Full Stack Project: Online Student Feedback & Resolution System

Subject Code: 23CSP-339 **Student:** Ronak Jain (23BCS10225)

Module 1: User Authentication and Roles

1. Objective

The primary goal of this foundational module was to establish a robust and secure authentication system. This system is the core security layer for the entire application, responsible for identifying users and differentiating their roles as either `ROLE_STUDENT` or `ROLE_ADMIN`. This enables role-based access control (RBAC), ensuring users can only see and interact with the features permitted for their role.

2. Technologies Used

- **Backend:** Spring Boot, Spring Security, Spring Data MongoDB, JSON Web Tokens (JWT)
- **Frontend:** React, React Router, React Context API, Axios
- **Database:** MongoDB (Local Instance)
- **Testing:** Postman (for API testing), MongoDB Compass (for database verification)

3. Backend Implementation (Spring Boot)

The backend is the single source of truth for user identity and permissions.

3.1. Project Setup & Database

The project was generated using **Spring Initializr** with key dependencies: `Spring Web`, `Spring Security`, `Spring Data MongoDB`, `Validation`, and `Lombok`.

Technical Adaptation: The original plan proposed PostgreSQL and JPA. A strategic decision was made to pivot to **MongoDB** and `MongoRepository`. This provides greater flexibility and simplifies the data structure for future modules (e.g., embedding replies directly within a feedback document, as planned in Module 5).

The `application.properties` file was configured to connect to the local MongoDB instance at `mongodb://localhost:27017/feedback_system`.

3.2. Data Models (`model` package)

- **`Role.java`:** An `Enum` was created to strongly type the user roles, preventing invalid data. It contains `ROLE_STUDENT` and `ROLE_ADMIN`.
- **`User.java`:** This class is annotated with `@Document(collection = "users")` to map it to the `users` collection in MongoDB. It contains fields for `id` (String), `name`, `email` (annotated with `@Indexed(unique = true)`), `password`, `role`, and `createdAt`.

3.3. Repository (`repository` package)

- **`UserRepository.java`:** An interface that extends `MongoRepository<User, String>`. This provides all standard CRUD (Create, Read, Update, Delete) methods out of the box.
- **Custom Methods:** Two methods were added: `Optional<User> findByEmail(String email)` and `Boolean existsByEmail(String email)`. Spring Data MongoDB automatically implements these queries based on their names.

3.4. Security Layer (`security` & `config` packages)

A stateless JWT-based authentication system was built using several key components:

- **`PasswordEncoder`:** A `BCryptPasswordEncoder` bean was defined in `SecurityConfig.java`. All passwords are encrypted using this bean before being saved to the database.
- **`CustomUserDetailsService.java`:** This service acts as the bridge between Spring Security and our `UserRepository`. It implements the `loadUserByUsername` method (which we use to load by *email*), providing the security framework with the correct user details and roles from our database.
- **`JwtTokenProvider.java`:** A helper class responsible for:
    1. Generating a new JWT upon successful login.
    2. Validating incoming tokens on subsequent requests.
    3. Extracting the user's email from a valid token.
- **`JwtAuthenticationFilter.java`:** The "security guard" for the API. This filter (extending `OncePerRequestFilter`) runs on *every* request. It reads the JWT from the `Authorization: Bearer` header, validates it using `JwtTokenProvider`, and sets the user's `Authentication` in the `SecurityContextHolder`, making the user "logged in" for that request.
- **`SecurityConfig.java`:** The "rulebook" that ties everything together. It:
    1. Disables CSRF (not needed for a stateless JSON API).
    2. Enables **CORS** for `http://localhost:3000` (allowing our React frontend to connect).
    3. Sets the session management policy to `STATELESS`.
    4. Defines URL permissions:
        - `/api/auth/**` (Login/Register) are `permitAll()`.

        - `/api/admin/**` (Module 4/5) requires `ROLE_ADMIN`.
        - All other routes (e.g., `/api/feedback/**`) require `authenticated()`.
    5. Registers our `JwtAuthenticationFilter` in the filter chain.
    6. Defines the `AuthenticationProvider` and `AuthenticationManager` to ensure they use our `CustomUserDetailsService` and `BCryptPasswordEncoder`.

3.5. API Endpoints (`controller` & `dto` packages)

- **DTOs:** Created `LoginRequest.java`, `RegisterRequest.java`, and `AuthResponse.java` to define clean data structures for API communication.
- **`AuthController.java`:** Exposes the two public authentication endpoints:
    - `POST /api/auth/register`: Checks if the email is unique, encrypts the password, sets the role to `ROLE_STUDENT`, and saves the new `User` to MongoDB.
    - `POST /api/auth/login`: Uses the `AuthenticationManager` to validate the user's credentials. If successful, it generates a JWT and returns it (along with the user's name, email, and role) in an `AuthResponse` DTO.

4. Frontend Implementation (React)

The frontend is responsible for the user's login experience and for managing the session state.

4.1. Services (`services` package)

- **`AuthService.js`:** A dedicated service file created to handle all authentication-related API calls using **Axios**.
    - `login(email, password)`: Calls the backend login endpoint. On success, it saves the user's data (token, role, etc.) to `localStorage` to persist the session.
    - `register(...)`: Calls the backend register endpoint.
    - `logout()`: Removes the user's data from `localStorage`.
    - `getCurrentUser()`: Reads the user's data from `localStorage`.

4.2. Global State (`context` package)

- **`AuthContext.js`:** The global state manager for the entire application, as planned.
    - It uses React's `useContext` and `useState` hooks to store the `currentUser`.
    - A `useEffect` hook runs on app load, calling `AuthService.getCurrentUser()` to restore the session from `localStorage`.

    - It provides `login` and `logout` functions to all child components, which handle calling the `AuthService` and updating the global state.

4.3. UI Components (`components` package)

- **`Login.js` & `Register.js`:** Simple form components that use `useState` to manage form data. On submit, they call the `login` or `register` functions from the `AuthContext` to perform the action and handle any errors.
- **`StudentDashboard.js` & `AdminDashboard.js`:** Placeholder pages created to prove the routing works. They display the user's details from `AuthContext`.

4.4. Protected Routes

- **`ProtectedRoute.js`:** A critical component built to enforce role-based access on the frontend. This component:
    1. Checks the `AuthContext` for a `currentUser`. If none exists, it redirects to `/login`.
    2. If a `role` prop is passed (e.g., `<ProtectedRoute role="ROLE_ADMIN">...`), it checks the `currentUser.role`. If the roles do not match, it redirects the user away (e.g., to their own dashboard).
- **`App.js`:** The main router file. It uses `react-router-dom` to define all application routes, wrapping the student and admin dashboards in the `ProtectedRoute` component to secure them.