HAW HAMBURG

Bachelorarbeit

Erim Medi

Entwicklung einer Full Stack Medienausleih-Applikation für eine Stadtteilschule mittels Spring-Boot und Vue

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department Computer Science

Erim Medi

Entwicklung einer Full Stack Medienausleih-Applikation für eine Stadtteilschule mittels Spring-Boot und Vue

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang *Bachelor of Science Angewandte Informatik*
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Stefan Sarstedt
Zweitgutachter: Prof. Dr. Ulrike Steffens

Eingereicht am: 26. September 2023

Erim Medi

Thema der Arbeit

Entwicklung einer Full Stack Medienausleih-Applikation für eine Stadtteilschule mittels Spring-Boot und Vue

Stichworte

Spring, Spring-Boot, Spring-Security, JSON Web Token, REST API, Vue,js, Vue-Router, Vuex, Java, JavaScript, Tailwind

Kurzzusammenfassung

In dieser Arbeit wird eine Full Stack Applikation entwickelt, welche im Kontext einer Stadtteilschule das Ziel hat, die Inventarisierung von physischen Medien (Bücher, iPads, Laptops etc.) und deren Ausleihe und Rücknahme zu gewährleisten. Die Applikation ist eine klassische Client-Server Architektur. Das Backend wird mittels Spring-Boot realisiert und die entstehende REST-API mit Spring-Security abgesichert. Für das Frontend wird Vue.js als Frontend-Framework und Tailwind CSS als Open-Source-CSS-Framework benutzt. Die Arbeit beschreibt die Anforderungsanalyse, Spezifikation, Entwurf sowie die Implementierung.

Erim Medi

Title of Thesis

Development of a full stack media lending application for a district school using Spring-Boot and Vue

Keywords

Spring, Spring-Boot, Spring-Security, JSON Web Token, REST API, Vue,js, Vue-Router, Vuex, Java, JavaScript, Tailwind

Abstract

In this work, a full stack application is developed, which in the context of a neighborhood school aims to ensure the inventory management of physical media (books, iPads, laptops,

etc.) and their lending and return. The application has a classic client-server architecture. The backend is implemented using Spring Boot and the resulting REST API is secured with Spring Security. Vue.js is used as a frontend framework and Tailwind CSS as an open source CSS framework for the frontend. The work describes the requirements analysis, specification, design and implementation.

Inhaltsverzeichnis

Abbildungsverzeichnis **viii**

Tabellenverzeichnis **x**

1 Einleitung **1**
1.1 Motivation . . . 1
1.2 Ziel . . . 2
1.3 Aufbau der Arbeit . . . 2

2 Grundlagen **3**
2.1 Spring . . . 3
2.1.1 Dependency Injection . . . 4
2.1.2 Spring-Container . . . 4
2.1.3 Spring Bean . . . 4
2.1.4 Aspektorientierte Programmierung (AOP) . . . 5
2.2 Spring-Boot . . . 5
2.2.1 Spring-Boot-Starters . . . 5
2.2.2 Datenbank: Spring Data JPA & Hibernate . . . 6
2.3 Vue.js . . . 6
2.3.1 Vue-Router . . . 7
2.3.2 Tailwind CSS . . . 8

3 Analyse **9**
3.1 Personas . . . 9
3.2 User Roles . . . 10
3.3 Anforderungen - User Storys & Use Cases . . . 12
3.3.1 User Storys . . . 13

4 Spezifikation **15**
4.1 Entity Relationship Model . . . 15
4.1.1 Beschreibung des Models . . . 16
4.1.2 Tabellen . . . 16
4.2 Use-Cases . . . 17
4.2.1 Use-Case Diagramm . . . 17
4.2.2 Anwendungsfall 1: Login . . . 18
4.2.3 Anwendungsfall 2: Ausgeliehene Medien lesen . . . 18
4.2.4 Anwendungsfall 3: Medien Ausleihe/ Rücknahme . . . 19
4.2.5 Anwendungsfall 4: Inventarisierung . . . 21
4.2.6 Anwendungsfall 6: Medien löschen . . . 23
4.2.7 Anwendungsfall 7: User-Import . . . 24
4.2.8 Anwendungsfall 8: Rechteverwaltung . . . 25

5 Entwurf **26**
5.1 Architektur . . . 26
5.2 Backend . . . 27
5.3 Frontend . . . 27

6 Implementierung **30**
6.1 Backend . . . 30
6.1.1 Spring-Boot aufsetzen . . . 30
6.1.2 Configs . . . 31
6.1.3 Entities . . . 32
6.1.4 Controller . . . 33
6.1.5 Repositories . . . 35
6.1.6 Services . . . 37
6.1.7 Endpoints . . . 39
6.2 Spring-Security . . . 42
6.2.1 Config . . . 44
6.2.2 JWT . . . 47
6.2.3 Erwähnungen . . . 49
6.3 Frontend . . . 50
6.3.1 Vue-Projekt aufsetzen . . . 50
6.3.2 Vue-Router . . . 51
6.3.3 View Beispiel: LoanView . . . 51

6.3.4 Vuex Store 54

7 Evaluation **59**
7.1 Tests 59
7.2 Nutzerbefragung 59
7.3 Fazit 61

8 Fazit / Zusammenfassung **64**
8.1 Retrospektive 65
8.2 Ausblick 66
8.2.1 Deployment 66
8.2.2 Datenbank 66
8.2.3 JWT Refresh-Token 66
8.2.4 Account zurücksetzen via Mail 67
8.2.5 Problem mit der User-Verwaltung 67

Literaturverzeichnis **68**

Listings **70**

A Anhang **71**
A.1 Tabellen 71
A.2 Abbildungen 73

Glossar **74**
Selbstständigkeitserklärung 75

Abbildungsverzeichnis

3.1 Empfohlene Aspkete einer Personabeschreibung nach Ian Sommerville. Abbildung wurde nachgebaut[19]. 10
3.2 Persona-Lisa-Bibliothekarin. Eigene Abbildung. 10
3.3 Kategorisierung der User Roles 12
3.4 Beispiel Use Case [16] 14

4.1 Entity Relationship Model 16
4.2 Use-Case Diagramm 18
4.3 Wireframe HomeView - Auflistung der aktuell ausgeliehenen Medien . . . 19
4.4 Wireframe - Suche Nutzer im System für Ausleih-/Rücknahmevorgang . . 20
4.5 Wireframe AusleihView - Medien-ID einscannen zum Ausleihen/zur Rücknahme 20
4.6 Ausleihprozess 21
4.7 Inventarisierungsprozess 21
4.8 Wireframe - Medienreihen Übersicht 22
4.9 Wireframe - Medium in MedienReihe aufnehmen 22
4.10 Wireframe - Neue Medienreihe erstellen. Es öffnet sich ein transparentes Fenster, indem sämtliche benötigten Informationen zur neuen Medienreihe übergeben werden. 23
4.11 Wireframe - Medium löschen. 24

5.1 Drei-Schichten-Architektur der Applikation 26
5.2 Fluss-Architektur einer Spring-MVC Applikation 29
5.3 Architektur des Frontend mit Vue. Übernommen und angepasst von bezkoder 29

6.1 Pagination - Postman Response Beispiel 37
6.2 Swagger-UI 40

6.3 Spring-Security-Fluss-Architektur. Diagramm übernommen und angepasst von spring-security-tutorial-jwt [17] 42
6.4 Vue-Dependencies 51
6.5 Router-Snippet 52
6.6 LoanView Screenshot 53
6.7 index.js File (Vuex-Store) 55

7.1 Wireframe - Kategorien in der Inventarübersicht 62

A.1 Persona-Katarina-Lehrerin. Eigene Abbildung. 73
A.2 Persona-Gunnar-Admin. Eigene Abbildung. 73

Tabellenverzeichnis

3.1 User Roles und ihre Rechte 12

4.1 Beispiel Tabelle MediaSeries 17
4.2 Tabelle für ein Feld annotiert mit @ElementCollection 17

6.1 API-Endpoints 41
6.2 Vorinitialisierte Nutzer zum Testen 41

A.1 Tabelle MediaSeries 71
A.2 Tabelle Medium 71
A.3 Tabelle LoanHistory 72
A.4 Tabelle Borrower 72
A.5 Tabelle für ein Feld annotiert mit @ElementCollection 72

1 Einleitung

In diesem Kapitel wird die Motivation hinter der Arbeit dargestellt. Außerdem wir das Ziel und der Aufbau der Arbeit erklärt.

1.1 Motivation

Die Stadtteilschule Mümmelmannsberg ist eine große Schule mit über 1000 Schülern. Als eines der ersten Schulen in Hamburg ist es ihr Ziel eine „iPad-Schule“ zu werden, sodass jedem Schüler und jeder Schülerin, aber auch jedem Lehrer und Lehrerin ein iPad bereitgestellt wird. Dieses Ziel wurde vor allem in der Corona-Pandemie vorangetrieben. Aktuell besitzt die Schule ungefähr 1.300 iPads. Die Verwaltung der iPads, Installationen von Apps, Restriktionen, konfigurieren des WLAN-Zugangs etc., geschieht über ein Mobile-Device-Management (MDM). Das MDM ermöglicht jedoch keine Zuweisung, welches Gerät, welchem Schüler bzw. Lehrer gehört. Das aktuelle Bibliothekssystem ermöglicht zwar eine Inventarisierung anhand einer internen nummerischen Id, die auf sämtliche Medien aufgeklebt wird, doch lässt sich die Information der Seriennummer nicht hinzufügen. Daher findet die Zuweisung mittels einer (riesigen) Exel-Datei statt, die eine Verknüpfung von Geräteseriennummer und Informationen des Leihenden (Name, Vorname, Klasse, etc.) herstellt. Hier geschahen im Verlaufe des Prozesses immer wieder Zuweisungsfehler, sodass eine mühsame Detektivarbeit im Nachhinein nötig war. Außerdem kann nicht mit sofortiger Gewissheit gesagt werden, ob der Ausleiher gerade *sein* Gerät zurückgibt oder das eines anderen, ohne in die Exel-Liste zu schauen, die eventuell Fehler hat, was dazu führt, den Ausleihvertrag zu suchen und die Seriennummer zu überprüfen.

Neben den knapp 1.300 iPads besitzt die Schule noch weitere Medien, die ausgeliehen werden können.

- Hochwertige iPad-Tastaturhüllen (Nur für die Oberstufen)
- Normale Hüllen
- HDMI-Adapter für iPads zum Anbinden an Smartboards in den Klassenräumen
- Apple Pencils
- Laptops

1.2 Ziel

Ziel der Arbeit ist es ein User freundliches Medienausleihsystem zu entwickeln, das neben Büchern auch andere Medien wie iPads inventarisieren und deren Ausleihe und Rückgabe gewährleisten kann. Das System soll im Stande sein für technische Geräte die Seriennummer mit aufzunehmen, sodass auf die bisherige Exel-Datei verzichtet werden kann.

1.3 Aufbau der Arbeit

Die Arbeit ist in acht Kapitel gegliedert. Im ersten Kapitel wurden Motivation und Ziel der Arbeit vorgestellt. In Kapitel 2 werden die zwei Frameworks Spring-Boot und Vue.js vorgestellt, die für die Implementierung der Applikation genutzt wurden. Das dritte Kapitel befasst sich mit der Anforderungsanalyse und das vierte mit der Spezifikation der Anforderungen. Kapitel 5 enthält den Entwurf der Applikation und mit Kapitel 6 folgt die Implementierung. In Kapitel 7 werden die Ergebnisse der Nutzerumfrage vorgestellt. Zuletzt wird in Kapitel 8 ein Fazit bzw. Zusammenfassung der Arbeit gegeben.

2 Grundlagen

In diesem Kapitel werden die zwei Frameworks näher betrachtet, die für die Implementierung der Applikation zum Einsatz kommen. Das sind Spring-Boot für das Backend und Vue.js für das Frontend.

2.1 Spring

Spring wurde 2002 als Idee vorgestellt, mit dem Ziel, die Entwicklung mit Java zu vereinfachen und Programmierpraktiken zu fördern. Ein Jahr später wurde es unter dem Namen Spring-Framework als Open Source veröffentlicht [18]. Spring bietet Infrastruktur auf Anwendungsebene an und übernimmt die Konfiguration, sodass die Entwickler ihren Fokus ganz auf die Implementierung der Geschäftslogik richten können [3, 18]. Kernfunktionen von Spring sind;

- Dependency Injection
- MVC basierte Webanwendungen und RESTful Webservices
- Grundlagen für JDBC, JPA etc.
- Aspektorientierte Programmierung (AOP)

Hierbei sind Dependency Injection und AOP zwei wichtige Grundlagen für die Infrasturktur.

2.1.1 Dependency Injection

Dependency Injection, früher bekannt als *Inversion of Controle* (IoC), ist ein Paradigma, das bei der Verdrahtung von Objekten zu komplexen Objektnetzen zum Einsatz kommt. Das Paradigma gewährleistet eine möglichst lose Koppelung und eine hohe Kohäsion für Objekte. Die Abhängigkeit eines Objektes zu einem anderen wird hierbei nicht hart codiert, sonder per Konfigurationsinformationen zur Laufzeit von außen in die Objekte per Konstruktor oder Setter injiziert [18]. Im Spring-Kontext wird dies *Auto-Wiring* genannt. Traditionell kommen hier XML-Dateien zum Einsatz, welche die Konfigurationsmetadaten enthalten.

2.1.2 Spring-Container

Der Spring-Container ist für die Verwaltung, Instanziierung, und Konfiguration der Beans verantwortlich. Welche Beans er instanziiert und welche Abhängigkeiten jene Beans haben, liest der Container aus Konfigurationsmetadaten. Entsprechend werden die Abhängigkeiten injiziert [4]. Der Container in Spring ist eine Implementation des Interfaces *Application Context*. Es stehen unterschiedliche Implementationen des Application Context zur Verfügung, welche je eine andere Möglichkeit bereitstellen, Beans zu instanziieren [18].

- **XML-Basiert:** Beans werden explizit in XML-Dateien gelistet und verdrahtet (traditionelle Art).
- **Annotationen:** Klassen werden mit Annotationen versehen und über einen *ComponentScan* gesucht.
- **Java-Basiert:** Beans werden in speziellen Konfigurationsklassen instanziiert.

2.1.3 Spring Bean

Ein Bean ist ein Objekt, welche vom Spring-Container instanziiert und konfiguriert wurde [6]. Ein Bean hat einen Lebenszyklus, der vom Spring-Container verwaltet wird. Per Default werden Beans als Singleton instanziiert. [18]

2.1.4 Aspektorientierte Programmierung (AOP)

AOP ist das zweite Kernfeature von Spring. AOP betrachtet Aspekte als *querschnittliche Themen oder crosscutting concerns* [18]. Crosscutting Concerns sind Anforderungen, die über die gesamte Applikation verteilt sind und sich nur schwer kapseln lassen wie bspw. Logging, Caching oder Security. Das Ziel, welches verfolgt wird, ist jene Aspekte losgelöst von den funktionalen Anforderungen, also der Geschäftslogik, zu halten.

2.2 Spring-Boot

Spring-Boot ist eine Erweiterung des Spring-Frameworks. Auf der offiziellen Webseite von Spring heißt es: *„Spring Boot makes it easy to create stand-alone, production-grade Spring based Applications that you can **just run**"*. Was bedeutet hier *just run*? Kurz gesagt bedeutet es, dass Spring-Boot die komplexe Konfiguration einer Spring-Anwendung übernimmt. Während Spring sich um die Kernfeatures wie Dependency Injection und AOP dreht, hat Spring-Boot den Fokus auf **Auto-Configuration**, mit dem Ziel, Boilerplate-Konfigurationen zu eliminieren wie bspw. die XML-Dateien für die Instanziierung der Beans, indem standardmäßig @ComponentScan aktiviert ist, sodass Spring-Boot sämtliche Klassen annotiert mit @Component oder einer ableitenden Annotation wie @Controller selbstständig verwaltet. Auf diese Weise reicht es aus, eine Klasse mit @Component zu annotierten statt eine XML-Configuration vorzunehmen.

2.2.1 Spring-Boot-Starters

Spring-Boot bietet diverse sogenannte *Starter*. Solche Starter werden als Abhängigkeiten deklariert. Sie bündeln sämtliche weitere Abhängigkeiten und Konfigurationen zusammen. Wird bspw. die Abhängigkeit spring-boot-starter-web deklariert, konfiguriert Spring-Boot automatisch Spring MVC, die Infrastruktur für die Annotation @Controller und einen Tomcat-Server. Das ist der große Vorteil von Spring-Boot, Auto-Configuration. In einer reinen Spring-Applikation ohne Spring-Boot bedarf es einer eigenständigen Konfiguration. Das Zusammenspiel der etwa 50+ spring-boot-starters und annotationsbasierten Beans Initialisierung ermöglicht es den Entwicklern, eine Spring-Applikation *einfach auszuführen.*

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

Listing 2.1: spring-boot-starter-web dependency aus der pom.xml

2.2.2 Datenbank: Spring Data JPA & Hibernate

Für das Backend ist der Starter *spring-boot-starter-data-jpa* hinzugefügt worden, sodass Spring Data JPA und Hibernate als JPA-Implementierung bereitgestellt wird. JPA (Jakarta Persistence API früher Java Persistence API) ist die allgemeine Persistenz-API für Java. Hibernate ist unter anderem ein Open-Source-Object-Relational-Mapping Framework. Es bildet Java-Objekte auf Tabelleneinträge und umgekehrt ab. Spring Data JPA baut auf Hibernate auf und vereinfacht die Implementierung der Repositories, indem es Boilerplate-Code durch Abstraktion reduziert. Mehr dazu im Abschnitt Repositories 6.1.5. Spring-Boot ermöglicht es, die Datenbank leicht auszutauschen. Dazu muss in der pom.xml die benötigte Datenbank-Dependency eingefügt und in der *application.yml* die Datenbankanbindung konfiguriert werden. Die *application.yml* befindet sich unter dem Pfad *src/main/resources/application.yml*.

Es bietet sich aus praktischen Gründen an, für die Entwicklung oder lokalem Testen eine in-Memory H2 Datenbank zu nutzen. Während der Entwicklung wurde jedoch eine PostgresSQL Datenbank erstellt und genutzt, um das Anbinden an eine Datenbank zu erproben, die weit verbreitet ist. Da, wie erwähnt der Austausch der Datenbank einfach ist, wird zum Zeitpunkt der Abgabe auf H2 umgestellt, sodass ein lokales Testen ermöglicht wird.

2.3 Vue.js

Vue ist ein Frontend-JavaScript-Framework zur Erstellung von Webanwendungen und glieder sich unter Angular und React ein. Es baut auf Standard-HTML, CSS und JavaScript auf und bietet ein deklaratives und komponentenbasiertes Programmiermodell. Eine Komponente, genannt Vue-SFC (Single-File-Component), kapselt die gesamte Logik (JavaScript), das Template (HTML) und die Stile (CSS) in eine Datei zusammen. Komponenten können wiederum aus anderen Komponenten bestehen und diese per Import

in sich einbinden. So entstehen Hierarchien bzw. Eltern-Kind Relationen zwischen den Komponenten. Eine beidseitige Kommunikation zwischen Eltern-Kind ist möglich. Vue setzt eine Installation von Node.js voraus.

