🔐JWT Authentication – Line-by-Line Technical Documentation (Spring Boot)

Source Code Reference
Repository: `SpringBootLearn`
Commit: `c566d61c5b3d3520bb7a76dd8b48fa8fb6014e28`

Goal of this document
- Deep technical understanding of JWT implementation
- Line-by-line explanation for interviews & revision
- Clear WHY + WHAT + FLOW for each class
- PDF-ready structured notes

Overall JWT Architecture (Big Picture)

JWT authentication in this project follows **stateless security** using Spring Security.

High-level Flow

```
Client → /authenticate → JWT generated
Client → Authorization: Bearer TOKEN
↓
JwtAuthenticationFilter
↓
JwtService (validate token)
↓
SecurityContextHolder
↓
Controller Access
```

✂️JwtService (JwtUtil / JwtService)

🕰️Purpose

This class is responsible for **creating, reading, and validating JWT tokens**. It acts as the **core JWT engine** of the application.

Without this class: - No token generation - No token validation - JWT authentication cannot work

---

🕰️Key Responsibilities

- Generate JWT token
- Extract username from token
- Extract expiration
- Validate token authenticity

🕰️Important Code & Line-by-Line Explanation

🕐Secret Key

```
private static final String SECRET_KEY = "mysecretkey";
```

What it does: - Used to **sign** and **verify** JWT tokens

Why required: - Prevents token tampering

If removed: - Tokens can be forged

Best Practice: - Store in `application.properties` - Use strong, long key

🕐Token Generation

```
Jwts.builder()
    .setSubject(username)
    .setIssuedAt(new Date())
    .setExpiration(new Date(System.currentTimeMillis() + 1000 * 60 * 60))
    .signWith(SignatureAlgorithm.HS256, SECRET_KEY)
    .compact();
```

Line-by-line

- **Jwts.builder()** → Starts JWT creation
- **setSubject(username)** → Stores username inside token
- **setIssuedAt** → Token creation time
- **setExpiration** → Token expiry
- **signWith** → Digitally signs token
- **compact()** → Generates final token string

Definition:
- **Claims** → Information inside token
- **HS256** → HMAC SHA-256 signing algorithm

Why use expiration?
- Prevents infinite token misuse

🕰️Token Validation

```
extractUsername(token).equals(userDetails.getUsername()) && !
isTokenExpired(token)
```

What it checks: - Token belongs to correct user - Token is not expired

If not checked: - Expired or stolen tokens may work

🕰️Flow Diagram (JwtService)

```
Username → JwtService
↓
Claims created
↓
Token signed
↓
JWT returned
```

🕰️Best Practices

- Use environment variables for secret key
- Short expiry time
- Add Refresh Token support

✒️JwtAuthenticationFilter

🕰️Purpose

Intercepts **every incoming HTTP request** and validates JWT before reaching controller.

This is the **gatekeeper** of the application.

🕰️Why OncePerRequestFilter?

```
extends OncePerRequestFilter
```

Definition: - Ensures filter runs **only once per request**

Why important? - Prevents duplicate authentication

🕰️Core Logic Breakdown

🕐Read Authorization Header

```
String authHeader = request.getHeader("Authorization");
```

Purpose: - Extract JWT from request header

🕐Check Bearer Token

```
if(authHeader != null && authHeader.startsWith("Bearer "))
```

Why: - Standard JWT format

If missing: - Token not processed

🕐Set Authentication

```
SecurityContextHolder.getContext().setAuthentication(authToken);
```

What it does: - Marks user as authenticated

If not done: - Spring Security will reject request

🕰️Flow Diagram (Filter)

```
Request
↓
JWT extracted
↓
JwtService validates
↓
SecurityContext updated
↓
Controller
```

🕰️Best Practices

- Handle token exceptions gracefully
- Log invalid token attempts

🛌SecurityConfig

🕰️Purpose

Defines **security rules** for the application.

Controls: - Which APIs are public - Stateless behavior - JWT filter order

---

🕰️Stateless Configuration

```
sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS)
```

Definition: - No server-side session

Why required: - JWT is stateless

🕰️Disable CSRF

```
csrf().disable();
```

Why: - CSRF attacks apply to session-based auth - JWT uses header-based auth

---

🕰️Filter Order

```
.addFilterBefore(jwtFilter, UsernamePasswordAuthenticationFilter.class)
```

Why important: - JWT must be validated before username-password auth

---

🕰️Flow Diagram (SecurityConfig)

```
Request
↓
JWT Filter
↓
Security Rules
↓
Controller
```

🕰️Best Practices

- Define role-based rules
- Secure actuator endpoints

🌂CustomUserDetailsService

🕰️Purpose

Loads user data for authentication.

Spring Security **always** uses this class.

🕰️Why UserDetailsService?

Definition: - Standard Spring Security interface

Why required: - AuthenticationManager depends on it

🕰️Flow Diagram

```
Username
↓
UserDetailsService
↓
UserDetails
```

🕰️Best Practices

- Fetch users from DB
- Encrypt passwords

🌡️ AuthController

🕰️Purpose

Handles login request and generates JWT.

🕰️Authentication Logic

```
authenticationManager.authenticate(
 new UsernamePasswordAuthenticationToken(username, password)
);
```

What happens: - Username validated - Password matched

🕰️Flow Diagram

```
Login Request
↓
AuthenticationManager
↓
JWT Generated
```

```
↓
Response
```

🕰️Best Practices

- Never expose password
- Add login rate limiting

📌Final Interview Summary

- JWT is stateless authentication
- Filter validates token before controller
- SecurityContext stores authentication
- JwtService is token engine

✅This document is designed to be directly converted into a PDF for interview prep and deep revision.