Spring Web Security for REST APIs with JWT Authentication

Table of Contents

1. Introduction: Why JWT for REST APIs?

REST APIs are inherently **stateless**. Unlike traditional web applications that rely on server-side sessions to maintain user state, REST APIs should treat each request as independent. This is where JWTs shine:

- **Statelessness:** The server doesn't need to store session information. All necessary user information (like ID, roles, expiration) is contained within the token itself.
- **Scalability:** Easier to scale horizontally, as any server instance can validate a token without needing to access a shared session store.
- **Mobile & Microservices Friendly:** Tokens can be easily passed in HTTP headers (e.g., `Authorization: Bearer <token>`) between clients (mobile apps, SPAs) and various microservices.
- **Security:** Cryptographically signed to prevent tampering.
- **Decoupling:** Authentication can be handled by a dedicated service, and tokens can be validated by other services.

How it works (High-Level):

1. User sends credentials (username/password) to a login endpoint.
2. Server authenticates credentials, and if successful, generates a JWT.
3. The JWT is sent back to the client.
4. For subsequent requests, the client includes the JWT in the `Authorization` header.
5. Server intercepts the request, validates the JWT, extracts user information, and sets up the security context before the request reaches the controller.

2. Core Spring Security Concepts

Before diving into code, let's refresh some fundamental Spring Security concepts crucial for understanding our JWT setup:

- **`FilterChainProxy`:** The main Spring Security filter responsible for delegating to a chain of security filters (`SecurityFilterChain`). It's the entry point for all security operations.

- **`SecurityFilterChain`:** A collection of security filters applied to specific request matchers. In modern Spring Security, you define this as a `Bean`.
- **`SecurityContextHolder`:** A central place where the `SecurityContext` is stored. The `SecurityContext` contains the `Authentication` object of the currently authenticated user. By default, it uses a `ThreadLocal` for `MODE_THREADLOCAL`, meaning the security context is available throughout the request processing for that specific thread.
  - *Reference:* Spring Security Docs - SecurityContextHolder
- **`Authentication` Object:** Represents the currently authenticated principal (user). It holds:
  - `principal`: The user's identity (often a `UserDetails` object or just a username).
  - `credentials`: The user's password (typically null after authentication).
  - `authorities`: A collection of `GrantedAuthority` objects (roles/permissions).
  - `authenticated`: A boolean indicating if the principal is authenticated.
  - *Reference:* Spring Security Docs - The Authentication Interface
- **`AuthenticationManager`:** An interface that processes an `Authentication` request. Its primary method is `authenticate(Authentication authentication)`. It delegates to a chain of `AuthenticationProvider`s.
  - *Reference:* Spring Security Docs - AuthenticationManager
- **`AuthenticationProvider`:** Performs the actual authentication. For username/password, a `DaoAuthenticationProvider` is common, which uses a `UserDetailsService` and `PasswordEncoder`.
- **`UserDetailsService`:** An interface with a single method `loadUserByUsername(String username)` that retrieves user details (username, password, authorities) based on a username. It returns a `UserDetails` object.
  - *Reference:* Spring Security Docs - UserDetailsService
- **`UserDetails`:** An interface providing core user information. Spring Security uses it to represent a user in the system.
- **`PasswordEncoder`:** Used to encode (hash) passwords before storing them and verify submitted passwords during authentication. **Never store plain-text passwords.** `BCryptPasswordEncoder` is recommended.
  - *Reference:* Spring Security Docs - Password Storage
- **`GrantedAuthority`:** Represents a permission or role granted to the principal.
- **`SessionCreationPolicy.STATELESS`:** Crucial for REST APIs with JWT. It tells Spring Security *not* to create or use HTTP sessions, making your application truly stateless.
  - *Reference:* Spring Security Docs - Session Management
- **`AuthenticationEntryPoint`:** Handles what happens when an unauthenticated user tries to access a protected resource. For REST, it typically returns a 401 Unauthorized status.
  - *Reference:* Spring Security Docs - Authentication Entry Point
- **`AccessDeniedHandler`:** Handles what happens when an authenticated user tries to access a resource they don't have permission for (e.g., wrong role). For REST, it typically returns a 403 Forbidden status.

