

Podcast Management System "FLOSScaster"

WP-Seminar "Quelloffene Software in der modernen Informatik" Projekt
Unter Beaufsichtigung von Dr. Chris Zimmermann

Frankfurt University of Applied Sciences

Brychcy, Patryk
patryk.brychcy@stud.fra-uas.de

Fluegel, Dwipa
dwipa.fluegel@stud.fra-uas.de

Karaman, Deniz
deniz.karaman@stud.fra-uas.de

Zimmermann, Alwin
alwin.zimmermann@stud.fra-uas.de

Filatoff, Michael
michael.filatoff@stud.fra-uas.de

Gavrilov, Sascha
sascha.gavrilov@stud.fra-uas.de

Lepore, Dominik
dominik.lepore@stud.fra-uas.de

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

Friday, 2025-05-23

<https://github.com/bzundel/flosscaster>

Inhaltsverzeichnis

| | |
|---|-----------|
| 1 Projektmanagement mit Scrum & GitHub Projects (Deniz Karaman) | 4 |
| 1.1 Darstellung Boards | 4 |
| 2 Mockup-Design und Gestaltung (Patryk Brychcy & Michael Filatoff) | 5 |
| 2.1 Planung | 5 |
| 2.2 Aufbau und Layout | 5 |
| 2.2.1 Hero-Sektion | 5 |
| 2.2.2 Latest Episodes | 6 |
| 2.2.3 Updates | 6 |
| 2.2.4 Umsetzung & Endergebnis | 6 |
| 2.2.5 Verwendete Designprinzipien | 7 |
| 3 Technische Dokumentation | 7 |
| 3.1 Frontend (Michael Filatoff) | 7 |
| 3.1.1 Komponenten | 8 |
| 3.1.2 Styling (Sascha Gav) | 9 |
| 3.2 Backend (Benedikt Zundel) | 10 |
| 3.2.1 Modell | 10 |
| 3.2.2 Endpunkt: /api/list | 11 |
| 3.2.3 Endpunkt: /api/create | 11 |
| 3.2.4 Endpunkt: /api/get_upload/<path> | 11 |
| 3.2.5 Endpunkt: /rss | 11 |
| 3.2.6 Dokumentation | 11 |
| 3.3 Dockerization (Benedikt Zundel) | 12 |
| 3.4 Deployment (Benedikt Zundel) | 12 |
| 3.4.1 Rekonfiguration der Container | 12 |
| 3.4.2 Nginx Konfiguration | 13 |
| 3.5 RSS-Feed Generator (Alwin Zimmermann) | 13 |
| 3.5.1 Ziel | 13 |
| 3.5.2 Struktur des RSS Feeds | 14 |
| 3.5.3 Probleme | 14 |
| 3.5.4 Das Script | 15 |
| 3.6 Publikation auf einem Social Media Feed (Dwipa Flügel) | 16 |
| 3.6.1 Ziel | 16 |
| 3.6.2 Auswahl und Entscheidung | 16 |
| 3.6.3 Zugriff auf die API | 16 |
| 3.6.4 Probleme | 16 |
| 3.6.5 Gelerntes | 17 |
| 4 Testbericht (Dominik Lepore) | 17 |
| 4.1 Zielsetzung | 17 |
| 4.2 Teststruktur | 17 |
| 4.3 Test-Fixtures | 18 |
| 4.4 Tests | 18 |
| 4.4.1 /api/create Testen | 18 |
| 4.4.2 /api/list Testen | 19 |
| 4.4.3 GetFileByFileName Testen | 20 |
| 4.4.4 GetRSS Testen | 20 |
| 4.5 Testplan | 21 |
| 4.6 Static Code-Analyse (Deniz Karaman) | 21 |

Abbildungsverzeichnis

| | | |
|---|--|---|
| 1 | Screenshot der GitHub Projects Ansicht des Backlogs | 4 |
| 2 | Screenshot der GitHub Projects Ansicht der Quality Assurance | 4 |
| 3 | Screenshot der GitHub Ansicht einer Story | 5 |
| 4 | Figma Design der Landingpage | 6 |
| 5 | Figam Design des Upload-Dialogs | 7 |
| 6 | Implementiertes Design der Landingpage | 7 |
| 7 | Implementiertes Design der Podcast-Episoden | 8 |

Listings

| | | |
|----|---|----|
| 1 | Implementation des Hero-Bilds | 8 |
| 2 | Implementation der Navbar | 8 |
| 3 | Implementation der Podcast-Episoden Klasse | 10 |
| 4 | Dokumentationskommentar des <code>/api/list</code> Endpunkts | 11 |
| 5 | Unterschiede im Backend Dockerfile | 12 |
| 6 | Unterschiede im Frontend Dockerfile | 13 |
| 7 | <code>Nginx proxy pass</code> Konfiguration | 13 |
| 8 | Feedparser Bug | 14 |
| 9 | RSS-Feed Generator imports | 15 |
| 10 | Erstellen eines neuen Feed-Items | 15 |
| 11 | Vollständige Funktion des RSS-Feed Generators | 15 |
| 12 | Vollständige Funktion des <i>Autootootings</i> | 17 |
| 13 | Beispiel einer Fixture | 18 |
| 14 | Test-Implementation von <code>/api/create</code> | 18 |
| 15 | Implementation der <code>client</code> -Fixture | 19 |
| 16 | Test-Implementation von <code>/api/list</code> | 19 |
| 17 | Test-Implementation von <code>/api/list</code> gegen eine leere Datenbank | 19 |
| 18 | Test-Implementation von <code>/api/get_upload</code> | 20 |
| 19 | Test-Implementation von <code>/rss</code> | 20 |
| 20 | Fixture zum löschen des RSS-Feeds | 20 |
| 21 | Test-Implementation von <code>/rss</code> bei nicht vorhandenem Feed | 21 |

1 Projektmanagement mit Scrum & GitHub Projects (Deniz Karaman)

Um unseren Entwicklungsprozess agil zu strukturieren, haben wir Scrum mithilfe von GitHub Projects implementiert. Dabei haben wir uns an klassischen Scrum-Prinzipien orientiert und diese wie üblich an unsere TeamgröSSe und die Projektanforderungen angepasst. Jede Story wird dabei als Issue angelegt.

Zunächst haben wir in unseren Issues sowohl Labels als auch Milestones definiert, um einen schnellen Überblick zu gewährleisten und einzelne Issues entsprechend ihrer Zuordnung hervorzuheben. Da wir unsere Stories im Team abgestimmt und deren Komplexität berücksichtigt haben, gelten die Definition of Done (DoD) sowie die Bewertung als feststehend.

Folgende Labels haben wir definiert: *backend, bug, documentation, enhancement, epic, frontend, story, testing*. Zusätzlich zu unseren Labels benötigen wir unterschiedliche Status, für die wir folgende verwenden: *Todo, In Progress, Needs Review, Done, Canceled*.

Durch die Nutzung von Custom Fields vom Typ Iteration können wir unsere Sprints zeitlich organisieren und in wöchentlichen Zyklen abbilden. Bei der Sprintplanung in unseren Meetings haben wir zudem die Möglichkeit, sowohl die Story Points als auch die Priorität (*Low, Medium, High, Critical*) über Custom Fields abzubilden.

