ChatGPT

# JWT Authentication in Spring Boot – Complete Guide

This guide shows **step-by-step** how to implement JWT-based authentication and authorization in Spring Boot. We cover login endpoint creation, JWT generation, role-based access control (RBAC), stateless session setup, token validation/filtering, Spring Security configuration, user storage (in-memory and MySQL), and refresh-token handling. All relevant classes, annotations, and configuration are explained in detail.

JWT (JSON Web Token) is a self-contained token format that encodes user identity and claims in a signed token. In a typical flow, a client submits credentials to a **/login** endpoint; if valid, the server returns a signed JWT access token (and a refresh token). The client then includes the access token in the `Authorization: Bearer <token>` header on subsequent requests. The server verifies the JWT on each request (without storing any session state) [1] [2]. Roles (like `ROLE_USER`, `ROLE_ADMIN`) embedded in the JWT enforce RBAC. We also handle refresh tokens so that short-lived JWTs can be renewed.

## 1. Project Setup & Maven Dependencies

First, create a Spring Boot project (for example via https://start.spring.io) and add the following dependencies in your `pom.xml`. These include Spring Web, Security, JPA (for MySQL), and the JJWT library for token handling:

```
<dependencies>
  <!-- Spring Web for REST controllers -->
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>

  <!-- Spring Security -->
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
  </dependency>

  <!-- Spring Data JPA (for MySQL) -->
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
  </dependency>
  <!-- MySQL Connector -->
  <dependency>
```

```
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
  </dependency>

  <!-- JWT library (JJWT) -->
  <dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-api</artifactId>
    <version>0.11.5</version>
  </dependency>
  <dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-impl</artifactId>
    <version>0.11.5</version>
    <scope>runtime</scope>
  </dependency>
  <dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-jackson</artifactId>
    <version>0.11.5</version>
    <scope>runtime</scope>
  </dependency>

  <!-- (Optional) For validation annotations -->
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-validation</artifactId>
  </dependency>

  <!-- (Optional) Lombok for boilerplate code -->
  <dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <optional>true</optional>
  </dependency>
</dependencies>
```

These dependencies provide: Spring MVC support (`spring-boot-starter-web`), security (`spring-boot-starter-security`), data JPA for database access, and the JJWT library (`io.jsonwebtoken`) to create/verify JWTs.

## 2. `application.properties` (Configuration)

Configure database connection, JPA, and JWT properties in `src/main/resources/application.properties` (or `application.yml`). For example, with MySQL:

```
spring.datasource.url=jdbc:mysql://localhost:3306/mydb?useSSL=false
spring.datasource.username=root
spring.datasource.password=yourpassword
spring.jpa.hibernate.ddl-auto=update
spring.jpa.database-platform=org.hibernate.dialect.MySQL8Dialect

# JWT secret and expiration times (in milliseconds)
jwt.secret=MySuperSecretKey12345
jwt.expirationMs=3600000        # 1 hour for access token
jwt.refreshExpirationMs=86400000 # 24 hours for refresh token
```

- **spring.datasource.***: JDBC URL, credentials for your MySQL database.
- **spring.jpa.hibernate.ddl-auto=update**: auto-create/update tables from entities (for development).
- **jwt.secret**: a strong secret key to sign JWTs (keep it safe!).
- **jwt.expirationMs**: how long an access token is valid.
- **jwt.refreshExpirationMs**: validity of the refresh token.

You can use environment variables or vaults to store secrets in production. In code we'll inject these values to generate and validate tokens.

## 3. Data Models (Entities & Repositories)

### 3.1. Role Entity and Enum

We define a **Role** entity to represent user roles (for RBAC). Each role has an ID and a name. It's common to prefix role names with `ROLE_` (Spring Security requires this when using `hasRole`, see below).

```
@Entity
@Table(name = "roles")
public class Role {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    @Enumerated(EnumType.STRING)
    @Column(length = 20, unique = true)
    private ERole name;  // ERole is an enum of role names

    // getters, setters, constructors
}
```

The `ERole` enum lists the valid role names:

```
public enum ERole {
    ROLE_USER,
    ROLE_ADMIN,
    ROLE_MODERATOR
}
```

Each user can have one or more roles (e.g. ROLE_USER, ROLE_ADMIN). We will store roles in the `roles` table and link users to roles.

