

# Soundtrack-Guesser

**DOKUMENTATION WEBSERVICES | 4. SEMESTER**

SIMON BESTLER [9295660] | FABIAN LOHMÜLLER [9514094] | NIEL  
ROHLING [8864957] | LUCA SCHIRMBRAND [4775194]

<b>Abbildungsverzeichnis.....</b>	<b>II</b>
<b>1 Projektidee und -ablauf.....</b>	<b>1</b>
1.1 Grundidee des Soundtrack-Guesser .....	1
1.2 Aufgabenverteilung.....	1
<b>2 Verwendete APIs.....</b>	<b>2</b>
2.1 TMDb-API .....	2
2.2 Spotify-API.....	2
2.2.1 Authentifizierung.....	2
2.2.2 Verwendete API-Methoden .....	3
<b>3 Backend .....</b>	<b>3</b>
3.1 Entrypoints .....	3
3.2 Services.....	4
3.3 Dataproviders .....	5
3.3.1 Spotify-API.....	5
3.3.2 TMDb-API .....	6
3.3.3 MariaDB.....	7
<b>4 Frontend .....</b>	<b>8</b>
4.1 Oberfläche - Einführung.....	8
4.2 Oberfläche - Startfenster .....	8
4.3 Oberfläche – Im Spiel .....	9
4.4 Oberfläche – Ende des Spiels .....	10
4.5 Oberfläche - Administratorbereich.....	10
4.6 Oberfläche – Match verwalten.....	11
4.7 Oberfläche – Matches hinzufügen.....	12
<b>5 Performancetests .....</b>	<b>13</b>
<b>6 Containerisierung.....</b>	<b>13</b>
<b>7 OpenAPI/SwaggerUI .....</b>	<b>14</b>

<b>8</b>	<b><i>BPMN-Modellierung</i></b> .....	<b>14</b>
<b>9</b>	<b><i>Links</i></b> .....	<b>16</b>
<b>10</b>	<b><i>Quellen</i></b> .....	<b>16</b>

## Abbildungsverzeichnis

Abbildung 1:	Authentifizierung Spotify Web API [2] .....	3
Abbildung 2:	RestController für die Route /game .....	4
Abbildung 3:	Beispiel eines Game-Objekts in JSON .....	5
Abbildung 4	Navigation: Startseite .....	8
Abbildung 5	Navigation: Spieleübersicht .....	9
Abbildung 6	Navigation: Nochmal spielen .....	10
Abbildung 7	Navigation: Administratorbereich .....	11
Abbildung 8	Navigation: Eintrag Einzelansicht.....	11
Abbildung 9	Navigation: Eintrag hinzufügen.....	12
Abbildung 10:	Performancetests - SOAP UI .....	13
Abbildung 11:	Multicontaineranwendung: Soundtrack-Guesser .....	14

# 1 Projektidee und -ablauf

## 1.1 Grundidee des Soundtrack-Guesser

Wir – die Projektmitglieder – sind sehr begeisterte Nutzer von Spotify, weswegen wir uns schnell einigen konnten, dass wir die sehr umfangreiche Spotify-API für unseren Webservice verwenden möchten. Die Grundüberlegung ging zunächst in Richtung eines Dienstes, der beispielsweise individualisierte Playlist erstellt (u. a. mit Fokus auf unbekanntere Künstler). Die Aufgabenstellung sieht aber eine Verzahnung mehrerer Services vor. Deswegen versuchten wir eine sinnvolle Kombination mit einem weiteren Interessenschwerpunkt unsererseits zu finden. Dies gelang uns mit dem Thema Filmen bzw. der Kombination zu Filmmusik.

Aufgrund der Coronapandemie erfreuen sich kleine Onlinespiele (wie [skribbl.io](https://skribbl.io) – eine Art Online Montagsmaler) großer Beliebtheit. Davon ausgehend entstand die Idee, ein kleines Onlinespiel zu bauen, bei dem die User ihre Fähigkeit Filmmusik zu erkennen unter Beweis stellen können. Das Spielkonzept ist dabei sehr simpel: Spotify liefert einen 30 Sekunden Ausschnitt aus einem Filmmusiktitel und der Nutzer bekommt drei Filme angezeigt und versucht den gehörten Ausschnitt dem passenden Film zuzuordnen. Die Daten zu den Filmen (wie Titel, Cover-Art) werden von der [TMDb](https://www.tmdb.org/) (The Movie Database) API bezogen.

## 1.2 Aufgabenverteilung

Name	Zuständigkeit
Simon Bestler	Backend
Fabian Lohmüller	Frontend
Niel Rohling	Backend
Luca Schirmbrand	Frontend

Die restlichen Aufgaben (Performancetests, Dokumentation, Präsentation, ...) wurden gemeinschaftlich erstellt, sodass eine klare Zuteilung nicht möglich ist.

## 2 Verwendete APIs

Wie bereits in der Projektbeschreibung dargelegt werden die APIs von Spotify (<https://developer.spotify.com/documentation/web-api/>) und von TMDb (<https://developers.themoviedb.org/3>) verwendet.

### 2.1 TMDb-API

The Movie Database (TMDb) ist eine kollaborative Datenbank, die Metadaten von Filmen und Serien bereitstellt. Die Daten werden dabei von Nutzern aus der Community gepflegt. TMDb zählt in diesem Bereich zu den führenden Anbietern und wird von 400.000 Entwicklern und Unternehmen genutzt. [1]

Die Authentifizierung erfolgt über einen Bearer-Token, der mit einem kostenlosen Entwickleraccount beantragt werden kann. Des Weiteren ist ein großer Vorteil, dass der Dienst kostenlos zur Verfügung steht, wobei noch nicht einmal die Anzahl der Abfragen begrenzt ist.

Für das Projekt war vor allem die Bereitstellung von qualitativ hochwertigen Filmcovern relevant.

### 2.2 Spotify-API

#### 2.2.1 Authentifizierung

Spotify setzt bei der Authentifizierung auf den OAuth2 Standard. Dieser ist in der Handhabung etwas komplexer als die Methode des Bearer-Tokens. OAuth2 bietet dafür grundsätzlich auch die Möglichkeit eine nutzerspezifische Authentifizierung durchzuführen. Dies ist notwendig, wenn man Nutzerdaten (also bspw. Playlists, Statistiken zum Nutzerverhalten) abrufen möchte. Da für das Projekt aber nur allgemeine Daten von Spotify benötigt wurden, konnte auf die vereinfachte Form der Authentifizierung mit Client-Credentials (Client-ID & Client-Secret) zurückgegriffen werden (siehe Abbildung 1).