Nach Abschluss eines Sprints könnte eine formale Retrospektive stattfinden. Da wir jedoch durch unsere flexiblen Arbeitszeiten agil bleiben und unsere Sprints fortlaufend anpassen wollten, haben wir auf eine feste Retrospektive verzichtet.

1.1 Darstellung Boards

Abbildung 1: Screenshot der GitHub Projects Ansicht des Backlogs

| Title | Assignees | Status | Sprints | Story Points | Priority |
|------------------------------------|---------------------|--------------|-----------|--------------|----------|
| 1 Implement a Testplan #12 | sgedodo | Needs review | Sprints 3 | 1 | Medium |
| 2 Implement testing for backend #2 | sgedodo | Todo | Sprints 2 | 2 | High |
| 3 Frontend view to consume API #4 | Nennmicha, Patss... | In Progress | Sprints 1 | 3 | Critical |

Um die Übersichtlichkeit weiter zu erhöhen, haben wir für jede Board-Ansicht vordefinierte Filter konfiguriert.

So erhält jede Ansicht genau die Issues, die für den jeweiligen Arbeitsschritt relevant sind, und das Team behält jederzeit den passenden Fokus.

Abbildung 2: Screenshot der GitHub Projects Ansicht der Quality Assurance

| Title | Assignees | Status | Sprints | Story Points | Priority |
|----------------------------|-----------|--------------|-----------|--------------|----------|
| 1 Implement a Testplan #12 | sgedodo | Needs review | Sprints 3 | 1 | Medium |

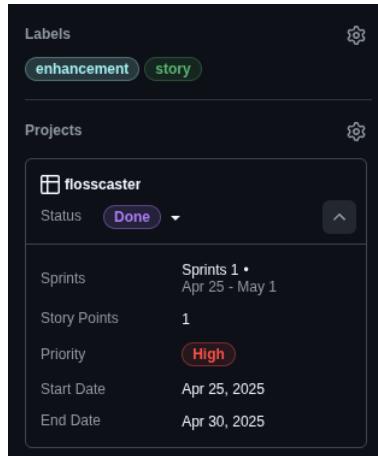
- **Backlog:** Alle geplanten User Stories und Tasks
- **Current Sprint:** Die aktuell bearbeiteten Items
- **Next Sprint:** Vorgemerkte Stories für den nächsten Sprint
- **Quality Assurance:** Abgenommene Features in der Testphase
- **Done:** Abgeschlossene Stories

Jeder Eintrag auf dem Board ist als GitHub Issue angelegt und hat folgende Felder

1. Story Points (zur Aufwandsschätzung in Fibonacci)

2. Priority-Labels (*Low, Medium, High, Critical*)
3. Sprint-Labels (z. B. *Sprint1, Sprint2*, etc.)
4. Status-Labels (*Todo, In Progress, Needs review, Done, Canceled*)

Abbildung 3: Screenshot der GitHub Ansicht einer Story



Durch diese enge Verzahnung von Issue-Tracking, Code-Repositories und agilen Meetings erreichen wir eine hohe Transparenz, schnelle Feedback-Zyklen und eine stetige Verbesserung unseres Workflows.

2 Mockup-Design und Gestaltung (Patryk Brychcy & Michael Filatoff)

2.1 Planung

In einem ersten Planungsschritt entschieden wir gemeinsam, dass die gesamte Funktionalität auf einer einzigen HTML-Webseite umgesetzt werden soll. Die Seite sollte so einfach, übersichtlich und benutzerfreundlich wie möglich gestaltet werden.

Daraufhin haben wir mithilfe des Online-Design-Tools *Figma* das erste Mockup erstellt. Dieses visuelle Konzept stellt eine minimalistische und funktionale Benutzeroberfläche dar, die sich auf die wesentlichen Elemente konzentriert.

Die Webseite wurde dabei so angelegt, dass es flexibel ausbaufähig bleibt, das *Flosscaster*-Logo oben am Rand ist eine Navigationsleiste (Navbar), wo man auch in der Zukunft eine Benutzerverwaltung integrieren könnte.

Die Webseite soll den Nutzern folgendes ermöglichen:

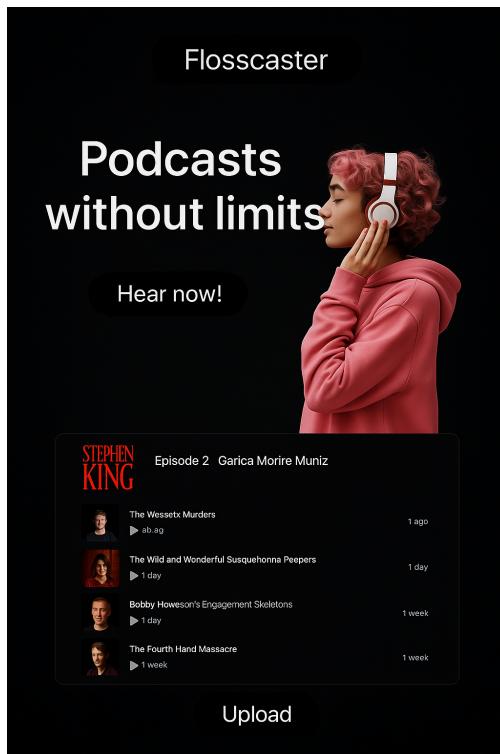
- Neue Podcast-Folgen in einem klar strukturierten Umfeld zu entdecken
- Eigene Podcasts hochzuladen über einen gut sichtbaren Upload-Button
- Intuitives Fenster für das Hochladen der Podcasts

2.2 Aufbau und Layout

2.2.1 Hero-Sektion

- **Titel & Claim:** Podcasts without limits in groSSer, fetter Schrift als Blickfang
- **Call-to-Action-Button:** Handlungsaufforderung als Hear now-Button direkt unter dem Text
- **Visuelles Element:** Ein Porträt einer Person mit Kopfhörern vor einem Pink-Gradient-Hintergrund, das für Emotionalität und Wiedererkennbarkeit sorgt

Abbildung 4: Figma Design der Landingpage



2.2.2 Latest Episodes

- **Überschrift:** Deutlich hervorgehoben mit weiSSer, serifloser Schrift
- **Upload-Button:** Oben rechts platziert, damit der User mit Leichtigkeit neue Podcasts hochladen kann.
- **Episoden-Karten:**
 - Abgerundete Ecken und leicht abgesetzter Hintergrund (dunkles Anthrazit)
- **Jede Karte enthält:**
 - Titel der Episode in fetter, weiSSer Schrift
 - Kurzbeschreibung in grauer Schrift (mehrzeilig, lesefreundlich)
 - Audioplayer mit Play/Pause-Funktion, Lautstärkeregelung und Fortschrittsbalken

2.2.3 Updates

Das Design wurde im Laufe der Tests in mehreren Punkten überarbeitet.

- Die Buttons sind pink und der Hero-Bereich hat pinke Akzente
- Die Liste mit Podcasts wurde nach unten versetzt und der Hear Now-Button scrollt nach unten auf die Podcasts-Liste
- Das Upload-Fenster wurde überarbeitet.

2.2.4 Umsetzung & Endergebnis

Durch gezielte Anpassungen in den CSS-Dateien konnte das Design final umgesetzt werden. Das Ergebnis ist eine schlichte, funktionale und optisch ansprechende Webseite.

