**JWT Util:**

This `JwtUtil` class is a utility for working with \*\*JSON Web Tokens (JWT)\*\* in a Spring Boot application. It handles three main tasks:

1. \*\*Generating JWT tokens\*\*.

2. \*\*Extracting information (e.g., username) from a token\*\*.

3. \*\*Validating a token\*\* for authenticity and expiration.

Let’s break down the code line by line and explain its purpose, along with examples of how it works.

---

### \*\*Key Components of the Code\*\*

#### 1. \*\*Annotations and Fields\*\*

```java

@Component

public class JwtUtil {

```

- \*\*`@Component`\*\*: Marks this class as a Spring-managed component, allowing it to be autowired into other classes.

```java

@Value("${secret.key}")

private String SECRET\_KEY;

```

- \*\*`@Value("${secret.key}")`\*\*: Injects the value of the `secret.key` property from the `application.properties` or `application.yml` file into the `SECRET\_KEY` field.

- \*\*`SECRET\_KEY`\*\*: A secret key used to sign and verify JWT tokens. It should be a strong, random string.

```java

private final long EXPIRATION\_TIME = 86400000; // 1 day in milliseconds

```

- \*\*`EXPIRATION\_TIME`\*\*: Defines the expiration time for the token (1 day in this case).

---

#### 2. \*\*Key Generation\*\*

```java

private Key getSigningKey() {

return Keys.hmacShaKeyFor(SECRET\_KEY.getBytes());

}

```

- \*\*Purpose\*\*: Converts the `SECRET\_KEY` into a `Key` object that can be used for signing and verifying JWTs.

- \*\*`Keys.hmacShaKeyFor(...)`\*\*: Generates a key for HMAC-SHA256, which is the algorithm used to sign the token.

---

#### 3. \*\*Token Generation\*\*

```java

public String generateToken(String username, String role) {

return Jwts.builder()

.setSubject(username) // Set the subject (username)

.claim("role", role) // Add a custom claim for the role

.setIssuedAt(new Date()) // Set the token's issue time

.setExpiration(new Date(System.currentTimeMillis() + EXPIRATION\_TIME)) // Set expiration time

.signWith(getSigningKey(), SignatureAlgorithm.HS256) // Sign the token

.compact(); // Convert to a compact string

}

```

- \*\*Purpose\*\*: Generates a JWT token with the provided `username` and `role`.

- \*\*Steps\*\*:

1. \*\*`setSubject(username)`\*\*: Sets the subject of the token (usually the username or user ID).

2. \*\*`claim("role", role)`\*\*: Adds a custom claim for the user's role.

3. \*\*`setIssuedAt(new Date())`\*\*: Sets the token's issue time to the current time.

4. \*\*`setExpiration(...)`\*\*: Sets the token's expiration time (current time + 1 day).

5. \*\*`signWith(...)`\*\*: Signs the token using the HMAC-SHA256 algorithm and the secret key.

6. \*\*`compact()`\*\*: Converts the token into a compact, URL-safe string.

---

#### 4. \*\*Extracting Username from Token\*\*

```java

public String extractUserName(String token) {

return Jwts.parserBuilder()

.setSigningKey(getSigningKey()) // Set the signing key

.build()

.parseClaimsJws(token) // Parse the token

.getBody() // Get the claims (payload)

.getSubject(); // Extract the subject (username)

}

```

- \*\*Purpose\*\*: Extracts the username (subject) from a JWT token.

- \*\*Steps\*\*:

1. \*\*`setSigningKey(...)`\*\*: Sets the signing key used to verify the token.

2. \*\*`parseClaimsJws(token)`\*\*: Parses the token and verifies its signature.

3. \*\*`getBody()`\*\*: Retrieves the token's payload (claims).

4. \*\*`getSubject()`\*\*: Extracts the subject (username) from the payload.

---

#### 5. \*\*Token Validation\*\*

```java

public boolean validateToken(String token) {

try {

Jwts.parserBuilder()

.setSigningKey(getSigningKey()) // Set the signing key

.build()

.parseClaimsJws(token); // Parse and verify the token

return true; // Token is valid

} catch (JwtException e) {

return false; // Token is invalid

}

}

```

- \*\*Purpose\*\*: Validates a JWT token by checking its signature and expiration.

- \*\*Steps\*\*:

1. \*\*`setSigningKey(...)`\*\*: Sets the signing key used to verify the token.

2. \*\*`parseClaimsJws(token)`\*\*: Parses the token and verifies its signature and expiration.

3. \*\*`JwtException`\*\*: Catches exceptions like `ExpiredJwtException` or `SignatureException` if the token is invalid.

---

### \*\*Example Workflow\*\*

#### \*\*1. Generating a Token\*\*

- \*\*Input\*\*:

```java

String username = "john\_doe";

String role = "USER";

```

- \*\*Code\*\*:

```java

String token = jwtUtil.generateToken(username, role);

```

- \*\*Output\*\*:

```plaintext

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJqb2huX2RvZSIsInJvbGUiOiJVU0VSIiwiaWF0IjoxNTE2MjM5MDIyLCJleHAiOjE1MTYyMzkwMjJ9.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV\_adQssw5c

```

#### \*\*2. Extracting Username from Token\*\*

- \*\*Input\*\*:

```java

String token = "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...";

```

- \*\*Code\*\*:

```java

String username = jwtUtil.extractUserName(token);

```

- \*\*Output\*\*:

```plaintext

john\_doe

```

#### \*\*3. Validating a Token\*\*

- \*\*Input\*\*:

```java

String token = "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...";

```

- \*\*Code\*\*:

```java

boolean isValid = jwtUtil.validateToken(token);

```

- \*\*Output\*\*:

- If the token is valid: `true`

- If the token is expired or tampered with: `false`

---

### \*\*Key Points\*\*

1. \*\*Token Structure\*\*:

- A JWT token consists of three parts:

- \*\*Header\*\*: Contains the algorithm and token type.

- \*\*Payload\*\*: Contains claims (e.g., subject, role, issue time, expiration time).

- \*\*Signature\*\*: Ensures the token's integrity.

2. \*\*Secret Key\*\*:

- The `SECRET\_KEY` is critical for signing and verifying tokens. It must be kept secure and should not be hardcoded in production.

3. \*\*Expiration\*\*:

- Tokens have an expiration time to enhance security. After expiration, the token becomes invalid.

4. \*\*Custom Claims\*\*:

- You can add custom claims (e.g., `role`) to the token payload to include additional information.

5. \*\*Error Handling\*\*:

- The `validateToken` method catches exceptions like `ExpiredJwtException` and `SignatureException` to determine if the token is invalid.

---

### \*\*Why This Code Is Important\*\*

- \*\*Authentication\*\*: JWTs are commonly used for stateless authentication in modern web applications.

- \*\*Security\*\*: By signing tokens with a secret key, the application ensures that tokens cannot be tampered with.

- \*\*Flexibility\*\*: Custom claims (e.g., `role`) allow the application to include additional user information in the token.

---

### \*\*Summary\*\*

| \*\*Method\*\* | \*\*Input\*\* | \*\*Output\*\* | \*\*Purpose\*\* |

|--------------------------|-------------------------------|-------------------------------------|--------------------------------------------------|

| `generateToken` | `username`, `role` | JWT token | Generates a signed JWT token with claims. |

| `extractUserName` | JWT token | `username` | Extracts the username from the token. |

| `validateToken` | JWT token | `true` or `false` | Validates the token's signature and expiration. |

This utility class is essential for implementing JWT-based authentication in your Spring Boot application. It handles token generation, extraction, and validation, ensuring secure and efficient user authentication.