PROG 2B

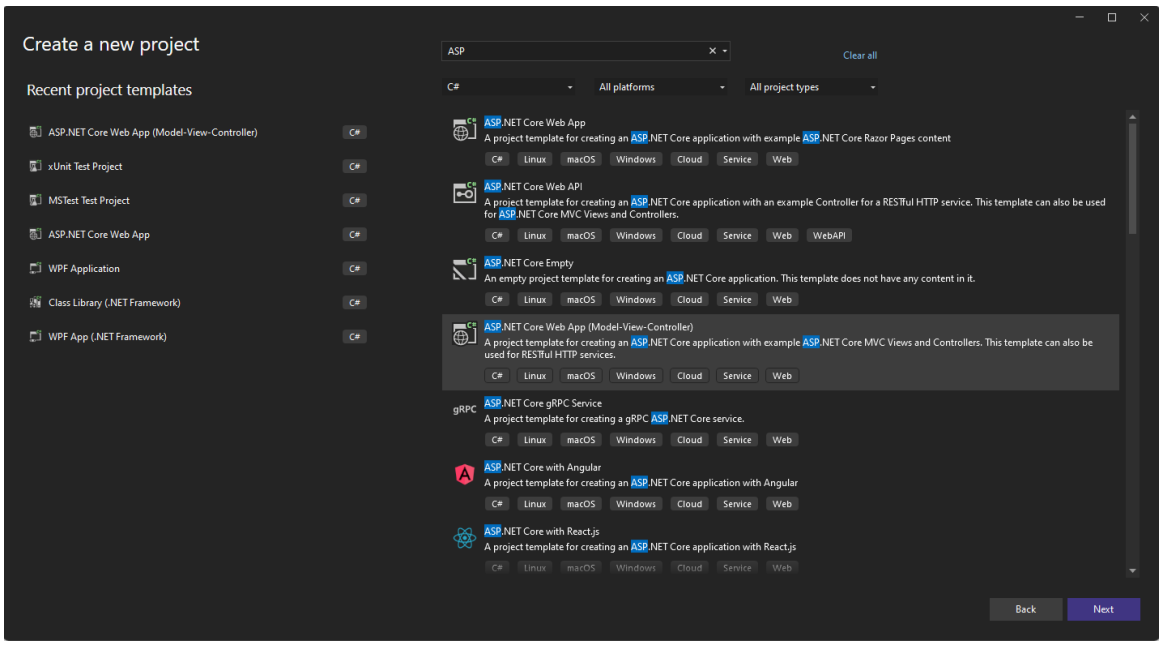
Basic ASP.Net Core Web App (MVC) using Individual Accounts

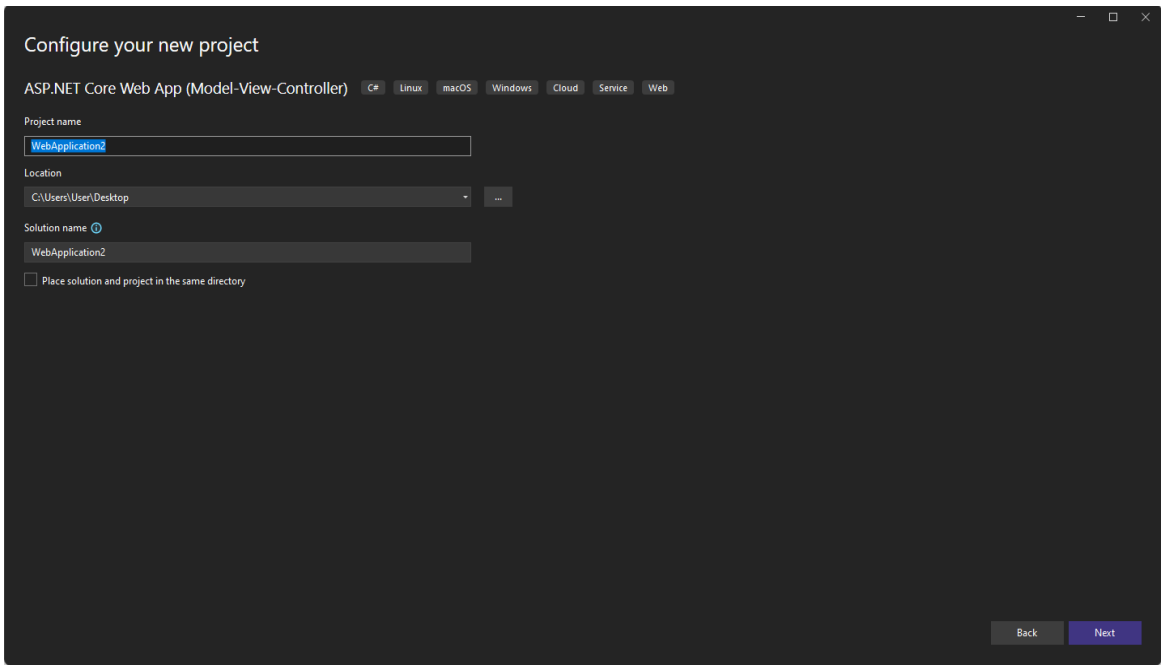
By Aaron Fourie

Contents

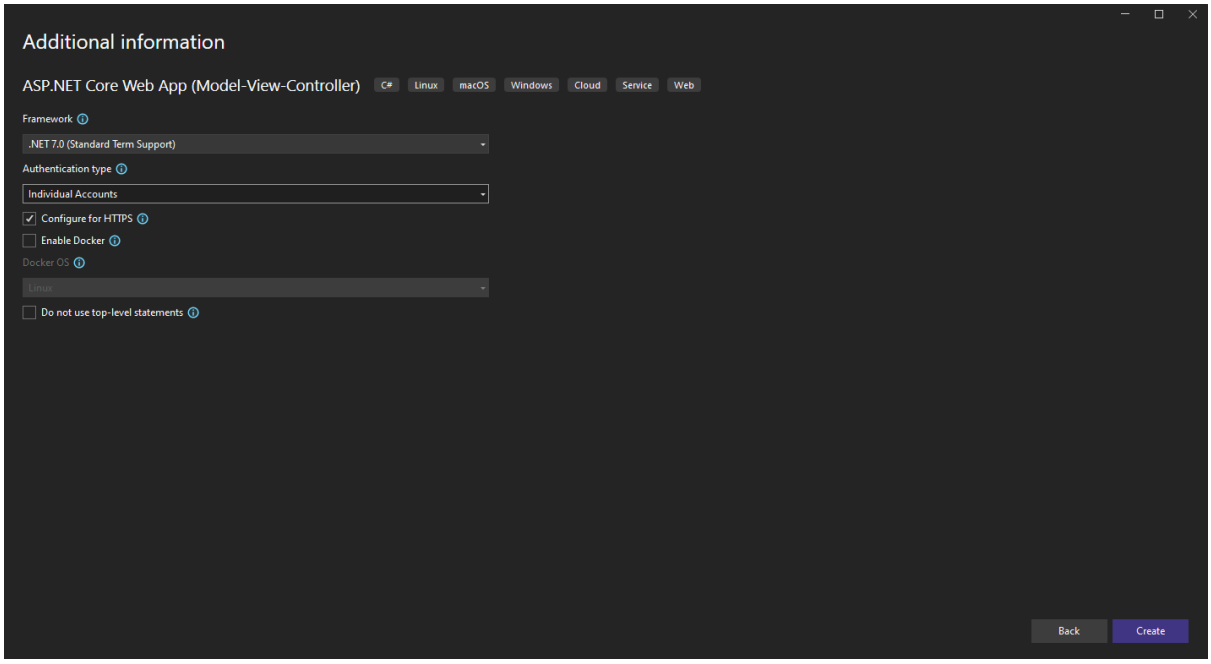
Creating the project

- Open Visual Studio and create a new project
- Select ASP.Net Core Web App (Model-View-Controller)

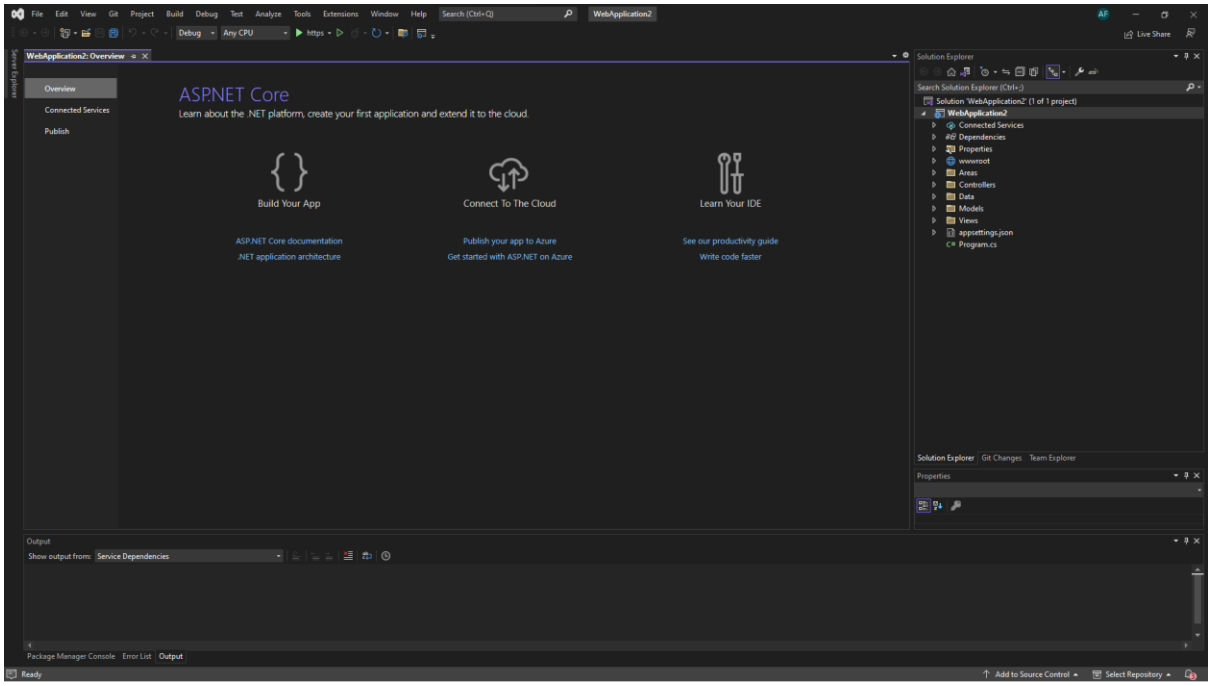




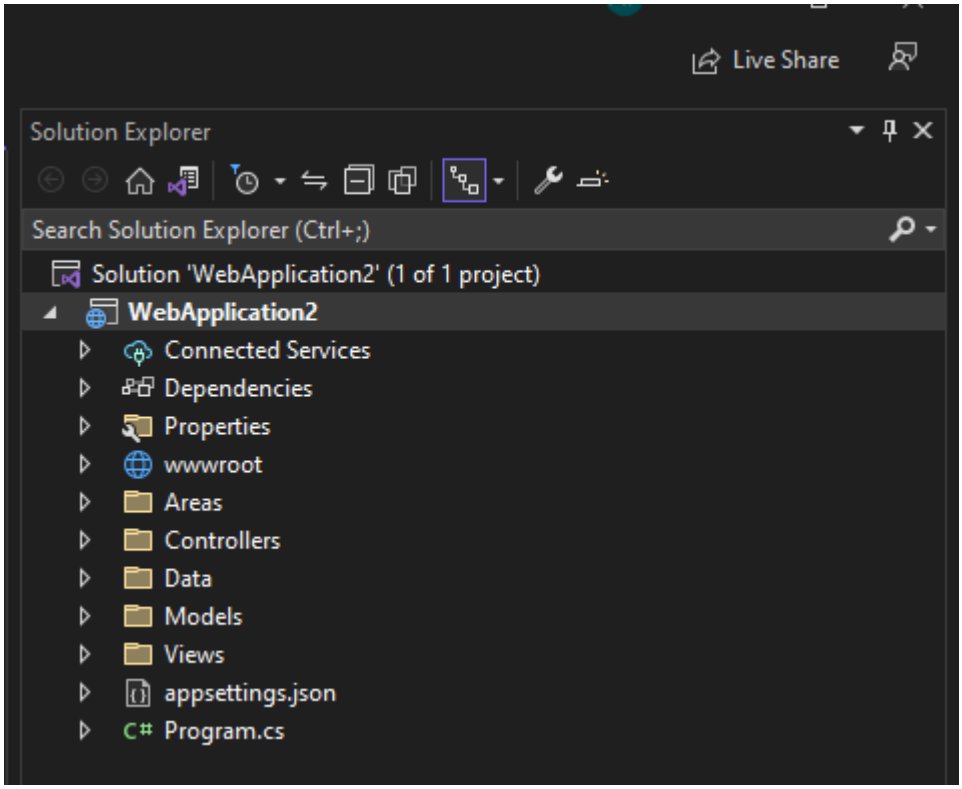
- Ensure that the latest stable version of Dot Net Framework is selected (.NET 7 or 8)
- Under Authentication Type select Individual Accounts
- Click Create



Your project should then look like this



Here’s a snapshot of the default Solution Explorer



Default Solution Explorer File Structure Breakdown

Important Folders:

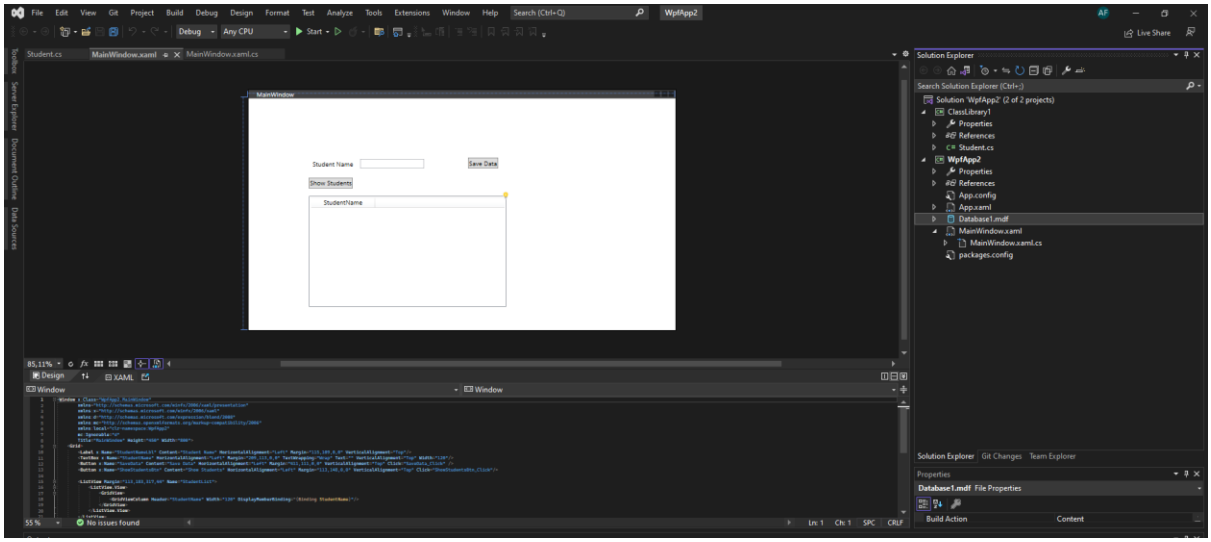
Folder	Importance	System Architectural Level
Areas	Identity folder for register, login and logout models and controllers	Business Logic Level
Controllers	Interaction business logic between models and views	Business Logic Level
Data	Database context and migrations	Database Level
Models	Model classes with data annotations	Database Level
Views	Cshtml razor pages	Presentation Level

Important Files:

File	Importance
appsettings.json	Configuration of app connection strings, logging, and allowed hosts
Program.cs	App execution file. Configuration of app builder and services including development environment and routing.

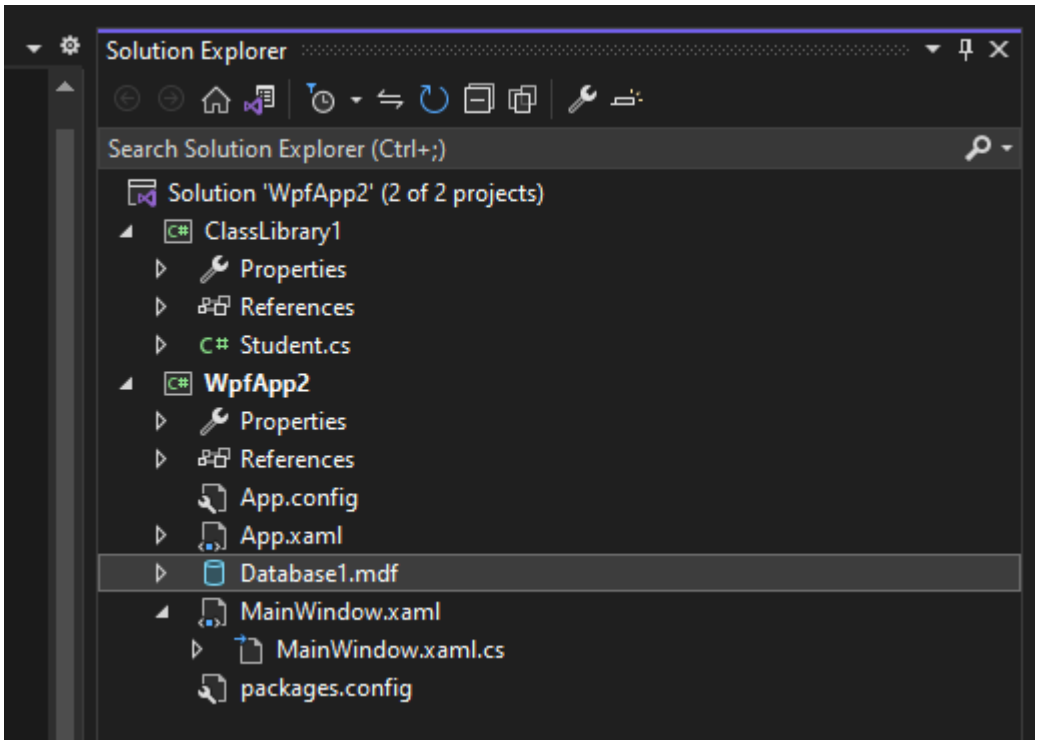
Part 2 WPF App Reference

Here is the Basic WPF app with SQL database from the last guide.



To continue this example, we are going to make use of this Class library and Service Based Database (You should use your WPF app from part 2).

Here is a closeup of the Solution Explorer for the WPF app



As we can see the Class Library and Database is in two different projects.