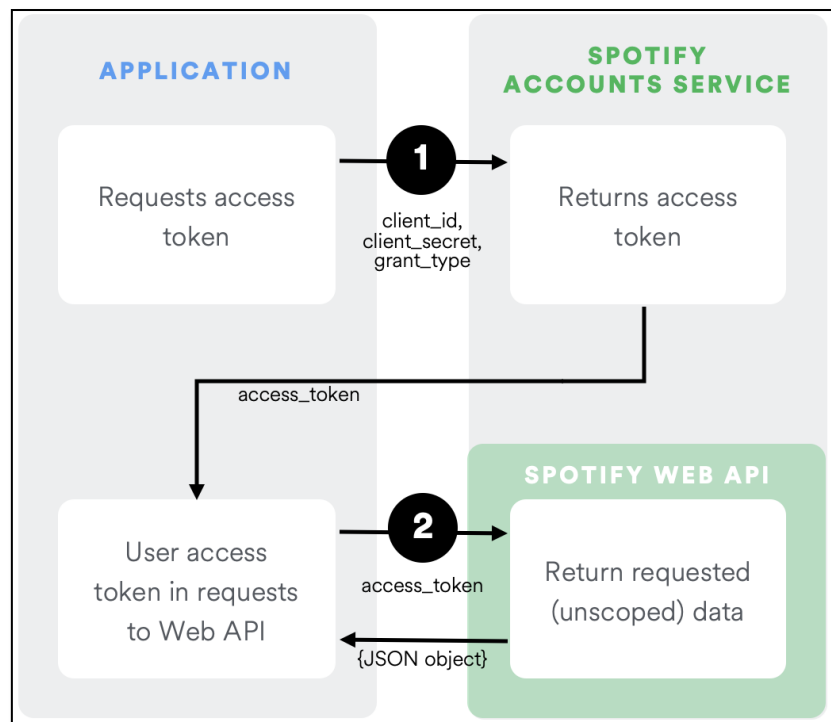


Abbildung 1: Authentifizierung Spotify Web API [2]

### 2.2.2 Verwendete API-Methoden

Für das Projekt waren für uns die API-Methoden *search* (Herausfinden der TrackID zu einem Songtitel) sowie *track* (Abrufen von Daten zu einer gegebenen TrackID) von Belang.

## 3 Backend

Das Backend stellt die Kernkomponente unseres Dienstes dar. Für die Entwicklung wurde aus das Java Framework Springboot gesetzt. Die Architektur gliedert sich in drei Teile: Die Entrypoints beschreiben die REST-Schnittstellen, die durch Spring-Boot bereitgestellt werden. Die **Entrypoints** sind nur für Ein- und Ausgabe zuständig und rufen die **Services** auf. Diese beinhalten die Geschäftslogik. Die Daten erhalten die Services auf Abfrage von den **Dataproviders**.

### 3.1 Entrypoints

Die Endpunkte der Rest-API werden durch vier Klassen beschrieben: MovieController (/movie), TrackController (/track), MatchController (/match) und GameController (/game). Durch die Annotation @RestController von Spring-Boot werden diese Klassen automatisch von Spring-Boot erzeugt und die Schnittstellen bereitgestellt.

```

@RestController
@RequestMapping("/game")
public class GameController {

    private final GameService gameService;

    public GameController(GameService gameService) {
        this.gameService = gameService;
    }

    @GetMapping("")
    public Game getNewGame(
        @RequestParam(name = "roundCount", defaultValue = "10") int roundCount,
        @RequestParam(name = "moviesPerRound", defaultValue = "3") int moviesPerRound) {
        return gameService.create(roundCount, moviesPerRound);
    }
}

```

Abbildung 2: RestController für die Route /game

Die API unterstützt folgende HTTP-Anfragen:

Methode	Route	Beschreibung
GET	/movie	Eine Liste aller in der Datenbank vermerkten Filme
GET	/movie/{id}	Film mit der entsprechenden ID
GET	/movie/search/{query}	Sucht nach Filmen von TMDB
GET	/movie/for/{trackID}	Film, der laut DB zum Lied mit der ID passt
GET	/track	Eine Liste aller in der Datenbank vermerkten Lieder
GET	/track/{id}	Lied mit der entsprechenden ID
GET	/track/search/{query}	Sucht nach Liedern von Spotify
GET	/track/for/{movieID}	Lied, das laut DB zum Film mit der ID passt
GET	/match	Alle Matches aus der DB
POST	/match	Neues Match erstellen; Body: {movieID, trackID}
PATCH	/match	Match bearbeiten; Body: {id, movieID, trackID}
GET	/match/{id}	Match mit bestimmter ID aus DB
DELETE	/match/{id}	Löscht Match aus DB
GET	/game	Generiert Daten für ein Spiel

### 3.2 Services

Die Services werden direkt von den Entrypoints aufgerufen. Sie beinhalten die Geschäftslogik. Hier ist lediglich der GameService interessant, da die anderen nur auf triviale Weise Daten von den Dataproviders abfragen/übergeben.

Der GameService stellt über die Methode create() Daten für ein gesamtes Spiel (standardmäßig 10 Runden) bereit. Dafür werden pro Runde zufällig drei Filme ausgesucht, aus denen ein Film bestimmt wird. Von diesem Film wird der richtige Soundtrack bestimmt. Zusammen mit den Filmen und dem Index des passenden Films wird der Soundtrack als Round-Objekt gespeichert. Die Runden werden als Game-Objekt zurückgegeben.

```
{
  "rounds": [
    {
      "movies": [
        {
          "id": 157336,
          "title": "Interstellar",
          "posterURL": "https://image.tmdb.org/t/p/w1280/gEU2QniE6E77NI6lCU6MxLNBvIX.jpg",
          "genres": [
            "Adventure",
            "Drama",
            "Science Fiction"
          ],
          "overview": "..."
        },
        {
          "id": 273248,
          "title": "The Hateful Eight",
          ...
        },
        {
          "id": 680,
          "title": "Pulp Fiction",
          ...
        }
      ],
      "correctIndex": 1,
      "soundTrack": {
        "id": "1dbf8JY7I2WJ8xgdlnEhTi",
        "name": "L'Ultima Dilligenza di Red Rock - From \"The Hateful Eight\" Soundtrack / Versione Integrale",
        "artistName": "Ennio Morricone",
        "previewURL": "https://p.scdn.co/mp3-preview/b6bfb623bc43655007725e41eb736bd25d021383?cid=a46f5c5745a14fbf826186da8da5ecc3",
        "url": "https://open.spotify.com/track/1dbf8JY7I2WJ8xgdlnEhTi",
        "albumName": "Quentin Tarantino's The Hateful Eight (Original Motion Picture Soundtrack)",
        "coverURL": "https://i.scdn.co/image/ab67616d0000b2731f81120f53150c3920057e48"
      }
    },
    { ... },
    { ... },
    { ... },
    { ... },
    { ... },
    { ... },
    { ... },
    { ... },
    { ... },
    { ... }
  ]
}
```