We also need a repository to fetch roles by name:

```
@Repository
public interface RoleRepository extends JpaRepository<Role, Integer> {
    Optional<Role> findByName(ERole name);
}
```

### 3.2. User Entity

The **User** entity stores user credentials and their assigned roles:

```
@Entity
@Table(name = "users")
public class User {
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(length = 50, unique = true)
    private String username;

    @Column(length = 100, unique = true)
    private String email;

    private String password;

    @ManyToMany(fetch = FetchType.LAZY)
    @JoinTable(name = "user_roles",
        joinColumns = @JoinColumn(name = "user_id"),
        inverseJoinColumns = @JoinColumn(name = "role_id"))
    private Set<Role> roles = new HashSet<>();

    // getters, setters, constructors
}
```

Key points:

- `@ManyToMany` with a join table `user_roles` links users and roles.
- Passwords should be stored **hashed** (we will configure a `PasswordEncoder` for this).
- We typically omit including the password in responses (we mark it hidden with `@JsonIgnore` if returning User objects).

Repository for users:

```
@Repository
public interface UserRepository extends JpaRepository<User, Long> {
    Optional<User> findByUsername(String username);
    boolean existsByUsername(String username);
    boolean existsByEmail(String email);
}
```

We'll use `findByUsername()` during authentication and for assigning roles.

### 3.3. Refresh Token Entity

To support refresh tokens, create a **RefreshToken** entity. Each refresh token links to a user and has an expiration time:

```
@Entity
@Table(name = "refresh_tokens")
public class RefreshToken {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @ManyToOne
    @JoinColumn(name = "user_id", nullable = false)
    private User user;

    @Column(nullable = false, unique = true)
    private String token;          // the refresh token string

    @Column(nullable = false)
    private Instant expiryDate;

    // getters, setters
}
```

When issued, the `token` can be a random UUID string, and `expiryDate` is set to (now + refreshExpirationMs).

Repository for refresh tokens:

```
@Repository
public interface RefreshTokenRepository extends JpaRepository<RefreshToken,
Long> {
    Optional<RefreshToken> findByToken(String token);
    void deleteByUser(User user);
}
```

We will **save** a RefreshToken record when a user logs in, and look it up by the token string when refreshing. Expired tokens should be cleaned up or rejected.

## 4. Security Utility Classes

### 4.1. JWT Utility (`JwtUtils`)

Create a component `JwtUtils` to generate and validate JWTs. It reads the secret and expiration from properties. Example using JJWT library:

```
@Component
public class JwtUtils {

    @Value("${jwt.secret}")
    private String jwtSecret;
    @Value("${jwt.expirationMs}")
    private int jwtExpirationMs;

    /** Generate JWT from username. */
    public String generateToken(UserDetails userDetails) {
        return Jwts.builder()
            .setSubject(userDetails.getUsername())
            .setIssuedAt(new Date())
            .setExpiration(new Date(System.currentTimeMillis() +
jwtExpirationMs))
            .signWith(SignatureAlgorithm.HS256, jwtSecret)
            .compact();
    }

    /** Extract username (subject) from a valid token. */
    public String getUsernameFromToken(String token) {
        return Jwts.parser()
            .setSigningKey(jwtSecret)
            .parseClaimsJws(token)
            .getBody()
            .getSubject();
```

```
    }

    /** Validate token signature and expiration. */
    public boolean validateToken(String authToken) {
        try {
            Jwts.parser().setSigningKey(jwtSecret).parseClaimsJws(authToken);
            return true;
        } catch (JwtException | IllegalArgumentException e) {
            // invalid token
        }
        return false;
    }
}
```

- `generateToken`: signs a new JWT containing the username.
- `getUsernameFromToken`: parses the token and returns the subject (the username).
- `validateToken`: checks signature and expiry; returns `false` if invalid or expired.

With these methods, we can create tokens on login and check them on each request.

