Institut für Architektur von Anwendungssystemen

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Design und Implementierung eines Storage-Backends für MUSE4Music

Fabian Bühler

Studiengang:	Softwaretechnik
Prüfer/in:	Prof. Dr. Dr. h.c. Frank Leymann
Betreuer/in:	Johanna Barzen M.A. Michael Wurster M.Sc.
Beginn am:	22. Mai 2017
Beendet am:	22. November 2017

Kurzfassung

Durch die fehlenden Zeitzeugen aus dem 19. Jahrhundert wird die Erforschung von dem Einfluss der Musikstücke aus dieser Zeit auf das Publikum bedeutend erschwert. Manche Quellen können zwar die Aussage eines Zeitzeugen ersetzen, die Quellenlage ist insgesamt aber unzureichend um Aussagen über eine große Bandbreite an Musikstücken belegen zu können. In dem Vision Paper von Barzen et al. [BBE+16] wurde für dieses Problem eine mögliche Lösung mittels computergestützter Mustersuche vorgeschlagen. Für die computergestützte Mustersuche müssen die Daten in einem Format vorliegen, in dem ein Algorithmus Muster erkennen kann. Für die Dateneingabe und spätere Mustersuche wurde in dem Paper ein Softwaresystem skizziert, mit dem die Mustersuche in den Partituren der Musikstücke stattfinden kann. In dieser Bachelorarbeit wird der Teil des skizzierten Softwaresystems, der für die Datenspeicherung verantwortlich ist, implementiert und die Architektur der zu implementierenden Software zusammen mit verschiedenen Implementierungsdetails beschrieben.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Listings

Verzeichnis der Algorithmen

1 Einleitung

Bei der Erforschung von historischen Musikstücken aus dem 19. Jahrhundert ist es nicht mehr möglich die Zeitzeugen zu befragen. Um die Wirkung einer bestimmten Komposition oder einer einzelnen Passage auf das damalige Publikum zu erfassen, kann man sich nur auf Quellen aus der damaligen Zeit beziehen. Die Partituren der Stücke selber haben sich über die Zeit zwar nicht verändert, sehr wohl aber die Hörgewohnheiten der heutigen Gesellschaft. Die Fülle an Musikrichtungen, die in den letzten Jahrzehnten entstanden sind und sich teilweise sehr stark von der Klassischen Musik des 19. Jahrhunderts unterscheiden, lassen es unwahrscheinlich erscheinen, dass ein Musikstück aus dem 19. Jahrhundert heute die gleichen Reaktionen auslöst wie damals [BBE+16].

Nur mit Quellenarbeit alleine ist es allerdings nicht möglich, eine Aussage über viele der Werke zu treffen. Die verfügbaren Quellen decken nicht alle verfügbaren Werke ab und viele Aussagen sind nicht detailliert genug [BBE+16]. Um dennoch Aussagen über Musikstücke treffen zu können, die nicht mit Quellen belegt werden können, haben sich die Autoren des Vision Papers zu MUSE4Music [BBE+16] verschiedene Methoden aus anderen Forschungsfeldern angesehen. Der von ihnen beschriebene Lösungsansatz für das Problem der unzureichenden Quellenlage nutzt computergestützte Mustererkennung. Besitzen verschiedene Passagen unterschiedlicher Werke eine hohe Ähnlichkeit zueinander, so ist mit einer gewissen Vorsicht anzunehmen, dass auch der Eindruck, den die infrage kommenden Passagen auf den damaligen Hörer hatten, ähnlich war. Die Eigenschaften, die diese Passagen aufweisen, können dann als Musikalisches Pattern (zu Deutsch: Muster) mitsamt Wirkung und belegenden Quellen dokumentiert werden. Ein Pattern kann dabei eine bestimmte Abfolge an Tönen oder Akkorden sein, die genutzt werden um eine gewisse Stimmung anzudeuten. Ein Musikalisches Pattern kann aber auch in der Art und Weise wie sich zwei Instrumentengruppen im Zusammenspiel verhalten begründet liegen. Als gemeinsames Merkmal der Vorkommen eines Patterns ist dabei die Assoziation des damaligen Publikums zu einer bestimmten Emotion oder einem Bild. Dazu müssen die Vorkommen des Patterns mit den vorhandenen Quellen referenziert werden. Mit den vorhandenen Quellen lässt sich die Qualität des Patterns überprüfen. Ein Pattern, das im Zusammenhang mit unterschiedlichen Reaktionen des Publikums steht, kann zum Beispiel keiner Reaktion eindeutig zugeordnet werden. Aus den so gewonnenen Patterns lässt sich dann eine Pattern Language [AIS77] erarbeiten, die Erkenntnisse zu den Kompositionsmethoden der Musik aus dem 19. Jahrhundert enthält.

In dem Vision Paper von Barzen et al. [BBE+16] wird dafür ein Softwaresystem skizziert, das im Wesentlichen aus drei Komponenten besteht: *(a) Erkennung der Eigenschaften eines Werkes (b) Auffinden von Patterns (c) Pattern Matching*. Die Komponente *(a)* umfasst dabei einen automatischen Analyseprozess, einen manuellen Überarbeitungsschritt und die Datenbank, in der die Werke mitsamt ihren Eigenschaften katalogisiert werden. In Komponente *(b)* werden mögliche Kandidaten für Patterns mittels Data-Mining Algorithmen aus der Datenbank extrahiert und von Experten begutachtet. Die in Komponente *(b)* erkannten Patterns werden dann in Komponente *(c)* genutzt um mittels Pattern Matching weitere Vorkommen zu finden. Die Komponenten *(b)*

und *(c)* benötigen einen umfangreichen Datensatz um Patterns mit einiger Zuverlässigkeit zu erkennen.

Das erste Ziel zur Umsetzung des beschriebenen Softwaresystems sollte also die Fertigstellung der Komponente *(a)* sein. Für eine erste Annäherung soll dabei ein Benutzerinterface zum Eingeben der Merkmale, sowie eine Datenbank, in der diese Merkmale abgelegt werden können, implementiert werden. Die Benutzeroberfläche soll, nach dem Vorbild von MUSE [Bar17], als *Single Page Application* umgesetzt werden. Als Single Page Application wird dabei eine Webseite bezeichnet, deren Inhalt dynamisch mittels Javascript angepasst wird. Anders als bei klassischen Webseiten muss bei einer Single Page Application keine neue Webseite vom Server abgerufen werden, wenn der Benutzer eine Unterseite aufruft. Die Benutzeroberfläche speichert die Daten dann mithilfe einer HTTP API in der Datenbank. Ziel dieser Bachelorarbeit ist die Entwicklung des Backends. Dazu gehört hier die Datenbank, sowie die zugehörige HTTP API. Die Entwicklung der Benutzeroberfläche ist Aufgabe einer zweiten parallel verlaufenden Bachelorarbeit.

Gliederung

Die Arbeit ist in folgender Weise gegliedert:

Kapitel 2 – Anforderungen: Hier werden die grundlegenden Anforderungen an das Backend vorgestellt. Außerdem werden verschiedene Designentscheidungen thematisiert.

Kapitel 3 – Architektur und Design: Hier wird das Design des MUSE4Music Backends erläutert.

Kapitel 4 – Zusammenfassung und Ausblick fasst die Ergebnisse der Arbeit zusammen und stellt Anknüpfungspunkte vor.

2 Anforderungen

2.1 Funktionale Anforderungen

Da die Arbeit in Zusammenarbeit mit dem musikwissenschaftlichen Institut der Universität zu Köln entstanden ist, war eine der Herausforderungen eine gemeinsame Sprache zu finden, die zwischen den Fachbereichen Musikwissenschaft und Informatik vermitteln kann. Die grundlegenden Anforderungen an MUSE4Music waren in mehreren Dokumenten in Form von Designentwürfen und den benötigten Taxonomien in Form von Mindmaps festgehalten. Diese sind auch Teil eines technischen Reports [EBH17]. Die einzelnen Taxonomien sind Teil einer Ontologie, mit der die Eigenschaften der Musik aus dem 19. Jarhundert erfasst werden können. In der Ontologie werden die Beziehungen zwischen den Taxonomien definiert. Die häufig hierarchisch strukturierten Taxonomien dienen als Klassifikationsschema für die einzelnen Eigenschaften, die in der Ontologie für die Musikstücke definiert sind. In mehreren Kundengesprächen wurden dabei die so dokumentierten Anforderungen auf Machbarkeit, Priorität und einige anderen Kriterien hin überprüft. Zudem wurden in den Kundengesprächen Detailfragen geklärt und neue Anforderungen, die sich erst im Verlauf der Bachelorarbeit ergeben hatten diskutiert. Im Folgenden werden die wichtigsten Kundenanforderungen an das Backend erläutert. Die Anforderungen beschreiben jeweils eine Eigenschaft der Anwendung aus der Sicht des Kundens.

Anforderung 1

Der Benutzer der Software möchte die Eigenschaften von *Werken*, *Personen*, *Werkausschnitten* und den *Stimmen* eines Werkausschnitts in der Software erfassen.

Ein Werk bezeichnet dabei eine musikalische Komposition aus dem 19. Jahrhundert. Ein Werkausschnitt steht für einen logisch abgegrenzten Teil eines Werkes. Dabei muss ein Werkausschnitt nicht mit den Satzgrenzen im Werk übereinstimmen. Personen sind entweder Komponisten eines Werkes, Dirigenten einer Uraufführung oder Teil eines Bezugs auf den Kompositionsstil einer Person. Eine Stimme ist eine Gruppe von Instrumenten, die in einem Werkausschnitt zusammen eine Melodie oder Begleitung spielen.

Aus Kundensicht wurde für diese User Story im Besonderen die graphische Oberfläche besprochen. Als Format für die Datenübertragung zwischen Backend und Frontend fiel die Wahl auf JSON, weil sich JSON aufgrund der übersichtlichen und intuitiven Darstellung von Objekten leicht verwenden lässt.

Aus Kundensicht gehört diese Anforderung zu den unbedingt umzusetzenden Anforderungen und hat dementsprechend die höchste Priorität.

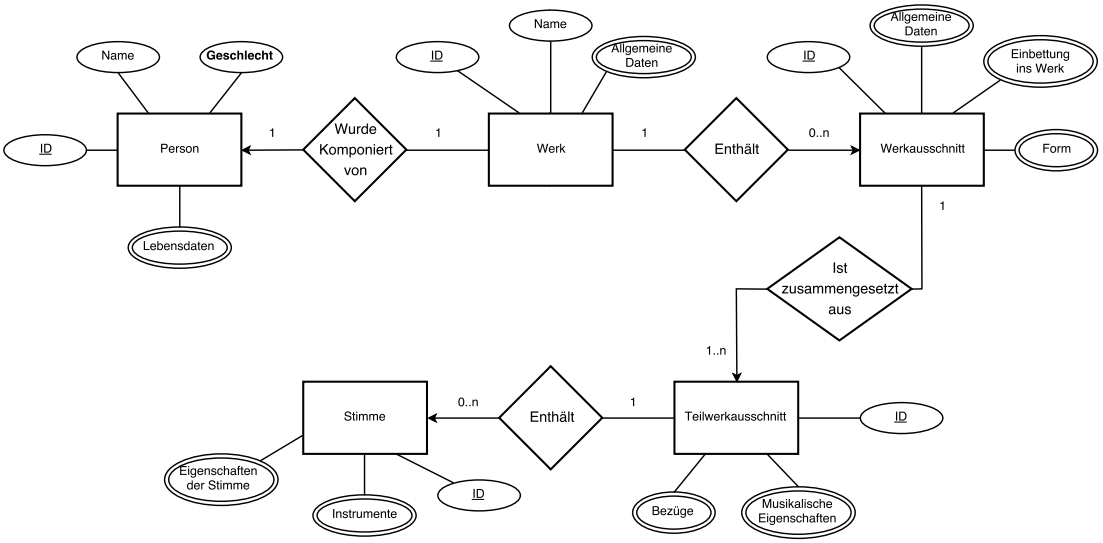

Abbildung 2.1: Abhängigkeiten zwischen Personen, Werken, Werkausschnitten, Teilwerkausschnitten und Stimmen

2.1.1 Anforderung 2

Werkausschnitte sollen nachträglich in mehrere Teilwerkausschnitte aufgeteilt werden können.

