### **1. Introduction**

#### **Purpose**

This document outlines the system design for the Library Management System (LMS). The LMS is designed to automate the functions of a library, including book management, user management, and borrowing and returning process.

#### **Scope**

The system will handle tasks such as:

* Managing books (adding, searching, updating, removing)
* Managing users (registration, authentication, role-based access)
* Managing transactions (book borrowing, returns, fines)

#### **Audience**

* Developers
* Project Managers
* Librarian

### **2. System Overview**

#### **System Description**

The LMS will be a web-based application that allows librarians and members (non-admins, admins) to manage books, borrow books, reserve books, and return books. The system will also track overdue books and apply fines.

#### **System Components**

* **Frontend (UI):** A web-based user interface for users to interact with the system.
* **Backend (API):** A Spring Boot RESTful API to handle business logic and interact with the database.
* **Database:** A relational database for storing data about books, users, transactions, etc.

### **3. Architecture Design**

#### **Architecture Overview**

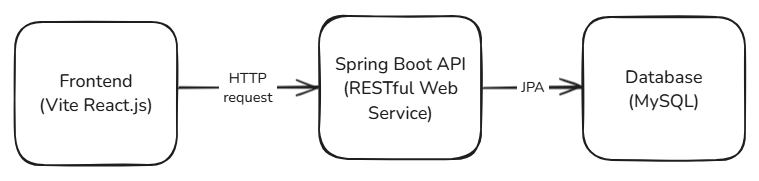
The system will follow a layered architecture consisting of:

1. **Presentation Layer (View):** The user interface (Web application).
2. **Business Logic Layer (Controller):** Spring Boot RESTful API to handle all application logic.
3. **Data Layer (Model):** Relational database (MySQL) to store and manage data.

#### **Tech Stack**

* **Frontend:** Vite React.js
* **Backend:** with Java for building the backend RESTful APIs.
* **Database:** MySQL
* **Development Environment:** Eclipse as the Integrated Development Environment (IDE) for backend development and VS code as the Integrated Development Environment (IDE) for frontend development.

#### **Component Diagram**

**

**Frontend (UI)**:

* **Web Application (Vite React.js)**: The user Interface part of the system that allows users/librarians to interact with the library (browsing books, borrowing books, managing books and account, etc.).

**Backend (API)**:

* **Spring Boot API**: The server-side application that handles business logic, such as managing users, books, and transactions. The backend processes requests from the frontend.

**Database**:

* **MySQL**: Stores all the persistent data, such as user, book, borrowing history, fines, etc.

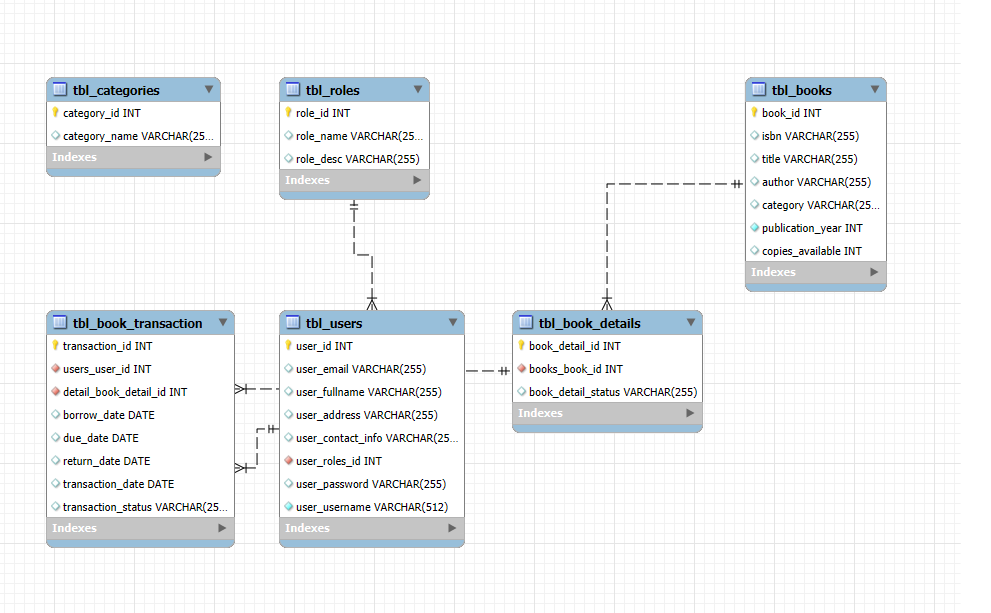
### **4. Database Design**

#### **Database Schema**

The system will use a relational database with the following tables:

