This tutorial features on step-by-step how to set up a DOTNET project as the Web API backend with example codes snippets, having Visual Studio installed is advised.

Part 1. Create new WebAPI project

Step 1. Create WebAPI project using the following command:

```
dotnet new webapi -n project-name
```

Step 2. Delete the weather forecast template files

Part 2. Configurations

Step 1. Configure the service container ('launchSettings.json' file):

1) open file 'launchSettings.json' inside the 'Properties' Folder

2) Modify the "applicationUrl" line to following:

```
"applicationUrl": "https://localhost:8080;http://localhost:8081",
```

Part 3. Entity Frame Work Setup

Step 1. Using the following command to install EntityFrameWork Tool on computer (Only need to do once per machine)

1) open cmd.exe in any directory

2) dotnet tool install –global dotnet-ef

3) dotnet tool update –global dotnet-ef

Step 2. Install 3 tools of EntityFrameWork:

1) Go to the project Folder (directory) in CMD

2) Install 3 packages for Database:

```
dotnet add package Microsoft.EntityFrameworkCore.Sqlite

dotnet add package Microsoft.EntityFrameworkCore.Tools

dotnet add package Microsoft.EntityFrameworkCore.Design
```

Part 4. DB Setup

Step 1. define the class/classes that represent data stored in the DB table/tables

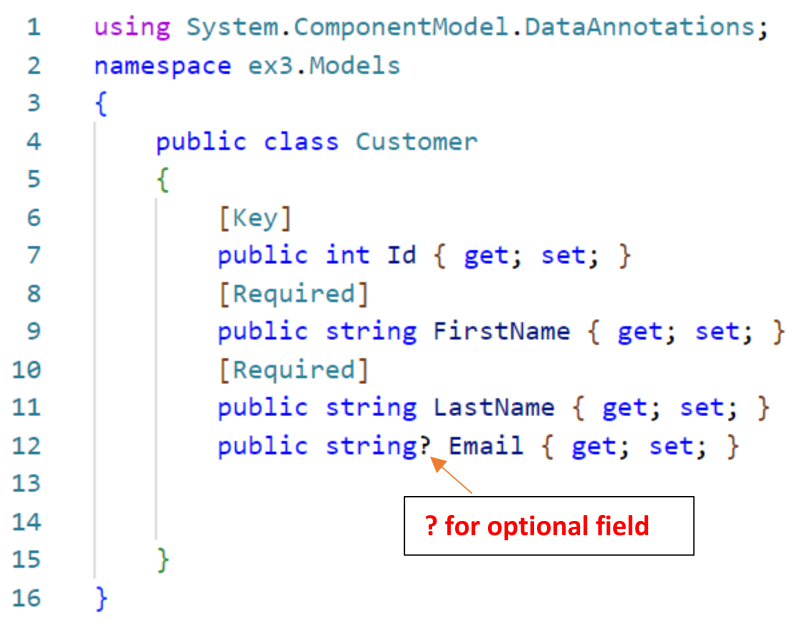
1) create a folder “Model” in our project

2) create a class “Customer.cs” in the **Model** folder

3) import annotation:

    using System.ComponentModel.DataAnnotations;

4) Make Annotations:



```
using System.ComponentModel.DataAnnotations;
namespace ex3.Models
{
    public class Customer
    {
        [Key]
        public int Id { get; set; }
        [Required]
        public string FirstName { get; set; }
        [Required]
        public string LastName { get; set; }
        public string? Email { get; set; }


    }
}
```

Step 2. define the class representing the DB (DBContext class)

1) create a folder “Data” in our project

2) create a new class WebAPIDBContext.cs

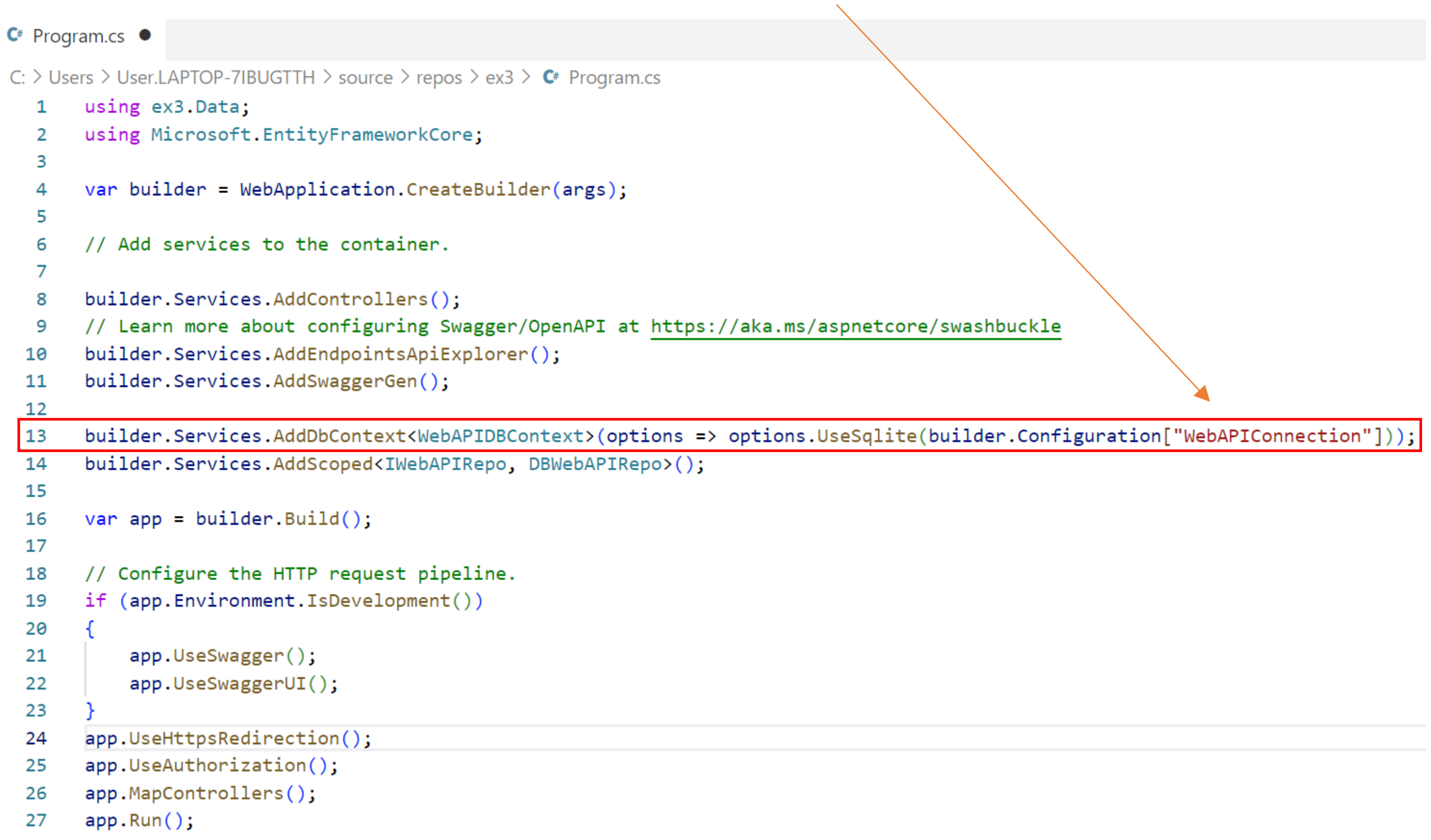
3) add constructor:

```
public WebAPIDBContext(DbContextOptions<WebAPIDBContext> options) : base(options) { }
```