Diese Anforderung wurde erst im Verlauf der Arbeit gestellt und wurde in mehreren Gesprächen weiter verfeinert. Dabei wurde unter Anderem geklärt welche technische Umsetzung der Anforderung am ehesten entspricht. Die Kundenanforderung hat hier nahegelegt die meisten Eigenschaften eines Werkausschnitts als Eigenschaften eines Teilwerkausschnitts neu zu definieren und Werkausschnitte immer als Gruppe von einem oder mehr Teilwerkausschnitten anzusehen. Dabei gelten manche Eigenschaften wie zum Beispiel die Taktangabe der ursprünglichen Werkausschnitte jetzt als Eigenschaft der Gruppe. Im Folgenden ist mit *Werkausschnitt* immer diese Gruppe gemeint.

Auch diese Anforderung gehört zu den Anforderungen, die unbedingt umgesetzt werden müssen. Die Abhängigkeiten zwischen Personen, Werken, Werkausschnitten, Teilwerkausschnitten und Stimmen sind in Abbildung 2.1 dargestellt.

Anforderung 3

Die Benutzer der Software sollen sich in der Software mit Benutzernamen und eigenem Passwort sicher anmelden, bevor sie die Daten bearbeiten.

Für das Backend ergeben sich dadurch die Anforderung einen Login Mechanismus bereitzustellen, sowie die Benutzernamen und Passwörter sicher zu speichern. Außerdem dürfen nur angemeldete Benutzer die API verwenden können. Der geschätzte Arbeitsaufwand für diese User Story ist im Vergleich zu User Story 1 und 2 eher gering, da Librarys verwendet werden können. Weil die

Software anfangs nur im vergleichsweise geschützten internen Netzwerk der Universität zu Köln verwendbar sein soll, ist ein Login Mechanismus nicht von höchster Priorität.

Anforderung 4

Die erfassten Daten sollen in einem Review Prozess überprüft werden können.

Um den Review Prozess softwareseitig zu unterstützen, wurde mit dem Kunden abgesprochen Indikatoren für *bereit zum Review* und *Review durchgeführt* in die Daten einzufügen. Diese müssen sowohl im Frontend als auch im Backend vorhanden sein. Die Indikatoren in das Datenschema einzubauen ist von geringem Aufwand. Allerdings sind Filter, um zum Beispiel nur Werkausschnitte, die bereit zum Review sind abzufragen, möglich. Dies würde den Aufwand der Implementierung erhöhen. Da die Reviews erst durchgeführt werden können, wenn die Daten für den entsprechenden Teil vollständig sind, können diese Indikatoren auch erst nachträglich implementiert werden. Diese User Story hat deshalb eine niedrige Priorität.

Anforderung 5

Die Mehrheit der Eigenschaften soll Mehrfachauswahl erlauben.

Diese Anforderung ist mehr eine Detailfrage, als eine User Story, hat aber ihre Berechtigung hier aufgeführt zu werden, da eine Mehrfachauswahl für die Mehrheit der Eigenschaften funktionieren soll. Es geht also nicht um einzelne Sonderfälle. Dadurch ergibt sich die Anforderung für das Backend effizient mit den durch Mehrfachauswahl entstandenen Listen umgehen zu können.

Anforderung 6

Verschiedene Eigenschaften sollen bei Bedarf weiter spezifiziert werden können.

Die Spezifikationen gelten dabei nicht pauschal für ganze Felder, sondern nur für jeweils ein ausgewähltes Taxonomie-Element. Auch erlauben nicht alle Einträge einer Taxonomie eine Spezifikationen. Deshalb kann der Aufwand für die Implementierung als sehr hoch eingeschätzt werden. Aus Kundensicht haben die Spezifikationen eine geringe Priorität.

2.2 Nicht-funktionale Anforderungen

Aus der Aufgabenstellung und dem Einsatzgebiet der fertigen Software ergeben sich weitere nicht-funktionale Anforderungen. Anforderungen an Eigenschaften des Quellcodes wie *Wartbarkeit* (zu Englisch *Maintanability*) oder *Erweiterbarkeit* ergeben sich dabei zum Beispiel aus Anforderungen, die erst zu einem späteren Zeitpunkt umgesetzt werden sollen, oder zu erwartenden Anpassungen, die erst nach einer Testphase der Software vorgenommen werden können. Andere Anforderungen lassen sich direkt aus den Anforderungen an das Schwesterprojekt MUSE [Bar17] ableiten. Das MUSE Projekt beschäftigt sich mit computergestützter Mustererkennung bei Filmkostümen.

Performanz

Erfahrungen aus dem Schwesterprojekt MUSE haben dabei gezeigt, dass eine Benutzerzahl im unteren zweistelligen Bereich zu erwarten ist. Auch sind Wartezeiten beim Speichern der Änderungen durchaus hinnehmbar, solange sie unter etwa zehn Sekunden bleiben und im Hintergrund ablaufen. Die Anforderungen an die Performanz des Systems sind also vergleichsweise gering.

Wartbarkeit und Erweiterbarkeit

Die Erfassung der Musikstücke wird sich voraussichtlich über mehrere Jahre erstrecken. Dabei ist es fast ausgeschlossen, dass alle Taxonomien unverändert bleiben. Dies zeigen die Erfahrungen aus dem MUSE Projekt. Da das vorrangige Ziel ist, die Datenerfassung so schnell wie möglich beginnen zu können, sind einige Anforderungen erst im laufenden Betrieb umsetzbar. Dafür wird auch die dahinterstehende Datenbank angepasst werden müssen. Es sind aber keine grundlegenden Änderungen in der Datenstruktur zu erwarten, da diese schon manuell mittels Word-Dokumenten getestet wurde.

Dokumentation

Da das Projekt in zwei Teile aufgeteilt wurde, ist zur Umsetzung die Kommunikation zwischen Backend und Frontend, sowie den entsprechenden Entwicklern unabdingbar. Dabei gilt es vor allem Missverständnisse zu vermeiden und auch Details nicht zu vergessen. Für diese Aufgabe eignet sich eine Dokumentation der entsprechenden Schnittstelle, über die die zwei Teile kommunizieren. Demnach hat die Dokumentation der Schnittstelle im Vergleich zur Dokumentation des Quellcodes eine höhere Priorität. Sie sollte zu keinem Zeitpunkt veraltet sein, da dies direkt zu Fehlern in der Kommunikation mit der Schnittstelle führen kann.

Benutzbarkeit (zu Englisch Usability)

Für das Backend bedeutet Benutzbarkeit in erster Linie Benutzbarkeit der API. Diese hängt dabei von vielen Faktoren ab. Die Dokumentation der API ist dabei einer der bedeutendsten Faktoren. Daneben haben die möglichen HTTP Requests an die API, das zum Austausch genutzte Datenformat und die angebotenen Methoden den größten Einfluss auf die Benutzbarkeit der API. Weiterhin können hilfreiche Fehlermeldungen die Benutzbarkeit positiv beeinflussen.

Wiederverwendbarkeit

Das Forschungsfeld der Digital Humanities ist noch vergleichsweise neu. In diesem Forschungsfeld werden Geisteswissenschaftliche Themen mit computergestützten Verfahren erforscht. Es ist deshalb anzunehmen, dass der in MUSE und MUSE4Music verwendete Forschungsansatz auch in weiteren Projekten Anwendung finden wird. Diese würden ein ähnliches Backend benötigen. Ist die aus dieser Arbeit entstandene Software möglichst einfach als Grundlage für ähnliche Projekte wiederzuverwenden, dann wäre das ein Vorteil für kommende Projekte. Dafür gilt es aber nicht nur

den Quellcode Modular aufzubauen, sondern diesen auch für Forscher aus Disziplinen, die nicht zur Informatik affin sind, wie zum Beispiel den klassischen Geisteswissenschaften, zugänglicher zu machen.

Sicherheit

In der Datenbank werden keine sicherheitskritischen Informationen wie z. B. personenbezogene Daten gespeichert. Die Ansprüche an die Sicherheit des Backends sind dementsprechend eher gering. Trotzdem sollten die eingegebenen Daten vor unbefugten Änderungen oder gar Löschungen geschützt werden. Dazu gehört auch die Anforderung 3. Für die Passwörter, die für den Login gespeichert werden müssen gelten allerdings höhere Sicherheitsansprüche. Dies liegt daran, dass Passwörter häufig wiederverwendet werden und deshalb immer entsprechend abgesichert gespeichert werden sollten.

3 Architektur und Design

3.1 Backend-Applikation

Das Backend ist als Flask [Ron] Anwendung implementiert. Flask ist ein Microframework für Webentwicklung, dass in Python geschrieben ist. Da Flask ein Microframework ist, sind nur ein Developement-Server, eine Routing Engine, eine Templating Engine und Logging direkt in dem Framework enthalten. Weitere Funktionen wie ein OR-Mapper können über Plugins hinzugefügt werden. Die Sprache Python zeichnet sich durch eine leicht verständlichen Syntax aus und arbeitet nach dem Prinzip der geringsten Überraschung [Pet04]. Im Wesentlichen bedeutet das, dass z. B. bei einer möglicherweise uneindeutigen Interpretierung eines Ausdrucks dieser entweder nicht gültig ist, oder zumindest eine Fehlermeldung zur Laufzeit generiert wird. Dadurch werden mögliche Fehler im Programm einfacher gefunden. Für Python gibt es im Internet zahlreiche Tutorials zu alle möglichen Themen. Dadurch eignet sich die Sprache besonders gut für Programmiereinsteiger.

Als Alternative für Flask bietet sich ein Node.js Server an. Die Anwendung würde dann in Javascript oder Typescript implementiert. Um die neuesten Funktionen von Javascript zusammen mit einem automatisch überprüften Typsystem zu nutzen, muss Typescript eingesetzt werden. Dafür nuss der Quellcode vor dem Ausführen nach Javascript kompiliert werden. Dadurch wird die Zeit bis eine Änderung im Quellcode getestet werden kann deutlich verlängert. Typescript bietet zusätzlich zu dem Typsystem noch einige andere Funktionen, die die Entwicklung in Typescript im vergleich zu purem Javascript angenehmer gestalten. Da Javascript implementierungsbedingt in einem einzelnen Thread läuft, werden für eine Node.js Anwendung Promises für Nebenläufigkeit verwendet. Diese sind gerade für Programmiereinsteiger schwer zu verstehen. Da die Anwendung aber möglicherweise von Programmiereinsteigern als Grundlage für ähnliche Projekte genutzt werden soll, ist eine einsteigerfreundliche Programmiersprache von Vorteil. Für Python lässt sich auch eine statische Typprüfung umsetzen, die keine auswirkungen auf die Laufzeit des Programms hat. Deshalb wurde für dieses Projekt Flask als Framework ausgewählt.

3.1.1 Gesamtarchitektur

Die MUSE4Music Anwendung besteht aus einem Web Frontend und dem Backend. In Abbildung 3.1 ist der Aufbau des MUSE4Music Backends im Zusammenhang mit der Umgebung dargestellt. Das Web Frontend wird dabei ebenfalls vom MUSE4Music Backend an den Client ausgeliefert. Dieser kann das Web Frontend über seinen Browser aufrufen und benutzen. Die Daten werden von dem Web Frontend über die HTTP API vom Backend geladen. Dieses ist dabei in einer Schichtenarchitektur umgesetzt. Die Anwendung wird über WSGI an den HTTP Server angebunden. WSGI ist ein universelles Interface zwischen HTTP Servern und Python Programmen. Das Flask Framework übernimmt dabei die Kommunikation mit dem HTTP Server. Die API ist in zwei APIs mit jeweils abgegrenzten Funktionen aufgeteilt. API und User API greifen über die Datenbankmodelle auf die Datenbank zu.