Abbildung 5: Figam Design des Upload-Dialogs

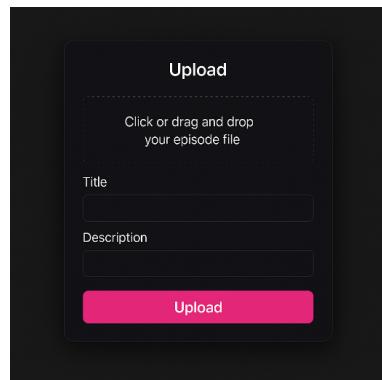
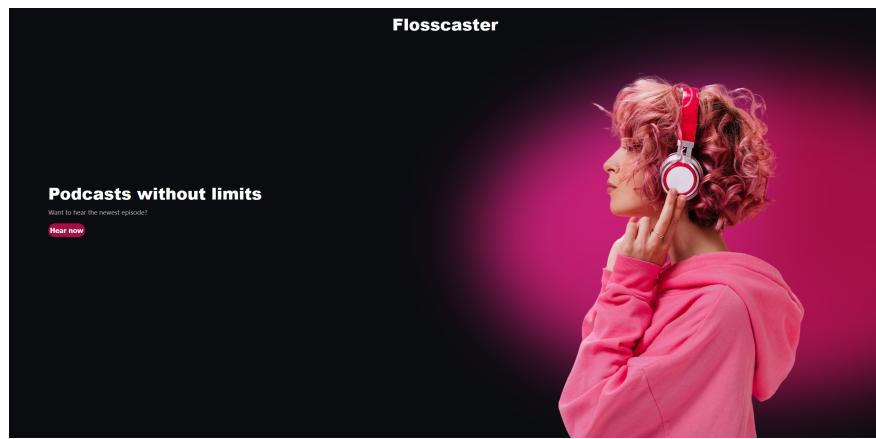


Abbildung 6: Implementiertes Design der Landingpage



2.2.5 Verwendete Designprinzipien

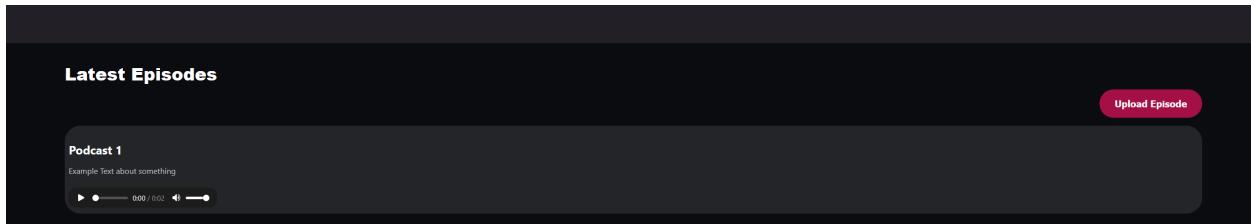
- **Dark Mode** mit pinken Akzenten
- **Akzentfarbe Pink** (#D81B60) für interaktive Elemente
- **WeiSSer Text** für Überschriften zur klaren Strukturierung
- **Grauer Text** für FlieSStexte und sekundäre Informationen
- **Abgerundete Buttons & Karten**
- **Intuitive Buttons** mit klarer Beschriftung
- **Font Smoothing**: antialiased für bessere Textdarstellung
- **Smoothed Interface** für visuelle Harmonie und Benutzerfreundlichkeit
- **Hover-Effekte** für die Buttons und das Navbar
- **Verzicht auf überflüssige Elemente**

3 Technische Dokumentation

3.1 Frontend (Michael Filatoff)

Das Frontend haben wir mit den folgenden drei Kerntechnologien erstellt. HTML5 wurde für die Struktur verwendet, CSS3 für das Styling und JavaScript für die Funktionalität. Grundlegend wurde die gesamte UI

Abbildung 7: Implementiertes Design der Podcast-Episoden



mit der populären JavaScript-Bibliothek `React` zur Erstellung verwendet. Da `React` den Entwicklungsprozess vereinfacht und strukturiert, basiert unsere Architektur auf der *Komponenten basierte Architektur*, die es uns ermöglicht, wiederverwendbare UI-Komponente zu erstellen. Außerdem benutzen wir die Zustandsmanagementfunktionen, welche Funktionen wie `useState`, `useEffect`, etc... anbieten. Diese helfen, die Daten, welche wir von dem API-Endpoint aus dem Backend erhalten, zu speichern und zu verwenden.

3.1.1 Komponenten

Grundlegend sind die Komponenten mit einer `.jsx` und `.module.css` Datei ausgestattet. Das `JSX` ist eine Erweiterung von JavaScript, die es ermöglicht, HTML Code innerhalb von JavaScript-Dateien zu schreiben. Die CSS Struktur basiert auf das Modul-Prinzip, wo Klassen und Namen nur lokal und nicht global gültig sind. Diese sind erst mit dem Import der CSS-Datei gültig.

3.1.1.1 Bar

Die Bar-Komponente dient als Balken der für die Trennung zwischen der HeroSection und der Podcastlist-Section fungiert.

3.1.1.2 Hero

Die Hero-Komponente enthält eine Sektion die aus einem Header, Paragraphen und einen Button besteht. Für den Button wurde der HTML tag `input` mit dem Typ `button` verwendet, welcher die `Id` von dem `div` aus der Datei `podcastLists` entnimmt und dorthin navigiert. Zusätzlich in der Hero-Sektion ist ein großes Bild, das sogenannte Hero-Bild.

Listing 1: Implementation des Hero-Bilds

```
1 
```

3.1.1.3 Navbar

Die Navbar-Komponente beinhaltet ausschließlich ein HTML tag `a`, welcher als Titel der Website dient. Zusätzlich besitzt der tag den `href` (hypertext reference), welcher leer ist und nach jedem Klicken wird man wieder zurück an den Start der Website navigiert.

Listing 2: Implementation der Navbar

```
1 <a href="/" className={styles.mainHeader}>
2   Flosscaster
3 </a>
```

3.1.1.4 Podcast List

Die `Podcastlist`-Komponente ist dafür zuständig, eine Liste von Podcast-Episoden anzuzeigen und dem User die Möglichkeit zu geben, neue Episoden hochzuladen. Diese interagiert mit dem Backend, um Episoden abzurufen und neue Episoden zu speichern.

Folgende Funktionalitäten sind in der `podcast list` enthalten:

- **State Management:** Die Komponente verwendet React Hooks, wie z.B `useState` oder `useEffect`, um den Zustand von Variablen zu verwalten und speichern.
- **Wichtige Funktionen:**
 - `handleOpen()` / `handleClose()`: Funktionen, um das Upload-Formular ein- bzw. auszublenden.
 - `handleSubmit(event)`: Wird beim Absenden des Upload-Formulars ausgelöst. Bereitet die Daten als `FormData` vor und sendet die per POST-Methode an den Backend-Endpunkt `/api/create`. Zum Schluss setzt er die Formularfelder zurück aktualisiert die Episodenliste bei Erfolg.
 - `WorkspaceEpisodes()`: Ruft die Liste der Episoden vom Backend-Endpunkt `/api/list` ab und aktualisiert den `episodes`-Zustand mit den empfangenen Daten.
- **Aufbau der UI:** Eine Überschrift, einen Button der bedingt das UploadFormular anzeigen lässt. Dieser besteht aus drei Eingabefeldern mit Titel, Beschreibung und eine Audio Datei, die mit einem *Submit Button* versendet wird.
- **Aufbau der Podcast-Episoden UI:** Jede Episode wird als *Karte* `podcastCard` dargestellt. Die zeigt den Titel, die Beschreibung und die abspielbare Audio Datei, vom Backend-Endpunkt `/api/getuploadstream`.
- **Backend-Kommunikation:** Die Basis-URL für Backend-Anfragen wird über die Umgebungsvariable `REACT_APP_BACKEND_HOST` konfiguriert oder verwendet standardmäßig `localhost`. Die Kommunikation erfolgt über den Port 1111.

