Managing Permissions in .NET Core Web API Using Claim-Based Policies with SQL Server

This guide covers how to manage user permissions, starting with basic access control and extending to specific actions such as Create, View, Add, Edit, and Delete. It also includes advanced operations like Export and Print.

It explains how to implement a role- and claim-based authorization model in a .NET Core Web API, using Microsoft SQL Server to store permissions and claim policies to enforce them within the API.

Here's a detailed guide for **Managing Permissions in a .NET Core Web API Using Claim-Based Policies with SQL Server**. It outlines how to secure endpoints based on user roles and fine-grained claims such as **Create**, **View**, **Add**, **Edit**, **Delete**, **Export**, and **Print**.

🔧 Step-by-Step Guide

✅ 1. Set Up the Database Schema

Use SQL Server to store:

- **Users**
- **Roles**
- **Permissions**
- **RolePermissions**
- **UserRoles**

```sql

-- Roles table
CREATE TABLE Roles (
    Id INT PRIMARY KEY IDENTITY,
    Name NVARCHAR(100) NOT NULL
);

-- Permissions table
CREATE TABLE Permissions (
    Id INT PRIMARY KEY IDENTITY,
    Name NVARCHAR(100) NOT NULL
);

-- Users table
CREATE TABLE Users (
    Id INT PRIMARY KEY IDENTITY,
    Username NVARCHAR(100) NOT NULL,
    PasswordHash NVARCHAR(256) NOT NULL -- or use Identity
);

-- UserRoles table
CREATE TABLE UserRoles (
    UserId INT,
    RoleId INT,
```

```
    PRIMARY KEY (UserId, RoleId),
    FOREIGN KEY (UserId) REFERENCES Users(Id),
    FOREIGN KEY (RoleId) REFERENCES Roles(Id)
);

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

🔒 2. Configure Authentication

Use **JWT Bearer Authentication** in your `Startup.cs` or `Program.cs` (depending on your .NET version):

```
csharp


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

🔐 3. Define and Add Claims for Permissions

When authenticating a user (e.g., during login), fetch their permissions from SQL Server and embed them into their JWT claims:

```
csharp


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

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

Generate the JWT:

```
csharp


var token = new JwtSecurityToken(
    issuer: "yourissuer.com",
    audience: "youraudience.com",
    claims: claims,
    expires: DateTime.UtcNow.AddHours(1),
```

```
    signingCredentials: new SigningCredentials(
        new SymmetricSecurityKey(Encoding.UTF8.GetBytes("Your_Secret_Key")),
        SecurityAlgorithms.HmacSha256
    )
);
```

🛡️ 4. Register Authorization Policies

In **`Program.cs`**:

```
csharp

builder.Services.AddAuthorization(options =>
{
    var permissions = new[] { "Create", "View", "Edit", "Delete", "Export", "Print" };
    foreach (var permission in permissions)
    {
        options.AddPolicy(permission, policy =>
            policy.RequireClaim("Permission", permission));
    }
});
```

🧱 5. Use Policies in Controllers

Decorate your controller actions with the **`[Authorize(Policy = "...")]`** attribute:

```
csharp

[Authorize(Policy = "View")]
[HttpGet]
public IActionResult GetData()
{
    return Ok("You can view data!");
}

[Authorize(Policy = "Delete")]
[HttpDelete("{id}")]
public IActionResult DeleteItem(int id)
{
    // Delete logic here
    return Ok($"Item {id} deleted.");
}
```

📥 6. Optional: Custom Authorization Handler (Advanced)

If permissions need to be more complex (e.g., resource-based access), create a custom handler:

```
csharp

public class PermissionRequirement : IAuthorizationRequirement
{
    public string Permission { get; }
    public PermissionRequirement(string permission) => Permission = permission;
}

public class PermissionHandler : AuthorizationHandler<PermissionRequirement>
{
    protected override Task HandleRequirementAsync(AuthorizationHandlerContext context,
PermissionRequirement requirement)
    {
```

```
        if (context.User.HasClaim(c => c.Type == "Permission" && c.Value ==
requirement.Permission))
        {
            context.Succeed(requirement);
        }
        return Task.CompletedTask;
    }
}
```

Register the handler:

```
csharp

builder.Services.AddSingleton<IAuthorizationHandler, PermissionHandler>();
builder.Services.AddAuthorization(options =>
{
    options.AddPolicy("Edit", policy =>
        policy.Requirements.Add(new PermissionRequirement("Edit")));
});
```

📌 Summary

Step	Task
1	Design SQL tables to store roles/permissions
2	Configure JWT-based authentication
3	Add user permissions as JWT claims
4	Register claims-based policies
5	Use **`[Authorize(Policy = "...")]`** in controllers
6	(Optional) Implement custom authorization handler

Would you like a **sample project** or **code template** to get started faster?

> make this dynamic

Got it! You want a **dynamic, flexible permissions system** in .NET Core Web API that can load policies and permissions from the database at runtime, rather than hardcoding them.

Here's a clean way to do **dynamic claim-based authorization with policies loaded from SQL Server**:

Dynamic Claim-Based Authorization with Policies Loaded from SQL Server

---

Overview

- Permissions and policies are stored in SQL Server.

- At app startup, permissions are loaded from DB dynamically.
- Policies are registered programmatically based on DB data.
- User claims contain permission strings fetched from DB.
- Controllers use policies by name — no need to update code when permissions change.

Step-by-step

1. Database structure for permissions

Assuming you have a **Permissions** table:

```sql
sql

CREATE TABLE Permissions (
    Id INT PRIMARY KEY IDENTITY,
    Name NVARCHAR(100) NOT NULL UNIQUE -- e.g., "Create", "Edit", "Delete", "Export"
);
```

---

2. Load permissions from DB on startup

Define an interface for permission repository:

```csharp
csharp

public interface IPermissionRepository
{
    Task<IEnumerable<string>> GetAllPermissionsAsync();
}
```

Implementation example with Entity Framework:

```csharp
csharp

public class PermissionRepository : IPermissionRepository
{
    private readonly YourDbContext _context;

    public PermissionRepository(YourDbContext context)
    {
        _context = context;
    }

    public async Task<IEnumerable<string>> GetAllPermissionsAsync()
    {
        return await _context.Permissions.Select(p => p.Name).ToListAsync();
    }
}
```

---

3. Register authorization policies dynamically

In `Program.cs` or `Startup.cs`:

```csharp
builder.Services.AddScoped<IPermissionRepository, PermissionRepository>();

builder.Services.AddAuthorization(); // Add Authorization services first

// Then after building service provider to get the repository instance
var serviceProvider = builder.Services.BuildServiceProvider();

var permissionRepo = serviceProvider.GetRequiredService<IPermissionRepository>();
var permissions = await permissionRepo.GetAllPermissionsAsync();

builder.Services.AddAuthorization(options =>
{
    foreach (var permission in permissions)
    {
        options.AddPolicy(permission, policy =>
            policy.RequireClaim("Permission", permission));
    }
});
```

> **Note:** The above snippet may require you to restructure your program startup to allow async calls before building the host, or preload permissions from DB in a separate step.

---

4. Add permission claims to users dynamically

When authenticating users (e.g., in your login method), fetch their permissions from DB and add as claims:

```csharp
var userPermissions = await _permissionRepository.GetUserPermissionsAsync(userId);

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

// Add all permission claims dynamically
foreach (var perm in userPermissions)
{
    claims.Add(new Claim("Permission", perm));
}
```

---

5. Use policy names directly in controllers

Since policies are created dynamically based on DB permission names, just reference them by name:

```csharp
[Authorize(Policy = "Edit")]
[HttpPut("{id}")]
public IActionResult UpdateItem(int id, ItemDto item)
{
    // update logic
    return Ok();
}
```

If later you add a new permission `Export` in DB and assign it, it will automatically be available as `[Authorize(Policy = "Export")]` without code changes.

---

6. Optional: Refresh policies at runtime

If you want to refresh policies without restarting the app, consider:

- Creating a custom `IAuthorizationPolicyProvider` that loads policies from DB on demand.
- This enables truly dynamic policies.

Example skeleton:

```
csharp

public class DynamicAuthorizationPolicyProvider : IAuthorizationPolicyProvider
{
    private readonly IPermissionRepository _permissionRepository;
    private readonly DefaultAuthorizationPolicyProvider _fallbackPolicyProvider;

    public DynamicAuthorizationPolicyProvider(IConfiguration configuration,
IPermissionRepository permissionRepository)
    {
        var options = Options.Create(new AuthorizationOptions());
        _fallbackPolicyProvider = new DefaultAuthorizationPolicyProvider(options);
        _permissionRepository = permissionRepository;
    }

    public async Task<AuthorizationPolicy> GetPolicyAsync(string policyName)
    {
        var permissions = await _permissionRepository.GetAllPermissionsAsync();
        if (permissions.Contains(policyName))
        {
            var policy = new AuthorizationPolicyBuilder()
                .RequireClaim("Permission", policyName)
                .Build();
            return policy;
        }
        return await _fallbackPolicyProvider.GetPolicyAsync(policyName);
    }

    public Task<AuthorizationPolicy> GetDefaultPolicyAsync()
    {
        return _fallbackPolicyProvider.GetDefaultPolicyAsync();
    }
}
```

Register in `Program.cs`:

