Windows Authentication & Role-Based Authorization

Overview

This application now uses **Windows Authentication** with **role-based authorization** configured through `appsettings.json`. No complex policies needed - just simple role annotations!

How It Works

1. Windows Authentication

- Uses built-in Windows/Negotiate authentication
- Captures Windows identity (domain\username or email)
- Works with IIS, IIS Express, or Kestrel with Windows Auth enabled

2. Role Mapping (appsettings.json)

JSON

```json
{
  "Authorization": {
    "Roles": [
      "LoggedIn",
      "User",
      "Admin",
      "Editor",
      "Viewer"
    ],
    "UserRoles": {
      "john@company.com": ["LoggedIn", "Admin", "Editor"],
      "jane@company.com": ["LoggedIn", "User", "Viewer"],
      "admin@company.com": ["LoggedIn", "Admin"]
    }
  }
}
```

3. Claims Transformation

- Custom `RoleClaimsTransformation` service reads user email
- Looks up roles in appsettings
- Adds role claims to `ClaimsPrincipal`
- Happens automatically on every request

4. Controller Authorization

Simple `[Authorize]` attributes:

```csharp
[Authorize(Roles = "Admin")]
public IActionResult AdminOnly() { }

[Authorize(Roles = "Editor,Admin")]
public IActionResult EditData() { }

[Authorize(Roles = "LoggedIn,Viewer,User,Editor,Admin")]
public IActionResult ViewData() { }
```

Role Definitions

Role	Description	Permissions
LoggedIn	Default for all authenticated users	Basic access
Viewer	Read-only access	GET endpoints only
User	Standard user	Read + limited write
Editor	Can modify data	Read + Create + Update
Admin	Full access	All operations including Delete

API Endpoint Permissions

Read Operations (GET)

- **Roles:** `LoggedIn`, `Viewer`, `User`, `Editor`, `Admin`
- **Endpoints:**
    - `GET /api/tables` - List tables
    - `GET /api/tables/{tableName}` - Table metadata
    - `GET /api/tables/{tableName}/rows` - Get rows (paginated)
    - `GET /api/tables/{tableName}/rows/{id}` - Get single row

Create Operations (POST)

- **Roles:** `Editor`, `Admin`
- **Endpoints:**
    - `POST /api/tables/{tableName}/rows` - Create new row

Update Operations (PUT)

- **Roles:** `Editor`, `Admin`
- **Endpoints:**
    - `PUT /api/tables/{tableName}/rows/{id}` - Update row

Delete Operations (DELETE)

- **Roles:** `Admin` only
- **Endpoints:**
    - `DELETE /api/tables/{tableName}/rows/{id}` - Delete row

Query Operations (POST)

- **Roles:** `LoggedIn`, `Viewer`, `User`, `Editor`, `Admin`
- **Endpoints:**
    - `POST /api/tables/{tableName}/query` - Query with filters
    - `POST /api/tables/query/join` - JOIN queries

Configuration

Adding New Users

Edit `appsettings.json`:

```JSON
{
  "Authorization": {
    "UserRoles": {
      "newuser@company.com": ["LoggedIn", "User"],
      "another@company.com": ["LoggedIn", "Editor", "Admin"]
    }
  }
}
```

No code changes needed! Just restart the app.

Adding New Roles

1. Add role to the `Roles` array in `appsettings.json`
2. Assign to users in `UserRoles`
3. Use in controller attributes: `[Authorize(Roles = "YourNewRole")]`

Default Behavior

If a user is authenticated but not in `UserRoles`:

- Automatically gets `LoggedIn` role
- Can access endpoints that allow `LoggedIn`
- Logged as a warning for visibility

UI Components

UserInfo Component

Displays in the sidebar:

- User avatar with initials
- User name and email
- Role badges (color-coded)

Role Badge Colors:

- 🔴 **Admin** - Red
- 🟡 **Editor** - Yellow
- 🔵 **User** - Cyan

- ⚫ **Viewer** - Gray
- 🟢 **LoggedIn** - Green

Testing Locally

Option 1: IIS Express (Recommended)

1. Open project in Visual Studio
2. Enable Windows Authentication in `launchSettings.json`:
3. Run with IIS Express
4. Your Windows identity will be used automatically