Abbildung 3: Beispiel eines Game-Objekts in JSON

### 3.3 Dataproviders

#### 3.3.1 Spotify-API

Da nicht alle Informationen, die von der Spotify-API über die Lieder bereitgestellt werden, für unseren Dienst benötigt werden bzw. in ein anderes Format übergeführt werden müssen, nutzen wir einen individuellen JSON-Deserialisierer, der aus JSON ein Java-Objekt erzeugt.



```

public class SpotifyTrackDeserializer extends StdDeserializer<Track> {

    public SpotifyTrackDeserializer() {
        this(null);
    }

    public SpotifyTrackDeserializer(Class<?> vc) {
        super(vc);
    }

    @Override
    public Track deserialize(JsonParser p, DeserializationContext ctx) throws IOException {
        JsonNode node = p.getCodec().readTree(p);

        String id = node.get("id").asText();
        String name = node.get("name").asText();

        List<String> artists = new ArrayList<>();
        JsonNode artistsList = node.get("artists");
        if (artistsList != null) {
            artistsList.forEach(artistNode -> artists.add(artistNode.get("name").asText()));
        }

        String previewURL = node.get("preview_url").asText();
        String spotifyURL = node.get("external_urls").get("spotify").asText();
        String albumName = node.get("album").get("name").asText();

        JsonNode imagesNode = node.get("album").get("images");
        String coverURL = imagesNode.isEmpty() ? "" : imagesNode.get(0).get("url").asText();

        return new Track(id, name, String.join(", ", artists), previewURL, spotifyURL, albumName, coverURL);
    }
}

```

Codeausschnitt 1: JSON-Decoder

Da von der Spotify-API nicht immer konsistent das Attribut „previewURL“ mit einem Wert gefüllt ist, sondern öfters den Wert „null“ übergibt, wir aber von jedem Lied die 30-sekündige Vorschau benötigen, mussten wir einen Workaround finden. Falls die API keine previewURL liefert, nutzen wir Webscraping auf der Seite des Spotify-Embedded-Player (<https://open.spotify.com/embed/track/{trackID}>). Die Klasse SpotifyWebScraper lädt dafür die Seite und sucht nach dem HTML Element mit der ID „resource“. Das Element ist ein Script-Tag, dass Daten über das Lied in JSON-Format (URL-Codiert) mit einer gültigen previewURL enthält.

Da oft viele Daten gleichzeitig angefragt werden, werden die Lieddaten von Spotify durch Spring-Boot gecachet. Dafür werden die Methoden im SpotifyProvider mit @Cacheable annotiert.

### 3.3.2 TMDb-API

Auch für die Filmdaten von der TMDb wird ein JSON-Deserialisierer benutzt, damit die Daten in ein für uns nützliches Format konvertiert werden. Auch die Anfragen an die TMDb werden im Backend durch Spring-Boot gecachet.

### 3.3.3 MariaDB

Neben den Datensätzen, die über die REST-APIs abgerufen werden, müssen die Zuordnung der Filme zur Filmmusik gespeichert werden. Diese Daten müssen persistent gespeichert werden. Für diesen Einsatzzweck werden die Daten in eigene MariaDB-Datenbank gespeichert. Für die Transaktionen zwischen der Datenbank und dem Backend wird auf ORM (Object-relational Mapping) mittels Hibernate gesetzt. Hierfür gibt es eine *MatchEntity*, also eine Java-Klasse, die die entsprechenden Attribute der Daten in der Datenbank besitzt. Die Daten werden der SpringBoot Anwendung mittels eines CrudRepositorys bereitgehalten.

## 4 Frontend

Für das Frontend wurde auf das JavaScript Framework React gesetzt.

### 4.1 Oberfläche - Einführung

Im Folgenden wird im Sinne der Dokumentation durch das Frontend des „Soundtrack-Guessers“ navigiert. Die Website beinhaltet mehrere Fenster, für den normalen Benutzer ist allerdings nur das Startfenster relevant

### 4.2 Oberfläche - Startfenster

Das Startfenster ist die initial aufgerufene Seite des Soundtrack-Guessers. Hier wird der Benutzer begrüßt und mit dem Ablauf des Spiels bekannt gemacht. Nun, da der Benutzer das Spiel verstanden hat, kann er durch Drücken des Knopfes „Start Game“ das Spiel starten. Hierbei sind einige Sekunden Ladezeit zu erwarten, da zu Beginn des Spiels alle benötigten Runden aus dem Backend geladen werden und anschließend im Cache gespeichert werden. Dies bewirkt, dass die Ladezeiten zwischen den einzelnen Runden minimiert werden und der Benutzer ein optimales Spielerlebnis erfährt. Außerdem ist ein manueller Start, da viele Browser eine manuelle Interaktion mit der Website erfordern um anschließend Audio automatisch abzuspielen.