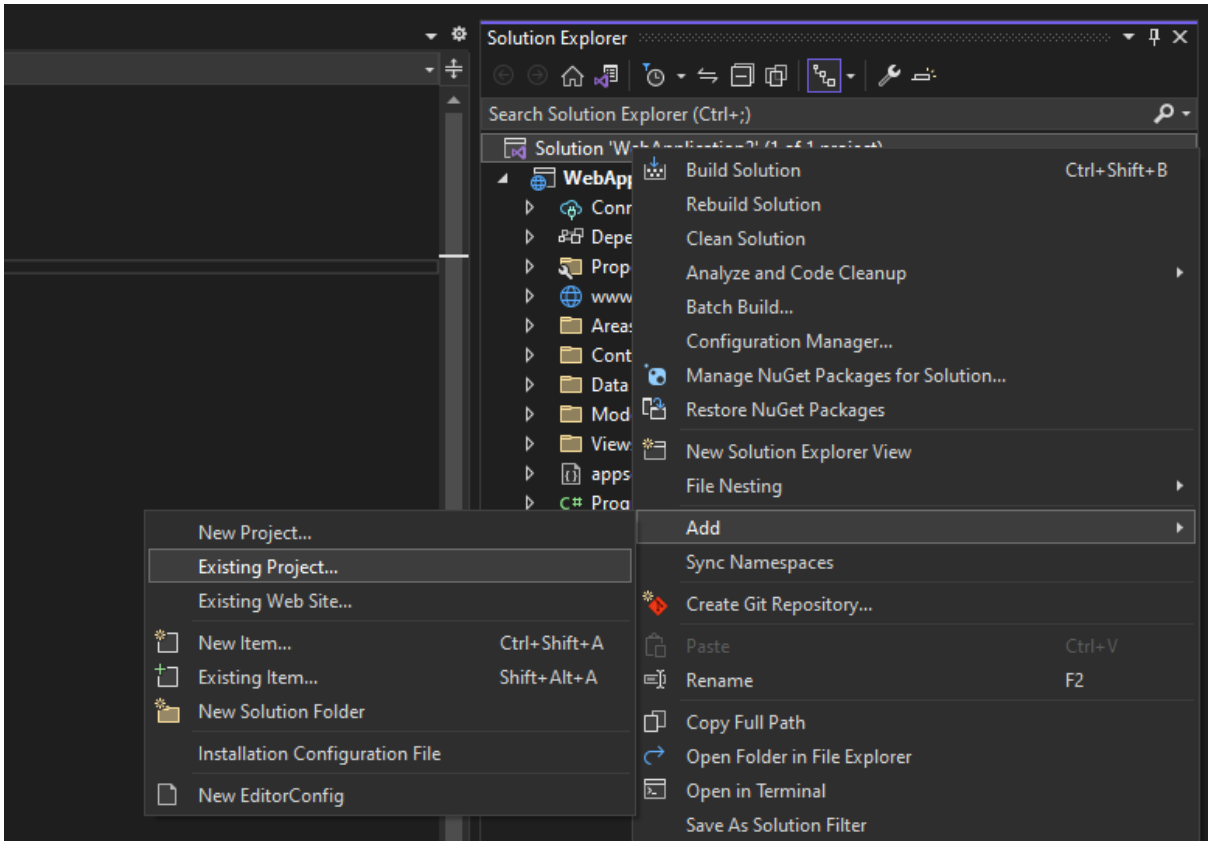
We need to bring both of them into our ASP.Net Core Web App

Let's first add in the Class library then the Database

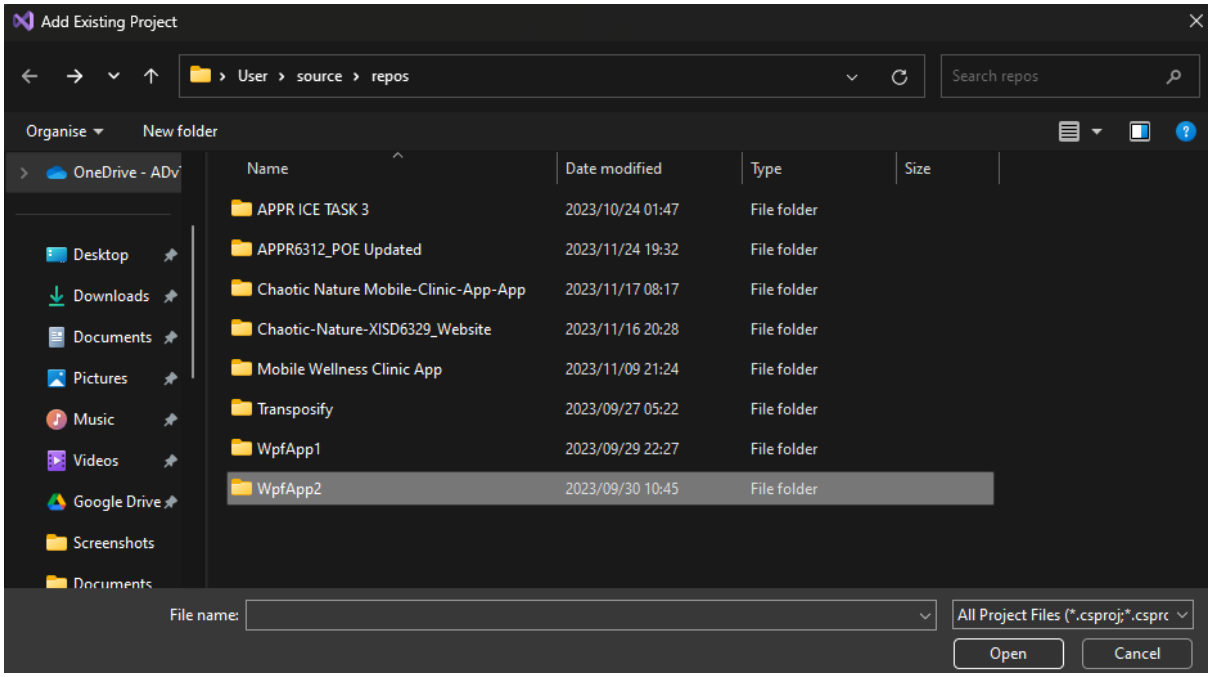
Setting up the Project

1) Adding the Class Library

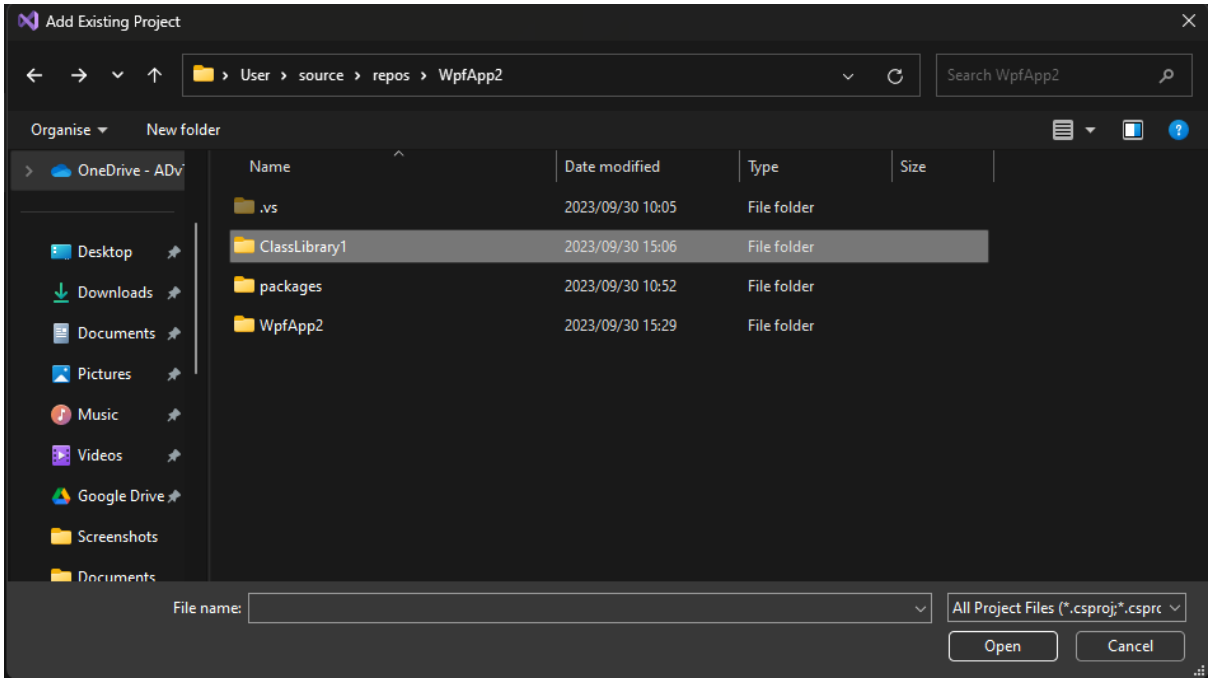
- Right Click the Solution -> Add -> Existing Project



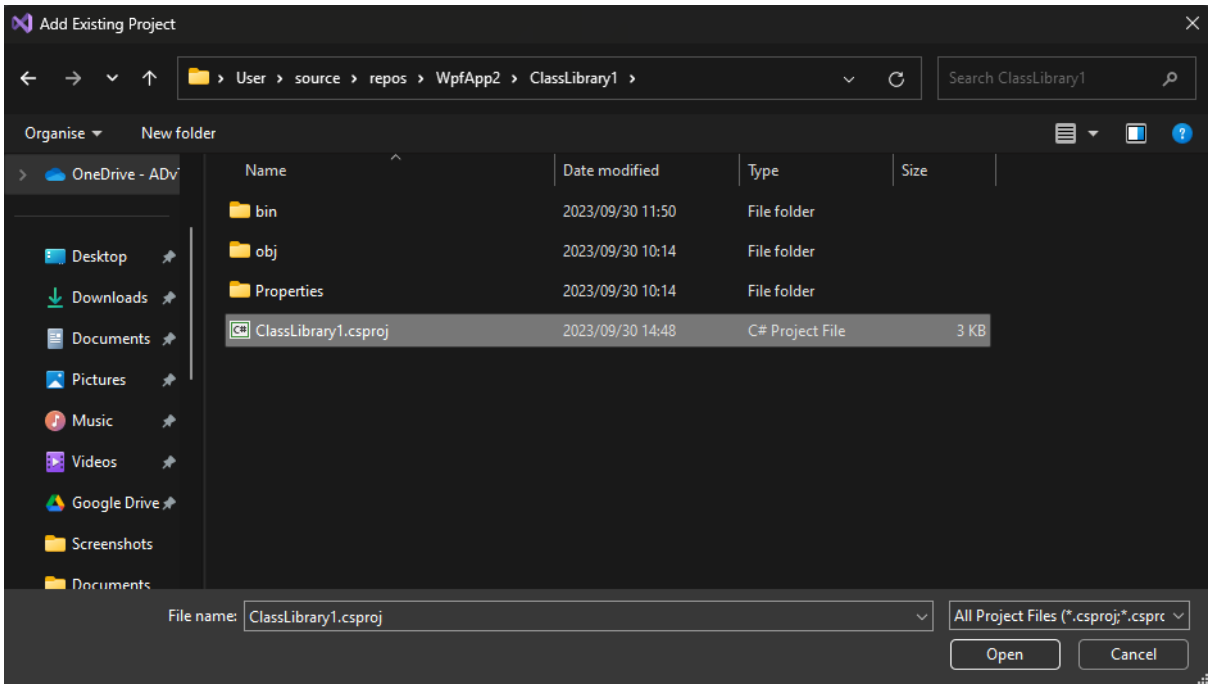
- Open the Part 2 WPF Project



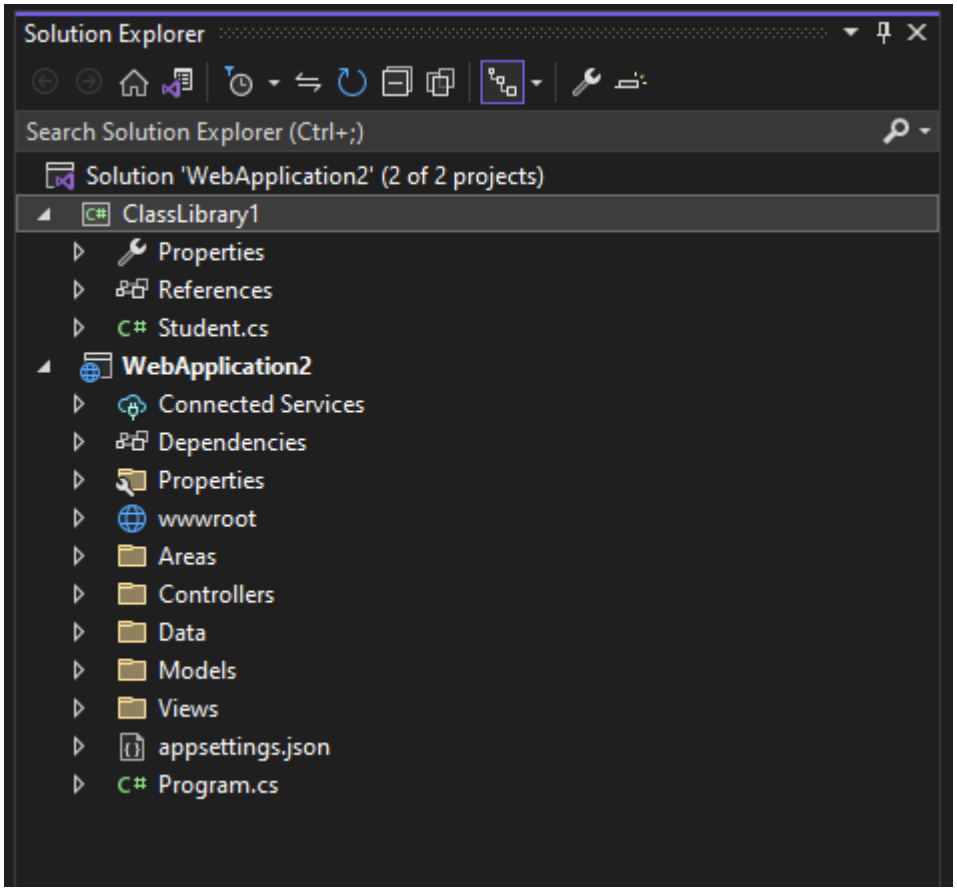
- Then Open the Class Library



- Open the .csproj file

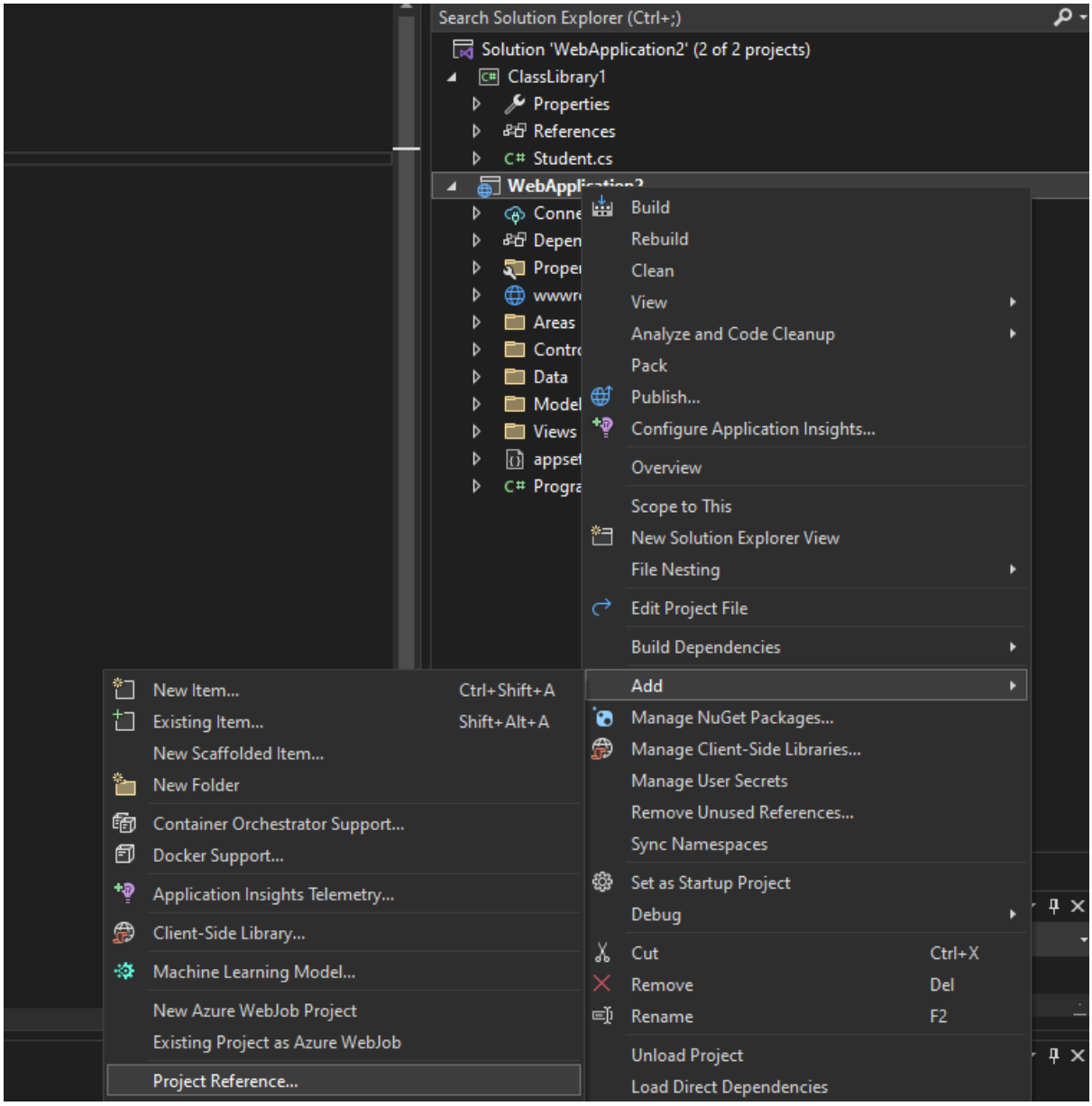


The class library should now be added to your solution explorer

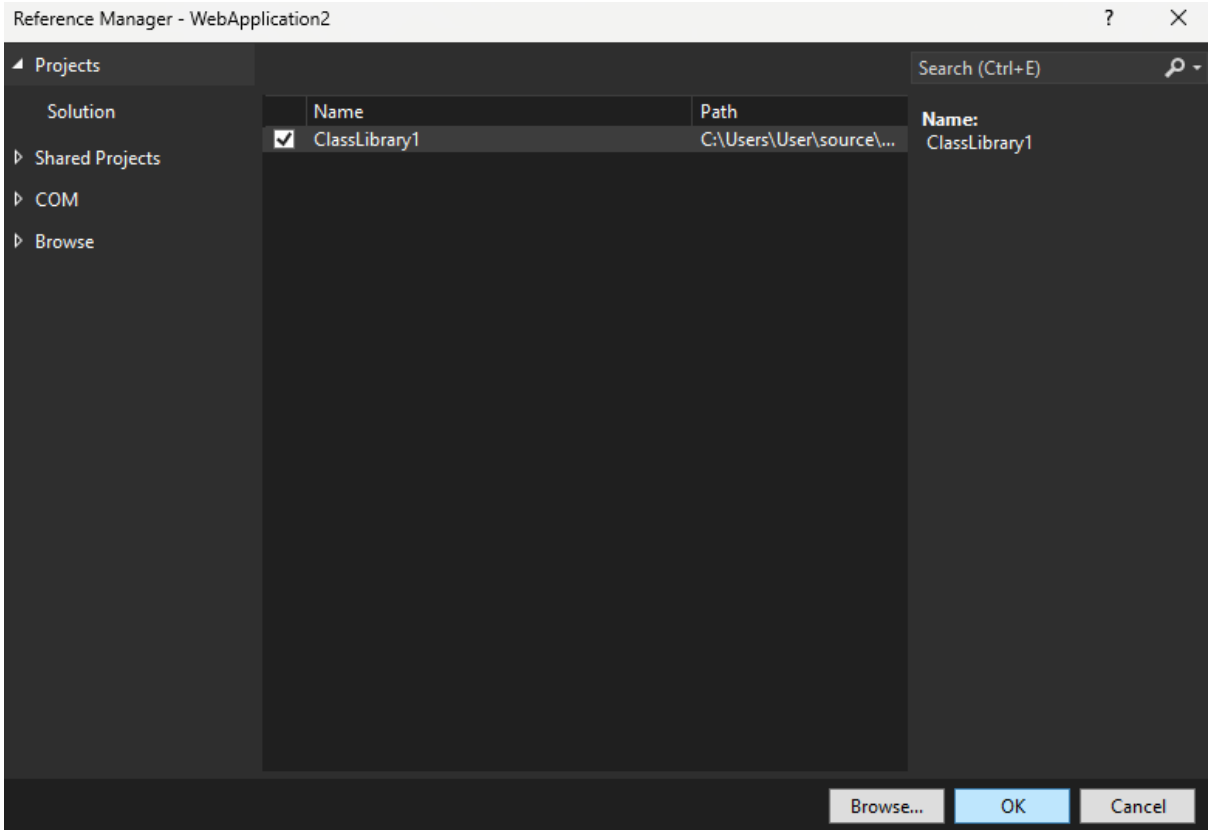


Now let’s add a reference to the class library so we can use it in or ASP.Net Core Web App

- Right Click your ASP.Net Core Web App -> Add-> Project Reference…



- Select The class library and hit OK



Your Class library is now setup successfully in the ASP.Net Core app.