### 4.2. UserDetails and UserDetailsService

Spring Security uses `UserDetails` to represent authenticated users. We implement it for our `User`:

```
public class UserDetailsImpl implements UserDetails {
    private Long id;
    private String username;
    private String email;
    @JsonIgnore
    private String password;
    private Collection<? extends GrantedAuthority> authorities;

    // Constructor
    public UserDetailsImpl(Long id, String username, String email, String
password,
                           Collection<? extends GrantedAuthority> authorities) {
        this.id = id;
        this.username = username;
        this.email = email;
        this.password = password;
        this.authorities = authorities;
    }

    /** Build UserDetailsImpl from User entity. */
    public static UserDetailsImpl build(User user) {
        // Convert roles (Role entities) to GrantedAuthority list
```

```
        List<GrantedAuthority> auths = user.getRoles().stream()
            .map(role -> new SimpleGrantedAuthority(role.getName().name()))
            .collect(Collectors.toList());
        return new UserDetailsImpl(
            user.getId(),
            user.getUsername(),
            user.getEmail(),
            user.getPassword(),
            auths);
    }

    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        return authorities;
    }
    @Override public String getPassword() { return password; }
    @Override public String getUsername() { return username; }
    // The following fields are left true (not expired/locked)
    @Override public boolean isAccountNonExpired() { return true; }
    @Override public boolean isAccountNonLocked() { return true; }
    @Override public boolean isCredentialsNonExpired() { return true; }
    @Override public boolean isEnabled() { return true; }

    // equals, hashCode, getters for id/email omitted for brevity
}
```

When a user authenticates, Spring Security will use this to load user data. The `authorities` list holds the user's roles as `GrantedAuthority` objects. In this code, we map each `Role.name()` to a `SimpleGrantedAuthority`. Note: by convention `ROLE_` prefix is added automatically if you use `hasRole()` (see below) [3] [4].

Now implement `UserDetailsService` to load users by username:

```
@Service
public class UserDetailsServiceImpl implements UserDetailsService {
    @Autowired
    private UserRepository userRepository;

    /** Load user by username (for authentication). */
    @Override
    @Transactional
    public UserDetails loadUserByUsername(String username) throws
UsernameNotFoundException {
        User user = userRepository.findByUsername(username)
            .orElseThrow(() ->
                new UsernameNotFoundException("User Not Found with username: "
```

```
+ username));
        return UserDetailsImpl.build(user);
    }
}
```

Here, when `AuthenticationManager` needs to authenticate, it will call `loadUserByUsername()`. We fetch the `User` from the DB and convert to `UserDetailsImpl`. If no user is found, we throw `UsernameNotFoundException`. Spring Security then checks the password, etc.

## 5. Security Configuration

Spring Security must be configured to use JWT and be stateless. We create a configuration class and expose necessary beans. In Spring Boot 3 / Security 6, use `SecurityFilterChain` beans (the old `WebSecurityConfigurerAdapter` is deprecated).

```
@Configuration
@EnableWebSecurity    // Enables web security
@EnableMethodSecurity  // Enable @PreAuthorize, etc. (replaces deprecated
@EnableGlobalMethodSecurity)
public class SecurityConfig {

    @Autowired
    private UserDetailsServiceImpl userDetailsService;
    @Autowired
    private AuthEntryPointJwt unauthorizedHandler; // handles unauthorized
errors

    @Bean
    public JwtTokenFilter authenticationJwtTokenFilter() {
        return new JwtTokenFilter();
    }

    /** Expose AuthenticationManager bean. */
    @Bean
    public AuthenticationManager
authenticationManager(AuthenticationConfiguration authConfig) throws Exception {
        return authConfig.getAuthenticationManager();
    }

    /** Password encoder to hash passwords. */
    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }

    /** Configure HTTP security (endpoints, CORS/CSRF, session policy, filter
```

```
order). */
    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        http.cors().and().csrf().disable()  // We use JWT, so no CSRF
            .exceptionHandling().authenticationEntryPoint(unauthorizedHandler).and()
            .sessionManagement()
                .sessionCreationPolicy(SessionCreationPolicy.STATELESS) //
*Stateless* session; no HttpSession
            .and()
            .authorizeHttpRequests()
                .requestMatchers("/api/auth/**").permitAll()  // public auth
endpoints
                .requestMatchers("/api/test/user").hasRole("USER")  // only
ROLE_USER
                .requestMatchers("/api/test/admin").hasRole("ADMIN") // only
ROLE_ADMIN
                .anyRequest().authenticated();

        // Add JWT filter before the default
UsernamePasswordAuthenticationFilter
        http.addFilterBefore(authenticationJwtTokenFilter(),
UsernamePasswordAuthenticationFilter.class);

        return http.build();
    }

    /** Configure DAO-based authentication provider to use our
userDetailsService & passwordEncoder. */
    @Bean
    public DaoAuthenticationProvider authenticationProvider() {
        DaoAuthenticationProvider authProv = new DaoAuthenticationProvider();
        authProv.setUserDetailsService(userDetailsService);
        authProv.setPasswordEncoder(passwordEncoder());
        return authProv;
    }
}
```