Auf der offiziellen Seite von Vue ist eine detaillierte Dokumentation zu finden, welche bei der Entwicklung des Frontends dieser Applikation stark einbezogen wurde [12].

```
<template>
    <!-- HTML -->
    <div class="content-text">
        <p>Hello world :) </p>
    </div>
</template>
<script>
    //JavaScript
    //Imports
    import { ref } from 'vue'

    //Properties; koennen von Parent gesetzt werden
    const props = defineProps(['foo'])

    //Variablen
    const count = ref(0)

    //Funktionen
    const function = () =>{
    //Implementation
    }
</script>
<style scoped>
    /* CSS */
    @import '../styles/HomeView.css';
</style>
```

Listing 2.2: Vue-SFC Beispiel

2.3.1 Vue-Router

Vue besitzt kein integriertes Routing, jedoch existiert eine offizielle Routing-Bibliothek Vue-Router. Wird ein neues Vue-Projekt in der Kommandozeile mittes des Befehls

npm create vue@latest initiert, kann Vue-Router mit installiert und konfiguriert werden. Alternativ lässt sich Vue-Router mittels *npm install vue-router@4* nach installieren lassen.

Das Routing ermöglicht es eine Vue-SFC auf eine URL zu mappen. So lässt sich bspw. eine Komponente HomeView.vue auf die Base-URL meine-webseite.de mappen, die immer dann gerendert wird, sobald der Client jene URL ansteuert. Auf diese Art lassen sich verschiedene Pfade erstellen, denen jeweils eine Komponente zugeordnet wird. Solche Komponenten werden *Views* genannt. Technisch unterscheiden sich Views und Komponenten nicht. Praktisch werden Komponenten mehrfach in einem Projekt verwendet, während Views einmalig vorkommen. Die namentliche Unterscheidung sorgt für mehr Klarheit und Struktur [13].

2.3.2 Tailwind CSS

In dieser Arbeit wird Tailwind CSS als Open-Source-CSS-Framework benutzt, ist jedoch für das Verständnis der Arbeit nicht relevant. Es sei gesagt, dass Tailwind den CSS-Part (Cascading Style Sheets) im Frontend erleichtert (subjektive Meinung).

3 Analyse

In diesem Abschnitt der Arbeit wird die Anforderungsanalyse beschrieben. Es werden Personas und User Roles definiert sowie User Storys ermittelt.

3.1 Personas

Um Software möglichst passend für den Kunden zu entwickeln, bedarf es einem gewissen Verständnis für die potenziellen Anwender ganz nach der Frage "Wer sind meine Zielbenutzer?"[19]. Um dieses Verständnis aufzubauen und ein Bild der potenziellen Anwender vor Augen zu haben, werden sogenannte Personas, fiktionale Charaktere, kreiert. Ian Sommerville fasst folgende vier Aspekte zusammen, die eine Persona besitzen sollte - visualisiert in Abbildung 3.1.

- **Personalisierung:** Personas sollen einen Namen haben und persönliche Informationen darstellen.
- **Berufsbezogene Informationen:** Informationen zur Arbeitswelt. Bei einigen Berufen, mit dem der Leser vertraut ist, wäre dies eventuell nicht erforderlich z.b der Lehrer.
- **Bildung:** Der Bildungsstand, das technische Niveau und Erfahrungen sollten beschrieben werden.
- **Relevanz:** Wenn möglich, sollte gesagt werden, warum die Personas Interesse am Produkt haben könnten.

Wie diese Informationen dargestellt werden, ist nicht standardisiert. Manche nutzen kleine Karten, andere Poster, die im Raum aufgehängt werden können und einige nutzen bevorzugt digitale Tools. Neben den erwähnten vier Aspekten sollte eine Persona auch ein Bild besitzen, um den Visualisierungsprozess für den Entwickler zu begünstigen [16], ganz nach dem Sprichwort “Ein Bild sagt mehr als tausend Worte”.

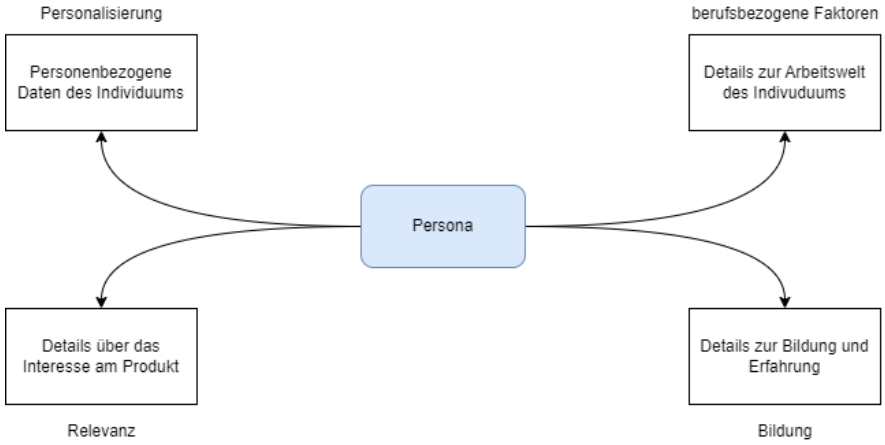

Abbildung 3.1: Empfohlene Aspkete einer Personabeschreibung nach Ian Sommerville. Abbildung wurde nachgebaut[19].

Es sind drei Personas entstanden: Lisa, die Bibliothekarin (Abb. 3.2). Gunnar, der Lehrer & Admin (Anhang A.2). Katarina, die Lehrerin (Anhang A.1).

Abbildung 3.2: Persona-Lisa-Bibliothekarin. Eigene Abbildung.

3.2 User Roles

Die in dieser Arbeit entstehende Applikation wird Benutzer (User) haben, welche mit dem System interagieren werden. Der Begriff User ist sehr allgemein und für die spätere Analyse und Spezifikation, bspw. für User Storys, nicht optimal. Welcher User ist

gemeint, wenn in einer User Story der generische Begriff User benutzt wird? Die User werden sich in dem, wie sie mit der Applikation interagieren, unterscheiden, daher sollten die User spezifiziert werden, sodass verschiedene *User-Roles* definierbar sind. Mike Cohn beschreibt User Roles wie folgt: *„Eine Nutzerrolle ist eine Menge von Attributen, die eine Population von Nutzern und ihre beabsichtigten Interaktionen mit dem System beschreibt."* [16].

Desweiteren nutzt er folgende vier Schritte um User Roles zu identifizieren:

- brainstorm an initial set of user roles
- organize the initial set
- consolidate roles
- refine the roles

Es wird nicht weiter auf die einzelnen Schritte eingegangen, sondern nur das Ergebnis dargestellt.

Das eine Schule Lehrer und Schüler hat steht außer Frage, somit entstehen die ersten zwei User Roles. Irgendjemand muss sich um die Technik an einer Schule kümmern. Dies kann eine externe Person sein oder wie im Falle der Stadtteilschule Mümmelmannsberg eine oder mehrere Lehrkräfte. Entsprechend könnte jene externe Fachkraft oder eine Lehrkraft die Administration der Applikation übernehmen. Selbstverständlich wird ein Bibliothekar bzw. Bibliothekarin benötigt, welche die Ausleihe von Bücher etc. an Schüler und Lehrer gewährleistet . Im Falle der Stadtteilschule Mümmelmannsberg übernimmt dies die Bibliothekarin der Schule. Falls die Bibliothekarin einmal krankheitsbedingt ausfällt oder, um eine Hilfe bereit zu stellen, könnten Lehrkräfte geringere Zugriffsrechte bekommen, sodass jene Lehrer bei der Inventur oder Ausleihe helfen können.

Letztendlich sind die User Roles in drei Kategorien, Ausleiher, Bibliothekare und Admins (Abbildung 3.3) aufteilbar. Sowohl Schüler als auch einfache Lehrer haben nur lesende Einsicht auf ihre eigenen ausgeliehenen Medien, daher lassen sie sich als Ausleiher zusammenfassen. Der Bibliothekar ist als *User* mit Ausleihe und Inventurrechten zu sehen. Zuletzt wird der Admin betrachtet, der selbst ein Lehrer sein kann, wie die Persona Gunnar (Abbildung A.2)

Tabelle 3.1: User Roles und ihre Rechte

User Role	Persona	Rechte
Admin	Gunnar	Administration, Voller Zugriff
Bibliothekar	Lisa	Ausleihe, Inventur
Schüler	-	Nur lesende rechte auf eigene ausgeliehene Medien
Lehrer	Katarina	Nur lesende rechte auf eigene ausgeliehene Medien
Inventur-Helfer	Katarina,Gunnar	Kann Bücher aus dem Inventar löschen oder hinzufügen
Ausleih-Helfer	Katarina,Gunnar	Kann Medien ausleihen oder zurück nehmen

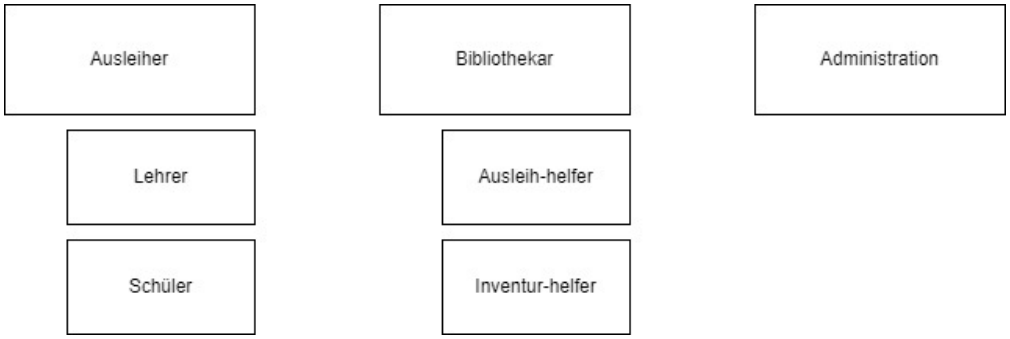

Abbildung 3.3: Kategorisierung der User Roles

3.3 Anforderungen - User Storys & Use Cases

In diesem Abschnitt der Arbeit werden die Anforderungen der Applikation analysiert. Die Anforderungen werden mittels User Storys erfasst. Neben User Storys gibt es noch *Use-Cases*. Es folgt eine kurze Unterscheidung zwischen User Story & Use-Case.

Eine User Story repräsentiert eine Anforderung und folgt einem gewissen Standardformat, welcher wie folgt aussieht:

Als <Rolle> <möchte/ will / muss> ich <etwas tun>, damit <Grund> [19]

Ein Use Case (Anwendungsfall) ist eine verallgemeinerte Beschreibung einer Reihe von Interaktionen zwischen dem System und einem oder mehreren Akteuren, wobei ein Akteur entweder ein Benutzer oder ein anderes System ist [16]. Eines der Hauptunterschiede zwischen User Storys und Use Cases ist der Umfang. Während eine User Story klein

gehalten wird, sodass sie problemlos mit ein bis zwei Sätzen auf eine Karte notiert werden kann, sind Use Cases umfangreicher und beschreiben unter anderem Systemabläufe. Wichtig ist hier die Perspektive. User Stories werden aus der Perspektive der End-User geschrieben und beinhalten die Motivation bzw. das Ziel/Ergebnis, welches sie erreichen möchten, während Use Cases aus Sicht des Systemflusses geschrieben werden, sprich wie der User das System ”triggered” und welche Abläufe er damit anstößt, um an sein Ziel zu gelangen. Use Cases können aus User Storys entstehen, da eine User Story oft als eine Teilmenge bzw. als ”Erfolgsszenario” eines Use Cases gesehen wird. Use Cases besitzen ebenfalls einen Standard, siehe Abbildung 3.4. Abschließend eine Antwort auf die Frage: ”Wann nutze ich was” ? Wie bereits erwähnt, ist der Umfang von Use Cases größer gegenüber User Storys, da sie ein Feature detaillierter beschreiben. Entsprechend würde es zu viel Zeit und Aufwand kosten, alle Features mit Use Cases zu beschreiben. Hier gilt der Ansatz, kritische Hauptfunktionalitäten, die geschäftskritisch oder umfangreich sind, mit Use Cases ausformulieren. Nebenfunktionen, wie "Login", können mit User Storys formuliert werden.

3.3.1 User Storys

Während der Analyse der Prozesse sowie in Gesprächen mit der Bibliothekarin und den Admin der Schule kamen folgende 15 User Storys hervor.

1. Als Ausleiher kann ich mich auf der Webseite einloggen.
2. Als Ausleiher kann ich meine ausgeliehenen Medien auf der Webseite sehen.
3. Als Ausleiher kann ich Medien ausleihen und zurückgeben.
4. Als Admin möchte ich die User-Verwaltung per csv-import anlegen können.
5. Als Admin kann ich Usern Rechte geben und entziehen.
6. Als Bibliothekarin kann ich neue Medien ins System inventarisieren.
7. Als Bibliothekarin kann ich Medien aus dem System löschen.
8. Als Bibliothekarin möchte ich Medien ausgeben können.
9. Als Bibliothekarin möchte ich Medien zurücknehmen können.
10. Als Bibliothekarin möchte ich die Ausleihhistorie eines Mediums sehen.

Use Case Title: Pay for a job posting
Primary Actor: Recruiter
Level: Actor goal
Precondition: The job information has been entered but is not viewable.
Minimal Guarantees: None
Success Guarantees: Job is posted; recruiter's credit card is charged.
Main Success Scenario:

1. Recruiter submits credit card number, date, and authentication information.
2. System validates credit card.
3. System charges credit card full amount.
4. Job posting is made viewable to Job Seekers.
5. Recruiter is given a unique confirmation number.

Extensions:

2a: The card is not of a type accepted by the system:
2a1: The system notifies the user to use a different card.
2b: The card is expired:
2b1: The system notifies the user to use a different card.
2c: The card is expired:
2c1: The system notifies the user to use a different card.
3a: The card has insufficient available credit to post the ad.
3a1: The system charges as much as it can to the current credit card.
3a2: The user is told about the problem and asked to enter a second credit card for the remaining charge. The use case continues at Step 2.

Abbildung 3.4: Beispiel Use Case [16]

11. Als Bibliothekarin möchte ich technische Medien mit ihrer Seriennummer inventarisieren können.
12. Als Inventur-Helfer kann ich neue Medien in das System inventarisieren.
13. Als Inventur-Helfer kann ich Medien aus dem System löschen.
14. Als Ausleih-Helfer möchte ich Medien an User ausleihen können.
15. Als Ausleih-Helfer möchte ich Medien von User zurücknehmen können.

4 Spezifikation

In diesem Kapitel wird der Entwurf der Datenbank sowie eine nähere Betrachtung der Use-Cases dargestellt. Zuerst wird das Entity Relationship Model (ERM) entworfen und im Anschluss folgen die Tabellen, welche aus dem ERM abgeleitet werden. Im zweiten Teil wird näher auf die Use-Cases eingegangen, samt Wireframes und BPMN.

4.1 Entity Relationship Model

In dieser Arbeit wird die ”Chen” Notation genutzt, um das Entity Relationship Model (ERM) zu modellieren. Entitäten werden in Form von Rechtecken dargestellt. Eine Entität bildet Dinge der realen Welt ab und ist in der Regel eine Tabelle in einer relationalen Datenbank. Die Ovale sind Attribute. Attribute sind Eigenschaften von Entitäten. Ist ein Attribut unterstrichen, so ist dies der Primärschlüssel jener Entität. Der Primärschlüssel dient der eindeutigen Identifizierung einer Entität. Die Rauten beschreiben die Beziehungen zwischen zwei Entitäten und die Zahlen bzw. Buchstaben die Kardinalität (Anzahl). Die Kardinalitäten sind:

- 1: Einfache Assoziation.
- C: Konditionelle Assoziation (0 oder 1).
- M: Multiple Assoziation (1 oder mehrere).
- MC: Multiple-konditionelle Assoziation (0, 1 oder mehrere).

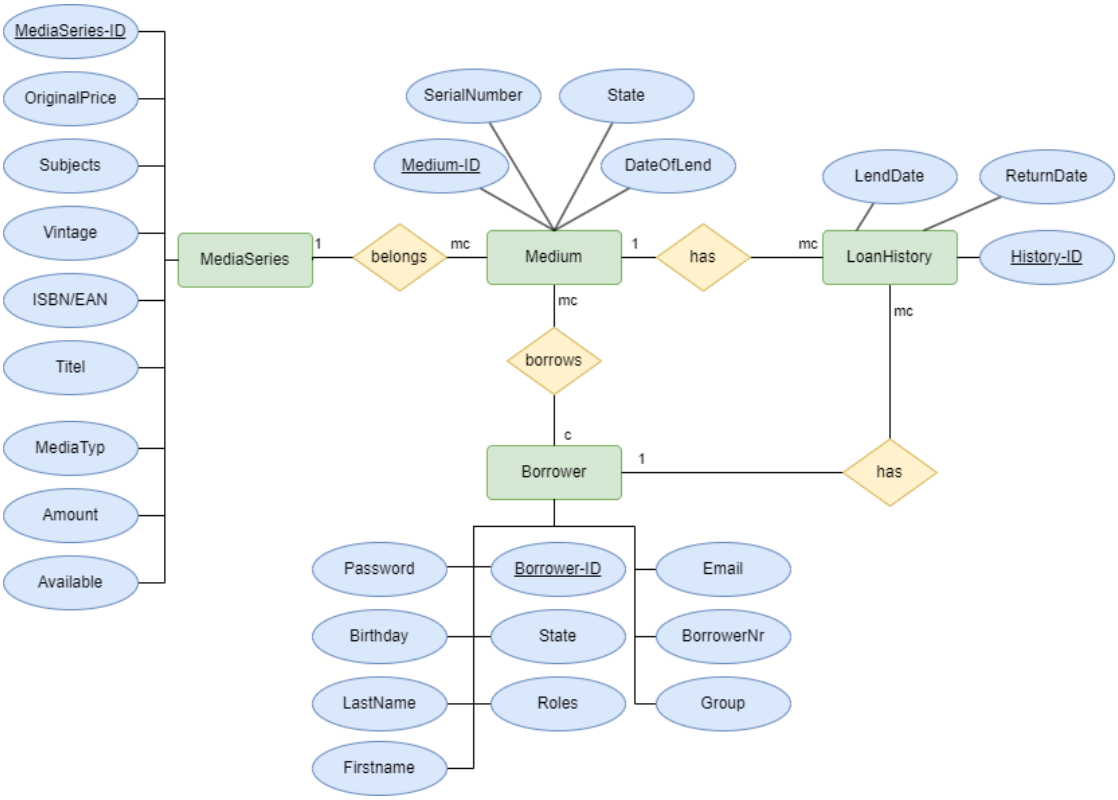

Abbildung 4.1: Entity Relationship Model

4.1.1 Beschreibung des Models

Eine Medien-Reihe (MediaSeries) beinhaltet kein, ein oder mehrere Medien. Ein Medium gehört immer genau zu einer Medien-Reihe. Ein Medium ist entweder nicht ausgeliehen oder von genau einem Ausleiher (Borrower) ausgeliehen. Ein Ausleiher kann kein, ein oder mehrere Medien ausleihen. Ein Medium hat keine, eine oder mehrere Ausleih-historien (LoanHistory). Eine Ausleih-Historie gehört genau zu einem Medium, sowie genau zu einem Ausleiher. Ein Ausleiher ist in keiner, einer oder mehrere Ausleih-Historien vermerkt.

4.1.2 Tabellen

Aus dem ERM lassen sich folgende Tabellen ableiten: Pro Entität eine Tabelle, sprich MediaSeries, Medium, Borrower und LoanHistory. Die Relationen zwischen den Entitäten werden mittels Fremdschlüssel abgebildet. Besitzt eine Entität Attribute, welche

als Listen realisiert werden können, wie bspw. MediaSeries mit zwei Attributen (Vintage, Subjects), so werden jene Attribute bzw. die Felder mit den Annotation *@ElementCollection* versehen. JPA bildet jene Felder jeweils als eigene Tabelle ab, welche von der Entität verwaltet werden. Jene Tabellen besitzen zwei Attribute. Einmal den Fremdschlüssel zur jeweiligen Entität und den entsprechenden Wert der zu persistieren ist (Tabelle 4.2). Für die restlichen Tabellen wird auf dem Anhang (Tabellen A.1) verwiesen.

