Eötvös Loránd University

Faculty of Informatics

Dept. of Software Technology and Methodology

A modern library management system featuring ML-powered recommendations and desk reservation services

Supervisor:
Gregory Morse
Lecturer

Author:
Jamal Mammadov
Computer Science BSc

Budapest, 2025

EÖTVÖS LORÁND UNIVERSITY
FACULTY OF INFORMATICS

Thesis Topic Registration Form

Student's Data:
Student's Name: Mammadov Jamal
Student's Neptun code: ZJUAR3

Educational Information:
Training programme: Computer Science BSc

I have an internal supervisor

Internal Supervisor's Name: *Morse Gregory Reynolds*
Supervisor's Home Institution: ***Department of Programming Languages and Compilers***
Address of Supervisor's Home Institution: ***1117, Budapest, Pázmány Péter sétány 1/C.***
Supervisor's Position and Degree: *Teaching Assistant, MSc in Computer Science*

Thesis Title: A modern library management system featuring ML-powered recommendations and desk reservation services

Topic of the Thesis:
(Upon consulting with your supervisor, give a 150-300-word-long synopsis os your planned thesis.)

Traditional library management systems often rely on outdated software, leading to inefficiencies for both administrators and users. My final thesis aims to address these issues by developing a modern web-based library management system that incorporates desk reservation services, a machine learning-powered book recommendation engine, book rental functionality, and an admin panel for streamlined management. Users will be able to register, log in, and manage their memberships, with different membership levels (bronze, silver, and gold) that provide varying privileges for book rentals and desk reservations. For example, gold members can book a desk for longer periods than bronze members.
The system will include a reliable database to store user data, book inventory, and desk reservation details. This will allow the application to track user activities efficiently, such as borrowing books and reserving desks, while providing an informative and real-time dashboard for both administrators and users. The inventory will be populated using a public book dataset, preferably from Kaggle, ensuring scalability and ease of management for books.
The machine learning-powered recommendation engine will allow users to input a book name, and the system will generate multiple book suggestions based on factors such as genre, publication date, and author of the chosen book. The recommendation system will be trained using recommendation algorithms, such as content-based filtering, with Python libraries. To achieve accurate results, the model will learn from a wide variety of data from the library's inventory. My hardware is capable of handling machine learning processes and dataset management.
The complexity of this project lies in developing the machine learning model using public datasets, creating the desk reservation system, and managing different membership levels. By addressing these challenges, the proposed library management system will offer a user-friendly solution that optimizes both administrative efficiency and user engagement. The system's scalability allows for the future implemenatiton of additional features.

Budapest, 2024. 10. 15.

Contents

Acknowledgements

I would like to express my deepest gratitude to everyone who has supported me throughout the journey of this thesis. First and foremost, I am incredibly thankful to my family for their unwavering love, encouragement, and patience. Their belief in me kept me motivated during the toughest moments. To my friends, thank you for your constant support, understanding, and for always being there to help.

I am also grateful to the professors and staff at the university for their invaluable guidance and knowledge. A special thanks to my supervisor, Mr. Gregory Morse, whose expertise, constructive feedback, and continuous encouragement have been instrumental in the completion of this work.

Chapter 1

Introduction

Library management systems modernize conventional libraries by replacing manual processes and enhancing user experience through automation and digital interfaces. `Kitabkhana` (meaning "Library" in Azerbaijani) is an intuitive system designed for desk reservations, book borrowing, and personalized suggestions, targeting both users and administrators. Motivated by the need to improve library accessibility and efficiency, it addresses challenges such as limited book tracking, lack of desk reservation services, and non-personalized suggestions through a modular, user-centric architecture [1]. This thesis focuses on the design, development, and assessment of `Kitabkhana`'s core functions, aiming to answer: How can technology optimize book rentals and desk reservations (see Section 3.2.4 and 3.2.7)? Can recommendation systems improve user engagement through personalized suggestions (see Section 3.2.5)?

The system leverages a scalable architecture with a flexible database design to ensure efficient resource management. Recommendations are driven by content-based filtering utilizing cosine similarity, suggesting books based on user inputs like titles or ratings [2]. Role-based access control (RBAC) governs user privileges across Bronze, Silver, and Gold membership levels, ensuring secure and effective operation [3]. The thesis covers architectural planning, Application Programming Interface (API) interactions [4], and user workflows, along with their implementation and testing, to meet performance and usability goals. `Kitabkhana` aims to contribute to efficient, user-centric library solutions through integrated software engineering and machine learning approaches.

Chapter 2

User Documentation

2.1 System Requirements and Configuration

This section outlines the setup for the **Kitabkhana** application locally, covering system requirements, external service setup, and configuration.

2.1.1 System Requirements

Ensure your system meets these requirements to run Kitabkhana:

Hardware Requirements

- **Operating System (OS)**: Microsoft Windows 10+ [5], Apple macOS 10.15+ [6], or Linux (e.g., Ubuntu 20.04) [7].
- **Processor**: Modern multi-core (e.g., Intel i5) [8].
- **RAM**: 8 Gigabytes (GB) minimum (16 GB recommended). [9]
- **Storage**: 2 GB free for dependencies and MongoDB.
- **Internet**: Required for downloads and services.

Software Requirements

Install the following tools, following their official guides:

1. **Node.js (v18+)**: For backend and frontend, includes Node Package Manager (npm).

   https://nodejs.org/en/download [10, 11].

2. **Python (v3.9+)**: For recommendation script.

   https://www.python.org/downloads/ [12].

3. **MongoDB**: For data storage (use MongoDB Atlas or local).

   https://www.mongodb.com/docs/manual/installation/ [13].

4. **Git**: To clone the repository. https://git-scm.com/downloads [14].

5. **Code Editor**: Visual Studio (VS) Code for editing files.

   https://code.visualstudio.com/download [15].

6. **Terminal**: Command Prompt (Windows) [16], Terminal (macOS) [17], or Git Bash [18]. https://www.freecodecamp.org/news/command-line-for-beginners/

7. **Browser**: A modern web browser such as Google Chrome [19], Mozilla Firefox [20], Microsoft Edge [21], or any compatible alternative to access the application. https://zapier.com/blog/best-web-browser/

2.1.2 Account Setup for External Services

Kitabkhana uses MongoDB Atlas for database hosting.

1. Set up an account:
   https://www.mongodb.com/docs/atlas/getting-started/

2. Initalize a free-tier cluster:
   http://www.mongodb.com/docs/atlas/tutorial/deploy-free-tier-cluster/

3. Create a database user:
   https://www.mongodb.com/docs/atlas/security-add-mongodb-users/

4. Whitelist your IP:
   https://www.mongodb.com/docs/atlas/security/ip-access-list/

5. Obtain the connection string:
   https://www.mongodb.com/docs/manual/reference/connection-string/.

2.1.3 Getting the Repository

- **Option 1: Clone via Git**

  Clone the repository from GitLab [22]:

  ```
  git clone https://gitlab.com/jamual/thesis_elte.git
  cd thesis_elte
  ```

- **Option 2: Download ZIP from Neptun**

  If you received the repository as a ZIP file [23] from Neptun, first extract, then navigate to the extracted folder.

2.1.4 Configure the Application

Environment Variables

Copy the `.env` template [24]: `cp server/.env.example server/.env`. Edit `server/.env` with:

MONGO_URI From Section 2.1.2 or `mongodb://localhost:27017/kitabkhana`.

JWTPRIVATEKEY Random secret key for JSON Web Tokens (JWT) authentication [25].

SALT Password hashing salt (default: `10`) [26].

PORT Backend port (default: `4000`) [27].

Install Dependencies

Run `npm run install-all` to install Node.js and Python dependencies, and initialize and activate Python Virtual Environment (venv) [28].

Initialize the Database and Admin User

Run `node server/importBooks.js` to populate the `Book` collection. Back up data, as this deletes existing books.

Additionally, run `node server/admin_seed.js` to create an admin user (refer to Section 2.3.1 for admin credentials).

2.1.5 Run the Application

Run `npm start` in the root folder to launch the backend (`http://localhost:4000`), Flask microservice (`http://localhost:5000`) [29, 30], and frontend (`http://localhost:3000`).

2.2 User-Side Usage Guide

This section provides a comprehensive user guide to assist users in navigating the application and utilizing its full functionalities. Basic functionalities such as login, registration, book rental, return, getting recommendations, reserving a desk, viewing reservations and book rentals, changing passwords, and upgrading membership levels will be covered.

2.2.1 Authentication Page

After successfully launching the application, the user must either register or log in to continue using it [31]. First-time users will see the login page (see Figure 2.1).

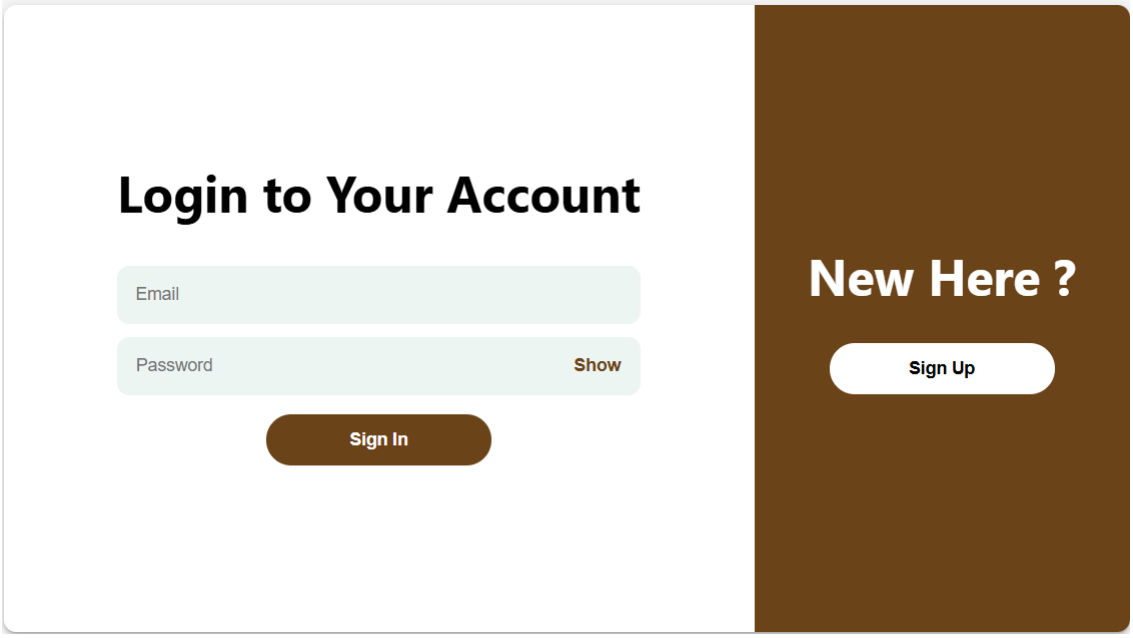


Figure 2.1: Login Page

On this page, to authenticate successfully, the user must enter a registered email address and password. The password input field includes a toggle button that allows users to show or hide the entered password.

For new users, clicking the *"Sign Up"* button navigates to the registration page (see Figure 2.2). All fields on this page are mandatory for successful sign-up:

- **First Name**
- **Last Name**
- **Email:** Users cannot register with an email address that is already in use. A formatter checks the input for a valid email format [32].
- **Password:** Must contain at least 8 characters, including one uppercase letter, one numeric digit, and one symbol [33].

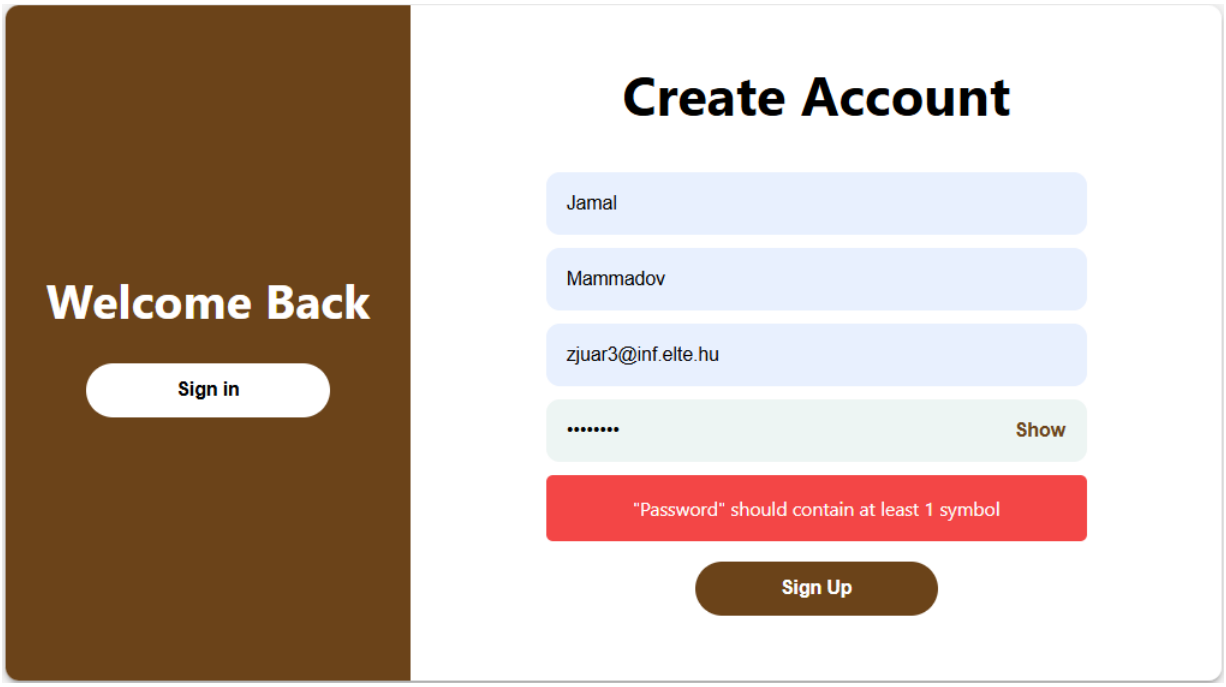


Figure 2.2: Registration Page

After successful registration, the user is redirected to the login page within one second and receives a toast notification confirming the registration (see Figure 2.3) [34].

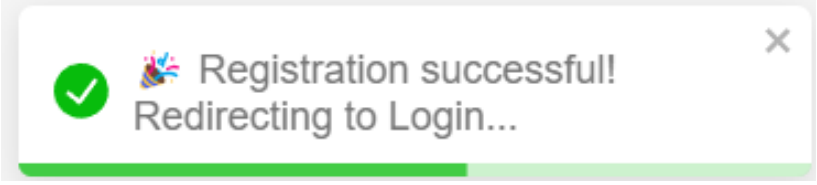


Figure 2.3: Toast Notification

2.2.2 Main Page

On the main page, users are welcomed by popular books, genre-based categories, and a navigation bar (see Figure 2.4) [35]. From here, they can scroll through different categories, explore books, navigate to other pages, or simply log out.

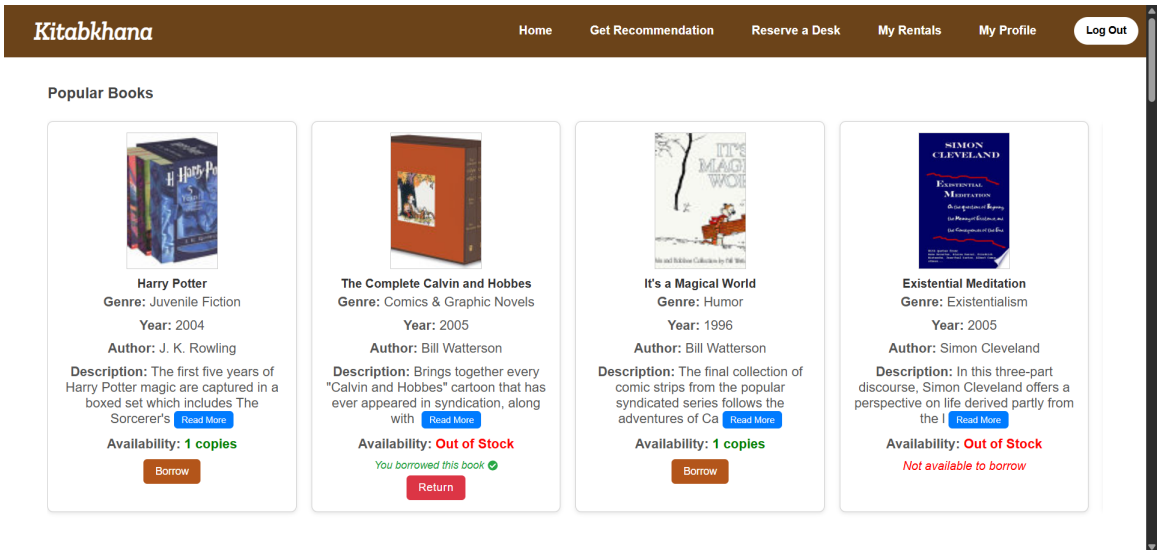


Figure 2.4: Main Page

Users can scroll horizontally to explore books within the same genre. The section automatically scrolls by one book every 2 seconds. A "Load More" button is available to load more categories (see Figure 2.5) [36]. Each time the button is clicked, three more genres are loaded.

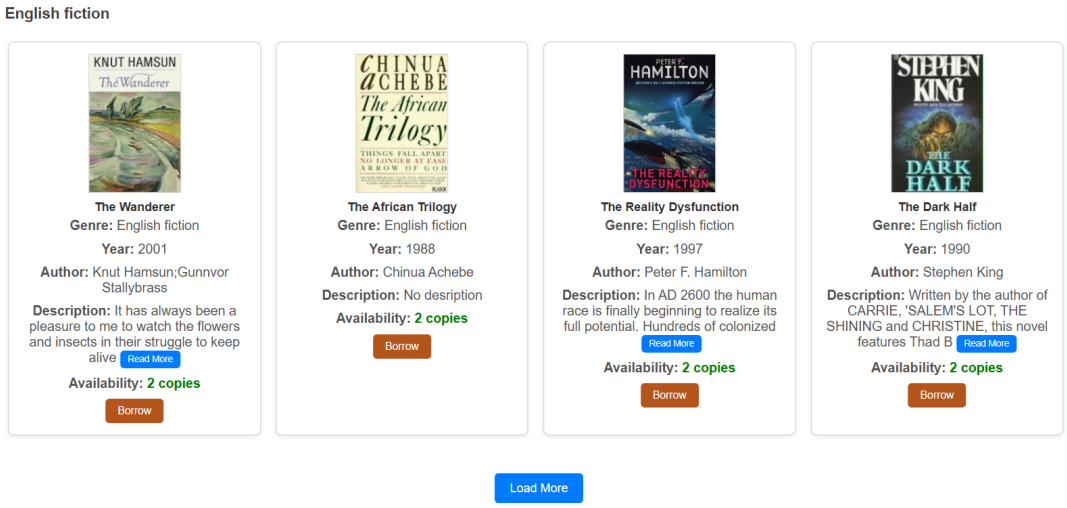


Figure 2.5: Load More button for Pagination

Each book is displayed as a book card, containing the following information:

- **Title**
- **Genre**
- **Year**
- **Author**
- **Description:** Some books feature a *"Read More"* button, which opens a modal window providing extended details, including the full book description [37]. The modal can be closed easily by clicking outside the window or on the close icon.
- **Availability:** Number of copies available for rental.
- **Borrow/Return Button:** Users can borrow or return books depending on their membership limits (see Figure 2.6). A notification is displayed for both successful and unsuccessful operations. If availability is 0, this button will be disabled (see Figure 2.7).

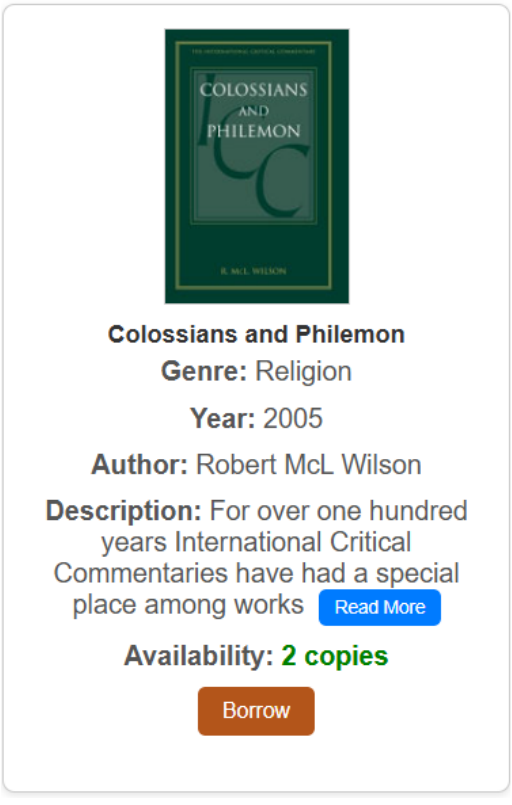


Figure 2.6: Available Book

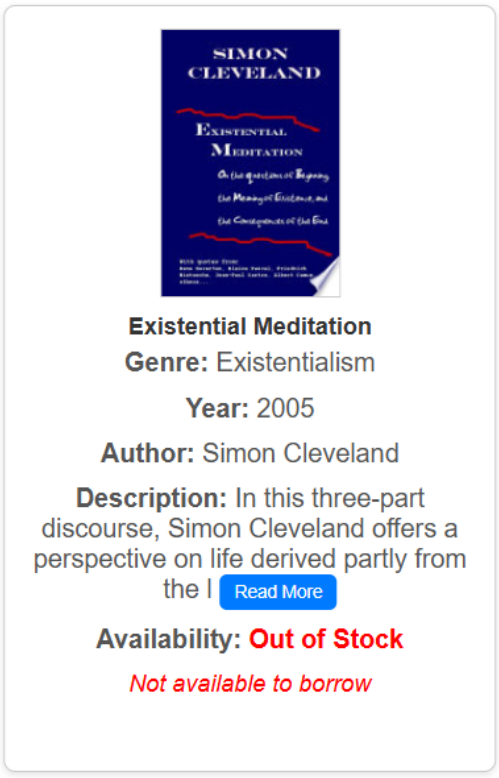


Figure 2.7: Out of Stock Book

2.2.3 Get Recommendation Page

This page offers two options for users to receive recommendations based on a machine learning algorithm (see Figure **??**)[38].

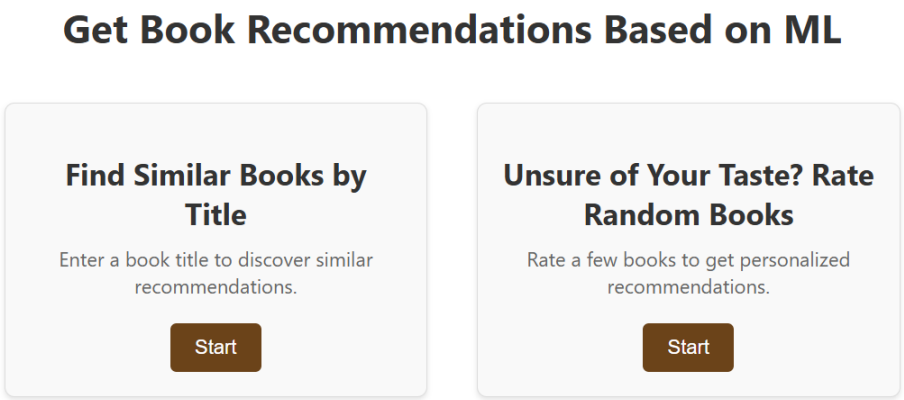


Figure 2.8: Navigation Bar

Find Similar Books by Title

The first option allows users to search for book titles and find similar ones (see Figure 2.9). At least one book must be entered.