1. **tbl\_users**
   * user\_id (Primary Key)
   * user\_fullname
   * user\_email
   * user\_address
   * user\_contact\_info
   * user\_roles\_id
   * user\_password
   * user\_username
2. **tbl\_books**
   * book\_id (Primary Key)
   * isbn
   * title
   * author
   * category
   * publication\_year
   * copies\_available
3. **tbl\_book\_transaction**
   * transaction\_id (Primary Key)
   * users\_user\_id(Foreign Key to tbl\_users)
   * detail\_book\_detail\_id(Foreign Key to tbl\_books)
   * borrow\_date
   * due\_date
   * return\_date
   * transaction\_date
   * transaction\_status
4. **tbl\_categories**
   * category\_id (Primary Key)
   * category\_name
5. **tbl\_roles**
   * role\_id (Primary Key)
   * role\_name
   * role\_desc
6. **tbl\_book\_details**
   * book\_detail\_id (Primary Key)
   * books\_book\_id (Foreign Key to tbl\_books)
   * book\_detail\_status

#### **ERD (Entity Relationship Diagram)**

**

### **5. API Design**

#### **Endpoints**

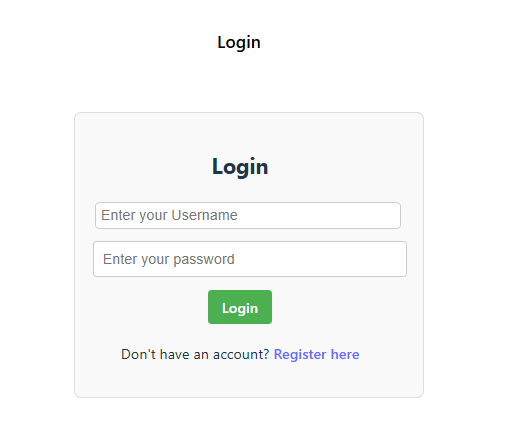
* **User Management**
  + POST /api/users/addUser: User Registration
  + GET /api/users/getAll: List all users
  + GET /api/users/getUsername/{username}: Get username
  + GET /api/users/verifyLogin: Check password
  + GET /api/users/getLoginDetails/{userId}: Get User’s username
* **Book Management**
  + GET /api/books: List all Books
  + POST /api/books: Add a New Book
  + PUT /api/books/update/{id}: Update Book Info
  + DELETE /api/books/{id}: Delete a Book
* **Book Details Management**
  + GET /api/bookdetails/{bookId}: List all BookDetails
  + GET /api/bookdetails/all: List all BookDetails with transaction
  + POST /api/bookdetails: Add a New Book Details
  + PUT /api/bookdetails/update/{id}: Update Book Details Info
  + DELETE /api/bookdetails/{id}: Delete a Book Details
* **Transaction Management**
  + GET /api/transactions/{userId}: View User current transaction
  + POST /api/transaction/borrow: Borrow a Book
  + POST /api/transactions/return: Return a Book
  + GET /api/transactions/borrow-details/{userId}: View User Transaction History
* **Categories**
  + GET /api/categories: Get Book’s categories
* **Roles**
  + GET /api/roles/getAll: Get all roles

### 

### **6. UI Mock-ups**

#### **User Interface Design**

* **Login Page:** A login form for user authentication.

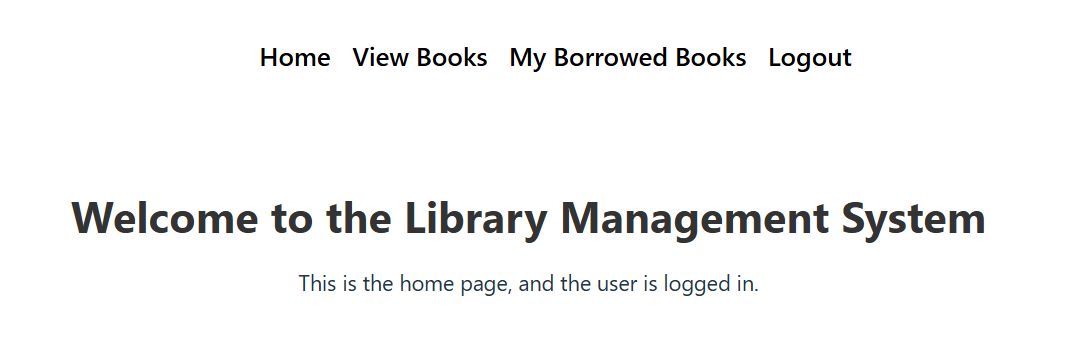
**

* **Librarian Homepage:** A central hub displaying the available books, transactions, etc.