MediaSeries:				
Attribute	Datentyp	Primarykey	Foreignkey	Not Null
media_series_id	bigInt	✓		✓
title	varchar			✓
media_typ	varchar			
isbn_ean	varchar			
original_price	double precision			
amount	bigint			
available	bigint			

Tabelle 4.1: Beispiel Tabelle MediaSeries

@ElementCollection:				
Attribute	Datentyp	Primarykey	Foreignkey	Not Null
entity_name_id	bigInt		✓	✓
value	<T>			✓

Tabelle 4.2: Tabelle für ein Feld annotiert mit @ElementCollection

4.2 Use-Cases

4.2.1 Use-Case Diagramm

Das Anwendungsfalldiagramm in Abbildung 4.2 gibt einen Überblick über die Anwendungsfälle. Unsere Akteure sind jene User Roles, welche im Abschnitt 3.2 User Roles definiert sind. *Admin*, *Bibliothekar*, *Inventur-Helfer* & *Ausleih-Helfer* sind als *Ausleiher* zu sehen. Jeder Ausleiher kann sich in das System einloggen und seine eigenen ausgeliehenen Medien einsehen. Der Admin kann sowohl die Ausleihe und Rücknahme der Medien durchführen als auch deren Inventarisierung. Des Weiteren verwaltet der Admin das Anlegen, Löschen und Verändern der User mittels CSV-Import. Zuletzt kann der Admin

anderen User die User-Roles Admin, Bibliothekar, Inventur-Helfer & Ausleih-Helfer geben oder nehmen. Der Bibliothekar kann sowohl die Ausleihe und Rücknahme der Medien durchführen als auch deren Inventarisierung. Der Ausleih-Helfer kann die Ausleihe und Rücknahme der Medien durchführen. Der Inventur-Helfer kann das Inventar verwalten, sprich Medien löschen und aufnehmen.

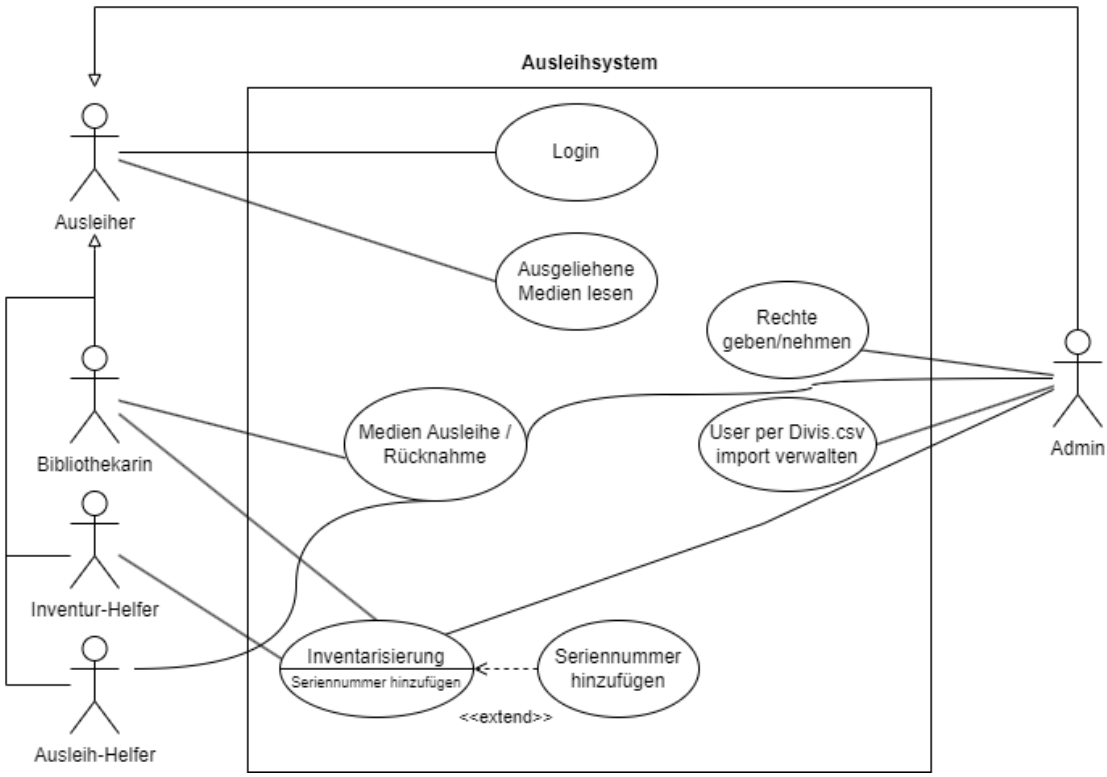

Abbildung 4.2: Use-Case Diagramm

4.2.2 Anwendungsfall 1: Login

Jeder Nutzer kann sich mit seinen Zugangsdaten (Email & Passwort) einloggen. Für das initiale Logging erhält der Nutzer ein Initialpasswort.

4.2.3 Anwendungsfall 2: Ausgeliehene Medien lesen

Jeder Nutzer kann seine ausgeliehenen Medien einsehen.

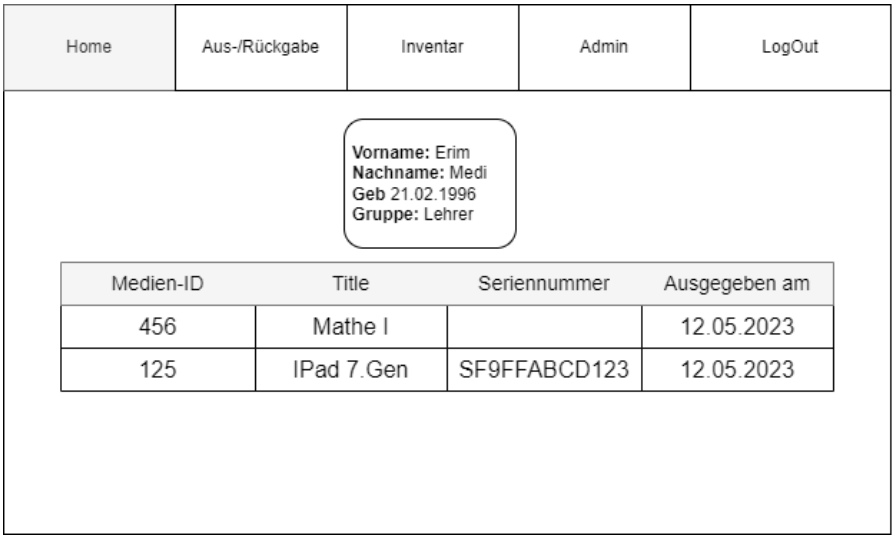

Abbildung 4.3: Wireframe HomeView - Auflistung der aktuell ausgeliehenen Medien

4.2.4 Anwendungsfall 3: Medien Ausleihe/ Rücknahme

Admin, Bibliothekar und Helfer-Ausleihe können Medien an Ausleiher ausleihen und wieder zurück nehmen. Der Prozess einer Ausleihe (Abb. 4.6) bzw. Rücknahme läuft folgend ab. Der Ausleiher möchte ein Medium ausleihen bzw. zurückgeben. Er wird im System gesucht (Abb. 4.4). Ist er nicht im System wird die Ausleihe verweigert. Ist er im System, kann das Medium ausgeliehen werden, indem der Barcode bzw. die MedienID des Mediums eingescannt oder getippt wird (Abb. 4.5). Die Ausleihe bzw. Rücknahme wird im System vermerkt.

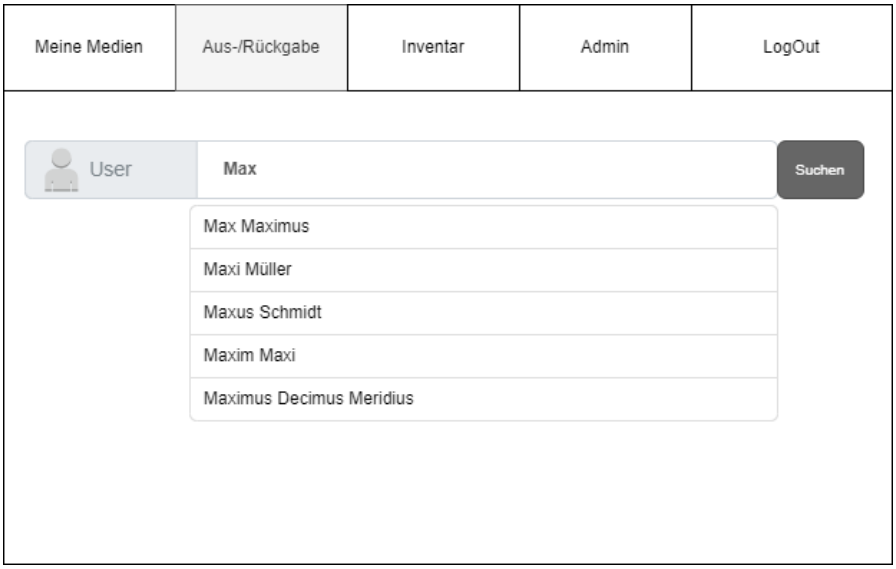

Abbildung 4.4: Wireframe - Suche Nutzer im System für Ausleih-/Rücknahmevorgang

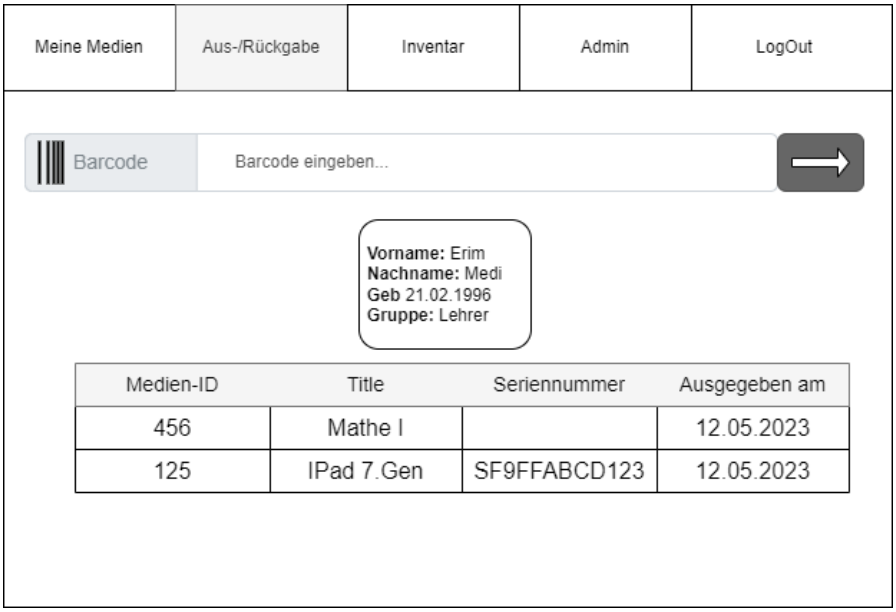

Abbildung 4.5: Wireframe AusleihView - Medien-ID einscannen zum Ausleihen/zur Rücknahme

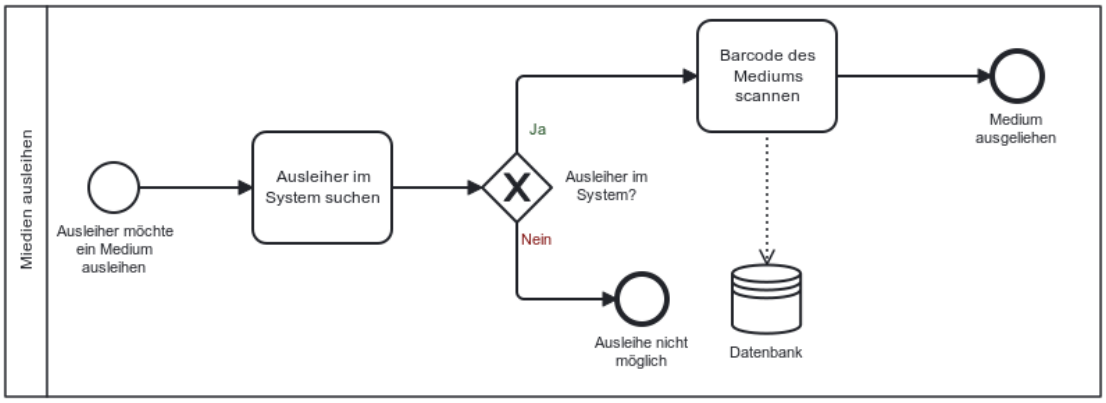

Abbildung 4.6: Ausleihprozess

4.2.5 Anwendungsfall 4: Inventarisierung

Ein Medium wird in das Inventar aufgenommen, indem es einer Medienreihe untergeordnet wird. Existiert keine passende Medienreihe, so muss diese zunächst erstellt werden (Abb. 4.10). Medienreihen fassen übergreifende Informationen gleicher Medien zusammen wie bspw. die ISBN oder den Titel. Das Medium wird mittels eines Barcodes inventarisiert, welche seine eindeutige ID (MedienID) abbildet. Im Falle technischer Medien wie iPads wird die optionale Möglichkeit geboten, jenes Medium mit einer Seriennummer zu inventarisieren (Abb. 4.9).

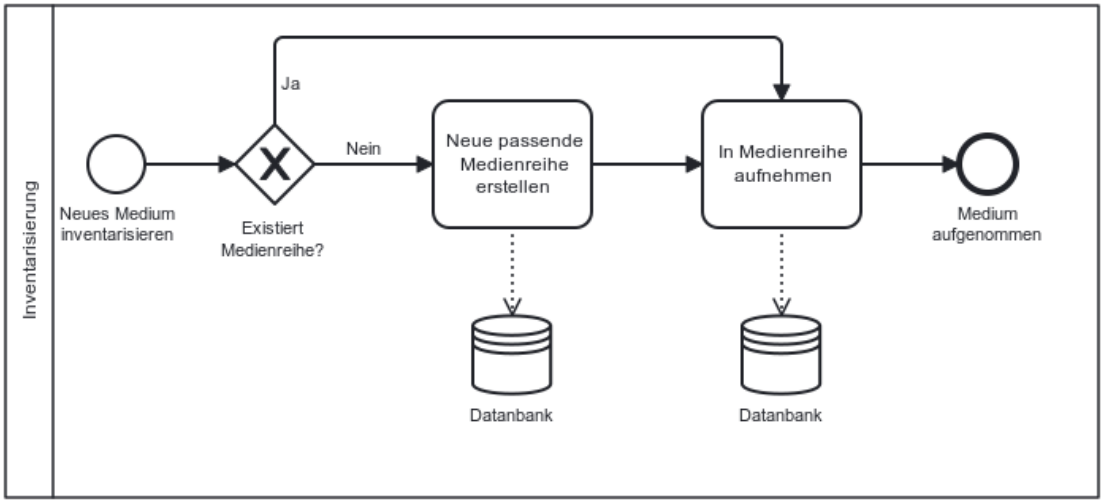

Abbildung 4.7: Inventarisierungsprozess

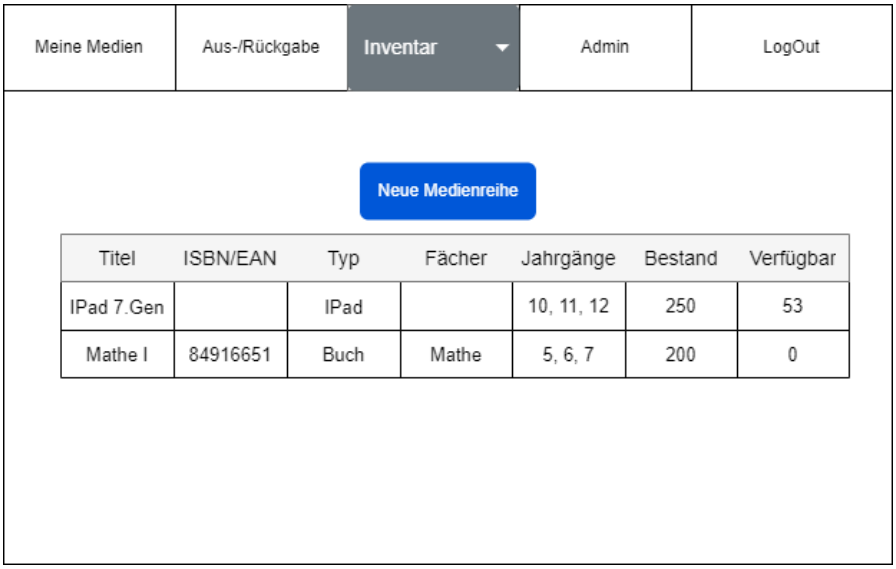

Abbildung 4.8: Wireframe - Medienreihen Übersicht

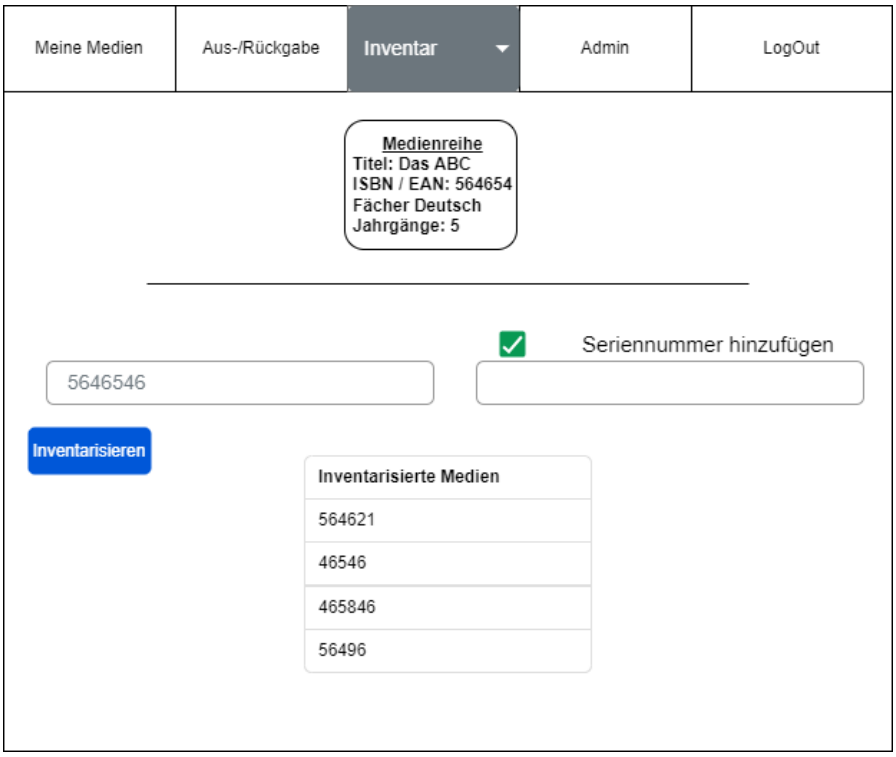

Abbildung 4.9: Wireframe - Medium in MedienReihe aufnehmen

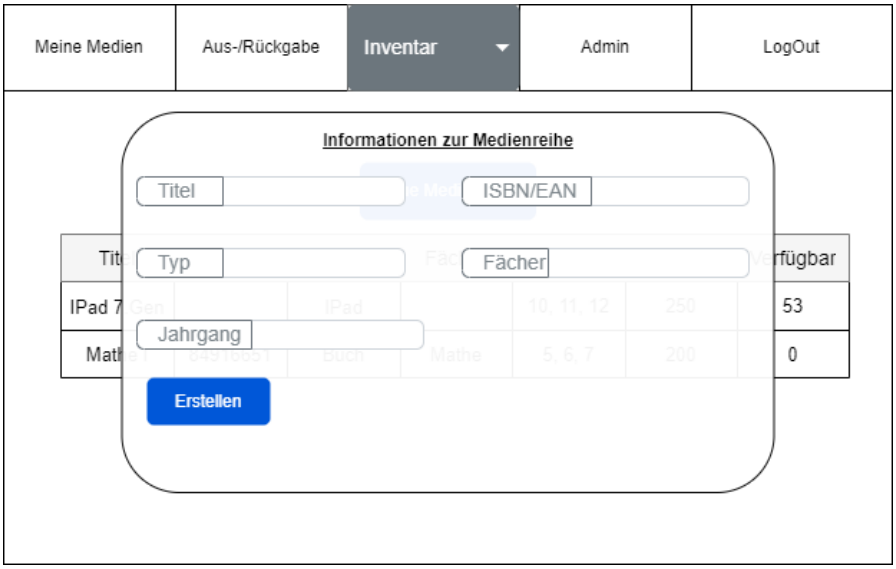

Abbildung 4.10: Wireframe - Neue Medienreihe erstellen. Es öffnet sich ein transparentes Fenster, indem sämtliche benötigten Informationen zur neuen Medienreihe übergeben werden.