Figure 2.9: Search box

To remove a book, hover over the book card and then click the red 'X' button in the upper-right corner of the card (see Figure 2.10). After selecting several books, the user should click the 'Get Recommendations' button to get recommendations. It will take a little to load the results. User will see loading page. The 'Clear All' button will remove all selected books.

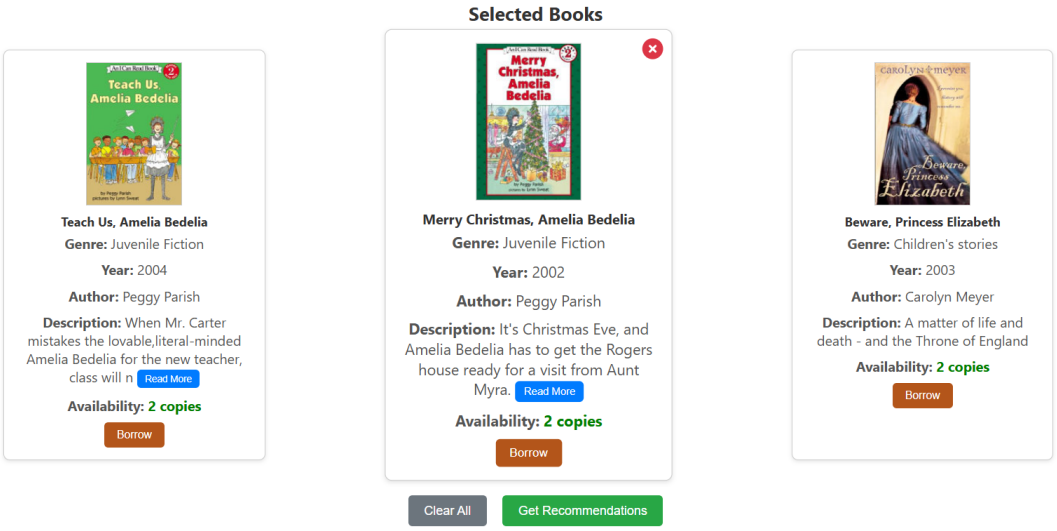


Figure 2.10: Selected books

The system will then recommend three similar books based on properties such as title, genre, author, and publication year of the selected books. Users can either borrow these books or start over by refreshing the page (see Figure 2.11).

Figure 2.11: Recommended books

Rate Books to Discover New Ones

The second option is designed for users who are unsure about their preferences or simply want to explore new books. On this page, they will see six random books, each of which must be rated using a 1 to 5-star scale [39].

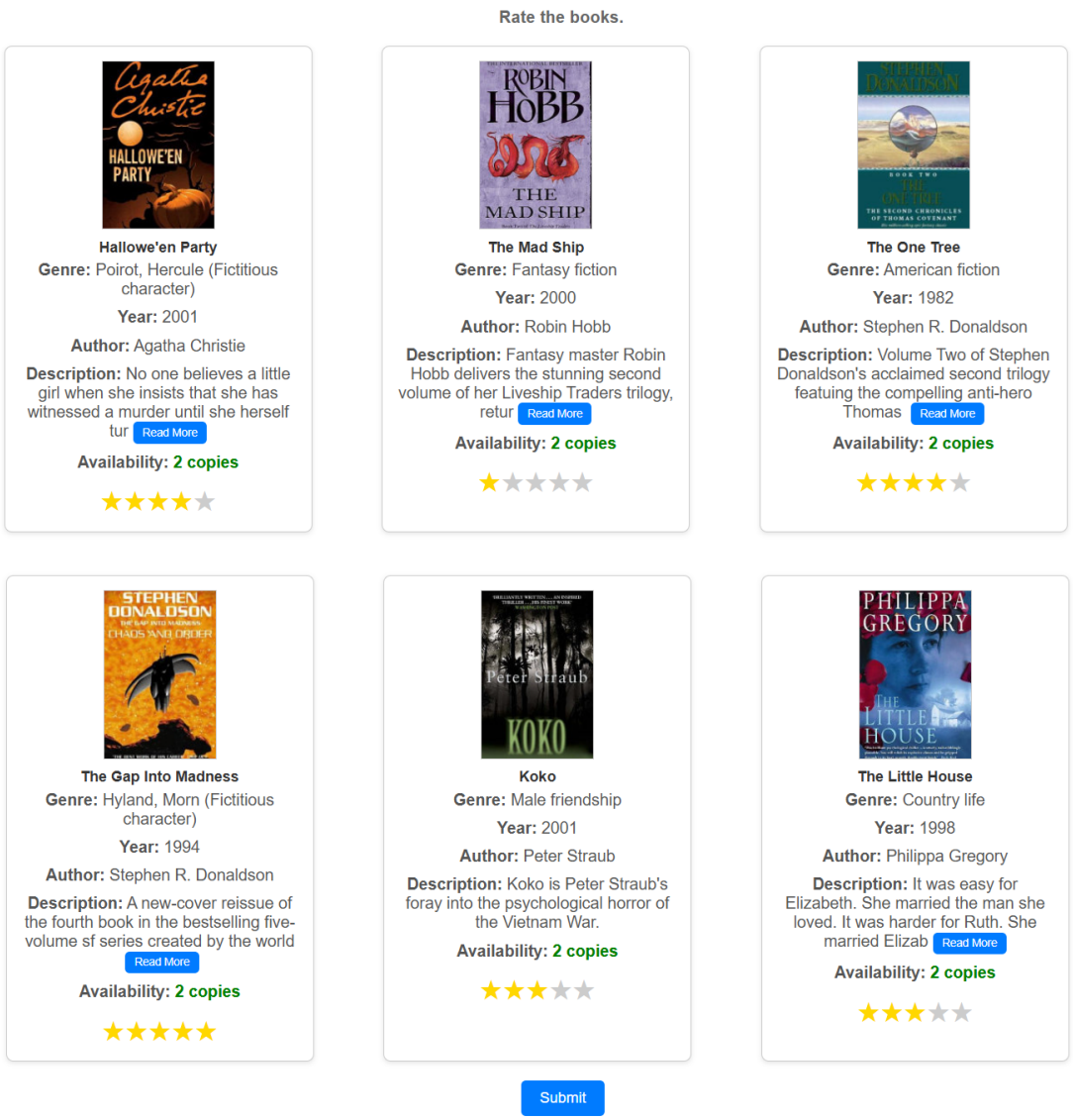


Figure 2.12: Rating Books

It is mandatory to rate all six books in order to receive recommendations. If the user clicks the submit button without rating all of them, the form will not be submitted, and a toast notification will appear.

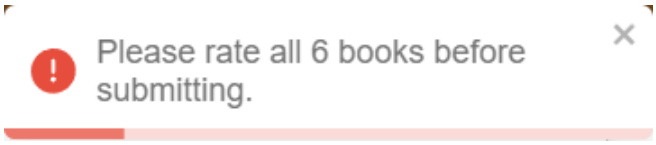


Figure 2.13: Toast Notification for Missing Ratings

After submitting their ratings, a loading page will appear for few seconds in order to fetch the results.

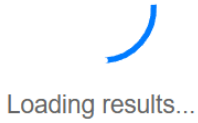


Figure 2.14: Loading page

At the end, users will receive three book recommendations based on their ratings. They can either borrow the recommended books or start over by refreshing the page, similar to the first recommendation option.

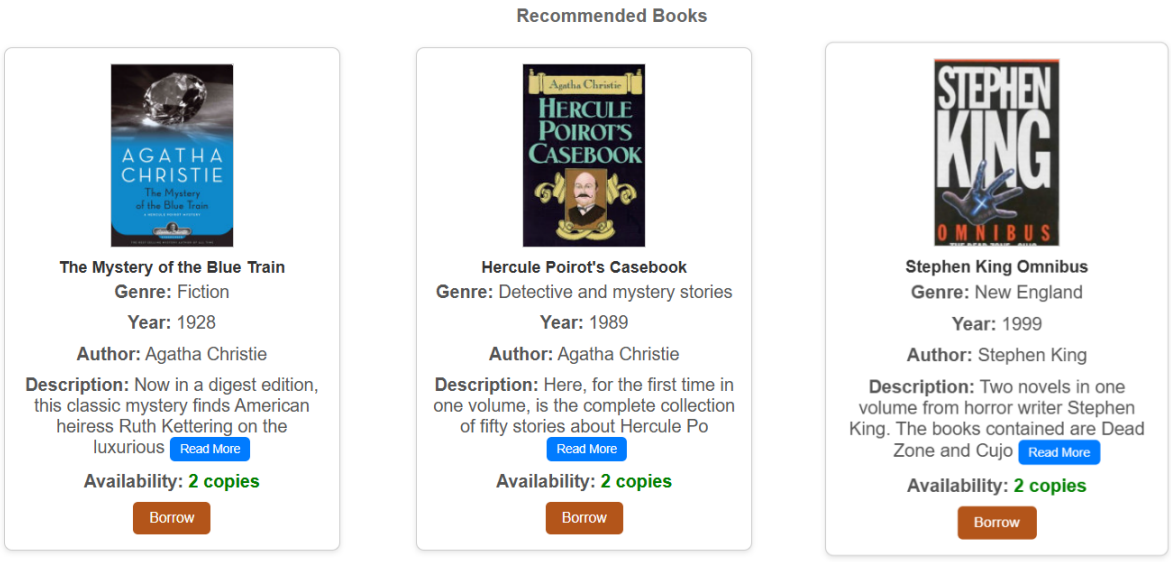


Figure 2.15: Recommended Books

2.2.4 Reserve a Desk Page

This page allows users to reserve a desk at one of the following libraries: Newton, Tesla, or Goethe. The page includes several input fields required for making a reservation (see Figure 2.16):

- **Select Library**: One of the library options—Newton, Tesla, or Goethe—must be selected.
- **Select Date**: By default, users can choose a date starting from tomorrow.
- **Select Starting Time**: Users can choose a time between 08:00 and 22:00.
- **Select Duration**: Depending on the user's membership level, the duration can range from 1 to 6, 9, or 12 hours.

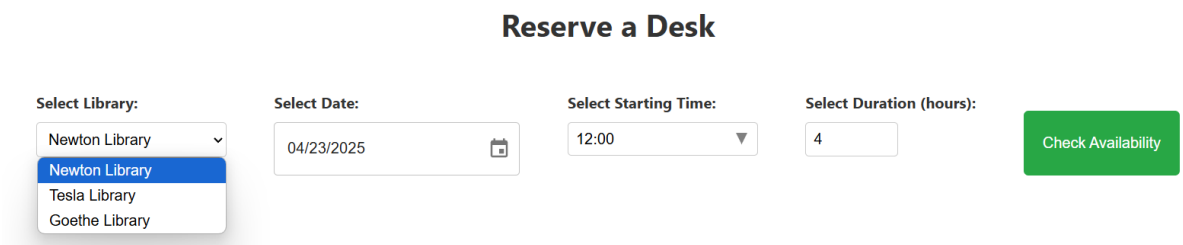


Figure 2.16: Reserve a Desk page

After filling in the input fields, the user should click the 'Check Availability' button to view available desks. If any of the input fields are changed, the user must click the button again to refresh the results accordingly.

At the next step, the user can choose one of the available seats (see Figure 2.17). In the seat grid, the meaning of each color is explained at the bottom. "Busy" indicates that the seat has already been booked by another user for the selected time. In this case, the user can change one of the input fields to search for available seats.

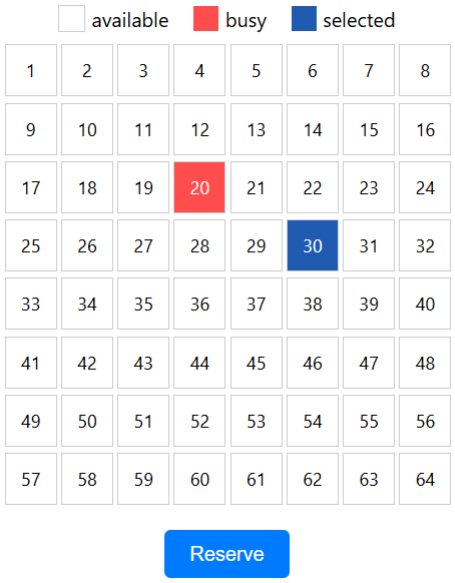


Figure 2.17: Seats grid of a library

The system checks for overlapping reservations, meaning a user cannot book another seat at any library if they already have a reservation that overlaps with the selected time. In case of submission user will get toast notification about overlapping reservation (see Figure 2.20).

Figure 2.18: Toast Notification for overlapping reservation

Finally, when user make a successful reservation, will also get a toast notification (see Figure 2.21). To change or cancel a reservation, the user must first cancel their existing reservation from the my profile page (See Section 2.2.6) and then make a new one.

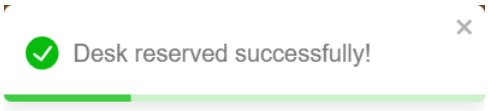


Figure 2.19: Toast Notification for successful reservation

2.2.5 My Rentals Page

On this page, users can view their currently borrowed books displayed in a book-card format (see Figure 2.20). When a user reaches their membership rental limit, they must return a book in order to borrow a new one. Alternatively, they can apply for a membership upgrade to increase their borrowing limits (see Section 2.2.6).

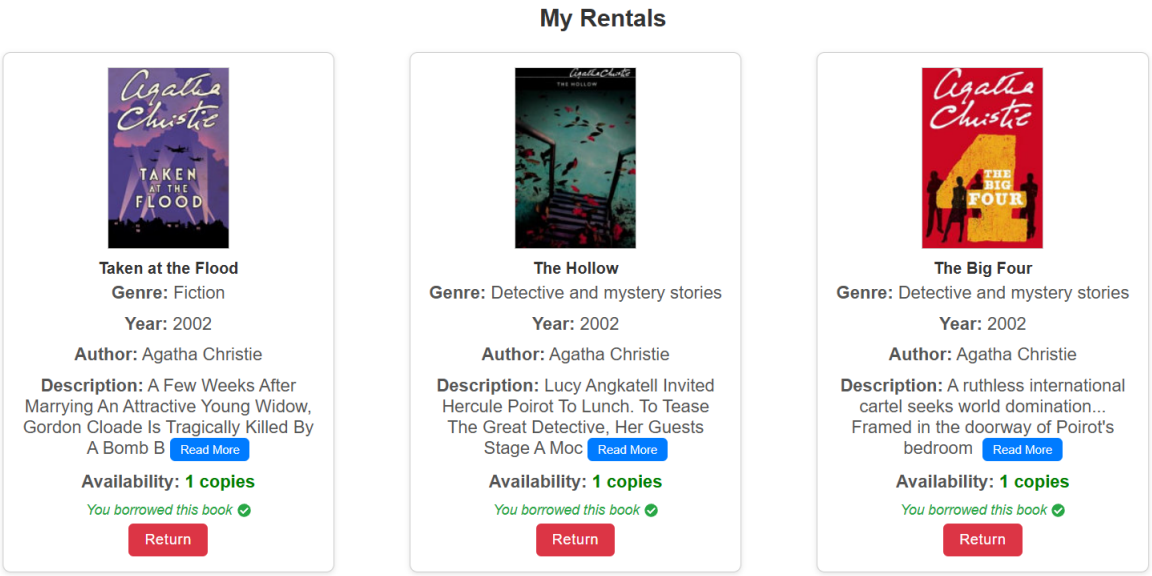


Figure 2.20: My Rentals Page

Users can also click the "Generate Recommendations" button at the bottom of the page (see Figure 2.21) to receive three book recommendations similar to their current rentals.

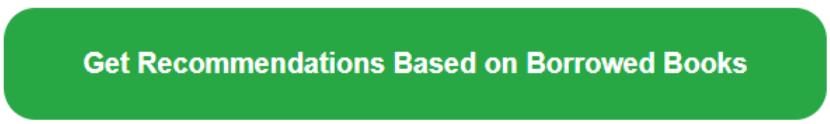


Figure 2.21: Generate Recommendations Button

2.2.6 My Profile Page

The final page on the user side, the My Profile page, serves as a central hub for managing various operations and viewing a range of data in a visual format. On the left side of the page a profile menu provides access to different sections.

Profile Sub-page

When users navigate to the My Profile page, the first section they see is the profile section (see Figure 2.22). This section includes the following information:

- **Profile Picture**: Users can see or change their profile picture. The uploaded image must be in JPEG, PNG, or GIF format and should not exceed 2MB in size.
- **Personal Data**: This area displays the user's full name, email address, date of registration, and membership level.

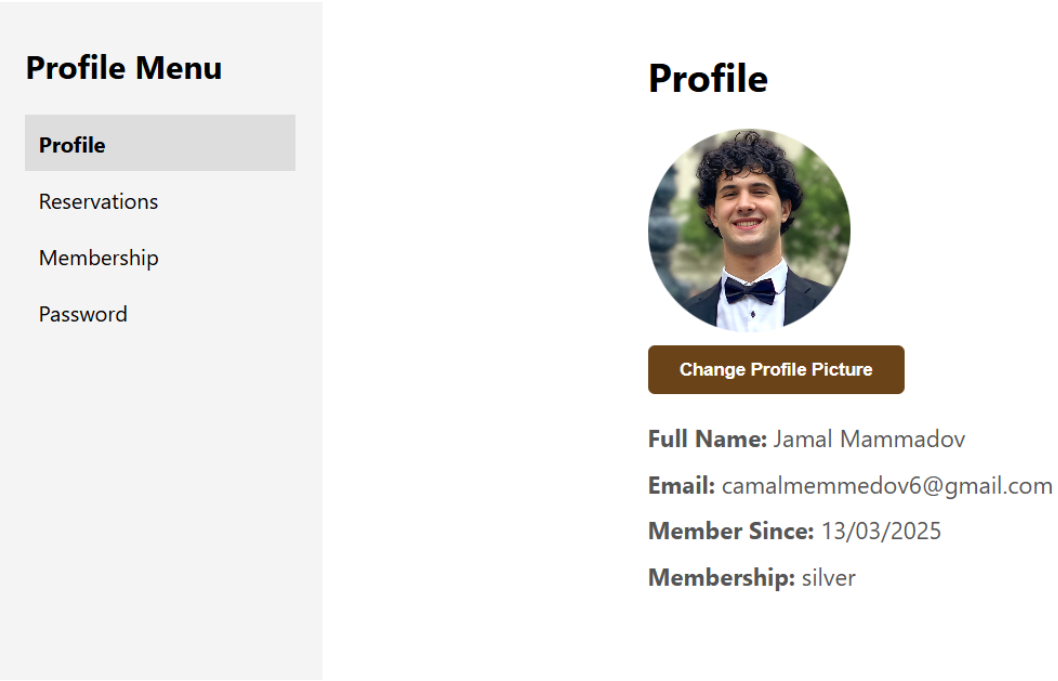


Figure 2.22: Profile Section

- **Real-Time Dashboard**: Users can view a pie chart and bar chart showing the genre distribution of their borrowed books and the number of active reservations per library, respectively (see Figure 2.23) [40, 41]. This information updates in real-time when a user borrows a new book or makes a new reservation.

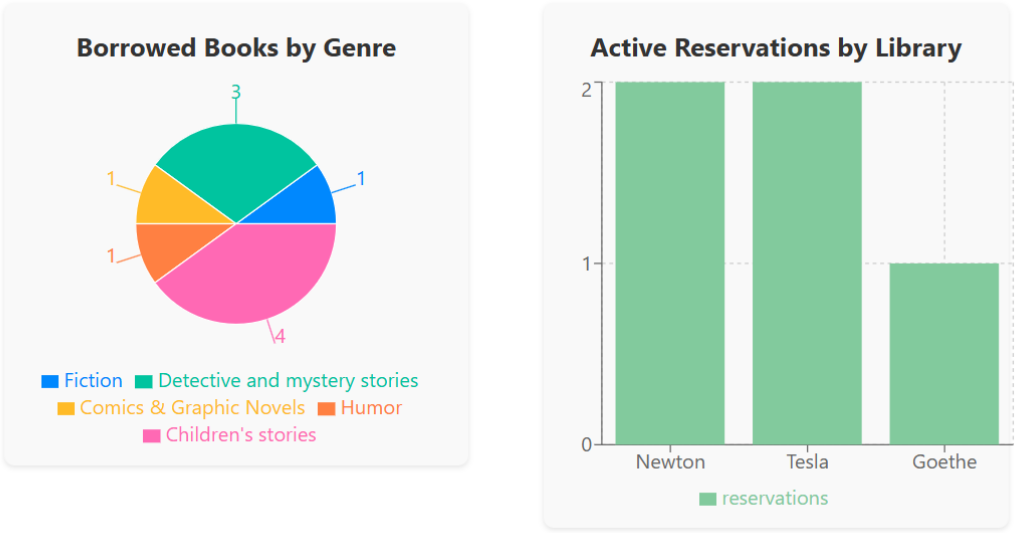


Figure 2.23: Real-Time Dashboard for User

By default, when a new user registers, there will be no data available, and the dashboard will indicate that there is no information to display (see Figure 2.24). Once the user borrows books or makes a reservation, the dashboard will begin to show relevant data. It updates in real-time as the user borrows or returns books, or cancels reservations.

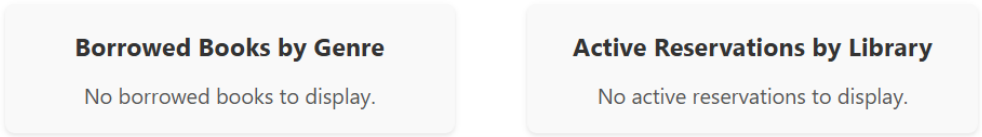


Figure 2.24: Real-Time Dashboard Displayed When No Data is Available

Reservations Sub-page

This section is designed for managing user reservations in single place (see Figure 2.25). In this page there are subsections with different categories:

- **Active Reservations**: Reservations that are active, and the date has not passed will be displayed here.
- **Cancelled Reservations**: If the reservation has been cancelled by user, or admin it will appear here.
- **Past Reservations**: When active reservations is used, and their date is already in the past (e.g yesterday and later), they can be found here.