```
csharp

builder.Services.AddSingleton<IAuthorizationPolicyProvider, DynamicAuthorizationPolicyProvider>
();
```

Summary

Feature	How to achieve
Dynamic permissions from DB	Load permissions from SQL Server
Dynamic policy registration	Register policies on startup from DB

Feature	How to achieve
User claims for permissions	Add permission claims on login
Controllers authorize by name	Use `[Authorize(Policy = "Permission")]`
Runtime dynamic policies (optional)	Custom `IAuthorizationPolicyProvider`

If you want, I can help with a **full working example project** or focus on the **dynamic policy provider** implementation next. Would that help?

Yes

Great! I'll provide a **minimal but complete example** showing how to implement a **dynamic policy provider** with claim-based authorization, loading permissions from SQL Server.

Complete Example: Dynamic Authorization Policy Provider in .NET Core Web API

1. Setup Entity Framework Models

```
csharp

public class Permission
{
    public int Id { get; set; }
    public string Name { get; set; }
}
public class ApplicationDbContext : DbContext
{
    public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options) : base(options)
{}
    public DbSet<Permission> Permissions { get; set; }
}
```

2. Permission Repository

```
csharp

public interface IPermissionRepository
{
    Task<IEnumerable<string>> GetAllPermissionsAsync();
}

public class PermissionRepository : IPermissionRepository
{
    private readonly ApplicationDbContext _context;

    public PermissionRepository(ApplicationDbContext context)
    {
        _context = context;
    }
```

```
    public async Task<IEnumerable<string>> GetAllPermissionsAsync()
    {
        return await _context.Permissions.Select(p => p.Name).ToListAsync();
    }
}
```

3. Dynamic Authorization Policy Provider

```
csharp

public class DynamicAuthorizationPolicyProvider : IAuthorizationPolicyProvider
{
    private readonly IPermissionRepository _permissionRepository;
    private readonly DefaultAuthorizationPolicyProvider _fallbackPolicyProvider;

    public DynamicAuthorizationPolicyProvider(IOptions<AuthorizationOptions> options,
IPermissionRepository permissionRepository)
    {
        _permissionRepository = permissionRepository;
        _fallbackPolicyProvider = new DefaultAuthorizationPolicyProvider(options);
    }

    public async Task<AuthorizationPolicy> GetPolicyAsync(string policyName)
    {
        var permissions = await _permissionRepository.GetAllPermissionsAsync();

        if (permissions.Contains(policyName))
        {
            var policy = new AuthorizationPolicyBuilder()
                .RequireClaim("Permission", policyName)
                .Build();
            return policy;
        }

        return await _fallbackPolicyProvider.GetPolicyAsync(policyName);
    }

    public Task<AuthorizationPolicy> GetDefaultPolicyAsync()
    {
        return _fallbackPolicyProvider.GetDefaultPolicyAsync();
    }

    public Task<AuthorizationPolicy> GetFallbackPolicyAsync()
    {
        return _fallbackPolicyProvider.GetFallbackPolicyAsync();
    }
}
```

4. Register Services in `Program.cs` (or `Startup.cs`)

```
csharp

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection")));

builder.Services.AddScoped<IPermissionRepository, PermissionRepository>();

builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
    .AddJwtBearer(options => {
        // configure JWT options here...
    });

builder.Services.AddAuthorization();
```

```
builder.Services.AddSingleton<IAuthorizationPolicyProvider, DynamicAuthorizationPolicyProvider>
();

var app = builder.Build();

app.UseAuthentication();
app.UseAuthorization();

app.MapControllers();

app.Run();
```

5. Adding Permission Claims to User Token (e.g., in Login Endpoint)

When issuing JWT tokens after user login, fetch permissions and add as claims:

```
csharp

var userPermissions = await permissionRepository.GetUserPermissionsAsync(userId);

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

foreach (var perm in userPermissions)
{
    claims.Add(new Claim("Permission", perm));
}

// Create JWT token with claims here...
```

6. Protect Controller Actions Using Dynamic Policies

```
csharp

[Authorize(Policy = "Edit")]
[HttpPut("{id}")]
public IActionResult EditItem(int id, ItemDto dto)
{
    return Ok($"Edited item {id}");
}

