RBAC MERN Application Documentation

Overview

- The application implements **Role-Based Access Control (RBAC)** to securely manage user permissions.
- Built with the **MERN stack** enhanced by **NestJS** for the backend:
  - **MongoDB** for data persistence.
  - **Express** integrated within NestJS framework.
  - **React.js** for a dynamic frontend experience.
  - **Node.js** runtime environment.
- Supports three user roles:
  - **Super Admin:** Full control over users, roles, and permissions.
  - **Admin:** Limited control, able to manage certain features but restricted from full admin privileges.
  - **User:** Basic access to personal profile and minimal functionalities.
- Enforces strict access control on backend API endpoints and frontend UI based on roles.

Features

- **User Registration**
  - New users register with a form and are assigned the `user` role by default.
  - The system prevents role manipulation at registration for security.
- **Authentication & Authorization**
  - Uses **JWT tokens** to authenticate users and secure communication.
  - Automatically logs in users immediately after successful registration.
- **Role-Based Dashboards**
  - Each role sees a tailored dashboard UI with only the features they are authorized to access.
- **Backend Route Protection**
  - Uses **NestJS Guards** (JwtAuthGuard and RolesGuard) to protect routes based on authentication and user roles.

- **Dynamic Permissions**
  - Permissions are created and stored in the database.
  - Roles have assigned permissions that control access at API level.
- **Frontend Role-Aware Rendering**
  - React app dynamically renders components and routes based on the user's role decoded from JWT.
- **Secure Password Management**
  - Passwords are hashed securely using bcrypt before storing in the database.
- **Error Handling**
  - Unauthorized access attempts receive proper HTTP status codes (403 Forbidden).
  - Frontend shows relevant error messages and redirects unauthorized users.

Technology Stack

- **Frontend:**
  - React.js for UI.
  - React Router for SPA routing.
  - Axios for HTTP API calls.
  - React Context API for managing authentication state.
  - React Toastify for notifications.
- **Backend:**
  - NestJS framework for scalable, modular backend.
  - TypeScript for type safety and maintainability.
  - Passport.js with JWT strategy for authentication.
  - Mongoose ORM for MongoDB interaction.
- **Database:**
  - MongoDB Atlas or local instance as NoSQL database.
- **Security:**
  - bcrypt for password hashing.
  - JWT for token-based stateless authentication.

Application Architecture

Backend

- Organized into modules for separation of concerns:
  - **AuthModule:** Handles user registration, login, JWT token creation, and validation.
  - **UsersModule:** Manages user data and role assignments.
  - **RolesModule:** Defines roles and manages their permissions.
  - **PermissionsModule:** CRUD operations for permissions.
- **Guards and Middleware:**
  - `JwtAuthGuard`: Checks JWT tokens to confirm authentication.
  - `RolesGuard`: Validates user roles and permissions to authorize access.
- **DTOs and Validation:**
  - Data Transfer Objects ensure input validation and type safety.
- **Controllers:**
  - Define RESTful routes exposed to frontend, each protected appropriately.
- **Services:**
  - Contain business logic to handle user registration, login, role/permission management.

Frontend

- **Pages:**
  - `RegisterPage`: User signup with automatic role assignment and login.
  - `LoginPage`: User login with error handling.
  - `SuperAdminDashboard`: Full feature set for managing users, roles, and permissions.
  - `AdminDashboard`: Restricted access UI for admin-level users.
  - `UserDashboard`: Basic profile and settings UI for regular users.
- **Components:**
  - `Navbar`: Displays links dynamically based on user role.
  - `PrivateRoute`: Protects routes and redirects unauthorized users.
  - `Select`: Reusable dropdown component for selecting roles or permissions.
- **Context:**
  - `AuthContext`: Provides authentication state and user info across the app.
- **State Management:**

- Tokens and user data are stored and retrieved securely.
- Role-based conditional rendering enforces UI access restrictions.

Setup & Installation

- The project is split into two separate folders for backend and frontend.
- Environment variables are required for sensitive data like MongoDB URI and JWT secret.
- Backend exposes RESTful APIs secured with JWT and role-based guards.
- Frontend consumes backend APIs, stores tokens, and renders role-specific interfaces.
- Both backend and frontend can be started independently.

Usage Flow

1. **User Registration:**
    a. New users register through the frontend form.
    b. The backend assigns them the `user` role automatically.
    c. Upon successful registration, users are immediately logged in.
2. **User Login:**
    a. Registered users log in with email and password.
    b. Successful login returns a JWT token containing user role information.
3. **Access Role-Specific Dashboards:**
    a. JWT token decoded on frontend to determine the user's role.
    b. Based on the role, the corresponding dashboard and UI elements are displayed.
4. **Role & Permission Management:**
    a. Super Admins can create permissions and assign them to roles.
    b. Roles control what API endpoints a user can access.
5. **Route Protection:**
    a. Backend routes are guarded by NestJS JWT and role guards.
    b. Frontend routes are protected by PrivateRoute components that check user roles.
6. **Unauthorized Access Handling:**

a. Unauthorized API requests receive 403 responses.
b. Frontend redirects unauthorized users to login or shows error messages.

Role-Based Access Control (RBAC) Flow

- **JWT Token Content:** Includes user ID, email, and roles.
- **Backend Validation:** Guards validate token authenticity and verify role permissions before allowing access to endpoints.
- **Frontend Control:** UI components, pages, and routes render conditionally based on roles.
- **Permission Checks:** Granular permissions assigned to roles dictate precise access controls.
- **Security:** Prevents unauthorized data access and enforces least privilege.

Key Implementation Details

- **Default Role Assignment:**
  - When a user registers, the backend assigns the `user` role by default without allowing input override.
- **JWT Payload Structure:**
  - Payload contains minimal data needed for role verification.
- **NestJS Guards:**
  - Modular guards that check for authentication and authorization on protected routes.
- **Frontend Route Guards:**
  - Custom React components that restrict access to routes depending on user roles.
- **Permissions Management:**
  - Permissions are stored as entities and assigned to roles, which are then assigned to users.
- **Error & Notification Handling:**
  - Frontend displays toast notifications for success/failure.
  - Proper HTTP status codes used for API responses.

Future Enhancements

- Build UI tools to dynamically assign permissions to roles in the admin dashboard.
- Add refresh token mechanism to enhance session security and usability.
- Implement audit logging to track changes to roles, permissions, and user activities.
- Expand unit and integration testing coverage on both frontend and backend.
- Improve frontend responsiveness and accessibility for diverse devices.
- Implement email verification and password reset flows.