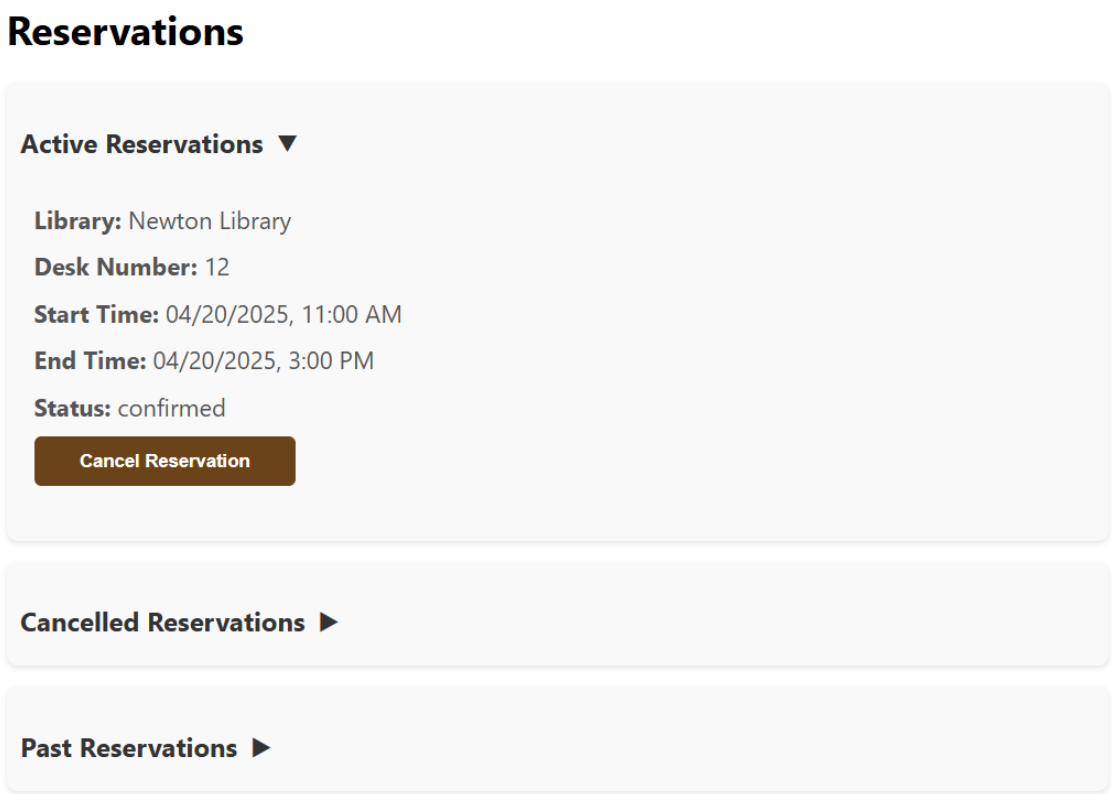


Figure 2.25: Reservations by Categories

Membership Section

In this section, users can find general information about the different membership levels (see Figure 2.26). A newly registered user is assigned the Bronze level by default. The benefits and limitations of each membership tier are detailed on this page.

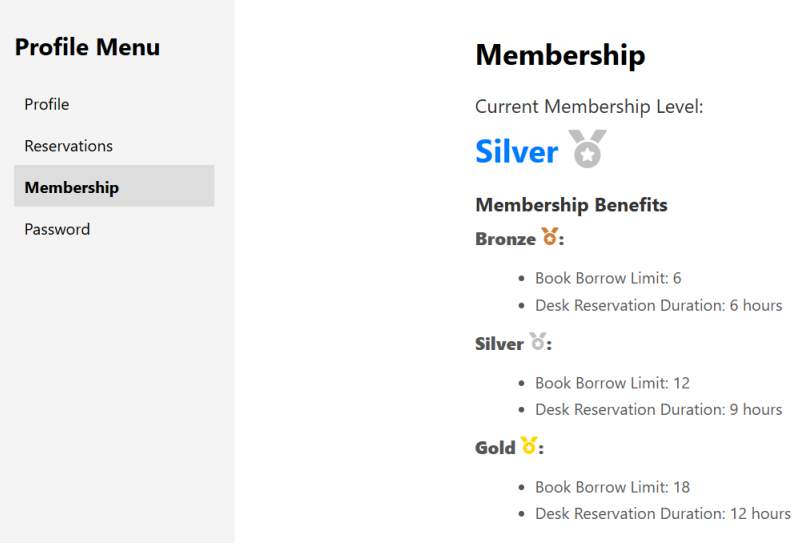


Figure 2.26: Membership Section and Benefits of Each Level

Users can also view different status of membership requests and submit a request for an upgrade (see Figure 2.27). All upgrade requests must be approved by an administrator. If the user already has a pending request or has reached the Gold level, they will no longer be able to submit a new upgrade request.

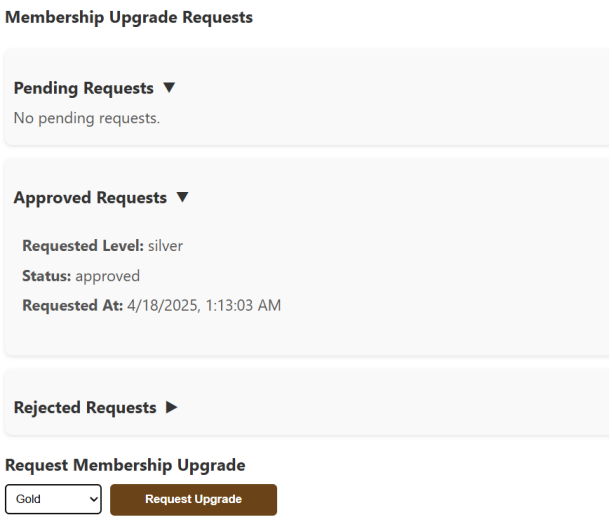


Figure 2.27: Membership Upgrade Requests

Password Section

The final section of the user profile allows users to change their password. To do so, they must enter their current password, followed by the new password entered twice for confirmation (see Figure 2.28). The password requirements remain consistent, and real-time error messages are displayed while filling out the inputs.

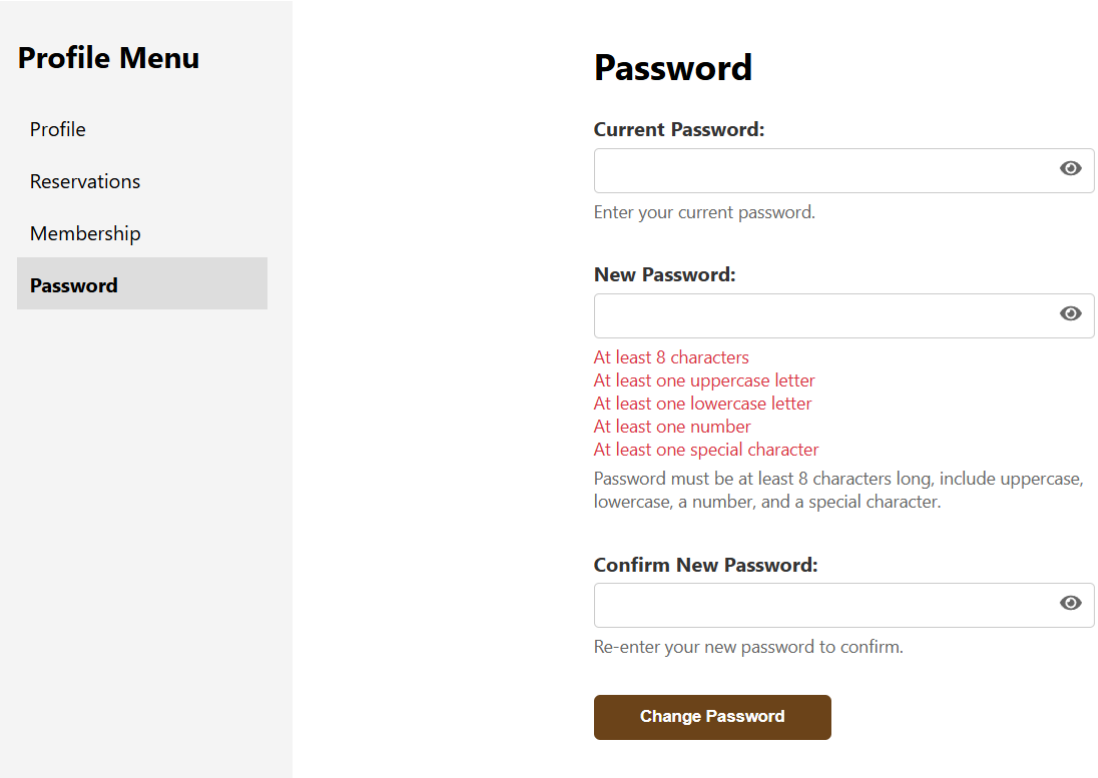


Figure 2.28: Password Change Section

In order to proceed, the current password must also be entered correctly. The system will verify it, and if the password is incorrect, an error message will be displayed (see Figure 2.29).

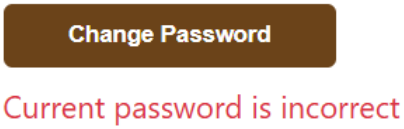


Figure 2.29: Incorrect Current Password Entry

2.3 Admin-Side Usage Guide

This section provides a detailed guide for administrators on how to effectively use `Kitabkhana` admin panel. The admin panel offers essential tools for managing the system, including access to a dashboard with visual performance metrics, handling membership upgrade requests, and user reservations.

2.3.1 Admin Login

To log in as an administrator, navigate to the following route: `http://localhost:3000/admin-login` (see Figure 2.30). In order to access the admin panel, users must enter valid login credentials and possess administrative privileges. If the credentials are incorrect or the user does not have admin rights, access to the panel will be denied, and an error message will be displayed.

Only authorized personnel should have access to this route, as it provides administrative control over various system functionalities. It is recommended to keep login credentials secure to prevent unauthorized access.

For demo an admin user has been initialized with following credentials:

- **Email:** `admin@gmail.com`
- **Password:** `Admin123*`

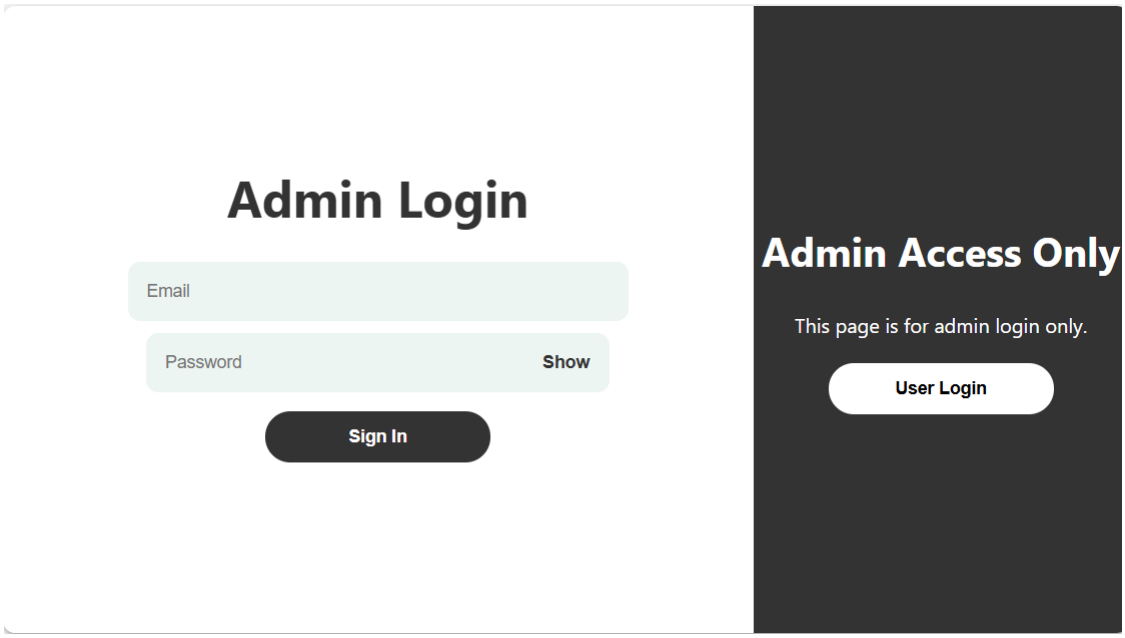


Figure 2.30: Admin Panel Login Page

2.3.2 Admin Dashboard

The admin dashboard serves as the main landing page for administrators and is updated in real time to reflect key system metrics (see Figure 2.31). It provides a visual overview of important data, enabling admins to monitor user activity and system usage effectively.

The dashboard includes the following visual components:

- **Total Users**: Displays the number of registered users in the system.
- **Total Borrowed Books**: Shows the total number of books borrowed by all users.
- **Membership Distribution**: A pie chart illustrating the proportion of users in each membership tier.
- **Reservations per Library**: A breakdown of reservations across different libraries, including all status categories.
- **Borrowed Books by Genre**: A genre-wise distribution of all borrowed books presented in a pie chart (see Figure 2.32).

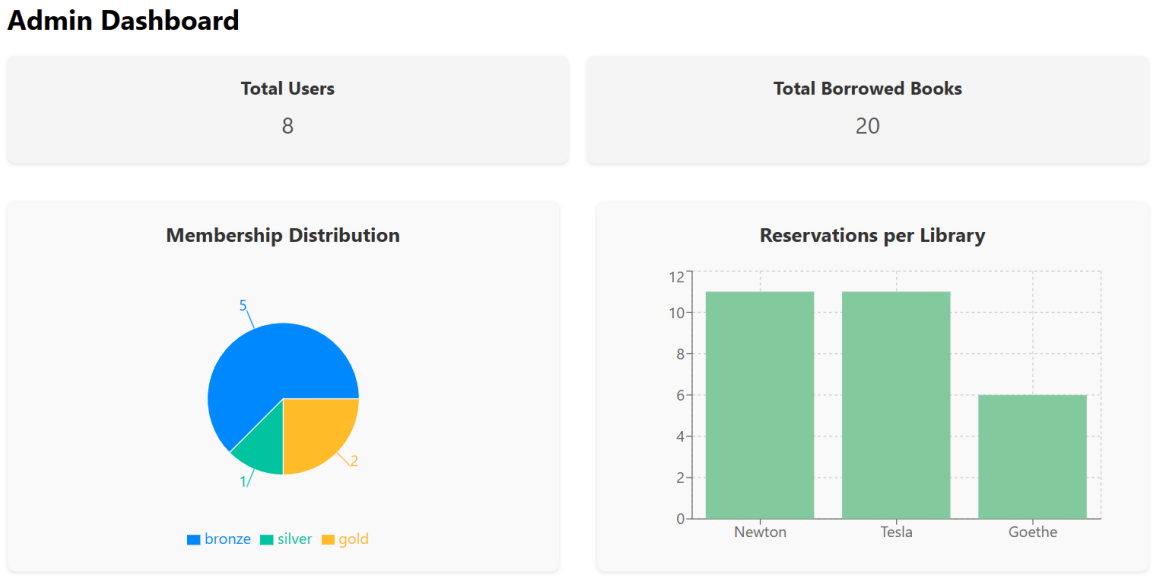


Figure 2.31: Admin Panel Dashboard

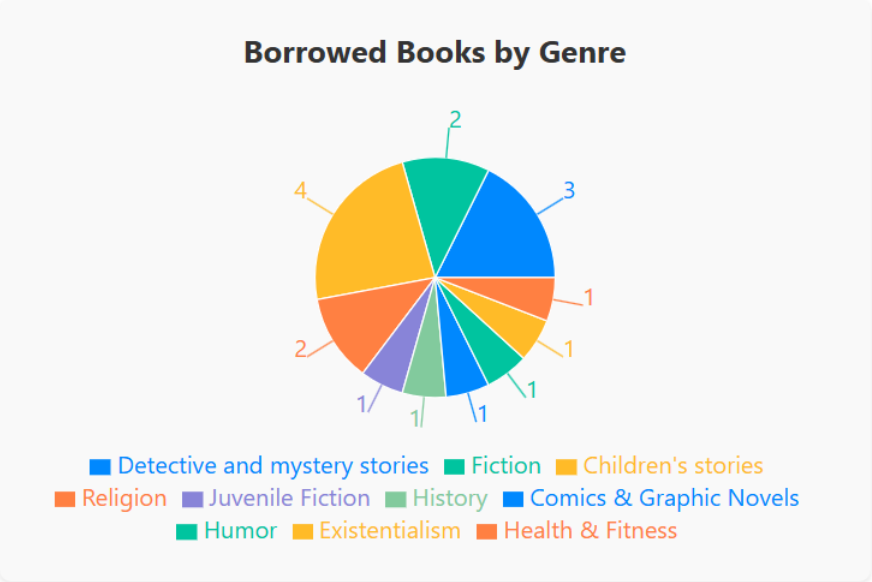


Figure 2.32: Pie Chart of Borrowed Books by Genre

2.3.3 Requests Page

In this section, administrators can view and manage membership upgrade requests submitted by users (see Figure 2.33). Each request is displayed in a tabular format, with the following information available in each row:

- **User**: The full name of the user.
- **Email**: The email address of the user.
- **Current Level**: The user's current membership level.
- **Requested Level**: The membership level requested by the user.
- **Actions**: Admins can either approve or reject the request using the corresponding action buttons.

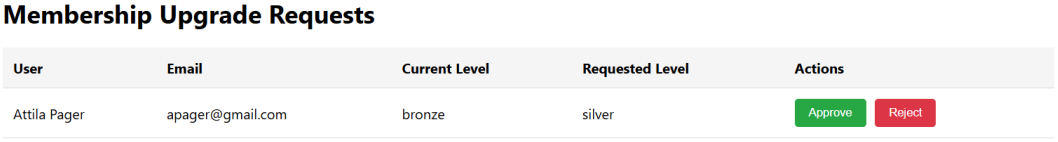


Figure 2.33: Membership Upgrade Requests Table

2.3.4 Reservations Page

On the Reservations page, administrators can view and manage all user reservations within the system (see Figure2.34). Reservations are grouped into three categories: **Active**, **Cancelled**, and **Past Reservations**, allowing for easier tracking and filtering of booking statuses.

Each reservation entry provides detailed information such as the user's name, email, selected library, desk number, reservation start time, end time, and current status. This layout helps administrators monitor usage patterns and handle reservations more efficiently.

Administrators also have the authority to cancel any active reservation when necessary—whether due to system errors, policy violations, or user requests. Upon cancellation, the reservation's status is updated accordingly and moved to the Cancelled Reservations category.

This page plays a crucial role in maintaining operation of the library booking system by giving admins control over reservations and helping them resolve issues.

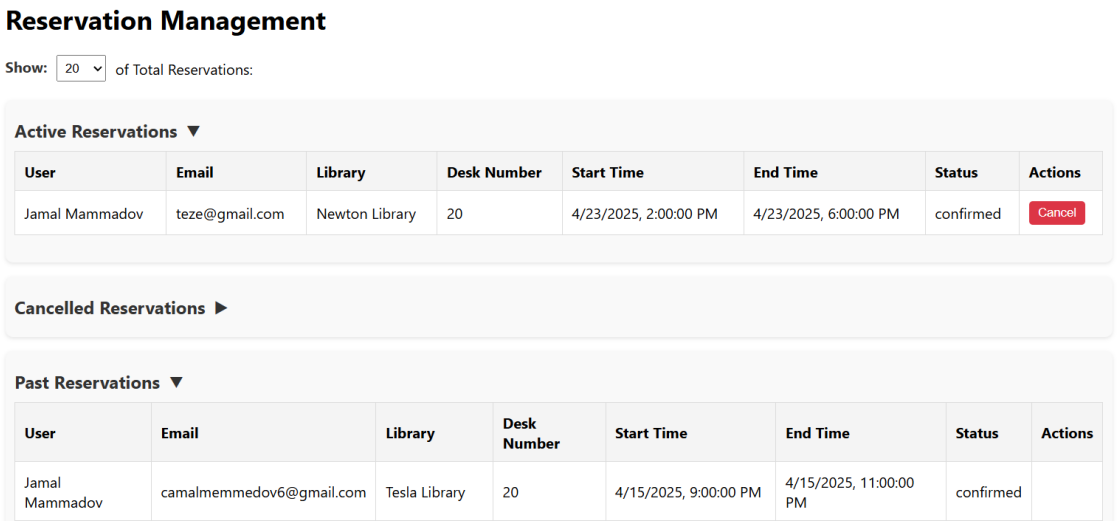


Figure 2.34: Reservation Management

2.3.5 Switching to User View

In this application, administrators are also considered regular users, meaning they have access to user-side features in addition to their administrative privileges. To access the user interface, admins can simply click the **Switch to User View** button from navbar (see Figure 2.35), which redirects them to the main page of the user-side application(see Figure 2.36).

This feature is particularly useful for testing or managing their own user-related activities, such as book rentals or reservations, without needing a separate user account.

To return to the admin interface, administrators must manually navigate to the following URL in their browser: `http://localhost:3000/admin`. This route is protected by admin-specific tokens, ensuring that only users with administrative privileges can access it. Non-admin users attempting to visit this route will be denied access, and will forward to main page of user-side.

This separation and protection of routes ensure both security and flexibility for admin users operating across both interfaces.

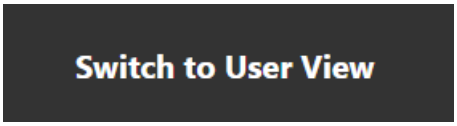


Figure 2.35: Switch to User View Button

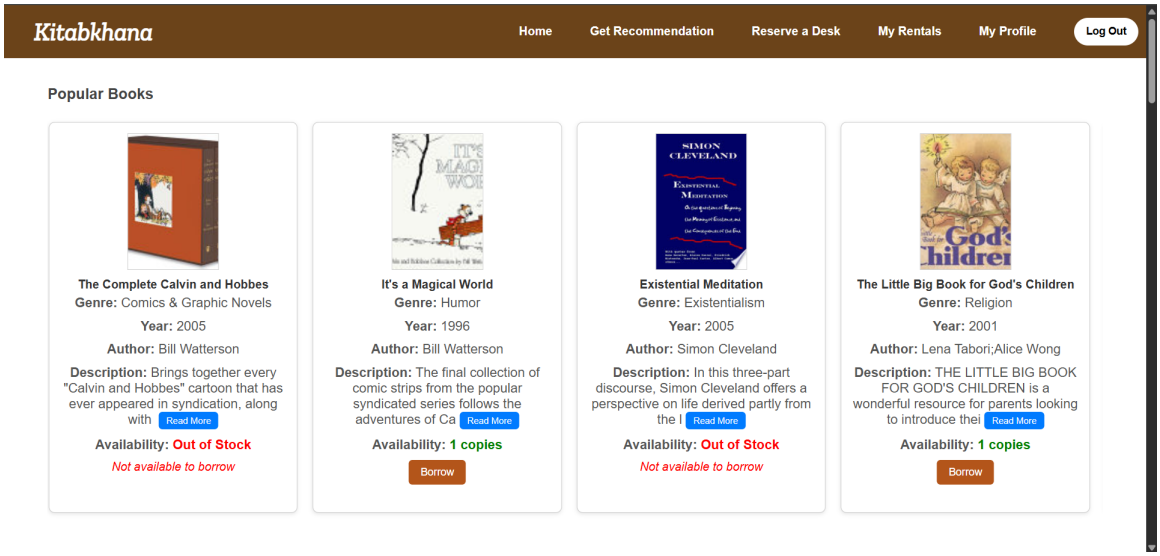


Figure 2.36: User view

Chapter 3

Developer Documentation

3.1 Design

Library management system design for `Kitabkhana` provides a modular and user-centric framework for library operations. It is a conceptual blueprint of the system, that describes its structural parts, interactions, and visual representations. Clarity and modularity are prioritized to achieve system acceptance by users and administrators, as well as provide a solid foundation for development and future enhancements.

The design is explored in five aspects. Wireframes [42] represent layouts for main components like the home page, desk reservation, and book recommendation pages. System architecture and component interaction will give an overview of the basic structure of the client, server, and data storage components. Also, the database design specifies how core entities such as books, users, rentals, and reservations schemas are structured to optimize data management and integrity preservation.

The Application Programming Interface (API) design also defines interfaces between client components and back-end services, including user profile and recommendation services with modular integration. Unified Modeling Language (UML) diagrams show system behaviour for book borrowing, recommendation generation, and membership requests in use case and workflow diagrams. All of which together provide an overview of `Kitabkhana`'s design using conceptual frameworks and visual models as a guide to implementation.

3.1.1 Wireframe

Wireframes illustrate the layout and user interactions in `Kitabkhana`, including the homepage, ML-powered recommendation system, and desk booking [42].