4.2.6 Anwendungsfall 6: Medien löschen

Ein Medium kann genau nur dann gelöscht werden, wenn es aktuell nicht entliehen ist. Ist das Medium entliehen, wird das Löschen verweigert mit einem Verweis auf den aktuellen Ausleiher. Das Löschen geschieht über die eindeutige Medien-ID. Ein Löschvorgang kann nicht rückgängig gemacht werden. Wird ein Medium versehentlich gelöscht, muss es neu inventarisiert werden (Abb. 4.11).

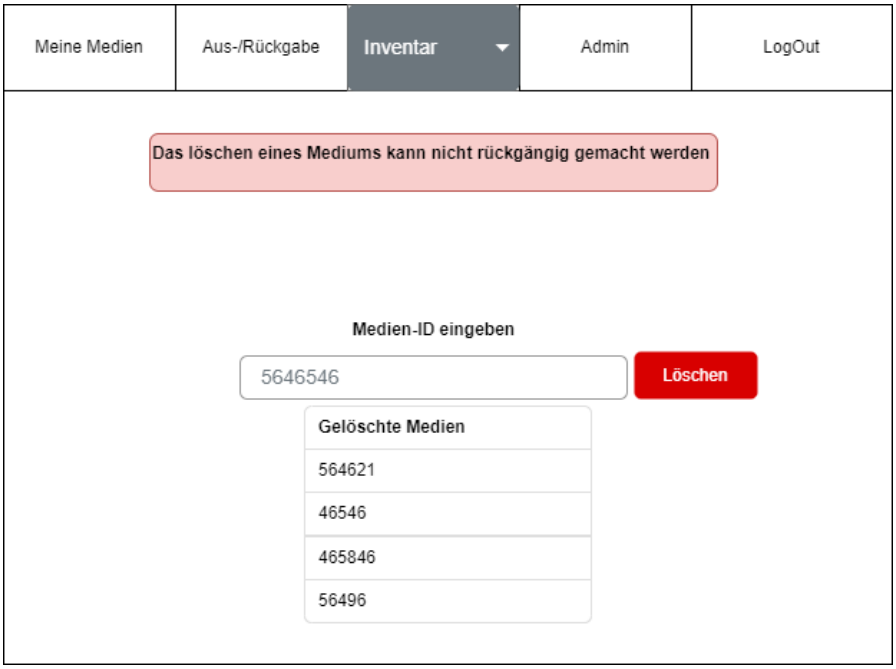

Abbildung 4.11: Wireframe - Medium löschen.

4.2.7 Anwendungsfall 7: User-Import

Die Schule nutzt "DiViS" als Schulmanagementsystem. Sämtliche Informationen zur Schülerschaft und Kollegium werden in DiViS eingespeist. DiViS besitzt die Datenhoheit. Aus DiViS werden wiederum per CSV-Export andere Schulsysteme befüllt bspw. die Schulplattform IServ. Das Erstellen, Bearbeiten und Deaktivieren der Nutzer soll ausschließlich über ein CSV-Import gewährleistet werden, um die größtmögliche Konsistenz der Daten zu gewähren. Die CSV-Datei besitzt fünf Spalten, welche eine fest definierte Reihenfolge haben.

1. Spalte: eindeutige nummerische ID
2. Spalte: Vorname
3. Spalte: Nachname
4. Spalte: Gruppe/Klasse
5. Spalte: Geburtstag
6. Spalte: Email

Die CSV-Datei wird mit der Datenbank verglichen. Filterkriterium ist hier die ID in der CSV. Die CSV-ID entspricht dem Fachlichenschlüssel *BorrowerNr* in der Borrower-Entität. Es sind drei Fälle zu unterscheiden:

1. ID in CSV, aber nicht in Datenbank → Neuer Nutzer
2. ID in CSV und in Datenbank → prüfe auf Änderungen
3. ID **nicht** in CSV, aber in Datenbank → deaktiviere Nutzer

Werden neue Nutzer hinzugefügt, wird eine CSV-Datei mit den neuen Nutzern inklusive ihrer Initialpasswörter erstellt und an den Client gesendet.

4.2.8 Anwendungsfall 8: Rechteverwaltung

Ein Admin kann anderen Nutzern Rechte bzw. Rollen verteilen. Die verteilbaren Rollen sind: Admin, Bibliothekar, Ausleih-Helfer und Inventar-Helfer

5 Entwurf

In diesem Kapitel wird der Entwurf der Applikation dargestellt, die eine Client-Server Architektur ist. Zuerst wird das Backend dargestellt und im Anschluss das Frontend.

5.1 Architektur

Die zugrunde liegende Architektur ist eine klassische (strikte) Drei-Schichten-Architektur (Abbildung 5.1). Das User-Interface (UI) stellt für die Applikation die Schnittstelle nach außen dar. Der fachliche Code wie Services und Entitäten ist in der Geschäftslogik anzufinden. Die Persistenzschicht übernimmt die klassischen CRUD-Operationen (CREATE, READ, UPDATE, DELETE) auf den Ressourcen.

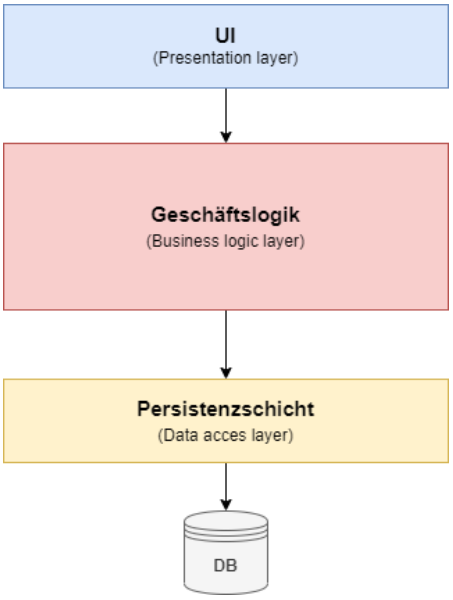

Abbildung 5.1: Drei-Schichten-Architektur der Applikation

5.2 Backend

Das Backend wird mittels Spring-Boot realisiert, welche eine Erweiterung des Spring-Frameworks ist. Durch die Popularität und Nutzung haben sich Standards etabliert. Abbildung 5.2 zeigt eine typische Spring-Boot MVC (Model-View-Controller) Architektur.

- **Controller:** Controller sind die Schnittstellen nach außen und definieren die Endpoints der API. Sie empfangen und validieren HTTP-Requests sowie delegieren, entsprechend der geforderten Ressourcen, Funktionsaufrufe auf den Services-Klassen. Das De/-Serialisieren von JSON zu Objekten und Objekten zu JSON geschieht in den Controllern.
- **Service:** Services repräsentieren die Geschäftslogik. Hier werden die Use-Cases implementiert. Sie erstellen, löschen oder manipulieren Models. Repositories werden per *Dependecy Injection* in den Services-Klassen injiziert.
- **Model:** Models sind ebenfalls Teil der Geschäftslogik. Sie besitzen keine Logik, sondern dienen der Modellierung realer Objekte. Sie bilden Entitäten ab und werden in einer Datenbank persistiert.
- **Repository:** Repositories bilden die Persistenzschicht und stellen CRUD-Funktionen auf unseren Models zur Verfügung. Repositories abstrahieren Datenbankabfragen.

Der API-Zugriff wird mit Spring-Security und JSON-Web-Tokens (JWT) gesichert. Mehr dazu in Kapitel Spring-Security 6.2.

5.3 Frontend

Das Frontend wird mittels Vue.js entwickelt. In Abbildung 5.3 ist die Architektur des Frontends zu sehen.

- **App:** App ist die Root-Komponente, welche alle anderen Komponenten/Views als Kinder beinhaltet. App besteht aus dem Router und der Navbar. App erhält von Vuex store den *state*, sprich ob ein User eingeloggt ist. Die Navbar wird passend zu den Rollen des User gerendert.

- **Views:** Das Frontend besteht aus mehreren Views. Die Views bestehen selbst aus anderen Vue-SFC. Die Views sind in fünf Kategorien eingeteilt.
  - LoginView: Für das Login. LoginView delegiert das Login an den Vuex-Store.
  - HomeView: Startseite nach dem Login. Rendert die ausgeliehen Medien des Users.
  - LoanViews: Für User-Suche & den Ausleih-/Rücknahmevorgang der Medien.
  - InventoryViews: Besteht aus mehreren Views für die Aufnahme & Löschung von Medien sowie eine Gesamtübersicht.
  - AdminViews: Besteht aus mehreren Views für User-Import, Rollenverwaltung und einer Gesamtübersicht.
- **Router:** Rendert die richtigen Views entsprechend der URL.
- **Vuex store:** Um das Login im Frontend zu gewährleisten, bedarf es einer Möglichkeit, den aktuellen Status einer Anwendung zu überwachen, sprich ob ein Nutzer eingeloggt ist oder nicht. Vuex bietet diese Möglichkeit. Vuex ist eine State-Management-Libary für Vue Anwendungen [15]. Mit Vuex lässt sich ein globaler Store sowie Aktionen auf den Store definieren. Ein Store ist ein Container der den Zustand der Andwendung speichert. Es lassen sich diverse referenzierbare Werte speichern, auf die alle Komponenten Zugriff haben können, sodass sie entsprechend einer Änderung korrekt agieren können. Der Store wird genutzt, um den User und den *loggedInState* zu speichern. Sendet *authService* ein erfolgreiches Login, wird der State und der User entsprechend gesetzt, analog zu einem fehlgeschlagenen Login.
- **authService:** Sendet via Axios eine HTTP-Anfrage für das Login ans Backend. Die Response (Succes/Error) wird zum Vuex-Store gesendet. Bei einem erfolgreichen Login wird der JWT (JSON Web Token) im *Local Storage* gespeichert.
- **Axios:** Axios ist eine Bibliothek zum Senden und Empfangen von HTTP Anfragen. Sämtliche HTTP Anfragen werden über Axios getätigt.

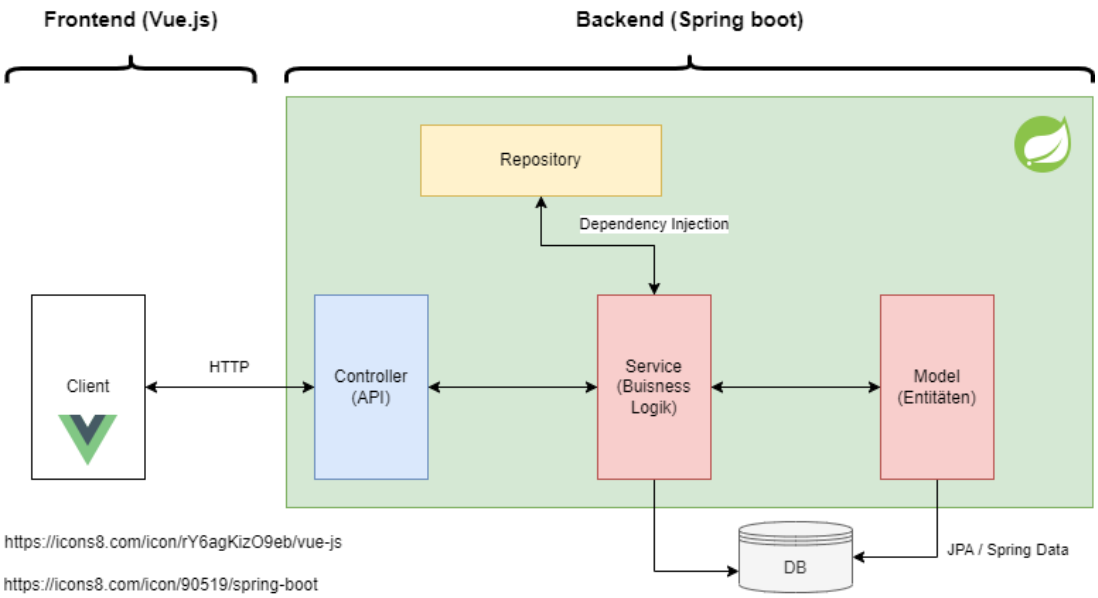

Abbildung 5.2: Fluss-Architektur einer Spring-MVC Applikation

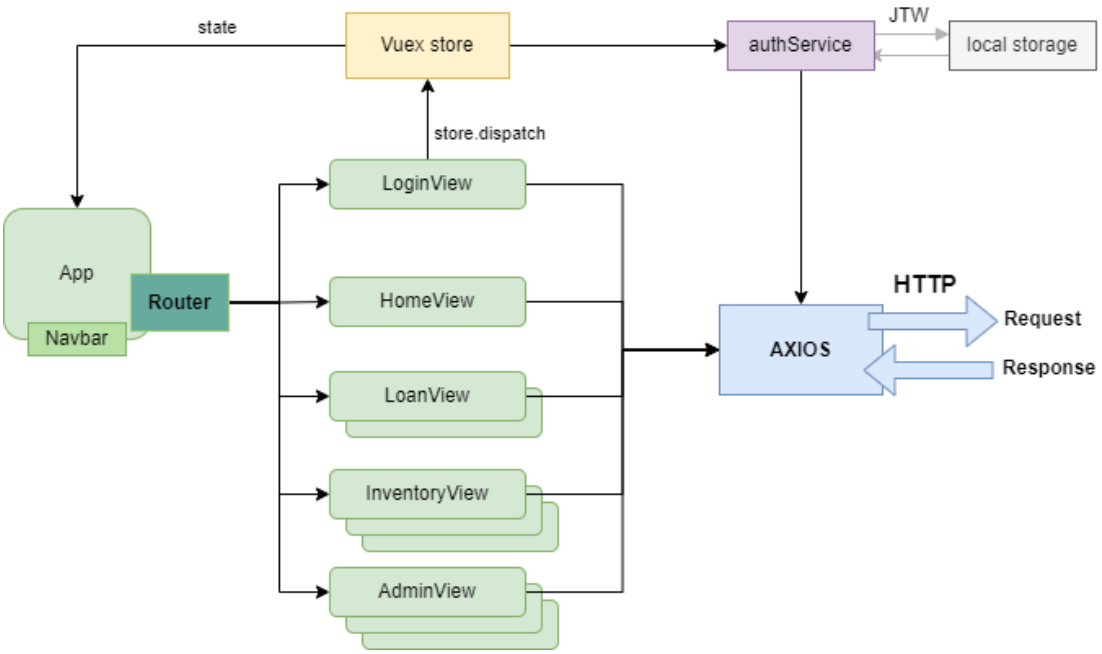

Abbildung 5.3: Architektur des Frontend mit Vue. Übernommen und angepasst von bezkoder

6 Implementierung

Das Folgende Kapitel behandelt die Implementierung des Backend mit Spring-Boot und Frontend mit Vue.js. Begonnen wird mit dem Backend. Anschließend folgt ein separates Kapitel für Spring-Security. Zuletzt folgt das Frontend.

6.1 Backend

Das Backend ist in folgende Packages aufgeteilt: *config*, *controller*, *dto*, *entities*, *enums*, *exceptions*, *helper*, *repositories*, *security*, & *service*. Um den Rahmen leserfreundlich klein zu halten, werden nicht sämtliche Implementationen vorgestellt. Es wird jeweils ein Beispiel zu den Packages *config*, *entities*, *repositories* & *service* gezeigt, da dies die Spring-Boot relevantesten Packages sind. Das *security* Package, sprich die Implementierung von *Spring-Security* wird im Abschnitt 6.2 erläutert. Das Package *helper* beinhaltet zwei Hilfs-Klassen: *CSVHelper* zum abbilden von CSV-Dateien in Borrower-Entitäten und umgekehrt & *PasswordGenerator* zum Erstellen von zufällig numerischen Passwörtern für das initiale Log-in. Das Package *dto* beinhaltet die *Data-Transfer-Objects* (DTO), welche Gebrauch in den Controller-Klassen finden. Ein DTO fasst die zu übertragenden Daten in einem neuen Objekt zusammen. Die Daten können aus unterschiedlichen Entitäten stammen[2].

6.1.1 Spring-Boot aufsetzen

Das Backend wurde mittels Spring Initializr erstellt. Mit Spring Initializr lässt sich eine Spring-Boot-Anwendung inklusive aller gewünschten Dependencies sehr leicht initialisieren. Die Dependencies sind:

- **spring-boot-starter-web:** Inkludiert alles zum Erstellen von RESTful-Applications. Darunter auch Spring-MVC.

- **spring-boot-starter-data-jpa:** Persistieren der Daten über die Jakarta Persistence API (JPA) mittels Hibernate und Spring Data.
- **spring-boot-starter-validation:** Erweiterte Validierungsannotationen für Hibernate.
- **lombok:** Annotationen zur Reduzierung von Boilerplate-Code.
- **PostgreSQL Driver:** Anbindung an einer Postgre Datenbank.

Nachträglich hinzugefügt:

- **commons-csv:** CSV-Parser.
- **modelmapper:** Wandelt Daten zu Objekten und umgekehrt.
- **jjwt-api, jjwt-impl, jjwt-jackson:** Erstellen von JSON Web Tokens.
- **springdoc-openapi-starter-webmvc-ui:** Swagger zum Testen der API.
- **spring-boot-starter-security:** Zur Sicherung der API.
- **com.h2database:** in-Memory H2 Datenbank.

6.1.2 Configs

Es gibt vier Config-Klassen.

- **InitializeConfig:** Initialisiert Dummy-Daten in die Datenbank. Konfiguriert und initialisiert Swagger-API und Modelmapper.
- **EnumMappingConfig:** Enums werden *case-insensitive* für Controller.
- **SecurityConfig & SecurityFilterConfig:** Siehe Abschnitt 6.2 Spring-Security.

6.1.3 Entities

Die Annotation @Entity deklariert eine Klasse als JPA-Entity, welche als Tabelle in der Datenbank persistiert wird. Eine Entität benötigt eine Id und einen parameterlosen Konstruktor. @Id deklariert den Primärschlüssel der Entität. Die Id wird für das Medium nicht automatisch generiert, sondern manuell bei der Inventarisierung übergeben. Wird @GeneratedValue unter @Id ergänzt, so übernimmt Spring-Boot bzw. Hibernate die automatische Generierung einer eindeutigen Id. Die Lombkok-Dependecy ermöglicht durch Annotationen Boilerplate-Code zu reduzieren. Beispielsweise generiert @Data automatisch Getter, Setter, ToString, Equals und HashCode. @AllArgsConstructor generiert einen Konstruktor mit allen Feldern. @NoArgsConcstructor generiert einen parameterlosen Konstruktor.

