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.

Use MediatR and write full project.

Models:
Users
Roles
Permissions
RolePermissions
UserRoles

Got it! You want a **full .NET Core Web API project** that implements **claim-based authorization** using **roles and permissions** stored in SQL Server, enforcing policies in the API with **MediatR** for command/query handling.

Summary of What We'll Build

- Models: **`Users`**, **`Roles`**, **`Permissions`**, **`RolePermissions`**, **`UserRoles`**
- Store permissions in SQL Server.
- Role-based + claim-based authorization (each permission is a claim).
- Policies based on permissions.
- Use MediatR for CQRS pattern (commands & queries).
- Basic CRUD + advanced permissions (Create, View, Add, Edit, Delete, Export, Print).
- Authentication is assumed (e.g. JWT), focus on Authorization.

Let's do a minimal working project structure step by step

---

1. Setup and Dependencies

- .NET 7 or 8 Web API project
- Entity Framework Core with SQL Server provider
- MediatR

- Microsoft.AspNetCore.Authentication.JwtBearer (for JWT support)
- Microsoft.AspNetCore.Authorization

2. Project Setup

```bash

dotnet new webapi -n PermissionDemo
cd PermissionDemo

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

3. Models

```csharp

public class User
{
    public int Id { get; set; }
    public string Username { get; set; } = null!;
    public string PasswordHash { get; set; } = null!;  // for simplicity
    public ICollection<UserRole> UserRoles { get; set; } = new List<UserRole>();
}

public class Role
{
    public int Id { get; set; }
    public string Name { get; set; } = null!;
    public ICollection<UserRole> UserRoles { get; set; } = new List<UserRole>();
    public ICollection<RolePermission> RolePermissions { get; set; } = new List<RolePermission>
();
}

public class Permission
{
    public int Id { get; set; }
    public string Name { get; set; } = null!;  // e.g. "Create", "View", "Edit", "Delete",
"Export", "Print"
    public string Description { get; set; } = null!;
    public ICollection<RolePermission> RolePermissions { get; set; } = new List<RolePermission>
();
}

public class UserRole
{
    public int UserId { get; set; }
    public User User { get; set; } = null!;
    public int RoleId { get; set; }
    public Role Role { get; set; } = null!;
}

public class RolePermission
{
    public int RoleId { get; set; }
    public Role Role { get; set; } = null!;
    public int PermissionId { get; set; }
    public Permission Permission { get; set; } = null!;
}
```

4. DbContext

```csharp
public class AppDbContext : DbContext
{
    public AppDbContext(DbContextOptions<AppDbContext> options)
        : base(options)
    {
    }

    public DbSet<User> Users => Set<User>();
    public DbSet<Role> Roles => Set<Role>();
    public DbSet<Permission> Permissions => Set<Permission>();
    public DbSet<UserRole> UserRoles => Set<UserRole>();
    public DbSet<RolePermission> RolePermissions => Set<RolePermission>();

    protected override void OnModelCreating(ModelBuilder builder)
    {
        base.OnModelCreating(builder);

        builder.Entity<UserRole>()
            .HasKey(ur => new { ur.UserId, ur.RoleId });
        builder.Entity<UserRole>()
            .HasOne(ur => ur.User)
            .WithMany(u => u.UserRoles)
            .HasForeignKey(ur => ur.UserId);
        builder.Entity<UserRole>()
            .HasOne(ur => ur.Role)
            .WithMany(r => r.UserRoles)
            .HasForeignKey(ur => ur.RoleId);

        builder.Entity<RolePermission>()
            .HasKey(rp => new { rp.RoleId, rp.PermissionId });
        builder.Entity<RolePermission>()
            .HasOne(rp => rp.Role)
            .WithMany(r => r.RolePermissions)
            .HasForeignKey(rp => rp.RoleId);
        builder.Entity<RolePermission>()
            .HasOne(rp => rp.Permission)
            .WithMany(p => p.RolePermissions)
            .HasForeignKey(rp => rp.PermissionId);
    }
}
```

---

5. Authorization Setup

- We will create **policies** dynamically from permissions stored in DB.
- Each permission corresponds to a claim of type **`"Permission"`**.

---

6. Custom Authorization Policy Provider

To dynamically create policies for each permission:

```csharp
using Microsoft.AspNetCore.Authorization;
using Microsoft.Extensions.Caching.Memory;

public class PermissionPolicyProvider : IAuthorizationPolicyProvider
{
    const string POLICY_PREFIX = "Permission:";
```

```
    private readonly DefaultAuthorizationPolicyProvider _fallbackPolicyProvider;
    private readonly IMemoryCache _cache;

    public PermissionPolicyProvider(IOptions<AuthorizationOptions> options, IMemoryCache cache)
    {
        _fallbackPolicyProvider = new DefaultAuthorizationPolicyProvider(options);
        _cache = cache;
    }

    public Task<AuthorizationPolicy?> GetPolicyAsync(string policyName)
    {
        if (policyName.StartsWith(POLICY_PREFIX, StringComparison.OrdinalIgnoreCase))
        {
            var permissionName = policyName.Substring(POLICY_PREFIX.Length);

            // Cache policies for performance
            return Task.FromResult(_cache.GetOrCreate(policyName, entry =>
            {
                entry.AbsoluteExpirationRelativeToNow = TimeSpan.FromMinutes(30);
                return new AuthorizationPolicyBuilder()
                    .RequireClaim("Permission", permissionName)
                    .Build();
            }));
        }

        // fallback to default
        return _fallbackPolicyProvider.GetPolicyAsync(policyName);
    }

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

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

7. Authentication & Claims Transformation

- Assuming JWT authentication is in place.
- We add a **`ClaimsTransformation`** service to load user permissions from DB and add them as claims.

```
csharp

public class PermissionsClaimsTransformer : IClaimsTransformation
{
    private readonly AppDbContext _dbContext;

    public PermissionsClaimsTransformer(AppDbContext dbContext)
    {
        _dbContext = dbContext;
    }

    public async Task<ClaimsPrincipal> TransformAsync(ClaimsPrincipal principal)
    {
        if (principal.Identity?.IsAuthenticated != true)
            return principal;

        var identity = (ClaimsIdentity)principal.Identity;

        // Avoid duplicate claims
        if (identity.Claims.Any(c => c.Type == "Permission"))
            return principal;
```

```
        var userIdClaim = identity.FindFirst(ClaimTypes.NameIdentifier);
        if (userIdClaim == null)
            return principal;

        if (!int.TryParse(userIdClaim.Value, out var userId))
            return principal;

        // Load permissions for user's roles
        var permissions = await (from ur in _dbContext.UserRoles
                                 where ur.UserId == userId
                                 join rp in _dbContext.RolePermissions on ur.RoleId equals
rp.RoleId
                                 join p in _dbContext.Permissions on rp.PermissionId equals p.Id
                                 select p.Name).Distinct().ToListAsync();

        foreach (var perm in permissions)
        {
            identity.AddClaim(new Claim("Permission", perm));
        }

        return principal;
    }
}
```

8. Configure `Program.cs`

```
csharp


var builder = WebApplication.CreateBuilder(args);

// Add services
builder.Services.AddDbContext<AppDbContext>(options =>
    options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection")));

builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
    .AddJwtBearer(options =>
    {
        options.Authority = "https://your-auth-server";
        options.Audience = "your-api";
        // Configure JWT validation here
    });

builder.Services.AddAuthorization(options =>
{
    // You can add default policies if needed
});

builder.Services.AddMemoryCache();

builder.Services.AddSingleton<IAuthorizationPolicyProvider, PermissionPolicyProvider>();
builder.Services.AddScoped<IClaimsTransformation, PermissionsClaimsTransformer>();

builder.Services.AddMediatR(typeof(Program));

builder.Services.AddControllers();

var app = builder.Build();

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

app.MapControllers();

app.Run();
```

