DSD605 Authorization and Security

Contents

1. Setting up for this project. ... 4
   The Identity system Database ... 5
2. Customise your Login parameters ... 6
   Customise Password criteria ... 6
   Scaffold the Identity pages you want to change or investigate ... 7
   Set Email confirmed to True ... 8
   Add Authentication and Authorization Middleware ... 9
3. Roles and Claims overview ... 10
   Role Based Authorization ... 10
   Claim Based Authorization ... 10
   Where to place your claims and role policies? ... 11
   Directly at the endpoints. On the pages. ... 11
   Restrict entire folders ... 13
4. Role based Authentication ... 15
   Building the Role UI - Viewing Roles ... 16
   Create the Users and Roles interface ... 16
   Create a DTO called UserRoles ... 16
   Create the Add Roles Page ... 20
   Note: Make sure there are no spaces around your Claims and Roles. ... 22
   Add roles to RoleManager ... 23
   Assigning roles to users ... 23
   Assign “Admin” as the value to the Roles property: ... 26
   Modify the Access Denied file. ... 27
   Access the AccessDenied file by adding the file to the Identity, ... 27
5. Using policies to apply role checks ... 28
   Lock out an entire folder with a Policy ... 28
6. Claims based Authorization ... 29
   Create the Claim Manager Page ... 31
   Create the Assign Claim Page ... 33
   Working with dates….. ... 36
7. Using policies to enforce claims-based authorization ... 37
   Common methods for building simple policies ... 37
   Using assertions for more complex requirements ... 39

Users are employed more than 6 months ago example 39
Using the Policy Builder 39
8. Create a Movie database 41
Add the following classes to the Models 41
Scaffold The Classes 42
Add Data 43
Create API's 44
Add Swashbuckle / Swagger 45
Client React App 48
Run the app and connect to Movie API – CORS stops transmission 52
9. Enable Cross-Origin Requests (CORS) in ASP.NET Core 52
10. Integration Testing with XUnit 54
Testing the API's 56
Using an Integration test to test the API 56
Create an XUnit Integration Test 57
Install Nuget packages 57
Check your Packages so that you have the following: 58
Create WebApplicationFactory 59
Modify the Program.cs class 60
Getting Sample Data for the test 64
Check the Json files are registered correctly. 65
Create a class to add the data to the InMemory Database 65
Capitalise the Json Keys. 66
Pass the mock data to the WebApplicationFactory 67
Create our Integration Tests. 68
Run the tests 69
Break the Test. 69
Create a Single Entry instead of importing them all 70
11. ASP.Net Core and ReactJS security tests 72
Cross-Site Scripting (XSS) tests: 72
Cross-Site Request Forgery (CSRF) tests: 74
Session Management tests: 75
Input Validation tests: 76
Configuration Management tests: 76
Penetration Testing: 77
12. Resources 79

Cryptographic Failures 79
Storing passwords 79
Resource extension. Custom authorization requirements and handlers 81
Making it more easy to Maintain. – Create separate Handlers 85
Using Multiple Requirements 87
Resource: Modify Users at database level 91
Resources Add new fields to the ASPNetUsers table 93

Find this Exercise here https://github.com/Netchicken/RolesForAssessment

React App https://github.com/Netchicken/webapiforcorsReact

Good resource [Policy-based authorization in ASP.NET Core]

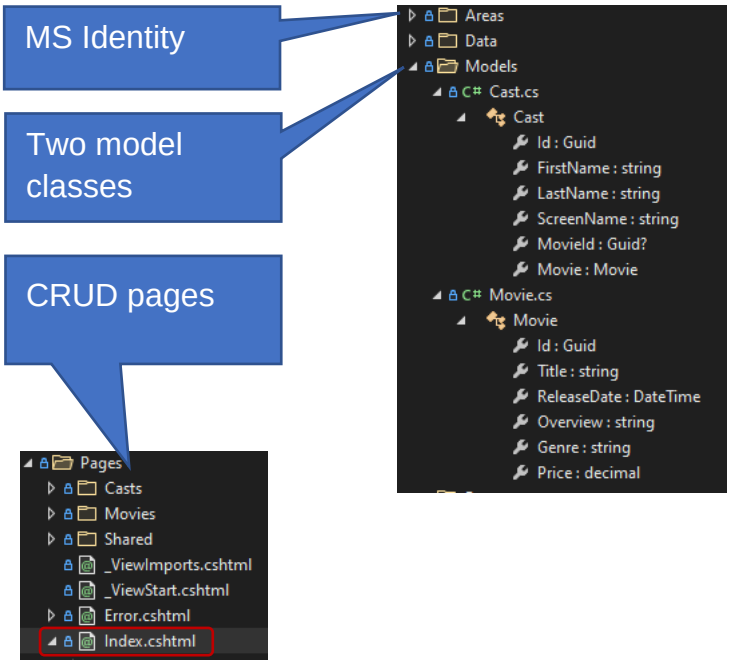
1. Setting up for this project.

You can use the **Movies Project** that you created earlier in the year for DSD603 Assessment 1 or start with a new project.

In that project we created

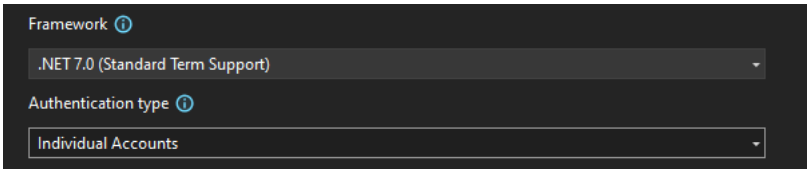
- An Identity system using MS Authentication for Individual Accounts

- A database with two tables, Movies and Casts.



You can also create a new project (I called mine RolesForAssessment, but that's because I am boring).

Make sure you add in the Authentication setting when creating it.



The instructions in this manual assume you are creating it from scratch.

The Identity system Database

When we choose Individual Accounts, and create the project, the ASP.net builds an Identity database that becomes the base for our system.

You can find the connection string to it in appsetting.json

```
"ConnectionStrings": {
  "DefaultConnection": "Server=(localdb)\\mssqllocaldb;Database=aspnet-
   RolesForAssessment-53bc9b9d-9d6a-45d4-8429-2a2761773502; Trusted_Connection=True;MultipleActiveResultSets=true"
},
```

This triggers the DbContext to build the IdentityDB

```
45 references
public class ApplicationDbContext : IdentityDbContext
{
```

We can see it in this Migration

```
Data
  Migrations
    00000000000000_CreateIdentitySchema.cs
```

That creates the following Database tables, that we will look at later

```
aspnet-RolesForAssessment-53bc9
  Tables
    System Tables
    External Tables
    dbo.__EFMigrationsHistory
    dbo.AspNetRoleClaims
    dbo.AspNetRoles
    dbo.AspNetUserClaims
    dbo.AspNetUserLogins
    dbo.AspNetUserRoles
    dbo.AspNetUsers
    dbo.AspNetUserTokens
```

We access this database to CRUD the data by injecting the ApplicationDbContext service into our code.

```
public class IndexModel : PageModel
{
    private readonly ApplicationDbContext _context;
    private readonly RoleManager<IdentityRole> _roleManager;
    public readonly UserManager<IdentityUser> _userManager;
    0 references
```

The context maps the fields from the Identity table

```
_context.u
s in _context.UserR
_context.
erRoles {
  Update
  Update<>
  UpdateRange
  UserClaims
  UserLogins
  UserRoles
  Users
  UserTokens
```

```
aspnet-RolesForAssessment-53bc9
  Tables
    System Tables
    External Tables
    dbo.__EFMigrationsHistory
    dbo.AspNetRoleClaims
    dbo.AspNetRoles
    dbo.AspNetUserClaims
    dbo.AspNetUserLogins
    dbo.AspNetUserRoles
    dbo.AspNetUsers
    dbo.AspNetUserTokens
```

2. Customise your Login parameters

Customise Password criteria

We will be creating and logging in frequently in this manual, by setting these parameters to their minimum settings it will speed your login process.

Add this to your **Program.cs** to make your passwords less onerous for debugging.

```
builder.Services.Configure<IdentityOptions>(configureOptions: options =>
{
    // Password settings.
    options.Password.RequireDigit = false;
    options.Password.RequireLowercase = false;
    options.Password.RequireNonAlphanumeric = false;
    options.Password.RequireUppercase = true;
    options.Password.RequiredLength = 6;
    options.Password.RequiredUniqueChars = 1;
    options.SignIn.RequireConfirmedEmail = false;

    // Lockout settings.
    options.Lockout.DefaultLockoutTimeSpan = TimeSpan.FromMinutes(value: 5);
    options.Lockout.MaxFailedAccessAttempts = 5;
    options.Lockout.AllowedForNewUsers = true;

    // User settings.
    options.User.AllowedUserNameCharacters =
    "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789-._@+";
    options.User.RequireUniqueEmail = false;
});
```

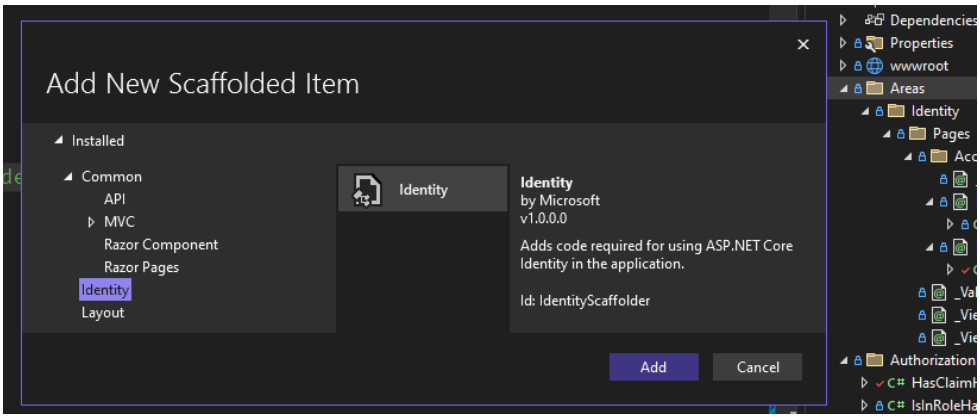
```
builder.Services.Configure<IdentityOptions>(options =>
{
    // Password settings.
    options.Password.RequireDigit = false;
    options.Password.RequireLowercase = false;
    options.Password.RequireNonAlphanumeric = false;
    options.Password.RequireUppercase = true;
    options.Password.RequiredLength = 6;
    options.Password.RequiredUniqueChars = 1;
    options.SignIn.RequireConfirmedEmail = false;

    // Lockout settings.
    options.Lockout.DefaultLockoutTimeSpan = TimeSpan.FromMinutes(5);
    options.Lockout.MaxFailedAccessAttempts = 5;
    options.Lockout.AllowedForNewUsers = true;

    // User settings.
    options.User.AllowedUserNameCharacters =
    "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789-._@+";
    options.User.RequireUniqueEmail = false;
});
```

Scaffold the Identity pages you want to change or investigate

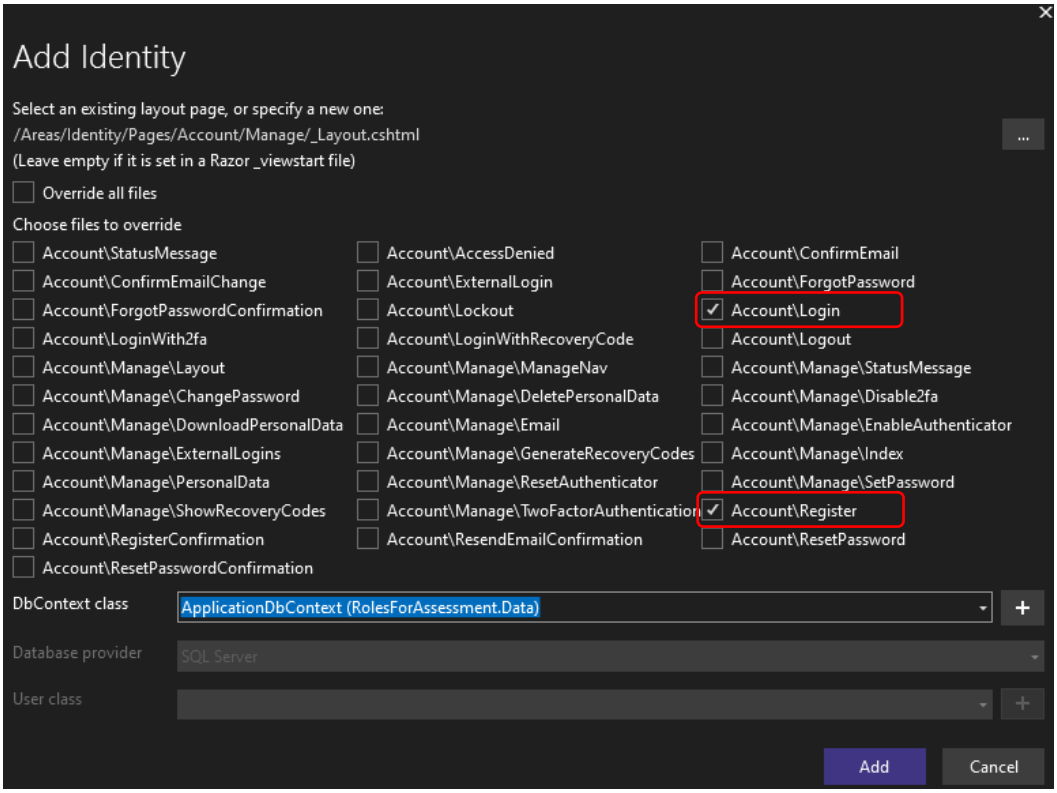
There are heaps of pages that are templates, and you can access and change them all.

Right click on Areas, go Add / New Scaffolded Item

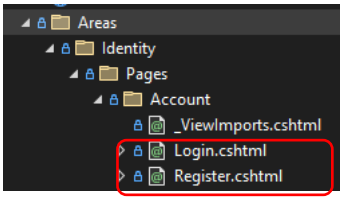
Click on Identity in the bottom left



Choose at least Account/Login and Account/Register, and give the DB Class



The pages will load under Account



You can go back to add or remove these pages at any time.

Set Email confirmed to True

When you make a new user it automatically sets the Email confirmed field to false which means that when you tried to log in with your new user you can't. It just gives an "Invalid User" error.

I had to go into this window and change it by hand. This was very frustrating ……

dbo.AspNetUsers [Data] | HasClaimHandler.cs | Program.cs | ViewRolesHandler.cs

Max Rows: 1000

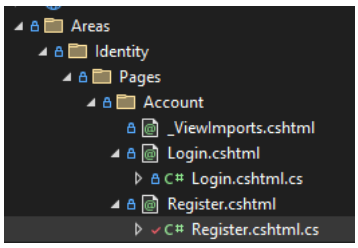
Id	UserName	NormalizedUse...	Email	NormalizedEm...	EmailConfirmed	PasswordHash
8f536075-01f3...	ccc@ccc.com	CCC@CCC.COM	ccc@ccc.com	CCC@CCC.COM	True	AQAAAAIAAYa...
ac60820f-41dc...	ddd@gmail.com	DDD@GMAIL.C...	ddd@gmail.com	DDD@GMAIL.C...	False	AQAAAAIAAYa...
b2eefcef-b4c2-...	bbb@bbb.com	BBB@BBB.COM	bbb@bbb.com	BBB@BBB.COM	True	AQAAAAIAAYa...
f0020d26-edbc...	aaa@aaa.com	AAA@AAA.COM	aaa@aaa.com	AAA@AAA.COM	True	AQAAAAIAAYa...
NULL	NULL	NULL	NULL	NULL	NULL	NULL

In Program.cs I added the following ..

```
options.SignIn.RequireConfirmedEmail = false;
options.User.RequireUniqueEmail = false;
```

But alas it didn't solve the problem.

So after googling fruitlessly I used [ChatGPT](#) and asked it "asp.net core 6 how to set emailconfirmed to true" and it gave an answer!



So with some modification go to the Register.cshtml.cs

And add in

```
user.EmailConfirmed = true; //added this line to automatically confirm email
```

```
103     public async Task<IActionResult> OnPostAsync(string returnUrl = null)
104     {
105         returnUrl ??= Url.Content(contentPath: "~/");
106         ExternalLogins = (await _signInManager.GetExternalAuthenticationSchemesAsync()).ToList();
107         if (ModelState.IsValid)
108         {
109             var user = CreateUser();
110
111             await _userStore.SetUserNameAsync(user, userName: Input.Email, cancellationToken: CancellationToken.None);
112             await _emailStore.SetEmailAsync(user, email: Input.Email, cancellationToken: CancellationToken.None);
113
114             user.EmailConfirmed = true; //added this line to automatically confirm email
```

Test it with a new user

UserName	NormalizedUserName	Email	NormalizedEm...	EmailConfirmed
emailconfirmed@xxx.com	EMAILCONFIRMED@XX...	emailconfirmed...	EMAILCONFIR...	True
ddd@ddd.com	DDD@DDD.COM	ddd@ddd.com	DDD@DDD.CO...	False

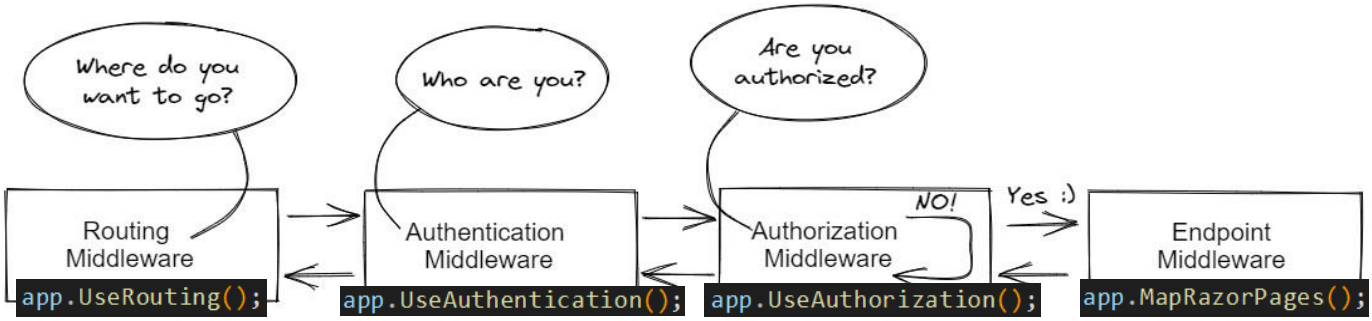
Add Authentication and Authorization Middleware

We do this by calling **app.UseAuthentication()** in the request pipeline. However, the positioning of this call is crucial to authentication working properly.

It must be placed after routing is added, but before endpoints are added by the MapRazorPages call.

```
app.UseAuthentication();
app.UseAuthorization();//Authorization middleware is enabled by
   default in the web application template by the inclusion of
  app.UseAuthorization() in the Program class.

app.MapControllers();

app.MapRazorPages();
```

```
app.UseAuthentication();
app.UseAuthorization();//Authorization middleware is enabled by default in the web application template
by the inclusion of app.UseAuthorization() in the Program class.
```



Middleware authorization order depends on knowing **who the user is**, **and where they are trying to go**.

If the user is not authorized, the pipeline is short-circuited.

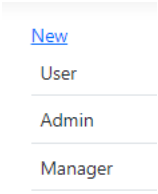
Otherwise the request flows through to Endpoint middleware and the page is executed.

3. Roles and Claims overview

This manual is a simple overview to applying roles and claims to the members of the system.

Role Based Authorization

Role-based authorization in ASP.NET Core or role-based access control (RBAC) is easier to understand and implement but far less flexible than Claims.



RBAC is implemented with the predefined ClaimTypes.Role.

Roles are broad criteria for to filter who can access what places.

Claim Based Authorization

Claims-based authorization in ASP.NET Core or claims-based access control (CBAC) is not easy. CBAC provides more granular control of what the user can access supporting the principle of least privilege.

CBAC is implemented with ClaimsIdentity and AuthorizationPolicy.

In many cases you might want to have a finer grain of criteria than using Roles. The example being “People who have been employed over 6 months”. Now this isn’t a Role, instead it’s a Claim.

It means that you can have criteria that are fluid, such as using time employed, or for a different app, maybe how many points you have scored. (Users with more than 100 points have access to the secret pages)

Eg:

emailconfirmed@xxx.com

Type	Value
DataEntry	
Joining Date	2023-1-18
Coffee Type	Latte

- These claims for this user are DataEntry – the user can enter data,
- You can calculate how long they have been employed from their Joining Date
- You can use Coffee Type as a claim type to filer out people who don’t drink coffee.

Basically anything you want can be a claim.

For example I only want Latte drinkers who have been employed over 6 months.

Where to place your claims and role policies?

There are a number of places where you can restrict access to endpoints.

Directly at the endpoints. On the pages.

You can use the Authorize Attribute to apply authorization to endpoints.

The attribute has some properties, among which are Roles and Policy.

At its most basic, when you apply the attribute to an endpoint, it prevents access to that endpoint to anonymous users. Users must authenticate to become authorized to proceed.

There are a number of ways in which you can apply the attribute to an endpoint. The simplest way to add it to the PageModel class.

```
namespace RolesForAssessment.Pages.RolesManager
{
    [Authorize(Roles = "Admin")]
    6 references
    public class AssignModel : PageModel
    {
        //We inject the UserManager and RoleManager services into the PageModel class
        private readonly RoleManager<IdentityRole> _roleManager;
        private readonly UserManager<IdentityUser> _userManager;
        0 references
```

```
C#

[Authorize(Roles = "Administrator")]
public class AdministrationController : Controller
{
    public IActionResult Index() =>
        Content("Administrator");
}
```

Multiple roles can be specified as a comma separated list:

The SalaryController is only accessible by users who are members of the HRManager role OR the Finance role.

```
C#

[Authorize(Roles = "HRManager,Finance")]
public class SalaryController : Controller
{
    public IActionResult Payslip() =>
                    Content("HRManager || Finance");
}
```

When multiple attributes are applied, an accessing user must be a member of all the roles specified. The following sample requires that a user must be a member of both the PowerUser AND ControlPanelUser role:

```
C#

[Authorize(Roles = "PowerUser")]
[Authorize(Roles = "ControlPanelUser")]
public class ControlPanelController : Controller
{
    public IActionResult Index() =>
        Content("PowerUser && ControlPanelUser");
}
```

We can change access so that anonymous users cannot access the page by any means by protecting it with the Authorize attribute.

```
Error.cshtml
Index.cshtml
  Index.cshtml.cs
Privacy.cshtml
```

```
using Microsoft.AspNetCore.Authorization;   //add this
using Microsoft.AspNetCore.Mvc.RazorPages;
namespace RolesForAssessment.Pages
{
    [Authorize] //add this

    8 references
    public class IndexModel : PageModel
```

A controller can be locked down but allow anonymous, unauthenticated access to individual actions:

```
C#

[Authorize]
public class Control3PanelController : Controller
{
    public IActionResult SetTime() =>
        Content("[Authorize]");

    [AllowAnonymous]
    public IActionResult Login() =>
        Content("[AllowAnonymous]");
}
```

In a Razor Pages application, the 401 forbidden response redirects to the login page, which by default is configured to be at **"/Identity/Account/Login"**.

