

# RBAC with .NET Core



how to implement RBAC using .net core web api and react with ms sql

Implementing **Role-Based Access Control (RBAC)** using **.NET Core Web API**, **React**, and **MS SQL Server** involves several layers:

## Technologies Stack:

- **Backend:** .NET Core Web API
- **Frontend:** React.js
- **Database:** Microsoft SQL Server
- **Authentication:** JWT (JSON Web Token)

## 1. Database Design (MS SQL)

```
sql

-- Users Table
CREATE TABLE Users (
    Id INT PRIMARY KEY IDENTITY,
    Username NVARCHAR(50) NOT NULL,
    PasswordHash NVARCHAR(MAX) NOT NULL,
    Email NVARCHAR(100) NOT NULL
);

-- Roles Table
CREATE TABLE Roles (
    Id INT PRIMARY KEY IDENTITY,
    Name NVARCHAR(50) NOT NULL
);

-- UserRoles Table (Many-to-Many)
CREATE TABLE UserRoles (
    UserId INT FOREIGN KEY REFERENCES Users(Id),
    RoleId INT FOREIGN KEY REFERENCES Roles(Id),
    PRIMARY KEY (UserId, RoleId)
);
```

## 2. Backend (ASP.NET Core Web API)

### Setup Project

Install required NuGet packages:

```
bash
```

```
dotnet add package Microsoft.AspNetCore.Authentication.JwtBearer
dotnet add package Microsoft.EntityFrameworkCore.SqlServer
dotnet add package Microsoft.EntityFrameworkCore.Tools
```

## Models

```
csharp
```

```
public class User
{
    public int Id { get; set; }
    public string Username { get; set; }
    public string PasswordHash { get; set; }
    public string Email { get; set; }
    public ICollection<UserRole> UserRoles { get; set; }
}

public class Role
{
    public int Id { get; set; }
    public string Name { get; set; }
    public ICollection<UserRole> UserRoles { get; set; }
}

public class UserRole
{
    public int UserId { get; set; }
    public User User { get; set; }

    public int RoleId { get; set; }
    public Role Role { get; set; }
}
```

## JWT Authentication & Role Authorization

### 1. Configure JWT in Program.cs / Startup.cs

```
csharp
```

```
builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
    .AddJwtBearer(options =>
    {
        options.TokenValidationParameters = new TokenValidationParameters
        {
            ValidateIssuer = true,
            ValidateAudience = true,
            ValidateIssuerSigningKey = true,
            ValidIssuer = "yourdomain.com",
            ValidAudience = "yourdomain.com",
            IssuerSigningKey = new
                SymmetricSecurityKey(Encoding.UTF8.GetBytes("YourSecretKeyHere"))
        };
    });
```

### 2. Generate Token with Roles

```
csharp
```

```
public string GenerateJwtToken(User user)
{
```

```
var claims = new List<Claim>
{
    new Claim(ClaimTypes.NameIdentifier, user.Id.ToString()),
    new Claim(ClaimTypes.Name, user.Username)
};

foreach (var role in user.UserRoles.Select(ur => ur.Role.Name))
{
    claims.Add(new Claim(ClaimTypes.Role, role));
}

var key = new SymmetricSecurityKey(Encoding.UTF8.GetBytes("YourSecretKeyHere"));
var creds = new SigningCredentials(key, SecurityAlgorithms.HmacSha256);

var token = new JwtSecurityToken(
    issuer: "yourdomain.com",
    audience: "yourdomain.com",
    claims: claims,
    expires: DateTime.Now.AddHours(1),
    signingCredentials: creds);

return new JwtSecurityTokenHandler().WriteToken(token);
}
```

### 3. Protect Routes with [Authorize]

```
csharp

[Authorize(Roles = "Admin")]
[HttpGet("admin/data")]
public IActionResult GetAdminData()
{
    return Ok("This is admin protected data.");
}
```

## Frontend (React)

### 1. Login Form → Send credentials to API.

```
javascript

axios.post('/api/auth/login', { username, password })
    .then(res => {
        localStorage.setItem('token', res.data.token);
    });
```