Let’s move on and add the Database

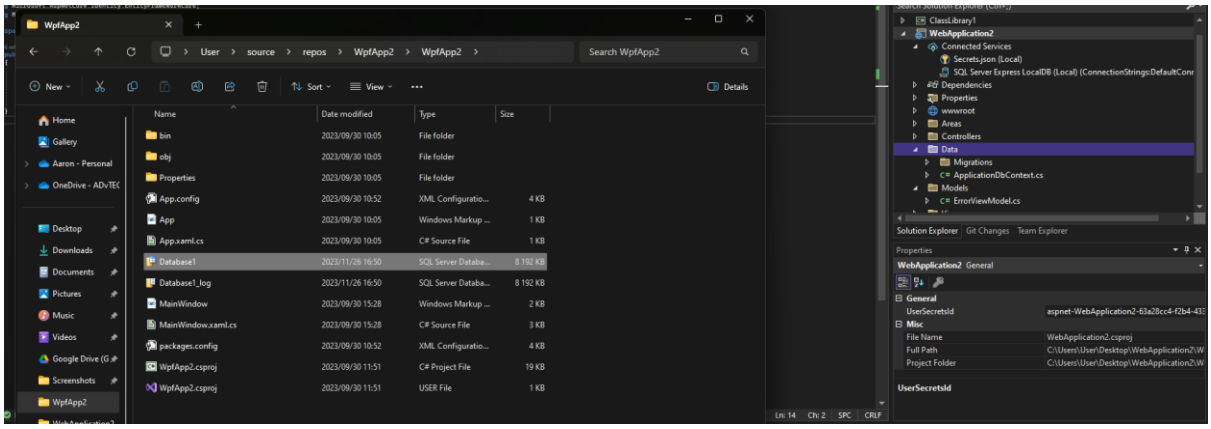
2) Adding the Database

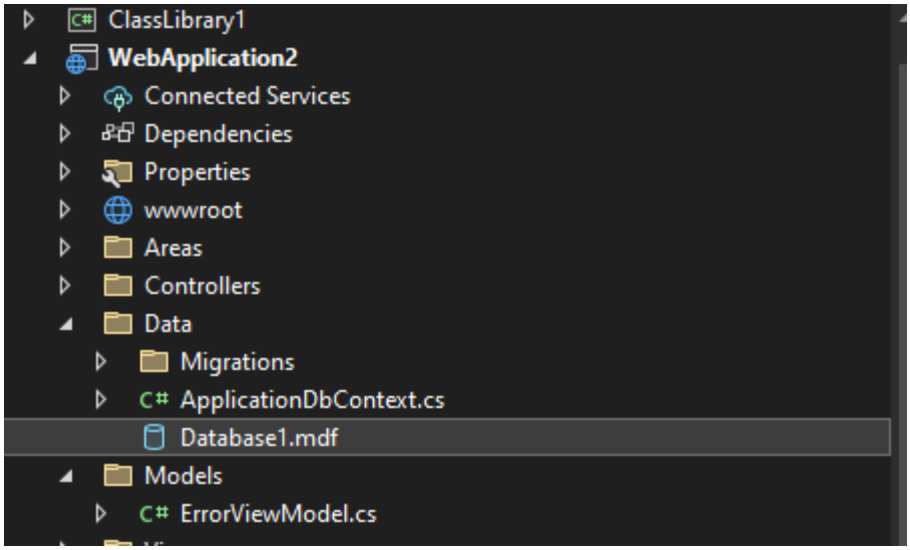
Since we are supposed to connect to the database using ADO.Net or Entity Framework (EF), we will need to first need to add in the connection.

The simplest wassy to do this would be to copy the mdf file from our wpf app and paste it in our ASP.NET Core app.

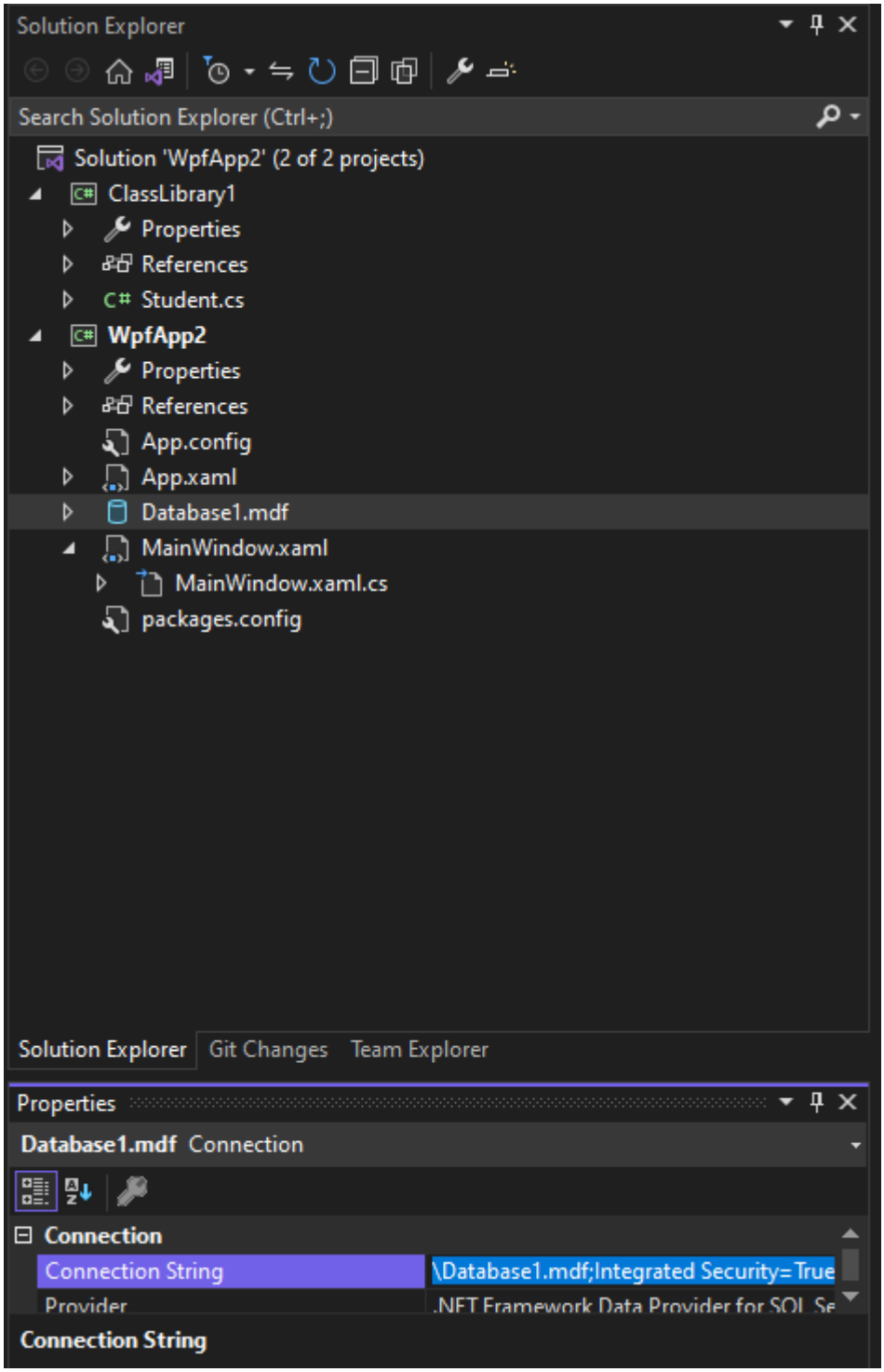
Lets do it.

Locate the database mdf file then drag and drop it to your Data folder in the ASP.NET Core App

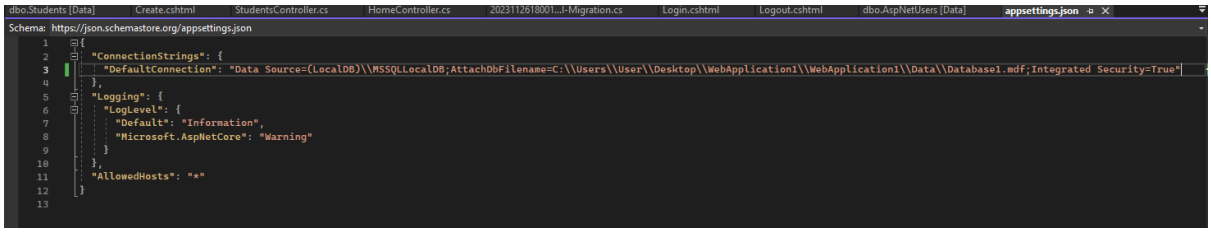




After it’s done you will be able to access your database from the SQL Server Object Explorer



Copy and replace the connection string to your appsettings.json file



As part of part 2 additional requirements, we need to connect to our database using EF or ADO.NET

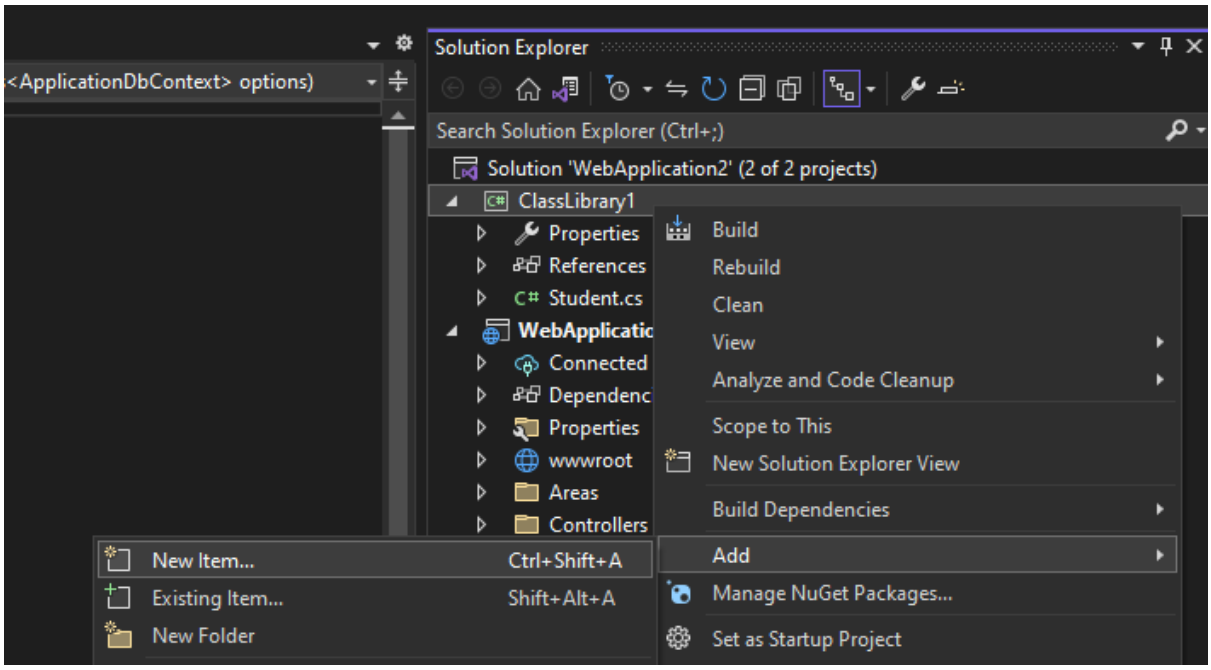
Let's follow the ADO.NET connection layer approach

3) Adding the ADO.NET Database Connection Layer

For this we will need to add an Item to our Class Library. This will generate the classes and dB context for you and It'll make the project a lot easier moving forward.

In your Solution explorer:

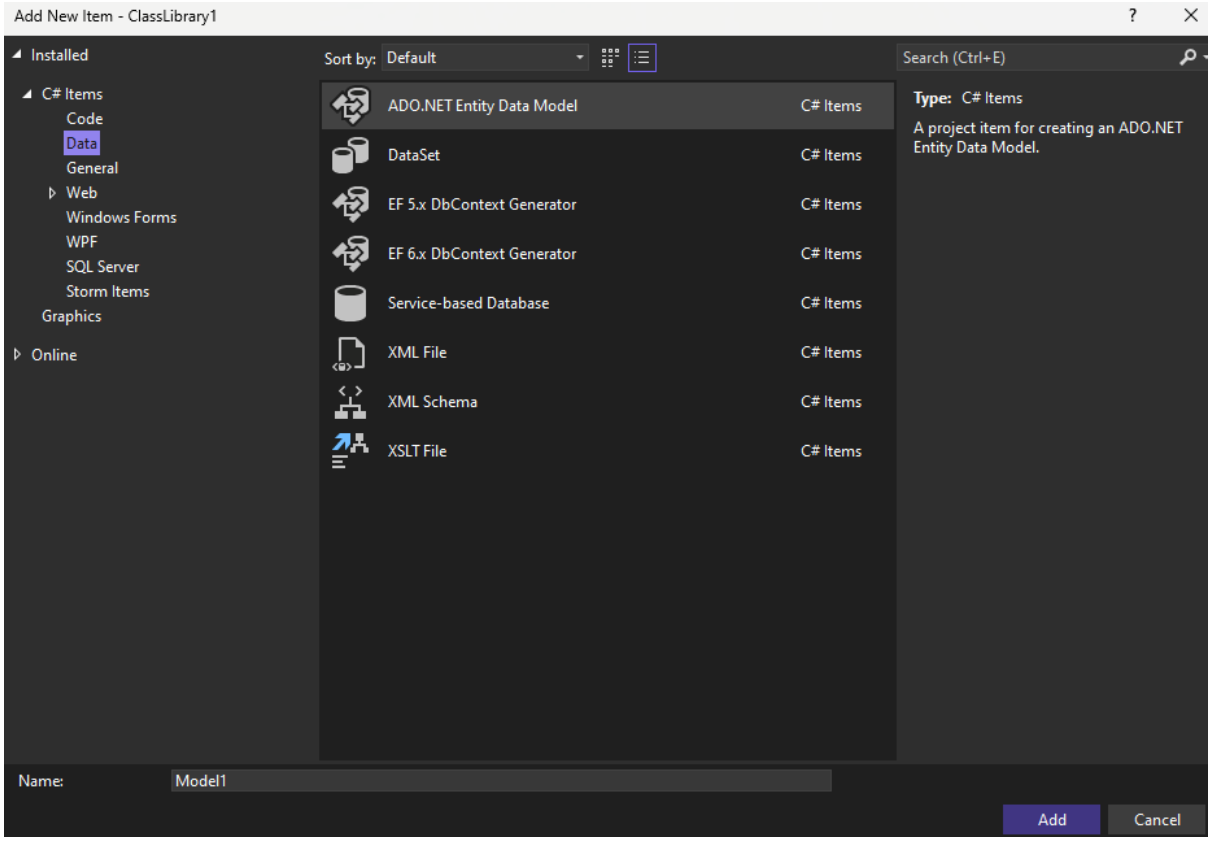
- Right click the Class Library -> Add -> New Item



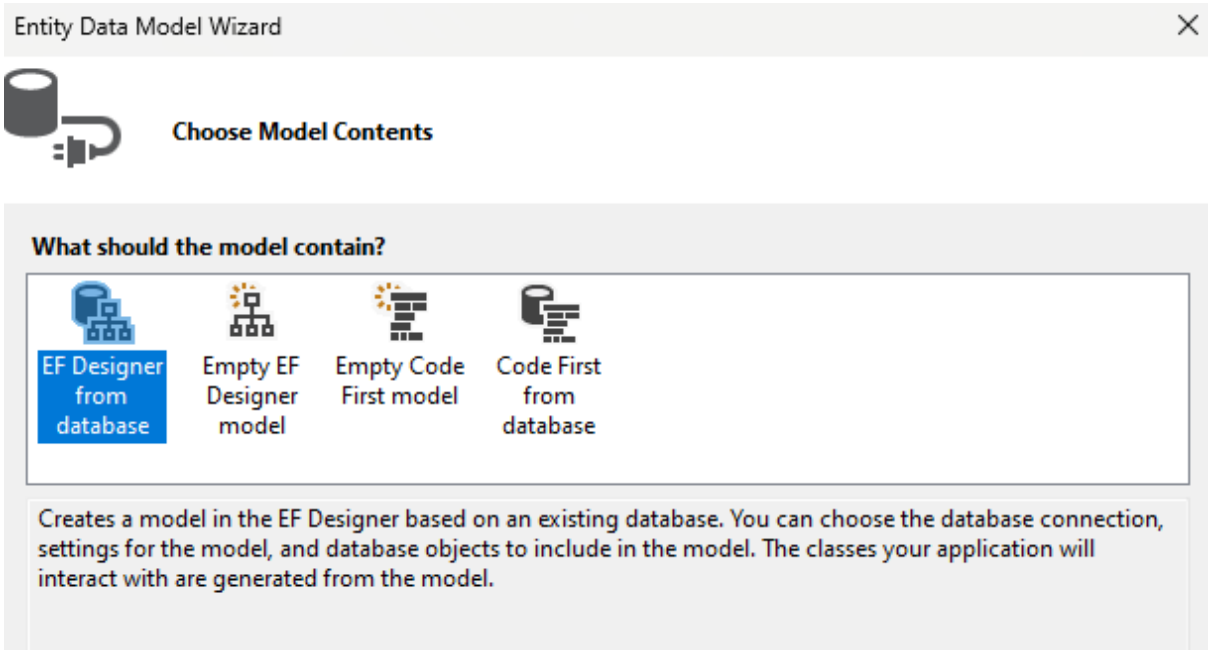
- Under C# Items -> Data -> ADO.NET Entity Data Model

3.1. Why are we using an ADO.NET Entity Data Model?

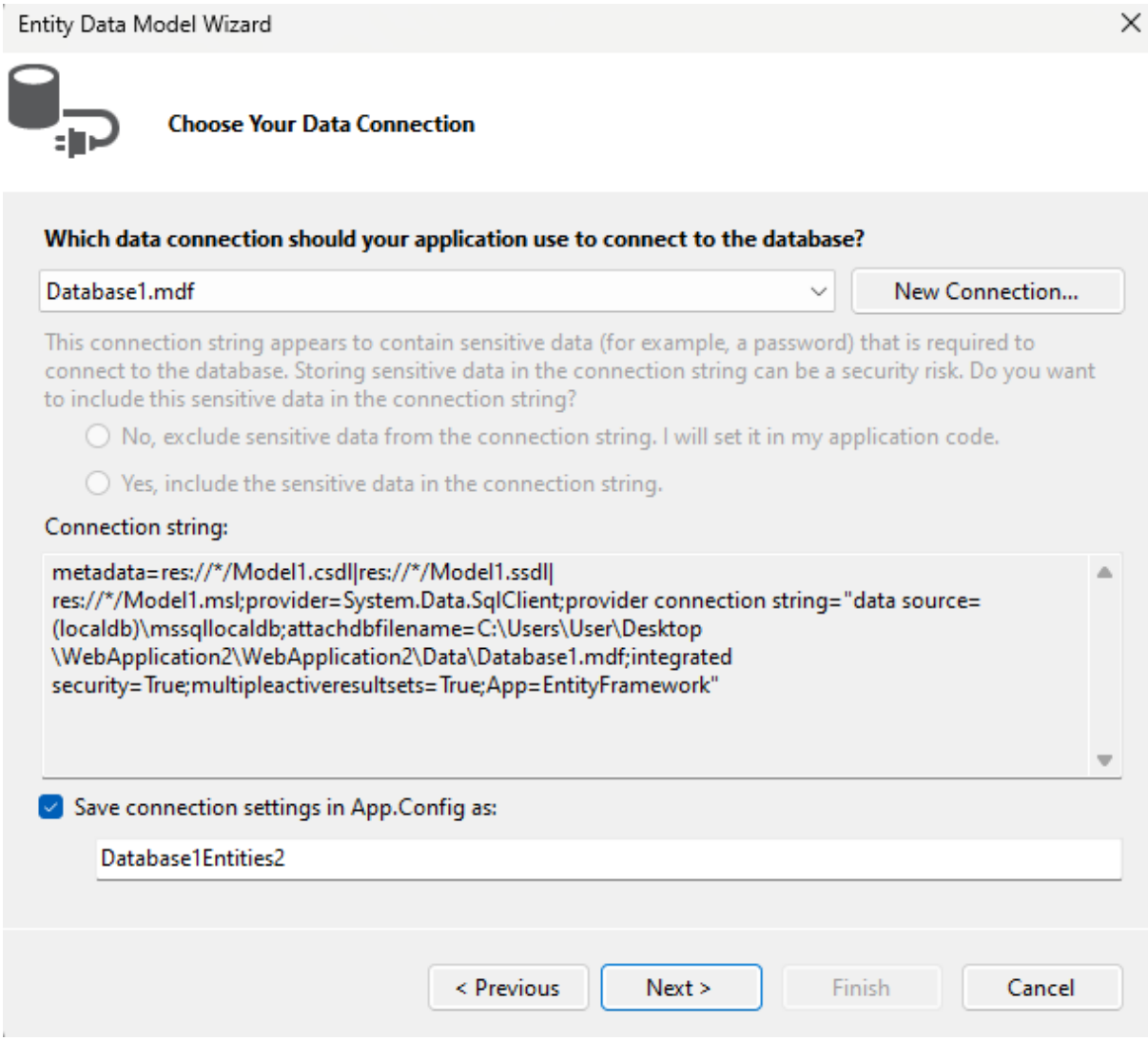
ADO.NET is the only data option that allows you to choose a database connection, settings for the model, and database objects to include in the model.