4) add DbSet and complete the class:

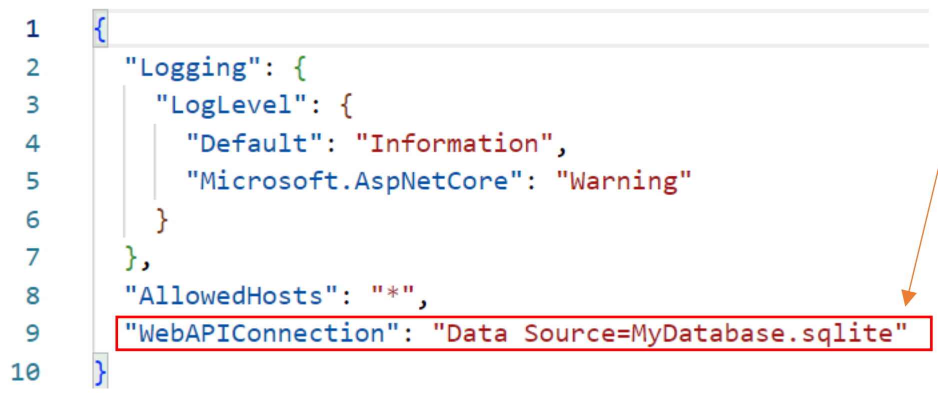
```
using Microsoft.EntityFrameworkCore;
using ex3.Models;

namespace ex3.Data
{
    public class WebAPIDBContext : DbContext
    {
        public WebAPIDBContext(DbContextOptions<WebAPIDBContext> options) : base(options) { }
        public DbSet<Customer> Customers { get; set; }


    }
}
```

5) Register this class with the service container (Main program)


Program.cs
C: > Users > User.LAPTOP-7IBUGTTH > source > repos > ex3 > Program.cs

```
using ex3.Data;
using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.

builder.Services.AddControllers();
// Learn more about configuring Swagger/OpenAPI at https://aka.ms/aspnetcore/swashbuckle
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();

builder.Services.AddDbContext<WebAPIDBContext>(options => options.UseSqlite(builder.Configuration["WebAPIConnection"]));
builder.Services.AddScoped<IWebAPIRepo, DBWebAPIRepo>();

var app = builder.Build();

// Configure the HTTP request pipeline.
if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();
}
app.UseHttpsRedirection();
app.UseAuthorization();
app.MapControllers();
app.Run();
```

6) Add Connection of database file in ‘appsettings.json’ file (from the root folder):



```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*",
  "WebAPIConnection": "Data Source=MyDatabase.sqlite"
}
```

Step 3. Tools to generate the database (can be skipped if supplied DB file)

1) open CMD to the project root folder

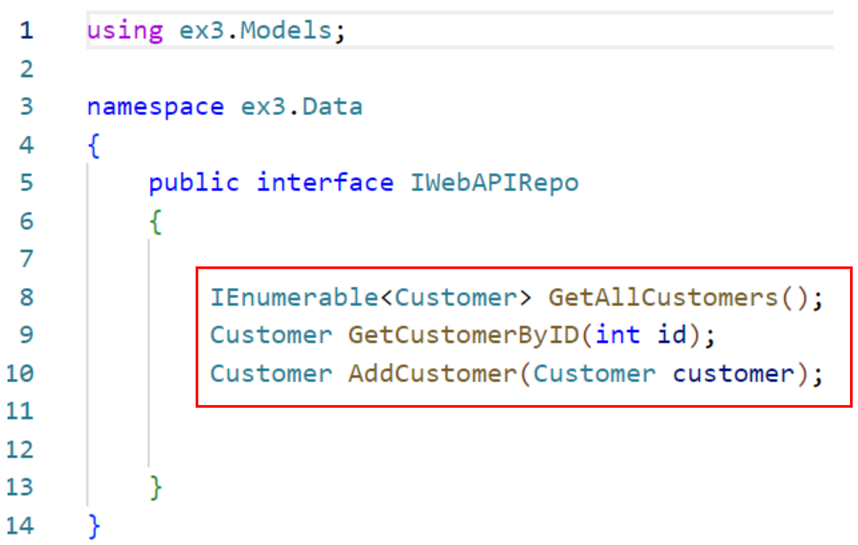
2) enter:

```
dotnet ef migrations add InitialCreate

dotnet ef database update
```

Part 5. Data Repository

Step 1. Create the interface 'IWebAPIRepo.cs' and the implementation of the interface 'DBWebAPIRepo.cs' in the 'Data' folder.

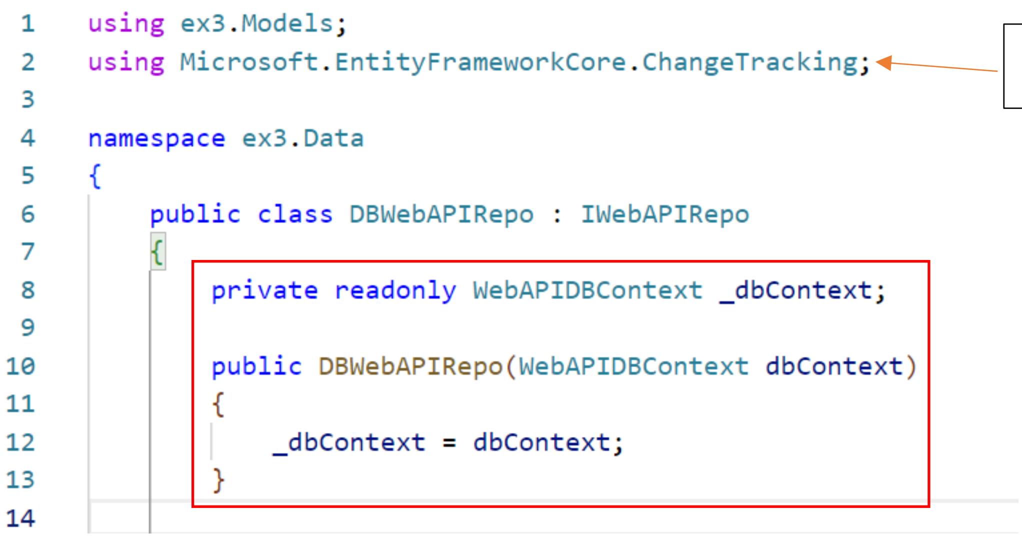
Step 2. in the interface 'IWebAPIRepo.cs', define all abstract methods to be implemented:



```
using ex3.Models;

namespace ex3.Data
{
    public interface IWebAPIRepo
    {

        IEnumerable<Customer> GetAllCustomers();
        Customer GetCustomerByID(int id);
        Customer AddCustomer(Customer customer);


    }
}
```

Step 3. implement of the interface in 'DBWebAPIRepo.cs'