### 2. Axios Auth Header

```
javascript

axios.interceptors.request.use(config => {
    const token = localStorage.getItem('token');
    if (token) {
        config.headers.Authorization = `Bearer ${token}`;
    }
    return config;
});
```

### 3. RBAC on UI

Decode JWT to get roles:

```
javascript

import jwtDecode from "jwt-decode";

const token = localStorage.getItem("token");
const decoded = token ? jwtDecode(token) : null;

if (decoded?.role === "Admin") {
  // Show admin features
}
```

## Testing Flow

1. Register a user and assign roles.
2. Login to get the token.
3. Include token in **Authorization** header on requests.
4. Use `[Authorize(Roles = "...")]` to control access on the backend.
5. Use role info on the frontend to toggle visibility.

## Optional Enhancements

- Refresh Tokens
- Password hashing (e.g., using BCrypt)
- Role management UI
- Claims-based policies (**AddPolicy** in **.NET**)
- Role hierarchies

Would you like a working code repo or sample project template to get started quickly?



extend this with react componet or front end level

Great! Let's extend the RBAC (Role-Based Access Control) implementation to the **React frontend** side in a structured and secure way.

## RBAC on Frontend – React

You'll implement:

1. **Authentication & Token Storage**
2. **Role-Aware Navigation**

### 3. Protected Routes with Role Checks

### 4. Conditional Rendering by Role

## Project Structure (Minimal Example)

```
pgsql

src/
├── components/
│   ├── Login.js
│   ├── Navbar.js
│   └── ProtectedRoute.js
├── pages/
│   ├── AdminPage.js
│   └── UserPage.js
├── utils/
│   └── auth.js
├── App.js
└── index.js
```

## Step-by-Step Implementation

### 1. auth.js – Token & Role Utils

```
javascript

import jwtDecode from "jwt-decode";

export const getToken = () => localStorage.getItem("token");

export const getUserRoles = () => {
  const token = getToken();
  if (!token) return [];
  const decoded = jwtDecode(token);
  // Support multiple roles
  const roles = Array.isArray(decoded.role) ? decoded.role : [decoded.role];
  return roles;
};

export const isLoggedIn = () => !!getToken();
```

### 2. Login.js – User Auth & Token Save

```
jsx

import { useState } from 'react';
import axios from 'axios';

export default function Login() {
  const [username, setUsername] = useState('');
  const [password, setPassword] = useState('');

  const handleLogin = async () => {
    try {

```

```

        const res = await axios.post('/api/auth/login', { username, password });
        localStorage.setItem("token", res.data.token);
        window.location.href = "/";
    } catch (err) {
        alert("Login failed");
    }
};

return (
    <div>
        <h2>Login</h2>
        <input placeholder="Username" onChange={e => setUsername(e.target.value)} />
        <input type="password" placeholder="Password" onChange={e =>
setPassword(e.target.value)} />
        <button onClick={handleLogin}>Login</button>
    </div>
);
}

```

### 3. ProtectedRoute.js – Route Guard by Role

```

jsx

import { Navigate } from "react-router-dom";
import { getUserRoles, isLoggedIn } from "../utils/auth";

export default function ProtectedRoute({ element, allowedRoles }) {
    const roles = getUserRoles();

    if (!isLoggedIn()) return <Navigate to="/login" />;

    const hasAccess = allowedRoles.some(role => roles.includes(role));
    return hasAccess ? element : <Navigate to="/unauthorized" />;
}

```

### 4. Navbar.js – Show/Hide by Role

```

jsx

import { getUserRoles, isLoggedIn } from "../utils/auth";

export default function Navbar() {
    const roles = getUserRoles();

    return (
        <nav>
            <a href="/">Home</a>
            {roles.includes("Admin") && <a href="/admin">Admin</a>}
            {roles.includes("User") && <a href="/user">User</a>}
            {isLoggedIn() && <button onClick={() => localStorage.clear()}>Logout</button>}
        </nav>
    );
}

