt alle Container.

Sie kennen die wichtigsten Optionen:

- --name: Gibt dem Container einen Namen.
- --rm: Container wird nach dem Stoppen automatisch gelöscht.
- --network: Bestimmt das Docker-Netzwerk, in dem der Container läuft.
- --ip: Setzt eine feste IP-Adresse (nur mit benutzerdefiniertem Netzwerk).
- -d: Startet den Container im Hintergrund (detached).
- -it: Interaktiv + Terminal (z. B. für Shell-Zugriff).
- -p: Portweiterleitung (z. B. 8080:80).
- -v: Mountet ein Volume oder Verzeichnis.
- -e: Setzt Umgebungsvariablen.

Ich kann mit «docker container exec» Commands in einem laufendem Container ausführen:

Mit «docker exec -it <container>» bash kannst du z. B. in den Container rein und Befehle ausführen.

In der Konsole dann:

- ls – Zeigt Dateien und Verzeichnisse
- pwd – Zeigt das aktuelle Verzeichnis
- cd – Verzeichnis wechseln
- cat – Zeigt Datei-Inhalte
- echo – Gibt Text aus (z. B. echo $PATH)
- env – Zeigt alle Umgebungsvariablen

Ich kann verschiedene Versionen (tags) eines Container-Images nutzen:

Images haben **Tags**, z. B. nginx:1.21 oder nginx:latest. Du kannst gezielt Versionen starten: docker run nginx:1.21

Ich kann Container mit umgeleiteten Ports starten:

Sie starten Container mit umgeleiteten Ports.

- Mit -p 8080:80 wird Port 80 im Container auf Port 8080 deines Rechners gemappt.

  Läuft beispielsweise im Container eine Webanwendung auf Port 80 (http), läss sich dieser Port beim Start des Containers mit dem Parameter -p mit dem Port 80 des Hostrechners so verbinden

  ```
  docker run  -p 80:80 <image>
  ```

  Auf dem Host lässt sich somit die Webseite des Containers mit `http://localhost` öffnen

  Die Syntax für -p lautet

  ```
  -p hostport:containerport
  ```

  Die erste Zahl bezieht sich also auf die Portnummer des Hostrechners und die zweite der Portnummer im Container drin. Die Syntax folgt der in Docker üblichen Reihenfolge auch bei anderen Zuordnungen, immer zuerst den Host und dann den Container anzugeben.

  Sollte der Port 80 des Hostrechners im obigen Beispeil bereits belegt sein, weil z.B. der Host selber ein Webserver ist und damit Port 80 verwendet, kann man einfach eine andere Portnummer verwenden. Für Webdienste üblich ist z.B. Port 8080:

  ```
  docker run  -p 8080:80 <image>
  ```

  Die Containerwebseite ist dann unter `http://localhost:8080` erreichbar.

Ich kann unbenannte, benannte und gemountete Volumes einsetzen:

Unbenannte Volumes:

Anonymous Volumes**:** Schauen wir dazu nochmals das Beispiel des mariadb-Servers an. Einen Image können Sie mit dem folgenden Kommando herunterladen und einen Container daraus starten

docker run -d --name mariadb-test -e MYSQL_ROOT_PASSWORD=geheim mariadb

Um herauszufinden wo nun die Daten aus /var/lib/mysql gelandet sind können Sie das Kommando

docker inspect -f '{{.Mounts}}' mariadb-test

ergibt:

vmadmin@ubuntu:~/bsp-apache-php$ docker inspect -f '{{.Mounts}}' mariadb-test

[{volume 752507751a42f0c781b96adacb4a3d73bdbbf2184ead3bd4874b4b5f065ee4eb /var/lib/docker/volumes/752507751a42f0c781b96adacb4a3d73bdbbf2184ead3bd4874b4b5f065ee4eb/_data /var/lib/mysql local true }]

vmadmin@ubuntu:~/bsp-apache-php$

Wenn beim Erstellen des Containers also nichts eingestellt wird, liegt das Verzeichnis in einem zufällig benannten Unterverzeichnis von /var/lib/docker/volumes des Hostrechners. Die zufällige Bezeichnung verunmöglicht eine praktikable Verwaltung von Volumes nahezu. Sollte z.B ein Volume gelöscht werden, müsste der vollständige Verzeichnisname angegeben werden: docker volume rm 7525077...

