

Permissions Management .NET Core



How to manage permissions, starting with basic access and including additional actions such as create, view, add, edit, delete, as well as extended options like export and print. How to implement the case in .net core web api and ms sql

Managing permissions in a **.NET Core Web API** with **MS SQL Server** typically involves designing a robust **Role-Based Access Control (RBAC)** system or a more flexible **Claim/Permission-based authorization** system.

Below is a detailed guide for implementing a flexible permission-based model supporting basic and extended actions like **create, view, add, edit, delete, export, print**, etc.

◆ 1. Database Design (MS SQL)

Start by modeling your permission system:

✓ Tables:

Users

```
sql

CREATE TABLE Users (
    Id INT PRIMARY KEY IDENTITY,
    Username NVARCHAR(100) NOT NULL,
    PasswordHash NVARCHAR(MAX),
    -- Other fields like email, etc.
);
```

Roles

```
sql

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

Permissions

```
sql

CREATE TABLE Permissions (
    Id INT PRIMARY KEY IDENTITY,
    Name NVARCHAR(100) NOT NULL UNIQUE -- e.g., 'User.Create', 'User.View', etc.
);
```

UserRoles

```
sql

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

RolePermissions

```
sql

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

◆ 2. .NET Core Backend Setup

✓ Models

```
csharp

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

public class Role
{
    public int Id { get; set; }
    public string Name { get; set; }
    public ICollection<RolePermission> RolePermissions { get; set; }
}

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

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

```
public int PermissionId { get; set; }
public Permission Permission { get; set; }
}
```

◆ 3. Seeding Permissions

Create a seed list of permissions:

```
csharp

var permissions = new List<string>
{
    "User.Create",
    "User.View",
    "User.Add",
    "User.Edit",
    "User.Delete",
    "User.Export",
    "User.Print"
};
```

Use EF Core to insert these into the **Permissions** table.

◆ 4. Authorization Logic

✓ Custom Authorization Attribute

```
csharp

public class HasPermissionAttribute : AuthorizeAttribute, IAuthorizationFilter
{
    private readonly string _permission;

    public HasPermissionAttribute(string permission)
    {
        _permission = permission;
    }

    public void OnAuthorization(AuthorizationFilterContext context)
    {
        var userId = context.HttpContext.User.FindFirst(ClaimTypes.NameIdentifier)?.Value;

        if (string.IsNullOrEmpty(userId))
        {
            context.Result = new ForbidResult();
            return;
        }

        // Access your DB context (use scoped services or a service locator)
        var db = context.HttpContext.RequestServices.GetRequiredService<ApplicationDbContext>();

        var hasPermission = db.UserRoles
            .Where(ur => ur.UserId == int.Parse(userId))
            .SelectMany(ur => ur.Role.RolePermissions)
            .Any(rp => rp.Permission.Name == _permission);

        if (!hasPermission)
        {
            context.Result = new ForbidResult();
        }
    }
}
```

```
}
}
```

◆ 5. Usage in Controller

```
csharp

[HttpPost]
[HasPermission("User.Create")]
public IActionResult CreateUser(UserDto user)
{
    // Logic here
    return Ok();
}

[HttpGet]
[HasPermission("User.View")]
public IActionResult GetUser(int id)
{
    // Logic here
    return Ok();
}
```

◆ 6. Authentication Setup

Use **JWT-based authentication** where the token includes the user's ID. From there, you can pull roles and permissions dynamically from the database.

◆ 7. Optional: Caching Permissions (Performance)

For performance, consider caching permissions in memory (e.g., using **IMemoryCache** or Redis) so you don't query the database every request.

◆ Summary

Feature	Implemented Via
User Auth	JWT, ASP.NET Core Identity
Role System	Roles, UserRoles
Permissions	Permissions, RolePermissions
Enforcement	Custom Attribute [HasPermission("X")]

If you'd like a working **GitHub sample project**, or help generating the EF Core migrations and seeding, I can assist with that too.



Yes

Here's a refined solution to implement a robust permission system in a .NET Core Web API with MS SQL, covering basic actions (*create, view, add, edit, delete*) and extended ones (*export, print*).

1. Data Model in MS SQL

Use a flexible table structure to store resources and actions:

```
sql

CREATE TABLE Users (
  Id INT PRIMARY KEY IDENTITY,
  Username NVARCHAR(100) NOT NULL
);

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

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

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

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

This RBAC structure supports arbitrary actions and resources.

2. Seed Permissions & Roles

In EF Core `OnModelCreating` or a seed script, insert all operations:

```
csharp

var actions = new[] { "Create", "View", "Add", "Edit", "Delete", "Export", "Print" };
var resources = new[] { "User", "Product", /* etc. */ };
foreach (var r in resources)
  foreach (var a in actions)
    permissions.Add(new Permission { Name = $"{r}.{a}" });
```

Also create core roles like "Admin", "Editor" etc., and assign permissions.

3. Authentication & Claims

When a user logs in, include their permissions as claims in the JWT:

csharp

```
// fetch perms list from DB for user
var claims = perms.Select(p => new Claim("permission", p.Name));
var identity = new ClaimsIdentity(claims, JwtBearerDefaults.AuthenticationScheme);
```

4. Policy-Based Authorization

Instead of custom attributes, use dynamic policies via a **PermissionPolicyProvider** and 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 ctx,
        PermissionRequirement req) {
        if (ctx.User.HasClaim("permission", req.Permission))
            ctx.Succeed(req);
        return Task.CompletedTask;
    }
}
```

Register the provider that builds a policy dynamically:

csharp

```
services.AddSingleton<IAuthorizationPolicyProvider, PermissionPolicyProvider>();
services.AddScoped<IAuthorizationHandler, PermissionHandler>();
```

Where **PermissionPolicyProvider** auto-creates policies based on policy strings like "User.Create"
codewithmukesh.com+10 [RedditReddit](#).

5. Applying Attributes

csharp

```
[Authorize(Policy = "User.Create")]
[HttpPost("users")]
public IActionResult CreateUser(UserDto dto) { ... }
```

This ensures only users with that specific permission can access the endpoint.

6. View-Level Checks (Optional)

In server-side UI or dashboards, you can use:

csharp

```
@if ((await AuthorizationService.AuthorizeAsync(User, "Product.Print")).Succeeded)
{
    <button>Print</button>
}
```

Filtered rendering based on permission claims [Reddit](#).

7. Best Practices and Performance

- **Caching:** Cache user permissions per request/session.
- **Dynamic discovery:** Tools like BlazorPlate auto-scan API actions to populate permissions [Reddit](#).
- **JWT vs DB:** Use claims in JWT for speed, but refresh tokens or use short TTL to reflect DB changes (see discussion on JWT limitations) [Reddit](#).
- **Policy combining:** You can stack or combine policies (e.g., all authenticated + custom claim) [Reddit](#).

Summary Table

Layer	Implementation
DB	Users, Roles, Permissions, Mappings
Auth	JWT with permission claims
Authorization	Custom policies and handlers
Controller	<code>[Authorize(Policy = "X.Y")]</code>
View/UI	<code>AuthorizationService.AuthorizeAsync</code>

Example Project

Most steps and full source code are available in tools like *CodeWithMukesh* (.NET 5 tutorial)

[codewithmukesh.com+8](#)

[DEV Community+3](#)

[Reddit+14](#)

, and Microsoft's policy-based auth samples

[Microsoft Learn+15](#)

Would you like a minimal GitHub starter with EF migrations, dynamic policy provider, seeding, and sample controller + view? Let me know and I'll generate it!