maids.cc
+ nannies + visa services

Library Management System API Documentation

Task Overview

Ali Asswad

1-How to Run the Application

1-1. Prerequisites

Ensure you have the following installed on your system:

- Java 17+
- Maven 3+
- MySQL Server
- An API testing tool (e.g., Postman)

2-2. Steps to Run the Application

1. **Clone the Repository**
2. **Configure the Database**
   - Ensure MySQL is running.
   - Create a database named library_management.
   - Update database credentials in application.properties or application.yml:

     Properties
3. **Run the Application**
   - Open the project in your IDE (e.g., IntelliJ, VS Code, Eclipse).
   - Locate the TaskApplication class.
   - Run the application using:

     - mvn spring-boot:run

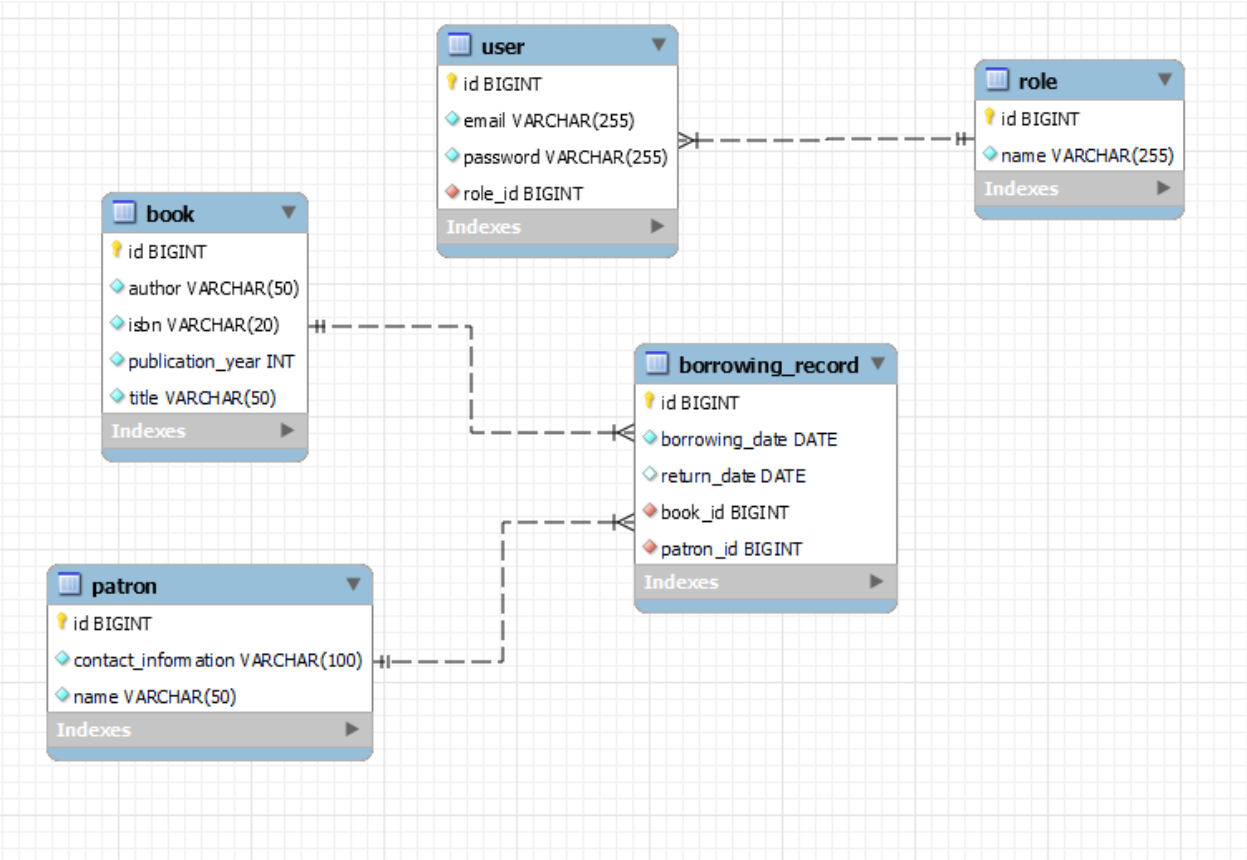
   - Alternatively, run it from your IDE's Run/Debug configurations.
4. **Verify API Endpoints**
   - Once the application starts, the API will be available at:
     - http://localhost: 8085

2.Database Structure

Database Type: MySQL

Database Name: library_management

Tables & Relationships :



User Roles & Permissions

Predefined Roles:

These roles are pre-seeded in the database upon the first application startup.

1. **ROLE_ADMIN**: Full access to manage books, patrons, , borrowing process and create users.
2. **ROLE_LIBRARIAN**: Can manage books and borrowing records but has limited administrative privileges.
3. **ROLE_PATRON**: Can view available books and borrow/return books.

These roles are seeded **only once**, when the application is first launched.

3.Authentication & Authorization

3-1. Authentication Mechanism

This API uses **JWT-based Authentication & Authorization** to secure endpoints. When a user logs in successfully, they receive a **JWT token**, which contains:

- **User ID**
- **User Role**

This token must be included in the **Authorization header** of all subsequent requests as follows:

Authorization: Bearer <JWT_TOKEN>

3-2. User Registration & Role Assignment

- Only **ROLE_ADMIN** users can create new users and assign roles.
- Initially, an **admin user** must be manually created in the database.

  -Run the following SQL command to create an admin user:

  ```
  INSERT INTO User (email, password, role_id)
  VALUES ('admin@example.com', 'securePassword123',
      (SELECT id FROM Role WHERE name = 'ROLE_ADMIN'));
  ```

- Alternatively, you can **temporarily allow all users to sign up**, create an admin user, and then restrict sign-ups again.

3-3.API Endpoints & Role-Based Access Control

- **Authentication Endpoints**

Endpoint	HTTP Method	Access Level	Description
`/api/auth/login`	`POST`	**Permit All**	Allows users to authenticate and receive a JWT token.
`/api/auth/signup`	`POST`	**ROLE_ADMIN Only**	Allows an admin to create new users and assign roles.

- **Book Management Endpoints**

Endpoint	HTTP Method	Access Level	Description
`/api/books`	`GET`	**ADMIN, LIBRARIAN, PATRON**	Retrieves a list of all books.
`/api/books/{id}`	`GET`	**ADMIN, LIBRARIAN, PATRON**	Retrieves details of a specific book.
`/api/books`	`POST`	**ADMIN, LIBRARIAN**	Adds a new book to the system.
`/api/books/{id}`	`PUT`	**ADMIN, LIBRARIAN**	Updates an existing book's details.
`/api/books/{id}`	`DELETE`	**ADMIN, LIBRARIAN**	Deletes a book from the system.

- **Patron Management Endpoints**

Endpoint	HTTP Method	Access Level	Description
`/api/patrons`	`GET`	**ADMIN, LIBRARIAN**	Retrieves a list of all patrons.
`/api/patrons/{id}`	`GET`	**ADMIN, LIBRARIAN**	Retrieves details of a specific patron.
`/api/patrons`	`POST`	**ADMIN, LIBRARIAN**	Adds a new patron to the system.
`/api/patrons/{id}`	`PUT`	**ADMIN, LIBRARIAN**	Updates an existing patron's details.
`/api/patrons/{id}`	`DELETE`	**ADMIN, LIBRARIAN**	Removes a patron from the system.

Borrowing Management Endpoints

Endpoint	HTTP Method	Access Level	Description
`/api/borrow`	`GET`	**ADMIN, LIBRARIAN, PATRON**	Retrieves all borrowing records.
`/api/borrow/{bookId}/patron/{patronId}`	`POST`	**ADMIN, LIBRARIAN, PATRON**	Allows a patron to borrow a book.
`/api/return/{bookId}/patron/{patronId}`	`PUT`	**ADMIN, LIBRARIAN, PATRON**	Records the return of a borrowed book.

Any other request requires authentication and will return 401 Unauthorized if a valid JWT token is not provided.

4. API Documentation:

1. Authentication Controller

Endpoint	Method	Headers	Request Body	Response
`/api/auth/login`	POST	`Content-Type: application/json`	`{ "email": "user@example.com", "password": "securePassword123" }`	`{ "token": "eyJhbGciOiJIUzI1NiIsInR..." }`
`/api/auth/signup`	POST	`Content-Type: multipart/form-data`	Form Data: - **email:** `user@example.com` - **password:** `securePassword123` - **role:** `ROLE_LIBRARIAN`	`200 OK` **Note:** Password is hashed and securely stored in the database. The response does not return sensitive information such as the password for security reasons.

Reason for multipart/form-data in SignUp Request

The multipart/form-data content type is commonly used when submitting data that might include files, though in this case, it's chosen to accommodate potential future extensions where user profile pictures or other attachments might be added to the registration process.

Requests :

{ "email": "user@example.com", "password": "securePassword123" }

- email: user@example.com
- password: securePassword123
- role: ROLE_LIBRARIAN

2. Book Controller

Endpoint	Method	Headers	Request Body	Response
`/api/books`	GET	`Authorization: Bearer {token}`	None	`200 OK` List of `BookDto` objects.
`/api/books/page`	GET	`Authorization: Bearer {token}`	`{ "page": 1, "size": 10, "sort": "title", "sortDirection": "asc" }`	`200 OK` Paginated response of `BookDto` objects.
`/api/books/{id}`	GET	`Authorization: Bearer {token}`	None	`200 OK` Book details as `BookDto` object.
`/api/books`	POST	`Authorization: Bearer {token}`	`BookCreateDto` object	`201 Created` Book created as `BookDto`.
`/api/books/{id}`	PUT	`Authorization: Bearer {token}`	`BookUpdateDto` object	`200 OK` Updated book as `BookDto`.
`/api/books/{id}`	DELETE	`Authorization: Bearer {token}`	None	`204 No Content` Book successfully deleted.

Requests :

1. BookCreateDto (for POST /api/books)

{ "title": "The Great Gatsby", "author": "F. Scott Fitzgerald", "publicationYear": 1925, "isbn": "9780743273565" }

2. BookUpdateDto (for PUT /api/books/{id})

{ "id": 1, "title": "The Great Gatsby", "author": "F. Scott Fitzgerald", "publicationYear": 1925, "isbn": "9780743273565" }

4. PageRequestDto (Request body for GET /api/books/page)

{ "page": 1, "size": 10, "sort": "title", "sortDirection": "asc" }

4. Borrowing Controller

Endpoint	Method	Headers	Request Body	Response
`/api/borrow/{bookId}/patron/{patronId}`	**POST**	`Authorization: Bearer {token}`	None	`201 Created` New borrowing record as `BorrowingRecordDto`.
`/api/borrow/{id}`	**GET**	`Authorization: Bearer {token}`	None	`200 OK` Borrowing record as `BorrowingRecordDto`.
`/api/return/{bookId}/patron/{patronId}`	**PUT**	`Authorization: Bearer {token}`	None	`200 OK` Updated borrowing record as `BorrowingRecordDto`.
`/api/borrow`	**GET**	`Authorization: Bearer {token}`	None	`200 OK` List of `BorrowingRecordDto` objects.

4. Patron Controller

Endpoint	Method	Headers	Request Body	Response
`/api/patrons`	**GET**	`Authorization: Bearer {token}`	None	`200 OK` List of `PatronDto` objects.
`/api/patrons/page`	**GET**	`Authorization: Bearer {token}`	`PageRequestDto`	`200 OK` Paginated `PatronDto` list as `PageResponse<PatronDto>`.
`/api/patrons/{id}`	**GET**	`Authorization: Bearer {token}`	None	`200 OK` Patron details as `PatronDto`.
`/api/patrons`	**POST**	`Authorization: Bearer {token}`	`PatronCreateDto`	`201 Created` New `PatronDto` object.
`/api/patrons/{id}`	**PUT**	`Authorization: Bearer {token}`	`PatronUpdateDto`	`200 OK` Updated `PatronDto` object.
`/api/patrons/{id}`	**DELETE**	`Authorization: Bearer {token}`	None	`204 No Content` Patron deleted.

Request :

1. POST /api/patrons (Create a New Patron)

```
{ "name": "Johnathan Doe", "contactInformation": "098-765-4321" }
```

2. PUT /api/patrons/{id} (Update a Patron)

```
{ "id": 1, "name": "Johnathan Doe", "contactInformation": "098-765-4321" }
```

Recommendation: Cascade Delete vs. Soft Delete

In this project, we used Cascade Delete to align with the requirement of automatically removing associated records when a main record (like a book or patron) is deleted, ensuring data integrity. However, Soft Delete can be a better approach for scenarios where data retention, auditing, and recovery are important. By marking records as deleted instead of physically removing them, we can preserve historical data, track deletions for auditing purposes, and easily recover accidentally deleted records, which is beneficial for analytics and ensuring data integrity over time.