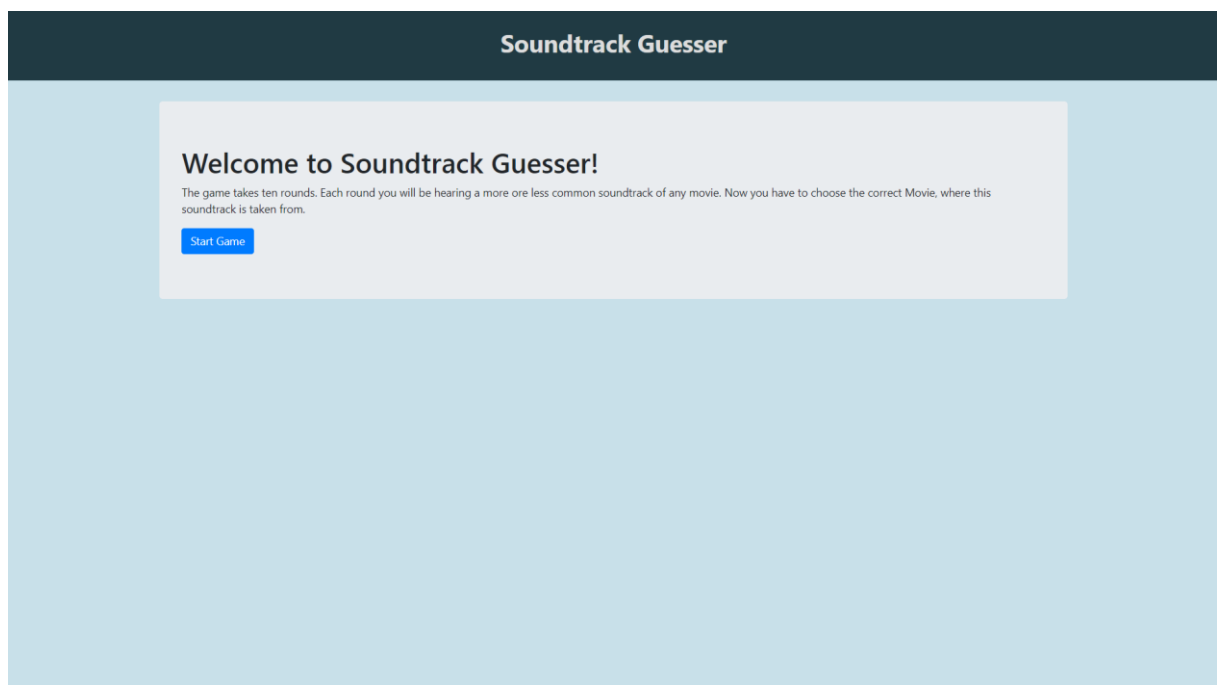


Abbildung 4 Navigation: Startseite

### 4.3 Oberfläche – Im Spiel

Nachdem der Spieler den besagten Knopf gedrückt hat, fängt das Spiel nun mit der ersten Runde an. An oberster Stelle sieht der Spieler seine aktuelle Punktzahl. Diese existiert nur während des Spiels, beginnt also jedes Mal bei null und kann maximal der Menge der Runden entsprechen. Ziel des Spiels ist logischerweise eine möglichst hohe Punktzahl zu erreichen, mit der man sich entweder mit seiner früheren Bestleistung und seinen Freunden messen kann, oder einfach Spaß haben kann. Unter der aktuellen Punktzahl ist die Runde des Spiels zu sehen, in der sich der Benutzer gerade befindet. Daneben wird ihm auch angezeigt, wie viele Runden das Spiel insgesamt hat, sodass der Benutzer immer den Überblick hat. Unterhalb der Rundenübersicht ist der Timer, visualisiert als blauer Balken mit einer ablaufenden Zeit, zu sehen. Initial hat der Timer einen Wert von 30 Sekunden. Unter dem ablaufenden Timer befinden sich die drei Auswahlmöglichkeiten, visualisiert durch das jeweilige Filmposter und den jeweiligen Filmnamen. Der Spieler muss seine Auswahl durch Klicken, auf das seiner Meinung nach richtige Filmposter bestätigen. Nach abgeschlossener Wahl wird dem Benutzer visuell, je nach Richtigkeit, durch einen eingefärbten Rahmen um die richtige Auswahl die Entscheidung validiert.

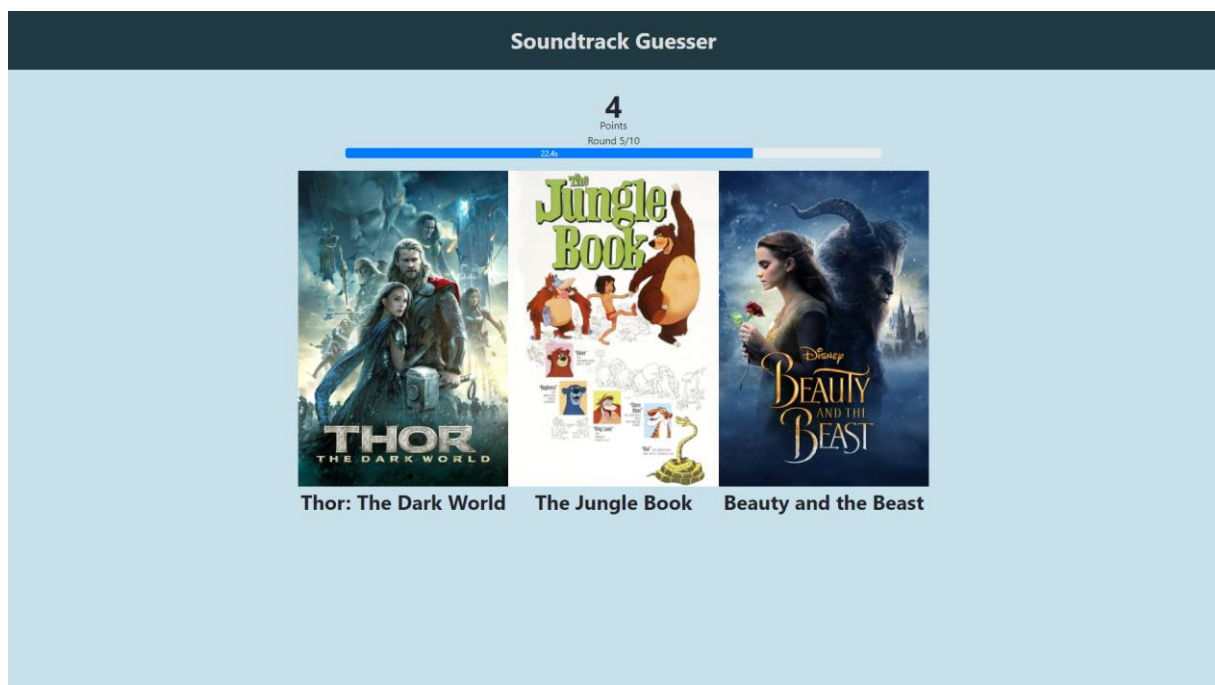
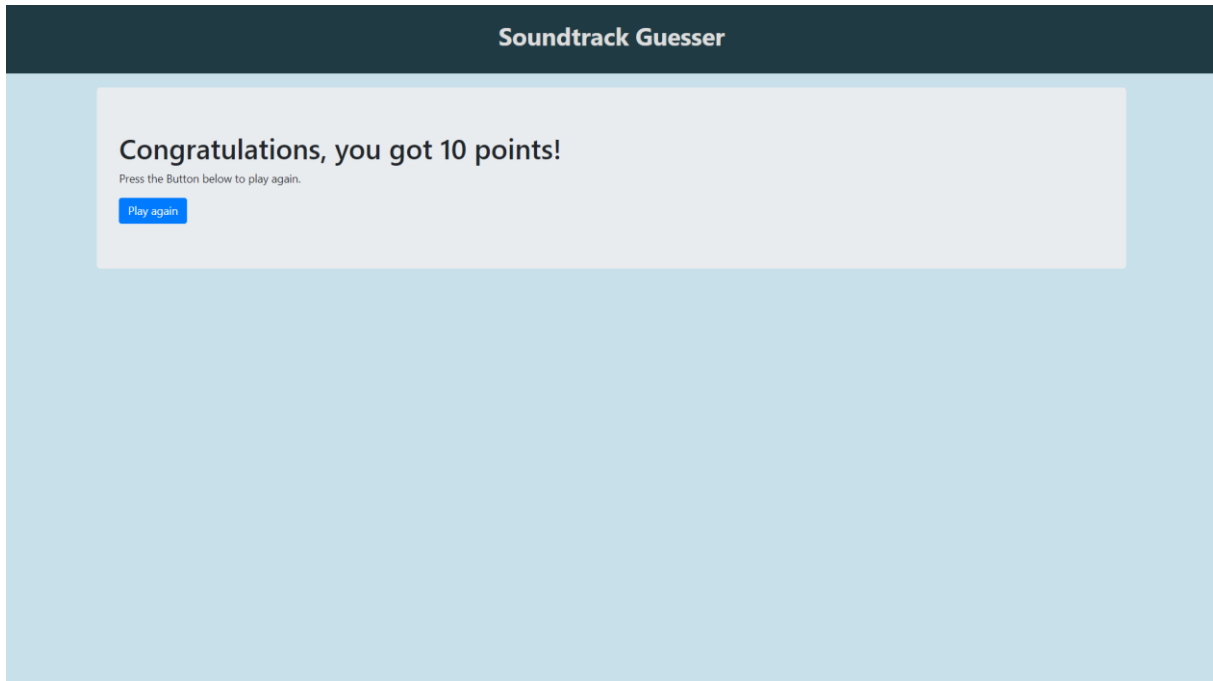