- Click Add
- Select EF Designer from Database



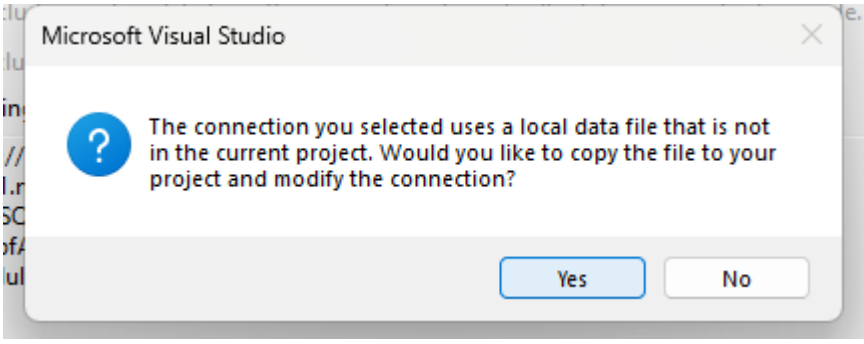
- Click OK
- Choose the Given connection (its already setup)



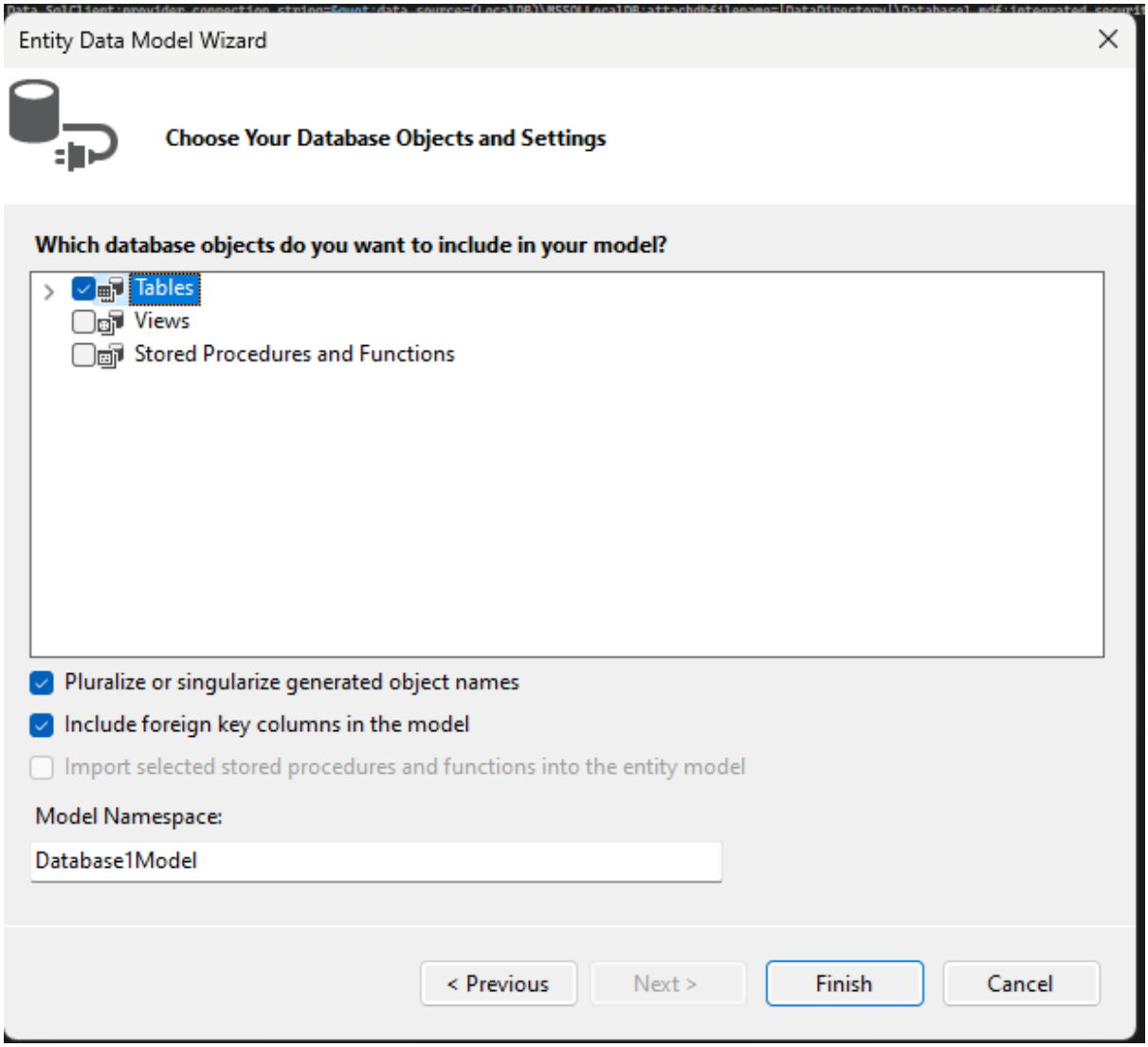
- Its connected successfully! Now click OK, and OK again

You can name it whatever but, in this example, let's leave it as default, then click Next

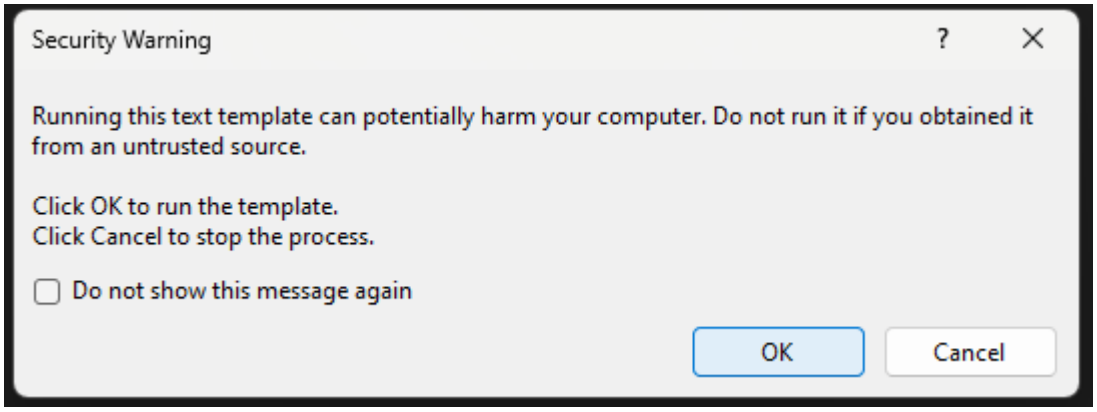
- Click Yes



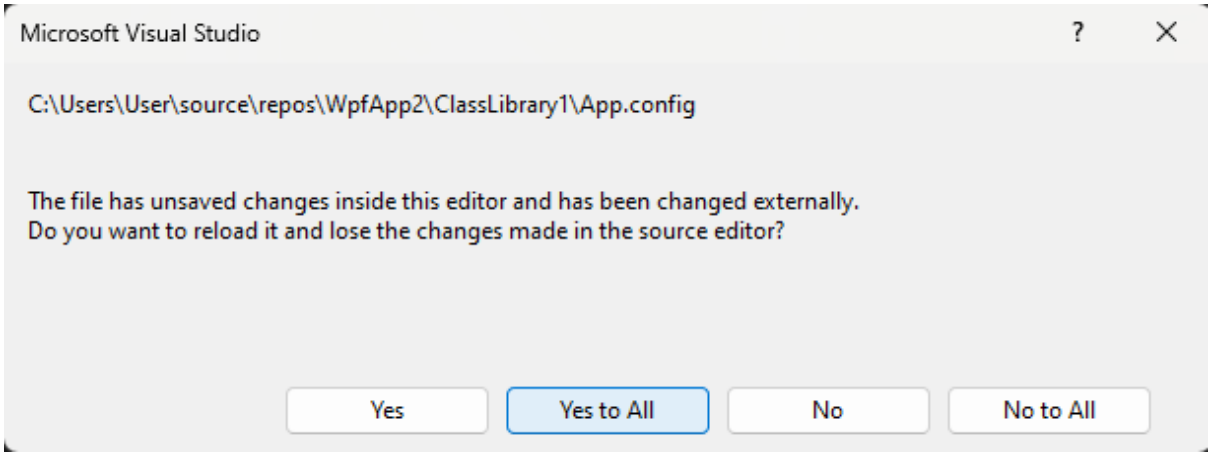
- Select Tables and click Finish



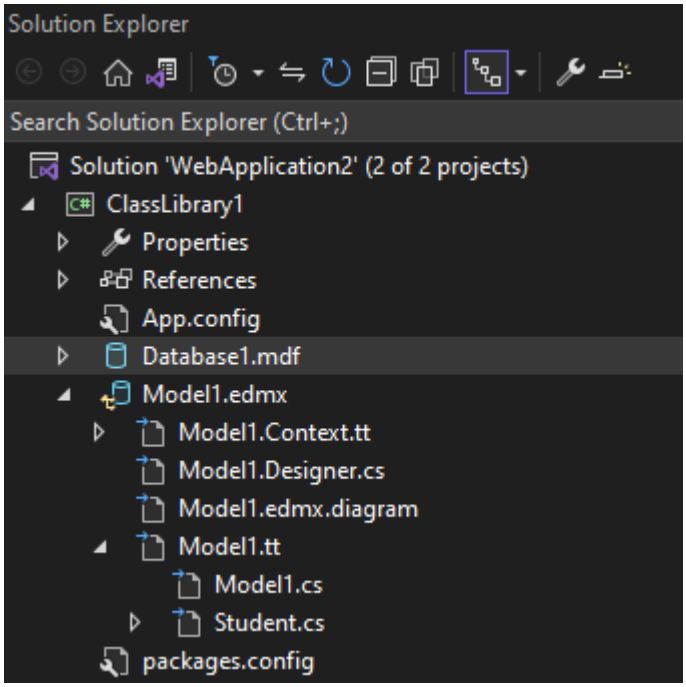
- Click OK by the security warnings

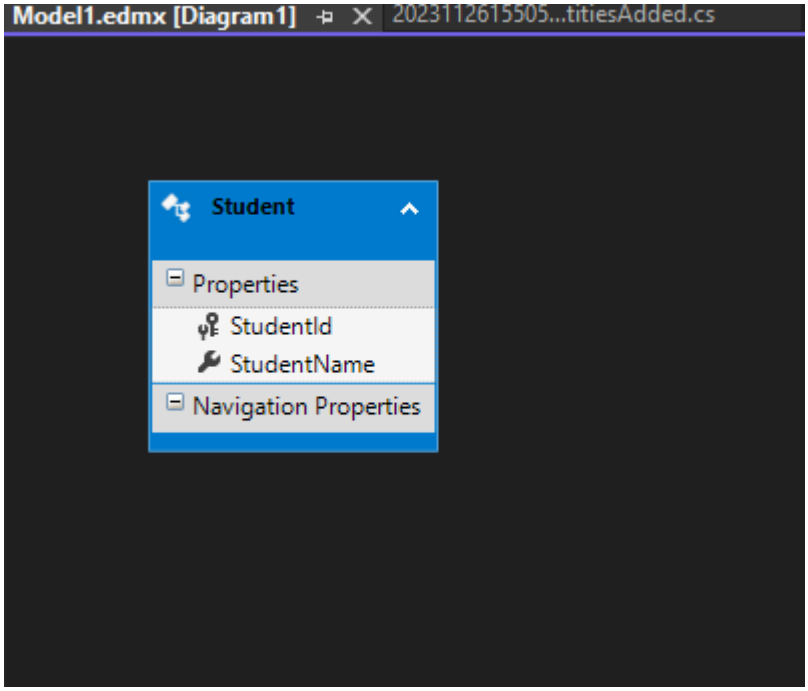


- Select Yes to All



The solution explorer should now look similar to this, you will have different Model classes





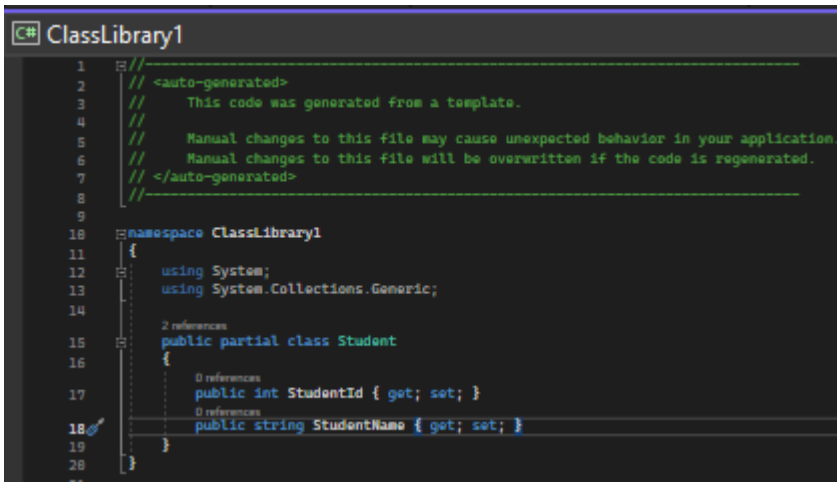
Now we brought in our database into our ASP.NET Core Project so we can query it from the Server Explorer anytime just by double clicking on the Database mdf file.

We won't have to manually add the database connection every time we open vs.

We also have our model classes created for us

We'll come back to this later

Here's our auto generated Students.cs located in the Model.tt file



Now we need to setup our Database Context

4) Setting up the Database Context

4.1. Working with the right Database Context

If you notice we have 2 DB Contexts, your auto generated Model Context in the class library and the default ApplicationDbContext in the ASP.NET Core web app.

```
WebApplication2
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore;

namespace WebApplication2.Data
{
    6 references
    public class ApplicationDbContext : IdentityDbContext
    {
        0 references
        public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
            : base(options)
        {
        }
    }
}
```

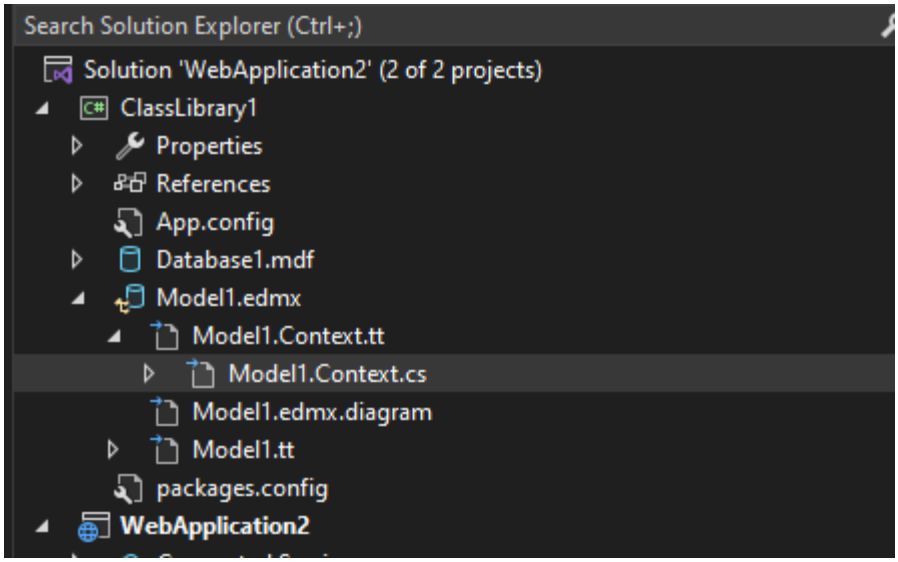
Figure 1: ApplicationDbContext

```
Model1.tt    Model1.edmx.diagram    Model1.Context.cs    Mo
ClassLibrary1
// <auto-generated>
//     This code was generated from a template.
//
//     Manual changes to this file may cause unexpected behavior in your application.
//     Manual changes to this file will be overwritten if the code is regenerated.
// </auto-generated>
//------------------------------------------------------------------------------

namespace ClassLibrary1
{
    using System;
    using System.Data.Entity;
    using System.Data.Entity.Infrastructure;

    1 reference
    public partial class DatabaseEntities : DbContext
    {
        0 references
        public DatabaseEntities()
            : base("name=DatabaseEntities")
        {
        }

        0 references
        protected override void OnModelCreating(DbModelBuilder modelBuilder)
        {
            throw new UnintentionalCodeFirstException();
        }

        0 references
        public virtual DbSet<Student> Students { get; set; }
    }
}
```

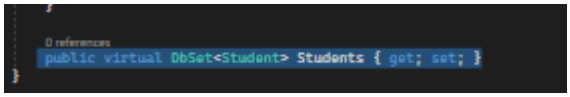
Figure 2: Auto generated DatabaseEntities DbContext

Our app is already setup to use ApplicationDbContext by default. So, let's move the dB Sets over from our class library to our ApplicationDbContext and then delete it

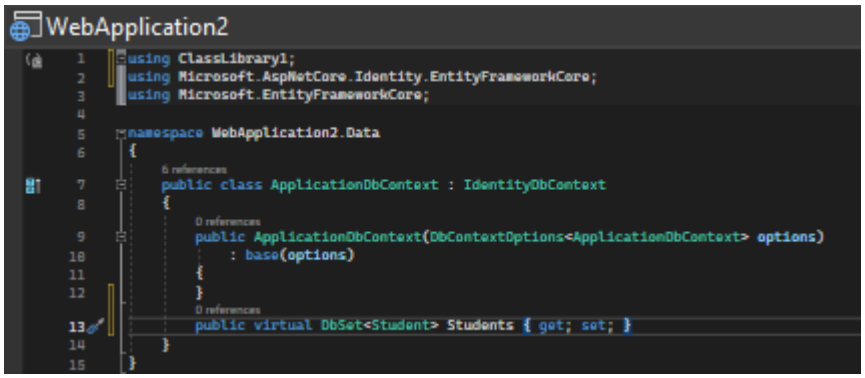
- Double Click on your Model.Context.cs file



- Copy the auto generated DbSets in the class library



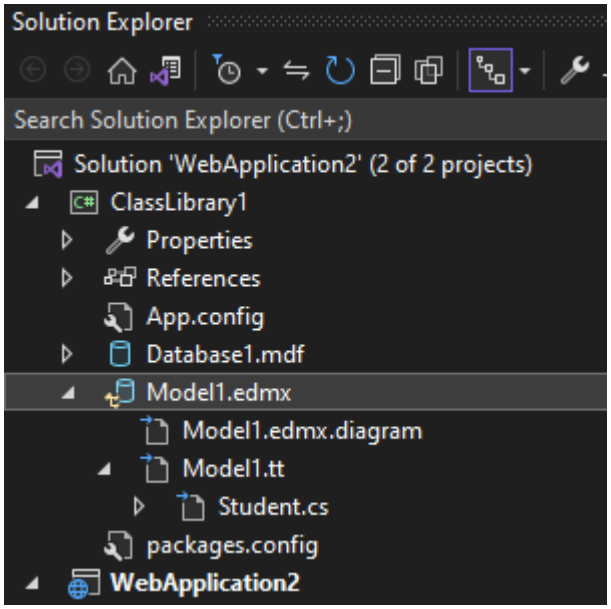
Paste them in your ApplicationDbContext.cs file under the constructor



If there are any auto generated relationships setup paste them here too

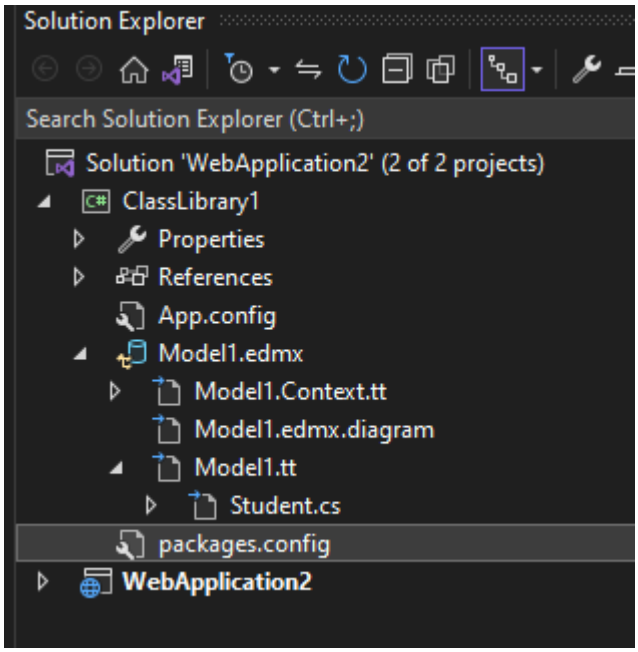
- Now delete your auto-generated Model.Context.cs class. We don’t need it anymore.