3.1.1.5 App

Die App-Komponente ist die Hauptkomponente. Sie definiert die grundlegende Struktur und das Layout der Webseite, indem sie verschiedene wiederverwendbare Unterkomponenten importiert und anordnet.

3.1.2 Styling (Sascha Gav)

3.1.2.1 module.css

Die Verwendung von `.module.css` ist eine spezielle Methode, um sicherzustellen, dass die in dieser Datei definierten Stile nur auf diesem spezifischen Teil der Webseite angewendet werden und nicht versehentlich das Aussehen anderer Teile ändern. `styles` wird zu einem Objekt, das wir verwenden können, um diese spezifischen Stile anzuwenden.

3.1.2.2 index.css

Die Datei `index.css` stylt die gesamte Anwendung. Es wird einen dunklen Hintergrund gesetzt, Browser-Default-Abstände entfernt, sanftes Scroll-Verhalten bei *In-Page-Links* aktiviert und ein globales *Font-Fallback* (System- und Webfonts) definiert.

3.1.2.3 Bar.module.css

Die Bar-Komponente `bar.module.css` fungiert in unserem Layout als Platzhalter und Trennelement zwischen dem Hero-Bereich und der Episoden-Liste. Ihre zentrale Klasse `.container` sorgt für einen halb-transparenten Hintergrund mit abgerundeten Ecken und genug Innenabstand.

3.1.2.4 Navbar.module.css

Das Navbar-Modul `navbar.module.css` definiert das Styling der Navigationsleiste. Die Klasse `.navbar` richtet den Navigations-Container als Flexbox aus, zentriert alle enthaltenen Elemente horizontal und fügt oben einen Abstand von 30 Pixeln hinzu. Innerhalb dieses Containers übernimmt die Klasse `.mainHeader` die Gestaltung des Textes: grossSe, fette Schrift in Weiß, keine Unterstreichung und ein sanfter Hover-Effekt, der die Schrift minimal vergrößert und den Hand-Cursor anzeigt, wenn man drauf zeigt.

3.1.2.5 Hero.module.css

Im Hero-Modul `hero.module.css` wird der Fullscreen-Bereich an oberster Stelle umgesetzt. Die Klasse `.container` teilt den Bereich mittels Flexbox in einen linken Textbereich und einen rechten Bildbereich, beide in voller Browserhöhe. Der linke Bereich, also der Textbereich `.content` stapelt Überschrift, Untertitel und *Hear now*-Button vertikal und zentriert sie auf der linken Seite, getrennt durch etwas Abstand zum Bild. Die Klassen `.mainHeader` und `.mainParagraph` legen Schriftart, Farben und Abstände für Titel und Untertitel fest. Der Button (`.heroButton`) erhält eine abgerundete Form, eine passende Farbe und ein sanftes Leuchten beim Hover. Das Bild selbst `.mainPicture` füllt seinen Bereich ohne Verzerrung aus. Dekorative Blur-Elemente im Hintergrund `.bottomBlur` erzeugen mit großer Fläche und Unschärfe einen modernen Tiefeneffekt.

3.1.2.6 PodcastLists.module.css

Schließlich beschreibt das PodcastLists-Modul `podcastLists.module.css` das Design der Episodenübersicht und des zugehörigen Upload-Fensters. `.container` stapelt die einzelnen Podcast-Karten vertikal mit gleichmäßiger Abstand. Eine Überschrift im Stil der Klasse (`.mainHeader`) führt in die Listen-Section ein. Jede Karte `.podcastCard` erhält durch halbtransparente Hintergrundfarbe, Unschärfe (Glas-Effekt), stark abgerundete Ecken und ein leichtes Skalieren bei Hover einen modernen Look. Innerhalb der Karten definieren die Klassen `.podcastTitle` und `.podcastDescription` die Schriftgrößen, die Schriftdicke und Abstände für Titel und Beschreibung. Der Audio-Player wird über eine eigene Klasse (`audio`) leicht abgerundet. Die Button Klasse `.showUploadBtn` gestaltet den *Upload Episode*-Button, der das Upload-Fenster öffnet. Er ist in der Akzentfarbe der Webseite gestaltet und zeigt beim Hover einen Schatten und einen gedrückten Zustand beim Klick. Das Upload-Fenster selbst wird von der Klasse `.top` komplett im Viewport zentriert und erhält eine dunkle, fast deckende Hintergrundfarbe, passend zum Stil der restlichen Webseite, und feste Breiten- und Höhenbegrenzung. Die Textfelder im Fenster sind vollbreit, mit hohem Kontrast und ausreichendem Innenabstand definiert.

Abschließend stylt `.submit` den *Submit*-button in Akzentfarbe und `.close` positioniert das Schließen-Icon oben rechts im Uploadfenster.

3.2 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.

3.2.1 Modell

Das unterliegende Modell, welches wir zur internen Darstellung der Podcast Episoden verwenden, besteht aus einer `dataclass` mit wenigen Feldern (vgl. Listing 3). 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 3: Implementation der Podcast-Episoden Klasse

```
1 @dataclass
2 class Podcast:
3     id: int
4     title: str
5     description: str
6     date: str
7     filepath: str
```

3.2.2 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.

3.2.3 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 angehängt 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.

3.2.4 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.

3.2.5 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.

3.2.6 Dokumentation

Die Dokumentation der API (verfügbar unter dem `/apidocs` Endpunkt) ist im Code enthalten und wird durch `swagger` [1] zur Laufzeit bereitgestellt. Listing 4 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 4: Dokumentationskommentar des `/api/list` Endpunkts

```
1 class List(Resource):
2     def get(self):
3         """Returns all podcasts metadata in the database
4         ---
5         definitions:
6             Podcast:
7                 type: object
8                 properties:
9                     id:
10                         type: integer
11                     title:
12                         type: string
13                     description:
14                         type: string
15                     date:
16                         type: string
17         responses:
18             200:
19                 description: A list containing all podcasts
20                 schema:
```

```

21     type: array
22     items:
23       $ref: "#/definitions/Podcast"
24   examples:
25   [
26     {
27       'id': 1,
28       'title': '6-Sekunden Podcast',
29       'description': 'Wir hatten keine Zeit um ein Thema anzusprechen.',
30       'date': '2025-04-22T20:00:00Z',
31     },
32     {
33       'id': 2,
34       'title': '6-Sekunden Podcast: Part 2',
35       'description': 'Wir hatten wieder keine Zeit um ein Thema
anzusprechen.',
36       'date': '2025-04-23T18:00:00Z',
37     }
38   ]
39   """
40 ...

```

3.3 Dockerization (Benedikt Zundel)

Um das Einrichten der Applikation 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 Laufzeit relevanten 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.