Abbildung 5 Navigation: Spieleübersicht

#### 4.4 Oberfläche – Ende des Spiels

Nachdem das Spiel beendet wurde, wird der Benutzer automatisch auf die Spielübersichtsseite weitergeleitet. Hier hat er Einsicht auf seine erzielte Punktzahl. Natürlich wird dem Benutzer hier auch die Möglichkeit geboten, das Spiel zu wiederholen.



*Abbildung 6 Navigation: Nochmal spielen*

#### 4.5 Oberfläche - Administratorbereich

Um dem Benutzer das Spielen zu ermöglichen muss zuerst das Backend mit Daten gefüllt werden. Dies geschieht mithilfe des eigens dafür eingerichteten Administratorbereich. Wie bereits im Teil „Backend“ der Dokumentation beschrieben, herrscht das Problem vor, dass ein Filmtitel nur in den seltensten Fällen gleichlautend zu dem verwendeten Musiktitel des Films ist. Auch gibt es keine Einträge in der TMDb, die Informationen über die verwendete Filmmusik geben. Dies bedeutet, dass ein automatisches Matching zwischen Filmtitel und Musiktitel nicht möglich ist, was allerdings dringend für den „Soundtrack-Guesser“ benötigt ist. Das resultiert darin, dass das Matching manuell erfolgen muss. Ein einzelnes Einfügen von Matches nach jedem Reboot des Backends wäre umständlich und langwierig, weshalb eine Import- und Exportfunktion eingeführt wurde, die den besagten Vorgang beschleunigt. Diese Import- und Exportfunktionalität funktioniert mithilfe von JSON-Dateien, die alle erstellten Matches beinhalten. Somit können die angelegten Matches jederzeit als Datei lokal exportiert und zu einem späteren Zeitpunkt wieder importiert werden.

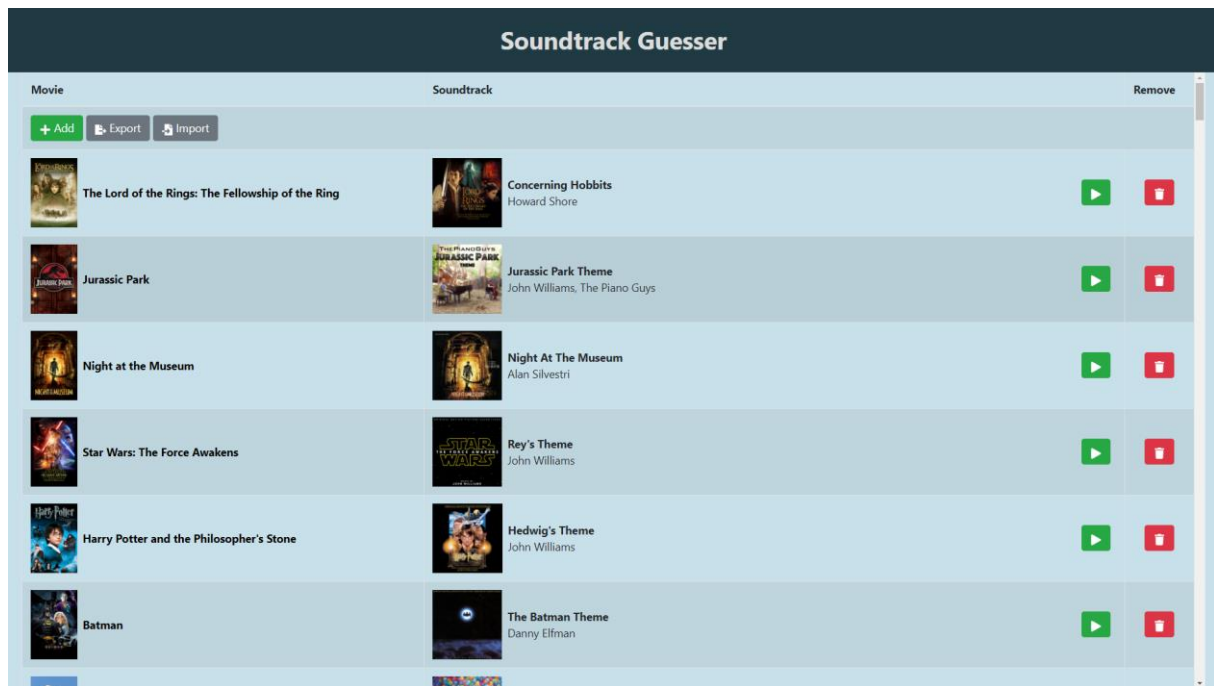


Abbildung 7 Navigation: Administratorbereich

#### 4.6 Oberfläche – Match verwalten

Weiterhin hat der Administrator die Möglichkeit den Soundtrack von existierenden Matches im Administratorbereich abzuspielen, was für die Validierung der Matches hilfreich ist.

Natürlich müssen weiterhin auch einzelne Einträge gelöscht und hinzugefügt werden, was durch das Drücken des „Add“- oder „Delete“-Buttons realisiert wurde.

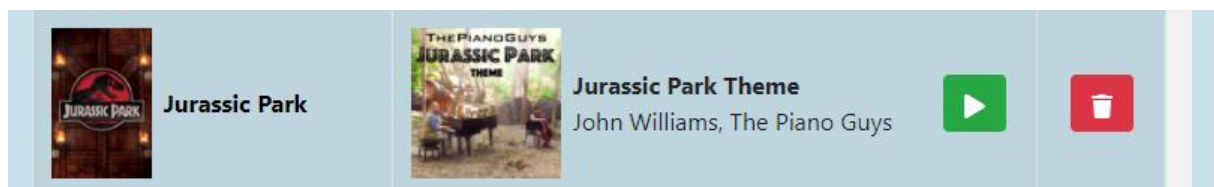


Abbildung 8 Navigation: Eintrag Einzelansicht

#### 4.7 Oberfläche – Matches hinzufügen

Das manuelle Matching eines einzelnen Eintrags wurde über ein eigenes Fenster realisiert, was nach Drücken des Buttons „Add“ automatisch geöffnet wird. Dieses ist mit einer Überschrift versehen, die dem Administrator den Überblick über die Aufgabe des aktuellen Fensters verschafft. Darunter sind zwei Inputs, realisiert als Textfelder, zu sehen. Diese sind beide jeweils mit einer API verbunden. Das linke Textfeld nutzt die TMDb um stichwortartig eingetragene Begriffe zu analysieren und die entsprechend in Frage kommenden Filme, mit jeweiligem Filmposter und Filmenamen, dem Administrator über eine Drop-Down-Liste vorzuschlagen, die dieser dann auswählen kann. Das rechte Textfeld ist sehr ähnlich realisiert, allerdings ist es mit der Spotify API verbunden und schlägt dem Administrator entsprechend Musiktitel, passend zur Eingabe, vor. In der sich öffnenden Drop-Down-Liste ist dieses Mal zusätzlich eine Audio-Vorschau realisiert, worüber der Administrator validieren kann, dass der vorgeschlagene Soundtrack tatsächlich der Gesuchte ist. Über die zwei Steuerelemente unterhalb des linken Textfeldes, realisiert als Buttons, kann der aktuelle Vorgang des Hinzufügens eines Matches entweder abgeschlossen, oder abgebrochen werden.

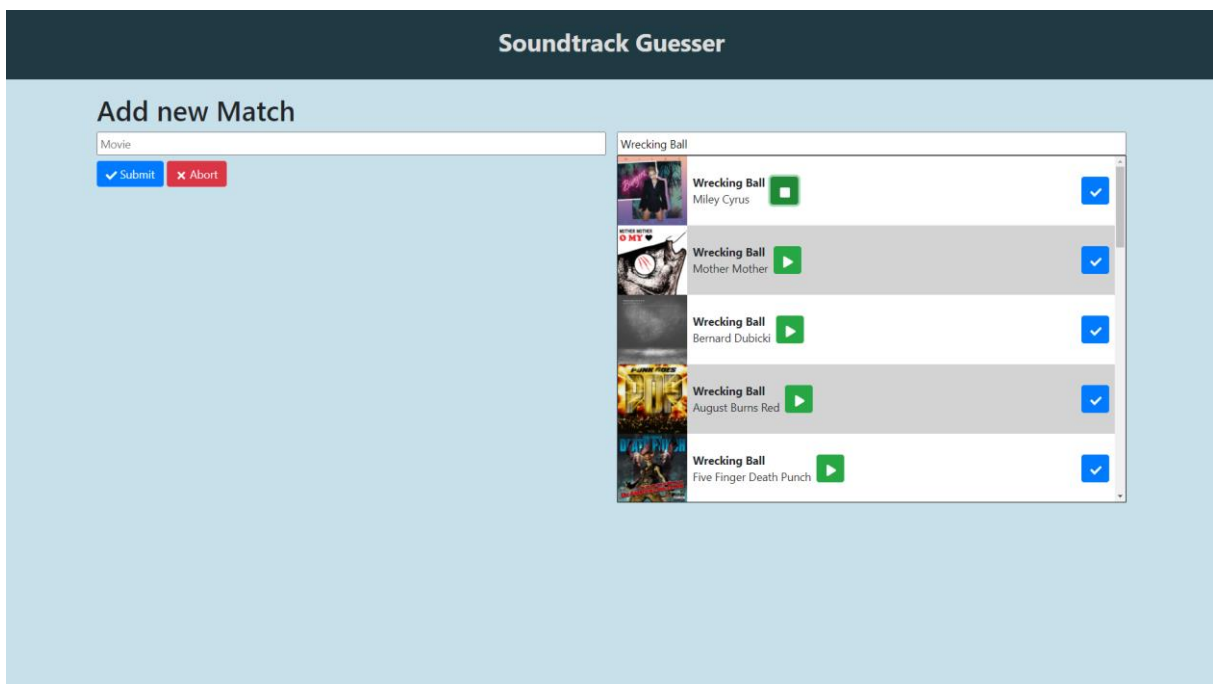
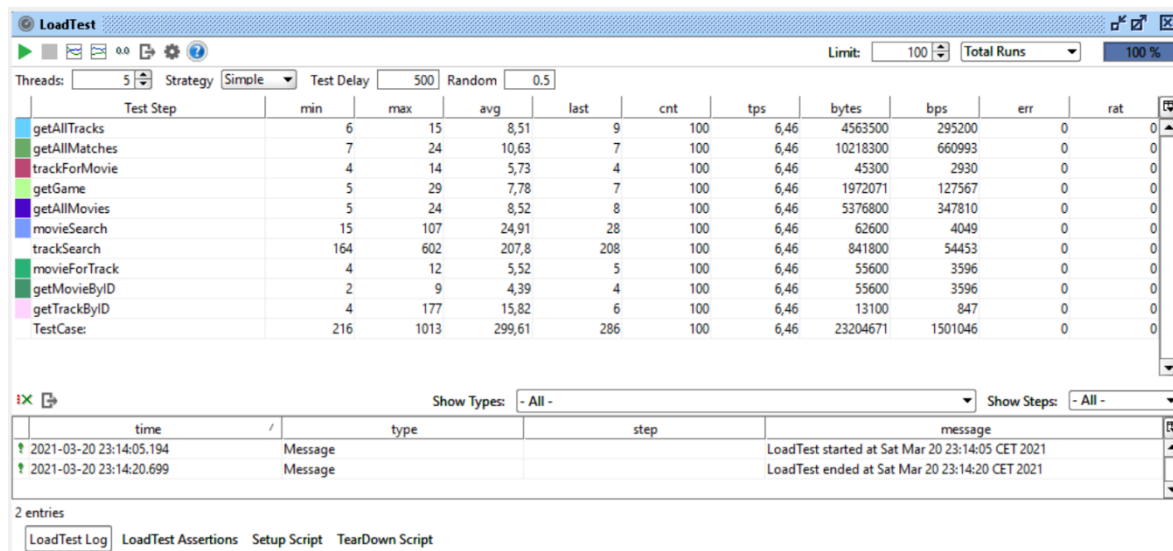


Abbildung 9 Navigation: Eintrag hinzufügen

## 5 Performancetests



The screenshot shows the SoapUI LoadTest interface. At the top, there are controls for Threads (5), Strategy (Simple), Test Delay (500), and Random (0.5). Below this is a table of test steps with columns: Test Step, min, max, avg, last, cnt, tps, bytes, bps, err, and rat. The test steps include getAllTracks, getAllMatches, trackForMovie, getGame, getAllMovies, movieSearch, trackSearch, movieForTrack, getMovieByID, getTrackByID, and TestCase. The results show that movieSearch and trackSearch have significantly higher response times (avg 24.91s and 207.8s respectively) compared to the other steps. Below the table, there are filters for Show Types and Show Steps, and a log section showing two entries: 'LoadTest started at Sat Mar 20 23:14:05 CET 2021' and 'LoadTest ended at Sat Mar 20 23:14:20 CET 2021'.

Test Step	min	max	avg	last	cnt	tps	bytes	bps	err	rat
getAllTracks	6	15	8,51	9	100	6,46	4563500	295200	0	0
getAllMatches	7	24	10,63	7	100	6,46	10218300	660993	0	0
trackForMovie	4	14	5,73	4	100	6,46	45300	2930	0	0
getGame	5	29	7,78	7	100	6,46	1972071	127567	0	0
getAllMovies	5	24	8,52	8	100	6,46	5376800	347810	0	0
movieSearch	15	107	24,91	28	100	6,46	62600	4049	0	0
trackSearch	164	602	207,8	208	100	6,46	841800	54453	0	0
movieForTrack	4	12	5,52	5	100	6,46	55600	3596	0	0
getMovieByID	2	9	4,39	4	100	6,46	55600	3596	0	0
getTrackByID	4	177	15,82	6	100	6,46	13100	847	0	0
TestCase	216	1013	299,61	286	100	6,46	23204671	1501046	0	0

Abbildung 10: Performancetests - SOAP UI

Die Performancetests wurde mit SOAP-UI durchgeführt (siehe Abbildung 10). An dieser Stelle zeigt sich ein gravierender Performanceunterschied zwischen den Endpunkten, die wir im Backend als *cacheable* annotiert haben und den Anfragen, die jedes Mal an Spotify bzw. TMDb weitergereicht werden (bspw. *movieSearch* und *trackSearch*). Auch ist ersichtlich, dass die Antwortzeit der Spotify-API länger ist als die der TMDb-API (wobei ein Teil auch auf die etwas größeren Datensätze der Spotify-API zurückzuführen ist.)

## 6 Containerisierung

Das Deployment des Projekts erfolgt mittels Docker. Das Frontend, Backend sowie die Datenbank werden in separaten Docker-Containern betrieben.

Der Datenbank-Container enthält eine aktuelle Instanz einer MariaDB-Datenbank. Beim Erstellen des Images, wird die Datenbank für die spätere Anwendung vorbereitet, indem ein Nutzer mit Passwort speziell für den Webservice angelegt wird. Des Weiteren wird die Datenbank initial erstellt. Das Anlegen der Tabellen wird durch Spring Boot bzw. Hibernate durchgeführt.

Das mit React entwickelte Frontend wird im Client-Container bereitgestellt. Hierfür wird die Webanwendung über npm mit react-scripts gebaut. Das Image basiert auf dem nginx-Server, der das Frontend bereitstellt.



Das Herzstück des Webservice ist das Backend. Dieses wird in einem Container mit Gradle gebaut und anschließend ein Image basierend auf einer aktuellen Java-Version (OpenJDK 15.0.2) mit der von Gradle gebauten JAR-Datei erstellt.

Um die Container zusammen zu starten wird Docker-Compose verwendet. Lokal gibt es dafür eine docker-compose.yml Datei, in der sich ein Anwendungsstapel (Stack) definieren lässt. Dadurch können die Images in den jeweiligen Unterordnern gleichzeitig gebaut und die Container gestartet werden.

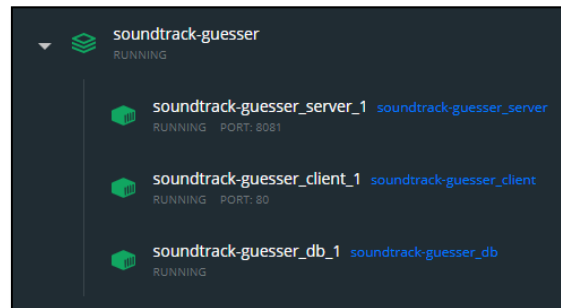


Abbildung 11: Multicontaineranwendung: Soundtrack-Guesser

Um die Container auch auf dem Docker-Host für die Abgabe starten zu können, werden durch Pipelines im git-Repository auf gitlab.com die Images gebaut und auf der gitlab-eigenen Image-Registry bereitgestellt. Der Stack kann nun direkt über Portainer auf dem Host für die Abgabe über docker-compose Syntax definiert werden. Der einzelne Unterschied zum lokalen Stack besteht darin, dass die Images nicht neu gebaut werden, sondern auf die Image-Registry von gitlab zugegriffen wird.

