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

Das Backend wurde mit dem Framework **FastAPI** in Python umgesetzt und ist die einzige Schnittstelle zur Datenbank **MongoDB**.

A. Datenbank

Um eine MongoDB lokal (für die Entwicklung) zu starten, gibt es mehrere Möglichkeiten:

- Herunterladen von MongoDB über die offizielle Website oder einen Package Manager (z.B. ABT)
- Starten eines einzelnen Docker Containers mittels **docker run –name mongodb -d -p 27017:27017 mongo:4.4**
- Starten mittels des existierenden Docker Compose Files **docker-compose up -d mongodb**

Die Datenbank wird dabei in der Standardkonfiguration genutzt, somit ist es nicht erforderlich Nutzer oder anderes anzulegen. Lediglich sollte die Datenbank über den Standardport 27017 erreichbar sein.

MongoDB enthält zu Beginn einige Tabellen die zur Konfiguration und zur internen Konsistenz genutzt werden. Die eigentlichen Daten werden dabei in einer neuen Datenbank (der Name dieser kann konfiguriert werden) gespeichert. Eine solche Datenbank kann mehrere Collectionen beinhalten, das Äquivalent zu Tabellen in einer SQL Datenbank. Diese Collectionen beinhalten i.d.R ähnliche Documente im Format BSON (Binary JSON).

B. Datenformat

Es werden zwei Datenformate genutzt, die jeweils in einer eigenen Collection gespeichert werden : **Kategorien** und **Artikel**.

Listing 1 zeigt ein Kategorie Objekt, wie es in der Datenbank gespeichert sein könnte.

```
{
    "_id": ObjectID(abc123def456),
    "kategorie": "Beispielkategorie",
    "parent_kategorie" : null,
    "subkategorien"    : ["ghi789jkl101"],
    "artikel" : [
        {
        "name":"Beispielartikel",
        "id"  :"lmnopqrs333"
        }]
}
```

Listing 1. Kategorie Objekt

Das Feld **'_id'** ist ein Primärschlüssel, welcher einzigartig ist und automatisch von MongoDB vergeben wird. Mit diesem können Objekte eindeutig referenziert werden. Über **'kategorie'** wird der Name einer Kategorie als String gespeichert. **'parent_kategorie'** und **'subkategorien'** dienen dazu, eine Hierarchie zwischen den Kategorien zu etablieren. Dabei handelt es sich um eine Doppelreferenz, ähnlich einer *Doppelt verketteten Liste*. Dabei kann eine Kategorie jedoch mehrere Subkategorien enthalten, wohingegen maximal eine Parent-Kategorie definiert werden kann. Ist der Parent jedoch None bzw. null, ist die Kategorie in der Hierarchie ganz oben. Das Feld **'artikel'** enthält eine Liste von Artikelname sowie dessen ID. Dadruch können die Artikel zugeordnet werden, sowie im Frontend passend der Name unter entsprechender Kategorie angezeigt werden. Das Beispiel zeigt bereits, dass eine Kategorie sowohl Subkategorien, als auch Artikel enthalten kann.

```
{
    "_id": ObjectID(lmnopqrs333),
    "artikel_name": "Beispielartikel",
    "artikel_text": "Artikel ueber ...",
    "kategorie" : "abc123def456",
    "current_version": 2,
    "tags":["beispiel","tags"],
    "created": 01-01-2020,
    "old_versions" : [
        {
        "name":"Beispielartikel_alt",
        "text"  :"Hier war ..."
        "version": 1
        }]
}
```

Listing 2. Artikel Objekt

Ein Artikel Objekt aus der Datenbank wird in Listing 2 dargestellt. Die meisten Felder dienen den Informationen des Artikels, wie etwa **artikel_name** (Überschrift) und **artikel_text**. Das **kategorie** Feld dient zur Verbindung

mit entsprechender Kategorie. Sollte ein Artikel geupdated werden, wird die alte Version in dem Feld **old_versions** hinterlegt, und die Inhalte der anderen Felder geupdated.