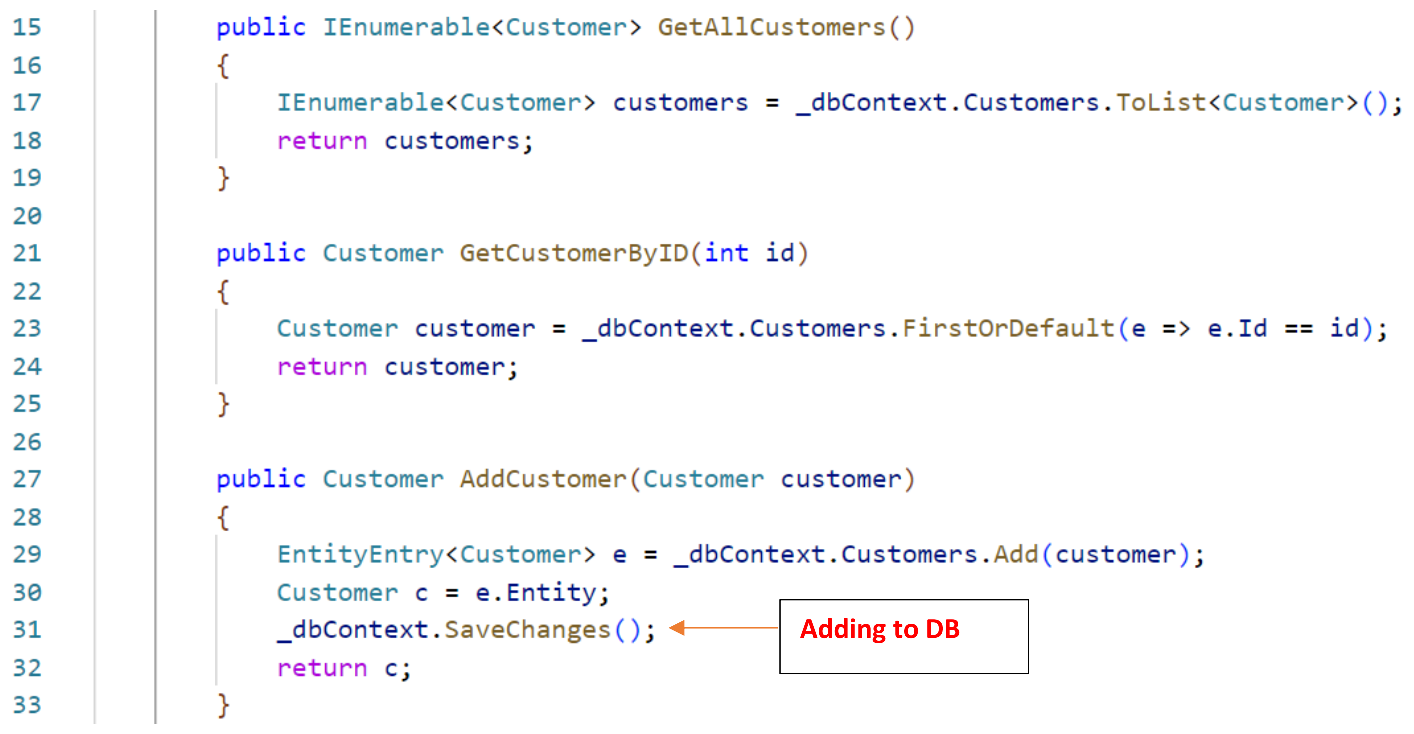
1) in constructor, pass a reference to the Database (DBContext)



```
using ex3.Models;
using Microsoft.EntityFrameworkCore.ChangeTracking;

namespace ex3.Data
{
    public class DBWebAPIRepo : IWebAPIRepo
    {
        private readonly WebAPIDBContext _dbContext;

        public DBWebAPIRepo(WebAPIDBContext dbContext)
        {
            _dbContext = dbContext;
        }

```

To be added when there is an "ADD to DB" method

2) implement the methods from the interface:

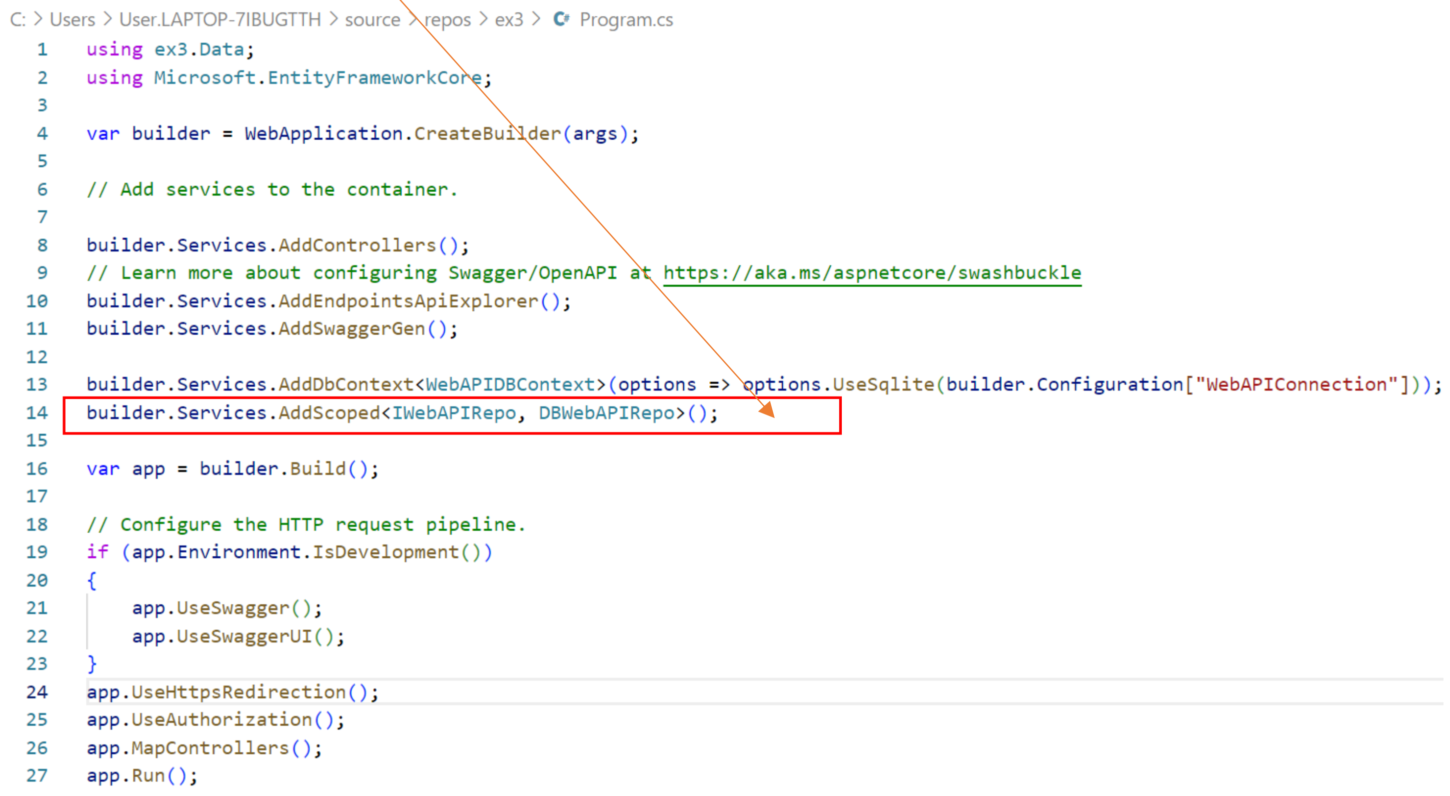


```
public IEnumerable<Customer> GetAllCustomers()
{
    IEnumerable<Customer> customers = _dbContext.Customers.ToList<Customer>();
    return customers;
}

public Customer GetCustomerByID(int id)
{
    Customer customer = _dbContext.Customers.FirstOrDefault(e => e.Id == id);
    return customer;
}

public Customer AddCustomer(Customer customer)
{
    EntityEntry<Customer> e = _dbContext.Customers.Add(customer);
    Customer c = e.Entity;
    _dbContext.SaveChanges();
    return c;
}
```