## 7 OpenAPI/SwaggerUI

Im git-Repository ist die OpenAPI-Datei gespeichert. Erreichbar ist sie auch unter der Route `/openapi.yaml` des Backends. SwaggerUI erreicht man unter `/swagger-ui.html`.

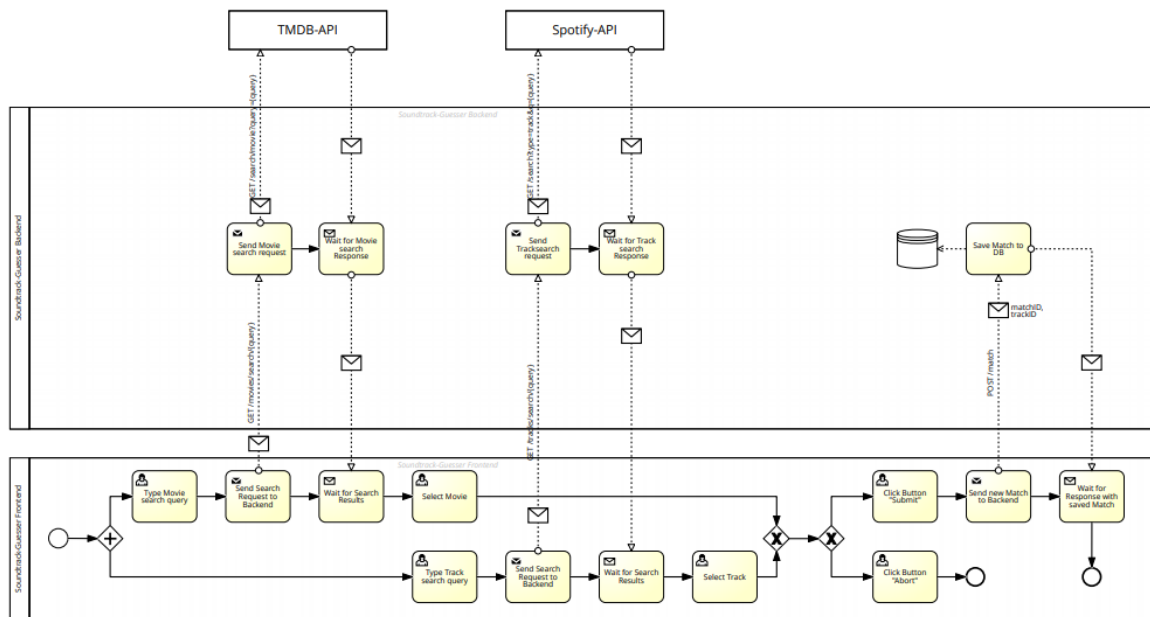
## 8 BPMN-Modellierung

Es wurden zwei Prozesse mit BPMN modelliert:

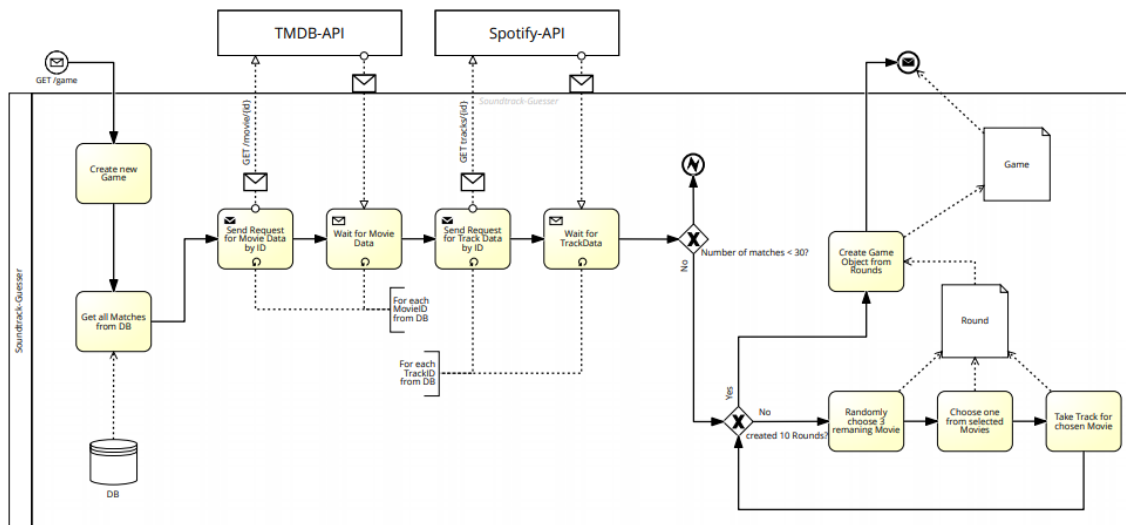
1. Anlegung eines neuen Matches in der DB durch den User.
2. Erstellung der Daten für ein Spiel, angestoßen durch ein GET-Request.

Die Modelle sind auf der folgenden Seite als Bilder eingefügt. Im git-Repository sind die Modelle auch in einer PDF-Datei gespeichert.

## Add Match



## Create Game



## 9 Links

### git-Repository

- [http://10.50.15.50/abgaben\\_ws/inf19/soundtrack-guesser/](http://10.50.15.50/abgaben_ws/inf19/soundtrack-guesser/)
- <https://gitlab.com/123niel/soundtrack-guesser/> (Pipelines und Docker-Registry)

### Frontend

- Spiel: <http://10.50.15.51:8880/>
- Adminbereich: <http://10.50.15.51:8880/admin>

### Backend

- <http://10.50.15.51:8881/>
- OpenAPI: <http://10.50.15.51:8881/openapi> oder  
<http://10.50.15.51:8881/openapi.yml>
- SwaggerUI: <http://10.50.15.51:8881/swagger-ui.html>

## 10 Quellen

[1] [Online]. Available: <https://www.themoviedb.org/about?language=de>.

[2] [Online]. Available:

<https://developer.spotify.com/documentation/general/guides/authorization-guide/>.