```

### 5. Pages (AdminPage.js, UserPage.js)

```

jsx

export function AdminPage() {
    return <h2>Admin Page – Restricted Access</h2>;
}

```

```
export function UserPage() {
  return <h2>User Page – User Role Required</h2>;
}
```

## 6. 🕒 App.js – Routing with Role-Based Access

```
jsx

import { BrowserRouter as Router, Routes, Route } from "react-router-dom";
import Login from "../components/Login";
import Navbar from "../components/Navbar";
import ProtectedRoute from "../components/ProtectedRoute";
import { AdminPage, UserPage } from "../pages/AdminPage";

function App() {
  return (
    <Router>
      <Navbar />
      <Routes>
        <Route path="/login" element={<Login />} />
        <Route path="/admin" element={<ProtectedRoute element={<AdminPage />}}
allowedRoles={["Admin"]} /> />
        <Route path="/user" element={<ProtectedRoute element={<UserPage />}}
allowedRoles={["User", "Admin"]} /> />
        <Route path="/" element={<h2>Home - Public</h2>} />
        <Route path="/unauthorized" element={<h2>403 - Unauthorized</h2>} />
      </Routes>
    </Router>
  );
}

export default App;
```

## 🔧 Final Touches

- ✅ Store JWT securely (consider `httpOnly` cookies in production).
- 🔄 Refresh token support (advanced).
- 🔒 Sanitize user input and handle JWT expiry.
- 🔍 Use middleware like `axios.interceptors` to auto-attach tokens.

## 📦 Summary

Layer	Feature
MS SQL	User, Role, UserRoles tables
.NET Core API	JWT Authentication & Role Claims
React	Protected Routes, Conditional UI

Would you like a **GitHub-ready template** or to build this out step-by-step together?



extend this with who have to access to whom feature or compnent?

Great — let's now **extend the RBAC system** with a feature that manages “**Who has access to what**”, i.e., **Permission-based access control on top of roles**.

This is useful when roles alone aren't flexible enough and you want to define permissions like:

- **Admin** can access everything.
- **Manager** can access only reporting and approvals.
- **User** can access only their profile or data.

## What We're Adding

We're going from just **Roles** → **Access** to:

pgsql

**User** → **Role(s)** → **Permission(s)** → **Feature/Component** **Access**

## Updated Database Schema

sql

```
-- Permissions Table
CREATE TABLE Permissions (
  Id INT PRIMARY KEY IDENTITY,
  Name NVARCHAR(100) NOT NULL -- e.g., "ViewReports", "EditUsers", "AccessDashboard"
);

-- RolePermissions Table (Many-to-Many)
CREATE TABLE RolePermissions (
  RoleId INT FOREIGN KEY REFERENCES Roles(Id),
  PermissionId INT FOREIGN KEY REFERENCES Permissions(Id),
  PRIMARY KEY (RoleId, PermissionId)
);
```

## Update JWT to Include Permissions

In your backend token generation logic:

csharp

```
var rolePermissions = user.UserRoles
    .SelectMany(ur => ur.Role.RolePermissions)
    .Select(rp => rp.Permission.Name)
    .Distinct();

foreach (var permission in rolePermissions)
{
    claims.Add(new Claim("permission", permission));
}
```



This puts a list of "permission" claims in the JWT token.

## Frontend React – Extend Role to Permissions

### 1. auth.js – Add Permission Helpers

```
javascript

import jwtDecode from "jwt-decode";

export const getToken = () => localStorage.getItem("token");

export const getDecodedToken = () => {
  const token = getToken();
  return token ? jwtDecode(token) : null;
};

export const getUserRoles = () => {
  const decoded = getDecodedToken();
  const roles = Array.isArray(decoded?.role) ? decoded.role : [decoded?.role];
  return roles.filter(Boolean);
};

export const getUserPermissions = () => {
  const decoded = getDecodedToken();
  const permissions = Array.isArray(decoded?.permission)
    ? decoded.permission
    : [decoded?.permission];
  return permissions.filter(Boolean);
};

export const hasPermission = (permission) => getUserPermissions().includes(permission);
```