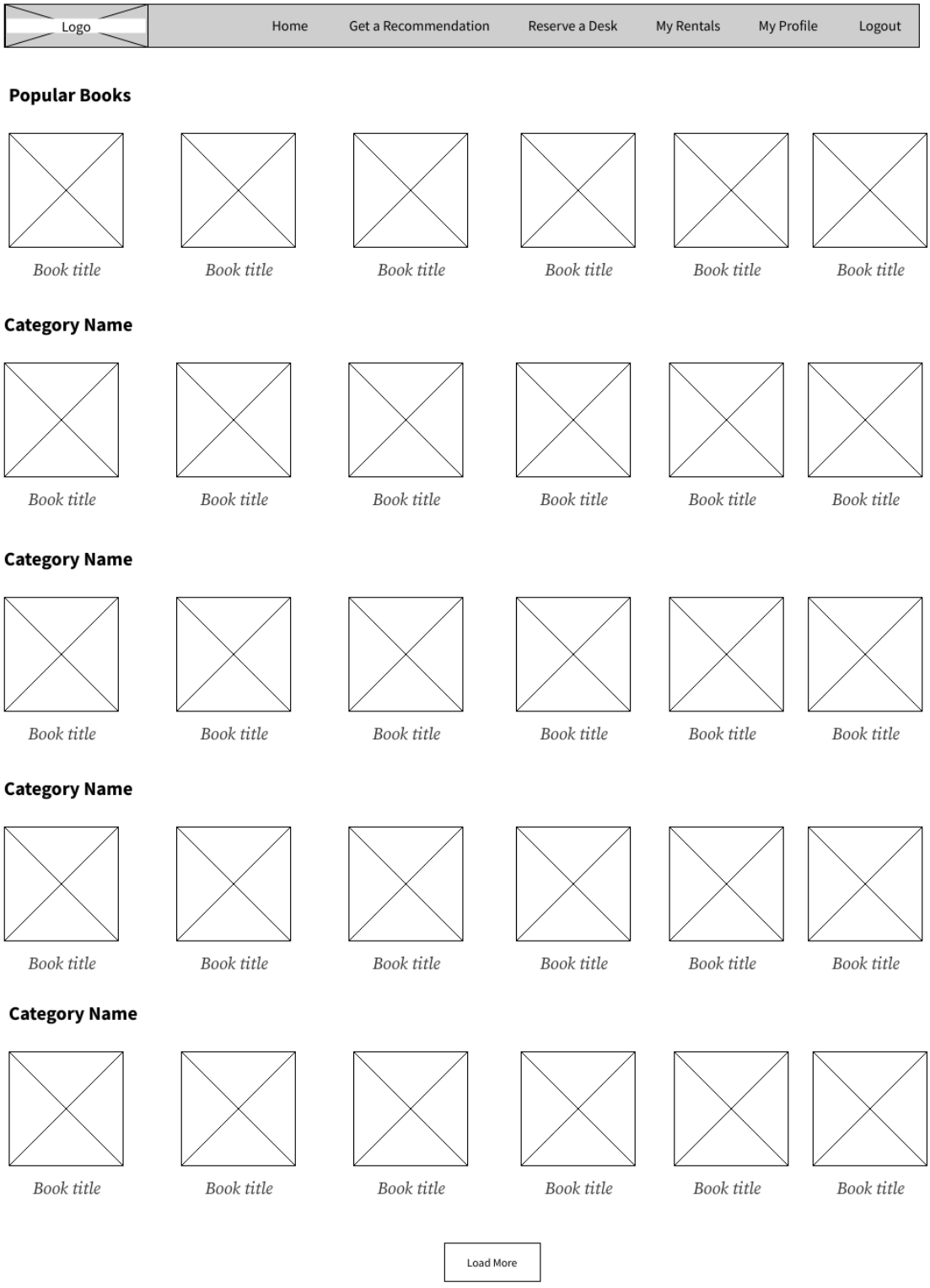


Figure 3.1: Wireframe for Home Page

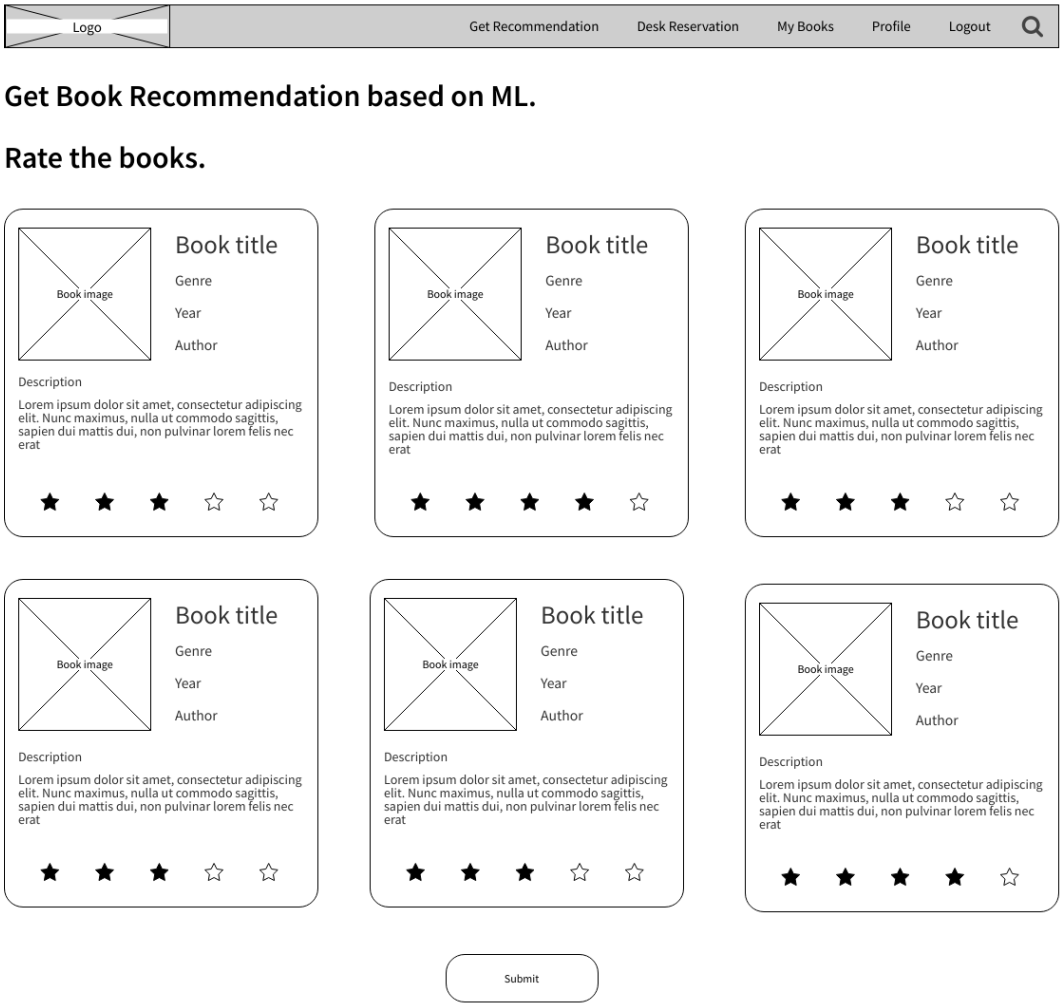


Figure 3.2: Wireframe for ML Recommendation Page

All design decisions focus on usability, improving user experience, and enabling users to browse through the application conveniently. They provide a visual roadmap to the structure and navigation flow.

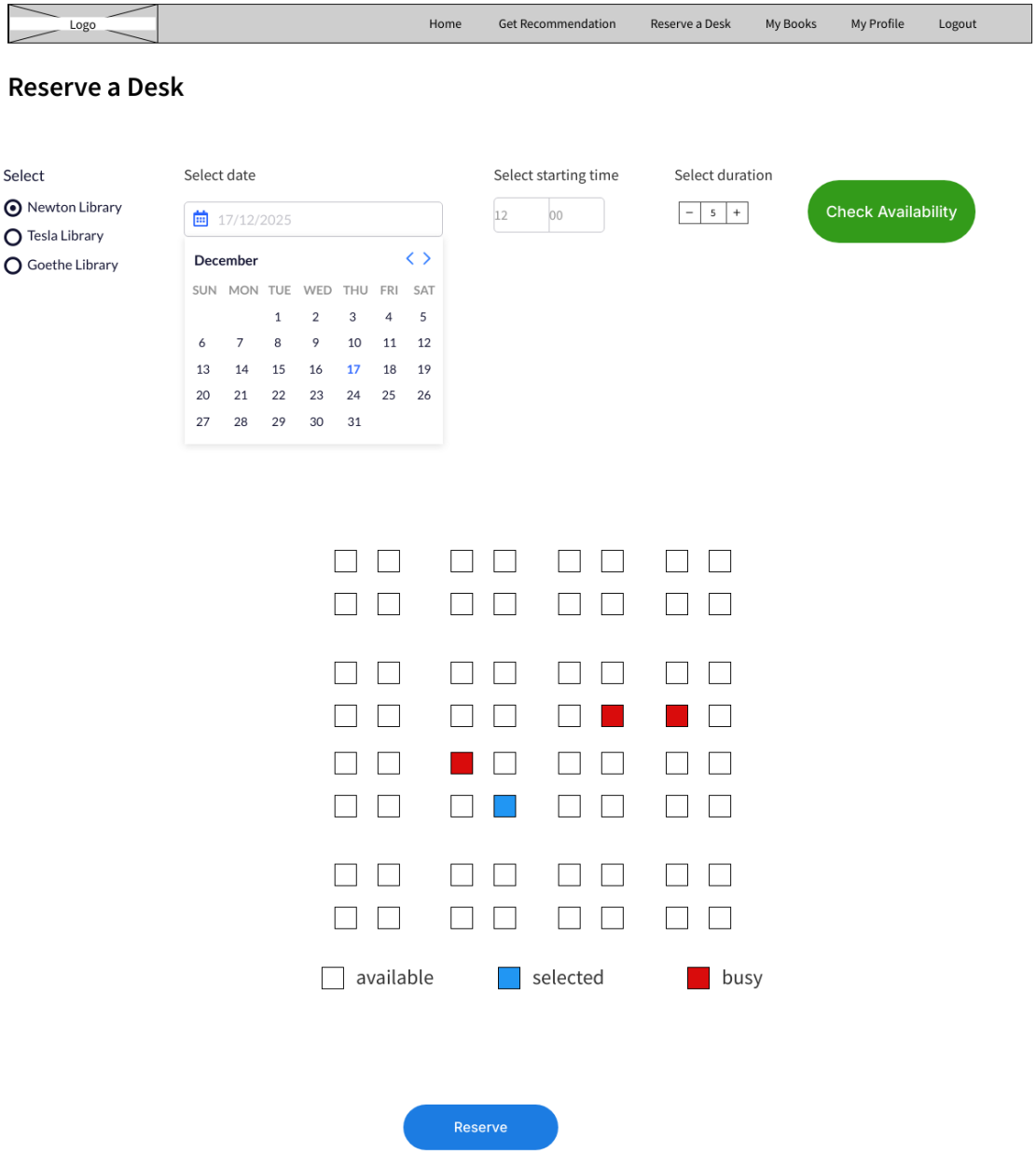


Figure 3.3: Wireframe for Desk Reservation Page

3.1.2 System Architecture and Component Interaction

Kitabkhana is based on a modular design to ensure scalability, easy maintainability, and easy integration with different APIs [1]. The application consists of integrated but independent parts, each servicing a specific range of functionalities.

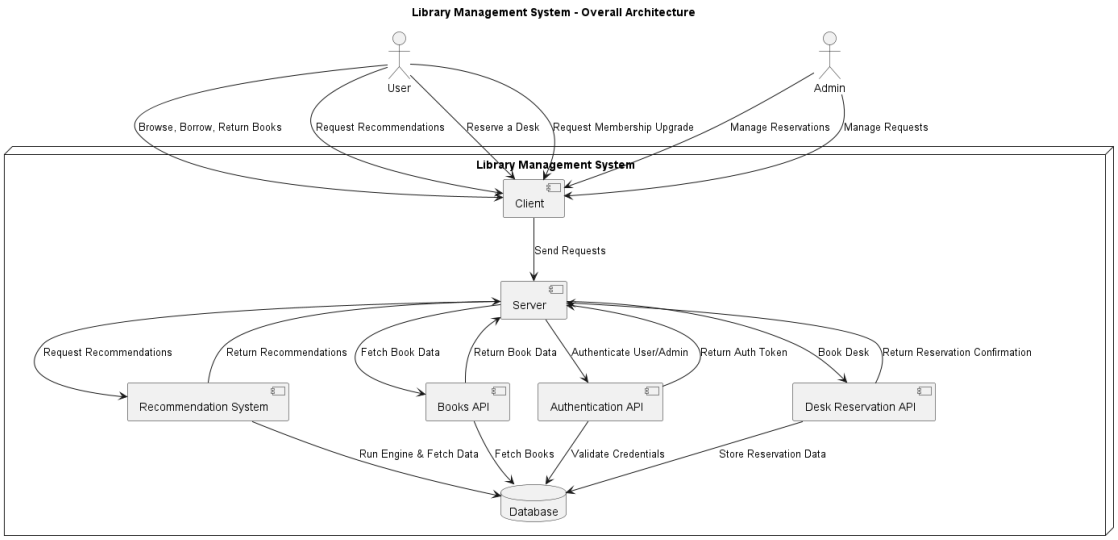


Figure 3.4: Overview of System Architecture

Client (Front-End): Provides an end-user interface, where users and admins can interact with the system. It facilitates actions such as:

- Browsing, borrowing, and returning books.
- Requesting book recommendations.
- Reserving library desks.
- Requesting membership upgrades.
- Allowing admins to manage reservations and approve membership upgrades.

The client communicates with the backend by sending requests, which are processed and responded to accordingly.

Server (Backend & API Layer): The backend acts as the center of the system, processing client requests and transactions of data. It communicates with different subsystems, including:

- Authentication Service – Validates user credentials, ensuring secure access for users and admins.
- Desk Reservation System – Handles desk booking requests, storing reservation details in the database.
- Recommendation System – Generates book recommendations by analyzing user's ratings or book title input based on content of books.
- Books APIs – Allow the client to interact with the server by sending requests and receiving responses, and fetching data from a database.

Database: The central database stores essential system data, including:

- User profiles, roles, and membership statuses.
- Book properties with ISBN number.
- Desk reservation details.
- Membership upgrade requests

The recommendation system also interacts with the database to fetch book data and set up content-based filtering to enhance suggestions.

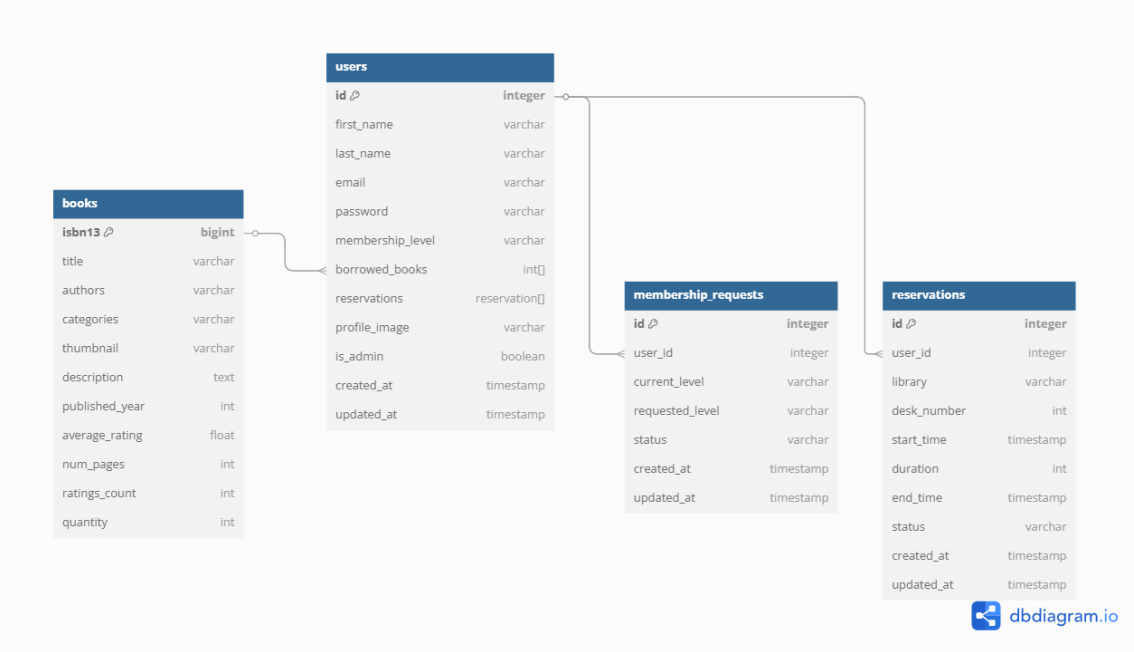


Figure 3.5: Database diagram

3.1.3 Machine Learning-Powered Recommendation

Systems that suggest items based on user preferences and item features are called recommendation systems and can be classified as collaborative filtering, content-based filtering, or hybrid approaches. The recommendation system of `Kitabkhana` filters recommendations based on content, focusing on item attributes (books) and user interactions with them [2]. Contrary to collaborative filtering that relies on user-user or item-item similarities, content-based filtering suggests items similar to those a user has previously interacted with based on item features. In `Kitabkhana`, users enter a book title or rate several books on a 1-5 star scale to receive recommendations based on book features that identify similar items.

Books are represented by attributes like title, genre, year, author, and description, which are transformed into feature vectors and identified by ISBN numbers [43]. Text fields like title, genre, and author description capture thematic similarities, while numerical fields like year give context. The system measures similarity between books by computing cosine similarity - the cosine of the angle of two feature vectors [44]. The cosine similarity between two vectors $\mathbf{x}$ and $\mathbf{y}$ is defined as:

$$\text{Cosine}(x, y) = \frac{\sum_{i=1}^{n} x_i y_i}{\sqrt{\sum_{i=1}^{n} x_i^2} \sqrt{\sum_{i=1}^{n} y_i^2}}$$

The system compares its feature vector with others in the library when a user types a book title. Or if the user rates several books, the ratings are added to a user preference vector which is compared with book feature vectors by cosine similarity. Table 3.1 illustrates an example dataset for books and Figure 3.6 shows cosine similarity calculations comparing genre-based feature vectors and similarity matrices identifying the most similar books.

Its recommendation system communicates with the broader architecture via defined interfaces to gather data for similarity calculations. It is a modular component that takes user input (book title or ratings) and returns a ranked list of book suggestions. Such a design integrates recommendations into the user interface thereby increasing discoverability and engagement with the library system.

Title	Genre	Year	Author	Description
The Great Gatsby	Computers	1995	F. Scott Fitzgerald	A young man newly rich tries to recapture the past and win back his former love, despite the fact she has married
To Kill a Mockingbird	Fiction	2006	Harper Lee	At the age of eight, Scout Finch is an entrenched free-thinker. She can accept her father's warning that it is a sin to kill a mockingbird, because mockingbirds harm no one and give great pleasure.

Table 3.1: Example book data in database

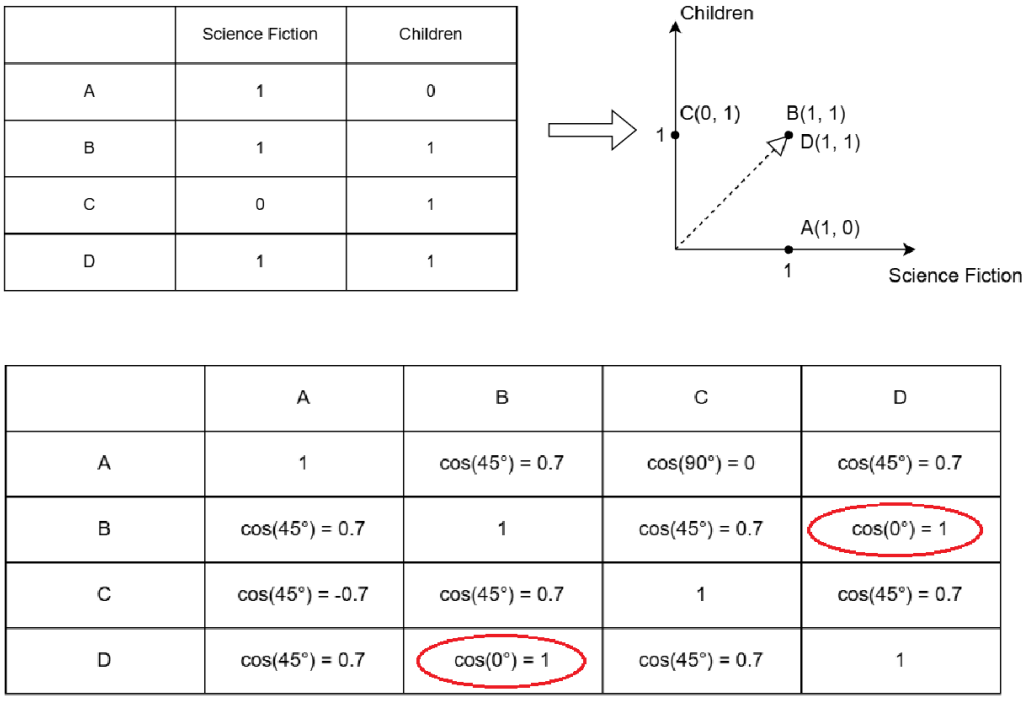


Figure 3.6: Cosine similarity calculation. The figure shows genre encoding for four books (A, B, C, D), a 2D vector representation of their feature vectors based on Science Fiction and Children genres, and a similarity matrix identifying the most similar books (B and D) with a cosine similarity of 1.

3.1.4 API Design and System Communication

The `Kitabkhana` system employs a structured Application Programming Interface (API) design to communicate with external services, following Representational State Transfer (REST) principles [4]. APIs are modular, with each module handling a specific function independently. This design supports maintainability and future extensions, such as integrating new external services. Security is incorporated through authentication and role-based access control to protect sensitive operations [3]. The system defines key APIs for internal interactions:

- **Authentication API**: Manages user login, registration, and role-based access control. It verifies credentials and restricts access to sensitive features like admin tasks or personal data for authorized users only.
- **Desk Reservation API**: Handles desk booking requests, checks availability, resolves conflicts, and updates reservation records.
- **Recommendation API**: Uses user inputs (e.g., book titles and ratings) with content-based filtering and cosine similarity to generate personalized book recommendations.
- **Admin API**: Supports administrative functions such as managing book inventories, processing membership upgrades, and handling desk reservations.

