Event Management System implement using Spring Security

Type: Projects

Skill: Java

Entity RelationshipsRESTful EndpointsRole-Based Access ControlSpring BootSpring Security

Medium

Develop a Spring Boot-based web application for a simplified Event Management System where users can view events, book tickets, and manage their bookings. The platform should support basic user authentication and authorization, and have two roles: USER (for regular users) and ADMIN (for event managers). Admins can create, update, and delete events, while Users can view and book tickets for events.

**Functional Requirements:**

1. User Authentication and Authorization:

User Registration:

* Implement a RESTful endpoint /users/register with a @PostMapping annotation to allow users to register a new account.
* The registration request should include the username, password, email, first name, last name, and role (either "USER" or "ADMIN").
* The password should be encrypted using BCryptPasswordEncoder before saving to the database.
* The API should accept User object in thr request body and return registered user object in the response.

User Login:

* Implement a login system using Spring Security to authenticate users with their username and password.
* After successful login, generate and return a JWT token to be used for further requests.
* API = POST /users/login
* Input: { "username": "user1", "password": "password123" }
* Output: { "token": "jwt-token" }
* You can use already provided Dto classes `AuthRequest` and `AuthResponse` for this API.
* DTO classes for login api are already provided in the project at location `src/main/java/com/wecp/event\_management\_system\_jwt/dto`
* If password is incorrect, return a 401 Unauthorized response.

Role-Based Authorization:

* USER role can view events and book tickets.
* ADMIN role can manage events, including creating, updating, and deleting events, as well as viewing all bookings.
* Check permission in SecurityConfig.java file.
* Use hasAuthority() method to check the role of the user.
* For example hasAuthority("ADMIN") or hasAuthority("USER")

2. Event Management (Admin Role):

Event Entity:

* An Event entity should have the following fields:
* id (Long) – Unique identifier (auto-generated).
* name (String) – Name of the event.
* description (String) – Description of the event.
* date (LocalDateTime) – Date and time of the event.
* location (String) – Venue or location of the event.
* availableTickets (int) – Total number of tickets available for the event.
* price (double) – Price per ticket.

CRUD Operations for Events:

* Create Event: Implement a RESTful endpoint /events with a @PostMapping annotation to allow ADMIN users to create new events.
* The Api should accept Event object in the request body and return created event object in the response.
* Update Event: Implement a RESTful endpoint /events/{id} with a @PutMapping annotation to allow ADMIN users to update event details (e.g., price, date, location).
* The API should accept Event object in the request body and return updated event object in the response.
* Delete Event: Implement a RESTful endpoint /events/{id} with a @DeleteMapping annotation to allow ADMIN users to delete events.
* View Events: Implement a RESTful endpoint /events with a @GetMapping annotation to return a list of all events. This should be accessible by both USER and ADMIN users.

3. Ticket Booking (User and Admin Role):

Booking Entity:

* A Booking entity should have the following fields:
* id (Long) – Unique identifier (auto-generated).
* user (User) – Many-to-one relationship with User entity.
* event (Event) – Many-to-one relationship with Event entity.
* quantity (integer) – Number of tickets booked.
* totalAmount (double) – Total booking amount (calculated as quantity \* price).
* bookingDate (LocalDateTime) – Date and time of booking.

Booking Tickets:

* Implement a RESTful endpoint /bookings?userId={userId}&eventId={eventId}&quantity={quantity}
* with a @PostMapping annotation to allow USER users to book tickets for an event.
* The booking request should include event ID, userId, quantity, and the total amount (calculated automatically based on the ticket price).
* Ensure that users cannot book more tickets than the available quantity for the event.
* The Api should accept userId, eventId and quantity in request parameters and return booking object in the response.

View User Bookings:

* Implement a RESTful endpoint /bookings/users/{userId} with a @GetMapping annotation to allow users to view their bookings.

Cancel Booking:

* Implement a RESTful endpoint /bookings/{id} with a @DeleteMapping annotation to allow USER users to cancel their own bookings.

**Entities:**

User Entity:

* id (Long) – Unique identifier (auto-generated).
* username (String) – Unique username.
* password (String) – Encoded password.
* firstName (String) – User's first name.
* lastName (String) – User's last name.
* email (String) – User's email address.
* role (String) – Role of the user, either "USER" or "ADMIN".

Event Entity:

* id (Long) – Unique identifier (auto-generated).
* name (String) – Name of the event.
* description (String) – Description of the event.
* date (LocalDateTime) – Date and time of the event.
* location (String) – Location or venue of the event.
* availableTickets (int) – Number of available tickets.
* price (double) – Price of each ticket.

Booking Entity:

* id (Long) – Unique identifier (auto-generated).
* user (User) – Many-to-one relationship with User entity.
* event (Event) – Many-to-one relationship with Event entity.
* quantity (integer) – Number of tickets booked.
* totalAmount (double) – Total amount paid for the booking.
* bookingDate (LocalDateTime) – Date and time when the booking was made.

Implement Getters and Setters of all fields in the entities as per standard java practices. Mapped the entities with the table names as users, events, bookings respectively.

**Application Requirement / Notes:**

* Spring Security should be configured to secure the system, allowing only authenticated users to book tickets and manage bookings.
* JWT Tokens should be used for stateless authentication.
* Password Encoding: Use BCryptPasswordEncoder to securely store passwords.
* Role-based Access Control: Implement role-based authorization so that only ADMIN users can manage events, while USER users can book tickets.
* Booking Validation: Ensure users cannot book more tickets than the available stock for an event.
* Implement Getters and Setters of all fields in the entities as per standard java practices.
* Mapped the entities with the table names as users, events, bookings respectively.
* Use hasAuthority() method to check the role of the user. For example hasAuthority("ADMIN") or hasAuthority("USER")
* You need to write API in a.http file to use it via Restclient.