9. Example Controller with Authorization by Permission

```csharp
[ApiController]
[Route("api/[controller]")]
public class DocumentsController : ControllerBase
{
    private readonly IMediator _mediator;

    public DocumentsController(IMediator mediator)
    {
        _mediator = mediator;
    }

    [HttpGet]
    [Authorize(Policy = "Permission:View")]
    public async Task<IActionResult> Get()
    {
        var docs = await _mediator.Send(new GetDocumentsQuery());
        return Ok(docs);
    }

    [HttpPost]
    [Authorize(Policy = "Permission:Create")]
    public async Task<IActionResult> Create(CreateDocumentCommand command)
    {
        var result = await _mediator.Send(command);
        return CreatedAtAction(nameof(GetById), new { id = result.Id }, result);
    }

    [HttpGet("{id}")]
    [Authorize(Policy = "Permission:View")]
    public async Task<IActionResult> GetById(int id)
    {
        var doc = await _mediator.Send(new GetDocumentByIdQuery(id));
        if (doc == null) return NotFound();
        return Ok(doc);
    }

    // Similarly for Edit, Delete, Export, Print with appropriate policies
}
```

10. Sample MediatR Request & Handler

Example **`GetDocumentsQuery`**:

```csharp
public record GetDocumentsQuery : IRequest<List<DocumentDto>>;

public class GetDocumentsHandler : IRequestHandler<GetDocumentsQuery, List<DocumentDto>>
{
    private readonly AppDbContext _context;

    public GetDocumentsHandler(AppDbContext context)
    {
        _context = context;
    }

    public async Task<List<DocumentDto>> Handle(GetDocumentsQuery request, CancellationToken
cancellationToken)
    {
        // Replace with actual entity and DTO mapping
        return await _context.Set<Document>().Select(d => new DocumentDto
        {
            Id = d.Id,
            Title = d.Title,
            Content = d.Content
        }).ToListAsync(cancellationToken);
```

```
    }
}
```

11. Database Seeding (Optional)

Seed roles, permissions, and their mappings for easy setup.

```
csharp

public static class SeedData
{
    public static void Initialize(AppDbContext context)
    {
        if (context.Permissions.Any()) return;

        var permissions = new[]
        {
            new Permission { Name = "Create", Description = "Create items" },
            new Permission { Name = "View", Description = "View items" },
            new Permission { Name = "Edit", Description = "Edit items" },
            new Permission { Name = "Delete", Description = "Delete items" },
            new Permission { Name = "Export", Description = "Export items" },
            new Permission { Name = "Print", Description = "Print items" }
        };
        context.Permissions.AddRange(permissions);

        var adminRole = new Role { Name = "Admin" };
        var userRole = new Role { Name = "User" };

        context.Roles.AddRange(adminRole, userRole);
        context.SaveChanges();

        context.RolePermissions.AddRange(
            permissions.Select(p => new RolePermission { RoleId = adminRole.Id, PermissionId =
p.Id })
        );

        context.RolePermissions.Add(
            new RolePermission { RoleId = userRole.Id, PermissionId = permissions.First(p =>
p.Name == "View").Id }
        );

        context.SaveChanges();
    }
}
```

Call this in **`Program.cs`** after building the app:

```
csharp

using (var scope = app.Services.CreateScope())
{
    var db = scope.ServiceProvider.GetRequiredService<AppDbContext>();
    db.Database.Migrate();
    SeedData.Initialize(db);
}
```

Summary

This setup gives you:

- SQL Server database storing users, roles, permissions.
- Claims-based policies dynamically created for each permission.
- User permissions loaded into claims on every request.
- Authorization using policies like `[Authorize(Policy = "Permission:Create")]`.
- MediatR pattern for request handling.
- Sample data seeding to bootstrap roles and permissions.

If you want, I can help generate:

- Complete sample project on GitHub gist.
- Detailed CQRS commands/queries.
- JWT auth setup example.
- Frontend integration ideas.

Let me know! Want me to generate a full project scaffold with everything?