ChatGPT

Authentication Service Classes Analysis

JwtService.java

- **Purpose & Role:** This Spring `@Service` bean handles JSON Web Token (JWT) operations. It generates signed JWT access tokens and validates them during authentication. In the application flow, `JwtService` is called by controllers (e.g. signup/login endpoints) to create tokens and by security filters or controllers to validate tokens on incoming requests.
- **Key Methods:**
- `GenerateToken(String username)`: Public method that returns a new JWT for the given username. It creates an empty claims map and delegates to `createToken(...)` [1].
- `createToken(Map<String,Object> claims, String username)`: Private helper that builds the JWT. It sets the subject to the username, issues it at the current time, and sets expiration to 1 minute (1000*60*1 ms) in the future [2]. The token is signed with an HMAC-SHA256 key derived from the `SECRET`. For example, the code calls:

```
Jwts.builder()
    .setClaims(claims)
    .setSubject(username)
    .setIssuedAt(new Date(...))
    .setExpiration(new Date(... + 1000*60*1))
    .signWith(getSignKey(), SignatureAlgorithm.HS256)
    .compact();
```

  This produces a compact JWT string [2].
- `extractUsername(String token)` and `extractExpiration(String token)`: Utility methods to parse a token's claims. `extractUsername` returns the `sub` claim (subject/username), and `extractExpiration` returns the `exp` claim [3]. Internally they call `extractAllClaims(token)`.
- `extractClaim(String token, Function<Claims,T> claimsResolver)`: Generic method to parse all claims and apply a function. It calls `extractAllClaims(token)` then applies `claimsResolver` to the `Claims` object [4].
- `validateToken(String token, UserDetails userDetails)`: Checks if the token is valid for the given user. It extracts the username from the token and ensures it matches `userDetails.getUsername()`, and that the token is not expired [5]. It returns `true` only if both checks pass.
- **Token Logic & Security:**
- **Signing Key:** The `getSignKey()` method decodes a base64-encoded `SECRET` constant and creates an HMAC-SHA256 key [6]. This key is used for both signing and verification. The secret is hardcoded as `35763879...4629` (base64) [7].

- **Expiration:** Access tokens have a very short lifespan (1 minute). This forces clients to use refresh tokens frequently. The code for expiration is: `.setExpiration(new Date(System.currentTimeMillis()+1000*60*1))` [8].
- **Validation:** The `validateToken` method ensures the token is not expired by comparing `exp` to the current date (`!isTokenExpired(token)`) and that the username inside matches the expected user [5]. The `isTokenExpired` helper checks if `exp.before(new Date())` [9]. If the token is invalid or expired, validation will fail (return `false`). Note: JJWT parsing (`parseClaimsJws`) can also throw exceptions (e.g. `ExpiredJwtException`, `MalformedJwtException`) if the token is invalid; this code does not catch those, so such exceptions would bubble up if thrown during extraction.
- **Dependencies & Injection:** `JwtService` has no Spring-injected dependencies besides the signing key. It is annotated `@Service`, so it's a singleton bean. It relies on the static `SECRET` and the JJWT library (`io.jsonwebtoken`). There are no `@Autowired` fields in this class.
- **Exception / Edge Handling:** The service does not explicitly catch exceptions. If an invalid token string is passed, `extractAllClaims()` will throw a runtime exception from the JJWT parser. The `validateToken` method only returns a boolean; it does not throw on failure. Expired tokens simply cause `validateToken` to return `false`.
- **Controller & Data Interaction:** In practice, controllers (e.g. `AuthController`) call `GenerateToken(username)` when issuing new access tokens. For example, upon signup the controller does:

```
RefreshToken refreshToken =
refreshTokenService.createRefreshToken(userInfoDto.getUsername());
String jwtToken = jwtService.GenerateToken(userInfoDto.getUsername());
```

(see AuthController) [10]. The service itself does not interact with the database or other repositories; it only handles token string creation and parsing.

