**Exercise 12: Online Bookstore - Securing RESTful Endpoints with Spring Security**

**Business Scenario:**

Secure your bookstore's RESTful endpoints using Spring Security with JWT-based authentication.

Maven Dependencies:

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-security</artifactId>

</dependency>

<dependency>

<groupId>io.jsonwebtoken</groupId>

<artifactId>jjwt</artifactId>

<version>0.9.1</version>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

Creating a Security Configuration Class:

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.security.authentication.AuthenticationManager;

import org.springframework.security.config.annotation.web.builders.HttpSecurity;

import org.springframework.security.config.annotation.web.builders.WebSecurity;

import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;

import org.springframework.security.config.http.SessionCreationPolicy;

import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;

import org.springframework.security.crypto.password.PasswordEncoder;

import org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter;

@Configuration

@EnableWebSecurity

public class SecurityConfig extends WebSecurityConfigurerAdapter {

@Override

protected void configure(HttpSecurity http) throws Exception {

http.csrf().disable()

.authorizeRequests()

.antMatchers("/api/auth/\*\*").permitAll()

.anyRequest().authenticated()

.and()

.sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS);

http.addFilterBefore(jwtAuthenticationFilter(), UsernamePasswordAuthenticationFilter.class);

}

@Override

public void configure(WebSecurity web) throws Exception {

web.ignoring().antMatchers("/h2-console/\*\*"); // For H2 console (if using)

}

@Bean

public PasswordEncoder passwordEncoder() {

return new BCryptPasswordEncoder();

}

@Bean

@Override

public AuthenticationManager authenticationManagerBean() throws Exception {

return super.authenticationManagerBean();

}

@Bean

public JwtAuthenticationFilter jwtAuthenticationFilter() {

return new JwtAuthenticationFilter();

}

}

Creating a JWT Utility Class:

import io.jsonwebtoken.Claims;

import io.jsonwebtoken.Jwts;

import io.jsonwebtoken.SignatureAlgorithm;

import org.springframework.security.core.userdetails.UserDetails;

import org.springframework.stereotype.Component;

import java.util.Date;

import java.util.HashMap;

import java.util.Map;

import java.util.function.Function;

@Component

public class JwtUtil {

private String secret = "your\_secret\_key"; // Change this to your actual secret key

public String extractUsername(String token) {

return extractClaim(token, Claims::getSubject);

}

public Date extractExpiration(String token) {

return extractClaim(token, Claims::getExpiration);

}

public <T> T extractClaim(String token, Function<Claims, T> claimsResolver) {

final Claims claims = extractAllClaims(token);

return claimsResolver.apply(claims);

}

private Claims extractAllClaims(String token) {

return Jwts.parser().setSigningKey(secret).parseClaimsJws(token).getBody();

}

private Boolean isTokenExpired(String token) {

return extractExpiration(token).before(new Date());

}

public String generateToken(UserDetails userDetails) {

Map<String, Object> claims = new HashMap<>();

return createToken(claims, userDetails.getUsername());

}

private String createToken(Map<String, Object> claims, String subject) {

return Jwts.builder().setClaims(claims).setSubject(subject).setIssuedAt(new Date(System.currentTimeMillis()))

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

.signWith(SignatureAlgorithm.HS256, secret).compact();

}

public Boolean validateToken(String token, UserDetails userDetails) {

final String username = extractUsername(token);

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

}

}

Creating a JWT Authentication Filter:

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.security.core.context.SecurityContextHolder;

import org.springframework.security.core.userdetails.UserDetails;

import org.springframework.security.core.userdetails.UserDetailsService;

import org.springframework.security.web.authentication.WebAuthenticationDetailsSource;

import org.springframework.stereotype.Component;

import org.springframework.web.filter.OncePerRequestFilter;

import javax.servlet.FilterChain;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

@Component

public class JwtAuthenticationFilter extends OncePerRequestFilter {

@Autowired

private JwtUtil jwtUtil;

@Autowired

private UserDetailsService userDetailsService;

@Override

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

throws java.io.IOException, javax.servlet.ServletException {

final String authorizationHeader = request.getHeader("Authorization");

String username = null;

String jwt = null;

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

jwt = authorizationHeader.substring(7);

username = jwtUtil.extractUsername(jwt);

}

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

UserDetails userDetails = this.userDetailsService.loadUserByUsername(username);

