06 - Implementing JWT Tokens

Table of Contents

Agenda:

- What is JWT (JSON Web Token)
- Why do we need JWT Tokens?
- Configuring JWT in Spring Security
- Adding JWT dependency (jjwt-api, jjwt-impl, jjwt-jackson)
- Understanding JWT Authentication EntryPoint, JWT Authentication Filter, and JWT Helper
- Adding Configuration for JWT

What is JWT (JSON Web Token)?

JWT (JSON Web Token) is a compact, URL-safe means of representing claims to be transferred between two parties. The claims in a JWT are encoded as a JSON object that is used as the payload of a JSON Web Signature (JWS) structure or as the plaintext of a JSON Web Encryption (JWE) structure, enabling the claims to be digitally signed or integrity protected with a Message Authentication Code (MAC) and/or encrypted.

A typical JWT is composed of three parts, separated by dots ('.'):

1. Header: Typically consists of two parts: the type of token (JWT) and the signing algorithm being used, such as HMAC SHA256 or RSA.

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

2. Payload: Contains the claims. This is the part of the token where you store the data you want to transmit. There are three types of claims: registered, public, and private claims.

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true
}
```

3. Signature: To create the signature part, you have to take the encoded header, the encoded payload, a secret, and the algorithm specified in the header, and sign that.

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  secret)
```

The output is three Base64-URL strings separated by dots that can be easily passed in HTML and HTTP environments, like so:

```
xxxxx.yyyyy.zzzzz
```

Why do we need JWT Tokens?

JWT tokens are useful for several reasons:

1. Stateless Authentication: JWTs allow for stateless authentication. This means that the server doesn't need to store session information. All the required information is stored in the token, which is sent with each request. This reduces the server's load and improves scalability.
2. Security: JWTs are signed and optionally encrypted, ensuring that the data they carry can be trusted and is not tampered with. The signature verifies the integrity of the claims contained within the token.
3. Interoperability: JWTs are self-contained and can be used across different domains and technologies. This makes them a great choice for microservices architectures and single sign-on (SSO) systems.
4. Performance: Since JWTs are compact and can be passed through URLs, POST parameters, or inside HTTP headers, they are ideal for use in HTTP-based environments.
5. Flexibility: JWTs can carry any kind of information, not just user authentication data. They can be used to exchange information securely between parties.
6. Ease of Use: Libraries for creating and verifying JWTs are available in many programming languages, making them easy to implement in various applications.

Hands-On

1. Download and set up the project - dependencies (Web, Lombok, Security)
2. Add JWT dependency (jjwt-api, jjwt-impl, jjwt-jackson)

```
<!-- https://mvnrepository.com/artifact/io.jsonwebtoken/jjwt-api -->
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-api</artifactId>
    <version>0.11.5</version>
</dependency>

<!-- https://mvnrepository.com/artifact/io.jsonwebtoken/jjwt-impl -->
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-impl</artifactId>
    <version>0.11.5</version>
    <scope>runtime</scope>
</dependency>

<!-- https://mvnrepository.com/artifact/io.jsonwebtoken/jjwt-jackson -->
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-jackson</artifactId>
    <version>0.11.5</version>
    <scope>runtime</scope>
</dependency>
```

3. Create `HomeController` in the `controllers` to add endpoints:

```
@RestController
@RequestMapping("/home")
public class HomeController {
    @GetMapping()
    public String usersPage(){
        return "Welcome to the Dashboard!";
    }
}
```

4. Create `AppConfig` in the `config` directory - for UserDetails

```
@Configuration
public class AppConfig {
    @Bean
    public UserDetailsService userDetailsService() {
        UserDetails userDetails = User.builder()
                .username("abc")
```

```
                .password(passwordEncoder().encode("abc123"))
                .roles("ADMIN")
                .build();
        return new InMemoryUserDetailsManager(userDetails);
    }

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

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

5. Add Template code -
   Under `security` directory:

`JwtAuthenticationEntryPoint`

```
@Component
public class JwtAuthenticationEntryPoint implements
AuthenticationEntryPoint {

    @Override
    public void commence(HttpServletRequest request, HttpServletResponse
response, AuthenticationException authException) throws IOException,
ServletException {
        response.setStatus(HttpServletResponse.SC_UNAUTHORIZED);
        PrintWriter writer = response.getWriter();
        writer.println("Access Denied !! " + authException.getMessage());
    }
}
```

`JwtHelper`

```
@Component
public class JwtHelper {

    //requirement :
    public static final long JWT_TOKEN_VALIDITY = 5 * 60 * 60;

    private final String secret =
```

```
"MEgCQQCwamEJZx3WqmutgfPhOJV+8OI+MkVbuulvZGKbv5ekLR0eXIfTRTjCT2ym7qDbWFM9V
jEzgwP9lhne85fHBeqTAgMBAAE=";