Option 2: Kestrel with Windows Auth

1. Install `Microsoft.AspNetCore.Authentication.Negotiate` (already added)
2. Run with: `dotnet run`
3. Configure your Windows account in `appsettings.json`

Option 3: IIS Deployment

1. Publish the application
2. Deploy to IIS
3. Enable Windows Authentication in IIS Manager:
    - Select your site
    - Authentication → Enable Windows Authentication
    - Disable Anonymous Authentication
4. Users will authenticate with their Windows credentials

Troubleshooting

"No roles found for user"

Cause: User email not in `appsettings.json`

Solution:

1. Check logs for the actual email being used

2. Add user to `UserRoles` in `appsettings.json`
3. Restart the app

"Could not extract email from user identity"

Cause: Windows identity doesn't include email claim

Solution:

1. Check logs for available claims
2. Update `RoleClaimsTransformation.GetUserEmail()` to handle your claim type
3. Or map Windows username to email in appsettings

401 Unauthorized in Swagger

Cause: Swagger doesn't automatically pass Windows credentials

Solution:

- Use browser to test endpoints (credentials passed automatically)
- Or configure Swagger to use Windows Auth (advanced)
- Or temporarily allow anonymous for testing

Testing Without Windows Auth

For development/testing without Windows Auth:

1. Comment out authentication in `Program.cs`:
2. Remove `[Authorize]` attributes temporarily
3. Or create a mock authentication handler for testing

Security Best Practices

1. Principle of Least Privilege

- Give users only the roles they need
- Start with `Viewer` or `User`, not `Admin`

2. Regular Audits

- Review `UserRoles` periodically
- Remove users who no longer need access

3. Logging

- Monitor authentication logs
- Watch for "No roles found" warnings
- Track authorization failures

4. Environment-Specific Configuration

- Use different `appsettings.json` per environment:
  - `appsettings.Development.json`
  - `appsettings.Production.json`
- Keep production role mappings secure

5. HTTPS Only

- Always use HTTPS in production
- Windows Auth credentials should never go over HTTP

Advanced: Custom Claims

Want to add more claims beyond roles?

Edit `RoleClaimsTransformation.TransformAsync()`:

```C#
// Add custom claims
identity.AddClaim(new Claim("Department", "Engineering"));
identity.AddClaim(new Claim("EmployeeId", "12345"));
```

Then access in controllers:

```C#
var department = User.FindFirst("Department")?.Value;
```

Code Structure

Key Files:

- `Program.cs` - Authentication/authorization setup
- `Services/RoleClaimsTransformation.cs` - Role mapping logic
- `Controllers/TablesController.cs` - Authorization attributes
- `Components/UserInfo.razor` - User display component
- `appsettings.json` - Role configuration
- `App.razor` - CascadingAuthenticationState wrapper

Flow:

1. User authenticates with Windows
2. `RoleClaimsTransformation` runs
3. User email extracted from Windows identity
4. Roles looked up in `appsettings.json`
5. Role claims added to `ClaimsPrincipal`
6. Controller checks `[Authorize(Roles = "...")]`
7. Access granted or denied

Example Scenarios

Scenario 1: View-Only User

JSON

```json
"viewer@company.com": ["LoggedIn", "Viewer"]
```

Can:

- View tables and metadata
- Query data
- Execute JOINs

Cannot:

- Create, update, or delete rows

Scenario 2: Data Editor

JSON

```
"editor@company.com": ["LoggedIn", "Editor"]
```

Can:

- Everything Viewer can do
- Create new rows
- Update existing rows

Cannot:

- Delete rows (Admin only)

Scenario 3: Administrator

```json
"admin@company.com": ["LoggedIn", "Admin"]
```

Can:

- Everything Editor can do
- Delete rows
- Full access to all endpoints

Scenario 4: Multi-Role User

```json
"poweruser@company.com": ["LoggedIn", "User", "Editor", "Admin"]
```

Can:

- Access any endpoint that requires any of these roles
- Roles are cumulative (OR logic)

Summary

✅ **Simple** - Just edit `appsettings.json`, no code changes

✅ **Flexible** - Add/remove users and roles easily

✅ **Secure** - Windows Authentication with role-based access

✅ **Transparent** - See current user and roles in UI

✅ **Standard** - Uses ASP.NET Core's built-in authorization

No complex policies, no custom middleware - just clean, simple role-based auth!