Die Datenbankobjekte werdem im Code als Pydantic Dataclasses hinterlegt, was das Parsen dieser Objekte (z.B. als JSON Payload) erleichtert. Dabei werden die Klassen mit dem Decorator **@dataclass** verziert und erhalten **TypeHints** mit entsprechden Datentypen

```
@dataclass
class Category():
    kategorie: str
    parent_kategorie: Optional[str] = None
    subkategorien: Optional[List[str]] =
        Field(default_factory=list)
    artikel: Optional[List[str]] =
        Field(default_factory=list)
```

Listing 3. Kategorie Dataclass

C. Backend

Mittels dem Python Framework **FastAPI** werden diverse Endpoints bereitgestellt, die das Erstellen und Ausgeben der Datenobjekte ermöglicht. Für Kategorien gibt es dabei folgende Datenpunkte:

- **GET /categories**: Alle Kategorien werden hierarchisch sortiert und ausgegeben.
- **GET /categories/{id oder name}**: Eine bestimmte Kategorie wird ausgegeben. Kann auch genutzt werden, um zu prüfen ob eine Kategorie existiert
- **POST /categories**: Anlegen einer neuen Kategorie
- **DELETE /categories/{id}**: Löschen einer Kategorie mittels der ID

Die Artikel können über folgende Routen gemanaged werden:

- **GET /article/{id}**: Gebe einen Artikel mit allen Infos (z.B. Artikeltext) aus.
- **POST /articles**: Erstelle einen neuen Artikel.
- **POST /articles/update**: Ermöglicht das Update eines Artikels, benötigt jedoch die ID des ursprünglichen Objekts.
- **DELETE /articles/{id}**:Löschen eines Artikels mittels der ID

Die Endpunkte (besonders die POST Routen) nutzen dabei ein automatisiertes Umwandeln zu entsprechenden Dataclasses. Dies sorgt dafür, dass Pflichtfelder übergeben werden und nicht benötigte Elemente verworfen werden.
Über einen PyMongo Client werden diese Dataclasses also entweder von Route zur Datenbank weitergeleitet, oder aus der Datenbank zur Route. Die Verbindung erfolgt dabei über eine **MONGO_URI**, welche Hostnamen und Port beinhaltet. Dieser String sieht wie folgt aus: *mongodb://localhost:27017/.*
Als tatsächlicher Webserver wird **uvicorn** genutzt, welcher im **__main__.py** gestartet wird. Dies ermöglicht es, das Backend mittels des Befehlt *python -m backend* zu starten

D. Tests

In einem separaten Testscript **test_backend.py** werden die Routen getestet. Hierbei wird jedoch eine Testtabelle in der Datenbank genutzt, um nicht mit anderen Daten zu interferieren. In einer Setup-Methode werden dabei die Referenzen auf die Tabelle ausgetauscht. Der Server wird ebenfalls durch einen **TestClient** ersetzt, welcher in dem FastAPI Framework integriert ist.
Um die Endpoints für die Kategorien zu testen, werden mehrere Kategorien angelegt. Dabei sind diese teilweise hierarchisch auf der gleichen Ebene, jedoch auch teilweise untergliedert. Nach den Inserts wird überprüft, ob gewünschte Hierarchie mit der echten Hierarchie übereinstimmt. Nach Löschen der Kategorien wird getestet, dass keine Kategorie mehr in der Datenbank angelegt ist.
Das Testen der Artikel simuliert einen ähnlichen Objektzyklus, jedoch wird immer auch auf die passende Verbindung zur Kategorie geachtet. So soll beispielsweise nach dem Updaten eines Artikels, ebenfalls der Artikelname in der Verknüpfung angepasst werden.
Die Tests können mittels des Befehls *python -m pytest -s* gestartet werden.

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 4 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 4. 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 5 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 5. 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 6 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 6. 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