Entwicklung einer anwenderorientierten Webanwendung zur intelligenten Verwaltung von Musiksammlungen

Bachelor-Thesis

zur Erlangung des akademischen Grades B.Sc

Christopher von Bargen

[redacted]

Hochschule für Angewandte Wissenschaften Hamburg

Fakultät Design, Medien und Information

Department Medientechnik

Erstprüfer: Prof. Dr. Andreas Plaß

Zweitprüfer: Dipl.-Ing. Thorsten Wagener

Hamburg, 26. 02. 2019

Abstract

Diese Arbeit dokumentiert Planung und Umsetzung einer entkoppelten Anwendung zur Verwaltung einer digitalen Musiksammlung in Datenbankform. Aus der Grundidee werden anwenderorientierte Anforderungen formuliert, die den gesamten Entstehungsprozess begleiten sollen. Es werden konkrete Konzepte entwickelt, die schließlich mit modernen Technologien wie *Angular* und *Spring* umgesetzt werden. Die besondere Anforderung dabei ist, das Projekt in zwei getrennte Arbeitsfelder – *Back-End* und *Front-End* – aufzuteilen und gleichzeitig zu betreuen. Es wird ein detaillierter Einblick in die Software-Architektur und Schnittstellengestaltung gewährt, besondere Problemstellungen erläutert und vielseitige Funktionalitäten prototypisch umgesetzt. Am Ende ist eine solide Basis geschaffen, die darauf abzielt, Erweiterbarkeit und Unabhängigkeit der einzelnen Anwendungen zu gewährleisten. Zudem wird ein offener Entwicklungsprozess sichtbar, der die Fortsetzung des Projekts eindeutig fördert. Wird von den konkreten Anforderungen und Konzepten abgesehen, kann diese Arbeit außerdem maßgeblich für die Umsetzung vergleichbarer *Full-Stack*-Projekte sein.

Inhaltsverzeichnis

1. Motivation

Die Verwaltung eigener Musiksammlungen ist in Zeiten von Musikstreamingdiensten vielleicht nicht mehr so bedeutend, wie sie einmal war. Nichtsdestotrotz gibt es Menschen, denen ihre eigene Sammlung ans Herz gewachsen ist und die die Unabhängigkeit von solchen Diensten schätzen. Die Grundidee hinter diesem Projekt ist es, Sammlungen die sich auf privaten Computern befinden in Netzwerken zugänglich zu machen, so dass viele Personen gleichzeitig die Sammlung nutzen können, um das gemeinsame Musikhören interaktiv zu gestalten. Darüber hinaus soll die Verwaltung auch intelligenter werden. Songs könnten beispielsweise von der Anwendung selbstständig passend ausgewählt oder Relationen zu Künstlern untereinander sichtbar gemacht werden. Im Vordergrund soll zunächst jedoch das Entwickeln einer Software stehen, dessen Handhabung einfach und intuitiv ist und mit der Spaß macht im Alltag zu interagieren.

Das Ziel des Projektes ist, eine moderne, also relativ komplexe und intelligente, Musikdatenverwaltung mit einer leichtgewichtigen, nutzerfreundlichen und endgeräteunabhängigen Anwendung zu vereinen.

2. Anforderungsanalyse

Um aus der Motivation heraus ein Konzept zu entwickeln, werden Anforderungen an das Projekt aus Sicht eines potentiellen Anwenders herausgearbeitet. Diese sollen während der Umsetzung stets maßgeblich dafür sein, welchen Weg die Entwicklung einschlägt. Welche Anforderungen nach dem Ende des Projekts tatsächlich umgesetzt werden und welche noch offen sind, wird sich im Fazit am Ende dieser Arbeit wiederfinden.

Übersicht

Der Nutzer möchte einen Überblick über seine Musiksammlung haben

Eine Sammlung von Musikdateien enthält idealerweise neben den Dateien selbst auch eine Reihe von Metadaten wie Künstler oder Album. Anhand dieser Metadaten soll eine vollständige und kategorisierte Übersicht über die Sammlung entstehen. Die Darstellung von Daten soll klar, ansprechend und intuitiv verständlich sein.

Sammlung erweitern

Der Nutzer möchte neue Musiktitel seiner Sammlung hinzufügen

Das Erweitern seiner Sammlung soll für den Nutzer ein leichtgewichtiger Prozess sein. Idealerweise beträgt die Anzahl an Interaktionen die dafür notwendig sind genau eins. Neue Titel sollen dann vollständig in die bestehenden Daten integriert werden, ohne dass ein lästiges Nachpflegen von Metadaten notwendig wird.

Abspielen von Musik

Der Nutzer möchte die Musik seiner Sammlung abspielen

Da das Abspielen von Musiktiteln als eine der Kernfunktionen einer solchen Anwendung verstanden wird, sollte es dem Nutzer möglichst einfach gemacht werden, zu jeder Zeit Einfluss auf die Wiedergabe zu nehmen. Diese Anforderung könnte man auch in zwei Teile teilen. Einerseits könnte Musik dort abgespielt werden, wo sie gespeichert ist, beispielsweise als zentraler Wiedergabeort in einem Haushalt, andererseits kann es erwünscht sein, die Wiedergabe auf dem Gerät des Endnutzers zu haben.

Erstellen von Wiedergabelisten

Der Nutzer möchte eigene Listen von Musiktiteln erstellen und speichern

Diese Anforderung ergibt sich nicht direkt aus der Motivation heraus, sondern eher aus der vorherigen Anforderung. Nicht nur das bloße Abspielen der Musik sondern auch das Bestimmen, welche Titel in welcher Reihenfolge laufen, stellt eine wichtige Funktion dar. Dem Nutzer soll ermöglicht werden, auf einfache Weise eigenständig Listen zu erstellen und zu bearbeiten. Diese Listen sollen natürlich nicht immer neu erstellt werden müssen, sondern ebenfalls in der Datenbank gespeichert werden.

Gleichzeitiger Zugriff

Mehrere Nutzer möchten gleichzeitig auf die Sammlung zugreifen

Musik wird häufig in Gesellschaft gehört. Was gespielt wird, soll in diesem Fall von allen potentiellen Nutzern beeinflusst werden können, um ein gemeinschaftliches Erlebnis zu erzeugen. Dabei ist auch zu beachten, dass vorherige Anforderungen für alle Nutzer gleichzeitig gelten sollen. Nimmt beispielsweise ein Nutzer Änderungen vor, sollten diese sofort für andere Nutzer sichtbar gemacht werden.

Intelligente Wiedergabe

Der Nutzer möchte, dass Musik selbstständig ausgewählt wird

Um die Anzahl von nötigen Interaktionen zur Musikwiedergabe zu minimieren, kommt eine automatische, intelligente Musikauswahl in Frage.

3. Konzeption

Diese Kapitel befasst sich mit dem Auflösen von Anforderungen in die Architektur des Projekts auf abstrakter Ebene. Dafür werden zunächst Anforderungen konkretisiert und in Applikations- und Datenstruktur überführt.

Die Hauptaufgabe dieses Projekts wird es sein, eine unbegrenzte Menge von Musiktiteln, in Form von Dateien, zentral zu speichern und in verarbeitungsfähige Daten zu überführen. Es müssen sinnvolle Funktionen zum Umgang mit diesen Daten und Dateien geschaffen werden. Hierfür werden Applikationen benötigt werden, die zum einen die Daten verwalten und bereitstellen, zum anderen sollen Daten veröffentlicht und eingesehen werden können.

3.1 Konzeptionelle Modelle und Relationen

Für die Erfassung von Daten wird zunächst die Modellierung einer Datenbankstruktur vorgenommen. In diesem Abschnitt wird erläutert, welche Entitäten erforderlich sind und wie deren Relationen zueinander sind.

Musiktitel, *Track*

Ein Musiktitel entspricht prinzipiell dem, was in einer Musikdatei gespeichert ist und steht im Zentrum der Datenstruktur.

Künstler, Interpret, *Artist*

Künstler sind Interpret mehrerer Musikstücke. Musikstücke können von mehreren Künstlern interpretiert werden.

Album, Veröffentlichung, *Release*

Ein Album enthält mehrere Titel, jeder Titel ist genau einem Album zugeordnet. Künstler eines Albums sind die Künstler aller Titel des Albums. Eine direkte Relation gibt es nicht.

Datei-Entität, *File-Entity*

Datei-Entitäten halten Informationen zum Dateiformat und erhalten die ursprünglichen Metadaten. Musiktitel können in mehreren Formaten (z.B. mp3 oder flac) vorliegen. Jede Datei ist genau einem Titel zugeordnet.

Wiedergabeliste, *Playlist*

Wiedergabelisten sind Kompositionen mehrerer Musiktitel, die von Nutzern angelegt werden. Titel können mehreren Listen zugeordnet werden.

Aktive Wiedergabe, *Player*

Player halten Informationen zur aktuellen Wiedergabe. Sie beinhalten mehrere Musiktitel, nämlich die, die für die Wiedergabe vorgesehen sind. Titel können auch verschiedenen Playern zugeordnet werden. Mehrere Player anzulegen soll dazu dienen, unterschiedliche Ausgabekanäle anzusteuern.

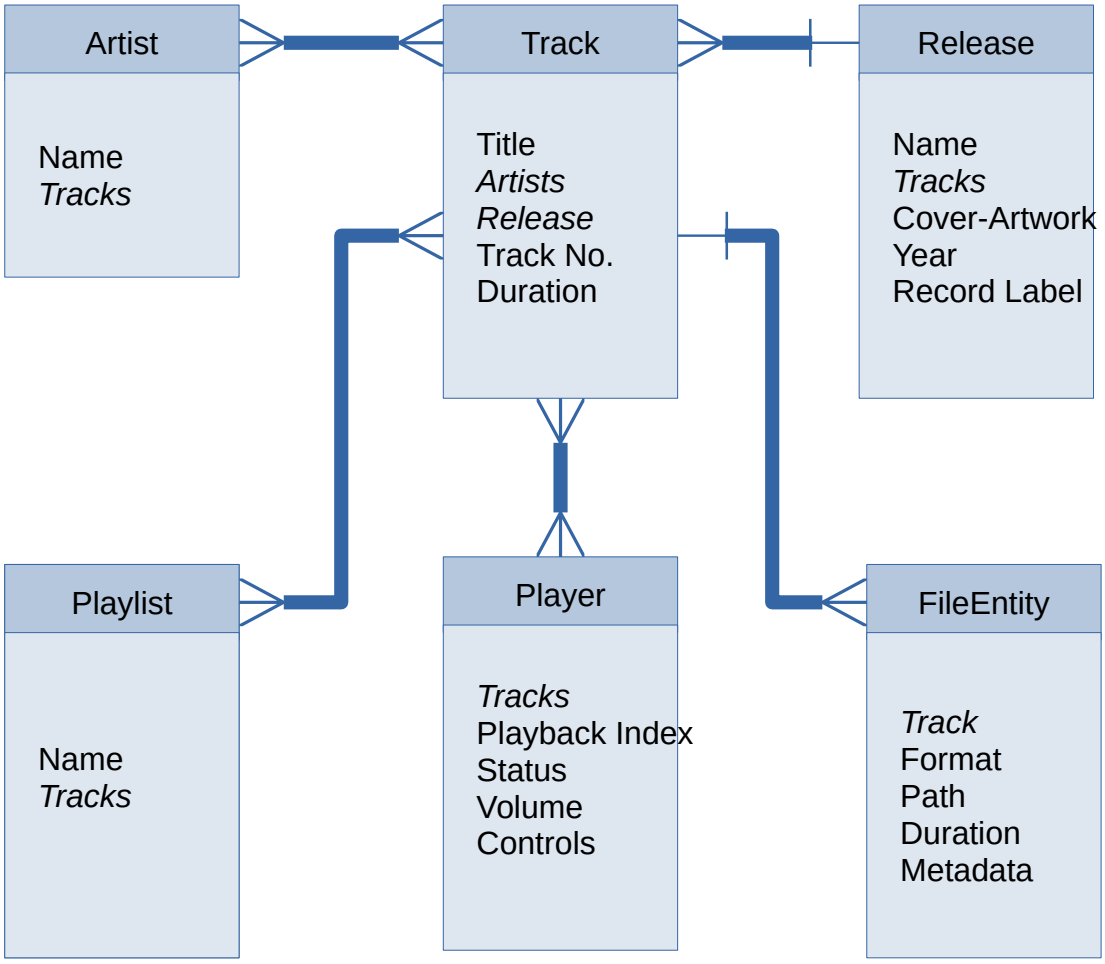

Abbildung 1: Konzeptionelle Modelle in Crow‘s-Foot-Notation

3.2 Datenverwaltung

Um ein System für die Bereitstellung der im vorherigen Abschnitt ermittelten Datenstrukturen zu errichten, wird eine alleinstehende Back-End-Applikation entworfen. Dafür ist im Inneren eine Datenbank zum Ablegen von Daten, sowie eine Geschäftslogik (*Business-Logik*) zum Verarbeiten der Daten erforderlich. Für die Darstellung nach außen wird das Gestalten einer Programmierschnittstelle (*API*) vorgenommen, welche als zentraler und strukturierter Zugriffspunkt dient. Hier können andere Applikationen Daten entnehmen oder einbringen.

Konkrete Anforderungen an das Back-End sind:

Datenzugriffe regulieren

Werden Entitäten angefordert, wird im System entschieden, welche Felder preisgegeben werden. Felder wie Dateipfad sind beispielsweise nicht für Konsumenten der API bestimmt. Außerdem ist es wichtig verknüpfte Entitäten zu verkapseln, um die Übertragungsmenge gering zu halten. Fordert eine Anwendung alle Datensätze von Künstlern an, ist es sinnvoll, zugeordnete Musiktitel nicht ebenfalls als Entiäten mit zu übermitteln, sondern einen Verweis darauf, wie diese zu erlangen sind.

Datensätze anlegen und verändern

Das Back-End soll Methoden zur Verfügung stellen, die es einer Anwendung ermöglichen, Daten in das System einzubringen. Daten müssen vom Back-End verifiziert und angepasst werden. Beispielsweise sollten Namen von Wiedergabelisten verändert werden können, niemals jedoch die ID, da es sich um den Primärschlüssel der Datenbank handelt.

Verarbeiten von Datei-Uploads

Aus einer Musikdatei gehen in der Regel mehrere Datensätze, wie Musiktitel, Künstler und Album hervor. Mp3-Dateien enthalten beispielsweise ID3-Header, welche Metadaten zu den oben genannten Attributen enthalten. Diese sollen im System extrahiert werden. Es kann sein, dass Künstler oder Album bereits in der Datenbank existieren, in dem Fall muss ein Titel den Datensätzen zugeordnet werden. Anderenfalls wird ein neuer Datensatz angelegt.

Wiedergabe von Musikdateien

Das Konzept sieht es zunächst vor, dass Musikdateien auf der Seite des Back-Ends abgespielt werden (zentrale Wiedergabe). Die Wiedergabe darf dabei keine anderen Prozesse blockieren.

Steuerung der Musikwiedergabe

Die Programmlogik muss in der Lage sein, Anfragen zur aktiven Musikwiedergabe zu verarbeiten. Soll beispielsweise ein Titel in der Wiedergabeliste übersprungen werden, so muss der nächste aus der Datenbank angefordert und an das Wiedergabesystem übermittelt werden. Anfragen müssen außerdem validiert werden. Zum Beispiel gilt, dass das überspringen von Titeln nur möglich ist, wenn nicht bereits das Ende der Wiedergabeliste erreicht ist.

3.3 Nutzeroberfläche

Für die Vermittlung zwischen Anwender und System sind Front-End-Anwendungen zuständig. Zunächst wird der Zugriff über eine Webseite realisiert, die von einem Server, auf dem sich auch das Back-End befindet, bereitgestellt wird. Auch eigenständige, mobile Anwendungen sind denkbar, die direkt auf die API zugreifen können und deutlich leichtgewichtiger gestaltet sind, als die Webseite. Das Front-End soll Anforderungen des Nutzers in Form von Interaktionen lösen.