---

3. Setting Up Your Spring Boot Project

First, add the necessary dependencies to your `pom.xml` (Maven) or `build.gradle` (Gradle):

Maven (`pom.xml`):

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>3.2.5</version> <!-- Use a recent Spring Boot version -->
        <relativePath/> <!-- lookup parent from repository -->
    </parent>
    <groupId>com.example</groupId>
    <artifactId>spring-security-jwt-rest</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>spring-security-jwt-rest</name>
    <description>Demo project for Spring Security with JWT for REST
APIs</description>

    <properties>
        <java.version>17</java.version>
        <jjwt.version>0.12.5</jjwt.version> <!-- Use a recent JJWT version -->
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-security</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jpa</artifactId> <!-- If using
database -->
        </dependency>
        <dependency>
            <groupId>io.jsonwebtoken</groupId>
            <artifactId>jjwt-api</artifactId>
            <version>${jjwt.version}</version>
        </dependency>
        <dependency>
            <groupId>io.jsonwebtoken</groupId>
            <artifactId>jjwt-impl</artifactId>
            <version>${jjwt.version}</version>
        </dependency>
        <dependency>
            <groupId>io.jsonwebtoken</groupId>
            <artifactId>jjwt-jackson</artifactId>
            <version>${jjwt.version}</version>
        </dependency>
        <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
            <optional>true</optional>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.springframework.security</groupId>
            <artifactId>spring-security-test</artifactId>
            <scope>test</scope>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <configuration>
                    <excludes>
                        <exclude>
                            <groupId>org.projectlombok</groupId>
                            <artifactId>lombok</artifactId>
                        </exclude>
                    </excludes>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

---

4. Spring Security Configuration for REST & JWT

We'll define a configuration class that sets up the `SecurityFilterChain` bean. This is the modern way since `WebSecurityConfigurerAdapter` is deprecated.

```
// src/main/java/com/example/security/config/SecurityConfiguration.java
package com.example.security.config;

import com.example.security.jwt.JwtAuthenticationFilter;
import lombok.RequiredArgsConstructor;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.authentication.AuthenticationProvider;
import
```

```
org.springframework.security.config.annotation.method.configuration.EnableMethodSe
curity;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import
org.springframework.security.config.annotation.web.configuration.EnableWebSecurity
;
import
org.springframework.security.config.annotation.web.configurers.AbstractHttpConfigu
rer;
import org.springframework.security.config.http.SessionCreationPolicy;
import org.springframework.security.web.SecurityFilterChain;
import
org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilt
er;

@Configuration
@EnableWebSecurity // Enables Spring Security features
@EnableMethodSecurity // Enables @PreAuthorize, @PostAuthorize, @Secured, etc.
@RequiredArgsConstructor
public class SecurityConfiguration {

    private final JwtAuthenticationFilter jwtAuthFilter;
    private final AuthenticationProvider authenticationProvider; // Our custom
AuthProvider
    private final JwtAuthenticationEntryPoint jwtAuthEntryPoint; // Handles
unauthorized access

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws
Exception {
        http
            .csrf(AbstractHttpConfigurer::disable) // Disable CSRF for stateless
REST APIs
            .authorizeHttpRequests(authorize -> authorize
                .requestMatchers("/api/auth/**").permitAll() // Allow
unauthenticated access to authentication endpoints
                .anyRequest().authenticated() // All other requests require
authentication
            )
            .sessionManagement(session -> session
                .sessionCreationPolicy(SessionCreationPolicy.STATELESS) //
Crucial: No sessions will be created or used
            )
            .authenticationProvider(authenticationProvider) // Register our custom
AuthenticationProvider
            .addFilterBefore(jwtAuthFilter,
UsernamePasswordAuthenticationFilter.class) // Add our JWT filter before Spring's
default UsernamePasswordAuthenticationFilter
            .exceptionHandling(exceptions -> exceptions
                .authenticationEntryPoint(jwtAuthEntryPoint) // Handle
unauthenticated access
                // .accessDeniedHandler(accessDeniedHandler) // Optional: Handle
access denied for authenticated users
            );
        return http.build();
    }
}
```