The 403 response includes a redirect to a page specified by the AccessDeniedPath option. The redirect location is **"/Identity/Account/AccessDenied"**.

You can customize the endpoints in Program.cs

```
builder.Services.ConfigureApplicationCookie(configure: options =>
{
    // Cookie settings
    options.Cookie.HttpOnly = true;
    options.ExpireTimeSpan = TimeSpan.FromMinutes(value: 5);

    options.LoginPath = "/Identity/Account/Login";
    options.AccessDeniedPath = "/Identity/Account/AccessDenied";
    options.SlidingExpiration = true;
});
```

The is quick and easy, but if you have a number of pages to protect against anonymous users, the only way to check that you have applied the attribute to the relevant PageModels is to look at each file individually.

If you want to protect the contents of an entire folder, you must remember to add the attribute to each and every page in the folder, This is problematic on a big site.

Restrict entire folders

Ideally, you want to centralize the code that applies authorization to endpoints so that you can tell, at a glance, which parts of the application are protected and to what degree.

Additional extension methods exist that enable us to apply authorization to individual pages and whole folders via conventions. Using these, we can establish our authorization rules in one place, in the Program class. The key methods are:

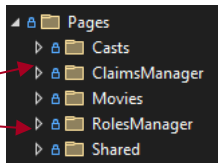
- AuthorizePage - adds authorization to a single page
- AuthorizeFolder - adds authorization to all pages in the specified folder
- AuthorizeAreaFolder - adds authorization to all pages in the specified folder within the specified area

```
C#

services.AddRazorPages(options =>
{
    options.Conventions.AuthorizePage("/Contact");
    options.Conventions.AuthorizeFolder("/Private");
    options.Conventions.AllowAnonymousToPage("/Private/PublicPage");
    options.Conventions.AllowAnonymousToFolder("/Private/PublicPages");
});
```

```
options.Conventions.
                    AuthorizeAreaFolder
                    AuthorizeAreaPage
                    AuthorizeFolder
                    AuthorizePage
```

Each one of these methods takes the name of the page, folder and/or area, and also includes an overload that takes the name of a policy.

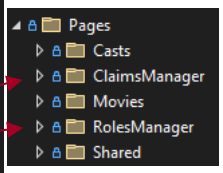
```
builder.Services.AddRazorPages(configure: options =>
{
    options.Conventions.AuthorizeFolder(folderPath: "/ClaimsManager");
    options.Conventions.AuthorizeFolder(folderPath: "/RolesManager");
});
```



We will look policies in detail later. Policies as represent authorization requirements beyond just being authenticated.

The AuthorizePage method in Program.cs replaces the Authorize attribute in the home page earlier.

```
//Having configured the policy named AdminPolicy, we can apply it to the AuthorizeFolder method
  to ensure that only members of the Admin role can access the content:
builder.Services.AddRazorPages(configure: options =>
{
    options.Conventions.AuthorizeFolder(folderPath: "/ClaimsManager", policy: "AdminPolicy");
    options.Conventions.AuthorizeFolder(folderPath: "/RolesManager", policy: "AdminPolicy");
});
```



There are further options here.

4. Role based Authentication

When an identity is created it may belong to one or more roles. For example, Tracy may belong to the Administrator and User roles while Scott may only belong to the User role.

How these roles are created and managed depends on the backing store of the authorization process.

Roles are exposed to the developer through the IsInRole method on the ClaimsPrincipal class. AddRoles must be added to Role services.

Before we can work with Roles we need to add the AddRoles Service to the application in the **Program.cs** class.

```
//the default identity of the user
builder.Services.AddDefaultIdentity<IdentityUser>(configureOptions: options => options.SignIn.RequireConfirmedAccount = true)
    .AddRoles<IdentityRole>()
    .AddEntityFrameworkStores<ApplicationDbContext>();
builder.Services.AddRazorPages();
```

The role type in the is IdentityRole

```
//the default identity of the user
builder.Services.AddDefaultIdentity<ApplicationUser>(options => options.SignIn.RequireConfirmedAccount =
true)
    .AddRoles<IdentityRole>()
    .AddEntityFrameworkStores<ApplicationDbContext>();
```

Keep the **Program.cs** open, we will use it throughout the manual.

There is no UI for managing roles, so we have to build our own.

Building the Role UI - Viewing Roles

Roles provide a simple mechanism for grouping together users who have the same level of access.

They are most useful in applications that are unlikely to grow in complexity and where it is easy to differentiate the access needs of different groups of users.

Create a New Role

List of All Roles

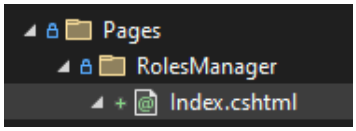
User
Admin
Manager

List of Users and their Roles

ccc@ccc.com	User
aaa@aaa.com	Admin
bbb@bbb.com	Manager

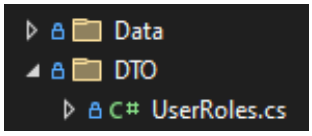
Create the Users and Roles interface

Add a folder to the Pages folder named *RolesManager*. Within that, **add a new Razor Page named *Index.cshtml*.**



Create a DTO called UserRoles

You need a DTO called **UserRoles** that will hold the user, and the role that they have.




```
namespace RolesForAssessment.DTO
{
    3 references
    public class UserRoles
    {
        2 references
        public string UserName { get; set; }
        2 references
        public string RoleName { get; set; }
    }
}
```

In the Index.cshtml page we are going to inject the RoleManager service and use its `Roles` property to populate a public `List<IdentityRole>` property

1. First of all we need to get the list of Users, from the Identity database to do that we inject in the **ApplicationDbContext**.

```
private readonly ApplicationDbContext _context;
```

2. Then we need to get the Roles that are available by injecting the RoleManager.

```
private readonly RoleManager<IdentityRole> _roleManager;
```

this will give us access to the roles we will add in.

dbo.AspNetRoles [Data] | Index.cshtml.cs

Max Rows: 1000

Id	Name	NormalizedN
2788a48e-e043...	User	USER
9dfcf9cb-f6b0-...	Admin	ADMIN
fadf7a51-b748...	Manager	MANAGER
NULL	NULL	NULL

3. Inject them into our Index page.

```
private readonly ApplicationDbContext _context;
private readonly RoleManager<IdentityRole> _roleManager;

0 references
public IndexModel(RoleManager<IdentityRole> roleManager, ApplicationDbContext context) {
    _roleManager = roleManager;
    _context = context;
}
```

4. Then we have to create a list of roles

```
//create a list of all the Roles
3 references
public List<IdentityRole> Roles { get; set; }
```

List of All Roles

User
Admin
Manager

5. As well as a list of Users and Roles

List of Users and their Roles

ccc@ccc.com	User
aaa@aaa.com	Admin
bbb@bbb.com	Manager

```
//create a list of all the Current Users and Roles
2 references
public List<UserRoles> UserAndRoles { get; set; }
//create the Users and Roles from the DB
1 reference
public List<UserRoles> GetUserAndRoles() {
    var list = (from user in _context.Users
                join userRoles in _context.UserRoles on user.Id equals userRoles.UserId
                join role in _context.Roles on userRoles.RoleId equals role.Id
                select new UserRoles { UserName = user.UserName, RoleName = role.Name }).ToList();
    return list;
}
```

6. Finally send the results to the front to be displayed

```
0 references
public void OnGet() {
    //pass the Roles to the front end
    Roles = _roleManager.Roles.ToList();
    //Pass the users and roles to the front end
    UserAndRoles = GetUserAndRoles();

}
```

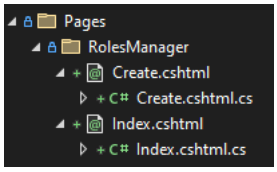
All code

```
[BindProperties]
    public class IndexModel : PageModel {
        private readonly ApplicationDbContext _context;
        private readonly RoleManager<IdentityRole> _roleManager;

        public IndexModel(RoleManager<IdentityRole> roleManager, ApplicationDbContext context) {
            _roleManager = roleManager;
            _context = context;
        }
        //create a list of all the Roles
        public List<IdentityRole> Roles { get; set; }
        //create a list of all the Current Users and Roles
        public List<UserRoles> UserAndRoles { get; set; }
        //create the Users and Roles from the DB
        public List<UserRoles> GetUserAndRoles() {
            var list = (from user in _context.Users
                        join userRoles in _context.UserRoles on user.Id equals userRoles.UserId
                        join role in _context.Roles on userRoles.RoleId equals role.Id
                        select new UserRoles { UserName = user.UserName, RoleName = role.Name }).ToList();
            return list;
        }

        public void OnGet() {
            //pass the Roles to the front end
            Roles = _roleManager.Roles.ToList();
            //Pass the users and roles to the front end
            UserAndRoles = GetUserAndRoles();
        }
    }
```

In the Razor page itself, check to see if there are any roles, and if so, display them in a table

```
@{
    ViewData["Title"] = "Roles and their Users";
}
<a asp-page="/RolesManager/Create">Create a New Role </a>
<hr />
@if (Model.Roles.Any())
{
    <h5>List of All Roles</h5>
    <div class="table-responsive">
    <table class="table table-striped table-bordered table-primary">
        @foreach (var role in Model.Roles)
        {
            <tr>
                <td>@role.Name</td>
            </tr>
        }
    </table>
    </div>
    <h5>List of Users and their Roles</h5>
    <div class="table-responsive">
    <table class="table table-striped table-bordered table-dark">
    @foreach (var result in Model.UserAndRoles)
    {
        <tr>
            <td>@result.UserName</td>
            <td>@result.RoleName</td>
        </tr>
    }
</table>
</div>
}
```

Create the Add Roles Page



Add a new Razor Page to the RoleManager folder named **Create**.

This will contain a form for creating a new role. The only piece of data we need to do that is a name.

We inject the RoleManager service into the page and use its CreateAsync method to add the new role. Once again, the code should look similar to the CRUD pages that we have already created.

First, the PageModel class.

```
namespace RolesForAssessment.Pages.RolesManager
{
    6 references
    public class CreateModel : PageModel
    {
        private readonly RoleManager<IdentityRole> _roleManager;

        0 references
        public CreateModel(RoleManager<IdentityRole> roleManager)
        {
            _roleManager = roleManager;
        }

        [BindProperty]
        1 reference
        public string Name { get; set; }
        0 references
        public async Task<IActionResult> OnPostAsync()
        {
            if (ModelState.IsValid)
            {
                var role = new IdentityRole { Name = Name }; await _roleManager.CreateAsync
                  (role);
                return RedirectToPage(pageName: "/RolesManager/Index");
            }
            return Page();
        }
    }

}
```

```
private readonly RoleManager<IdentityRole> _roleManager;
        public CreateModel(RoleManager<IdentityRole> roleManager)
        {
            _roleManager = roleManager;
        }

        [BindProperty]
        public string Name { get; set; }
        public async Task<IActionResult> OnPostAsync()
        {
            if (ModelState.IsValid)
            {
  var role = new IdentityRole { Name = Name.Trim() };
await _roleManager.CreateAsync(role);
                return RedirectToPage("/RolesManager/Index");
            }
            return Page();
        }
```

The Front page is just a simple form

Create Role

Name

Assign

```
@page
@model RolesForAssessment.Pages.RolesManager.CreateModel
@{
    ViewData["Title"] = "Create Role";
}
<h4>Create Role</h4>

<div class="row">
    <div class="col-md-8">
        <form method="post">
            <div class="form-group mb-3">
                <label asp-for="Name" class="control-label"></label>
                <input asp-for="Name" class="form-control" />
                <span asp-validation-for="Name" class="text-danger"></span>
            </div>
            <div class="form-group">
                <input type="submit" value="Assign" class="btn btn-primary" />
            </div>
        </form>
    </div>
</div>

@section scripts{
    <partial name="_ValidationScriptsPartial" /> }
```

```
<h4>Create Role</h4>
<div class="row">
    <div class="col-md-8">
        <form method="post">
            <div class="form-group mb-3">
                <label asp-for="Name" class="control-label"></label>
                <input asp-for="Name" class="form-control" />
                <span asp-validation-for="Name" class="text-danger"></span>
            </div>
            <div class="form-group">
                <input type="submit" value="Assign" class="btn btn-primary" />
            </div>
        </form>
    </div>
</div>
@section scripts{
    <partial name="_ValidationScriptsPartial" /> }
```

Note: Make sure there are no spaces around your Claims and Roles.

This issue took too long to solve, later in the code you will check if the user has a claim

```
context.User.HasClaim(type: "Permission", value: "View Roles")
```
this returns a True/false.

However “View Roles” entered have a space at the front, that was so small I didn’t see it in the DB – like this

Max Rows: 1000

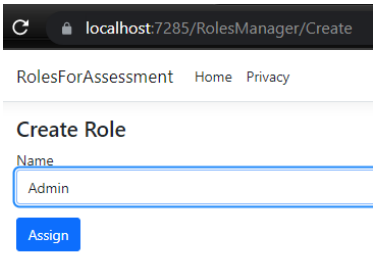
UserId	ClaimType	ClaimValue
f0020d26-edbc...	Permission	View Roles
f0020d26-edbc...	Joining Date	2020-1-12
8f536075-01f3...	Permission	View Roles
8f536075-01f3...	Joining Date	2021-1-12
b2eefcef-b4c2-...	Joining Date	2022-1-12
f0020d26-edbc...	Admin	
5d790e78-647...	Joining Date	2022-1-12
5d790e78-647...	Permission	View Roles
NULL	NULL	NULL

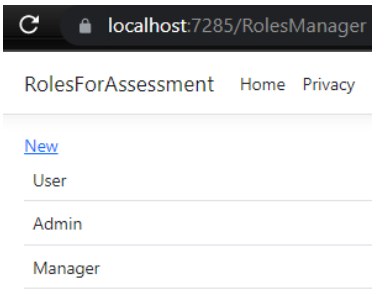
As a result hours were wasted trying to find why
```
context.User.HasClaim(type: "Permission", value: "View Roles")
```
returned false.

Check that your entries are being trimmed `Name.Trim()` to remove spaces.

Add roles to RoleManager

Go to your Create page and add in Admin, Manager, and User

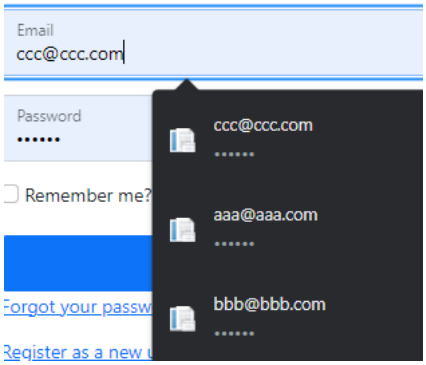




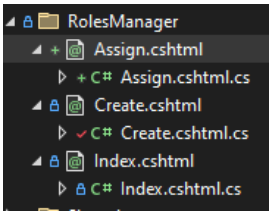
Assigning roles to users

Before we can assign roles to users, we need some users.

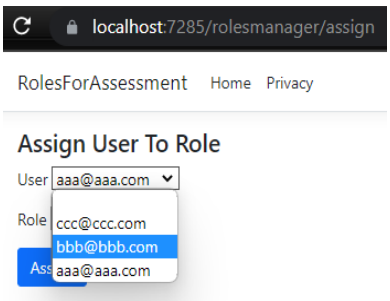
1. Register three users see mine right, in the application and for simplicity, the same password.

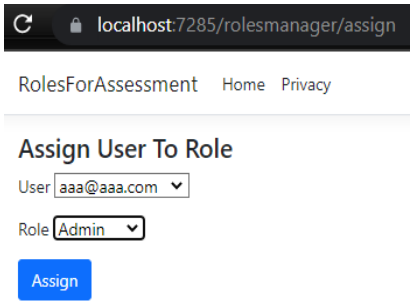


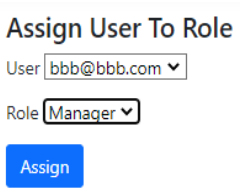
2. Next, add a new Razor page to the RolesManager folder named **Assign**.



3. In this page, we will obtain a list of all users, and a list of all roles, and present them in select lists which will be used to assign the selected user to the selected role. Its quite simplistic but works.







Change the AssignModel code to the following listing.

```
6 references
public class AssignModel : PageModel
{
    //We inject the UserManager and RoleManager services into the PageModel class
    private readonly RoleManager<IdentityRole> _roleManager;
    private readonly UserManager<IdentityUser> _userManager;
    0 references
    public AssignModel(RoleManager<IdentityRole> roleManager, UserManager<IdentityUser> userManager)
    {
        _roleManager = roleManager;
        _userManager = userManager;
    }
    2 references
    public SelectList Roles { get; set; }
    2 references
    public SelectList Users { get; set; }

    [BindProperty, Required, Display(Name = "Role")]
    4 references
    public string SelectedRole { get; set; }
    [BindProperty, Required, Display(Name = "User")]
    4 references
    public string SelectedUser { get; set; }

    0 references
    public async Task OnGet()
    {
        await GetOptions();
    }

    0 references
    public async Task<IActionResult> OnPostAsync()
    {
        if (ModelState.IsValid)
        {//We get the user with the selected name and assign the selected user to the selected role
            var user = await _userManager.FindByNameAsync(userName: SelectedUser); await _userManager.AddToRoleAsync
              (user, role: SelectedRole); return RedirectToPage(pageName: "/RolesManager/Index");
        }
        await GetOptions(); return Page();
    }

    //We declare a private method that assign users and roles to SelectList object
    2 references
    public async Task GetOptions()
    {
        var roles = await _roleManager.Roles.ToListAsync();
        var users = await _userManager.Users.ToListAsync();
        Roles = new SelectList(items: roles, selectedValue: nameof(IdentityRole.Name));
        Users = new SelectList(items: users, selectedValue: nameof(IdentityUser.UserName));
    }
}
```

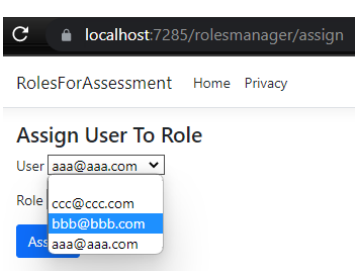
```
public class AssignModel : PageModel
    {
        //We inject the UserManager and RoleManager services into the PageModel class
        private readonly RoleManager<IdentityRole> _roleManager;
        private readonly UserManager<IdentityUser> _userManager;
        public AssignModel(RoleManager<IdentityRole> roleManager, UserManager<IdentityUser> userManager)
        {
            _roleManager = roleManager;
            _userManager = userManager;
        }
        public SelectList Roles { get; set; }
        public SelectList Users { get; set; }

        [BindProperty, Required, Display(Name = "Role")]
        public string SelectedRole { get; set; }
        [BindProperty, Required, Display(Name = "User")]
        public string SelectedUser { get; set; }

        public async Task OnGet()
        {
            await GetOptions();
        }

        public async Task<IActionResult> OnPostAsync()
        {
            if (ModelState.IsValid)
            {//We get the user with the selected name and assign the selected user to the selected role
var user = await _userManager.FindByNameAsync(SelectedUser);
await _userManager.AddToRoleAsync(user, SelectedRole);
return RedirectToPage("/RolesManager/Index");
```

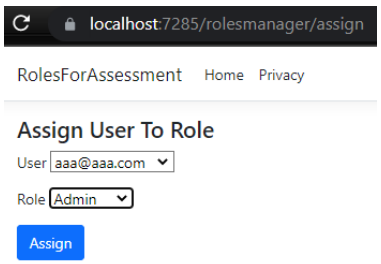
```
            }
            await GetOptions(); return Page();
        }

        //We declare a private method that assign users and roles to SelectList object
        public async Task GetOptions()
        {
            var roles = await _roleManager.Roles.ToListAsync();
            var users = await _userManager.Users.ToListAsync();
            Roles = new SelectList(roles, nameof(IdentityRole.Name));
            Users = new SelectList(users, nameof(IdentityUser.UserName));
        }
    }
```

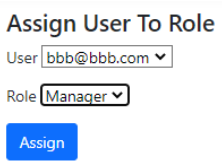
Front code.

```
<h4>Assign User To Role</h4>
<div class="row">
    <div class="col-md-8">
        <form method="post">
            <div class="form-group mb-3">
                <label asp-for="SelectedUser" class="control-label"></label>
                <select asp-for="SelectedUser" asp-items="Model.Users" class="formcontrol">
                    <option></option>
                </select>
                <span asp-validation-for="SelectedUser" class="text-danger"></span>
            </div>
            <div class="form-group mb-3">
                <label asp-for="SelectedRole" class="control-label"></label>
                <select asp-for="SelectedRole" asp-items="Model.Roles" class="formcontrol">
                    <option></option>
                </select>
                <span asp-validation-for="SelectedRole" class="text-danger"></span>
            </div>
            <div class="form-group">
                <input type="submit" value="Assign" class="btn btn-primary" />
            </div>
        </form>
    </div>
</div>

@section scripts{
    <partial name="_ValidationScriptsPartial" /> }
```

Assign aaa to Admin role, and bbb to Manager Role





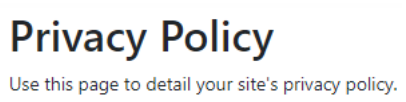


Then add an Authorize attribute to the AssignModel class,

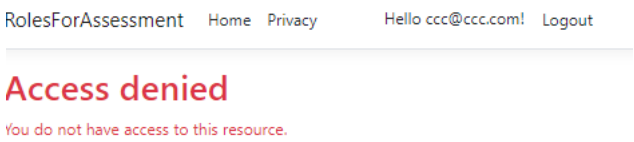
Assign “Admin” as the value to the Roles property:

```
namespace RolesForAssessment.Pages.RolesManager
{
    [Authorize(Roles = "Admin")]
    6 references
    public class AssignModel : PageModel
```

Admin can log in

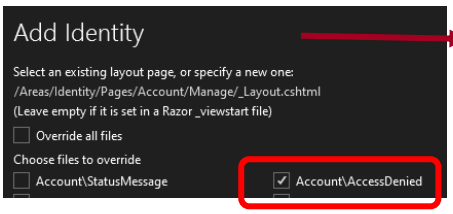


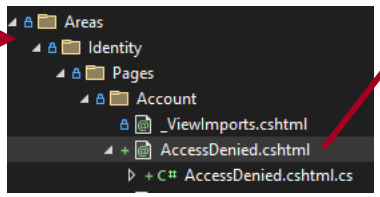
Others cannot.

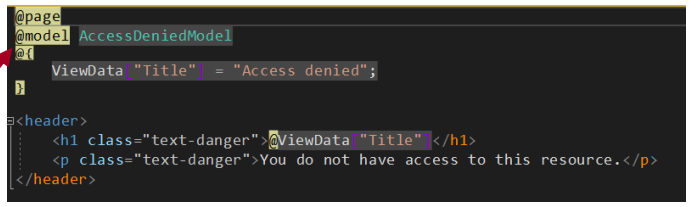


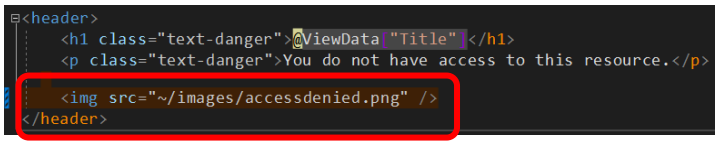
Modify the Access Denied file.

