Spring Boot-based web application for a simplified Library Management System implement using Spring Security

Type: Projects

Skill: Java

Entity DesignRESTful EndpointsRole-Based Access ControlSecurity ImplementationSpring Boot

Medium

Develop a Spring Boot-based web application for a Library Management System with basic Spring Security features to manage book records, users, and borrowing operations. The system should enable users to register, borrow books, view available books, and manage user accounts. Additionally, there should be role-based access control with roles such as "USER" (who can borrow books) and "ADMIN" (who can manage books and users).

**Functional Requirements:**

1. User Registration and Authentication:

User Registration:

* Implement a RESTful endpoint /users/register with a @PostMapping annotation to allow users to register a new account. The registration should require a unique username and password.
* Roles should be assigned during registration: "USER" or "ADMIN".
* Passwords should be encoded before storing them in the database using Spring Security.
* The Api should accept User object in request body and return a message in response.

User Login:

* Implement a login system with proper authentication using Spring Security. endpoints /users/login with a @PostMapping annotation to allow users to log in to the system.
* A user can authenticate using their username and password to gain access to the system.
* The Api should accept AuthRequest object in request body and return AuthResponse object in response with a JWT token for session management

**Note that:** AuthRequest and AuthResponse are custom classes and already provided under '/src/main/java/com/wecp/library\_management\_system\_jwt/dto/' that represent the request and response objects for user authentication.

Role-Based Access Control:

Implement role-based access control where:

* "USER" role allows the user to borrow books and view book details.
* "ADMIN" role allows the user to add, update, delete books, and view all users.

2. Book Management (Admin Role):

Book CRUD Operations:

a. Create Book:

* Implement a RESTful endpoint /books with a @PostMapping annotation to allow ADMIN users to add new books to the library.
* The Api should accept Book object in request body and return a created book object in response.

b. Update Book:

* Implement a RESTful endpoint /books/{id} with a @PutMapping annotation to allow ADMIN users to update book details.
* The Api should accept Book object in request body, bookId in path variable and return a updated book object in response.

c.Delete Book:

* Implement a RESTful endpoint /books/{id} with a @DeleteMapping annotation to allow ADMIN users to delete a book.
* The Api should accept bookId in path variable and return http status no content in response.

d.View All Books:

* Implement a RESTful endpoint /books with a @GetMapping annotation to retrieve a list of all books in the library. This should be accessible to all users, but only authenticated users should have access to the borrowing feature.
* The Api should return a list of all books in the library with their availability status.

3. Borrowing Books (User Role):

Borrow Book:

* Implement a RESTful endpoint /books/{bookId}/borrow with a @PostMapping annotation. This should allow USER users to borrow books.
* Ensure that only available books can be borrowed. When a book is borrowed, its availability field should be set to false.
* If a user attempts to borrow a book that is already borrowed, they should receive an error message.
* The Api should accept bookId in path variable and return a message in response.

Return Book:

* Implement a RESTful endpoint /books/{bookId}/return with a @PostMapping annotation. This should allow USER users to return books they have borrowed.
* Once the book is returned, its availability should be set to true.
* The Api should accept bookId in path variable and return a message in response.

4. User Account Management:

View User Details:

* Implement a RESTful endpoint /users/{userId} with a @GetMapping annotation to allow users to retrieve details of their own account.
* ADMIN users should be able to view any user's account details by specifying a userId.
* User and Admin both should be able to view user details.

Update User Details:

* Implement a RESTful endpoint /users/{userId} with a @PutMapping annotation to allow users to update their own account details, such as username or password.
* The Api should accept userId in path variable and User object in request body and return updated user object in response.
* User and Admin both should be able to update user details.

**Entities for the System:**

1. Book Entity: A Book entity should contain the following fields:

* id (Long) – Unique identifier (auto-generated).
* title (String) – Title of the book.
* author (String) – Author of the book.
* description (String) – A brief description of the book.
* availability (boolean) – Whether the book is available for borrowing.

2. User Entity: A User entity should contain the following fields:

* id (Long) – Unique identifier (auto-generated).
* username (String) – Unique username.
* password (String) – Encoded password.
* role (String) – Role of the user, e.g., "USER" or "ADMIN".

Implement all Getters and Setters for both entities as per standard Java conventions.

Table name for the entities should be "books" and "users" respectively.

**Security Key points:**

* Generate JWT tokens for user authentication and session management.
* Use BCryptPasswordEncoder for password encoding.
* Ensure that the application is secure and follows best practices for authentication and authorization.

Spring Security should be configured to:

* Implement the Security configuration in SecurityConfig.java.
* Use Authorities for role-based access control. For example: hasAuthority("USER") or hasAuthority("ADMIN").
* Allow registration and login without authentication.
* Secure endpoints like book creation, borrowing, and user management based on roles ("USER" and "ADMIN").
* Ensure that only authenticated users with the appropriate role can access certain endpoints.

Password encoding:

* You need to encode the password before saving the user in the database using BCryptPasswordEncoder provided in Spring Security.
* Role-based access control: The application must have a system for managing user roles such as "USER" and "ADMIN" and restrict access to certain actions based on the role.

Exception Handling:

* Proper error messages should be returned when:
* Users attempt to borrow books that are not available.
* Unauthorized access is attempted.
* JWT Authentication: For managing authenticated sessions, use JWT tokens for stateless authentication.

**Test Cases:**

1. User Registration: Test the POST /users/register endpoint to register a new user and check if the user is saved in the database.
2. Login: Test the POST /users/login endpoint to authenticate a user and ensure that a valid JWT token is returned.
3. Create Book: Test the POST /books endpoint with an "ADMIN" role to create a book and check if it is added to the database.
4. Update Book: Test the PUT /books/{bookId} endpoint as an "ADMIN" role to update book details and check if the changes are reflected.
5. Delete Book: Test the DELETE /books/{bookId} endpoint as an "ADMIN" role to delete a book and verify that it is removed from the database.
6. View All Books: Test the GET /books endpoint to retrieve a list of all books and verify that the availability status is correct. ADMIN and USER roles should have access to this endpoint.
7. Borrow Book: Test the POST /books/{bookId}/borrow endpoint as a "USER" role to borrow an available book and check if the availability status is updated.
8. Return Book: Test the POST /books/{bookId}/return endpoint to return a borrowed book and check if the availability is updated.
9. View User Details: Test the GET /users/{userId} endpoint to retrieve user information and ensure that only the logged-in user or "ADMIN" can access the details.