E-Commerce Platform with Order Management and User Roles using Spring Security

Type: Projects

Skill: Java

Entity RelationshipsRESTful EndpointsRole-Based AuthorizationSpring BootSpring Security

Medium

Develop a Spring Boot-based web application for a simplified E-Commerce Platform where users can browse products, place orders, and manage their personal details. The platform should incorporate basic Spring Security features to ensure secure user authentication and authorization. There will be two roles: "USER" and "ADMIN", with different access permissions for managing products and orders.

**Functional Requirements:**

1. User Authentication and Authorization:

Role-Based Authorization:

* USER role allows browsing products and placing orders.
* ADMIN role allows managing products, viewing orders, and managing user accounts.
* Check permission in SecurityConfig.java file.
* Use hasAuthority() method to check the role of the user.
* For example hasAuthority("ADMIN") or hasAuthority("USER")

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".

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 user’s username, password, first name, last name, email, and role (either "USER" or "ADMIN").
* Passwords should be encrypted using BCryptPasswordEncoder before storing them in the database.
* The API should accept User objects in RequestBody and return created User objects in ResponseBody.

User Login:

* Implement a login system using Spring Security to authenticate users with their username and password.
* After successful login, a JWT token should be returned to be used for subsequent 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/order\_management\_system\_jwt/dto`
* If password is incorrect, return a 401 Unauthorized response.

**Registration and login should be accessible by all users, including unauthenticated users.**

2. Product Management (Admin Role):

Product Entity:

* A Product entity should have the following fields:
* id (Long) – Unique identifier (auto-generated).
* name (String) – Name of the product.
* description (String) – Description of the product.
* price (double) – Price of the product.
* stock (int) – Available stock for the product.

Product CRUD Operations (Admin only):

a.Create Product:

* Implement a RESTful endpoint /products with a @PostMapping annotation to allow ADMIN users to add new products to the platform.
* The Api Should accept Product objects in RequestBody and return created Product objects in ResponseBody.

b.Update Product:

* Implement a RESTful endpoint /products/{id} with a @PutMapping annotation to allow ADMIN users to update product details (e.g., price, stock).
* The API should accept Product objects in RequestBody and return updated Product objects in ResponseBody.

c.Delete Product:

* Implement a RESTful endpoint /products/{id} with a @DeleteMapping annotation to allow ADMIN users to delete a product.
* View All Products: Implement a RESTful endpoint /products with a @GetMapping annotation that returns a list of all products. This should be accessible by all users.

3. Order Management (User and Admin Roles):

Order Entity:

An Order entity should have the following fields:

* id (Long) – Unique identifier (auto-generated).
* user (User) – Many-to-one relationship with the User entity.
* product (Product) – Many-to-one relationship with the Product entity.
* quantity (int) – Number of units ordered.
* totalAmount (double) – Total order amount (calculated as quantity \* price).
* status (String) – Order status (e.g., "PENDING", "SHIPPED", "DELIVERED").

Placing an Order:

* Implement a RESTful endpoint /orders?productId={productId}&userId={userId} with a @PostMapping annotation to allow USER users to place an order.
* The API should accept productId and userId as query parameters, Order Object in RequestBody and return created Order objects in ResponseBody.
* calculate total amount (calculated automatically based on the price of the product).
* Ensure that the requested quantity does not exceed the available stock for the product.
* Once an order is placed, update the stock of the product.

Viewing Orders:

* Implement a RESTful endpoint /orders/{userId} with a @GetMapping annotation that allows users to view their own orders.
* ADMIN users should have access to view all orders, not just their own.
* The API should accept userId as a path variable and return a list of orders in ResponseBody.

Updating Order Status:

* Implement a RESTful endpoint /orders/{id}?status={status} with a @PutMapping annotation to allow ADMIN users to update the status of an order (e.g., from "PENDING" to "SHIPPED").
* The API should accept status as a query parameter and return updated Order objects in ResponseBody.

4. User Management (Admin Role):

View User Information:

* Implement a RESTful endpoint /users/{userId} with a @GetMapping annotation to allow ADMIN users to view details of any user.

Update User Information:

* Implement a RESTful endpoint /users/{userId} with a @PutMapping annotation to allow ADMIN users to update a user's personal details (e.g., name, email, role).
* The API should accept User objects in RequestBody and return updated User objects in ResponseBody.