Explanation of key points:

- `@EnableWebSecurity` marks this class as a security config [6].
- `@EnableMethodSecurity` (in Spring Security 6 / Boot 3, replacing `@EnableGlobalMethodSecurity`) allows annotations like `@PreAuthorize` on controllers [5].
- In `filterChain(HttpSecurity http)`:
- **CORS and CSRF:** We disable CSRF (`csrf().disable()`) since JWT tokens protect against CSRF differently, and we enable CORS as needed.

- **SessionManagement:** We set `SessionCreationPolicy.STATELESS`. This means Spring Security will **not** create or use an HTTP session; every request must be authenticated via token [1]. This makes the API *stateless*.
- **Endpoint security:** We permit public access to `/api/auth/**` (login, refresh, etc.), and require roles for others using `.hasRole("USER")` or `.hasRole("ADMIN")`. Note that `hasRole("USER")` checks for authority `ROLE_USER` (Spring adds the `ROLE_` prefix) [3]. Other endpoints require any authenticated user.
- **JWT Filter:** We register our custom `JwtTokenFilter` *before* `UsernamePasswordAuthenticationFilter`. This ensures that every request passes through our filter to check for a valid JWT (extracting user identity) before any secured endpoint logic [7].

We also define an `AuthEntryPointJwt` (not shown here) to send HTTP 401 if an unauthorized request is made (it implements `AuthenticationEntryPoint`). It's injected into exception handling.

Finally, we expose an `AuthenticationManager` bean so that we can use it in our login controller to authenticate credentials. We also configure a `DaoAuthenticationProvider` with our `UserDetailsService` and `BCryptPasswordEncoder` (so passwords will be compared using BCrypt hashing).

## 6. JWT Filter (`JwtTokenFilter`)

We need a filter that runs once per request, extracts the JWT from the `Authorization` header, validates it, and sets the authentication in the Spring context. Example:

```
@Component
public class JwtTokenFilter extends OncePerRequestFilter {

    @Autowired
    private JwtUtils jwtUtils;
    @Autowired
    private UserDetailsServiceImpl userDetailsService;

    @Override
    protected void doFilterInternal(HttpServletRequest request,
                                    HttpServletResponse response,
                                    FilterChain filterChain)
                                    throws ServletException, IOException {
        String header = request.getHeader("Authorization");
        String jwt = null;
        if (header != null && header.startsWith("Bearer ")) {
            jwt = header.substring(7);
        }

        if (jwt != null && jwtUtils.validateToken(jwt)) {
            String username = jwtUtils.getUsernameFromToken(jwt);
            // Load user associated with token
```

```
            UserDetails userDetails =
userDetailsService.loadUserByUsername(username);
            // Create auth token
            UsernamePasswordAuthenticationToken authToken =
                new UsernamePasswordAuthenticationToken(
                    userDetails, null, userDetails.getAuthorities());
            authToken.setDetails(new
WebAuthenticationDetailsSource().buildDetails(request));
            // Set authenticated user in context
            SecurityContextHolder.getContext().setAuthentication(authToken);
        }
        // Continue filter chain
        filterChain.doFilter(request, response);
    }
}
```