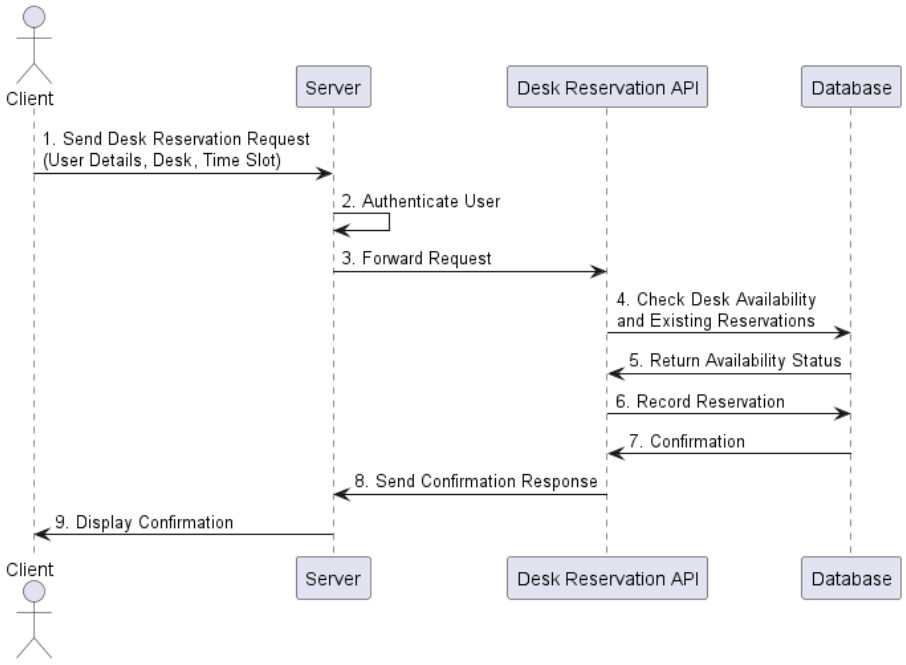


Figure 3.7: API Workflow Diagram for Desk Reservation API

3.1.5 UML Diagrams

To provide a clear visual representation of Kitabkhana's system design and workflows, several UML diagrams have been created [45].

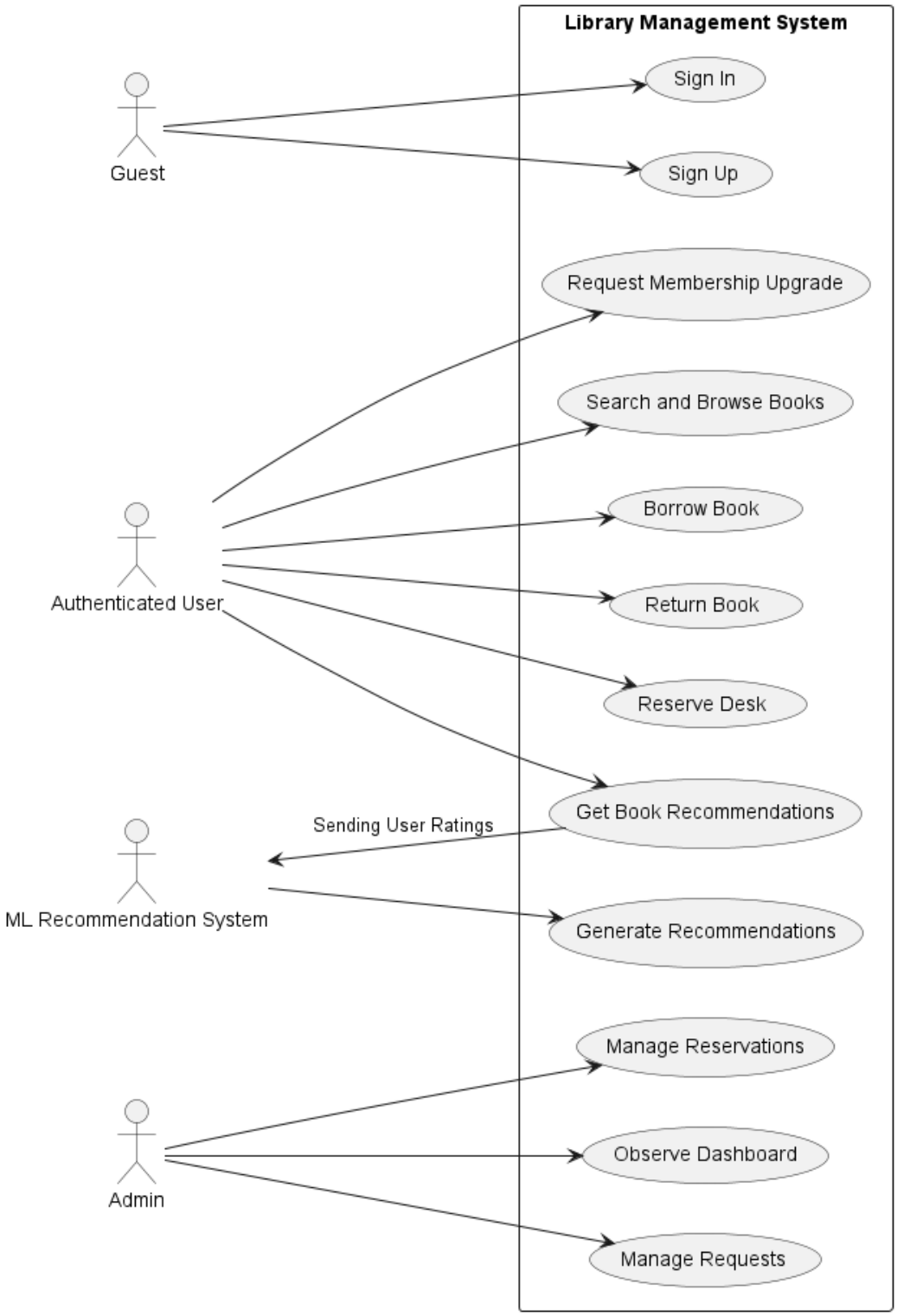


Figure 3.8: The Use Case Diagram provides a high-level overview of system functionalities.

The Book Borrowing use case describes how users borrow and return books. The process starts when an authenticated user sends a request via the client interface to the Book Borrowing API to check availability. If available, the API updates the database to add the book to the user's rental list, and a confirmation appears in the `My Rentals` section. To return a book, the user submits a return request, and the API verifies its presence, removes it from the database, and updates the interface accordingly (see Figure 3.9).

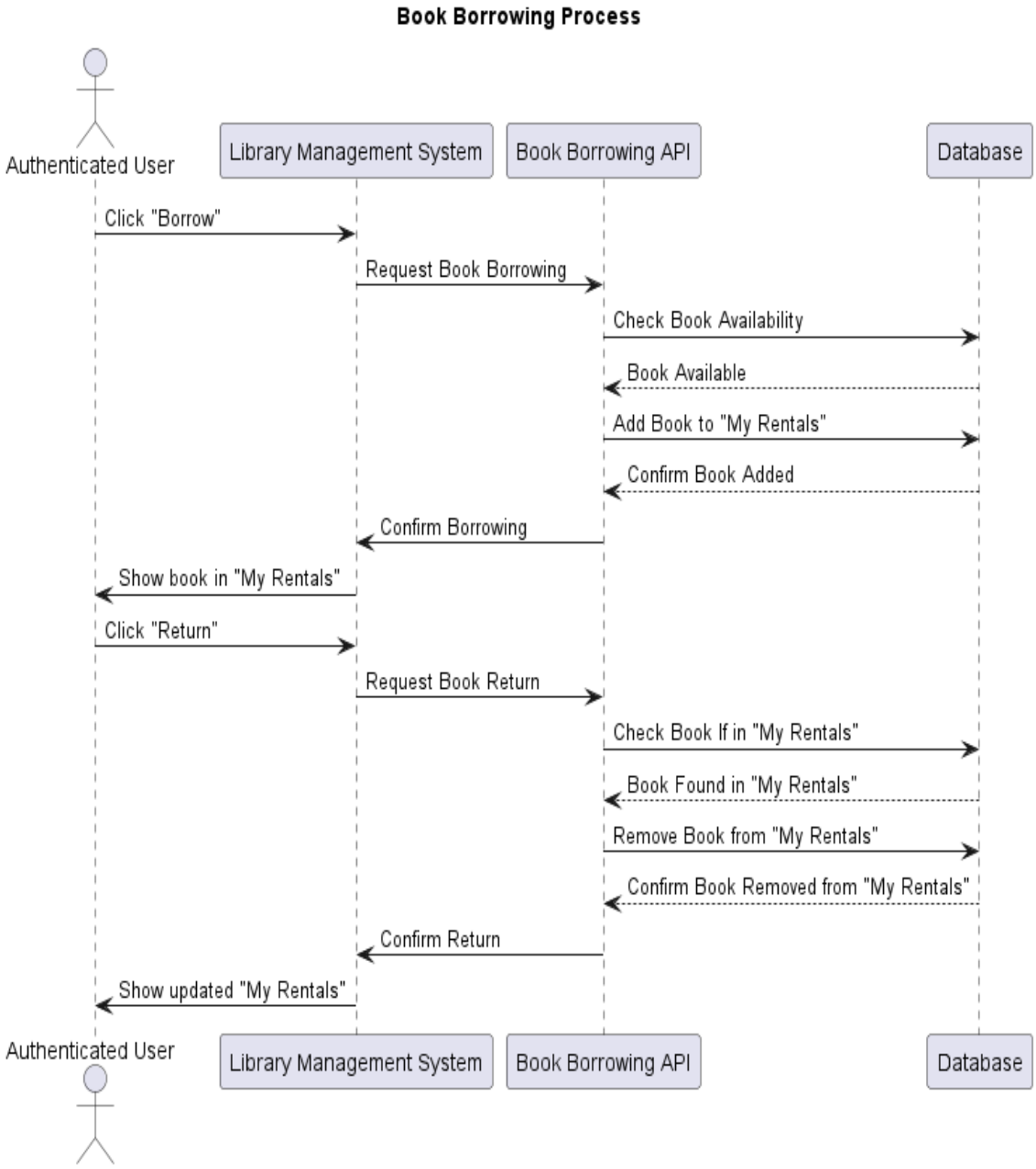


Figure 3.9: Book Borrowing Process Workflow

The Machine Learning Recommendation use case depicts how the system suggests books to users based on user's ratings and books' properties. The recommendation engine processes those values to generate personalized book recommendations, improving user engagement.

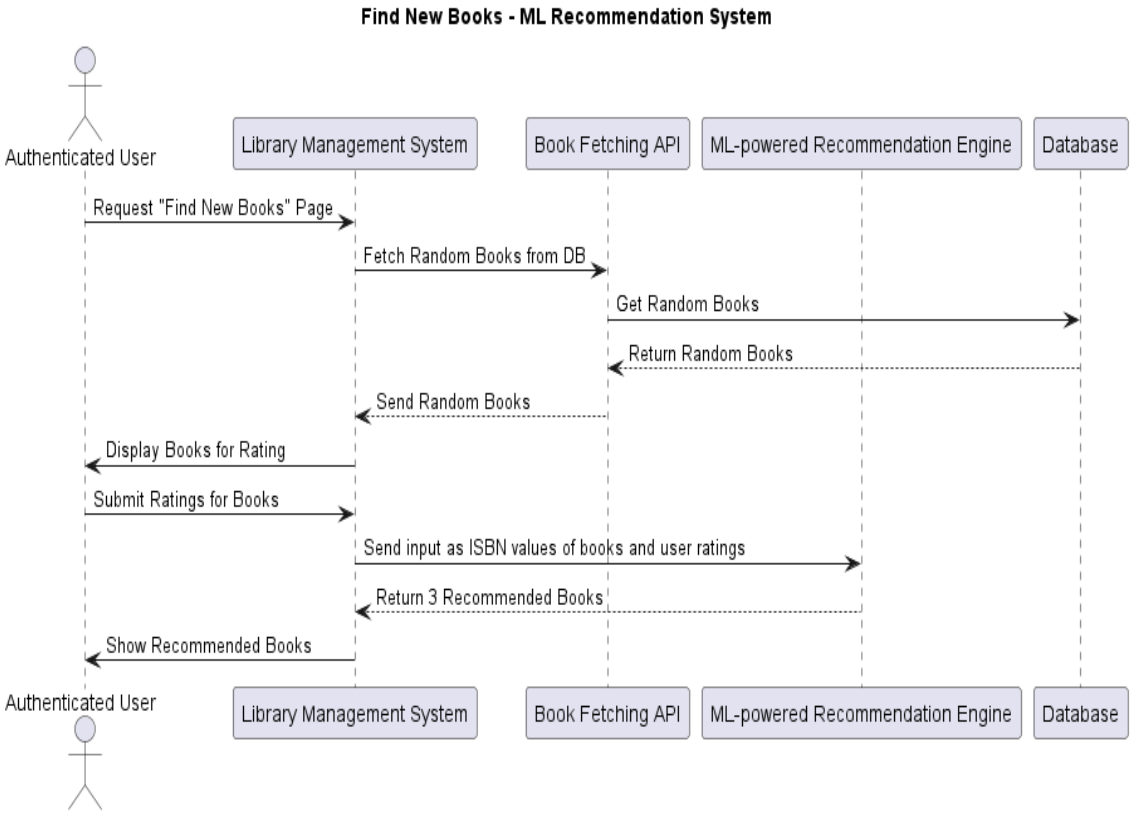


Figure 3.10: ML Recommendation Process Workflow

The Request Membership Upgrade use case represents the process of users upgrading their membership. A user selects a membership plan (Silver, or Gold) and submits a request. The system validates the request, admin approves it, and the user's privileges are updated accordingly.

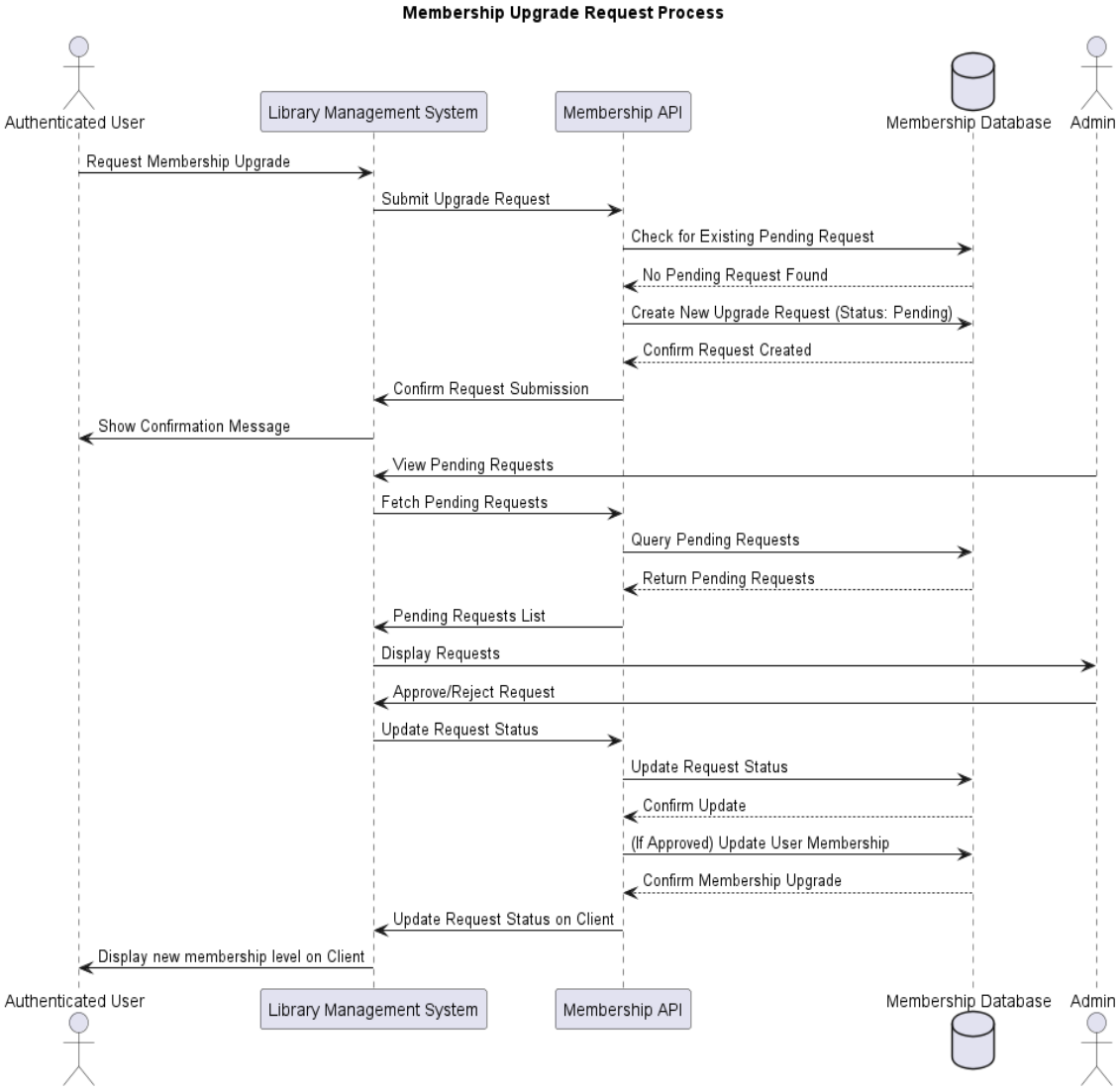


Figure 3.11: User Requesting Membership Workflow

3.2 Implementation

In this section, we will dive into the implementation details of the key components of the application. We will explore how core features were developed, highlight technical challenges encountered during the development process, and explain the problem-solving strategies and architectural decisions made to overcome them.

3.2.1 System Architecture and Technology Stack

This application is built using the MERN stack, a full-stack JavaScript framework that consists of the initials of MongoDB (database), Express.js (backend framework), React.js (frontend library), and Node.js (JavaScript runtime) [46]. It follows a client-server architecture, with the front-end communicating with the server via REpresentational State Transfer (RESTful) Application Programming Interface (API) endpoints [4]. The book recommendation model, developed in Python, operates as a Flask microservice, ensuring modularity and scalability [12, 29, 30]. The technology stack is categorized by component roles below.

Front-End Technologies

The front-end provides an interactive user interface.

- **React.js**: JavaScript library for dynamic, component-based UI developed by Facebook (Meta) [47].
- **HTML5**: Structures the front-end interface [48].
- **CSS3 and CSS modules**: Styles the front-end for responsive design [49, 50].
- **JavaScript (ECMAScript 2020) and JSX (JavaScript XML extension language)**: Enables client-side interactivity [51, 52, 53].
- **Recharts**: Charting library for React dashboards [41].
- **React Toastify**: Toast notifications for user feedback [34].

Back-End Technologies

The back-end handles server-side logic and service integration.

- **Node.js**: JavaScript runtime for scalable server operations [10].
- **Express.js**: Web framework on Node.js for API routes [54].
- **npm**: Manages dependencies and runs the Node.js server [11].
- **Python**: Used for the recommendation model [12].
- **Flask**: Deploys the recommendation microservice [29, 30].

APIs and Communication

Technologies facilitating client-server and external communication.

- **REST APIs**: Enables client-server communication [4].
- **HTTP/1.1**: Protocol for API requests [55].
- **Axios**: HTTP client for front-end API requests [56].
- **Cross-Origin Resource Sharing (CORS)**: A browser mechanism which enables controlled access to resources. [57, 58, 59]
- **Socket.IO**: WebSocket library for real-time dashboard updates [60, 40].

Security Technologies

Security mechanisms protect user data and access.

- **JSON Web Tokens (JWT)**: Handles user authentication [25].
- **Password Hashing with Salt**: Secures password storage [26].
- **Role-Based Access Control (RBAC)**: Restricts access by user roles [3].

Database Technologies

The database layer manages data storage.

- **MongoDB**: NoSQL database for flexible data storage [61].
- **MongoDB Atlas**: Cloud-hosted MongoDB for deployment [13].

Development and Collaboration Tools

Tools supporting development and collaboration.

- **Git**: Version control system [14].
- **GitLab**: Repository hosting platform [22].
- **Visual Studio Code**: Code editor for development [15].

Testing and Debugging Tools

Tools used for testing and debugging the application.

- **Jest**: JavaScript testing framework for unit testing [62].
- **Postman**: Platform for API testing and validation [63].
- **Chrome DevTools**: Browser tool for inspecting network, DOM, and console [64].

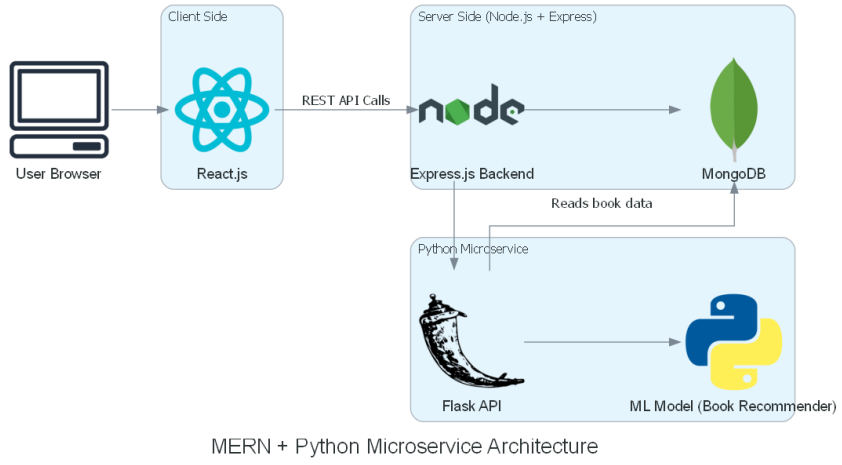


Figure 3.12: High-Level System Architecture

3.2.2 User Authentication and Authorization

Ensuring secure access to the application is a critical part of the system's design. This section outlines how user authentication and authorization are implemented on both the server and client sides to protect sensitive routes and user data.

Authentication is handled using JSON Web Tokens (JWT), which are generated upon successful login and are used to verify user identity on protected routes. Key components include:

- **User Model (`models/User.js`)**: The `User` model, defined using Mongoose [65], structures the user data in the MongoDB database.
  - `email` and `password` are used to identify and authenticate users during registration and login.
  - `isAdmin`: A boolean variable assigned to users during registration (default value: `false`). It determines whether the user has administrative privileges and is essential for authorizing access to admin-specific routes.
  - `generateAuthToken` method: This method generates a JWT by signing the user's id with a private key defined in .env file and setting an expiration of 7 days:

```
1 userSchema.methods.generateAuthToken = function () {
2     const token = jwt.sign({ userId: this._id },
3         process.env.JWTPRIVATEKEY, {
4             expiresIn: "7d",
5         });
6     return token;};
```

- **JWT Authentication**: When a user logs in, the server generates a JWT containing the user's ID and other relevant claims. This token is then sent to the client for storage.
- **authMiddleware (middleware/auth.js)**: This middleware extracts the `x-auth-token` header from incoming requests, verifies the token using a secret key, and attaches the corresponding `userId` to the request object for downstream use.

- **adminMiddleware (middleware/admin.js)**: To protect admin routes, this middleware checks whether the authenticated user has the `isAdmin` flag set to true in the database.
- **uploadMiddleware** (`middleware/upload.js`): This middleware, implemented using Multer [66], handles profile picture uploads. It validates the file format (JPEG, PNG, GIF) and enforces a size limit of 2MB, storing the uploaded files locally in the `uploads/` directory.
- **User Routes (routes/users.js)**:
  - `POST /api/signup`: Registers a new user by saving their credentials and details to the database.
  - `POST /api/login`: Verifies user credentials and returns a signed JWT on success.

On the client side, the JWT is stored in `localStorage` upon successful login. This token is then included in the `x-auth-token` header of subsequent API requests to authenticate the user.

- **Protected Routes**: Pages such as the admin dashboard are protected by conditional logic. If the user is not authenticated, they are redirected to the login page.
- **Conditional Rendering**: The UI components are dynamically rendered based on the authentication and authorization state of the user.

3.2.3 Main Page, Books Model, and API

The main page of **Kitabkhana** serves as the primary interface for users to explore books, offering sections for popular books and genre-based categories.

