Podcast Management System "FLOSScaster"

WP-Seminar "Quelloffene Software in der modernen Informatik" Projekt

Frankfurt University of Applied Sciences

dkar001
dkar001@stud.fra-uas.de

Dominik
Dominik@stud.fra-uas.de

dwippah
dwippah@stud.fra-uas.de

Michael F
MichaelF@stud.fra-uas.de

Patryk
Patryk@stud.fra-uas.de

salat17
salat17@stud.fra-uas.de

Sascha Gab
Sascha@stud.fra-uas.de

Zundel, Benedikt
benedikt.zundel@stud.fra-uas.de

Friday, 2025-05-23

Contents

Listings

1 Technische Dokumentation

1.1 Backend

Als Kommunikationsschnittstelle mit dem Frontend haben wir uns für ein minimales restful *application programming interface* (API) entschieden. Dafür haben wir unterliegend `flask`, insbesondere `flask-restful`, verwendet. Diese API legt die wesentlichen Endpunkte, die verwendet werden, um Podcast Episoden abzufragen und neue zu erstellen, frei. Zum sichern der Episoden verwenden wir eine dynamische Lösung, mit der Metadaten in einer leichtgewichtigen `SQLite` Datenbank abgelegt werden und die zugehörigen Audiodaten im Dateisystem gespeichert werden, mit einem Verweis auf den Pfad in der Datenbank. Rückwirkend wird entsprechend der Eintrag aus der Datenbank geholt und die Datei aus dem Dateisystem geladen.

1.1.1 Modell

Das unterliegende Modell, welches wir zur internen Darstellung der Podcast Episoden verwenden, besteht aus einer `dataclass` mit wenigen Feldern (vgl. Listing 1). Sie enthält eine `id`, welche eine eindeutige Identifikation des Elements darstellt, die Felder `title`, `description` und `date` in denen Metadaten der Episoden liegen und das Feld `filepath`, in dem der interne Pfad zur Audiodatei gesichert wird. Eine `dataclass` war für unsere Zwecke ausreichend, da es keine Anforderungen für komplexe Operationen gibt und somit keine Funktionen innerhalb der Klasse nötig sind, weshalb das Modell als reine Datenklasse implementiert werden kann.

Listing 1: Implementation der Podcast-Episoden Klasse

```python
@dataclass
class Podcast:
    id: int
    title: str
    description: str
    date: str
    filepath: str
```

1.1.2 Endpunkt: `/api/list`

Der Endpunkt `/api/list` dient der Lieferung der vollständigen Auflistung aller Episoden und ist somit als `GET` Endpunkt definiert. Es wird kein Parameter im Aufruf erwartet. Intern wird bei einem Aufruf eine `SELECT *` SQL-Abfrage an die Datenbank gemacht. Das Ergebnis wird dann auf die Modellstruktur *gecastet* und als `JSONArray` als Antwort zurückgeschickt.

1.1.3 Endpunkt: `/api/create`

Bei dem Endpunkt `/api/create` handelt es sich um einen `POST` Endpunkt, der für das Erstellen neuer Episoden verantwortlich ist. Bei einem Aufruf werden die Parameter `title` & `description` erwartet, welche die Metadaten der zu erstellenden Episode repräsentieren, und die Audiodatei `audio`, welche dem `MP3`-Format entsprechen soll und im Dateisystem abgelegt wird. Die Datei wird auf die formalen Kriterien geprüft und bei Erfolg mit einer `UUID` versehen im Dateisystem gesichert um Kollisionen zu vermeiden. Anhand der übergebenen Metadaten und dem generierten Dateinamen wird ein Eintrag in die Datenbank angelegt. Sollten alle Operationen erfolgen, wird der neue Eintrag dem RSS-Feed angehangen und ein *Toot* von dem verbundenen Mastodon Account gesendet. Die Antwort an die aufrufende Instanz besteht aus dem Erfolgscode `200` mit der internen `id` des erstellten Eintrags.