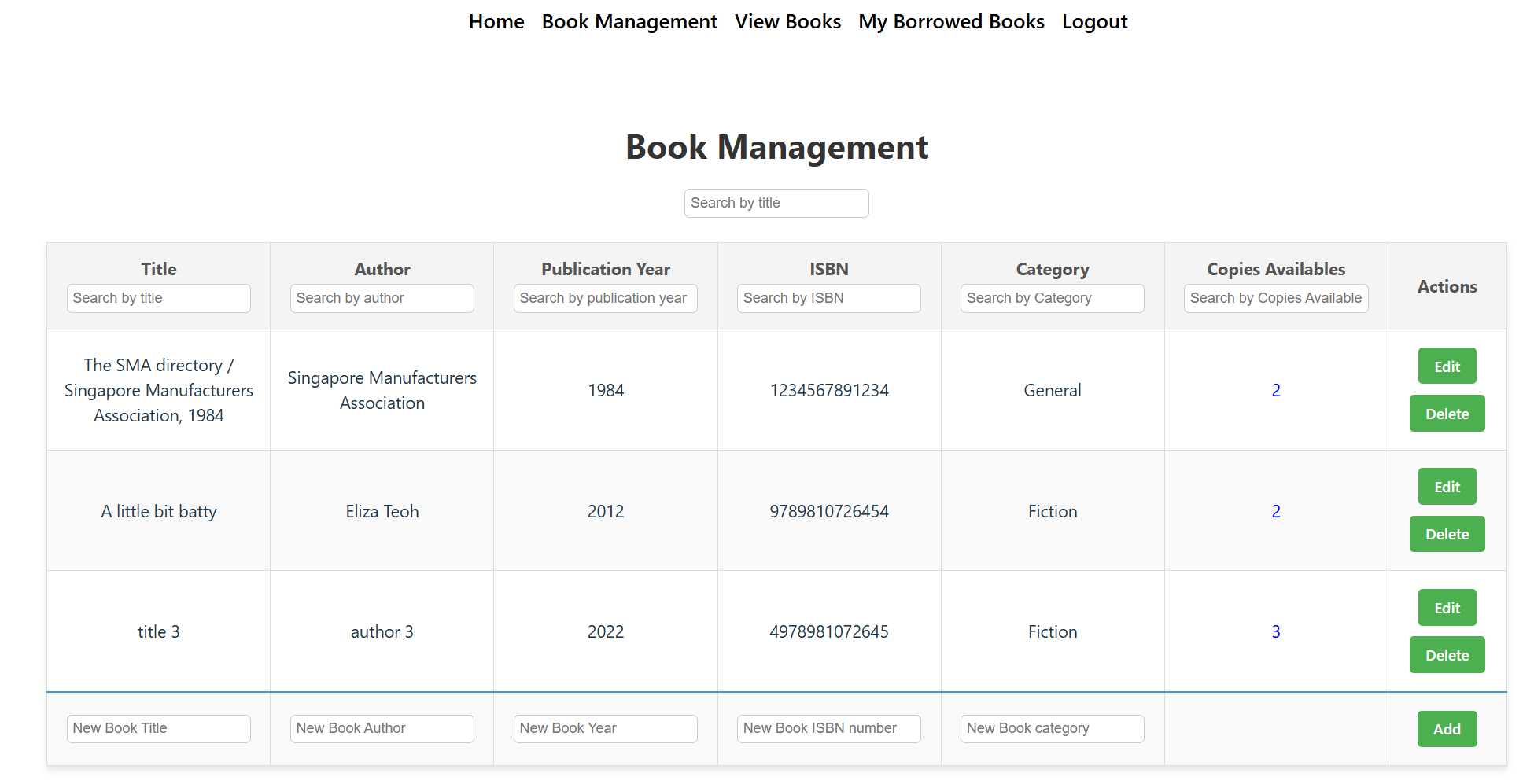
A screenshot of a computer

AI-generated content may be incorrect.

* **Member Homepage:** A central hub displaying the available books, transactions, etc.



* **Book Management:** Displays a list of books with search and filter options.