Die Relationen zwischen den Entitäten werden durch @OneToMany und @ManyToOne Annotationen deklariert. Die Eltern-Entität einer Relation besitzt eine Liste vom Typ der Kind-Entität. Beispielsweise ist Medium die Eltern-Entität der Relation zwischen Medium und LoanHistory. Entsprechend wird das Feld in Medium "List<LoanHistory> loanHistories" mit der Annotation @OneToMany (Ein Medium kann mehrere LoanHistories haben) markiert. Auf der anderen Seite der Relation wird in LoanHistory das Feld "Medium medium" mit @ManyToOne (Mehrere LoanHistories gehören zu einem Medium) markiert. Durch beidseitiges Markieren der Relation wird diese Relation eine bidirektional Relation. Ein Vorteil einer bidirektionalen Relation ist die beidseitige Navigation. Ein Nachteil kann eine schlechtere Performance sein [1]. Das Attribute "mappedBy" wird in bidirektionalen Relationen benutzt, um der Datenbank mitzuteilen wem die Relation "gehört", sprich wer die Fremdschlüssel verwaltet. In der Regel ist es die @ManyToOne Seite. Der Wert in mappedBy ist der Name des Feldes in der @ManyToOne Seite (Medium **medium**).

Analog sieht die Implementierung der anderen Entitäten aus.

```
1 //Imports...
2 @Data
3 @Builder
4 @Entity
5 @AllArgsConstructor
6 @NoArgsConstructor
7 public class Medium {
8     @Id
9     private Long mediumID;
```

```
10      private String serialNr;
11      private LocalDate dateOfLend;
12      @Enumerated(EnumType.STRING)
13      private Status status;
14
15      //Many instances of  Medium are mapped to one instance Borrower
16      @ManyToOne
17      @JoinColumn(name = "borrower_id")
18      @JsonIgnoreProperties("mediumList")
19      private Borrower borrower;
20
21      @ManyToOne
22      @JoinColumn(name = "media_series_id")
23      private MediaSeries mediaSeries;
24
25      @OneToMany(mappedBy = "medium",cascade=CascadeType.ALL)
26      private List<LoanHistory> loanHistories = new ArrayList<>();
27
28      //More Methods...
29  }
```

Listing 6.1: Medium Entity Snippet

6.1.4 Controller

Controller bilden die API-Endpoints, welche von einem Client angesprochen werden können. Die @RestController Annotation inkludiert @Controller (markiert Klasse als Web-Request-Handler) und @ResponseBody (automatisches serialisieren der Response zu JSON). Anhand der Methode *addMedium* in Listing 6.2 ist zu sehen, wie Controller prinzipiell funktionieren. Spricht der Client zum Inventarisieren eines Medium den Endpoint *api/v1/inventory/series/{seriesID}/media* an, wird die Methode addMedium ausgeführt. Die Methode erwartet im Http-Body der Request ein JSON-Object bestehend aus ID und Seriennummer des Mediums (MediumRequest). Die MediumRequest wird in eine Medium-Entität abgebildet und der Serviceklasse übergeben. Wirft der Service keine Exception, wurde das Medium erfolgreich inventarisiert und ein DTO (MediumResponse) mit den Informationen zum neuen Medium wird zurück gesendet. Andernfalls wird passend zur Exception eine Response zum Client gesendet. Analog sieht die Implementierung der restlichen Controller aus (BorrowerController, AuthController, LoanController)

Spring-MVC Annotationen

Die Annotation @RequestMapping auf Klassenlevel, zusammen mit den spezifischeren @GetMapping, @PostMapping, @PutMapping und @DeleteMapping auf Methodenlevel, mappen entsprechende Http-Requests auf die korrekten Methoden. Hierbei sollte der Basis-URL-Pfad durch @ReqeustMapping definiert werden (Siehe Listing 6.2, Zeile 3). @RequestBody (Listing 6.2, Zeile 11) ermöglicht eine automatische Deserialisierung des Http-Body in der Request. Es bildet den Http-Body auf den Parameter in der Methode ab. Mit @PathVariable (Listing 6.2, Zeile 11) lassen sich URI-Template Platzhalter(Listing 6.2, Zeile 10 { seriesID }) auf Parameter in der Methode abbilden [18].

ResponseEntity<T> & ResponseStatusException

Die Klasse ResponseEntity erweitert die Klasse HttpEntity, die den Header & Body einer Request/Response enthält, um HttpStatusCodes. Hiermit kann, basierend auf dem Verarbeitungsergebnis der Request, ein passender Http-Status dem Client mitgesendet werden. Im Fehlerfall kann bspw. der Body leer gelassen und nur ein Http-Status code gesendet werden. Das ist nicht immer ideal, da der Client eventuell mehr Kontext benötigt. In diesem Fall eignet sich die Klasse *ResponseStatusException*, da hier nicht nur ein Http-Code, sonder auch eine individuelle Nachricht mitgesendet werden kann.

```
1  //Imports...
2  @RestController
3  @RequestMapping(path = "api/v1/inventory")
4  @Tag(name = "Inventory")
5  public class InventoryController {
6      //Constructor and Fields...
7      //more endpoints...
8
9      //add new Medium under a MediaSeries
10     @PostMapping(path = "series/{seriesID}/media" ,consumes = "application/
       json", produces = "application/json")
11     public ResponseEntity<MediumResponse> addMedium(@RequestBody
       MediumRequest mediumRequest, @PathVariable Long seriesID) {
12         Medium newMedium;
13         Medium medium = null;
14         try {
15             medium = convertToMedium(mediumRequest);
16             newMedium = inventoryService.addNewMedium(medium, seriesID);
```

```
17              return new ResponseEntity<>(convertToDTO(newMedium), HttpStatus.
    OK);
18          } catch (NoSuchElementException e) {
19              throw new ResponseStatusException(HttpStatus.NOT_FOUND, "
    mediaSeries: " +seriesID+" nicht gefunden");
20          } catch (MediumIdExistsException e) {
21              throw new ResponseStatusException(HttpStatus.NOT_ACCEPTABLE, "
    Medium ID: " +medium.getMediumID()+ " bereits belegt");
22          }
23      }
24      //more endpoints...
25  }
```

Listing 6.2: Inventory Controller Snippet

6.1.5 Repositories

Mittels Repositories werden Manipulationen auf der Datenbank durchgeführt. Dadurch, dass *spring-boot-starter-data-jpa* als Abhängigkeit hinzugefügt ist, wird automatisch Spring Data JPA und Hibernate als JPA-Implementierung bereitgestellt. Spring Data JPA baut auf Hibernate auf und vereinfacht die Implementierung der Repositories, indem es Boilerplate-Code durch Abstraktion reduziert. Für die Implementierung der Repositories wurde für je eine Entity-Klasse eine Repository-Klasse erstellt, die jeweils von JpaRepository<T,ID> ableiten. JpaRepository stellt viele Methoden bereit, welche direkt genutzt werden können wie bspw. *findAll, count, safe, delete* etc. [18].

Eigene Queries

In Listing 6.3, Zeile 7&8 ist je ein Beispiel zu sehen, wie nur durch Definition von Methodennamen eigene Queries gestaltet werden können, ohne aufwändigen Code zu schreiben. Spring Data JPA analysiert die Methodennamen und implementiert die Abfrage. Hierfür muss auf Standardausdrücke zurückgegriffen werden, wie bspw. *find...By* [7].

Eine weitere Möglichkeit ist der Gebrauch der Java Persistence Query Language (JPQL). Sie ähnelt der SQL-Syntax und erlaubt, Abfragen zu schreiben, die unabhängig von der darunter liegenden Datenbank sind. Spring übersetzt die JPQL-Abfrage immer im Dialekt der darunter liegenden Datenbank (Listing 6.3, Zeile 3).

Spring Data - Pagination (Seitennummerierung)

Fordert ein Client große Datensätze an, bspw. *getAllUsers*, wäre es ungünstig dem Client alle User in einer einzigen Response zu senden. Um den Datentransfer klein zu halten gibt es die Möglichkeit große Datensätze in kleineren "Häppchen" zu übermitteln. Diese Häppchen werden durchnummeriert und haben eine Größe die konfigurierbar ist. Spring Data bietet dafür bestimmte Interfaces an, die für genau diesen Zweck geeignet sind. In Zeile 10, Listing 6.3 ist zu sehen, dass die Query als Parameter unteranderem ein *Pageable* erwartet. Pageable ist ein Interface, welches Informationen über Seitenanzahl, Seitengröße, Sortierung etc. besitzt. Das Pageable Objekt wird im Controller durch die statische Methode PageRequest.of(pageNumber, pageSize,Sort.by(orders)) erstellt und über die Serviceklasse der Query übergeben. Als Ergebnis wird ein Objekt vom Typ "Page" zurückgegeben, welches genau das Häppchen bzw. die Seite ist, mit der Elementgröße, die in der PageRequest.of Methode definiert wurde. Neben dem Inhalt, sprich die Borrowers, besitzt das Page-Objekt weitere Informationen wie Seitennummer, Seitengröße und Gesamtseitenanzahl. Mit dem Page-Objekt kann eine passende Response zum Client gesendet werden. In Abbildung 6.1 ist ein Beispiel für Request und Response zu sehen. Der Client kann durch Query Paramater die gewünschte Seite (pageNumber) und Seitengröße (pageSize) festlegen. Pagination beginnt standardmäßig in Spring bei Seite 0. Die *erste* Seite wird somit durch PageRequest.of(**0**, pageSize) abgefragt. PageRequest.of ist mehrfach überladen. Während *pageNumber* und *pageSize* benötigt werden, kann die Sortierung ausgelassen werden [5].

```
1  //Imports...
2  public interface BorrowerRepository extends JpaRepository<Borrower,Long> {
3      @Query("select u from Borrower u where (:firstName is null or u.firstName
       ilike :firstName%)"
4      +" and (:lastName is null or u.lastName  ilike :lastName%)")
5      List<Borrower> searchByFirstAndOrLastName(@Param("firstName") String
       firstName, @Param("lastName") String lastName);
6
7      Optional<Borrower> findBorrowerByBorrowerNr(Long borrowerNr);
8      Optional<Borrower> findBorrowerByEmail(String email);
9
10     Page<Borrower> findAllByBorrowerState(Pageable pageable,BorrowerState
       borrowerState );
11     //More Queries...
12 }
```

Listing 6.3: BorrowerRepository Snippet

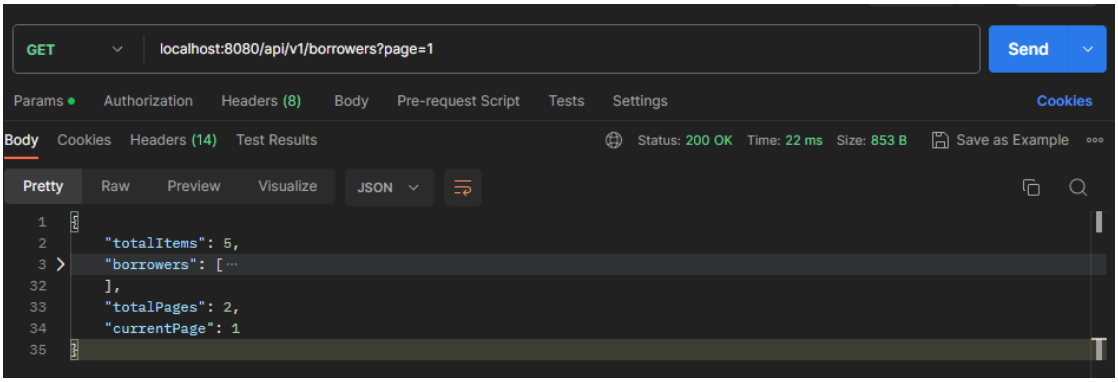

Abbildung 6.1: Pagination - Postman Response Beispiel

6.1.6 Services

Serviceklassen implementieren die Geschäftslogik. Die Annotation @Service leitet wie @Controller und @Repository von @Component ab. Anders als @Controller und @Repository bietet @Service keine besonderen Funktionen. Es markiert eine Klasse als Service, mehr nicht. Der LoanService implementiert den Use-Case Medien Ausleihe/Rücknahme (4.2.4). Des Weiteren wird die Ausleihhistorie eines Mediums durch den LoanService bei jeder Ausleihe und Rücknahme vermerkt.

In Listing 6.4 ist die Implementierung des LoanService zu sehen, der aus folgenden Methoden besteht:

- **loanUnloanMedium:** Die Methode zum Ausleihen & Zurücknehmen. Übergeben als Parameter werden ID des auszuleihenden Mediums und des Borrowers (Ausleihers). Es sind drei Fälle zu unterscheiden:
  - Medium **nicht** ausgeliehen → Leihe *mediumID* an *borrowerID* aus.
  - Medium ist von borrowerID ausgeliehen → Medium wird zurück genommen.
  - Medium ist **nicht** von borrowerID ausgeliehen → Ausleihe verweigern. Wirf Exception.

  Wird das Medium oder der Borrower nicht gefunden, wirft die Methode eine passende Exception.
- **getLoanHistories:** Gibt eine Liste der Ausleihhistorie des Mediums wieder.

- **unloanFromBorrower:** Helfermethode - Rücknahme des Mediums. Setzt den Status des Mediums auf "Verfügbar". Rücknahme wird vermerkt.
- **loanToBorrower:** Helfermethode - Ausleihe des Mediums. Setzt den Status des Mediums auf "Verliehen". Ausleihe wird vermerkt.

```
1  //imports...
2  @Service
3  public class LoanService implements ILoanService {
4
5     //Fields and Constructor...
6
7      @Override
8      public Borrower loanUnloanMedium(Long borrowerID, Long mediumID) throws
       MediumIsBorrowedException {
9          Borrower borrower;
10         Medium medium;
11
12         borrower = borrowerRepository.findById(borrowerID).orElseThrow(()->
       new NoSuchElementException("user with id: "+borrowerID+" not found"));
13         medium   = mediumRepository.findById(mediumID).orElseThrow(()-> new
       NoSuchElementException("medium with id: "+mediumID+" not found"));
14
15         if(medium.isBorrowed()){
16             //Borrower != medium.getBorrower
17             if(!Objects.equals(medium.getBorrower().getBorrowerID(), borrower
       .getBorrowerID()))
18                 throw new MediumIsBorrowedException("MediumID:"+mediumID+ "
       ist bereits verliehen an "+medium.getBorrower().getFullName());
19             else
20 // Medium is already loan to this borrower ==> take it back
21                 return unloanFromBorrower(borrower,medium);
22         }
23         else
24             return loanToBorrower(borrower,medium);
25     }
26
27     @Override
28     public List<LoanHistory> getLoanHistories(Long mediumID) {
29         return loanHistoryRepository.findAllLoanHistoryByMediumMediumID(
       mediumID);
30     }
31
32
```

```
33      private Borrower unloanFromBorrower(Borrower borrower, Medium medium) {
34          medium.setStatus(Status.AVAILABLE);
35          medium.setBorrower(null);
36          medium.setDateOfLend(null);
37
38          //latest LoanHistory
39          LoanHistory loanHistory = medium.getLoanHistories().get(medium.
        getLoanHistories().size()-1);
40          loanHistory.setDateOfReturn(LocalDate.now());
41          mediumRepository.save(medium);
42          return  borrower;
43      }
44
45      private Borrower loanToBorrower(Borrower borrower, Medium medium) {
46          medium.setStatus(Status.RENT);
47          medium.setBorrower(borrower);
48          LocalDate now = LocalDate.now();
49          medium.setDateOfLend(now);
50
51          LoanHistory loanHistory = new LoanHistory(now,borrower,medium);
52          medium.addNewLoanHistory(loanHistory);
53          mediumRepository.save(medium);
54          return  borrower;
55      }
56  }
```

Listing 6.4: LoanService

6.1.7 Endpoints

Abbildung 6.1 gibt eine kurze Übersicht über die API. Für eine detailliertere Sicht und Testung der API wird auf Swagger-UI verwiesen (Abb. 6.2). In der Swagger-UI ist zu sehen wie eine korrekte Request auszusehen hat, bspw. was als RequestBody erwartet wird und wie die Response aussieht. Nach dem Start des Backends ist die UI unter http://localhost:8080/swagger-ui/index.html/ zu finden. Da die API mit Spring-Security gesichert ist, muss zuerst ein Login unternommen werden. Der JWT in der Response kann dann mittels des “Authorize“-Button (Siehe Abb. 6.2) für jede weitere Request eingebunden werden. Zum Testen werden einige Nutzer beim Start der Applikation mit initialisiert. Die Zugangsdaten können aus der Tabelle 6.2 entnommen werden.

