**Lab 01. Building an Product Management Application using ASP.NET Core Web App (Model-View-Controller)**

# 1. Introduction

Imagine you're an employee of a store named **ProductStore**. Your manager has asked you to develop a WPF application for product management. The application has to support adding, viewing, modifying, and removing products—a standardized usage action verbs better known as Create, Read, Update, Delete (CRUD).

This lab explores creating an application using ASP.NET Core Web App (Model-View-Controller). An **SQL Server** **Database** will be created to persist the car's data that will be used for reading and managing product data by **Entity Framework Core**.

# 2. Lab Objectives

In this lab, you will:

* Use the Visual Studio.NET to create ASP.NET Core Web App (MVC) and Class Library (.dll) projects.
* Create a SQL Server database named MyStoreDB that has a Product, Category, AccountMember tables.
* Apply Repository pattern in a project.
* Add CRUD action methods to ASP.NET Core Web App (MVC).
* Run the project and test the application actions.

# 3. Database Design (MyStore)

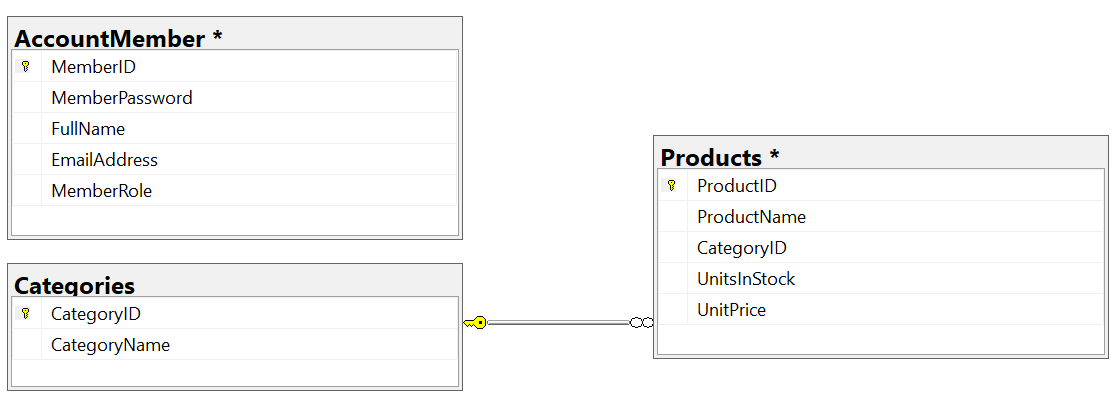


Table AccountMember

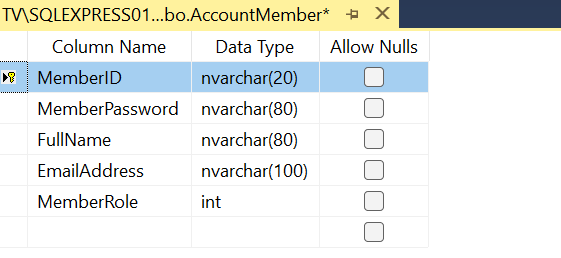


Table Categories

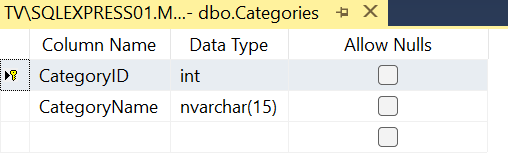
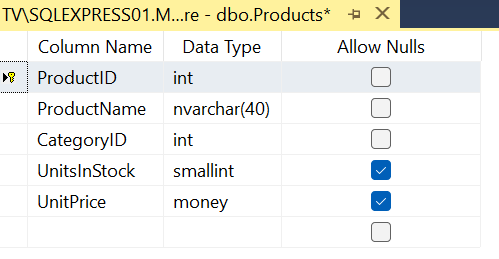