**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".

Product Entity:

* id (Long) – Unique identifier (auto-generated).
* name (String) – Name of the product.
* description (String) – Description of the product.
* price (double) – Price of the product.
* stock (int) – Available stock of the product.

Order Entity:

* id (Long) – Unique identifier (auto-generated).
* user (User) – Many-to-one relationship with User entity.
* product (Product) – Many-to-one relationship with Product entity.
* quantity (Integer) – Number of units ordered.
* totalAmount (Double) – Total order amount.
* status (String) – Order status (e.g., "PENDING", "SHIPPED").

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

**Notes:**

* Spring Security should be configured to secure the system, allowing only authenticated users to place orders and perform other sensitive actions.
* JWT Tokens should be used for stateless authentication.
* Password Encoding: Ensure that passwords are securely encoded using BCryptPasswordEncoder.
* Role-based Access Control: Implement role-based authorization so that only ADMIN users can manage products and user information, while USER users can place orders and view their personal orders.
* Stock Management: Ensure that orders cannot exceed the available stock for a product. If the stock is insufficient, the user should receive an error message.
* Mapped the entities with the table names as users, products, orders respectively.
* Implement Getters and Setters of all fields in the entities as per standard java practices.

// ============ ENTITIES ============

@Entity

@Table(name = "users")

public class User {

@Id @GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

@Column(unique = true, nullable = false)

private String username;

@Column(nullable = false)

private String password;

private String firstName;

private String lastName;

private String email;

private String role; // USER or ADMIN

// Getters and Setters

}

@Entity

@Table(name = "products")

public class Product {

@Id @GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

private String description;

private double price;

private int stock;

// Getters and Setters

}

@Entity

@Table(name = "orders")

public class Order {

@Id @GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

@ManyToOne(fetch = FetchType.LAZY)

@JoinColumn(name = "user\_id")

private User user;

@ManyToOne(fetch = FetchType.LAZY)

@JoinColumn(name = "product\_id")

private Product product;

private int quantity;

private double totalAmount;

private String status;

// Getters and Setters

}

// ============ DTOs ============

public class AuthRequest {

private String username;

private String password;

// Getters and Setters

}

public class AuthResponse {

private String token;

public AuthResponse(String token) { this.token = token; }

// Getter and Setter

}

// ============ REPOSITORIES ============

public interface UserRepository extends JpaRepository<User, Long> {

Optional<User> findByUsername(String username);

}

public interface ProductRepository extends JpaRepository<Product, Long> {}

public interface OrderRepository extends JpaRepository<Order, Long> {

List<Order> findByUserId(Long userId);

}

// ============ SERVICES ============

@Service

public class UserService implements UserDetailsService {

@Autowired private UserRepository userRepository;

@Autowired private PasswordEncoder passwordEncoder;

public User registerUser(User user) {

user.setPassword(passwordEncoder.encode(user.getPassword()));

return userRepository.save(user);

}

public User updateUser(Long id, User updatedUser) {

User user = userRepository.findById(id).orElseThrow();

user.setFirstName(updatedUser.getFirstName());

user.setLastName(updatedUser.getLastName());

user.setEmail(updatedUser.getEmail());

user.setRole(updatedUser.getRole());

return userRepository.save(user);

}

public User getUser(Long id) {

return userRepository.findById(id).orElseThrow();

}

@Override

public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {

User user = userRepository.findByUsername(username).orElseThrow(() -> new UsernameNotFoundException("User not found"));

return new org.springframework.security.core.userdetails.User(

user.getUsername(),

user.getPassword(),

Collections.singleton(new SimpleGrantedAuthority(user.getRole()))

);

}

}

@Service

public class JwtService {

@Autowired private JwtUtil jwtUtil;

@Autowired private AuthenticationManager authenticationManager;

public String authenticateAndGenerateToken(String username, String password) {

authenticationManager.authenticate(new UsernamePasswordAuthenticationToken(username, password));

return jwtUtil.generateToken(new org.springframework.security.core.userdetails.User(username, password, new ArrayList<>()));