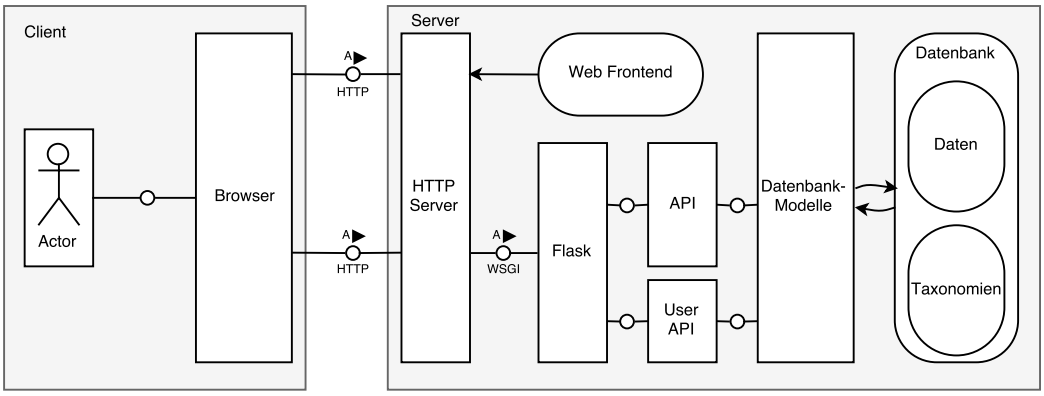

Abbildung 3.1: Gesamtarchitektur MUSE4Music

3.1.2 Datenbank und OR-Mapper

Wie in dem Schwesterprojekt MUSE [Bar17] sollen die Daten in einer Relationalen Datenbank wie MySQL oder MariaDB gespeichert werden. Für den Zugriff auf die Datenbank wird dabei das Flask-SQLAlchemy Plugin genutzt. Dieses erlaubt das automatische Anlegen der benötigten Datenbanktabellen nach den definierten Modellen. Das hat den Vorteil, dass kein SQL Script zum Erstellen der Datenbank an anderer Stelle aktuell gehalten werden muss. Die erstellte Datenbank passt immer zu den Modellen im Quellcode.

Die von Flask-SQLAlchemy genutzte Library SQLAlchemy [SQL] bietet einen großen Funktionsumfang. Sie unterstützt mehrere SQL-fähige Datenbanksysteme. So kann zum Testen eine SQLite Datenbank verwendet werden, die keine Installation eines Datenbankservers benötigt, während im Produktivsystem MySQL, MariaDB oder PostgreSQL eingesetzt werden. Zur Performanzoptimierung gibt es die Möglichkeit die automatisch generierten SQL-Statements durch optimierte Varianten zu ersetzen. Mit Flask-QueryInspect gibt es auch ein Plugin, dass bei der Analyse der SQL-Statements behilflich ist. Falls nachträgliche Änderungen an der Datenbank nötig sein sollten, existiert mit Flask-Migrate ein Plugin, dass die nötigen SQL-Statements soweit möglich automatisch generiert.

3.1.3 HTTP API

Auch für den Bau einer HTTP API gibt es einige Plugins für Flask. Unter Berücksichtigung der Anforderungen an die Dokumentation (Abschnitt 2.2) sowie die Benutzbarkeit (Abschnitt 2.2) der API wurde das Flask RestPlus [noi] Plugin ausgewählt. Dieses Plugin erlaubt die Dokumentation der API nach der OpenAPI Spezifikation 2.0 [Ope14]. Aus dieser Dokumentation lässt sich außerdem direkt Code für Angular 2 generieren. Die für die Dokumentation benötigten Modelle werden von der Library ebenfalls für Marshalling und Validierung verwendet. Dadurch wird eine hohe Benutzbarkeit der API bei minimalem Mehraufwand für Validierung und Dokumentation gewährleistet.

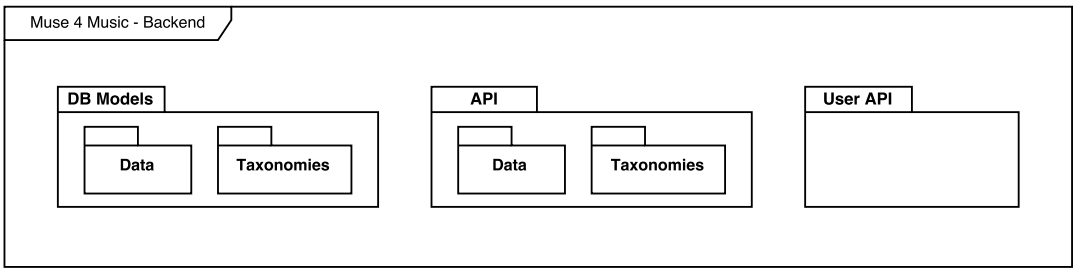

Abbildung 3.2: Vereinfachtes Paketdiagramm

3.1.4 Architektur

Auf der obersten Ebene ist die Anwendung wie in Abbildung 3.2 in drei Teile aufgeteilt. Ein Teil enthält dabei alle Datenbankmodelle. Die HTTP API ist in Zwei APIs aufgeteilt. Die User API, deren Aufgabe die Authentifizierung und Verwaltung der Benutzer ist, wurde in ein eigenständiges Modul ausgelagert. Diese Trennung erfolgt dabei nicht nur über das Python eigene Modulsystem, sondern auch über die von Flask angebotenen Blueprints. Eine Flask Anwendung kann dabei mehrere Blueprints besitzen. Ein Blueprint bildet dann für das Routing eine abgeschlossene Einheit.

Die benötigten Datenstrukturen lassen sich in zwei Gruppen aufteilen: Stammdaten und Bewegungsdaten. Als Stammdaten gelten dabei die Taxonomien und Personen. Bewegungsdaten, also Daten die sich häufig ändern, sind Werke, Werkausschnitte, Teilwerkausschnitte und Stimmen. Die Taxonomien haben eine gewisse Sonderstellung, da sie im Normalfall nicht von den Benutzern der Anwendung verändert werden. Stattdessen werden sie beim ersten Start der Anwendung initialisiert. Die Daten, wie zum Beispiel die verschiedenen Komponisten, sollen hingegen von den Benutzern verändert werden können. Dabei sind die Werte der Mehrheit der Felder eines Objekts auf jeweils die Elemente einer Taxonomie beschränkt. Es ist also sinnvoll, die Taxonomien von den anderen Daten zu trennen. Diese Trennung ist dabei sowohl bei den Datenbankmodellen, als auch in der API umgesetzt. Die Datenbankmodelle, die für die Benutzerverwaltung benötigt werden sind direkt im „DB Models“-Modul. Eine vollständige Trennung wie bei der User API ist in der Datenbank nicht möglich, da die Datenobjekte später auch den Benutzer, von dem sie erstellt worden sind, speichern können sollen.

In der Abbildung 3.2 nicht gezeigt ist das „debug-routes“ Modul. Es enthält mehrere Funktionen die beim Debugging nützlich sind und wird auch nur im debug Modus geladen. Da das Modul im Produktivbetrieb nicht verfügbar ist, wurde es auch in Abbildung 3.2 weggelassen.

3.1.5 Taxonomien

Die Taxonomien bilden die Grundlage für die Datenerfassung sowohl im MUSE, als auch im MUSE4Music Projekt. In beiden Projekten gibt es zwei Arten von Taxonomien: Listen und Bäume. Dies spiegelt sich auch in dem Klassendiagramm Abbildung 3.3 wider. In der Sprache Python ist Mehrfachvererbung erlaubt. Die Taxanomie-Klassen, die hier als Interfaces dargestellt sind,

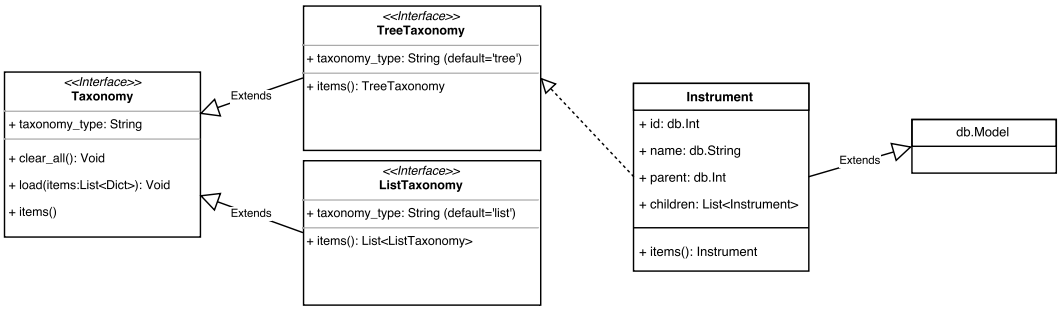

Abbildung 3.3: Auschschnitt aus dem Klassendiagramm – Taxonomien

enthalten deshalb direkt die Implementierungen der entsprechenden Methoden. Dadurch, dass alle Taxonomien von der gleichen Klasse erben, können sie einfach von anderen Datenbankmodellen unterschieden werden.

Die Taxonomien-Tabellen müssen nach dem Erstellen der Datenbank direkt mit Werten befüllt werden. Eine typische Vorgehensweise eine Datenbanktabelle mit Werten zu bestücken ist es ein SQL Script anzulegen, das die entsprechenden `INSERT`-Statements enthält. Was für einen Datenbankingenieur intuitiv funktioniert, ist in diesem Kontext eher ein Nachteil, denn SQL-Statements erschweren die Lesbarkeit der Taxonomien und brauchen andere Werkzeuge zum Bearbeiten, als beispielsweise eine Excel Tabelle. Da gerade die Taxonomien auch von den Geisteswissenschaftlern bearbeitet werden, gilt es hier möglichst viel der zugrundeliegenden technischen Implementierung zu abstrahieren, um dem Ziel der Wiederverwendbarkeit (Abschnitt 2.2) zu folgen.

Die Taxonomien von MUSE4Music wurden in Mindmaps mit dem Programm Freeplane[1] erstellt. Diese Mindmaps eignen sich besonders dazu Taxonomien darzustellen, da sie ebenfalls auf einer Baumstruktur beruhen. Das auf XML basierende Dateiformat der Mindmaps ist dabei gut dokumentiert und leicht verständlich, was ein Parsen dieser Dateien zum Erstellen der benötigten SQL-Statements möglich macht. Durch die in Freeplane vorhandenen Möglichkeiten zur Formatierung einzelner Knoten der Mindmap und deren Beschriftungen, ist ein Parser, der den Text aus dem Knotem extrahiert, ohne dass dieser von einem Mensch überprüft werden muss, zu aufwändig für dieses Projekt. Aus diesem Grund werden die Mindmaps nicht direkt in ein SQL-Script übersetzt, sondern stattdessen zunächst in eine CSV-Datei übersetzt. Diese lässt sich mit jedem handelsüblichen Tabellenkalkulationsprogramm bearbeiten. In Listing 3.1 ist ein Ausschnitt einer Taxonomie im CSV Format zu sehen.

Flask erlaubt es eigene Administrationskomandos zu implementieren. Diese können über die Komandozeile des jeweiligen Betriebssystems ausgeführt werden und funktionieren auch wenn die App ausgeführt wird. Sie eignen sich damit zum Beispiel für die Initialisierung der Datenbank und der Taxonomien. Um ein wiederholtes Ausführen des Kommandos für jede einzelne Taxonomie zu verhindern, wird beim Import der CSV Dateien ein Ordnerpfad verlangt. Alle CSV Dateien,

[1] www.freeplane.org

Listing 3.1 Beispiel einer Taxonomie im CSV Format

```
name,parent,description
root,,
niedrig,root,"Saemtliche Klaenge lassen sich aus dem diatonischen Material des tonalen
    Zentrums oder dem Doppeldominantklang bilden."
eher niedrig,root,"Einsatz von Zwischendominanten und -subdominanten, Verwendung von
    Terzverwandtschaften 2. bis 4. Ordnung im Umfang von 20% der verwendeten Klaenge."
hoch,root,"Einsatz erweiterter Zwischendominantik, Ellipsen, Verwendung von
    Terzverwandtschaften 2. bis 4. Ordnung im Umfang von mehr als 20% der verwendeten
    Klaenge."
```

die sich direkt in dem angegebenen Ordner befinden (Unterordner ausgenommen), werden dann anhand des Dateinamens einer Taxonomieklasse zugeordnet. Dazu werden sowohl der Dateiname als auch der Klassenname zu einem Namen in Großbuchstaben normalisiert. Die eventuell in der Datenbank enthaltenen Elemente werden standardmäßig nicht gelöscht, was nachträgliche Ergänzungen zu schon vorhandenen Taxonomien über diesen Importmechanismus erlaubt. Beim Import der Einträge aus den CSV Dateien wird die Reihenfolge der Einträge beibehalten. Dadurch ist es möglich den Taxonomieeinträgen eine Sortierung zu geben, die nicht vom Namen des Eintrags abhängig ist.