RefreshTokenService.java

- **Purpose & Role:** This `@Service` class manages long-lived refresh tokens stored in the database. When a user logs in or signs up, a new refresh token is created; later, when the user's short-lived JWT expires, the client can present a refresh token to obtain a new JWT. This service handles creating, looking up, and verifying refresh tokens.
- **Dependencies & Injection:** It injects two Spring Data repositories:
- `RefreshTokenRepository refreshTokenRepository` – for saving and querying `RefreshToken` entities.
- `UserRepository userRepository` – to look up the `UserInfo` (user account) by username. Both are `@Autowired` fields (which means Spring will auto-wire the repository implementations) [11].
- **Key Methods:**
- `createRefreshToken(String username)`: Generates a new refresh token for the given user. It looks up the `UserInfo` entity via `userRepository.findByUsername(username)` [12]. It then builds a `RefreshToken` entity with:
    - `userInfo` set to the found user,
    - `token` set to a random UUID string (`UUID.randomUUID().toString()`),

    - `expiryDate` set to 10 minutes in the future (`Instant.now().plusMillis(600000)`) [13].

      Finally it saves the entity with `refreshTokenRepository.save(...)` and returns it [14].
- `findByToken(String token)`: Retrieves a refresh token from the database by its string value. It returns an `Optional<RefreshToken>` by delegating to `refreshTokenRepository.findByToken(token)` [15].
- `verifyExpiration(RefreshToken token)`: Checks if a given `RefreshToken` has expired. It compares `token.getExpiryDate()` to the current time. If the token is expired (`expiryDate < now`), it deletes the token from the repository and throws a `RuntimeException` with a message indicating expiration [16]. If not expired, it simply returns the token.
- **Token Generation & Validation Logic:**
- **Generation:** When creating a token, this service assigns a fresh UUID as the token string and sets its expiry 600,000 milliseconds (10 minutes) ahead [13]. It uses a Lombok `@Builder` on the `RefreshToken` entity (not shown) to set fields fluently.
- **Storage:** The token is a persistent entity linking to `UserInfo`. The service uses `refreshTokenRepository` to save and query tokens in the database.
- **Validation:** The `verifyExpiration` method must be called during a refresh request to ensure the token is still valid. If expired, it removes the token and signals failure by throwing an exception [16]. The calling code (e.g. a controller handling refresh) should catch this exception to prompt the user to re-authenticate.
- **Exception / Edge Handling:**
- If the refresh token is expired, `verifyExpiration()` deletes it and throws a `RuntimeException` (with message like `"XYZ Refresh token is expired. Please make a new login..!"`) [16]. This is the only explicit exception thrown by this service.
- If a username does not exist in `createRefreshToken`, the code would get a `null` `userInfoExtracted`, which could lead to a null reference when building the token. (This edge is not handled here; it assumes the user exists.)
- If `findByToken` is called with a token that doesn't exist, it returns `Optional.empty()`. The service does not throw in that case; calling code should check the optional.
- **Controller & Data Interaction:**
- **Controller:** In signup or login controllers, after authenticating credentials, the code calls `refreshTokenService.createRefreshToken(username)` to issue a new refresh token [10]. In a token refresh endpoint, one would call `findByToken(tokenString)`, then `verifyExpiration(...)` on the returned token to check validity, and then issue a new JWT. (That refresh flow isn't shown here, but would logically follow.)
- **Data Layer:** Uses `refreshTokenRepository` (likely a `JpaRepository<RefreshToken, …>`) to persist and query refresh tokens in the database. It also uses `userRepository` (a `JpaRepository<UserInfo, …>`) to associate the token with the correct user.

UserDetailsServiceImpl.java

- **Purpose & Role:** This Spring bean implements `UserDetailsService` to bridge the application's user model with Spring Security. It handles loading user data by username (used during login authentication) and also provides a signup method to create new user accounts. This service is the application's user management component in the security flow.

