Aufbau der Anwendung

Da diese Anwendung der Fachschaft Informatik zur Verfügung gestellt werden soll und es wahrscheinlich ist, dass sie sich aus verschiedenen Gründen mit dem Code vertraut machen muss, wird in diesem Kapitel kurz der Aufbau der Anwendung erklärt, um die Einarbeitung zu erleichtern.

Die Anwendung besteht aus vier größeren Komponenten: Der UltrastarFileParser, die SQLite-Datenbank, das FastAPI Backend und das Sveltekit Frontend, auf welche im Folgenden näher eingegangen wird. Der Aufbau von Sveltekit wird dabei etwas ausführlicher erläutert, da es ein spezifisches Framework ist, welches eventuell nicht jeder Person bekannt ist.

Backend

Für den Aufbau des Backends habe ich mich an einer Mischung aus dem FastAPI Template Projekt von Ramírez (2019a) und der von Yerassyl Z (2021) Vorgeschlagenen Struktur in dem Github Repository "FastAPI Best Practices" orientiert. Da das Template Projekt auf dem vorgegebenen Tutorial von dem FastAPI Entwickler Ramírez (2018) basiert, werden sich auch innerhalb des Codes einige Ähnlichkeiten oder übernommene Passagen finden.

Letztendlich sieht die Struktur so aus:

```
├──song_examples
│   └──Creative Commons
│       └──...
├──src
│   ├──alembic
│   │   └──versions
│   ├──app
│   │   ├──admin
│   │   ├──auth
│   │   ├──queue
│   │   └──songs
│   ├──logging
│   └──ultrastar_file_parser
└──tests
    └──unit
        └──api
```

(Aufbau des Ordners `backend`)

UltrastarFileParser

Der `UltrastarFileParser` enthält alle nötigen Funktionen um eine Ultrastar-Datei zu parsen.

Es gibt die Möglichkeit aus einem angegebenen Ordner mögliche Ultrastar-Dateien zu filtern, sodass die Dateipfade nicht einzeln angegeben werden müssen.
Momentan ist das Identifizieren der Ultrastar-Dateien nicht wirklich Ultrastar spezifisches Verhalten, da aus dem angegebenen Ordner alle `.txt`-Dateien rausgefiltert werden, doch ich habe diese Funktion beim `UltrastarFileParser` belassen, da man gegebenenfalls die Namen der Dateien auf ein bestimmtes Muster matchen könnte, falls eines bekannt ist, dem die Benennung der Dateien folgt.

Es gibt eine Funktion mit der aus einer Ultrastar-Datei die Attribute ausgelesen und in ein Dictionary geschrieben werden.
Als Attribute gelten hierbei die Zeilen, die dem vorgegeben Muster folgen:

```
#TITLE: Best title ever
#ARTIST: Some random artist
...
```

(Abb 2: Beispiel der Attributszeilen einer UltraStar-Datei)

Aus der Datei werden ebenso die Lyrics ausgelesen und in dem Dictionary unter dem Attribut `lyrics` gespeichert.
Es ist zu beachten, dass die Lyrics - dem Format entsprechend - nicht als grammatikalisch korrekte Sätze angegeben sind, wodurch sie auch nicht als solche gespeichert werden. Häufig fehlen Satzzeichen vollständig.

Wenn für die Datei kein Encoding angegeben wurde, wird versucht das Encoding selbstständig zu ermitteln. Bei seltenen Encodings kann es vorkommen, dass diese nicht richtig erkannt werden, wodurch ein `UltrastarMatchingError` geworfen wird, obwohl es sich um eine Ultrastar-Datei handelt. Mit den vorgegebenen Einstellungen wird der Fehler in eine eigene Logging-Datei geschrieben, was es erleichtern soll bei Bedarf manuell zu überprüfen, ob eine Ultrastar-Datei wegen eines falsch identifizierten Encodings nicht als solche erkannt wurde.

Anm.: Während des Zeitraums meiner Bachelorarbeit hat randompersona1 (2024) einen Ultrastar-Parser veröffentlicht. Zu diesem Zeitpunkt hatte ich den Parser bereits fertiggestellt und habe demnach keinen Sinn darin gesehen meinen eigenen Parser durch diese Fremdbibliothek zu ersetzen.

