Project 5 - Role-Based Access Control (RBAC) - Key Concepts

1. Authentication vs. Authorization: The Core Difference

Concept Overview

- Authentication (AuthN) is the process of verifying a user's identity. It answers the question: "Who are you?" This was the entire focus of Project 4. When a user provides a username and password, the system authenticates them and confirms they are who they claim to be.
- Authorization (AuthZ) is the process of determining if an *authenticated* user has permission to access a specific resource or perform a particular action. It

answers the question: "What are you allowed to do?" This is the focus of Project 5.

Analogy: The Office Building

- Authentication is showing your ID badge to the security guard at the front door. The guard verifies you are a legitimate employee and lets you into the building.
- Authorization is what happens *inside* the building. Your ID badge may grant you access to the main office floor, but it won't open the door to the server room or the CEO's office. Your access level (your role) determines which doors you can open.

Why the Distinction Matters

Separating these two concerns is a fundamental principle of security design. A system must first authenticate a user before it can authorize them. Failing to properly authorize users can lead to serious security breaches, where legitimate, logged-in users can access data or functionality they should not be able to.

2. Role-Based Access Control (RBAC) Explained

Concept Overview

RBAC is a popular and effective authorization strategy. Instead of assigning permissions directly to individual users, permissions are assigned to roles, and then roles are assigned to users.

Core Components

- User: An individual who can log in to the system (e.g., `john.doe@example.com`).
- Role: A collection of permissions. A role represents a specific job function or access level within the system (e.g., `admin`, `user`, `editor`).

- Permission: A specific action that can be performed on a resource (e.g., `read_document`, `delete_user`, `access_dashboard`). *(Note: For this project, we will keep it simple and our "permissions" will be implicitly tied to the role).*

How It Works

1. A user is assigned one or more roles. In our project, a user will have one role: either `user` or `admin`.
2. When the user logs in, their role is retrieved and included in their session information (in our case, the JWT).
3. When the user tries to access a resource (like an API endpoint or a UI component), the system checks if the user's role has the necessary permission.
4. Access is granted or denied based on that check.

Advantages of RBAC

- Scalability: Managing roles is much easier than managing permissions for thousands of individual users.
- Maintainability: To change what a group of users can do, you only need to modify the permissions for their role, not for each user.
- Clarity: It provides a clear, understandable structure for access rights that often mirrors an organization's structure.

3. Extending JWTs for Authorization

Concept Overview

In Project 4, our JWT payload was simple, containing claims like `sub` (user ID) and `exp` (expiration). To support authorization, we need to add a new claim for the user's role.

The role Claim

We will add a `role` claim to the JWT payload. When a user logs in, we will fetch their role from the database and include it in the token.

Example JWT Payload (Before vs. After)

Project 4 Payload (Authentication only):

```JSON
{
  "sub": "12345",
  "exp": 1678886400,
  "iat": 1678882800
}
```

Project 5 Payload (Authentication + Authorization):

```JSON
{
  "sub": "12345",
  "role": "admin",  // <-- The new authorization claim
  "exp": 1678886400,
  "iat": 1678882800
}
```

Security Implication: The JWT is the Source of Truth

Once the JWT is issued, the backend and frontend should trust the `role` claim within it for the duration of the token's life. This is efficient because it avoids a database lookup on every request. However, it also means that if a user's role is changed in the database, the change will not take effect until their current JWT expires and they log in again to get a new one.

4. Securing API Endpoints with Role Checks

Concept Overview

Protecting an API endpoint with RBAC involves creating a dependency (or middleware) that not only verifies the JWT's validity (authentication) but also inspects its payload to check the user's role (authorization).

Implementation in FastAPI

In FastAPI, this is elegantly handled by creating a dependency that requires a specific role.

```python
# A simplified dependency to check for an admin role
from fastapi import Depends, HTTPException, status

# Assume get_current_user is the dependency from Project 4
# that validates the JWT and returns the user model.
from .auth import get_current_user
from ..models.user import User

def get_current_admin_user(current_user: User =
Depends(get_current_user)):
    """
    A dependency that checks if the current user is an admin.
    If not, it raises an HTTP 403 Forbidden error.
    """
    if current_user.role != "admin":
        raise HTTPException(
            status_code=status.HTTP_403_FORBIDDEN,
            detail="The user does not have sufficient privileges"
```

```
    )
  return current_user

# Protecting an endpoint with the dependency
@router.get("/admin/dashboard")
def get_admin_dashboard_data(admin_user: User =
Depends(get_current_admin_user)):
  # This code will only execute if the user is an admin.
  return {"message": "Welcome to the Admin Dashboard!"}
```

HTTP Status Codes

- 401 Unauthorized: This should be returned if the user provides no token or an invalid token. This is an *authentication* failure.
- 403 Forbidden: This should be returned if the user provides a *valid* token, but their role does not grant them access to the resource. This is an *authorization* failure.

5. Conditional UI Rendering Based on Roles

Concept Overview

A key part of a good user experience is only showing users the UI elements they are allowed to interact with. We can use the `role` from the JWT on the frontend to conditionally render components.

Implementation in React

In React, we can achieve this by accessing the user's role from our `AuthContext` and using it in our rendering logic.

```
None
// Example of a protected component in React
import React from 'react';
import { useAuth } from '../context/AuthContext';

const AdminDashboardLink = () => {
  const { user } = useAuth(); // Assume user object contains the role

  // If the user's role is 'admin', render the link. Otherwise, render nothing.
  if (user && user.role === 'admin') {
    return (
      <a href="/admin/dashboard">Admin Dashboard</a>
    );
  }

  return null; // Render nothing for non-admin users
};

export default AdminDashboardLink;
```

Security Note: Frontend Hiding is Not Enough

Hiding a button in the UI is for user experience, not for security. A savvy user can still attempt to access the API endpoint directly. True security is always enforced on the backend. Our frontend logic simply improves the UX by not showing options that would lead to a "Forbidden" error anyway.