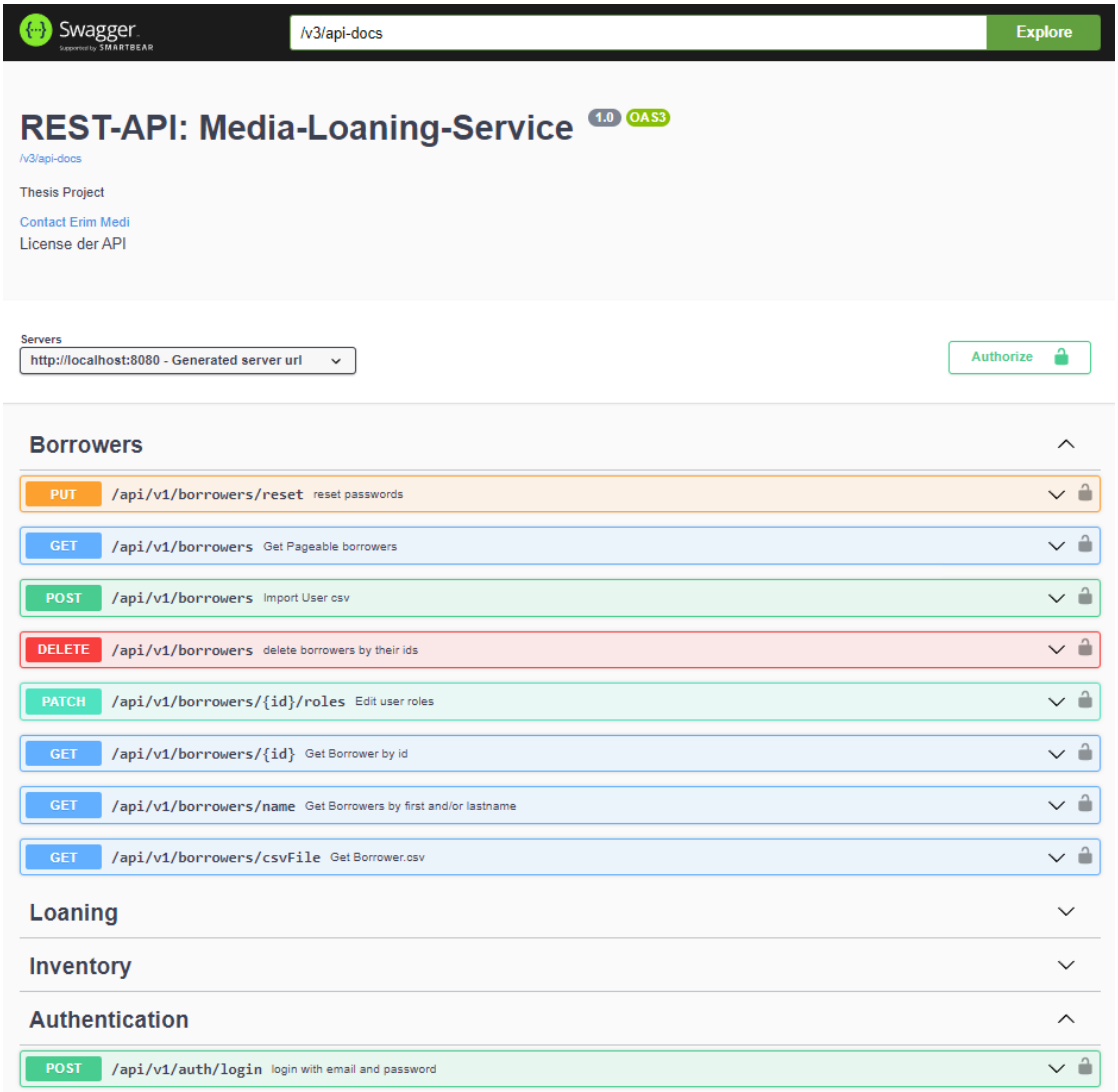

Abbildung 6.2: Swagger-UI

Base-URI: /api/v1	
URI	Methode
/auth/login	POST
/auth/initialLogin	POST
/loan	POST
/loan/histories/{mediumID}	GET
/inventory	GET
/inventory/{mediumID}	GET,DELETE
/inventory/serialNr	GET
/inventory/series	POST
/inventory/series/{seriesID}	GET,POST,DELETE
/inventory/series/{seriesID}/media	GET,POST
/borrowers	GET,POST,DELETE
/borrowers/reset	PUT
/borrowers/name	GET
/borrowers/csvFile	GET
/borrowers/{id}	GET
/borrowers/{id}/roles	PATCH

Tabelle 6.1: API-Endpoints

E-Mail	Passwort	User-Roles
admin	user	ADMIN
librarian	user	LIBRARIAN
inventoryHelper	user	INVENTORY_HELPER
loanHelper	user	LOAN_HELPER
user	user	

Tabelle 6.2: Vorinitialisierte Nutzer zum Testen

6.2 Spring-Security

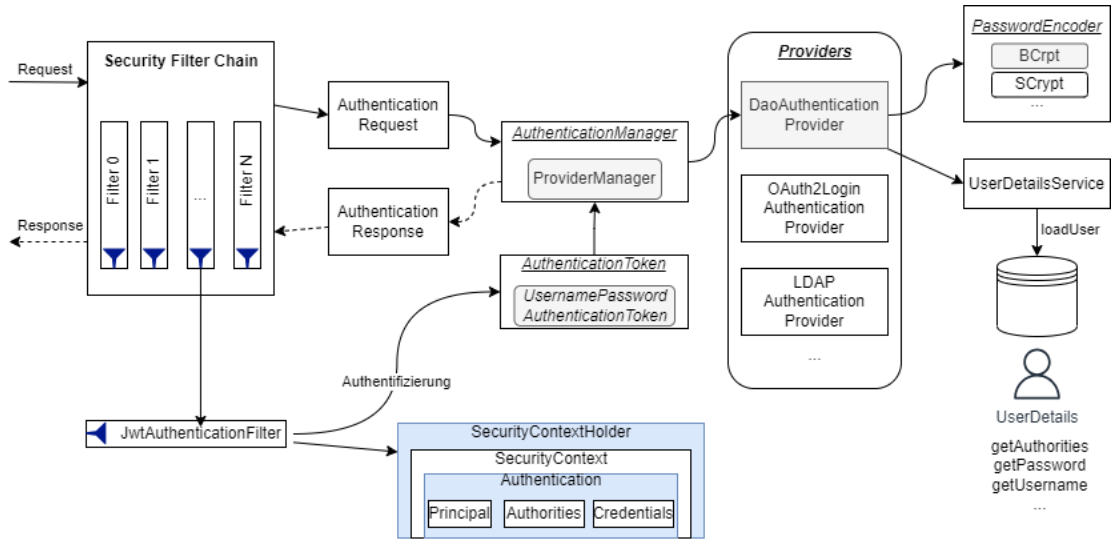

Abbildung 6.3: Spring-Security-Fluss-Architektur. Diagramm übernommen und angepasst von spring-security-tutorial-jwt [17]

In dieser Arbeit wird mithilfe von Spring-Security die API vor einem unautorisierten und nicht authentifizierten Zugriff geschützt. Wird Spring-Security einer Spring-Anwendung hinzugefügt und konfiguriert, werden Filter ”vor” die Anwendung angelegt, welche sämtliche Http-Requests abfangen und bearbeiten gewissermaßen wie Türsteher, die Klienten nur eintreten lassen, wenn sie bestimmte Kriterien erfüllen. Spring-Security ist hochgradig konfigurierbar und bietet diverse Interfaces mit samt gelieferten Implementationen an. In Abbildung 6.3 ist die Fluss-Architektur hinter Spring-Security zu sehen, welche im Folgenden näher beleuchtet wird.

Jede Request muss zuerst eine Kette bestehend aus mehreren Filtern (Filter Chain) passieren. Spring-Security bietet diverse Filter an, von denen einige bereits in der Filter Chain integriert sind und andere durch eine Konfiguration eingebunden werden können. Ob und welche Filter aufgerufen werden, hängt wiederum von der Konfiguration ab. Es lassen sich auch eigene Filter erstellen und in die Filter Chain an gewünschter Stelle einbinden, wie der in dieser Arbeit erstelle und genutzte “JwtAuthenticationFilter“. Die Filter reichen die Request bzw. Response immer zum nächsten Filter in der Kette weiter, nachdem sie ihre Checks durchgeführt haben.

Möchte ein Client Zugriff auf die (private) API, muss zunächst eine Authentifizierung stattfinden, sprich ein Login. Dazu sendet er eine *Authentication Request*, die seine Zugangsdaten enthalten (Email & Passwort). Die Authentication Request durchläuft den Authentifizierungsprozess, der in Abb. 6.3 zu sehen ist. Ist die Authentifizierung erfolgreich, erhält der Client als *Authentication Response* ein JWT (JSON Web Token). Der JWT kann für sämtliche weitere Http-Requests als Beweis seiner Authentifizierung im Http-Header eingefügt werden. Ist die Authentifizierung erfolglos, wirft Spring eine BadCredentialsException, entsprechend wird dies dem Client mitgeteilt in Form eines Http-Status-Code 400 (BAD_REQUEST). Folgend wird der Authentifizierungsprozess näher betrachtet.

Der **JwtAuthenticationFilter** prüft, ob die eingehende Request auf die private API ein JWT besitzt und dieser gültig ist. Ist der JWT gültig, wird ein "Authentication-Token" generiert, welcher dem "SecurityContextHolder" übergeben wird. Somit wird die Request als authentifiziert gesetzt. *„Im SecurityContextHolder speichert Spring Security die Details darüber, wer authentifiziert ist"*[8].

Der **AuthenticationManager** ist ein Interface mit einer einzigen Methode *authenticate (Authentication authentication)*. Die Standardimplementierung ist der ProviderManager [9]. Unter dem AuthenticationManager lassen sich mehrere AuthenticationProviders registrieren. Die Authentication Request wird an den passenden Provider weiter delegiert. Konkret wird in diesem Fall ein *UsernamePasswordAuthenticationToken* erstellt, den der DaoAuthenticationProvider für die Authentifizierung benötigt.

DaoAuthenticationProvider ist eine Standardimplementierung des AuthenticationProvider Interfaces. Es verifiziert den UsernamePasswordAuthenticationToken, welche es vom ProviderManager erhält. Der DaoAuthenticationProvider benötigt einen PasswordEncoder zum Hashen der Passwörter und eine Referenz zum UserDetailsService. Über den UserDetailsService wird der gewünschte User geladen, sodass ein Abgleich der Credentials (Passwort & Username) zwischen UsernamePasswordAuthenticationToken und dem gespeicherten User in der Datenbank stattfinden kann. Schlägt die Authentifizierung fehl, wird eine Exception geworfen. Ist die Authentifizierung erfolgreich, wird ein UsernamePasswordAuthenticationToken zurückgesendet mit den UserDetails und *Authorities*, außerdem wird im Token authenticated auf true gesetzt [11].

Mithilfe von *Authorities* lassen sich die API-Endpoints restriktiver gestalten. Sendet ein authentifizierter User eine HTTP-Anfrage auf Endpoint X, welche die Authority "ADMIN" erwartet, wird jenem User der Zugriff gewährt, wenn er tatsächlich die Authority

bzw. User-Role “ADMIN“ hat. Das heißt, der User muss einerseits authentifiziert sein, aber auch die benötigten Zugriffsrechte für Endpoint X besitzen. Werden die zwei Bedingungen nicht erfüllt, wird der Zugriff verweigert [10].

Bei einem erfolgreichen Login gilt der User als authentifiziert und erhält einen JWT. Jede Request auf einen gesicherten API-Endpunkt erfordert im Http-Header den JWT. Dieser JWT ist der Beweis für die Authentifizierung des Users, so ist es nicht nötig, erneut den ganzen Ablauf vom ProviderManager hin zum UserDetailsService und zurück durch zu laufen. Nichtsdestotrotz muss der JWT auf Gültigkeit überprüft werden. Dies geschieht im JwtAuthenticationFilter. Der Ablauf der Authentifizierung mit Username und Passwort, sowie JWT Generierung findet nur beim Login statt.

6.2.1 Config

Spring-Security lässt sich mittels einer Dependency in der pom.xml leicht hinzufügen, bedarf aber einer geringfügigen Konfiguration.

```
1     <!-- Spring security -->
2     <dependency>
3       <groupId>org.springframework.boot</groupId>
4       <artifactId>spring-boot-starter-security</artifactId>
5     </dependency>
```

Listing 6.5: spring-boot-starter-security dependency

SecurityConfig

Die benötigten Klassen (Abb. 6.3) UserDetailsService, AuthenticationProvider, PasswordEncoder & AuthenticationManager werden mit @Bean instanziiert und verdrahtet.

```
1 \\imports
2 @Configuration
3 @RequiredArgsConstructor
4 public class SecurityConfig {
5     private final BorrowerRepository repository;
6     @Bean
7     public UserDetailsService userDetailsService(){
8         return username -> repository.findBorrowerByEmail(username).
    orElseThrow(() -> new UsernameNotFoundException("User not found"));
9     }
```

```
10
11      @Bean
12      public AuthenticationProvider authenticationProvider(UserDetailsService
        userDetailsService,PasswordEncoder passwordEncoder){
13          DaoAuthenticationProvider authProvider = new
        DaoAuthenticationProvider();
14          authProvider.setUserDetailsService(userDetailsService);
15          authProvider.setPasswordEncoder(passwordEncoder);
16          return  authProvider;
17      }
18      @Bean
19      public PasswordEncoder passwordEncoder() {
20          return new BCryptPasswordEncoder();
21      }
22
23      @Bean
24      public AuthenticationManager authenticationManager(
        AuthenticationConfiguration config) throws Exception {
25          return config.getAuthenticationManager();
26      }
27  }
```

Listing 6.6: SecurityConfig

SecurityFilterConfig

Die SecurityFilterConfig Klasse dient der Konfiguration der FilterChain. In Zeile 12-13 wird mittels *.requestMatchers(AUTH_ WHITE_ LIST)* die Public-API definiert und mit *.permitAll()* der unautorisierte Zugriff. In Zeile 15-21 werden die geforderten Zugriffsrechte der Private-API definiert. Zeile 21-22 sorgt dafür, dass alle anderen Anfragen, abgesehen der Public-API, authentifiziert sein müssen. In Zeile 28 wird der AuthenticationProvider gesetzt und in Zeile 29 wird der JwtAuthenticationFilter der FilterChain hinzugefügt. Die restlich benötigte Konfiguration wird durch die Annotation @EnableWebSecurity von Spirng-Boot übernommen.

```
1  //Imports
2  @Configuration
3  @RequiredArgsConstructor
4  @EnableWebSecurity(debug = true)
5  @EnableMethodSecurity // Enable @PreAuthorize at Method-Level
6  public class SecurityFilterConfig {
7     //Fields...
```

```
8       @Bean
9       public SecurityFilterChain securityFilterChain(HttpSecurity http) throws
        Exception {
10          http.csrf().disable()
11                  .authorizeHttpRequests()
12                  .requestMatchers(AUTH_WHITE_LIST)//Public API
13                  .permitAll()
14                  //Configure Borrower-Admin endpoints
15                  .requestMatchers(HttpMethod.POST, "/api/v1/borrowers/**").
        hasAuthority(ADMIN)
16                  .requestMatchers(HttpMethod.DELETE, "/api/v1/borrowers/**").
        hasAuthority(ADMIN)
17                  .requestMatchers(HttpMethod.PUT, "/api/v1/borrowers/**").
        hasAuthority(ADMIN)
18                  //Configure Inventory endpoints
19                  .requestMatchers( "/api/v1/inventory/**").hasAnyAuthority(
        ADMIN,LIBRARIAN,INVENTORY_HELPER)
20                  .requestMatchers( "/api/v1/loan/histories/*").hasAnyAuthority
        (INVENTORY_HELPER,ADMIN,LIBRARIAN,LOAN_HELPER)
21                  .requestMatchers( "/api/v1/loan/**").hasAnyAuthority(ADMIN,
        LIBRARIAN,LOAN_HELPER)
22                  .anyRequest()
23                  .authenticated()
24                  .and()
25                  .sessionManagement()
26                  .sessionCreationPolicy(SessionCreationPolicy.STATELESS)
27                  .and()
28                  .authenticationProvider(authenticationProvider)
29                  .addFilterBefore(jwtAuthFilter,
        UsernamePasswordAuthenticationFilter.class);
30          return http.build();
31      }
32  }
```

Listing 6.7: SecurityFilterConfig

6.2.2 JWT

JSON Web Token ist ein Standard (RFC 7519) zur kompakten und sicheren Übermittlung von Informationen zwischen Kommunikationspartner als JSON-Object. JWT werden mittels eines Secret Key oder public/private key pair signiert. Durch die Signatur kann der Empfänger sichergehen, dass der Sender der ist, für den er sich ausgibt. In der codierten kompakten Form besteht ein JWT aus drei Teilen, Header, Payload und der Signatur, welche durch ein Punkt getrennt werden. So entsteht folgende Struktur; xxxx.yyyy.zzzz

Dekodiert sehen die Teile in der Regel wie folgt aus:

```
Beispiel Header:
{
  "alg": "HS256",
  "typ": "JWT"
}
Beispiel Payload: Informationen zum User, Ausstellung-/Ablaufdatum des Tokens
{
  "sub": "1234567890",
  "name": "John Doe",
  "iat": 1516239022
  ...
}
Beispiel Signatur:
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
your-256-bit-secret
)
```

Listing 6.8: Beispiel JWT (Dekodiert)

Bei einem erfolgreichen Login erstellt und signiert der Server ein JWT und sendet diesem dem Client. Der Client fügt für weitere Http-Anfragen auf private Endpoints den JWT im Authorization header ein: Authorization: Bearer <token>

Für die Generierung und Validierung der JWT wurde eine Serviceklasse implementiert. Der generierte JWT besitzt eine Gültigkeit von 12 Stunden (Listing 6.9, Zeile 24). Zum Signieren wird ein 256-Bit Secret Key verwendet. Zum Generieren eines JWT werden die erwähnten Dependencies in Abschn. 6.1.1 benutzt. So ist in Listing. 6.9, Zeile 18 zu sehen, wie mittels der Utility-Klasse *Jwts* sehr einfach ein JWT erstellt wird.

```java
//Imports
@Service
public class JwtService {
    //Fields
    public String extractUsername(String jwt) {
        return extractClaim(jwt,Claims::getSubject);
    }

    public <T> T extractClaim(String jwt, Function<Claims,T> claimsResolver){
        final Claims claims = extractAllClaims(jwt);
        return claimsResolver.apply(claims);
    }

    public String generateJwt(UserDetails userDetails){
        return  generateJwt(new HashMap<>(),userDetails);
    }
    public String generateJwt(Map<String,Object> extraClaims, UserDetails
    userDetails){
        return Jwts
                .builder()
                .setClaims(extraClaims)
                .setHeaderParam("typ","JWT")
                .setSubject(userDetails.getUsername())
                .setIssuedAt(new Date(System.currentTimeMillis()))
                .setExpiration(new Date(System.currentTimeMillis() + 12*HOUR)
    )
                .signWith(getSigningKey(), SignatureAlgorithm.HS256)
                .compact();
    }
    public boolean isJwtValid(String jwt, UserDetails userDetails){
        final String username = extractUsername(jwt);
        return (username.equals(userDetails.getUsername())) && !isJwtExpired(
    jwt);
    }

    private boolean isJwtExpired(String jwt) {
        return extractExpiration(jwt).before(new Date());
    }

    private Date extractExpiration(String jwt) {
       return extractClaim(jwt, Claims::getExpiration);
    }


```

```
42      private Claims extractAllClaims(String jwt) {
43          return Jwts
44                  .parserBuilder()
45                  .setSigningKey(getSigningKey())
46                  .build()
47                  .parseClaimsJws(jwt)
48                  .getBody();
49      }
50      private Key getSigningKey() {
51          byte[] keyBytes = Decoders.BASE64.decode(SECRET_KEY);
52          return Keys.hmacShaKeyFor(keyBytes);
53      }
54  }
```

Listing 6.9: JwtService

6.2.3 Erwähnungen

Als Informationsquellen und Referenz für die Implementierung von Spring-Security wurden neben der offiziellen Spring-Security Dokumentation verschiedenen Quellen benutzt. Maßgeblich für die Implementierung wurde jedoch Bezug zu Tutorials von @amigoscode genommen.

6.3 Frontend

Die Frontendkomponenten (Views und Vue-SFC) funktionieren prinzipiell alle gleich. Der Userinput wird aufgegriffen, in eine für das Backend passende Request verpackt und mittels Axios versendet. Die Response wird verarbeitet und gerendert. Ähnlich wie im Backendteil kann und wird nicht auf alle Komponenten eingegangen.

6.3.1 Vue-Projekt aufsetzen

Ein Vue.js Projekt lässt sich leicht aufsetzen. Vorausgesetzt wird die Installation von Node.js. Mittels *npm create vue@latest* wird im aktuellen Verzeichnis ein neues Projekt erstellt. Bei der Erstellung kann aus einer kleinen Auswahl an Bibliotheken entschieden werden, ob jene Bibliotheken mit installiert und konfiguriert werden sollen. Einzig Vue-Router wurde hierbei ausgewählt. Alle anderen benutzten Bibliotheken wurden nachträglich installiert (Abb. 6.4).

