Project 5: Role-Based Access Control (RBAC)

Project 5: Role-Based Access Control (RBAC) **1**
Objective (Why?) 1
Core Requirements (Must-have) 2
Development Approach: Milestone-Based Progression 3
Milestone 1: Backend RBAC Implementation 3
Milestone 2: Frontend Integration & Conditional UI 4
Stretch Goals (Nice-to-have) 4
Evaluation Rubric (100 Points) 5

Objective (Why?)

Extend the complete authentication system from Project 4 with a robust authorization layer using Role-Based Access Control (RBAC). This project transitions from *who* can access the system (authentication) to *what* they can do within it (authorization). You will practice:

- Authorization Concepts: Implementing Role-Based Access Control (RBAC).
- Database Schema Extension: Modifying existing database models and managing migrations.

- Secure Backend Development: Creating role-protected API endpoints and embedding roles in JWTs.
- Conditional Frontend Rendering: Dynamically showing or hiding UI components based on user roles.

Core Requirements (Must-have)

Layer	Requirement
Backend	FastAPI + PostgreSQL • Add a role column to the users table (e.g., 'admin', 'user'). • Include the user's role in the JWT payload. • Create a protected API endpoint accessible only by 'admin' users. • Ensure regular 'user' roles cannot access the admin endpoint.
Frontend	React + Vite + Tailwind CSS • Update the authentication context to store and expose the user's role. • Conditionally render an "Admin Dashboard" link in the navigation, visible only to 'admin' users. • Create a placeholder page for the Admin Dashboard.
Database	PostgreSQL • Create and apply a new Alembic migration to add the role column. • Assign a default role to new users upon registration. • Provide a mechanism to manually update a user's role to 'admin' for testing.

Security	Authorization • Secure the admin endpoint using a FastAPI dependency that checks the role from the JWT. • Prevent unauthorized access attempts with proper HTTP status codes (403 Forbidden). • Ensure the JWT payload is the single source of truth for the user's role on the frontend.

Development Approach: Milestone-Based Progression

Philosophy: This is a focused, enhancement-based project. The goal is to carefully extend a production-ready system, focusing on precision and security. Each milestone must pass all relevant review templates.

Milestone 1: Backend RBAC Implementation

Deliverables:

- Database Migration: A new Alembic migration script to add a role column to the users table.
- Updated User Model: The SQLAlchemy User model and Pydantic User schema are updated to include the role.
- JWT Enhancement: The user's role is successfully embedded into the JWT access token upon login.
- Admin Seeding: A script or manual process to assign the 'admin' role to at least one user for testing.
- Role-Protected Endpoint: A new API endpoint (e.g., /api/admin/dashboard) that is protected and only accessible to users with the 'admin' role.

Review Requirements (Must Pass to Proceed):

- Security Review: JWT payload is secure; endpoint protection is robust; no unauthorized access is possible.

- Architecture Review: RBAC implementation is clean, scalable, and well-integrated.
- Code Quality Review: Code is clean, documented, and follows best practices.

Milestone 2: Frontend Integration & Conditional UI

Deliverables:

- Updated Auth Context: The React AuthContext is updated to decode the JWT and store the user's role.
- Conditional UI Element: An "Admin" link or button appears in the dashboard sidebar/navigation *only* if the logged-in user has the 'admin' role.
- Admin Page: A basic, placeholder "Admin Dashboard" page that is accessible when the admin link is clicked.
- Testing: Confirmed that a regular 'user' does not see the admin link and cannot access the admin page or API endpoint directly.

Review Requirements:

- Security Review: Frontend correctly interprets roles and does not expose admin routes or components to unauthorized users.
- Code Quality Review: Frontend code is clean, and the conditional rendering logic is efficient and secure.
- User Experience Review: The experience is seamless for both admin and regular users.

Stretch Goals (Nice-to-have)

- Permission-Based Control: Implement a more granular system where roles have specific permissions (e.g., read:data, write:users) and protect endpoints based on those permissions.
- Admin User Management: Build a UI in the Admin Dashboard where an admin can view all users and change their roles.
- Dynamic Role Updates: If an admin changes a user's role, ensure the user's session/token is refreshed to reflect the new role immediately.

Deliverables

1. Updated GitHub Repository.
2. Demonstrating the different experiences for a regular user and an admin user.
3. RBAC_DEMO.md - Include:
    - Screenshots of the UI for a logged-in user.
    - Screenshots of the UI for a logged-in admin (showing the admin link).
    - Screenshots of a successful API call to the admin-only endpoint (using an admin token).
    - Screenshots of a failed API call to the admin-only endpoint (using a user token, showing a 403 Forbidden error).
4. Updated Technical_Learnings.md.

Evaluation Rubric (100 Points)

Criterion	Points	Details
Backend Implementation	40 pts	➢ Role added to DB/model correctly. ➢ JWT includes role. ➢ Admin endpoint is created and properly secured. ➢ Correct HTTP status codes used.
Frontend Implementation	30 pts	➢ Auth context correctly handles the role. ➢ Admin UI elements are rendered conditionally and securely. ➢ Routing to the admin page works correctly for admins.
Security & Testing	20 pts	➢ Robust protection against unauthorized access. ➢ Clear evidence of testing both admin and user roles. ➢ No security loopholes.

Code Quality & Docs	10 pts	➢ Code is clean, maintainable, and well-documented. ➢ The migration script is clean and functional.