Benannte Volumes:

Named Volumes**:** Eine bessere Variante ist es deshalb, Volumes beim Erstellen eines Containers zu benennen:

docker run -d --name mariadb-test2 -v myvolume:/var/lib/mysql -e MYSQL_ROOT_PASSWORD=geheim mariadb

vmadmin@ubuntu:~/bsp-apache-php$ docker run -d --name mariadb-test2 -v myvolume:/var/lib/mysql -e MYSQL_ROOT_PASSWORD=geheim mariadb

2ccb55f675a5867efeefaa8bfd02e896cf0e0e1b66fe098465d6d7cc3e6a9bc6

vmadmin@ubuntu:~/bsp-apache-php$ docker inspect -f '{{.Mounts}}' mariadb-test2

[{volume myvolume /var/lib/docker/volumes/myvolume/_data /var/lib/mysql local z true }]

Wie Sie sehen, wird nun der gewählte Name für das Unterverzeichnis verwendet. Die Syntax lautet somit:

-v volumename:containerverzeichnis

Mounted Volumes:

Mounted Volumes**:** Anstelle eines Namens für das Hostvolume kann auch ein Verzeichnis angegeben werden:

-v hostverzeichnis:containerverzeichnis

Damit wird das Volume ganz aus der Dockerumgebung herausgelöst und kann an einer beliebigen Stelle platziert werden:

mkdir /home/vmadmin/databases

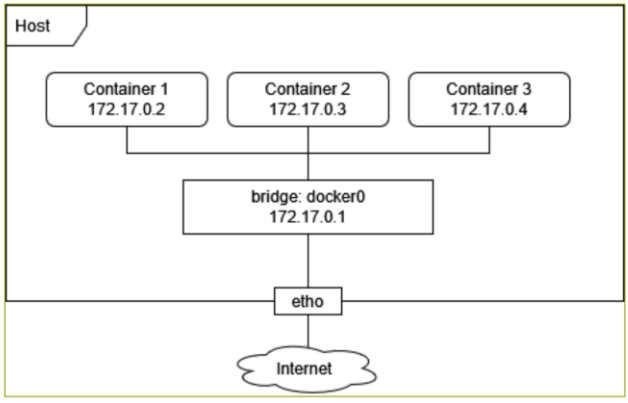
docker run -d --name mariadb-test3 -v /home/vmadmin/databases:/var/lib/mysql -e MYSQL_ROOT_PASSWORD=geheim mariadb

Ich kann ein Docker-Netzwerk einrichten und kann ihm verschiedene Container zuweisen:

Standardnetzwerk:

Standardnetzwerk

Das Netzwerk mit dem Namen bridge ist das Standardnetzwerk und wird verwendet wenn nichts anderes angegeben wird. Die Netzwerkarchitektur lässt sich wie folgt darstellen:



Bridge Netzwerk

Der Befehl:

- docker network ls

Zeigt alle Netzwerke an.

Zuerst wird ein Ubuntu-Container gestartet mit zum Beispiel:

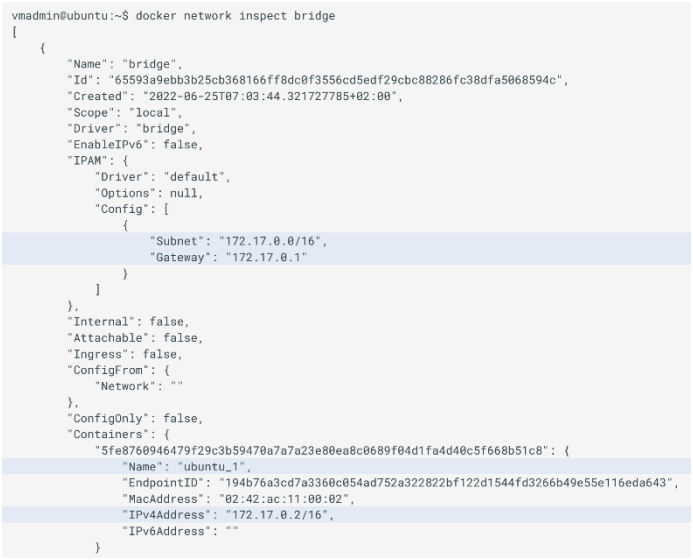
- docker run -it --name ubuntu_1 ubuntu:latest

Dann ist man in der Ubuntu-Konsole.

In einer zweiten Konsole kann nun das Kommando

- docker network inspect bridge

ausgeführt werden



Eigene Netzwerke:

Alle Container landen standardmässig im selben Netzwerk, dem bridge-Netzwerk. Dies ist aus sicherheitstechnischen Gründen nicht ideal, wenn unterschiedliche Anwendungen voneinander isoliert sein sollen. Es lassen sich deshalb eigene Netzwerke definieren und diese den Containern zuordnen.

docker network create \

--driver=bridge \

--subnet=10.10.10.0/24 \

--gateway=10.10.10.1 \

my_net

Überprüfen mit

```
vmadmin@ubuntu:~$ docker network ls
NETWORK ID     NAME                                DRIVER    SCOPE
65593a9ebb3b   bridge                              bridge    local
364521a9eaa2   host                                host      local
049d1ccb15e7   my_net                              bridge    local
```

ip addr zeigt die neue Schnittstelle an:

- ip addr

Ein Container kann nun beim Start diesem Netzwerk zugeordnet werden:

- docker run -it --name ubuntu_2 --network=my_net ubuntu:latest

und

- docker network inspect my_net

zeigt die IP-Adresse des neuen Containers

vmadmin@ubuntu:~$ docker network inspect my_net

...

"Config": [

{

"Subnet": "10.10.10.0/24",

```
            "Gateway": "10.10.10.1"
...
    "Containers": {
        "8848cacd27a445a7805e34bed6542c21535aa5e7cbb370e4cae7e26cd7d19f15": {
            "Name": "ubuntu_2",
            "IPv4Address": "10.10.10.2/24",
...
```


Die IP-Adresse für den Container wird dabei von docker via DHCP aus dem definierten Netzwerk vergeben. Alternativ kann eine fixe IP-Adresse beim Start des Containers angegeben werden.

- docker run -it --name ubuntu_2 --ip="10.10.10.10" --network=my_net ubuntu:latest

Als nächstes soll nun der weiteroben dem Netzwerk bridge zugeordnete Container ubuntu_1 dem Netzwerk my_net zugeordnet werden. Dazu trennen wir ihn zuerst von bridge mit

- docker network disconnect bridge ubuntu_1

anschliessend wird er zu my_net hinzugefügt und neu gestartet

- docker network connect my_net ubuntu_1
- docker start -i ubuntu_1

docker inspect zeigt nun beide Container an:

docker network inspect my_net


```
...
    "Containers": {
        "5fe8760946479f29c3b59470a7a7a23e80ea8c0689f04d1fa4d40c5f668b51c8": {
            "Name": "ubuntu_1",
            "IPv4Address": "10.10.10.3/24",
            "IPv6Address": ""
        },
        "8848cacd27a445a7805e34bed6542c21535aa5e7cbb370e4cae7e26cd7d19f15": {
            "Name": "ubuntu_2",
            "IPv4Address": "10.10.10.2/24",
            "IPv6Address": ""
        }
...
```

Prüfen:

Um zu überprüfen, ob die beiden Container tatsächlich mit einander kommunizieren können, installieren wir auf ubuntu_1 das Paket iputils-ping:

- apt update
- apt install iputils-ping

Anschliessend wird ubuntu_2 angepingt. Ubuntu_1 und _2 sind beides einzelne Container die so zum Beispiel erstellt werden können:

- docker run -it --name ubuntu_1 ubuntu:latest
- docker run -it --name ubuntu_2 --network=my_net ubuntu:latest
- ping 10.10.10.2

Dabei ist es nicht einmal nötig die IP-Adresse von ubuntu_2 zu kennen, da man auch den Namen direkt verwenden kann

- ping ubuntu_2

Nachdem alle Container, die zu einem Netzwerk hinzugefügt wurden, gestoppt oder getrennt wurden, können Sie das Netzwerk mit folgendem Befehl löschen:

- docker network rm my_net

Ich kann ein mir nicht bekanntes Image anhand der offiziellen Dokumentation erstellen:

Standardimage:

Wird eine umfangreichere oder bekanntere Linux Distribution benötigt, kann man z.B. das ubuntu Image verwenden. Dieses ist deutlich grösser als alpine. Das image kann so ausprobiert werden:

- docker run -it --name ubuntu-test ubuntu:latest

eine spezifische Version mit

- docker run -it --name ubuntu-test ubuntu:20.04

oder

- docker run -it --name ubuntu-test ubuntu:latest

Der Container lässt sich mit exit beenden und kann mit

- docker start -i ubuntu-test

auch wieder neu gestartet werden.

Eigenes Image:

Benötigte Dateien:

Erstellen Sie im Homeverzeichnis von vmadmin ein neues Verzeichnis apache-php

- mkdir bsp-apache-php

und wechseln Sie gleich ins neu erstellte Verzeichnis

- cd bsp-apache-php

Erstellen Sie in diesem Verzeichnis eine neue Datei mit Namen Dockerfile und dem abgebildeten Inhalt:

```
# Datei: bsp-apache-php/Dockerfile

FROM php:8-apache

COPY index.php /var/www/html
```

Diese Datei gibt das Rezept an, wie ein Image erstellt wird. **FROM** gibt das verwendete Basisimage an. **COPY** kopiert die Datei index.php ins Verzeichnis /var/www/html im Container. Dieses Verzeichnis ist bei Apache das Standardverzeichnis für Webseiten, d.h. beim Aufruf der Webseite wird index.php aufgerufen von php bearbeitet und an den aufrufenden Browser ausgeliefert.

Die Datei index.php muss natürlich noch erstellt werden und hat folgenden Inhalt:

```
<!DOCTYPE html >

<!-- Datei index.php -->

<html >

<head >

<title >Beispiel</title >

<meta charset ="utf-8" />

</head >

<body >

<h1>Beispiel apache/php</h1>

Serverzeit : <?php echo date("j. F Y, H:i:s, e "); ?>

</body >

</html >
```

Image builden:

Nun wird das Image mit folgendem Befehl erstellt:

- docker build -t bsp-apache-php .

Der Schalter -t gibt den Namen des Images an, hier also bsp-apache-php. Üblicherweise startet man dieses Kommando aus dem Verzeichnis in dem sich das Dockerfile befindet. Somit gibt der Punkt im Kommando einfach an, dass Sie sich bereits in diesem Verzeichnis befinden. In zwei Schritten wird nun das Basisimage heruntergeladen und extrahiert und die Indexdatei kopiert.

Überprüfen Sie den Erfolg mit dem Kommando

- docker image ls

Hier sollte (unter anderem) nun 2 Images aufgeführt werden:

Container starten:

Nun kann aus dem Image ein Container gestartet werden:

- docker run -d --name bsp-apache-php-container -p 8080:80 bsp-apache-php

Nun können Sie die Webseite http://localhost:8080 aufrufen:

Ich kann die 12 verschiedenen Anweisungen von Dockerfiles erklären:

- FROM: Basis-Image definieren.
- RUN: Befehl beim Bauen ausführen (z. B. Software installieren).
- CMD: Standardbefehl beim Start des Containers.
- ENTRYPOINT: Definiert, was IMMER ausgeführt wird beim Start.
- COPY: Dateien in das Image kopieren.
- ADD: Wie COPY, aber mit mehr Features (z. B. URL-Download, Entpacken).
- ENV: Setzt Umgebungsvariablen.
- EXPOSE: Gibt an, welche Ports das Image nutzt.
- WORKDIR: Setzt das Arbeitsverzeichnis.
- USER: Setzt den Nutzer im Container.
- LABEL: Metadaten setzen.
- VOLUME: Gibt Verzeichnisse an, die als Volumes dienen sollen.

Ich kann ENTRYPOINT und CMD sowie COPY und ADD unterscheiden:

- ENTRYPOINT ist „fix“, CMD ist „optional“ – CMD kann von der Kommandozeile überschrieben werden, ENTRYPOINT eher nicht.
- COPY kopiert nur lokale Dateien, ADD kann auch URLs und Archive (automatisch entpacken) verarbeiten.