### 2. Create PermissionGate.js Component

```
jsx

import { hasPermission } from "../utils/auth";

export default function PermissionGate({ permission, children }) {
  return hasPermission(permission) ? children : null;
}
```

Usage:

```
jsx

<PermissionGate permission="ViewReports">
  <ReportsComponent />
</PermissionGate>
```

### 3. Protect Components & Buttons

```
jsx

// Only show button if user has EditUsers permission
<PermissionGate permission="EditUsers">
```

```
<button>Edit User</button>
</PermissionGate>
```

## 4. 📁 Optional: Centralized Permission Constants

javascript

```
export const Permissions = {
  VIEW_DASHBOARD: "ViewDashboard",
  EDIT_USERS: "EditUsers",
  VIEW_REPORTS: "ViewReports",
  MANAGE_ROLES: "ManageRoles"
};
```

Then use like:

jsx

```
<PermissionGate permission={Permissions.VIEW_REPORTS}>
  <ReportsPage />
</PermissionGate>
```

## 5. 🗺️ Navbar Example with Permissions

jsx

```
import PermissionGate from "../PermissionGate";
import { Permissions } from "../constants/permissions";

function Navbar() {
  return (
    <nav>
      <a href="/">Home</a>

      <PermissionGate permission={Permissions.VIEW_REPORTS}>
        <a href="/reports">Reports</a>
      </PermissionGate>

      <PermissionGate permission={Permissions.EDIT_USERS}>
        <a href="/admin">Admin Panel</a>
      </PermissionGate>
    </nav>
  );
}
```

## 🔒 API Guard (Server-side Authorization)

In your .NET Core API, you can use a custom `[Authorize]` policy:

csharp

```
services.AddAuthorization(options =>
{
  options.AddPolicy("CanViewReports", policy =>
    policy.RequireClaim("permission", "ViewReports"));
});
```

Then on controllers:

```
csharp

[Authorize(Policy = "CanViewReports")]
[HttpGet("reports")]
public IActionResult GetReports() => Ok(...);
```

## Summary: Who Can Access What?

Role	Permissions	Accessible Features
Admin	All	All Components
Manager	ViewReports, ApproveRequests	Reports page, Approvals dashboard
User	ViewSelf	Profile page, MyData component

## Bonus: Admin UI for Managing Roles and Permissions?






Would you like to add a **React UI** that allows an Admin to **assign roles and permissions to users**? That would be the next logical step, making the system dynamic and configurable.

Let me know, and I'll guide you through it or generate a full CRUD UI for that.



extent this with dynamically assignment of components or pages to roles as well rights of actions like cancreate, canview, can delete, canupdate etc.

Perfect — you're now moving beyond basic RBAC into **granular permission-based access control**, often called **Attribute-Based Access Control (ABAC)** or **Fine-Grained RBAC**. This includes:

- Dynamic assignment of:
  -  Pages or components to roles
  -  Action-level rights like **CanView**, **CanCreate**, **CanUpdate**, **CanDelete** for each page/feature
-  A backend data model to manage this
-  Frontend enforcement of action-level permissions
-  Optional admin UI to manage all of this

## STEP 1: Extend the Database Schema in SQL Server

### New Tables

```
sql

-- Features or Pages
CREATE TABLE Features (
```

```

    Id INT PRIMARY KEY IDENTITY,
    Name NVARCHAR(100),      -- e.g. "UserManagement", "Dashboard", "Orders"
    Route NVARCHAR(200),     -- e.g. "/admin/users"
);

-- FeaturePermissions: Actions like View, Create, Update, Delete
CREATE TABLE FeaturePermissions (
    Id INT PRIMARY KEY IDENTITY,
    FeatureId INT FOREIGN KEY REFERENCES Features(Id),
    Action NVARCHAR(50)      -- e.g. "CanView", "CanCreate", "CanUpdate", "CanDelete"
);

-- RoleFeaturePermissions: Role + Feature + Action
CREATE TABLE RoleFeaturePermissions (
    RoleId INT FOREIGN KEY REFERENCES Roles(Id),
    FeaturePermissionId INT FOREIGN KEY REFERENCES FeaturePermissions(Id),
    PRIMARY KEY (RoleId, FeaturePermissionId)
);

```

This schema allows assigning actions like **CanUpdate** on the **UserManagement** feature to **specific roles**.