- **@fortawesome & @heroicons:** Für diverse Icons.
- **@headlessui:** Unstyled-UI Komponenten (MenuDropDown).
- **jwt-decode:** JWT Parser.
- **papaparse:** CSV Parser.
- **vee-validate & yup:** Html-Form Validierung.
- **vuex:** Für das State-Management des Frontends.
- **vue-router:** Für das Routing
- **qs:** Querystring parsing und *stringifying* library

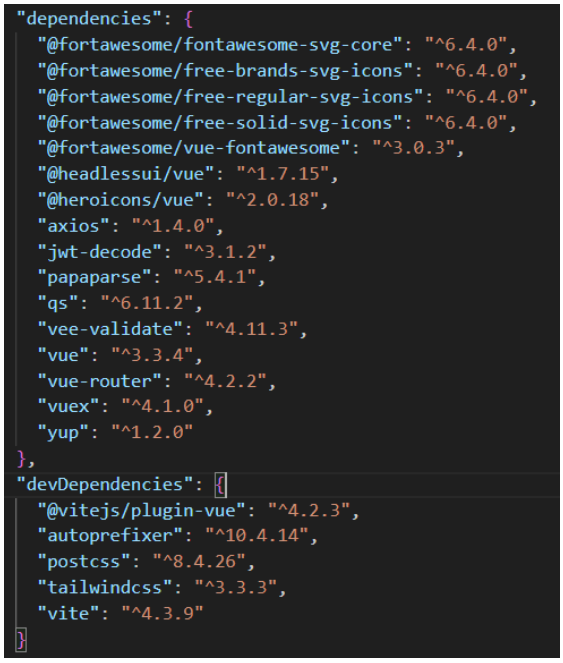

Abbildung 6.4: Vue-Dependencies

6.3.2 Vue-Router

Unter dem Package "router" ist die Config-Datei (index.js) für Vue-Router zu finden, welche Vue bei der Initialisierung automatisch mitgeneriert hat. In jener Datei werden sämtliche internen Routes definiert sowie die dazu gehörige View zum Rendern. Wird der entsprechende "Path" aufgerufen rendert Vue die dazugehörige View (Abb. 6.5). Um den Router in einer Komponente zu nutzen, muss zuerst der Router importiert werden durch *"import {useRouter} from 'vue-router';"*. Anschließend wird eine Konstante definiert mit *"const router = useRouter();"*. Das Navigieren zu anderen Views bzw. Routes geschieht durch *router.push(<path>)* oder *router.push(<name>)* (Listing 6.12, Zeile 16).

6.3.3 View Beispiel: LoanView

Als Implementierungsbeispiel einer Vue-SFC wird hier die *LoanView* vorgestellt. Die LoanView eignet sich sehr gut als Vorzeigebeispiel. Anhand der LoanView kann eine Kernfunktionalität die Ausleihe von Medien präsentiert werden sowie Vue spezifische Funktionen. Die LoanView besteht im Wesentlichen aus zwei Teilen, dem InputField zur Eingabe der Medien-Id für die Ausleihe und der Komponente "<AsyncBorrower>" zur

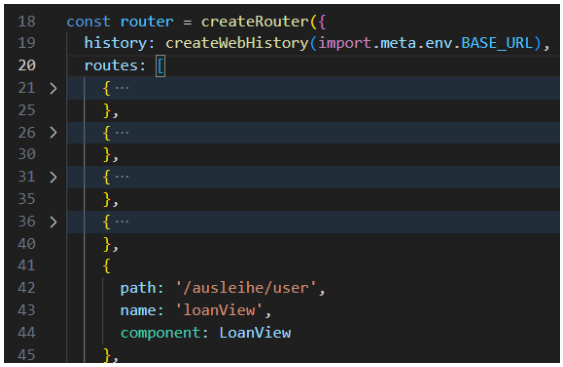

Abbildung 6.5: Router-Snippet

Darstellung der Userinfos und aktuell ausgeliehenen Medien eines *Borrowers*.
Die <AsyncBorrower> Komponente sendet, wenn sie gerendert wird, eine Request ans Backend, um die Userinfos und aktuell ausgeliehenen Medien zu erfragen. Dafür wird ihr die User-Id übergeben (Zeile 7, Listing 6.10), welche aus der URI-Query entnommen wird. Bei Eingabe der Medien-Id triggert die Funktion "loanMedia" (Zeile 31-50, Listing 6.10), die mittels Axios eine Http-POST Request samt Parameter (User-ID & Medien-ID) ans Backend sendet. Sendet das Backend als Response einen Fehler, werden im Catch-Block die Variablen *err* und *errorMessage* gesetzt. Mittels *v-if*, zu sehen in Listing 6.10 Zeile 3, bietet Vue *bedingtes Rendering* an. Ein Block mit v-if wird nur dann gerendert, wenn der Ausdruck der Direktive einen wahrheitsgemäßen Wert zurückgibt. In diesem Fall, wenn *err* true ist, wird die *errorMessage* gerendert. Ist die Response positiv, bedeutet es, dass die Ausleihe bzw. Rücknahme erfolgreich war. Die Informationen in der <AsyncBorrower> Komponente sind daher veraltet und ein neu rendern wird benötigt. Um nicht die gesamte Seite neu laden zu müssen, sondern genau die Komponente die ein neu rendern benötigt, kann jener Komponente das Attribut *:key=<value>* gegeben werden (Listing 6.10, Zeile 7). Jedes Mal, wenn sich der :key-<value> ändert, triggert Vue ein neu rendern der Komponente, die wiederum in diesem Fall eine erneute Abfrage ans Backend sendet, um den aktuellsten Stand zu erfragen. Als Anstöß zum neu rendern, dient die Funktion *forceRerender*, die den *componentKey* inkrementiert. Die Funktion *forceRerender* wird bei einer erfolgreichen Request in Zeile 46, Listing 6.10 getriggert. In Abb. 6.6 ist ein Screenshot der implementierten LoanView zu sehen. In der Browserleiste ist der Pfad zur LoanView *ausleihe/user* und die Query *user?id=2* für die AsyncBorrower-Komponente zu sehen. Außerdem ist der Inputfield für die Medien-ID und die gerenderte AsyncBorrower-Komponente darunter zu sehen.

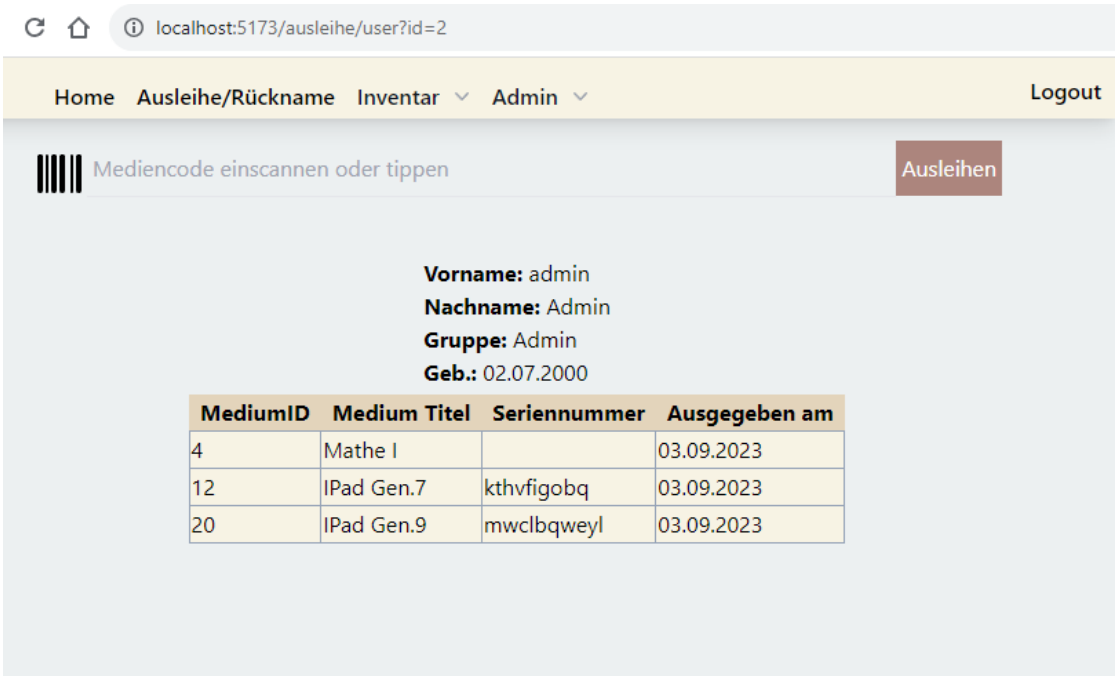

Abbildung 6.6: LoanView Screenshot

```vue
<template>
<!-- Reduzierte Darstellung des templates für bessere Lesbarkeit  -->
<div v-if="err" >
    <p>{{ errorMessage }}</p>
</div>
<Suspense>
    <AsyncBorrower :key="componentKey" :user-id="route.query.id" />
    <template #fallback>
        <p>loadning...</p>
    </template>
</Suspense>
</template>

<script setup>
import AsyncBorrower from '../components/AsyncBorrower.vue';
import { ref } from 'vue';
import axios from "axios";
import { useRoute } from "vue-router";
import authHeader from '../services/authHeader';

const err = ref(false);
const errorMessage = ref("");
```

```
const componentKey = ref(0);
const route = useRoute();
const mediaID = ref("");
//Re-render component if necessary
const forceRerender = () => {
    componentKey.value += 1;
};

const loanMedia = async () => {
    err.value = false;
    if (mediaID.value !== "") {
        const borrowerID = route.query.id;
        let config = {
            headers: authHeader(),
            params: { borrowerID: borrowerID, mediumID: mediaID.value }
        }
        await axios.post('/api/v1/loan', null, config)
            .catch(function (error) {
                if (error) {
                    err.value = true;
                    errorMessage.value = error.response.data.message
                }
            });
        forceRerender();
        //reset mediaID to blank
        mediaID.value = "";
    }
}
</script>
```

Listing 6.10: LoanView-Snippet

6.3.4 Vuex Store

Wie im Kapitel 5 Abschnitt 5.3 erwähnt, wird Vuex für das *State-Management* des Frontend benötigt. Konkret, ob ein User eingeloggt ist oder nicht. Die Implementierung des Vuex-Store setzt sich aus zwei Dateien zusammen, die unter dem Package "store" zu finden sind, *index.js* (Abb. 6.7) & *auth.module.js* (Listing 6.11). Die Index-Datei ist der Store und auth.module ein registriertes Modul im Store . Es lassen sich unterschiedliche Module zu unterschiedlichen Zwecken unter einem Store registrieren. Das auth.module besteht aus drei Teilen, dem *state*, den *actions* und den *mutations*.

```
1  import { createStore } from "vuex";
2  import { auth } from "./auth.module";
3
4  const store = createStore({
5    modules: {
6      auth,
7    },
8  });
9
10 export default store;
```

Abbildung 6.7: index.js File (Vuex-Store)

Einfach gesagt, triggert eine *action* eine *mutation* aus, die den *state* manipuliert. In einer *action* wird über die Vuex-Funktion *commit(<mutation>)* eine mututation getriggert. Eine *action* selbst kann durch alle Komponenten getriggert werden, die den Store nutzen möchten. Im auth.module sind Login und Logout implementiert in Form von *actions*. Die LoginView nutzt die Action *login*, um das Login anzustoßen. Dies geschieht durch *store.dispatch("auth/login", form)* (Zeile 10, Listing 6.12). Hierbei ist *"auth"* das Modul im Store, *"login"* die action im Modul und *"form"* die Usercredentials.

Das auth.modul nutzt intern den AuthService, der zwei Funktionen erfüllt. Erstens über Axios eine Http-Request für das Login ans Backend zu senden. Im Falle, dass in der Response ein JWT ist, sprich die Authentifizierung war erfolgreich, speichert es den Token im Localstorage des Browsers. Zweites beim ausloggen den JWT wieder aus dem Localstorage zu entfernen. Im Falle, dass AuthService kein Error zurück an das auth.modul sendet und der Login kein Initial-Login ist, gilt der Login als Erfolgreich und dem entsprechend wird über *commit("loginSuccess")* die passende mutation getriggerd, welche den state passend setzt. Des Weiteren wird ein Timer für das automatische Ausloggen initiiert (Listing 6.11, Zeile 28-30). Der Timer triggert die action *auto_logout* aus, sobald der JWT seine Gültigkeit verliert. Das führt dazu, dass der State, konkret *autoLogout*, auf true gesetzt wird. Die Wurzelkomponente *App.vue* überwacht *autoLogout*, sodass bei einer Änderung der Client ausgeloggt wird. Der State wird auch von Vue-Router vor jedem Navigieren zu einer Route geprüft, ob der aktuelle User eingeloggt ist. Wenn nicht, wird der User zur LoginView umgeleitet (Listing 6.13). So wird gewährleistet, dass nur ein User, der tatsächlich eingeloggt ist, die Applikation nutzen darf. Für die Vuex-Store Implementierung wurde bezkoder[14] als Referenz genommen.

```
//Imports...
//Auto-LogOut Timer
let timer ='';
const user = JSON.parse(localStorage.getItem('user'));
const initialState = user
  ? { status: { loggedIn: true,autoLogout:false }, user }
  : { status: { loggedIn: false ,autoLogout:true}, user: null };

export const auth = {
  namespaced: true,
  state: initialState,
  actions: {
    login({ commit,dispatch}, user) {
      return AuthService.login(user).then(
        user => {
          if(user.initialLogin){
            commit('loginFailure');
            return Promise.resolve(user);
          }
          else{
            commit('loginSuccess', user);

            const decoded =jwt_decode(user.jwt)
            let expDate = new Date(decoded.exp *1000)
            let now = new Date();
            let expiresIn = expDate.getTime() - now.getTime()

            timer = setTimeout(()=>{
              dispatch("auto_logout")
            },expiresIn )
            return Promise.resolve(user);
          }
        },
        error => {
          commit('loginFailure');
          return Promise.reject(error);
        }
      );
    },
    //More Actions...logout,auto_logout
  },
  mutations: {
    loginSuccess(state, user) {
      state.status.loggedIn = true;
```

```
45        state.status.autoLogout = false;
46        state.user = user;
47      },
48      //More mutations...
49    }
50  };
```

Listing 6.11: auth.module-snippet

```
1  <!-- template Omitted for readability and brevity -->
2  <script setup>
3  //More imports...
4  import { useStore } from 'vuex';
5  //more Variables, consts
6  const store = useStore();
7  //dispatch 'auth/login' Action to Vuex Store. If the login is successful,
       go to Profile Page, otherwise, show error message.
8  const handleLogin = async (form) => {
9      //login
10     store.dispatch("auth/login", form).then(
11         (response) => {
12             if (response.initialLogin) {
13                 showInitLogin.value = true;
14             }
15             else {
16                 router.push("/myProfile")
17             }
18         },
19         (error) => {
20             message.value = error;
21         }
22     )
23 };
24 </script>
```

Listing 6.12: LoginView-Snippet

```
1  //Before routing check if user is logged-in. If not redirect to loginView
2  router.beforeEach(async(to, from) => {
3      const loggedIn = store.state.auth.status.loggedIn
4      if(!loggedIn &&
5      // Avoid an infinite redirect
6      to.name !== 'login'
7      ){
8      return {name: 'login'}
9      }
10 })
```

Listing 6.13: Vue-Router-Snippet beforeEach Funktion des Routers

7 Evaluation

In diesem Kapitel wird kurz erklärt, wie die Applikation getestet wurde und im Anschluss folgt die Nutzerbefragung, die an der Schule stattfand.

7.1 Tests

Die REST-API wurde während der Entwicklung stetig mittels Swagger und Postman getestet. Für die Repository und Service-Klassen wurden JUnit Tests hinzugefügt, die bzgl. Umfang und Komplexität durchaus ausgeprägter ausfallen könnten. Es wurden keine Frontendtests implementiert, da hier keinerlei *Know-How* vorhanden war und eine Recherche als zu zeitintensive angesehen wurde. Ein ausgeprägtes und professionelles Testen ist sehr aufwändig in Recherche und Implementation, daher wurde zu Gunsten einer kürzeren Bearbeitungszeit darauf verzichtet.

7.2 Nutzerbefragung

Die Applikation wurde dem Admin, der Bibliothekarinin und einem Lehrer vorgestellt. Es wurden Fragen zur Ausleihfunktion und Inventarisierung der Medien sowie Administration der Nutzer gestellt.

Ausleihe & Rückgabe:

1. Gefällt Ihnen die Ausleihe und Rückgabefunktion?
    - Bibliothekarin: ☆☆☆☆☆
    - Admin: ☆☆☆☆☆
    - Lehrer: ☆☆☆☆☆

2. Ist die Nutzung einfach(5)/schwer(0)?
   - Bibliothekarin: ★★★★★ (Kommentar: Sehr übersichtlich)
   - Admin: ★★★★★
   - Lehrer:★★★★★
3. Verbesserungen: „Schnellrücknahme“ - Nur die MedienID einscannen ohne Namenssuche zur Rücknahme.

Inventarverwaltung:

1. Gefällt Ihnen das Aufnehmen von neuen Medien ?
   - Bibliothekarin: ★★★★★
   - Admin: ★★★★★
   - Lehrer: ★★★★★
2. Gefällt Ihnen das Löschen?
   - Bibliothekarin: ★★★★★
   - Admin: ★★★★★
   - Lehrer: ★★★★★
3. Gefällt Ihnen die Inventarübersicht?
   - Bibliothekarin: ★★★★☆ (Kommentar: Preis Anzeige wäre toll)
   - Admin: ★★★☆☆ (Kommentar: Kategorien, Mathe,Deutsch etc. wären gut)
   - Lehrer: ★★★★☆
4. Das Verwalten des Inventars ist generell einfach (5) / schwer (0) verständlich?
   - Bibliothekarin: ★★★★★ (Kommentar: Ausleihhistorie ist toll)
   - Admin: ★★★★★
   - Lehrer: ★★★★★

5. Verbesserungen: „Klassenübersicht“ - Eine Auflistung der Schüler in Klasse X mit ihren ausgeliehen Medien, die druckbar ist. „Kategorien“ für das Inventar (Mathe, Deutsch, etc.).

User-Verwaltung (Nur für den Admin):

1. Gefällt Ihnen der User-Import: ★★★★½
2. Ist der User-Import simple (5) oder schwer (0): ★★★★★
3. Gefällt Ihnen die „User-Import-Preview“: ★★★★★
4. Ist die Rechtevergabe simple (5) oder schwer (0): ★★★★★
5. Verbesserungen: „Frontend-Design“ - könnte *besser* sein. Mehr bzw. anderen Style.

7.3 Fazit

Anhand der Nutzerbefragung lässt sich sagen, dass die Funktionalität (Backend) gut ankommt. Die Anwendung ist leicht verständlich und nutzbar. Zum Frontend lässt sich sagen, dass es zwar nicht überladen wirkt, aber auch nicht besonders viel *Style* hat (*Design ist simpel, erfüllt seinen Zweck*).

Zu den Verbesserungsvorschlägen sei Folgendes gesagt:

- **Preisanzeige in der Inventarübersicht:** Sehr leicht implementierbar. Die benötigte Information wird dem Client bereits mitgegeben, jedoch nicht gerendert. In der Vue-SFC <AsyncInventoryList> muss im <template> die Tabelle um ein weiteres Feld für die Preisanzeige ergänzt werden.
- **Kategorien in der Inventarübersicht:** Dieses Feature kann Frontend technisch implementiert werden. Jede Medienreihe besitzt das Attribut *Fächer/Subjects*. Anhand dieses Attributes ließe sich eine Kategorisierung im Frontend vornehmen lassen. Die Filterung bzw. Kategorisierung würde somit rein auf den Client verlegt werden. Das kann bei einer großen Anzahl an Medienreihen zu schlechterer Performance führen. In diesem Fall wäre eine Erweiterung der API inkl. Service und Repository-Abfrage besser, sodass der Server die Last übernimmt.