Your Model.edmx file now only needs these files



Delete the Database mdf file in the class library, we already have it in the ASP.NET Core Web App

Your Model.edmx file should look like this



4.2. Adding the Identity User table Foreign Key to other tables

Then Add this line of code to add the Id Foreign key to the tables where you need them (probably all the tables)

```
ClassLibrary1
//------------------------------------------------------------------------
// <auto-generated>
//     This code was generated from a template.
//
//     Manual changes to this file may cause unexpected behavior in y
//     Manual changes to this file will be overwritten if the code is
// </auto-generated>
//------------------------------------------------------------------------

namespace ClassLibrary1
{
    using System;
    using System.Collections.Generic;

    2 references
    public partial class Student
    {
        0 references
        public int StudentId { get; set; }
        0 references
        public string StudentName { get; set; }

        // Foreign key to reference the Identity User Id
        0 references
        public string UserId { get; set; }
    }
}
```

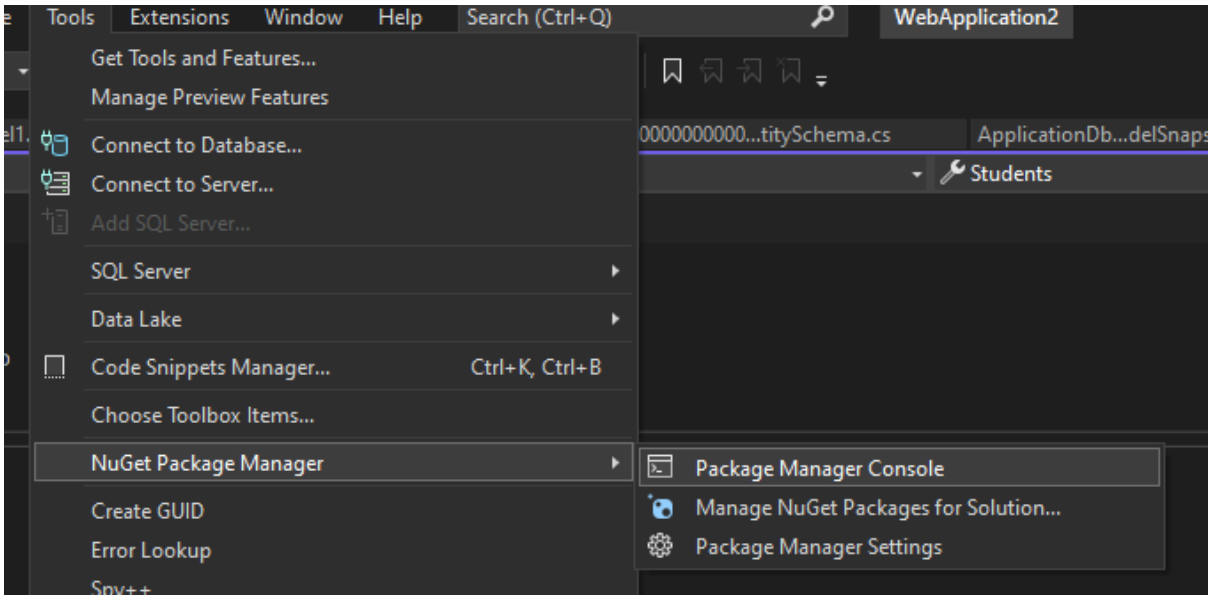
Then Add this code to the ApplicationDbContext,

You might need to add the modelbuilder.Entity for multiple tables whichever table you have that's not the User table

Now would be a great time to save all your work

In order to apply the migrations and update the database we need to make use of the Package Manager Console

- Go to Tools -> NuGet Package Manager -> Package Manager Console



Enter the command "**EntityFrameworkCore\Add-Migration"** or "**EntityFrameworkCore\add-migration**" followed by a space and a suitable migration name and hit Enter

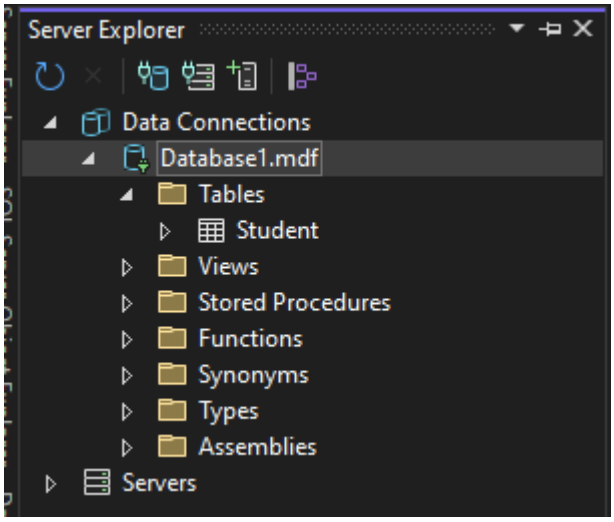
Open Package Manager Console and Add the new Migration

```
PM> EntityFrameworkCore\Add-Migration NewMigration
Both Entity Framework Core and Entity Framework 6 are
Build started...
Build succeeded.
To undo this action, use Remove-Migration.
```

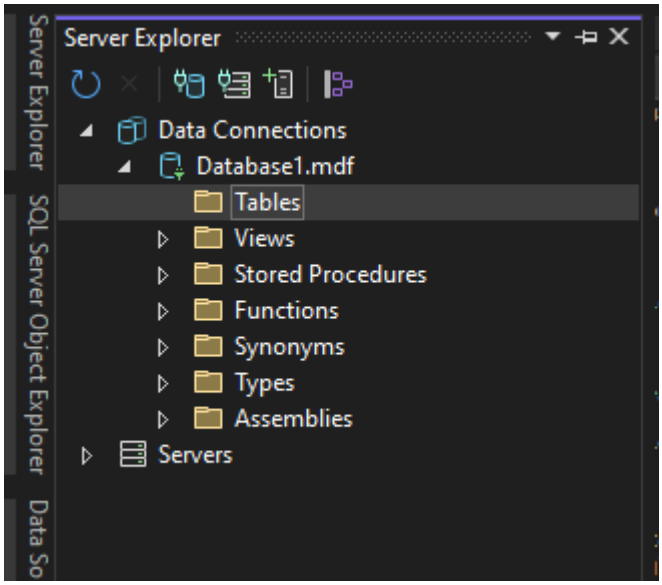
Here is the New Migration

```
WebApplication1                                          WebApplication1.D
1   using Microsoft.EntityFrameworkCore.Migrations;
2
3   #nullable disable
4
5   namespace WebApplication1.Data.Migrations
6   {
7       /// <inheritdoc />
        1 reference
8       public partial class NewMigration : Migration
9       {
10          /// <inheritdoc />
            0 references
11          protected override void Up(MigrationBuilder migrationBuilder)
12          {
13              migrationBuilder.CreateTable(
14                  name: "Students",
15                  columns: table => new
16                  {
17                      StudentId = table.Column<int>(type: "int", nullable: false)
18                          .Annotation("SqlServer:Identity", "1, 1"),
19                      StudentName = table.Column<string>(type: "nvarchar(max)", nullable: true),
20                      UserId = table.Column<string>(type: "nvarchar(max)", nullable: true)
21                  },
22                  constraints: table =>
23                  {
24                      table.PrimaryKey("PK_Students", x => x.StudentId);
25                  });
26          }
27
28          /// <inheritdoc />
            0 references
29          protected override void Down(MigrationBuilder migrationBuilder)
30          {
31              migrationBuilder.DropTable(
32                  name: "Students");
33          }
34      }
35  }
```

Delete the Student table from the database we are going to create it again



This is what our Database looks like before we update it in the console



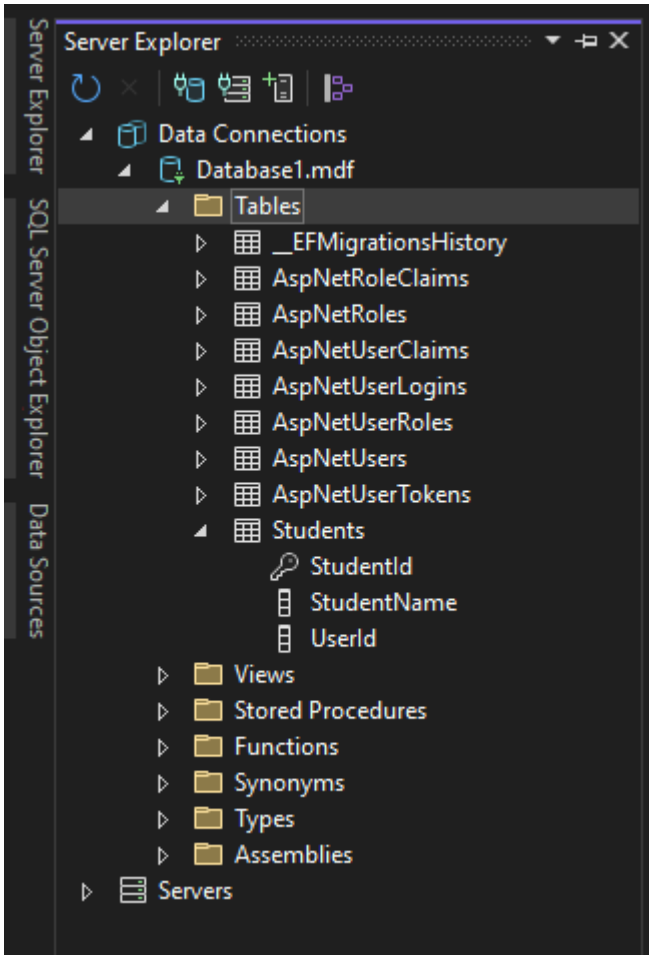
Now lets run the Update database command

```
PM> EntityFrameworkCore\update-database
```

```
Package source: All                    Default project: WebApplication1
        [StudentId] int NOT NULL IDENTITY,
        [StudentName] nvarchar(max) NULL,
        [UserId] nvarchar(max) NULL,
        CONSTRAINT [PK_Students] PRIMARY KEY ([StudentId])
    );
Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (0ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
      INSERT INTO [__EFMigrationsHistory] ([MigrationId], [ProductVersion])
      VALUES (N'20231126180019_Initial-Migration', N'7.0.2');
Done.
PM>
```

Its finished

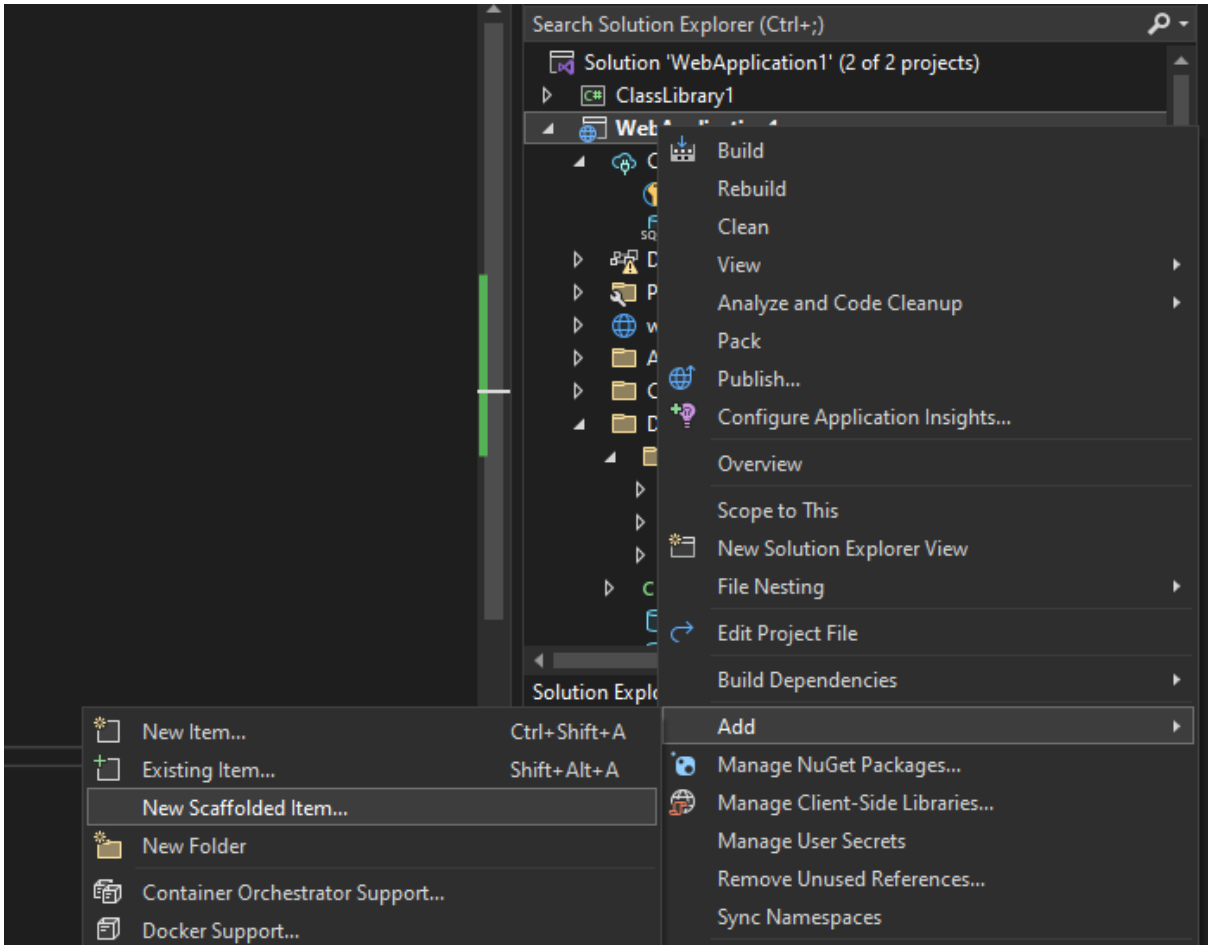
Now the database should look something like this



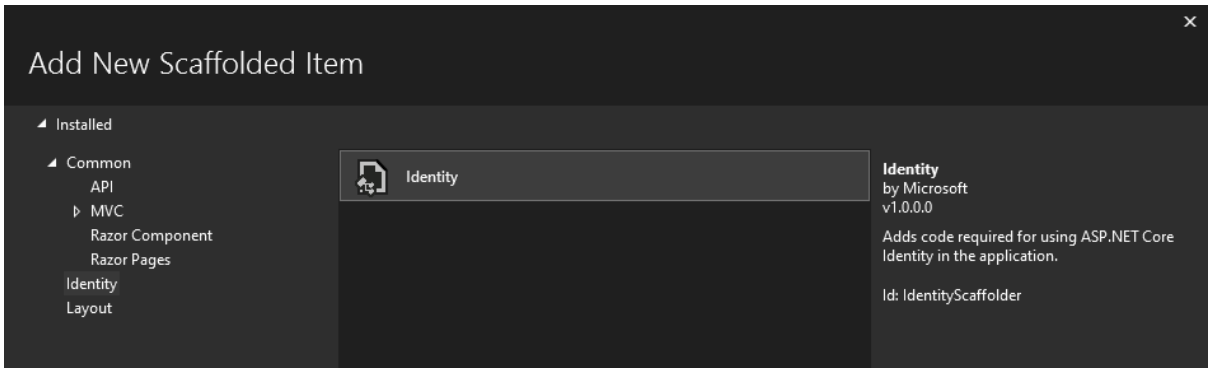
Scaffolding Identity Pages

Lets add some Login, Register and Logout functionality to our app using Identity

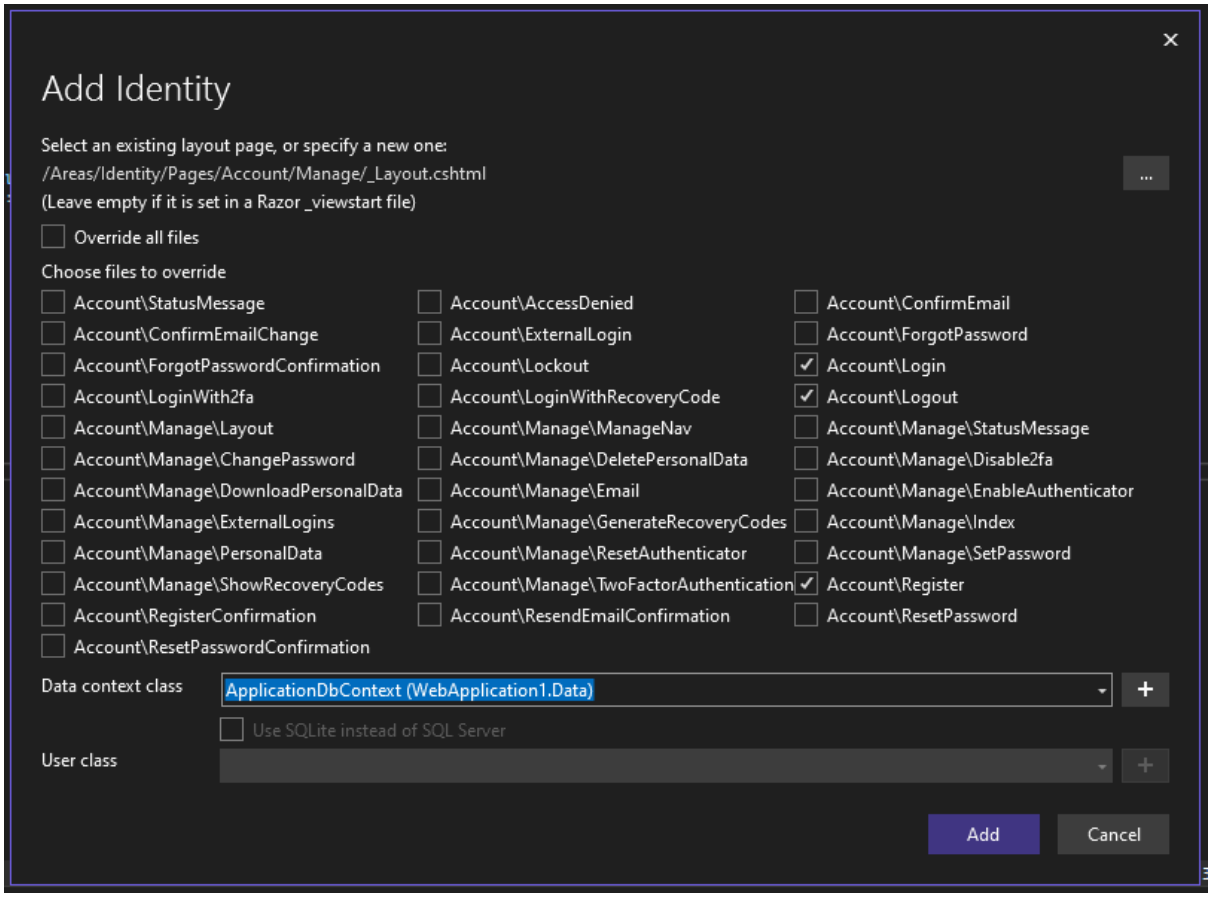
Right click on the project -> Add -> New Scaffolded Item