The server-side implementation involves defining the `Book` model, sourcing data, and creating API endpoints to fetch books for the main page.

- **Book Model (`models/Book.js`)**: The `Book` model (see Code 3.1 is defined using Mongoose [65] to structure book data in the MongoDB database.

```
const bookSchema = new mongoose.Schema({
    isbn13: { type: Number, required: true, unique: true },
    title: { type: String, required: true },
    authors: { type: String, required: true },
    categories: { type: String, required: true }, // Used for
       genre grouping
    thumbnail: { type: String, required: true },
    description: { type: String, required: true },
    published_year: { type: Number, required: true },
    average_rating: { type: Number, required: true,
    min: 0,
    max: 5,
    default: 0, get: (v) => Number(v.toFixed(2)),
    },
    ratings_count: { type: Number, required: true, default: 0
       }, // Imported from public dataset
    num_pages: { type: Number, required: true },
    quantity: { type: Number, required: true,
    default: 2, // Initially, 2 copies are available
    min: 0,
    },
});
```

Code 3.1: The `Book` model schema in Mongoose

The schema defines the key attributes used in the application. The model includes fields such as `title`, `categories`, `published_year`, `thumbnail`, `quantity`, and `description`, which are used to populate the UI components for each book.

The `categories` field is used to group books by genre, while `average_rating` and `ratings_count` are used to calculate popularity for the popular books section.

- **Data Source**: The book data was obtained from a public dataset on Kaggle [67]. The dataset originally contained more fields, but only the relevant ones have been used. The data was preprocessed to remove unavailable entries, and missing values were replaced with `Unknown`.
- **Popular Books API** (`GET /api/books/popular`): This endpoint fetches the top 15 popular books based on a custom popularity metric (see Code 3.2).

```
popularity_score = book.average_rating * Math.sqrt(book.
    ratings_count)
});
```

Code 3.2: Popularity Score Formula

- **Categories API** (`GET /api/books/categories`): This endpoint groups books by their `categories` field and returns a categorized list (see Code 3.3). Each category is limited to 12 books to avoid overwhelming the client with large datasets.

```
// Route for categories
router.get("/categories", async (req, res) => {
  try {
    const books = await Book.find().lean();
    const categories = {};
    books.forEach((book) => {
      const category = book.categories?.trim();
      if (!category || category.toLowerCase() === "unknown") {
        return;
      }
      if (!categories[category]) {
        categories[category] = [];
      }
      categories[category].push(book);
    });
```

```
    const groupedCategories = Object.fromEntries(
      Object.entries(categories).map(([category, books]) => [
        category,
        books.slice(0, 12), // Limit to 12 books per category
      ])
    );
    res.json(groupedCategories);
    ...
  }
});
```

Code 3.3: API endpoint for grouping books based on genre

The client-side of the application fetches data from API endpoints and renders the main page with two sections: Popular Books and Genre Categories. Each is displayed in a wide, scrollable container with hidden overflow and horizontal scrolling. An auto-scrolling feature further enhances navigation and user experience.

Displaying genres introduced several challenges. Some genres had very few associated books, which disrupted layout balance. To fix this, a client-side filter was added to display only genres with at least ten books.

The large number of genres also affected performance by making the main page too long. To improve this, pagination was introduced: six genres are shown initially, with three more added on each "Load More" click. This keeps load times fast and content manageable.

Lastly, books with the categories field set to "unknown" led to an unnecessary "Unknown" genre. The GET /api/books/categories endpoint was updated to exclude such entries (see Code 3.3), ensuring only relevant genres appear on the page.

3.2.4 Book Rental and Return

Book rental and return functionality are the core functionalities of **Kitabkhana** application, allowing users to borrow and return books efficiently.

On the server-side API endpoints have been created for borrowing and returning books, and the `User` and `Book` models adopted to manage user data and book inventory. The `User` model, defined using Mongoose [65], includes the `borrowedBooks` field, an array of `isbn13` numbers representing borrowed books, and the `membershipLevel` field, which determines borrowing limits (see Section 2.2.6 for membership details). The `Book` model, also defined using Mongoose [65], features the `isbn13` field as a unique identifier and the `quantity` field to track available copies, defaulting to 2 for this project (see Code 3.1 for the full book schema). Two key endpoints handle the rental and return processes, each split into condition checks and implementation for clarity.

Borrow Functionality

The `POST /api/rental/borrow` route enables borrowing a book with `isbn13` by authenticated users. It's initiated by a set of checks to determine if the request is valid. It checks the `isbn13` for valid format, user exists, user has not borrowed the maximum number of books based on `membershipLevel` limits, whether the book is already borrowed by the user, and whether the book is available `(quantity > 0)` (see Code 3.4.

On successful checks, the implementation is carried out by updating the database and sending WebSocket events (see WebSocket section) for real-time notifications. The `quantity` for the book is reduced by 1, `isbn13` is appended to `user.borrowedBooks` array, and following WebSocket events are sent:

- `quantityUpdate` for updating book quantity for all clients,
- `borrowedBooksUpdate` for updating user borrowed books and dashboard,
- `totalBorrowedBooksUpdate` and `borrowedByGenreUpdate` for the dashboards of admins.

The response confirms a successful borrowing with the updated borrowedBooks array, and shows toast notification on user-side.

```
// Borrow a book (add to borrowedBooks)
router.post("/borrow", authMiddleware, async (req, res) => {
  try {
    // .. Different condition checkings before implementation

    // Check borrowing limit based on membershipLevel
    const borrowLimits = {
      bronze: 6,
      silver: 12,
      gold: 18,
    };
    const limit = borrowLimits[user.membershipLevel] || 6;
    if (user.borrowedBooks.length >= limit) {
      return res.status(400).json({
        message:
          "You have reached your borrow limit, please return one of
              the books, or request for membership upgrade.", }); }

    // .. Check if already borrowed
    // .. Check book availability with isbn13 and quantity

    // Decrease the book quantity
    // Add the book to the user's borrowedBooks
    // ... Websocket events
});
```

Code 3.4: "Membership limit checking on Borrow API endpoint"

Return Functionality

The POST /api/rental/return handles the process of returning, also separated into condition checks and implementation. The condition checks verify the isbn13, that the user exists, and that the book is present within the user's borrowedBooks array, returning error messages if a check is not passed. After a successful pass of the checks, the implementation removes the isbn13 from the user's borrowedBooks, increases the quantity of the book by 1, and emits WebSocket events to update UI in real-time. These events are mirrored within the borrow endpoint, maintaining consistency within real-time updates across clients.

3.2.5 ML-Powered Book Recommendation Engine

To deliver intelligent and personalized book suggestions, a **machine learning-powered content-based recommendation engine** was developed and implemented as a modular microservice. The engine utilizes content-based filtering techniques, combined with natural language processing (NLP) and machine learning algorithms, to generate accurate recommendations based on a user's book preferences.

System Overview

The core recommendation logic relies on **TF-IDF vectorization** [68] and **cosine similarity** [69], which allows the system to interpret textual metadata and assess the similarity between books. Source data is retrieved from a MongoDB collection, containing attributes such as title, authors, categories, descriptions, publication years, and ISBN identifiers (see Code 3.5). This data is sourced from a public dataset on Kaggle [67].

```
# Connect to MongoDB using the URI from .env
mongo_uri = os.getenv('MONGO_URI')
if not mongo_uri:
    raise ValueError("MONGO_URI is not set in .env file")

client = MongoClient(mongo_uri)
db = client['test']
books_collection = db['books']
```

Code 3.5: Importing Book Dataset from MongoDB

Preprocessing is conducted on selected fields by converting text to lowercase, removing punctuation and stopwords, and applying lemmatization using the `NLTK` library [70]. These fields are then concatenated into a `combined_features` column, which serves as the content profile for each book. See Code 3.6 for the preprocessing function.

```
def preprocess_text(text):
    if pd.isna(text):
        return ""
    text = text.lower()                    # Lowercase
    text = re.sub(r'[^\w\s]', '', text)    # Remove punctuation
    text = re.sub(r'\s+', ' ', text).strip()
    words = [
        lemmatizer.lemmatize(word)
        for word in text.split()
        if word not in stop_words ]
    return ' '.join(words)
```

Code 3.6: Preprocessing of dataset

These fields are then concatenated into a `combined_features` column that represents the content profile of each book (see Code 3.7).

```
books_cleaned['combined_features'] = (
    books_cleaned['categories'] + ' ' +
    books_cleaned['authors']    + ' ' +
    books_cleaned['published_year'].astype(str) + ' ' +
    books_cleaned['description'] )
```

Code 3.7: Combining features column with keywords from preprocessed fields

```
print(books_cleaned['combined_features'])
```

```
0       fiction marilynne robinson 2004.0 novel reader...
1       detective mystery story charles osborneagatha ...
2       american fiction stephen r donaldson 1982.0 vo...
3       fiction sidney sheldon 1993.0 memorable mesmer...
4       christian life clive staple lewis 2002.0 lewis...
                              ...
6805    philosophy sri nisargadatta maharajsudhakar di...
6806                      mysticism khalil gibran 1993.0
6807                    book burning ray bradbury 2004.0
6808    history georg wilhelm friedrich hegel 1981.0 s...
6809    literary criticism helena gricetim wood 1998.0...
Name: combined_features, Length: 6810, dtype: object
```

Figure 3.13: `combined_features` column

The resulting dataset is then transformed into numerical vectors using Term Frequency-Inverse Document Frequency (TF-IDF), reducing text into a machine-readable format suitable for similarity calculations (see Code 3.8).

```
tfidf = TfidfVectorizer(stop_words='english', max_features=5000)
tfidf_matrix =
tfidf.fit_transform(books_cleaned['combined_features'])
```

Code 3.8: TF-IDF vector

User Interaction and Learning Strategy

Users may interact with the engine via two modes:

1. **Manual Rating Input**: Users provide ratings for random 6 books, and client sends these ratings as ISBN:rating pairs (see Code 3.9). Each rating influences the user's content profile, and these weights are factored into the recommendation scoring using cosine similarity.

2. **Search-Driven Input**: A custom search bar lets users select a book by title from database. Then the system assigns a *default five-star rating* to simulate strong preference.

```
user_ratings = {
    9780439554893: 5,  # Harry Potter and the Chamber of Secrets
    9780439651417: 4,  # Hunting the Hunter
    9780439574273: 1,  # The Invisible Man
    9780439574280: 4,  # The Adventures of Sherlock Holmes
    9780439812313: 5   # Runny Babbit
}
```

Code 3.9: Ratings dictionary is used to construct the profile

The TF-IDF matrix acts as a learned representation of the book dataset, and the user profile vector is constructed by aggregating the TF-IDF vectors of rated books. The model computes similarity scores between this profile and the rest of the catalog, ranks them, and returns the top N recommendations while excluding already rated books.

```
user_profile = np.zeros(tfidf_matrix.shape[1])
total_weight = 0

for isbn, rating in user_ratings.items():
    idx = find_book_index_by_isbn(isbn, books_cleaned)
    user_profile += tfidf_matrix[idx].toarray().flatten() * rating
    total_weight  += rating

if total_weight > 0:
    user_profile /= total_weight
```

Code 3.10: Each rated book's TF-IDF vector is multiplied by its rating, summed into a single profile vector, and normalized

```
scores = cosine_similarity(
    user_profile.reshape(1, -1), tfidf_matrix
)[0]
recommended_indices = np.argsort(scores)[::-1][:3]
recommended_books   = books_cleaned.iloc[recommended_indices]
```

Code 3.11: Cosine similarity against all books produces a ranked list of recommendations.

Microservice Architecture and Integration

The recommendation engine is built as an independent Flask microservice, ensuring scalability and clear separation from the main app. When a POST request is sent to the /recommend endpoint, the service loads and preprocesses data, creates `combined_features` column, initializes TF-IDF matrix, build user profile, computes similarity scores, and returns JavaScript Object Notation (JSON)-formatted recommendations [71] (see Code 3.12). An Express.js route (recommend.js) handles authentication and bridges communication between the client and the microservice (see Code 3.13).

```python
@app.route('/recommend', methods=['POST'])
def recommend_books():
    user_ratings = request.json
    tfidf_matrix, books_df = update_book_dataset()
    # ... build user_profile & compute similarities ...

    # Include isbn13 in the returned fields
    recommended_books = books_df.iloc[recommended_indices][
        ['title', 'authors', 'categories', 'published_year', '
           thumbnail', 'description', 'isbn13', 'quantity']
    ].to_dict(orient='records')
    for book in recommended_books:
        book['isbn13'] = int(float(book['isbn13']))  # Ensure
           isbn13 is an integer
        book['quantity'] = int(book['quantity'])

    return jsonify(recommended_books)
```

Code 3.12: Flask microservice for sending results to server

```javascript
router.post("/recommend", authMiddleware, async (req, res) => {
  const userRatings = req.body;
  try {
    const response = await axios.post(
      "http://localhost:5000/recommend",
      userRatings,
      {
        headers: { "Content-Type": "application/json" },
      }
    );
    res.json(response.data);
  } catch (error) {
    console.error("Error fetching recommendation:", error.message);
    res.status(500).json({ error: "Recommendation service
       unavailable" });
  } });
```

Code 3.13: API endpoint in server for communcation between client and microservice

Example Test Run

To illustrate the engine in action, consider the following user input (only ISBN and rating):

ISBN13	Rating	Title	Genre	Year	Author
9780007113804	5	Murder in Mesopotamia	Detective and mystery stories	2001	Agatha Christie
9780007154845	5	The Mysterious Mr. Quin	Detective and mystery stories	2003	Agatha Christie
9781579126896	5	Death on the Nile	Fiction	2007	Agatha Christie

Table 3.2: User Ratings and Book Metadata for the Test Run

The microservice returns the following top-3 recommendations:

Title	Genre	Year	Author
Hercule Poirot's Casebook	Detective and mystery stories	1989	Agatha Christie
Hallowe'en Party	Poirot, Hercule	2001	Agatha Christie
The Regatta Mystery and Other Stories	Fiction	1984	Agatha Christie

Table 3.3: Output Recommendation

As shown in Table 3.3, the system performs effectively, offering recommendations that reflect a strong understanding of the user's preferences. It not only identifies books by the same author in same genres but also suggests titles within the different genre by the same author, showing an awareness of both content and style. One of the recommended books also shares a similar publication year (2001), indicating that the model is sensitive to temporal patterns as well. Overall, the results suggest that the system is capable of capturing subtle relationships in the data and delivering personalized suggestions that feel relevant and well-matched.

3.2.6 Recommendation Pages

Overview of Recommendation Features

In **Kitabkhana**, recommendation features are integrated into three areas to enhance user experience: recommendation by title, by ratings of random books, and based on rented books. These features were designed to offer tailored book suggestions, addressing varied user needs. This section explores their implementation and challenges.

Recommendation by Title

The recommendation-by-title feature was developed to suggest books based on a user-entered title. A custom search bar was created, allowing the system to search the database and suggest titles as the user types. This is powered by a server-side API endpoint, `/search`, which uses a case-insensitive string-match search on the `title` field in the MongoDB `books` collection, returning up to 5 matches with details like `isbn13` and `title`, as shown in Code 3.14. Selected books are given a default rating of 5 and sent to the recommendation microservice.

```
// Route for searching books
router.get("/search", async (req, res) => {
    // .. Check if query exists
    const { query } = req.query;

    // .. Fetch all books (or a subset) from the database

    // Use KMP to filter books where the title contains the query
    const matchingBooks = books.filter((book) => kmpSearch(book.
        title, query));

    // Limit to 5 suggestions
    const limitedBooks = matchingBooks.slice(0, 5);

    res.json(limitedBooks); // Return found books
});
```

Code 3.14: High-Level overview of API endpoint for book title search

Recommendation by Ratings of Random Books

The recommendation-by-ratings feature was crafted to help users unsure of their tastes, offering suggestions based on ratings of varied books. An API endpoint, `/all`, fetches 50 books from MongoDB, from which 6 are randomly selected for rating. Users rate these on a 1-to-5 scale, chosen for its balance of detail, UI simplicity, and manageable calculations. Ratings and `isbn13` numbers are sent to the microservice, which suggests the top 3 similar books based on attributes like `title` and `categories`.

A key challenge was deciding which books to show: same ones each time, random, or one per genre. A random selection of 6 from 50 books was chosen to ensure variety, likely including different genres. The 1-to-5 rating scale was selected for its clarity and efficiency over scales like 1-to-10 (see Code 3.15).

```
router.get("/all", async (req, res) => {
  try {
    console.log("Fetching books from MongoDB...");
    const books = await Book.find().limit(50);
    console.log("Books fetched:", books.length, "documents");

    // .. Error handling
});
// ... Client
const dataLength = response.data.length;
const randomIndices = [];
while (randomIndices.length < 6 && randomIndices.length <
    dataLength) {
    const randomIndex = Math.floor(Math.random() * dataLength);
    if (!randomIndices.includes(randomIndex)) {
        randomIndices.push(randomIndex);
    }
}
const randomBooks = randomIndices.map((index) => response.data[
    index]);
```

Code 3.15: This API endpoint fetches 50 books from database, then client chooses 6 random books.

Recommendation Based on My Rentals

The recommendation based on My Rentals feature suggests books based on a user's rented books, mirroring real-world recommendations. The "My Rentals" page lists borrowed books (see Section 2.2.5). Clicking "Get Recommendations" sends the `isbn13` numbers to the microservice with a default rating of 5, returning the top 3 similar books for display.

The main challenge was ensuring meaningful recommendations with few rented books. A default rating of 5 was assigned to each, assuming positive user interest, allowing the microservice to focus on content-based similarity.

3.2.7 Desk Reservation System

In **Kitabkhana**, a desk reservation system was created to help users book study spaces in one of three libraries: Newton, Tesla, or Goethe. Each library has 64 desks arranged in an 8x8 grid, and users can reserve a desk for a specific time and duration based on their membership level.

Reservation Model and Implementation

The desk reservation system was designed to let users book a desk in a library for a chosen time, ensuring a smooth and fair process. A `Reservation` model was defined using MongoDB to store details like the user's ID (`userId`), library name, desk number (1 to 64), start time, duration, end time, and status, which can be "pending", "confirmed", or "cancelled". The system enforces rules based on membership levels—bronze (6 hours max), silver (9 hours), and gold (12 hours)—and limits reservations to one month in advance.

The reservation process begins when a user selects a library, date, start time, and duration. Availability is checked via the `/availability` endpoint, which identifies unbooked desks within the selected time slot. If desks are available, the user selects one and confirms the booking using the `POST /reservations` endpoint. The reservation is saved as "confirmed" after validating against overlaps and membership limits, and real-time updates are pushed to both user and admin dashboards via socket.io.

Users can cancel a reservation using the `DELETE /reservations/:id` endpoint, provided it belongs to them and hasn't passed. If valid, the reservation status is

set to "cancelled" and updates are sent via socket.io. Other key endpoints include `GET /reservations`, which fetches current reservations and marks completed ones, and `GET /availability`, which ensures accurate desk availability. Each endpoint handles proper validation and keeps the system in sync in real time.

Challenges

A main challenge was ensuring desks are not double-booked and users do not have overlapping reservations. A helper function, `checkOverlap`, was created to compare the requested time slot with existing reservations for the same user and desk, as shown in Code 3.16. Another issue was enforcing membership limits and time rules, which was handled by checking the user's membership level and validating the reservation time against the one-month limit during the booking process.

```
1  const checkOverlap = async (
2    userId,
3    startTime,
4    endTime,
5    excludeReservationId = null
6  ) => {
7    const overlappingReservations = await Reservation.find({
8      userId,
9      status: { $ne: "cancelled" },
10     startTime: { $lt: endTime }, // Existing reservation starts
           before new endTime
11     endTime: { $gt: startTime }, // Existing reservation ends after
           startTime
12     ...(excludeReservationId && { _id: { $ne: excludeReservationId
           } }),
13   });
14
15   return overlappingReservations.length > 0;
16 };
```

Code 3.16: Function to check for overlapping reservations

3.2.8 WebSockets

WebSockets were set up to allow instant updates across the application, such as book quantities, user reservations, and admin dashboard data. On the server side, a WebSocket server was created using `socket.io`, as shown in Code 3.17.

A challenge was ensuring updates reach the right user without delays. The room-based system (`user-userId` for users, `admin-room` for admins) was used to target updates accurately. When a client connects, they join their specific room based on their token, and updates like `quantityUpdate` or `reservationsUpdate` are sent to the right room, ensuring users and admins see changes instantly.

```
const io = new Server(httpServer, {
cors: {
    origin: "http://localhost:3000",
    methods: ["GET", "POST"],
    credentials: true,
    },
});

io.on("connection", (socket) => {
socket.on("joinAdminRoom", async (token) => {
    socket.join("admin-room");
    socket.emit("totalUsersUpdate", totalUsers);
    // .. Similar logic for other entities
    ...
    });

socket.on("joinUserRoom", (token) => {
    const decoded = jwt.verify(token, process.env.JWTPRIVATEKEY);
    const userId = decoded.userId;
    const userRoom = user-${userId};
    socket.join(userRoom);
    ...
    });
...
});
```

Code 3.17: WebSocket setup on the server

3.2.9 Membership Requests

The membership upgrade system allows users to request a higher membership level for additional benefits. On the server side, the `POST /membership/upgrade` endpoint checks if the requested level is valid (silver or gold), ensures the user upgrades only one level at a time, and verifies no existing pending requests. If valid, a `MembershipRequest` document is created and marked as "pending" in the user's profile. On the client side, users choose their desired level from the profile page, triggering a request to the server. Upon success, the profile reflects the pending status, and a confirmation message is shown.

A key challenge was preventing users from submitting multiple requests at once, which was solved by checking for existing pending requests before allowing a new one. Another issue was ensuring users can only upgrade to the next level, not skip levels, which was handled by comparing the current and requested levels in the defined order (bronze, silver, gold).

Admin Review Process

Admins can review and manage membership upgrade requests through a dedicated interface. Two endpoints handle these actions: `PUT /membership-request/approve/:requestId` to approve, and `PUT /membership-request/reject/:requestId` to reject requests. Approval updates the user's membership level, marks the request as "approved", and triggers a WebSocket event to refresh the admin dashboard. Rejection simply updates the request status to "rejected". On the client side, admins see a list of pending requests with user details and can approve or reject with one click, updating the list in real time.

To ensure data integrity, status checks were added so only pending requests can be processed.

3.3 Shortcomings and Future Implementations

Due to limited time for implementing this project, several shortcomings were identified, which impacted the system's functionality and user experience. These limitations are outlined below, along with proposed future implementations to address them, enhancing the overall robustness and usability of **Kitabkhana**.

3.3.1 Design Shortcomings

Design shortcomings reflect the inherent limitations in the system's architecture, user experience, and how the features are conceptualized. These issues often arise from initial design choices that need to be addressed to improve overall usability and scalability.

- **Inaccurate ML Recommendations**: The machine learning recommendation engine (see Section 3.2.5) can generate inaccurate genre-based recommendations due to books with "unknown" genres in the dataset. It happens, because at preprocessing step missing values replaced with "unknown". Future work should involve better data cleaning to exclude or impute missing values accurately, or source a more complete dataset.

- **Limited Recommendation Variety**: The recommendation-by-ratings feature (see Section 3.2.6) randomly selects 6 books from first 50 books for rating, which may not cover diverse genres. Future enhancements could ensure books are selected from distinct genres, increasing recommendation diversity.

- **Missing Personalized Recommendations**: The recommendation based on My Rentals (see Section 3.2.6) requires manual initiation. A future "For You" section on the main page, displaying recommendations after at least 3 rentals, could improve accessibility, with caching to manage request frequency.