Table Products



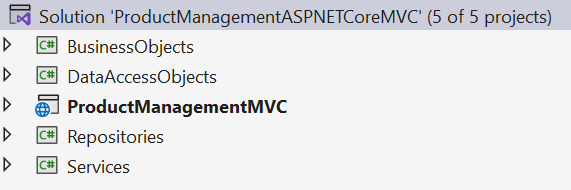
# Activity 01: Build a solution by Visual Studio.NET

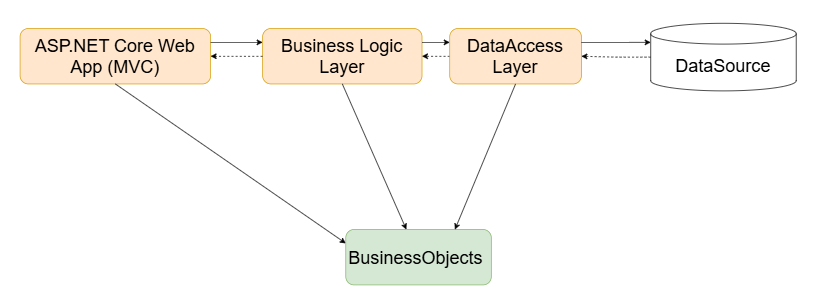
Create a Blank Solution named **ProductManagementASPNETCoreMVC** then add new a **Class Library** project named **BusinessObjects, DataAccessObjects, Repositories, Services** and a ASP.NET Core Web App (MVC) project named **ProductManagementMVC**

## **Step 01**. Create a Blank solution.

## **Step 02.** Create 4 **Class Library** projects.

## **Step 03.** Create a project (ASP.NET Core Web App (MVC).



****

Note:

* **Data Source** in this case is the SQL Server Database
* **Services Project** – This project represents a layer or component responsible for implementing the business logic of an application.
* **Repository Project** – This project provides an abstraction layer between the application’s business logic and the underlying data source.
* **Data Access Layer Project** – This project used to abstract and encapsulate the logic for accessing data from a data source, such as a database.

# Activity 02: Write codes for the BusinessObjects project

## **Step 01**. Install the following packages from NuGet:

* Microsoft.EntityFrameworkCore.SqlServer --version 8.0.2
* Microsoft.EntityFrameworkCore.Tools --version 8.0.2
* Microsoft.Extensions.Configuration.Json --version 8.0.0

Check the tool for EFCore (install/uninstall tool if needed) (dotnet SDK 8.0.202)

dotnet tool install --global dotnet-ef --version 8.0.2

dotnet tool uninstall --global dotnet-ef

## **Step 02**. Right-click on project , select **Open In Terminal.** On **Developer PowerShell** dialog execute the following commands to generate model:

* Implement ORM

dotnet ef dbcontext scaffold “**Server=(local); Database=MyStore; Uid=sa; Pwd=1234567890**” Microsoft.EntityFrameworkCore.SqlServer --output-dir ./

* Change the connection string in OnConfiguring() function of MyStoreContext.cs

using System.IO;

         using Microsoft.Extensions.Configuration.Json;

*private string GetConnectionString()*

*{*

*IConfiguration configuration = new ConfigurationBuilder()*

*.SetBasePath(Directory.GetCurrentDirectory())*

*.AddJsonFile("appsettings.json", true, true).Build();*

*return configuration["ConnectionStrings:DefaultConnectionString"];*

*}*

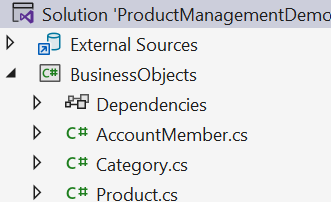
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

        {

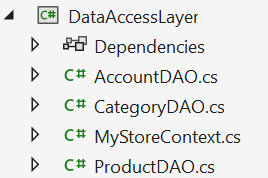
***optionsBuilder.UseSqlServer(GetConnectionString());***