Custom `ApplicationConfig` for `UserDetailsService`, `PasswordEncoder`, and `AuthenticationManager`:

```
// src/main/java/com/example/security/config/ApplicationConfig.java
package com.example.security.config;

import com.example.security.user.UserRepository;
import lombok.RequiredArgsConstructor;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.authentication.AuthenticationManager;
import org.springframework.security.authentication.AuthenticationProvider;
import org.springframework.security.authentication.dao.DaoAuthenticationProvider;
import
org.springframework.security.config.annotation.authentication.configuration.Authen
ticationConfiguration;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.core.userdetails.UsernameNotFoundException;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.crypto.password.PasswordEncoder;

@Configuration
@RequiredArgsConstructor
public class ApplicationConfig {

    private final UserRepository userRepository; // Assume you have a
UserRepository for your custom User entity

    @Bean
    public UserDetailsService userDetailsService() {
        return username -> userRepository.findByEmail(username) // Or
findByUsername if your User has a username field
                .orElseThrow(() -> new UsernameNotFoundException("User not
found"));
    }

    @Bean
    public AuthenticationProvider authenticationProvider() {
        DaoAuthenticationProvider authProvider = new DaoAuthenticationProvider();
        authProvider.setUserDetailsService(userDetailsService());
        authProvider.setPasswordEncoder(passwordEncoder());
        return authProvider;
    }

    @Bean
    public AuthenticationManager authenticationManager(AuthenticationConfiguration
config) throws Exception {
        return config.getAuthenticationManager();
```

```java
    }

    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }
}
```

User Entity & Repository (Example):

```java
// src/main/java/com/example/security/user/User.java
package com.example.security.user;

import jakarta.persistence.*;
import lombok.AllArgsConstructor;
import lombok.Builder;
import lombok.Data;
import lombok.NoArgsConstructor;
import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.authority.SimpleGrantedAuthority;
import org.springframework.security.core.userdetails.UserDetails;

import java.util.Collection;
import java.util.List;

@Data
@Builder
@NoArgsConstructor
@AllArgsConstructor
@Entity
@Table(name = "_user") // _user to avoid conflict with 'user' keyword in some DBs
public class User implements UserDetails {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;
    private String firstname;
    private String lastname;
    private String email;
    private String password;

    @Enumerated(EnumType.STRING)
    private Role role; // Enum for roles

    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        return List.of(new SimpleGrantedAuthority(role.name()));
    }

    @Override
    public String getUsername() {
        return email; // Using email as username
    }

    @Override
    public boolean isAccountNonExpired() {
        return true;
    }

    @Override
    public boolean isAccountNonLocked() {
        return true;
    }

    @Override
    public boolean isCredentialsNonExpired() {
        return true;
    }

    @Override
    public boolean isEnabled() {
        return true;
    }
}
```

```
// src/main/java/com/example/security/user/Role.java
package com.example.security.user;

public enum Role {
    USER,
    ADMIN
}
```

```
// src/main/java/com/example/security/user/UserRepository.java
package com.example.security.user;

import org.springframework.data.jpa.repository.JpaRepository;

import java.util.Optional;

public interface UserRepository extends JpaRepository<User, Integer> {
    Optional<User> findByEmail(String email);
}
```

---

5. JWT Service: Generation & Validation

This service will handle creating, parsing, and validating JWTs. We'll use the `io.jsonwebtoken` (JJWT) library.

```java
// src/main/java/com/example/security/jwt/JwtService.java
package com.example.security.jwt;

import io.jsonwebtoken.Claims;
import io.jsonwebtoken.Jwts;
import io.jsonwebtoken.SignatureAlgorithm;
import io.jsonwebtoken.io.Decoders;
import io.jsonwebtoken.security.Keys;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.stereotype.Service;

import java.security.Key;
import java.util.Date;
import java.util.HashMap;
import java.util.Map;
import java.util.function.Function;

@Service
public class JwtService {

    @Value("${application.security.jwt.secret-key}")
    private String secretKey; // Load from application.properties
    @Value("${application.security.jwt.expiration}")
    private long jwtExpiration; // Token validity in milliseconds
    @Value("${application.security.jwt.refresh-token.expiration}")
    private long refreshExpiration; // Refresh token validity

    // Extract username (subject) from token
    public String extractUsername(String token) {
        return extractClaim(token, Claims::getSubject);
    }

    // Extract a specific claim from token
    public <T> T extractClaim(String token, Function<Claims, T> claimsResolver) {
        final Claims claims = extractAllClaims(token);
        return claimsResolver.apply(claims);
    }

    // Generate token with extra claims
    public String generateToken(Map<String, Object> extraClaims, UserDetails
userDetails) {
        return buildToken(extraClaims, userDetails, jwtExpiration);
    }

    // Generate token without extra claims
    public String generateToken(UserDetails userDetails) {
        return generateToken(new HashMap<>(), userDetails);
    }

    // Generate refresh token
    public String generateRefreshToken(UserDetails userDetails) {
        return buildToken(new HashMap<>(), userDetails, refreshExpiration);
    }

    // Build the JWT
    private String buildToken(Map<String, Object> extraClaims, UserDetails userDetails, long expiration) {
        return Jwts
                .builder()
                .setClaims(extraClaims)
                .setSubject(userDetails.getUsername())
                .setIssuedAt(new Date(System.currentTimeMillis()))
                .setExpiration(new Date(System.currentTimeMillis() + expiration))
                .signWith(getSignInKey(), SignatureAlgorithm.HS256)
                .compact();
    }

    // Validate token against user details
    public boolean isTokenValid(String token, UserDetails userDetails) {
        final String username = extractUsername(token);
        return (username.equals(userDetails.getUsername())) && !isTokenExpired(token);
    }

    // Check if token is expired
    private boolean isTokenExpired(String token) {
        return extractExpiration(token).before(new Date());
    }

    // Extract expiration date from token
    private Date extractExpiration(String token) {
        return extractClaim(token, Claims::getExpiration);
    }

    // Extract all claims from token
    private Claims extractAllClaims(String token) {
        return Jwts
                .parserBuilder()
                .setSigningKey(getSignInKey())
                .build()
                .parseClaimsJws(token)
                .getBody();
    }

    // Get the signing key from the secret key string
    private Key getSignInKey() {
        byte[] keyBytes = Decoders.BASE64.decode(secretKey);
        return Keys.hmacShaKeyFor(keyBytes);
    }
}
```

`application.properties` (or `application.yml`):

```
# Generate a secret key (e.g., using:
System.out.println(io.jsonwebtoken.SignatureAlgorithm.HS256.key().build().encodeTo
String()))
# Or use a online tool to generate a random Base64 encoded string of sufficient
length (e.g., 256-bit or 32-byte)
application.security.jwt.secret-
key=404E635266556A586E3272357538782F413F4428472B4B6250645367566B5970
application.security.jwt.expiration=86400000 # 24 hours in milliseconds
application.security.jwt.refresh-token.expiration=604800000 # 7 days in
milliseconds
```

---

6. JWT Authentication Filter

This is a custom filter that will intercept every request, extract the JWT, validate it, and set the `Authentication` object in the `SecurityContextHolder`.

```
// src/main/java/com/example/security/jwt/JwtAuthenticationFilter.java
package com.example.security.jwt;

import jakarta.servlet.FilterChain;
import jakarta.servlet.ServletException;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import lombok.RequiredArgsConstructor;
import org.springframework.lang.NonNull;
import
org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
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 java.io.IOException;

@Component
@RequiredArgsConstructor
public class JwtAuthenticationFilter extends OncePerRequestFilter {

    private final JwtService jwtService;
    private final UserDetailsService userDetailsService;

    @Override
    protected void doFilterInternal(
            @NonNull HttpServletRequest request,
            @NonNull HttpServletResponse response,
            @NonNull FilterChain filterChain
    ) throws ServletException, IOException {
        final String authHeader = request.getHeader("Authorization");
        final String jwt;
        final String userEmail; // Or username

        // 1. Check if token exists and is in correct format
        if (authHeader == null || !authHeader.startsWith("Bearer ")) {
            filterChain.doFilter(request, response);
            return;
        }

        // 2. Extract JWT token
        jwt = authHeader.substring(7);
        userEmail = jwtService.extractUsername(jwt); // Extract user
email/username from JWT

        // 3. Validate token
        if (userEmail != null &&
SecurityContextHolder.getContext().getAuthentication() == null) {
            // User not yet authenticated in the current security context
            UserDetails userDetails =
this.userDetailsService.loadUserByUsername(userEmail);

            if (jwtService.isTokenValid(jwt, userDetails)) {
                // Token is valid, create an Authentication object
                UsernamePasswordAuthenticationToken authToken = new
UsernamePasswordAuthenticationToken(
                        userDetails,
                        null, // Credentials are not needed once authenticated via
JWT
                        userDetails.getAuthorities()
                );
                // Set authentication details (remote IP, session ID etc.)
                authToken.setDetails(
                        new WebAuthenticationDetailsSource().buildDetails(request)
                );
                // Set the Authentication object in the SecurityContextHolder
                // This marks the user as authenticated for the current request
                SecurityContextHolder.getContext().setAuthentication(authToken);
            }
        }
        filterChain.doFilter(request, response); // Continue with the filter chain
    }
}
```

---

7. Authentication Endpoint (Login)

This endpoint will handle user login, authenticate credentials using `AuthenticationManager`, and return a JWT.

```java
// src/main/java/com/example/security/auth/AuthenticationController.java
package com.example.security.auth;

import lombok.RequiredArgsConstructor;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/api/auth")
@RequiredArgsConstructor
public class AuthenticationController {

    private final AuthenticationService service;

    @PostMapping("/register")
    public ResponseEntity<AuthenticationResponse> register(
            @RequestBody RegisterRequest request
    ) {
        return ResponseEntity.ok(service.register(request));
    }

    @PostMapping("/authenticate")
    public ResponseEntity<AuthenticationResponse> authenticate(
            @RequestBody AuthenticationRequest request
    ) {
        return ResponseEntity.ok(service.authenticate(request));
    }
}
```

Request and Response DTOs:

```java
// src/main/java/com/example/security/auth/RegisterRequest.java
package com.example.security.auth;

import lombok.AllArgsConstructor;
import lombok.Builder;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@Builder
@AllArgsConstructor
@NoArgsConstructor
public class RegisterRequest {
    private String firstname;
    private String lastname;
    private String email;
    private String password;
}
```

```
// src/main/java/com/example/security/auth/AuthenticationRequest.java
package com.example.security.auth;

import lombok.AllArgsConstructor;
import lombok.Builder;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@Builder
@AllArgsConstructor
@NoArgsConstructor
public class AuthenticationRequest {
    private String email;
    private String password;
}
```

```
// src/main/java/com/example/security/auth/AuthenticationResponse.java
package com.example.security.auth;

import lombok.AllArgsConstructor;
import lombok.Builder;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@Builder
@AllArgsConstructor
@NoArgsConstructor
public class AuthenticationResponse {
    private String token;
    // Potentially add a refresh token here
}
```

Authentication Service:

```
// src/main/java/com/example/security/auth/AuthenticationService.java
package com.example.security.auth;

import com.example.security.jwt.JwtService;
import com.example.security.user.Role;
import com.example.security.user.User;
import com.example.security.user.UserRepository;
import lombok.RequiredArgsConstructor;
```

```java
import org.springframework.security.authentication.AuthenticationManager;
import
org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.stereotype.Service;

@Service
@RequiredArgsConstructor
public class AuthenticationService {

    private final UserRepository userRepository;
    private final PasswordEncoder passwordEncoder;
    private final JwtService jwtService;
    private final AuthenticationManager authenticationManager;

    public AuthenticationResponse register(RegisterRequest request) {
        var user = User.builder()
                .firstname(request.getFirstname())
                .lastname(request.getLastname())
                .email(request.getEmail())
                .password(passwordEncoder.encode(request.getPassword()))
                .role(Role.USER) // Default role for new users
                .build();
        userRepository.save(user);
        var jwtToken = jwtService.generateToken(user);
        return AuthenticationResponse.builder()
                .token(jwtToken)
                .build();
    }

    public AuthenticationResponse authenticate(AuthenticationRequest request) {
        authenticationManager.authenticate(
                new UsernamePasswordAuthenticationToken(
                        request.getEmail(),
                        request.getPassword()
                )
        );
        // If authentication successful, load user details and generate token
        var user = userRepository.findByEmail(request.getEmail())
                .orElseThrow(); // Handle exception properly in a real app (e.g.,
custom exception)
        var jwtToken = jwtService.generateToken(user);
        return AuthenticationResponse.builder()
                .token(jwtToken)
                .build();
    }
}
```

8. Authorization: Securing API Endpoints

Once a user is authenticated via JWT, Spring Security's authorization mechanisms can be used to control access to resources based on roles or permissions.

Using `@PreAuthorize`: With `@EnableMethodSecurity` in `SecurityConfiguration`, you can use expression-based access control.

```java
// src/main/java/com/example/security/demo/DemoController.java
package com.example.security.demo;

import org.springframework.http.ResponseEntity;
import org.springframework.security.access.prepost.PreAuthorize;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/api/demo")
public class DemoController {

    @GetMapping("/user-only")
    @PreAuthorize("hasRole('USER')") // Only users with the 'USER' role can access
    public ResponseEntity<String> userOnly() {
        return ResponseEntity.ok("Hello from a secured endpoint for USER!");
    }

    @GetMapping("/admin-only")
    @PreAuthorize("hasRole('ADMIN')") // Only users with the 'ADMIN' role can
access
    public ResponseEntity<String> adminOnly() {
        return ResponseEntity.ok("Hello from a secured endpoint for ADMIN!");
    }

    @GetMapping("/any-authenticated")
    @PreAuthorize("isAuthenticated()") // Any authenticated user can access
    public ResponseEntity<String> anyAuthenticated() {
        return ResponseEntity.ok("Hello from a secured endpoint for any
authenticated user!");
    }
}
```

Note: For `hasRole()`, Spring Security typically prefixes role names with `ROLE_` (e.g., `ROLE_USER`). However, if your `UserDetailsService` or `AuthenticationProvider` does not add this prefix, you might need to adjust your `hasRole` expressions or ensure your `GrantedAuthority` objects properly reflect `ROLE_USER`. In our `User` entity, we directly use the `Role` enum name, so `hasRole('USER')` works as Spring Security converts `USER` to `ROLE_USER` by default in expressions. If you stored "USER" directly as an authority string without the "ROLE_" prefix, you would use `hasAuthority('USER')`.

---

9. Error Handling for Authentication & Authorization

For REST APIs, you want to return appropriate HTTP status codes and JSON responses for security errors.

Authentication Entry Point (401 Unauthorized):

```
// src/main/java/com/example/security/jwt/JwtAuthenticationEntryPoint.java
package com.example.security.jwt;

import com.fasterxml.jackson.databind.ObjectMapper;
import jakarta.servlet.ServletException;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import org.springframework.http.MediaType;
import org.springframework.security.core.AuthenticationException;
import org.springframework.security.web.AuthenticationEntryPoint;
import org.springframework.stereotype.Component;

import java.io.IOException;
import java.util.HashMap;
import java.util.Map;

@Component
public class JwtAuthenticationEntryPoint implements AuthenticationEntryPoint {

    @Override
    public void commence(HttpServletRequest request, HttpServletResponse response,
AuthenticationException authException) throws IOException, ServletException {
        response.setStatus(HttpServletResponse.SC_UNAUTHORIZED);
        response.setContentType(MediaType.APPLICATION_JSON_VALUE);

        Map<String, Object> errorDetails = new HashMap<>();
        errorDetails.put("timestamp", new Date());
        errorDetails.put("status", HttpServletResponse.SC_UNAUTHORIZED);
        errorDetails.put("error", "Unauthorized");
        errorDetails.put("message", "Authentication required or invalid token: " +
authException.getMessage());
        errorDetails.put("path", request.getRequestURI());

        ObjectMapper mapper = new ObjectMapper();
        mapper.writeValue(response.getOutputStream(), errorDetails);
    }
}
```

Register this in `SecurityConfiguration` as `exceptionHandling().authenticationEntryPoint(jwtAuthEntryPoint)`.