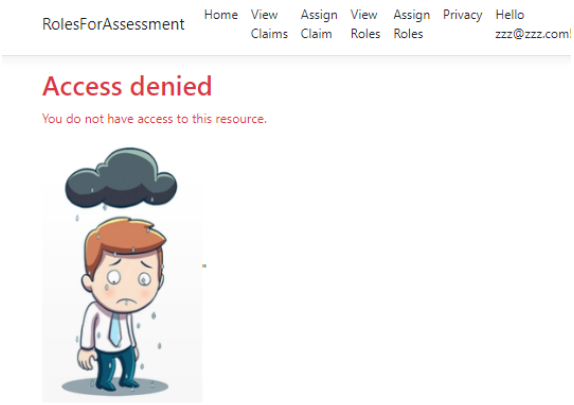
Access the AccessDenied file by adding the file to the Identity,









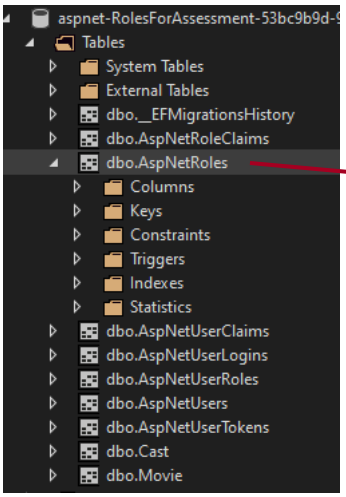


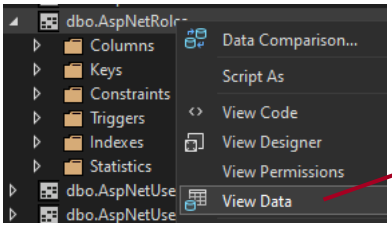
This is the absolute basics you can do. By locking out a page using Authorise.

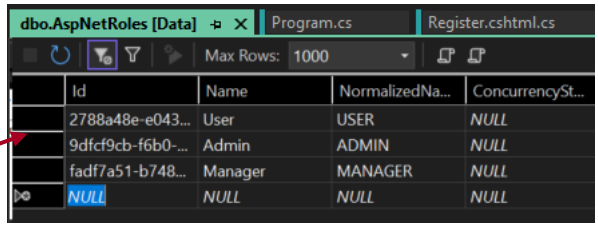
Comment out the [Authorise] when finished as it will affect future authorizations.

```
//  [Authorize(Roles = "Admin")]
6 references
public class AssignModel : PageModel
```

The Roles are stored in the AspNetRoles table. You can add or modify them there as well. (although its not a good policy) Right click on the table and choose View Data.







5. Using policies to apply role checks

So far we have extended the Authorize attribute `[Authorize(Roles = "Admin")]` to check that the current user belongs to the specified role.

Lock out an entire folder with a Policy

If we want lock the contents of an entire folder, we can use an **overload of the AuthorizeFolder** method that takes a policy.

You can consider a policy as representing the requirements that need to be satisfied to determine whether the current user is authorized to access the requested endpoint.

For relatively simple policies, we can use AuthorizeOptions within the **AddAuthorization** method to configure a role-based policy.

```
builder.Services.AddAuthorization(configure: options =>
{
    options.AddPolicy(name: "AdminPolicy", configurePolicy: policyBuilder => policyBuilder.RequireRole(roles: "Admin"));
```

The AddPolicy method takes the name of the policy, `name: "AdminPolicy"` and an AuthorizationPolicyBuilder which has a RequireRole method, `RequireRole(roles: "Admin")` enabling us to state which roles are required

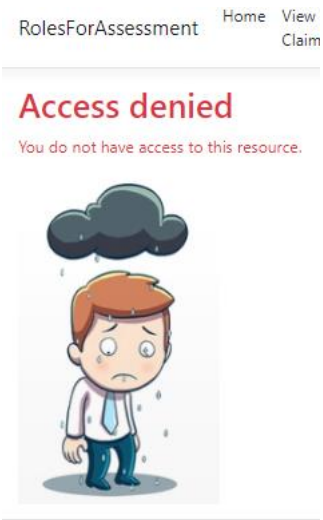
We then add the **AdminPolicy** policy to the AuthorizeFolder method with the path to the folder we want checked.

```
builder.Services.AddRazorPages(configure: options =>
{
    options.Conventions.AuthorizeFolder(folderPath: "/RolesManager", policy: "AdminPolicy");
});
```

Modify the Program file as below.

```
//The AddPolicy method takes the name of the policy, and an AuthorizationPolicyBuilder which has a RequireRole method,
  enabling us to state which roles are required
builder.Services.AddAuthorization(configure: options =>
{
    options.AddPolicy(name: "AdminPolicy", configurePolicy: policyBuilder => policyBuilder.RequireRole(roles: "Admin"));
});
//Having configured the policy named AdminPolicy, we can apply it to the AuthorizeFolder method to ensure that only
  members of the Admin role can access the content:
builder.Services.AddRazorPages(configure: options => {
    options.Conventions.AuthorizeFolder(folderPath: "/RolesManager", policy: "AdminPolicy");
});
```

Test it out

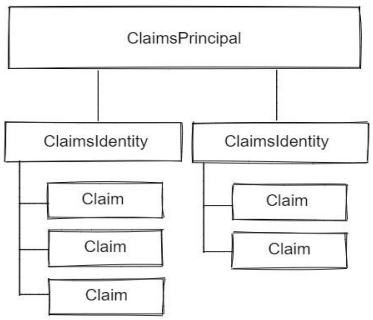


Comment it out when finished

6. Claims based Authorization

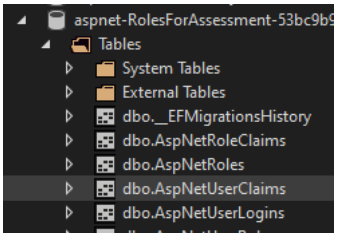
Claims are simply name-value pairs that represent items of data that we know about the user. They are attached to a ClaimsIdentity, which is then attached to the ClaimsPrincipal:

A ClaimsPrincipal supports multiple identities, each supporting multiple claims



Within .NET, claims are represented by the Claim class. Among its properties are **Type, Value and Issuer**. The last of these, the Issuer, is the authority that issued the claim.

Claims stored on the DB with the UserID



Id	UserId	ClaimType	ClaimValue
1	f0020d26-edbc...	Permission	View Roles
2	f0020d26-edbc...	Joining Date	2020-1-12
3	8f536075-01f3...	Permission	View Roles
4	8f536075-01f3...	Joining Date	2021-1-12
5	b2eefcef-b4c2-...	Joining Date	2022-1-12
6	f0020d26-edbc...	Admin	
8	5d790e78-647...	Joining Date	2022-1-12
9	5d790e78-647...	Permission	View Roles
10	549030a7-cbe3...	DataEntry	
11	549030a7-cbe3...	Joining Date	2023-1-18
12	5d790e78-647...	Coffee Type	Long Black
13	64b41401-f3e3...	Coffee Type	Latte
14	549030a7-cbe3...	Coffee Type	Latte

When you assign claims to users in your application, the issuer is LOCAL_AUTHORITY by default. If you incorporate external authentication providers like Google or Facebook in your application, any claims that they add to the identity that they authenticate will be issued by them.

You can choose which version of a claim to use, based on how much weight you give the issuer. For example, an external authentication service like FaceBook may well prove an email claim, but the email address may not exist.

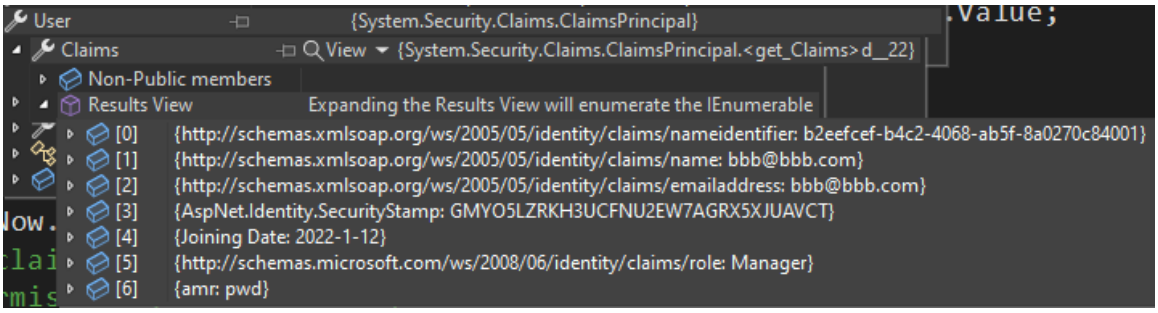
There is no limit imposed on what the Type represents. Widely used claim types are represented by URIs at the domain schemas.xmlsoap.org.

This shows the claim types that you are more likely to find yourself working with.

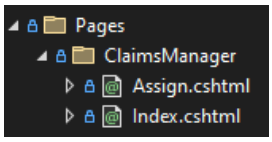
- Claim Type — Description
- ClaimTypes.Name — Represents the user name of the user
- ClaimTypes.Email — Used for the user's email address
- ClaimTypes.GivenName — The user's first name
- ClaimTypes.Surname — The user's last name

- ClaimTypes.NameIdentifier The user's unique identifier

When the user is authenticated in our application and has a ClaimsIdentity assigned to it, the authentication system adds various claims to it. These include the Name, Email and NameIdentifier:



Create the Claim Manager Page

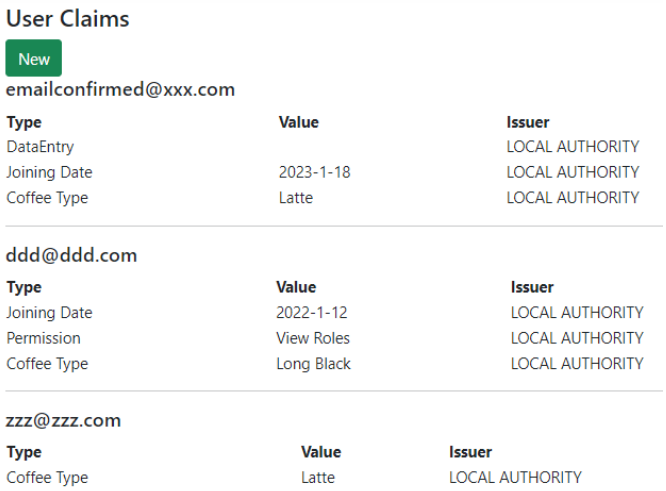


Add a new folder to the Pages folder named ClaimsManager. Within that, we add a new Razor page named Index. This will list all users who have additional claims assigned to them, and details of the claims that have been assigned.

The PageModel code for ClaimsManager Index page takes the UserManager as an injected dependency and assigns it to a public property so that it is accessible to the Razor part of the page via its Model property. It is also used within the OnGetAsync method to populate a collection of IdentityUser objects (listing 10.15).

```
public class IndexModel : PageModel
{
    //import the userManger and generate a list of users
    2 references
    public UserManager<IdentityUser> UserManager { get; set; }
    0 references
    public IndexModel(UserManager<IdentityUser> userManager)
    {
        UserManager = userManager;
    }
    1 reference
    public List<IdentityUser> Users { get; set; }
    0 references
    public async Task OnGetAsync()
    {
        Users = await UserManager.Users.ToListAsync();
    }

}
```

```
public class IndexModel : PageModel
    {
        //import the userManger and generate a list of users
        public UserManager<IdentityUser> UserManager { get; set; }
        public IndexModel(UserManager<IdentityUser> userManager)
        {
            UserManager = userManager;
        }
        public List<IdentityUser> Users { get; set; }
        public async Task OnGetAsync()
        {
            Users = await UserManager.Users.ToListAsync();
        }
    }
```



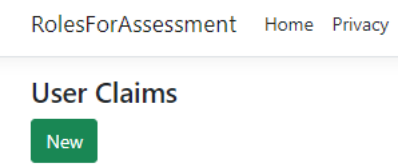
The front just shows all the claims.

```
@page
@model RolesForAssessment.Pages.ClaimsManager.IndexModel
@{
}
<h4>User Claims</h4>
<a class="btn btn-success" asp-page="/ClaimsManager/Assign">New</a>

@foreach (var user in Model.Users)
{
    var claims = await Model.UserManager.GetClaimsAsync(user); if (claims.Any())
    {
        <h5>@user.UserName</h5>
        <table class="table-striped col-12">
            <tr>
                <th>Type</th>
                <th>Value</th>
                <th>Issuer</th>
            </tr>
            @foreach (var claim in claims)
            {
                <tr>
                    <td>@claim.Type</td>
                    <td>@claim.Value</td>
                    <td>@claim.Issuer</td>
                </tr>

            }
        </table>

    }
}
```

```
<h4>User Claims</h4>
<a class="btn btn-success" asp-page="/ClaimsManager/Assign">New</a>

@foreach (var user in Model.Users)
{
    var claims = await Model.UserManager.GetClaimsAsync(user); if (claims.Any())
    {
        <h5>@user.UserName</h5>
        <table class="table-striped col-12">
            <tr>
                <th>Type</th>
                <th>Value</th>
                <th>Issuer</th>
            </tr>
            @foreach (var claim in claims)
            {
                <tr>
                    <td>@claim.Type</td>
                    <td>@claim.Value</td>
                    <td>@claim.Issuer</td>
                </tr>

            }
        </table>
    }
}
```

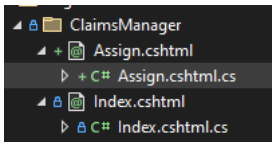
When you run this page for the first time, all the database calls are made but there is no data to be displayed, so you will only be presented with the button inviting you to add a new claim. It goes nowhere at the moment, because we have yet to create the page.

But always check your code is working before moving to the next step and push it to Github.



Create the Assign Claim Page

Add a new page to the *ClaimsManager* folder named *Assign*. This page will feature a list of users in a select list, and inputs for a claim type and value. We will use this page to create claims and assign them to users.



Code over the page.

```
6 references
public class AssignModel : PageModel
{

    private readonly UserManager<IdentityUser> _userManager;
    0 references
    public AssignModel(UserManager<IdentityUser> userManager)
    {
        _userManager = userManager;
    }

    2 references
    public SelectList Users { get; set; }
    [BindProperty, Required, Display(Name = "User")]
    4 references
    public string SelectedUserId { get; set; }
    [BindProperty, Required, Display(Name = "Claim Type")]
    4 references
    public string ClaimType { get; set; }
    [BindProperty, Display(Name = "Claim Value")]
    3 references
    public string ClaimValue { get; set; }
    0 references
    public async Task OnGetAsync()
    {
        await GetOptions();
    }
```

```
public async Task<IActionResult> OnPostAsync()
{
    if (ModelState.IsValid)
    {
        var claim = new Claim(type: ClaimType, value: ClaimValue ?? String.Empty);
        var user = await _userManager.FindByIdAsync(userId: SelectedUserId);
        await _userManager.AddClaimAsync(user, claim);
        return RedirectToPage(pageName: "/ClaimsManager/Index");
    }
    await GetOptions(); return Page();
}

2 references
public async Task GetOptions()
{
    var users = await _userManager.Users.ToListAsync();
    Users = new SelectList(items: users, dataValueField: nameof(IdentityUser.Id), dataTextField: nameof
      (IdentityUser.UserName));
}
```

```
public class AssignModel : PageModel
    {
        private readonly UserManager<IdentityUser> _userManager;
        public AssignModel(UserManager<IdentityUser> userManager)
        {
            _userManager = userManager;
        }
        public SelectList Users { get; set; }
        [BindProperty, Required, Display(Name = “User”)]
        public string SelectedUserId { get; set; }
        [BindProperty, Required, Display(Name = “Claim Type”)]
        public string ClaimType { get; set; }
        [BindProperty, Display(Name = “Claim Value”)]
        public string? ClaimValue { get; set; }
        public async Task OnGetAsync()
        {
            await GetOptions();
        }

        public async Task<IActionResult> OnPostAsync()
        {
            if (ModelState.IsValid)
            {
                var claim = new Claim(ClaimType, ClaimValue ?? String.Empty);
                var user = await _userManager.FindByIdAsync(SelectedUserId);
                await _userManager.AddClaimAsync(user, claim);
                return RedirectToPage(“/ClaimsManager/Index”);
            }
            await GetOptions(); return Page();
        }
        public async Task GetOptions()
        {
            var users = await _userManager.Users.ToListAsync();
Users = new SelectList(users, nameof(IdentityUser.Id), nameof(IdentityUser.UserName));
        }
    }
```

Create the following front end.

```
@page
@model RolesForAssessment.Pages.ClaimsManager.AssignModel
@{
}
<h4>Assign Claim To User</h4>

<div class="row">
    <div class="col-md-8">
        <form method="post">
            <div class="form-group mb-3">
                <label asp-for="SelectedUserId" class="control-label"></label>
                <select asp-for="SelectedUserId" asp-items="Model.Users" class="form-control">
                    <option></option>
                </select>
                <span asp-validation-for="SelectedUserId" class="text-danger"></span>
            </div>
            <div class="form-group mb-3">
                <label asp-for="ClaimType" class="control-label"></label>
                <input asp-for="ClaimType" class="form-control" />
                <span asp-validation-for="ClaimType" class="text-danger"></span>
            </div>
            <div class="form-group mb-3">
                <label asp-for="ClaimValue" class="control-label"></label>
                <input asp-for="ClaimValue" class="form-control" />
            </div>
            <div class="form-group">
                <input type="submit" value="Assign" class="btn btn-primary" />
            </div>
        </form>
    </div>
</div>

@section scripts{
    <partial name="_ValidationScriptsPartial" />
}
```

```
<h4>Assign Claim To User</h4>
<div class=”row”>
    <div class=”col-md-8”>
        <form method=”post”>
            <div class=”form-group mb-3”>
                <label asp-for=”SelectedUserId” class=”control-label”></label>
   <select asp-for=”SelectedUserId” asp-items=”Model.Users” class=”form-control”>
                    <option></option>
                </select>
                <span asp-validation-for=”SelectedUserId” class=”text-danger”></span>
            </div>
            <div class=”form-group mb-3”>
                <label asp-for=”ClaimType” class=”control-label”></label>
                <input asp-for=”ClaimType” class=”form-control” />
                <span asp-validation-for=”ClaimType” class=”text-danger”></span>
            </div>
            <div class=”form-group mb-3”>
                <label asp-for=”ClaimValue” class=”control-label”></label>
                <input asp-for=”ClaimValue” class=”form-control” />
            </div>
            <div class=”form-group”>
                <input type=”submit” value=”Assign” class=”btn btn-primary” />
            </div>
        </form>
    </div>
</div>

@section scripts{
    <partial name=”_ValidationScriptsPartial” />
}
```

Now create the following claims for your Users

User Claims

New

ccc@ccc.com

Type	Value
Permission	View Roles
Joining Date	2021-1-12

bbb@bbb.com

Type	Value
Joining Date	2022-1-12

aaa@aaa.com

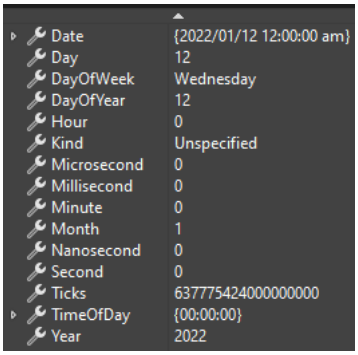
Type	Value
Permission	View Roles
Joining Date	2020-1-12
Admin	

Having assigned a claim, the next step is to use it as part of an authorization policy.

Working with dates.....

Looking at the dates we inserted, I thought it was yyyy-dd-mm, actually its yyyy-mm-dd

This comes from looking at your claim for the date. Make your you don't make the mistake I did 😊



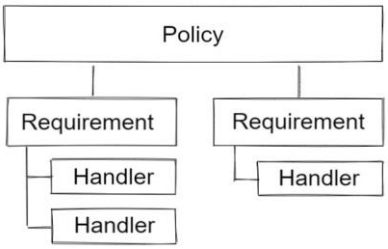
Date	{2022/01/12 12:00:00 am}
Day	12
DayOfWeek	Wednesday
DayOfYear	12
Hour	0
Kind	Unspecified
Microsecond	0
Millisecond	0
Minute	0
Month	1
Nanosecond	0
Second	0
Ticks	637775424000000000
TimeOfDay	{00:00:00}
Year	2022

7. Using policies to enforce claims-based authorization

Claims-based authorization is dependent on policies, which we've touched on before**.**

A policy consists of one or more requirements.

Authorization is granted when **all** of the requirements in a policy have been met. It is the job of one or more authorization handlers to evaluate whether each requirement within the policy has been satisfied.



Using this pattern, it is possible to build complex authorization policies that enable fine grained control over who can reach which parts of an application.

In addition to protecting endpoints, it is also possible to apply authorization policies within a Razor page itself, so, for example, you can toggle the visibility of parts of the UI depending on the current user's claims.

When we used the AuthorizationPolicyBuilder.**RequireRole** method earlier, a requirement of type RolesAuthorizationRequirement was created that specifies that the nominated role is required. We can also use **RequireClaim** in the same manner.

```
options.AddPolicy(name: "AdminPolicy", configurePolicy: policyBuilder => policyBuilder.RequireRole(roles: "Admin"));
options.AddPolicy(name: "AdminPolicy", configurePolicy: policyBuilder => policyBuilder.RequireClaim(claimType: "Admin"));
```

Other methods are available on the AuthorizationPolicyBuilder that enable us to express common policies simply by using other built-in requirements and handlers.

Common methods for building simple policies

Method	Description
RequireClaim(string claimtype)	The user must have the specified claim
RequireClaim(string claimtype, params string[] allowedValues) Or RequireClaim(string claimtype, IEmumerable<string> allowedValues)	The user must have the specified claim with one of the specified values
RequireUserName	The user must have the specified name
RequireAuthenticatedUser	The user must be authenticated
RequireAssertion	Takes a delegate that represents an assertion to be tested to determine authorization status

The variations of the RequireClaim method create a ClaimsAuthorizationRequirement with a handler that returns true if the specified claim exists, and if values are specified, **that at least one of them is found.**

We can test this by changing the code for the existing policy to use the RequireClaim method instead of RequireRole:

```
//The AddPolicy method takes the name of the policy, and an AuthorizationPolicyBuilder which has a RequireRole method,
  enabling us to state which roles are required
builder.Services.AddAuthorization(configure: options =>
{
    options.AddPolicy(name: "AdminPolicy", configurePolicy: policyBuilder => policyBuilder.RequireRole(roles: "Admin"));
    options.AddPolicy(name: "AdminPolicy", configurePolicy: policyBuilder => policyBuilder.RequireClaim(claimType: "Admin"));
});
```

```
options.AddPolicy("AdminPolicy", policyBuilder => policyBuilder.RequireClaim("Admin"));
```

We can also apply this policy to pages individually by passing the name of the policy to the Policy property of the AuthorizeAttribute: This is applied to the Privacy page when you log in, as an example.

```
[Authorize(Policy = "AdminPolicy")]
8 references
public class PrivacyModel : PageModel
{
```

Using assertions for more complex requirements

The RequireAssertion method `policyBuilder.RequireAssertion` is provided to cater for more complex requirements than the other methods can handle.

