**Library Management System API Documentation**

**Table of Contents**

1. [Introduction](#introduction)
2. [Prerequisites](#prerequisites)
3. [Getting Started](#getting-started)
   * [Clone the Repository](#clone-the-repository)
   * [Build the Project](#build-the-project)
   * [Run the Application](#run-the-application)
4. [API Documentation](#api-documentation)
5. [Interacting with the API](#interacting-with-the-api)
   * [Available Endpoints](#available-endpoints)
   * [Request and Response Formats](#request-and-response-formats)
6. [Error Handling](#error-handling)
7. [Testing](#testing)
8. [Common Issues and Troubleshooting](#common-issues-and-troubleshooting)
9. [Conclusion](#conclusion)

## Introduction

This documentation provides an overview of the Library Management System API, developed using Spring Boot. The API allows users to manage books, patrons, and borrowing records within a library system. This document covers how to set up, run, and interact with the API.

## Prerequisites

Before running the application, ensure you have the following installed:

* **Java 11 or higher**
* **Maven 3.6.0 or higher**
* **Git** (optional, for cloning the repository)

## Getting Started

### Clone the Repository : git clone

To get started, clone the repository from your version control system (e.g., GitHub):

Build the Project : mvn clean, mvn install

Run the Application : mvn spring-boot:run

## API Documentation

### Swagger UI

<http://localhost:8080/swagger-ui.html>

## Interacting with the API

### Available Endpoints

Below is a list of available API endpoints grouped by resource:

* **Books**
  + POST /books - Create a new book
  + PUT /books/{id} - Update a book by ID
  + GET /books - Get all books
  + GET /books/{id} - Get a book by ID
  + DELETE /books/{id} - Delete a book by ID
* **Patrons**
  + POST /patrons - Create a new patron
  + PUT /patrons/{id} - Update a patron by ID
  + GET /patrons - Get all patrons
  + GET /patrons/{id} - Get a patron by ID
  + DELETE /patrons/{id} - Delete a patron by ID
* **Borrowing Records**
  + POST /borrow/{bookId}/patron/{patronId} - Borrow a book
  + PUT /return/{bookId}/patron/{patronId} - Return a borrowed book

### Request and Response Formats

All endpoints accept and return JSON data. Below are example request and response formats for key operations:

**Create a Book**:

**Request**: {

"title": "Clean Code",

"author": "Robert C. Martin",

"isbn": "9780132350884",

"publicationYear": 2008,

"description": "this is a book discussing best practices to write clean, readable code"

}

Response: {

"id": 1,

"title": "Clean Code",

"author": "Robert C. Martin",

"isbn": "9780132350884",

"publicationYear": 2008,

"description": "this is a book discussing best practices to write clean, readable code"

}

**Borrow a Book**:

**Request**: No request body needed.

**Response**: {

"id": 1,

"book": {

"id": 1,

"title": "Clean Code",

"author": "Robert C. Martin",

"isbn": "9780132350884",

"publicationYear": 2008,

"description": "this is a book discussing best practices to write clean, readable code"

},

"patron": {

"id": 1,

"name": "John Doe"…..

},

"borrowDate": "2024-08-01T10:15:30",

"returnDate": null

}

## Error Handling

The API uses custom exception handling to return meaningful error messages. Common error responses include:

* **400 Bad Request**: Returned for validation errors.
* **404 Not Found**: Returned when a requested resource (e.g., book or patron) does not exist.
* **409 Conflict**: Returned for data integrity violations (e.g., duplicate entries).

### Example Error Response

{

"timestamp": "2024-08-01T10:15:30",

"status": 404,

"error": "Not Found",

"message": "Book not found",

"path": "/books/999"

}

## Service Aspect

In the **Library Management System**, an aspect-oriented programming (AOP) approach has been applied to cross-cutting concerns for controller methods. This aspect allows you to add functionality like logging, authentication checks, or other reusable code without modifying the core controller logic.

### Purpose

The controller aspect is implemented to:

* Log method calls and execution time.
* Handle repetitive checks (e.g., validation or authorization).

### Example

Before method call: “A method in BookService is about to be executed"

After method call: “A method in BookService has been executed successfully"

## Caching Behavior

The **Library Management System** caches frequently accessed data to improve response times and reduce database load. This section explains the expected caching behavior for the main CRUD operations on entities like books and patrons.

### Get All (e.g., getAllBooks)

* **First Request**: When the getAll endpoint is called for the first time, the database is queried to retrieve all items. The result is then cached and a message will appear in the console.log “Fetching all books from the database”.
* **Subsequent Requests**: For future calls to getAll, the cached list is returned immediately, bypassing the database, unless the cache has been invalidated.

### Get By ID (e.g., getBookById)

* **First Request**: On the first call to getById with a specific ID, the item is fetched from the database and then cached and a message will appear in the console.log "Fetching book details from the database for book ID: (bookId)".
* **Subsequent Requests**: Future requests for the same ID will return the cached item instead of querying the database, unless the cache for this item has been evicted.

### Create, Update, and Delete Operations

* **Create**: When a new item is created, the cache for the entire list (getAll) and any individual items are evicted to ensure that future requests retrieve the updated data from the database.
* **Update**: Upon updating an item, the cache for both getAll and the updated item (getById) is cleared to avoid serving outdated information.
* **Delete**: Similarly, deleting an item evicts its cache entry and the getAll cache to ensure that subsequent requests do not return stale data.

This approach ensures that the application serves up-to-date data while minimizing redundant database calls for frequently accessed information.

## Security and Authentication

The **Library Management System** uses basic authentication to secure endpoints, utilizing in-memory users with assigned roles (USER or ADMIN). The following outlines the expected behavior and requirements for calling each endpoint:

### Authentication

* **Basic Authentication**: Each request to a secured endpoint requires Basic Authentication headers. The client must include the username and password in the request headers.
* **In-Memory Users**:
  + **User**:
    - **Username**: user
    - **Password**: password
    - **Role**: USER
  + **Admin**:
    - **Username**: admin
    - **Password**: adminpass
    - **Role**: ADMIN

### Endpoint Access Control

1. **User Role (USER)**:
   * Can access GET endpoints for patrons.
   * This role has permission for all patrons actions including GET, POST, PUT, and DELETE operations.
2. **Admin Role (ADMIN)**:
   * Can access all endpoints, including GET, POST, PUT, and DELETE operations.
   * Full permissions to create, update, and delete records in addition to reading data.

### Example Requests with Basic Authentication

To make an authenticated request to the system, include the Authorization header in the following format:

Authorization: Basic base64(username:password)

For instance:

* For **User Access** to retrieve books:

**Summary of Access Control**

| **Endpoint** | **HTTP Method** | **User Role** | **Admin Role** |
| --- | --- | --- | --- |
| /patrons | POST | ✅ | ✅ |
| /patrons/{id} | GET | ✅ | ✅ |
| /books | POST | ❌ | ✅ |
| /books/{id} | PUT | ❌ | ✅ |
| /books/{id} | DELETE | ❌ | ✅ |

This security setup provides flexibility for testing and role-based access control without requiring persistent user storage.

## Testing

Unit tests are provided for the service and controller layers. To run the tests, use the following command:

mvn test

The tests are located in the src/test/java directory and cover various scenarios to ensure the API behaves as expected

## Common Issues and Troubleshooting

### Common Issues

* **Application Fails to Start**: Ensure your database connection settings in application.properties are correct.
* **Swagger UI Not Loading**: Ensure Swagger dependencies are correctly configured and the application is running on port 8080.

### Troubleshooting

* Check logs for detailed error messages.
* Ensure all required dependencies are included in the pom.xml.

## Conclusion

This documentation provides a comprehensive guide to running and interacting with the Library Management System API. For any further questions or contributions, feel free to reach out or open an issue on the project's repository.