[Authorize(Policy = "Export")]
[HttpGet("export")]
public IActionResult ExportData()
{
    return Ok("Exported data");
}
```

How it works

- When a request arrives, **`.NET`** will ask the **`DynamicAuthorizationPolicyProvider`** for the policy named **`"Edit"`** (or any other).
- The provider checks the DB for that permission name.
- If found, it creates a policy requiring the **`"Permission"`** claim with that name.

- The user's JWT token must have that claim to access the endpoint.

Next Steps / Enhancements

- Implement caching in `DynamicAuthorizationPolicyProvider` to avoid DB calls on every request.
- Add role support or resource-based authorization if needed.
- Secure your permissions management endpoints to allow admins to add/remove permissions dynamically.

If you want, I can help generate a **GitHub repo skeleton** with this setup or code samples for JWT generation and seeding permissions in the DB — just say the word!

> Yes

Awesome! Below is a **GitHub-style repo skeleton** outline with sample files and code snippets for:

- EF Core setup + seed permissions
- JWT token generation with permission claims
- DynamicAuthorizationPolicyProvider
- Sample controller with policy-based authorization

Repo Skeleton: `DynamicPermissionsDemo`

```
pgsql

/DynamicPermissionsDemo
│
├── Controllers
│   └── ItemsController.cs
│
├── Data
│   ├── ApplicationDbContext.cs
│   └── DbSeeder.cs
│
├── Authorization
│   └── DynamicAuthorizationPolicyProvider.cs
│
├── Models
│   └── Permission.cs
│
├── Services
│   └── PermissionRepository.cs
│
├── Program.cs
├── appsettings.json
├── README.md
└── DynamicPermissionsDemo.csproj
```

---

1. Models/Permission.cs

```
csharp

namespace DynamicPermissionsDemo.Models;

public class Permission
{
    public int Id { get; set; }
    public string Name { get; set; }
}
```

2. Data/ApplicationDbContext.cs

```
csharp

using DynamicPermissionsDemo.Models;
using Microsoft.EntityFrameworkCore;

namespace DynamicPermissionsDemo.Data;

public class ApplicationDbContext : DbContext
{
    public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options) : base(options)
{ }

    public DbSet<Permission> Permissions { get; set; }
}
```

3. Data/DbSeeder.cs

```
csharp

using DynamicPermissionsDemo.Models;

namespace DynamicPermissionsDemo.Data;

public static class DbSeeder
{
    public static async Task SeedAsync(ApplicationDbContext context)
    {
        if (!context.Permissions.Any())
        {
            var permissions = new[]
            {
                new Permission { Name = "Create" },
                new Permission { Name = "View" },
                new Permission { Name = "Edit" },
                new Permission { Name = "Delete" },
                new Permission { Name = "Export" },
                new Permission { Name = "Print" }
            };

            context.Permissions.AddRange(permissions);
            await context.SaveChangesAsync();
        }
    }
}
```

4. Services/PermissionRepository.cs

```
csharp


using DynamicPermissionsDemo.Data;
using Microsoft.EntityFrameworkCore;

namespace DynamicPermissionsDemo.Services;

public interface IPermissionRepository
{
    Task<IEnumerable<string>> GetAllPermissionsAsync();
}

public class PermissionRepository : IPermissionRepository
{
    private readonly ApplicationDbContext _context;

    public PermissionRepository(ApplicationDbContext context)
    {
        _context = context;
    }

    public async Task<IEnumerable<string>> GetAllPermissionsAsync()
    {
        return await _context.Permissions.Select(p => p.Name).ToListAsync();
    }
}
```

5. Authorization/DynamicAuthorizationPolicyProvider.cs

```
csharp


using Microsoft.AspNetCore.Authorization;
using Microsoft.Extensions.Options;
using DynamicPermissionsDemo.Services;

namespace DynamicPermissionsDemo.Authorization;

public class DynamicAuthorizationPolicyProvider : IAuthorizationPolicyProvider
{
    private readonly IPermissionRepository _permissionRepository;
    private readonly DefaultAuthorizationPolicyProvider _fallbackPolicyProvider;

    public DynamicAuthorizationPolicyProvider(IOptions<AuthorizationOptions> options,
IPermissionRepository permissionRepository)
    {
        _permissionRepository = permissionRepository;
        _fallbackPolicyProvider = new DefaultAuthorizationPolicyProvider(options);
    }

    public async Task<AuthorizationPolicy> GetPolicyAsync(string policyName)
    {
        var permissions = await _permissionRepository.GetAllPermissionsAsync();

        if (permissions.Contains(policyName))
        {
            return new AuthorizationPolicyBuilder()
                .RequireClaim("Permission", policyName)
                .Build();
        }

        return await _fallbackPolicyProvider.GetPolicyAsync(policyName);
    }

    public Task<AuthorizationPolicy> GetDefaultPolicyAsync()
    {
        return _fallbackPolicyProvider.GetDefaultPolicyAsync();
    }

    public Task<AuthorizationPolicy> GetFallbackPolicyAsync()
```

```
    {
        return _fallbackPolicyProvider.GetFallbackPolicyAsync();
    }
}
```

6. Controllers/ItemsController.cs

