JWT Authentication System - Implementation Guide

Project Overview

A Spring Boot application implementing JWT-based authentication with user registration and login functionality.

Step-by-Step Implementation Order

Step 1: Create User Entity

File: `User.java`

The foundation of the system - represents the user table in the database.

```
Properties:
- id (Long, auto-generated)
- username (String)
- password (String, will be encrypted)
```

Step 2: Create DTOs (Data Transfer Objects)

Request DTOs

File: `RegisterRequestDTO.java`

- Used when user registers
- Fields: username, password

File: `LoginRequestDTO.java`

- Used when user logs in
- Fields: username, password

Response DTOs

File: `RegisterResponseDTO.java`

- Response after successful registration
- Fields: id, username, message

File: `AuthResponseDTO.java`

- Response after successful login
- Fields: token (JWT)

Step 3: Create Mappers

File: `toEntity.java`

- Maps `RegisterRequestDTO` → `User` entity
- **Important:** Encrypts password using BCryptPasswordEncoder before saving

File: `toResponse.java`

- Maps `User` entity → `RegisterResponseDTO`
- Maps JWT token string → `AuthResponseDTO`

File: `UtilDTO.java` (Helper)

- Extracts username and password from `LoginRequestDTO`

Step 4: Create Repository

File: `UserRepo.java`

- Interface extending JpaRepository
- Method: `findByUsername(String username)` returns `Optional<User>`

Step 5: Create UserService

File: `UserService.java`

Implements: `UserDetailsService` (Spring Security interface)

Key Methods:

1. `registerUser(RegisterRequestDTO dto)` → `RegisterResponseDTO`
    - Checks if username exists
    - Maps DTO to User entity
    - Saves to database
    - Returns response DTO

2. `loadUserByUsername(String usernameOrEmail)` → `UserDetails`
    - **Crucial for authentication**
    - Called by Spring Security's AuthenticationManager
    - Fetches user from database
    - Returns Spring Security's UserDetails object with username, password, and authorities
3. Helper methods: `userExists()`, `getUserEntity()`, `saveUserDirect()`

Step 6: Create JwtUtil

File: `JwtUtil.java`

Independent utility class - does not depend on other files

Key Methods:

1. `generateToken(String username)` → `String`
    - Creates JWT token with username as subject
    - Sets expiration (24 hours)
    - Signs with HS256 algorithm
2. `extractUsername(String token)` → `String`
    - Extracts username from token claims
3. `validateToken(String token)` → `boolean`
    - Validates token signature and expiration
4. `getClaims(String token)` → `Claims` (private)
    - Parses and returns token claims

Configuration:

- SECRET: Signing key for JWT
- EXPIRATION: 86400000 ms (1 day)

Step 7: Create JwtAuthFilter

File: `JwtAuthFilter.java`

Extends: `OncePerRequestFilter`

Dependencies: UserService, JwtUtil

Purpose: Middleware to intercept every request and validate JWT

Flow:

1. Extracts "Authorization" header
2. Checks if header starts with "Bearer "
3. Extracts token (substring from index 7)
4. Validates token and extracts username
5. Loads UserDetails using UserService
6. Creates Authentication object
7. Sets authentication in SecurityContext
8. Passes request to next filter (like `next()` in Express.js)

Step 8: Create SecurityConfig

File: `SecurityConfig.java`

Annotations: `@Configuration`, `@EnableWebSecurity`

Dependencies: UserService, JwtUtil (to create JwtAuthFilter)

Beans Configured:

1. `SecurityFilterChain`
    - Disables CSRF (not needed for JWT APIs)
    - Sets session policy to STATELESS
    - Permits `/auth/**` endpoints (signup/login)
    - Requires authentication for all other requests
    - Adds JwtAuthFilter before UsernamePasswordAuthenticationFilter
2. `AuthenticationManager`
    - Handles authentication logic
    - Used by AuthService
3. `PasswordEncoder` (BCryptPasswordEncoder)
    - Used for encoding passwords during registration
    - Used for comparing passwords during login

Step 9: Create AuthService

File: `AuthService.java`

Dependencies: AuthenticationManager

Key Method:

`login(LoginRequestDTO loginDTO)` → `String`

1. Extracts username and password from DTO
2. Creates UsernamePasswordAuthenticationToken
3. Calls authenticationManager.authenticate()
    - **Internally calls** UserService.loadUserByUsername()
    - Compares provided password with stored encrypted password
4. Returns authenticated username

Step 10: Create AuthController

File: `AuthController.java`

Endpoint: `/auth`

Dependencies: AuthService, UserService, JwtUtil

Endpoints:

1. `POST /auth/signup`
    - Receives: `RegisterRequestDTO`
    - Calls: `userService.registerUser()`
    - Returns: `RegisterResponseDTO`
2. `POST /auth/login`
    - Receives: `LoginRequestDTO`
    - Calls: `authService.login()` (validates credentials)
    - Calls: `jwtUtil.generateToken()` (creates JWT)
    - Returns: `AuthResponseDTO` with token

