Database Project

Denchanadula Rushikesh Chary 220336 drushikesh22@iitk.ac.in
Mohammed Anas 220654 mohdanas22@iitk.ac.in
Sandela Amruta Varshini 220960 ssaiamruta22@iitk.ac.in
Nimmala Rithika Reddy 220716 nrithikar22@iitk.ac.in
Chinthapudi Gowtham Chand 220313 cgchand22@iitk.ac.in

CS315 Final Project

Abstract

This project implements a command-line based Library Management System using a Node.js backend and MySQL database. The system supports functionalities such as OTP-based login, book issue and return tracking, role-based access for users and managers, and fine calculation for overdue returns. Designed with modular routes and secure database interactions, this system models a real-world digital library management solution with a focus on core database principles and access control mechanisms.

1 Motivation and Problem Statement

Traditional library management methods often involve manual tracking, which can lead to inefficiencies, data inconsistency, and lack of scalability. Our aim was to build a database-driven web application that mimics real-world library workflows and introduces features like secure login, book transactions, and role-based access. The project serves as a practical application of the relational database concepts, transaction management, and backend integration taught in the course.

2 Methodology

The design of the Library Management System follows a modular and layered architecture aimed at ensuring clarity, separation of concerns, and ease of development. The project is built using a client-server model, with the backend implemented using `Node.js` and `Express`, and the database built on `MySQL`.

The project is structured to include clear roles for users and managers. Users can register, log in, search for books, and issue or return them. Managers have access to administrative functionalities such as viewing issued books and managing the library records. The main design principles that guided our development are **modularity**, **security**, and **scalability**.

At a high level, we separated the system into three main layers:

- **Frontend (Client Layer)**: Responsible for handling user interactions. It communicates with the backend using HTTP requests. While the frontend is minimal in this project, it sets the foundation for future development with API integration through `Axios`.

- **Backend (Application Layer)**: This is the core of the system, where business logic resides. `Express.js` is used to define routes for handling book-related operations, user authentication, and manager functionality. The backend is modular, with routes and database logic organized into separate files for clarity and maintainability.
- **Database (Data Layer)**: `MySQL` is used to store persistent data. The schema includes tables for users, books, authors, categories, publishers, and issued books. Relations between tables are designed using foreign keys to ensure consistency.

Security is addressed by encrypting passwords using `bcrypt` and verifying user identity through email-based OTPs using `nodemailer`. Sensitive credentials are managed using environment variables via the `dotenv` package.

Overall, the architecture is designed to be extensible and allows for easy addition of new features, like fine tracking, notifications, and analytics. The use of standard frameworks and libraries helps ensure that the project is compatible with common deployment and testing workflows.

3 Implementation and Results

3.1 Backend Overview

The backend is written using `Express.js`. The entry point is `server.js`, which sets up middleware, routes, and database connectivity. Database access is abstracted into a `db/connection.js` file, which uses the `mysql2` package to manage queries and connections.

3.2 Book Routes

The book routes handle various user operations related to books in the library, such as searching, issuing, returning, and requesting books. These operations ensure user fine constraints are respected, book availability is maintained, and request queues are handled properly.

- `GET /search`: Allows users to search for books based on title, author, or category. The result includes book ID, title, available copies, category name, and a comma-separated list of authors.
- `GET /issued-books`: Returns all books currently issued by the user, along with book ID, title, available copies, category, and authors.
- `POST /issue`: Issues a book to a user if:
  - The user has not already issued the book.
  - The user has not already requested the book.
  - The user's total fine is below the defined `FINE_LIMIT`.
  - There are copies of the book available.

  If no copies are available, the user is informed to request the book.
- `POST /request`: Adds a request entry in the `book_request` table for a specific book and user, along with the request timestamp.

- `POST /return_request`: Marks a return request for a book by the user. This allows the system to defer the actual return processing to the manager or a background job. The request is recorded with the user's ID and the book's ID.