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

UsernamePasswordAuthenticationToken authenticationToken = new UsernamePasswordAuthenticationToken(

userDetails, null, userDetails.getAuthorities());

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

SecurityContextHolder.getContext().setAuthentication(authenticationToken);

}

}

chain.doFilter(request, response);

}

}

Implementing UserDetailsService and UserDetails:

import org.springframework.security.core.userdetails.User;

import org.springframework.security.core.userdetails.UserDetails;

import org.springframework.security.core.userdetails.UserDetailsService;

import org.springframework.security.core.userdetails.UsernameNotFoundException;

import org.springframework.stereotype.Service;

import java.util.ArrayList;

@Service

public class CustomUserDetailsService implements UserDetailsService {

@Override

public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {

// For simplicity, using a hardcoded user

// In a real-world scenario, load user details from a database

if ("user".equals(username)) {

return new User("user", new BCryptPasswordEncoder().encode("password"), new ArrayList<>());

} else {

throw new UsernameNotFoundException("User not found with username: " + username);

}

}

}

Creating Authentication Controller:

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.security.authentication.AuthenticationManager;

import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;

import org.springframework.security.core.userdetails.UserDetails;

import org.springframework.security.core.userdetails.UserDetailsService;

import org.springframework.web.bind.annotation.\*;

@RestController

@RequestMapping("/api/auth")

public class AuthController {

@Autowired

private AuthenticationManager authenticationManager;

@Autowired

private JwtUtil jwtUtil;

@Autowired

private UserDetailsService userDetailsService;

@PostMapping("/login")

public ResponseEntity<?> createAuthenticationToken(@RequestBody AuthRequest authRequest) throws Exception {

authenticationManager.authenticate(

new UsernamePasswordAuthenticationToken(authRequest.getUsername(), authRequest.getPassword()));

final UserDetails userDetails = userDetailsService.loadUserByUsername(authRequest.getUsername());

final String jwt = jwtUtil.generateToken(userDetails);

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

}

}

Creating Authentication Request and Response Models:

public class AuthRequest {

private String username;

private String password;

// getters and setters

}

public class AuthResponse {

private String jwt;

public AuthResponse(String jwt) {

this.jwt = jwt;

}

public String getJwt() {

return jwt;

}

}

Adding CORS Configuration:

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.web.cors.CorsConfiguration;

import org.springframework.web.cors.UrlBasedCorsConfigurationSource;

import org.springframework.web.filter.CorsFilter;

import java.util.Arrays;

@Configuration

public class CorsConfig {

@Bean

public CorsFilter corsFilter() {

UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();

CorsConfiguration config = new CorsConfiguration();

config.setAllowCredentials(true);

config.setAllowedOrigins(Arrays.asList("http://localhost:3000")); // Adjust based on your frontend's URL

config.setAllowedHeaders(Arrays.asList("Origin", "Content-Type", "Accept", "Authorization"));

config.setAllowedMethods(Arrays.asList("GET", "POST", "PUT", "DELETE", "OPTIONS"));

source.registerCorsConfiguration("/\*\*", config);

return new CorsFilter(source);

}

}

Testing the Secured Endpoints:

curl -X POST http://localhost:8080/api/auth/login -H "Content-Type: application/json" -d '{"username":"user", "password":"password"}'

curl -X GET http://localhost:8080/api/books -H "Authorization: Bearer <your-jwt-token>"

**Exercise 13: Online Bookstore - Unit Testing REST Controllers**

**Business Scenario:**

Write unit tests for your bookstore's REST controllers using JUnit and Mockito.

Maven Dependencies:

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

<exclusions>

<exclusion>

<groupId>org.mockito</groupId>

<artifactId>mockito-core</artifactId>

</exclusion>

</exclusions>

</dependency>

<dependency>

<groupId>org.mockito</groupId>

<artifactId>mockito-core</artifactId>

<version>5.0.0</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.mockito</groupId>

<artifactId>mockito-junit-jupiter</artifactId>

<version>5.0.0</version>

<scope>test</scope>

</dependency>

Using MockMvc to Write Unit Tests for Your REST Controllers

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.annotation.\*;

import java.util.List;

@RestController

@RequestMapping("/api/books")

public class BookController {

@Autowired

private BookService bookService;

@GetMapping

public ResponseEntity<List<Book>> getAllBooks() {

List<Book> books = bookService.findAll();

return ResponseEntity.ok(books);