- **Collaborative Filtering**: A potential enhancement to the recommendation system could involve incorporating `collaborative filtering`, which would suggest books based on other users' book rentals and preferences [72]. However, this approach requires a significant volume of user data, which is currently lacking. To enable this, the system should store and analyze user borrowing

history, ensuring adequate data is available to generate meaningful recommendations. This would require changes in data storage, potentially introducing a more complex user profiling system.

- **Hardcoded Membership Limits**: Membership limits (e.g., borrowing limits, reservation durations) are hardcoded (see Sections 3.2.4 and 3.2.7), in case of update it requires changes in different places in source code, which can cause potential bugs and requires time for identification every time. A centralized configuration file or database collection should be implemented to manage these limits dynamically. With this approach, new membership levels addition could be possible, such as platinum, diamond etc.

- **Inflexible Desk Reservations**: Users cannot edit desk reservations (see Section 3.2.7) and must cancel and rebook, which is inefficient. Adding a modification feature, along with email confirmations and reminders for reservations, would enhance usability.

- **Limited Admin Controls**: The admin panel (see Section 3.2.9) lacks comprehensive user and inventory management. Future implementations should allow admins to view user lists, suspend users, promote to admin, manage book inventory in a table format, add or modify books, add new libraries, and send emails to users.

- **Inefficient Machine Learning Pipeline**: The current machine learning recommendation engine (see Section 3.2.5) retrains the TF-IDF vectorizer and reconstructs the similarity matrix every time a recommendation request is made. This leads to unnecessary computational overhead, slower response times, and scalability issues as the dataset grows. Future improvements should involve training the model once at server startup and caching the preprocessed data in memory, refreshing it only when the underlying book dataset changes.

3.3.2 Implementation Shortcomings

Implementation shortcomings refer to specific technical or coding decisions made during development that directly impact the system's performance, security, or ease of maintenance. These issues often arise from time constraints or suboptimal solutions chosen during the project's lifecycle.

- **Vulnerable Token Storage**: Storing JWT in `localStorage` (see Section 3.2.2) is prone to XSS attacks. Future implementations should use `HttpOnly` cookies to securely store tokens, reducing vulnerability to such attacks [73].

- **Duplicate Book Additions in Recommendations**: In the recommendation-by-title feature (see Section 3.2.6), users can add the same book multiple times because the system does not prevent re-adding already selected titles. A future implementation could disable adding a book again once it has been selected, improving the user experience.

- **Profile Picture Security Risks**: Profile picture uploads get checked with middleware called `upload.js` for image format (jpg, png, gif) and size (<2MB), but remain vulnerable to attacks, such as renaming malicious files to bypass checks. Future enhancements should include stricter validation, such as verifying file content, and implement additional security measures like sandboxing uploads.

- **Missing Account Security Features**: The system lacks a "forgot password" option, account verification via email to prevent bot users, and email notifications for reservation actions or membership requests (not previously detailed). Future implementations should include email-based password recovery, account verification during registration, and notifications for reservation confirmations, cancellations, and membership request evaluations.

- **Limited Database Storage and Scalability**: The current implementation uses the free-tier MongoDB, which has a storage limit of 512MB. This restriction hampers the ability to handle high traffic and large volumes of data effectively. As traffic increases, the system may face performance bottlenecks or data storage issues. Future implementations should consider upgrading to

a paid tier of MongoDB or transitioning to more scalable database solutions. Additionally, leveraging technologies such as Kubernetes as a load balancer could improve the system's ability to handle high traffic, ensuring better scalability and performance.

- **Suboptimal Book Referencing with ISBN**: Books are referenced using their ISBN numbers in features like borrowed books and rated books (see Section 3.2.4). While functional, this approach is less efficient in a NoSQL database like MongoDB, as ISBN lookups require additional indexing and can lead to slower query performance. A more effective solution would be to use MongoDB's object references (e.g., `ObjectId`) to directly link records, improving query efficiency and maintaining referential integrity [74].

3.4 Testing

This part details the testing strategy for the text-based application including unit testing, automated API testing with Postman, manual UI testing, and webSocket testing. All types of testing verify the reliability, functionality, and real behavior of the application.

3.4.1 Unit Testing

Core algorithms such as string matching, quick sort are verified by unit tests for key backend functions using Jest [62]. The tests themselves were written in JavaScript with expectations handled by Jest's built-in assertion library. Test files under the test folder in the server part. These functions were tested:

- `kmpSearch`: Tested pattern matching for search functionality using `test` blocks [75]. For example, a test case verified that searching for "Great" in "The Great Gatsby" returns the correct index (4). Assertions used `expect(kmpSearch(...)).toBe(...)`, and all test cases passed.

- `computeLPSArray`: Validated the Longest Proper Prefix Suffix array computation for the KMP algorithm [75]. A test case checked that `computeLPSArray("AABAACAAB")` returns the array `[0, 1, 0, 1, 2, 0, 1, 2, 0]`. Assertions used `expect(computeLPSArray(...)).toEqual(...)`, and all tests passed.

- `quicksort`: Ensured correct sorting of book arrays by popularity in descending order [76]. Test cases included sorting an array of book objects (e.g., `[popularity: 5, popularity: 10]`), expecting `[popularity: 10, popularity: 5]`). Edge cases like empty arrays were tested using `expect(quicksort(...)).toEqual(...)`, and all tests passed.

- `partition`: Confirmed the partitioning logic for quicksort [76]. A test case verified that `partition` correctly positions the pivot in an array (e.g., `[5, 2, 9, 1]` with pivot 5 returns the correct pivot index). Assertions used `expect(partition(...)).toBe(...)`, and all tests passed.

- `checkOverlap`: Tested reservation overlap detection. A test case checked that two reservations (e.g., 2025-04-24T10:00:00Z to 2025-04-24T12:00:00Z and

2025-04-24T11:00:00Z to 2025-04-24T13:00:00Z) are correctly identified as overlapping using `expect(checkOverlap(...)).toBe(true)`. All test cases passed.

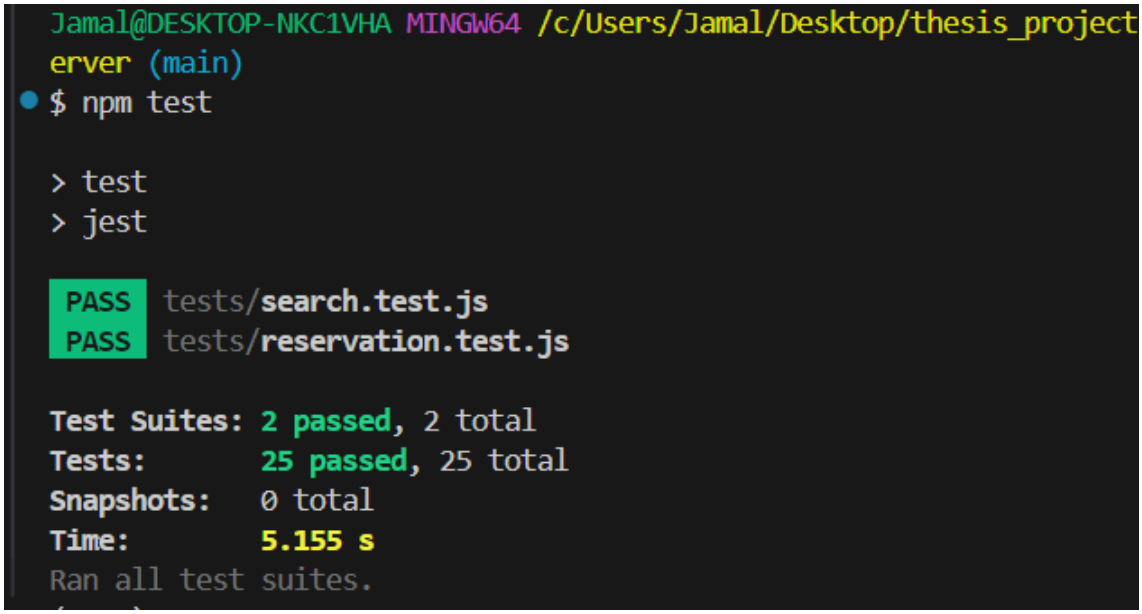


Figure 3.14: A total of 25 unit test cases were executed using Jest, all of which passed, ensuring the reliability of the backend algorithms.

3.4.2 Automated API Testing with Postman

Automated API tests were conducted using Postman [63] to validate the functionality and performance of the application's endpoints. A collection named `Kitabkhana API Tests` was created, covering 15 endpoints, including `GET /api/books/search`, `POST /api/rental/borrow`, `POST /api/reservations`, and admin routes like `GET /api/admin/membership-requests`. The testing was divided into functional and performance testing, as described below.

Functional Testing

Functional tests were conducted to ensure the API endpoints work as expected. Key aspects of the testing process include:

- **Setup**: Tests were organized into folders (Books, Rentals, Reservations, User, Admin) and with environment variables have been created for user, admin tokens, isbn13 number and reservation.

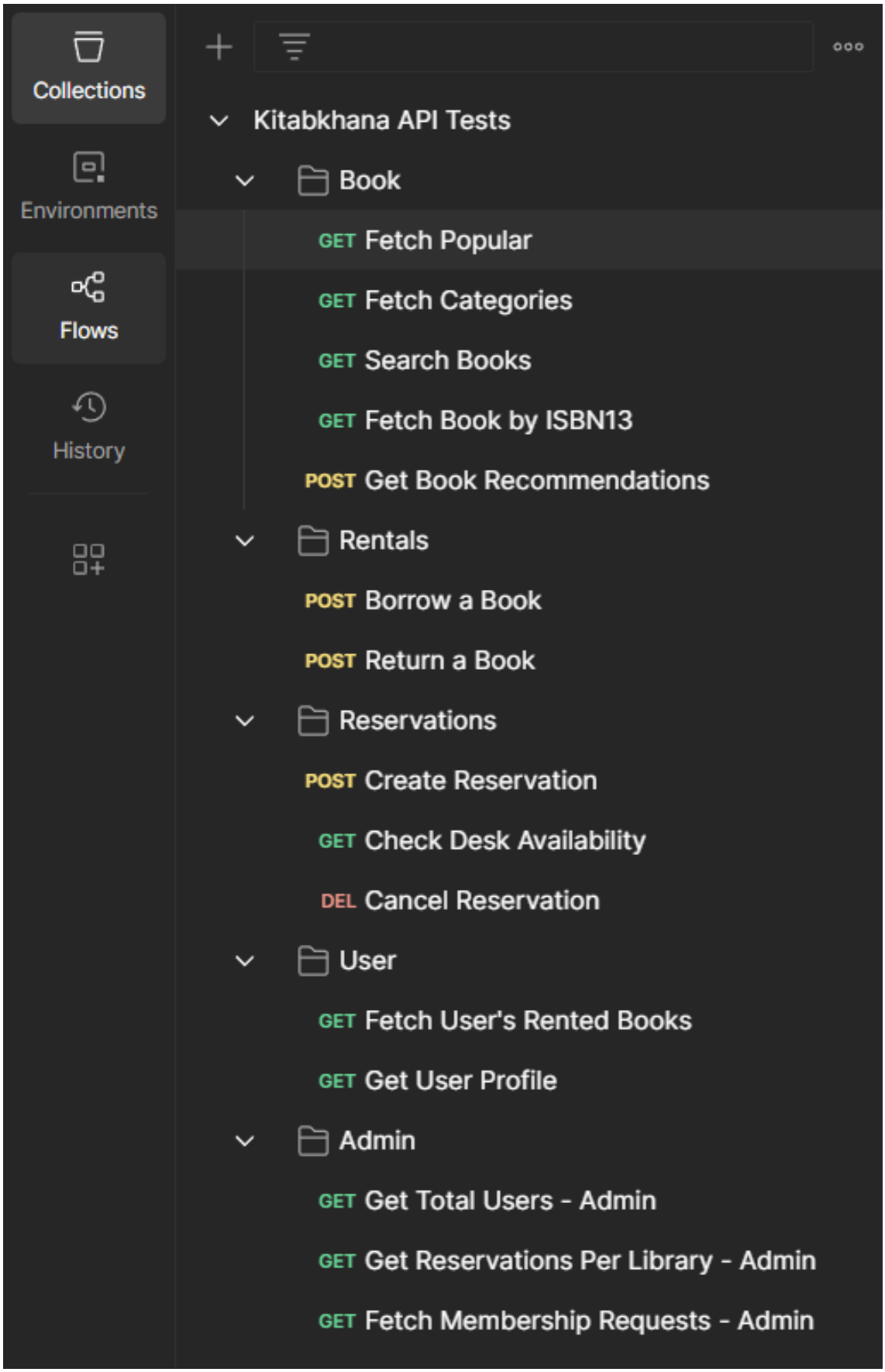


Figure 3.15: API endpoints structure in Postman

- **Test Scripts**: Each request included test scripts written in JavaScript using the Chai.js [77] assertion library (via Postman's pm.expect) to validate status codes, response formats, and data integrity.

```
pm.test("Status code is 200", function () {
    pm.response.to.have.status(200);
});

pm.test("Response is an array of 3 book recommendations",
    function () {
    const response = pm.response.json();
    pm.expect(response).to.be.an("array");
    pm.expect(response).to.have.lengthOf(3);
});

pm.test("Each recommendation has required book fields",
    function () {
    const response = pm.response.json();
    response.forEach(book => {
        pm.expect(book).to.be.an("object");
        pm.expect(book).to.have.property("isbn13");
        pm.expect(book).to.have.property("title");
        pm.expect(book).to.have.property("authors");
    });
});
```

Code 3.18: Test scripts for `POST /api/recommend/recommend` API endpoint

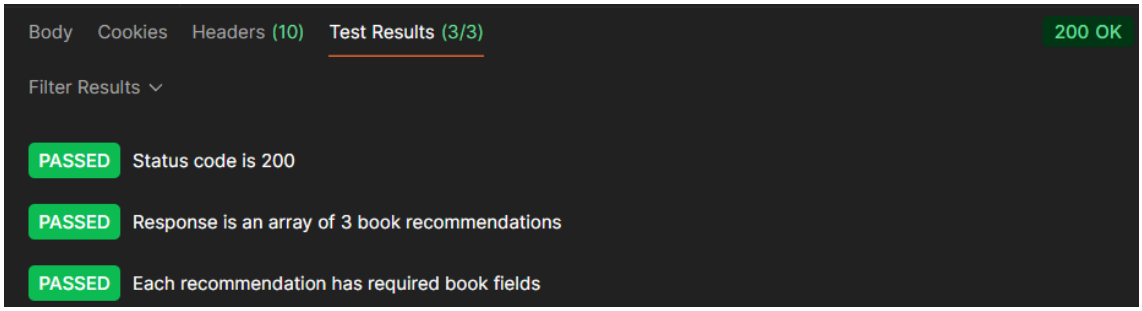


Figure 3.16: API testing script results

- **Results**: A total of 190 assertions were executed across 5 sequential iterations, with 38 assertions in each iteration. All tests passed successfully.

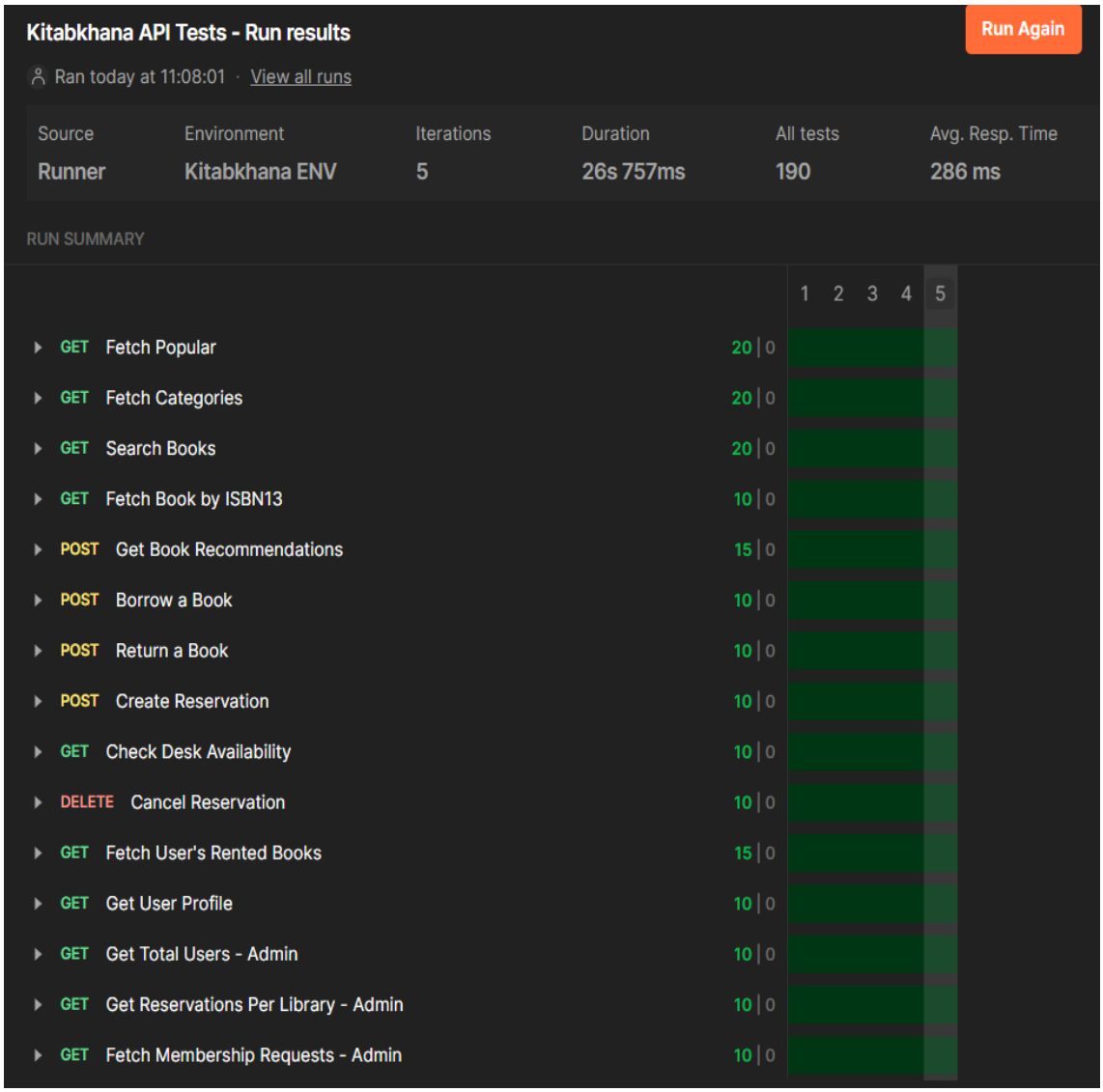


Figure 3.17: Functional Tests Run Result

Performance Testing

Performance tests were conducted to evaluate the API's behavior under load using Postman's performance testing feature. The test conditions and results are detailed below:

- **Setup**: A peak load profile was used with 10 virtual users (VUs) over 1 minute. The test started with 2 VUs for 12 seconds, ramp up to 10 VUs over 12 seconds, decrease to 2 over 12 seconds, maintains 2 VUs for 12 seconds, executing all requests sequentially.
- **Results**: A total of 390 requests were sent at 5.61 requests/second. The average response time was 738 ms. The error rate was 1.54%, indicating good reliability under load. See the attached performance test report on the next page.

Performance Test Report - Apr 23, 2025 (#8)

Open in Postman

Postman collection: Kitabkhana API Tests
Report exported on: Apr 23, 2025, 14:42:00 (GMT+2)

Test setup

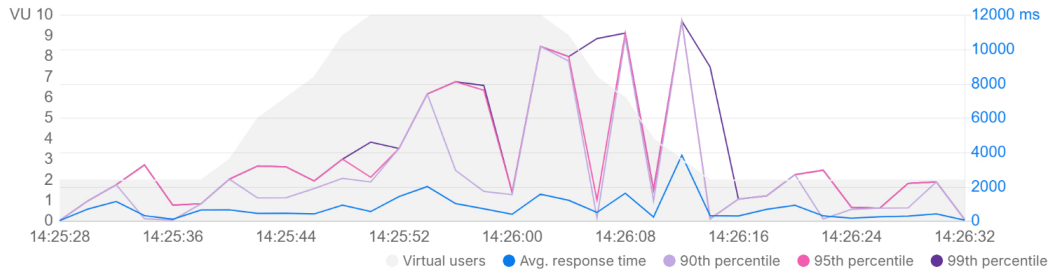
Virtual users	Start time	Load profile
10 VU	Apr 23, 14:25:24 (GMT+2)	Peak
Duration	**End time**	**Environment**
1 minute	Apr 23, 14:26:32 (GMT+2)	Kitabkhana ENV

1. Summary

Total requests sent	Throughput	Average response time	Error rate
390	5.70 requests/second	738 ms	1.54 %

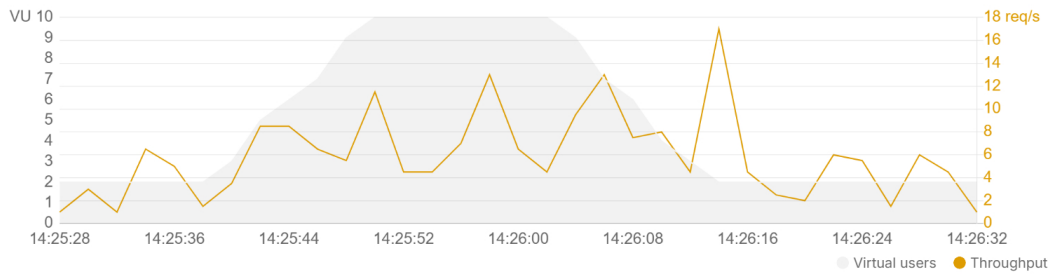
1.1 Response time

Response time trends during the test duration.



1.2 Throughput

Rate of requests sent per second during the test duration.



1.3 Requests with slowest response times

Top 5 slowest requests based on their average response times.

Request	Resp. time (Avg ms)	90th (ms)	95th (ms)	99th (ms)	Min (ms)	Max (ms)
POST Get Book Recommendations http://localhost:4000/api/recommend/recommend	6,655	10,700	10,970	11,675	2,141	11,675
GET Fetch Categories http://localhost:4000/api/books/categories	1,685	2,982	3,464	4,254	779	4,254
GET Search Books http://localhost:4000/api/books/search?query=Great	1,486	2,301	2,917	3,241	711	3,241
POST Borrow a Book http://localhost:4000/api/rental/borrow	190	283	302	368	21	368
POST Return a Book http://localhost:4000/api/rental/return	174	281	286	288	22	288

1.4 Requests with most errors

Top 5 requests with the most errors, along with the most frequently occurring errors for each request.

Request	Total error count	Error 1	Error 2	Other errors
POST Borrow a Book http://localhost:4000/api/rental/borrow	2	400 Bad Request (2)	-	0
POST Return a Book http://localhost:4000/api/rental/return	2	400 Bad Request (2)	-	0
POST Create Reservation http://localhost:4000/api/reservations	2	400 Bad Request (2)	-	0

2. Metrics for each request

The requests are shown in the order they were sent by virtual users.