Step 4. Register the interface and the implementation with the service container

1) Open 'Program.cs' from the root folder

2) Add the Service:



C: > Users > User.LAPTOP-7IBUGTTH > source > repos > ex3 > Program.cs

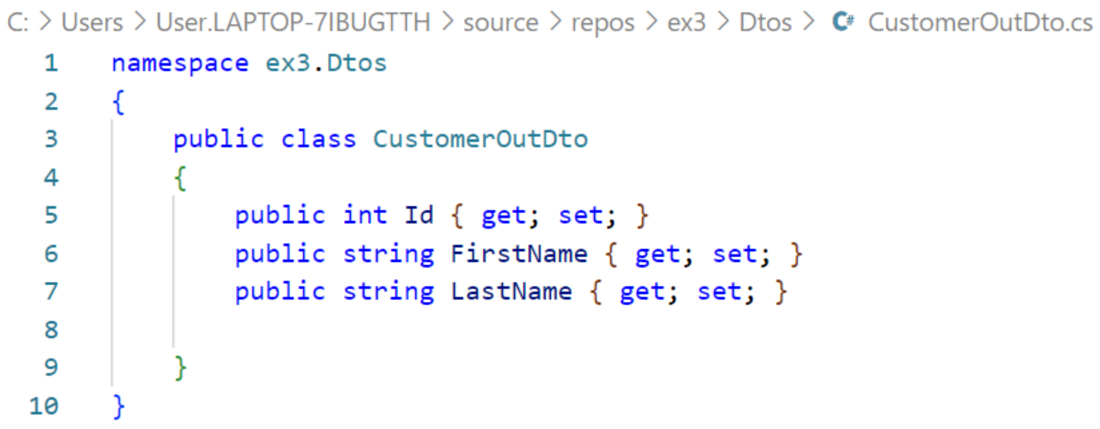
```
using ex3.Data;
using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.

builder.Services.AddControllers();
// Learn more about configuring Swagger/OpenAPI at https://aka.ms/aspnetcore/swashbuckle
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();

builder.Services.AddDbContext<WebAPIDBContext>(options => options.UseSqlite(builder.Configuration["WebAPIConnection"]));
builder.Services.AddScoped<IWebAPIRepo, DBWebAPIRepo>();

var app = builder.Build();

// Configure the HTTP request pipeline.
if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();
}
app.UseHttpsRedirection();
app.UseAuthorization();
app.MapControllers();
app.Run();
```

Part 6. Data Transfer Objects (DTO)

Step 1. Create DTO classes

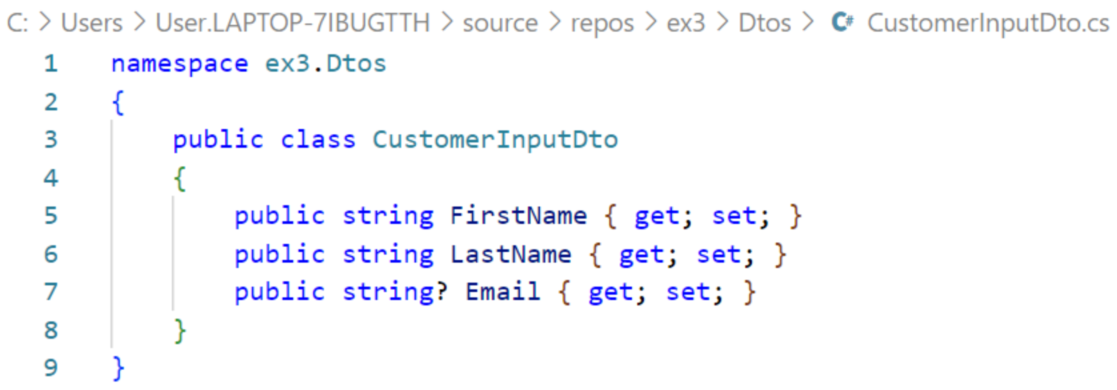
1) create folder ‘Dtos’ in the root folder

2) create ‘CustomerOutDto.cs’ in the ‘Dtos’ folder


C: > Users > User.LAPTOP-7IBUGTTH > source > repos > ex3 > Dtos > CustomerOutDto.cs

```
namespace ex3.Dtos
{
    public class CustomerOutDto
    {
        public int Id { get; set; }
        public string FirstName { get; set; }
        public string LastName { get; set; }

    }
}
```

3) Create ‘CustomerInputDto’ in the ‘Dtos’ folder (NOTE: ‘Id’ needs to be unique, So, it is automatically generated by the system when the customer’s record is inserted into the table).


C: > Users > User.LAPTOP-7IBUGTTH > source > repos > ex3 > Dtos > CustomerInputDto.cs

```
namespace ex3.Dtos
{
    public class CustomerInputDto
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public string? Email { get; set; }
    }
}
```

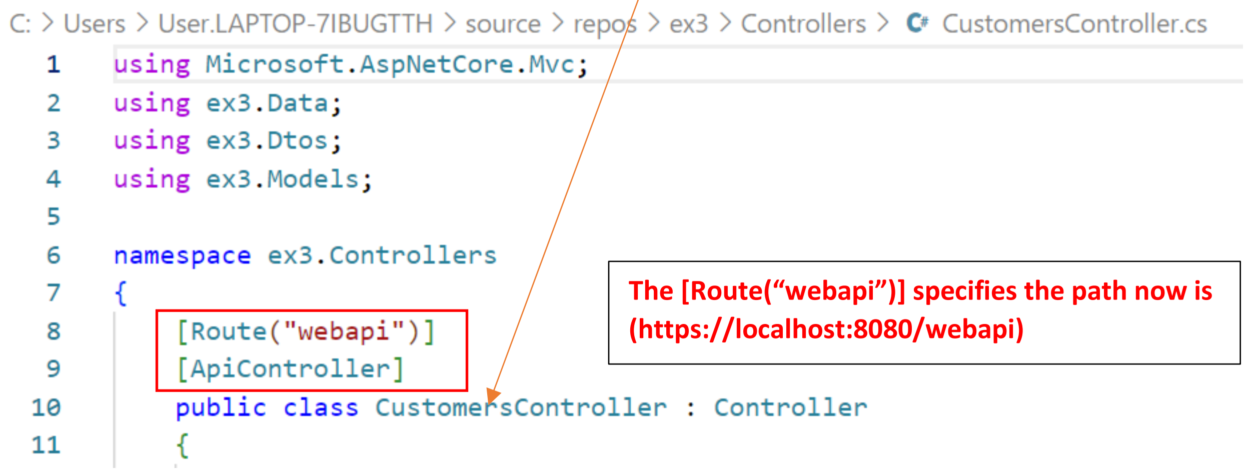
Part 7. Controllers

Step 1. Create Controllers classes

1) Right click the Controllers folder from the VS

2) Add -> Controller -> Choose the first option “MVC Controller – Empty”

3) Add annotation [ApiController] and [Route(“webapi”)]