Da alle Taxonomien aus den Mindmaps extrahiert wurden, haben alle CSV Dateien auch eine „parent“ Spalte. Der Wurzelknoten wird dabei nach Konvention immer `root` heißen. Durch die Identifikation der Einträge allein über den Namen ergibt sich die Einschränkung, dass kein Eintrag den gleichen Namen besitzen darf. Gleichzeitig wird aber auch der Vaterknoten über den Namen referenziert, was die Lesbarkeit der Taxonomie im CSV Format fördert. Da die Listentaxonomien weder einen `root`-Knoten haben, noch eine Beziehung zum Vaterknoten benötigen, werden diese zusätzlich angegebenen Informationen beim Import ignoriert, wenn es sich bei der Taxonomie um eine `ListTaxonomy` handelt.

Für den Import der Taxonomien werden alle Taxonomie-Klassen automatisch erkannt, damit sie mit den CSV Dateien verglichen werden können. Diese Funktion wird auch von der API genutzt um die Taxonomien bereitzustellen. Für neue Taxonomie muss lediglich im Taxonomies Paket ein neues Modell angelegt werden, das von `TreeTaxonomy` oder `ListTaxonomy` erbt. Die neue Taxonomie wird dann automatisch erkannt und sowohl beim Import, als auch von der API berücksichtigt. Dadurch kann ein API Endpunkt die Aufgabe vieler einzeln angelegter API Endpunkte übernehmen.

3.1.6 Datenmodelle

Die Datenbankmodelle für die Taxonomien unterscheiden sich stark von den Datenbankmodellen für die anderen Datenstrukturen wie zum Beispiel Personen, Werke oder Werkausschnitte. Während Taxonomien untereinander keine Beziehungen haben, haben diese Datenmodelle Beziehungen zu Taxonomien und zu anderen Datenmodellen. Ein Werk hat zum Beispiel eine Person als Komponist und mehrere Werkausschnitte. Manche Attribute eines Datenmodells, wie der Name einer Person, sind direkt in der jeweiligen Tabelle gespeichert und können einfach abgerufen werden. Die Mehrheit der Attribute ist jedoch ein Verweis auf eine Taxonomie oder ein anderes Datenmodell. Für diese Attribute wird in der Datenbank lediglich eine Spalte mit einem Foreign-Key Constraint angelegt. In dieser Spalte befindet sich die ID des referenzierten Ojektes. Um den

Namen und die Beschreibung eines referenzierten Taxonomieeintrages zu bekommen, muss dieser extra von der Datenbank angefordert werden. Dies geschieht entweder in einer zweiten Anfrage an die Datenbank, oder über ein `JOIN` in der Anfrage zu dem Objekt, in dem der Taxonomieeintrag referenziert wurde.

Damit dem Endbenutzer der Anwendung von referenzierten Taxanomieeinträgen der Namen des jeweiligen Eintrags angezeigt werden kann, muss die Referenz vor dem Anzeigen aufgelöst werden. Dabei gilt es zu beachten, dass die Anwendung als Webanwendung konzipiert ist, und die Verbindung zwischen API und der Webseite, die an den Endbenutzer ausgeliefert wird, sehr viel schlechtere Performanzeigenschaften haben wird, als die Verbindung zwischen API und Datenbank, die beide auf derselben Maschine laufen sollen. Es bietet sich also für die API an bei den gesendeten Objekten alle Referenzen im Voraus aufzulösen.

Der eingesetzte OR-Mapper SQLAlchemy erlaubt es deshalb für referenzierte Objekte anzugeben, wie diese geladen werden sollen. Neben der Möglichkeit das Objekt beim ersten Zugriff aus der Datenbank nachzuladen, gibt es auch die Möglichkeit, das Objekt mit einem `JOIN` oder einer *SubQuery* zu laden. Die unterschiedlichen Strategien haben dabei jeweils andere Auswirkungen auf die Performanz einer Anfrage. Die tatsächlichen Auswirkungen auf die Performanz lassen sich jedoch erst mit einer einigermaßen gefüllten Datenbank verlässlich messen. Deshalb sind alle Entscheidungen, welche Strategie für welche Relationen zu verwenden ist, als vorläufig anzusehen.

Haben die Objekte, die auf diese Weise automatisch geladen wurden, weitere Abhängigkeiten, die ebenfalls automatisch geladen werden, so kann leicht ein Schneeballeffekt ausgelöst werden. Um dies zu verhindern, dürfen die Referenzen, die Objekte referenzieren, die weitere Objekte referenzieren, nur dann automatisch aufgelöst werden werden, wenn das Objekt, das die Referenz enthält, direkt abgerufen wird. SQLAlchemy erlaubt es dazu die Strategie zum Auflösen der Referenzen für jede Datenbankabfrage einzeln zu konfigurieren. Die Konfiguration einzelner Datenbankabfragen benötigt dabei einen hohen Grad an Fachwissen und ist sehr fehleranfällig, da jede Änderung eines Datenbankmodells eine Anpassung aller betreffenden Abfragen erfordern kann. Dabei können die einzelnen Abfragen weit über den Quellcode verteilt liegen, was die Wahrscheinlichkeit erhöht, das eine der Abfragen bei einer Änderung übergangen wird.

Als Lösung bietet sich hier die Nutzung eines Mixins [LRS15] an. Das Mixin stellt dabei eine „`get`“ Funktion bereit, die die Datenbankabfrage entsprechend vorkonfiguriert. Für die Konfiguration der Abfrage werden dabei lediglich die Namen der Attribute benötigt, die automatisch geladen werden sollen. Diese müssen händisch in ein Klassenattribut eingetragen werden. Dadurch können alle Anpassungen zum Ändern der Strategie zum automatischen Auflösen der Referenzen innerhalb des Datenbankmodells vorgenommen werden. Die durch das Mixin hinzugefügte `get` Funktion versteckt dabei gleichzeitig einen großen Teil der Komplexität der Datenbankabfragen. Als Eingabe für die `get` Funktion ist dabei sowohl die ID des Objektes, als auch ein Python Dictionary (entspricht einer HashMap) mit einem ID Eintrag erlaubt. Dadurch können die JSON Objekte der API direkt genutzt werden, um das entsprechende Objekt aus der Datenbank zu laden, ohne zuerst die ID aus dem Objekt in einer temporären Variable zu speichern. Dadurch wird der Code insgesamt übersichtlicher.

Listing 3.2 Beispiel API-Modell zum Anlegen einer Person

```
person_post = api.model('PersonPOST', {
   'name': fields.String(default='', required=True, example='admin'),
   'gender': GenderField(required=True, example='male', enum=['male', 'female', 'other'])
})
```

3.1.7 API Modelle

Das verwendete Plugin RestPlus nutzt eigene Modelle für die Dokumentation der API, die Serialisierung der Objekte und die Überprüfung von ankommenden Daten. Listing 3.2 enthält ein solches API-Modell. Das eigentliche Modell besteht dabei aus einer Sammlung von Felddefinitionen in einem Dictionary. Der Key des Felds im Dictionary entspricht dabei dem Namen, mit dem das Feld später serialisiert wird, und, sofern nicht anders angegeben, auch dem Namen des Attributs in dem zu serialisierenden Objekt. Der `default` Wert wird dabei verwendet, falls das entsprechende Attribut nicht gefunden wurde oder kein Wert gesetzt wurde (dies entspricht in Python dem Wert `None`). Für die API-Dokumentation werden aus den Modellen auch direkt benutzbare Beispiele generiert, die in der SwaggerUI[2] bereitgestellt werden. Dazu wird, falls vorhanden der `example` Wert des jeweiligen Feldes verwendet. Für komplexere Datentypen ist es möglich, eigene `Field` Klassen zu erstellen. Das in Listing 3.2 verwendete `GenderField` ist ein eigens definierter Feldtyp. Diese Ansammlung von Feldern wird mit dem Aufruf von `api.model` bei der API unter dem angegebenen Namen registriert.

Diese API-Modelle sind auf den ersten Blick redundant zu den Datenbankmodellen. Die Informationen zum Datentyp und den default Werten sind schon in den Datenbankmodellen enthalten. Allerdings enthalten die API-Modelle zusätzliche Informationen für die Dokumentation der API. Über die API-Modelle kann auch nur eine Teilansicht des Objekts realisiert werden, in der nur ausgewählte Attribute des ursprünglichen Objekts verfügbar sind. Diese Informationen können nicht sinnvoll von einem Algorithmus generiert werden. Sehr wohl aber lässt sich aus den Datenbankmodellen automatisch eine Liste entsprechender Felder generieren, deren Datentyp und default Werte aus dem Datenbankmodell ausgelesen werden. Die generierten Felder können dann direkt in den API-Modellen verwendet werden. Die zusätzlichen Informationen für die Dokumentation der API werden den Feldern nachträglich hinzugefügt. Da eine solche Funktion nur während der Entwicklung hilfreich ist, wurde sie in dem in Abschnitt 3.1.4 erwähnten „debug-routes“ Modul implementiert. Dieses Modul wird nur im Debug-Modus geladen, weshalb die Funktion keinen Einfluss auf den Produktivbetrieb hat.

Neben der Serialisierung von Objekten unter der Dokumentation der API, können die API-Modelle auch für die Validierung von serialisierten Objekten verwendet werden. Die Validierung erfolgt dabei nach dem JSON-Schema V4 Standard [GZC13]. Dieser Standard ist weitestgehend ein Superset des OpenAPI Standards [Ope14] zur Dokumentation von Objekten. So unterstützt der OpenAPI Standard zur Vererbung von Modellen nur der Operator `AllOf` während in JSON-Schema zusätzlich `AnyOf` und `OneOf` unterstützt werden. Dies hat zufolge, dass Polymorphie vom OpenAPI Standard nur unzureichend unterstützt wird. Auch bei den zur Auswahl stehenden Datentypen unterscheiden sich die beiden Standards. Der OpenAPI Standard erlaubt keinen Datentyp `null` und

[2] https://swagger.io/swagger-ui/

Algorithmus 3.1 Update Algorithmus

```
procedure UPDATE(self, json_object)
    for all (attribute, class) in self.attributes_to_update do
        new_value = json_object.GET(attribute)
        if ISUPDATEABLE(class) and not attribute in self.reference_only_attributes then
                                                        // Update eines geschachtelten Objekts
            GETATTRIBUTE(self, attribute).UPDATE(new_value)
        else if ISDBMODEL(class) then
                                                                // Update einer Referenz
            new_object = class.GETOBJECTBYID(new_value)
            SETATTRIBUTE(self, attribute, new_object)
        else if class ∈ {int, float, str, bool} then
                        // Diese Klassen entsprechen den Datentypen aus dem JSON Standard.
            SETATTRIBUTE(self, attribute, new_value)
        else if class = CustomType then
                                        // Hier können eigene Datentypen hinzugefügt werden.
            new_value = CONVERTTOCUSTOMTYPE(new_value)
            SETATTRIBUTE(self, attribute, new_value)
        end if
    end for
end procedure
```

damit auch keine `null`-Werte in Objekten. Da diese beiden Unterschiede Auswirkungen auf die Dokumentation und Validierung haben, beeinflussen sie indirekt auch die API. Die Auswirkungen auf die API werden in Abschnitt 3.3.4 weiter ausgeführt.