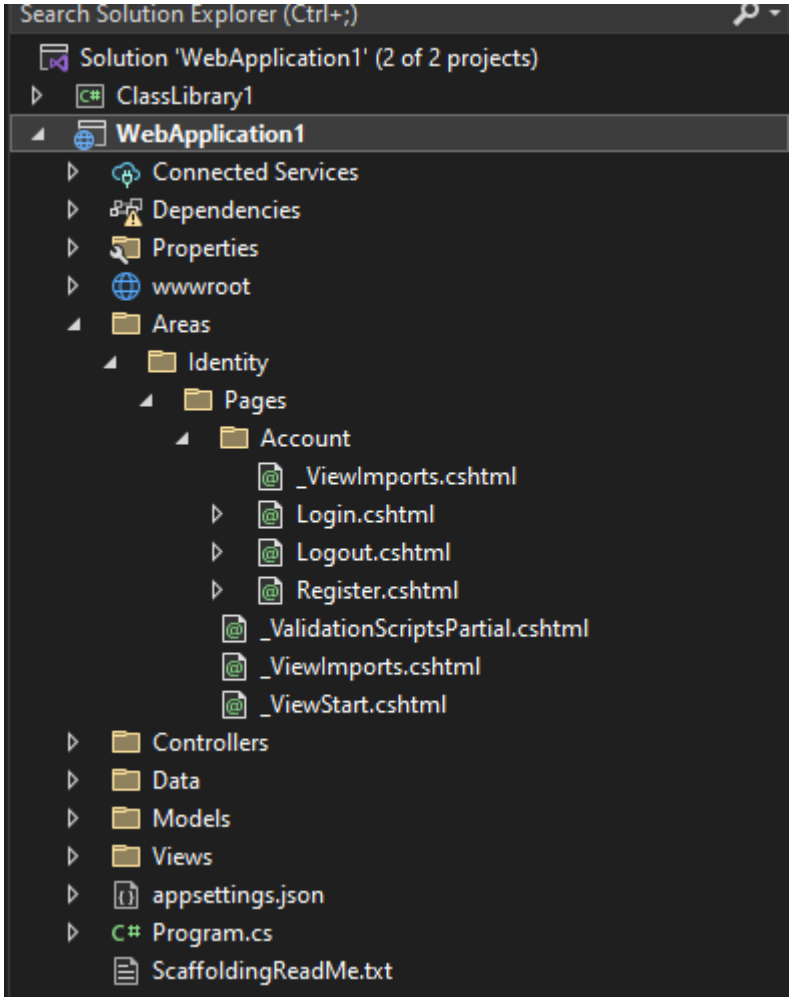
- Choose Identity and click Add



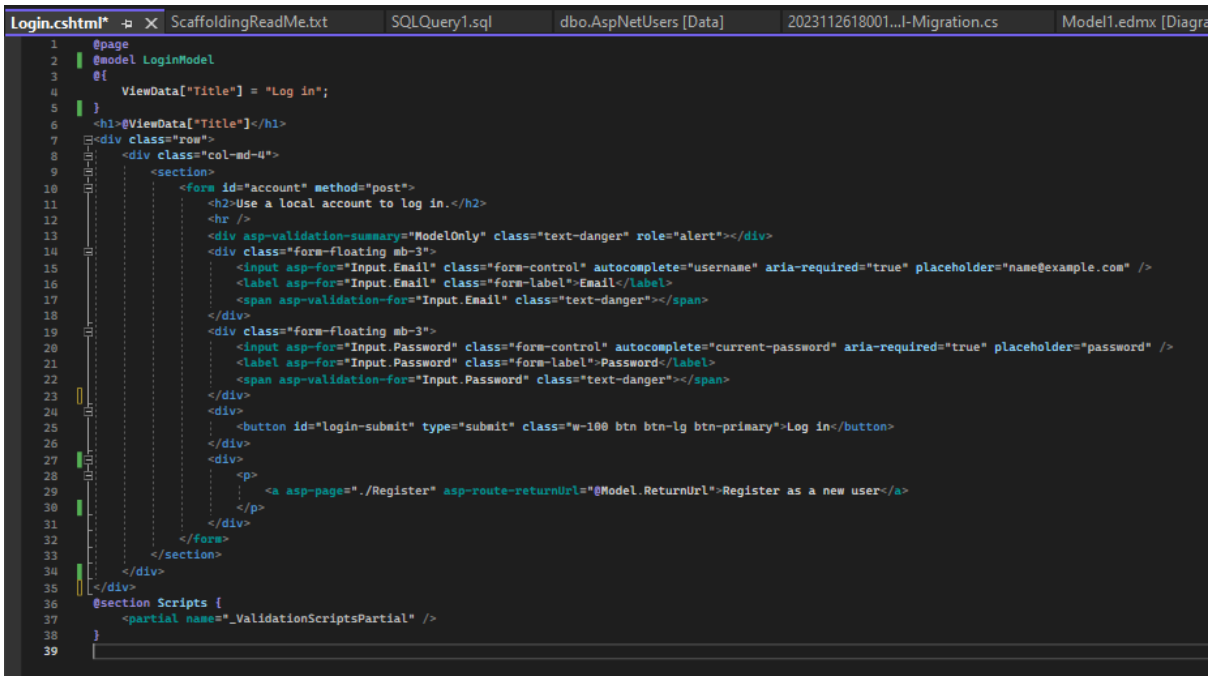
- Select only Account\Login, Account\Register and Account\Logout
- Select the ApplicationDbContext from the dropdown
- Click Add



- Your Areas Folder should now be update with the added Identity Items



- Edit the login cshtml page (optional) to remove any call to actions we don't need



I'd do the same for the Register cshtml (Optional)

Register.cshtml* | Logout.cshtml | Login.cshtml* | ScaffoldingReadMe.txt | SQLQuery1.sql | dbo.AspNetUsers [Data] | 202311

```
@page
@model RegisterModel
@{
    ViewData["Title"] = "Register";
}

<h1>@ViewData["Title"]</h1>

<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.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>
                <span asp-validation-for="Input.Password" class="text-danger"></span>
            </div>
            <div class="form-floating mb-3">
                <input asp-for="Input.ConfirmPassword" class="form-control" autocomplete="new-password" aria-required="true" placeholder="password" />
                <label asp-for="Input.ConfirmPassword">Confirm Password</label>
                <span asp-validation-for="Input.ConfirmPassword" class="text-danger"></span>
            </div>
            <button id="registerSubmit" type="submit" class="w-100 btn btn-lg btn-primary">Register</button>
        </form>
    </div>
</div>
@section Scripts {
    <partial name="_ValidationScriptsPartial" />
```

Edit the Login.cshtml.cs like this

```
public string ErrorMessage { get; set; }
1 reference
public class InputModel
{
    [Required]
    4 references
    public string UserName { get; set; }

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

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

```
0 references
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.IsLockedOut)
        {
            _logger.LogWarning("User account locked out.");
            return RedirectToPage("./Lockout");
        }
        else
        {
            ModelState.AddModelError(string.Empty, "Invalid login attempt.");
            return Page();
        }
    }
    // If we got this far, something failed, redisplay form
    return Page();
```

I'd Modify the Register.cshtml.cs file like this because we don't need any email confirmation

```
1 reference
public class InputModel
{
    [Required]
    [StringLength(20, ErrorMessage = "The {0} must be at least {2} and at max {1} characters long.", MinimumLength = 6)]
    [Display(Name = "Username")]
    4 references
    public string UserName { get; set; }

    [Required]
    [StringLength(12, ErrorMessage = "The {0} must be at least {2} and at max {1} characters long.", MinimumLength = 8)]
    [DataType(DataType.Password)]
    [Display(Name = "Password")]
    4 references
    public string Password { get; set; }

    [DataType(DataType.Password)]
    [Display(Name = "Confirm password")]
    [Compare("Password", ErrorMessage = "The password and confirmation password do not match.")]
    3 references
    public string ConfirmPassword { get; set; }
}
```

```
0 references
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);
        var result = await _userManager.CreateAsync(user, Input.Password);

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

                await _signInManager.SignInAsync(user, isPersistent: false);
                return LocalRedirect(returnUrl);

        }
        foreach (var error in result.Errors)
        {
            ModelState.AddModelError(string.Empty, error.Description);
        }
    }
    return Page();
}
```

- Go to Program.cs and set SignInRequireConfiremedAccount from true to false

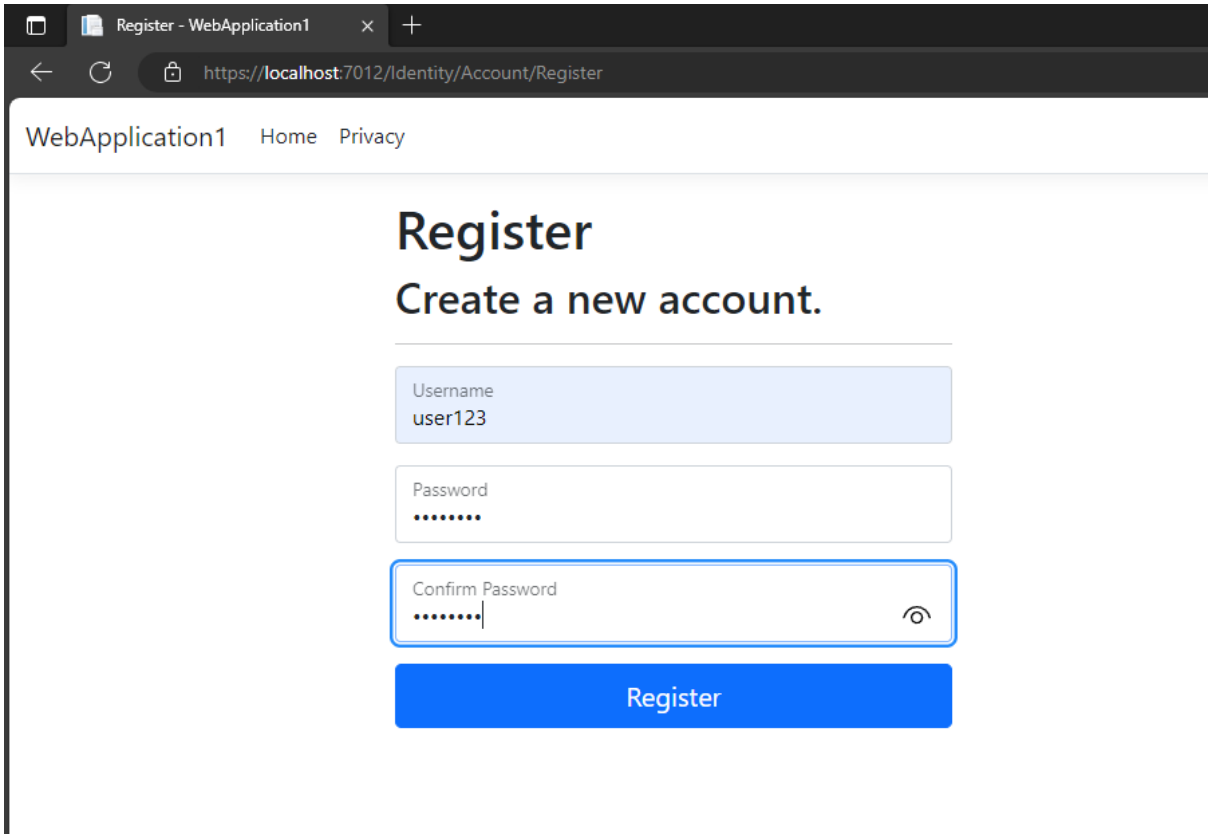
```
// Add services to the container.
var connectionString = builder.Configuration.GetConnectionString("DefaultConnection") ?? throw new InvalidOperationException("Connecti
builder.Services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(connectionString));
builder.Services.AddDatabaseDeveloperPageExceptionFilter();

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

var app = builder.Build();
```

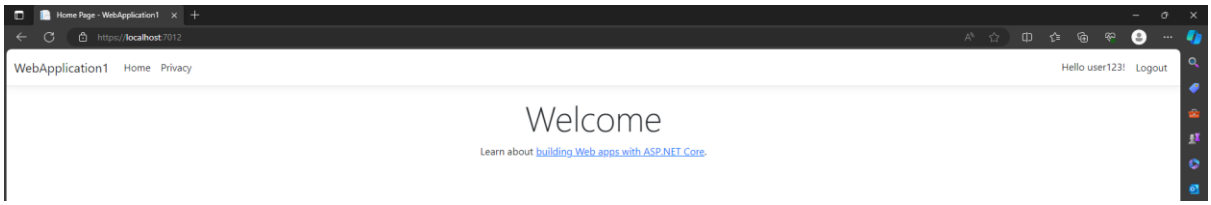
- Now go back to the Login.cshtml and change the email form controls to UserName