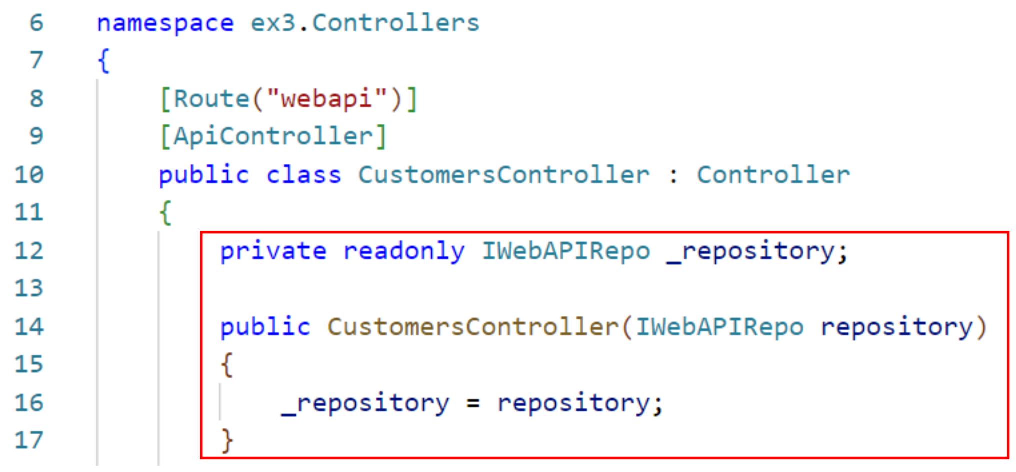


C: > Users > User.LAPTOP-7IBUGTTH > source > repos > ex3 > Controllers > CustomersController.cs

```
using Microsoft.AspNetCore.Mvc;
using ex3.Data;
using ex3.Dtos;
using ex3.Models;

namespace ex3.Controllers
{
    [Route("webapi")]
    [ApiController]
    public class CustomersController : Controller
    {
```

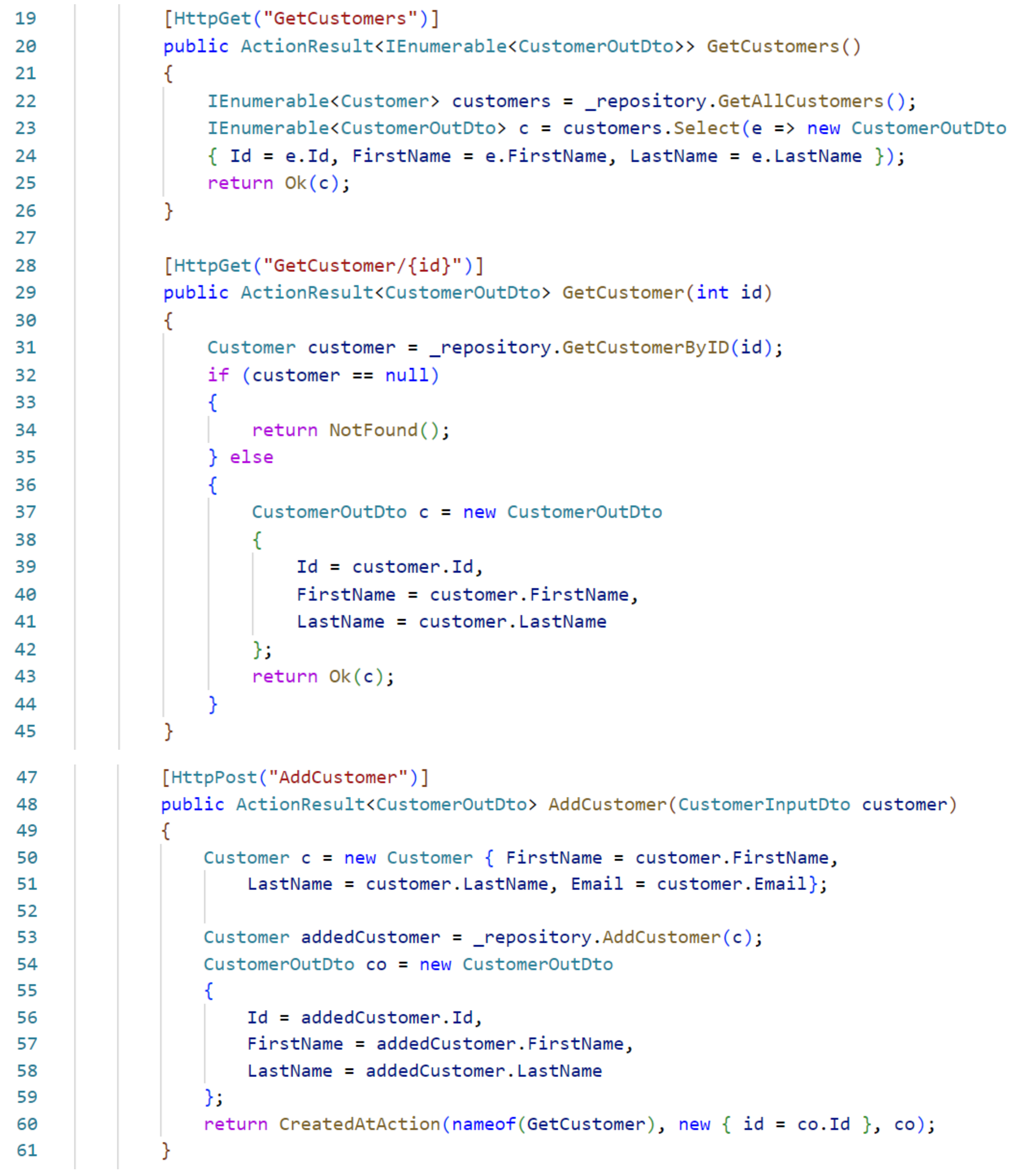
The [Route(“webapi”)] specifies the path now is (https://localhost:8080/webapi)

4) Add constructor:



```
namespace ex3.Controllers
{
    [Route("webapi")]
    [ApiController]
    public class CustomersController : Controller
    {
        private readonly IWebAPIRepo _repository;

        public CustomersController(IWebAPIRepo repository)
        {
            _repository = repository;
        }
```

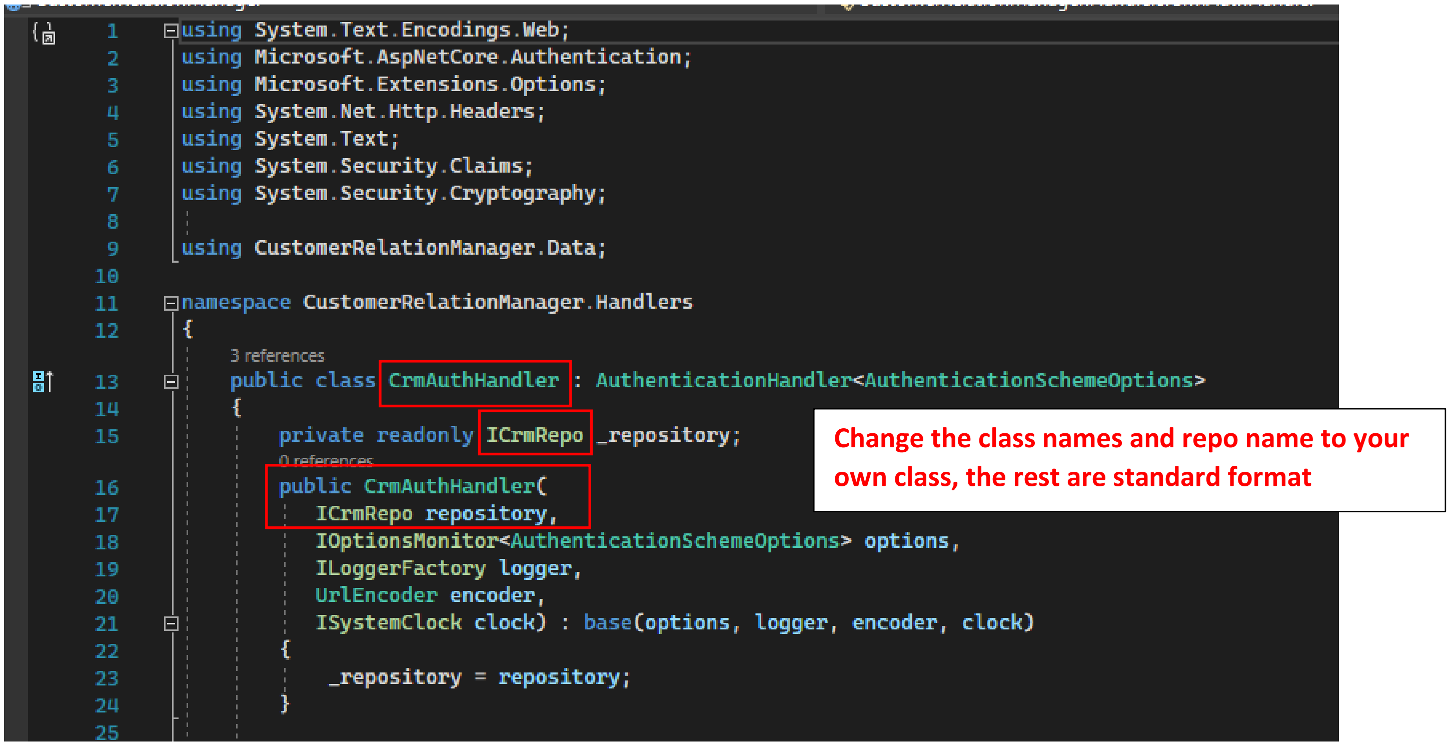
5) Add API Methods:



```csharp
[HttpGet("GetCustomers")]
public ActionResult<IEnumerable<CustomerOutDto>> GetCustomers()
{
    IEnumerable<Customer> customers = _repository.GetAllCustomers();
    IEnumerable<CustomerOutDto> c = customers.Select(e => new CustomerOutDto
    { Id = e.Id, FirstName = e.FirstName, LastName = e.LastName });
    return Ok(c);
}

[HttpGet("GetCustomer/{id}")]
public ActionResult<CustomerOutDto> GetCustomer(int id)
{
    Customer customer = _repository.GetCustomerByID(id);
    if (customer == null)
    {
        return NotFound();
    } else
    {
        CustomerOutDto c = new CustomerOutDto
        {
            Id = customer.Id,
            FirstName = customer.FirstName,
            LastName = customer.LastName
        };
        return Ok(c);
    }
}

[HttpPost("AddCustomer")]
public ActionResult<CustomerOutDto> AddCustomer(CustomerInputDto customer)
{
    Customer c = new Customer { FirstName = customer.FirstName,
        LastName = customer.LastName, Email = customer.Email};

    Customer addedCustomer = _repository.AddCustomer(c);
    CustomerOutDto co = new CustomerOutDto
    {
        Id = addedCustomer.Id,
        FirstName = addedCustomer.FirstName,
        LastName = addedCustomer.LastName
    };
    return CreatedAtAction(nameof(GetCustomer), new { id = co.Id }, co);
}
```

Part 8. Authentication

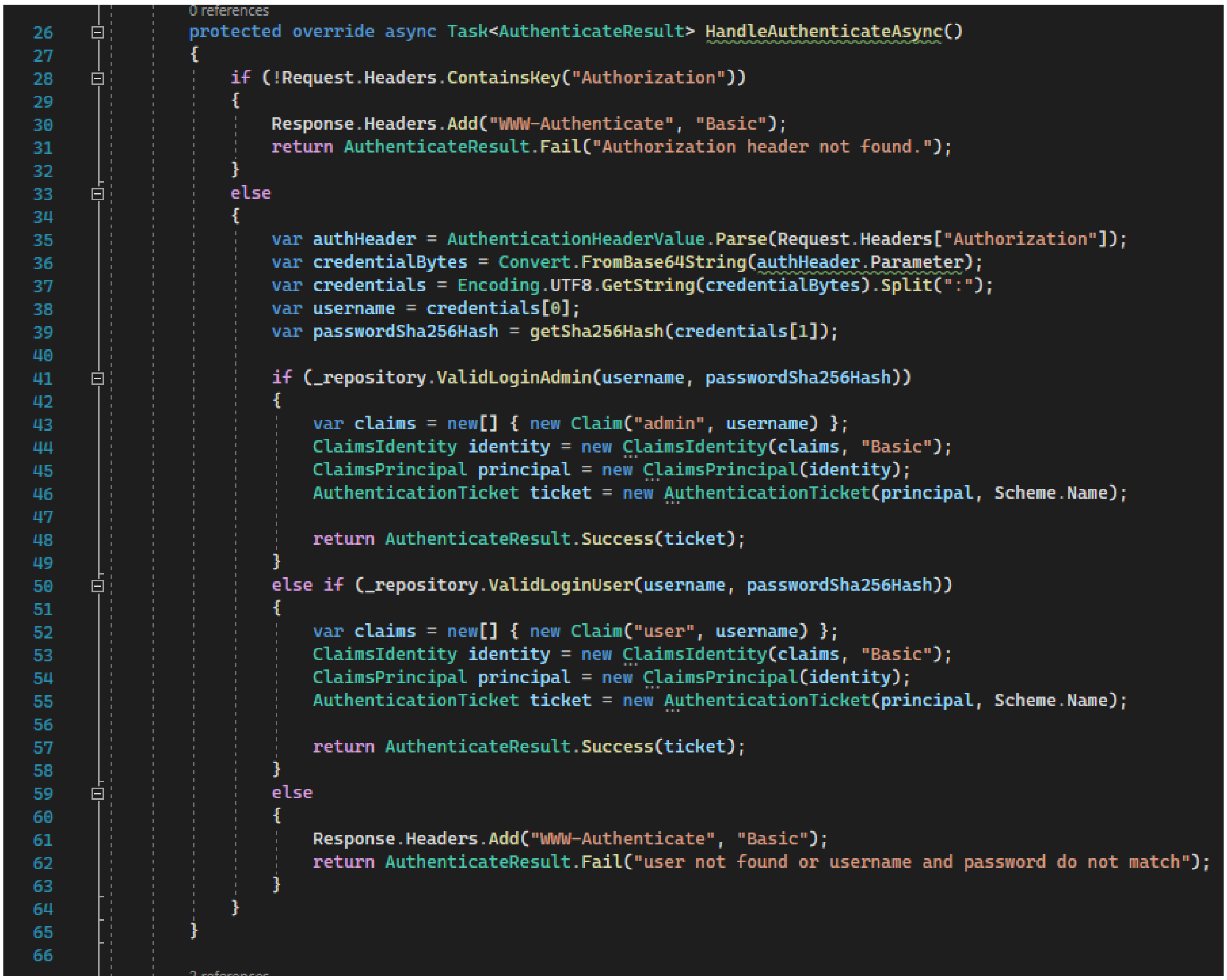
Step 1. Add the authentication handler class and create the standard constructor:



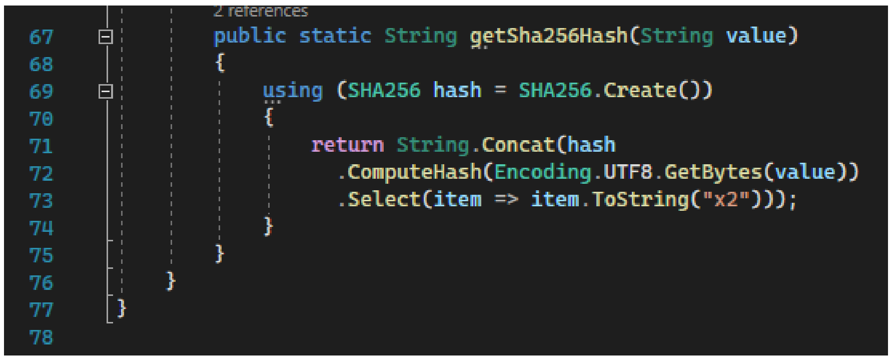
```
using System.Text.Encodings.Web;
using Microsoft.AspNetCore.Authentication;
using Microsoft.Extensions.Options;
using System.Net.Http.Headers;
using System.Text;
using System.Security.Claims;
using System.Security.Cryptography;

using CustomerRelationManager.Data;

namespace CustomerRelationManager.Handlers
{
    3 references
    public class CrmAuthHandler : AuthenticationHandler<AuthenticationSchemeOptions>
    {
        private readonly ICrmRepo _repository;
        0 references
        public CrmAuthHandler(
            ICrmRepo repository,
            IOptionsMonitor<AuthenticationSchemeOptions> options,
            ILoggerFactory logger,
            UrlEncoder encoder,
            ISystemClock clock) : base(options, logger, encoder, clock)
        {
            _repository = repository;
        }
```

Change the class names and repo name to your own class, the rest are standard format

Step 2. Add the handle method, you can make adjustment to the method according to your need:

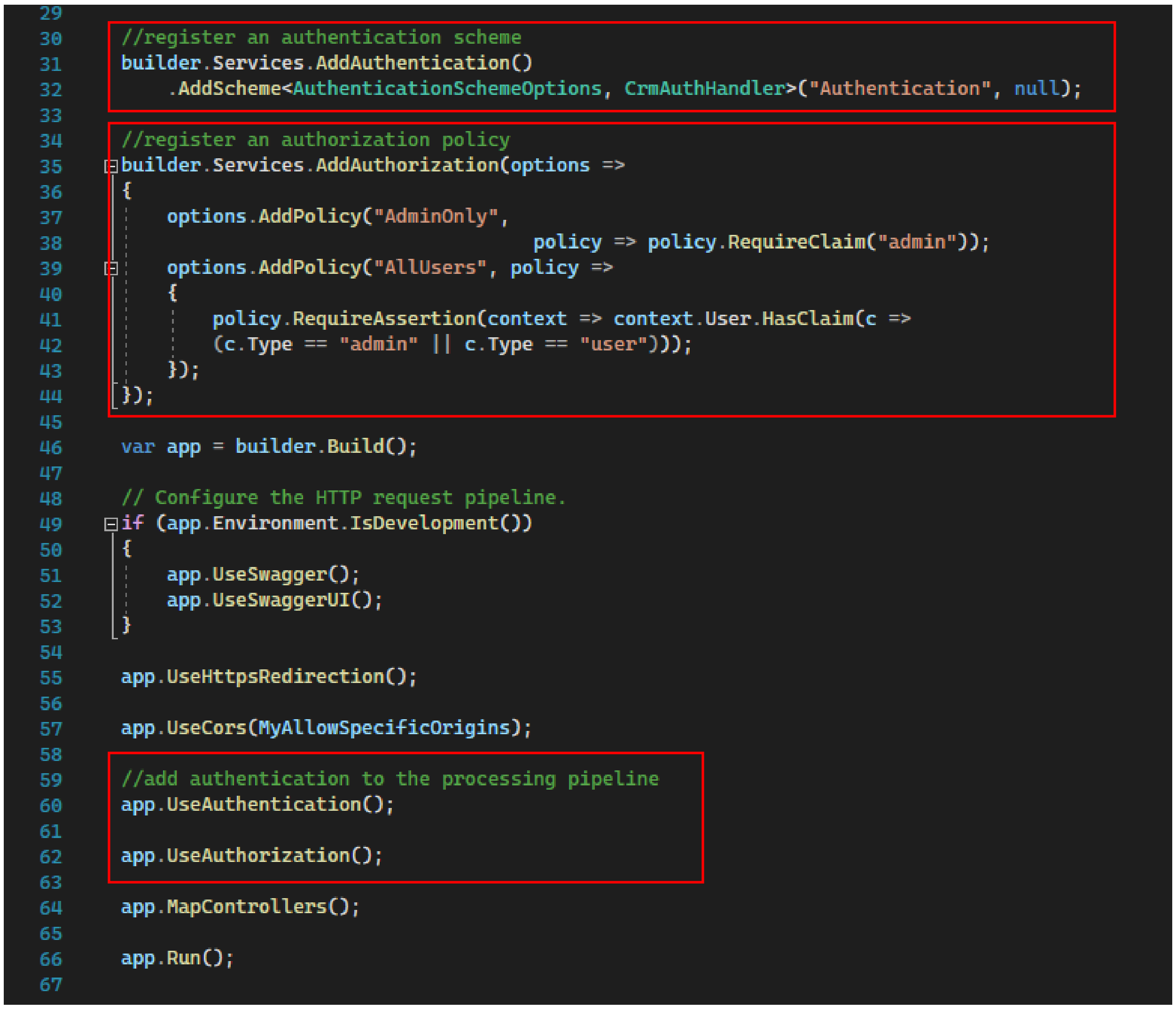


```
        protected override async Task<AuthenticateResult> HandleAuthenticateAsync()
        {
            if (!Request.Headers.ContainsKey("Authorization"))
            {
                Response.Headers.Add("WWW-Authenticate", "Basic");
                return AuthenticateResult.Fail("Authorization header not found.");
            }
            else
            {
                var authHeader = AuthenticationHeaderValue.Parse(Request.Headers["Authorization"]);
                var credentialBytes = Convert.FromBase64String(authHeader.Parameter);
                var credentials = Encoding.UTF8.GetString(credentialBytes).Split(":");
                var username = credentials[0];
                var passwordSha256Hash = getSha256Hash(credentials[1]);

                if (_repository.ValidLoginAdmin(username, passwordSha256Hash))
                {
                    var claims = new[] { new Claim("admin", username) };
                    ClaimsIdentity identity = new ClaimsIdentity(claims, "Basic");
                    ClaimsPrincipal principal = new ClaimsPrincipal(identity);
                    AuthenticationTicket ticket = new AuthenticationTicket(principal, Scheme.Name);

                    return AuthenticateResult.Success(ticket);
                }
                else if (_repository.ValidLoginUser(username, passwordSha256Hash))
                {
                    var claims = new[] { new Claim("user", username) };
                    ClaimsIdentity identity = new ClaimsIdentity(claims, "Basic");
                    ClaimsPrincipal principal = new ClaimsPrincipal(identity);
                    AuthenticationTicket ticket = new AuthenticationTicket(principal, Scheme.Name);

                    return AuthenticateResult.Success(ticket);
                }
                else
                {
                    Response.Headers.Add("WWW-Authenticate", "Basic");
                    return AuthenticateResult.Fail("user not found or username and password do not match");
                }
            }
        }

```



```
        public static String getSha256Hash(String value)
        {
            using (SHA256 hash = SHA256.Create())
            {
                return String.Concat(hash
                  .ComputeHash(Encoding.UTF8.GetBytes(value))
                  .Select(item => item.ToString("x2")));
            }
        }
    }
}

```

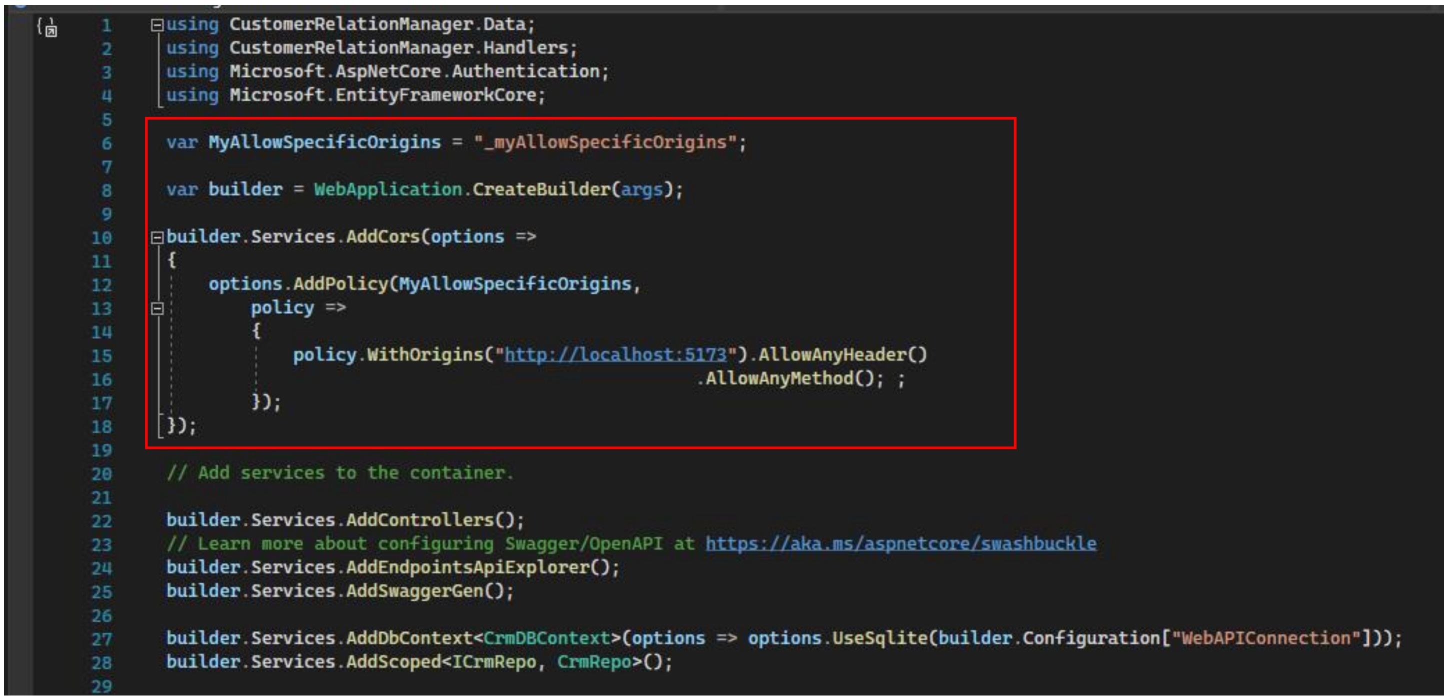
Step 3. In your main Program.cs, register the authentication handler class and authorization policies:



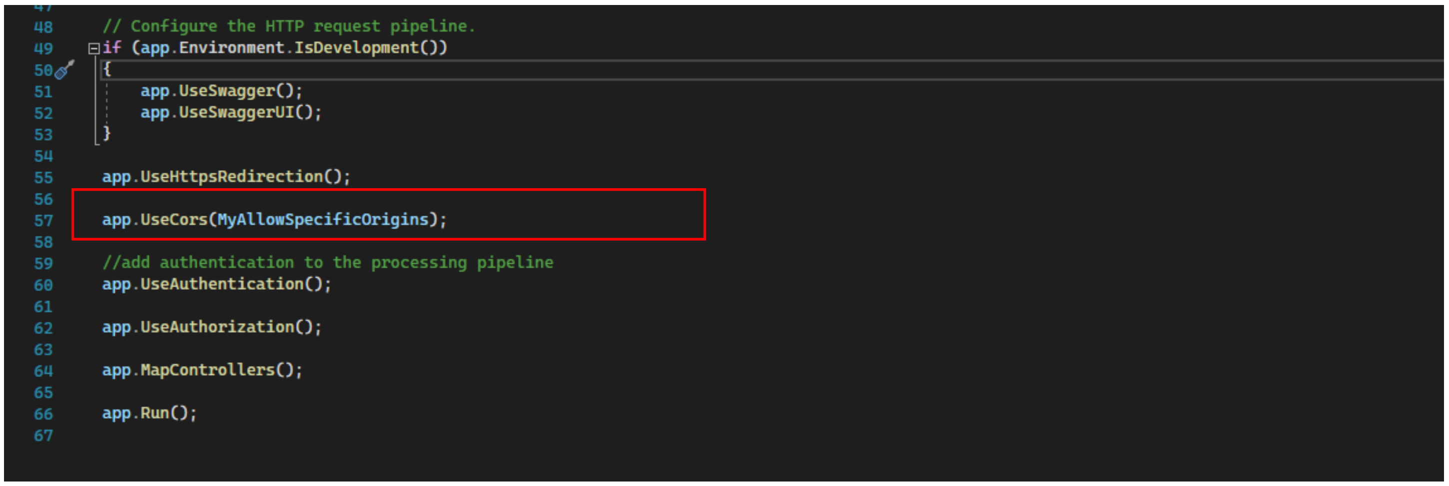
```
//register an authentication scheme
builder.Services.AddAuthentication()
    .AddScheme<AuthenticationSchemeOptions, CrmAuthHandler>("Authentication", null);

//register an authorization policy
builder.Services.AddAuthorization(options =>
{
    options.AddPolicy("AdminOnly",
                                    policy => policy.RequireClaim("admin"));
    options.AddPolicy("AllUsers", policy =>
    {
        policy.RequireAssertion(context => context.User.HasClaim(c =>
        (c.Type == "admin" || c.Type == "user")));
    });
});

var app = builder.Build();

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

app.UseHttpsRedirection();

app.UseCors(MyAllowSpecificOrigins);

//add authentication to the processing pipeline
app.UseAuthentication();

app.UseAuthorization();

app.MapControllers();

app.Run();
```

Part 9. CORS configuration

For some of the front-end server to connect to this backend server, the CORS policy has to be configured in the main Program.cs class, as below:



```csharp
using CustomerRelationManager.Data;
using CustomerRelationManager.Handlers;
using Microsoft.AspNetCore.Authentication;
using Microsoft.EntityFrameworkCore;

var MyAllowSpecificOrigins = "_myAllowSpecificOrigins";

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddCors(options =>
{
    options.AddPolicy(MyAllowSpecificOrigins,
        policy =>
        {
            policy.WithOrigins("http://localhost:5173").AllowAnyHeader()
                                                        .AllowAnyMethod(); ;
        });
});

// Add services to the container.

builder.Services.AddControllers();
// Learn more about configuring Swagger/OpenAPI at https://aka.ms/aspnetcore/swashbuckle
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();

builder.Services.AddDbContext<CrmDBContext>(options => options.UseSqlite(builder.Configuration["WebAPIConnection"]));
builder.Services.AddScoped<ICrmRepo, CrmRepo>();
```



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

app.UseHttpsRedirection();

app.UseCors(MyAllowSpecificOrigins);

//add authentication to the processing pipeline
app.UseAuthentication();

app.UseAuthorization();

app.MapControllers();

app.Run();
```

That's all the steps that you need to set up the backend framework, now you can add methods into the project and make new API endpoints!