This type provides us with access to the current user through its User property. From that, we can examine their claims.

RequireAssertion returns a True or False

Users are employed more than 6 months ago example

For example, let's say that we want to implement a requirement that says that a user can access the roles management area if they have a claim with a particular value, but **only** if they have been with the business for more than six months.

In order to be able to determine this, **we record the user's joining date as a claim**.

ccc@ccc.com

Type	Value
Permission	View Roles
Joining Date	2021-1-12

Then we need to **convert the value to a DateTime and compare it to the current date** to determine how long the user has been with the business.

Using the Policy Builder

You use the PolicyBuilder to register the policy, passing in a suitable value for the months parameter.

In Program.cs give the new users the "Permission" claim with the value "View Roles":

```
builder.Services.AddAuthorization(configure: options =>
{
    options.AddPolicy(name: "AdminPolicy", configurePolicy: policyBuilder => policyBuilder.RequireRole(roles: "Admin"));
    options.AddPolicy(name: "AdminPolicy", configurePolicy: policyBuilder => policyBuilder.RequireClaim(claimType: "Admin"));

    //We use the RequireAssertion method, which takes an AuthorizationHandlerContext as a parameter providing access to the
      current user
    options.AddPolicy(name: "ViewRolesPolicy", configurePolicy: policyBuilder => policyBuilder.RequireAssertion(handler: context =>
    {
      // We use the FindFirst method to access a claim and obtain its value(if there is one) and convert it to a DateTime
       var joiningDateClaim = context.User.FindFirst(match: c => c.Type == "Joining Date")?.Value;
       var joiningDate = Convert.ToDateTime(value: joiningDateClaim);

       //We use the HasClaim method to establish that a claim with the specified value exists
       //We compare the joining date value with DateTime.MinValue and the current date to ensure that the claim is not null,
          and that the date is earlier than six months ago
       return context.User.HasClaim(type: "Permission", value: "View Roles") && joiningDate > DateTime.MinValue && joiningDate <
          DateTime.Now.AddMonths(months: -6);
    }));
});
```

```
options.AddPolicy("ViewRolesPolicy", policyBuilder => policyBuilder.RequireAssertion(context =>
    {
        var joiningDateClaim = context.User.FindFirst(c => c.Type == "Joining Date")?.Value;
        var joiningDate = Convert.ToDateTime(joiningDateClaim);
        return context.User.HasClaim("Permission", "View Roles") && joiningDate > DateTime.MinValue &&
joiningDate < DateTime.Now.AddMonths(-6);
    }));
```

User Claims

New

ccc@ccc.com

Type	Value
Permission	View Roles
Joining Date	2021-1-12

bbb@bbb.com

Type	Value
Joining Date	2022-1-12

aaa@aaa.com

Type	Value
Permission	View Roles
Joining Date	2022-1-12
Admin	

Now we have a new “View Roles” policy, update the AuthorizeFolder policy to ViewRolesPolicy so your example person (ccc) can access it since they are older than 6 months.

```
...
//Having configured the policy named AdminPolicy, we can apply it to the
  AuthorizeFolder method to ensure that only members of the Admin role can access the
  content:
builder.Services.AddRazorPages(configure: options =>
{
...
    options.Conventions.AuthorizeFolder(folderPath: "/RolesManager", policy:
      "ViewRolesPolicy");
    //options.Conventions.AuthorizeFolder("/ClaimsManager", "ViewClaimsPolicy");
    // options.Conventions.AuthorizeFolder("/RolesManager", "AdminPolicy");
});
```

So

bbb@bbb.com

Type	Value
Joining Date	2022-1-12

Does not have a joining date 6 months before this date Therefore he cannot view the view roles page. Nor can he view the Assign Roles page, **because the whole folder has been locked out to him.**

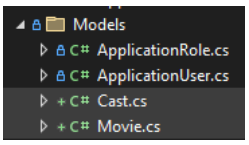
RolesForAssessment Home View Claims Assign Claim View Roles Assign Roles Privacy Hello bbb@bbb.com! Logout

Access denied

You do not have access to this resource.

8. Create a Movie database

For the next section we need a database, lets use the movie one we made earlier in the year.



Add the following classes to the Models

```
public class Cast
    {
        public Guid Id { get; set; }
        public string? FirstName { get; set; }
        public string? LastName { get; set; }
        public string? ScreenName { get; set; }
        public Guid? MovieId { get; set; }
        //Navigation
        public Movie? Movie { get; set; }
    }
```

```
public class Movie
    {
        public Guid Id { get; set; }
        public string Title { get; set; } = string.Empty;
        [DataType(DataType.Date)]
        public DateTime ReleaseDate { get; set; }
        public string Overview { get; set; } = string.Empty;
        public string Genre { get; set; } = string.Empty;
        public decimal Price { get; set; }
    }
```

Update your ApplicationDBContext

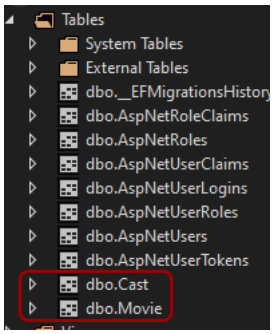
```
8 references
public class ApplicationDbContext : IdentityDbContext
{
    0 references
    public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
        : base(options)
    {
    }

    0 references
    public DbSet<Movie> Movie { get; set; }
    0 references
    public DbSet<Cast> Cast { get; set; }

}
```

Create your Migration

```
PM> add-migration movie
Build started...
Build succeeded.
```

Update the Database

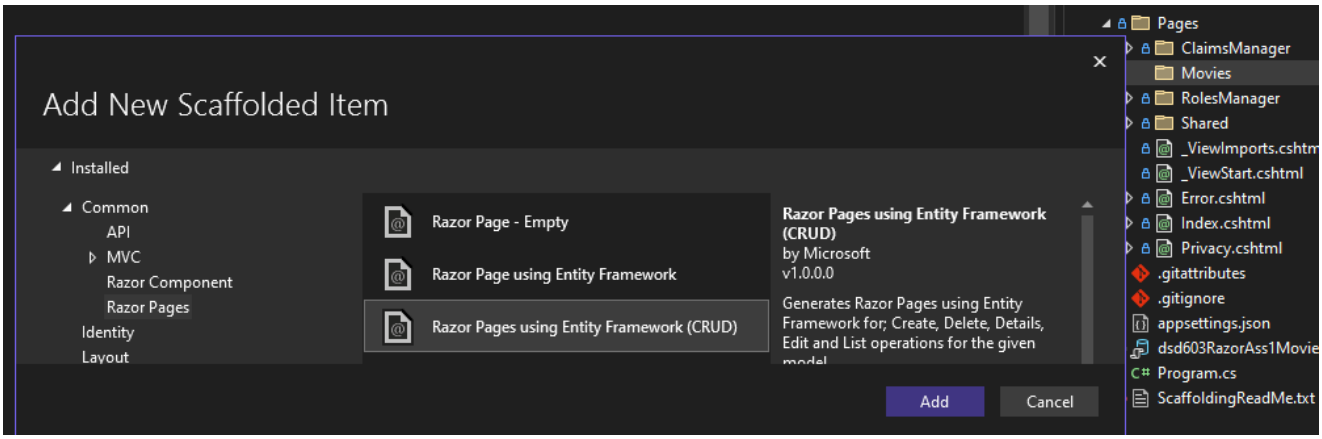
```
PM> update-database movie
Build started...
Build succeeded.
```

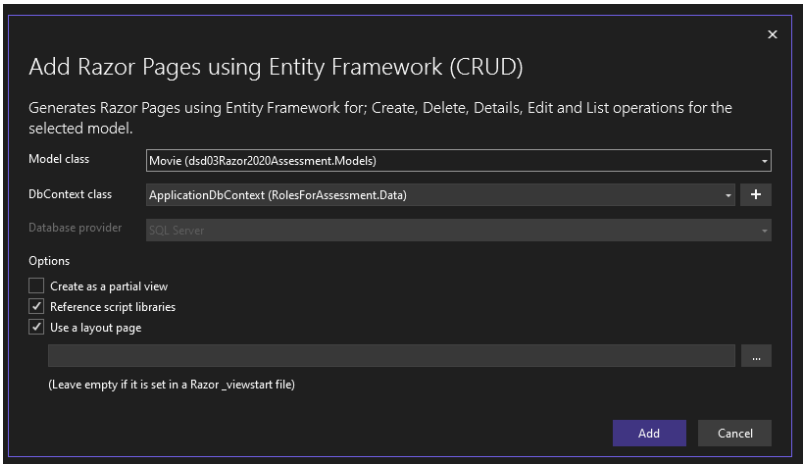


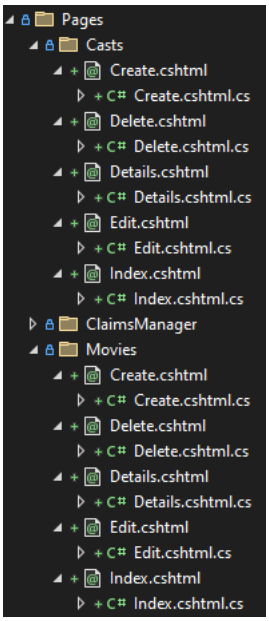
Scaffold The Classes

Create a **Movies** and a **Casts** Folders, (You have pluralize the folder names to avoid errors)) and then right click on each Folder and scaffold out Razor Pages with CRUD.

We need this to add some data to our database.

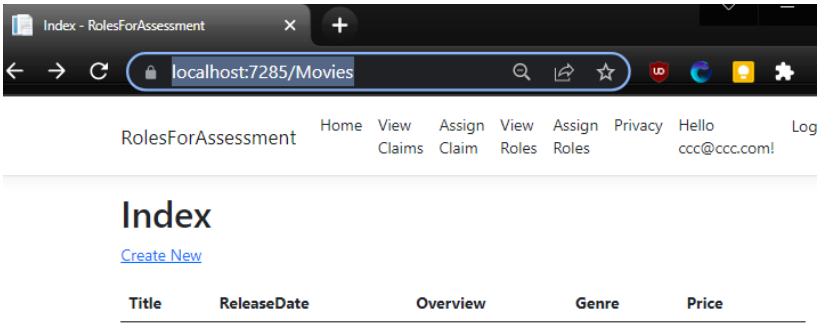




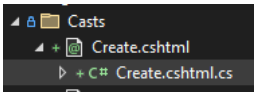


Run the Program

Change the URL to /Movies and add some movies

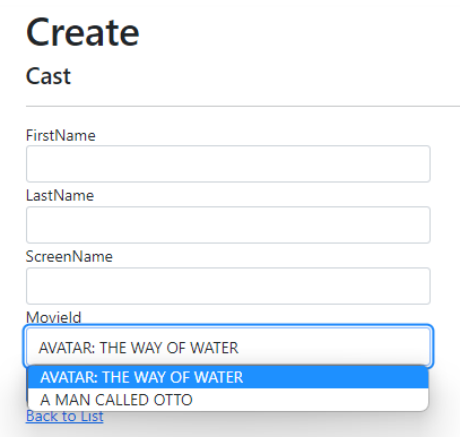


Remember in the Create to change the DataTextField below to Title. Otherwise you only see the ID of the movie and not the name.




```
public IActionResult OnGet()
{
ViewData[index: "MovieId"] = new SelectList(items: _context.Movie, dataValueField: "Id", dataTextField: "Title");
```

Add Data



My data (yours will be different)

Cast

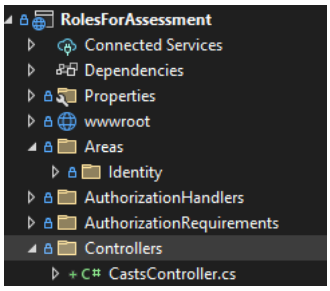
FirstName	LastName	ScreenName	Movie	
Sam	Worthington	Jame	AVATAR: THE WAY OF WATER	Edit | Details | Delete
Zoe	Saldana	Neytiri	AVATAR: THE WAY OF WATER	Edit | Details | Delete
Sigourney	Weaver	Kiri	AVATAR: THE WAY OF WATER	Edit | Details | Delete
Tom	Hanks	Otto Anderson	A MAN CALLED OTTO	Edit | Details | Delete
Mack	Bayda	Malcolm	A MAN CALLED OTTO	Edit | Details | Delete
Mariana	Trevino	Marisol	A MAN CALLED OTTO	Edit | Details | Delete
Colin	Farrell	Pádraic Súilleabháin	THE BANSHEES OF INISHERIN	Edit | Details | Delete
Brendon	Gleeson	Colm Doherty	THE BANSHEES OF INISHERIN	Edit | Details | Delete
Kerry	Condon	Siobhan Súilleabháin	THE BANSHEES OF INISHERIN	Edit | Details | Delete

Movies

Title	ReleaseDate	Overview	Genre	Price	
AVATAR: THE WAY OF WATER	2023/01/16	Set more than a decade after the events of the first film, "Avatar: The Way of Water" begins to tell the story of the Sully family (Jake, Neytiri, and their kids), the trouble that follows them, the lengths they go to keep each other safe, the battles they fight to stay alive, and the tragedies they endure.	Sci-Fi, Adventure, Action, Fantasy	123.00	Edit | Details | Delete
A MAN CALLED OTTO	2023/01/16	Based on the comical and moving New York Times bestseller, A Man Called Otto tells the story of Otto Anderson (Tom Hanks), a grumpy widower whose only joy comes from criticizing and judging his exasperated neighbors. When a lively young family moves in next door, he meets his match in quick-witted and very pregnant Marisol, leading to an unexpected friendship that will turn his world upside-down.	Comedy, Drama	123.00	Edit | Details | Delete
THE BANSHEES OF INISHERIN	2023/01/16	Set on a remote island off the west coast of Ireland, THE BANSHEES OF INISHERIN follows lifelong friends Pádraic and Colm, who find themselves at an impasse when Colm unexpectedly puts an end to their friendship. A stunned Pádraic, aided by his sister Siobhán and troubled young islander Dominic, endeavours to repair the relationship, refusing to take no for an answer. But Pádraic's repeated efforts only strengthen his former friend's resolve and when Colm delivers a desperate ultimatum, events swiftly escalate, with shocking consequences.	Comedy	123.00	Edit | Details | Delete

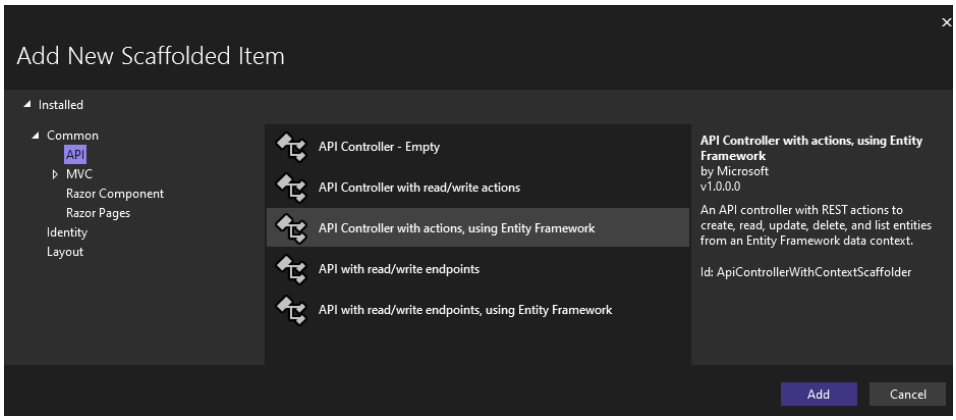
Create API's

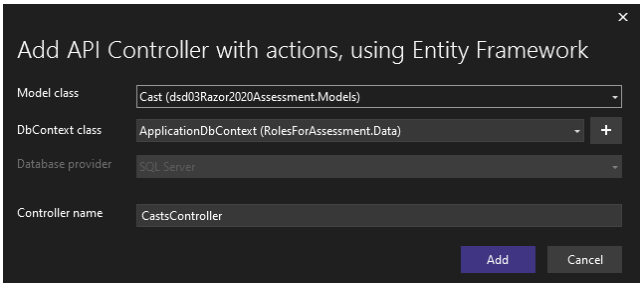
https://learn.microsoft.com/en-us/aspnet/core/web-api/?view=aspnetcore-7.0

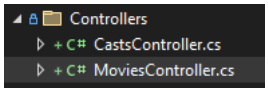


Create a new Folder called Controllers

Right click on the Controller folder and scaffold a new AP! Controller for Cast. Repeat for Movies







Add MapControllers to your Program file

```
app.MapControllers();

app.MapRazorPages();

app.Run();
```

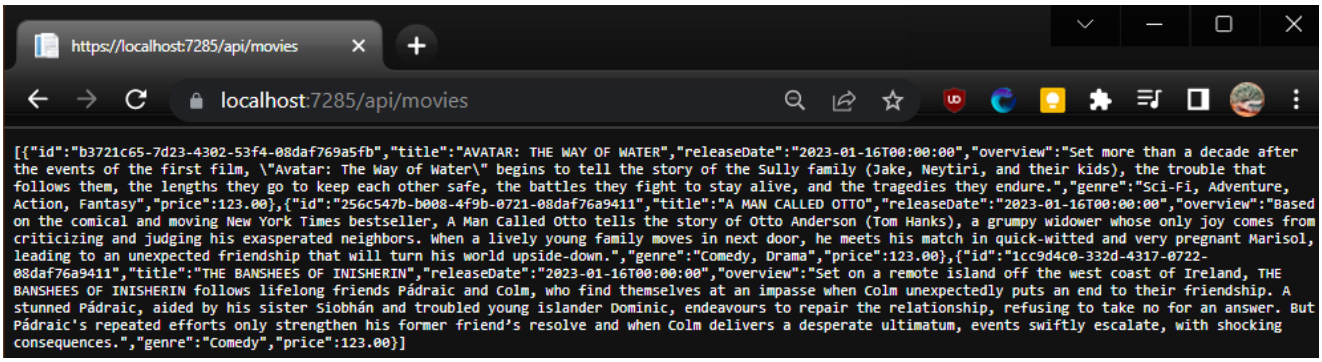
Run your Project.

We are going to access the APIs we have created. In each controller are the endpoints of the CRUD added as comments.

For the Movies it is this one // GET: api/Movies

```
// GET: api/Movies
[HttpGet]
0 references
public async Task<ActionResult<IEnumerable<Movie>>> GetMovie()
{
    if (_context.Movie == null)
    {
        return NotFound();
    }
    return await _context.Movie.ToListAsync();
}
```

Change your browser path to /api/movies



Your data is working!!

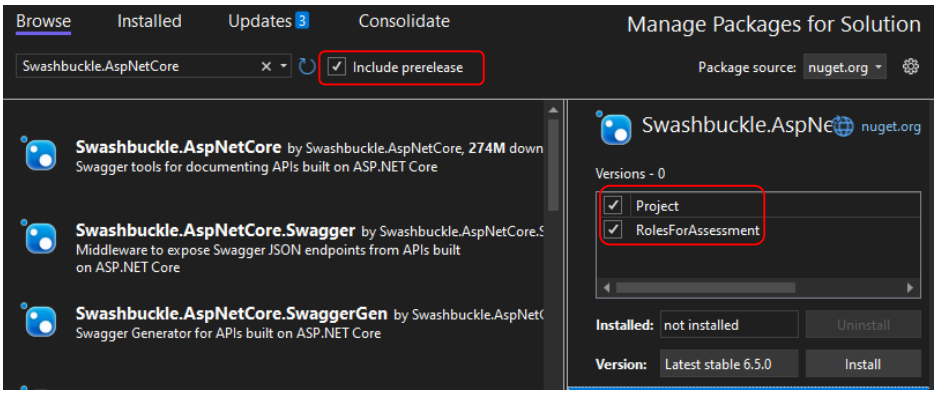
Add Swashbuckle / Swagger

The raw data is pretty hard to work with in the browser. MS have a tool you can add in that provides an interface for your API's.

https://learn.microsoft.com/en-us/aspnet/core/tutorials/getting-started-with-swashbuckle?view=aspnetcore-7.0&tabs=visual-studio

In the **Manage NuGet Package For Solution** type in **Swashbuckle.AspNetCore**

Make sure you tick Include Prerelease



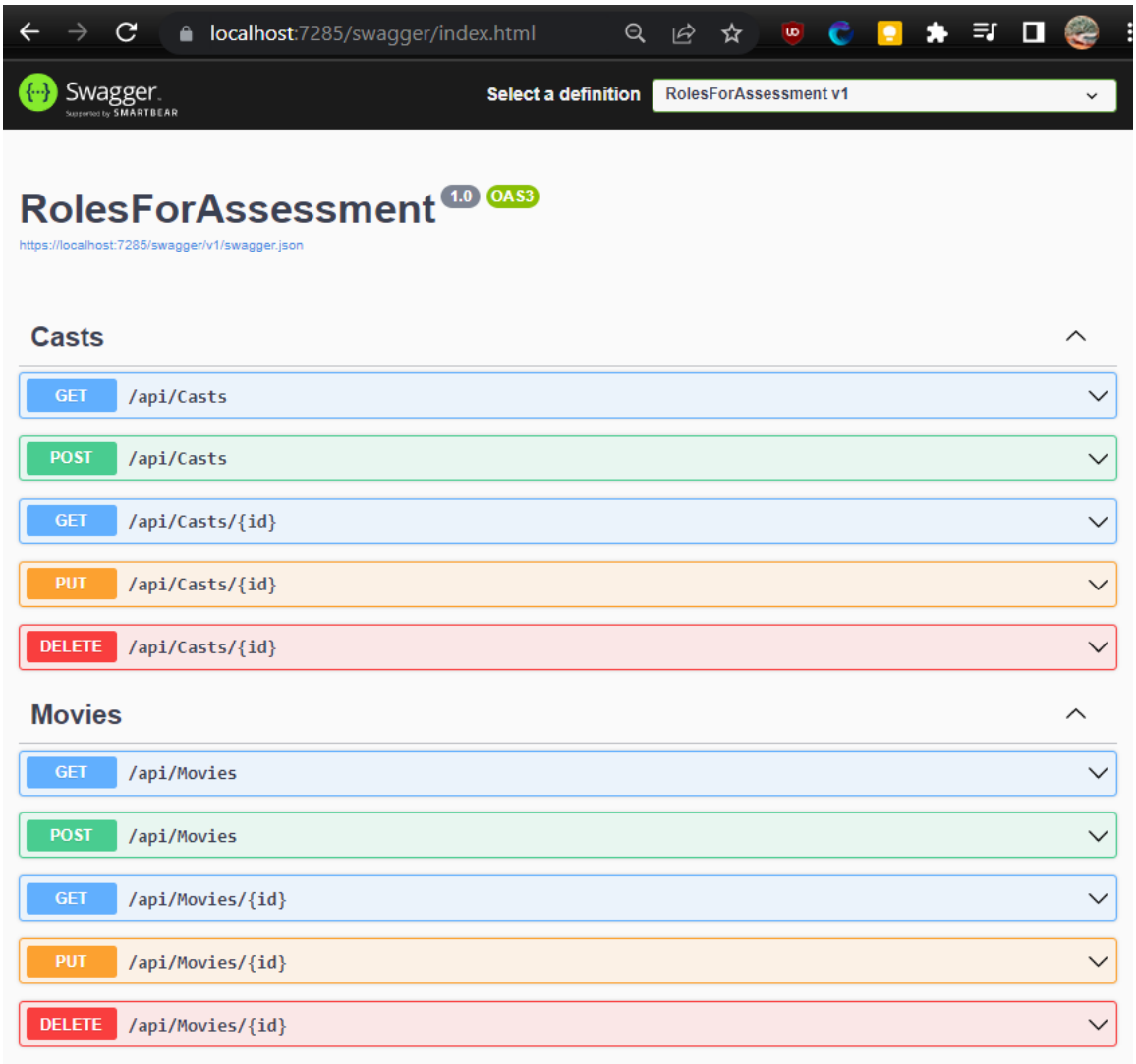
Add the following to the Program.cs

```
builder.Services.AddSwaggerGen();

  app.UseSwagger();
  app.UseSwaggerUI();
```

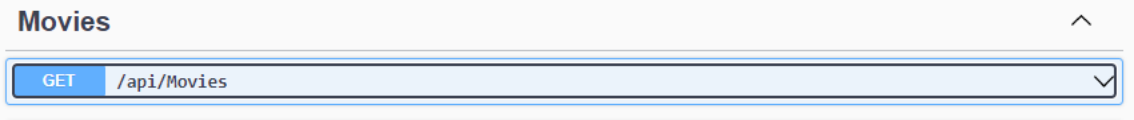
```
//=============END NEW SECURITY=========
builder.Services.AddSwaggerGen();
...
var app = builder.Build();

// Configure the HTTP request pipeline.
if (app.Environment.IsDevelopment())
{
    app.UseMigrationsEndPoint();
    app.UseSwagger();
    app.UseSwaggerUI();
}
else
{
```

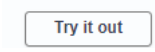
Run your program and add /swagger to your URL.