Dockerfile mit CMD:

FROM ubuntu:latest

CMD ["echo", "Hallo, ich bin CMD!"]

Wenn ich jetzt das so starte:

- docker run cmd-test echo "Neuer Befehl!"

Dann wird CMD überschrieben und die Ausgabe ist «Neuer Befehl»

Dockerfile mit ENTRYPOINT:

FROM ubuntu:latest

ENTRYPOINT ["echo", "Hallo, ich bin ENTRYPOINT!"]

Wenn ich es jetzt genau gleich starte mit:

- docker run entrypoint-test echo "Neuer Befehl!"

Dann sieht die Ausgabe so aus: Hallo, ich bin ENTRYPOINT! echo Neuer Befehl!

Das heisst ENTRYPOINT wird nicht überschrieben.

Unterschied zwischen COPY und ADD:

COPY kann nur lokale Dateien kopieren, ADD kann auch entpacken und von URLs herunterladen.

Beispiel COPY:

FROM ubuntu:latest

COPY lokaledatei.txt /app/

Kopiert lokaledatei.txt aus dem lokalen Verzeichnis ins Image.

Beispiel ADD mit automatischem entpacken:

FROM ubuntu:latest

ADD meinarchiv.tar.gz /app/

meinarchiv.tar.gz wird automatisch entpackt ins /app/-Verzeichnis!

Beispiel ADD mit URL:

FROM ubuntu:latest

ADD https://example.com/meinedatei.txt /app/

Docker lädt die Datei direkt aus dem Internet herunter.

Ich kann ein eigenes Docker-Image (Single-State) mit Hilfe eines Dockerfiles erstellen:

Hier ist die Zusammenfassung des Videos mit allen Codes und Dateien:

1. Erstellen der Node.js-Anwendung

a. Initialisiere ein neues Node.js-Projekt:

npm init -y

Dieser Befehl erstellt eine package.json-Datei mit Standardwerten.

b. Installiere die benötigten Abhängigkeiten:

npm install express

Dies installiert das Web-Framework Express.

c. Erstelle die Datei app.js mit folgendem Inhalt:

```
const express = require('express');

const app = express();

const PORT = process.env.PORT || 3000;


app.get('/', (req, res) => {

  res.send('Hello, World!');

});


app.listen(PORT, () => {

  console.log(` Server is running on port ${PORT}` );

});
```

Diese einfache Anwendung startet einen Webserver auf Port 3000 und gibt "Hello, World!" zurück.

2. Erstellen der Docker-Konfigurationsdateien

a. Erstelle eine .dockerignore-Datei, um unnötige Dateien vom Docker-Build auszuschließen:

```
node_modules

npm-debug.log
```

b. Erstelle ein Dockerfile mit folgendem Inhalt:

```
# Basis-Image

FROM node:14


# Arbeitsverzeichnis im Container

WORKDIR /usr/src/app


# Kopiere package.json und package-lock.json

COPY package*.json ./


# Installiere Abhängigkeiten

RUN npm install
```

Kopiere den restlichen Anwendungscode

COPY . .

Exponiere den Port, auf dem die App läuft

EXPOSE 3000

Definiere den Startbefehl

CMD ["node", "app.js"]

Das Dockerfile verwendet ein Node.js-Image als Basis, setzt das Arbeitsverzeichnis, kopiert die Dateien, installiert Abhängigkeiten und startet die Anwendung.

3. Bauen und Ausführen des Docker-Containers

a. Baue das Docker-Image:

docker build -t node-app .

Dieser Befehl erstellt ein Docker-Image mit dem Namen node-app.

b. Starte einen Container aus dem erstellten Image:

docker run -p 3000:3000 node-app

Dieser Befehl startet einen Container und mappt den Container-Port 3000 auf den Host-Port 3000.

c. Zugriff auf die Anwendung: Öffne einen Webbrowser und gehe zu http://localhost:3000. Du solltest die Nachricht "Hello, World!" sehen.

Durch diese Schritte hast du eine einfache Node.js-Anwendung erstellt, sie mit Docker containerisiert und lokal ausgeführt.