```
<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="User1" />
                    <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>
                    <button id="login-submit" type="submit" class="w-100 btn btn-lg btn-primary">Log in</button>
                </div>
                <div>
                    <p>
                        <a asp-page="./Register" asp-route-returnUrl="@Model.ReturnUrl">Register as a new user</a>
                    </p>
                </div>
            </form>
        </section>
    </div>
</div>
```

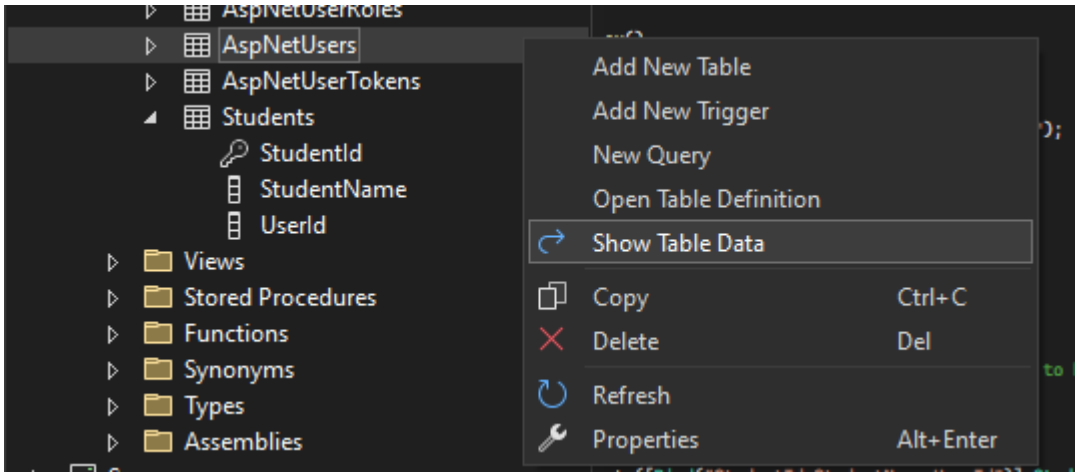
- Save all and run the application

Testing the Register and Login

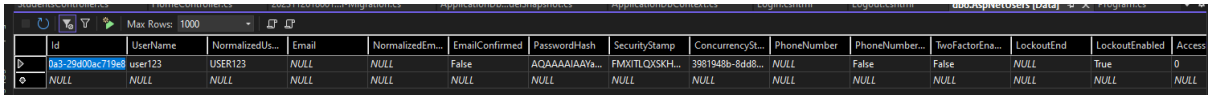




Let's check the database

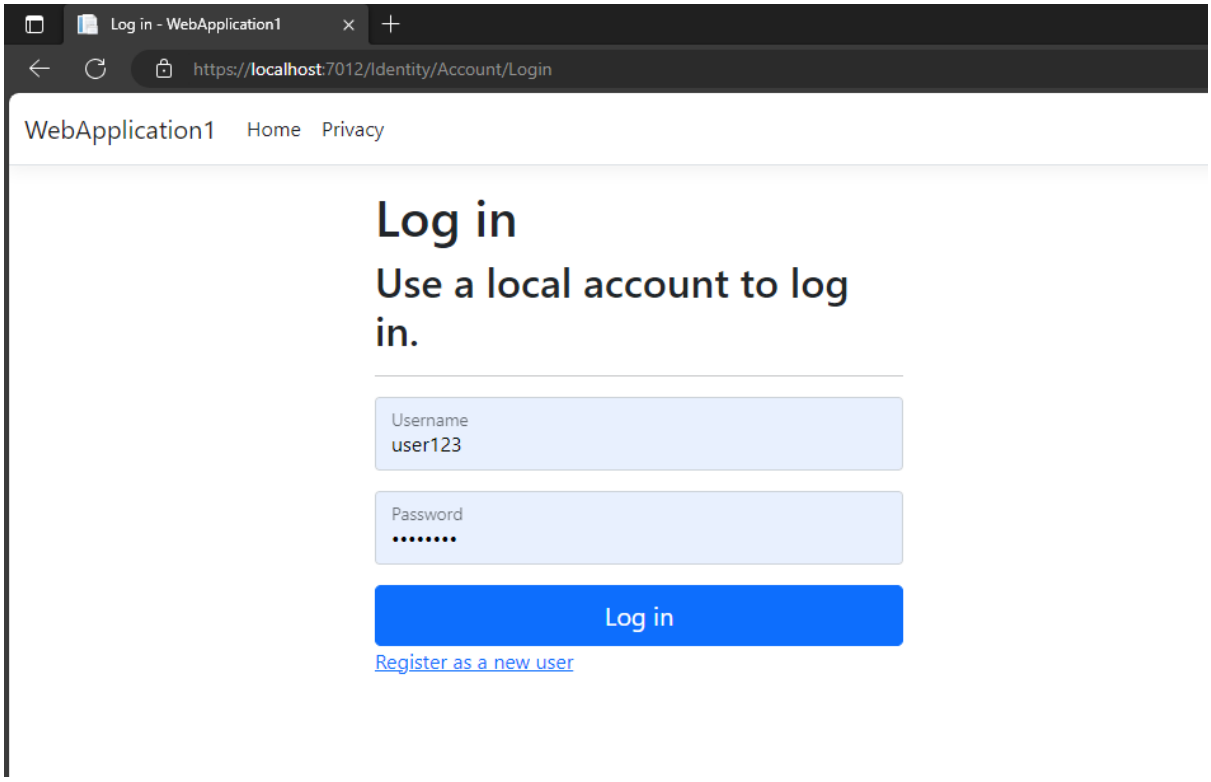


Hit the blue refresh icon to see the added user



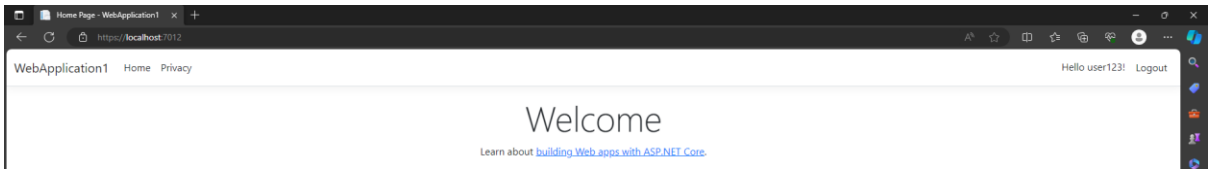
Nice. If you got this far then your app is setup correctly and the rest of the app should be fairly easy

Let's log out and login using user123's account we just created



The login works too with no errors

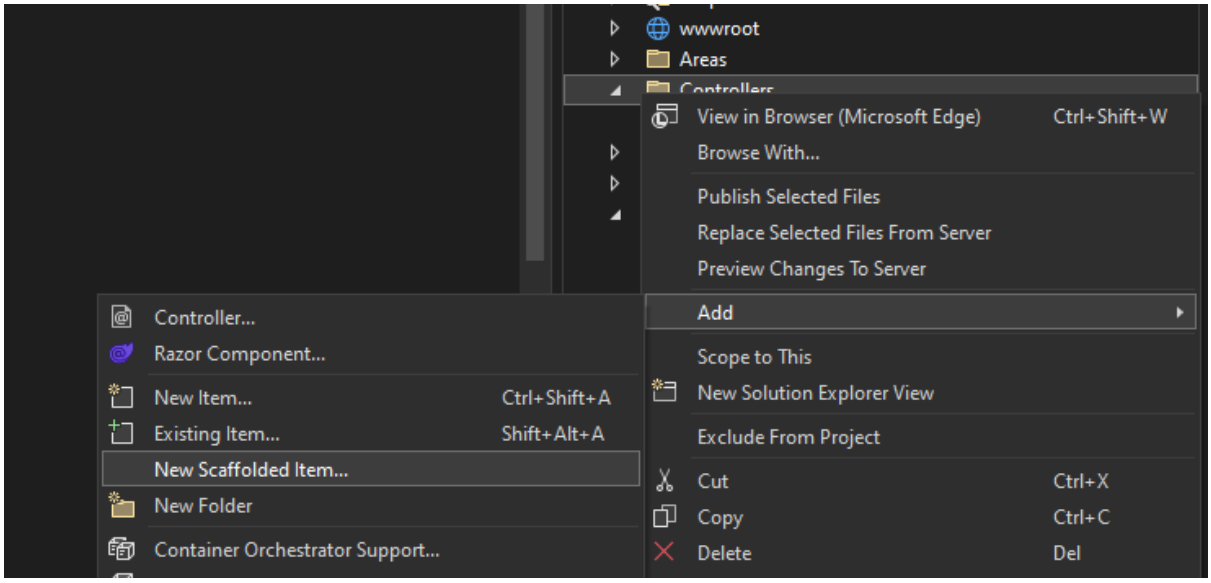
The User logs in automatically and their Username is displayed in the Hello message instead of their email



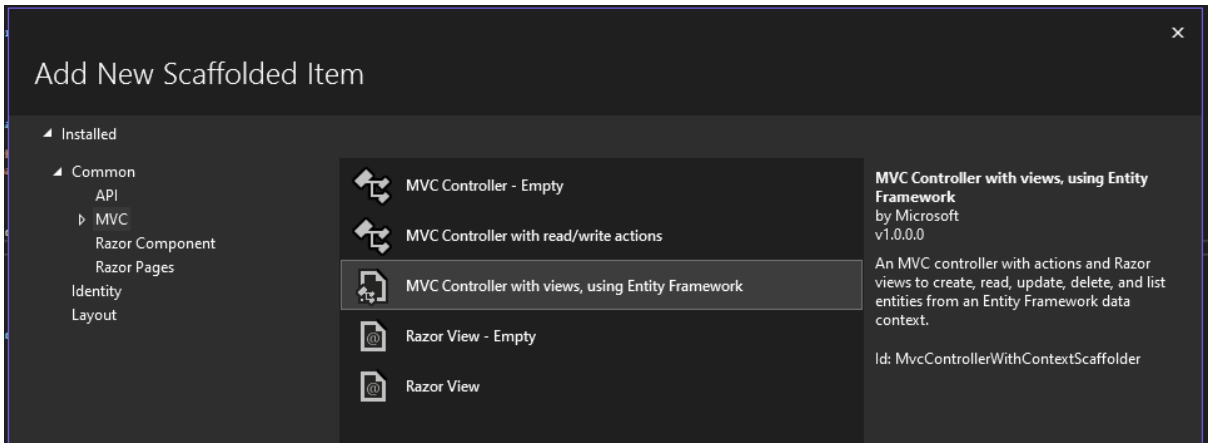
Now lets scaffold a controller and views for Adding a student.

Scaffolding a Model Controller and Views with EF

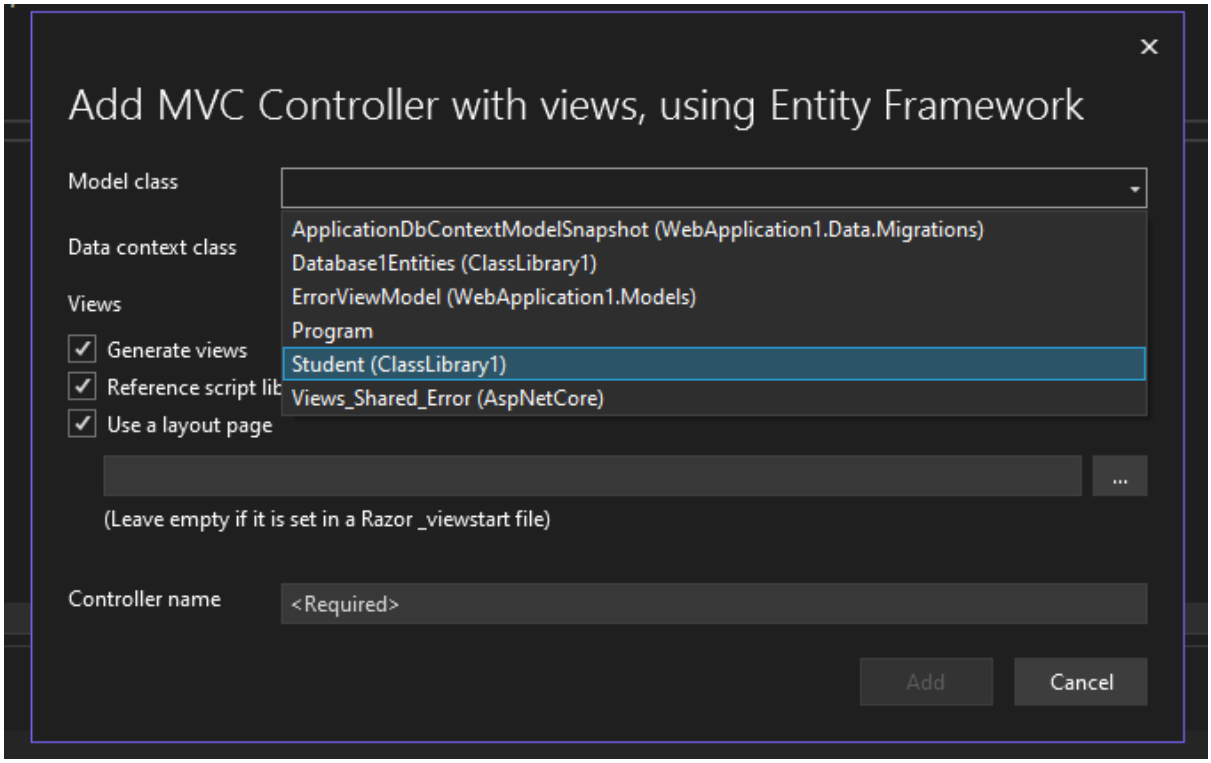
- Right click on the Controllers Folder -> Add ->New Scaffolded Item



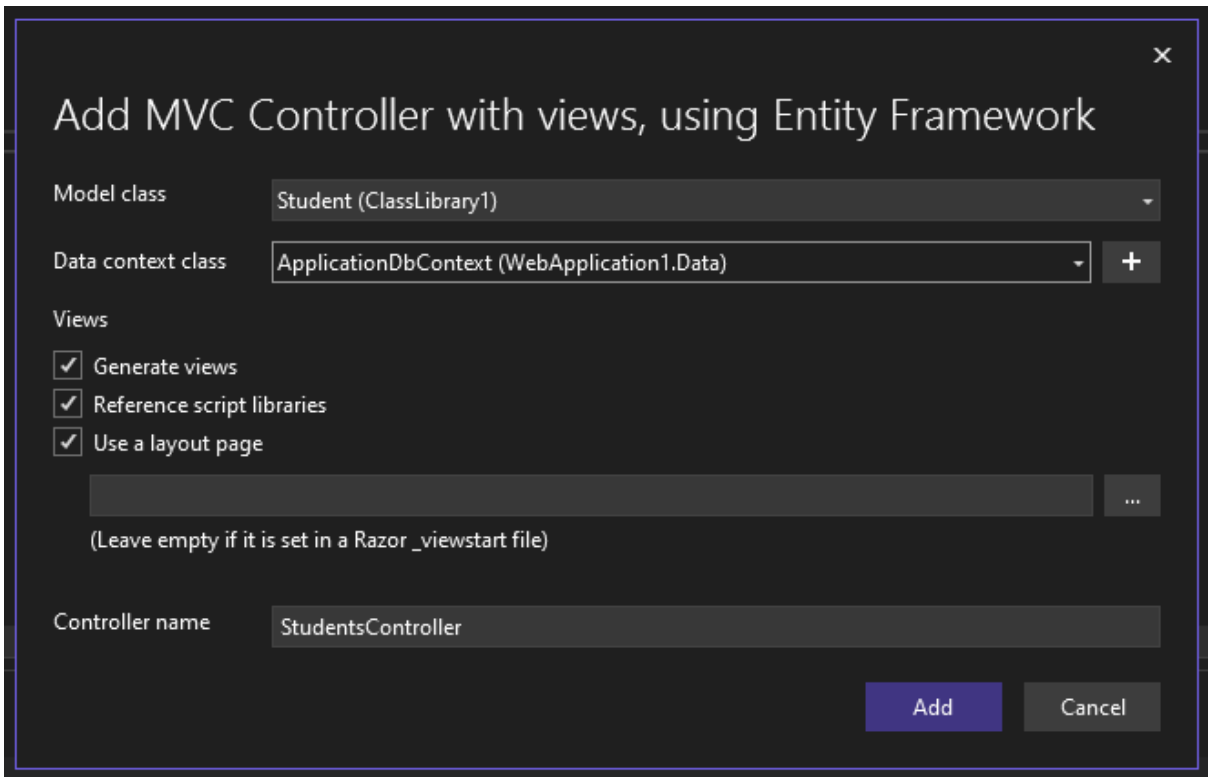
- Select under Common -> MVC -> MVC Controller with views, using Entity Framework



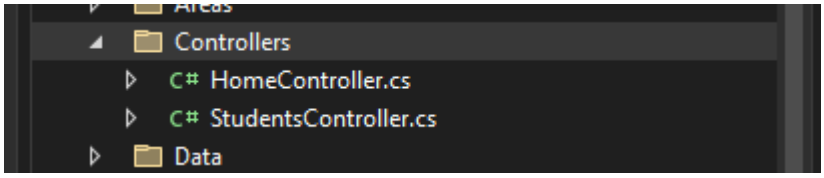
- Choose one of your Model Classes in your class library, in this example we are going to store Student data



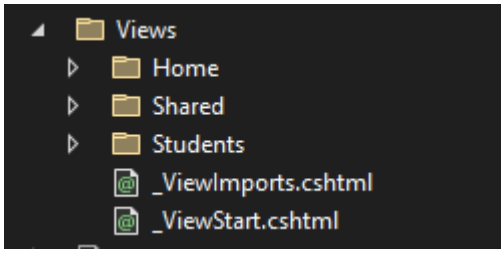
- Select the default ApplicationDbContext



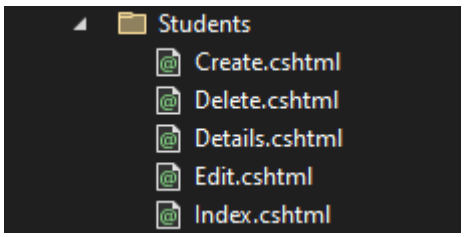
- Click Add
- The Students Controller is now added



Let's check the Views



There is a View folder for Students added



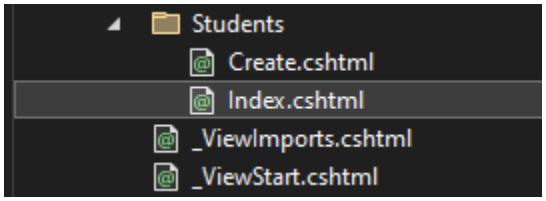
In the Students folder there are 4 default pages, Create, Delete, Details, Edit and Index

For your project you only need Index and Create.

Why?

Because it's a study tracking app. The purpose of the app is to store and view data, so you only need the Index and Create. Maybe Details if you want a better view but its not required.

So lets delete the other razor pages that's not required



Alright now let's remove the action logic in the controller for the pages we removed

Here’s the updated Students controller

```
WebApplication1                                                    WebApplication1.C

namespace WebApplication1.Controllers
{
    1 reference
    public class StudentsController : Controller
    {
        private readonly ApplicationDbContext _context;

        0 references
        public StudentsController(ApplicationDbContext context)
        {
            _context = context;
        }

        // GET: Students
        1 reference
        public async Task<IActionResult> Index()
        {
            return _context.Students != null ?
                        View(await _context.Students.ToListAsync()) :
                        Problem("Entity set 'ApplicationDbContext.Students'  is null.");
        }

        // GET: Students/Create
        0 references
        public IActionResult Create()
        {
            return View();
        }

        // POST: Students/Create
        // To protect from overposting attacks, enable the specific properties you want to bind to.
        // For more details, see http://go.microsoft.com/fwlink/?LinkId=317598.
        [HttpPost]
        [ValidateAntiForgeryToken]
        0 references
        public async Task<IActionResult> Create([Bind("StudentId,StudentName,UserId")] Student student)
        {
            if (ModelState.IsValid)
            {
                _context.Add(student);
                await _context.SaveChangesAsync();
                return RedirectToAction(nameof(Index));
            }
            return View(student);
        }
    }
```

Now lets modify the controller to save and display to and for the specific signed in user

StudentsController.cs | HomeController.cs | 2023112618001...I-Migration.cs | ApplicationDb...delSnapshot.cs

WebApplication1 | WebApplication1.Controllers.Stude

```
using System;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Rendering;
using Microsoft.EntityFrameworkCore;
using Microsoft.AspNetCore.Identity; // Add this namespace for Identity related operations
using ClassLibrary1;
using WebApplication1.Data;

namespace WebApplication1.Controllers
{
    1 reference
    public class StudentsController : Controller
    {
        private readonly ApplicationDbContext _context;
        private readonly UserManager<IdentityUser> _userManager; // Add UserManager

        0 references
        public StudentsController(ApplicationDbContext context, UserManager<IdentityUser> userManager)
        {
            _context = context;
            _userManager = userManager;
        }

        // GET: Students
        1 reference
        public async Task<IActionResult> Index()
        {
            // Get the currently signed-in user's identity
            var currentUser = await _userManager.GetUserAsync(User);

            if (currentUser != null)
            {
                // Filter students by UserId (assuming UserId in Student refers to the Identity User's Id)
                var students = await _context.Students.Where(s => s.UserId == currentUser.Id).ToListAsync();
                return View(students);
            }
            else
            {
                return NotFound(); // Or handle accordingly if user not found
            }
        }
```

```
        // GET: Students/Create
        0 references
        public IActionResult Create()
        {
            return View();
        }

        // POST: Students/Create
        [HttpPost]
        [ValidateAntiForgeryToken]
        0 references
        public async Task<IActionResult> Create([Bind("UserId,StudentId,StudentName")] Student student)
        {
            if (ModelState.IsValid)
            {
                var currentUser = await _userManager.GetUserAsync(User);

                if (currentUser != null)
                {
                    student.UserId = currentUser.Id; // Set UserId to currently signed-in user's Id

                    _context.Add(student);
                    await _context.SaveChangesAsync();
                    return RedirectToAction(nameof(Index));
                }
            }
            return View(student);
        }
    }
}
```

Ok Now before we run it, we have to make the input for UserId Hidden in the Student create razor page

```
<h1>Create</h1>

<h4>Student</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form asp-action="Create">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <div class="form-group">
                <label asp-for="StudentName" class="control-label"></label>
                <input asp-for="StudentName" class="form-control" />
                <span asp-validation-for="StudentName" class="text-danger"></span>
            </div>
            <div class="form-group">
                <input asp-for="UserId" type="hidden" />
                <label asp-for="UserId" class="control-label" style="display: none;"></label>
            </div>
            <div class="form-group">
                <input type="submit" value="Create" class="btn btn-primary" />
            </div>
        </form>
    </div>
</div>
```

- Now lets modify the Index to take away actions we don't need and remove the UserId from being displayed in the table

```
Index.cshtml*    Create.cshtml*    StudentsController.cs
1   @model IEnumerable<ClassLibrary1.Student>
2
3   @{
4       ViewData["Title"] = "Index";
5   }
6
7   <h1>Index</h1>
8
9   <p>
10      <a asp-action="Create">Create New</a>
11  </p>
12  <table class="table">
13      <thead>
14          <tr>
15              <th>
16                  @Html.DisplayNameFor(model => model.StudentName)
17              </th>
18              <th></th>
19          </tr>
20      </thead>
21      <tbody>
22  @foreach (var item in Model) {
23          <tr>
24              <td>
25                  @Html.DisplayFor(modelItem => item.StudentName)
26              </td>
27          </tr>
28  }
29      </tbody>
30  </table>
31
```

- Delete the Privacy cshtml page and replace the privacy nav link for the Students Controller Index action

```
_Layout.cshtml   Index.cshtml   Index.cshtml   Create.cshtml   StudentsController.cs   HomeController.cs   20231126180011
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>@ViewData["Title"] - WebApplication1</title>
    <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.min.css" />
    <link rel="stylesheet" href="~/css/site.css" asp-append-version="true" />
    <link rel="stylesheet" href="~/WebApplication1.styles.css" asp-append-version="true" />
</head>
<body>
    <header>
        <nav class="navbar navbar-expand-sm navbar-toggleable-sm navbar-light bg-white border-bottom box-shadow mb-3">
            <div class="container-fluid">
                <a class="navbar-brand" asp-area="" asp-controller="Home" asp-action="Index">WebApplication1</a>
                <button class="navbar-toggler" type="button" data-bs-toggle="collapse" data-bs-target=".navbar-collapse" aria-controls="navbarSupportedContent"
                        aria-expanded="false" aria-label="Toggle navigation">
                    <span class="navbar-toggler-icon"></span>
                </button>
                <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="Students" asp-action="Index">Students</a>
                        </li>
                    </ul>
                    <partial name="_LoginPartial" />
                </div>
            </div>
        </nav>
    </header>
    <div class="container">
        <main role="main" class="pb-3">
            @RenderBody()
        </main>
    </div>
```

Now lets add logic to only show specific actions if the user is signed in

- Add the using for Identity and inject the SignInManager