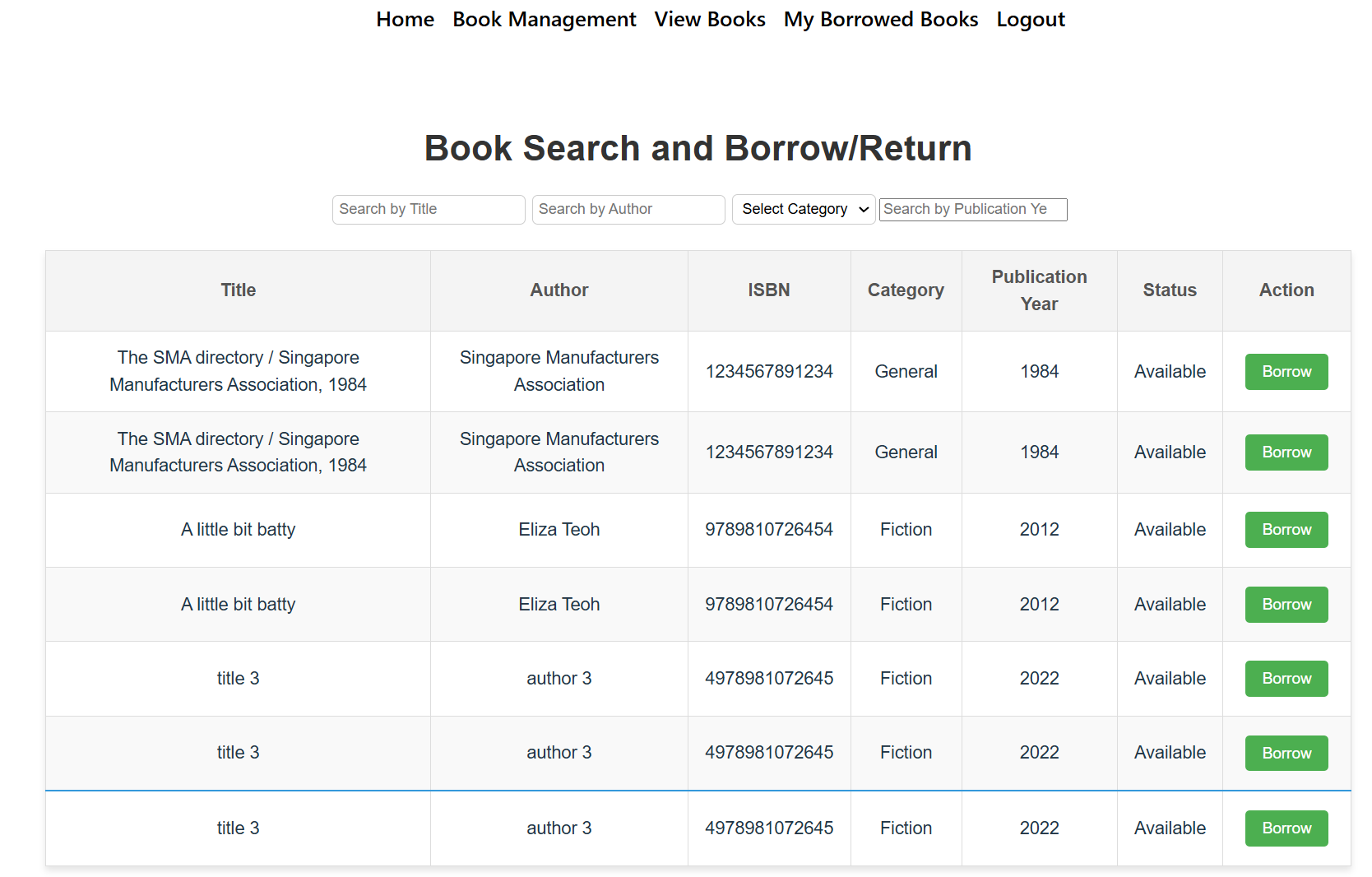


* **User Management:** Displays a list of users for librarians to edit email or role or to remove users.

A screenshot of a website

AI-generated content may be incorrect.

* **View Books:** A page with details about a single book, including availability and reserve option.



* **My Borrowed Books:** A page showing current transactions for the user.

A screenshot of a credit card

AI-generated content may be incorrect.

### **7. Non-Functional Requirements**

#### **Performance**

* API calls should respond within 200ms under normal load.

#### **Scalability**

* The system should be scalable to handle thousands of users and books.

#### **Backup and Recovery**

* Daily backups of the database should be performed, and a disaster recovery plan should be in place.

#### **Availability**

* The system should maintain 99.9% uptime.

### **8. Testing**

#### **Testing Strategy**

* **Unit Testing:** Use frameworks Jest.
* **Integration Testing:** Ensure that API endpoints interact correctly with the database.
* **User Acceptance Testing (UAT):** Verify the system with end-users to ensure it meets their requirements.

### **9. Deployment and Maintenance**

#### **Deployment**

* Use a CI/CD pipeline to automate deployment.
* Host the application on AWS, Heroku, or any other cloud platform.

#### **Maintenance**

* Regular bug fixes and updates will be scheduled.
* Periodic monitoring of the system to identify and resolve issues.