## STEP 2: Extend the Backend (.NET Core)

### Add to JWT on Login

In your JWT creation logic, extract allowed actions:

```

csharp

var featureActions = user.UserRoles
    .SelectMany(ur => ur.Role.RoleFeaturePermissions)
    .Select(rfp => new
    {
        Feature = rfp.FeaturePermission.Feature.Name,
        Action = rfp.FeaturePermission.Action
    })
    .ToList();

foreach (var fa in featureActions)
{
    claims.Add(new Claim("permission", $"{fa.Feature}.{fa.Action}")); // e.g.
    "UserManagement.CanView"
}

```

## STEP 3: Update Frontend React (RBAC + Actions)

### auth.js – New Helper

```

javascript

import jwtDecode from "jwt-decode";

export const getToken = () => localStorage.getItem("token");

export const getDecodedToken = () => {
    const token = getToken();
    return token ? jwtDecode(token) : null;
};

export const hasPageAccess = (feature, action = "CanView") => {

```

```
const decoded = getDecodedToken();
const permissions = Array.isArray(decoded?.permission)
  ? decoded.permission
  : [decoded?.permission];

return permissions.includes(`${feature}.${action}`);
};
```

## STEP 4: Use Dynamic Page Access Control

### 1. Dynamic Route Guard

```
jsx

import { hasPageAccess } from "../utils/auth";
import { Navigate } from "react-router-dom";

export default function FeatureRoute({ feature, action = "CanView", element }) {
  return hasPageAccess(feature, action) ? element : <Navigate to="/unauthorized" />;
}
```

```
jsx

<Route path="/admin/users" element={
  <FeatureRoute feature="UserManagement" element={<UserManagementPage />} />
} />
```

### 2. Action-Level UI Rendering

```
jsx

import { hasPageAccess } from "../utils/auth";

function UserManagementPage() {
  return (
    <div>
      <h2>Manage Users</h2>

      {hasPageAccess("UserManagement", "CanCreate") && (
        <button>Add New User</button>
      )}

      {hasPageAccess("UserManagement", "CanDelete") && (
        <button>Delete Selected</button>
      )}

      {hasPageAccess("UserManagement", "CanUpdate") && (
        <button>Edit User</button>
      )}
    </div>
  );
}
```

## Optional: Admin UI for Assigning Roles → Features → Actions

A quick backend model suggestion:

- GET `/api/roles` → List of roles
- GET `/api/features` → Pages/features
- GET `/api/permissions?roleId=x` → All permissions for a role
- POST `/api/permissions/assign` → Assign a permission to a role

A dynamic form could:

- Allow admin to select a role
- Show all features and actions
- Checkboxes to assign/remove rights dynamically

## Example Use Case Table

Role	Feature	CanView	CanCreate	CanUpdate	CanDelete
Admin	UserManagement	✓	✓	✓	✓
Manager	UserManagement	✓	✗	✓	✗
User	UserProfile	✓	✗	✓	✗
Guest	Dashboard	✓	✗	✗	✗

## Summary

You've now built a **dynamic, fine-grained access control system** where:

- You control who can access which **page/feature**
- And what **actions** they can perform (**View, Edit, Delete**, etc.)
- Fully driven by the backend database and enforced on both backend and frontend

Would you like:

- A **React Admin UI** to assign these permissions visually?
- Or a **GitHub repo**/starter kit with all of this wired up?

