This is a \*\*Spring Boot-based authentication system\*\* with JWT (JSON Web Token) for secure login and access control. The system is built to handle user registration, login, and JWT-based authentication for accessing protected endpoints.

"The project follows a typical 3-layer architecture consisting of:

1. \*\*Controllers\*\* - Handle HTTP requests and responses.

2. \*\*Services\*\* - Contain the business logic and communicate with repositories.

3. \*\*Repositories\*\* - Provide database interaction using \*\*Spring Data JPA\*\*.

Additionally, we use \*\*Spring Security\*\* to manage authentication and authorization, and \*\*JWT\*\* for securing the user’s sessions.”

Controllers:

### \*\*Component Breakdown:\*\*

1. \*\*Controllers:\*\*

In my project, I’ve set up two main controllers — AuthenticationController and UserController — and both play a key role in securing and managing user access using JWT authentication.

The AuthenticationController is responsible for user registration and login. In the POST /auth/signup endpoint, users provide their name, email, and password. I encrypt the password using Spring Security’s PasswordEncoder before saving it to the database, so passwords are never stored in plain text.

Then there’s the POST /auth/login endpoint. Here, the user enters their credentials, and if they are valid, I generate a JWT token. This token is signed with a secret key and returned to the client. That token is then used for all future requests — it proves the user is authenticated, and we don’t need to store any sessions on the server.

The UserController handles authenticated and role-based access. The GET /users/me endpoint is used by logged-in users to get their own profile. I decode the JWT token from the request header, extract the username, and return the user’s details.

There’s also a GET /users endpoint, which lists all users in the system. But this one is protected with role-based access control — only users with an admin role can access it, even if they have a valid token. So I not only validate the JWT, but also check their role before allowing access.

I also added a GET /userprofile endpoint to demonstrate manual token validation. It extracts the JWT from the header, decodes it to get the username, and returns a welcome message. It’s a simple but effective example of using JWT to secure endpoints without storing any session state.

The /me endpoint returns the profile of the currently authenticated user. I use Spring Security’s SecurityContextHolderto access the current Authentication object, which holds the user’s details. From there, I extract the authenticated user using getPrincipal(), cast it to my custom User entity, and return it in the response.

Overall, this structure keeps the system stateless, secure, and scalable — and it's easy to extend with more features like refresh tokens or password resets if needed.

### \*\*Component Breakdown:\*\*

1. \*\*Controllers:\*\*

\* \*\*AuthenticationController\*\*: This controller exposes two main endpoints:

\* `POST /auth/signup`: he user provides their email, password, and full name. The password is encrypted using Spring Security’s `PasswordEncoder` before saving it to the database.

\* `POST /auth/login`: the user provides their credentials. If they are correct, a \*\*JWT token\*\* is generated using a secret key, and this token is returned to the user for subsequent requests.

\* \*\*UserController\*\*: This handles user-specific endpoints:

\* `GET /users/me`: Returns the authenticated user’s profile by decoding the JWT token in the request.

\* `GET /users`: Lists all users in the database. (Only accessible to admin users with valid tokens).

@GetMapping("/userprofile") endpoint is to access the resources with JWT-based authentication

2. \*\*Services:\*\*

\* \*\*AuthenticationService\*\*: Manages user registration (`signup`) and login (`authenticate`). It validates credentials and generates a JWT token on successful login.

\* \*\*JwtService\*\*: Handles JWT-related functionality, including token creation, extraction of claims, and validation. The service also checks whether the token has expired or is valid.

\* \*\*UserService\*\*: Provides business logic related to users. This includes retrieving all users and getting user details.

\* \*\*CustomUserDetailsService\*\*: This class implements `UserDetailsService` to load user details (like email and password) from the database when Spring Security needs them for authentication.

3. \*\*Configuration:\*\*

\* \*\*SecurityConfig\*\*: Configures Spring Security, including enabling JWT authentication via `JwtAuthenticationFilter`. This filter checks if the incoming request has a valid JWT token and authenticates the user accordingly.

\* \*\*JwtAuthenticationFilter\*\*: This filter is responsible for checking each incoming request for a valid JWT token in the `Authorization` header. If a valid token is found, the filter authenticates the user and populates the security context.

\* \*\*ApplicationConfiguration\*\*: Configures Spring Security-related beans, including `AuthenticationManager`, `PasswordEncoder`, and `UserDetailsService`.

4. \*\*Database:\*\*

\* \*\*User Entity\*\*: The user is represented by a `User` class annotated with `@Entity`, which maps to the `users` table in the database. The `User` object contains fields like `id`, `email`, `password`, `fullName`, and timestamps for when the user was created or updated.

