**Library Management System**

**API Documentation**

**Welcome** to the API documentation for the Library Management System (LMS)! This documentation provides comprehensive insights into the endpoints, functionalities, and usage of the LMS API. Whether you're a developer integrating with the LMS or an administrator managing the system, this guide will assist you in understanding how to interact with the API effectively.

**Overview**

The Library Management System API facilitates the management of books, patrons, borrowing records, and more within a library environment. It offers endpoints for performing CRUD operations on books and patrons, as well as functionalities for borrowing and returning books.

**Getting Started**

Before diving into the API endpoints, it's essential to understand the basic concepts and authentication mechanisms:

* **Authentication**: The API endpoints are protected using Spring Security. Users need to authenticate themselves via username and roles, which are provided as part of the HTTP request headers.

**Endpoints**

The API provides the following endpoints for various operations:

1. **Book Endpoints**: Allows users to perform CRUD operations on books, including retrieval, creation, updating, and deletion.
2. **Patron Endpoints**: Facilitates the management of library patrons, including CRUD operations and patron-specific functionalities.
3. **Borrowing Endpoints**: Enables patrons to borrow and return books, as well as view their borrowing history.

**Testing**

To ensure the reliability and correctness of the API, unit tests have been implemented for each controller endpoint. These tests cover various scenarios, including successful operations, error handling, and edge cases.

**Usage**

Developers can integrate with the LMS API by sending HTTP requests to the appropriate endpoints. The API responses adhere to RESTful principles and provide meaningful information for effective interaction.

**M V C**

**1. Overview**

The Library Management System (LMS) is designed to manage the operations of a library, including handling book and patron management and the borrowing processes. This document covers the MVC architecture used in the LMS, focusing on controllers, services, and models.

**2. Controllers**

**BookController**

* **Purpose**: Manages all operations related to books.
* **Endpoints**:
  + **GET /user/api/books**: Retrieves all books.
  + **GET /user/api/books/{id}**: Retrieves a book by its ID.
  + **POST /user/api/books**: Creates a new book.
  + **PUT /user/api/books/{id}**: Updates an existing book.
  + **DELETE /user/api/books/{id}**: Deletes a book.

**PatronController**

* **Purpose**: Manages all operations related to patrons.
* **Endpoints**:
  + **GET /user/api/patrons**: Retrieves all patrons.
  + **GET /user/api/patrons/{id}**: Retrieves a patron by ID.
  + **POST /user/api/patrons**: Creates a new patron.
  + **PUT /user/api/patrons/{id}**: Updates an existing patron.
  + **DELETE /user/api/patrons/{id}**: Deletes a patron.

**BorrowingController**

* **Purpose**: Manages the borrowing and returning of books.
* **Endpoints**:
  + **POST /user/api/borrow/{bookId}/patron/{patronId}**: Borrow a book.
  + **PUT /user/api/return/{bookId}/patron/{patronId}**: Return a borrowed book.

**3. Models**

**Book**

* **Attributes**:
  + **id**: Primary key.
  + **title**: Title of the book.
  + **author**: Author of the book.
  + **publicationYear**: Year the book was published.
  + **isbn**: International Standard Book Number.
  + **borrowingRecords**: List of borrowing records associated with the book.

**Patron**

* **Attributes**:
  + **id**: Primary key.
  + **name**: Name of the patron.
  + **contactInfo**: Contact details of the patron.
  + **borrowingRecords**: List of borrowing records associated with the patron.

**BorrowingRecord**

* **Attributes**:
  + **id**: Primary key.
  + **book**: Book borrowed.
  + **patron**: Patron who borrowed the book.
  + **borrowDate**: Date the book was borrowed.
  + **returnDate**: Date the book was returned.

**4. Services**

**BookService**

* **Functions**:
  + **findAllBooks()**: Retrieves all books.
  + **findBookById(id)**: Retrieves a specific book by ID.
  + **saveBook(book)**: Saves a new or updated book.
  + **updateBook(id, bookDetails)**: Updates a book.
  + **deleteBook(id)**: Deletes a book.

**PatronService**

* **Functions**:
  + **findAllPatrons()**: Retrieves all patrons.
  + **findPatronById(id)**: Retrieves a specific patron by ID.
  + **savePatron(patron)**: Saves a new or updated patron.
  + **updatePatron(id, patronDetails)**: Updates a patron.
  + **deletePatron(id)**: Deletes a patron.

**BorrowingService**

* **Functions**:
  + **borrowBook(bookId, patronId)**: Records the borrowing of a book by a patron.
  + **returnBook(bookId, patronId)**: Records the return of a borrowed book.

**5-Error Handling**

**Common Error Types**

* **404 Not Found**:
  + **Cause**: The requested resource (book or patron) does not exist.
  + **Response**: Returns a status code of 404 with a message indicating the resource was not found.
* **400 Bad Request**:
  + **Cause**: The request parameters or body did not validate correctly.
  + **Response**: Returns a status code of 400 with a detailed message about the validation errors.
* **500 Internal Server Error**:
  + **Cause**: Unexpected conditions encountered on the server (e.g., database down, runtime exceptions).
  + **Response**: Returns a status code of 500 with a message to contact support or try again later.

**Validation Errors**

* **Trigger**: Occur when data submitted in requests does not meet validation rules set in the models.
* **Handled by**: **MethodArgumentNotValidException** within the **GlobalExceptionHandler**.
* **Response Example**:
  + Invalid book creation where mandatory fields are missing.
  + Response: HTTP 400 Bad Request with a detailed breakdown of each validation failure.

**Custom Exception Handling with GlobalExceptionHandler**

The **GlobalExceptionHandler** class centralizes error handling for the entire application. It captures exceptions and formats responses in a user-friendly manner, ensuring consistency across the system.

**Handling Validation Exceptions**

Here’s how the LMS handles validation errors thrown by the Spring framework when data does not meet the model constraints:

* **MethodArgumentNotValidException**: Captures exceptions related to validation failures in the request body.
* **Error Structure**: Each error consists of the field name causing the validation failure and a message describing the issue.
* **Response**: Constructs a **ResponseEntity** containing the errors mapped by field names, returned with a status code of 400 (Bad Request).

**Validation Rules in Models**

Validation in the LMS is enforced at the model level using annotations from the **jakarta.validation.constraints** package. Here are examples from the **Book** and **Patron** classes:

* **Book Model**:
  + **@NotNull**: Ensures fields like **title**, **author**, and **isbn** are not null.
  + **@Min** and **@Max**: Ensure the publication year falls within acceptable ranges.
  + **@Pattern**: Validates the ISBN format.
* **Patron Model**:
  + **@NotNull**: Ensures that the name and contact information fields are not empty.
  + **@Digits**: Validates that the contact information consists only of digits up to a specified length.

**Security**

**1. Overview**

The security configuration for the Library Management System leverages Spring Security for robust authentication and authorization mechanisms, with a focus on managing API access rights and handling JSON Web Tokens (JWT) for secure communication. Here’s a detailed breakdown:

**Security Configuration Overview**

The SecurityConfiguration class is critical in defining the security protocols for the system, including how requests are authenticated and authorized and how tokens are generated and decoded**.**

**1. Security Configuration Setup**

* SecurityFilterChain: Configures HTTP security, setting authorization rules for different endpoints and enabling JWT-based authentication.
* DaoAuthenticationProvider: Integrates with Spring Security to provide authentication using a data access object (DAO).
* PasswordEncoder: Uses BCrypt hashing for securing passwords.

**2. JWT Handling**

* JwtEncoder and JwtDecoder: Setup for encoding and decoding JWTs. The system uses RSA keys for signing and verifying JWTs.
* JwtAuthenticationConverter: Converts JWT tokens into authentication objects, extracting authorities and mapping them to roles.

**3. Endpoint Security**

* Public Endpoints (/auth/\*\*): Open for registration and login.
* Admin Endpoints (/admin/\*\*): Restricted to users with the "ADMIN" role.
* User Endpoints (/user/\*\*): Accessible to users with "USER" or "ADMIN" roles.

**Registration and Login Endpoints Overview**

These endpoints are part of the authentication process, allowing users to register for an account and log in to the system. They are crucial for user management and securing access to the system.

**1. Registration Endpoint (/auth/register)**

**Purpose:**

Allows new users to create an account in the system. This is the first step for users to gain access to the library management features.

**Request:**

* **Method**: POST
* **URL**: **/auth/register**
* **Body**:

{ "username": "newuser", "password": "password123" }

* **Content-Type**: **application/json**

**Response:**

* **Success (200 OK)**:

{ "userId": 1, "username": "newuser", "authorities": [ { "authority": "ROLE\_USER" } ] }

* **Error (400 Bad Request)**: If the username is already taken or the input data is invalid.

**Security:**

* The password provided by the user is hashed using BCrypt before storing it in the database to ensure security.
* The endpoint checks for duplicate usernames to prevent registration with an already existing username.

**2. Login Endpoint (/auth/login)**

**Purpose:**

Authenticates a user by their username and password, returning a JWT for accessing protected resources.

**Request:**

* **Method**: POST
* **URL**: **/auth/login**
* **Body**:

{ "username": "existinguser", "password": "password123" }

* **Content-Type**: **application/json**

**Response:**

* **Success (200 OK)**:

{ "jwt": "eyJhbGciOiJIUzI1NiIsInR5cCIgOiAi..." }

* **Error (401 Unauthorized)**: If the username or password is incorrect.

**Security:**

* Authentication is performed using Spring Security's **AuthenticationManager**.
* The JWT includes claims such as the user's username and roles, encrypted using RSA keys, ensuring that only an entity with the private key can sign the JWTs.
* JWTs ensure that after initial authentication, subsequent requests do not need to send credentials, reducing the risk of credential interception.

**Aspect-Oriented Programming (AOP)**

Aspect-Oriented Programming (AOP) in Spring Framework is a powerful way to manage cross-cutting concerns such as logging, security, and data management across an application. In your Library Management System, AOP can significantly enhance modularity, making your system easier to maintain and audit.

**Overview of AOP Integration**

The provided **LoggingAspect** class encapsulates logging behavior that can be applied across various components of the system, particularly within service layers. This approach ensures that logging is managed consistently without cluttering the business logic with boilerplate logging code.

**AOP Components**

**1. Aspect Configuration**

**LoggingAspect**

* **Role**: Provides centralized logging for entering methods, method returns, and handling exceptions within specified packages.
* **Methods**:
  + **@Before**: Logs before the execution of a method.
  + **@AfterReturning**: Logs after a method successfully returns a result.
  + **@AfterThrowing**: Logs when a method throws an exception.
  + **@Around**: Monitors method execution time and logs both normal and exceptional returns.

**Pointcut Declaration**

* **Pointcut Name**: **applicationPackagePointcut**
* **Definition**: Targets all methods within the service layers.
* **Scope**:
  + **within(com.maids.Library\_Management\_System.Services..\*)**
  + **within(com.maids.Library\_Management\_System.Security.Services..\*)**

**2. Advices Defined in LoggingAspect**

**Before Advice**

Logs an entry message before any method defined within the pointcut's target packages executes.

@Before("applicationPackagePointcut()") public void logBeforeMethod(JoinPoint joinPoint) { logger.info("Entering in Method : " + joinPoint.getSignature().getName()); }

**After Returning Advice**

Logs a message after a method completes execution successfully, showing the returned result.

@AfterReturning(pointcut = "applicationPackagePointcut()", returning = "result") public void logAfterReturning(JoinPoint joinPoint, Object result) { logger.info("Method Return : " + joinPoint.getSignature().getName() + " with result : " + result); }

**After Throwing Advice**

Logs an error message if a method throws an exception, including the cause of the exception.

@AfterThrowing(pointcut = "applicationPackagePointcut()", throwing = "exception") public void logAfterThrowing(JoinPoint joinPoint, Throwable exception) { logger.error("An exception has been thrown in " + joinPoint.getSignature().getName()); logger.error("Cause : " + exception.getCause()); }

**Around Advice**

Monitors execution time of method calls, logging detailed timings and capturing exceptions if they occur.

@Around("applicationPackagePointcut()") public Object logAround(ProceedingJoinPoint joinPoint) throws Throwable { long start = System.currentTimeMillis(); try { Object result = joinPoint.proceed(); long elapsedTime = System.currentTimeMillis() - start; logger.info("Method execution time for " + joinPoint.getSignature().getName() + " : " + elapsedTime + " milliseconds."); return result; } catch (Exception e) { long elapsedTime = System.currentTimeMillis() - start; logger.error("Method execution time for " + joinPoint.getSignature().getName() + " with exception : " + elapsedTime + " milliseconds."); throw e; } }

**Best Practices and Considerations**

* **Modularity**: AOP helps keep the application modular by separating logging and monitoring logic from business logic.
* **Performance**: While AOP provides powerful capabilities, it's essential to consider its impact on application performance, especially when using **@Around** advices which could add overhead.
* **Security and Compliance**: Logging sensitive information should be handled carefully to comply with data protection regulations.

**Unit Tests Using Mockito**

Unit tests play a crucial role in ensuring the correctness and reliability of your application's controllers. They help verify that the endpoints behave as expected under various conditions. Below are the unit tests using Mockito for the controllers in your Library Management System:

**BookControllerTests**

1. **testGetAllBooks**: Verifies that the endpoint for retrieving all books returns the correct list of books with the expected properties.
2. **testGetBookById\_WhenFound**: Tests the endpoint for retrieving a book by ID when the book is found in the database. It ensures that the correct book details are returned.
3. **testGetBookById\_WhenNotFound**: Tests the scenario where the requested book ID does not exist in the database. It verifies that the endpoint returns a 404 Not Found status.
4. **testCreateBook**: Validates the functionality of the endpoint for creating a new book. It ensures that the book is successfully created and that the correct response is returned.
5. **testUpdateBook\_WhenFound**: Tests the endpoint for updating a book when the book is found in the database. It verifies that the book details are updated correctly.
6. **testUpdateBook\_WhenNotFound**: Tests the scenario where the requested book ID for updating does not exist in the database. It ensures that the endpoint returns a 404 Not Found status.
7. **testDeleteBook\_Successful**: Verifies the functionality of the endpoint for deleting a book when the operation is successful. It ensures that the correct status code is returned, and the book is deleted from the database.
8. **testDeleteBook\_NotFound**: Tests the scenario where the requested book ID for deletion does not exist in the database. It ensures that the endpoint returns a 404 Not Found status.

**BorrowingControllerTests**

1. **testBorrowBook\_Success**: Validates the functionality of the endpoint for borrowing a book. It ensures that the book is successfully borrowed by the patron.
2. **testReturnBook\_Success**: Tests the endpoint for returning a borrowed book. It verifies that the book is successfully returned by the patron.

**PatronControllerTests**

1. **getAllPatrons\_ReturnsPatronList**: Verifies that the endpoint for retrieving all patrons returns the correct list of patrons with the expected properties.
2. **getPatronById\_Exists\_ReturnsPatron**: Tests the endpoint for retrieving a patron by ID when the patron exists in the database. It ensures that the correct patron details are returned.
3. **getPatronById\_NotFound\_ReturnsNotFound**: Tests the scenario where the requested patron ID does not exist in the database. It verifies that the endpoint returns a 404 Not Found status.
4. **createPatron\_SavesNewPatron\_ReturnsSavedPatron**: Validates the functionality of the endpoint for creating a new patron. It ensures that the patron is successfully created and that the correct response is returned.
5. **updatePatron\_Exists\_UpdatesAndReturnsPatron**: Tests the endpoint for updating a patron when the patron exists in the database. It verifies that the patron details are updated correctly.
6. **updatePatron\_NotFound\_ReturnsNotFound**: Tests the scenario where the requested patron ID for updating does not exist in the database. It ensures that the endpoint returns a 404 Not Found status.
7. **deletePatron\_Exists\_DeletesAndReturnsOk**: Verifies the functionality of the endpoint for deleting a patron when the operation is successful. It ensures that the correct status code is returned, and the patron is deleted from the database.
8. **deletePatron\_NotFound\_ReturnsNotFound**: Tests the scenario where the requested patron ID for deletion does not exist in the database. It ensures that the endpoint returns a 404 Not Found status.