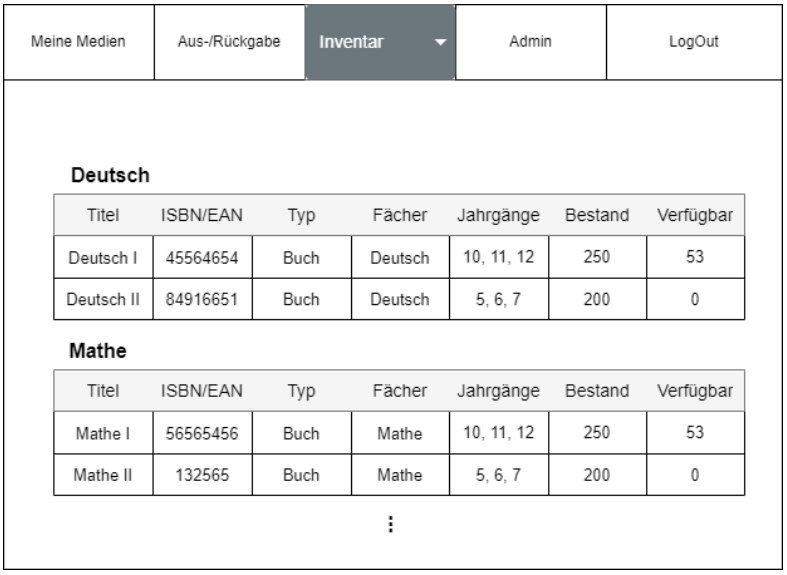

Abbildung 7.1: Wireframe - Kategorien in der Inventarübersicht

- **Schnellrücknahme:** Dieses Features lässt sich relativ einfach implementieren. Hierzu muss die Klasse *LoanService* um eine Methode erweitert werden, die als Parameter die MedienId übergeben bekommt. Über die MedienId wird das Medium gesucht. Falls das Medium ausgeliehen ist, kann der *Borrower* ebenfalls über das Medium gesucht werden (Bidirektionale Relation). Medium & Borrower kann der bereits vorhandenen Methode *unloanFromBorrower(Borrower borrower, Medium medium)* übergeben werden. Außerdem wird ein neuer API-Endpoint im *LoanController* für die Schnellrücknahme benötigt, der leicht hinzuzufügen ist. Für das Frontend kann, grob erklärt, die View für die Ausleihe so angepasst werden, dass zwischen *Usersuche*-Inputfield und *Schnellrücknahme*-Inputfield per Buttonclick gewechselt wird.

- **Klassenübersicht:** Die Implementierung dieses Feature gestaltet sich komplizierter. Das Frontend & Backend müsste angepasst werden. Über das Attribut *BorrowerGroup* lässt sich eine Gruppierung der *Borrower* bzgl. Gruppe/Klasse realisieren. Da eine bidirektionale Relation zwischen *Borrower* & *Medium* existiert, ist die Information "ausgeliehene Medien" über den *Borrowern* extrahierbar.

  Das **Frontend** benötigt zwei neue Views, eine für die allgemeine Klassenauflistung und die andere für die interne individuelle Klassensicht, die als Liste druckbar sein würde.

Zum **Backend** sei Folgendes gesagt: Es empfiehlt sich eventuell für die *Borrower-Group* eine separate Tabelle zu erstellen, statt wie jetzt nur als Attribut zu definieren. Durch eine separate *BorrowerGroup*-Tabelle sind die Datenbankanfragen für die erste View (allgemeine Klassenauflistung) effizienter, da jene Tabelle definitiv weniger Einträge haben wird als die *Borrower*-Tabelle. Somit muss nicht die gesamte Borrower-Tabelle nach BorrwerGroup abgefragt werden. Für die individuelle Klassensicht würde ein join zwischen der BorrowerGroup und Borrwer-Tabelle stattfinden. Die Ergebnisse müssten in ein passendes DTO abgebildet werden. Das DTO würde zum Client gesendet werden, der die Informationen passend rendert.

Um die druckbare Liste zu implementieren, kann die Open-Source Libary itext genutzt werden. Die Libary lässt sich einfach per Dependency in die pox.xml einfügen. Mit itext lassen sich Pdf´s generieren oder bestehende Pdf´s bearbeiten. Wie konkret itext zu implementieren ist, müsste untersucht werden.

8 Fazit / Zusammenfassung

In dieser Arbeit wurde im Kontext einer Schule eine Full Stack Medienausleih-Applikation mit Spring-Boot und Vue.js entwickelt. Die (Web)Applikation ist in der Lage unterschiedlichste Medien zu inventarisieren inkl. Seriennummern, falls dies gewünscht ist. Letzteres ermöglicht die im Abschnitt Ziel 1.2 erwähnte CSV-Datei für die Verknüpfung zwischen Seriennummer und Ausleiher überflüssig zu machen. Jedes Medium führt eine eigene *Ausleihhistorie*, sodass ersichtlich ist, welcher Ausleiher jenes Medium für welche Zeitspanne ausgeliehen hat. Dies *kann* praktisch für den Fall sein, dass ein Medium unbemerkt beschädigt entgegen genommen wird. Es lässt sich somit rückwirkend einsehen, wer jenes Medium im Besitz hatte und es in jenem Zustand abgegeben hat. Die Applikation ermöglicht die Vergabe von *User-Roles* (Abschn. 3.2) durch den Admin, sodass Inventarmanagement sowie Ausleihe & Rückgabe der Medien auf mehrere Angestellte aufteilbar ist. Dadurch kann vermieden werden, den Betrieb der Bibliothek von einer einzigen Person abhängig sein zu lassen. Fällt eine Person aus, kann eine andere einspringen. Gemäß der Nutzerumfrage ist die Nutzung der Applikation einfach. Ein aufwändiges einarbeitet ist daher nicht nötig. Die Vergabe von User-Roles ist auch bei der jährlichen Inventur hilfreich, da *Inventur-Helfer* Rechte vergeben werden können. Das Inventarisieren neuer Medien und löschen älterer aus dem Bestand kann daher effizienter gestaltet werden. Ist die Inventur erledigt, kann der Admin jenen Helfern die Inventur-Helfer Rechte wieder entziehen. Jeder Nutzer, auch Schüler, erhält Log-In-Daten (Email & Passwort), mit denen er sich anmeldet kann und entsprechend seiner User-Roles eine zugriffsbeschränkte Sicht hat. Für Schüler kann der Zugang zum System durchaus praktisch sein, da sie ihre ausgeliehenen Medien online einsehen und so im besten Fall eine vollständige Rückgabe rechtzeitig vorbereiten können. Um einen unautorisierten & nicht authentifizierten Zugriff auf die API zu verhindern, wurde die API mit Spring-Security gesichert. Sobald sich ein Client erfolgreich authentifiziert, generiert das Backend ein JWT und übermittelt es dem Client. Der JWT dient ihm fortan für eine bestimmte Zeitspanne als Beweis seiner Identität für jede weitere Anfrage.

Abschließend lässt sich sagen, dass die Applikation gut ankommt hinsichtlich der Funktionen, aber durchaus mehr Schliff bzgl. Frontend-Design vertragen kann. Mit mehr Expertise & Erfahrung in Frontend-Design ließe sich dies gewährleisten. Aus Gründen geringer Kenntnisse bzgl. Software-Testing wurde und konnte nur geringfügig getestet werden. Ohne eine professionelle Testung und Sicherheitsanalyse sollte davon abgesehen werden, die Applikation in Produktion zu nehmen. Die ermittelten User-Stories in Abschnitt 3.3.1 wurde alle erfüllt. Durch die Evaluation traten neue Anforderungen auf, die prinzipiell umsetzbar sind. Eine erste Beschreibung der Umsetzung wurde im Fazit der Evaluation gegeben (Abschn. 7.3). Ein wichtiger Punkt muss hier noch angesprochen werden. Die Applikation weist in der User-Verwaltung ein Problem auf und zwar, dass der Admin sich praktisch selbst aus dem System löschen kann. Zwar wird er frontendtechnisch darauf hingewiesen, dass er dabei ist, sich selbst zu löschen, doch aufgehalten wird er nicht. Im Falle eines einzigen Admins, der sich selbst löscht, ist das System nicht mehr verwaltbar. Im Abschnitt Ausblick 8.2.5 wird darauf näher eingegangen.

Das Projekt kann in GitHub eingesehen werden. Eine README-Datei ist in GitHub eingepflegt, welche die Nutzung und Installation der Applikation für die lokale Testung erläutert.

8.1 Retrospektive

Retrospektiv betrachtet kann gesagt werden, dass die Entwicklung des Frontends unterschätze wurde. Frontend-Entwicklung ist nicht unbedingt weniger aufwendig oder komplex als Backend-Entwicklung. Nicht umsonst existiert der spezialisierte Beruf des Frontend-Entwickler, die oft eng mit ein oder mehreren UX/UI-Designer zusammen arbeiten, um für den Nutzer ein nicht nur optisch ansehnliches, sondern auch leicht verständliches User-Interfaces zu entwickeln. Ohne die Nutzung eines Frontend-Framework wie Vue.js wäre die Entwicklung des Frontends dieser Arbeit undenkbar. Vue.js ist eine Erleichterung für die Frontend-Entwicklung wie es Spring bzw. Spring-Boot für das Backend ist.

8.2 Ausblick

Folgend soll ein kurzer Ausblick auf einige Punkte gegeben werden, die ungeklärt geblieben sind bzw. nicht weiter untersucht wurden.

8.2.1 Deployment

Sowohl Frontend als auch das Backend müssen auf Servern *deployed* werden. Es stellt sich die Frage, wie die Applikation deployed werden soll. Beide benutzten Frameworks thematisieren das Deployment in ihrer Dokumentation (Deployment-Spring-Boot, Vue-Deployment). Wie genau das Deployment funktioniert, müsste ferner untersucht werden.

8.2.2 Datenbank

Die In-Memory-H2 Datenbank eignet sich nur für das Entwickeln und Testen einer Applikation. Im Falle eines Deployment und Nutzung der Applikation sollte die Datenbank daher bspw. gegen eine Postgres-Datenbank ausgetauscht werden. Wie in Abschn. 2.2.2 erwähnt, ist der Austausch der Datenbank relativ einfach. Zuvor muss die Datenbank jedoch erstellt werden, um das Anbinden mittels der application.yml Datei zu ermöglichen.

8.2.3 JWT Refresh-Token

Wie Abschn. 6.2.2 erwähnt, generiert das Backend ein JWT bei einem erfolgreichen Log-in. Der JWT dient dem Client als Beweis seiner Identität und gewährleistet ihm Zugriff auf die private API. Läuft das Verfallsdatum des JWT ab, wird der Client automatisch ausgeloggt. Eine alternative wäre die Implementierung eines *Refresh-Token*. Grob angerissen sehe dies wie Folgt aus. Bei einem initialen Log-in wird nicht ein einziger JWT, sonder zwei unterschiedliche JWTs generiert. Ein *Access-Token* und ein *Refresh-Token*. Der Access-Token dient dem Zugang zur API und besitzt ein kurzes Verfallsdatum (wenige Stunden). Der Refresh-Token wird im Local Storage gespeichert. Es besitzt ein langes Verfallsdatum (Wochen oder Monate) und dient dem generieren eines neuen Access-Token, sobald der aktuelle Access-Token ausläuft. Diese Variante bietet mehr Sicherheit.

Wird das Verfallsdatum eines Access-Token klein gehalten, ist die Zeitspanne, in der jemand mit einem kompromittierten Access-Token Schaden anstellen kann, ebenfalls klein. In der Variante mit einem JWT muss der Client sich alle N Stunden erneut einloggen und seine Nutzerdaten eingeben. Mit der alternativen Variante bedarf es einem erneuten Log-in erst, wenn der Refresh-Token ausläuft. Das kann für eine bessere User-Experience sorgen. Problematisch wird es, wenn der Refresh-Token kompromittiert wird, da derjenige dann lange unbefugten Zugriff hätte. Was gute Verfallsdaten für die Tokens sind und wie konkret eine Implementierung aussieht, müsste genauer untersucht werden.

8.2.4 Account zurücksetzen via Mail

Vergisst ein Nutzer sein Passwort, besteht aktuell nur die Möglichkeit, durch ein Zurücksetzen des Accounts über den Admin wieder Zugang zu erhalten. Setzt der Admin ein oder mehrere Accounts zurück, wird eine CSV-Datei generiert mit nummerischen *Einmal-Passwörtern* pro zurückgesetzten Account. Diese Einmal-Passwörter muss der Admin an jene Nutzer weiterleiten. Eine *Passwort vergessen* Option wurde nicht implementiert. Um die Funktion anbieten zu können, müsste in Erfahrung gebracht werden, wie das Backend E-Mails mit den Informationen zur Zurücksetzung an den Nutzer senden kann.

8.2.5 Problem mit der User-Verwaltung

Wie im Fazit erwähnt, hat die Applikation aktuell ein Problem mit der User-Verwaltung. Die Applikation kann nicht verwaltbar werden, wenn der einzige Admin sich selbst löscht. Das Problem entsteht auch, wenn der letzte Admin seine Zugangsdaten vergisst. Letzteres ließe sich relativ gut angehen, wenn es eine *Account zurücksetzen via Mail* Funktion gäbe. Wenn ein Admin dabei ist, sich selbst zu löschen, wird zwar im Frontend darauf hingewiesen, jedoch wird der Vorgang nicht aufgehalten. Praktisch ließe sich im Backend mit wenigen Zeilen Code, der oder die Admins aus dem Löschvorgang ausschließen lassen, doch sobald ein Admin die Schule verlässt, muss er oder sie aus dem System entfernbar sein. Sollte es einen nicht löschbaren Super-Admin geben, dessen Zugangsdaten von Admin zu Admin weitergegeben werden? Reicht es aus, nur den letzten Admin nicht löschbar zu machen, bis ein zweiter auserwählt wurde? Diese Problemstellung bedarf einer genaueren Untersuchung, gefolgt von einer Implementierung.

Literaturverzeichnis

[1] *Bidirectional Associations JPA/Hibernate*. – URL https://www.baeldung.com/jpa-hibernate-associations. – [Online; accessed 20-07-2023]

[2] *Data-Transfer-Object*. – URL https://www.baeldung.com/java-dto-pattern. – [Online; accessed 20-07-2023]

[3] *Introduction to Spring Framework*. – URL https://docs.spring.io/spring-framework/docs/3.2.x/spring-framework-reference/html/overview.html. – [Online; accessed 20-07-2023]

[4] *Spring-Container*. – URL https://docs.spring.io/spring-framework/reference/core/beans/basics.html. – [Online; accessed 20-07-2023]

[5] *Spring-doc-pageRequest*. – URL https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/domain/PageRequest.html. – [Online; accessed 01-08-2023]

[6] *Spring-docs-beans*. – URL https://docs.spring.io/spring-framework/reference/core/beans/basics.html. – [Online; accessed 20-07-2023]

[7] *Spring-docs-jpa-querykeywords*. – URL https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#repository-query-keywords. – [Online; accessed 20-07-2023]

[8] *Spring-security-doc-securitycontextholder*. – URL https://docs.spring.io/spring-security/reference/servlet/authentication/architecture.html#servlet-authentication-securitycontextholder. – [Online; accessed 02-08-2023]

[9] *spring-security-docs-authenticationmanager*. – URL https://docs.spring.io/spring-security/reference/servlet/authentication/architecture.html#servlet-authentication-authenticationmanager. – [Online; accessed 02-08-2023]

[10] *spring-security-docs-authorizing-endpoints*. – URL https://docs.spring.io/spring-security/reference/servlet/authorization/authorize-http-requests.html#authorizing-endpoints. – [Online; accessed 02-08-2023]

[11] *spring-security-docs-dao-authentication-provider*. – URL https://docs.spring.io/spring-security/reference/servlet/authentication/passwords/dao-authentication-provider.html. – [Online; accessed 02-08-2023]

[12] *Vue-Docs*. – URL https://vuejs.org/guide/introduction.html. – [Online; accessed 15-08-2023]

[13] *Vue-Router-Docs*. – URL https://router.vuejs.org/guide/. – [Online; accessed 15-08-2023]

[14] *Vuex-Bezkoder*. – URL https://www.bezkoder.com/jwt-vue-vuex-authentication/. – [Online; accessed 15-08-2023]

[15] *Vuex-Docs*. – URL https://vuex.vuejs.org/guide/. – [Online; accessed 15-08-2023]

[16] Cohn, Mike: *User Stories Applied*. Addison-Wesley, 2004. – ISBN 0-321-20568-5

[17] Gordadze, Ioram: *spring-security-tutorial*. – URL https://www.toptal.com/spring/spring-security-tutorial. – [Online; accessed 02-08-2023]

[18] Simons, Michael: *Spring Boot 2 - Moderne Softwareentwicklung mit Spring 5*. dpunk.verlag GmbH. – ISBN 978-3-86490-525-4

[19] Sommerville, Ian: *Modernes Software-Engineering Entwurf und Entwicklung von Softwareprodukten*. Pearson. – ISBN 9783868943962

Listings

2.1 spring-boot-starter-web dependency aus der pom.xml 6
2.2 Vue-SFC Beispiel 7

6.1 Medium Entity Snippet 32
6.2 Inventory Controller Snippet 34
6.3 BorrowerRepository Snippet 36
6.4 LoanService 38
6.5 spring-boot-starter-security dependency 44
6.6 SecurityConfig 44
6.7 SecurityFilterConfig 45
6.8 Beispiel JWT (Dekodiert) 47
6.9 JwtService 48
6.10 LoanView-Snippet 53
6.11 auth.module-snippet 55
6.12 LoginView-Snippet 57
6.13 Vue-Router-Snippet beforeEach Funktion des Routers 58

A Anhang

A.1 Tabellen

MediaSeries:				
Attribute	Datentyp	Primarykey	Foreignkey	Not Null
id	bigInt	✓		✓
title	varchar			✓
mediaTyp	varchar			
isbn_ean	varchar			
originalPrice	double precision			
amount	bigint			
available	bigint			

Tabelle A.1: Tabelle MediaSeries

Medium:				
Attribute	Datentyp	Primarykey	Foreignkey	Not Null
medium_id	bigInt	✓		✓
seriel_nr	varchar			
status	varchar			
date_of_lend	date			
borrower_id	bigInt		✓	
media_series_id	bigInt		✓	✓

Tabelle A.2: Tabelle Medium

LoanHistory:				
Attribute	Datentyp	Primarykey	Foreignkey	Not Null
loan_history_id	bigInt	✓		✓
date_of_lend	date			
date_of_return	date			
mediun_id	bigInt		✓	✓
borrower_id	bigInt		✓	

Tabelle A.3: Tabelle LoanHistory

Borrower:				
Attribute	Datentyp	Primarykey	Foreignkey	Not Null
borrower_id	bigInt	✓		✓
borrower_nr	bigInt			
borrower_state	varchar			
date_of_birth	date			
email	varchar			
first_name	varchar			
last_name	varchar			
password	varchar			
roles	varchar			
class	varchar			

Tabelle A.4: Tabelle Borrower

@ElementCollection:				
Attribute	Datentyp	Primarykey	Foreignkey	Not Null
entity_name_id	bigInt		✓	✓
value	<T>			✓

Tabelle A.5: Tabelle für ein Feld annotiert mit @ElementCollection

A.2 Abbildungen

Abbildung A.1: Persona-Katarina-Lehrerin. Eigene Abbildung.

Abbildung A.2: Persona-Gunnar-Admin. Eigene Abbildung.

Glossar

AOP Aspektorientierte Programmierung.

BPMN Business Process Model and Notation.

DiViS Modernes Schulmanagement-System an beruflichen Schulen.

ERM Entity Relationship Mode.

IoC Inversion of Controle.

JDBC Java Database Connectivity.

JPA Jakarta Persistence API.

MDM Mobile-Device-Management.

MVC Model-View-Controller Architekturmuster.

REST Representational State Transfer.

SFC Single File Component.

URI Uniform Resource Identifier.

URL Uniform Resource Locator.

Erklärung zur selbstständigen Bearbeitung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Ort	Datum	Unterschrift im Original