Technical Report : OTH-Wiki

Dominik Smrekar, Johannes Horst, Patrick Sabau, Saniye Ogul und Tobias Schotter

I. EINLEITUNG

OTH-Wiki ist eine Web-Anwendung, welche relevante Informationen für Studenten bereits stellt.
Nutzer der Seite, sehen dabei nur die Single-Page-Applikation, welche mittels Angular erstellt wurde. Diese wird mittels eines Nginx-Servers ausgeliefert, welcher sich in einem Docker-Container befindet. Um immer aktuelle Inhalte anzuzeigen, wird mittels einem Backend kommuniziert, welches mittels Python und FastAPI Informationen bereitstellt, und in einer MongoDB Datenbank sichert.

II. BACKEND + DATENBANK

III. FRONTEND

IV. DEPLOYMENT

Das Deployment basiert primär auf Containerisierung mittels Docker. Das Backend und die Datenbank sind dabei abgekapselt, während der Frontend-Container zusätzlich einen Nginx Server enthält.
Im Folgenden werden die einzelnen Bestandteile des Deployments detailliert erläutert.

A. Backend Dockerfile

Für das Backend wird ein relativ simpler Container gebaut. Listing 1 zeigt die entsprechendes Dockerfile.

```
FROM python:3.10-slim

COPY ./requirements.txt .
RUN python -m pip install -r requirements.txt

COPY . /

ENTRYPOINT ["python"]
CMD ["-m", "backend"]
```

Listing 1. Backend Dockerfile

Als Basis wird dabei ein Python Container genutzt (Zeile 1). In diesen werden die Requirements rein kopiert und anschließend installiert (Zeile 2-3). Der restliche Code wird erst im Anschluss kopiert (Zeile 6). Dies hat den Vorteil, dass der Container nach Zeile 4 gecached werden kann. Dadurch müssen die Requirements nur neu installiert werden, wenn sich eine Dependency geändert hat, nicht jedoch nach Codeänderungen. Da in Kapitel II bereits beschrieben wurde, dass der Python Code die Konfiguration des Servers übernimmt, reicht ein simpler ***python -m backend*** Befehl, um die Anwendung im Container zu starten (Zeile 8-9). Eine zusätzliche Optimierung dieses Containers ist die Nutzung einer **.dockerignore**-File. Diese enthält das Virtual-Environment der lokalen Entwicklung. Dies sorgt dafür, dass durch den Befehl **COPY . /** aus Zeile 6, nur selbst entwickelter Code übernommen wird, nicht jedoch lokal installierte Pakete.

B. Frontend Dockerfile

Angular kann aus allen Komponenten und Modulen eine einzige **html**-Seite erzeugen, welche danach mittels eines Webservers bereit gestellt werden kann. Diese beiden Aufgaben werden von dem Frontend Dockerfile übernommen. Dieses wird in Listing 2 dargestellt.

```
FROM node:16.15-bullseye as build

ENV PATH /app/node_modules/.bin:$PATH

WORKDIR /app

COPY package.json /app/package.json
RUN npm install
RUN npm install -g @angular/cli

COPY . /app
RUN ng build --output-path=dist

FROM nginx:1.21

COPY --from=build /app/dist
      /usr/share/nginx/html

COPY nginx.conf.template
     /etc/nginx/conf.d/default.conf

EXPOSE 80
EXPOSE 443
CMD ["nginx","-g","daemon off;"]
```

Listing 2. Backend Dockerfile

Ein **NodeJS**-Container wird als *build*-Container genutzt (Zeile 1). Dies sorgt dafür, dass dieser Teil nur temporär (nur zum Bauen der HTML-Seite) erzeugt wird, und zur Laufzeit wieder entfernt wird. Dies reduziert die Größe des laufenden Containers. In diesen *build*-Container werden (wie beim Python Container) die Dependencies reingeladen und installiert (Zeile 7-9). Nach Kopieren des Codes (Zeile 11) wird der Befehl **ng build** ausgeführt, welcher die tatsächliche Singe-Page erstellt (Zeile 12).
Als laufender Container wird ein Nginx Container genutzt (Zeile 14). In diesen muss neben der *nginx.conf*-File (Zeile 19-20) auch die HTML-Seite aus dem *build*-Container kopiert werden (Zeile 16-17). Ports eines *Docker*-Containers können in der Dockerfile geöffnet werden (Zeile 22-23), jedoch aber z.B. in der docker-compose File. Der Container wird schlussendlich mit dem **nginx** Befehl gestartet (Zeile

24).
Details der Nginx Konfiguration werden in IV-D beschrieben

C. Docker Compose

Der Container des Frontends und des Backends, sowie ein separater Container für die Datenbank werden in einer einzigen docker-compose File zusammengefasst, wodurch die gesamte Anwendung auf einmal gestartet werden kann. Listing 3 zeigt entsprechende **docker-compose.yml**.

```
---
version: '3'
services:
  mongodb:
    image: mongo:4.4
    hostname: mongodb
    container_name: mongodb
    ports:
      - 27017:27017
    #volumes:
    #  - /data/mongodb:/data/db
  backend:
    build: ./backend
    hostname: backend
    container_name: backend
    ports:
      - 5000:5000
    depends_on:
      - mongodb
  frontend:
    build: ./frontend
    hostname: frontend
    container_name: frontend
    depends_on:
      - mongodb
      - backend
    ports:
      - 80:80
```

Listing 3. docker-compose

Der **MongoDB** Service (Zeile 4-11) benutzt dabei ein *mongo*-Containerimage in der Standardausführung. Durch Mounten eines Laufwerkordners (was hier ausgeklammert ist), könnten die Daten persistent auf dem Host gespeichert werden, wodurch diese unabhängig von dem Container vorliegen würden.
Der **Backend** Service (Zeile 12-19) nutzt das bereits beschriebene Dockerfile. Eine Besonderheit hierbei ist, dass der Service von der MongoDB abhängig ist, und erst startet, sobald die Datenbank gestartet wurde.
Der **Frontend** Service (Zeile 20-28) ist abhängig von den anderen beiden, weshalb dieser als letztes startet.

Neben Host- und Containername werden noch Ports spezifiziert. Diese sorgt dafür, dass Ports innerhalb eines Containers, auch durch Ports des Hosts erreicht werden können. Die Services untereinander können jedoch auch ohne diese Port-Exposures kommunizieren, da Docker Compose intern ein Docker Network startet und somit die Services verknüpft. Sollten also alle Services auf dem gleichen Server liegen, sollte nur der Port 80 des Frontend-Services geöffnet werden, nicht jedoch Ports der anderen Services (zumindest sofern keine weiteren Security-Features integriert wurden).

Sofern Docker läuft, kann die ganze Webanwendung mittels des Befehls **docker-compose up [−−flags]** gestartet werden.

D. Nginx

- Serven der Website
- Proxy für das Backend

V. AUSBLICK