1.1.4 Endpunkt: `/api/get_upload/<path>`

Dieser Endpunkt ist ein weiter `GET` Endpunkt, der dem Senden der Audiodatei aus dem Dateisystem dient. Als URL-Parameter wird der Dateiname der gewünschten Episode erwartet, der von dem Client aus dem `JSON`-Objekt gelesen wird. Mit Hilfe des Dateinamens und dem Wissen über das Verzeichnis, in dem die Audiodateien gespeichert sind, ließt die API die Datei aus dem Dateisystem und sendet diese als Antwort. Sollte die Datei nicht gefunden werden, Antwortet die API mit dem Fehlercode `404`.

1.1.5 Endpunkt: `/rss`

Zur Bereitstellung des RSS-Feeds wird der Endpunkt `/rss` verwendet, der, auf Basis der Umgebungsvariable, die den Pfad zur RSS-Feed Datei bestimmt, die Datei aus dem Dateisystem lädt und als Antwort sendet.

1.1.6 Dokumentation

Die Dokumentation der API (verfügbar unter dem `/apidocs` Endpunkt) ist im Code enthalten und wird durch `swagger` [1] zur Laufzeit bereitgestellt. Listing 2 ist ein Beispiel, welches die Dokumentation des `/api/list` Endpunkts bestimmt. Erkennbar ist, dass die Dokumentation in einem `YAML`-ähnlichen Format geschrieben wird. Dort lassen sich Schemata definieren (in diesem Fall die Struktur eines Podcast-Episoden Objekts), Rückgabecodes der API und Parameter, die für eine Abfrage relevant sind.

Listing 2: Dokumentationskommentar des `/api/list` Endpunkts

```
class List(Resource):
    def get(self):
        """Returns all podcasts metadata in the database
        ---
        definitions:
            Podcast:
                type: object
                properties:
                    id:
                        type: integer
                    title:
                        type: string
                    description:
                        type: string
                    date:
                        type: string
        responses:
            200:
                description: A list containing all podcasts
                schema:
                    type: array
                    items:
                        $ref: "#/definitions/Podcast"
                examples:
                    [
                      {
                        'id': 1,
                        'title': '6-Sekunden Podcast',
                        'description': 'Wir hatten keine Zeit um ein Thema anzusprechen.',
                        'date': '2025-04-22T20:00:00Z',
                      },
                      {
                        'id': 2,
                        'title': '6-Sekunden Podcast: Part 2',
                        'description': 'Wir hatten wieder keine Zeit um ein Thema
    anzusprechen.',
                        'date': '2025-04-23T18:00:00Z',
                      }
                    ]
        """
...
```

1.2 Dockerization

Um das Einrichten der Application auf einem Server zu vereinfachen, haben wir uns für die *dockerisierung* (i.e. Befähigung mit Docker zu bauen) entschieden. Das Projekt lässt sich in zwei primäre Kategorien teilen: Das Frontend und das Backend. Beide Teile benötigen einen `Dockerfile`, der die jeweiligen Teile als isolierte *Container* darstellt. Inhalt eines `Dockerfiles` ist eine Beschreibung des Bauvorgangs, der das Installieren von Abhängigkeiten, Kopieren von laufzeitrelevanten Dateien und Ausführen der Applikation beinhaltet. Um zu verhindern, dass mühselig zwei Container parallel gestartet werden müssen, haben wir

`docker-compose` zur *Container orchestrierung* eingesetzt. Mit `docker-compose` lassen sich multi-Container Anwendungen mit einem Kommando ausführen, indem ihr Verhalten in einer `YAML`-Datei definiert wird. Darin wird, im Falle dieses Projekts, beschrieben, welche Container gestartet, welche Ports offengelegt und welche Umgebungsvariablen verfügbar gemacht werden sollen. Anhand dieser Informationen werden die Container gebaut und isoliert gestartet. Eine direkte Kommunikation mit den Containern ist nur durch die freigegebenen Ports möglich, was zu einer erhöhten Sicherheit der Anwendung führt.

1.3 Deployment

Didn't deploy yet, not much to say.

References

[1] Swagger: Api documentation & design tools for teams. `https://swagger.io/`. (Accessed: 2025-05-17).