3.1.8 Updates

Die API-Modelle können für die Serialisierung von Objekten verwendet werden, nicht aber für die Deserialisierung. Diese Funktion ist von der Library nicht vorgesehen. Ankommende Objekte im JSON Format können aber weiterhin von Flask deserialisiert werden. Dabei werden die primitiven Datentypen wie zum Beispiel `String` und `Boolean` auf die äquivalenten Datentypen in Python abgebildet. JSON Objekte werden in Python zu Dictionaries und JSON Arrays werden zu Listen. Da der JSON Standard keine Datentypen für Datumsangaben oder Enums enthält, müssen diese vor einer Serialisierung in das JSON Format in einen JSON kompatiblen Datentyp umgewandelt werden. Entsprechend müssen diese Datentypen nach der Deserialisierung aus dem JSON Format wieder zurück in ihren Ursprungsdatentyp umgewandelt werden.

Die Werte aus diesen Dictionaries müssen dann in die entsprechenden Datenobjekte übertragen werden. Dazu muss, überall wo ein Objekt verändert werden soll, für jedes infrage kommende Attribut des Objekts der neue Wert aus dem deserialisierten JSON Objekt übernommen werden. Der Code für die einzelnen Attribute würde sich dabei lediglich in dem Attribut und der möglicherweise notwendigen Konvertierung des Datentyps unterscheiden. Deshalb bietet es sich auch hier an die Funktionalität, die für ein Update benötigt wird, in eine Mixin Klasse zu kapseln.

Die `update` Methode aus dem Mixin ist in vereinfachter Fassung in Algorithmus 3.1 beschrieben. Als Eingabe bekommt die Methode das zu aktualisierende Objekt als `self` und das schon auf die Python Datentypen abgebildete `json_object`, in dem die neuen Werte für die Attribute stehen. Die Methode iteriert dabei zunächst über alle Attribute des Objekts. In einer Klassenvariable, die von dem jeweiligen Datenbankmodell überschrieben werden kann, sind die Attribute, die auf diese Weise aktualisiert werden sollen, jeweils mit ihrem Datentyp enthalten. Anhand des angegebenen Datentyps entscheidet der Algorithmus dann, wie mit dem neuen Wert für das Attribut umzugehen ist. Enthält das Attribut ein Objekt, das ebenfalls das Update Mixin implementiert hat, dann wird dieses über einen Aufruf der `update` Methode ebenfalls aktualisiert. Ein solches Verhalten ist allerdings nicht immer erwünscht. Bei der Aktualisierung eines Werkes sollte der Komponist nicht verändert, sehr wohl aber mit einem anderen Komponisten ausgetauscht werden können. Deshalb sind in einer zweiten Klassenvariable diejenigen Attribute enthalten, die nicht aktualisiert, aber ersetzt werden dürfen. Für diese Attribute wird also lediglich ein anderes Objekt referenziert. Dieses Verhalten gilt auch für Datenbankmodelle die nicht das Update Mixin implementiert haben. Da die von JSON unterstützten Datentypen in Python äquivalente Datentypen haben, können Werte vom Typ `bool`, `int`, `float` und `str` direkt übernommen werden. Für eigene Datentypen kann die Methode an dieser Stelle durch weitere `if` Bedingungen ergänzt werden.

In der vereinfachten Fassung des Update Algorithmus (Algorithmus 3.1) nicht enthalten sind kleinere Optimierungen und Randfälle. In der Implementierung des Algorithmus werden die Attribute nur dann neu gesetzt, wenn sich ihr Wert tatsächlich ändert. Dadurch werden unnötige Updates vermieden und die Minimallaufzeit einer Aktualisierung bei großen Objekten verkürzt. Zu den Randfällen gehören unter anderem die Fälle, bei denen ein Attribut, dass vor dem Update den Wert `None` hat, ein aktualisierbares Datenbankobjekt enthält. In diesem Fall muss zuerst ein neues Objekt instantiiert werden, bevor dieses aktualisiert werden kann. Ebenfalls fehlt in Algorithmus 3.1 die Überprüfung, ob das entsprechende Attribut auch in dem `json_object` enthalten ist. Der Datentyp des neuen Wertes muss an dieser Stelle nicht überprüft werden, da das `json_object` schon mittels eines API-Modells validiert wurde, bevor es an die `update` Methode übergeben wurde. Die zusätzliche Überprüfung, ob das Attribut auch wirklich in dem `json_object` enthalten ist, hilft dabei Felder in dem API-Modell zu entdecken, die versehentlich nicht als `required` also benötigt markiert wurden.

Mit dieser Implementierung des Update Algorithmus bleiben alle Informationen, die zum Aktualisieren eines Datenbankobjekts benötigt werden, in dem entsprechenden Datenbankmodell. Um ein Datenbankobjekt zu aktualisieren genügt es deshalb ein validiertes `json_object` an seine `update` Methode zu übergeben. Der Update Algorithmus beachtet dabei Randfälle und enthält eine Laufzeitoptimierung. Gleichzeitig werden die Datenbankobjekte einfacher zu Benutzen und zu Erstellen. Soll ein Datenbankobjekt aktualisierbar sein, dann müssen lediglich alle aktualisierbaren Attribute mit dem entsprechenden Datentyp in eine weitere Klassenvariable im Datenbankmodell abgelegt werden und das Datenbankmodell muss von dem Update Mixin erben. Wie bei den Feldern der API-Modelle gibt es auch hierfür entsprechende Hilfsfunktionen in dem „debug-routes“ Modul. In direkter Konsequenz zu der Kapselung der `update` Methode in einem Mixin, wird der Code in einem API Endpunkt, in dem ein Datenbankobjekt aktualisiert wird, übersichtlicher. Der API Endpunkt enthält dann nur noch den Code, der für die Behandlung der verschiedenen URL Parameter notwendig ist, sowie einen Aufruf von `update`.

Algorithmus 3.2 Listupdate Algorithmus

```
procedure LIST_UPDATE(self, old_values, new_values, item_class, item_class, mapping_class)
    to_delete = old_values \ new_values
    ids_to_add = EXTRACTIDs(new_values \ old_values)
    items_to_add = item_class.GETLISTBYID(ids_to_add)
    for all item in items_to_add do
        new_item = mapping_class(self, item)
        ADDTOSESSION(new_item)
    end for
    DELETE(to_delete)
end procedure
```

Wie in Anforderung 5 beschrieben, sollen viele Attribute Mehrfachauswahl unterstützen. Hier bietet es sich ebenfalls an, den Update Algorithmus für Listen in ein Mixin auszulagern. In Algorithmus 3.2 ist eine vereinfachte Fassung des implementierten Algorithmus beschrieben. Um dabei die nötigen Anfragen an die Datenbank zu minimieren, werden zuerst alle zu erstellenden Objekte und alle zu löschenden Objekte bestimmt. Da eine Relationale Datenbank keine Listen unterstützt wird hier eine `mapping_class` benötigt, die ein Datenbankmodell mit zwei Foreign Key Spalten ist. Die erste Spalte zeigt dabei auf das Objekt, das die Liste enthält, wärend die zweite Spalte das Objekt, das in der Liste enthalten sein soll referenziert. Mit der Methode `getListByID` wird eine Liste von Objekten mit einer Anfrage an die Datenbank geholt. Mit `addToSession` werden neue Objekte zu der laufenden Session hinzugefügt und beim Commit der Session in die Datenbank geschrieben. Die Reihenfolge der Elemente ist dabei nicht relevant, da eine Mehrfachauswahl, wie sie in Anforderung 5 beschrieben ist, keine Reihenfolge definiert. Die beiden Eingabeparameter `old_values` und `new_values` werden in dem Algorithmus aus diesem Grund als Menge behandelt. Der operator \ ist dabei die Mengenoperation der Differenz.

3.2 Datenbankdesign

Das Datenbankschema folgt im Wesentlichen direkt aus der Ontologie, wie sie in den Mindmaps ausgearbeitet ist (vgl. Abschnitt 2.1). Unter Berücksichtigung der Anforderung 2 ergeben sich die Abhängigkeiten aus Abbildung 3.4. Dabei haben alle Tabellen eine `ID` Spalte als Primary Key. Alternativ würde sich für Werk und Person auch der Name als Primary Key anbieten. Dann darf ein Name aber nur ein einziges mal benutzt werden. Die Namen sind aber nicht unbedingt eindeutig, da ein Name wie „Duett in d-Dur“ durchaus mehrfach vorkommen kann. Deshalb ist es sinnvoll, die Werke und auch Personen mit einer eindeutig generierten ID als Primary Key zu versehen. Werkausschnitt, Teilwerkausschnitt und Stimme haben keine eindeutigen Merkmale, die sich als Primary Key eignen. Sie benötigen deshalb einen generierten Primary Key. Da ein Werkausschnitt in dem Datenmodell prinzipiell nicht ohne ein Werk existieren kann, ist auch ein zusammengesetzter Primary Key der aus der `ID` des Werks und der `ID` des Werkausschnitts im Werk besteht denkbar. Analog gilt das Gleiche für Werkausschnitt und Teilwerkausschnitt, sowie Teilwerkausschnitt und Stimme. Für die Stimme würde der Primary Key aus insgesamt vier `ID` Spalten bestehen. Da immer alle Primary Keys bei einer Anfrage an die Datenbank berücksichtigt werden müssen, werden die SQL Anfragen an die Datenbank insgesamt

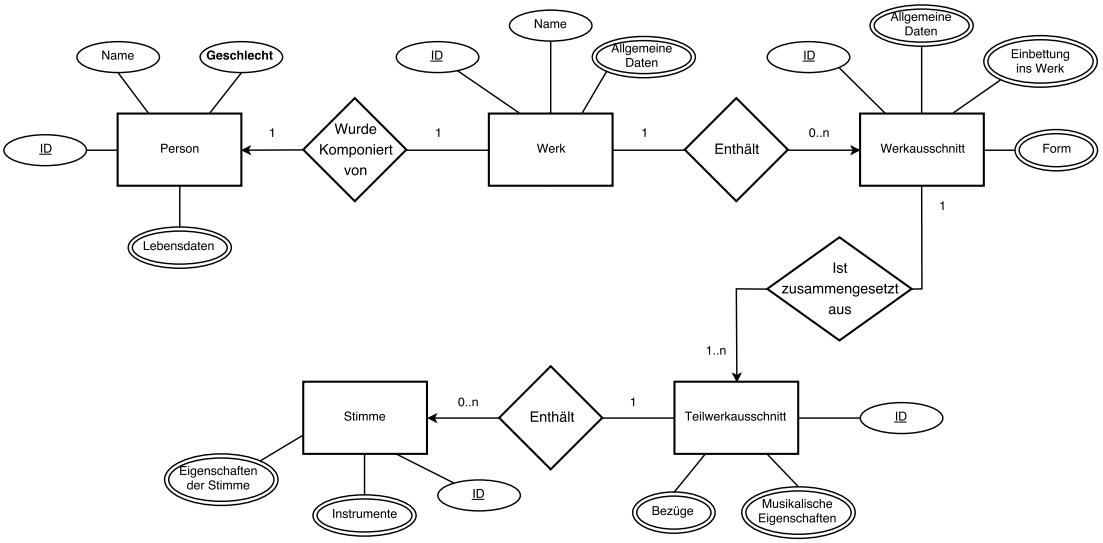

Abbildung 3.4: Datenbankschema

komplizierter. Der Vorteil von den zusammengesetzten Primary Keys ist, dass ein Werkausschnitt niemals ohne ein zugehöriges Werk existieren kann. Dies lässt sich aber auch durch eine Referenz auf das zugehörige Werk im Werkausschnitt erreichen, wenn diese ausgefüllt sein muss. Aus diesem Grund haben auch Werkausschnitt, Teilwerkausschnitt und Stimme jeweils nur eine `ID` generierte ID als Primary Key.