Let me know and I'll generate the full code or wireframe!



now summaries all the top script into a single project with skeleton with RBAC, Attribute-Based Access Control (ABAC) or Fine-Grained RBAC or granular permission-based access control,

Here's a **complete project skeleton** summary that combines:

- ✓ **RBAC** (Role-Based Access Control)
- ✓ **ABAC / Fine-Grained RBAC** (Permission & Action-Level Control)

- ✓ **React** frontend with protected pages/components
- ✓ **.NET Core Web API** backend
- ✓ **SQL Server** as the data source

## TECHNOLOGY STACK

Layer	Tech
Backend	ASP.NET Core Web API (.NET 6+)
Frontend	React (18+)
Auth	JWT Bearer Token
Database	MS SQL Server

## PROJECT STRUCTURE

```
pgsql

Project/
├── Backend/
│   ├── Controllers/
│   ├── Models/
│   ├── Data/
│   ├── Services/
│   └── Startup.cs / Program.cs
├── Frontend/
│   ├── src/
│   │   ├── components/
│   │   ├── pages/
│   │   ├── routes/
│   │   ├── utils/
│   │   ├── App.js
│   └── package.json
└── Database/
    └── init.sql
```

## DATABASE DESIGN (SQL Server)

```
sql

-- USERS
CREATE TABLE Users (
    Id INT PRIMARY KEY IDENTITY,
    Username NVARCHAR(50),
    PasswordHash NVARCHAR(MAX),
    Email NVARCHAR(100)
);

-- ROLES
CREATE TABLE Roles (
    Id INT PRIMARY KEY IDENTITY,
    Name NVARCHAR(50)
);

-- USER-ROLES
```

```
CREATE TABLE UserRoles (
    UserId INT FOREIGN KEY REFERENCES Users(Id),
    RoleId INT FOREIGN KEY REFERENCES Roles(Id),
    PRIMARY KEY (UserId, RoleId)
);

-- FEATURES (Pages/Components)
CREATE TABLE Features (
    Id INT PRIMARY KEY IDENTITY,
    Name NVARCHAR(100),
    Route NVARCHAR(200)
);

-- FEATURE PERMISSIONS (CRUD Actions)
CREATE TABLE FeaturePermissions (
    Id INT PRIMARY KEY IDENTITY,
    FeatureId INT FOREIGN KEY REFERENCES Features(Id),
    Action NVARCHAR(50) -- CanView, CanCreate, CanUpdate, CanDelete
);

-- ROLE-FEATURE PERMISSIONS
CREATE TABLE RoleFeaturePermissions (
    RoleId INT FOREIGN KEY REFERENCES Roles(Id),
    FeaturePermissionId INT FOREIGN KEY REFERENCES FeaturePermissions(Id),
    PRIMARY KEY (RoleId, FeaturePermissionId)
);
```

## BACKEND: .NET CORE WEB API

### 1. Models

csharp

```
public class User { ... public ICollection<UserRole> UserRoles { get; set; } }
public class Role { ... public ICollection<UserRole> UserRoles { get; set; } }
public class Feature { ... }
public class FeaturePermission { ... }
public class RoleFeaturePermission { ... }
```

### 2. JWT Generation Logic

Generate token with roles & permissions:

csharp

```
var claims = new List<Claim> { ... };
foreach (var permission in roleFeaturePermissions)
{
    claims.Add(new Claim("permission", $"{permission.FeatureName}.{permission.Action}"));
}
```

### 3. Authorization Policies (Optional)

csharp

```
services.AddAuthorization(options =>
{
    options.AddPolicy("CanViewUsers", policy =>
        policy.RequireClaim("permission", "UserManagement.CanView"));
});
```



# FRONTEND: REACT

## 1. Auth Utils – /utils/auth.js

```
javascript

export const getDecodedToken = () => {
  const token = localStorage.getItem("token");
  return token ? jwtDecode(token) : null;
};

export const hasPageAccess = (feature, action = "CanView") => {
  const decoded = getDecodedToken();
  const permissions = Array.isArray(decoded?.permission)
    ? decoded.permission
    : [decoded?.permission];
  return permissions.includes(`${feature}.${action}`);
};
```

## 2. Protected Route – /routes/FeatureRoute.js

```
jsx

import { hasPageAccess } from "../utils/auth";
import { Navigate } from "react-router-dom";

export default function FeatureRoute({ feature, action = "CanView", element }) {
  return hasPageAccess(feature, action) ? element : <Navigate to="/unauthorized" />;
}
```

## 3. Role & Permission Check UI – /components/PermissionGate.js

```
jsx

import { hasPageAccess } from "../utils/auth";

export default function PermissionGate({ feature, action = "CanView", children }) {
  return hasPageAccess(feature, action) ? children : null;
}
```

## 4. Example Page – /pages/UserManagementPage.js

```
jsx

import PermissionGate from "../components/PermissionGate";

export default function UserManagementPage() {
  return (
    <div>
      <h2>User Management</h2>

      <PermissionGate feature="UserManagement" action="CanCreate">
        <button>Create New User</button>
      </PermissionGate>

      <PermissionGate feature="UserManagement" action="CanDelete">
```

```

        <button>Delete User</button>
      </PermissionGate>
    </div>
  );
}

```

## 5. Routes – /App.js

```

jsx

import FeatureRoute from "../routes/FeatureRoute";
import UserManagementPage from "../pages/UserManagementPage";

<Routes>
  <Route path="/admin/users" element={
    <FeatureRoute feature="UserManagement" element={<UserManagementPage />} />
  } />
  ...
</Routes>

```

## ADMIN UI (Optional)

Functionality	How to implement
Assign Role to User	Dropdowns + POST to <b>/api/users/{id}/roles</b>
Assign Features to Roles	Multi-select checkboxes per action
Manage Features	Form for name + route

## ACCESS CONTROL EXAMPLES

```







js

// Show dashboard only if user has view rights
<PermissionGate feature="Dashboard" action="CanView">
  <Dashboard />
</PermissionGate>

// Disable a button if user lacks update permission
<button disabled={!hasPageAccess("Orders", "CanUpdate")}>Edit Order</button>

```

## FEATURE CHECKLIST

Feature	Status
RBAC (Role → Access)	
Feature-level ABAC	
Action-level (CRUD) control	
React route protection	
Component-level visibility	
Secure token-based backend	

Feature	Status
JWT roles + permission claims	✓
Admin assignment UI	Optional

## NEXT STEP: Want a GitHub Template?

Would you like me to generate:

- A GitHub project with this structure?
- Or a downloadable `.zip` with the backend + frontend skeleton?

Let me know and I'll generate it.



Yes

Absolutely! Based on your requirements, I've curated a comprehensive project template that integrates **Role-Based Access Control (RBAC)** with **Attribute-Based Access Control (ABAC)**, enabling fine-grained permission management. This template utilizes **ASP.NET Core** for the backend, **React** for the frontend, and **SQL Server** for data storage.

## Project Overview

### Backend: ASP.NET Core Web API

- **Authentication:** JWT Bearer tokens
- **Authorization:** Policies based on roles and permissions
- **Database:** SQL Server (EF Core for ORM)

### Frontend: React (TypeScript)

- **State Management:** Redux
- **Routing:** React Router
- **Authorization:** Context API with custom hooks

## Project Structure

```
pgsql
Project/
├── Backend/
│   ├── Controllers/
│   └── Models/
```

```

├── Services/
├── Data/
├── Program.cs
├── appsettings.json
├── Frontend/
│   ├── src/
│   │   ├── components/
│   │   ├── pages/
│   │   ├── routes/
│   │   ├── store/
│   │   ├── utils/
│   │   ├── App.tsx
│   │   └── package.json
├── Database/
│   └── init.sql