- **Annotations & Injection:** It's annotated `@Component` and uses Lombok's `@AllArgsConstructor` and `@Data`, which together provide constructor injection for final fields [17]. The class has three injected fields:
- `UserRepository userRepository` – to query and save `UserInfo` entities (the data layer).
- `PasswordEncoder passwordEncoder` – to hash passwords before storing.
- `ValidateEmailPass validateEmailPass` – a utility for email/password format validation (note: it's injected here but not actually used inside this class; validation is done earlier in the controller).
- **Key Methods:**
- `loadUserByUsername(String username)`: As required by `UserDetailsService`, this loads a user's details by username. It uses `userRepository.findByUsername(username)` [18]. If the user is not found (`null`), it throws `UsernameNotFoundException` with a message [19]. Otherwise, it returns a new `CustomUserDetails` wrapping the `UserInfo` entity [20]. (The `CustomUserDetails` class is not shown here, but it implements `UserDetails` so Spring Security can read username, password, roles, etc.)
- `checkIfUserAlreadyExists(UserInfoDto userInfoDto)`: Helper method that looks up a user by username from the `UserInfoDto`. It simply calls `userRepository.findByUsername(...)` and returns the result [21].
- `signupUser(UserInfoDto userInfoDto)`: Creates a new user account from registration data. It first checks `checkIfUserAlreadyExists(...)`; if a user is found, it returns `false` indicating signup failed due to duplicate username [22]. Otherwise, it generates a random user ID (`UUID.randomUUID()`) [23]. Inside a `try` block it:
    - Encodes the password: `String hashedPassword = passwordEncoder.encode(userInfoDto.getPassword())` [24].
    - Saves a new `UserInfo` entity: `userRepository.save(new UserInfo(userId, userInfoDto.getUsername(), hashedPassword, new HashSet<>()))` [25]. (Here `new HashSet<>()` represents an empty set of roles or authorities.)
    - Prints a success message and returns `true` [26].
      If any `IllegalArgumentException` is thrown during encoding or saving, it is caught, logged, and the method returns `false` [27].
- **User Management Logic:**
- This service enforces unique usernames by checking the repository first. It handles password hashing so that the raw password is never stored. It does **not** assign any default roles (the code uses an empty `HashSet` of authorities).
- The `loadUserByUsername` method ensures Spring Security can authenticate users. It wraps the found `UserInfo` in `CustomUserDetails`, which presumably exposes the username, password, and granted authorities to the framework.
- **Exception / Edge Handling:**
- In `loadUserByUsername`, if the user is not found, a `UsernameNotFoundException` is thrown [28]. This is how Spring Security knows authentication failed.
- In `signupUser`, duplicate usernames cause an early `false` return (no exception). If password encoding or saving fails (caught as `IllegalArgumentException`), it prints a message and returns `false` [27]. Other exceptions (e.g. database errors) would not be caught here and would bubble up.
- **Controller & Data Interaction:**
- **Controller:** The `AuthController` calls `signupUser(...)` during the signup endpoint. For example:

```
Boolean isSignUped = userDetailsService.signupUser(userInfoDto);
if (!isSignUped) {
    // return 400 error
}
```

If signup is successful, the controller then creates tokens [29] [10]. The `loadUserByUsername` method is implicitly used by Spring Security during login (e.g. in an authentication filter or provider), although that login flow isn't shown here.

- **Data Layer:** The service directly uses `UserRepository` (a JPA repo) to query and save `UserInfo` records. It maps between the DTO (`UserInfoDto`) and the entity (`UserInfo`). It does not call any other external services.

Each service class thus encapsulates a specific part of the authentication flow: `JwtService` handles JWT tokens (creation and parsing), `RefreshTokenService` handles persistent refresh tokens (creation, lookup, expiration), and `UserDetailsServiceImpl` handles user data (authentication and registration). Controllers invoke these services for signup, login, and token refresh, and the services in turn use repositories (`UserRepository`, `RefreshTokenRepository`) to interact with the database. Together they enable a flow where a new user can register, obtain an access token and refresh token, and later use those tokens for authenticated requests or refreshing credentials.

Sources: The behavior and code structure above are directly inferred from the provided source files (e.g. `JwtService.java` [2], `RefreshTokenService.java` [14] [16], `UserDetailsServiceImpl.java` [18] [25]) and the `AuthController` usage (e.g. calls to these services [10]).

---

[1] [2] [3] [4] [5] [6] [7] [8] [9] JwtService.java
file://file-7qAKtMSs9FaQU8naA5zzuB

[10] [29] AuthController.java
file://file-KNBmR88xysyKmQYQnTVNbU

[11] [12] [13] [14] [15] [16] RefreshTokenService.java
file://file-1huPjv4A4p4ncdo4eGY3Tr

[17] [18] [19] [20] [21] [22] [23] [24] [25] [26] [27] [28] UserDetailsServiceImpl.java
file://file-7PD9jarZ1zZfn5JdYTvxnS