JWT Authentication Backend:

Step 1: User Registration

When a new user registers on your application, their details like username, email, password, and role are stored in the database.

Step 2: Login Request

When a user attempts to log in, they send a POST request to the **/login** endpoint with their username and password in the request body.

Step 3: Authentication

The **AuthenticationController** receives the login request and passes the username and password to the **AuthenticationService**.

The **AuthenticationService** uses Spring Security's **AuthenticationManager** to authenticate the user by creating an **UsernamePasswordAuthenticationToken** and passing it to the **AuthenticationManager**.

The **AuthenticationManager** receives a **UsernamePasswordAuthenticationToken** with the user's credentials (username and password).

It then tries to authenticate the user by matching the provided credentials with the stored credentials in the database.

If the credentials are correct, authentication succeeds; otherwise, it throws an exception indicating authentication failure.

Step 4: User Retrieval

Once authentication is successful, the

repository.findByUsername(request.getUsername()) method is used to retrieve the user details from the database based on the provided username.

Step 5: JWT Token Generation

After successfully retrieving the user details, the **jwtService.generateToken(user)** method generates a JSON Web Token (JWT) for the authenticated user.

The JWT typically contains the user's username, expiration time, and possibly other relevant information.

The token is signed using a secret key known only to the server to ensure its authenticity.

Step 6: Sending Token to Client

The JWT token is returned to the client-side (frontend) as part of the authentication response.

An **AuthenticationResponse** object is created using the generated JWT, a success message indicating a successful login, the username of the user, and their role.

This response object is then returned from the **authenticate** method to the calling controller or service.

Step 7: Handling the Response

In your controller or service, when the **authenticate** method is called, it receives the **AuthenticationResponse**.

The controller or service can then use this response to send back the JWT token and any additional information to the client (frontend) as part of the login response.

The client can store this JWT token and include it in subsequent requests to access protected endpoints.

In summary, the **authenticate** method handles the user authentication process using Spring Security's **AuthenticationManager**. It generates a JWT token upon successful authentication, which is then returned in an **AuthenticationResponse** object for further handling and usage in your application.

Step 8: Subsequent Requests

For subsequent requests that require authentication (e.g., accessing protected endpoints), the client includes the JWT in the **Authorization** header of the HTTP request as **Bearer <token>**.

The **JwtAuthenticationFilter** intercepts these requests and extracts the JWT from the **Authorization** header.

Step 9: Token Validation

The **JwtAuthenticationFilter** validates the JWT by checking its signature and expiration time.

If the JWT is valid, it extracts the username from the token and loads the user details using the **UserDetailsService**.

Step 10: User Details Loading

The **UserDetailsService** (implemented as **UserDetailsServiceImpl**) fetches the user details from the database based on the username extracted from the JWT.

Step 11: Authorization

Once the user details are loaded, Spring Security performs authorization checks based on the user's role and the requested endpoint's required authorities.

If the user has the necessary authorities (roles), the request is allowed to proceed.

If not, a 403 Forbidden error is returned.

Process in SECURITY CONFIG:

The Java configuration class **SecurityConfig** is responsible for configuring the security settings for your Spring Boot application using Spring Security.

- **@Configuration**: Indicates that this class declares one or more **@Bean** methods and can be processed by the Spring container to generate bean definitions and service requests for those beans at runtime.
- **@EnableWebSecurity**: Enables Spring Security's web security features.
- **@EnableMethodSecurity**: Enables Spring Security's method-level security features.
- The constructor injects dependencies needed for security configuration: **UserDetailsServiceImpl** and **JwtAuthenticationFilter**.
- **Bean Definitions**:
  - **securityFilterChain**: Configures the security filter chain for URL-based access control, authentication, and authorization.
    - Disables CSRF protection.
    - Defines URL patterns and required authorities using lambda expressions.
    - Configures the **UserDetailsService** for authentication.
    - Sets session management to be stateless.
    - Adds the JWT authentication filter before the standard UsernamePasswordAuthenticationFilter.
    - Configures exception handling for access denied and unauthorized access.

- **passwordEncoder**: Defines a bean for password encoding using BCryptPasswordEncoder.
- **authenticationManager**: Configures an AuthenticationManager bean for handling authentication requests.
- **Session Management**: Configures Spring Security to use stateless sessions (**SessionCreationPolicy.STATELESS**), suitable for JWT-based authentication.
- **Exception Handling**: Configures how Spring Security should handle access denied and unauthorized access exceptions.

- If the JWT is valid and not expired, the server grants access to the requested resource based on the user's role and authorities.
- If the JWT is invalid or expired, the server denies access and sends an appropriate response (e.g., 401 Unauthorized).

•Overall, this configuration ensures that the application's endpoints are secured based on URL patterns and required authorities, handles authentication using JWT tokens, uses BCrypt for password encoding, and sets up proper exception handling for security-related issues.

Authentication Frontend:

Step 1: AuthProvider Component

The **AuthProvider** component is a context provider that wraps your entire application or a part of it where you need to manage user authentication.

It uses the **createContext** and **useState** hooks to create an authentication context and manage the authentication state.

The **auth** state contains information about the logged-in user (**user**) and their access token (**accessToken**).

By using **AuthContext.Provider**, it makes the **auth** state available to all child components that are wrapped inside it.

Step 2: RegistrationComponent

When the user sign up with respective details and the handleSubmit function is triggered. After validation completes, the userData object gets created with the entered details like username, password, email and role.

The **UserService.registerUser(userData)** function is called to send a POST request to the **/register** endpoint on your backend server with the user data.

Upon successful registration, the user is navigated to the home page (**"/"**) using **navigate("/")**.

Step 3: Login Component

When the user submits the login form, the **handleSubmit** function is triggered. In `handleSubmit`, a POST request is sent to the `/login` endpoint on your backend server using `UserService.userLogin({ username, password })`.

If the server responds with a valid access token (**response.data.token**), it means the user is authenticated.

The **setAuth** function from the authentication context is used to update the authentication state with the logged-in user's details (**user**, **accessToken**).

Depending on the user's role (**userRole**), the user is navigated to their respective dashboard using **navigate("/")**, **navigate("/admindashboard")**, etc.

Step 4: RequireAuth Component

This component is used to protect routes that require authentication, ensuring that only authenticated users can access them.

It uses the **useContext(AuthContext)** hook to access the authentication context and check if the user is authenticated (**auth.user** is not null).

auth?.user checks if the **user** property exists in the **auth** object. The **?.** operator is used for optional chaining to avoid errors if **auth** is null or undefined.

If **auth?.user** is truthy (meaning the user is authenticated), it renders **<Outlet />**, which is a placeholder for rendering child components or routes that require authentication.

If **auth?.user** is falsy (meaning the user is not authenticated), it triggers navigation to the login page (**"/login"**).

The **<Navigate />** component is used to perform navigation programmatically in React Router.

The **to="/login"** prop specifies the destination URL, which is the login page in this case.

The **state** prop is used to pass additional state information along with the navigation.

- **from: location** provides information about the current location (URL) the user is trying to access before being redirected to the login page. This can be useful for implementing redirect-back functionality after successful login.
- **accessToken: getAccessToken()** includes the access token in the state, although it's not common to pass sensitive information like tokens in URL state. It's safer to use cookies or localStorage for storing tokens.

The **replace** prop indicates that the navigation should replace the current entry in the history stack, which prevents the user from going back to the previous page after logging in.

Summary of RequireAuth:

We have an **AuthContext** created using **createContext()** in your **AuthProvider** component. This context provides a way to pass the authentication state and related functions throughout your application.

In the **RequireAuth** component, you're using the **useContext(AuthContext)** hook to access the authentication state (**auth**) stored in the context.

The **getAccessToken** function is a helper function defined within **RequireAuth**. It accesses the **auth** object and extracts the **accessToken** property using optional chaining (**auth?.accessToken || ""**).

If the **auth** object is not **null** or **undefined**, it returns the **accessToken**; otherwise, it returns an empty string (**""**).

In the **Navigate** component inside the **RequireAuth** component's return statement, you're passing the **state** prop with an object that includes **from: location** and **accessToken: getAccessToken()**.

Overall, we're storing the access token in the **auth** object within the **AuthContext** provided by **AuthProvider**. The **getAccessToken** function retrieves this token from the context when needed, such as when navigating to protected routes that require authentication.

The flow involves registering and logging in users through API requests, managing authentication state with the **AuthProvider** component, and protecting routes that require authentication using the **RequireAuth** component.