What this does:

1. **Get Authorization header:** It looks for a header named `Authorization`. If absent or not starting with `"Bearer "`, the filter just continues (`doFilter`).
2. **Extract token:** If present, remove `"Bearer "` prefix to get the raw JWT string.
3. **Validate token:** Using `jwtUtils.validateToken(jwt)`. If invalid (bad signature or expired), we skip authentication.
4. **Parse user info:** Extract username from token (`jwtUtils.getUsernameFromToken(jwt)`).
5. **Load UserDetails:** We call our `UserDetailsServiceImpl` to load the user by username.
6. **Create Authentication:** We build a `UsernamePasswordAuthenticationToken` with the user details and authorities. The constructor `new UsernamePasswordAuthenticationToken(principal, credentials, authorities)` creates an authenticated token (we pass `null` for credentials since we already trust this token).
7. **Set in context:** Call `SecurityContextHolder.getContext().setAuthentication(authToken)` so downstream Spring Security considers the user as authenticated for this request.

By placing this filter before `UsernamePasswordAuthenticationFilter`, we ensure all secured endpoints see the user as authenticated when a valid JWT is provided [8] [9]. This filter enforces **stateless** auth: no session is created or used; each request is checked independently.

# 7. Authentication Controller (Login & Refresh)

## 7.1. Login Endpoint

Create a REST controller `AuthController` (or similar) for login. This endpoint will:

- Receive a login request (username & password).
- Use `AuthenticationManager` to authenticate credentials.
- If authentication succeeds, generate an access token and a refresh token.

- Return tokens to the client (often in JSON).

Example:

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

    @Autowired
    AuthenticationManager authenticationManager;
    @Autowired
    JwtUtils jwtUtils;
    @Autowired
    RefreshTokenService refreshTokenService;

    @PostMapping("/login")
    public ResponseEntity<?> authenticateUser(@RequestBody LoginRequest
loginReq) {
        // Perform authentication
        Authentication authentication = authenticationManager.authenticate(
            new UsernamePasswordAuthenticationToken(
                loginReq.getUsername(), loginReq.getPassword()));
        SecurityContextHolder.getContext().setAuthentication(authentication);

        // Generate JWT token
        String jwt = jwtUtils.generateToken(authentication.getPrincipal());

        // Generate or retrieve refresh token
        UserDetailsImpl userDetails = (UserDetailsImpl)
authentication.getPrincipal();
        RefreshToken refreshToken =
refreshTokenService.createRefreshToken(userDetails.getId());

        // Collect roles for response
        List<String> roles = userDetails.getAuthorities().stream()
            .map(auth -> auth.getAuthority())
            .collect(Collectors.toList());

        // Return tokens and user info
        return ResponseEntity.ok(new JwtResponse(jwt, refreshToken.getToken(),
roles));
    }
}
```

**LoginRequest DTO:** This class simply holds the incoming username/password:

```
public class LoginRequest {
    private String username;
    private String password;
    // getters and setters
}
```

**Explanation:**

- We call `authenticationManager.authenticate(...)` with a `UsernamePasswordAuthenticationToken`. This effectively checks the credentials against our `UserDetailsServiceImpl` and `PasswordEncoder`. If invalid, a `BadCredentialsException` is thrown (Spring will return 401). Otherwise, it returns an `Authentication` object.
- We then generate a JWT access token. (Here we passed `authentication.getPrincipal()` to `generateToken`, which should be cast to `UserDetails` inside `JwtUtils` or adjusted; you may also pass `userDetails` directly.)
- We also create a refresh token via `refreshTokenService`. This typically generates a random token (e.g. UUID), sets an expiry, and saves it with the user.
- Finally, we return a JSON response containing the JWT and refresh token. The `JwtResponse` might look like:

```
public class JwtResponse {
    private String accessToken;
    private String refreshToken;
    private List<String> roles;
    private String tokenType = "Bearer";
    // constructor, getters
}
```

Clients will store both tokens. The access token is sent in `Authorization: Bearer <accessToken>` on protected requests. The refresh token is sent only to the refresh endpoint.

### 7.2. Refresh Token Endpoint

A refresh endpoint allows a client with an **expired** access token (but a valid refresh token) to obtain a new access token. We validate the refresh token and issue a new JWT. Example:

```
@PostMapping("/refresh")
public ResponseEntity<?> refreshToken(@RequestBody TokenRefreshRequest request)
{
    String requestToken = request.getRefreshToken();

    return refreshTokenService.findByToken(requestToken)
        .map(refreshTokenService::verifyExpiration)
        .map(RefreshToken::getUser)
```

```
        .map(user -> {
            String newAccessToken = jwtUtils.generateToken(user.getUsername());
            return ResponseEntity.ok(new TokenRefreshResponse(newAccessToken,
requestToken));
        })
        .orElseThrow(() -> new RuntimeException("Refresh token is not in
database!"));
}
```

**TokenRefreshRequest DTO:**

```
public class TokenRefreshRequest {
    private String refreshToken;
    // getters and setters
}
```

**TokenRefreshResponse DTO:**

```
public class TokenRefreshResponse {
    private String accessToken;
    private String refreshToken;
    private String tokenType = "Bearer";
    // constructor, getters
}
```

**How it works:**

- The client calls `POST /api/auth/refresh` with the JSON `{ "refreshToken": "<token>" }`.
- The service checks `findByToken(refreshToken)`. If not found, throw error.
- It verifies expiration: if expired, delete it and throw (so it cannot be reused) [10].
- If valid, generate a new access token for `user.getUsername()`.
- Return the new access token (and you can optionally return the same refresh token or issue a new one).

For example, on logout you would delete the refresh token so it can't be reused.

**Refresh Token Service:** We mentioned `refreshTokenService.createRefreshToken(userId)`. Implementation sketch:

```
@Service
public class RefreshTokenService {
    @Value("${jwt.refreshExpirationMs}")
    private Long refreshTokenDurationMs;
    @Autowired
```

```
    private RefreshTokenRepository refreshTokenRepo;
    @Autowired
    private UserRepository userRepository;

    /** Create and save a refresh token for a user. */
    public RefreshToken createRefreshToken(Long userId) {
        RefreshToken token = new RefreshToken();
        token.setUser(userRepository.findById(userId).get());
        token.setExpiryDate(Instant.now().plusMillis(refreshTokenDurationMs));
        token.setToken(UUID.randomUUID().toString());
        return refreshTokenRepo.save(token);
    }

    /** Find token by string. */
    public Optional<RefreshToken> findByToken(String token) {
        return refreshTokenRepo.findByToken(token);
    }

    /** Verify expiry: if expired, delete and throw. */
    public RefreshToken verifyExpiration(RefreshToken token) {
        if (token.getExpiryDate().isBefore(Instant.now())) {
            refreshTokenRepo.delete(token);
            throw new
RuntimeException("Refresh token expired. Please make a new login request");
        }
        return token;
    }
}
```

This service creates a new `RefreshToken` with a random UUID string, saves it, and handles expiration. When validating a token, it deletes it if expired. This matches the flow in recent tutorials [10] [11]: the user can present the refresh token after the access token expires, and the server will issue a new access token without re-authenticating the user [11] [10].

## 8. Role-Based Access Control (RBAC)

Roles are stored in the `roles` table and linked to users. We already saw in the `SecurityConfig` how to require roles for certain endpoints:

```
// Example in SecurityConfig.filterChain():
.authorizeHttpRequests()
    .requestMatchers("/api/test/user").hasRole("USER")
    .requestMatchers("/api/test/admin").hasRole("ADMIN")
    .anyRequest().authenticated();
```

We can also use method-level security. For example, in a controller method:

```
@PreAuthorize("hasRole('ADMIN')")
@GetMapping("/api/admin/data")
public ResponseEntity<String> adminData() {
    return ResponseEntity.ok("Admin content");
}
```

The `@PreAuthorize` annotation checks the current user's roles (Authorities). Note: `hasRole("ADMIN")` is equivalent to `hasAuthority("ROLE_ADMIN")` [3]. Spring by default adds the `ROLE_` prefix. So if your role is named `ROLE_ADMIN` in the database, you use `hasRole("ADMIN")`.

**Important terms explained:**

- `@EnableWebSecurity`: enables Spring Security's web security support and allows us to configure it [6].
- `AuthenticationManager`: the main entry point for Spring Security's authentication process. It takes an `Authentication` request (like username/password) and delegates it to appropriate providers [12] [13].
- `UsernamePasswordAuthenticationToken`: a Spring Security `Authentication` implementation used for simple username/password auth. We create one with credentials to attempt authentication [14] [2]. After authentication, Spring replaces it with an authenticated token (with `isAuthenticated=true`).
- **Stateless authentication:** means the server does not maintain any user session. With JWTs, each request carries all needed info (the token). We achieve this by disabling sessions (`SessionCreationPolicy.STATELESS`) and not using HTTP sessions or cookies [1].

## 9. Statelessness and Security Filters

By setting **stateless** in the config (`http.sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS)`), Spring Security will not create an `HttpSession`. Instead, each HTTP request must include a valid JWT. We also disable CSRF (cross-site request forgery protection) because we are not using cookies for authentication [1].

The filter chain is important. When a request arrives:

1. Our `JwtTokenFilter` runs (set before the standard filters). It checks the header for a valid JWT and sets the user in the `SecurityContext` if so.
2. Spring then invokes authorization checks (based on endpoint or `@PreAuthorize`).
3. If the user is not authenticated or not authorized, the `AuthEntryPointJwt` sends a 401 response.

This is the **JWT-based stateless authentication** flow outlined in Spring Security guides [1]. No session is stored server-side – if a client loses its token, they must log in again (or use a refresh token to get a new token).

## 10. In-Memory vs. MySQL User Storage

For **testing** or simple apps, you can use in-memory users. For example, in your security configuration, you could define a `UserDetailsService` bean with hardcoded users:

```
@Bean
public InMemoryUserDetailsManager inMemoryUserDetailsManager(PasswordEncoder
encoder) {
    UserDetails user = User.withUsername("user")
        .password(encoder.encode("pass"))
        .roles("USER")
        .build();
    UserDetails admin = User.withUsername("admin")
        .password(encoder.encode("pass"))
        .roles("ADMIN", "USER")
        .build();
    return new InMemoryUserDetailsManager(user, admin);
}
```

This registers two users in memory (passwords encoded via BCrypt). Use this instead of the JPA-based `UserDetailsServiceImpl` for quick tests. You would skip defining JPA entities in this case.

For **production/real** use, we rely on the JPA entities above. You'd typically:

- Create a database schema (the JPA `ddl-auto=update` can auto-generate tables). Example SQL for reference:

```
CREATE TABLE users (
  id BIGINT AUTO_INCREMENT PRIMARY KEY,
  username VARCHAR(50) UNIQUE NOT NULL,
  email VARCHAR(100) UNIQUE NOT NULL,
  password VARCHAR(255) NOT NULL
);

CREATE TABLE roles (
  id INT AUTO_INCREMENT PRIMARY KEY,
  name VARCHAR(20) UNIQUE NOT NULL
);

CREATE TABLE user_roles (
  user_id BIGINT NOT NULL,
  role_id INT NOT NULL,
  PRIMARY KEY(user_id, role_id),
  FOREIGN KEY(user_id) REFERENCES users(id),
  FOREIGN KEY(role_id) REFERENCES roles(id)
```

```
);

CREATE TABLE refresh_tokens (
  id BIGINT AUTO_INCREMENT PRIMARY KEY,
  user_id BIGINT NOT NULL,
  token VARCHAR(255) UNIQUE NOT NULL,
  expiry_date DATETIME NOT NULL,
  FOREIGN KEY (user_id) REFERENCES users(id) ON DELETE CASCADE
);
```

- Insert initial role data, e.g.:

```
INSERT INTO roles(name) VALUES('ROLE_USER'), ('ROLE_MODERATOR'),
('ROLE_ADMIN');
```

(This matches many examples [15] [16].)

- You can then create users (either via a signup endpoint or SQL). Assign roles by inserting into `user_roles`.

The JPA repositories handle loading users/roles. With this setup, your `UserDetailsServiceImpl` (backed by `UserRepository`) will fetch from MySQL.

## 11. Refresh Token Handling

As seen, refresh tokens allow renewing expired JWTs. In our implementation:

- **Issuing:** On login, we call `refreshTokenService.createRefreshToken(userId)`. This creates a random token (e.g. UUID string) stored in the `refresh_tokens` table with an expiry (e.g. 24 hours ahead).
- **Storing:** Each refresh token is linked to a user. We store both the token string and expiry date in the database.
- **Validation:** When a refresh request comes in, we look up the token (`findByToken`). If not found or expired (checked via `verifyExpiration`), we reject it [10].
- **Renewing:** If valid, we generate a new access JWT for that user. We may choose to issue a new refresh token or reuse the old one until it expires. In our example, we reused the same refresh token string.
- **Revocation:** To handle logout, you would delete the refresh token (e.g. `refreshTokenRepo.deleteByUser(user)`), so it cannot be used again.

As one source notes: *"Access tokens are short-lived for security reasons. Refresh tokens let us issue new access tokens without re-authentication."* [11]. Our implementation follows that pattern. For example, see the pseudocode in [24] that checks expiry and issues a new token if valid [10].

## 12. Summary

This completes a full JWT authentication setup in Spring Boot:

- **Dependencies**: Spring Boot security, web, JPA, JWT library.
- **Config**: `application.properties` for DB and JWT settings.
- **Entities**: `User`, `Role`, `RefreshToken` with JPA repositories.
- **Security Config**: Custom `SecurityFilterChain` with `@EnableWebSecurity` and stateless session.
- **JWT Utils**: Methods to create/parse tokens.
- **Filter**: `JwtTokenFilter` to validate token on each request.
- **UserDetailsService**: loads users from DB.
- **Controller**: `/api/auth/login` to authenticate and return tokens; `/api/auth/refresh` to renew access tokens.
- **Password Encoding**: Always use `PasswordEncoder` (e.g. BCrypt) for user passwords.
- **RBAC**: Use `hasRole("USER")` or `@PreAuthorize` to secure endpoints by role, remembering Spring's `ROLE_` prefix convention [3].

Throughout this flow, Spring Security remains **stateless**: no HTTP session is used, so each request must contain a valid JWT for authentication [1]. This is ideal for REST APIs.

**References:** The implementation above follows best practices from official and community sources [1] [2] [11] [7]. All classes and annotations are explained inline. Further details on Spring Security filters and authentication flow can be found in Spring's documentation and tutorials [7] [2].

---


[1] [3] [7] [8] [9] Spring Security JWT Tutorial | Toptal®
https://www.toptal.com/spring/spring-security-tutorial

[2] [13] Secure Authentication and Authorization with JWT in Spring Boot 3 and Spring Security 6: Step-by-Step Guide | Medium
https://medium.com/@truongbui95/jwt-authentication-and-authorization-with-spring-boot-3-and-spring-security-6-2f90f9337421

[4] [5] [6] Spring Boot Token based Authentication with Spring Security & JWT - BezKoder
https://www.bezkoder.com/spring-boot-jwt-authentication/

[10] [11] User Registration and JWT Authentication with Spring Boot 3: Part 3— Refresh Token & Logout | by Max Di Franco | Medium
https://medium.com/@max.difranco/user-registration-and-jwt-authentication-with-spring-boot-3-part-3-refresh-token-logout-ea0704f1b436

[12] [14] Spring Security Authentication Process | Geek Culture
https://medium.com/geekculture/spring-security-authentication-process-authentication-flow-behind-the-scenes-d56da63f04fa

[15] GitHub - bezkoder/spring-security-refresh-token-jwt: Spring Security Refresh Token using JWT in Spring Boot example with HttpOnly Cookie - Expire and Renew JWT Token
https://github.com/bezkoder/spring-security-refresh-token-jwt

16 GitHub - bezkoder/spring-boot-refresh-token-jwt: Spring Boot Refresh Token using JWT example - Expire and Renew JWT Token
https://github.com/bezkoder/spring-boot-refresh-token-jwt