Request	Total requests	Requests/s	Min (ms)	Avg (ms)	90th (ms)	Max (ms)	Error %
GET Fetch Popular http://localhost:4000/api/books/popular	34	0.50	27	40	54	71	0
GET Fetch Categories http://localhost:4000/api/books/categories	34	0.50	779	1,685	2,982	4,254	0
GET Search Books http://localhost:4000/api/books/search?query=Great	25	0.37	711	1,486	2,301	3,241	0
GET Fetch Book by ISBN13 http://localhost:4000/api/books/book/{{isbn13}}	25	0.37	20	27	37	50	0
POST Get Book Recommendations http://localhost:4000/api/recommend/recommend	25	0.37	2,141	6,655	10,700	11,675	0
POST Borrow a Book http://localhost:4000/api/rental/borrow	25	0.37	21	190	283	368	8

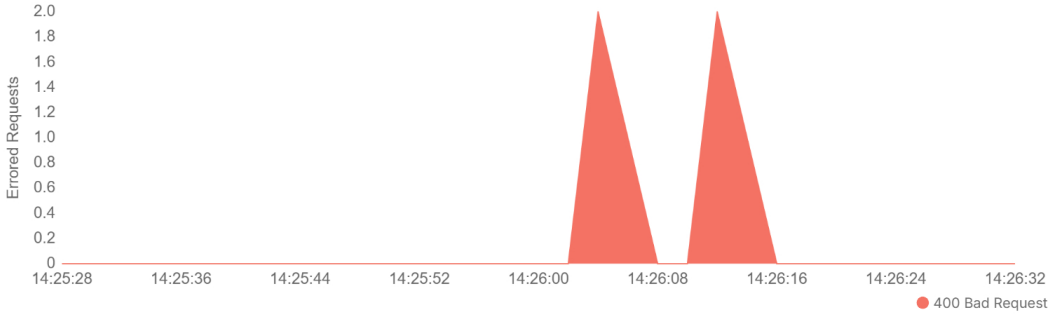
POST Return a Book http://localhost:4000/api/rental/return	25	0.37	22	174	281	288	8
POST Create Reservation http://localhost:4000/api/reservations	25	0.37	42	167	247	270	8
GET Check Desk Availability http://localhost:4000/api/reservations/availability?library=Newton Library&startTime=2025-04-24T10:00:00Z&duration=2	25	0.37	20	27	38	48	0
DELETE Cancel Reservation http://localhost:4000/api/reservations/{{reservationId}}	25	0.37	77	102	161	171	0
GET Fetch User's Rented Books http://localhost:4000/api/rental/books	25	0.37	39	54	74	92	0
GET Get User Profile http://localhost:4000/api/user/profile	25	0.37	67	102	133	324	0

GET Get Total Users - Admin http://localhost:4000/api/admin/users/count	24	0.35	40	55	75	86	0
GET Get Reservations Per Library - Admin http://localhost:4000/api/admin/reservations/count-per-library	24	0.35	40	56	80	106	0
GET Fetch Membership Requests - Admin http://localhost:4000/api/admin/membership-requests?status=pending	24	0.35	57	82	126	146	0

3. Errors

3.1 Error distribution over time

Top 5 error classes observed during the test duration.



3.2 Error distribution for requests

Errored requests grouped by error class, along with the error count for each class.

Error class	Total counts
400 Bad Request	6
POST Borrow a Book	2
POST Return a Book	2
POST Create Reservation	2

Testing API performance on Postman

Postman enables you to simulate user traffic and observe how your API behaves under load. It also helps you identify any issues or bottlenecks that affect performance.

Learn more about testing API performance.

3.4.3 Manual Testing

Manual tests were performed through the User Interface (UI) to validate end-to-end functionality, focusing on user workflows such as login, borrowing, returning, reservations, and membership upgrades. The tests were executed using a Bronze user and an admin user depending on test case. The test cases are summarized in Table 3.4.

Test Case Type	**Description**	**Test Steps**	**Expected Result**
Functionality	Login with valid creds	Open login, enter creds, submit	User logs in, sees dashboard
Functionality	Borrowed book in rentals	As a user borrow a book	Appears on My Rentals, quantity down by 1
Functionality	Returned book removed	As a user return a book	Book removed from My Rentals
Functionality	Book desk if available	As a user book a free desk	Desk booked, listed on Profile page
Functionality	Membership upgrade req	As a user request upgrade	Request visible to admin
Functionality	Admin can see upgrade request from users	Admin approves user request	User upgraded to Silver
Security	Block unauthorized admin	As a user access admin page from router /admin/reservations	User redirected to user end main page.
Usability	Navigation links work	As a user click nav links	Links navigate correctly
Usability	Clear error messages	As a user book reserved desk	"Desk booked" error shown

Table 3.4: Revised Manual Test Plan for MyLibrary Application

3.4.4 WebSocket Testing with Network Tools

Real-time updates were tested using Chrome DevTools [64] to monitor WebSocket events [40], ensuring the application correctly handles live updates via Socket.IO [60]. The tests were conducted with the frontend running on `http://localhost:3000` and the backend on `http://localhost:4000`. Two key scenarios were tested:

- **Book Quantity Update**:
  - **Steps**: 1. Open the frontend in two browser tabs (Tab A and Tab B) as a user. 2. In Tab A, view the main page for the book. 3. In Tab B, borrow and return a book. 4. In Tab A, monitor WebSocket messages in Chrome DevTools (Network tab, WS filter).
  - **Result**: A `quantityUpdate` event was received with data `{ isbn13: 9780941807555, quantity: 0 }`, and the UI updated the quantity in real-time.]

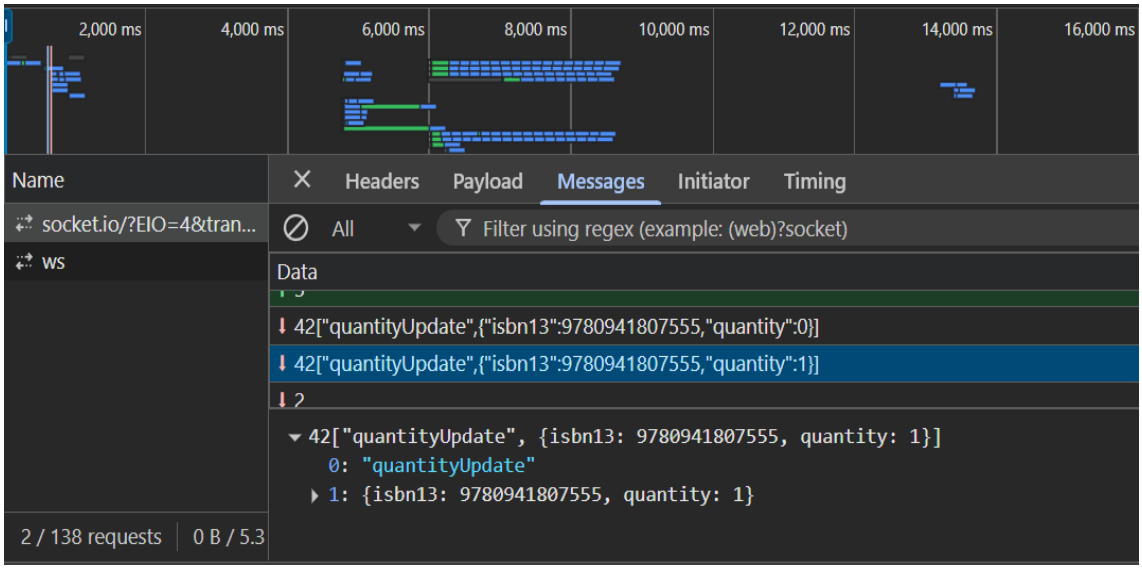


Figure 3.18: WebSocket Events in Chrome DevTools for Book Quantity Change

- **Reservation Update**:
  - **Steps**: 1. Log in as a user. 2. Open two tabs. 2. In Tab A navigate to profile section for reservations dashboard, and Open Chrome DevTools (Network tab, WS filter). 3. In tab B navigate to the reservations page and create a reservation 4. In Tab A Monitor WebSocket messages for a `reservationsUpdate` event.
  - **Result**: A `reservationsUpdate` event was received, and the UI displayed the new reservation.

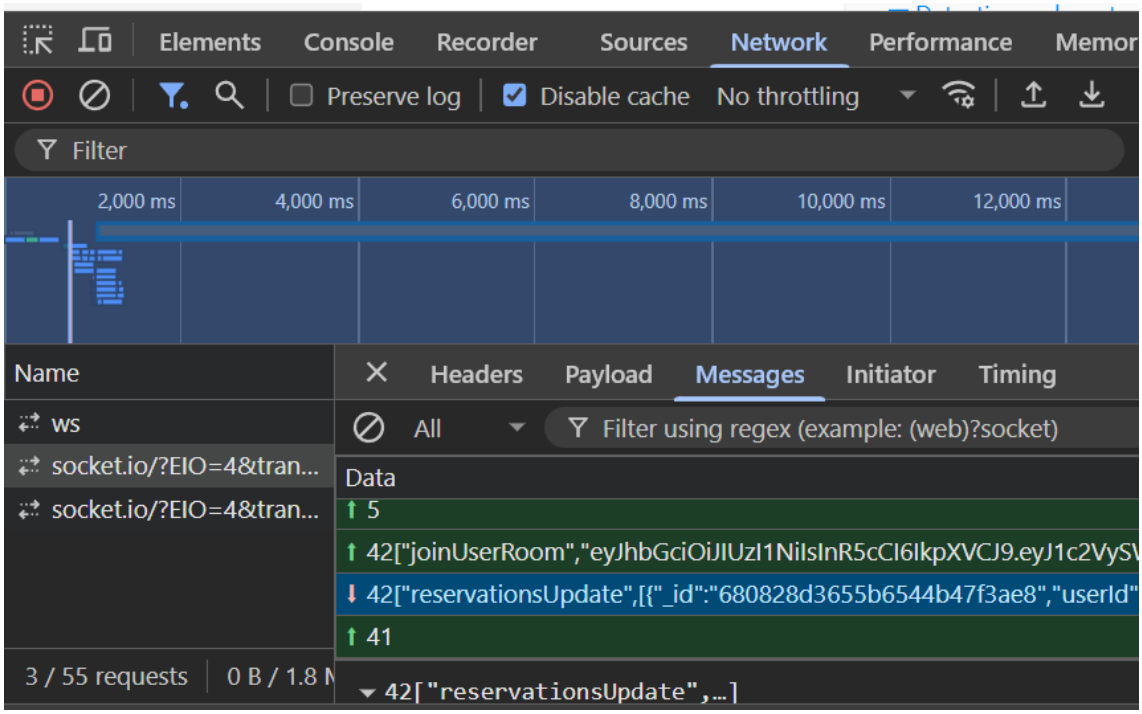


Figure 3.19: WebSocket Events in Chrome DevTools for Reservations

Chapter 4

Conclusion

The `Kitabkhana` project successfully developed a library management system that addressed the predefined challenges. The application optimized book rentals and desk reservations through automated processes and conflict-free scheduling, ensuring efficient resource allocation (see Sections 3.2.4 and 3.2.7). The recommendation system enhanced user engagement by offering personalized book suggestions using content-based filtering with cosine similarity, aligning with individual preferences (Section 3.2.5). Secure access was enforced via role-based access control (RBAC), effectively managing admin and user roles across membership levels within a shared environment (Section 3.2.2) [3].

The system demonstrated reliability through diverse testing techniques and leveraged a scalable architecture, real-time communication mechanisms, and a modular design. However, the current shortcomings need to be addressed before real-world deployment (see Section 3.3).

In a conclusion, the development of `Kitabkhana` contributed to a broader understanding of how software engineering, machine learning, and secure systems design can be applied to overcome library management challenges. The project not only achieved its initial objectives but also established a solid foundation for future advancements, contributing understanding into building efficient, user-focused library systems that balance functionality, security, and scalability.

Bibliography

[1] IBM. *Modular Design in Software Development*. Accessed: April 2025. 2025. URL: https://www.ibm.com/think/topics/modular-design.

[2] IBM. *What is Content-Based Filtering?* Accessed: April 2025. 2025. URL: https://www.ibm.com/think/topics/content-based-filtering.

[3] NIST. *Role-Based Access Control*. Accessed: April 2025. 2025. URL: https://csrc.nist.gov/projects/role-based-access-control.

[4] IBM. *What are REST APIs?* Accessed: April 2025. 2025. URL: https://www.ibm.com/think/topics/rest-apis.

[5] Microsoft. *Windows 10 System Requirements*. Accessed: April 2025. 2025. URL: https://www.microsoft.com/en-us/windows/windows-10-specifications.

[6] Apple. *macOS System Requirements*. Accessed: April 2025. 2025. URL: https://support.apple.com/en-us/HT211683.

[7] Canonical. *Ubuntu System Requirements*. Accessed: April 2025. 2025. URL: https://ubuntu.com/download/desktop.

[8] Intel. *Intel Core i5 Processors*. Accessed: April 2025. 2025. URL: https://www.intel.com/content/www/us/en/products/processors/core/i5-processors.html.

[9] Wikipedia. *Random Access Memory (RAM)*. Accessed: April 2025. 2025. URL: https://en.wikipedia.org/wiki/Random-access_memory.

[10] Node.js. *Node.js Introduction*. Accessed: April 2025. 2025. URL: https://nodejs.org/en/learn/getting-started/introduction-to-nodejs.

[11] npm. *npm Documentation*. Accessed: April 2025. 2025. URL: https://docs.npmjs.com/.

[12] Python Software Foundation. *Python Documentation*. Accessed: April 2025. 2025. URL: https://docs.python.org/3/using/index.html.

[13] MongoDB. *MongoDB Documentation*. Accessed: April 2025. 2025. URL: https://www.mongodb.com/docs/.

[14] Git. *Git Documentation*. Accessed: April 2025. 2025. URL: https://git-scm.com/docs/gittutorial.

[15] Visual Studio Code. *Visual Studio Code Guide*. Accessed: April 2025. 2025. URL: https://code.visualstudio.com/docs/getstarted/getting-started.

[16] Microsoft. *Command Prompt Documentation*. Accessed: April 2025. 2025. URL: https://learn.microsoft.com/en-us/windows-server/administration/windows-commands/cmd.

[17] Apple. *Terminal User Guide for macOS*. Accessed: April 2025. 2025. URL: https://support.apple.com/guide/terminal/welcome/mac.

[18] Git. *Git Bash Documentation*. Accessed: April 2025. 2025. URL: https://git-scm.com/docs/git-bash.

[19] Google. *Google Chrome Browser*. Accessed: April 2025. 2025. URL: https://developer.chrome.com/docs.

[20] Mozilla. *Firefox*. Accessed: April 2025. 2025. URL: https://developer.mozilla.org/en-US/docs/Mozilla/Firefox.

[21] Microsoft. *Microsoft Edge*. Accessed: April 2025. 2025. URL: https://learn.microsoft.com/en-us/microsoft-edge/.

[22] GitLab. *GitLab User Documentation*. Accessed: April 2025. 2025. URL: https://docs.gitlab.com/ee/user/.

[23] Wikipedia. *ZIP*. Accessed: April 2025. 2025. URL: https://en.wikipedia.org/wiki/ZIP_(file_format).

[24] Node.js. *Working with Environment Variables in Node.js*. Accessed: April 2025. 2025. URL: https://habtesoft.medium.com/using-env-files-in-node-js-a-complete-guide-c9e80619c1a7.

[25] JWT. *Introduction to JSON Web Tokens*. Accessed: April 2025. 2025. URL: https://www.rfc-editor.org/rfc/rfc7519.

[26] auth0. *Adding Salt to Hashing*. Accessed: April 2025. 2025. URL: https://auth0.com/blog/adding-salt-to-hashing-a-better-way-to-store-passwords/.

[27] Cloudflare. *What is a computer port*. Accessed: April 2025. 2025. URL: https://www.cloudflare.com/en-in/learning/network-layer/what-is-a-computer-port/.

[28] Python Software Foundation. *Virtual Environments and Packages*. Accessed: April 2025. 2025. URL: https://docs.python.org/3/tutorial/venv.html.

[29] Flask. *Flask Documentation*. Accessed: April 2025. 2025. URL: https://flask.palletsprojects.com/en/3.0.x/.

[30] Microservices.io. *Microservices Architecture*. Accessed: April 2025. 2025. URL: https://microservices.io/.

[31] OWASP. *Authentication Cheat Sheet*. Accessed: April 2025. 2025. URL: https://cheatsheetseries.owasp.org/cheatsheets/Authentication_Cheat_Sheet.html.

[32] WHATWG. *HTML Standard: Email Input Type*. Accessed: April 2025. 2025. URL: https://html.spec.whatwg.org/multipage/input.html#email-state-(type=email).

[33] NPMJS. *joi-password-complexity*. Accessed: April 2025. 2023. URL: https://www.npmjs.com/package/joi-password-complexity.

[34] React Toastify. *React Toastify Documentation*. Accessed: April 2025. 2025. URL: https://fkhadra.github.io/react-toastify/introduction.

[35] Mailchimp. *Website Navigation Guide: Basics and Best Practices*. Accessed: April 2025. 2025. URL: https://mailchimp.com/resources/best-practices-for-website-navigation/.

[36] Halo Lab. *Pagination Design Tips and Examples*. Accessed: April 2025. 2025. URL: https://www.halo-lab.com/blog/pagination-design-tips-and-examples.

[37] Nielsen Norman Group. *Modal Windows in UX Design*. Accessed: February 2025. 2017. URL: https://www.nngroup.com/articles/modal-nonmodal-dialog/.

[38] Nvidia. *Recommendation System*. Accessed: April 2025. 2025. URL: https://www.nvidia.com/en-eu/glossary/recommendation-system/.

[39] Hossein Sharafi. *Designing the user experience of a rating system*. Accessed: April 2025. 2022. URL: https://uxdesign.cc/designing-the-user-experience-of-a-rating-system-2c6a4d33bb11.

[40] WebSocket.org. *What is WebSocket?* Accessed: April 2025. 2025. URL: https://www.websocket.org/aboutwebsocket.html.

[41] Recharts.js. *Recharts.js Documentation*. Accessed: April 2025. 2025. URL: https://recharts.org/en-US.

[42] Figma. *What is Wireframing?* Accessed: April 2025. 2025. URL: https://www.figma.com/resource-library/what-is-wireframing/.

[43] Towards Data Science. *Feature Vectors in Machine Learning*. Accessed: April 2025. 2025. URL: https://towardsdatascience.com/understanding-feature-vectors-in-machine-learning-123.

[44] Naomy Gomes. *The Cosine Similarity and its Use in Recommendation Systems*. Accessed: April 2025. 2025. URL: https://naomy-gomes.medium.com/the-cosine-similarity-and-its-use-in-recommendation-systems-cb2ebd811ce1.

[45] GeeksforGeeks. *Unified Modeling Language (UML) Introduction*. Accessed: April 2025. 2025. URL: https://www.geeksforgeeks.org/unified-modeling-language-uml-introduction/.

[46] MongoDB. *Introduction to the MERN Stack*. Accessed: April 2025. 2025. URL: https://www.mongodb.com/languages/mern-stack-tutorial.

[47] React.js. *React.js Documentation*. Accessed: April 2025. 2025. URL: https://react.dev/.

[48] WHATWG. *HTML5 Standard*. Accessed: April 2025. 2025. URL: https://html.spec.whatwg.org/.

[49] W3C. *Cascading Style Sheets Level 3 (CSS3)*. Accessed: April 2025. 2025. URL: https://www.w3.org/Style/CSS/.

[50] Css-modules. *Documentation about css-modules*. Accessed: April 2025. 2024. URL: https://github.com/css-modules/css-modules.

[51] ECMA International. *ECMAScript 2020 Language Specification*. Accessed: April 2025. 2025. URL: https://262.ecma-international.org/11.0/.

[52] Facebook. *JavaScript XML extension language*. Accessed: April 2025. 2022. URL: https://facebook.github.io/jsx/.

[53] W3C. *Extensible Markup Language (XML)*. Accessed: April 2025. 2008. URL: https://www.w3.org/TR/xml/.

[54] Express.js. *Express.js Documentation*. Accessed: April 2025. 2025. URL: https://expressjs.com/.

[55] IETF. *Hypertext Transfer Protocol – HTTP/1.1*. Accessed: April 2025. 1999. URL: https://www.rfc-editor.org/rfc/rfc2616.

[56] Axios. *Axios Documentation*. Accessed: April 2025. 2025. URL: https://axios-http.com/docs/intro.

[57] Mozilla. *Cross-Origin Resource Sharing (CORS)*. Accessed: April 2025. 2025. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/CORS.

[58] Portswigger.net. *Cross-origin resource sharing (CORS)*. Accessed: April 2025. 2025. URL: https://portswigger.net/web-security/cors.

[59] NPM. *cors*. Accessed: April 2025. 2025. URL: https://www.npmjs.com/package/cors.

[60] Socket.IO. *Socket.IO Documentation*. Accessed: April 2025. 2025. URL: https://socket.io/docs/v4/.

[61] MongoDB. *What is NoSQL?* Accessed: April 2025. 2025. URL: https://www.mongodb.com/databases/what-is-nosql.

[62] Jest. *Jest Documentation*. Accessed: April 2025. 2025. URL: https://jestjs.io/docs/getting-started.

[63] Postman. *Postman Documentation*. Accessed: April 2025. 2025. URL: https://learning.postman.com/docs/getting-started/introduction/.

[64] Google. *Chrome DevTools Documentation*. Accessed: April 2025. 2025. URL: https://developer.chrome.com/docs/devtools/.

[65] Mongoose. *Mongoose Documentation*. Accessed: April 2025. 2025. URL: https://mongoosejs.com/docs/.

[66] Multer. *Multer Documentation*. Accessed: April 2025. 2025. URL: https://github.com/expressjs/multer.

[67] Abdallah Wagih. *Books Dataset*. Accessed: April 2025. 2025. URL: https://www.kaggle.com/datasets/abdallahwagih/books-dataset.

[68] scikit-learn. *TfidfVectorizer Documentation*. Accessed: April 2025. 2025. URL: https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html.

[69] scikit-learn. *cosine_similarity Documentation*. Accessed: April 2025. 2025. URL: https://scikit-learn.org/stable/modules/generated/sklearn.metrics.pairwise.cosine_similarity.html.

[70] NLTK. *Natural Language Toolkit Documentation*. Accessed: April 2025. 2025. URL: https://www.nltk.org/.

[71] Tim Bray. *JavaScript Object Notation (JSON)*. Accessed: April 2025. 2017. URL: https://datatracker.ietf.org/doc/html/rfc8259.

[72] Google. *Collaborative filtering*. Accessed: April 2025. 2025. URL: https://developers.google.com/machine-learning/recommendation/collaborative/basics.

[73] OWASP. *HttpOnly*. Accessed: April 2025. 2025. URL: https://owasp.org/www-community/HttpOnly.

[74] MongoDB. *ObjectId()*. Accessed: April 2025. 2025. URL: https://www.mongodb.com/docs/manual/reference/method/ObjectId/.

[75] Aakash Das. *Preprocessing algorithm for KMP search (LPS array algorithm)*. Accessed: April 2025. 2021. URL: https://medium.com/@aakashjsr/preprocessing-algorithm-for-kmp-search-lps-array-algorithm-50e35b5bb3cb.

[76] Tobias Candela. *Quick Sort Algorithm: A Quick Overview*. Accessed: April 2025. 2023. URL: https://medium.com/@tobiascandela/quick-sort-algorithm-a-quick-overview-4ad3601ae42a.

[77] Chai.js. *Chai.js Documentation*. Accessed: April 2025. 2025. URL: https://www.chaijs.com/guide/.

List of Figures

List of Tables

List of Codes