\* \*\*UserRepository\*\*: A repository interface extending `JpaRepository`, which provides CRUD operations for interacting with the user data in the database.

### \*\*Security and JWT Implementation:\*\*

1. \*\*JWT Token\*\*:

\* The user provides email and password for login.

\* If the credentials are valid, the system generates a JWT token and sends it back to the user. This token contains claims such as the user's email, expiration time, and the secret key to validate the token.

\* This token is used for authenticating subsequent requests by including it in the `Authorization` header as a `Bearer` token.

2. \*\*Security Flow\*\*:

\* Upon login, the user receives the JWT token.

\* For every protected endpoint (like `/users/me`), the user must include the token in the `Authorization` header.

\* The `JwtAuthenticationFilter` checks the token on each request, extracting the username (email), validating the token, and loading user details via `CustomUserDetailsService`.

3. \*\*Password Security\*\*:

\* The user’s password is hashed using \*\*BCryptPasswordEncoder\*\* before being saved in the database.

\* During login, the password provided by the user is compared with the hashed password stored in the database.

### Why This Approach:

\* Using \*\*JWT\*\* ensures that the authentication system is stateless, meaning there’s no need to store session data on the server.

\* \*\*Spring Security\*\* ensures robust security for user authentication and authorization.

\* The system is scalable and can be easily extended with features like role-based access control, password resets, and more.

---

\*\*Interviewer\*\*: "That sounds interesting. How did you handle JWT expiration?"

\*\*You\*\*:

"Great question! The JWT token is issued with an expiration time, which is configured in the `application.properties` file. After the token expires, the user would need to log in again to get a new token. I handle this by checking the expiration of the token before granting access to protected endpoints. If the token is expired, access is denied."

---

\*\*Interviewer\*\*: "How did you test this project?"

\*\*You\*\*:

"I wrote both unit tests and integration tests. For unit tests, I mocked dependencies like the `UserRepository` to test the business logic in the `UserService` and `JwtService`. For integration tests, I tested the actual API endpoints to ensure that the signup, login, and profile retrieval work as expected. I also tested the JWT generation and validation logic to ensure secure access control.”

The /me endpoint returns the profile of the currently authenticated user. I use Spring Security’s SecurityContextHolderto access the current Authentication object, which holds the user’s details. From there, I extract the authenticated user using getPrincipal(), cast it to my custom User entity, and return it in the response.

Perfect — thanks for sharing! Your project structure is \*\*well-organized\*\* and aligns with modern Spring Boot practices for building a secure authentication system using \*\*JWT\*\*.

Let me break it down in a way you can \*\*explain confidently in an interview\*\* — especially if you're asked about how your app is structured or how different layers work together.

---

### ✅ \*\*Interview-Style Explanation of Your Project Structure\*\*

> In my Spring Boot project, I followed a clean layered architecture to separate concerns properly. Everything is under the base package `com.tericcabrel.authapi`, and I’ve organized it like this:

>

> \* \*\*controllers\*\*: This has the main REST endpoints. For example, `AuthenticationController` handles `/auth/signup` and `/auth/login`, while `UserController` handles user-specific routes like `/me`, `/users`, and `/userprofile`.

>

> \* \*\*dtos\*\*: I created DTO classes like `LoginUserDto` and `RegisterUserDto` to decouple the request/response models from the entity. This makes the API cleaner and avoids exposing sensitive fields like passwords.

>

> \* \*\*entities\*\*: The `User` class is the core entity, mapped to the database, and it implements `UserDetails` so it can integrate with Spring Security.

>

> \* \*\*exceptions\*\*: I use a `GlobalExceptionHandler` to catch and manage exceptions in a consistent way, returning meaningful HTTP error responses.

>

> \* \*\*configs\*\*: This package includes `SecurityConfiguration` for setting up Spring Security, `JwtAuthenticationFilter` to intercept and validate JWT tokens, and `ApplicationConfiguration` for general bean configurations.

>

> \* \*\*repositories\*\*: I use a `UserRepository` interface that extends `JpaRepository` to interact with the database.

>

> \* \*\*services\*\*: This is where the core logic lives. `AuthenticationService` handles signup and login, `JwtService` handles token creation and validation, `UserService` manages user operations, and `CustomUserDetailsService` is used by Spring Security to load user details during authentication.

>

> Overall, this structure keeps everything modular, testable, and easy to maintain. It also supports adding features like role-based access or refresh tokens without changing much of the core logic.

---

Would you like a visual diagram or a one-slide summary to present this in a project showcase or interview presentation?