Request-Response Flow

Registration Flow

```
1. Client sends POST /auth/signup
   Body: { "username": "john", "password": "secret123" }

2. AuthController.registerUser() receives RegisterRequestDTO

3. UserService.registerUser() is called
   ↓
4. Check if username exists in database
   ↓
5. toEntity.mapRegisterRequestDTOtoUser() converts DTO to User entity
   - Password is encrypted with BCryptPasswordEncoder
   ↓
6. UserRepo.save() saves user to database
   ↓
7. toResponse.mapToRegisterResponseDTO() creates response
   ↓
8. Return RegisterResponseDTO
   Response: { "id": 1, "username": "john", "message": "User registered successfully" }
```

Login Flow

```
1. Client sends POST /auth/login
   Body: { "username": "john", "password": "secret123" }

2. AuthController.login() receives LoginRequestDTO

3. AuthService.login() is called
   ↓
4. Extracts username and password from DTO
   ↓
5. Creates UsernamePasswordAuthenticationToken(username, password)
   ↓
6. AuthenticationManager.authenticate() is called
   ↓
7. Spring Security internally calls UserService.loadUserByUsername()
   ↓
8. UserRepo.findByUsername() fetches user from database
   ↓
9. Returns UserDetails object with encrypted password
   ↓
```

10. AuthenticationManager compares provided password with stored password
(using PasswordEncoder)
↓
11. If valid, authentication succeeds and returns username
↓
12. JwtUtil.generateToken(username) creates JWT token
↓
13. toResponse.mapToAuthResponseDTO() wraps token in DTO
↓
14. Return AuthResponseDTO
Response: { "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9..." }

Protected Endpoint Access Flow

1. Client sends request to protected endpoint
Header: Authorization: Bearer <jwt-token>

2. JwtAuthFilter.doFilterInternal() intercepts request
↓
3. Extracts "Authorization" header
↓
4. Checks if header starts with "Bearer "
↓
5. Extracts token (removes "Bearer " prefix)
↓
6. JwtUtil.extractUsername(token) gets username from token
↓
7. JwtUtil.validateToken(token) validates token signature
↓
8. UserService.loadUserByUsername(username) loads user details
↓
9. Creates UsernamePasswordAuthenticationToken with UserDetails
↓
10. Sets authentication in SecurityContextHolder
↓
11. Request proceeds to controller (filterChain.doFilter())
↓
12. Controller processes request with authenticated user context
↓
13. Returns response to client

File Dependencies Map

```
JwtauthApplication.java (Entry Point)
  ↓
SecurityConfig.java
  ├── JwtAuthFilter.java
  │   ├── UserService.java
  │   │   ├── UserRepo.java
  │   │   │   └── User.java
  │   │   └── toEntity.java, toResponse.java
  │   └── JwtUtil.java
  └── PasswordEncoder (Bean)

AuthController.java
  ├── AuthService.java
  │   └── AuthenticationManager (from SecurityConfig)
  │       └── UserService.java (for loadUserByUsername)
  ├── UserService.java
  └── JwtUtil.java

DTOs:
  ├── RegisterRequestDTO.java
  ├── RegisterResponseDTO.java
  ├── LoginRequestDTO.java
  └── AuthResponseDTO.java
```

Key Spring Security Concepts

Authentication Flow

1. **AuthenticationManager** coordinates authentication
2. **UserDetailsService** (UserService) loads user from database
3. **PasswordEncoder** verifies password
4. **SecurityContext** stores authenticated user

JWT Filter Chain

1. Request → JwtAuthFilter (validates token)
2. Sets authentication in SecurityContext
3. Request → Controller (with authenticated context)

Testing the Application

1. Register a User

bash

```
POST http://localhost:8080/auth/signup
Content-Type: application/json

{
 "username": "testuser",
 "password": "password123"
}
```

2. Login

bash

```
POST http://localhost:8080/auth/login
Content-Type: application/json

{
 "username": "testuser",
 "password": "password123"
}
```

3. Access Protected Endpoint

bash

```
GET http://localhost:8080/protected-endpoint
Authorization: Bearer <token-from-login-response>
```

Important Notes

1. **Password Encryption**: Passwords are encrypted in `toEntity.mapRegisterRequestDTOtoUser()` using BCryptPasswordEncoder
2. **Stateless Sessions**: Application uses JWT, so no server-side sessions (SessionCreationPolicy.STATELESS)
3. **Token Expiration**: Tokens expire after 24 hours (configurable in JwtUtil)
4. **Public Endpoints**: `/auth/**` endpoints are public (permitAll in SecurityConfig)
5. **Protected Endpoints**: All other endpoints require valid JWT token

6. **Filter Order**: JwtAuthFilter runs before UsernamePasswordAuthenticationFilter in the security chain