

Permission Management Model



how to handle permissions like basic with additional like create, view, add, edit, delete and additional like export, print etc

Handling permissions in an application where users can have basic permissions (like create, view, edit, delete) and additional permissions (like export, print, share, etc.) requires a structured approach to permission management. Here's a practical way to design and manage this:



1. Define a Permission Model

Start by creating a standard structure for your permissions. For example:

Permission Categories:

- **Basic:** create, view, edit, delete
- **Advanced/Additional:** export, print, share, archive, etc.

These can be grouped per **entity** (e.g., Document, User, Report, etc.)



2. Database Schema Design (Relational DB)

Tables:

```
plaintext

Users
Roles
Permissions
RolePermissions (role_id, permission_id)
UserRoles (user_id, role_id)
```

Permissions Table Example:

id	entity	action	description
1	document	view	View document
2	document	edit	Edit document
3	document	delete	Delete document
4	document	export	Export document
5	report	print	Print report

3. Permission Key Structure

Use a structured naming convention:

```
php-template

<entity>.<action>
```

Examples:

- `document.view`
- `report.export`
- `user.delete`
- `invoice.print`

4. Assigning Permissions

- Assign **roles** to users (e.g., Admin, Editor, Viewer)
- Each role has a set of permissions
- Alternatively, assign specific permissions directly to users (if you support overrides)

5. Checking Permissions in Code

Use a utility or middleware function:

```
python

def has_permission(user, entity, action):
    permission_key = f"{entity}.{action}"
    return permission_key in user.get_all_permissions()
```

Or, with a framework like Django or Laravel:

- Use decorators or middleware to enforce permission checks.
- Create permission classes.

6. UI-Level Enforcement

In the frontend:

- Hide or disable buttons based on permissions.
- For example, disable the "Export" button if `!user.can("document.export")`

7. Best Practices

- **Least privilege:** Only grant users the minimum permissions they need.
- **Auditing:** Keep logs of permission changes and access.
- **Permission groups:** Allow grouping of permissions for simpler management.
- **Versioning:** When permissions evolve, maintain backward compatibility.

Example: JSON-based Role Definition

```
json
{
  "Admin": {
    "document": ["view", "create", "edit", "delete", "export", "print"],
    "report": ["view", "print"]
  },
  "Viewer": {
    "document": ["view"],
    "report": ["view"]
  }
}
```

Would you like an implementation example in a specific language or framework (like Django, Node.js, or Laravel)?



how to implement this in .net core web api complete example

Here's a full example of implementing fine-grained permissions (like create, view, add, edit, delete, export, print) in a .NET Core Web API using **roles and policies**:

1. Set Up Models

```
csharp
public class Permission
{
    public string Name { get; set; } // e.g. "document.view", "report.export"
}

public class Role
{
    public string Name { get; set; }
    public List<Permission> Permissions { get; set; }
}

public class User
{
    public string Username { get; set; }
    public List<Role> Roles { get; set; }
}
```

2. Service: Authentication & Claims

You'll inject a custom authentication that supplies user roles/permissions as claims.

```
csharp

public class UserService
{
    public Task<User> GetUserAsync(string username)
    {
        // Load from your DB or hard-code for demo
        return Task.FromResult(new User {
            Username = username,
            Roles = new List<Role> {
                new Role {
                    Name = "Editor",
                    Permissions = new List<Permission> {
                        new Permission { Name = "document.view" },
                        new Permission { Name = "document.create" },
                        new Permission { Name = "document.edit" },
                        new Permission { Name = "document.export" }
                    }
                }
            }
        });
    }
}
```

3. Configure Authentication & Authorization

In **Startup.cs** (or **Program.cs** for .NET 6+):

```
csharp

builder.Services.AddSingleton<UserService>();

builder.Services.AddAuthentication("Basic")
    .AddScheme<AuthenticationSchemeOptions, BasicAuthHandler>("Basic", null);

builder.Services.AddAuthorization(options =>
{
    // Create a policy for each permission
    var permissions = new[] {
        "document.view", "document.create", "document.edit",
        "document.delete", "document.export", "document.print"
    };

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

BasicAuthHandler (simplified):

```
csharp

protected override async Task<AuthenticateResult> HandleAuthenticateAsync()
{
    // Parse credentials (e.g. from headers)
    var username = "...";

    var user = await _userService.GetUserAsync(username);
}
```

```

if (user == null)
    return AuthenticateResult.Fail("Invalid user");

var claims = user.Roles
    .SelectMany(r => r.Permissions)
    .Select(p => new Claim("permission", p.Name))
    .ToList();

var identity = new ClaimsIdentity(claims, Scheme.Name);
var principal = new ClaimsPrincipal(identity);
var ticket = new AuthenticationTicket(principal, Scheme.Name);

return AuthenticateResult.Success(ticket);
}

```

4. Controller Example

Apply policies to endpoints:

```

csharp

[ApiController]
[Route("api/[controller]")]
public class DocumentsController : ControllerBase
{
    [HttpGet]
    [Authorize(Policy = "document.view")]
    public IActionResult GetAll() { /* ... */ }

    [HttpPost]
    [Authorize(Policy = "document.create")]
    public IActionResult Create([FromBody] Document doc) { /* ... */ }

    [HttpPut("{id}")]
    [Authorize(Policy = "document.edit")]
    public IActionResult Update(int id, [FromBody] Document doc) { /* ... */ }

    [HttpDelete("{id}")]
    [Authorize(Policy = "document.delete")]
    public IActionResult Delete(int id) { /* ... */ }

    [HttpGet("{id}/export")]
    [Authorize(Policy = "document.export")]
    public IActionResult Export(int id) { /* ... */ }

    [HttpGet("{id}/print")]
    [Authorize(Policy = "document.print")]
    public IActionResult Print(int id) { /* ... */ }
}

```

5. Configure Middleware

Still in `Program.cs` or `Startup.cs`:

```

csharp

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

app.MapControllers();

```





6. Frontend (Optional)

In a front-end app you would check permissions and conditionally show buttons:

```
html
```

```
<button disabled="@(!User.HasClaim("permission", "document.export") ? false : true)">Export</button>
```

7. Summary Table

Area	Implementation
 Authentication	Custom handler issues permission claims
 Authorization	Policies defined for each permission
 Controller	Use [Authorize(Policy="entity.action")]
 Extensible	Add new permissions (export, print) centrally

What Next?

- Replace demo **UserService** with your real user/role database.
- Use strong authentication (JWT, IdentityServer).
- Cache claims, or fetch from DB on login.
- Log failed authorizations for security auditing.

Let me know if you want a full working repo example, or integration with JWT, EF Core, or Identity!