}

@GetMapping("/{id}")

public ResponseEntity<Book> getBookById(@PathVariable Long id) {

return bookService.findById(id)

.map(ResponseEntity::ok)

.orElse(ResponseEntity.notFound().build());

}

@PostMapping

public ResponseEntity<Book> createBook(@RequestBody Book book) {

Book createdBook = bookService.save(book);

return ResponseEntity.status(201).body(createdBook);

}

@PutMapping("/{id}")

public ResponseEntity<Book> updateBook(@PathVariable Long id, @RequestBody Book bookDetails) {

return bookService.findById(id)

.map(book -> {

book.setTitle(bookDetails.getTitle());

book.setAuthor(bookDetails.getAuthor());

book.setPrice(bookDetails.getPrice());

Book updatedBook = bookService.save(book);

return ResponseEntity.ok(updatedBook);

})

.orElse(ResponseEntity.notFound().build());

}

@DeleteMapping("/{id}")

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

return bookService.findById(id).map(book -> {

bookService.deleteById(id);

return ResponseEntity.noContent().build();

}).orElse(ResponseEntity.notFound().build());

}

}

Example Unit Tests Using MockMvc and Mockito:

import org.junit.jupiter.api.BeforeEach;

import org.junit.jupiter.api.Test;

import org.junit.jupiter.api.extension.ExtendWith;

import org.mockito.InjectMocks;

import org.mockito.Mock;

import org.mockito.MockitoAnnotations;

import org.mockito.junit.jupiter.MockitoExtension;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;

import org.springframework.boot.test.mock.mockito.MockBean;

import org.springframework.http.MediaType;

import org.springframework.test.web.servlet.MockMvc;

import org.springframework.test.web.servlet.setup.MockMvcBuilders;

import java.util.Arrays;

import java.util.Optional;

import static org.mockito.ArgumentMatchers.any;

import static org.mockito.Mockito.\*;

import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.\*;

import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.\*;

@ExtendWith(MockitoExtension.class)

@WebMvcTest(BookController.class)

public class BookControllerTest {

@Autowired

private MockMvc mockMvc;

@MockBean

private BookService bookService;

private Book book1;

private Book book2;

@BeforeEach

public void setUp() {

book1 = new Book(1L, "Book One", "Author One", 9.99);

book2 = new Book(2L, "Book Two", "Author Two", 14.99);

}

@Test

public void testGetAllBooks() throws Exception {

when(bookService.findAll()).thenReturn(Arrays.asList(book1, book2));

mockMvc.perform(get("/api/books")

.contentType(MediaType.APPLICATION\_JSON))

.andExpect(status().isOk())

.andExpect(jsonPath("$[0].title").value("Book One"))

.andExpect(jsonPath("$[1].title").value("Book Two"));

verify(bookService, times(1)).findAll();

}

@Test

public void testGetBookById() throws Exception {

when(bookService.findById(1L)).thenReturn(Optional.of(book1));

mockMvc.perform(get("/api/books/1")

.contentType(MediaType.APPLICATION\_JSON))

.andExpect(status().isOk())

.andExpect(jsonPath("$.title").value("Book One"));

verify(bookService, times(1)).findById(1L);

}

@Test

public void testGetBookById\_NotFound() throws Exception {

when(bookService.findById(1L)).thenReturn(Optional.empty());

mockMvc.perform(get("/api/books/1")

.contentType(MediaType.APPLICATION\_JSON))

.andExpect(status().isNotFound());

verify(bookService, times(1)).findById(1L);

}

@Test

public void testCreateBook() throws Exception {

when(bookService.save(any(Book.class))).thenReturn(book1);

mockMvc.perform(post("/api/books")

.contentType(MediaType.APPLICATION\_JSON)

.content("{\"title\":\"Book One\",\"author\":\"Author One\",\"price\":9.99}"))

.andExpect(status().isCreated())

.andExpect(jsonPath("$.title").value("Book One"));

verify(bookService, times(1)).save(any(Book.class));

}

@Test

public void testUpdateBook() throws Exception {

when(bookService.findById(1L)).thenReturn(Optional.of(book1));

when(bookService.save(any(Book.class))).thenReturn(book1);

mockMvc.perform(put("/api/books/1")

.contentType(MediaType.APPLICATION\_JSON)

.content("{\"title\":\"Updated Book\",\"author\":\"Updated Author\",\"price\":12.99}"))

.andExpect(status().isOk())

.andExpect(jsonPath("$.title").value("Updated Book"));

verify(bookService, times(1)).findById(1L);

verify(bookService, times(1)).save(any(Book.class));

}

@Test

public void testDeleteBook() throws Exception {

when(bookService.findById(1L)).thenReturn(Optional.of(book1));

mockMvc.perform(delete("/api/books/1")

.contentType(MediaType.APPLICATION\_JSON))

.andExpect(status().isNoContent());

verify(bookService, times(1)).findById(1L);

verify(bookService, times(1)).deleteById(1L);

}

@Test

public void testDeleteBook\_NotFound() throws Exception {

when(bookService.findById(1L)).thenReturn(Optional.empty());

mockMvc.perform(delete("/api/books/1")

.contentType(MediaType.APPLICATION\_JSON))

.andExpect(status().isNotFound());

verify(bookService, times(1)).findById(1L);

}

}

**Exercise 14: Online Bookstore - Integration Testing for REST Services**

**Business Scenario:**

Write integration tests for your bookstore's RESTful services.

Maven Dependencies:

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</dependency>

<dependency>

<groupId>com.h2database</groupId>

<artifactId>h2</artifactId>

<scope>test</scope>

</dependency>

Example Test Class Configuration:

import org.junit.jupiter.api.Test;

import org.junit.jupiter.api.extension.ExtendWith;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.boot.test.autoconfigure.web.servlet.AutoConfigureMockMvc;

import org.springframework.boot.test.context.SpringBootTest;

import org.springframework.test.context.junit.jupiter.SpringExtension;

import org.springframework.test.web.servlet.MockMvc;

import org.springframework.test.web.servlet.MvcResult;

import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.\*;

import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.\*;

@ExtendWith(SpringExtension.class)

@SpringBootTest // Loads the entire Spring context

@AutoConfigureMockMvc // Auto-configures MockMvc

public class BookControllerIntegrationTest {

@Autowired

private MockMvc mockMvc;

@Test

public void testGetAllBooks() throws Exception {

MvcResult result = mockMvc.perform(get("/api/books")

.contentType("application/json"))

.andExpect(status().isOk())

.andReturn();

// Additional assertions can be made based on the response

}

@Test

public void testCreateBook() throws Exception {

String bookJson = "{\"title\":\"New Book\",\"author\":\"New Author\",\"price\":19.99}";

mockMvc.perform(post("/api/books")

.contentType("application/json")

.content(bookJson))

.andExpect(status().isCreated())

.andExpect(jsonPath("$.title").value("New Book"))

.andExpect(jsonPath("$.author").value("New Author"))

.andExpect(jsonPath("$.price").value(19.99));

}

}

Example of End-to-End Test:

@Test

public void testGetBookById() throws Exception {

// First, create a new book

String bookJson = "{\"title\":\"Book One\",\"author\":\"Author One\",\"price\":9.99}";

mockMvc.perform(post("/api/books")

.contentType("application/json")

.content(bookJson))

.andExpect(status().isCreated());

// Then, retrieve the book by ID

mockMvc.perform(get("/api/books/1")

.contentType("application/json"))

.andExpect(status().isOk())

.andExpect(jsonPath("$.title").value("Book One"))

.andExpect(jsonPath("$.author").value("Author One"))

.andExpect(jsonPath("$.price").value(9.99));

}

Configure H2 Database:

spring.datasource.url=jdbc:h2:mem:testdb

spring.datasource.driverClassName=org.h2.Driver

spring.datasource.username=sa

spring.datasource.password=password

spring.jpa.database-platform=org.hibernate.dialect.H2Dialect

spring.h2.console.enabled=true

Example of a Test That Interacts with the Database:

@Test

public void testUpdateBook() throws Exception {

// Create a book to update later

String bookJson = "{\"title\":\"Original Title\",\"author\":\"Original Author\",\"price\":9.99}";

mockMvc.perform(post("/api/books")

.contentType("application/json")

.content(bookJson))

.andExpect(status().isCreated());

// Update the book

String updatedBookJson = "{\"title\":\"Updated Title\",\"author\":\"Updated Author\",\"price\":12.99}";

mockMvc.perform(put("/api/books/1")

.contentType("application/json")

.content(updatedBookJson))

.andExpect(status().isOk())

.andExpect(jsonPath("$.title").value("Updated Title"))

.andExpect(jsonPath("$.author").value("Updated Author"))

.andExpect(jsonPath("$.price").value(12.99));

}

**Scenario 15: Online Bookstore - API Documentation with Swagger**

**Business Scenario:**

Document your bookstore's REST APIs using Swagger and Springdoc.