3.4 Deployment (Benedikt Zundel)

Die Migration des Projekts in eine Produktionsumgebung war, trotz Einsatzes von Docker, nicht trivial. Als Hostingplattform bat sich ein Linux VPS mit Debian GNU/Linux 12 an, da dies mein persönlicher Server ist, auf dem das Standardwerkzeug wie Docker und eine Nginx reverse-proxy bereits installiert und konfiguriert ist. Ein naives Klonen des Repositorys und starten der Docker Container war zunächst erfolgreich und das Frontend war auch erreichbar, jedoch wies der Browser einige SSL-Probleme auf, da die Entwicklungsserver von Flask und React keine gute, native Unterstützung für verschlüsselte Verbindungen haben, ich jedoch ein SSL-Zertifikat von *Let's Encrypt* einrichtete. Es folgte also eine Migration auf **unicorn** im Falle des Backends und **serve** im Falle des Frontends.

3.4.1 Rekonfiguration der Container

3.4.1.1 Gunicorn

Der Kontrast des Bau- und Ausführprozesses zwischen Entwicklungsumgebung und Produktionsumgebung hielt sich im Bereich des Backends gering. Die relevanten Änderungen beschränken sich auf das Installieren des **unicorn** Pakets mittels **pip** und dessen Verwendung zum starten der API, im Gegensatz zum Flask Entwicklungsserver.

Listing 5: Unterschiede im Backend Dockerfile

```

1 -CMD ["python", "src/main.py"]
2 +ENV PYTHONPATH=src
3 +

```

```
4 +CMD ["gunicorn", "--bind", "0.0.0.0:1111", "src.main:app"]
```

Die Umgebungsvariable PYTONPATH muss mitgeliefert werden, sodass die Module, auf die `src/main.py` verweist, gefunden werden.

3.4.1.2 Serve

Die Herangehensweise von `serve`, einem kleinen Webserver aus der `nodejs`-Welt, ist ein wenig anders. Es wird erwartet, dass statische Dateien angeboten werden, weshalb das React Projekt erst mittels `npm run build` gebaut wird. Dieser Prozess baut das Projekt in einen Ordner `build`, welcher dann zum hosten des Dienstes verwendet werden kann.

Listing 6: Unterschiede im Frontend Dockerfile

```
1 +RUN npm install -g serve
2 ...
3 -CMD ["npm", "start"]
4 +RUN npm run build
5 +
6 +CMD ["serve", "-s", "build", "-l", "3000"]
```

3.4.2 Nginx Konfiguration

Durch das hosten von Front- und Backend in Webservern, die einen Port freigeben, über die die Dienste erreichbar sind, statt direkt statische Dateien zu liefern, lag die Lösung via `Nginx` mehrere *proxy passes* einzurichten nahe. Mit dieser Konfiguration wird Verkehr, der an die eingerichtete Subdomain geschickt wird, an die entsprechenden Ports weitergeleitet.

Listing 7: Nginx proxy pass Konfiguration

```
1 server {
2     server_name flosscaster.bzun.de;
3
4     location / {
5         proxy_pass http://127.0.0.1:3000;
6     }
7
8     location /api {
9         proxy_pass http://127.0.0.1:1111;
10    }
11
12    location /rss {
13        proxy_pass http://127.0.0.1:1111/rss;
14    }
15
16    ... # ssl configurations
17 }
```

Aus Listing 7 ist zu entnehmen, dass Anfragen an den `/` Endpunkt an das Frontend, welches unter Port 3000 läuft, weiterzuleiten sind. Anfragen an den Endpunkt beginnend mit `/api` werden entsprechend an das Backend weitergeleitet und dort verarbeitet. Eine Sonderregelung hat der Endpunkt `/rss`, der einen direkten Endpunkt auf der API abruft, um die rohe RSS-Feed Datei zu liefern, die, beispielsweise, von einem Aggregatoren verwendet werden kann.

3.5 RSS-Feed Generator (Alwin Zimmermann)¹

3.5.1 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

¹Ich 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.

beachtet werden müssen. Eine XML-Datei hat eine Baumstruktur, im Kopf stehen Infos wie: XML-Version, Encoding, RSS-Version.

3.5.2 Struktur des RSS Feeds

Nach dem Kopf steht die Channel-Sektion, in der der Channel die wichtigsten Werte zum RSS feed speichert. 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 [6].

3.5.3 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 unverhältnismäßigigen 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` [4] 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 8: Feedparser Bug

```
1 def load_existing_feed(self):
2     """Laedt den bestehenden RSS-Feed, falls vorhanden."""
3     if os.path.exists(self.feed_file_path):
4         feed = feedparser.parse(self.feed_file_path)
5         for entry in feed.entries:
6             item = {
7                 'title': entry.title,
8                 'description': entry.description,
9                 'date': entry.published if 'published' in entry else
10                    datetime.now().isoformat(),
11                 'enclosure_url': entry.enclosure.href if 'enclosure' in entry else None,
12                 'enclosure_type': entry.enclosure.type if 'enclosure' in entry else None,
13                 'enclosure_length': entry.enclosure.length if 'enclosure' in entry else None
14             }
15             self.items.append(item)
16             print(self)
```

3.5.4 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 9: RSS-Feed Generator imports

```
1 import os
2 import argparse
3 import datetime
4 import pytz
5 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
- GröSSe der Datei in Bytes

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 10: Erstellen eines neuen Feed-Items

```
1 # Erstelle ein neues Item
2 new_item = etree.Element('item')
3 etree.SubElement(new_item, 'title').text = title
4 etree.SubElement(new_item, 'description').text = description
5 etree.SubElement(new_item, 'pubDate').text = new_episode_pubDate
6 enclosure = etree.SubElement(new_item, 'enclosure')
7 enclosure.set('url', url)
8 enclosure.set('length', length)
9 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 11: Vollständige Funktion des RSS-Feed Generators

```
1 def add_episode_to_podcast(title: str, url: str, description: str, length: str):
2     with open(RSS_FILE, 'rb') as f:
3         feed_content = f.read()
4
5     # XML-Parser initialisieren
6     root = etree.fromstring(feed_content)
7
8     # Neue Episode hinzufügen
9     new_episode_pub_date =
10    datetime.datetime.now(pytz.timezone('Europe/Berlin')).strftime('%a, %d %b %Y %H:%M:%S
11    %z')
12
13    # Erstelle ein neues Item
14    new_item = etree.Element('item')
15    etree.SubElement(new_item, 'title').text = title
16    etree.SubElement(new_item, 'description').text = description
17    etree.SubElement(new_item, 'pubDate').text = new_episode_pub_date
18    enclosure = etree.SubElement(new_item, 'enclosure')
19    enclosure.set('url', url)
20    enclosure.set('length', length)
21    enclosure.set('type', 'audio/mpeg')
22
23    # Füge das neue Item zum Channel hinzu
```

```

22     channel = root.find('channel')
23     channel.append(new_item)
24
25     # Aktualisiere lastBuildDate
26     last_build_date = channel.find('lastBuildDate')
27     last_build_date.text =
28     datetime.datetime.now(pytz.timezone('Europe/Berlin')).strftime('%a, %d %b %Y %H:%M:%S
29     %z')
30
31     # Speichern des aktualisierten Feeds mit Zeilenumbrüchen und Einrückungen
32     with open(RSS_FILE, 'wb') as f:
33         f.write(etree.tostring(root, pretty_print=True, xml_declaration=True,
34         encoding='UTF-8'))

```

3.6 Publikation auf einem Social Media Feed (Dwipa Flügel)

3.6.1 Ziel

Das Ziel ist es auf mindestens einem Social Feed eine Ankündigung zu veröffentlichen, wenn eine neue Folge des Podcastes veröffentlicht wird. Vorteilhaft sollte diese Ankündigung dabei den Titel und die URL des Podcastes erhalten, damit die Zuhörer einen kleinen Überblick bekommen, als auch die Möglichkeit bei Interesse direkt auf die neue Episode zuzugreifen über die URL.

3.6.2 Auswahl und Entscheidung

Es gibt eine groSSe Auswahl an Social Media Plattformen auf denen man seine Ankündigungen veröffentlichen kann, haben uns aber im Kontext dieses FLOSS-Projektes für Mastodon entschieden. Mastodon ist eine quelloffene dezentralisierte Social Media Plattform und Teil des "Fediverse", einem Netzwerk von Servern die unabhängig voneinander operieren, welche das ActivityPub-Protokoll benutzen um miteinander zu kommunizieren. Hierbei können sich Benutzer frei aussuchen auf welcher Instanz sie ihren Account erstellen und trotzdem mit anderen Instanzen interagieren. In unserem Falle haben wir uns für die Hauptinstanz mastodon.social entschieden, da diese Leuten frei erlaubt einen Account herzustellen und nicht Einladungs-basierend ist, als auch die Benutzung von Bots nicht untersagt ist; da jede Instanz ihre eigenen Regeln besitzt, sollten diese auch befolgt werden.

3.6.3 Zugriff auf die API

Um mit unserer Mastodon-Instanz zu kommunizieren wird eine API-Verbindung gebraucht, da wir uns entschieden haben das Backend unseres Projektes in Python zu schreiben, bietet sich hierfür das `Mastodon.py` Modul an, welches "feature complete" ist. Zudem werden für die Benutzung der API Zugriffsdaten benötigt, welche generiert werden indem man auf der ausgewählten Instanz eine Anwendung registriert. Hier empfiehlt es sich dies über die UI der Instanz zu machen, welche unter <https://mastodon.social/settings/applications> gefunden werden kann.

3.6.4 Probleme

Während meiner Forschung bezüglich der Möglichkeiten, wie man Podcast-Episoden automatisch ankündigt bin ich auf einen Blog-Post gestoSSen, der ein Utility-Tool vorstellt, bei dem neu hinzugefügte RSS-Items automatisch auf einen Mastodon-Account veröffentlicht werden [3]. In diesem wurde ein Utility-Tool vorgestellt, bei dem neu hinzugefügte RSS-Items automatisch auf einen Mastodon-Account veröffentlicht werden. Dadurch das der Podcast auf einem RSS-Feed aufgebaut ist, hatte mich dieser Ansatz neugierig gemacht und ich hatte vor unsere eigene Version zu programmieren, welche auf SQLite aufgebaut war und nicht auf Amazons DynamoDB. Hier wurde das Paket feedparser benutzt, welches wie oben von Alwin beschrieben gewisse Probleme mit sich brachte, die sich im Verlauf der Entwicklung problematisch herausgestellt hat. Das Umgehen des RSS-Feedes hatte sich dadurch Problematisch gemacht, da das Paket beim parsing ein Feed-ParserDictionary ausgibt, wodurch sich das sortieren des RSS-Feedes auf das neuste Item mittels `sorted()` als nicht möglich erwiesSS. Ein Ansatz, der hier funktioniert hätte, welcher vom Autor des obig genannten Blog-Postes benutzt wurde, war das kreieren einer eigenen Klasse um gewisse Hilffunktionen aufzurufen, welchen ich als unverhältnismäSSigen Aufwand ansah. Nachdem ich mir einige Gedanken gemacht hatte

und mir Alwins Code angeschaut habe, da er verantwortlich war für den RSS-Feed, musste ich mir eingestehen, dass das Design und die vorangehensweise sich als schlicht ineffizient herausgestellt hat. Dadurch das der Feed von uns verwaltet wird und die neuen Podcast-Episoden als neue Items in den Feed hinzufügt werden mit allen relevanten Informationen für das veröffentlichten eines Tootes, hatte es weitaus mehr Sinn ergeben, ein Skript zu bauen, welches den gleichen Input benutzt und direkt danach aufgerufen wird. Da wir hier mit Zugangsdaten arbeiten, sollte diese Informationen als Umgebungsvariablen gespeichert werden. Während des entwickeln hatte ich hierfür eine `.env`-Datei benutzt, welche ich natürlich in einer `gitignore` inkludiert hatte.

Listing 12: Vollständige Funktion des *Autootootings*

```

1 def masttoot(title: str, url: str, description: str):
2     """Publishes a new toot using the input, also sends a reply to the announcement with
3     the description."""
4
5     post_text = "The Flosscasters have released a new podcast episode: " + title + ". Check
6     it out @ " + url
7
8     #Mastodon - Instanz
9     mastodon = Mastodon(
10         client_id = os.getenv("MASTODON_CLIENT_KEY"),
11         client_secret = os.getenv("MASTODON_CLIENT_SECRET"),
12         access_token = os.getenv("MASTODON_ACCESS_TOKEN"),
13         api_base_url = os.getenv("MASTODON_BASE_URL")
14     )
15
16     #Publish the toot and reply to it with the description
17     to_reply = mastodon.status_post(post_text)
18     mastodon.status_post(description, in_reply_to_id=to_reply.id)

```

3.6.5 Gelerntes

Ich habe leider verhältnismäSSig viel Zeit investiert in ein Design, das sich nachträglich als ineffizient herausgestellt hat während der Entwicklung, wenn es eine weitaus einfache Lösung gab. Im Kontext eines Uni-Projektes sollte dies etwas weniger Konsequenzen haben, aber im zukünftigen Kontext der Industrie sind das redundante Personalkosten. Da ich selber bei einem IT-Dienstleister arbeite, jedoch im 1st-Level Support, ist dies für mich eine wertvolle Lektion, da es schwer ist bei einem Kunden gewisse Stunden zu rechtfertigen, wenn diese nicht effektiv genutzt werden.

4 Testbericht (Dominik Lepore)

4.1 Zielsetzung

Tests haben das Ziel, zu überprüfen, dass alle Funktionalitäten wie erwartet funktionieren. Um dies bei unserem Projekt umzusetzen, testen wir jeden von uns definierten REST-API-Endpunkt. Insgesamt haben wir vier API-Endpunkte definiert, die bereits in diesem Bericht detailliert dokumentiert wurden. Aus Gründen der Redundanz beschränken wir uns in diesem Abschnitt nur auf die Beschreibung der Test-Vorgehensweise und nicht auf die Funktionsweisen der Endpunkte selbst.

4.2 Teststruktur

Die Testumgebung inklusive aller Dateien und Konfigurationen befindet sich im Verzeichnis `/flosscaster/backend/tests`. Dieses Verzeichnis beinhaltet auch eine `README.md` Datei, mit einer Anleitungen zur Einrichtung und Ausführung der Tests.

Da wir die Python Bibliothek Flask zur Implementierung der API-Endpunkte benutzen, verwenden wir gemäSS derer Empfehlung die Funktionalitäten aus der Pytest Bibliothek zum Testen. Pytest bietet umfangreiche Test-Features wie Fixtures und die Parametrisierung. Dies ermöglichte uns eine strukturierte und übersichtliche Teststruktur aufzubauen, welche leicht angepasst oder erweitert werden kann.

4.3 Test-Fixtures

Alle Test-Fixtures haben wir in der Datei `conftest.py` hinterlegt. Das ist die von Pytest empfohlene Datei, damit die automatische Testerkennung fehlerfrei funktioniert.

Eine typische Fixture zur Initialisierung der Testumgebung sieht folgendermaßen aus:

Listing 13: Beispiel einer Fixture

```
1 @pytest.fixture()
2 def init_testing():
3     app.config.update({'TESTING': True})
4     if not os.path.exists(DATABASE_FILE):
5         con = sqlite3.connect(DATABASE_FILE)
6         cur = con.cursor()
7         cur.execute("CREATE TABLE IF NOT EXISTS podcasts(id INTEGER PRIMARY KEY, title,
8             description, date, filepath)")
9         con.close()
10    yield app
11    app.config.update({'TESTING': False})
```

Diese Fixture-Funktion kann von jedem Test aufgerufen werden. Sie initialisiert die Test-Datenbank und setzt die TESTING-Flag auf True. Für das Testen von Flask-Objekten sollte man die TESTING-Flag auf True setzen, um alle Test-Funktionen nutzen zu können. Fixtures haben den weiteren Vorteil, dass Sie nach der Ausführung des Tests noch Code ausführen kann. Das ist sehr nützlich, um für die Testumgebung spezifische Änderungen wieder rückgängig zu machen. In dem Code Beispiel nutzen wir direkt diese Funktionalität um die TESTING-Flag im Anschluss wieder zurückzusetzen. Für das Projekt wurden noch weitere Test Fixtures definiert, diese werde ich später, bei Ihrer Verwendung in den jeweiligen Tests, zeigen und erklären.

4.4 Tests

Die Tests werden in vier Kapiteln aufgelistet sein. Jedes Kapitel wird sich auf ein REST-API-Endpunkt beziehen. In jedem dieser Kapitel werden die Testmethoden genauer erläutert, wobei nicht nur die Funktionalität genannt wird, sondern auch die Beweggründe für die Gestaltung dieser Tests erklärt werden. Anschließend an den Kapiteln wird es auch ein Kapitel mit kleinen Testplänen geben, die nach James Whittakers Blogbeitrag The 10 Minute Test Plan gestaltet wurden [5].

4.4.1 /api/create Testen

Die Create-Funktion lässt uns Audiodateien als Podcasts hochladen. Wir müssen mit dem Testen sicherstellen, dass der Upload funktioniert hat und ob eine Fehlermeldung auftritt, wenn eine falsche Datei gesendet wird.

Listing 14: Test-Implementation von /api/create

```
1 @pytest.mark.parametrize("title, description, audio", [
2     ("First FLOSScast", "The very first flosscast", generate_mp3_tts("Hello to the very
3         first flosscast episode.", lang="en")),
4     (generate_random_ascii(length=10), generate_random_ascii(length=99),
5         generate_mp3_size(1024)),
6     (generate_random_ascii(length=24, letters=False, numbers=False, punctuation=True),
7         generate_random_ascii(length=240, punctuation=True), generate_mp3_duration(30))
8 ])
9
10
11 def test_create_podcasts(client, title, description, audio):
12     response = client.post("/api/create", content_type="multipart/form-data",
13                             data={
14         "title": title,
15         "description": description,
16         "audio": open(audio, 'rb')
17     })
18     assert response.status_code == 200
```

Für das Testen der Create-Funktion benutzen wir die Parametrisierung-Funktion, die Pytest uns zur Verfügung stellt. Die Parametrisierung umfasst die obere Hälfte des Codes und hat den Nutzen, dass wir für mehrere Inputs direkt testen können. Für zufällige Audiodateien habe ich Hilfsfunktionen geschrieben, die

sich unter `flosscaster/backend/tests/random_file_generator.py` einsehen lassen. Diese werden dann in der Parametrisierung erstellt und dann jeweils ihrer Variable zugewiesen. Somit haben wir im oberen Beispiel einen Datensatz mit dem man 3 mal testen kann. Die Variablen aus der Parametrisierung kann man dem Testen übergeben und die Tests werden dann mehrfach ausgeführt. In unserem Fall drei mal. Bei der Gestaltung des Tests ergab es sich als äußert schwierig, die Audiodatei als Datei zu übergeben. Ein Pfad zu der Datei allein genügt nicht. Nach längerer Bemühung haben wir eine Lösung gefunden, die sogar sehr schlank und schick ist.

Listing 15: Implementation der `client`-Fixture

```

1 @pytest.fixture()
2 def client(init_testing):
3     with app.test_client() as client:
4         yield client

```

Vielleicht ist es Ihnen als Leser aufgefallen, in der Testfunktion war neben den Variablen aus der Parametrisierung auch eine `client` Variable aufgeführt. Diese Variable ist ein Fixture, genauso eine Fixture wie wir Sie im letzten Kapitel gezeigt haben. Fixtures gibt man als Funktionsparameter an den Test weiter. Die `client` Fixture ist für das Testen von REST-API-Schnittstellen äußerst hilfreich. Sie ermöglicht es GET und POST Requests simpel aufzurufen. Eine POST Anfrage haben wir soeben beim Testen der Create Funktion gesehen und viele weitere GET Anfragen werden mit Hilfe von `client` in den nächsten Tests noch folgen.

Natürlich reicht es beim Testen nicht nur aus, den gewollten und erwarteten Ablauf zu testen, man möchte auch testen, dass eine falsche Nutzung der Funktion erkannt und richtig behandelt wird. Dafür findet man im Repository noch eine weitere Testfunktion bei der ich diesmal eine Datei erstellt habe, die keine Audiodatei ist. Der Test überprüft ob in diesem Fall der Status Code 400, der für einen Fehler steht, als Antwort kommt.

4.4.2 /api/list Testen

Die API-Schnittstelle List lässt sich um einiges einfacher als die zuvor beschriebene Create Schnittstelle, testen. List ist eine GET-Funktion, die keine Parameter annimmt. Als Antwort bekommt man eine JSON Liste mit allen Datenbankeinträgen. Zum Testen müssen wir nur 3 Szenarien überprüfen, um die gewünschte Funktionalität sicherzustellen. Diese drei Szenarien sind in drei Tests in der `test_list.py` beschrieben.

Listing 16: Test-Implementation von /api/list

```

1 def test_list(database_entry_id, client):
2     response = client.get("/api/list")
3     assert response.status_code == 200
4     assert f'\"id\":{database_entry_id}'.encode("utf8") in response.data
5     assert b'\"title\":\"' in response.data
6     assert b'\"date\":\"' in response.data
7     assert b'\"description\":\"' in response.data
8     assert b'\"filepath\":\"' in response.data

```

Zuallererst gehen wir von einem normalen Aufruf der Funktion auf. Der Nutzer erwartet eine Liste mit Informationen zu allen Podcasts als Antwort. Um dies zu testen, nutzen wir die `database_entry_id` Fixture. Sie stellt sicher das mindestens ein Eintrag in der Datenbank eingetragen ist und gibt deren `id` zurück. Damit lässt sich schon die Funktion testen, indem wir überprüfen, ob der Status-Code 200 übermittelt wird und ob alle Beschreibungen, die ein Podcast beinhaltet, übermittelt werden. Unter anderem überprüfen wir, ob der von unserer Fixture erstellte Eintrag vorhanden ist. Dies machen wir, indem wir für die `id` suchen, die von der `database_entry_id` Fixture übergeben wurde.