}

}

// ============ SECURITY CONFIGURATION ============

@Component

public class JwtUtil {

private final String SECRET\_KEY = "secret";

public String generateToken(UserDetails userDetails) {

return Jwts.builder()

.setSubject(userDetails.getUsername())

.setIssuedAt(new Date())

.setExpiration(new Date(System.currentTimeMillis() + 1000 \* 60 \* 60 \* 10))

.signWith(SignatureAlgorithm.HS256, SECRET\_KEY)

.compact();

}

public String extractUsername(String token) {

return Jwts.parser().setSigningKey(SECRET\_KEY).parseClaimsJws(token).getBody().getSubject();

}

public boolean validateToken(String token, UserDetails userDetails) {

return extractUsername(token).equals(userDetails.getUsername()) && !isTokenExpired(token);

}

private boolean isTokenExpired(String token) {

return Jwts.parser().setSigningKey(SECRET\_KEY).parseClaimsJws(token).getBody().getExpiration().before(new Date());

}

}

@Component

public class JwtAuthenticationFilter extends OncePerRequestFilter {

@Autowired private JwtUtil jwtUtil;

@Autowired private UserDetailsService userDetailsService;

@Override

protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain filterChain)

throws ServletException, IOException {

String authHeader = request.getHeader("Authorization");

String token = null;

String username = null;

if (authHeader != null && authHeader.startsWith("Bearer ")) {

token = authHeader.substring(7);

username = jwtUtil.extractUsername(token);

}

if (username != null && SecurityContextHolder.getContext().getAuthentication() == null) {

UserDetails userDetails = userDetailsService.loadUserByUsername(username);

if (jwtUtil.validateToken(token, userDetails)) {

UsernamePasswordAuthenticationToken authToken = new UsernamePasswordAuthenticationToken(

userDetails, null, userDetails.getAuthorities());

authToken.setDetails(new WebAuthenticationDetailsSource().buildDetails(request));

SecurityContextHolder.getContext().setAuthentication(authToken);

}

}

filterChain.doFilter(request, response);

}

}

@Configuration

@EnableWebSecurity

public class SecurityConfig {

@Autowired private JwtAuthenticationFilter jwtFilter;

@Autowired private UserDetailsService userDetailsService;

@Bean

public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {

http.csrf().disable()

.authorizeHttpRequests(auth -> auth

.requestMatchers("/users/register", "/users/login").permitAll()

.requestMatchers("/products/\*\*", "/orders/\*\*", "/users/\*\*").authenticated()

)

.sessionManagement(sm -> sm.sessionCreationPolicy(SessionCreationPolicy.STATELESS))

.addFilterBefore(jwtFilter, UsernamePasswordAuthenticationFilter.class);

return http.build();

}

@Bean

public PasswordEncoder passwordEncoder() {

return new BCryptPasswordEncoder();

}

@Bean

public AuthenticationManager authManager(HttpSecurity http) throws Exception {

return http.getSharedObject(AuthenticationManagerBuilder.class)

.userDetailsService(userDetailsService)

.passwordEncoder(passwordEncoder())

.and().build();

}

}

// ============ CONTROLLERS ============

// [AuthController, UserController, ProductController, OrderController are already included below ↓ ↓ ↓]

// ============ AuthController ============

@RestController

@RequestMapping("/users")

public class AuthController {

@Autowired private UserService userService;

@Autowired private JwtService jwtService;

@PostMapping("/register")

public ResponseEntity<User> register(@RequestBody User user) {

User savedUser = userService.registerUser(user);

return new ResponseEntity<>(savedUser, HttpStatus.CREATED);

}

@PostMapping("/login")

public ResponseEntity<AuthResponse> login(@RequestBody AuthRequest request) {

try {

String token = jwtService.authenticateAndGenerateToken(request.getUsername(), request.getPassword());

return ResponseEntity.ok(new AuthResponse(token));

} catch (AuthenticationException e) {

return ResponseEntity.status(HttpStatus.UNAUTHORIZED).build();

}

}

}

// ============ UserController ============

@RestController

@RequestMapping("/users")

public class UserController {

@Autowired private UserService userService;

@GetMapping("/{userId}")

@PreAuthorize("hasAuthority('ADMIN')")

public ResponseEntity<User> getUserById(@PathVariable Long userId) {

User user = userService.getUser(userId);

return ResponseEntity.ok(user);

}

@PutMapping("/{userId}")

@PreAuthorize("hasAuthority('ADMIN')")

public ResponseEntity<User> updateUser(@PathVariable Long userId, @RequestBody User updatedUser) {

User user = userService.updateUser(userId, updatedUser);

return ResponseEntity.ok(user);

}

}

// ============ ProductController ============

@RestController

@RequestMapping("/products")

public class ProductController {

@Autowired private ProductRepository productRepository;

@GetMapping

public ResponseEntity<List<Product>> getAllProducts() {

return ResponseEntity.ok(productRepository.findAll());

}

@PostMapping

@PreAuthorize("hasAuthority('ADMIN')")

public ResponseEntity<Product> createProduct(@RequestBody Product product) {

return new ResponseEntity<>(productRepository.save(product), HttpStatus.CREATED);

}

@PutMapping("/{id}")

@PreAuthorize("hasAuthority('ADMIN')")

public ResponseEntity<Product> updateProduct(@PathVariable Long id, @RequestBody Product updatedProduct) {

Product product = productRepository.findById(id).orElseThrow();

product.setName(updatedProduct.getName());

product.setDescription(updatedProduct.getDescription());

product.setPrice(updatedProduct.getPrice());

product.setStock(updatedProduct.getStock());

return ResponseEntity.ok(productRepository.save(product));

}

@DeleteMapping("/{id}")

@PreAuthorize("hasAuthority('ADMIN')")

public ResponseEntity<Void> deleteProduct(@PathVariable Long id) {

productRepository.deleteById(id);

return ResponseEntity.noContent().build();

}

}

// ============ OrderController ============

@RestController

@RequestMapping("/orders")

public class OrderController {

@Autowired private OrderRepository orderRepository;

@Autowired private ProductRepository productRepository;

@Autowired private UserRepository userRepository;

@PostMapping

@PreAuthorize("hasAuthority('USER')")

public ResponseEntity<?> placeOrder(@RequestParam Long productId, @RequestParam Long userId, @RequestBody Order orderRequest) {

Product product = productRepository.findById(productId).orElseThrow();

User user = userRepository.findById(userId).orElseThrow();

if (orderRequest.getQuantity() > product.getStock()) {

return ResponseEntity.badRequest().body("Insufficient stock");

}

Order order = new Order();

order.setUser(user);

order.setProduct(product);

order.setQuantity(orderRequest.getQuantity());

order.setTotalAmount(orderRequest.getQuantity() \* product.getPrice());

order.setStatus("PENDING");

product.setStock(product.getStock() - orderRequest.getQuantity());

productRepository.save(product);

return new ResponseEntity<>(orderRepository.save(order), HttpStatus.CREATED);

}

@GetMapping("/{userId}")

@PreAuthorize("hasAuthority('USER') or hasAuthority('ADMIN')")

public ResponseEntity<List<Order>> getOrders(@PathVariable Long userId, Authentication auth) {

User user = userRepository.findByUsername(auth.getName()).orElseThrow();

if (user.getRole().equals("ADMIN") || user.getId().equals(userId)) {

return ResponseEntity.ok(orderRepository.findByUserId(userId));

} else {

return ResponseEntity.status(HttpStatus.FORBIDDEN).build();

}

}

@PutMapping("/{id}")

@PreAuthorize("hasAuthority('ADMIN')")

public ResponseEntity<Order> updateOrderStatus(@PathVariable Long id, @RequestParam String status) {

Order order = orderRepository.findById(id).orElseThrow();

order.setStatus(status);

return ResponseEntity.ok(orderRepository.save(order));

}

}

Application.properties

# ========== Server Config ==========

server.port=5000

# ========== MySQL Database Config ==========

spring.datasource.url=jdbc:mysql://localhost:3306/ecommerce\_db?useSSL=false&serverTimezone=UTC

spring.datasource.username=root

spring.datasource.password=your\_password\_here

spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

# ========== JPA / Hibernate ==========

spring.jpa.hibernate.ddl-auto=update

spring.jpa.show-sql=true

spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL8Dialect

# ========== JWT Secret (Optional external config) ==========

# jwt.secret=your\_jwt\_secret\_key\_here