**Library Management System**

### **1. Detailed Structure of the Tables**

The following tables are used in your Library Management System:

#### **1.1. users Table**

* Stores information about users (librarians and members).

| **Column Name** | **Type** | **Description** |
| --- | --- | --- |
| id | Integer | Primary key |
| username | String | Unique username for login |
| password\_hash | String | Encrypted password |
| role | String | Either LIBRARIAN or MEMBER |
| is\_active | Boolean | Indicates if the account is active |

#### **1.2. books Table**

* Stores information about books available in the library.

| **Column Name** | **Type** | **Description** |
| --- | --- | --- |
| id | Integer | Primary key |
| title | String | Title of the book |
| author | String | Author of the book |
| status | String | Either AVAILABLE or BORROWED |
| borrower\_id | Integer | Foreign key referencing users.id |

#### **1.3. history Table**

* Stores the history of books issued and returned.

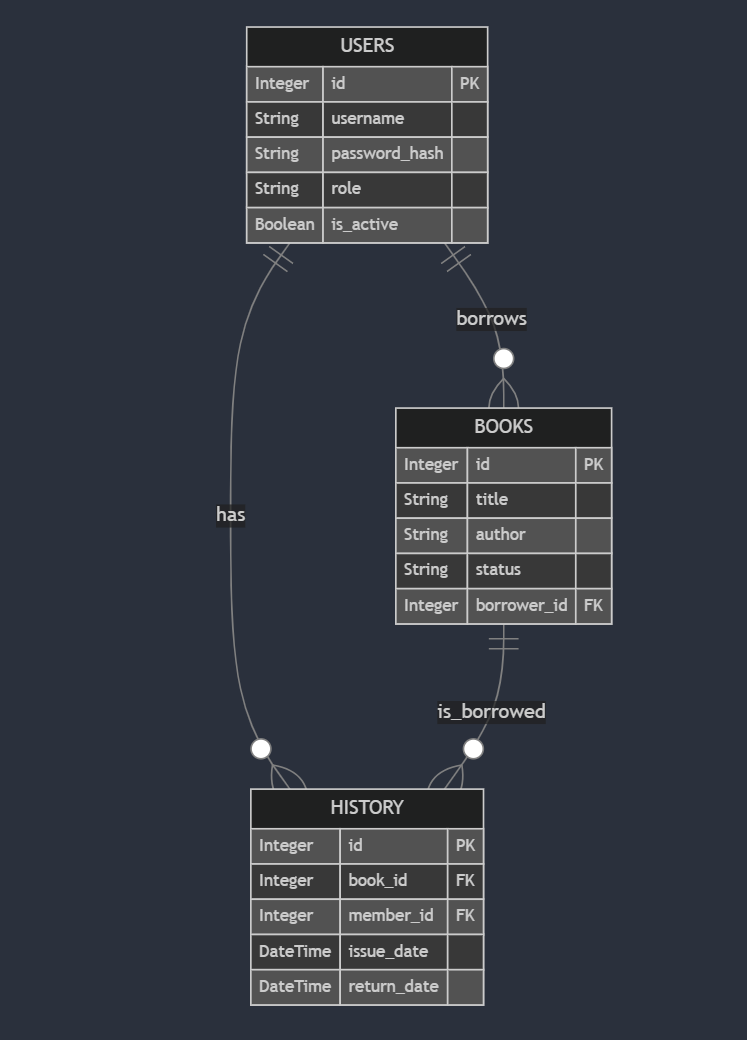
| **Column Name** | **Type** | **Description** |
| --- | --- | --- |
| id | Integer | Primary key |
| book\_id | Integer | Foreign key referencing books.id |
| member\_id | Integer | Foreign key referencing users.id |
| issue\_date | DateTime | The date when the book was issued |
| return\_date | DateTime | The date when the book was returned |

### 

### 

### 

### **2. Database Diagram**

Here is a database diagram showing the relationships between tables:

### **3. API Documentation**

Below is a description of each API endpoint, including requirements, output, and errors.

#### **3.1. Authentication APIs**

* POST /auth/login
  + Description: Authenticates the user and returns a JWT token.

Request Body:  
{

"username": "string",

"password": "string"

}

Response:  
{

"access\_token": "string",

"token\_type": "bearer"

}

* + Errors:
    - 400: Incorrect username or password.
    - 422: Missing or invalid fields in the request body.

#### **3.2. User Management APIs**

* POST /auth/signup
  + Description: Registers a new user.

Request Body:  
{

"username": "string",

"password": "string",

"role": "LIBRARIAN or MEMBER"

}

Response:  
{

"id": "integer",

"username": "string",

"role": "LIBRARIAN or MEMBER",

"is\_active": true

}

* + Errors:
    - 400: User already exists.
    - 422: Missing or invalid fields.
* GET /members (Librarian Only)
  + Description: Fetches a list of all members.
  + Response: List of users with the role MEMBER.

[

{

"id": "integer",

"username": "string",

"is\_active": true

}

]

#### **3.3. Book Management APIs**

* GET /books
  + Description: Fetches a list of all books.

Response:  
[

{

"id": "integer",

"title": "string",

"author": "string",

"status": "AVAILABLE or BORROWED"

}

]

* POST /books (Librarian Only)
  + Description: Adds a new book.

Request Body:  
{

"title": "string",

"author": "string"

}

Response:  
{

"id": "integer",

"title": "string",

"author": "string",

"status": "AVAILABLE"

}

* PUT /books/{book\_id} (Librarian Only)
  + Description: Updates an existing book.

Request Body:  
{

"title": "string",

"author": "string",

"status": "AVAILABLE or BORROWED"

}

* DELETE /books/{book\_id} (Librarian Only)
  + Description: Deletes a book.

### **4. Frontend and flow of Software**

The frontend (HTML + JavaScript) interacts with the backend using the following flow:

#### User Authentication:

* + The user logs in via the login.html page, which sends credentials to /auth/login. On successful authentication, the JWT token is stored in localStorage and used in subsequent API calls.

#### Dashboard Access:

* + Upon successful login, users are redirected to either the librarian\_dashboard.html or member\_dashboard.html depending on their role.

#### Book Management:

* + Librarians can add, edit, delete, and view books.
  + Members can only view available books and borrow/return them.

#### Member Management:

* + Librarians can add, edit, and delete members. Members can view their own borrowing history.

#### Flow of the Software

#### **1. User Authentication Flow (Login/Signup)**

* When the user navigates to the **login** page (login.html), they provide their **username** and **password**.
* The frontend sends a **POST request** to the backend (/auth/login) with the login credentials.
* If authentication is successful, a **JWT token** is returned by the backend and stored in the browser’s localStorage.
* The user is then redirected to the appropriate dashboard based on their role (librarian\_dashboard.html for Librarians or member\_dashboard.html for Members).

#### **2. Librarian Workflow**

1. **Dashboard Overview**:
   * After login, Librarians are taken to the **Librarian Dashboard** (librarian\_dashboard.html), where they can view the list of books and members.
2. **Book Management**:
   * **View Books**: All books are fetched via a **GET request** to /books.
   * **Add Book**: Librarians can add books using a form, which sends a **POST request** to /books.
   * **Edit Book**: Librarians can edit book details by sending a **PUT request** to /books/{id}.
   * **Delete Book**: Books can be deleted via a **DELETE request** to /books/{id}.
3. **Member Management**:
   * **View Members**: Librarians can view a list of members by sending a **GET request** to /members.
   * **Add Member**: Librarians can add new members by sending a **POST request** to /members.
   * **Edit Member**: Librarians can update member details by sending a **PUT request** to /members/{id}.
   * **Delete Member**: Members can be removed by sending a **DELETE request** to /members/{id}.
4. **View Borrowing History**:
   * Librarians can view the borrowing history of all members via a **GET request** to /history.

#### **3. Member Workflow**

1. **Dashboard Overview**:
   * After login, members are taken to the **Member Dashboard** (member\_dashboard.html), where they can view the list of available books and manage their borrowing history.
2. **Book Borrowing/Returning**:
   * **View Books**: Members can view available books via a **GET request** to /books.
   * **Borrow Book**: Members can borrow books by sending a **POST request** to /borrow/{book\_id}.
   * **Return Book**: Members can return borrowed books by sending a **POST request** to /return/{book\_id}.
3. **View Borrowing History**:
   * Members can view their borrowing history via a **GET request** to /history.

### **5. Hosting Instructions**

#### **Frontend Hosting (Vercel):**

1. Host the frontend files (index.html, app.js, etc.) on Vercel.
2. Ensure the API\_BASE\_URL in app.js points to the deployed backend URL(currently local).

#### **Backend Hosting (Render):**

1. Deploy the FastAPI backend to Render.
2. Make sure the backend is configured to accept CORS requests from the frontend domain.
3. Due to my office hours, I couldn’t get enough time to host properly. You can still check my Github repository for my work. Please consider.

#### **Database Setup:**

1. The project uses SQLite, which is file-based. Ensure that the DATABASE\_URL is configured in Heroku to persist the database, or consider switching to a cloud database like PostgreSQL.

### **6. Design Approach and Choices**

#### **Backend (FastAPI):**

* FastAPI was chosen for its speed and ease of use. It's asynchronous, which makes handling multiple API requests efficient.
* Pydantic models are used to validate incoming request data and generate error responses when data is invalid.
* JWT authentication is implemented for secure user login. Tokens are validated for every API call that requires authentication.
* SQLAlchemy is used for ORM to interact with the SQLite database. The structure is simple and can be extended to PostgreSQL or other databases in the future.

#### **Frontend (Vanilla JS):**

* The frontend uses vanilla JavaScript and fetch API to communicate with the backend. This keeps the project lightweight without the need for external libraries like React.
* The frontend dynamically updates the DOM based on the API responses for better user interaction (CRUD operations on books and members).

#### **Security:**

* All protected API routes require JWT authentication, and role-based access control (RBAC) is implemented to ensure that only librarians can perform certain actions (like adding books).