3.3.1 Anforderungen an die Nutzeroberfläche

Konkrete Anforderungen an die Nutzeroberfläche sind:

Ansprechende Übersicht über die Daten

Die Anwendung sollte grundsätzlich die Möglichkeit bieten, alle Datensätze einer Tabelle wie Titel, Album oder Künstler einsehen zu können. Dazu gehört auch, Datensätze beispielsweise mit einer Suchfunktion zu filtern. Es werden zu dem Entscheidungen getroffen, welche Attribute einer Entität dargestellt werden. Primärschlüssel der Datenbank sind in der Regel dafür nicht vorgesehen. Das Überladen von Listenansichten ist ebenfalls zu vermeiden. Dafür sind Detailansichten für einzelne Datensätze vorgesehen.

Auswahl von Titeln zur Wiedergabe

Das Abspielen von Titeln ist eine der Kernaufgaben des Anwendungen. Sie sollte intuitiv und universell zugänglich sein, beispielsweise durch Schaltflächen an den Titeln.

Steuerung der Wiedergabe

Steuerelemente sollten stets sichtbar sein und außerdem Auskunft darüber geben, in welchem Zustand sich die aktive Wiedergabe befindet.

Anlegen eigener Wiedergabelisten

Wiedergabelisten sollten vom Nutzer angelegt und benannt werden können.

Hinzufügen neuer Titel

Um Titel einer Wiedergabeliste hinzuzufügen, sollte sie sich im Vordergrund verankern lassen können. Dadurch hat der Anwender die Möglichkeit ihr aus jeder Ansicht heraus Titel hinzuzufügen und gleichzeitig das Resultat einzusehen.

3.3.2 Wire Frames

Die Nutzeroberfläche sollte Komponenten wie den Player und das Auswahlmenü stets an der Oberfläche halten. In mobilen Ansichten können sie ausgeblendet werden, sind aber über Schaltflächen immer zugänglich.

Listenansichten mit Menü (links) und Player (rechts)

Abbildung 2: Wireframe Listenansicht

Detailansicht Künstler

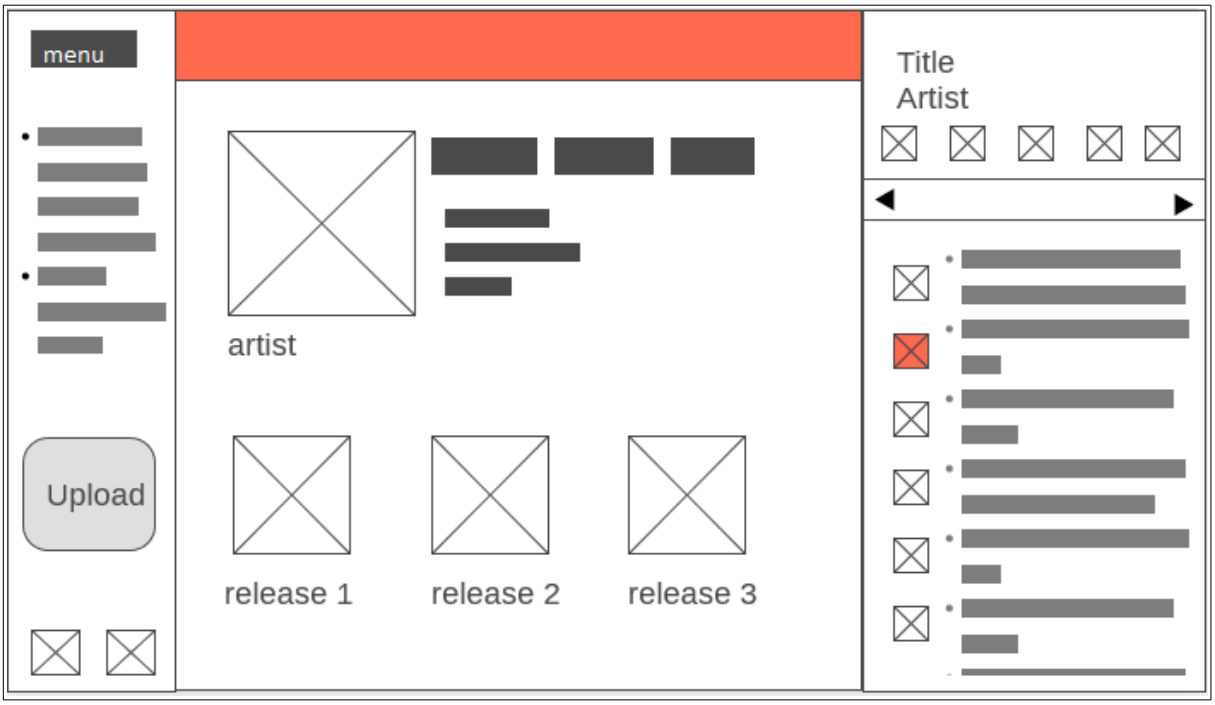

Abbildung 3: Detailansicht Künstler

Einzelelement Album

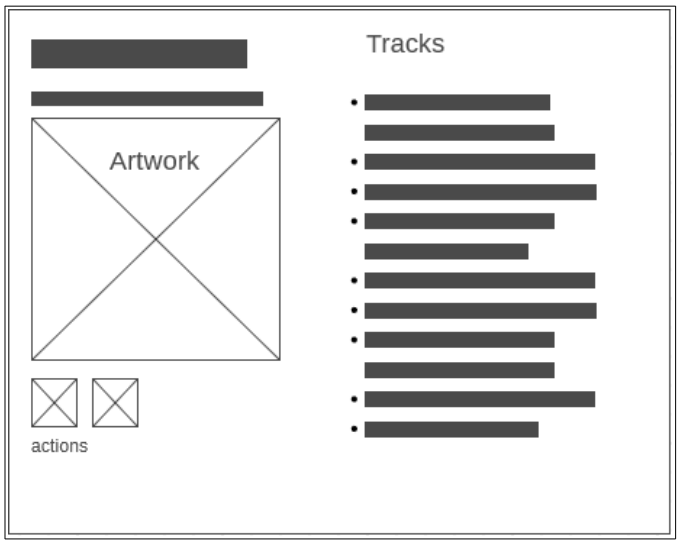

Abbildung 4: Wire Frame Album-Element

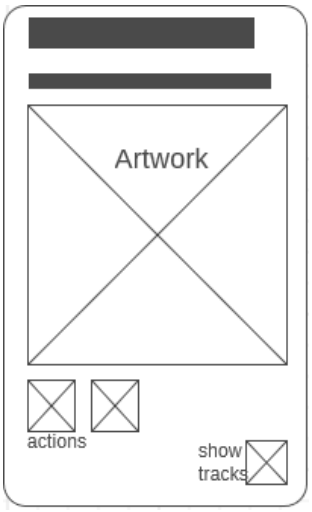

Abbildung 5: Wire Frame Kompakte Detailansicht Album

4. Technologien

Um die Konzepte umzusetzen wurden folgende Technologien gewählt:

4.1 Spring

Das auf Java basierende *Spring Framework* verwaltet gewöhnliche Java Objekte und vernetzt sie per konfigurierter *Depencency Injection* als sogenannte Spring *Beans*. Spring bietet viele einsteigerfreundliche Guides und Tutorials und ist daher für Java-Entwickler sehr interessant.
(Rahn, 2019)

Spring Boot als Teil des *Spring Frameworks* ist darauf spezialisiert mit geringem Konfigurationsaufwand solche vernetzten Architekturen zu realisieren. Da Spring auf Java basiert hat man neben Java als Programmiersprache auch die Möglichkeit ***Kotlin*** zu nutzen. Die Wahl viel für dieses Projekt auf letztere. (Spring.io, 2019, Kotlin 2019)

Im Projekt wird außerdem eine **MySQL** Datenbank genutzt, da diese ebenfalls von Spring unterstützt wird. Spring nutzt dafür die **Java Persistance API**. (Oracle, 2019)

Zum extrahieren der Metadaten aus Mp3-Dateien wird die Bibliothek ***Jaudiotagger*** verwendet. (JThink, 2019)

4.2 Angular

Angular ist ein Framework für Web-Anwendungen. Anders als sein Vorgänger AngularJS wird nicht in JavaScript sondern in ***TypeScript*** programmiert, eine Sprache die klassenbasiert und objektorientiert ist. Dadurch wird es ansprechender für solche Entwickler, die Programmiersprachen wie Java gewohnt sind.
(TypeScript, 2019, Angular.de 2019)

Angular Material

Angular Material liefert vielfältige, ansprechendee Design-Komponenten für Angular. Es wird im Projekt eingesetzt, um den Aufwand bei der Gestaltung der Front-End-Anwendung zu verringern, damit mehr Zeit in die eigentliche Programmierung gesteckt werden kann.
(AngularMaterial, 2019)

5. Umsetzung

Das Projekt unterteilt sich in zwei voneinander relativ unabhängige Applikationen, die über eine Programmierschnittstelle (API) miteinander kommunizieren. Relativ unabhängig daher, weil die Applikationen selbstständig lauffähig sind, allerdings ist die Front-End-Applikation ohne bereitstehende API des Back-Ends nicht sinnvoll zu gebrauchen. Anders herum läuft das Back-End zwar ohne Front-End theoretisch problemlos, lässt sich aber nicht steuern. Für die Back-End-Applikation muss außerdem eine Datenbank initialisiert werden und im Hintergrund bereitstehen. Damit die Back-End-Applikation Dateien speichern kann, müssen gegebenenfalls Berechtigungen dafür zur Verfügung stehen.

In den folgenden Abschnitten werden zunächst die Klassen der Anwendungen auf einer höheren Abstraktionsebene erläutert. Eine tabellarische, detaillierte Dokumentation befindet sich im Anhang. (Kapitel 10)

5.1 Back-End-Applikation – Projektname „Copper“

Das Back-End besteht im Wesentlichen aus Applikation, Datenbank und Dateispeicher, wobei letzterer nichts weiter als ein lokales Verzeichnis ist. Die Applikation ist in *Kotlin* geschrieben und nutzt das *Spring Framework*. Sie verarbeitet eingehende Anfragen an der API in Zugriffe auf die Datenbank. Die Datenbank ist eine MySQL Datenbank, wobei natives SQL in den selteneren fällen im Code zu finden ist.

Packages

Die eigentliche Applikation ist strukturiert in Pakete, die den konzeptionellen Datenmodellen aus Kapitel 3.1 folgen. Jedes Paket besteht aus:

Datenklassen:	Hauptelemente und Definition für Tabellen der Datenbank
Controller:	Definieren die API leiten dort eingehende Anfragen bezüglich der Datenklasse weiter
Datentransferobjekte:	Nehmen eingehende Anfragen als Objekt entgegen
Services:	Entscheiden, wie eingehende Anfragen verarbeitet werden
Repositories:	Speichern und entnehmen Daten aus der Datenbank

Ressourcenklassen: Definieren die ausgehenden Daten an der API

Resource-Assembler: Erzeugen Ressourcenklassen aus Entitäten und betten andere Ressourcen oder Links zu anderen Ressourcen ein

Außerhalb der Paketstruktur gesellen sich noch noch die Klasse *CopperApplication*, welche den Einstiegspunkt bereithält, Konfigurationsdateien sowie die *MediaPlayer*-Applikations-Klasse, die für die Wiedergabe von Musik in einem eigenen *Thread* zuständig ist und noch einige weitere unterstützende Klassen.

5.1.1 Datenklassen

```
package com.cv.copper.release

import ...

@Entity
@Table(name = "Releases")
data class Release (
        @Id
        @GeneratedValue(generator = "uuid2")
        @GenericGenerator(name = "uuid2", strategy =
"org.hibernate.id.UUIDGenerator")
        @Column(length = 36)
        var id: String?,
        @Column
        var name: String?,
        @Column
        var pictureUri: String?,
        @Column
        @OneToMany(mappedBy = "release")
        var tracks: List<Track>? = mutableListOf(),
        @Column
        var year: String?,
        @Column
        var label: String?,
        @Lob
        @Column(length = 66000)
        var coverArt: String? = null
) : Auditable() {
        override fun toString(): String {
                return "Release(id=$id, name=$name, year=$year)"
        }

}
```

Abbildung 6: Aufbau einer Datenklasse am Beispiel von Track.kt

Die Hauptklassen des Back-Ends sind die sogenannten Datenklassen oder Entitätenklassen. Sie legen fest, welche Daten gehalten werden, wie Entitäten der Datenbank generiert und wie die Relationen zu anderen Entitäten sind. Prinzipiell findet hier die gesamte Modellierung der Datenbank statt. Das geschieht über so genante *Annotations* der *Java Persistance API* sowie

den Feldnamen der Klassen. (Oracle, 2019) Folgende *Annotations* finden wurden im Projekt bisher genutzt:

@Entity	Definiert eine Klasse als Entität
@Table	Definiert den Tabellennamen
@Id	Legt den Primärschlüssel fest
@Column	Spezifiziert Eigenschaften wie Spaltenname und Datengröße
@OneToOne	Definiert eine 1 zu 1 Relation
@OneToMany	Definiert eine 1 zu n Relation
@ManyToMany	Definiert eine m zu n Relation und legt fest, wie das äquivalente Feld der assoziierten Klasse heißt. Die Zuordnung findet immer nur auf einer Seite der Relation statt.
@Lob	Large Objects: Sehr große Felder, wie beispielsweise die kodierten Cover-Artworks eines Albums
@Convert	Konvertiert Typen (z.B.) Path in speicherbare Datentypen
@Transient	Markiert Felder, die nicht in der Datenbank angelegt werden sollen.

Hier kommen außerdem *Kotlins ‚data classes‘* zum Einsatz, welche die Codemenge durch automatisches Ableiten von Feldern aus Konstruktor-Paramentern gering halten. Auch *getter*- und *setter*-Methoden müssen nicht explizit angegeben werden. Durch die Relationen von Entitäten untereinander ist darauf zu achten *toString*-Methoden zu überschreiben. Anderenfalls werden Endlosschleifen erzeugt, da der String der ersten Entität den String der verknüpften Entität darstellt, welcher wiederum den der ersten darstellt und so weiter. Datenklassen müssen das Interface *Serializable* implementieren, damit sie in der Datenbank abgelegt werden können. Außerdem ist die Superklasse der Datenklassen *Auditable*. Dadurch werden für die Entitäten Erstell- und Änderungsdatum generiert. Die IDs werden als Universally Unique Identifier (UUID) generiert mit Ausnahme der der Player-Entität.

5.1.2 Repositories

Für das Speichern und Abfragen von Entitäten aus einer Datenbank sind die sogenannten Repository-Interfaces zuständig. Sie implementieren das *JpaRepository-Interface* unter Angabe der gewünschten Datenklasse. Durch Vererbung werden sie für andere Spring Komponenten injizierbar und erhalten außerdem die Möglichkeit, SQL-Abfragen aus Methodennamen abzuleiten. Diese werden ausschließlich deklariert und erhalten ihre Funktionalität über die Auswahl an Schlüsselwörtern, Entitäten und deren Feldern.

(Oracle, 2019)

```
package com.cv.copper.artist

import org.springframework.data.jpa.repository.JpaRepository
import org.springframework.data.domain.Page
import org.springframework.data.domain.Pageable
import org.springframework.data.jpa.repository.Query

interface ArtistRepository : JpaRepository<Artist, String> {

    fun findByName(name: String): Artist?
    fun findAllByNameContains(p: Pageable, query: String): Page<Artist>
    @Query("SELECT * FROM db_copper.artists ORDER BY RAND() LIMIT ?1 ;",
nativeQuery = true)
    fun findRandom(number: Long): List<Artist>

}
```

Abbildung 7: Beispielhafter Aufbau eines Repositories

Häufig genutzte Abfragen wie *findAll* (Alle Datensätze einer Tabelle) oder *findById* (Datensatz mit entsprechender *ID*) werden ebenfalls durch das *JpaRepository-Interface* vererbt und müssen daher nicht zusätzlich in den eigens geschriebenen Interfaces deklariert werden.

Die Rückgabewerte sind bis auf wenige Ausnahmen Datenklassen selbst oder Listen bzw. Seiten-Objekte davon *(List<T>, Page<T>)*. Um ein Seitenobjekt zurückzugeben, wird der Methode zuvor ein Objekt vom Typ *Pageable* übergeben, welches Daten wie Seitenzahl und Seitennummer enthält, um den entsprechenden Ausschnitt an Datensätzen zu finden. Einige Methoden die nur sehr schwer oder eventuell gar nicht über die Methodennamen abzuleiten waren, sind in nativem SQL geschrieben.

5.1.3 Services

Die Services bilden das Zentrum der Verarbeitung von Anfragen. Sie vermitteln zwischen API und Datenbank. Sie entscheiden, wie angehende Anfragen auf der API und ausgehende Daten aus der Datenbank gehandhabt beziehungsweise verarbeitet werden.

Es gilt als gute Praxis, Services in Interface und Implementationsklassen zu teilen (Vikdor, 2019). Dadurch ist es nämlich möglich, mehrere Implementationsklassen des selben Services zu schreiben und je nach Anforderung zur Laufzeit eine bestimmte bereitzustellen. Beispielsweise können für Testzwecke Implementationsklassen, die Pseudo-Daten liefern injiziert werden, anstatt wie im Normalbetrieb die Daten der *Repositories* zu übermitteln. Derzeit findet sich im Projekt zwar jeweils nur eine Implementationsklasse wieder, allerdings wird sich das in Zukunft wahrscheinlich noch ändern, weshalb sich diese Praxis auch in der Projektstruktur wiederfindet.

Einfache Anfragen wie *findAll* – also finde alle Entitäten einer Tabelle – werden in der Regel nur ans Repository weitergeleitet und von dort wieder zurück an den Controller. Sie enthalten meistens nur eine oder wenige Zeilen Code, doch es gibt auch wesentlich komplexere Anfragen, wie das Verarbeiten von Datei-Uploads. Um die Prozesse im Back-End zu beschreiben werden im folgenden Abschnitt die einzelnen Anforderungen an die Logik und gegebenenfalls besondere Herausforderungen erläutert.

Abfragen, Erstellen, Ändern und Löschen von Entitäten

Dies sind die am häufigsten genutzten und einfachsten Prozesse, da sie in der Regel direkt an Datenbankoperationen weitergeleitet werden. In einigen Methoden werden Rückgabewerte der Datenbank von Listen-Objekten in Seiten-Objekte überführt, da Listen-Objekte fehlerhaft in Ressourcen überführt werden.

Beim Erstellen von neuen Entitäten aus Datentransferobjekten wird festgelegt, welche Felder des Datentransferobjekts im Konstruktor der Entität übernommen werden und welche ignoriert werden. Beispielsweise werden *ID*-Felder immer ignoriert, da diese beim Speichern der Entität in der Datenbank generiert werden.

Änderungen an bestehenden Wiedergabelisten durchführen

Änderungen an Wiedergabelisten können sein, dass ihr ein Musiktitel hinzugefügt oder ein vorhandener Titel bewegt wird. In beiden Fällen wird zunächst verifiziert, ob Musiktitel sowie Wiedergabeliste als Entität in der Datenbank existieren. Da es durchaus sein kann, dass mehrere Front-Ends gleichzeitig die selben Listen bearbeiten oder ansehen, wird nach jeder Änderung ein Web-Socket-Event ausgelöst, was den Front-Ends ein aktualisieren der Daten signalisiert.

Künstler eines Musikalbums finden und sortieren

Der oder die Künstler eines Musikalbums werden über die Titel des Albums bezogen. Eine direkte Relation von Künstler und Album gibt es bewusst nicht. Da ein Album möglicherweise mehrere verschiedene Künstler hat, die dazu auch noch in verschiedener Häufigkeit darauf vorkommen können, wird die Rückgabe so verarbeitet, dass die Künstler nach auftretender Häufigkeit umsortiert werden und mehrfach auftretende nur einmal in der Rückgabe zu finden sind.

Steuerung des Media Players

Eine weitere wichtige Aufgabe der Geschäftslogik ist das steuern des Media Players. Von den Controllern können Datentransferobjekte eingehen, die eine Operation am Player signalisieren. Die Felder des Objektes werden dann innerhalb der Logik in Methoden des Media Players übersetzt. Auch der Media Player selbst beauftragt einen Service mit dem finden des nächsten Musiktitels, wenn eine Wiedergabe beendet ist.

Hochladen von Dateien

Das Hochladen von Musikdateien stellt eine komplexe Anforderung an die Logik. Es wird geprüft, ob die Dateiendung valide ist – derzeit ist nur .mp3 erlaubt – und falls ja, werden die aus dem ID3-Header der Datei extrahierten Metadaten ausgewertet. Die Logik prüft dann, ob Künstler oder Album bereits existieren, anderenfalls werden sie vorher generiert. Vor dem Speichern des neuen Titels wird noch einmal geprüft, ob ein Musiktitel mit gleichen Metadaten bereits existiert.

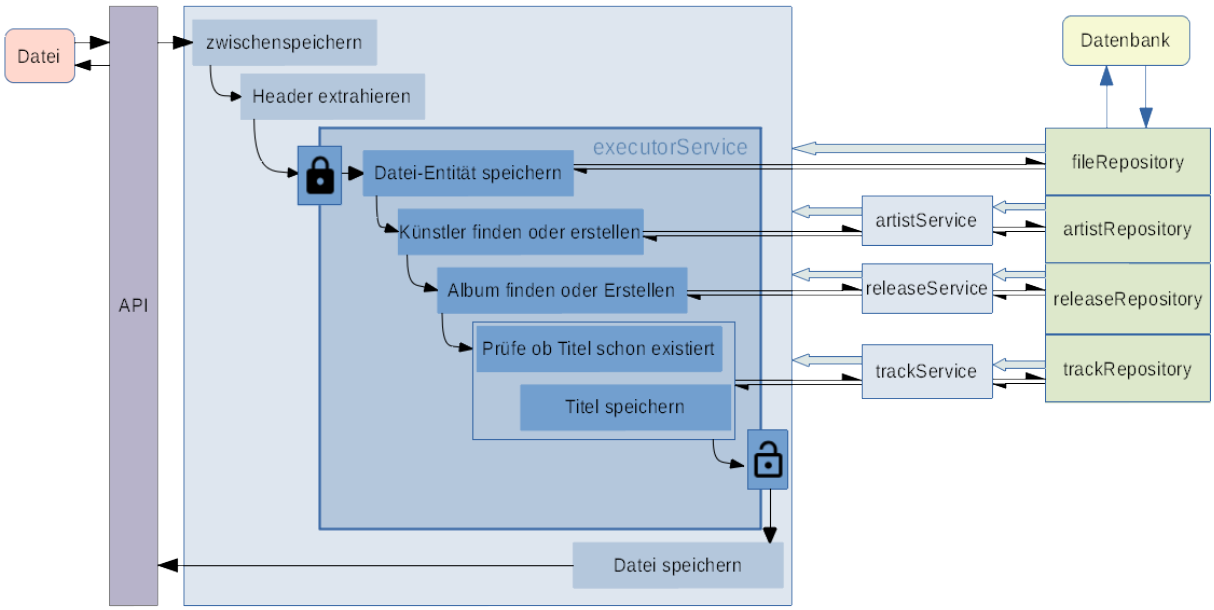

Abbildung 8: Dateiupload Flow Chart mit Executor-Service

Die „Lock“-Symbole stammen von Googles Material Design (Material.io, 2019)

Werden mehrere Dateien in schneller Abfolge hochgeladen, ergibt sich eine besondere Herausforderung. Die Datenbankzugriffe sind unter Umständen langsamer, als das Eingehen der Dateien. Dies hat zur Folge, dass beispielsweise ein neuer Künstler mehrfach erstellt wird, bevor erkannt wird, dass er bereits existiert. Um diesem Problem entgegenzuwirken, wird *Kotlins Executor-Service* genutzt, welcher sicherstellt, dass gleiche Prozeduren nur einmal zur Zeit laufen.

5.1.4 Ressourcen und Resource-Assembler

Die Ressourcenklassen dienen als Vorlage für ausgehende Ressourcen über die API. Durch ihre Superklasse werden ihnen außerdem Felder für eingebettete oder verlinkte Ressourcen bereitgestellt. Beispielsweise werden in Album Ressourcen Künstler eingebettet und die einzelnen Musiktitel verlinkt. Durch den Link können Titel dann bei Bedarf nachgeladen werden und müssen nicht bei jeder Anfrage übermittelt werden. Das Erzeugen einer Ressource aus einer Entität wird von den *Resource-Assemblern* übernommen. Über Controller-Methoden werden andere Ressourcen eingebettet oder verlinkt. Für jede Entität wird eine Ressourcenklasse bereitgestellt, welche genau festlegt, welche Daten über, die API rausgehen. Es eine Ausnahme bilden die kodierten *Cover-Artworks* der Musikalben, welche auf Grund ihrer Größe über eine extra Ressource und nur bei konkreter Anfrage ausgegeben werden.

5.1.5 Controller und Datentransferobjekte

Die Controllerklassen definieren die API des Webservers und leiten eingehende Anfragen an Services weiter. Außerdem definieren sie das Format der ausgehenden Ressourcen.

Jeder Methode im Controller ist ein Endpunkt zugewiesen, also eine Adresse unter der sie zu erreichen ist, sowie einer erforderlichen HTTP-Methode. Außerdem ist noch festgelegt, welche Pfadvariablen es gibt und wie der *HTTP-Message-Body* auszusehen hat. Für letzteres werden Datentransferobjekte genutzt, welche von der Struktur her den Datenklassen entsprechen, allerdings nur über primitive Datentypen verfügen. Neben HTTP-Endpunkten werden auch solche für Web-Sockets bereitgestellt. Eingehende Anfragen liefern immer eine *ResponseEntity* zurück, also eine HTTP-Antwort mit Statuscode. Eine genaue Dokumentation der API befindet sich im Anhang. (SpringDocs, 2019)

```
package com.cv.copper.track

import...

@RestController
@RequestMapping("", produces = [MediaTypes.HAL_JSON_VALUE])
class TrackController {

    @Autowired
    private lateinit var trackResourceAssembler: TrackResourceAssembler
    @Autowired
    private lateinit var trackService: TrackService

    @GetMapping("/tracks")
    fun findAll(@RequestParam(value = "query", required = false) query: String?,
pageable: Pageable, assembler: PagedResourcesAssembler<Track>):
PagedResources<TrackResource> {
        val trackPage = if (query == null) trackService.findAll(pageable) else
trackService.findAll(pageable, query)
        return assembler.toResource(trackPage, trackResourceAssembler)
    }
    @GetMapping("/tracks/{trackId}")
    fun findById(@PathVariable("trackId") trackId: String):
ResponseEntity<TrackResource> {
        val track = trackService.findById(trackId)
        track?.let { return
ResponseEntity.ok(trackResourceAssembler.toResource(track))} ?: return
ResponseEntity.notFound().build()
    }
}
```

Abbildung 9: Ausschnitt aus einer Controller Klasse

5.1.6 MediaPlayer

Diese Klasse *MediaPlayerApplication* erbt von der Klasse *Application* und ist damit in einem eigenen *Thread* ausführbar. Sie wird beim starten des Programms initialisiert und wartet dann auf eingehende Musikdateien zur Wiedergabe. Diese wird von einer Instanz des Typs *MediaPlayer* übernommen, die Außerdem Methoden für *stop*, *start* und pausieren der Wiedergabe enthält. Ist die Wiedergabe eines Titels am Ende angelangt, wird der *PlayerService* mit dem finden des nächsten Titels beauftragt.

5.1.7 Konfigurationsklassen

In den drei vorhandenen Konfigurationsklassen befinden sich Einstellungen für Web Sockets sowie *Auditing*, also das Erzeugen von Erstell- und Änderungsdatum für Entitäten. Außerdem wird in der *AppConfig* festgehalten, welche Herkunftsadressen für einen Zugriff auf die API erlaubt sind und welche HTTP-Methoden sie anwenden dürfen.

Einstellungen für die Datenbank befinden sich in der Datei *application.properties.*

5.2 Front-End-Applikation – Arbeitstitel „Azurite“

Für das Front-End wird das *Angular Framework* verwendet und in *TypeScript* geschrieben. Es besteht im wesentlichen aus Komponenten, injizierbaren Services und *Model*-Klassen, die den konzeptionellen Modellen aus Kapitel 3.1 nachempfunden sind. Die Komponenten sind wiederverwertbare Elemente, und setzen sich aus HTML und CSS-Datei zur Darstellung, sowie einer *TypeScript*-Datei für die Funktionen und Attribute zusammen. Die Services werden in Komponenten injiziert und enthalten alle Methoden, die für die Anfragen von Daten an der API erforderlich. Sie halten außerdem veränderbare Daten die dann von Komponenten beobachtet werden können (*Observables*). Änderungen dieser Daten werden dann auch von abonnierenden Komponenten wahrgenommen und angepasst. Daneben wird noch eine Direktive verwendet, die wiederum in Komponenten aufgerufen werden kann, um ihnen die Funktionalität des *Drag-And-Drop-Uploads* mitzugeben. Der Code dieser Direktive basiert auf einem Tutorial von Luis Moncaris, veröffentlicht auf Scotch.io (Moncaris, 2017).

Das Projekt enthält neben den selbst geschriebenen Komponenten außerdem noch solche, die über *Angular Material* Module geladen werden. Die Angular Material Bibliothek bietet viele vorgefertigte, ansprechend gestaltete Komponenten, wie Tabellen, Eingabefeldern und Navigationsleisten und wird im Projekt vielseitig eingesetzt. Außerdem wird ein

vorgefertigtes Farbschema genutzt. Das Verwenden von Angular Material verringert den Design-Aufwand deutlich und ermöglicht mehr Zeit in die Programmierung zu stecken.

Im folgenden Abschnitt werden die einzelnen Ansichten des Front-Ends vorgestellt und deren Komponenten und Funktionalität beschrieben.

5.2.1 Ansichten

Die Ansichten bestehen immer aus der Navigationskomponente, der Player-Komponente, der Titelleiste und dem eigentlichen Inhalt der Ansicht. Auf kleinen Bildschirmen werden Navigation und Player ausgeblendet und lassen sich über Schaltflächen in der Titelleiste wieder einblenden.

Verwendete Symbole stammen von Googles Material Design (Material.io, 2019)

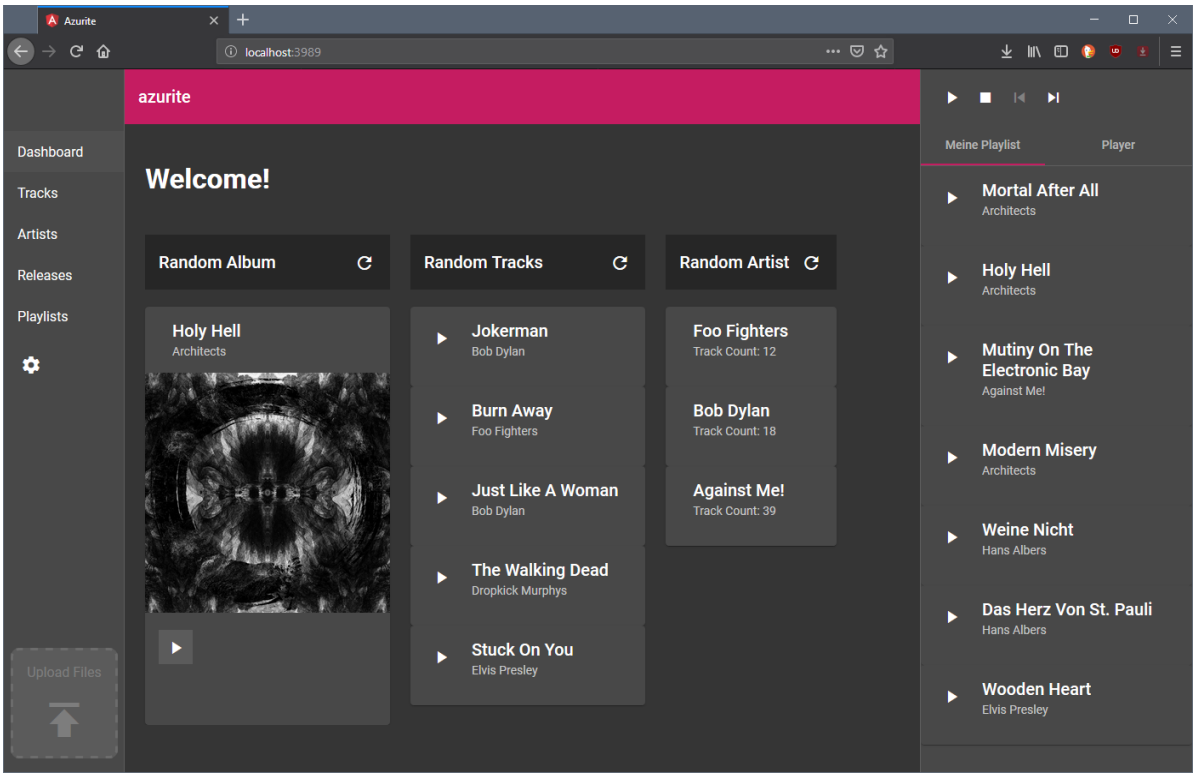

Abbildung 10: Dashboard-Ansicht mit Player und Menü

5.2.1.1 Navigation

Die Navigationsleiste führt neben der eigentlichen Navigation außerdem noch eine Schaltfläche für ein Einstellungsmenü, in der die Adresse der API geändert werden kann.

Unten sieht man einen Bereich der für den Upload von Dateien gedacht ist. Mp3-Dateien können hier hereingezogen (*Drag and Drop*) werden, um sie dann ins Back-End zu übermittelt. Ein Klick auf den Bereich öffnet zudem einen Dialog zur Dateiauswahl.

5.2.1.2 Player

Die *Player*-Komponente befindet sich am rechten Rand der Ansicht. Hier werden alle Musiktitel inklusive Interpret der aktiven Wiedergabe gelistet. Im oberen Bereich befindet sich ein Steuerungselement zum Anhalten, Fortführen oder Zurück- bzw. Überspringen von Wiedergaben. Der aktive Titel ist mit einem farbigen *Play*-Symbol markiert, die anderen Titel haben weiße *Play*-Symbole. Ein Klick darauf startet die Wiedergabe des jeweiligen Titels.

Eine zunächst versteckte Funktion des Players ist der sogenannte *Playlist*-Editor. Hier können eigens erstellte Wiedergabenlisten verankert werden, um sie zu bearbeiten. Dadurch ist die Liste immer sichtbar, wenn der Nutzer durch die übrigen Ansichten geht, um Titel der Liste hinzuzufügen. Der *Playlist*-Editor wird aktiviert, sobald ein Nutzer eine Wiedergabeliste zum Editieren in der *Playlist*-Ansicht ausgewählt hat. Danach kann vom *Playlist*-Editor Modus zum Player-Modus über Reiter am oberen Rand gewechselt werden.

Die Reihenfolge der Titel kann hier ebenfalls mittels *Drag-and-Drop* geändert werden. Das Löschen von Titeln aus Wiedergabelisten ist bisher noch nicht implementiert.

Web Sockets

Beim Initialisieren der Ansicht werden Web-Socket Verbindungen aufgebaut. Werden in den Entiäten in dieser Ansicht Änderungen im Back-End vorgenommen, leiten die zuständigen Services eine Benachrichtigung über einen Web-Socket ein, was dann über die Controller ausgeführt wird. Das Front-End lädt die jeweiligen Daten dann neu, sodass allen Anwendern stets die gleichen Informationen vorliegen. Ohne diese Vorkehrung würde das simultane Bearbeiten von Listen zu unvorhersehbaren Resultaten führen.
Mögliche Änderungen sind:

- Die Wiedergabe wird gestoppt oder fortgesetzt
- Eine Wiedergabe ist durchgelaufen und der nächste Titel startet
- Die Reihenfolge der Titel wird geändert

- Die Titel der Liste werden neu gesetzt oder es werden Titel hinzugefügt
- Ein anderer Titel in der vorliegenden Liste wird gestartet

5.2.1.3 Dashboard

Die *Dashboard*-Ansicht bildet den Einstiegspunkt in das Front-End. Sie ist konzipiert für eine Reihe verschiedenartiger Operationen. Bisher umgesetzte Operation sind das Anzeigen von zufällig ausgewählten Titeln, Künstlern oder auch Musikalben.

Die zufällige Auswahl wird bei jedem Aufrufen der Ansicht verändert. Darüber hinaus gibt es die Möglichkeit über einen *Refresh-Button* eine neue Auswahl zu erhalten.

Ein Klick auf auf die Elemente „*Random* Artist“ oder „*Random* Release“ öffnet die jeweilige Detailseite, für Musiktitel gibt es derzeit keine Detailansicht. Titel und Alben können über den vorhandenen *Play-Button* direkt abgespielt werden

5.2.1.4 Listen-Ansichten

Die übrigen Ansichten, die in der Navigation zu finden sind, beinhalten Sammlungen von Musiktiteln, Alben, Künstlern oder Wiedergabelisten. Diese Ansichten zeigen zunächst eine Seite aller Einträge einer Entität als Tabellenansicht. Durch das seitenweise Laden von Daten wird die Übertragungsmenge und damit die Ladezeit der jeweiligen Ansicht reduziert. Sie alle benutzen die *Collection-Container-Komponente* als Vorlage, dadurch werden sie mit einem Freitext-Suchfeld und einem *Scroll*-Modul ausgestattet, in der die eigentliche Tabelle angezeigt wird. Dieses *Scroll*-Modul verfügt über eine Methode zum Erkennen, ob das Ende der Tabelle erreicht ist. In diesem Fall werden die Komponenten informiert und beauftragen das Laden der nächsten Seite, die dann unten angefügt wird. Dadurch entsteht das Gefühl die gesamte Datenmenge einsehen zu können, ohne dass diese durch Interaktion des Nutzers nachgeladen werden müssen oder dass alles zuvor geladen werden muss.

Die Freitext-Suche übermittelt eine Suchanfrage an das Back-End zum finden aller Datensätze, die die eingegebe Zeichenfolge im Namen enthalten.

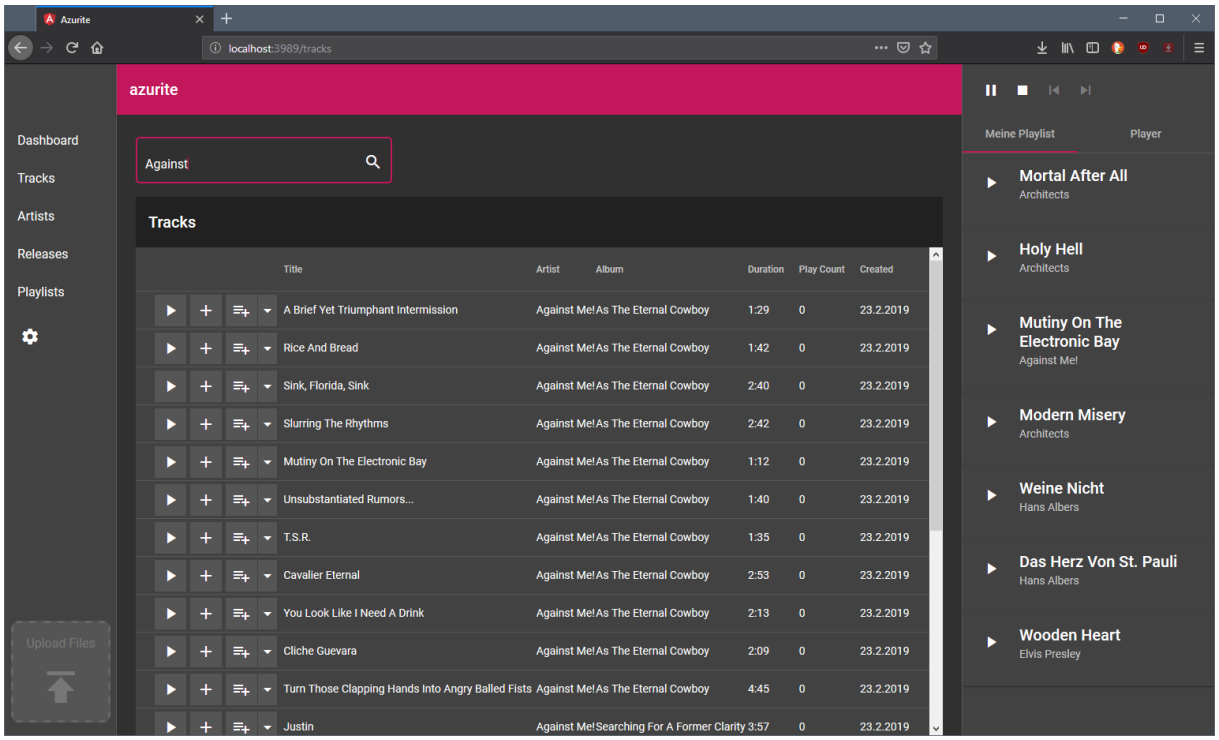

Abbildung 11: Listenansicht: ‚Tracks' mit Suchanfrage

5.2.1.5 *Tracks*-Ansicht

Diese Ansicht ist die erste der vier Ansichten die in der Navigation verankert sind. Sie liefert eine Tabelle aller Musiktitel mit dazugehörigen Daten wie Künstler, Album, Dauer, Erstelldatum und der Anzahl, wie oft ein Titel gespielt wurde. Klickt man auf den Titel einer Spalte, wird eine Sortierung nach dem jeweiligen Feld ausgeführt. Diese Sortierung findet ebenfalls im Back-End statt, es wird also nach allen Datensätzen sortiert und nicht nur nach denen, die zuvor geladen wurden. Zusätzlich wird jeder Datensatz mit Aktionen ausgestattet.

Die Aktionen sind in der Komponente *Track-Actions-Component* festgelegt und beinhalten:

- das Abspielen eines Titels
- Das Anhängen eines Titels an die aktive Wiedergabe *(Player)*
- Das Anhängen eines Titels, an die im *Playlist-Editor* geöffnete Liste
- Auswahl zum Anhängen eines Titels an eine andere Wiedergabeliste

5.2.1.6 *Releases*-Ansicht

Die *Releases*-Ansicht liefert eine Übersicht aller Veröffentlichungen – auch Alben genannt. In der Kopfzeile neben dem Suchfeld, kann die Ansichtsweise umgestellt werden. Standardmäßig ist die Tabellenansicht gewählt. Diese enthält die Daten Titel, Künstler und Erstelldatum. Ein Klick auf einen Datensatz öffnet die Detailansicht des Albums.

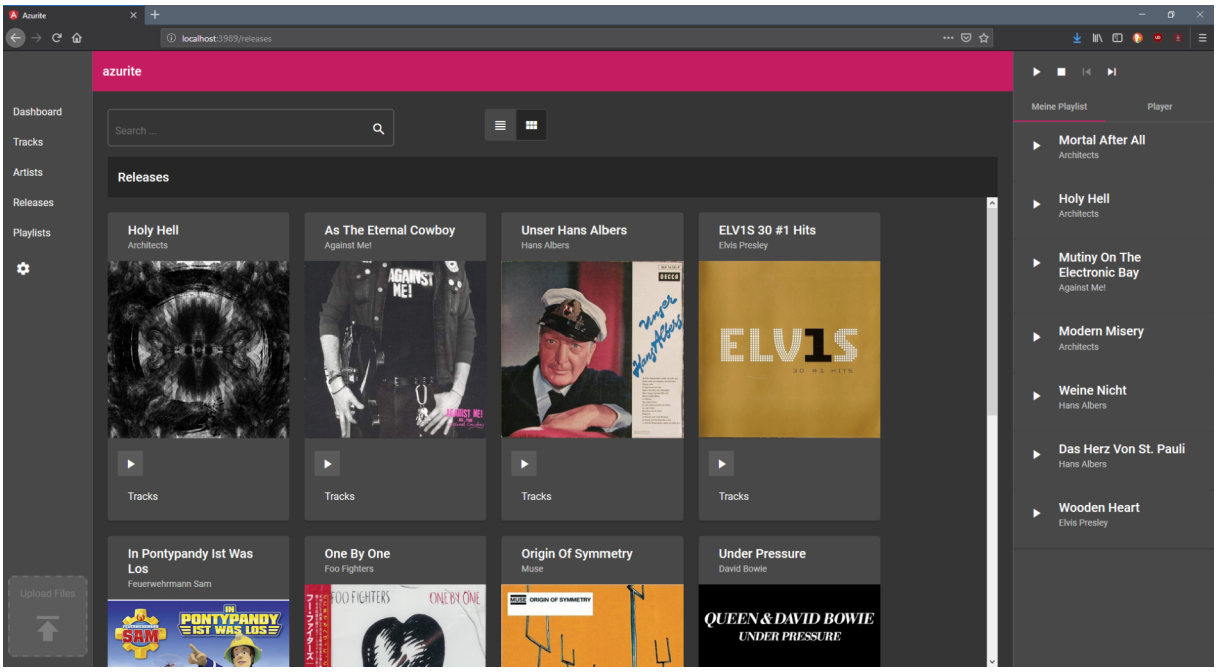

Abbildung 12: Kartenansicht: Releases

Die zweite Darstellungsansicht ist die sogenannte Karten-Ansicht, in der die Alben nebeneinander als *Release-Card-Component* angezeigt werden Diese Komponenten beinhalten Titel und Künstler im Kopfteil und darunter das *Cover-Artwork*, sofern vorhanden. Im unteren Teil befinden sich zwei Schaltflächen. Eine zum Abspielen des Albums, die untere mit dem Titel „*Tracks*“ lädt die Titel aus der Datenbank und zeigt sie in einem neuen Fenster

rechts der Karte an. Ein Klick auf einen Titel öffnet dann wiederum die im vorherigen Abschnitt erwähnten Musiktitel-Aktionen.

5.2.1.7 *Artists*-Ansicht

Diese Ansicht enthält alle Künstler und liefert dazugehörige Daten wie Titelanzahl und letztes Änderungsdatum. Ein Klick auf einen Datensatz öffnet eine Detailseite.

5.2.1.8 *Playlists*-Ansicht

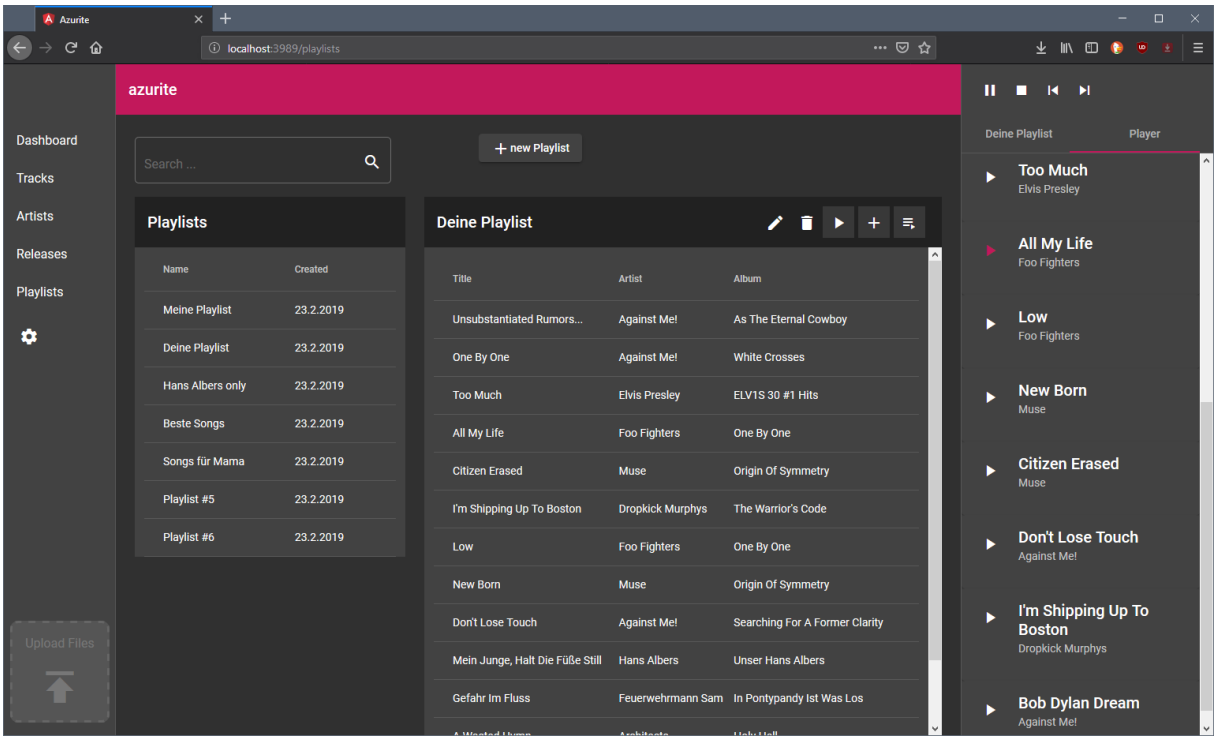

Abbildung 13: Playlist Ansicht

Die letzte der vier Sammlungsansichten beinhaltet zwei Spalten. Links sieht der Nutzer eine Liste aller Wiedergabelisten. Wählt er eine aus, so wird diese geladen und in der rechten Spalte mit allen Titeln detailliert angezeigt. In der Kopfzeile werden zusätzliche Aktionen angezeigt. Hier können unter Anderem der Name der Liste geändert oder die Liste gelöscht werden. Weitere Aktionen sind Wiedergabe der Liste und das Setzen als aktive Wiedergabeliste, was bedeutet, dass sie im *Playlist-Editor* angezeigt wird, um ihr dort aus anderen Ansichten Titel hinzufügen zu können.

5.2.1.9 *Detail*-Ansichten

Die Detailansichten sind bisher noch nicht sehr ausgereift umgesetzt. Öffnet man die Ansicht für ein Album wird dieses in ausgeklappter Karten-Ansicht dargestellt. Die Detailseite eines Künstlers liefert bisher alle Alben des Künstlers, ebenfalls als Karten-Ansicht. Die Karten-Ansicht ist im Absatz Releases-Ansicht beschrieben. Musiktitel haben bisher keine Detailansicht, sind allerdings als *Track*-Komponente im Player und *Playlist-Editor* zu sehen.

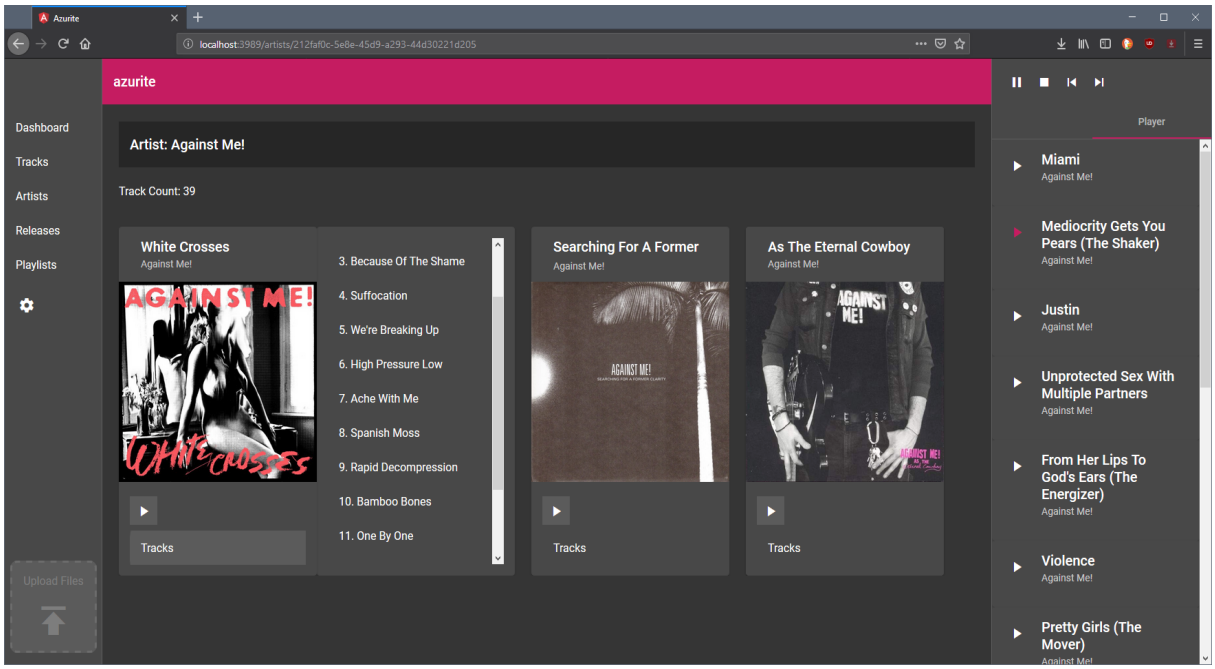

Abbildung 14: Detailansicht Künstler

5.2.1.10 Mobile Ansichten

Auf kleinen Bildschirmen werden Menü und Player-Komponente ausgeblendet und lassen sich über Schaltflächen einbinden.

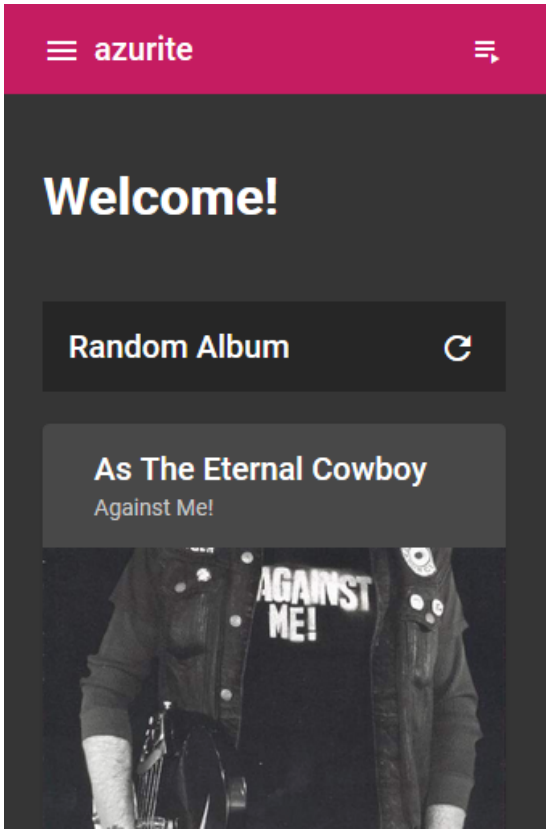

Abbildung 16: Dashboard in Mobilansicht

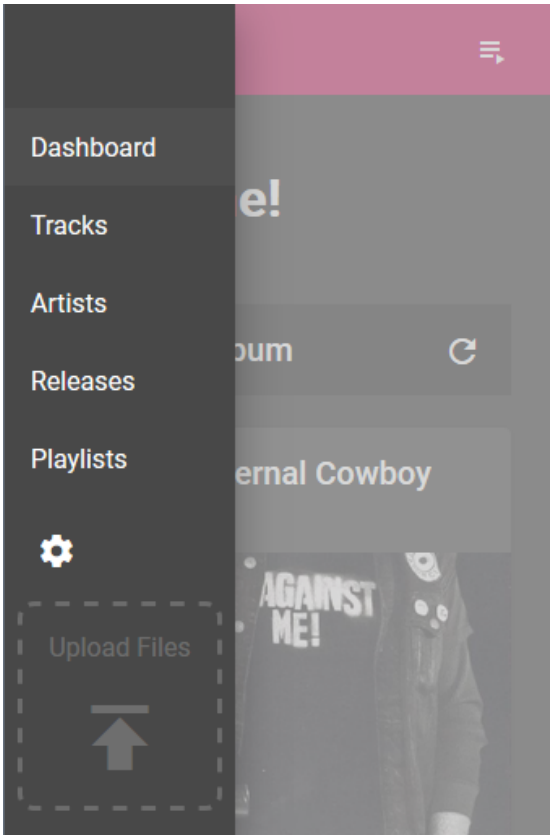

Abbildung 15: Menü in Mobilansicht

5.2.2 Models

Die *Model*-Klassen sind den konzeptionellen Modellen aus Kapitel 3.1 nachempfunden. Sie bilden im Front-End das Äquivalent zu den Entitäten der Datenbank und dienen der Überführung von Ressourcen aus dem Back-End in objektorientierte *TypeScript*-Klassen. In der Regel existieren ausschließlich Felder für Werte, die in den Ressourcen vorhanden sind oder sein können. Die Klassen enthalten die statischen Methoden *deserialize* und *deserializeArray*, welche für die Konstruktion neuer, typisierter Instanzen aus untypisierten Ressourcen-Objekten konzipiert sind. In ihnen werden alle Felder der Ressource in entsprechende Felder der Klasse überführt.

Eine genaue Übersicht über die vorhandenen *Model*-Klassen befindet sich in der Dokumentation im Anhang.

5.2.3 Services

Die Services des Front-Ends sind vor Allem für die API-Anfragen über HTTP verantwortlich. Pro Entitätenmodell der Datenbank gibt es einen zuständigen Service, der die jeweiligen Endpunkte am Back-End abfragt.

Alle Services verfügen in der Regel über Methoden zum Abfragen einer oder mehrerer Entitäten, sowie zum ändern von Entitäten. Weitere Methoden existieren zum Beispiel für das nachladen verlinkter Entitäten oder das Erzeugen neuer Entitäten. Eine schematische und beispielhafte Auflistung der speziellen Methoden befindet sich im Anhang, im Abschnitt „Dokumentation“. Im folgenden Abschnitt werden die typischen Methoden beispielhaft beschrieben.

Typische Methoden: Finden sich in den meisten Services wieder

find*Entity*(id)	Findet eine Entität mittels ID
Endpunkt:	*GET …/entity/*id
Beispiel:	*trackService.findTrack(123) => GET: .../tracks/123*

Find*All* **...**(*HttpParameter*)	Findet alle Datensätze einer Entität. Übermittelt zusätzlich einen einen *query*-String sowie weitere HTTP-Parameter mit Seiten und Sortieranforderungen.
Endpunkt:	*GET …/entities*
Beispiel:	*artistService.findAll => GET: .../artists?...httpParameter...*
findAll*Entities*(id) **find*Entities*(id)**	Findet alle Datensätze einer Entität in Relation zu einer anderen
Endpunkt:	*GET …this_entities/*id*/entities*
Beispiel:	*releaseService.findTracks*(123) => GET: …/releases/123/tracks
findRandom(int)	Findet eine gebene Anzahl zufälliger Entitäten
Endpunkt:	*GET …/entities/random/*int
Beispiel:	*trackService.findRandom(5) => GET: …/releases/random/5*

Subjects und Observables

Neben den Methoden halten der Player- sowie der Playlist-Service noch Quellen vom Typ *Subject* für veränderliche Daten, die von Komponenten als sogenannte *Observables* abonniert werden können.
Der Nutzen dahinter ist, dass so die Komponenten auf Änderungen der Quelldaten reagieren und ihre eigenen Werte daran anpassen, anstatt diese bei Initialisierung auf feste Werte zu setzen.

Folgende Quelldaten existieren:

Service	***Subject***	**Beschreibung**
Player Service	playerDataSource	Player-Daten wie Wiedergabe-Index, Lautstärke, Laufzeit, oder Pause-Modus
Player Service	trackDataSource	Die Musiktitel in der aktiven Wiedergabeliste
Playlist Service	editorTrackDataSource	Die Musiktitel im Playlist-Editor
Playlist Service	editorTitleSource	Titel des Playlist-Editors (Name der aktiven Wiedergabeliste)

5.2.4 Die Upload Direktive

Zum ermöglichen des Drag-And-Drop Uploads wurde eine Direktive erstellt, welche in Komponenten einbezogen wird, um ihnen diese Funktionalität mitzugeben. Sie ändert die Gestaltung der Komponente, wenn Dateien herübergezogen werden und leidet losgelassene Dateien über die Komponente an den Service weiter. Die Vorlage für den Code liefert ein Tutorial von Luis Moncaris von der Internetseite Scotch.io (Moncaris, 2017)
Durch das Nutzen von Direktiven wird die jeweilige Funktionalität wiederverwendbar und in theoretisch jeder dargestellten Komponente einsetzbar.

6. Fazit

Dieses Projekt war vor Allem sehr lehrreich und wesentlich umfangreicher als zunächst angenommen. Das Arbeiten an zwei verschiedenen Baustellen – Front-End und Back-End – war mit den beiden verschiedenen Programmiersprachen und Frameworks nicht immer leicht. Das sichtbare Ergebnis wirkt zunächst etwas schwach und bietet nicht alle Funktionalitäten des Entwurfs. Allerdings wurde eine große, funktionierende Basis erschaffen, die dazu einlädt, an dem Projekt weiterzuarbeiten. Hieraus lässt sich ableiten, dass es vielleicht besser gewesen wäre, kleinteilige Konzepte zu verfassen und sofort umzusetzen, anstatt ein breites Konzept anzulegen, von dem nur ein Teil realisiert wird. Diese Umstände wurden im späteren Verlauf der Umsetzung – nämlich bei der Programmierung des Front-Ends – bereits sichtbar. Zu diesem Zeitpunkt wurde daher der Fokus darauf gesetzt, möglichst viele verschiedene Funktionalitäten nur prototypisch einzubauen. Außerdem wurde viel Wert darauf gelegt, dass diese fehlerfrei laufen.
Folgende Funktionalitäten sind nun beispielsweise im Projekt enthalten:

- Listen mit Such- und Sortierfunktionen
- Stetiges, nahtloses nachladen von Daten
- *Drag and Drop* von Komponenten in einer Liste
- *Drag and Drop* Datei-Upload
- Bereitstellen und Beobachten veränderlicher Daten („*Observables*“)
- Extraktion von Metadaten aus Mp3-Dateien inklusive *Cover-Artwork*
- Web Sockets zum Informieren des Front-Ends über das Back-End
- Informationen werden im Browser gespeichert („Local Storage“)
- Sinnvolle Trennung von Ressourcen
- Verwalten zeitkritischer Datenbankabfragen mittels *Executor-Service* und *Coroutines*

In Zukunft kann das durch diese Funktionen angelernte Wissen dafür genutzt werden, deutlich schneller neue Funktionen zu implementieren.

Somit kann diese Arbeit auch hilfreich für andere Entwickler sein, die ein vergleichbares *Full-Stack*-Projekt realisieren möchten.

Ausblick

Die Weiterentwicklung des Projekts könnte zunächst bedeuten, die nicht umgesetzten Funktionalitäten umzusetzen. Besonders die Idee, Entitäten auf mehrere Weisen zu verknüpfen, um komplexere Relationen zu erzeugen, als bloß beispielsweise Verbindungen von Künstlern zu ihren Musikstücken, wurde noch nicht umgesetzt. Beispiele hierfür sind:

- Versionen von Titeln wie *Live-Version, Cover* oder *Remix* ihren originalen zuordnen
- Künstler über Genres oder Plattenlabel anderen Künstlern zuordnen
- Schlagwörter und Attribute für Musiktitel einführen, um sie vergleichbar zu machen

Zweck dieser Relationen wird es dann sein, intelligente Empfehlungen der Musikauswahl für den Nutzer zu erzeugen, wie im es im Kapitel Anforderungsanalyse erfasst ist. Weitere Ideen sind das Implementieren von Streaming oder Dateidownloads am Front-End. Die Front-End-Komponenten können generell noch reichlich erweitert werden, an erster Stelle steht hier das Überarbeiten des *Dashboard*s und Players, sowie die Gestaltung der Detailseiten. Auch mehr Einstellungsmöglichkeiten, beispielsweise ein helles Farbschema, sind vorgesehen.

Weiterhin interessant ist zudem die Anbindung von externen Metadatenquellen, wie beispielsweise durch die API von *discogs.com* (Discogs, 2019). Dadurch könnte das Aufbereiten der bestehenden Metadaten in den Musikdateien überflüssig werden.

7. Quellenverzeichnis

Sämtliche Quellen wurden zuletzt am 24.02.2019 aufgerufen

1. Rahn (2019) *https://www.frank-rahn.de/einfuehrung-spring-framework/*
2. Spring (2019) *https://spring.io/guides2 Spring,io*
3. Kotlin (2019) *https://kotlinlang.org*
4. Oracle, 2019) *https://www.oracle.com/technetwork/java/javaee/tech/persistence-jsp-140049.html*
5. JThink (2019) *http://www.jthink.net/jaudiotagger*
6. TypeScript (2019) *https://www.typescriptlang.org/docs/handbook/typescript-in-5-minutes.html*
7. Angular.de (2019) *https://angular.de/artikel/was-ist-angular*
8. AngularMaterial (2019) *https://material.angular.io*
9. Vikdor (2012) Nutzer Vikdor auf stackoverflow.com *https://stackoverflow.com/questions/12899372/spring-why-do-we-autowire-the-interface-and-not-the-implemented-class*
10. SpringDocs (2019) *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/http/ResponseEntity.html*
11. Discogs.com (2019) *https://www.discogs.com/developers*
12. Angular.io (2019) *https://angular.io/guide/lifecycle-hooks*
13. Material.io (2019) *https://material.io/tools/icons*
14. Moncaris (2017) Luis Moncrais auf *https://scotch.io/@minrock/how-to-create-a-drag-and-drop-file-directive-in-angular2-with-angular-cli-part-1*

8. Abbildungsverzeichnis

Abbildungsverzeichnis

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Bachelor-Thesis mit dem Titel:

Entwicklung einer anwenderorientierten Webanwendung zur intelligenten Verwaltung von Musiksammlungen

selbständig und nur mit den angegebenen Hilfsmitteln verfasst habe. Alle Passagen, die ich wörtlich aus der Literatur oder aus anderen Quellen wie z. B. Internetseiten übernommen habe, habe ich deutlich als Zitat mit Angabe der Quelle kenntlich gemacht.

Lübeck, den 25.02.2019 Christopher von Bargen

Anhang

Anhangsverzeichnis

A1 Eingesetzte Tools, Software und Assets

Oracle Java Runtime Environment *und* ***Development Kit****: Laufzeitumgebung und Compiler*

https://www.oracle.com/java

JetBrains intelliJ IDEA Ultimate *als Entwicklungsumgebung*

https://www.jetbrains.com/idea/

Oracle MySQL Server, Connector/J und Workbench*: Datenbank-Server, Java-Connector und Datenbank-Inspektion*

https://www.mysql.com/

Apache Maven *(Dependency und Build Management Back-End)*

https://maven.apache.org/

Node.js *als Laufzeitumgebung (Front End)*

https://nodejs.org/en/

Angular CLI *als Compiler und zum Generieren von Angular Komponenten*

https://angular.io/

Angular Material Components und Component DevKit *im Front-End*

https://material.angular.io/

Material Design Icons *im Front-End*

https://material.io/tools/icons/

Jthink Jaikoz *zum Vorbereiten der MP3-Metadaten*

http://www.jthink.net/jaikoz/

Postman zum Testen der API

https://www.getpostman.com/

git *als Versionierungsystem*

https://git-scm.com/

Atlassian BitBucket *Versionierung und Backup*

https://bitbucket.org/

A2 Installationshinweise und Quelltext

Quelltext:

Back-End: **`https://bitbucket.org/bargicorp/copper.git`**

Front-End: **`https://bitbucket.org/bargicorp/azurite.git`**

MySQL

Zunächst wird eine MySQL Datenbank mit Namen *db_copper* benötigt. Ein User mit Namen *Springuser* und Kennwort *ThePassword* muss angelegt werden. Dies kann auch über Tools wie *MySQL Workbench* geschehen. *MySQL Server* und *MySQL Connector/J* sind erforderlich

MySQL benötigt Python 2.7 *https://www.python.org/download/releases/2.7/*

Server und Connector via *Installer* *https://dev.mysql.com/downloads/installer/*

Standardeinstellungen übernehmen (*Port 3306*), *Root*-Kennwort festlegen

MySQL Command Line Client starten und mit *Root*-Kennwort anmelden

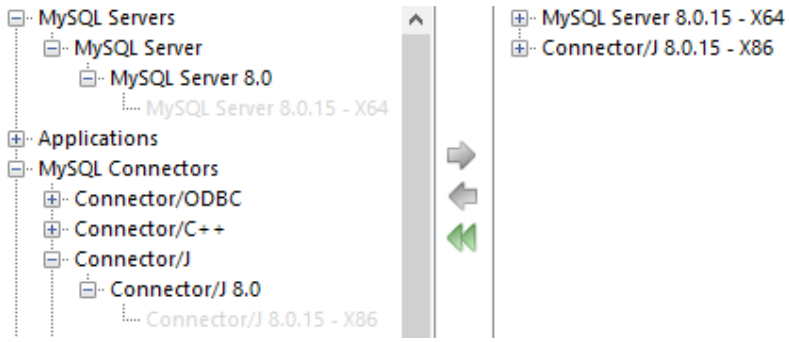

Abbildung 17: MySQL Installer Selection

Im MySQL Command Line Client:

Datenbank anlegen:
```
mysql> create database db_copper;
```

Nutzer anlegen:
```
mysql> create user 'springuser'@'%'
       identified by 'ThePassword';
```

Rechte gewähren:
```
mysql> grant all on db_copper.* to
       'springuser'@'%';
```

Spring

Für Spring wird *JavaDK 1.7* oder höher benötigt. Mit *Apache Maven* wird das Projekt dann direkt aus dem Quelltext kompiliert und gestartet

Jdk 1.7+	*https://www.oracle.com/technetwork/java/javase/downloads*
Maven	*binaries* von: *https://maven.apache.org/download.cgi*

Umgebungsvariablen für Java und Maven einrichten (Windows 10)

Umgebungsvariablen → Nutzervariablen → Neu

Name: JAVA_HOME

Wert: Pfad des JDK, z.B.: C:\Program Files\Java\jdk1.8.0_181

Maven:

Umgebungsvariablen → Nutzervariablen → Variable: Path *→Bearbeiten*

Neu →Wert: Pfad von Maven, z.B.: *C:\Program Files\apache-maven-3.6.0\bin*

Weitere Informationen: *https://maven.apache.org/install.html*

Projektordner mit Kommandozeile öffnen:

Projekt mit *Maven* kompilieren und starten ...\copper> `mvn spring-boot:run`

Die API ist jetzt unter localhost:8080 erreichbar. Fenster nicht schließen!

Hinweis: Soll bei wiederholtem Starten die Datenbank nicht neu erstellt und damit geleert werden, so muss in der *application.properties* im Verzeichnis ...*copper**src**main**resources*\ der Wert `spring.jpa.hibernate.ddl-auto` vor dem starten von `create-drop` auf `update` gesetzt werden.

Angular

Node.js installieren: *https://nodejs.org/*

Kommandozeile im Projekt-Source-Folder öffnen:

Angular CLI installieren:	`..\azurite\src> npm i -g @angular/cli`
Hot-Server installieren:	`..\azurite\src> npm i -g hot-server`
npm-Module laden:	`..\azurite\src> npm update`
Build:	`..\azurite\src> ng build`
Hot-Server im Dist-Folder starten	`..\azurite\dist\azurite> hot-server`

Der Server ist jetzt unter *localhost:3989/* über einen Browser erreichbar, Fenster nicht schließen!

A3 Dokumentation

A3.1 Back-End

A3.1.1 Datenklassen

Dieser Abschnitt liefert eine detaillierte Übersicht über die Hauptklassen. Daraus werden außerdem die Tabellen der Datenbank generiert.

Track.kt

Ein Musikstück

Feld	Typ	Beschreibung
`id`	`String`	Primärschlüssel
`title`	`String`	Titel des Musikstücks
`duration`	`Duration`	Dauer des Musikstücks
`version`	`Enum`	Versionstyp wie Studioaufname, Liveaufname etc. (bisher ungenutzt)
`trackNo`	`Int`	Titelnummer auf dem assoziieren Album
`files`	`List <FileEntity>`	Zugeordnete Dateientitäten
`artists`	`List<Artist>`	Assoziierte Künstler
`releases`	`Release`	Assoziierte Veröffentlichung
`players`	`Player`	Assoziierte Player
`playCount`	`Int`	Anzahl, wie oft ein Musikstück gespielt wurde
`lastPlayed`	`Date`	Zeitstempel, wann ein Track zuletzt gespielt wurde

Artist.kt

Ein Künstler

Feld	Typ	Beschreibung
`id`	`String`	Primärschlüssel
`name`	`String`	Name des Künstlers
`tracks`	`Track`	Assoziierte Musikstücke
`trackCount`	`Long`	Anzahl Tracks

Release.kt

Eine Veröffentlichung von Musikstücken

Feld	Typ	Beschreibung
`id`	`String`	Primärschlüssel
`name`	`String`	Name der Veröffentlichung
`pictureUri`	`String`	Externe Adresse für Cover Art
`tracks`	`List<Track>`	Assoziierte Musikstücke
`year`	`String`	Veröffentlichungsjahr
`label`	`String`	Plattenlabel
`coverArt`	`String`	Base64 kodiertes Albumcover

FileEntity.kt

Datensatz einer Musikdatei

Feld	Typ	Beschreibung
`id`	`String`	Primärschlüssel
`extension`	`Enum`	Dateiendung
`path`	`Path`	Dateipfad
`directory`	`Path`	Dateiverzeichnis
`duration`	`Duration`	Dauer der Musikdatei
`id3header`	`ID3Header`	Hält die Metadaten der Musikdatei
`track`	`Track`	Assoziiertes Musikstück

`filename`	`String`	Ursprünglicher Dateiname, genutzt für Initialisierung von Pfad und Dateiendung, danach verworfen (@Transient)

Methoden	**(Parameter): Rückgabe**	**Beschreibung**
`SanitizeAndSplitTag`	`(String): List<String>`	Trennt den Artist Eintrag der Metadaten in ggf. mehrere Künstler auf und entfernt Trennzeichen.
`extractHeader`		Extrahiert hinterlegte Metadaten einer Audiodatei und speichert sie im Feld id3-header
`generatePath`		Generiert den gesamten Zieldateipfad aus Metadaten, je nach dem, welche Metadaten vorhanden sind

innere Klassen	**Felder**	**Beschreibung**
`ID3-Header`	`title, artist, album, trackNo, year, coverArt, label`	Klasse für Metadaten, wie sie in der Datenbank gespeichert werden sollen. Datentypen sind String, bzw. List<String> bei artist. Das Erhalten der originalen Metadaten ist für die Wiederherstellung der Originaldatei vorgesehen, allerdings noch nicht implementiert
`PathConverter`		Enthält die Methoden *convertToDatabaseColumn* sowie *convertToEntityAttribute*, welche Pfade in String und umgekehrt konvertieren.

Player.kt

Datensatz für die Aktuelle Wiedergabe. In der aktuellen Version gibt es genau einen Player

Feld	**Typ**	**Beschreibung**
`id`	`String`	Primärschlüssel
`volume`	`Double`	Lautstärke der aktuellen Wiedergabe
`tracks`	`List<Track>`	Alle Musikstücke, die für die aktuelle Wiedergabe vorgesehen sind
`index`	`Int`	Wiedergabeindex innerhalb der Musikliste
`time`	`Int`	Zeitfortschritt des gegenwärtig spielenden Musikstücks
`paused`	`Boolean`	Gibt an, ob die Wiedergabe pausiert ist

Playlist.kt

Eine Liste von Musikstücken

Feld	Typ	Beschreibung
`id`	`String`	Primärschlüssel
`name`	`String`	Titel der Wiedergabeliste
`tracks`	`List<Track>`	Assoziierte Musikstücke (geordnet)
`creationIndex`	`Long`	Generierter Index für den Standardnamen. Eine neue Playlist erhält beispielsweise den Namen „Playlist#1“, die nächste den Namen „Playlist#2“ usw.

A3.1.2 Repositories

Durch Vererbung erhalten alle Repository-Interfaces folgende Methoden:

Methode	(Parameter): Rückgabe	Beschreibung
`findAll<T>`	`(Pageable): Page<T>`	Gibt eine Seite aller Datensätze zurück
`findByID<T>`	`(String): T`	Gibt genau eine Entität zurück

Im Folgenden sind Methoden die natives SQL enthalten gekennzeichnet.

TrackRepository.kt

Verwaltet Datenbankzugriffe auf *Track* Entitäten

Methode	(Parameter): Rückgabe	Beschreibung
`findAllByTitleContains`	`(Pageable, String): Page<Track>`	Findet Musiktitel nach Namen
`findAllByTitleContains OrAtists_NameContainsO rRelease_NameContains`	`(String, List<Artist>, Release): Track`	Findet Musiktitel nach Namen, Künstlernamen oder Albumnamen
`findByTitleAndArtistAn dRelease`	`(Pageable, String, String, String): Page<Track>`	Findet nach Namen, Künstler und Album
`findByReleaseOrderByTr ackNo`	`(Release): List<Track>`	Findet alle Titel eines Albums und sortiert nach Titelnummer
`findAllByTitleContains AndArtists`	`(String, Artist, Pageable): Page<Track>`	Findet alle Titel eines Künstlers mit enthaltender Zeichenfolge
`findAllByArtists`	`(Artist, Pageable): Page<Track>`	Findet alle Titel eines Künstlers

`countAllByArtists`	`(artist: Artist): Long`	Zählt die Anzahl Titel, die zu einem Künstler gehören
`findRandom`	`(number: Long): List<Track>`	Findet eine gegebene Anzahl zufälliger Titel

* natives SQL: `"SELECT * FROM db_copper.tracks ORDER BY RAND() LIMIT ? ;"`

Für ? wird der Parameter eingesetzt

ArtistRepository.kt

Verwaltet Datenbankzugriffe auf *Artist* Entitäten

Methode	(Parameter): Rückgabe	Beschreibung
`findByName`	`(String): Artist`	Findet Künstler nach Name
`findAllByNameContains`	`(Pageable, String): Page<Artist>`	Findet alle Künstler, deren Namen eine bestimmte Zeichenfolge enthält
`findRandom*`	`(Long): List<Artist>`	Findet eine gegebene Anzahl zufälliger Künstler

* natives SQL: `SELECT * FROM db_copper.artists ORDER BY RAND() LIMIT ? ;`

Für ? wird der Parameter eingesetzt

ReleaseRepository.kt

Verwaltet Datenbankzugriffe auf *Release* Entitäten

Methode	(Parameter): Rückgabe	Beschreibung
`findByName`	`(String): Release`	Findet eine Veröffentlichung nach Namen
`findAllByNameContains`	`(Pageable, String): Page<Release>`	Findet alle Veröffentlichungen, deren Namen eine bestimmte Zeichenfolge enthält
`FindRandom *`	`(Long): List<Release>`	Findet eine gegebene Anzahl zufälliger Künstler
`FindByArtistId **`	`(String): List<Release>`	Findet alle Alben eines Künstlers

* natives SQL:

```
SELECT * FROM db_copper.artists ORDER BY RAND() LIMIT ? ;
```

Für ? wird der Parameter eingesetzt

** natives SQL:

```
SELECT * FROM db_copper.releases INNER JOIN (SELECT
distinct tracks.release_id AS releaseId
FROM db_copper.tracks INNER JOIN db_copper.tracks_artists
ON tracks.track_id = tracks_artists.tracks_track_id
AND tracks_artists.artists_artist_id = ?1) releaseIds
WHERE id = releaseId ORDER BY releases.year DESC;
```

Für ? wird der Parameter eingesetzt

FileRepository.kt

Verwaltet Datenbankzugriffe auf *File* Entitäten

Methode	(Parameter): Rückgabe	Beschreibung
`findAllByFilename`	`(Pageable, String):`	Findet alle Datei-Entitäten, deren Name eine bestimmte Zeichenfolge enthält

PlayerRepository.kt

Verwaltet Datenbankzugriffe auf *Player* Entitäten

Methode	(Parameter): Rückgabe	Beschreibung
`findById`	`(id: Int): Player`	Findet einen Player mit gegebener ID. Das überschreiben der Superklassenmethode ist notwendig, da die ID hier nicht vom Typ String sondern Int ist

PlaylistRepository.kt

Verwaltet Datenbankzugriffe auf *Playlist* Entitäten

Methode	(Parameter): Rückgabe	Beschreibung
`findAllByNameContains`	`(Pageable, String): Page<Playlist>`	Findet alle Wiedergabelisten, deren Name eine bestimmte Zeichenfolge enthält

A3.1.3 Services

In den folgenden Dokumentationstabellen ist oft die rede von „leitet ... weiter“. Damit ist gemeint, dass eine Anfrage von den Controllern unverändert an die Repositories übermittelt wird und der Rückgabewert ebenso unverändert an die Controller zurückgegeben wird. Ist die Rede davon, dass Entitäten Ermittelt werden ist ebenfalls davon auszugehen, dass dafür Repositories hinzugezogen werden.

Injizierte Objekte werden nicht dokumentiert

TrackService.kt, TrackServiceImpl.kt

Verarbeitet musiktitelspezifische Anfragen

Methode	(Parameter): Rückgabe	Beschreibung
`findAll`	`(Pageable): Page<Track>`	Leitet eine Abfrage aller Musiktitel-Entitäten weiter
`findAll`	`(Pageable, String): Page<Track>`	Leitet eine Abfrage aller Musiktitel-Entitäten weiter, die eine bestimmte Zeichenkette im Namen haben
`findById`	`(String): Track`	Leitet eine Abfrage einer bestimmten Musiktitel-Entität weiter
`findByArtist`	`(String, Pageable): Page<Track>`	Leitet eine Abfrage aller Musiktitel-Entitäten weiter, die einem Künstler zugeordnet sind
`findByArtist`	`(String, Pageable, String): Page<Track>`	Leitet eine Abfrage aller Künstler-Entitäten weiter, die eine bestimmte Zeichenkette im Namen haben und einem Künstler zugeordnet sind
`findByReleaseId`	`(String): List<Track>`	Liefert alle Musiktitel eines Albums
`create`	`(TrackDto): Track`	Instanziiert ein *Title*-Objekt und übermittelt es zum Speichern an das Repository
`createSave`	`(Track): Track`	Prüft ob ein Musiktitel mit Titel, Künstler und Album noch nicht existiert und speichert ggf.
`update`	`(Track): Track`	Speichert einen Musiktitel
`buildTrackList`	`(List<String>): List<Track>`	Erzeugt eine Liste von Titeln aus Titel-IDs
`FindByTitleAndArtists ...AndRelease`	`(Track): Track`	Findet Musiktitel anhand von Titel, Künstler und Album
`findRandom`	`(Pageable, Long): Page<Artist>`	Leitet eine Abfrage einer Anzahl zufälliger Musiktitels weiter

ArtistService.kt, ArtistServiceImpl.kt

Verarbeitet künstlerspezifische Anfragen

Methode	(Parameter): Rückgabe	Beschreibung
`findByNameOrCreate`	`(String): Artist`	Ermittelt einen Künstler nach Namen, ist dieser nicht vorhanden, wird eine neue Entität erstellt
`create`	`(String): Artist`	Instanziiert ein *Artist*-Objekt und übermittelt es zum Speichern an das Repository
`findAll`	`(Pageable): Page<Artist>`	Leitet eine Abfrage aller Künstler-Entitäten weiter
`findAll`	`(Pageable, String): Page<Artist>`	Leitet eine Abfrage aller Künstler-Entitäten weiter, die eine bestimmte Zeichenkette im Namen haben
`findById`	`(String): Artist`	Leitet eine Abfrage einer bestimmten Künstler-Entität weiter
`getArtistsByReleaseId`	`(String): List<Artist>`	Ermittelt den oder die Künstler der Musiktitel eines Albums, sortiert nach Häufigkeit
`updateTrackCount`	`(Artist)`	Ermittelt die Anzahl aller Musiktitel eines Künstlers und speichert diese beim Künstler. Wartet 3s auf weitere Updates bevor ausgeführt wird *
`findRandom`	`(Pageable, Long): Page<Artist>`	Leitet eine Abfrage einer Anzahl zufälliger Künstler weiter

*Anmerkung: Diese Methode wird nicht mehr benutzt, da dass Abspeichern der Titelanzahl im Künstler redundant ist. Aufgrund ihrer Funktionalität wird sie erhalten

ReleaseService.kt, ReleaseServiceImpl.kt

Verarbeitet albumspezifische Anfragen

Methode	(Parameter): Rückgabe	Beschreibung
`findAll`	`(Pageable): Page<Release>`	Leitet eine Abfrage aller Album-Entitäten weiter
`findAll`	`(Pageable, String): Page<Release>`	Leitet eine Abfrage aller Album-Entitäten weiter, die eine bestimmte Zeichenkette im Titel haben

`findById`	`(String): Release`	Leitet eine Abfrage einer bestimmten Album-Entität weiter
`create`	`(ReleaseDto): Release`	Instanziiert ein *Release*-Objekt und übermittelt es zum Speichern an das Repository
`create`	`(String): Release`	Instanziiert ein *Release*-Objekt mit Namen und übermittelt es zum Speichern an das Repository
`findByName`	`(String): Release`	Ermittelt einen Künstler nach Namen
`findByNameOrCreate`	`(String): Release`	Ermittelt einen Künstler nach Namen, ist dieser nicht vorhanden, wird eine neue Entität erstellt
`findByArtistId`	`(Pageable, String): Page<Release>`	Leitet Abfrage an alle Alben eines Künstlers weiter
`findRandom`	`(Pageable, Long): Page<Release>`	Leitet eine Abfrage einer Anzahl zufälliger Künstler weiter

FileService.kt, FileServiceImpl.kt

Verarbeitet dateientitätenspezifische Anfragen

Methode	**(Parameter): Rückgabe**	**Beschreibung**
`create`	`(String): File`	Speichert eine Datei-Entität im Repository
`findAll`	`(Pageable): Page<File>`	Leitet eine Abfrage aller Datei-Entitäten weiter
`findAll`	`(Pageable, String): Page<File>`	Leitet eine Abfrage aller Datei-Entitäten weiter, die eine bestimmte Zeichenkette im Dateinamen haben
`handleUpload`	`(MultipartFile): FileEntity`	Speichert eine Datei im Dateisystem, erzeugt ggf. Entitäten (Künstler, Album, Musiktitel) aus Metadaten, wenn diese nicht schon existieren.

PlayerService.kt, PlayerServiceImpl.kt

Verarbeitet playerspezifische Anfragen

Methode	(Parameter): Rückgabe	Beschreibung
`create`	`(PlayerDto): Player`	Speichert eine Player-Entität im Repository
`findById`	`(Int): Player`	Leitet eine Abfrage einer bestimmten Player-Entität weiter
`update`	`(PlayerDto): Player`	Ändert Daten eines Players und passt sie an die MediaPlayer-Klasse an.
`next`		Übermittelt den nächsten Titel der Wiedergabe an die MediaPlayer-Klasse
`play`	`(Track)`	Übermittelt einen Titel an den MediaPlayer und erhöht den playCount des Titels
`addTrack`	`(String, Int): Track`	Ermittelt einen Titel und fügt ihn der aktuellen Wiedergabe hinzu
`moveTrack`	`(Int, Int): Track`	Bewegt einen Titel innerhalb der Reihenfolge der aktuellen Wiedergabe

PlaylistService.kt, PlaylistServiceImpl.kt

Verarbeitet wiedergabelistenspezifische Anfragen

Methode	(Parameter): Rückgabe	Beschreibung
`findAll`	`(Pageable): Page<Playlist>`	Leitet eine Abfrage aller Playlist-Entitäten weiter
`findAll`	`(Pageable, String): Page<Playlist>`	Leitet eine Abfrage aller Playlist-Entitäten weiter, die eine bestimmte Zeichenkette im Dateinamen haben
`findById`	`(String): Playlist`	Leitet eine Abfrage einer bestimmten Playlist-Entität weiter
`create`	`(name): Playlist`	Speichert eine Playlist-Entität im Repository
`addTrack`	`(String, String, Int): Track`	Ermittelt Playlist und Titel und fügt Titel zur Playlist hinzu
`update`	`(String, PlaylistDto): Playlist`	Ändert eine Playlist Entität und speichert sie
`moveTrack`	`(String, Int, Int): Track`	Ermittelt eine Playlist und bewegt einen Titel innerhalb der Reihenfolge der Titelliste
`delete`	`(String)`	Löscht eine Playlist

A3.1.4 Ressourcen

In diesem Absatz werden die Felder der einzelnen Ressourcen aufgelistet. Da sie sich an den Datenklassen orientieren, sind Typen und Beschreibungen zu ihnen identisch. Letztere werden daher weggelassen

TrackRessource.kt	**ArtistResourcen.kt**	**ReleaseResourcen.kt**

`id: String`
`title: String`
`duration: Long`
`playCount: Int`
	`id: String`
	`name: String`
	`trackCount: Int`
		`id: String`
		`name: String`
		`pictureUri: String`
		`year: String`
		`label: String`

FileResourcen.kt	**PlayerResourcen.kt**	**PlaylistResourcen.kt**

`id: String`
`filename: String`
`path: String,`
`duration: Long`
`iD3Header: ID3Header`
	`id: Int`
	`volume: Double`
	`index: Int`
	`time: Int`
	`paused: Boolean`
		`id: String`
		`name: String`

A3.1.5 Resource-Assembler

Die Resource-Assembler haben jeweils nur die Methode toResource:

Methode	**(Parameter): Rückgabe**	**Beschreibung**
`toResource`	`(<`*`entity`*`>): <`*`entity`*`>Resource`	Konstruiert eine Ressource aus einer Entität, bettet weitere Resourcen ein oder setzt links auf Resourcen

TrackRessourceAssembler.kt	**ArtistResourcenAssembler.kt**

`Embedded:`
`artists`
`release`
	`Links:`
	`tracks`
	`self`

ReleaseResourcenAssembler.kt	**FileResourcenAssembler.kt**

`Links:`
`tracks`
`self`
`coverArt`
	`Embedded:`
	`track`

PlayerResourcenAssembler.kt	**PlaylistResourcenAssembler.kt**

`Links:`
`tracks`
	`Links:`
	`tracks`
	`self`

A3.1.6 Controller und API

Die Controllermethoden bilden gleichzeitig die Endpunkte der API. Die Beschreibung orientiert sich daher an der Sicht von außen. Der Rückgabewert bezieht sich immer auf die zugehörige Entität. In der Dokumentation wird nur festgehalten, ob die Ressourcen einzeln, als Liste oder als Seite ausgegeben werden. Pfadvariablen sind mit {} gekennzeichnet. Sie werden von der Methode extrahiert.

TrackController.kt

	Endpunkt	Methode
GET	**/tracks** Param query: String	**findAll: Page** Findet alle Titel
POST	**/tracks** Body TrackDto	**create: Single** Erzeugt einen Titel
GET	**/tracks/{trackId}**	**findById: Single** Findet einen Titel
GET	**/artists/{artistId}/tracks**	**findByArtist: Page** Findet alle Titel eines Künstlers
GET	**/releases/{releaseId}/tracks**	**findByRelease: List** Findet alle Titel eines Albums
GET	**/player/tracks**	**findPlayerTracks: List** Findet alle Titel des Players
PATCH	**/player/tracks/{index}** Body TrackDto	**movePlayerTrack: Single** Bewegt einen Titel innerhalb der Player-Liste
POST	**/player/tracks** Body TrackDto	**addTrack: Single** Fügt dem Player einen Titel hinzu
GET	**/playlist/tracks**	**findPlaylistTracks: List** Findet alle Titel einer Wiedergabeliste
POST	**/playlist/{playlistId}/tracks** Body TrackDto	**addTrackToPlaylist: Single** Fügt einer Wiedergabeliste einen Titel hinzu

PATCH	**/playlist/{playlistId}/tracks/{index}** Body TrackDto	**movePlaylistTracks: Single** Bewegt einen Titel innerhalb einer Wiedergabeliste
GET	**/tracks/random/{number}**	**findRandom: Page** Findet eine Anzahl zufälliger Titel

ArtistController.kt

	Endpunkt	Methode
GET	**/artists** Param query: String	**findAll: Page** Findet alle Künstler
GET	**/artists/random/{number}**	**findRandom: Page** Findet eine Anzahl zufälliger Künstler
GET	**/artists/{artistId}**	**findById: Single** Findet einen Künstler

ReleaseController.kt

	Endpunkt	Methode
POST	**/releases** Body ReleaseDto	**create: Single** Erzeugt neues Album
GET	**/releases**	**findAll: Page** Findet alle Alben
GET	**/releases/{releaseID}**	**findById: Single** Findet ein Album
GET	**/artists/{artistId}/releases**	**findAllByArtistId: Page** Findet alle Alben eines Künstlers
GET	**/releases/{releaseID}/cover**	**findCoverArt: Single** (CoverArtResource) Findet das Cover-Artwork eines Albums
GET	***/releases/*random/{number}**	**findRandom: Page** Findet eine Anzahl zufälliger Alben

FilesController.kt

	Endpunkt	Methode, Beschreibung
GET	**/files**	**findAll: Page** Findet alle Datei-Entitäten
POST	**/files** Param MultipartFile	**uploadFile: Single** Erzeugt eine Titel

PlayerController.kt

	Endpunkt	Methode, Beschreibung
PATCH	**/player** Param query: String	**patch: Page** Findet alle Titel
GET	**/player** Body TrackDto	**create: Single** Erzeugt eine Titel
GET	**/artists/{artistId}/tracks**	**findByArtist: Page** Findet alle Titel eines Künstlers
WebS.	**/updates/player**	**push: Player** WebSocket: Player wurde aktualisiert; übermittelt Player-ID
WebS.	**/updates/player/tracks**	**PushTracks: Player** WebSocket: Player Titel wurden aktualisiert; übermittelt Player-ID

PlaylistController.kt

	Endpunkt	Methode, Beschreibung
POST	**/playlists** Body PlaylistDto	**create: Single** Erzeugt eine Wiedergabeliste
GET	**/playlists** Param query: String	**findAll: Page** Findet alle Wiedergabelisten
GET	**/playlists/{playlistId}**	**findById: Single** Findet eine Wiedergabeliste

PATCH	**/playlists/{playlistId}**	**update: Single** Ändert eine Wiedergabeliste
DELETE	**/playlists/{playlistId}**	**delete: Single** Löscht eine Wiedergabeliste
WebS.	**/updates/playlists**	**findPlayerTracks: List** WebSocket: Wiedergabeliste wurde aktualisiert; übermittelt Playlist-ID

A3.1.7 Datentransferobjekte

In diesem Absatz werden die Felder der einzelnen Datentransferobjekte aufgelistet. Da sie sich an den Datenklassen orientieren, sind Typen und Beschreibungen zu ihnen identisch. Letztere werden daher weggelassen.

TrackDto.kt	**ArtistDto.kt**	**ReleaseDto.kt**
id: String title: String duration: Duration filepath: String trackNo: Int artistIds: List<String> releaseId: String	id: String name: String	id: String name: String pictureUri: String year: String label: String

PlayerDto.kt	**PlaylistDto.kt**
id: Int volume: Double index: Int time: Int trackIds: List<String> paused: Boolean	id: String name: String trackIds: List<String>

A3.2 Front-End

A3.2.1 Models

Die Modell-Klassen orientieren sich an den Entitäten des Back-Ends. Beschreibungen der Felder sind identisch und dort einzusehen. Zusätzlich zu den Entitäten enthalten sie allerdings noch Felder für Links, die sich wiederum an den Ressourcen des Back-Ends orientieren

Alle Models haben außerdem die statischen Methoden serialize und serializeArray, mit denen ein Model-Objekt aus einem JSON-Objekt erzeugt wird.

track.model.ts	artist.model.ts	release.model.ts

`id: string`
`title: string`
`duration: number`
`playCount: number`
`artists: Artist[]`
`release: Release`
	`id: string`
	`name: string`
	`trackCount: number`
		`id: string`
		`title: string`
		`coverArt: string`
		`artists: Artist[]`
		`tracks: Track[]`

player.model.ts	**playlist.model.ts**	**resource-object.model.ts***

`id: string`
`volume: number`
`index: number`
`time: number`
`tracks: Track[]`
`paused: boolean`
	`id: string`
	`name: string`
	`tracks: Track[]`
		`createdDate: Date`
		`modifiedDate: Date`
		
		**Superklasse für andere Models*

A3.2.2 Komponenten

Implementierte Angular *Life-Cycle-Hooks* werden nicht dokumentiert (Angular.io, 2019)

artist.component.ts **Detailansicht Künstler**

Feld	Typ	Beschreibung
`loaded`	`Promise<boolean>`	Künstler ist geladen
`artist`	`Artist`	Künstler
`releases`	`Release[]`	Alben des Künstlers
`title`	`string`	Titelzeile

artist-page.component.ts **Listenansicht Künstler**

Feld	Typ	Beschreibung
`displayedColumn`	`string[]`	Angezeigte Tabellenspalten
`artists`	`Artist[]`	Künstler
`filter`	`string`	Suchwort
`pageData`	`PageData`	Seiteninformationen

Methoden	(Parameter): Rückgabe	Beschreibung
`loadArtistData`	`(HttpParams)`	Lädt alle Künstler vom Service
`applyFilter`	`(string)`	Übermittelt Suchfilter und aktualisiert Daten

collection-container.component.ts **Listenansicht**

Feld	Typ	Beschreibung
`viewport`	`CdkVirtualScroll ...Viewport`	Element für die eingebettete Listenansicht
`loadMore`	`EventEmitter<boolean>`	output: *Scroll*-Ende erreicht
`filter`	`EventEmitter<string>`	output: Suchwort
`itemSize`	`number`	Größe der Elemente im *Scroll-Viewport*

Methoden	(Parameter): Rückgabe	Beschreibung
`handleScroll`		Sendet LoadMore emitter wenn Scroll-Ende
`applyFilter`	`(string)`	Sendet Suchwort und aktualisiert Daten

dashboard.component.ts **Einstiegsseite**

Feld	Typ	Beschreibung
`tracks`	`Track[]`	Musiktitel
`artists`	`Artist[]`	Künstler
`releases`	`Release[]`	Alben
`tracksLoaded`	`Promise<boolean>`	Musiktitel geladen
`artistsLoaded`	`Promise<boolean>`	Künstler geladen
`releasesLoaded`	`Promise<boolean>`	Alben geladen

Methoden	(Parameter): Rückgabe	Beschreibung
`randomizeTracks`		Lädt neue, zufällige Titel
`randomizeArtists`		Lädt neue, zufällige Künstler
`randomizeReleases`		Lädt neue, zufällige Releases
`Play Track`	`(string)`	Spielt selektierten Track mit trackId

nav.component.ts **Navigationsleiste**

Feld	Typ	Beschreibung
`player`	`PlayerComponent`	Musiktitel
`apiAdress`	`FormControl`	Künstler

Methoden	(Parameter): Rückgabe	Beschreibung
`changeApiAdress`		Setzt API-Adresse aus FormControl im environment

player.component.ts **Player und PlaylistEditor**

Feld	Typ	Beschreibung
`playlists`	`Playlist[]`	Wiedergabelisten im Playlist-Editor
`player`	`Player`	Aktive Wiedergabe
`playlistEditorTracks`	`Track[]`	Titel im Playlist-Editor
`tracks`	`Track[]`	Titel in der aktiven Wiedergabe
`playlistEdtorTitle`	`string`	Titelzeile des Playlist-Editor-Tabs
`matTabGroup`	`MatTabGroup`	Auswahltabs Playlist-Editor | Player

Methoden	(Parameter): Rückgabe	Beschreibung
`onResize`		Passt Elemente an Bildgröße an
`move`	`(CdkDragDrop<string[]>, Playlist)`	Bewegt Element in Liste
`loadPlayerData`		Lädt Player
`loadTrackData`		Lädt Titel von Player
`previous`		Spielt vorherigen Titel
`next`		Spielt nächsten Titel
`stop`		Stoppt Wiedergabe
`togglePlayPause`		Pausiert oder setzt Wiedergabe fort
`play`	`(number)`	Spielt gewählten Titel im Player ab
`startPlaylist`	`(Playlist, number)`	Spielt gesamte Playlist
`HandlePlaylistEditor...ContentChange`	`(string)`	Setzt den Tab-Titel des Playlist-Editors

playlist-actions.component.ts **Bediefeld für Playlist**

Feld	Typ	Beschreibung
`playlist`	`Playlist`	Wiedergabeliste

Methoden	(Parameter): Rückgabe	Beschreibung
`sendToPlaylistEditor`		Setzt Playlist als aktive Editor-Liste
`play`		Spielt gesamte Playlist
`enqueue`		Hängt die Playlist an die Wiedergabe an

playlist-page.component.ts **Playlist-Listenansicht**

Feld	Typ	Beschreibung
`playlists`	`Playlist[]`	Wiedergabelisten
`nothingToShowText`	`string`	Alternativer Text
`selectedPlaylist`	`Playlist`	Ausgewählte Wiedergabeliste
`tracklistColumns`	`string[]`	Spalten der Titelliste der gewählten Playlist
`playlistColumns`	`string[]`	Spalten der Playlist-Liste
`edit`	`boolean`	Editierfunktion aktiviert

Methoden	(Parameter): Rückgabe	Beschreibung
`onResize`		Passt Elemente an Bildgröße an
`load`		Lädt wiedergabelisten
`selectPlaylist`		Wählt Playlist für die Detailansicht
`toggleEdit`		schaltet Editierfunktion um
`save`		Speichert gewählte Liste
`delete`		Löscht gewählte Liste

release.component.ts **Detailansicht Album**

Feld	Typ	Beschreibung
`loaded`	`Promise<boolean>`	Album ist geladen
`release`	`Release`	Album
`title`	`string`	Titelzeile
`card`	`ReleaseCardComponent`	ReleaseCardComponent-Element

release-actions.component.ts **Bedienfeld für Album**

Feld	Typ	Beschreibung
`release`	`Release`	input: Album

Methoden	(Parameter): Rückgabe	Beschreibung
`play`		Lädt Titel und startet Wiedergabe
`playTracks`		Spielt alle Titel

release-card.component.ts **Kartenansicht für Album**

Feld	Typ	Beschreibung
`tracksDisabled`	`boolean`	input: Schaltfläche Titelladen ist deaktiviert
`release`	`Release`	input: Album
`loaded`	`Promise<boolean>`	Titel des Albums sind geladen
`extracted`	`boolean`	ist ausgeklappt

Methoden	(Parameter): Rückgabe	Beschreibung
`toggleTracks`		klappt die Titelliste auf und zu

release-page.component.ts **Listenansicht Alben**

Feld	Typ	Beschreibung
`sort`	`MatSort`	Sortierparameter
`dataSource`	`MatTableDataSource`	Datenquelle für Alben-Tabelle
`displayedColumns`	`string[]`	Angezeigte Tabellenspalten
`filter`	`string`	Suchtext
`releases`	`Release[]`	Alben
`pageData`	`PageData`	Seiten Informationen der Tabelle

Methoden	(Parameter): Rückgabe	Beschreibung
`loadReleaseData`		Lädt Alben
`applyFilter`	`(string)`	Setzt Suchfilter

track.component.ts **Musiktitelkomponente für die Player-Ansicht**

Feld	Typ	Beschreibung
`active`	`boolean`	input: Wird gerade wiedergegeben
`paused`	`boolean`	input: Ist pausiert
`track`	`Track`	input: Musiktitel
`play`	`EventEmitter`	output
`pause`	`EventEmitter<boolean>`	output

Methoden	(Parameter): Rückgabe	Beschreibung
`togglePause`		emittiert inversen Pausestatus
`playTrack`		emittiert play

track-actions.component.ts **Bedienelement für Musiktitel**

Feld	Typ	Beschreibung
`trackId`	`string`	ID des zu steuernden Titels
`playlists`	`playlists`	Drop Down Menu Playlisten

Methoden	(Parameter): Rückgabe	Beschreibung
`play`		Spielt Titel
`enqueue`		Hängt Titel an aktive Wiedergabe an
`addToPlaylistEditor`		Hängt Titel an Playlist Editor an

track-page.component.ts **Listenansicht Musiktitel**

Feld	Typ	Beschreibung
`sort`	`MatSort`	Sortierparameter
`dataSource`	`MatTableDataSource<Track>`	Datenquelle der Titel
`paginator`	`MatPaginator`	Seitennavigationselement
`tracks`	`Track[]`	Musiktitel
`displayedColumns`	`string[]`	Angezeigte Spalten der Tabelle
`filter`	`string`	Suchtext
`totalElements`	`number`	Gesamte Titel in der Datenbank
`loading`	`boolean`	Ladevorgang läuft

Methoden	(Parameter): Rückgabe	Beschreibung
`loadTracks`	`(HttpParameter)`	Lädt Titel
`applyFilter`	`(string)`	Lade Titel mit Suchtext
`setFilter`	`(string)`	Setzt Suchtext
`handleScroll`		Lade nächste Seite bei Scroll-Ende

upload.component.ts **Uploadbereich**

Feld	Typ	Beschreibung
`files`	`FileList`	Dateien zum Upload
`changeFile`		Setzt neue Dateien zum Upload
`paginator`	`MatPaginator`	Seitennavigationselement
`tracks`	`Track[]`	Musiktitel
`displayedColumns`	`string[]`	Angezeigte Spalten der Tabelle
`filter`	`string`	Suchtext
`totalElements`	`number`	Gesamte Titel in der Datenbank
`loading`	`boolean`	Ladevorgang läuft

Methoden	(Parameter): Rückgabe	Beschreibung
`changeFile`		Setzt neue Dateien zum Upload
`fileDrop`	`(files)`	Lädt Dateien hoch via Drag and Drop
`upload`		Startet Dateiupload

A3.2.3 Services

Ein Überblick über die generischen Methoden der *Services* und die *Observables* befindet sich im Abschnitt 5.2.3. Hier sind dazu noch spezialisierte Methoden der Services gelistet.

Spezielle Methoden: Erfüllen besondere Anforderungen an die jeweilige Entität des Services

create()	Erzeugt eine neue *Playlist*-Entität
Endpunkt:	*POST: …/playlists*
Service:	*Playlist Service*
upload()	Sendet eine Datei im an das Back-End
Endpunkt:	*POST: …/files*
Service:	*File Service*
upload()	Sendet eine Datei im an das Back-End
Endpunkt:	*POST: …/files*
Service:	*File Service*
findCover(release)	Fordert das Cover-Artwork eines Albums an
Endpunkt:	*GET: .../releases/*release.id*/cover*
Service:	*Release Service*
addTrackToPlay ...(index), addTrackToActive ...EditorPlaylist(id, index)	*Fügt einen Musiktitel zu der aktiven Wiedergabe (player) bzw. der aktiven Liste im Editor (Playlist) hinzu. Index ist optional, wenn gesetzt ist es der Zielindex des Titels innerhalb der Liste.*
Endpunkt:	*POST: .../player/tracks* *POST: .../playlists/*activeEditorPlaylist.id*/tracks*
JSON-Body:	*{trackId:* id, *index:* index*}*
Services:	*Player Service, Playlist Service*
moveTrack ...(playlist, ...oldIndex, ...newIndex)	*Ändert den Index eines Titels innerhalb einer Liste. Der Parameter Playlist muss nur im PlaylistService angegeben werden*
Endpunkt:	*PATCH: .../player/tracks/*oldIndex *PATCH: …/playlists/*playlist.id*/tracks/oldIndex*
JSON-Body	*{trackNo:* newIndex*}*
Service:	*Player Service, Playlist Service*

setTracks(trackIds[ ], index)	*Ersetzt die Titel im Player durch einen oder mehrere andere. Der Parameter Index ist optional und definiert den Titel, der als erstes gespielt wird.*
Endpunkt:	*PATCH: .../player*
JSON-Body	*{id: 0, index:* index, *trackIds:* trackIds*, paused: false}*
Service:	*Player Service*
play(index) **play(trackId)**	*Startet die Wiedergabe eines Titels*
Endpunkt:	*PATCH: .../player*
JSON-Body	*{id: 0, index: index, paused: false}, {id: 0, index: 0, trackIds: [trackId], paused: false}*
Service:	*Player Service (play(index)), Track Service (play(trackId))*
stop()	*Stoppt die aktuelle Wiedergabe*
Endpunkt:	*PATCH: .../player*
JSON-Body	*{time: 0, paused = true}*
Service:	*Player Service*
togglePause (paused)	*Aktuelle Wiedergabe wird pausiert oder fortgesetzt*
Endpunkt:	*PATCH: .../player*
JSON-Body	*{paused =* !paused*}*
Service:	*Player Service*

A3.2.4 Upload-Direktive

upload.directive.ts **Upload-Direktive**

Wird in Komponenten eingebracht um Drag-and-Drop Uploads zu ermöglichen

Feld	Typ	Beschreibung
`files`	`FileList`	Dateien zum Upload
`background`		Hintergrund des Hosteements
`border`	`MatPaginator`	Rahmen des Hostelements

Methoden	(Parameter)	Beschreibung
`onDrop`	`(event)`	Emittiert Dateien an das Hostelement
`onDragOver`	`(event)`	Ändert Style (*background* und *border*)
`onDragLeave`	`(event)`	Setzt Style zurück auf Ursprungswerte