    //Retrieve username from JWT token
    public String getUsernameFromToken(String token) {
        return getClaimFromToken(token, Claims::getSubject);
    }

    //Retrieve expiration date from JWT token
    public Date getExpirationDateFromToken(String token) {
        return getClaimFromToken(token, Claims::getExpiration);
    }

    public <T> T getClaimFromToken(String token, Function<Claims, T>
claimsResolver) {
        final Claims claims = getAllClaimsFromToken(token);
        return claimsResolver.apply(claims);
    }

    //Retrieving any information from the token we will need the secret
key
    private Claims getAllClaimsFromToken(String token) {
        return
Jwts.parser().setSigningKey(secret).parseClaimsJws(token).getBody();
    }

    //Check if the token has expired
    private Boolean isTokenExpired(String token) {
        final Date expiration = getExpirationDateFromToken(token);
        return expiration.before(new Date());
    }

    //Generate token for user
    public String generateToken(UserDetails userDetails) {
        Map<String, Object> claims = new HashMap<>();
        return doGenerateToken(claims, userDetails.getUsername());
    }

    //While creating the token -
    //1. Define  claims of the token, like Issuer, Expiration, Subject,
and the ID
    //2. Sign the JWT using the HS512 algorithm and secret key.
    //3. According to JWS Compact Serialization
(https://tools.ietf.org/html/draft-ietf-jose-json-web-signature-
41#section-3.1) compaction of the JWT to a URL-safe string

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

        return Jwts
```

```
                        .builder()
                        .setClaims(claims)
                        .setSubject(subject)
                        .setIssuedAt(new Date(System.currentTimeMillis()))
                    .setExpiration(new Date(System.currentTimeMillis() +
JWT_TOKEN_VALIDITY * 1000))
                .signWith(SignatureAlgorithm.HS512, secret).compact();
    }

    //validate token
    public Boolean validateToken(String token, UserDetails userDetails) {
        final String username = getUsernameFromToken(token);
        return (username
                        .equals(userDetails
                                .getUsername()) &&
!isTokenExpired(token));
    }
}
```

6. Create `JwtRequest` and `JwtResponse` entities in `entities` directory

`JwtRequest`

```
public class JwtRequest {
    private String email;
    private String password;
}
```

`JwtResponse`

```
public class JwtResponse {
    private String jwtToken;
    private String username;
}
```

7. Add the `SecurityConfig` file in the `config` directory

`SecurityConfig`

```
@Configuration
public class SecurityConfig {
    @Autowired
    private JwtAuthenticationEntryPoint point;
    @Autowired
    private JwtAuthenticationFilter filter;

    @Bean
```

```
    public SecurityFilterChain securityFilterChain(HttpSecurity http)
throws Exception {

        http
                .csrf(csrf -> csrf.disable())
                .authorizeRequests()
                .requestMatchers("/auth/login").permitAll()
                .anyRequest()
                .authenticated()
                .and().exceptionHandling(ex ->
ex.authenticationEntryPoint(point))
                .sessionManagement(session ->
session.sessionCreationPolicy(SessionCreationPolicy.STATELESS));
        http.addFilterBefore(filter,
UsernamePasswordAuthenticationFilter.class);
        return http.build();
    }

}
```

8. Add `AuthController` in the `config`

`AuthController`

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

    private final Logger logger =
LoggerFactory.getLogger(AuthController.class);
    @Autowired
    private UserDetailsService userDetailsService;
    @Autowired
    private AuthenticationManager manager;
    @Autowired
    private JwtHelper helper;

    @PostMapping("/login")
    public ResponseEntity<JwtResponse> login(@RequestBody JwtRequest
request) {

        this.doAuthenticate(request.getEmail(), request.getPassword());


        UserDetails userDetails = userDetailsService
                                .loadUserByUsername(request
                                                .getEmail());
        String token = this.helper.generateToken(userDetails);
```

```
        JwtResponse response = JwtResponse
                        .builder()
                .jwtToken(token)
                .username(userDetails.getUsername()).build();

        return new ResponseEntity<>(response, HttpStatus.OK);
    }

    private void doAuthenticate(String email, String password) {

        UsernamePasswordAuthenticationToken authentication = new
UsernamePasswordAuthenticationToken(email, password);
        try {
            manager.authenticate(authentication);
        } catch (BadCredentialsException e) {
            throw new BadCredentialsException(" Invalid Username or
Password  !!");
        }
    }

    @ExceptionHandler(BadCredentialsException.class)
    public String exceptionHandler() {
        return "Credentials Invalid !!";
    }

}
```

9. Test your code using Postman