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

Sascha Gab
Sascha@stud.fra-uas.de

Zimmermann, Alwin
salat17@stud.fra-uas.de

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

Friday, 2025-05-23

Contents

Listings

1 Technische Dokumentation

1.1 Backend (Benedikt Zundel)

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 (Benedikt Zundel)

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 (Benedikt Zundel)

Didn't deploy yet, not much to say.

1.4 RSS-Feed Generator (Alwin Zimmermann)[1]

1.4.1 Ziel

Das Ziel des Generators ist es, dass ein sogenannter Podcatcher den RSS Feed unseres Podcasts einlesen kann. Der RSS Feed muss hierfür von unserem Server als .XML (oder .rss) bereitgestellt werden. Bei einem RSS Feed gibt es einige wichtige Punkte zu beachten, die für den Aufbau eines syntaktisch richtigen Feeds beachtet werden müssen. Eine XML-Datei hat eine Baumstruktur, im Kopf stehen Infos wie: XML-Version, Encoding, RSS-Version.

1.4.2 Struktur des RSS Feeds

Nach dem Kopf steht die Channel-Sektion, in der der Channel die wichtigsten Werte erhält. Diese wären:

- Titel
- Link zur Webseite
- Beschreibung
- Sprache
- Veröffentlichungsdatum
- Info, wann der Feed das letzte Mal erstellt/aktualisiert wurde

Ab diesem Bereich beginnen die Items, in denen die Infos über die neuen Podcast-Folgen hinzugefügt werden können. Jedes Item (jede Folge) hat:

- Titel
- Link zur Episode
- Beschreibung
- Enclosure URL
- Veröffentlichungsdatum

Nach allen Items wird der Channel geschlossen und der RSS Feed beendet.

[1] Ich erkläre hiermit, dass ich bei der Erstellung dieses Projektes Unterstützung von ChatGPT, einem KI-gestützten Sprachmodell von OpenAI, in Anspruch genommen habe. Alle Inhalte wurden von mir überprüft und bearbeitet.

1.4.3 Probleme

Das Paket `feedparser` [2] hat leider das Problem, dass es die sogenannte Enclosure URL nicht parsen kann. Diese ist jedoch essenziell für den Podcatcher, um die Datei finden zu können (Issue: #285 enclosure not parsed). Das Problem besteht nicht beim ersten Erstellen des Feeds, sondern erst beim Wiedereinlesen des Feeds. Eine mögliche Lösung hätte sein können, die Infos alle redundant zu speichern, jedoch habe ich diese Lösung schnell verworfen, da sie mit einem unverhältnismäßigen Aufwand verbunden wäre. Eine weitere Lösung könnte das Modul RSS in Ruby sein, jedoch wollte ich die Problematik in Python lösen, da die meisten Teile unseres Projekts bereits in Python geschrieben waren und ich selber bisher noch kein Ruby geschrieben habe. Deshalb habe ich mich für das Paket `lxml` [3] entschieden, welches ein generisches XML-Bearbeitungstool ist. Dieses ist zwar nicht speziell für RSS entworfen, sollte jedoch manuell in der Lage sein, den Feed entsprechend anzupassen.

Listing 3: Feedparser Bug

```
def load_existing_feed(self):
  """Laedt den bestehenden RSS-Feed, falls vorhanden."""
  if os.path.exists(self.feed_file_path):
      feed = feedparser.parse(self.feed_file_path)
      for entry in feed.entries:
          item = {
              'title': entry.title,
              'description': entry.description,
              'date': entry.published if 'published' in entry else
    datetime.now().isoformat(),
              'enclosure_url': entry.enclosure.href if 'enclosure' in entry else None,
              'enclosure_type': entry.enclosure.type if 'enclosure' in entry else None,
              'enclosure_length': entry.enclosure.length if 'enclosure' in entry else None
          }
          self.items.append(item)
          print(self)
```

1.4.4 Das Script

Das Script ist ein einfaches Python-Modul, das eine Funktion zum Hinzufügen von Podcasts in den Channel ermöglicht. Wichtig ist natürlich das Importieren der benötigten Python-Libraries.

Listing 4: RSS-Feed Generator imports

```
import os
import argparse
import datetime
from lxml import etree
```

Als Input-Werte benötigt es lediglich:

- einen Titel als String
- eine URL zu der Folge, die hinzugefügt werden möchte
- eine Beschreibung der Folge

Das Veröffentlichungsdatum wird als aktuelles Datum gewählt, da das Script automatisiert ausgeführt werden soll. Zudem wird der Audio-Typ auf MP3 festgelegt, da wir das FLAC-Format von Anfang an verworfen haben. Mithilfe von `lxml` wird nun ein neues `<item>` Objekt hinzugefügt mit den angegebenen Werten.

Listing 5: Erstellen eines neuen Feed-Items

```
# Erstelle ein neues Item
new_item = etree.Element('item')
etree.SubElement(new_item, 'title').text = title
etree.SubElement(new_item, 'description').text = description
etree.SubElement(new_item, 'pubDate').text = new_episode_pubDate
enclosure = etree.SubElement(new_item, 'enclosure')
```

```
enclosure.set('url', url)
enclosure.set('type', 'audio/mpeg')
```

Auch wird im Kopf das `lastBuildDate` auf den aktuellen Moment abgeändert. Zuletzt wird der neue XML-Feed dann in einen Ordner mit dem Namen `rss` abgespeichert und kann vom Server publiziert werden.

Listing 6: Vollständige Funktion des RSS-Feed Generators

```
def add_episode_to_podcast(title: str, url: str, description: str):
    # Verzeichnis und Dateiname
    directory = './rss'  # Aktuelles Verzeichnis mit dem Unterordner 'rss'
    filename = 'podcast.xml'  # Name der RSS-Datei

    # Vollstaendiger Pfad zur Datei
    file_path = os.path.join(directory, filename)

    # RSS-Feed laden
    with open(file_path, 'rb') as f:
        feed_content = f.read()

    # XML-Parser initialisieren
    root = etree.fromstring(feed_content)

    # Neue Episode hinzufuegen
    new_episode_pubDate = str(datetime.datetime.now())

    # Erstelle ein neues Item
    new_item = etree.Element('item')
    etree.SubElement(new_item, 'title').text = title
    etree.SubElement(new_item, 'description').text = description
    etree.SubElement(new_item, 'pubDate').text = new_episode_pubDate
    enclosure = etree.SubElement(new_item, 'enclosure')
    enclosure.set('url', url)
    enclosure.set('type', 'audio/mpeg')

    # Fuege das neue Item zum Channel hinzu
    channel = root.find('channel')
    channel.append(new_item)

    # Aktualisiere lastBuildDate
    last_build_date = channel.find('lastBuildDate')
    last_build_date.text = str(datetime.datetime.now())

    # Speichern des aktualisierten Feeds mit Zeilenumbruechen und Einrueckungen
    with open(file_path, 'wb') as f:
        f.write(etree.tostring(root, pretty_print=True, xml_declaration=True,
    encoding='UTF-8'))

    print(f'Die neue Episode "{title}" wurde erfolgreich zu {filename} hinzugefuegt.')
```

References

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

[2] Kurt McKee & Contributors. feedparser. https://github.com/kurtmckee/feedparser. (Accessed: 2025-05-18).

[3] scoder & Contributors. lxml. https://github.com/lxml/lxml. (Accessed: 2025-05-18).