3.3 User Routes

The user-facing routes provide a wide range of functionalities from authentication to book interaction features. These routes are essential for allowing users to register, login, manage fines, and interact with books via recommendations, likes, dislikes, and requests. The major user routes include:

- `POST /send-otp`: This route takes an email as input and sends a One-Time Password (OTP) to it using a utility service. It ensures that the email is valid before sending.
- `POST /verify-otp`: This route verifies the OTP received by the user. It expects both the email and OTP and returns whether the OTP is valid or not.
- `POST /register`: This route registers a new user after verifying their email and OTP. It validates required fields, ensures uniqueness of email and mobile number, and stores user details in the database.
- `POST /login`: This route logs in a user by checking if the provided email exists and if the password matches the stored one.
- `GET /fine`: This route fetches all outstanding fines for a specific user. It returns a list of books with corresponding fine details.
- `POST /pay-fine`: This route allows users to pay their fines. Users can either clear all fines or selectively pay for specific fine IDs.
- `POST /recommendations`: This route returns book recommendations using stored procedures. It gives two types of results: collaborative filtering-based and category-based recommendations.
- `POST /like`: This route allows a user to like a book, only if they have issued it before. It invokes a stored procedure that validates this condition.
- `POST /dislike`: This route allows a user to dislike a previously liked book. Like the `/like` route, this also calls a stored procedure to ensure proper checks.
- `GET /requested-books`: This route fetches all books currently requested by a user. It joins multiple tables to return details like title, authors, category, and available copies.
- `POST /cancel-request`: This route enables users to cancel a book request they had previously made. It validates the presence of the request before deletion.

3.4 Manager Routes

- `POST /login`: Authenticates a manager using email and password. On successful login, a session or token can be issued (optional JWT or session-based management).
- `POST /add-manager`: Adds a new manager to the system with email, password, name, and mobile number. Used for expanding administrative access.

- `GET /user`: Searches for a user by user_id to view their profile and borrowing history. Allows the manager to manually inspect user activity.
- `POST /logout`: Logs out the currently authenticated manager. (Token/session invalidation to be implemented as required.)
- `GET /dashboard`: Provides dashboard statistics such as the number of users, books, and currently issued books. Useful for managers to get a quick system overview.
- `POST /add-book`: Allows the manager to add a new book with fields like title, author(s), publication, category, and location (rack number and room number).
- `GET /not-returned`: Lists users who have not returned their books within the allowed timeframe. This route helps in fine calculation and overdue tracking.
- `GET /return-requests`: Displays all book return requests submitted by users. Each request typically includes user ID, book ID, and return timestamp.
- `POST /manage-return`: Approves or rejects a return request. If approved, the book issue record is deleted and inventory is updated. If rejected, the return request remains pending.

3.5 Utility Services

The file `utils/otpService.js` handles OTP generation and sending via email. The OTP is generated as a random 6-digit number and is sent to the user's registered email address for verification purposes. However, the OTP is not stored or verified using a secure session mechanism, which is one of the limitations discussed later.

The email functionality is supported by `utils/mailer.js`, which configures and utilizes the `nodemailer` package to send emails. It sets up a transporter using environment variables for the mail service credentials, allowing OTP and other system-generated emails (like password reset requests or alerts) to be dispatched securely. The modular structure of `mailer.js` allows for reusable and consistent email formatting across different parts of the system.

3.6 Database Schema

The library management system utilizes a relational database designed in MySQL to store and manage data efficiently. The schema is normalized and supports various entities such as books, users, authors, and related activities like issuing, liking, and requesting books. The schema ensures data integrity through foreign keys, constraints, procedures, and triggers.

Tables

- **category**: Stores book categories.
- **publisher**: Stores publishers with associated languages.
- **location**: Maps books to a physical floor and shelf.
- **author**: Stores author information.

- **book**: Stores book details, foreign keys to category, publisher, and location.
- **book_author**: Many-to-many relationship between books and authors.
- **user**: Stores library users including credentials and contact info.
- **book_request**: Tracks book requests by users.
- **book_issue**: Stores issued books with issue and return dates.
- **fine_due**: Tracks fines on late returns, linked to book issues.
- **manager**: Stores manager login and profile information.
- **book_like**: Stores user likes on books. Supports recommendations.
- **book_return**: Stores information about book return requests made by users, including user ID, book ID, and the date of return.

Procedures

- **get_recommendations(uid)**: Recommends books based on user likes and categories.
- **like_book_if_issued(uid, bid)**: Allows liking a book only if previously issued.
- **dislike_book(uid, bid)**: Removes like only if user had liked it.

Triggers

- **check_fine_after_update**: Trigger to calculate and record fines on late return.
- **increment_likes, decrement_likes**: Triggers to update `no_of_likes` in `book`.
- **set_return_date**: Automatically sets the default return date to 14 days after issue.