```
csharp

using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;

namespace DynamicPermissionsDemo.Controllers;

[ApiController]
[Route("api/[controller]")]
public class ItemsController : ControllerBase
{
    [Authorize(Policy = "View")]
    [HttpGet]
    public IActionResult Get()
    {
        return Ok("Viewing items");
    }

    [Authorize(Policy = "Create")]
    [HttpPost]
    public IActionResult Create()
    {
        return Ok("Created an item");
    }

    [Authorize(Policy = "Edit")]
    [HttpPut("{id}")]
    public IActionResult Edit(int id)
    {
        return Ok($"Edited item {id}");
    }

    [Authorize(Policy = "Delete")]
    [HttpDelete("{id}")]
    public IActionResult Delete(int id)
    {
        return Ok($"Deleted item {id}");
    }
}
```

7. Program.cs

```
csharp

using DynamicPermissionsDemo.Authorization;
using DynamicPermissionsDemo.Data;
using DynamicPermissionsDemo.Services;
using Microsoft.AspNetCore.Authentication.JwtBearer;
using Microsoft.EntityFrameworkCore;
using Microsoft.IdentityModel.Tokens;
using Microsoft.AspNetCore.Authorization;
using System.Text;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection")));

builder.Services.AddScoped<IPermissionRepository, PermissionRepository>();
```

```
builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
    .AddJwtBearer(options =>
    {
        options.TokenValidationParameters = new TokenValidationParameters
        {
            ValidateIssuer = true,
            ValidateAudience = true,
            ValidateLifetime = true,
            ValidateIssuerSigningKey = true,
            ValidIssuer = builder.Configuration["Jwt:Issuer"],
            ValidAudience = builder.Configuration["Jwt:Audience"],
            IssuerSigningKey = new
SymmetricSecurityKey(Encoding.UTF8.GetBytes(builder.Configuration["Jwt:Key"]))
        };
    });

builder.Services.AddAuthorization();

builder.Services.AddSingleton<IAuthorizationPolicyProvider, DynamicAuthorizationPolicyProvider>
();

builder.Services.AddControllers();

var app = builder.Build();

// Seed database permissions
using (var scope = app.Services.CreateScope())
{
    var dbContext = scope.ServiceProvider.GetRequiredService<ApplicationDbContext>();
    await DbSeeder.SeedAsync(dbContext);
}

app.UseAuthentication();
app.UseAuthorization();

app.MapControllers();

app.Run();
```

8. appsettings.json

```
json


{
  "ConnectionStrings": {
    "DefaultConnection":
"Server=YOUR_SERVER;Database=DynamicPermissionsDb;Trusted_Connection=True;"
  },
  "Jwt": {
    "Issuer": "yourissuer.com",
    "Audience": "youraudience.com",
    "Key": "Your_Secret_Key_Here_ChangeThisToSomethingSecure"
  },
  "Logging": {
    "LogLevel": {
      "Default": "Information"
    }
  },
  "AllowedHosts": "*"
}
```

9. JWT Token Generation Snippet (Example)

In your login or token service:

```csharp
var userPermissions = await permissionRepository.GetUserPermissionsAsync(userId);

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

foreach (var perm in userPermissions)
{
    claims.Add(new Claim("Permission", perm));
}

var key = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(configuration["Jwt:Key"]));
var creds = new SigningCredentials(key, SecurityAlgorithms.HmacSha256);

var token = new JwtSecurityToken(
    issuer: configuration["Jwt:Issuer"],
    audience: configuration["Jwt:Audience"],
    claims: claims,
    expires: DateTime.UtcNow.AddHours(1),
    signingCredentials: creds);

string tokenString = new JwtSecurityTokenHandler().WriteToken(token);
```

Ready to Run

- Run migrations and create the DB.
- Seed permissions run automatically at startup.
- Generate JWT tokens including permission claims.
- Protect controllers dynamically based on DB permissions.

If you want, I can also help with:

- EF Core migrations and commands
- Full token authentication/login controller sample
- Postman or HTTP client request examples for testing

Just let me know!