         }

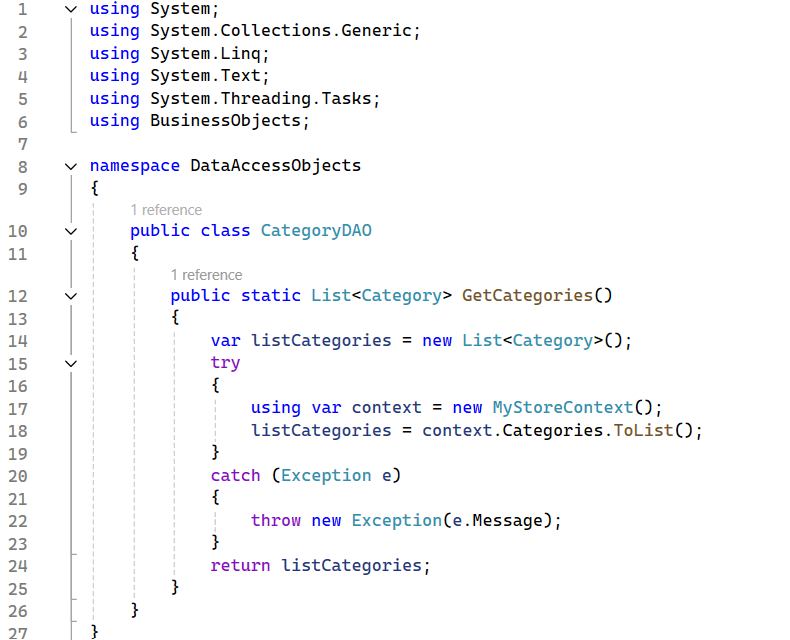
* Move the MyStoreContext.cs to DataAccessLayer Project



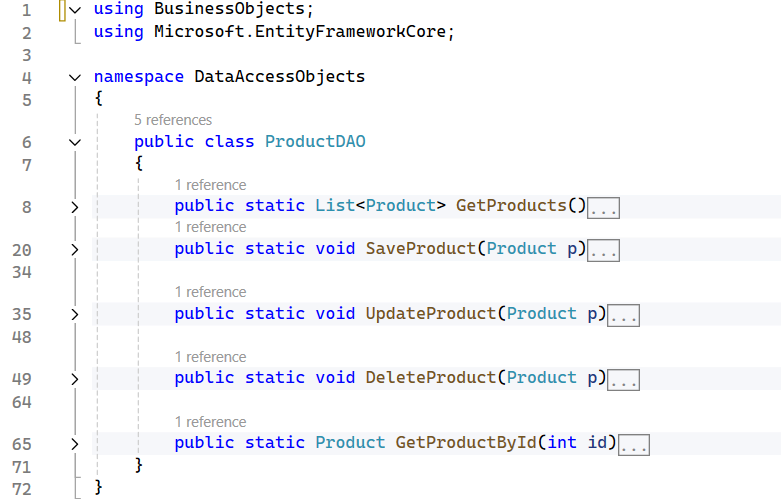
# Activity 03: Write codes for the DataAccessLayer project



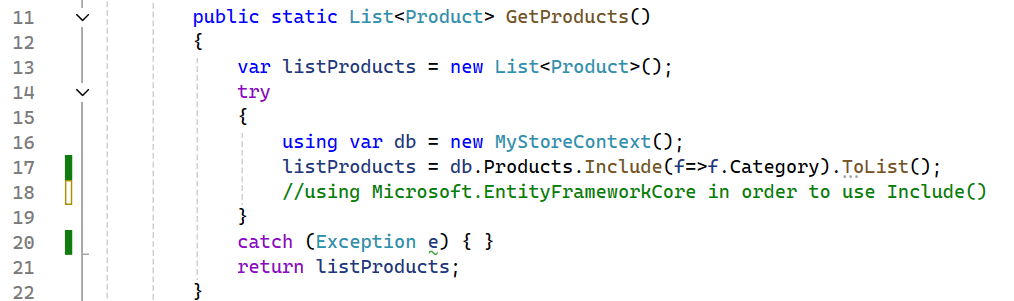
## **Step 01.** On the **DataAccessObjects** project, add a class named **CategoryDAO.cs** and write codes as follows:

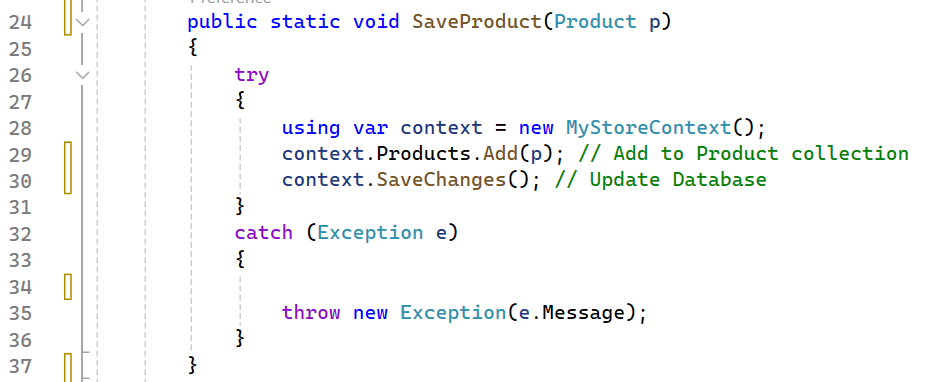


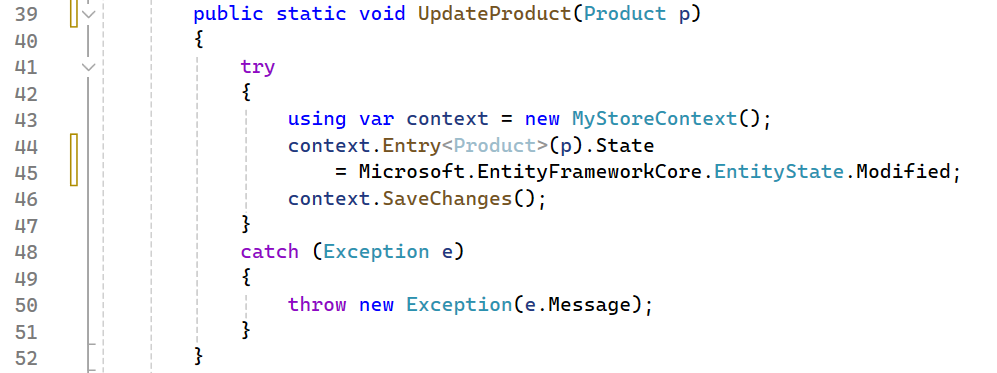
## **Step 02**. On the **DataAccessObjects** project, add a class named **ProductDAO.cs** and write codes as follows:

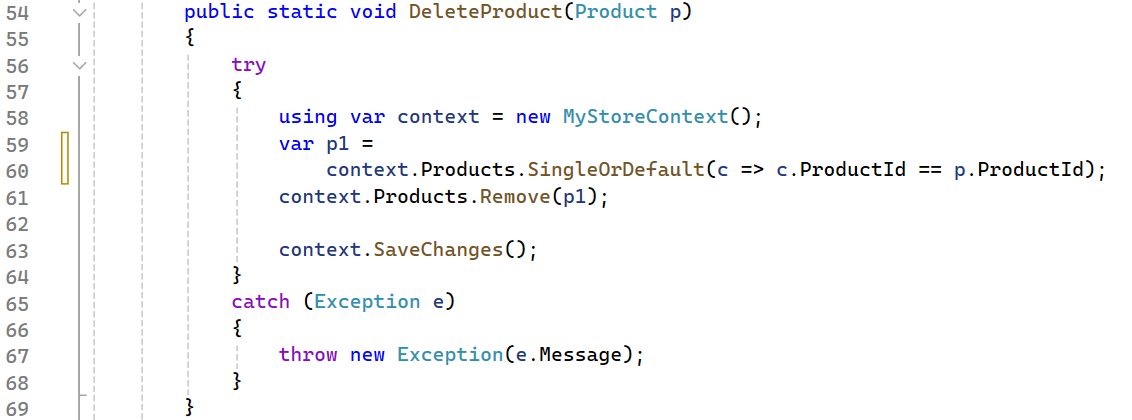


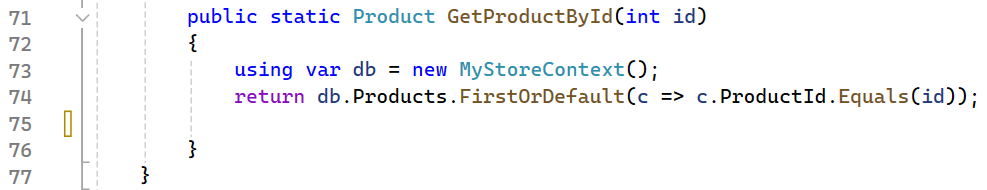
The details of functions in ProductDAO.cs:



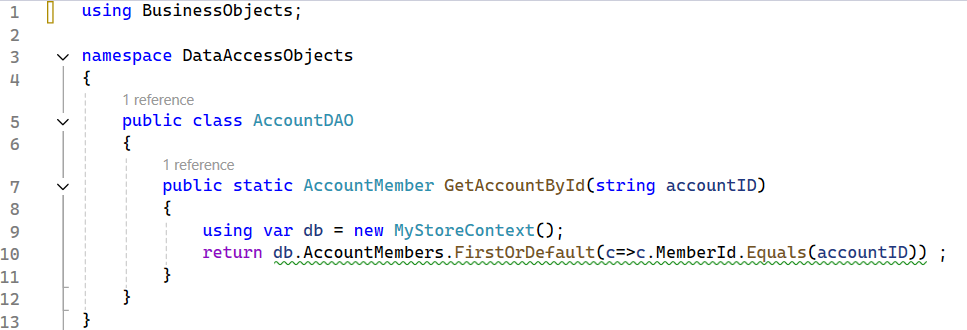




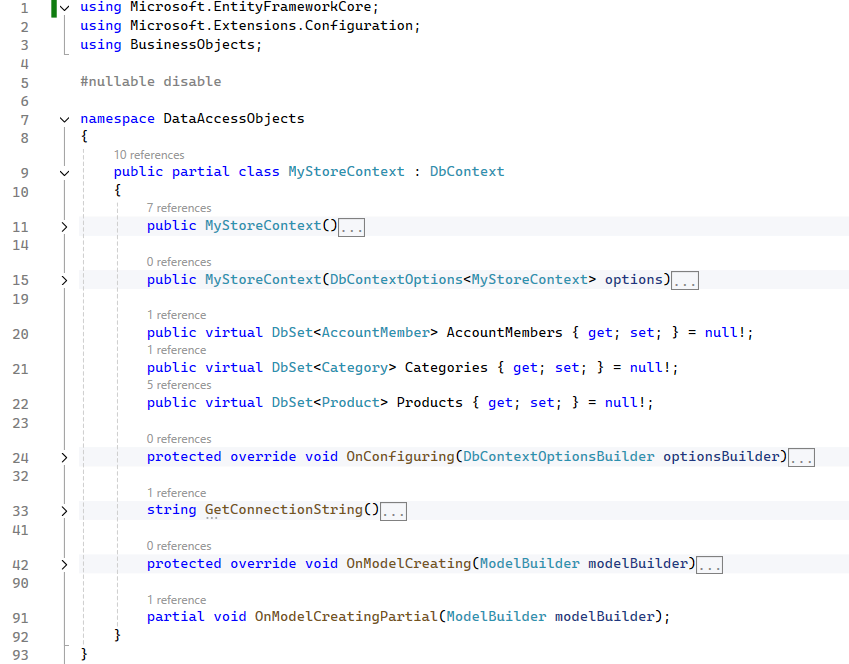




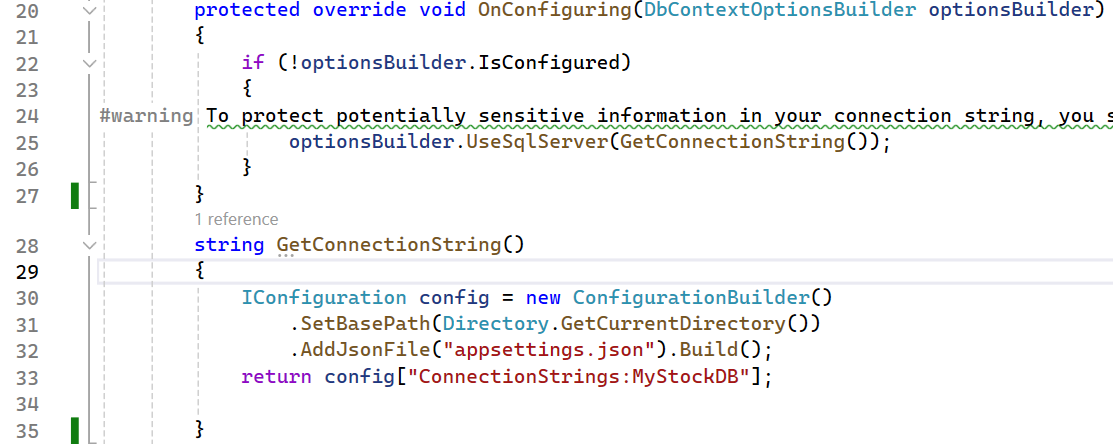
## **Step 03**. Write codes for **AccountDAO.cs** as follows:



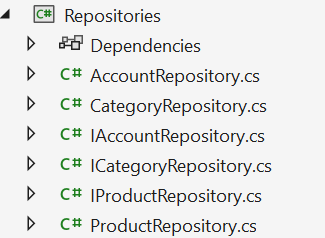
**Step 04**. The codes for **MyStoreContext.cs**:



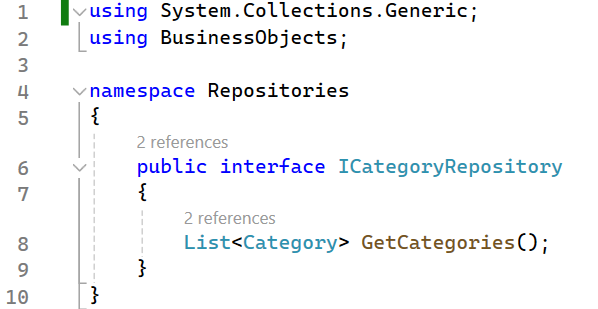
The details for GetConnectionString() and OnConfiguring() functions



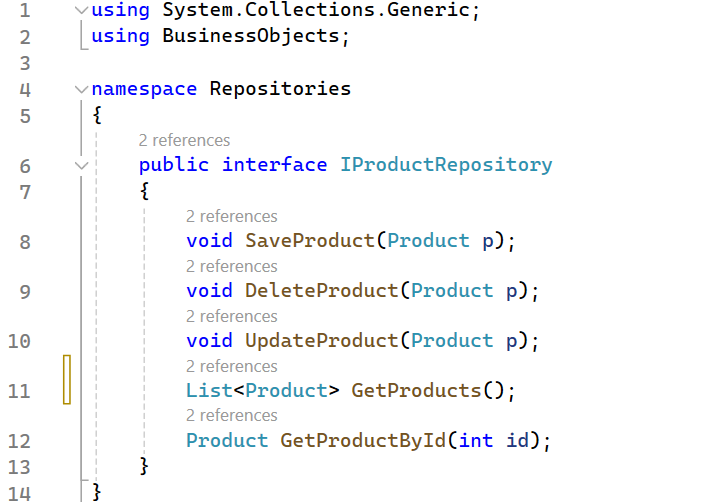
# Activity 04: Write codes for the Repositories project