```
_Layout.cshtml*   Index.cshtml   Index.cshtml   Create.cshtml
1  @using Microsoft.AspNetCore.Identity
2  @inject SignInManager<IdentityUser> SignInManager
3  <!DOCTYPE html>
4  <html lang="en">
5  <head>
6      <meta charset="utf-8" />
7      <meta name="viewport" content="width=device-width, initial-scale=1.0" />
8      <title>@ViewData["Title"] - WebApplication1</title>
9      <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.min.css" />
10     <link rel="stylesheet" href="~/css/site.css" asp-append-version="true" />
```

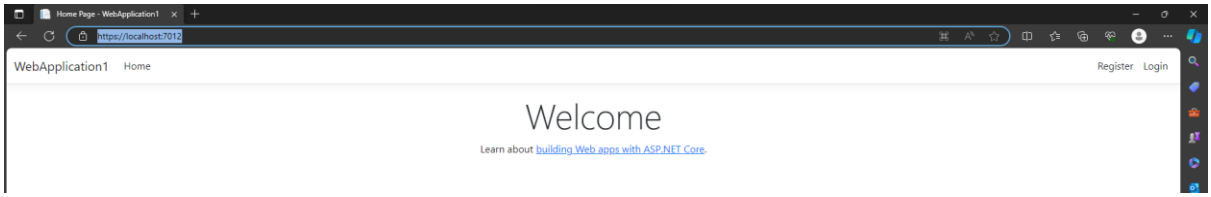
- Add the if statement in the nav to check if the user is signed in

```
<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>
        @if (SignInManager.IsSignedIn(User))
        {
            <li class="nav-item">
                <a class="nav-link text-dark" asp-area="" asp-controller="Students" asp-action="Index">Students</a>
            </li>
        }
    </ul>
    <partial name="_LoginPartial" />
```

Great! Everything is setup

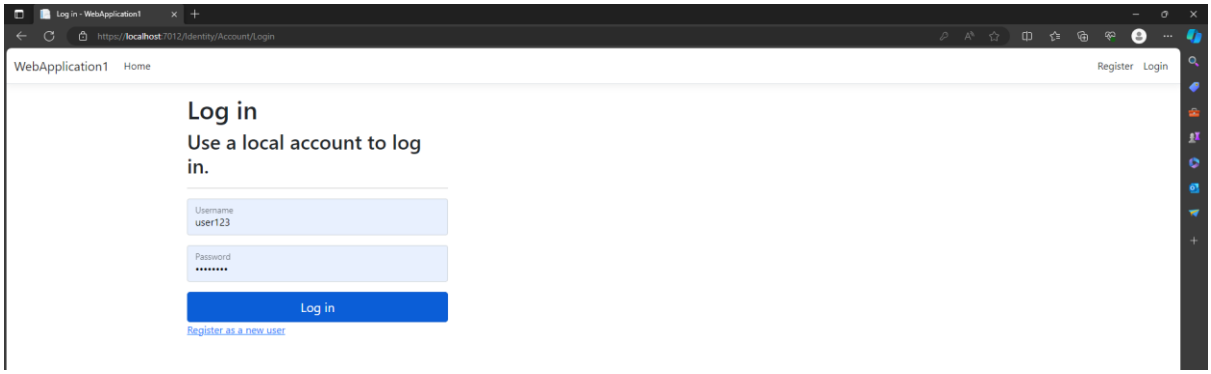
Now Lets run and our app to test if only the currently signed in users data is stored and displayed

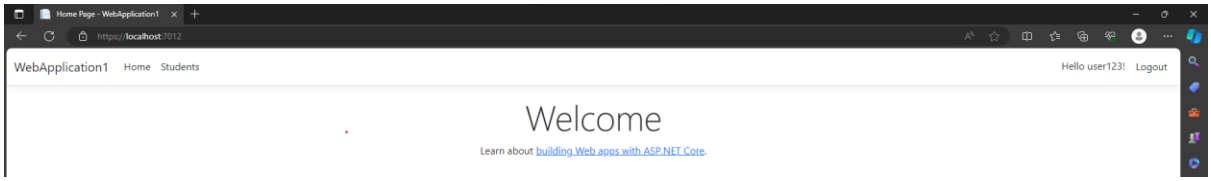
Testing the Scaffolded Controllers and Views



As you can see the Students Navigation item isn't yet being displayed

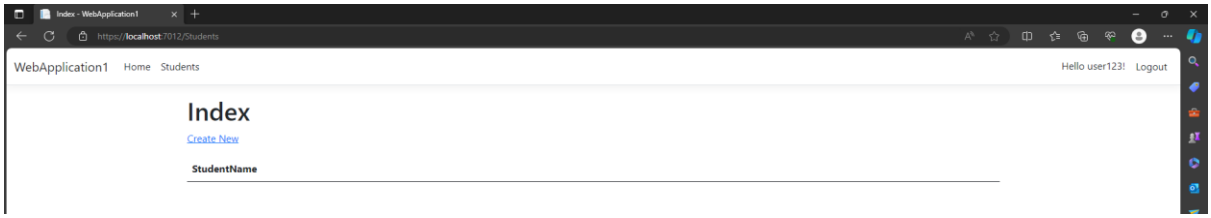
Let's login using the account we created earlier

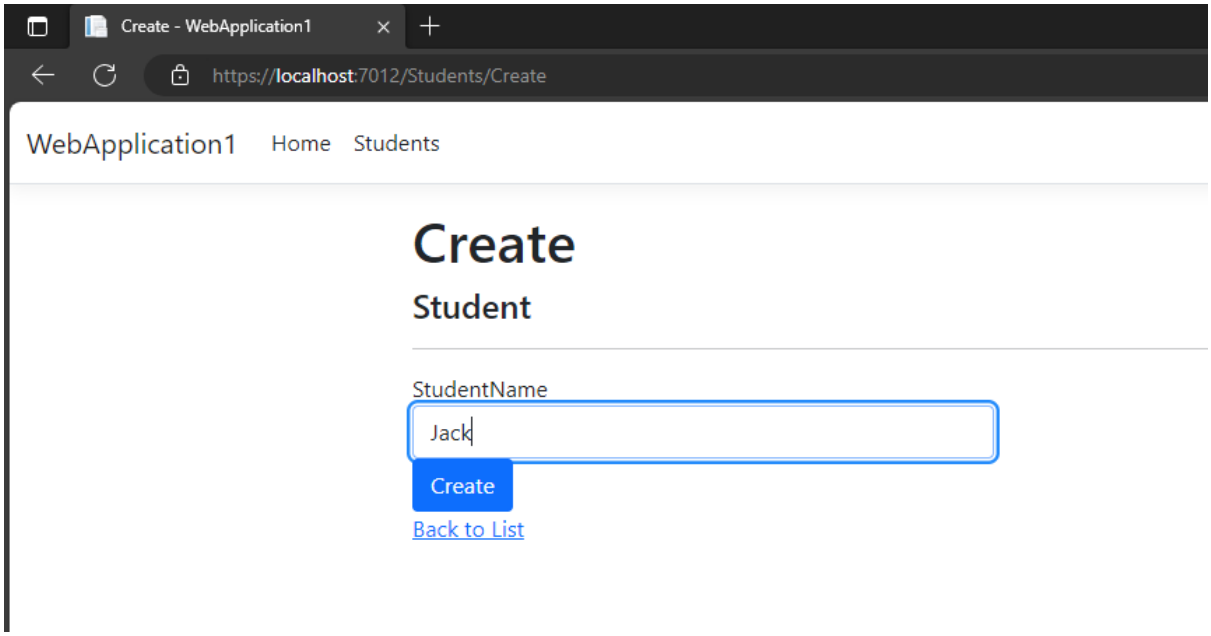




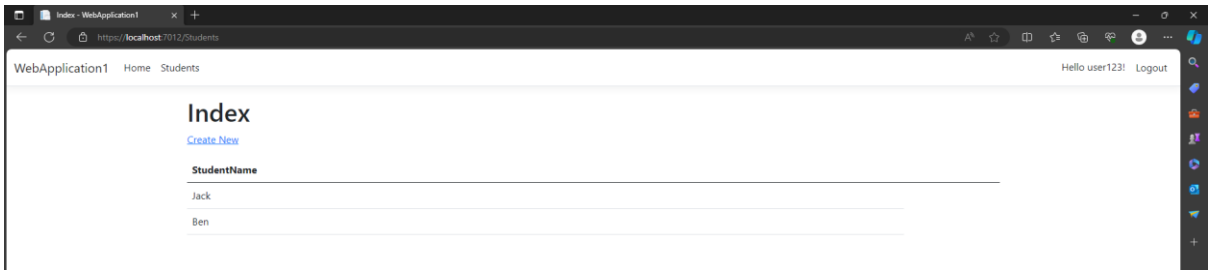
We can now see the Students menu item works as its intended. Use this for all your Navigation Items other than Home, Login, Register and Logout

If we click on Students, there are no students added yet so lets add some

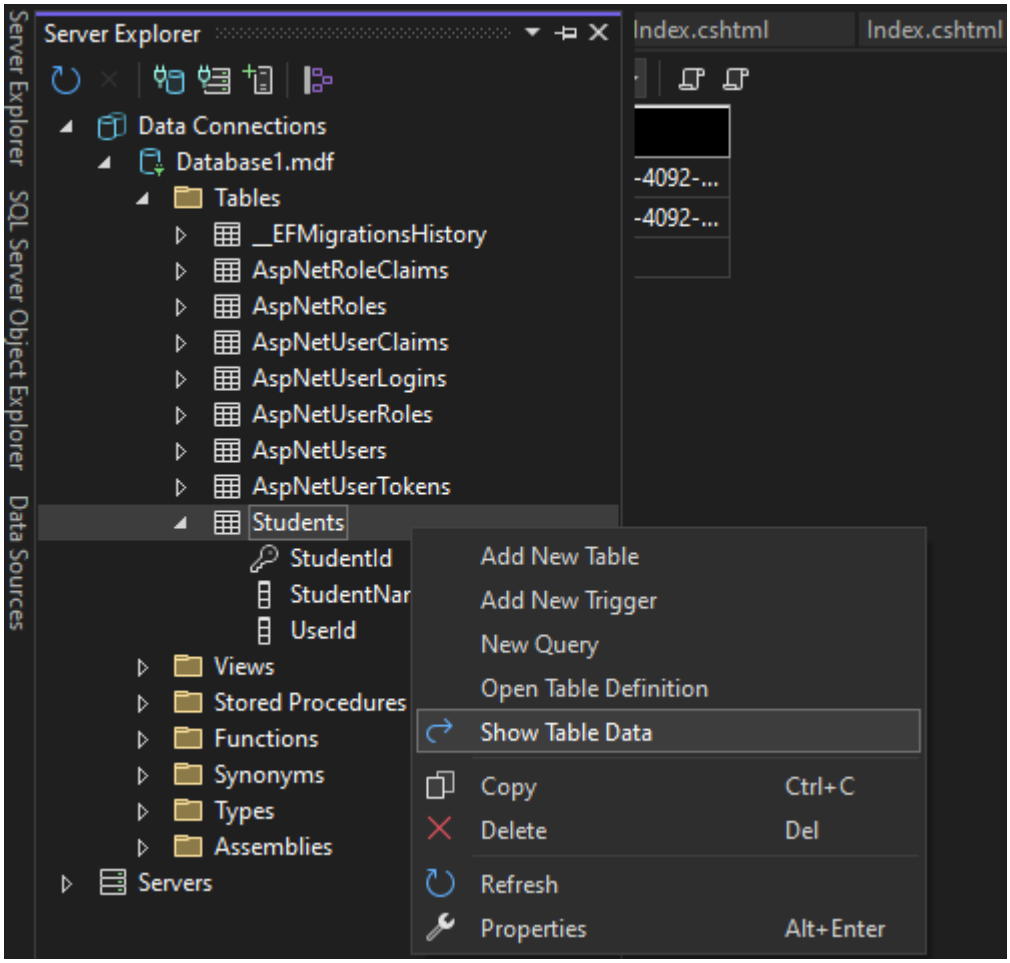


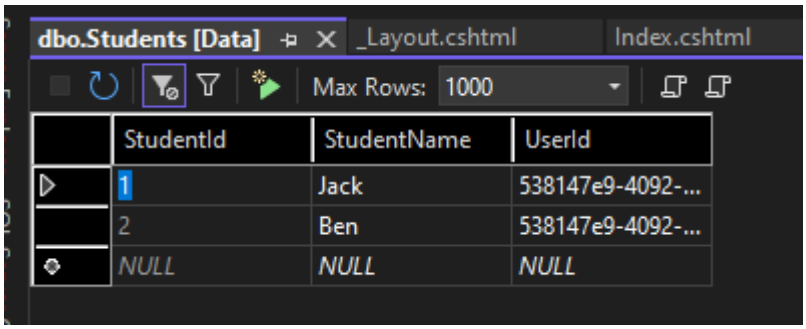


We've added Jack and Ben



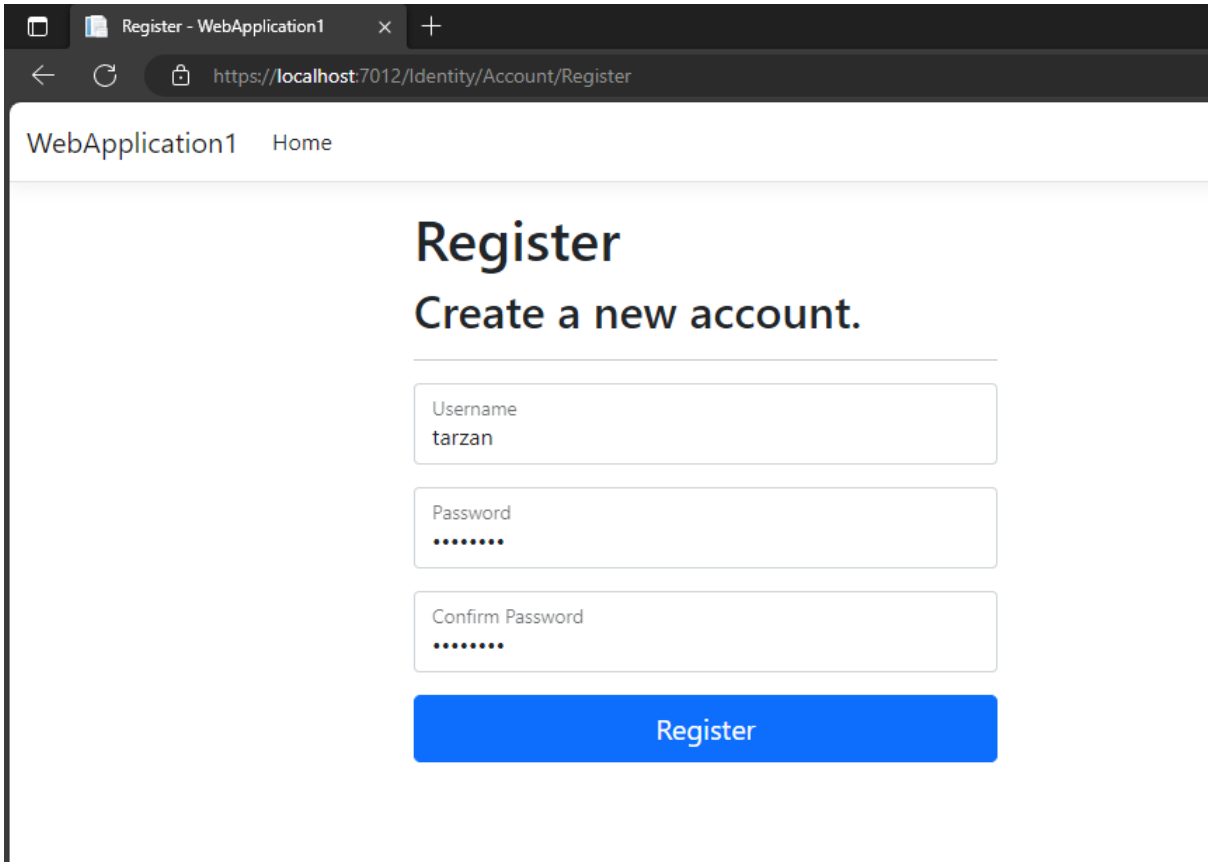
Now lets check the database to see if Jack and Ben are added to the correct user



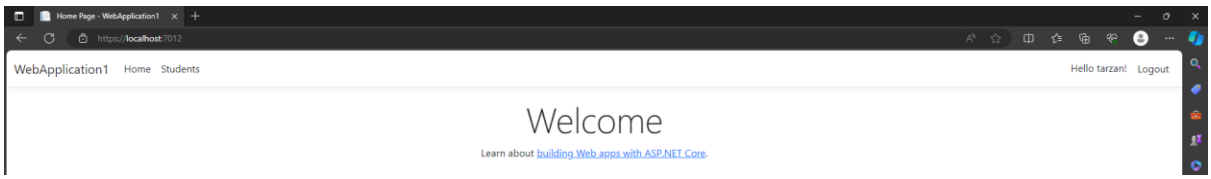


Cool. Now for one last check. Lets see if I create another account, whether user123's Students will also display there. We checking if the user specific data is implemented correctly

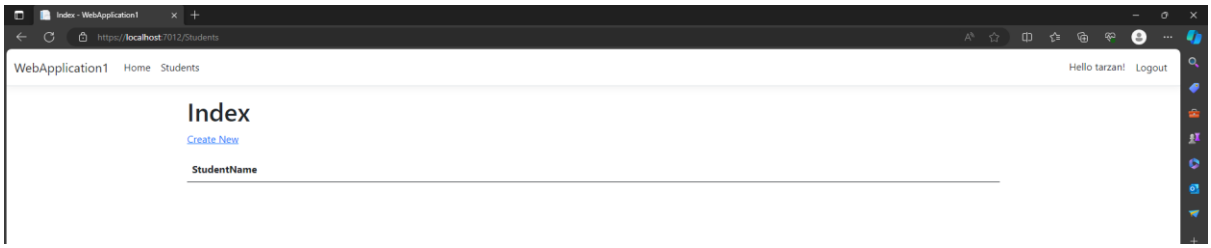
Lets register a new account with the username "tarzan"



Tarzan is now logged in

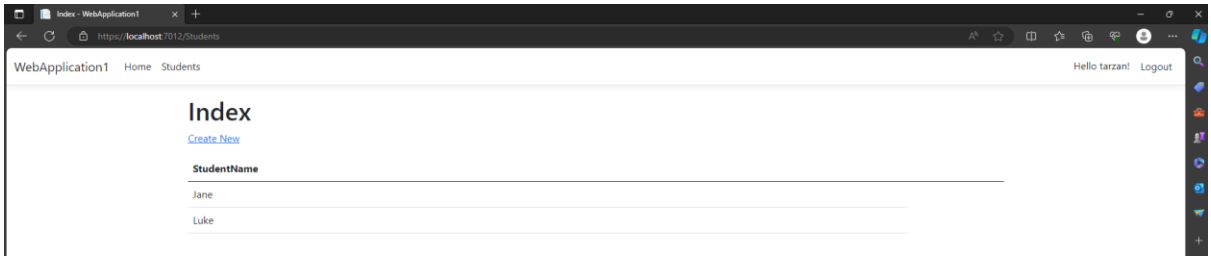


Lets check his students



As we can see tarzan does not have user123's Student data displaying. The data is user specific

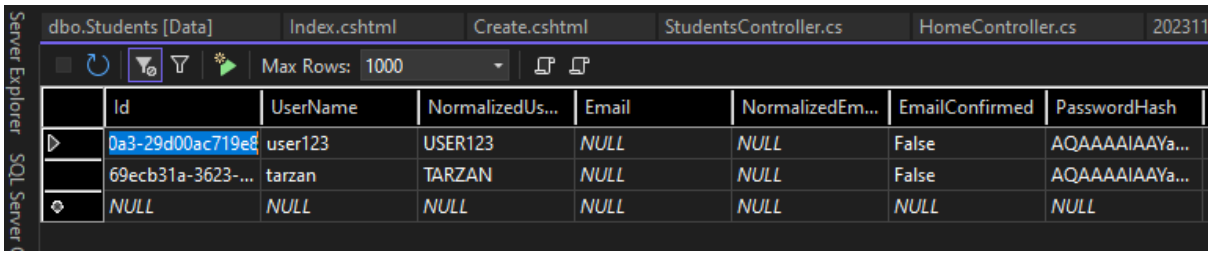
Let's create some students for tarzan. We added Jane and Luke



Now lets check the database again

To see if the data is reflecting properly

Here are the two users in the AspNetUsers table. Notice the difference in Id



Here’s the Student table