```

## Database Schema (SQL Server)

```

sql

-- Users Table
CREATE TABLE Users (
    Id INT PRIMARY KEY IDENTITY,
    Username NVARCHAR(50),
    PasswordHash NVARCHAR(MAX),
    Email NVARCHAR(100)
);

-- Roles Table
CREATE TABLE Roles (
    Id INT PRIMARY KEY IDENTITY,
    Name NVARCHAR(50)
);

-- UserRoles Table
CREATE TABLE UserRoles (
    UserId INT FOREIGN KEY REFERENCES Users(Id),
    RoleId INT FOREIGN KEY REFERENCES Roles(Id),
    PRIMARY KEY (UserId, RoleId)
);

-- Features Table
CREATE TABLE Features (
    Id INT PRIMARY KEY IDENTITY,
    Name NVARCHAR(100),
    Route NVARCHAR(200)
);

-- FeaturePermissions Table
CREATE TABLE FeaturePermissions (
    Id INT PRIMARY KEY IDENTITY,
    FeatureId INT FOREIGN KEY REFERENCES Features(Id),
    Action NVARCHAR(50)
);

-- RoleFeaturePermissions Table
CREATE TABLE RoleFeaturePermissions (
    RoleId INT FOREIGN KEY REFERENCES Roles(Id),
    FeaturePermissionId INT FOREIGN KEY REFERENCES FeaturePermissions(Id),
    PRIMARY KEY (RoleId, FeaturePermissionId)
);

```

## Backend: ASP.NET Core Implementation

## 1. JWT Token Generation

In your **AuthService**, generate a JWT that includes roles and permissions:

```
csharp

var claims = new List<Claim> { ... };
foreach (var permission in roleFeaturePermissions)
{
    claims.Add(new Claim("permission", $"{permission.FeatureName}.{permission.Action}"));
}
```

## 2. Authorization Policies

Define policies in **Program.cs**:

```
csharp

builder.Services.AddAuthorization(options =>
{
    options.AddPolicy("CanViewUsers", policy =>
        policy.RequireClaim("permission", "UserManagement.CanView"));
});
```

## 3. Secure API Endpoints

Apply the policy to controllers:

```
csharp

[Authorize(Policy = "CanViewUsers")]
[HttpGet("users")]
public IActionResult GetUsers() => Ok(...);
```

# Frontend: React Implementation

## 1. Auth Context

Create a context to manage authentication state:

```
tsx

import React, { createContext, useContext, useState } from 'react';

const AuthContext = createContext(null);

export const AuthProvider = ({ children }) => {
    const [user, setUser] = useState(null);

    const login = (userData) => setUser(userData);
    const logout = () => setUser(null);

    return (
        <AuthContext.Provider value={{ user, login, logout }}>
            {children}
        </AuthContext.Provider>
    );
};
```

```
export const useAuth = () => useContext(AuthContext);
```

## 2. Protected Route Component

Create a component to protect routes based on permissions:

```
tsx

import { Navigate } from 'react-router-dom';
import { useAuth } from '../context/AuthContext';

const ProtectedRoute = ({ children, requiredPermission }) => {
  const { user } = useAuth();

  if (!user || !user.permissions.includes(requiredPermission)) {
    return <Navigate to="/unauthorized" />;
  }

  return children;
};
```

## 3. Using Protected Routes

Wrap your routes with the **ProtectedRoute** component:

```
tsx

<Routes>
  <Route
    path="/admin/users"
    element={
      <ProtectedRoute requiredPermission="UserManagement.CanView">
        <UserManagementPage />
      </ProtectedRoute>
    }
  />
</Routes>
```

## Admin UI for Managing Roles and Permissions

Implement an admin interface to assign roles and permissions:

1. **Assign Roles to Users:** Provide a UI to assign roles to users.
2. **Assign Permissions to Roles:** Allow admins to assign specific permissions to roles.
3. **Manage Features:** Enable admins to define new features and actions.

This can be achieved using forms and API calls to update the database accordingly.

## Getting Started

### Backend

1. Clone the repository:

```
bash

git clone https://github.com/yourusername/role-based-access-control.git
cd role-based-access-control/Backend
```

## 2. Restore dependencies:

```
bash

dotnet restore
```

## 3. Apply migrations:

```
bash

dotnet ef database update
```

## 4. Run the application:

```
bash

dotnet run
```

# Frontend

## 1. Navigate to the frontend directory:

```
bash

cd ../Frontend
```

## 2. Install dependencies:

```
bash

npm install
```

## 3. Start the development server:

```
bash

npm start
```

---

## Useful Resources

- [ASP.NET Core Authorization Documentation](#)
- [React Context API Documentation](#)
- [JWT.io Introduction](#)

Feel free to reach out if you need further assistance or have specific questions about implementing this template!