TaDa!

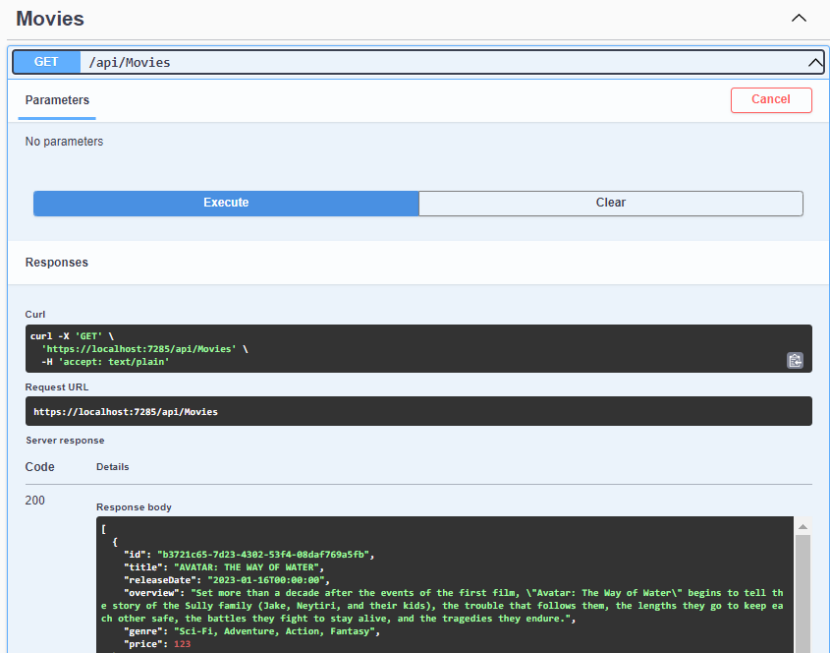


Go to Movies and open the screen 



Click 

Now when you press execute you can see the output from the API



This gives us a tool to work with our API.

Client React App

Download it here

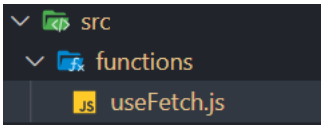
https://github.com/Netchicken/webapiforcorsReact

This nice little skeleton app gives us the basis for connecting to API's

A single Fetch function can take in any connection string and return back the data, meaning we don't have to call a separate one for each connection.

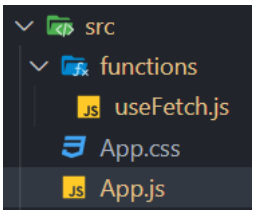
This is where I got the idea from https://dev.to/shaedrizwan/building-custom-hooks-in-react-to-fetch-data-4ig6 well worth following through.

Its running in a useEffect function so that it triggers when the URL changes.




```
You, 2 minutes ago | 1 author (You)
import { useEffect, useState } from "react";
import axios from "axios";

//pass in the URL that you want to fetch data for
export default function useFetch(url) {
  const [data, setData] = useState(null);
  const [error, setError] = useState(null);
  const [loading, setLoading] = useState(false);

  useEffect(() => {
    (async function () {
      try {
        setLoading(true);
        const response = await axios.get(url);
        setData(response.data); //have data set to data
      } catch (err) {
        setError(err); //otherwise set error
      } finally {
        setLoading(false); //set loading to false
      }
    })();
  }, [url]); //The only dependency we're going to put in the useEffect dependency array is Url because if the Url
  changes, we have to request new data.
  console.log(data);
  return { data, error, loading };
}

// https://dev.to/shaedrizwan/building-custom-hooks-in-react-to-fetch-data-4ig6
```

In App.js we just call this function



This is the data that we are getting back. I always make a record in the code to check

```
//data that we are getting back from the API
  // {
  //   "id": "3fa85f64-5717-4562-b3fc-2c963f66afa6",
  //   "title": "string",
  //   "releaseDate": "2023-01-16T00:00:00",
  //   "overview": "string",
  //   "genre": "string",
  //   "price": 0
  // }
```

```
import useFetch from "./functions/useFetch";
import Card from "react-bootstrap/Card";
import Container from "react-bootstrap/Container";
import Flex from "@react-css/flex";
//pass in the URL to the useFetch function if successful the data holds the data from the API
function App() {
  const { data, loading, error } = useFetch(
    "https://localhost:7285/api/Movies"
  );
  if (error) {
    console.log(error);
  }
  return (
    <Container fluid>
      {loading && <div>Loading...{error}</div>}
      {data && (
        <Flex flexDirection='row' justifyContent='center'>
          {data.map((item) => (
            <Card style={{ width: "28rem", padding: "10px" }}>
              <Card.Body>
                <h2 className='bodytext-Title'> {item.title}</h2>
                <p>Release Date... {item.releaseDate}</p>
                <p>Overview... {item.overview}</p>
                <h4>Genre ... {item.genre}</h4>
              </Card.Body>
            </Card>
          ))}
        </Flex>
      )}      You, 8 months ago • completed …
    </Container>
  );
}
export default App;
```

This passes in the URL, and returns back the Data the loading message, and any error,

```
const { data, loading, error } = useFetch(
  "https://localhost:7285/api/Movies"
);
if (error) {
  console.log(error);
}
```

This part is just the display. I am using react-bootstrap and flex.

```
import Card from "react-bootstrap/Card";
import Container from "react-bootstrap/Container";
import Flex from "@react-css/flex";
```

Its not great, but just a fast and simple way to see the output. We map through the data to extract out each item and display it in a Card.

```
return (
  <Container fluid>
    {loading && <div>Loading...{error}</div>}
    {data && (
      <Flex flexDirection='row' justifyContent='center'>
        {data.map((item) => (
          <Card style={{ width: "28rem", padding: "10px" }}>
            <Card.Body>
              <h2 className='bodytext-Title'> {item.title}</h2>
              <p>Release Date... {item.releaseDate}</p>
              <p>Overview... {item.overview}</p>
              <h4>Genre ... {item.genre}</h4>
            </Card.Body>
          </Card>
        ))}
      </Flex>      You, 21 hours ago • fixed css …
    )}
  </Container>
);
```

```
import useFetch from "./functions/useFetch";
import Card from "react-bootstrap/Card";
import Container from "react-bootstrap/Container";
import Flex from "@react-css/flex";
//pass in the URL to the useFetch function if successful the data holds the data from the API
function App() {
  const { data, loading, error } = useFetch(
    "https://localhost:7285/api/Movies"
  );
  if (error) {
    console.log(error);
  }
  return (
    <Container fluid>
      {loading && <div>Loading...{error}</div>}
      {data && (
        <Flex flexDirection='row' justifyContent='center'>
          {data.map((item) => (
            <Card style={{ width: "28rem", padding: "10px" }}>
              <Card.Body>
                <h2 className='bodytext-Title'> {item.title}</h2>
                <p>Release Date... {item.releaseDate}</p>
                <p>Overview... {item.overview}</p>
                <h4>Genre ... {item.genre}</h4>
              </Card.Body>
            </Card>
          ))}
        </Flex>
      )}
    </Container>
  );
}
export default App;
```

Run the app and connect to Movie API – CORS stops transmission

The first time we do this we get nothing. Which is not very helpful.

However going F12 brings up the console and we can see that CORS is preventing us from connecting.

```
Access to XMLHttpRequest at 'https://localhost:7285/api/Movies' from origin 'http://localhost: localhost/:1
3000' has been blocked by CORS policy: No 'Access-Control-Allow-Origin' header is present on the requested
resource.
▸GET https://localhost:7285/api/Movies net::ERR_FAILED 200                                  bundle.js:1107
```

9. Enable Cross-Origin Requests (CORS) in ASP.NET Core

https://learn.microsoft.com/en-us/aspnet/core/security/cors?view=aspnetcore-7.0

In Program.cs

Add in the following code, taken from above.

```
using RolesForAssessment.Data;

var CORSAllowSpecificOrigins = "_CORSAllowed";

var builder = WebApplication.CreateBuilder(args);


builder.Services.AddCors(setupAction: options =>
{
    options.AddPolicy(name: CORSAllowSpecificOrigins,
                      configurePolicy: policy =>
                      {
                          policy.WithOrigins(origins: "http://localhost:3000", "http://www.contoso.com");
                      });
});
```

In React here is my server path that we need to add in

```
You can now view webapiforcors in the browser.

  Local:            http://localhost:3000
  On Your Network:  http://10.7.200.4:3000
```

In → .vscode / {..} launch.json

```
var CORSAllowSpecificOrigins = "_CORSAllowed";
var builder = WebApplication.CreateBuilder(args);
builder.Services.AddCors(options =>
{
    options.AddPolicy(name: CORSAllowSpecificOrigins,
                      policy =>
                      {
 policy.WithOrigins("http://localhost:3000", "http://www.contoso.com");
                      });
});
```

Also in Program.cs add in this near the end

```
app.UseRouting();


app.UseCors(policyName: CORSAllowSpecificOrigins);

app.UseAuthentication();
```

```
app.UseCors(CORSAllowSpecificOrigins);
```

TaDa!

Once we get a connection made then the

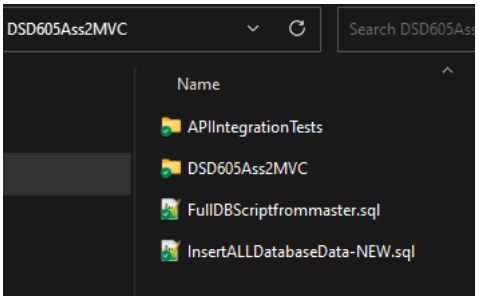
10. Integration Testing with XUnit

https://xunit.net/

xUnit.net is a free, open source, community-focused unit testing tool for the .NET Framework. Written by the original inventor of NUnit v2, **xUnit.net is the latest technology for unit testing** C#, F#, VB.NET and other .NET languages. xUnit.net works with ReSharper, CodeRush, TestDriven.NET and Xamarin.

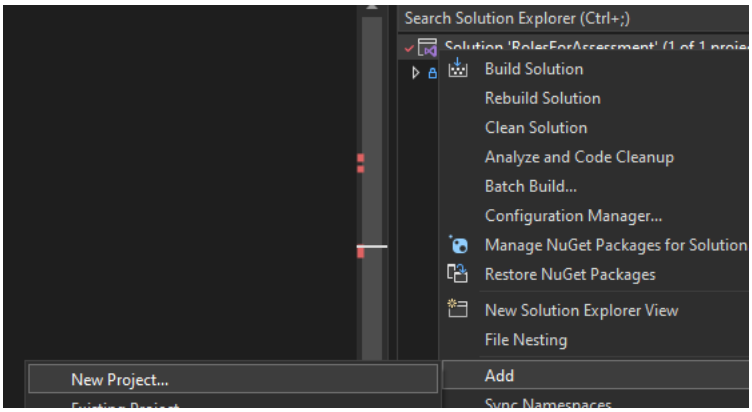
Unit and Integration tests are their own projects, so they will have their own folder and are not in the folder of your project.

As a result you need to make a folder to hold your Project and your Unit test like this.

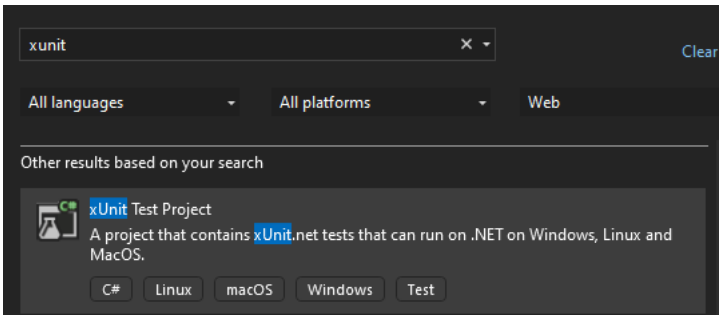


https://xunit.net/docs/getting-started/netcore/visual-studio

Go Solution / Add / New Project

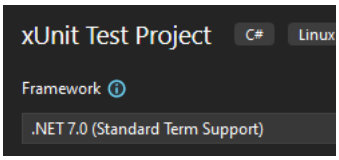


Install

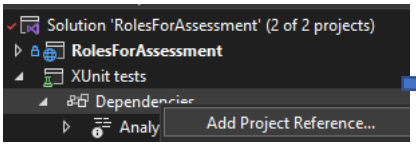


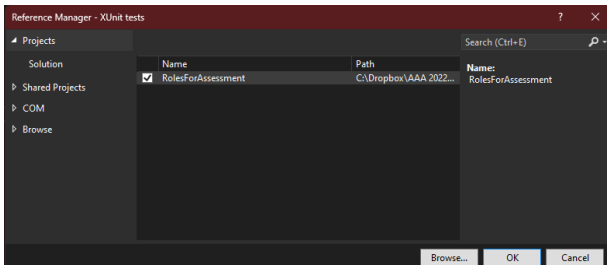
Name XUnit Tests

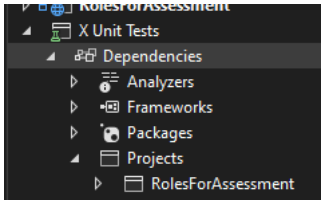
Choose the framework level that you made your project in. 6 or 7



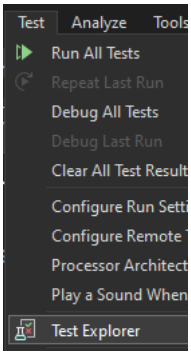
Right click on Dependencies and choose Add Project Reference

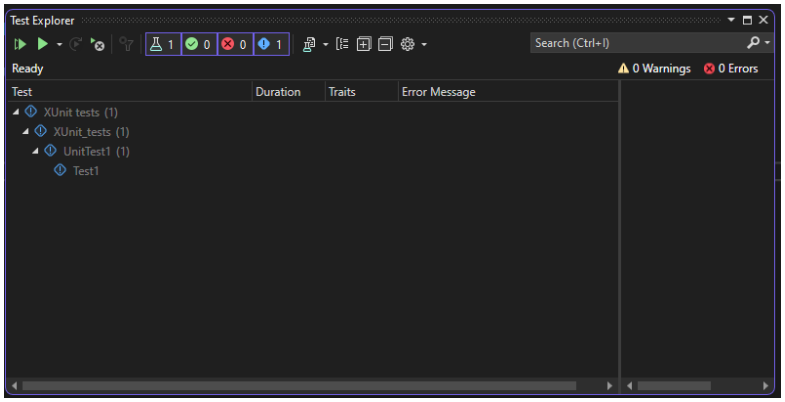






Open the test Explorer





https://gunnarpeipman.com/aspnet-core-integration-tests-users-roles/

https://gunnarpeipman.com/aspnet-core-test-controller-fake-user/

https://github.com/gpeipman/AspNetCoreTests

https://www.youtube.com/watch?v=ULJ3UEezisw Unit Testing Web API

https://code-maze.com/unit-testing-aspnetcore-web-api/

Testing the API's

Problem: We can't test the code that runs in the API because it pulls data from the database.

So we can't test that database unless the project is running. But we can't run our unit test when the project is running. Hmmmm…

Using an Integration test to test the API

https://learn.microsoft.com/en-us/aspnet/core/test/integration-tests?view=aspnetcore-7.0

Integration tests evaluate an app's components on a broader level than unit tests. Unit tests are used to test isolated software components, such as individual class methods.

Integration tests confirm that two or more app components work together to produce an expected result, possibly including every component required to fully process a request.

These broader tests are used to test the app's infrastructure and whole framework, often including the following components:

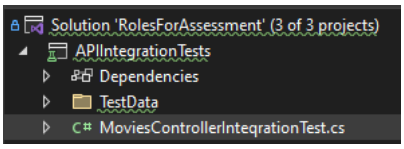
- Database
- File system
- Network appliances
- Request-response pipeline

In contrast to unit tests, integration tests:

- Use the actual components that the app uses in production.
- Require more code and data processing.
- Take longer to run.

Create an XUnit Integration Test

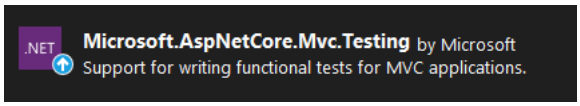
Make a Unit test Project but name it **APIIntegrationTests**. After making the Project rename UnitTest1 to MoviesControlerIntegrationTest



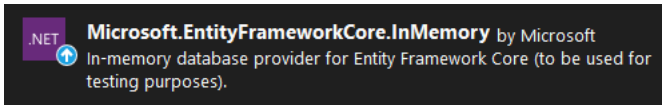
Install Nuget packages

Install to the main project AND the **APIIntegrationTests**.

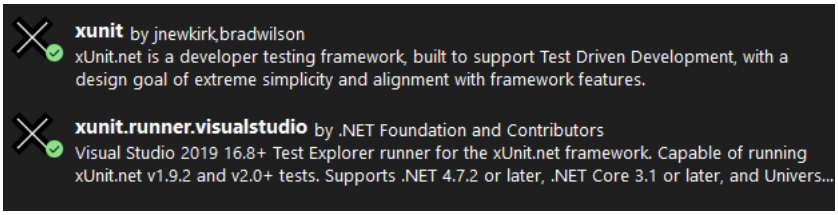
- AspNetCore.Mvc.Testing



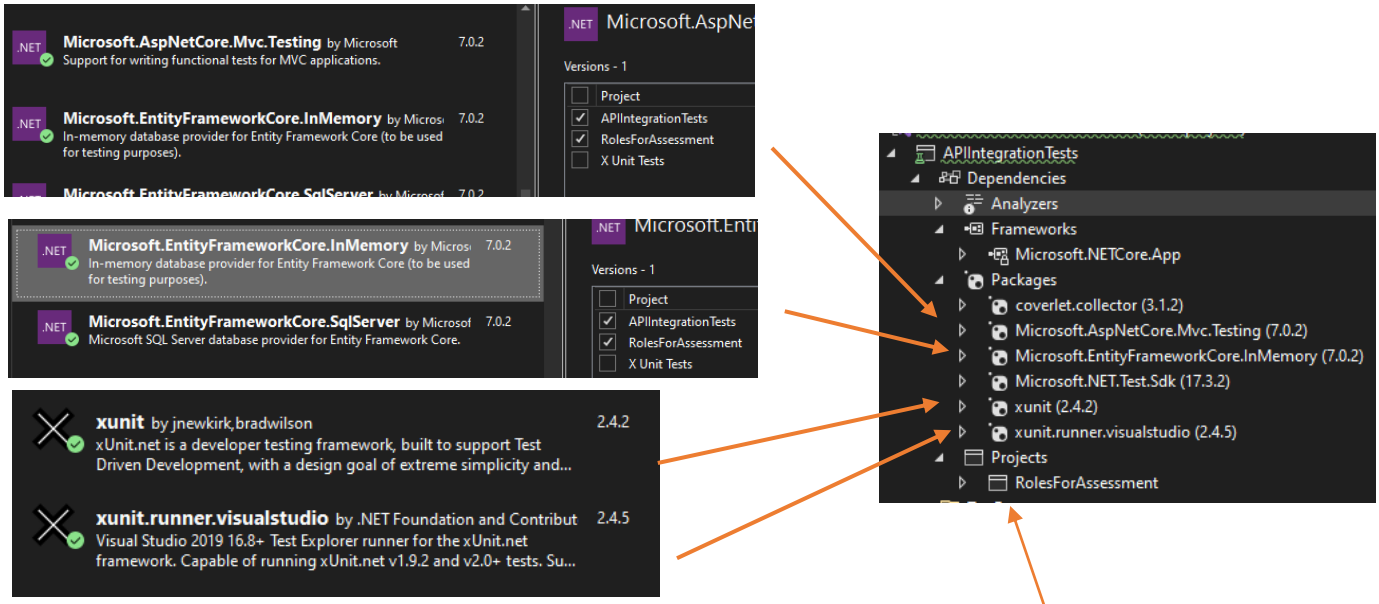
- Microsoft.EntityFrameworkCore.InMemory



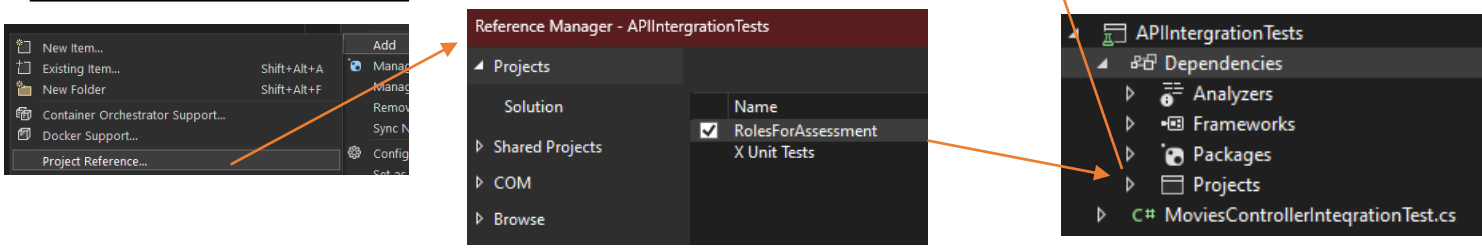
- Xunit – if not installed



Check your Packages so that you have the following:



Reference the Project



Create WebApplicationFactory

We need to store the results from WebApi's in the memory so the Tests can use them.

So we can use the WebApplicationFactory class from **Microsoft.EntityFrameworkCore.InMemory** to create an **in-memory** factory configuration.

An in-memory database is a database that resides in volatile memory instead of on a physical disk.

Naturally, reads and writes for an in-memory database are many times faster than for a disk-based database, because the application need not wait for the data to be physically written to or read from the disk. Reading and writing data stored on a physical disk is a resource-intensive operation.

In-memory databases are often used for caching purposes, as they can hold a copy of often-used data in memory for quick access. You can also take advantage of in-memory databases to store transient data, i.e., data that does not need to be persisted to the disk.

Create a new class **TestingWebAppFactory** in our **APIIntegrationTests** program. It will throw an error immediately.

```
using Microsoft.AspNetCore.Mvc.Testing;

namespace RolesForAssessment
{
    0 references
    public class TestingWebAppFactory<TEntryPoint> : WebApplicationFactory<Program> where TEntryPoint : Program
    {
        0 references
        protected override void ConfigureWebHost(IWebHostBuilder builder)
        {

        }
    }
}
```

```
Solution 'RolesForAssessment' (3 of 3 projects)
  APIIntegrationTests
    Dependencies
    TestData
    MoviesControllerInteqrationTest.cs
    TestingWebAppFactory.cs
    Usings.cs
```

```
using Microsoft.AspNetCore.Mvc.Testing;

namespace RolesForAssessment
{
    public class TestingWebAppFactory<TEntryPoint> : WebApplicationFactory<Program> where TEntryPoint :
Program
    {
        protected override void ConfigureWebHost(IWebHostBuilder builder)
        {

        }
    }
}
```

```
CS0060  Inconsistent accessibility: base class 'WebApplicationFactory<Program>' is less accessible than class 'TestingWebAppFactory<TEntryPoint>'
CS0703  Inconsistent accessibility: constraint type 'Program' is less accessible than 'TestingWebAppFactory<TEntryPoint>'
```

The class doesn't recognize the **Program** class even though we have the reference from the main project.

That's because in .NET compiler generates the Program class behind the scenes as the internal class, thus making it inaccessible in our integration testing project.

Modify the Program.cs class

So to solve this, we can create a **public partial Program class** in the Program.cs file in the main project:

```
164    app.MapControllers();
165
166    app.MapRazorPages();
167
168    app.Run();
169
       3 references
170    public partial class Program { }
```

You can see then that its available to the TestingWebAppFactory class we made

```
RolesForAssessment\Program.cs (1)
    1 : using Microsoft.AspNetCore.Authorization;
RolesForAssessment\TestingWebAppFactory.cs (2)
    5 : public class TestingWebAppFactory<TEntryPoint> : WebApplicationFactory<Program> where TEntryPoint : Program
    5 : public class TestingWebAppFactory<TEntryPoint> : WebApplicationFactory<Program> where TEntryPoint : Program
Collapse All
3 references
public partial class Program { }
```

```
public partial class Program { }
```

Having fixed that lets add in the code for the ConfigureWebHost to our TestingWebAppFactory class

The WebApplicationFactory class is a factory that we can use to bootstrap an application in memory for functional end-to-end tests.

```
0 references
public class TestingWebAppFactory<TEntryPoint> : WebApplicationFactory<Program> where TEntryPoint : Program
{

    0 references
    protected override void ConfigureWebHost(IWebHostBuilder builder)
    {
        builder.ConfigureServices(configureServices: services =>
        {
            var descriptor = services.SingleOrDefault(predicate: d => d.ServiceType ==
                    typeof(DbContextOptions<ApplicationDbContext>));
            // we remove the ApplicationDbContext registration from the Program class
            if (descriptor != null)
                services.Remove(item: descriptor);

            //we add the database context to the service container and instruct it to use the in-memory
              database instead of the real database
            services.AddDbContext<ApplicationDbContext>(optionsAction: options =>
            {
                options.UseInMemoryDatabase(databaseName: "InMemoryMoviesTest");
            });


            //Finally, we ensure that we seed the data from the ApplicationDbContext class (The same data you
               inserted into a real SQL Server database).
            var sp = services.BuildServiceProvider();
            using (var scope = sp.CreateScope())

            using (var appContext = scope.ServiceProvider.GetRequiredService<ApplicationDbContext>())
            {
                try
                {
                    appContext.Database.EnsureCreated();
                }
                catch (Exception ex)
                {
                    //Log errors or do anything you think it's needed
                    throw;
                }
            }
        });
    }
```

```
builder.ConfigureServices(services =>
            {
                var descriptor = services.SingleOrDefault(d => d.ServiceType ==
                        typeof(DbContextOptions<ApplicationDbContext>));
// we remove the ApplicationDbContext registration from the Program class
                if (descriptor != null)
                    services.Remove(descriptor);

//we add the database context to the service container and instruct it to use the in-memory database
instead of the real database
                services.AddDbContext<ApplicationDbContext>(options =>
                {
                    options.UseInMemoryDatabase("InMemoryMoviesTest");
                });
//Finally, we ensure that we seed the data from the ApplicationDbContext class (The same data you
inserted into a real SQL Server database).
                var sp = services.BuildServiceProvider();
                using (var scope = sp.CreateScope())

                using (var appContext = scope.ServiceProvider.GetRequiredService<ApplicationDbContext>())
                {
                    try
                    {
                        appContext.Database.EnsureCreated();
                    }
                    catch (Exception ex)
                    {
                        //Log errors or do anything you think it's needed
                        throw;
                    }
                }
            });
```

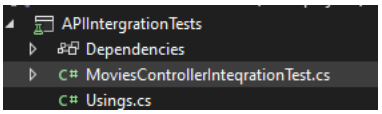
Here are the links to this tutorial and the project on Github

[integration-testing]

[https://github.com project]

Now we can go back to our testing area and bring in this class we created to our Integration test.

Replace the content of MoviesControllerInteqrationTest with the code that injects in our TestingWebAppFactory.




```
namespace APIIntergrationTests
{
    //The first thing we have to do is to implement the previously created TestingWebAppFactory class:
    1 reference
    public class MoviesControllerIntegrationTest : IClassFixture<TestingWebAppFactory<Program>>
    {
        private readonly HttpClient _client;
        //passing in the class using Injection and across to _client
        0 references
        public MoviesControllerIntegrationTest(TestingWebAppFactory<Program> factory)
            => _client = factory.CreateClient();

    }
}
```

This below is just the constructor done in a shorthand with the { } missing

```
//passing in the class using Injection and across to _client
0 references
public MoviesControllerIntegrationTest(TestingWebAppFactory<Program> factory)
    => _client = factory.CreateClient();
```

We implement the TestingWebAppFactory class with the IClassFixture interface and inject it in a constructor, where we create an instance of the HttpClient.

The IClassFixture interface is a decorator which indicates that tests in this class rely on a fixture to run. We can see that the fixture is our TestingWebAppFactory class.

We use this when you want to create a single test context and share it among all the tests in the class, and have it cleaned up after all the tests in the class have finished.

Sometimes test context creation and cleanup can be very expensive.

If you were to run the creation and cleanup code during every test, it might make the tests slower than you want. You can use the **class fixture** feature of xUnit.net **to share a single object instance among all tests in a test class.**

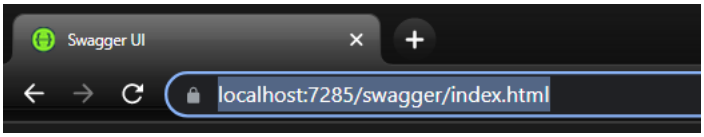
https://xunit.net/docs/shared-context

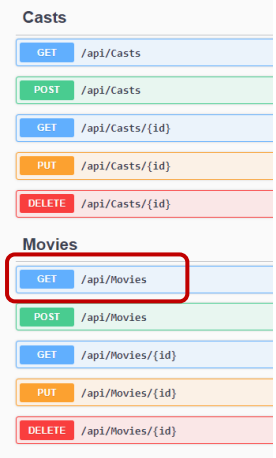
Getting Sample Data for the test

There are simpler ways of doing this but

So here is our movie data coming through the Swagger. The database that we are making doesn't hold any data, so we need to make some sample data for it.

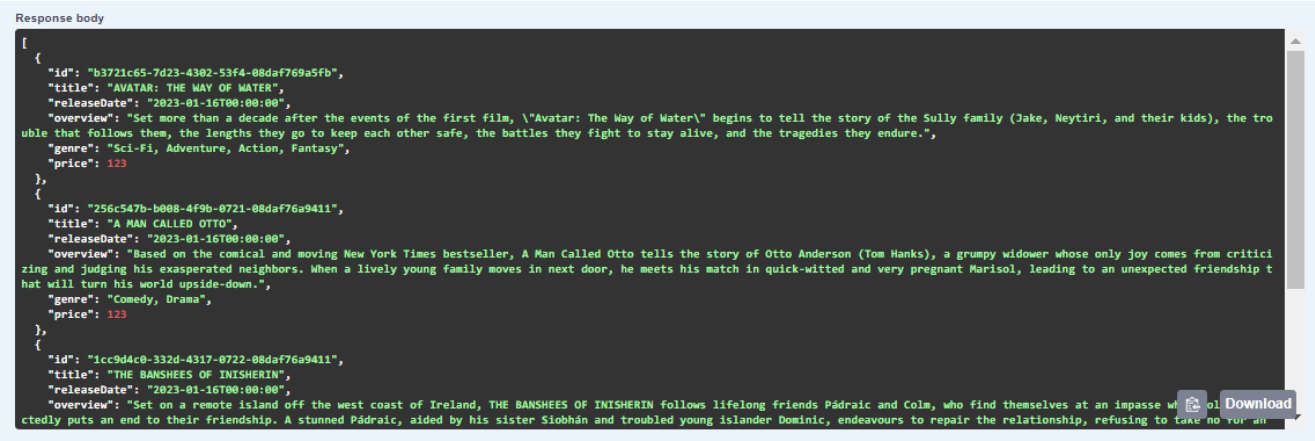
Actually its best to do this anyway. Imagine using real data in your test and someone changes the real data. That's not what we want, we don't care about the actual data, we care about the process of the data in the database.





The request URL shows the path we need later

In Get Movies, the response body holds all the data.

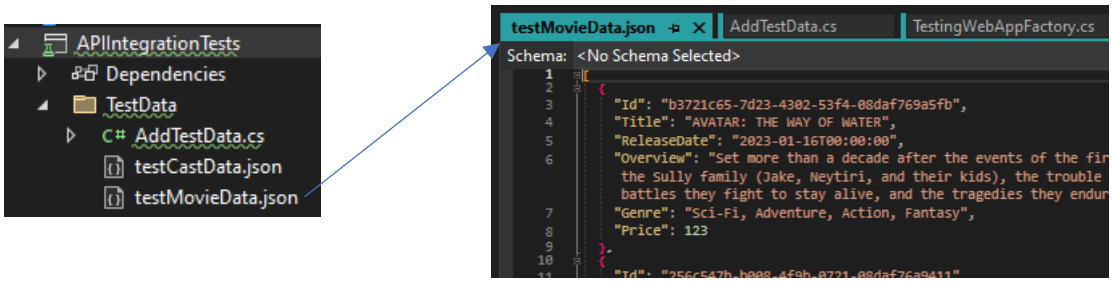



```
[
  {
    "id": "b3721c65-7d23-4302-53f4-08daf769a5fb",
    "title": "AVATAR: THE WAY OF WATER",
    "releaseDate": "2023-01-16T00:00:00",
    "overview": "Set more than a decade after the events of the first film, \"Avatar: The Way of Water\" begins to tell the story of the Sully family (Jake, Neytiri, and their kids), the tro
uble that follows them, the lengths they go to keep each other safe, the battles they fight to stay alive, and the tragedies they endure.",
    "genre": "Sci-Fi, Adventure, Action, Fantasy",
    "price": 123
  },
  {
    "id": "256c547b-b008-4f9b-0721-08daf76a9411",
    "title": "A MAN CALLED OTTO",
    "releaseDate": "2023-01-16T00:00:00",
    "overview": "Based on the comical and moving New York Times bestseller, A Man Called Otto tells the story of Otto Anderson (Tom Hanks), a grumpy widower whose only joy comes from critici
zing and judging his exasperated neighbors. When a lively young family moves in next door, he meets his match in quick-witted and very pregnant Marisol, leading to an unexpected friendship t
hat will turn his world upside-down.",
    "genre": "Comedy, Drama",
    "price": 123
  },
  {
    "id": "1cc9d4c0-332d-4317-0722-08daf76a9411",
    "title": "THE BANSHEES OF INISHERIN",
    "releaseDate": "2023-01-16T00:00:00",
    "overview": "Set on a remote island off the west coast of Ireland, THE BANSHEES OF INISHERIN follows lifelong friends Pádraic and Colm, who find themselves at an impasse wh
ctedly puts an end to their friendship. A stunned Pádraic, aided by his sister Siobhán and troubled young islander Dominic, endeavours to repair the relationship, refusing to take no
```


Create a folder called **TestData** in APIIntegrationTests

Create a new file called **testMoveData.json**

Copy the Json in the Response body into the new file.



Do the same for the Cast to a file named testCastData.json

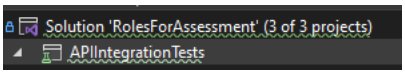
Check the Json files are registered correctly.

I had great trouble with the project not being able to find the files. This is quite typical for ASP.net, and there is a system of saving files in the www folder and calling them in.

However we don't have that in our Integration Project, and even if we do put it the main project, we want to keep the files OUT of the main project.

Eventually I checked the code behind the project and found that it was not listed there.

So check that you have the paths, when you click on the project file




```
<ItemGroup>
  <Content Include="TestData\testMovieData.json">
    <CopyToOutputDirectory>PreserveNewest</CopyToOutputDirectory>
    <ExcludeFromSingleFile>true</ExcludeFromSingleFile>
    <CopyToPublishDirectory>PreserveNewest</CopyToPublishDirectory>
  </Content>
</ItemGroup>
<ItemGroup>
  <Content Include="TestData\testCastData.json">
    <CopyToOutputDirectory>PreserveNewest</CopyToOutputDirectory>
    <ExcludeFromSingleFile>true</ExcludeFromSingleFile>
    <CopyToPublishDirectory>PreserveNewest</CopyToPublishDirectory>
  </Content>
</ItemGroup>
```

Create a class to add the data to the InMemory Database

1. Create a new class in the TestData folder called AddTestData.cs.
2. Make it a **static** class.
3. Create the **AddMovieData** method

```
//import json array and add to database context
1 reference
public static void AddMovieData(ApplicationDbContext context)
{
    var jsonString = File.ReadAllText(path: "TestData/testMovieData.json");

    //need to stop it being case sensitive the model is capital case and the json is not
    var options = new JsonSerializerOptions
    {//stop changing the case from uppercase to lower case for the first letter of the Key
        PropertyNameCaseInsensitive = true
    };

    var list = JsonSerializer.Deserialize<Movie[]>(json: jsonString, options);
    {
        foreach (var item in list)
        {
            context.Movie.Add(entity: item);
        }
        //save to the in memory database
        context.SaveChanges();
    }
}
```

```
//import json array and add to database context
        public static void AddMovieData(ApplicationDbContext context)
        {
            var jsonString = File.ReadAllText("TestData/testMovieData.json");

            //need to stop it being case sensitive the model is capital case and the json is not
            var options = new JsonSerializerOptions
            {//stop changing the case from uppercase to lower case for the first letter of the Key
                PropertyNameCaseInsensitive = true
            };

            var list = JsonSerializer.Deserialize<Movie[]>(jsonString, options);
            {
                foreach (var item in list)
                {
                    context.Movie.Add(item);
                }
                //save to the in memory database
                context.SaveChanges();
```

```
        }
    }
```

This was an interesting class to write. Firstly, we have to get all the data in the Json file into the method.

```
var jsonString = File.ReadAllText(path: "TestData/testMovieData.json");
```

That didn't work for a long time because the project couldn't find the file. This was because this was missing, that we checked above.

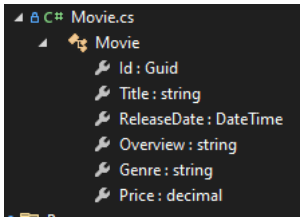
```
<ItemGroup>
  <Content Include="TestData\testMovieData.json">
```

Once I got the file in as a giant string I Deserialized it and pass it to the Movies model so it can be added to the DB.

```
var list = JsonSerializer.Deserialize<Movie[]>(json: jsonString, options);
```

But that too failed. Why? Case Sensitivity.

The Properties in the class are Capitalized. While the Json was lower case. So I made the Json capitalized as well




```
"Id": "b3721c6
"Title": "AVAT
"ReleaseDate":
"Overview": "S
 story of the
 each other sa
"Genre": "Sci-
"Price": 123
```

But it STILL didn't work. Because the Jason made the upper case letters Lower case in the Deserialization process.

So I found the PropertyNameCaseInsensitive = true option and added it. And it worked.

```
//need to stop it being case sensitive the model is capital case and the json is not
var options = new JsonSerializerOptions
{//stop changing the case from uppercase to lower case for the first letter of the Key
    PropertyNameCaseInsensitive = true
};

var list = JsonSerializer.Deserialize<Movie[]>(json: jsonString, options);
{
```

Capitalise the Json Keys.

Like below, for each entry.

```
{
  "Id": "b3721c65-7
  "Title": "AVATAR:
  "ReleaseDate": "2
  "Overview": "Set
   story of the Sul
   each other safe,
  "Genre": "Sci-Fi,
  "Price": 123
},
```

Now do the same for the Cast data

```
public static void AddCastData(ApplicationDbContext context)
{
    var jsonString = File.ReadAllText(path: "TestData/testCastData.json");

    var options = new JsonSerializerOptions
    {
        PropertyNameCaseInsensitive = true
    };

    var list = JsonSerializer.Deserialize<Cast[]>(json: jsonString, options);
    {
        foreach (var item in list)
        {
            context.Cast.Add(entity: item);
        }
        context.SaveChanges();
    }

}
```

That gives us 2 methods in the class

```
TestData
  C# AddTestData.cs
    AddTestData
      AddMovieData(ApplicationDbContext) : void
      AddCastData(ApplicationDbContext) : void
```

Pass the mock data to the WebApplicationFactory

In your TestingWebAppFactory class add in the methods that save the mock data to the in memory database

```
using (var appContext = scope.ServiceProvider.GetRequiredService<ApplicationDbContext>())
{
    try
    {
        appContext.Database.EnsureCreated();

        // Seed the database with test data.
        AddTestData.AddMovieData(context: appContext);
        AddTestData.AddCastData(context: appContext);
```

Now we can make some tests!!!

Create our Integration Tests.

We are going to make two tests that check that the database is returning data.

[aspnet-core-integration-testing](#) instructions [Github page](#)

This line is the key, it returns a Boolean True/False

```
Assert.Contains("A MAN CALLED OTTO", responseString);=
```

If the page contains A MAN CALLED OTTO the test passes, otherwise it fails.

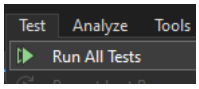
```
namespace APIIntegrationTests
{
    //The first thing we have to do is to implement the previously created TestingWebAppFactory class:
    1 reference
    public class MoviesControllerIntegrationTest : IClassFixture<TestingWebAppFactory<Program>>
    {
        private readonly HttpClient _client;
        //passing in the class using Injection and across to _client in the constructor
        0 references
        public MoviesControllerIntegrationTest(TestingWebAppFactory<Program> factory)
            => _client = factory.CreateClient();

        // GET: api/Movies
        [Fact]
        0 references
        public async Task IndexReturnsMovies()
        {
            var response = await _client.GetAsync(requestUri: "/Movies");
            response.EnsureSuccessStatusCode();
            var responseString = await response.Content.ReadAsStringAsync();
            Assert.Contains(expectedSubstring: "A MAN CALLED OTTO", actualString: responseString);
        }

        // GET: api/Casts
        [Fact]
        0 references
        public async Task IndexReturnsCast()
        {
            var response = await _client.GetAsync(requestUri: "/Casts");
            response.EnsureSuccessStatusCode();
            var responseString = await response.Content.ReadAsStringAsync();
            Assert.Contains(expectedSubstring: "Sigourney", actualString: responseString);
        }
    }
}
```

```
public class MoviesControllerIntegrationTest : IClassFixture<TestingWebAppFactory<Program>>
    {
        private readonly HttpClient _client;
        //passing in the class using Injection and across to _client in the constructor
        public MoviesControllerIntegrationTest(TestingWebAppFactory<Program> factory)
            => _client = factory.CreateClient();

        // GET: api/Movies
        [Fact]
        public async Task IndexReturnsMovies()
        {
            var response = await _client.GetAsync("/Movies");
            response.EnsureSuccessStatusCode();
            var responseString = await response.Content.ReadAsStringAsync();
            Assert.Contains("A MAN CALLED OTTO", responseString);
        }
```

```
        // GET: api/Casts
        [Fact]
        public async Task IndexReturnsCast()
        {
            var response = await _client.GetAsync("/Casts");
            response.EnsureSuccessStatusCode();
            var responseString = await response.Content.ReadAsStringAsync();
            Assert.Contains("Sigourney", responseString);
        }
    }
```

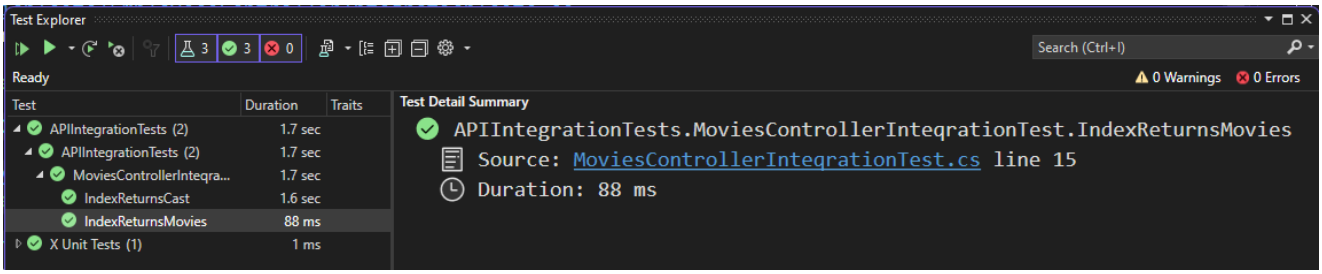
Run the tests

Go Test/Run All Tests



That should bring up our Test Explorer and run the tests.

Success!!! Happy ticks everywhere.



That also puts a happy tick on your method as well

```
// GET: api/Movies
[Fact]
0 references
public async Task IndexReturnsMovies()
{
```

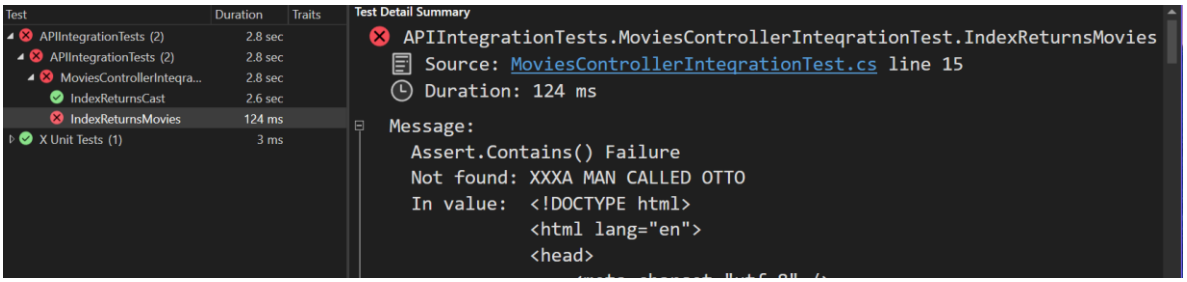
Break the Test.

You actually learn more about it by breaking the test.

Modify your Assert

```
Assert.Contains(expectedSubstring: "XXXA MAN CALLED OTTO", actualString: responseString)
```

Get a sad cross




```
// GET: api/Movies
[Fact]
0 references
public async Task IndexReturnsMovies()
{
```

Note that it tells you what the problem is, and where it looked.

```
Message:
  Assert.Contains() Failure
  Not found: XXXA MAN CALLED OTTO
```

It is also checking the entire page, not just the DB. This means you can use it for checking on your page as well.

```
In value:  <!DOCTYPE html>
           <html lang="en">
           <head>
               <meta charset="utf-8" />
               <meta name="viewport" conter
               <title>Index - RolesForAsses
               <link rel="stylesheet" href=
               <link rel="stylesheet" href=
               <link rel="stylesheet" href=
           </head>
           <body>
               <header>
```

Create a Single Entry instead of importing them all

This just adds a single movie.

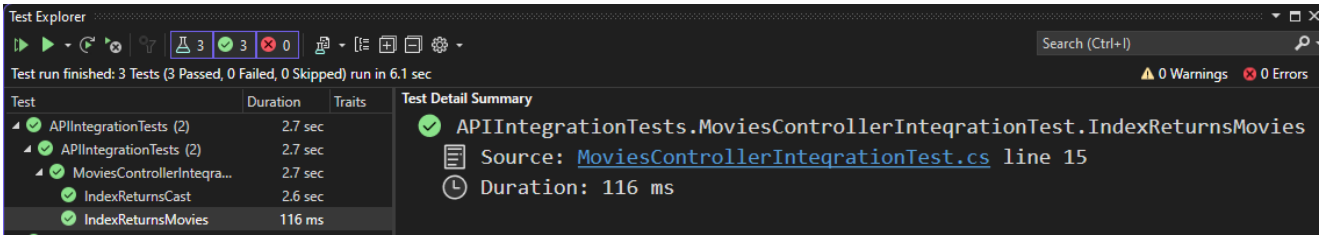
```
0 references
public static void AddSingleMovieData(ApplicationDbContext context)
{
    Movie movie = new Movie();

    movie.Id = Guid.Parse(input: "b3721c65-7d23-4302-53f4-08daf769a5fb");
    movie.Title = "A MAN CALLED OTTO";
    movie.ReleaseDate = DateTime.Parse(s: "2023 - 01 - 16T00: 00:00");
    movie.Overview = "Based on the comical and moving New York Times bestseller, A Man Called Otto tells the story
       of Otto Anderson (Tom Hanks), a grumpy widower whose only joy comes from criticizing and judging his
      exasperated neighbors. When a lively young family moves in next door, he meets his match in quick-witted and
       very pregnant Marisol, leading to an unexpected friendship that will turn his world upside-down.";
    movie.Genre = "Sci-Fi, Adventure, Action, Fantasy";
    movie.Price = 123;

    context.Movie.Add(entity: movie);
    //save to the in memory database
    context.SaveChanges();
}
```

```
public static void AddSingleMovieData(ApplicationDbContext context)
        {
            Movie movie = new Movie();
            movie.Id = Guid.Parse("b3721c65-7d23-4302-53f4-08daf769a5fb");
            movie.Title = "A MAN CALLED OTTO";
            movie.ReleaseDate = DateTime.Parse("2023 - 01 - 16T00: 00:00");
            movie.Overview = "Based on the comical and moving New York Times bestseller, A Man Called
Otto tells the story of Otto Anderson (Tom Hanks), a grumpy widower whose only joy comes from criticizing
and judging his exasperated neighbors. When a lively young family moves in next door, he meets his match
in quick-witted and very pregnant Marisol, leading to an unexpected friendship that will turn his world
upside-down.";
            movie.Genre = "Sci-Fi, Adventure, Action, Fantasy";
            movie.Price = 123;

            context.Movie.Add(movie);
            //save to the in memory database
            context.SaveChanges();
        }
```

Modify the method name in TestWebAppFactory

```
// Seed the database with test data.
AddTestData.AddSingleMovieData(context: appContext);
AddTestData.AddCastData(context: appContext);
```

Run the test – Happy Tick!



11. ASP.Net Core and ReactJS security tests

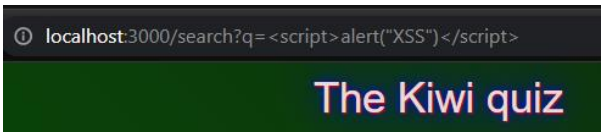
Here are some common security tests to provide technical support in ReactJS and ASP.net Core 7:

Cross-Site Scripting (XSS) tests:

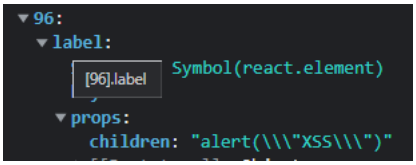
Test the application for vulnerabilities that allow malicious scripts to be executed within the user's browser.

ReactJS

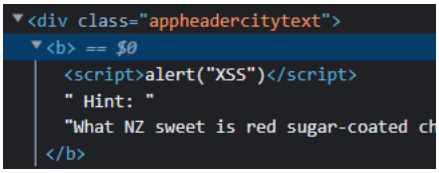
1. Verify that user-supplied data is properly sanitized before being displayed in the application. For example, when displaying user comments or messages, make sure that any HTML or JavaScript code entered by the user is properly escaped to prevent XSS attacks.
2. Test for reflected XSS attacks by attempting to inject malicious code into query parameters in the URL. For example, try accessing the application with a URL that includes a script tag, such as: http://example.com/search?q=<script>alert("XSS")</script>. If the application is vulnerable, the script should execute when the page loads.



3. Test for stored XSS attacks by attempting to inject malicious code into fields that are stored in the application's database. For example, try entering a comment or message that includes a script tag, such as: <script>alert("XSS")</script>. If the application is vulnerable, the script should execute whenever the comment or message is displayed.



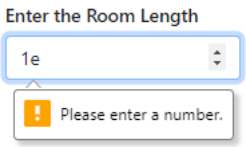
4. Test for XSS attacks in dynamically generated content by attempting to inject malicious code into fields that are generated by the application. For example, try entering a user name that includes a script tag, such as: <script>alert("XSS")</script>. If the application is vulnerable, the script should execute whenever the user name is displayed.



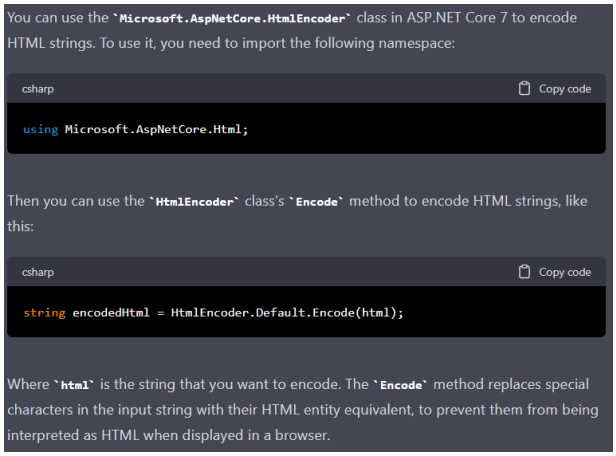
Asp.net Core

To test for Cross-Site Scripting (XSS) vulnerabilities in ASP.NET Core 7, you can follow these steps:

1. Identify user input fields: Look for all fields where users can input data, such as search boxes, comment forms, contact forms, etc.
2. Test each field: Attempt to inject malicious code into each field to see if it is reflected on the page. For example, try entering the following in a text field: <script>alert("XSS")</script>



3. Validate user input: Use the System.Web.HtmlEncode method to encode any user input before it is displayed on the page. – protect HTML code.



You can use the `Microsoft.AspNetCore.HtmlEncoder` class in ASP.NET Core 7 to encode HTML strings. To use it, you need to import the following namespace:

csharp Copy code

```csharp
using Microsoft.AspNetCore.Html;
```

Then you can use the `HtmlEncoder` class's `Encode` method to encode HTML strings, like this:

csharp Copy code

```csharp
string encodedHtml = HtmlEncoder.Default.Encode(html);
```

Where `html` is the string that you want to encode. The `Encode` method replaces special characters in the input string with their HTML entity equivalent, to prevent them from being interpreted as HTML when displayed in a browser.

4. Use anti-XSS libraries: Implement an anti-XSS library such as Microsoft Anti-XSS Library to automatically encode user input and protect your application. https://learn.microsoft.com/en-us/aspnet/core/security/cross-site-scripting?view=aspnetcore-7.0

Cross-Site Request Forgery (CSRF) tests:

Verify that the application properly implements measures to prevent unauthorized actions performed on behalf of the user.

ReactJS

1. Verify that the application implements CSRF protection by checking for a unique token in each form and API request. The token should be stored in a cookie on the user's device and passed as a parameter in each request to ensure that the request is coming from the same user who initiated the action.
2. Test for CSRF vulnerabilities by attempting to execute unauthorized actions on behalf of the user. For example, try submitting a form or making an API request from a different website or application. If the application is vulnerable, the request should be executed as if it came from the user.
3. Verify that the application properly handles invalid CSRF tokens by checking for error messages or exceptions when submitting a form or making an API request with an invalid token. If no errors are displayed, the application is likely vulnerable to CSRF attacks.
4. Test for CSRF attacks in dynamically generated content by attempting to submit a form or make an API request with an invalid token. For example, try submitting a form that includes a hidden field with an invalid CSRF token. If the application is vulnerable, the request should be executed as if it came from the user.

Asp.net Core

To test for Cross-Site Request Forgery (CSRF) vulnerabilities in ASP.NET Core 7, you can follow these steps:

1. Create a malicious website: Create a malicious website that will submit a form to your ASP.NET Core 7 application with a hidden input field.
2. Send a request: Send a request to your ASP.NET Core 7 application with the hidden input field, modifying the field in the Chrome debugger attempting to perform a privileged action such as updating a user's profile or submitting a form. https://stackoverflow.com/questions/72998610/asp-net-core-not-encoding-input-value-on-post-back value="<script>alert('test')

```
▼<div b-ei2zun9p2v class="container">
  ▼<main b-ei2zun9p2v role="main" class="pb-3">
    ▼<form action="/submit" method="post">
       <!-- hidden input field -->
       <input type="hidden" name="hiddenField" value="<script>alert('test')"> == $0
```

to do

3. Check the response: Observe the response from the application and determine whether the privileged action was performed or not. If the action was performed, your application may be vulnerable to CSRF.
4. Implement CSRF protection: Implement CSRF protection in your ASP.NET Core 7 application by using the Microsoft.AspNetCore.Antiforgery package, which provides an API for generating and validating anti-forgery tokens.
5. Test again: Test your application again to see if the CSRF vulnerability has been mitigated.

Session Management tests:

Verify that the application properly manages user sessions, including the protection of session IDs and the termination of sessions after a period of inactivity.

ReactJS

Session Management overview – if your App is using session management

1. Session Timeout Test: Test to ensure that the user session is automatically logged out after a specified amount of time.
2. Session Renewal Test: Test to ensure that the user session is renewed if the user is still active within the specified timeout period.
3. Session Invalidation Test: Test to ensure that the user session is invalidated if the user logs out or the session is terminated due to inactivity.
4. Session Persistence Test: Test to ensure that the user session is persistent across multiple pages and browser sessions.
5. Session ID Management Test: Test to ensure that the session ID is properly managed and encrypted to prevent unauthorized access.
6. Session Management Error Handling Test: Test to ensure that the system handles errors related to session management, such as incorrect session IDs or invalidated sessions.
7. Session Token Management Test: Test to ensure that the session token is properly managed and encrypted to prevent unauthorized access.
8. Session Renewal After Login Test: Test to ensure that the user session is renewed after the user logs in again.
9. Session Renewal After Logout Test: Test to ensure that the user session is not renewed after the user logs out.

Asp.net Core

To test for session management vulnerabilities in ASP.NET Core 7, you can follow these steps:

Session and state management in ASP.NET Core – setting up your sessions.

1. Test session fixation: Try to set a user's session ID to a specific value and see if the application accepts it, which could allow an attacker to hijack the user's session.
2. Test session hijacking: Attempt to hijack a user's session by intercepting and modifying the session ID, which could allow an attacker to gain access to sensitive information or perform actions on behalf of the user.
3. Test session timeout: Test the session timeout mechanism of the application to see if it is set to a reasonable value that balances security and usability.
4. Test session renewal: Test the session renewal mechanism of the application to see if it is renewing sessions frequently enough to prevent session timeouts and ensure a secure user experience.
5. Test session data protection: Test the protection mechanism of the session data to see if it is being stored securely and encrypted when stored on the client-side.

Input Validation tests:

Test the application's ability to validate user input and reject malicious data.

ReactJS

1. Email Validation Test: Test to ensure that the system only accepts valid email addresses as input.
2. Phone Number Validation Test: Test to ensure that the system only accepts valid phone numbers as input.
3. Required Field Validation Test: Test to ensure that required fields are not left blank.
4. String Length Validation Test: Test to ensure that input strings are of the required length.
5. Special Character Validation Test: Test to ensure that input data does not contain special characters that can cause security threats.
6. Numeric Validation Test: Test to ensure that input data is a valid number.
7. URL Validation Test: Test to ensure that input data is a valid URL.
8. Date Validation Test: Test to ensure that input data is a valid date.
9. Time Validation Test: Test to ensure that input data is a valid time.
10. File Format Validation Test: Test to ensure that uploaded files are of the required format and size.

Configuration Management tests:

Verify that the application's configuration is secure, including the protection of sensitive data and the proper use of encryption.

ReactJS

1. **Environment Configuration Test:** Test to ensure that the system is configured correctly for different environments, such as development, testing, and production.
2. **Dependency Management Test:** Test to ensure that the system is using the correct versions of dependencies and libraries, and that they are up to date.
3. **Build Configuration Test:** Test to ensure that the system's build process is configured correctly and can be repeated consistently.
4. **Deployment Configuration Test:** Test to ensure that the system is deployed correctly and can be deployed consistently.
5. **Security Configuration Test:** Test to ensure that the system's security settings are configured correctly and that the system is protected against threats.
6. **Performance Configuration Test:** Test to ensure that the system's performance settings are configured correctly and that the system is optimized for performance.
7. **Logging Configuration Test:** Test to ensure that the system's logging settings are configured correctly and that logs are being generated and stored properly.
8. **Monitoring Configuration Test:** Test to ensure that the system's monitoring settings are configured correctly and that the system is being monitored effectively.
9. **Backup Configuration Test:** Test to ensure that the system's backup settings are configured correctly and that backups are being performed and stored properly.
10. **Scalability Configuration Test:** Test to ensure that the system's scalability settings are configured correctly and that the system can scale to accommodate changes in traffic.

Asp.net Core

To test for configuration management vulnerabilities in ASP.NET Core 7, you can follow these steps:

1. **Test sensitive data exposure:** Check if sensitive information, such as connection strings, secrets, and API keys, are stored in a secure manner, such as in environment variables or encrypted configuration files.
2. **Test configuration file exposure:** Verify that configuration files are not stored in a publicly accessible location, such as the application's root directory, to prevent unauthorized access to sensitive information.
3. **Test configuration file updates:** Test the mechanism for updating configuration files to ensure that they are being updated securely and that the application continues to function as expected after the update.
4. **Test configuration file parsing:** Test the configuration file parsing mechanism to see if it is properly handling malformed configuration files and preventing code execution.
5. **Test configuration file encryption:** Test the encryption mechanism used to secure configuration files to ensure that sensitive information is protected.

Penetration Testing:

Simulate an attacker attempting to exploit known vulnerabilities in the application to gain unauthorized access.

ReactJS

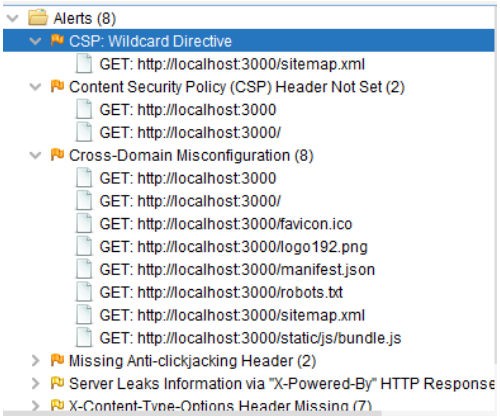
1. **Information Gathering:** The tester will gather information about the ReactJS application, such as its architecture, components, and APIs, to identify potential attack surfaces.
2. **Scanning:** The tester will scan the application for known vulnerabilities and weaknesses using automated tools such as Nessus or OWASP ZAP.

OWASP® Zed Attack Proxy (ZAP)

The world's most widely used web app scanner. Free and open source. Actively maintained by a dedicated international team of volunteers. A GitHub Top 1000 project.

Note that ZAP requires Java 11+ first in order to run.

Another Owasp Zap Tutorial Run the automatic test on your ReactJS project



3. **Exploitation:** The tester will attempt to exploit any vulnerabilities identified in the scanning phase, such as SQL injection or cross-site scripting attacks.
4. **Post-Exploitation:** The tester will evaluate the impact of the exploitation, such as whether sensitive data can be accessed or modified.
5. **Reporting:** The tester will provide a comprehensive report detailing the findings and recommendations for remediation.

Asp.net Core

To perform a penetration test on ASP.NET Core 7, you can follow these steps:

1. **Plan and scope the test:** Define the objectives of the test, the systems and applications to be tested, and the methods to be used. This will help ensure that the test is comprehensive and covers the areas of greatest risk.
2. **Gather information:** Conduct reconnaissance to gather information about the target system, including IP addresses, software versions, and open ports.
3. **Identify vulnerabilities:** Use tools and techniques to identify vulnerabilities in the target system, including network scans, application scans, and manual testing.
4. **Exploit vulnerabilities:** Attempt to exploit vulnerabilities in the target system to gain access to sensitive information or perform actions on behalf of the user.
5. **Report findings:** Document the results of the test, including any vulnerabilities found, how they were exploited, and recommendations for remediation.

12. Resources

Cryptographic Failures

Cryptography, the underpinning of a secure authentication framework, is a highly specialized area of work that is best left to experts.

Cryptographic failures result in sensitive data being leaked, and are the second most common types of vulnerabilities found in applications according to the OWASP. In this section we will look at a few specific examples of cryptographic failures and how to avoid them.

Storing passwords

Rule number one in the web security handbook (if there is such a thing) is never store user passwords in plain text. People are creatures of habit. Many of them will use the same password on your site that they use for every other site they have an account with. In the event of a data breach, you could be giving crooks the keys to all other websites that your users access.

The website, Have I Been Pwned lists over 500 websites that have suffered data breaches, many of them familiar names, and a significant number of them yielding up passwords in plain text.

So how should you store users' passwords and other sensitive data?

Some developers are tempted to encrypt passwords so that they can be decrypted for comparison with the password submitted in a login form. Because encryption is designed to be reversed, it is not secure should someone get access to the means for decrypting the data.

The Identity framework secures user passwords by hashing them. Hashing is a one-way process that should always produce the same output for a given input. For a hashing algorithm to be considered cryptographically secure, it should be infeasible to reverse a hash through brute force computation, based on the resources (equipment and time) required to do the job. The reality is that processing reduces in cost and increases in power all the time, which has resulted in older hashing algorithms such as MD5 to become "broken".

ASP.NET Core Identity uses the PBKDF2 hashing algorithm (which is one of the options recommended by the OWASP). The number of times that the actual hashing function is iterated (the work factor) determines the difficulty of breaking PBKDF2 hashes. The recommended number of iterations increases all the time as hardware gets faster. In 2010, the number of iterations hardcoded into ASP.NET was 1000.

The default number of iterations for PBKDF2 (Password-Based Key Derivation Function 2) is typically around 10,000 to 100,000 iterations. The exact number of iterations can vary depending on the implementation and the desired level of security. The number of iterations is used to increase the computational cost of deriving the key from the password, making it more difficult for an attacker to crack the password.

The advice is to increase this number to the largest that the authenticating server will tolerate. The OWASP currently recommends 310,000 iterations if you need to conform to the US Federal Information Processing Standards (FIPS).

Password hashing in Identity is performed by the PasswordHasher class. You can configure the number of iterations of the hashing function through the PasswordHasherOptions class. The following listing demonstrates how to increase the number of iterations to the number recommended to conform to FIPS.

Increase the PasswordHasher PBKDF2 iteration count to 310,000

```
builder.Services.Configure<PasswordHasherOptions>(options =>
{     options.IterationCount = 310000; });
```

When the PasswordHasher processes a password for storage, it uses a salt - a cryptographically generated random value that is added to the password before it is hashed.

This prevents passwords from being cracked through the use of Rainbow Tables - collections of hashes of common passwords. The salt is stored along with the hashed password, the number of iterations that were applied to the hashing function and the hashing function that was used (SHA256 by default).

When a user logs in, the salt and the iteration count are retrieved and then used to hash the submitted password. The result is compared with the stored hash to validate the user.

If the user is validated, and the iteration count configured via PasswordHasherOptions is higher than the iteration count stored with the password, the password is rehashed using the higher iteration count and updated in the database.

This enables you to increase the iteration count over time to keep ahead of technical advances. This operation is not reversible. If you reduce the iteration count, all passwords saved with the higher iteration account will continue to be subject to that higher count when they are verified.

Resource extension. Custom authorization requirements and handlers

As you can imagine everything is getting pretty complicated by now on the Program.cs page.

Lets take our policies and move them out to their own classes. This will keep our program.cs smaller and easier to read.

Authorization requirement classes implement the IAuthorizationRequirement interface.

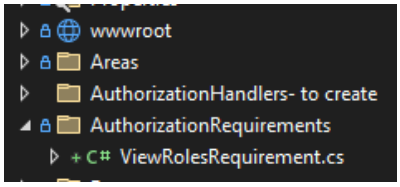
Handlers are represented by the IAuthorizationHandler interface which defines the HandleAsync method that takes an AuthorizationHandlerContext object as a parameter and returns a Task.

Classes implementing this interface are able to make a decision if authorization is allowed.

The logic for handling the requirement is placed in this method.

Requirements can have multiple handlers, but where there is a one to one relationship between the requirement and a handler, it is common to see the code for both placed in the same class which implements both interfaces.

The following example illustrates how to migrate the policy we created earlier into such a class, which takes a parameter representing the number of months.



Note: Its **AuthorizationRequirements as in plural, because you can add in more than one. Not AuthorizationRequirement, as the manual might accidently show.**

First rename the Policy in Program.cs to OLDViewRolesPolicy. We could delete it all out, but you should keep a copy to see how we are progressing.

```
//all of this gets replaced by the Authorization Handlers later on in the course
options.AddPolicy(name: "OLDViewRolesPolicy", configurePolicy: policyBuilder => policy
```

Add a new folder to the project with the name **AuthorizationRequirements**, and then add a new class file named **ViewRolesRequirement**s

Add the following code

How many months they have to be employed

Get users joining date

Convert joining date to date/time

Does user have permission to view roles and worked longer than 6 months

```
//This class implements both the IAuthorizationRequirement and IAuthorizationHandler interfaces
1 reference
public class ViewRolesRequirement : IAuthorizationRequirement, IAuthorizationHandler
{
    2 references
    public int Months { get; }

    //The constructor takes an int as a parameter and ensures that it is not a positive number
    0 references
    public ViewRolesRequirement(int months)
    {
        Months = months > 0 ? 0 : months;
    }

    //The HandleAsync method is implemented as required by the IAuthorizationHandler interface
    0 references
    public Task HandleAsync(AuthorizationHandlerContext context)
    {
        //The user is checked to see if they have a Joining Date claim. If not, the handler is exited
        var joiningDateClaim = context.User.FindFirst(match: c => c.Type == "Joining Date")?.Value;
        if (joiningDateClaim == null)
        {
            return Task.CompletedTask;
        }
        //The joining date is assessed to see if it exists and if its value is older than the age passed in.
        var joiningDate = Convert.ToDateTime(value: joiningDateClaim);

        if (context.User.HasClaim(type: "Permission", value: "View Roles") && joiningDate > DateTime.MinValue &&
            joiningDate < DateTime.Now.AddMonths(months: Months))

            context.Succeed(requirement: this);
        }

        //If the requirement is not satisfied, Task.CompletedTask is returned to satisfy the HandleAsync method signature
        return Task.CompletedTask;
    }
```

```
//This class implements both the IAuthorizationRequirement and IAuthorizationHandler
interfaces
    public class ViewRolesRequirement : IAuthorizationRequirement, IAuthorizationHandler
    {
        public int Months { get; }

        //The constructor takes an int as a parameter and ensures that it is not a
positive number
        public ViewRolesRequirement(int months)
        {
            Months = months > 0 ? 0 : months;
        }

        //The HandleAsync method is implemented as required by the IAuthorizationHandler
interface
        public Task HandleAsync(AuthorizationHandlerContext context)
        {
            //The user is checked to see if they have a Joining Date claim. If not, the
handler is exited
            var joiningDateClaim = context.User.FindFirst(c => c.Type == "Joining
Date")?.Value;
            if (joiningDateClaim == null)
            {
                return Task.CompletedTask;
            }
            //The joining date is assessed to see if it exists and if its value is older
than the age passed in.
            var joiningDate = Convert.ToDateTime(joiningDateClaim);
```

```
            if (context.User.HasClaim("Permission", "View Roles") && joiningDate >
DateTime.MinValue && joiningDate < DateTime.Now.AddMonths(Months))
            {
                context.Succeed(this);
            }

            //If the requirement is not satisfied, Task.CompletedTask is returned to
satisfy the HandleAsync method signature
            return Task.CompletedTask;
        }
    }
```

The requirement is satisfied if it is marked as being successfully evaluated.

This is achieved by calling the Succeed method of the AuthorizationHandlerContext class.

This class also offers a Fail method, which you can call to ensure that authorization is not successful. You would use this method if your handler allows all users except those that meet the specified condition, for example.

Of course we need to declare this new class in the Program.cs, as we will for the next two as well, when we make them.

```
builder.Services.AddSingleton<IAuthorizationHandler, ViewRolesHandler>();
```

```
builder.Services.AddSingleton<IAuthorizationHandler, ViewRolesHandler>();
```

You use the PolicyBuilder to register the policy, **passing in a suitable value** for the months parameter

```
options.AddPolicy(name: "ViewRolesPolicy", configurePolicy: policyBuilder =>
  policyBuilder.AddRequirements(requirements: new ViewRolesRequirement(months: -6)));
```

```
options.AddPolicy("ViewRolesPolicy", policyBuilder => policyBuilder.AddRequirements(new
ViewRolesRequirement(months: -6)));
```

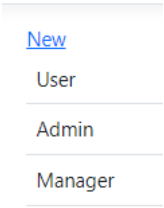
```
Program.cs                                                    HasClaimHandler.cs
RolesForAssessment
61  //all of this gets replaced by the Authorization Handlers later on in the course
62  options.AddPolicy(name: "OLDViewRolesPolicy", configurePolicy: policyBuilder =>
      policyBuilder.RequireAssertion(handler: context =>
63  {
64      //if they have a claim of type "Joining Date" and the value is less than 6 months ago, and they
          have Permission and View Roles they can view roles
65      // We use the FindFirst method to access a claim and obtain its value(if there is one) and
          convert it to a DateTime
66      var joiningDateClaim = context.User.FindFirst(match: c => c.Type == "Joining Date")?.Value;
67      var joiningDate = Convert.ToDateTime(value: joiningDateClaim);
68
69
70      return context.User.HasClaim(type: "Permission", value: "View Roles") && joiningDate >
          DateTime.MinValue && joiningDate < DateTime.Now.AddMonths(months: -6);
71
72  }));
73
74  //this policy only allows people who can view roles and have been employed longer than 6 months and
        comes from the ViewRolesRequirement Class
75  options.AddPolicy(name: "ViewRolesPolicy", configurePolicy: policyBuilder =>
      policyBuilder.AddRequirements(requirements: new ViewRolesRequirement(months: -6)));
76
```

Now we have moved the policy to its own class, we still have to apply the policy to the folder

```
//Having configured the policy we can apply it to the AuthorizeFolder method to ensure that only
  members of the Admin role can access the content:
builder.Services.AddRazorPages(configure: options =>
{
    options.Conventions.AuthorizeFolder(folderPath: "/RolesManager", policy: "ViewRolesPolicy");
});
```

```
builder.Services.AddRazorPages(options =>
{
    options.Conventions.AuthorizeFolder("/RolesManager", "ViewRolesPolicy");
});
```

So ccc can view the roles folder



ccc@ccc.com

Type	Value	Issuer
Permission	View Roles	LOCAL AUTHORITY
Joining Date	2021-1-12	LOCAL AUTHORITY

ZZZ can't because he hasn't been there for 6 months and doesn't have the View Roles permission

zzz@zzz.com

Type	Value	Issuer
Coffee Type	Latte	LOCAL AUTHORITY

DDD can't view either because he hasn't been employed long enough

ddd@ddd.com

Type	Value	Issuer
Joining Date	2022-1-12	LOCAL AUTHORITY
Permission	View Roles	LOCAL AUTHORITY
Coffee Type	Long Black	LOCAL AUTHORITY

Making it more easy to Maintain. – Create separate Handlers

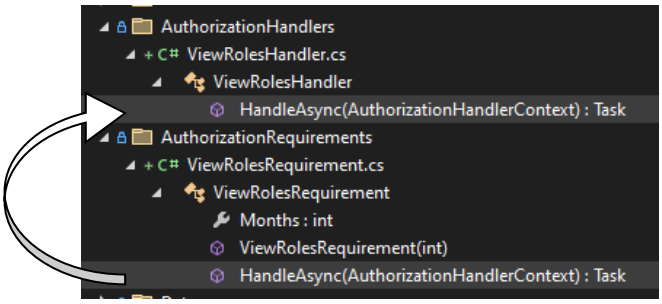
Building a combined requirement and handler is fine for simple use cases, but more often, you may want to create your handler as a separate class to enable reuse.

When you do this, there are a couple of changes you need to make to your general approach.

The existing requirement itself is trimmed down to just an implementation of IAuthorizationRequirements. We are taking the HandlerAsync method out to its own class

```
7 references
public class ViewRolesRequirement : IAuthorizationRequirement
{
    3 references
    public int Months { get; }

    //The constructor takes an int as a parameter and ensures that it is NOT a positive number
    1 reference
    public ViewRolesRequirement(int months)
    {
        Months = months > 0 ? 0 : months;
    }

};
```

1. Create a new Folder **AuthorizationHandlers**
2. Create a new class **ViewRolesHandler**
3. Inside of that class we are going to make a close copy of the public Task HandleAsync(AuthorizationHandlerContext context) Method



The ViewRolesHandler Class

```
public class ViewRolesHandler : IAuthorizationHandler
{
    0 references
    public Task HandleAsync(AuthorizationHandlerContext context)
    {
        // The PendingRequirements of the current user return all unsatisfied requirements. They need to be passed to a list so
          that we can execute operations against them.
        foreach (var requirement in context.PendingRequirements.ToList())
        {
            //We use pattern matching to identify ViewRolesRequirements in the original class and assign them to a local variable
              req
            if (requirement is ViewRolesRequirement req)
            {
                var joiningDateClaim = context.User.FindFirst(match: c => c.Type == "Joining Date")?.Value;
                if (joiningDateClaim == null) //no joining date
                {
                    return Task.CompletedTask;
                }
                var joiningDate = Convert.ToDateTime(value: joiningDateClaim); //convert it to a datetime
                if (context.User.HasClaim(type: "Permission", value: "View Roles") && joiningDate < DateTime.Now.AddMonths(months:
                  req.Months)) //if the date is greater than 6 months and they have the claim to View Roles then return suceed for
                  that reqirement
                {
                    context.Succeed(requirement);
                }
            }
        }
        return Task.CompletedTask;
    }
}
```

```
public Task HandleAsync(AuthorizationHandlerContext context)
        {
```

```
// The PendingRequirements of the current user return all unsatisfied requirements. They
need to be passed to a list so that we can execute operations against them.
            foreach (var requirement in context.PendingRequirements.ToList())
            {
                //We use pattern matching to identify ViewRolesRequirements in the
original class and assign them to a local variable  req
                if (requirement is ViewRolesRequirement req)
                {
                    var joiningDateClaim = context.User.FindFirst(c => c.Type == "Joining
Date")?.Value;
                    if (joiningDateClaim == null) //no joining date
                    {
                        return Task.CompletedTask;
                    }
var joiningDate = Convert.ToDateTime(joiningDateClaim); //convert it to a datetime
                    if (context.User.HasClaim("Permission", "View Roles") && joiningDate
< DateTime.Now.AddMonths(req.Months)) //if the date is greater than 6 months and they
have the claim to View Roles then return suceed for that reqirement
                    {
                        context.Succeed(requirement);
                    }
                }
            }
            return Task.CompletedTask;
        }
```

We also need to register the handler with the service container as an implementation of IAuthorizationHandler:

```
builder.Services.AddSingleton<IAuthorizationHandler, HasClaimHandler>();
builder.Services.AddSingleton<IAuthorizationHandler, ViewRolesHandler>();
```

Once you have done this, the authorization policy works in the same way as the previous approach that employed the combined requirement/handler combo.

The Code will automatically combine the two classes we made, from the single class at runtime because the

```
HandleRequirementAsync(AuthorizationHandlerContext context, ViewRolesRequirement req)
```

Calls in the ViewRolesRequirement

Using Multiple Requirements

As we mentioned before, requirements can have multiple handlers.

This is usually the case when there are alternative ways to satisfy the requirement.

Let's suppose that users can view roles if they have the "Permission" claim with the value "View Roles" and have been with the business for at least six months, or they're in the Admin role.

Now we have two alternative ways to authorize the user.

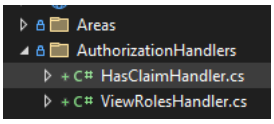
We could add more code to the existing handler to check whether the user is in the specified role, but there is a possibility of that code growing into a mess as more alternatives arise over time. Instead, we will implement an additional handler for our ViewRolesRequirement.

This time, however, we will take an alternative approach to crafting the handler that types it specifically to the requirement and therefore negates the need to filter all pending requirements.

We will derive from the abstract AuthenticationHandler<TRequirement> class, where TRequirement represents the requirement type that the handler is designed for.

The handler logic is placed in an overridden HandleRequirementAsync method which, in addition to an AuthorizationHandlerContext, takes the requirement type as a parameter.

Create a new class HasClaimHandler and add in the following class that checks if the user can view the roles because they have been there longer than 6 months.




```
public class HasClaimHandler : AuthorizationHandler<ViewRolesRequirement>
{
    0 references
    protected override Task HandleRequirementAsync(AuthorizationHandlerContext context, ViewRolesRequirement req)
    {
        //pass in the current user and look for their joining date value
        var joiningDateClaim = context.User.FindFirst(match: c => c.Type == "Joining Date")?.Value;
        if (joiningDateClaim == null) //if there isn't one then return
        {
            return Task.CompletedTask;
        }
        // if there is async joining date then check the date and see if it is less than 6 months and if the person has the
          persmission to see roles
        var joiningDate = Convert.ToDateTime(value: joiningDateClaim);

        if (context.User.HasClaim(type: "Permission", value: "View Roles") &&
            joiningDate < DateTime.Now.AddMonths(months: req.Months))
        {
            context.Succeed(requirement: req); //they have the permissions
        }
        return Task.CompletedTask;
    }
}
```

Context.Succeed is called to mark the specified requirement as being successfully evaluated.

```
public class HasClaimHandler : AuthorizationHandler<ViewRolesRequirement>
    {
        protected override Task HandleRequirementAsync(AuthorizationHandlerContext
context, ViewRolesRequirement req)
        {
            //pass in the current user and look for their joining date value
var joiningDateClaim = context.User.FindFirst(c => c.Type == "Joining Date")?.Value;
            if (joiningDateClaim == null) //if there isn't one then return
            {
                return Task.CompletedTask;
            }
            // if there is async joining date then check the date and see if it is less
than 6 months and if the person has the persmission to see roles
            var joiningDate = Convert.ToDateTime(joiningDateClaim);

            if (context.User.HasClaim("Permission", "View Roles") && joiningDate <
DateTime.Now.AddMonths(req.Months)) //if the date is greater than 6 months and they have
the claim to View Roles then return suceed for that reqirement
            {
                context.Succeed(req); //they have the permissions
            }
            return Task.CompletedTask;
        }
    }
```

We can see this running when the user who has worked longer than 6 months (ccc) tries to go to the RoleManager index page.

User details are passed in the `AuthorizationHandlerContext context` field.

Drilling down into the context under User shows the identities

```
[0]   {http://schemas.xmlsoap.org/ws/2005/05/identity/claims/nameidentifier: 8f536075-01f3-4860-8906-2bdeb02b4006}
[1]   {http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name: ccc@ccc.com}
[2]   {http://schemas.xmlsoap.org/ws/2005/05/identity/claims/emailaddress: ccc@ccc.com}
[3]   {AspNet.Identity.SecurityStamp: DQTWACIB75VHRBUGD63CCUUESUY7O6JA}
[4]   {Permission:  View Roles}
[5]   {Joining Date: 2021-1-12}
[6]   {http://schemas.microsoft.com/ws/2008/06/identity/claims/role: User}
[7]   {amr: pwd}
```

And Pending Requirements are shown as -6 months

```
PendingRequirements   View   Count = 1
[0]   {RolesForAssessment.AuthorizationRequirements.ViewRolesRequirement}
Months   -6
```

ViewRolesRequirement also shows this

```
text context, ViewRolesRequirement req)
req   {RolesForAssessment.AuthorizationRequirements.ViewRolesRequirement}
Months   -6
```

The User has an identity of

```
[4]   {Permission:  View Roles}
[5]   {Joining Date: 2021-1-12}
```

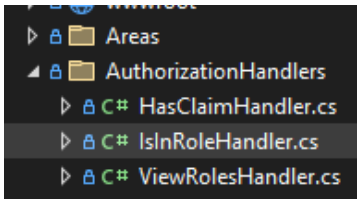
As we step into the code it shows us the joiningDate

```
var joiningDateClaim = context.User.FindFirs
if (joiningDateCla   joiningDateClaim   View   "2021-1-12"
```

Converted to DateTime

```
var joiningDate = Convert.ToDateTime
joiningDate    {2021/01/12 12:00:00 am}
```

Create a final third handler IsInRoleHandler




```
0 references
public class IsInRoleHandler : AuthorizationHandler<ViewRolesRequirement>
{
    0 references
    protected override Task HandleRequirementAsync(AuthorizationHandlerContext context,
        ViewRolesRequirement req)
    {
        if (context.User.IsInRole(role: "Admin")) //does the user have the role Admin?
        {
            context.Succeed(requirement: req);
        }
        return Task.CompletedTask;
    }
}
```

```
public class IsInRoleHandler : AuthorizationHandler<ViewRolesRequirement>
    {
        protected override Task HandleRequirementAsync(AuthorizationHandlerContext
context,
            ViewRolesRequirement req)
        {
            if (context.User.IsInRole("Admin")) //does the user have the role Admin?
            {
                context.Succeed(req);
            }
            return Task.CompletedTask;
        }
    }
```

All 3 handlers need to the registered with the service container in program.cs:

```
builder.Services.AddSingleton<IAuthorizationHandler, IsInRoleHandler>();
builder.Services.AddSingleton<IAuthorizationHandler, HasClaimHandler>();
builder.Services.AddSingleton<IAuthorizationHandler, ViewRolesHandler>();
```

```
...
// All 3 handlers need to the registered with the service container in program.cs:
builder.Services.AddSingleton<IAuthorizationHandler, IsInRoleHandler>();
builder.Services.AddSingleton<IAuthorizationHandler, HasClaimHandler>();
builder.Services.AddSingleton<IAuthorizationHandler, ViewRolesHandler>();

...
...
builder.Services.AddRazorPages();
```

Resource: Modify Users at database level

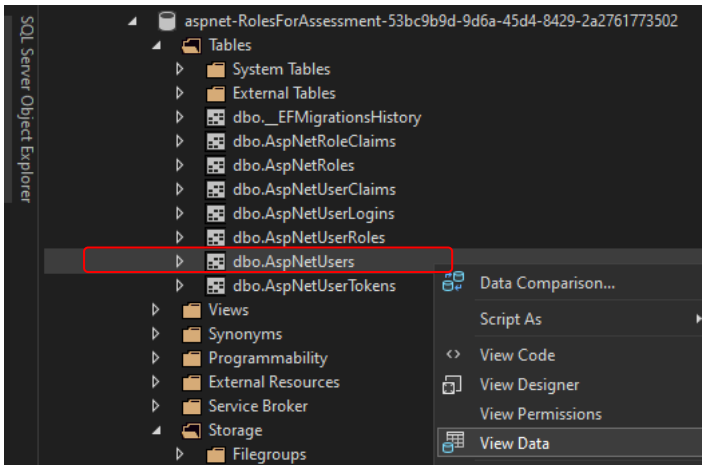
Modifying the existing user database

Find your DB name in the appsettings.json. Mine is below

```
"ConnectionStrings": {
  "DefaultConnection": "Server=(localdb)\\mssqllocaldb;Database=aspnet-RolesForAssessment-53bc9b9d-9d6a-45d4-8429-2a2761773502;
   Trusted_Connection=True;MultipleActiveResultSets=true"
},
```

Go to your database in the SQLServer Object Explorer Tab in Visual Studio.

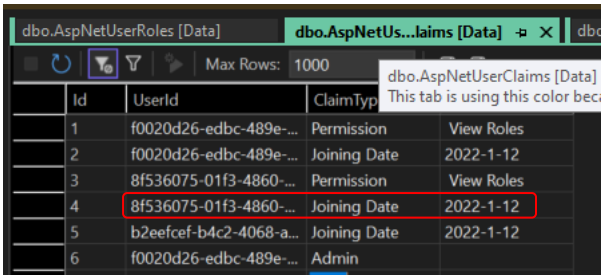
Right click on the ASPNetUsers and choose View Data



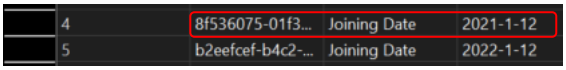
So my user CCC has an ID starting with 8f5

Id	UserName
8f536075-01f3-4860-8906...	ccc@ccc.com
b2eefcef-b4c2-4068-ab5f-...	bbb@bbb.com
f0020d26-edbc-489e-b24...	aaa@aaa.com
NULL	NULL

If I then view data in **AspNetUserClaims** I see the row that holds the users Joining date



I can click in and change that field to be older than 6 months.

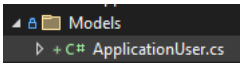


Close the table, then reopen it to check that the change has been saved. Sometimes you need to tab to the end of the line before it saves.

Resources Add new fields to the ASPNetUsers table

https://learn.microsoft.com/en-us/aspnet/core/security/authentication/customize-identity-model?view=aspnetcore-7.0

Create a new Class called ApplicationUser



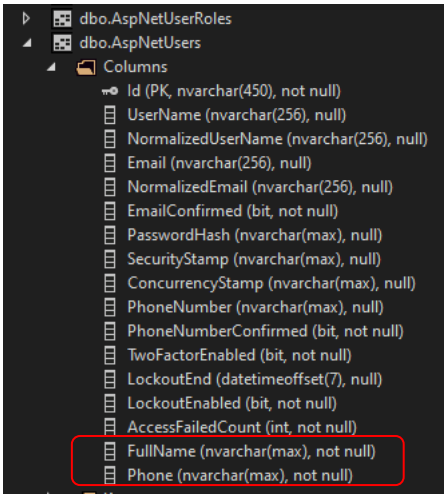
Add into that class the fields you want to add to your User. Then inherit IdentityUser

```
using Microsoft.AspNetCore.Identity;

namespace RolesForAssessment.Models
{
    1 reference
    public class ApplicationUser : IdentityUser
    {
        0 references
        public string FullName { get; set; }
        0 references
        public string Phone { get; set; }

    }
}
```

Create a migration to update the database table

```
PM> add-migration extendUser
```

```
PM> update-database extendUser
```



TaDa!

In your _LoginPartial, change IdentityUser to ApplicationUser. Also include the path to the class.

```
_LoginPartial.cshtml
1  @using Microsoft.AspNetCore.Identity
2  @using RolesForAssessment.Models   //path to ApplicationUser
3  @inject SignInManager<ApplicationUser> SignInManager
4  @inject UserManager<ApplicationUser> UserManager
5
```

In the Program.cs

Change IdentityUser to ApplicationUser

```
//the default identity of the user
builder.Services.AddDefaultIdentity<ApplicationUser>(configureOptions: options =>
  options.SignIn.RequireConfirmedAccount = true)
    .AddRoles<ApplicationUser>()
    .AddEntityFrameworkStores<ApplicationDbContext>();
```