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
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
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	`[Authorize(Policy = "X.Y")]`
View/UI	`AuthorizationService.AuthorizeAsync`

🔪 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!

> how to use claim policy in it

To use **Claim-based Policies** in your .NET Core Web API authorization system (along with the permission model), you define **policies based on user claims**, such as `"permission"` claims issued at login.

🔐 What is Claim-based Policy?

It means access is granted based on a user having a **specific claim**, like:

```
json

{
  "sub": "1",
  "username": "admin",
  "permission": ["User.Create", "User.Edit"]
}
```

.NET Core allows you to enforce policies such as:

```
csharp

[Authorize(Policy = "User.Create")]
```

Which checks if the current user has a `"permission"` claim with value `"User.Create"`.

✅ Step-by-Step Implementation

🧱 1. Add Claims to JWT During Login

When authenticating, fetch the user's permissions from the DB and embed them in the JWT as multiple `permission` claims.

```
csharp

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

foreach (var perm in userPermissions)
{
    claims.Add(new Claim("permission", perm.Name)); // e.g., "User.Create"
}

var token = new JwtSecurityToken(
    issuer: "your-app",
    audience: "your-app",
    claims: claims,
    expires: DateTime.UtcNow.AddHours(1),
    signingCredentials: creds);
```

- Use `"permission"` as the claim type, or customize it.

🧩 2. Define Policies Based on Claims

You define a custom **authorization policy provider** so you can create policies dynamically (instead of hardcoding each one in `Startup.cs`).

✅ Custom Policy Provider

```
csharp

public class PermissionPolicyProvider : IAuthorizationPolicyProvider
{
    public DefaultAuthorizationPolicyProvider FallbackPolicyProvider { get; }

    public PermissionPolicyProvider(IOptions<AuthorizationOptions> options)
    {
        FallbackPolicyProvider = new DefaultAuthorizationPolicyProvider(options);
    }

    public Task<AuthorizationPolicy> GetPolicyAsync(string policyName)
    {
        // Build a policy dynamically for the given permission
        var policy = new AuthorizationPolicyBuilder()
            .RequireAuthenticatedUser()
            .AddRequirements(new PermissionRequirement(policyName)) // e.g., "User.Create"
            .Build();

        return Task.FromResult(policy);
    }

    public Task<AuthorizationPolicy> GetDefaultPolicyAsync() =>
FallbackPolicyProvider.GetDefaultPolicyAsync();
    public Task<AuthorizationPolicy?> GetFallbackPolicyAsync() =>
FallbackPolicyProvider.GetFallbackPolicyAsync();
}
```

✅ Authorization Requirement & 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("permission", requirement.Permission))
        {
            context.Succeed(requirement);
        }
        return Task.CompletedTask;
    }
}
```

🧱 3. Register in `Startup.cs` or `Program.cs`

```
csharp

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

This allows **`[Authorize(Policy = "User.Create")]`** to dynamically check the **`"permission"`** claim.

🔪 4. Use in Controller

```csharp
[Authorize(Policy = "User.Create")]
[HttpPost("users")]
public IActionResult CreateUser(UserDto dto)
{
    // Logic here
    return Ok();
}
```

Now, this action is protected, and only users with the `"permission": "User.Create"` claim can access it.

🧠 Tips

Task	Method
Dynamic permissions?	Just add them to the DB and JWT
Role support too?	Add a `"role"` claim and create parallel policies
Performance	Cache claims in JWT to avoid DB hit per request
Claim list large?	Use short JWT TTL or reference tokens (e.g., Redis)

✅ Summary

Layer	Implementation
Claim Type	`"permission"`
Policy	Matches permission value
Auth Check	`[Authorize(Policy = "X.Y")]`
Token	JWT contains multiple `"permission"` claims
Dynamic	Yes, via `IAuthorizationPolicyProvider`

Would you like a full working starter project with this permission + JWT + policy setup scaffolded? I can generate one or share a GitHub repo template.

> how to implement this in react typescript?

To implement **claim-based permissions** in a **React + TypeScript** frontend, you'll need to:

✅ Goal

- Read **JWT claims** (like `permission: User.Create`, etc.)
- Store and use them for UI rendering and route guarding
- Optionally guard routes using a wrapper or component

📦 1. Decode JWT and Extract Claims

Install a JWT decoding library:

```
bash

npm install jwt-decode
```

`auth.ts` (Utility to read token and claims)

```
ts

import jwtDecode from "jwt-decode";

export interface DecodedToken {
  sub: string;
  username: string;
  permission?: string[]; // or just 'permission' if it's a string
  exp: number;
}

export function getDecodedToken(): DecodedToken | null {
  const token = localStorage.getItem("accessToken");
  if (!token) return null;

  try {
    return jwtDecode<DecodedToken>(token);
  } catch {
    return null;
  }
}

export function hasPermission(permission: string): boolean {
  const decoded = getDecodedToken();
  if (!decoded?.permission) return false;

  const userPermissions = Array.isArray(decoded.permission)
    ? decoded.permission
    : [decoded.permission];

  return userPermissions.includes(permission);
}
```

🧩 2. Guard Components and Routes

💡 Protect Buttons or UI Sections

```
tsx

import { hasPermission } from './auth';

const UserManagement = () => (
  <>
    <h2>Users</h2>

    {hasPermission("User.Create") && (
      <button>Create User</button>
    )}

    {hasPermission("User.View") ? (
      <p>You can view users.</p>
    ) : (
```

```
      <p>No permission to view users.</p>
    )}
  </>
);
```

🔐 Guard Routes

You can create a **`ProtectedRoute`** wrapper:

```tsx
// ProtectedRoute.tsx
import React from "react";
import { Navigate } from "react-router-dom";
import { hasPermission } from "./auth";

interface ProtectedRouteProps {
  permission: string;
  children: React.ReactNode;
}

export const ProtectedRoute: React.FC<ProtectedRouteProps> = ({
  permission,
  children,
}) => {
  return hasPermission(permission) ? (
    <>{children}</>
  ) : (
    <Navigate to="/unauthorized" replace />
  );
};
```

Use it in your Router

```tsx
<Routes>
  <Route
    path="/admin"
    element={
      <ProtectedRoute permission="Admin.Access">
        <AdminPage />
      </ProtectedRoute>
    }
  />
</Routes>
```

💡 Bonus: Global Context or Auth Hook

You can also create a **React Context** to hold the token, decoded claims, and permissions.

```tsx
const AuthContext = createContext<DecodedToken | null>(null);
```

Then wrap your app in an **`<AuthProvider>`** that decodes the token once and provides **`hasPermission`**.

📄 Example JWT Payload

Make sure your backend includes this structure in JWT:

```
json

{
  "sub": "1",
  "username": "admin",
  "permission": ["User.Create", "User.View", "User.Print"],
  "exp": 1719632400
}
```

✅ Summary

Feature	Implementation
Decode JWT	**`jwt-decode`**
Check permission	**`hasPermission('User.Create')`**
Guard UI elements	Conditional rendering
Guard routes	**`<ProtectedRoute permission="User.Create" />`**
Store token	**`localStorage`** or **`sessionStorage`**

Would you like a working React + TypeScript + Role-based permissions boilerplate project or repo? I can generate that for you too.