Access Denied Handler (403 Forbidden - Optional but Recommended): This handles cases where an *authenticated* user tries to access a resource they don't have permission for.

```
// src/main/java/com/example/security/config/CustomAccessDeniedHandler.java
package com.example.security.config;
```

```java
import com.fasterxml.jackson.databind.ObjectMapper;
import jakarta.servlet.ServletException;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import org.springframework.http.MediaType;
import org.springframework.security.access.AccessDeniedException;
import org.springframework.security.web.access.AccessDeniedHandler;
import org.springframework.stereotype.Component;

import java.io.IOException;
import java.util.Date;
import java.util.HashMap;
import java.util.Map;

@Component
public class CustomAccessDeniedHandler implements AccessDeniedHandler {

    @Override
    public void handle(HttpServletRequest request, HttpServletResponse response,
AccessDeniedException accessDeniedException) throws IOException, ServletException
{
        response.setStatus(HttpServletResponse.SC_FORBIDDEN);
        response.setContentType(MediaType.APPLICATION_JSON_VALUE);

        Map<String, Object> errorDetails = new HashMap<>();
        errorDetails.put("timestamp", new Date());
        errorDetails.put("status", HttpServletResponse.SC_FORBIDDEN);
        errorDetails.put("error", "Forbidden");
        errorDetails.put("message", "Access Denied: You don't have permission to
access this resource.");
        errorDetails.put("path", request.getRequestURI());

        ObjectMapper mapper = new ObjectMapper();
        mapper.writeValue(response.getOutputStream(), errorDetails);
    }
}
```

You would register this in your `SecurityConfiguration` like this:

```java
// ... inside securityFilterChain method
            .exceptionHandling(exceptions -> exceptions
                .authenticationEntryPoint(jwtAuthEntryPoint)
                .accessDeniedHandler(accessDeniedHandler) // Add this line
            );
```

And inject `CustomAccessDeniedHandler` into `SecurityConfiguration`'s constructor.

---

10. Best Practices & Considerations

- **HTTPS (SSL/TLS):** Always use HTTPS in production. JWTs are signed, but they are base64 encoded, not encrypted. If intercepted over plain HTTP, anyone can read the claims. HTTPS ensures the token is encrypted in transit.
- **Secret Key Management:** The JWT secret key (`application.security.jwt.secret-key`) is critical. Do not hardcode it in production. Use environment variables, Spring Cloud Config, or a dedicated secret management service (e.g., HashiCorp Vault, AWS Secrets Manager).
- **Token Expiration:**
  - **Short Lifespan for Access Tokens:** Keep access tokens valid for a short period (e.g., 15 minutes to 24 hours). This limits the window of opportunity if a token is stolen.
  - **Refresh Tokens:** Implement refresh tokens for better user experience. When an access token expires, the client sends a longer-lived refresh token to a dedicated `/refresh-token` endpoint to obtain a new access token (and optionally a new refresh token). This allows users to stay logged in without re-entering credentials frequently.
- **Token Revocation:** JWTs are stateless, meaning once issued, they are valid until they expire. Revocation is not trivial.
  - **Short expiration:** The primary defense.
  - **Blacklisting:** For critical cases (e.g., user logs out from all devices, admin revokes access), you can maintain a server-side blacklist of revoked JWTs. Every incoming JWT would be checked against this list. This adds state, but can be necessary.
- **Store Tokens Securely on Client-Side:**
  - **Web Browsers (SPAs):** Storing in `HttpOnly` cookies is generally preferred to `localStorage` to mitigate XSS attacks. However, this reintroduces CSRF vulnerability (which we disabled for REST), so you'd need to consider CSRF protection for token delivery via cookies. Storing in `localStorage` is common but vulnerable to XSS; use robust XSS prevention.
  - **Mobile Apps:** Store in secure storage mechanisms (e.g., Android KeyStore, iOS Keychain).
- **Logging & Monitoring:** Log authentication attempts, token generation, and validation failures. Monitor for unusual patterns.
- **Claim Management:** Only include necessary and non-sensitive information in JWT claims. Anything sensitive should be fetched from a secure backend store after authentication.
- **Rate Limiting:** Protect your `/api/auth/authenticate` (login) endpoint against brute-force attacks by implementing rate limiting.
- **Password Hashing:** Always use strong, one-way hashing algorithms like BCrypt (as implemented) for passwords.
- **CORS (Cross-Origin Resource Sharing):** If your frontend is on a different domain/port, configure CORS properly in your Spring Boot application.