FastAPI

Die für FastAPI relevanten Dateien liegen in `/src/app`. Innerhalb dieses Ordners sieht es folgendermaßen aus:

```
├──...
├──admin
│   └──routes.py
├──auth
│   ├──crud.py
│   ├──dependencies.py
│   ├──exceptions.py
│   ├──models.py
│   ├──routes.py
│   ├──schemas.py
│   └──utils.py
├──queue
│   ├──config.py
│   ├──exception.py
│   ├──routes.py
│   ├──schemas.py
│   └──service.py
├──songs
│   ├──crud.py
│   ├──exceptions.py
│   ├──models.py
│   ├──routes.py
│   └──schemas.py
├──config.py
├──database.py
├──dependencies.py
└──main.py
```

(Aufbau des Ordners `backend/src/app`)

Die App ist in 4 Bereiche eingeteilt: `admin`, `auth`, `queue` und `songs`.
Jeder dieser Teilbereiche hat eine eigene `routes.py`-Datei, welche einen Router enthält, der die FastAPI-Endpoints für den entsprechenden Teilbereich bündelt.
Die `exceptions.py` enthalten unter anderem `HTTPExceptions`, welche in den Endpoint Funktionen geworfen werden und ein JSON Objekt zusammen mit dem in der `HTTPException` gespeicherten HTTP-Status an den Aufrufer zurückgeben.
Ansonsten sind für FastAPI noch die `dependencies.py` interessant. Diese enthalten Dependencies, welche in FastAPI-Funktionen injected werden können, um zum Beispiel das Session handling der Datenbank oder die Authentifizierung zu übernehmen.
In `main.py` wird die FastAPI-App aufgebaut.

Datenbank

Die Datenbank ist eine asynchrone SQLite-Datenbank. In ihr werden die Informationen zu den Songs, sowie die Admin-Zugangsdaten gespeichert.

Innerhalb der Unterordner in `backend/src/app` (siehe Abb. 3) werden die Datenbank-Modelle der Songs und der User in verschiedenen `models.py`-Dateien gespeichert.
Die Funktionen, welche auf die Datenbank zugreifen, werden in `crud.py`-Dateien gespeichert.
In `database.py` sind Funktionen für ein manuelles Setup der Datenbank gespeichert.

Frontend

Sveltekit Allgemein

Für das Frontend wurde Sveltekit verwendet. Sveltekit ist ein Meta-Framework, welches auf Svelte aufbaut. Svelte selbst wiederum ist ein Komponenten basiertes Framework, welches auf HTML, CSS und JavaScript aufbaut. (vgl. Svelte contributors 2021a)

Im Folgenden wird die Struktur einer Sveltekit-Anwendung kurz erläutert.

routes

Durch die Ordnerstruktur innerhalb von `routes` wird der Pfad für die Frontend-Routes vorgegeben.
Demnach wäre die über die Route `/search/result` zu erreichende Seite gespeichert in:

```
└──routes
    └──search
        └──result
```

(Beispielhafter Aufbau des Ordners `routes`)

Ordnernamen, die mit eckigen Klammern umschlossen sind, dienen als Pfadparameter, deren Inhalt von der Seite eingelesen werden kann um Inhalte dynamisch zu laden. (vgl. Svelte contributors 2021d)
Zum Beispiel kann mit `/1/` der Song mit der ID 1 geladen werden.

```
└──routes
    └──[id]
        └──add
```

(Beispielhafter Aufbau des Ordners `routes` mit einem Pfadparameter)

Spezielle Dateien

Es gibt verschiedene Dateien, deren Benennung von Sveltekit vorgegeben werden.

+page.svelte

In dieser Datei wird eine Page der Webapplikation definiert. Sie enthält JavaScript-Funktionen und das HTML für die Page. Die Page wird sowohl auf dem Server gerendert (SSR), als auch auf dem Client (CSR). (vgl. Svelte contributors 2021d)

+page.js

Eine `+page.js` Datei gehört immer zu einer `+page.svelte` Datei, welche im selben Ordner liegen muss. In der `+page.js` Datei kann eine `load`-Funktion definiert werden, welche Daten laden kann, bevor die Komponenten der dazugehörigen `+page.svelte` gerendert werden. Die Funktion wird sowohl auf dem Server als auch auf dem Client ausgeführt. Außerdem kann hier das Verhalten der Page angepasst werden, sodass sie z. B. vorab gerendert wird. (vgl. Svelte contributors 2021d)

+page.server.js

Diese Datei erfüllt die selbe Funktion wie `+page.js`, nur dass die Funktionen nur auf dem Server, und nicht auf dem Client ausgeführt werden. (vgl. Svelte contributors 2021d)

+error.svelte

Sollte in der `load` Funktion der `+page(.server).js` ein Fehler auftreten, wird die Error-Page aus der nächstgelegenen `+error.svelte` gerendert (nächstgelegen bedeutet in diesem Fall: eine Datei im selben Ordner, ansonsten in übergeordneten Ordnern). Sollte keine `+error.svelte` gefunden werden, wird eine Default-Error-Page angezeigt. (vgl. Svelte contributors 2021d)

+layout.svelte

In diesen Dateien lassen sich Layouts definieren, welche als Basis für alle `+page.svelte`-Pages in dem selben Order und seinen Sub-Ordnern verwendet werden. Standardmäßig erben `+layout.svelte`-Dateien von anderen `+layout.svelte`-Dateien, welche in ihnen übergeordneten Ordnern liegen. (vgl. Svelte contributors 2021d)

+layout.js

Die Datei erfüllt die gleiche Funktion für die `+layout.svelte`-Datei, welche die `+page.js`-Datei für `+page.svelte` erfüllt. Allerdings sind bei dieser Funktion die Daten aus `load` auch für die Pages, welche von diesem Layout erben, verfügbar. Außerdem wird hier definiertes Verhalten einer Page als Default für alle erbenden Pages benutzt. (vgl. Svelte contributors 2021d)

+layout.server.js

Erfüllt die selbe Funktion wie `+layout.js`, allerdings werden die Funktionen nur auf dem Server ausgeführt. (vgl. Svelte contributors 2021d)

+server.js

Diese Dateien können benutzt werden um innerhalb von Sveltekit eine API zu erstellen. (vgl. Svelte contributors 2021d)

Stores

Svelte benutzt Stores, um variable Daten zu speichern, welche von mehreren Komponenten einer App abgerufen werden müssen. Stores können von Komponenten subscribed werden, welche dann informiert werden, wenn sich die Daten in einem Store ändern, und sich entsprechend aktualisieren können. (vgl. Svelte contributors 2022)

Sveltekit in dieser Anwendung

Der relevante Teil der Sveltekit Anwendung liegt in `frontend/src/`.

```
├──lib
│   ├──Alert.svelte
│   ├──backend_routes.js
│   ├──custom_utils.js
│   ├──index.js
│   ├──Login.svelte
│   ├──Navbar.svelte
│   ├──settings_funcs.js
│   └──SongTables.svelte
├──params
│   └──number.js
├──routes
│   ├──[id=number]
│   │   ├──add
│   │   │   ├──+page.js
│   │   │   └──+page.svelte
│   │   ├──+page.js
│   │   └──+page.svelte
│   ├──login
│   │   └──+page.svelte
│   ├──processed-songs
│   │   └──+page.svelte
│   ├──queue
│   │   ├──reload
│   │   │   └──+page.svelte
│   │   └──+page.svelte
│   ├──search
│   │   ├──result
│   │   │   └──+page.svelte
│   │   └──+page.svelte
```

```
│   ├──settings
│   │   └──+page.svelte
│   ├──+layout.svelte
│   ├──+page.server.js
│   └──+page.svelte
├──app.html
├──hooks.server.js
└──stores.js
```

(Aufbau des Ordners `frontend/src/` inklusive enthaltener Dateien)

Die Frontend-Routes, die es gibt, lassen sich hier aus der Ordnerstruktur in `/routes` ablesen. Innerhalb der Ordner liegen die Komponenten für die jeweiligen Pages.
Die `/routes/+layout.svelte`-Datei dient hierbei als Layout für alle Pages. Sie integriert die Navigationsleiste aus `/lib/Navbar.svelte`, sowie das System um Nutzer*innen Alerts anzeigen zu können aus `/lib/Alert.svelte`.
Die Basis für jede aufgebaute HTML-Seite ist in `/app.html` definiert.
In `/stores.js` sind benutzerdefinierte Svelte-Stores gespeichert.
In `/params` sind matching-Funktionen gespeichert, um das matching von Pfadparametern einzuschränken. In diesem Fall wird durch `/params/number.js` ein Pfad `[id=number]` nur zugeordnet, wenn sich der Pfad zu einem Integer parsen lässt.

Verknüpfung der Komponenten

Beim Starten der App wird der `UltrastarFileParser` aufgerufen, welcher den angegebenen Pfad nach möglichen Ultrastar-Dateien durchsucht. Danach werden die Ultrastar-Dateien geparsed und es wird versucht die Länge der zur jeweiligen `.txt`-Datei gehörenden Audiodatei zu ermitteln, um sie als `audio_duration` einzutragen. Sollte keine Datei gefunden werden, wird `None` als `audio_duration` eingetragen.
Anschließend werden die Songs mit den Attributen `title`, `artist`, `audio_duration` und `lyrics` in die Datenbank geschrieben. Dabei wird den Songs von der Datenbank eine eindeutige ID zugewiesen, worüber diese in anderen Teilen der Anwendung eindeutig identifiziert werden.
Wenn das Beschreiben der Datenbank mit den Songs abgeschlossen ist, werden `username` und `password` des Admins in die Datenbank geschrieben.
Das alles findet in `backend/src/app/main.py` statt.

Anm.: Zur Befüllung der Datenbank beim Start gibt es eine crud-Funktion um einen neuen User in die Datenbank einzufügen, allerdings gibt es keinen API-Endpoint um diese Funktion aufzurufen.

Wenn das Ganze durch ist, wird die App hochgefahren.
Es ist schwierig die API als Komponente vom Backend zu trennen, da an einigen Stellen das Backend direkt in die Web-API integriert ist. Ich hatte überlegt im Geiste der Aufgabenteilung

eine weitere Abstraktionsebene dazwischen zu legen, um die API vollständig vom Backend getrennt zu haben, aber da an besagten Stellen keine komplexere Logik vorkommt, habe ich das als überflüssig empfunden, da sonst mehr Code entstehen würde, der keinen richtigen Mehrwert bietet.
Bis auf die Verwaltung der Queue, die einen eigenen Service hat, findet die Logik direkt in den Funktionen statt, die mit den Endpoints verknüpft sind. Demnach ruft die API direkt crud-Funktionen auf, welche die Datenbankanfragen stellen.
In Belangen der Queue kommuniziert die API mit dem `QueueService`, der die Verwaltung derselbigen managed.

Das Frontend schickt Anfragen an die API-Endpoints. Um die Schnittstelle zu den Endpoints gesammelt an einem Ort zu haben, sind die Routes in `frontend/src/lib/backend_routes.js` gespeichert und können dort geändert werden, sollten sich die Endpoints ändern.

Es ist zu beachten, dass beim derzeitigen Aufbau der App die API-Endpoints an den Client übermittelt werden. Da der Zugriff auf die Endpoints durch CORS eingeschränkt wird, sollte dies kein größeres Problem darstellen, aber falls dies nicht erwünscht ist, muss das Frontend umgestellt werden, sodass die Anfragen serverseitig passieren und nur die Ergebnisse an den Client übermittelt werden. Sveltekit bietet die nötigen Tools, um dies zu bewerkstelligen.

Tests

Die Tests sind zwar keine eigene Komponente der Anwendung, sollten an dieser Stelle aber trotzdem kurz erwähnt werden.

In dem Ordner `backend/tests/unit/api` sind Tests enthalten, welche alle API-Endpoints testen. Als Framework für die Tests wurde Pytest[1] verwendet, da FastAPI direkte Unterstützung für dieses Framework bietet. (vgl. Ramírez 2019b)
Es gibt jeweils eine eigene Test-Datei für jeden einzelnen Router der Anwendung, also `admin`, `auth`, `queue` und `songs`.

---

1. https://docs.pytest.org/↩