Das in Abbildung 3.4 dargestellte Diagramm ist dabei nicht vollständig. Die Attribute der einzelnen Entitäten wurden in dem Diagramm aufgrund ihrer Anzahl teilweise zu komplexen Attributen, die aus mehreren Werten bestehen können, zusammengefasst. Dies dient der besseren Übersicht im Diagramm und erlaubt es die verschiedenen Eigenschaften unter einfach verständlichen Namen zu gruppieren. Die musikalischen Eigenschaften des Teilwerkausschnitts umfassen dabei unter Anderem die notierte Lautstärke (Dynamik), das notierte Tempo, die Tonart, die verwendeten musikalischen Stilmittel wie zum Beispiel besondere Akkordfolgen, sowie Eigenschaften der Melodielinie. Diese wiederum bestehen selbst aus mehreren Attributen, von denen einige Listen sind. Folgt man der Ontologie, würde man diese Eigenschaften in jeweils eigene Tabellen auslagern und diese Tabellen dann im Teilwerkausschnitt referenzieren. Die Attribute der einzelnen Eigenschaften können aber auch direkt als Attribute des Teilwerkausschnitts übernommen werden. Dadurch werden insgesamt weniger Tabellen angelegt und eine Anfrage eines Teilwerkausschnitts benötigt weniger `joins`, was sich positiv auf die Performanz der SQL Anfrage auswirkt. Allein schon durch die Menge der Eigenschaften ist es jedoch sinnvoll, diese zu gruppieren. Dadurch wird zum Einen die Tabelle Teilwerkausschnitt übersichtlicher, zum Anderen werden auch teilweise Aktualisierungen erlaubt, für die ansonsten der gesamte Teilwerkausschnitt aktualisiert werden müsste. In der Weboberfläche werden die Eigenschaften des Teilwerkausschnitts in einzelnen Tabs bearbeitet. Diese Tabs halten sich dabei an die Gruppierung aus der Ontologie. Damit teilweise Aktualisierungen möglich sind, und wegen der besseren Übersicht, ist es sinnvoll die Attribute soweit möglich entsprechend der Ontologie auf eigene Tabellen aufzuteilen.

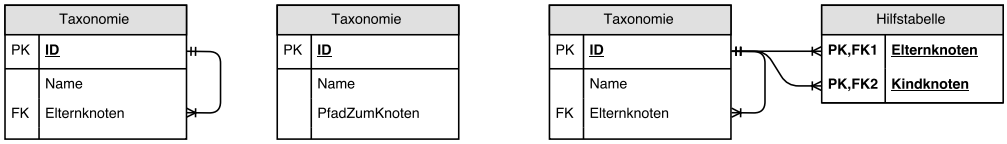

Abbildung 3.5: Alternative Darstellungen von Baumstrukturen in einer Relationalen Datenbank

3.2.1 Taxonomietabellen

Ebenfalls nicht in Abbildung 3.4 dargestellt sind die vielen Taxonomien. Diese machen einen großen Teil der Datenbanktabellen aus und sollten deshalb genauer betrachtet werden. In der Ontologie für MUSE4Music gibt es zwei Arten von Taxonomien. Die Taxonomien, die aus einer Auflistung von Werten bestehen, sind trivial in eine Datenbanktabelle zu speichern. Jede Zeile enthält einen Eintrag aus der Taxonomie. Für die Taxonomien, die eine Baumstruktur haben, ist eine Relationale Datenbank ein denkbar schlechter Ort um sie zu speichern. In Abbildung 3.5 sind drei Möglichkeiten dargestellt eine solche Taxonomie in einer Relationalen Datenbank zu speichern.

Die erste Möglichkeit entspricht der Tabelle ganz links in Abbildung 3.5. Dabei hat jeder Eintrag, bis auf den erste Eintrag, einen Verweis auf denjenigen Eintrag, der in der Hierarchie direkt über dem Eintrag steht. Dieser Ansatz hat neben seiner Einfachheit noch andere Vorteile. Zum Einen wird die gesamte Baumstruktur direkt mit den Mitteln der Relationalen Datenbank abgebildet. Dadurch ist es möglich die gesamte Struktur nur mit SQL Anfragen zu manipulieren. Zum Anderen lassen sich ganze Teilbäume einfach verschieben oder löschen. Um einen Teilbaum zu verschieben muss im Wurzelknoten des Teilbaums lediglich die Referenz zum Elternknoten angepasst werden. Sind die Foreign Keys in der Tabelle auf `CASCADE` eingestellt, dann lässt sich der gesamte Teilbaum löschen, indem der Wurzelknoten des Teilbaums gelöscht wird. Ebenfalls lassen sich neue Elemente an jedem Knoten des Baums leicht einfügen. Von Nachteil ist jedoch, dass SQL keine Rekursion erlaubt, weshalb alle direkt in SQL formulierten Anfragen die Hierarchie nur bis zu einer bestimmten Tiefe abrufen können. Diese Anfragen sind durch die geschachtelten `joins` außerdem sehr ineffizient.

Die zweite Möglichkeit, die der zweiten Tabelle in Abbildung 3.5 entspricht, versucht diese Einschränkung zu umgehen. Dazu wird der gesamte Pfad bis zu dem aktuellen Knoten kodiert in einer eigenen Spalte gespeichert. Mit einer klug gewählten Kodierung lassen sich Abfragen zu allen Kindern und Kindeskindern eines Knotens in einer SQL Anfrage formulieren. In der Anfrage muss lediglich überprüft werden, ob in dem Pfad zu einem Knoten die `ID` des Ausgangsknotens enthalten ist. Die Anfrage nach allen Elternknoten wird ebenfalls trivial, da diese in der Kodierung des Pfads zu dem Knoten enthalten sind. Gleichzeitig werden alle Änderungen am Baum aufwändiger. Um einen neuen Knoten einzufügen, muss der Pfad zu dem Knoten kodiert werden. Wird ein Teilbaum verschoben, dann muss für jeden Knoten in dem Teilbaum der Pfad entprechend angepasst werden. Außerdem können die Pfade nicht automatisch von der Datenbank überprüft werden.

Die dritte Möglichkeit aus Abbildung 3.5 benötigt eine Hilfstabelle. Ohne die Hilfstabelle entspricht die dritte Möglichkeit der Ersten. Um den Nachteil der nicht vorhandenen Rekursion in SQL auszugleichen, werden hier neben der Referenz auf den direkten Elternknoten in der Hilfstabelle

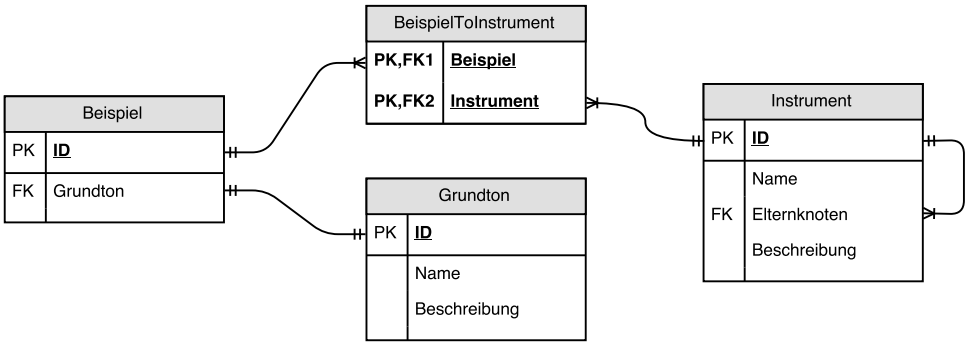

Abbildung 3.6: Relationen von Taxonomietabellen zu anderen Datenbanktabellen

auch alle Elternknoten bis hin zum Wurzelknoten referenziert. In der Hilfstabelle wird also die gleiche Information gespeichert wie in dem kodierten Pfad aus der zweiten Möglichkeit. Anders als bei der zweiten Möglichkeit, können hier die Pfade von der Datenbank automatisch überprüft werden. Mit entsprechenden Triggern lässt sich die Hilfstabelle auch automatisch verwalten. Nachteil ist hier der deutlich höhere Konfigurationsaufwand und die zusätzliche Tabelle.

Dass die Taxonomien relativ kleine Bäume mit einer geringen Tiefe sind, ist ein wichtiger Fakt, der die Entscheidung beeinflussen kann. Es spricht also nichts dagegen, die vollständige Tabelle abzufragen und die Baumstruktur erst im Arbeitspeicher wiederherzustellen. Der OR-Mapper von SQLAlchemy löst mit den Standardeinstellungen alle rekursiven Referenzen der verwendeten Taxonomien problemlos auf und speichert die Objekte im Arbeitspeicher zur späteren Verwendung. Dadurch ist der Nachteil der ersten Möglichkeit, keine rekursiven Anfragen mit SQL stellen zu können, vernachlässigbar. Da diese Möglichkeit auch den geringsten Aufwand bei der Umsetzung hat, fiel die Wahl auf die erste Möglichkeit.

In Abbildung 3.6 ist beispielhaft dargestellt, wie die Taxonomien mit den anderen Datenbanktabellen verknüpft werden. Dabei gilt es zwei Fälle zu unterscheiden. Im Normalfall sind die Taxonomien direkt über einen Foreign Key mit den anderen Tabellen verbunden. Dieser Fall ist in der Abbildung 3.6 unten dargestellt. Viele der einzutragenden Eigenschaften erfordern allerdings die Möglichkeit mehrere Elemente einer Taxonomie auswählen zu können. Für diesen Fall wird eine zusätzliche Verknüpfungstabelle zwischen der Taxonomietabelle und der anderen Datenbanktabelle benötigt. Diese ist in Abbildung 3.6 oben in der Mitte zu sehen. Da der Primary Key dieser Verknüpfungstabelle aus den beiden Foreign Keys zusammengesetzt ist, erlaubt diese Verknüpfung keine doppelte Auswahl. Dafür würde eine zusätzliche `ID` Spalte in der Verknüpfungstabelle benötigt, die die Rolle des Primary Keys übernimmt. Diese Verknüpfungstabellen folgen dabei, zur einfachen Erkennung im gesamten Projekt, einem festen Namensschema.

Auffällig ist, dass für die Liste an Instrumenten, die mit der „Beispiel“ Tabelle verknüpft ist, keine extra Spalte in der „Beispiel“ Tabelle benötigt wird. Enthält ein Objekt ausschließlich Listen, dann ist die entsprechende Datenbanktabelle bis auf die `ID` leer. Das entsprechende Datenbankobjekt ist trotzdem nicht leer, da der OR-Mapper die referenzierten Zeilen aus der Verknüpfungstabelle in einer Liste in dem Objekt zur Verfügung stellt. Auch zur Laufzeit des Programms entspricht damit

die Aufteilung der Attribute auf die Objekte der Aufteilung aus der Ontologie. Da die Datenbank direkt aus diesen Objekten generiert wird, werden die entstandenen „leeren“ Tabellen für eine bessere Aufteilung der Attribute auf Objektebene in Kauf genommen.

3.3 HTTP API

Die API des Backends orientiert sich an dem von Fielding [Fie00] beschriebenen REST Standard. Eine vollständige und gewissenhafte Umsetzung der REST Prinzipien benötigt allerdings viel Zeit und Arbeitsaufwand. Dabei war eine vollständig REST konforme API zu keinem Zeitpunkt eine Anforderung an die API des Backends. Die API wird auch nur von einem speziell für diese Anwendung entwickelten Client, der Weboberfläche von MUSE4Music, benutzt. Da Client und Server aufeinander abgestimmt werden können und es unwahrscheinlich ist, dass die API von anderen Clients benutzt wird, ist der Aufwand die API vollständig nach den REST Prinzipien zu implementieren nicht zu rechtfertigen. Trotzdem ist es sinnvoll bei der Implementierung weitestgehend den REST Prinzipien zu folgen, da diese weit verbreitet sind und eine gute Grundlage für die Architektur einer auf HTTP basierten API bieten.

Da die Gesamtanwendung, von der das Backend im Rahmen dieser Bachelorarbeit entwickelt wird, aus einem Client und einem Server besteht, sind die das Netzwerk betreffenden Eigenschaften des REST Standards nur von geringer Bedeutung für diese Anwendung. Auch muss der Server nicht mehrere Repräsentationen einer Ressource ausliefern können, da der Client nur eine dieser Repräsentationen anfragen wird. Die Möglichkeit zu der Ressource Code mitzuliefern, wie diese darzustellen ist, ist dabei bereits im REST Standard optional und hier ebenfalls unbedeutend, da Client und Server aufeinander abgestimmt werden können. Der Server arbeitet dafür aber Stateless. Alle nötigen Eingaben für eine Anfrage beim Server, werden vom Client der Anfrage angehängt. Cacheing wird dabei von den verwendeten Plugins nicht unterstützt. Für serverseitiges Cacheing existieren entsprechende Flask Plugins. Da ein guter Cache, der keine veralteten Objekte ausliefert und trotzdem die Performanz der Anwendung steigert, nicht einfach zu implementieren ist, und die Performanz der Anwendung nicht die höchste Priorität hat, wird in dieser Bachelorarbeit kein Cache in der API verwendet. Dieser kann nachträglich implementiert werden, ohne dass die Semantik der API verändert werden muss. Bei einer späteren Implementierung des Caches stehen außerdem bessere Daten zur Nutzung der Anwendung zur Verfügung.

3.3.1 Strukturierung der Ressourcen

Die wichtigste Abstraktion im REST Standard ist die Ressource [Fie00]. Dabei bezieht sich eine Ressource nicht zwangsläufig auf ein Objekt. Alles was benannt werden kann, kann eine REST Ressource sein. Da im MUSE4Music Backend ausschließlich Daten verwaltet werden, entsprechen fast alle REST Ressourcen auch einem Datenbankobjekt.

Für die MUSE4Music API wurden die folgenden Ressourcen festgestellt. Diese entsprechen weitestgehend auch den in Abbildung 3.4 gezeigten Entitäten.

routes Die `routes` Ressource enthält Links zu den weiterführenden REST Ressourcen. Diese können von einem Client dazu genutzt werden die API zu entdecken.

Pfad	Ressource (GET, PUT)	Ressource (POST)	Methoden			
			GET	PUT	POST	DELETE
/	routes	–	GET			
/taxonomies	[taxonomy]	–	GET			
/taxonomies/list/{taxonomy}	taxonomy	tax.-item	GET		POST	
/taxonomies/list/{taxonomy}/{item_id}	tax.-item	–	GET	PUT		
/taxonomies/tree/{taxonomy}	taxonomy	–	GET			
/taxonomies/tree/{taxonomy}/{item_id}	tax.-item	tax.-item	GET	PUT	POST	DELETE
/persons	[person]	person	GET		POST	
/persons/{id}	person	–	GET	PUT		DELETE
/opuses	[opus]	opus	GET		POST	
/opuses/{id}	opus	–	GET	PUT		DELETE
/opuses/{id}/parts	[part]	part	GET		POST	
/parts	[part]	–	GET			
/parts/{id}	part	–	GET	PUT		DELETE
/parts/{id}/subparts	[subpart]	subpart	GET		POST	
/subparts	[subpart]	–	GET			
/subparts/{id}	subpart	–	GET	PUT		DELETE
/subparts/{id}/voices	[voice]	voice	GET		POST	
/subparts/{id}/voices/{id}	voice	–	GET	PUT		DELETE

Tabelle 3.1: Struktur der HTTP API

taxonomy Die `taxonomy` Ressource enthält eine Taxonomie. Dazu gehört der Name der Taxonomie, der Typ der Taxonomie (`list` oder `tree`) sowie die Einträge der Taxonomie.

taxonomy-item Diese Ressource entpricht einem Eintrag einer Taxonomie. Ein Taxonomieeintrag hat dabei einen Namen, eventuell eine genauere Beschreibung und je nach Typ der Taxonomie eine Liste an untergeordneten Taxonomieeinträgen.

person Diese Ressource beschreibt eine einzelne Person.

opus Die `opus` Ressource beinhaltet ein Werk.

part Die `part` Ressource entspricht einem Werkausschnitt.

subpart Die Ressource `subpart` enthält einen Teilwerkausschnitt.

voice Diese Ressource enthält eine Stimme eines Teilwerkausschnitts.

Diese Ressourcen müssen in der API eindeutig adressiert werden können. Dazu werden URLs verwendet, die jeweils für die einzelne Ressource parametrisiert sind. Für die Datenbankobjekte bietet sich dabei der Primary Key der Tabelle als Parameter der URL an. Bei den Taxonomien wird, ähnlich wie beim Import der Taxonomien, der normalisierte Klassenname als Parameter für die Taxonomie verwendet. Dadurch kann die API eine große Menge an Taxonomien bereitstellen, ohne umstrukturiert werden zu müssen, da nicht für jede Taxonomie eine feste URL angelegt werden muss.

In der Tabelle 3.1 sind alle URLs zu den Ressourcen aufgezählt. Zu jeder URL sind auch die unterstützten Methoden aus dem HTTP 1.1 Standard [IETF14] angegeben. Mit der `GET` Methode lässt sich eine Ressource abfragen. Die `PUT` Methode wird genutzt, um eine Ressource zu aktualisieren. Um eine Ressource anzulegen, wird die `POST` Methode genutzt, und zum Löschen einer Ressource, wird die `DELETE` Methode verwendet. Steht eine Ressource in eckigen Klammern, wie zum Beispiel `[taxonomy]`, dann ist damit eine Liste, deren Einträge der Ressource in den Klammern entsprechen, gemeint. Dabei werden die Ressourcen zu einer URL nach `GET` und `PUT` sowie nach `POST` Ressourcen unterschieden. Die unter `POST` angegebenen Ressourcen gelten dabei ausschließlich für die `POST` Methode.

Trotz der eindeutig erkennbaren Hierarchie von Werk, Werkausschnitt, Teilwerkausschnitt und Stimme in Abbildung 3.4 sind die Ressourcen der API nicht mit entsprechend hierarchisch aufgebauten URLs adressiert. Die Stimmen wären ansonsten unter `/opuses/{id}/parts/{id}/subparts/{id}/voices/{id}` zu finden. Während bei dieser Lösung die Hierarchie auch in den URLs zu sehen ist, befinden sich oft genutzte Ressourcen in einer tieferen Ebene der Hierarchie. Um diese leichter zugänglich zu machen, wurde die Hierarchie nicht in den URLs übernommen. Dies wurde auch erst durch die Entscheidung keine zusammengesetzten Primary Keys in der Datenbank zu verwenden ermöglicht. Die Hierarchie bleibt aber in den URLs teilweise erhalten, da die Ressourcen zu Werkausschnitt, Teilwerkausschnitt und Stimme jeweils unter den Ressourcen zu Werk, Werkausschnitt und Teilwerkausschnitt angelegt werden. Deshalb unterstützt zum Beispiel die URL `/parts` nicht die `POST` Methode.

3.3.2 JSON + HAL

In der API wird JSON als Format für den Austausch von Ressourcen verwendet. Der JSON Standard bietet eine einfache und leicht zu lesende Darstellung von Objekten. Anders als zum Beispiel XML bietet JSON jedoch keine Möglichkeit, Links zu anderen Ressourcen besonders zu behandeln. Diese Links können in JSON nur in Textform gespeichert werden. Will man aber vermeiden, dass die verwendete URL Struktur dem Client im Voraus bekannt ist (ein Indikator für eine starke Kopplung zwischen Client und Server), dann muss die API von der ersten URL aus vollständig entdeckbar sein. Dazu müssen in jeder Ressource die notwendigen weiterführenden Ressourcen in Form von Links auf diese Ressourcen eingebettet werden. Genau für diesen Einsatzzweck ist der Standard JSON + HAL [Kel16] gedacht. Dieser Standard befindet sich aktuell noch in der Entwurfsphase. Trotzdem ist es sinnvoll, den Standard einzusetzen, da dieser recht ausgereift wirkt und es bereits viele Libraries gibt, die den Standard umsetzen.

In dem JSON + HAL Standard sind dabei zwei reservierte Felder (`_links` und `_embedded`) definiert. In dem `_links` Feld sind dabei neben dem `self` link, der auf die eigene Ressource zeigt, auch alle weiterführenden Links enthalten. Diese Links können dabei auch als URL Templates vorliegen. In dem `_embedded` Feld sind die Ressourcen enthalten, von denen anzunehmen ist, dass sie von dem Client direkt gebraucht werden. Da diese Ressourcen in einem extra Feld gekapselt vorliegen, können sie leicht von der eigentlichen Ressource unterschieden werden. Damit wird es dem Client ermöglicht die eingebetteten Ressourcen in einem Cache abzulegen [Kel16]. Falls die Ressource vom Client gebraucht wird, so wird sie im Cache unter dem `self` Link der Ressource gefunden. Dieser Cache muss allerdings von der Anwendung implementiert werden, da die üblichen Browser JSON + HAL nicht selbstständig interpretieren können.

Das `_links` Feld aus dem JSON + HAL Standard bietet eine standardisierte Möglichkeit Links in einem JSON Objekt abzubilden. Dadurch wird es dem Client ermöglicht die URLs, die von der API benutzt werden, zu entdecken, indem er den Links in diesem Feld folgt. Ein Client, der den Standard JSON + HAL jedoch nicht implementiert, kann das `_links` Feld einfach ignorieren und wird nicht davon beeinflusst. Anders verhält es sich mit dem `_embedded` Feld. Ein Client der JSON + HAL nicht implementiert muss entweder für jede eigentlich mitgelieferte Ressource eine neue Anfrage an die API formulieren, oder das `_embedded` Feld, dessen Inhalt sich laut dem Standard jederzeit ändern kann, parsen. Letzteres würde bedeuten, dass der Client gezwungen wird Teile des JSON + HAL Standards zu implementieren. Das Ziel, welches mit dem Standard erreicht werden soll, nämlich die Traversierbarkeit der API mittels Links in den Ressourcen, wird allein durch das `_links` Feld erreicht. Unter Absprache mit der Weboberfläche wurde deshalb nur das `_links` Feld aus dem JSON + HAL Standard umgesetzt. Dies erlaubt dem Client einen einfacheren Umgang mit der API.

3.3.3 Die Taxonomie Ressource

Die URL zu den Taxonomie Ressourcen enthält neben dem Namen der Taxonomie auch den Typ der Taxonomie. Dieser ist eigentlich nicht notwendig um die Taxonomie eindeutig zu identifizieren. Dazu reicht der angegebene Name aus. Der Typ der Taxonomie steht trotzdem in der URL, weil er bei der Dokumentation der Ressourcen die Unterscheidung zwischen den zwei Taxonomietypen ermöglicht. Die beiden Taxonomietypen unterscheiden sich in dem Feld `items`, dass die Einträge der Taxonomie enthält. Für Listentaxonomien ist in diesem Feld eine Liste von Taxonomieeinträgen gespeichert. Bei Baumtaxonomien befindet sich in `items` direkt der Eintrag, der dem Wurzelknoten des Baums entspricht. Die OpenAPI Spezifikation Version 2 [Ope14] erlaubt sowohl Vererbung von Modellen, als auch in eingeschränkter Form Polymorphie, letztere wird aber nur innerhalb eines Objektes erlaubt und funktioniert nicht direkt für eine Ressource.

Da die API die Schnittstelle zwischen der Weboberfläche und dem Backend darstellt, sollte sie ausführlich dokumentiert sein. Die bevorzugte Lösung wäre eine URL `/taxonomies/{taxonomy}` ohne den Datentyp, die mit einem polymorphen Rückgabewert dokumentiert ist. Genau das ist aber mit der OpenAPI Spezifikation nicht möglich.

Um dieses Problem bei der Dokumentation zu umgehen bieten sich mehrere Ansätze an. Da sich die zwei Taxonomietypen im Wesentlichen nur in dem `items` Feld unterscheiden, ist es möglich die Taxonomietypen einander anzugleichen und mit dem selben Schema zu beschreiben. Dazu würde bei Baumtaxonomien der Wurzelknoten nicht direkt in `items` gespeichert, sondern in einer Liste in `items`. Alle anderen Felder, die entweder für Listentaxonomien oder für Baumtaxonomien gelten, werden in dem Schema als optional markiert. Mit diesem Schema kann dann die URL `/taxonomies/{taxonomy}` dokumentiert werden. Dabei geht aber die klare Trennung zwischen Listentaxonomien und Baumtaxonomien verloren. Außerdem ist nicht eindeutig, wie mit Baumtaxonomien zu verfahren ist, wenn diese mehrere Einträge im `items` Feld haben. Dieser Fall ist aber in dem angegeben Schema eingeschlossen.

Um die Polymorphie mit den in der OpenAPI Spezifikation verfügbaren Mitteln zu dokumentieren, muss die Taxonomie in ein zusätzliches Hilfsobjekt verpackt werden. Dieses könnte in JSON Notation in etwa so aussehen: `{"data":null}`. Anstelle des `null` Werts würde eine Taxonomie eingefügt. In diesem Fall lässt sich das `data` Feld als ein polymorphes Feld dokumentieren. Dieser

Umschlag hat allerdings keine weitere Funktion. Falls der Umschlag nur bei den Taxonomien eingesetzt wird, werden die Rückgabewerte der API inkonsistent. Außerdem muss der Client alle verpackten Objekte vor Benutzung wieder entpacken.

Eine weitere Möglichkeit, eine klare Dokumentation der beiden Taxonomietypen zu ermöglichen, ohne die Taxonomien in einen Umschlag zu verpacken, ist es den Typ der Taxonomie in die URL mit aufzunehmen. Dadurch lassen sich die parametrisierten URLs für Listentaxonomien und Baumtaxonomien klar unterscheiden und damit auch einzeln dokumentieren. Durch die Unterscheidung nach Typ in der URL sind auch die jeweils unterschiedlich platzierten `POST` Methoden zum Erstellen neuer Taxonomieeinträge möglich. Bei einer Listentaxonomie sollten keine Taxonomieeinträge an einem Taxonomieeintrag angehängt werden können. Umgekehrt dürfen bei einer Baumtaxonomie auf der Ebene des Wurzelknotens keine weiteren Taxonomieeinträge angelegt werden. Mit den Methoden zum Anlegen, Ändern und Löschen von Taxonomieeinträgen ist zumindest in der API die Grundlage geschaffen, die später von einen Taxonomieeditor in der Weboberfläche benutzt werden kann. Unter Berücksichtigung der Wiederverwendbarkeit (Abschnitt 2.2) wird diese Grundlage in naher Zukunft hilfreich sein.

3.3.4 `null` Werte

Der Datentyp `null` ist in der OpenAPI Spezifikation [Ope14] nicht vorhanden. Die JSON Objekte, die von der API erstellt oder verarbeitet werden, dürfen deshalb keine `null` Werte in den Feldern enthalten. Die Validierung der Objekte erfolgt zwar mittels JSON Schema, welches den Datentyp `null` unterstützt, es gibt allerdings keine Möglichkeit zur Validierung andere Modelle zu verwenden als zur Dokumentation. Deshalb müssen alle Felder mit Standardwerten belegt werden.

3.3.5 User API

Wie in Anforderung 3 beschrieben, sollen sich die Benutzer der Weboberfläche mit einem Benutzernamen und Passwort anmelden können. Hierfür müssen die Benutzer im Backend abgespeichert werden. Außerdem muss das Backend eine Möglichkeit für die Weboberfläche anbieten Benutzername und Passwort zu verifizieren. Da die Anforderung 3 eine geringere Priorität hat, wurden die API Endpunkte zum Anmelden von Benutzern in eine eigene API ausgelagert. Dadurch kann das Webfrontend zunächst die Standard API ohne Authentifizierung benutzen, bis die Login Methoden der User API vom Webfrontend unterstützt werden. Zu diesem Zeitpunkt kann dann auch die Standard API mit der Authentifizierung durch die User API vor Missbrauch geschützt werden. Die User API bleibt aber unabhängig von der Standard API und kann deshalb auch von nachfolgenden Projekten übernommen werden. Dies ist interessant, weil die sicherheitskritischen Entscheidungen zum Speichern der Passwörter oder dem Session Handling nicht ohne entsprechendes Hintergrundwissen getätigt werden sollten.

Die Passwörter werden in der Datenbank als bcrypt Hash gespeichert. Dies ist eine aktuell noch nicht gebrochene Hashfunktion für Passwörter. In dem MUSE4Music Backend werden zwar keine personenbezogenen Daten (zu lebenden Personen) gespeichert, die verwendeten Passwörter könnten aber auch von den Benutzern in anderen Diensten verwendet worden sein. In diesem Fall könnte ein Diebstahl der Passwörter ernsthafte Folgen für die betreffende Person haben.

Deshalb sollte auch bei Anwendungen, deren Sicherheitsanforderungen eher gering sind, darauf geachtet werden, dass die Passwörter der Benutzer nicht trivial geknackt werden können, sollte die Datenbank kompromittiert worden sein.

Damit ein Benutzer nicht für jede neue Anfrage an die API sein Passwort mitschicken muss, bekommt er nach der erfolgreichen Überprüfung des Passworts einen JSON Web Token [JMB+15]. Dieser kann von dem Benutzer gelesen, aber nur vom Aussteller verändert werden. Dadurch kann der Benutzer für alle folgenden Interaktionen mit der API diesen Token angeben. Der State der Session wird also vom Benutzer verwaltet. Der Server braucht die Tokens nicht zu speichern, da sie vom Benutzer nicht verändert werden können. Er bleibt stateless. Die Tokens haben ein festes Ablaufdatum und können dem Benutzer zugeordnet werden. Dadurch wird der mögliche Schaden minimiert, falls ein Token abgegriffen wird. Durch die Zuordnung zu einem Benutzer ist es auch möglich den Benutzern unterschiedliche Rechte zum Bearbeiten der Daten zu geben.

Diese die Sicherheit betreffenden Entscheidungen sollten bei einem späteren Einsatz des Tools überprüft werden, da jederzeit ein Angriff auf die verwendeten kryptografischen Funktionen entdeckt werden kann.

4 Zusammenfassung und Ausblick

In dieser Arbeit wurden zunächst die Anforderungen an das Backend für MUSE4Music erhoben. Neben den funktionalen Anforderungen, die direkt vom Kunden formuliert worden sind, wurden auch die nicht funktionalen Anforderungen, die sich teilweise aus den Begebenheiten der Software ableiten lassen, genannt. In dem gegebenen Zeitraum konnten dabei aber nicht alle Anforderungen vollständig umgesetzt werden. Die Anmeldung mit Benutzername und Passwort (Abschnitt 2.1.1) wird vom Backend mit der User API unterstützt, aber bei der Benutzung der Standard API wird zum jetzigen Zeitpunkt keine Anmeldung vorausgesetzt Die Indikatoren für Objekte, die zum Review freigegeben sind (Abschnitt 2.1.1), wurden ebenfalls noch nicht implementiert. Zu den Spezifikationen (Abschnitt 2.1.1) besteht noch Klärungsbedarf mit dem Kunden. Außerdem ist diese Anforderung in dem aktuellem Umfang mit einem großen Implementierungsaufwand verbunden, weshalb sie zunächst noch zurückgehalten wird.

Im weiteren Verlauf der Arbeit wurde die Architektur und die Implementierung des MUSE4Music Backends behandelt. Dabei wurde insbesondere auf die nicht funktionale Anforderung der Wiederverwendbarkeit eingegangen. So wurde speziell beim Import der Taxonomien darauf geachtet, ein Dateiformat auszuwählen, dass sowohl Informatikern als auch Forschern anderer Disziplinen zugänglich ist. Auch wurde durch den Einsatz mehrerer Mixin Klassen der Quellcode übersichtlicher gestaltet. Bei dem Design der Datenbank wurde neben der Gesamtstruktur speziell auf die Darstellung der Taxonomien und deren Relationen mit den Datentabellen geachtet. Anschließend wurde der Aufbau der API näher beschrieben. Dazu wurden zunächst die einzelnen Ressourcen identifiziert und anschließend auf eine passende URL Hierarchie abgebildet. Insbesondere wurde auch bei der API auf die Taxonomie Ressource eingegangen.

Ausblick

Während die Anwendung im Einsatz getestet wird, sollten größere Performanzprobleme erkannt und behoben werden. Weiterhin steht auch die Umsetzung der nicht implementierten funktionalen Anforderungen an. Auch ist es nicht auszuschließen, dass das Datenmodell nach einer gewissen Zeit im Einsatz noch einmal etwas überarbeitet wird. Auf lange Sicht steht auch die Auswertung der gesammelten Daten an.

Das Backend wurde von Anfang an mit dem Ziel geplant, als Grundlage für ähnliche Projekte den Quellcode bereitstellen zu können. Deshalb wurde die Anforderung der Wiederverwendbarkeit in den Vordergrund gestellt. Die Vorteile, die die Mixin Klassen beim Implementieren der Datenstruktur bieten, haben sich dabei schon während der Bachelorarbeit bemerkbar gemacht. Diese Mixin Klassen sind für ähnliche Projekte ebenfalls einsetzbar. Darüber hinaus ist es denkbar, eine Hilfsanwendung zu entwickeln, die bei der Erstellung der Datenbankmodelle unterstützt.

Literaturverzeichnis

[AIS77] C. Alexander, S. Ishikawa, M. Silverstein. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, Aug. 1977 (zitiert auf S. 7).

[Bar17] J. Barzen. *MUSE – Muster Suchen und Erkennen*. 2017. URL: www.iaas.uni-stuttgart.de/forschung/projects/MUSE/ (zitiert auf S. 8, 11, 16).

[BBE+16] J. Barzen, U. Breitenbücher, L. Eusterbrock, M. Falkenthal, F. Hentschel, F. Leymann. „The vision for MUSE4Music. Applying the MUSE method in musicology“. In: *Computer Science - Research and Development* (Nov. 2016), S. 1–6. DOI: 10.1007/s00450-016-0336-1. URL: http://www.iaas.uni-stuttgart.de/RUS-data/ART-2016-17%20-%20The%20vision%20for%20MUSE4Music.pdf (zitiert auf S. 3, 7).

[EBH17] L. Eusterbrock, J. Barzen, F. Hentschel. *Eine Ontologie symphonischer Musik des 19. Jahrhunderts*. Techn. Ber. Universität Stuttgart – Fakultät Informatik, Elektrotechnik und Informationstechnik, 2017. URL: ftp://ftp.informatik.uni-stuttgart.de/pub/library/ncstrl.ustuttgart_fi/TR-2017-02/TR-2017-02.pdf (zitiert auf S. 9).

[Fie00] R. T. Fielding. „Architectural Styles and the Design of Network-based Software Architectures.“ In: *Doctoral dissertation, University of California, Irvine* (2000). URL: http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm (zitiert auf S. 28).

[GZC13] F. Galiegue, K. Zyp, G. Court. *JSON Schema: core definitions and terminology*. 2013. URL: json-schema.org/draft-04/json-schema-core.html (zitiert auf S. 21).

[IETF14] Internet Engineering Task Force. *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content*. 2014. URL: https://tools.ietf.org/html/rfc7231#section-4.3 (zitiert auf S. 30).

[JMB+15] M. Jones, Microsoft, J. Bradley, P. Identity, N. Sakimura. *JSON Web Token (JWT)*. 2015. URL: tools.ietf.org/html/rfc7519 (zitiert auf S. 33).

[Kel16] M. Kelly. *JSON Hypertext Application Language – draft-kelly-json-hal-08 (work in progress)*. 2016. URL: tools.ietf.org/html/draft-kelly-json-hal-08 (zitiert auf S. 30).

[LRS15] B. Lahres, G. Raýman, S. Strich. „Mixin-Module statt Mehrfachvererbung“. In: *Objektorientierte Programmierung*. 3. Aufl. Rheinwerk Computing, 2015. Kap. 5.4.3. ISBN: 978-3-8362-3514-3 (zitiert auf S. 20).

[noi] noirbizarre. *Flask RestPlus*. URL: github.com/noirbizarre/flask-restplus (zitiert auf S. 16).

[Ope14] Open API Initiative. 2014. URL: github.com/OAI/OpenAPI-Specification/blob/master/versions/2.0.md (zitiert auf S. 16, 21, 31, 32).

[Pet04] T. Peters. *PEP 20 – The Zen of Python*. 2004. URL: www.python.org/dev/peps/pep-0020/ (zitiert auf S. 15).

[Ron] A. Ronacher. *Flask*. URL: flask.pocoo.org (zitiert auf S. 15).

[SQL] SQLAlchemy. *The Python SQL Toolkit and Object Relational Mapper*. URL: www.sqlalchemy.org (zitiert auf S. 16).

Alle URLs wurden zuletzt am 19. 11. 2017 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift