Tutorial for Class number 8

This exercise intends develop a web application to test the use of **ASP.NET Core security.**

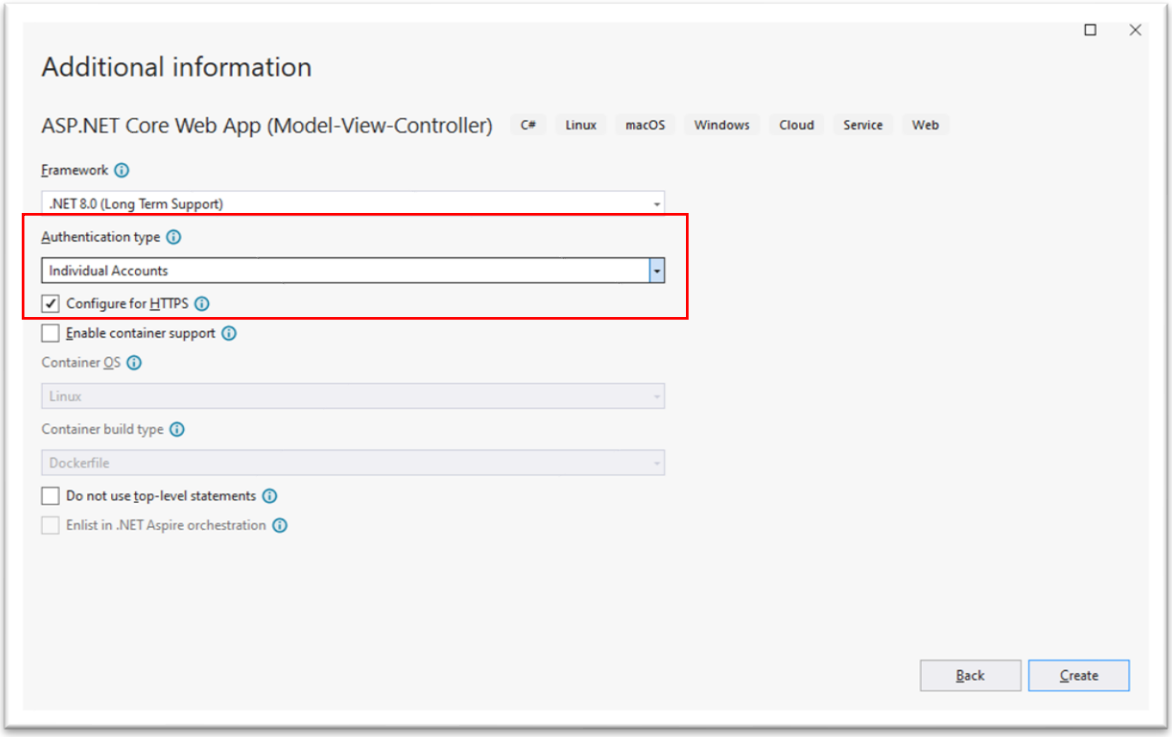
ASP.NET Core provides many tools and libraries to secure your apps including built-in identity providers, but you can use third-party identity services such as Facebook, Twitter, and LinkedIn. With ASP.NET Core, you can easily manage app secrets, which are a way to store and use confidential information without having to expose it in the code.

Authentication vs. Authorization

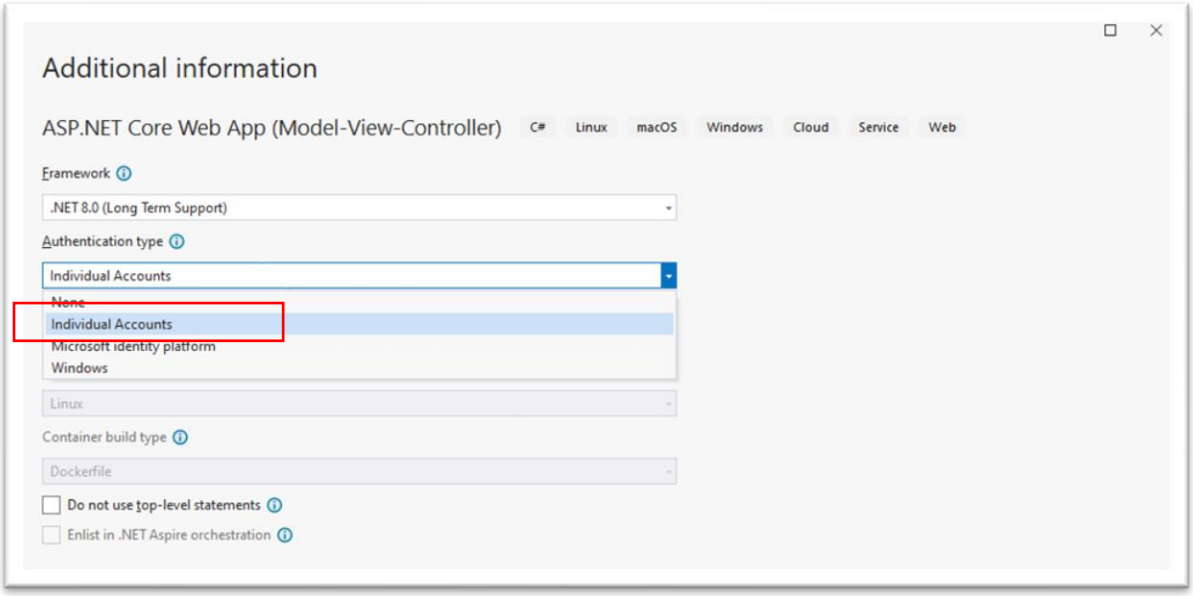
Authentication is a process in which a user provides credentials that are then compared to those stored in an operating system, database, app or resource. If they match, users authenticate successfully, and can then perform actions that they're authorized for, during an authorization process. The **authorization** refers to the process that determines what a user is allowed to do.

First step – Creating a project with authentication

- Create a new “ASP.NET Core Web App (Model-View-Controller)” project.



In Authentication Type information, select the **Individual Accounts** option. In this configuration, the user information is stored into a local database.



- See the generated or modified files:
  - **appsettings.json** with a new connection string. We can change where to save the user database.

```
"ConnectionStrings": {
  "DefaultConnection": "Server=(localdb)\\mssqllocaldb;Database=aspnet-Class08-6e503136-ca9e-4c
},
```

  - in the **Data/Migrations** folder, was created one migration (**CreateIdentitySchema**) with the tables definition.

```
mespace Class08.Data.Migrations

  public partial class CreateIdentitySchema : Migration
  {
      protected override void Up(MigrationBuilder migrationBuilder)
      {
          migrationBuilder.CreateTable(
              name: "AspNetRoles",
              columns: table => new
              {
                  Id = table.Column<string>(nullable: false),
                  Name = table.Column<string>(maxLength: 256, nullable: true),
                  NormalizedName = table.Column<string>(maxLength: 256, nullable: true),
                  ConcurrencyStamp = table.Column<string>(nullable: true)
              },
              constraints: table =>
              {
                  table.PrimaryKey("PK_AspNetRoles", x => x.Id);
              });
```

  - and a database context:

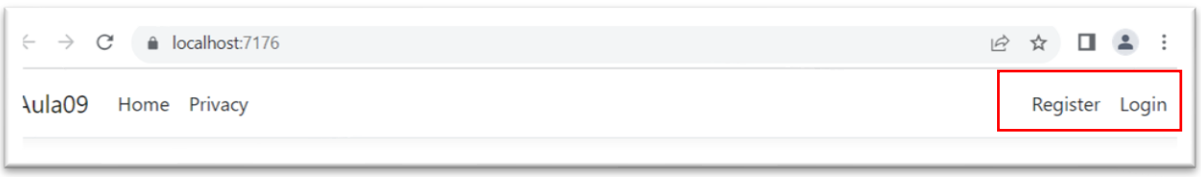
```
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore;

namespace Class08.Data
{
    public class ApplicationDbContext : IdentityDbContext
    {
        public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
            : base(options)
        {
        }
    }
}
```

- in the **Program.cs**

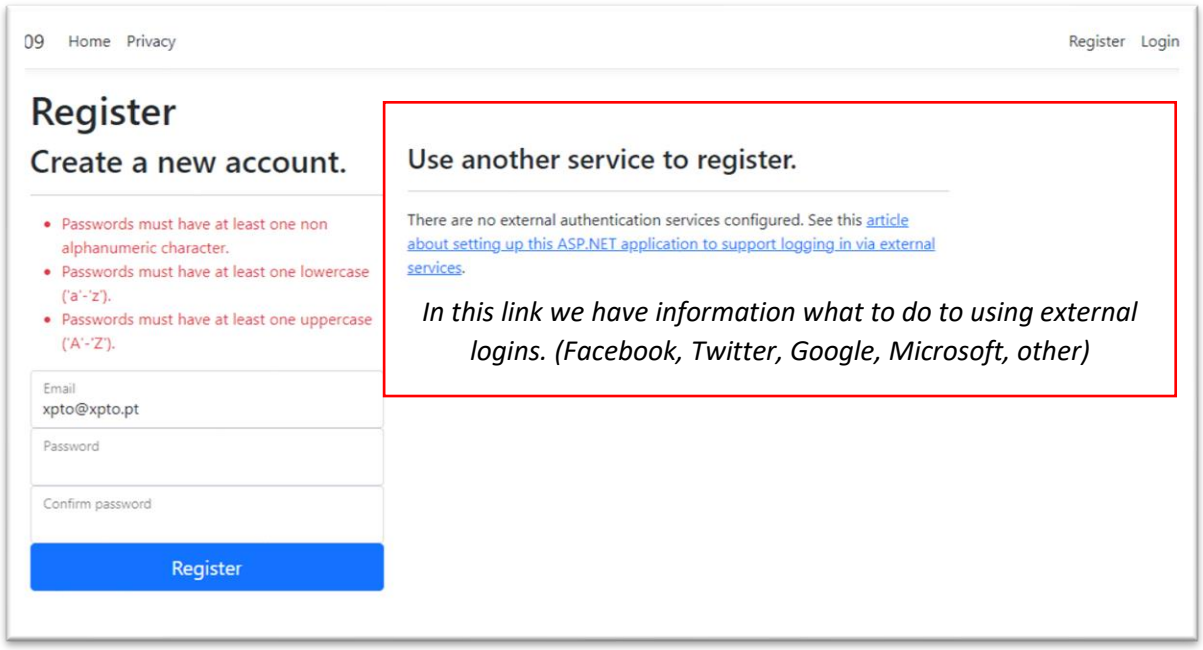
```
// Add services to the container.
var connectionString = builder.Configuration.GetConnectionString("DefaultConnection") ?? throw new InvalidOper
builder.Services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(connectionString));
builder.Services.AddDatabaseDeveloperPageExceptionFilter();

builder.Services.AddDefaultIdentity<IdentityUser>(options => options.SignIn.RequireConfirmedAccount = true)
    .AddEntityFrameworkStores<ApplicationDbContext>();

builder.Services.AddControllersWithViews();

var app = builder.Build();

// Configure the HTTP request pipeline.
if (app.Environment.IsDevelopment())...
else...

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");
```

- Apply the migrations to initialize the database. Run the following command in the Package Manager Console (PMC):
  PM> Update-Database

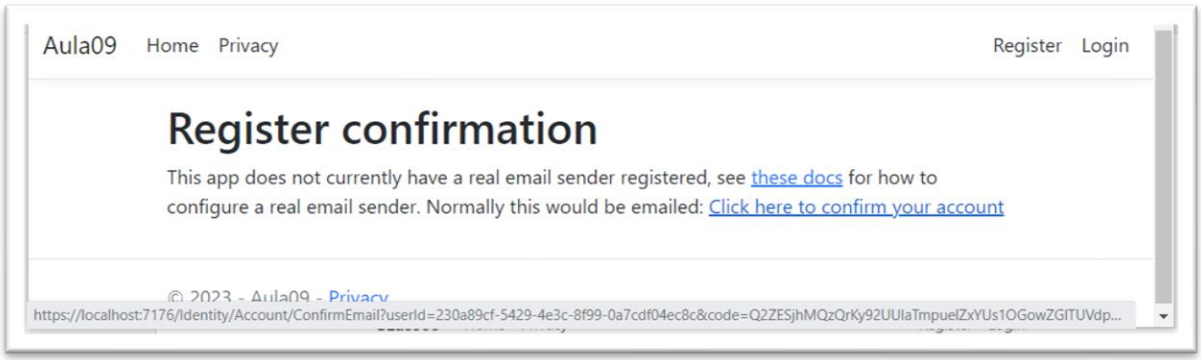
- Run the application
  In the application menu we have a **Register** and **Login** options.



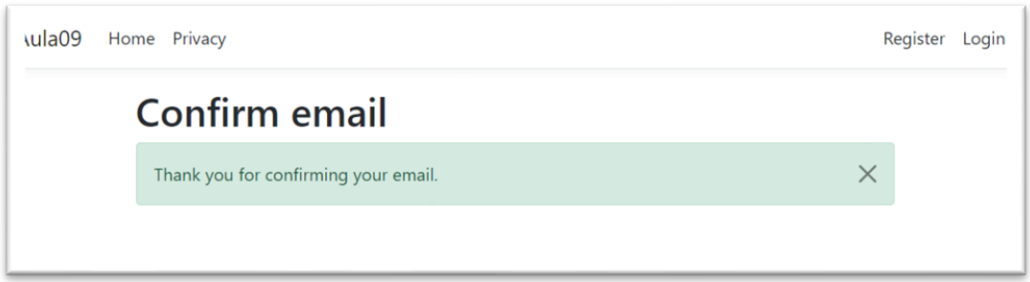
- Register a new user:



After the register, the system need a confirmation sending an email.



Because we don´t have a mail sender registered, we select “Click here to confirm your account” to simulate the email confirmation.



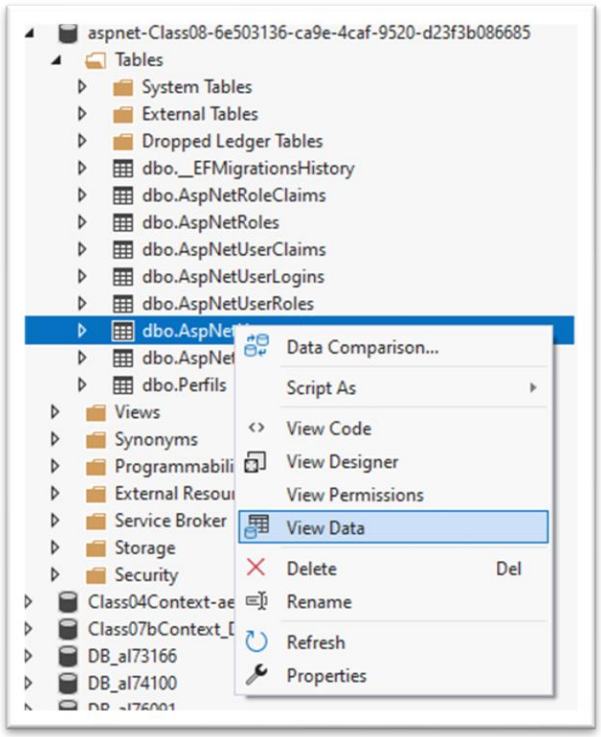
We can disable this requirement, changing in the **Program.cs**

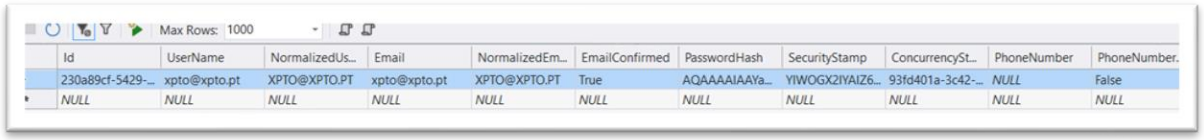
```
builder.Services.AddDatabaseDeveloperPageExceptionFilter();

builder.Services.AddDefaultIdentity<IdentityUser>(options => options.SignIn.RequireConfirmedAccount = false)
    .AddEntityFrameworkStores<ApplicationDbContext>();

builder.Services.AddControllersWithViews();
```

The next code configures **Identity** with default option values. We must change to our needs.

```
builder.Services.AddDefaultIdentity<IdentityUser>(options => options.SignIn.RequireConfirmedA
    .AddEntityFrameworkStores<ApplicationDbContext>();

builder.Services.Configure<IdentityOptions>(options =>
{
    //passsword default settings, we can change this to our needs
    options.Password.RequireDigit= true;
    options.Password.RequireLowercase= true;
    options.Password.RequireNonAlphanumeric = true;
    options.Password.RequireUppercase= true;
    options.Password.RequiredLength = 6;
    options.Password.RequiredUniqueChars= 1;

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

    // user settings
    options.User.AllowedUserNameCharacters =
    "abcdefghijklmnopqrstuvxzwyABCDEFGHIJKLMNOPQRSTUVXZWY0123456789-_@#";
    options.User.RequireUniqueEmail= true;
});

builder.Services.AddControllersWithViews();
```

See the data stored in the table **AspNetUsers**



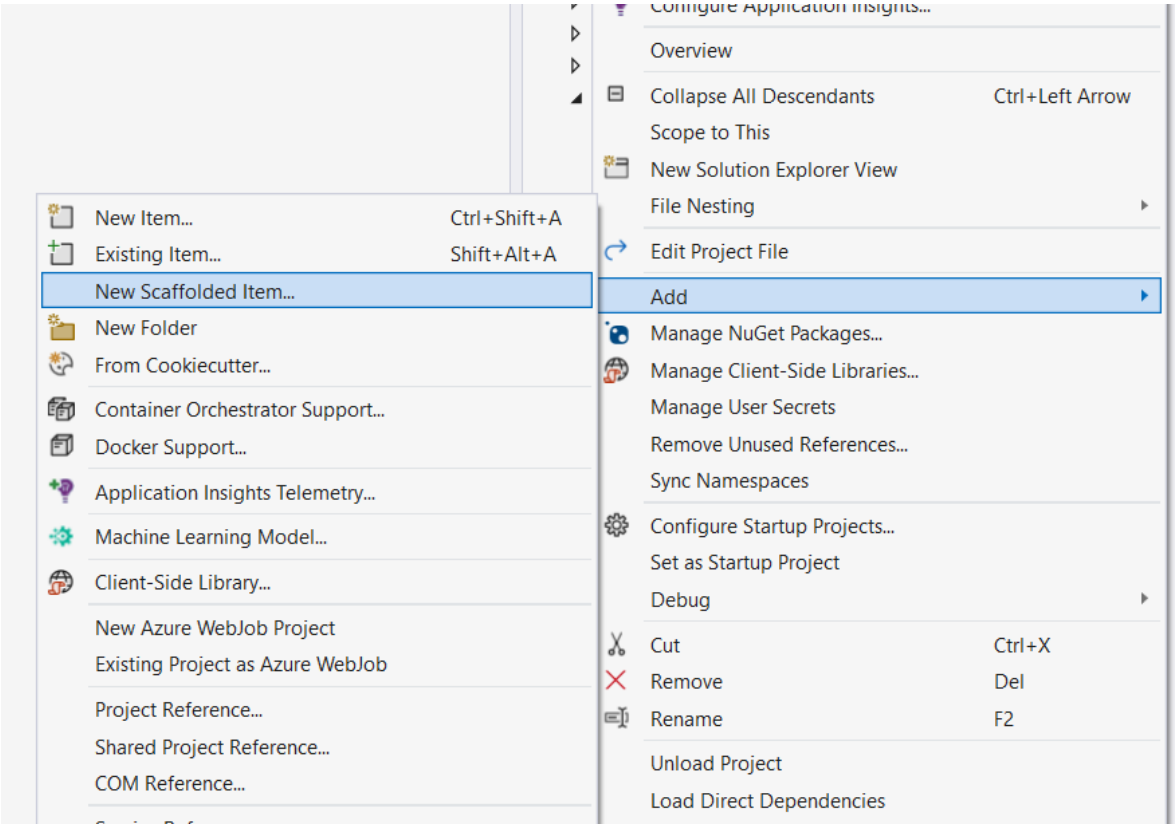


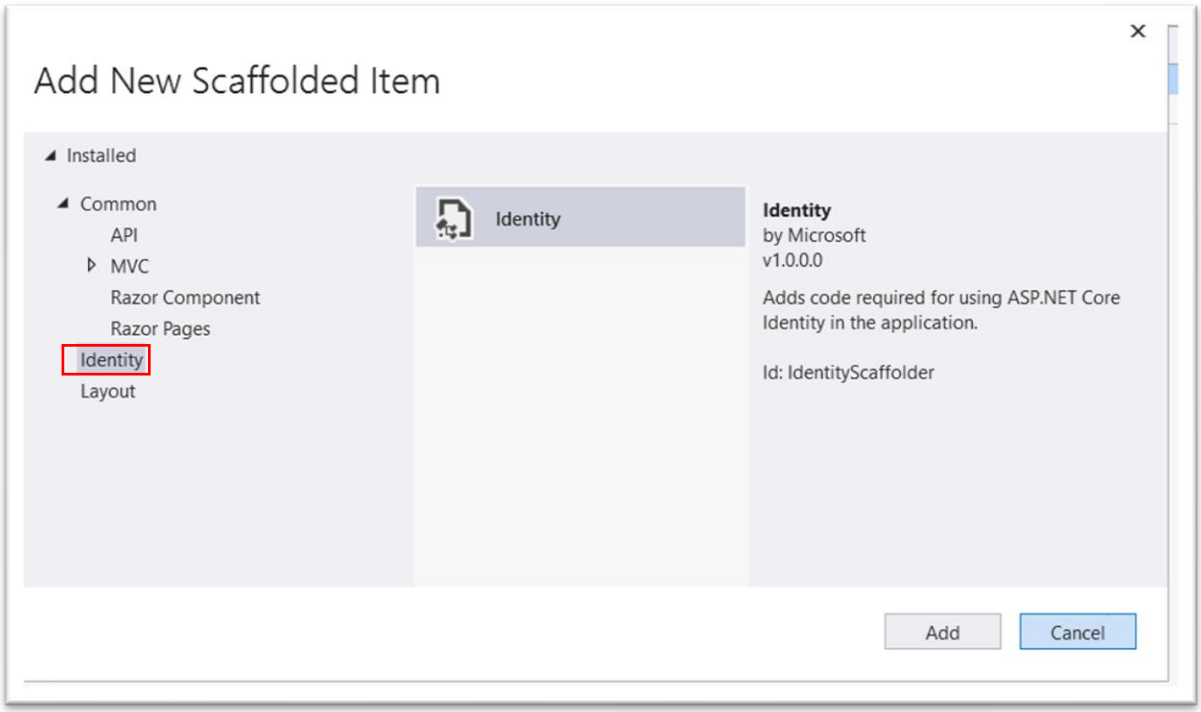
We can verify the data stored for each user. Note that Password is encrypted. By default, the UserName is the Email. In the next steps we will modify this.

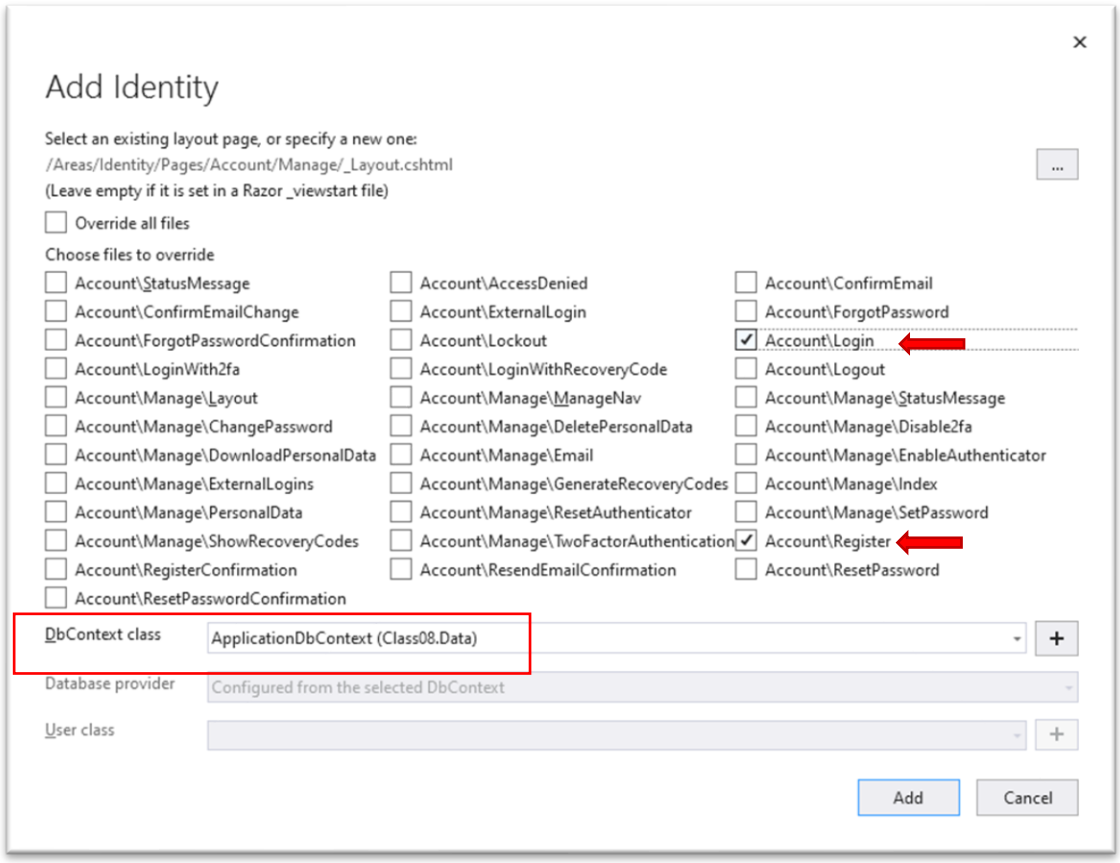
- Change authentication views.

These services used to register or login a user in system uses internal code. If we wish do modify then, we first have to pull this code to the project. To access the views and code of the actions associated with authentication, we need to generate that:

-Right click in project name (Class08) and choose **Add->New Scaffold Item…**

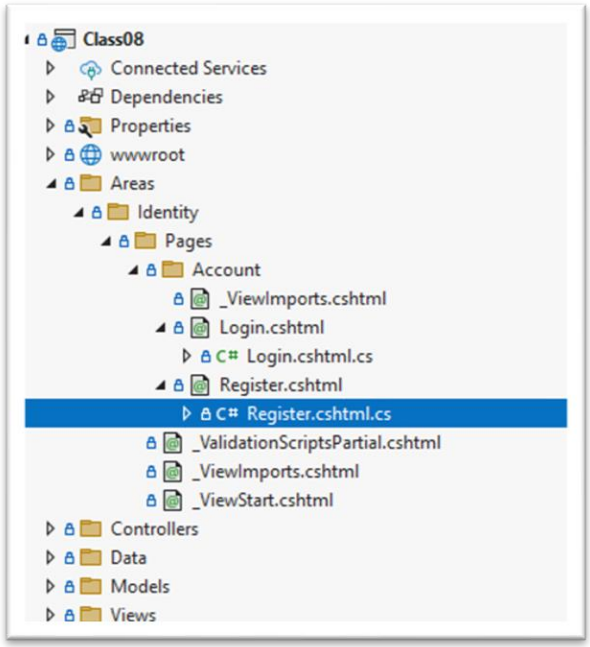

Configure Application Insights...
Overview
Collapse All Descendants Ctrl+Left Arrow
Scope to This
New Solution Explorer View
File Nesting
Edit Project File
Add
Manage NuGet Packages...
Manage Client-Side Libraries...
Manage User Secrets
Remove Unused References...
Sync Namespaces
Configure Startup Projects...
Set as Startup Project
Debug
Cut Ctrl+X
Remove Del
Rename F2
Unload Project
Load Direct Dependencies
New Item... Ctrl+Shift+A
Existing Item... Shift+Alt+A
New Scaffolded Item...
New Folder
From Cookiecutter...
Container Orchestrator Support...
Docker Support...
Application Insights Telemetry...
Machine Learning Model...
Client-Side Library...
New Azure WebJob Project
Existing Project as Azure WebJob
Project Reference...
Shared Project Reference...
COM Reference...


Add New Scaffolded Item
Installed
Common
API
MVC
Razor Component
Razor Pages
Identity
Layout
Identity
Identity
by Microsoft
v1.0.0.0
Adds code required for using ASP.NET Core Identity in the application.
Id: IdentityScaffolder
Add
Cancel



Select the views we want to change (Login, Register)

Expand de folder to see the created files:



These files use the **Blazor** format. The View and code are grouped in two files: **.cshtml** and **.cshtml.cs**

- Insert a Username field in the register process

Open the file **Register.cshtml.cs** and add **UserName** to **InputModel** declaration.

```
public class InputModel
{
    [Required]
    [Display(Name = "User Name")]
    0 references
    public string UserName { get; set; }

    [Required]
    [EmailAddress]
    [Display(Name = "Email")]
    7 references
```

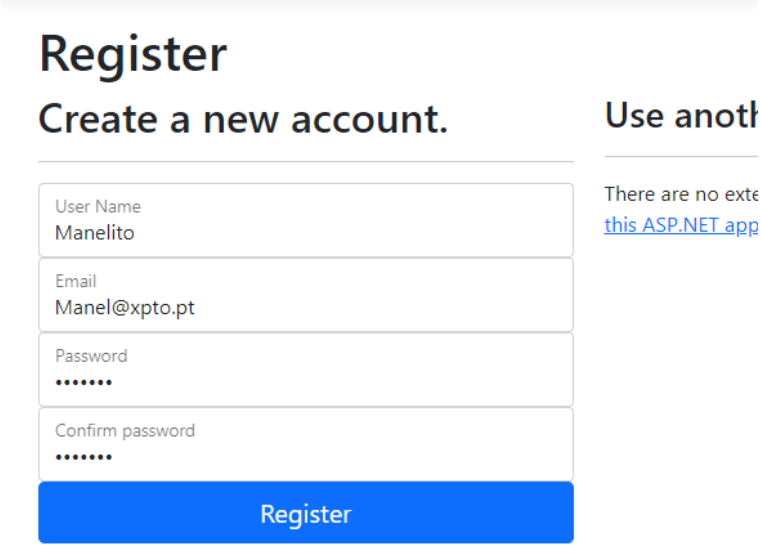
In same file, use this new information when creating an **IdentityUser** instance.

```
public async Task<IActionResult> OnPostAsync(string returnUrl = null)
{
    returnUrl ??= Url.Content("~/");
    ExternalLogins = (await _signInManager.GetExternalAuthenticationSchemesAsync()).ToList();
    if (ModelState.IsValid)
    {
        var user = CreateUser();

        await _userStore.SetUserNameAsync(user, Input.UserName, CancellationToken.None);
        await _emailStore.SetEmailAsync(user, Input.Email, CancellationToken.None);
        var result = await _userManager.CreateAsync(user, Input.Password);

        if (result.Succeeded)
        {
            _logger.LogInformation("User created a new account with password.");

            var userId = await _userManager.GetUserIdAsync(user);
            var code = await _userManager.GenerateEmailConfirmationTokenAsync(user);
```

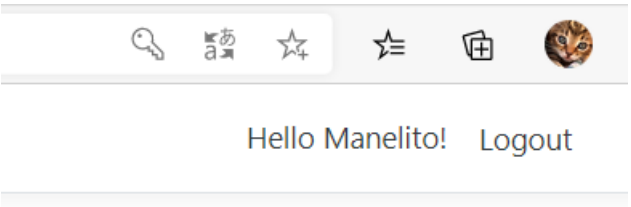
And finally, alter de view code to show a **UserName** field.

```
<div class="row">
    <div class="col-md-4">
        <form id="registerForm" asp-route-returnUrl="@Model.ReturnUrl" method="post">
            <h2>Create a new account.</h2>
            <hr />
            <div asp-validation-summary="ModelOnly" class="text-danger" role="alert"></div>
            <div class="form-floating mb-3">
                <input asp-for="Input.UserName" class="form-control" autocomplete="username" aria-required="true" placeholder="username" />
                <label asp-for="Input.UserName">UserName</label>
                <span asp-validation-for="Input.UserName" class="text-danger"></span>
            </div>
            <div class="form-floating mb-3">
                <input asp-for="Input.Email" class="form-control" autocomplete="username" aria-required="true" placeholder="name@example.com" />
                <label asp-for="Input.Email">Email</label>
                <span asp-validation-for="Input.Email" class="text-danger"></span>
            </div>
            <div class="form-floating mb-3">
                <input asp-for="Input.Password" class="form-control" autocomplete="new-password" aria-required="true" placeholder="password" />
                <label asp-for="Input.Password">Password</label>
```

Register a new user. After making these changes, we now have to insert the Username, Email and Password fields.



And the result is:



Now we also need modify the **Login** process to accept the **UserName**.

Open **Login.cshtml.cs** file and proceed with these modifications:

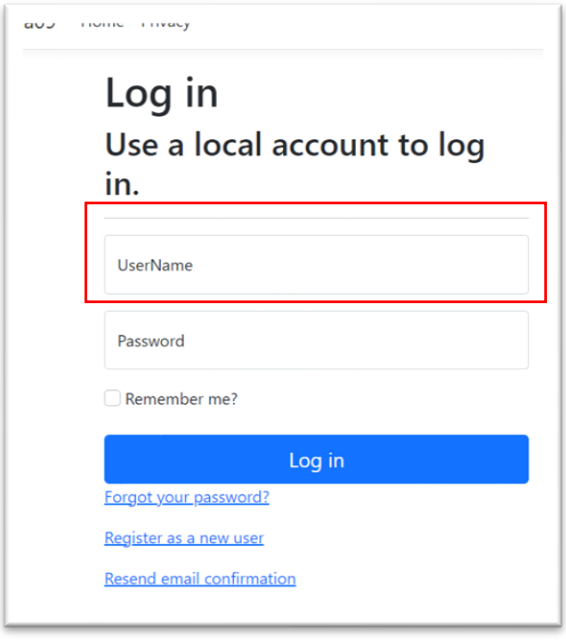
```
public class InputModel
{
    [Required]
    [Display(Name ="User Name")]
    4 references
    public string UserName { get; set; }

    [Required]
    [DataType(DataType.Password)]
    4 references
    public string Password { get; set; }


    [Display(Name = "Remember me?")]
    5 references
    public bool RememberMe { get; set; }
}
```

```
public async Task<IActionResult> OnPostAsync(string returnUrl = null)
{
    returnUrl ??= Url.Content("~/");

    ExternalLogins = (await _signInManager.GetExternalAuthenticationSchemesAsync()).ToList();

    if (ModelState.IsValid)
    {
        // This doesn't count login failures towards account lockout
        // To enable password failures to trigger account lockout, set lockoutOnFailure: true
        var result = await _signInManager.PasswordSignInAsync(Input.UserName, Input.Password, Input.RememberMe, lockoutOnFailure: false);
        if (result.Succeeded)
        {
            _logger.LogInformation("User logged in.");
            return LocalRedirect(returnUrl);
        }
        if (result.RequiresTwoFactor)
        {
            return RedirectToPage("./LoginWith2fa", new { ReturnUrl = returnUrl, RememberMe = Input.RememberMe });
```

Alter the View (Login.cshtml)

```
<h1>@ViewData["Title"]</h1>
<div class="row">
    <div class="col-md-4">
        <section>
            <form id="account" method="post">
                <h2>Use a local account to log in.</h2>
                <hr />
                <div asp-validation-summary="ModelOnly" class="text-danger" role="alert"></div>
                <div class="form-floating mb-3">
                    <input asp-for="Input.UserName" class="form-control" autocomplete="username" aria-required="true" placeholder="username" />
                    <label asp-for="Input.UserName" class="form-label">UserName</label>
                    <span asp-validation-for="Input.UserName" class="text-danger"></span>
                </div>
                <div class="form-floating mb-3">
                    <input asp-for="Input.Password" class="form-control" autocomplete="current-password" aria-required="true" placeholder="password"
                    <label asp-for="Input.Password" class="form-label">Password</label>
                    <span asp-validation-for="Input.Password" class="text-danger"></span>
                </div>
                <div class="checkbox mb-3">
                    <label asp-for="Input.RememberMe" class="form-label">
                        <input class="form-check-input" asp-for="Input.RememberMe" />
                        @Html.DisplayNameFor(m => m.Input.RememberMe)
                    </label>
                </div>
                <div>
```

- Test the new login



- See the _Layout.cshtml:

```
            <div class="navbar-collapse collapse d-sm-inline-flex justify-content-between">
                <ul class="navbar-nav flex-grow-1">
                    <li class="nav-item">
                        <a class="nav-link text-dark" asp-area="" asp-controller="Home" asp-action="Index">Home</a>
                    </li>
                    <li class="nav-item">
                        <a class="nav-link text-dark" asp-area="" asp-controller="Home" asp-action="Privacy">Privacy</a>
                    </li>
                </ul>
                <partial name="_LoginPartial" />
            </div>
        </div>
    </nav>
</header>
<div class="container">
    <main role="main" class="pb-3">
        @RenderBody()
    </main>
</div>
```

In here is inserted a partial view **_LoginPartial.cshtml**

```
@inject SignInManager<IdentityUser> SignInManager
@inject UserManager<IdentityUser> UserManager

<ul class="navbar-nav">
@if (SignInManager.IsSignedIn(User))
{
    <li class="nav-item">
        <a  class="nav-link text-dark" asp-area="Identity" asp-page="/Account/Manage/Index" title="Manage">Hello @User.Identity?.Name!</a>
    </li>
    <li class="nav-item">
        <form  class="form-inline" asp-area="Identity" asp-page="/Account/Logout" asp-route-returnUrl="@Url.Action("Index", "Home", new { area = "" })">
            <button  type="submit" class="nav-link btn btn-link text-dark">Logout</button>
        </form>
    </li>
}
else
{
    <li class="nav-item">
        <a class="nav-link text-dark" asp-area="Identity" asp-page="/Account/Register">Register</a>
    </li>
    <li class="nav-item">
        <a class="nav-link text-dark" asp-area="Identity" asp-page="/Account/Login">Login</a>
    </li>
}
</ul>
```

- Add other information to the User.
  If we need to store any other information of the user profile, we can create another model with the new data.

  - Create a new model “Perfil.cs” in **Model** folder.

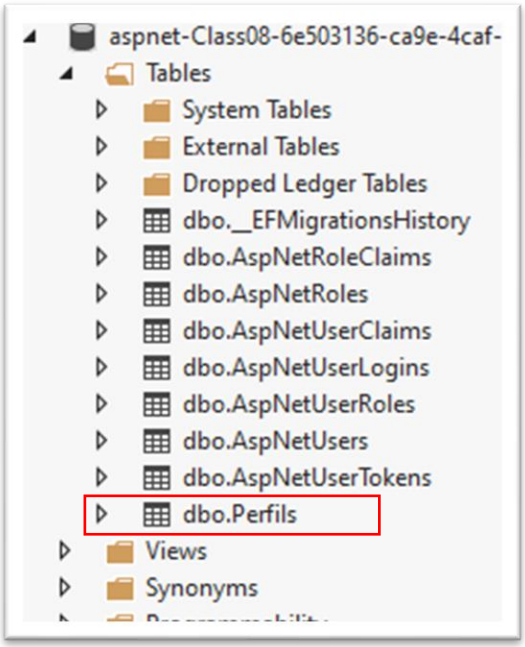
```
0 references
public class Perfil
{
    0 references
    public int Id { get; set; }

    [DataType(DataType.Date)]
    0 references
    public DateTime Birthday { get; set; }

    // we can put other atributes
    0 references
    public string? UserName { get; set; } //username of the User in AspNetUsers
}
```

-add a new DbSet to the **ApplicationDbContext** class.

```
namespace Class08.Data
{
    public class ApplicationDbContext : IdentityDbContext
    {
        public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
            : base(options)
        {
        }
        public DbSet<Perfil> Perfils { get; set; }
    }
}
```

- Add a new Migration
  PM> add-migration addPerfil
- Update the database
  PM> update-database

- Confirm the new table in the database



-alter the register page to accept the new information in the **InputModel**.

```
public class InputModel
{

    [Required]
    [Display(Name = "User Name")]
    4 references
    public string UserName { get; set; }

    [DataType(DataType.Date)]
    0 references
    public DateTime Birthday { get; set; }

    [Required]
    [EmailAddress]
    [Display(Name = "Email")]
    6 references
    public string Email { get; set; }
```

Inject the **ApplicationDbContext** in the code to access the tables of the database

```
public class RegisterModel : PageModel
{
    private readonly SignInManager<IdentityUser> _signInManager;
    private readonly UserManager<IdentityUser> _userManager;
    private readonly IUserStore<IdentityUser> _userStore;
    private readonly IUserEmailStore<IdentityUser> _emailStore;
    private readonly ILogger<RegisterModel> _logger;
    private readonly IEmailSender _emailSender;
    private readonly ApplicationDbContext _dbcontext;

    0 references
    public RegisterModel(
        UserManager<IdentityUser> userManager,
        IUserStore<IdentityUser> userStore,
        SignInManager<IdentityUser> signInManager,
        ILogger<RegisterModel> logger,
        IEmailSender emailSender,
        ApplicationDbContext dbContext)
    {
        _userManager = userManager;
        _userStore = userStore;
        _emailStore = GetEmailStore();
        _signInManager = signInManager;
        _logger = logger;
        _emailSender = emailSender;
        _dbcontext = dbContext;
    }
```