Beim zweiten Szenario übergeben wir falsche API-Aufrufe und erwarten den Status-Code 404. Wir hängen an den Aufruf einfach weitere Parameter an und checken den Status-Code.

Listing 17: Test-Implementation von /api/list gegen eine leere Datenbank

```

1 def test_list_empty_db(empty_database, client):
2     response = client.get("/api/list")
3     assert response.status_code == 200
4     assert b'[]' in response.data

```

Das dritte Szenario ist für den Fall, dass die Liste leer ist. Dazu nutzen wir wieder eine Fixture. Die `empty_database` Fixture erstellt, wie der Name schon hergibt, eine Datenbank ohne Einträge. Hier wollen wir für den Fall der Fälle überprüfen, ob sich die Funktion noch so verhält wie erwartet. Wir erwarten in diesem speziellen Fall eine leere JSON-Datei und den Status-Code 200, der für einen Erfolg des Funktionsaufrufes steht.

Mit diesen 3 getesteten Szenarien lassen sich so ziemlich alle Funktionalitäten der `List` Funktion garantieren.

4.4.3 `GetFileByFileName` Testen

Die `getFileByFileName` Funktion sucht nach einer Datei, die mit dem an die Funktion übermittelten Namen übereinstimmt. Wir testen die zwei Szenarien, wenn ein richtiger Dateiname übergeben wird und wenn nicht.

Listing 18: Test-Implementation von `/api/get_upload`

```

1 def test_getFileByFilename(upload_file, client):
2     response = client.get(f"/api/get_upload/{upload_file}")
3     assert response.status_code == 200
4
5     assert "audio/mpeg" in response.headers["content-type"]

```

Für das Testen ist eigentlich nur der erste Fall interessant, weil man mit einer Fixture erstmal die Datei im richtigen Verzeichnis erstellen muss und anschließend muss auch überprüft werden, ob die Datei eine Audiodatei ist. Um eine korrekte Ausführung sicherzustellen, erwarten wir den Status Code 200 und das die übermittelte Datei vom `content-type` `audio/mpeg` ist.

Im zweiten Fall, bei dem ein falscher Dateiname übermittelt wird, überprüfen wir nur, ob der Status Code 404 als Antwort kommt.

4.4.4 `GetRSS` Testen

Die `GetRSS` API-Schnittstelle ist nochmal eine Funktion, bei der nur die 2 Szenarien getestet werden, ob die angefragte Datei existiert oder nicht. Obwohl sich dieser Testablauf mit dem Testen von `GetFileByFileName` sehr ähnelt, hat das Programmieren dieser Test um einiges mehr Spaß gemacht.

Listing 19: Test-Implementation von `/rss`

```

1 def test_getrss(create_rss, client):
2     response = client.get("/rss")
3     assert "application/xml" in response.headers["content-type"]
4     assert response.status_code == 200 or response.status_code == 304

```

Für den Testfall der korrekten Ausführung der Schnittstelle müssen wir mehr als nur den Status Code 200 überprüfen. Auch der Status Code 304 sagt eine erfolgreiche Ausführung aus, da dieser Code dafür steht, dass sich die Datei seit der letzten Anfrage nicht verändert hat. Außerdem mussten die Fixtures sicherstellen, dass eine RSS Datei existiert und gegebenenfalls eine neue erstellen. Zuallerletzt überprüfen wir auch, dass die Datei tatsächlich eine RSS Datei ist, dies machen wir in dem wir den `content-type` Header überprüfen. RSS Dateien haben den `content-type` `application/xml`.

Listing 20: Fixture zum löschen des RSS-Feeds

```

1 @pytest.fixture()
2 def delete_rss(init_testing):
3     rss_path = os.getenv("RSS_FILE")
4     if os.path.exists(rss_path):
5         file = open(rss_path, "r")
6         rss_file = file.read()
7         file.close()
8         os.remove(rss_path)
9         return rss_file
10    return "failure"

```

Für den letzten Fall will ich nochmal eine Fixture zeigen. Diese Fixture löscht die vorhandene RSS-Feed Datei, aber speichert vorher den Inhalt ab. Dieser Inhalt wird dem Test, der unten auch aufgeführt ist, übergeben, damit er im Nachhinein die RSS Datei wiederherstellt.

Listing 21: Test-Implementation von /rss bei nicht vorhandenem Feed

```

1 def test_getrss_no_file(delete_rss, client):
2     response = client.get("/rss")
3     assert response.status_code == 404
4     if not delete_rss == "failure":
5         rss_path = os.getenv("RSS_FILE")
6         with open(rss_path, "w") as file:
7             file.write(delete_rss)

```

4.5 Testplan

| Test-Name | Attributes | Components | Capabilities |
|------------------------------|-------------------------------|--|----------------------------|
| test_create_podcast | reliable, usable | POST, main.Create, os, tests.random_file_generator | upload podcast |
| test_create_not_audio | reliable, error, safety | POST, main.Create, os, tests.random_file_generator | upload podcast |
| test_list_empty_db | edge case | main.List | see all podcasts |
| test_list | correctness, wholeness | main.List | see all podcasts |
| test_list_with_param | safety, security | main.List | wrong usage |
| test_getFileByFileName | fast, precise | main.getFileByName | search file on server |
| test_getFileByFileName_wrong | fast, error detection | main.getFileByName, tests.random_file_generator | search for wrong file |
| test_getrss | fast, functional, easy to use | main.getRSS | retrieve current RSS |
| test_getrss_no_file | error detection, reliability | main.getRSS | get RSS if no RSS is saved |

4.6 Static Code-Analyse (Deniz Karaman)

Für die statische CodeAnalyse im Backend haben wir die integrierte Inspektions-Engine von PyCharm (JetBrains) eingesetzt. Dabei konnten wir verschiedene Probleme im Code identifizieren. Zu den häufigsten Befunden gehörten unter anderem:

- Unresolved references
- Method is not declared static
- PEP 8 naming convention violation
- Redundant parentheses
- Shadowing built-in names
- Shadowing names from outer scopes
- Using equality operators to compare with None

Dank dieser statischen Prüfung konnten wir gezielt Code-Stellen verbessern etwa fehlende Imports ergänzen, Methoden korrekt deklarieren und PEP 8-konforme Bezeichner verwenden. Durch die frühzeitige Erkennung und Behebung dieser Probleme haben wir die Qualität, Wartbarkeit und Lesbarkeit unserer Anwendung weiter gesteigert.

Literatur

- [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] Chris Halstead. <https://c20d.blog/posts/2023/03/tootrss-utility/>. (Accessed: 2025-05-21).
- [4] scoder & Contributors. lxml. <https://github.com/lxml/lxml>. (Accessed: 2025-05-18).
- [5] James Whittaker. <https://testing.googleblog.com/2011/09/10-minute-test-plan.html>. (Accessed: 2025-05-21).
- [6] Dave Winer. <https://cyber.harvard.edu/rss/rss.html>. (Accessed: 2025-05-20).