# Project Documentation: Library Management System API

## ❖ Project Overview

The Library Management System API is designed to simplify and automate the management of library resources, including books, patrons, and borrowing records. Built using Spring Boot, this RESTful API enables librarians to perform a variety of actions such as adding new books, registering patrons, and managing loans.

## ❖ Technology Stack

Spring Boot: For creating the RESTful services.

Java: Programming language.

Maven: Dependency management.

MySQL: Primary database for storing all persistent data.

JUnit, Mockito, Spring Boot Test: For testing the application.

Spring Security: For implementing security measures.

Spring Data JPA: For database interactions.

H2 Database: Used for running integration tests.

Swagger UI: For API documentation and testing.

System Requirements

Java JDK 11 or newer

Maven 3.6 or newer

MySQL Server 8.0 or newer (or any compatible database system)

Any IDE that supports Spring Boot (e.g., IntelliJ IDEA, Eclipse, VS Code)

## ❖ Installation Steps

- **Clone the repository:**

  git clone https://github.com/yourusername/library-management-system.git

  cd library-management-system

- **Database setup:**

  Create a new MySQL Or Postgresql database named library.

  Update the src/main/resources/application.properties file with your database credentials:

  spring.datasource.url=jdbc:mysql://localhost:3306/library_db

  spring.datasource.username=<your_username>

  spring.datasource.password=<your_password>

- **Build the application: Run Application**

# Project Structure Overview

```
library-management-system/
├── src/
│   ├── main/
│   │   ├── java/
│   │   │   └── com/
│   │   │       └── yourcompany/
│   │   │           └── librarymanagement/
│   │   │               ├── config/          # Configuration classes and security config
│   │   │               ├── controller/      # REST API controllers
│   │   │               ├── model/           # Domain model entities
│   │   │               ├── repository/      # Spring Data JPA repositories
│   │   │               ├── service/         # Service classes for business logic
│   │   │               ├── dto/             # Data Transfer Objects
│   │   │               └── LibraryManagementApplication.java # Spring Boot application starter class
│   │   ├── resources/
│   │   │   ├── static/                  # Contains static resources like images, CSS, JavaScript
│   │   │   ├── templates/               # Contains server-side templates
│   │   │   ├── application.properties   # Application configurations
│   │   │   └── db/
│   │   │       └── migration/           # Database migration scripts like Flyway or Liquibase
│   ├── test/
│   │   ├── java/
│   │   │   └── com/
│   │   │       └── yourcompany/
│   │   │           └── librarymanagement/
│   │   │               ├── controller/      # Tests for controllers
│   │   │               ├── service/         # Tests for services
│   │   │               └── repository/      # Tests for repositories
│   ├── resources/
│   │   └── application-test.properties   # Configuration for testing
│
├── .gitignore                    # Specifies intentionally untracked files to ignore
├── mvnw and mvnw.cmd             # Maven wrapper script
├── pom.xml                       # Maven project file
└── README.md
```

# *Project APIS URL's*

Base URL: http://localhost:8080

| Method | Endpoint | Description |
|---|---|---|
| GET | /api/books | Retrieves a list of all books. |
| GET | /api/books/{id} | Retrieves details of a specific book. |
| POST | /api/books | Adds a new book to the library. |
| PUT | /api/books/{id} | Updates an existing book. |
| DELETE | /api/books/{id} | Removes a book from the library. |
| GET | /api/patrons | Retrieves a list of all patrons |
| GET | /api/patrons/{id} | Retrieves details of a specific patron |
| POST | /api/patrons | Adds a new patron to the system. |
| PUT | /api/patrons/{id} | Updates an existing patron's information. |
| DELETE | /api/patrons/{id} | Removes a patron from the system |
| POST | /api/borrow/{bookId}/patron/{patronId} | Allows a patron to borrow a book |
| PUT | /api/return/{bookId}/patron/{patronId} | Records the return of a borrowed book. |

Add Some Endpoints like :

| Method | Endpoint | Description |
|---|---|---|
| Post | /api/auth/token | To be Authorize User and Take JWT TOKEN |

# Bonus Criteria

- Security

  JWT  authentication is implemented for all API endpoints.

  JWT  Algorithm Build with RSA

  Users must provide a valid username and password to access the API.

  ( username  : john , password : jo )

- Transaction Management

  The system uses @Transactional to ensure that all database transactions are handled securely and consistently.

- Caching

  Frequently accessed data such as book details and patron information is cached to improve performance.

- Aspects

  AOP is used to log method calls, exceptions, and to monitor performance metrics.

- Testing

  Comprehensive unit tests have been written using JUnit and Mockito.

- **Functionality**

  Objective: Ensure that all CRUD operations for managing books, patrons, and borrowing records are functional and perform as expected.

  Verification Method: Conduct thorough testing using both automated tests and manual testing to verify that:

  Books can be created, read, updated, and deleted.

  Patrons can be registered, retrieved, updated, and removed.

  Borrowing records accurately track when books are borrowed and returned by patrons.

  Criteria for Success:

  All endpoints must respond with the correct status codes and data according to the operations performed.

  The system should handle relationships between books, patrons, and borrowing records correctly without data integrity issues.

- **Code Quality**

  Objective: Assess the code for readability, maintainability, and adherence to best practices.

  Verification Method: Review code to ensure that:

  It follows clean code principles such as meaningful names, small functions, clear and concise comments where necessary, and proper structuring.

  It adheres to the DRY (Don't Repeat Yourself) principle to avoid redundancy.

  It uses design patterns and best practices appropriate for Spring Boot applications.

  Code is easy to understand and well-documented.Application architecture follows a clear separation of concerns between controllers, services, and repositories.

- ## Error Handling ( Custom Validation )

  Objective: Ensure robust error handling and validate proper handling of edge cases and validation errors.

  Verification Method: Test the API to ensure that:

  All inputs are validated, and errors return appropriate HTTP status codes along with clear, actionable error messages.

  The system gracefully handles unexpected situations, such as database failures, network issues, or bad input data.

  Criteria for Success:

  Invalid requests should consistently return a 400 Bad Request status.

  Requests for non-existent resources should return a 404 Not Found status.

  Server errors should return a 500 Internal Server Error status.

  - **Create Custom Validation to Check on Contact Information of Patron**

- ## Testing

  Objective: Assess the coverage and effectiveness of unit tests.

  Verification Method:

  Utilize tools like JUnit and Mockito alongside Spring Boot's testing capabilities to write and execute tests.

  Measure test coverage using a tool like JaCoCo to ensure significant parts of the codebase are tested.

  Criteria for Success:

  Achieve a test coverage goal (e.g., 80% coverage of classes and methods).

  All tests should pass consistently, and the suite should include tests for both typical use cases and edge cases.

- **Bonus Features**

  Objective: Evaluate the implementation of additional features such as authorization, transactions, caching, and aspect-oriented programming.

  Verification Method:

  Authorization: Ensure that endpoints are protected and only accessible to authenticated users, using either basic authentication or JWTs.

  Transactions: Verify that all operations that involve multiple writes to the database are wrapped in transactions and are atomic.

  Caching: Check that frequently accessed data, such as book details or patron information, is cached effectively to improve performance.

  Aspects: Assess the usage of AOP for logging, performance metrics, and other cross-cutting concerns.

  Criteria for Success:

  Authentication prevents unauthorized access.

  Transactions roll back on failure without leaving the database in an inconsistent state.

  Cached data reduces the number of database hits during typical usage scenarios.

  AOP provides valuable insights and error handling without cluttering business logic.