ER Diagram

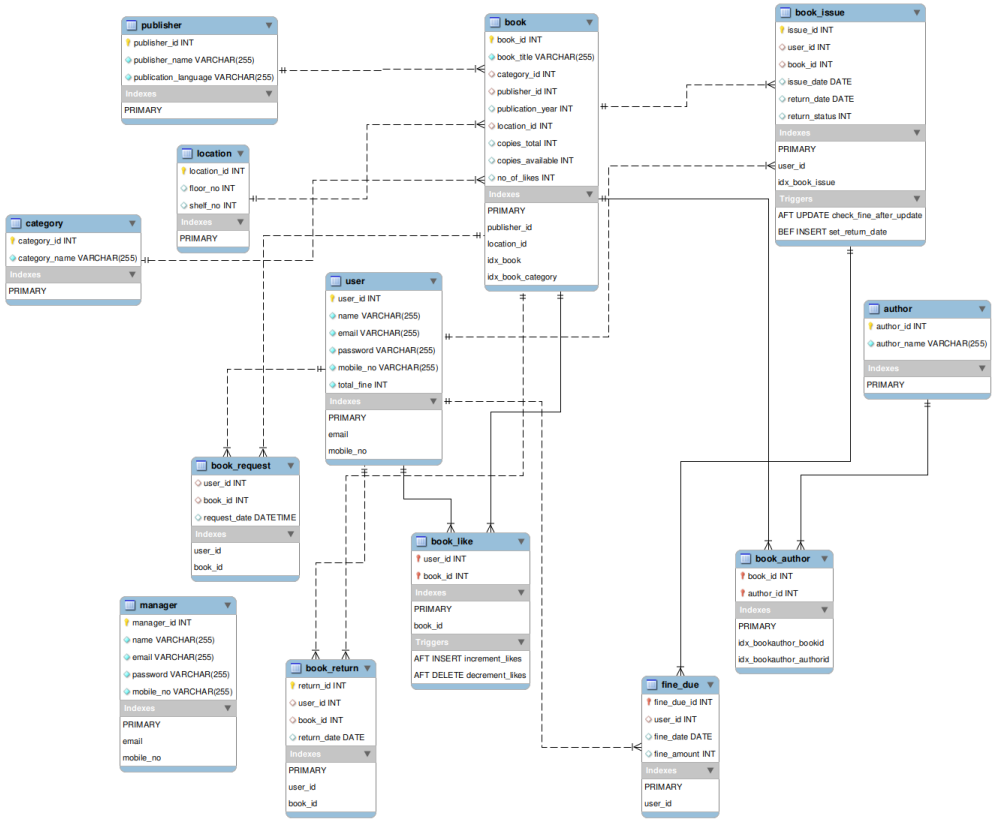

Figure 1: ER Diagram of the MySQL Database

3.7 Results

The backend supports:

- Secure user registration and login.
- OTP-based verification to ensure identity.
- Book search and borrow/return operations with rules around fines.
- Basic manager access to oversee library data.

The system works end-to-end in a local environment, with API calls successfully managing records and enforcing borrowing policies. The modular design makes the backend easy to extend, test, and debug.

Codebase: Available at **https://github.com/RushikeshChary/LibraryManagementSystem**

4 Discussion and Limitations

While functional and well-structured for academic purposes, the system does have several limitations:

- OTPs are not stored persistently — refreshing the server resets all OTPs
- No advanced UI — interactions are done via command-line interface
- No rate-limiting or brute-force prevention for OTP login
- Lacks deployment setup (Docker, CI/CD, etc.)

Future improvements could include:

- Web-based frontend with session management
- Persistent OTP expiration with Redis or SQL TTL
- Integration with cloud-based mail services and deployment on platforms like Heroku or Vercel
- Audit logs and statistics dashboard for managers

5 Contributions

Name	**Contributions**
Rushikesh Chary	Designed the database schema, implemented server functionality and did API testing.
Mohammed Anas	Fixed some bugs, added a notification feature, created an ER diagram, and wrote the report.
Nimmala Rithika Reddy	Improvised client and server functionalities, added few API routes, did testing and fixed most bugs, and updated the database.
Sandela Sai Amruta Varshini	Implemented the client functionalities, integrated API routes, and built the OTP mailing service.
Gowtham Chand	Helped writing report and designing ER diagram