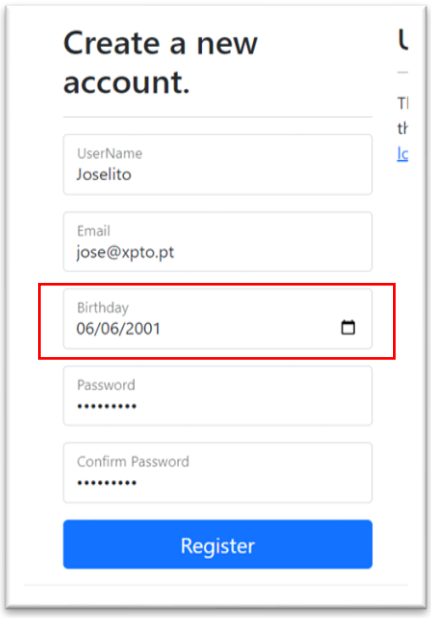
- If we success registering a new user, do an insert into table Perfils with the other information.

```
await _userStore.SetUserNameAsync(user, Input.UserName, CancellationToken.None);
await _emailStore.SetEmailAsync(user, Input.Email, CancellationToken.None);
var result = await _userManager.CreateAsync(user, Input.Password);

if (result.Succeeded)
{
    _logger.LogInformation("User created a new account with password.");

    Perfil newPerfil= new Perfil { UserName = user.UserName, Birthday = Input.Birthday };

    _dbcontext.Perfils.Add(newPerfil);
    _dbcontext.SaveChanges();

    var userId = await _userManager.GetUserIdAsync(user);
    var code = await _userManager.GenerateEmailConfirmationTokenAsync(user);
    code = WebEncoders.Base64UrlEncode(Encoding.UTF8.GetBytes(code));
    var callbackUrl = Url.Page(
```

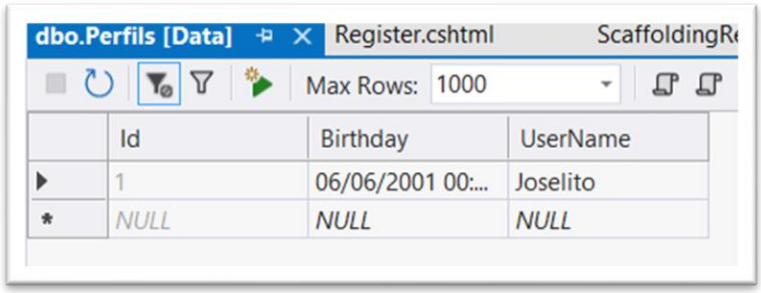
- Alter the view code to insert the new information:

```
<div class="form-floating mb-3">
    <input asp-for="Input.Email" class="form-control" autocomplete="username" aria-required="true" placeholder="name@example.com" />
    <label asp-for="Input.Email">Email</label>
    <span asp-validation-for="Input.Email" class="text-danger"></span>
</div>
<div class="form-floating mb-3">
    <input asp-for="Input.Birthday" class="form-control" autocomplete="birthday" aria-required="true" placeholder="birthday" />
    <label asp-for="Input.Birthday">Birthday</label>
    <span asp-validation-for="Input.Birthday" class="text-danger"></span>
</div>
```

-Register a new User:



- Confirm the data in the perfils table



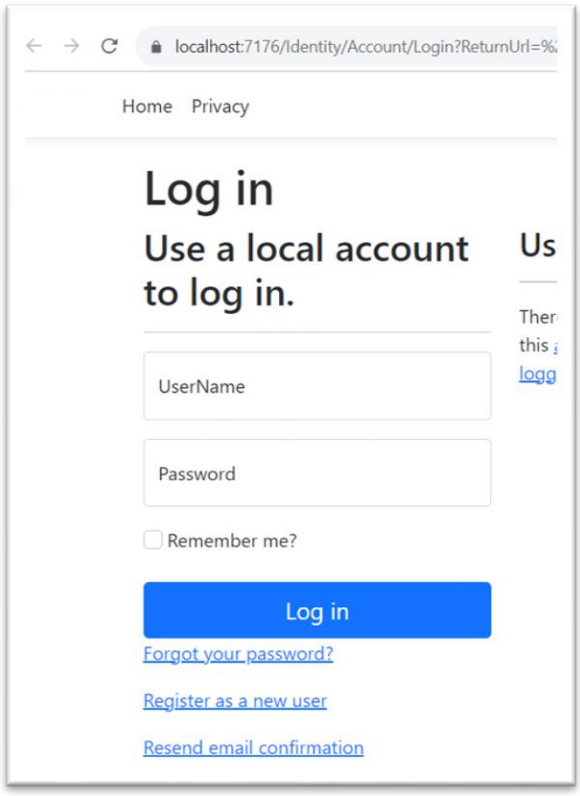
Second step – Using Authorization

The **authorization** refers to the process that determines what a User is allowed to do.

Open the **Home** controller:

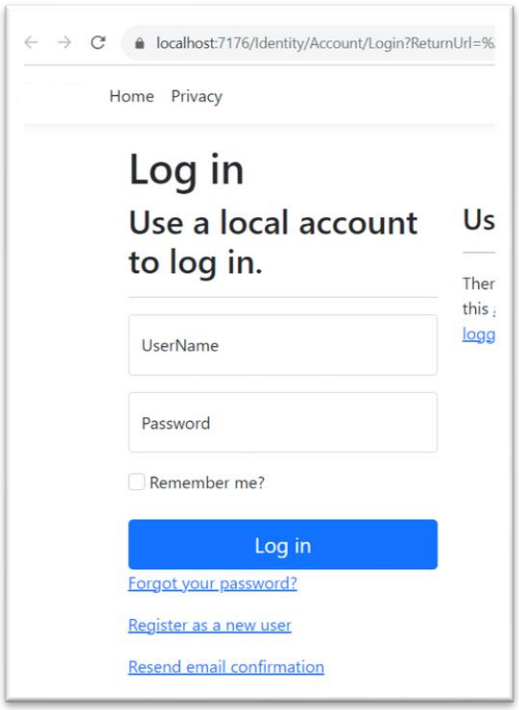
```
namespace Class08.Controllers
{
    [Authorize]
    public class HomeController : Controller
    {
        private readonly ILogger<HomeController> _logger;

        public HomeController(ILogger<HomeController> logger)
        {
            _logger = logger;
        }
```

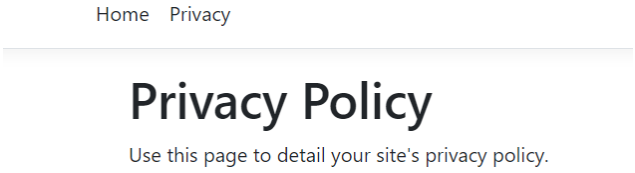
The [Authorize] filter, says that to access this controller (Home) the user needs to be authenticated. If he is not authenticated then is redirect to the login page.



- We can authorize some actions or controllers to be access to anonymous users:

- In this case, the controller is protected, but the action Privacy is not.





Privacy Policy

Use this page to detail your site's privacy policy.

Third step – Using Authorization with Roles

The **authorization** can be done using **Roles**

- Add Roles in configuration:
  - Open **Program.cs** file

```
builder.Services.AddDefaultIdentity<IdentityUser>(options => options.SignIn.Re
    .AddRoles<IdentityRole>()
    .AddEntityFrameworkStores<ApplicationDbContext>();
```

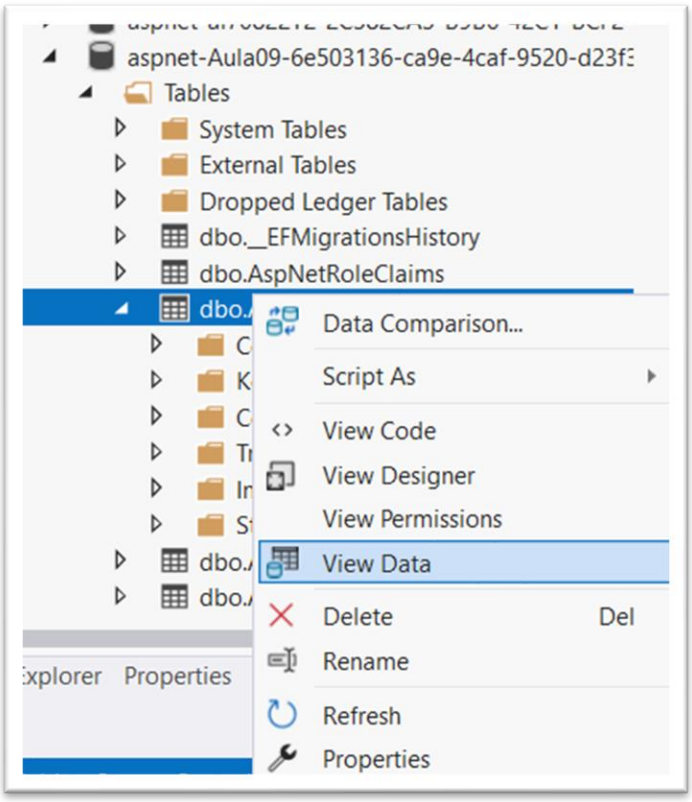
  - Create a new classe **SeedRoles.cs** in, the **Data** folder, to seed the predefined Roles:

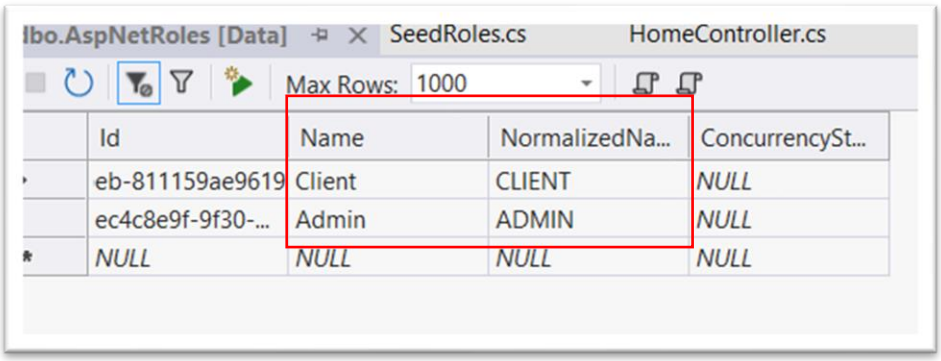
```
namespace Class08.Data
{
    public static class SeedRoles
    {
        public static void Seed(RoleManager<IdentityRole> roleManager)
        {
            if (roleManager.Roles.Any() == false)
            {
                roleManager.CreateAsync(new IdentityRole("Admin")).Wait();
                roleManager.CreateAsync(new IdentityRole("Client")).Wait();
            }
        }
    }
}
```

In the **Program.cs**, call this code:

```
builder.Services.AddControllersWithViews();

builder.Services.AddScoped<RoleManager<IdentityRole>>();

var app = builder.Build();

using(var scope = app.Services.CreateScope())
{
var roleManager=scope.ServiceProvider.GetRequiredService<RoleManager<IdentityRole>>();
    SeedRoles.Seed(roleManager);
}
```

- Run the application and close.
- Verify the existence of the new Roles.





- Alter the Register process to choose an Role:
  - Open Register.cshtml.cs file
  - Inject a RoleManager:

```
public class RegisterModel : PageModel
{
    private readonly SignInManager<IdentityUser> _signInManager;
    private readonly UserManager<IdentityUser> _userManager;
    private readonly IUserStore<IdentityUser> _userStore;
    private readonly IUserEmailStore<IdentityUser> _emailStore;
    private readonly ILogger<RegisterModel> _logger;
    private readonly IEmailSender _emailSender;
    private readonly ApplicationDbContext _dbcontext;
    private readonly RoleManager<IdentityRole> _roleManager;

    0 references
    public RegisterModel(
        UserManager<IdentityUser> userManager,
        IUserStore<IdentityUser> userStore,
        SignInManager<IdentityUser> signInManager,
        ILogger<RegisterModel> logger,
        IEmailSender emailSender,
        ApplicationDbContext dbContext,
        RoleManager<IdentityRole> roleManager
        )
    {
        _userManager = userManager;
        _userStore = userStore;
        _emailStore = GetEmailStore();
        _signInManager = signInManager;
        _logger = logger;
        _emailSender = emailSender;
        _dbcontext = dbContext;
        _roleManager= roleManager;
    }
```

  - Add a new property in the InputModel

```
6 references
public string Email { get; set; }

[Required]
0 references
public string Role { get; set; }
```

- Send the existing **Roles** to the view using the **ViewData** structure.

```
public async Task OnGetAsync(string returnUrl = null)
{
    ViewData["roles"] = _roleManager.Roles.ToList();
    ReturnUrl = returnUrl;
    ExternalLogins = (await _signInManager.GetExternalAuthenticationSchemesAsync()).To
}
```

- Alter the View

```
<div class="form-floating mb-3">
    <label asp-for="Input.Role"></label>
    <select asp-for="Input.Role" class="form-control" asp-items='new SelectList(ViewBag.roles,"Name","Name")'>
        <option value="">-- Select a Role --</option>
    </select>
    <span asp-validation-for="Input.Role" class="text-danger"></span>
</div>
```

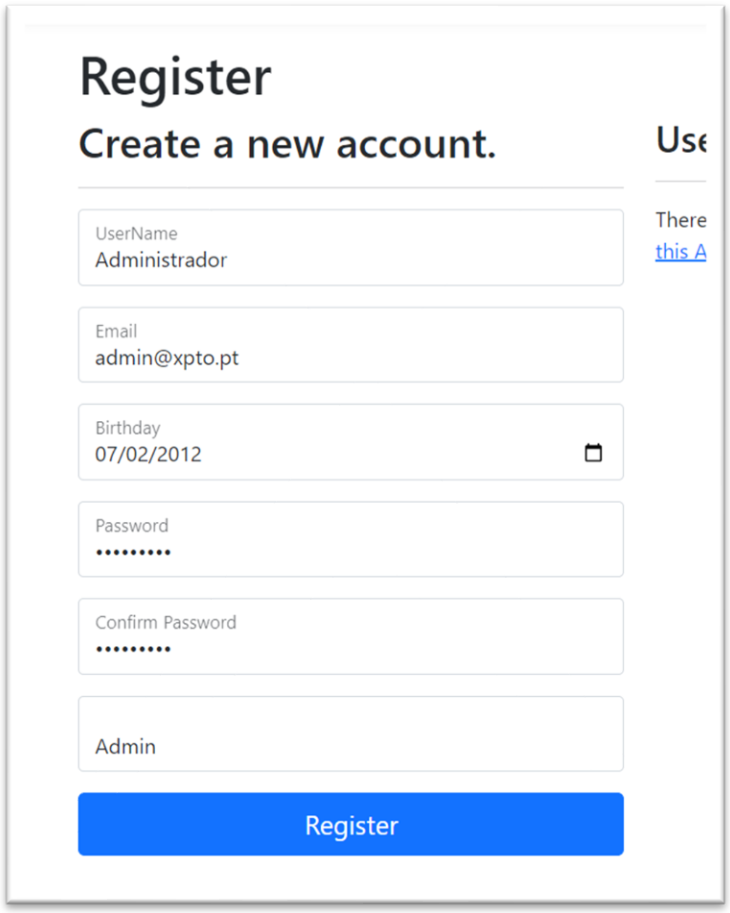
- Alter de OnPostAsync method

```
if (result.Succeeded)
{
    _logger.LogInformation("User created a new account with password.");

    await _userManager.AddToRoleAsync(user, Input.Role);

    Perfil newPerfil= new Perfil { UserName = user.UserName, Birthday = Input.Birt

    _dbcontext.Perfils.Add(newPerfil);
```

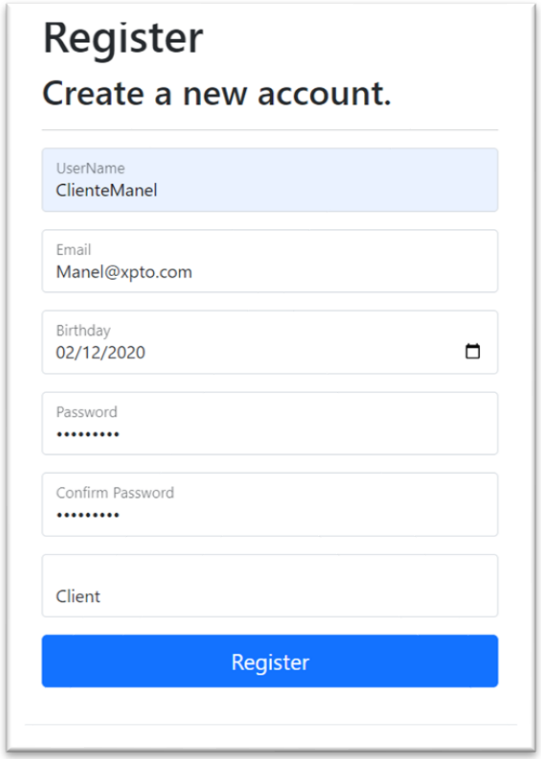
…

```
    }

    // If we got this far, something failed, redisplay form
    ViewData["roles"] = _roleManager.Roles.ToList();
    return Page();
}
```

- Test register a new user of the Role “Admin”



- Create a Client user



- With the roles created and users of those roles, we can do the protection through the roles:

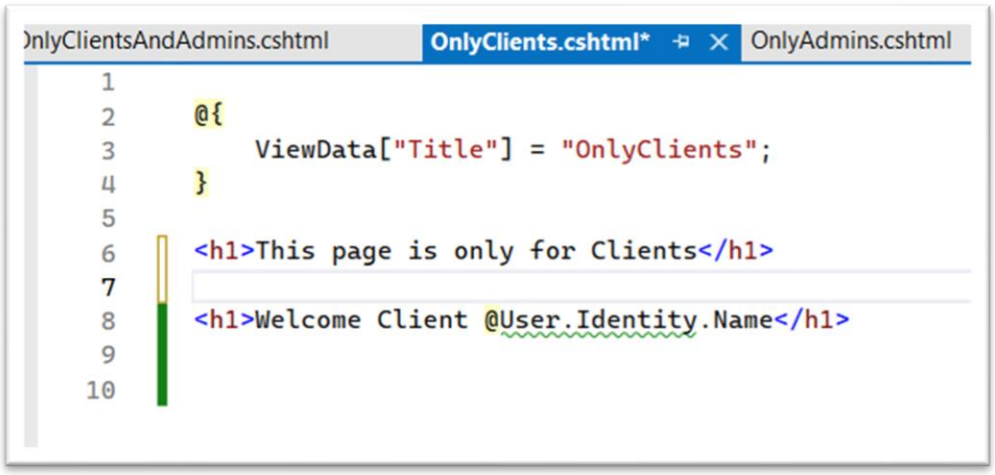
- open the Home Control and create the Actions

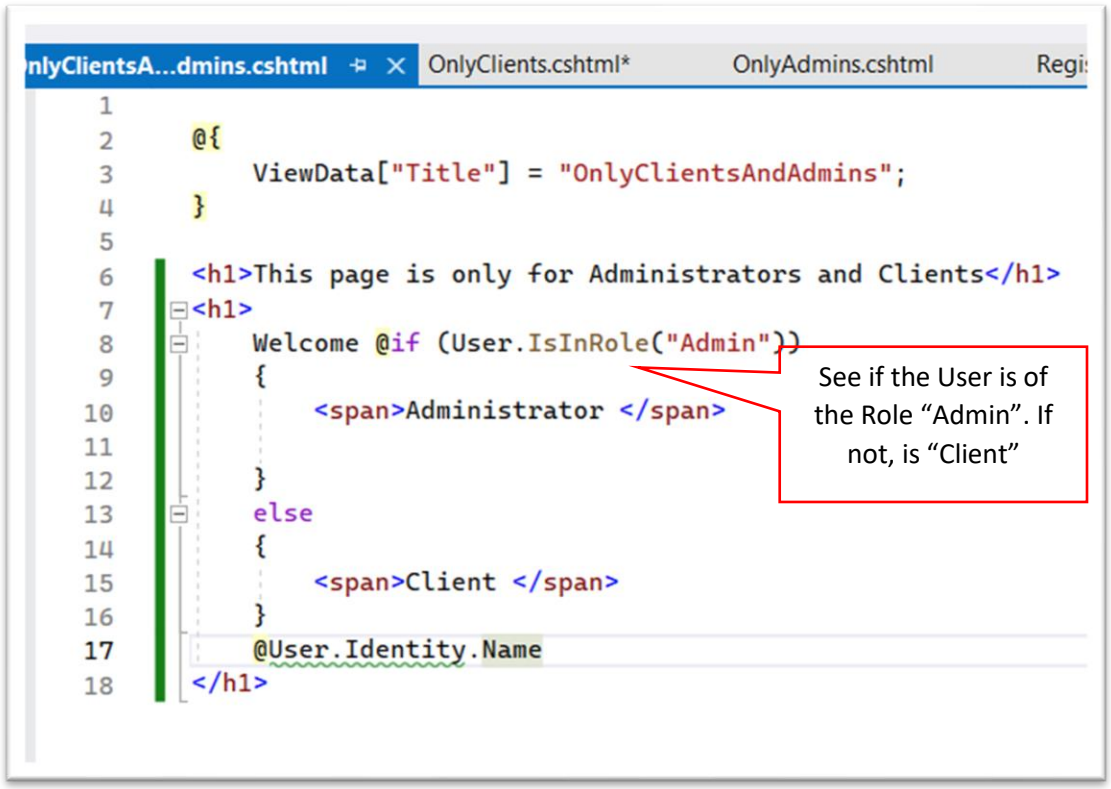
```
[Authorize(Roles ="Admin")]
0 references
public IActionResult OnlyAdmins()
{
    return View();
}

[Authorize(Roles = "Client")]
0 references
public IActionResult OnlyClients()
{
    return View();
}

[Authorize(Roles = "Admin, Client")]
0 references
public IActionResult OnlyClientsAndAdmins()
{
    return View();
}
```

- Create the correspondents Views

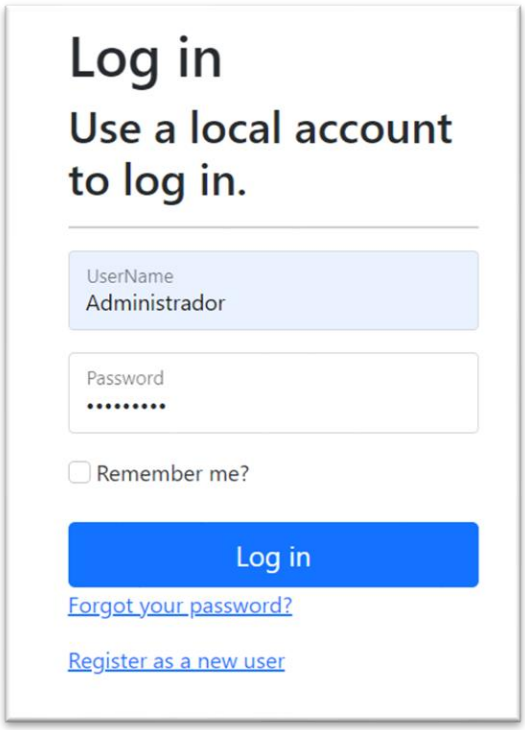
OnlyClientsAndAdmins.cshtml | OnlyClients.cshtml | OnlyAdmins.cshtml

```
1
2   @{
3       ViewData["Title"] = "OnlyAdmins";
4   }
5
6   <h1>This page is only for Administrators</h1>
7
8   <h1>Welcome Administrator @User.Identity.Name</h1>
9
10
11
```


OnlyClientsAndAdmins.cshtml
OnlyClients.cshtml*
OnlyAdmins.cshtml
@{
ViewData["Title"] = "OnlyClients";
}
<h1>This page is only for Clients</h1>
<h1>Welcome Client @User.Identity.Name</h1>


nlyClientsA...dmins.cshtml
OnlyClients.cshtml*
OnlyAdmins.cshtml
Regi
@{
ViewData["Title"] = "OnlyClientsAndAdmins";
}
<h1>This page is only for Administrators and Clients</h1>
<h1>
Welcome @if (User.IsInRole("Admin"))
{
Administrator
}
else
{
Client
}
@User.Identity.Name
</h1>
See if the User is of the Role “Admin”. If not, is “Client”

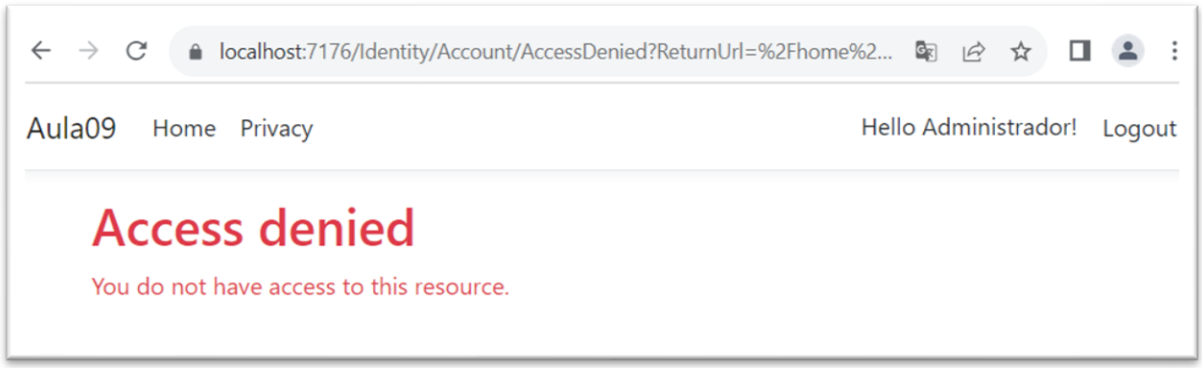
- Run the application and Login with Administrator account



- go to OnlyAdmins action: https://localhost:44328/home/onlyadmins.

Have access, because the user is in the Role “Admin”

- go to OnlyAdmins action: https://localhost:44328/home/onlyclients.



Do not have access because the user is not a “Client”

- go to OnlyAdmins action: https://localhost:44328/home/OnlyClientsAndAdmins.

- Test the application whit a user of Role “Client”.

- Adapt the Menu to show only the options allowed to the role of the user authenticated.
  - Open the _Layout.cshtml file: