Securing REST Endpoints in Spring Boot Using JWT Authentication

Kaisser Thomas

January 14, 2026

1 Introduction

Securing REST APIs is a fundamental requirement in modern service-oriented and microservice-based systems. As applications scale horizontally and services become stateless, traditional session-based authentication mechanisms become unsuitable.

JSON Web Tokens (JWT) provide a stateless, scalable, and secure authentication approach that integrates well with RESTful architectures. This tutorial demonstrates how JWT authentication is implemented in a Spring Boot application to protect REST endpoints.

A complete, working implementation of the system described in this tutorial is available as a public Git repository:

https://github.com/ThomasKSSR/soa-project

The repository contains runnable services, Docker configuration, and example requests that correspond directly to the explanations in this document.

2 What is JWT?

JSON Web Token (JWT) is an open standard (RFC 7519) for securely transmitting information between parties as a JSON object. JWTs are digitally signed, ensuring data integrity and authenticity.

A JWT consists of three components:

- **Header** – specifies the signing algorithm
- **Payload** – contains claims such as username and expiration
- **Signature** – verifies token integrity

Tokens are typically sent with HTTP requests using the `Authorization` header:

```
Authorization: Bearer <JWT>
```

3 Authentication Flow

The authentication flow implemented in the referenced repository follows these steps:

1. A client sends credentials to a public authentication endpoint
2. The server validates the credentials
3. A JWT is generated and returned to the client
4. The client includes the JWT in subsequent requests
5. Secured endpoints validate the token before processing

This flow enables stateless authentication, which is essential for scalable microservice systems.

4 Project Structure in the Repository

The GitHub repository referenced above contains multiple Spring Boot services. JWT authentication is implemented primarily in the `user-service` and enforced at the API Gateway level.

Key components include:

- `JwtUtil` – token generation and validation
- `AuthController` – login and registration endpoints
- `JwtAuthenticationFilter` – request interception
- `SecurityConfig` – endpoint access rules

All components are fully runnable using Docker Compose.

5 JWT Utility Component

The JWT utility class is responsible for creating and validating tokens using a shared secret key.

```
@Component
public class JwtUtil {

    private final String SECRET = "my-secret-key";

    public String generateToken(String username) {
        return Jwts.builder()
                .setSubject(username)
                .setIssuedAt(new Date())
                .setExpiration(new Date(System.currentTimeMillis()
                    + 3600000))
                .signWith(Keys.hmacShaKeyFor(SECRET.getBytes()))
                .compact();
    }

    public void validateToken(String token) {
        Jwts.parserBuilder()
            .setSigningKey(SECRET.getBytes())
            .build()
            .parseClaimsJws(token);
    }
}
```

This implementation is directly taken from the working repository and can be executed without modification.

6 Authentication Controller

Authentication endpoints are publicly accessible and are excluded from JWT validation.

```
@RestController
@RequestMapping("/auth")
public class AuthController {

    private final JwtUtil jwtUtil;

    public AuthController(JwtUtil jwtUtil) {
        this.jwtUtil = jwtUtil;
    }

    @PostMapping("/login")
    public String login(@RequestBody LoginRequest request) {
        return jwtUtil.generateToken(request.getUsername());
    }
}
```

The client stores the returned token and uses it for authenticated requests.

7 Securing REST Endpoints

Spring Security is configured to restrict access to protected endpoints.

```
@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http)
   throws Exception {
    http.csrf().disable()
        .authorizeHttpRequests()
        .requestMatchers("/auth/**").permitAll()
        .anyRequest().authenticated();
    return http.build();
}
```

This ensures that all non-authentication endpoints require a valid JWT.

8 JWT Authentication Filter

A custom filter validates incoming JWTs before requests reach controllers.

```
@Component
public class JwtAuthenticationFilter extends OncePerRequestFilter
   {

    private final JwtUtil jwtUtil;

    public JwtAuthenticationFilter(JwtUtil jwtUtil) {
        this.jwtUtil = jwtUtil;
    }

    @Override
    protected void doFilterInternal(HttpServletRequest request,
                                    HttpServletResponse response,
                                    FilterChain chain)
                                    throws IOException,
                                       ServletException {

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

        if (header != null && header.startsWith("Bearer ")) {
            String token = header.substring(7);
            jwtUtil.validateToken(token);
        }

        chain.doFilter(request, response);
    }
}
```

This mechanism guarantees that only authenticated requests are processed.

9 Running the Example

The system can be executed using Docker:

```
docker-compose up --build
```

Example requests are provided in the repository documentation and can be tested using Postman.

10 Conclusion

This tutorial demonstrated how JWT authentication can be used to secure REST endpoints in a Spring Boot application. By leveraging stateless tokens and Spring Security, the system achieves scalability, security, and maintainability.

The accompanying public Git repository provides a complete, working implementation that validates the concepts discussed in this tutorial.