Maven Dependencies:

<dependency>

<groupId>org.springdoc</groupId>

<artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>

<version>2.1.0</version> <!-- Use the latest version available -->

</dependency>

Example of Annotating REST Controller:

import io.swagger.v3.oas.annotations.Operation;

import io.swagger.v3.oas.annotations.media.Content;

import io.swagger.v3.oas.annotations.media.Schema;

import io.swagger.v3.oas.annotations.responses.ApiResponse;

import io.swagger.v3.oas.annotations.responses.ApiResponses;

import io.swagger.v3.oas.annotations.tags.Tag;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.annotation.\*;

import java.util.List;

@RestController

@RequestMapping("/api/books")

@Tag(name = "Book", description = "The Book API")

public class BookController {

@Autowired

private BookService bookService;

@Operation(summary = "Get all books", description = "Retrieve a list of all books")

@ApiResponses(value = {

@ApiResponse(responseCode = "200", description = "Found the books",

content = {@Content(mediaType = "application/json",

schema = @Schema(implementation = Book.class))}),

@ApiResponse(responseCode = "404", description = "Books not found", content = @Content)

})

@GetMapping

public ResponseEntity<List<Book>> getAllBooks() {

List<Book> books = bookService.findAll();

return ResponseEntity.ok(books);

}

@Operation(summary = "Get a book by ID", description = "Retrieve a book by its ID")

@ApiResponses(value = {

@ApiResponse(responseCode = "200", description = "Found the book",

content = {@Content(mediaType = "application/json",

schema = @Schema(implementation = Book.class))}),

@ApiResponse(responseCode = "404", description = "Book not found", content = @Content)

})

@GetMapping("/{id}")

public ResponseEntity<Book> getBookById(@PathVariable Long id) {

return bookService.findById(id)

.map(ResponseEntity::ok)

.orElse(ResponseEntity.notFound().build());

}

@Operation(summary = "Create a new book", description = "Add a new book to the collection")

@ApiResponses(value = {

@ApiResponse(responseCode = "201", description = "Book created successfully",

content = {@Content(mediaType = "application/json",

schema = @Schema(implementation = Book.class))}),

@ApiResponse(responseCode = "400", description = "Invalid input", content = @Content)

})

@PostMapping

public ResponseEntity<Book> createBook(@RequestBody Book book) {

Book createdBook = bookService.save(book);

return ResponseEntity.status(201).body(createdBook);

}

@Operation(summary = "Update a book", description = "Update an existing book")

@ApiResponses(value = {

@ApiResponse(responseCode = "200", description = "Book updated successfully",

content = {@Content(mediaType = "application/json",

schema = @Schema(implementation = Book.class))}),

@ApiResponse(responseCode = "404", description = "Book not found", content = @Content)

})

@PutMapping("/{id}")

public ResponseEntity<Book> updateBook(@PathVariable Long id, @RequestBody Book bookDetails) {

return bookService.findById(id)

.map(book -> {

book.setTitle(bookDetails.getTitle());

book.setAuthor(bookDetails.getAuthor());

book.setPrice(bookDetails.getPrice());

Book updatedBook = bookService.save(book);

return ResponseEntity.ok(updatedBook);

})

.orElse(ResponseEntity.notFound().build());

}

@Operation(summary = "Delete a book", description = "Delete a book by its ID")

@ApiResponses(value = {

@ApiResponse(responseCode = "204", description = "Book deleted successfully", content = @Content),

@ApiResponse(responseCode = "404", description = "Book not found", content = @Content)

})

@DeleteMapping("/{id}")

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

return bookService.findById(id).map(book -> {

bookService.deleteById(id);

return ResponseEntity.noContent().build();

}).orElse(ResponseEntity.notFound().build());

}

}

Generating and Review the API Documentation:

<http://localhost:8080/swagger-ui/index.html>

Review the API documentation to ensure that all endpoints are correctly documented, and descriptions are clear. Swagger UI allows you to interact with the API directly, making it easier to test the endpoints and see the expected behaviour.