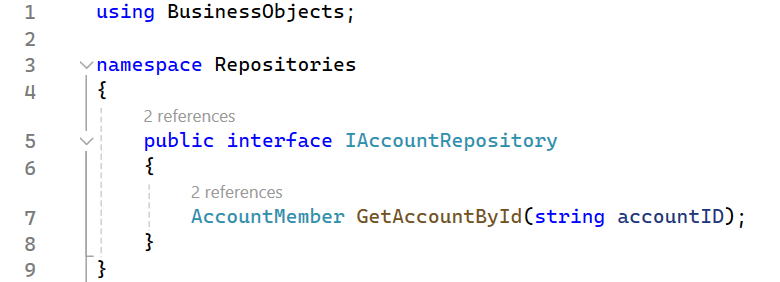
## **Step 01.** On the **Repositories** project, add an interface named **ICatergoryRepository.cs** and write codes as follows:



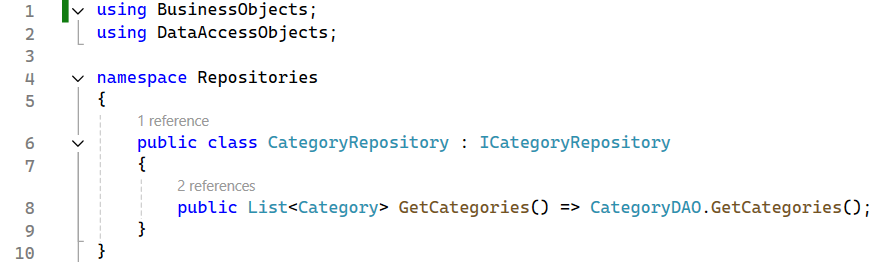
## **Step 02**. On the **Repositories** project, add an interface named **IProductRepository.cs** and write codes as follows:



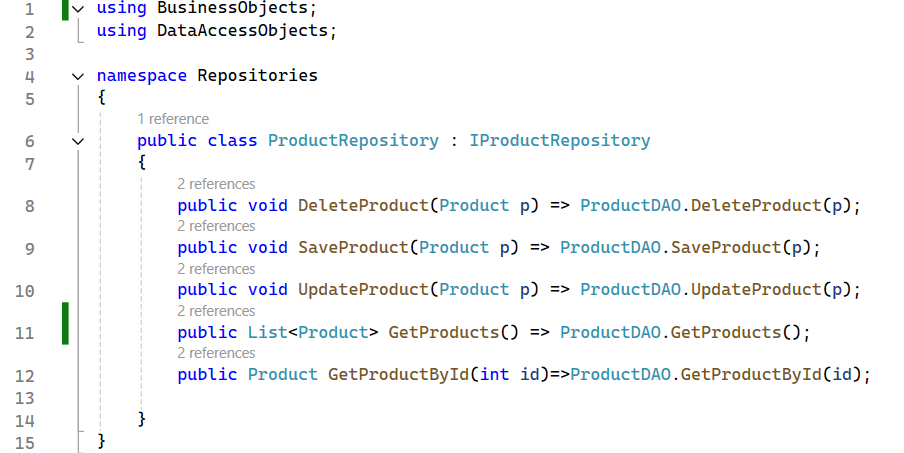
## **Step 03**. On the **Repositories** project, add an interface named **IAccountRepository.cs** and write codes as follows:



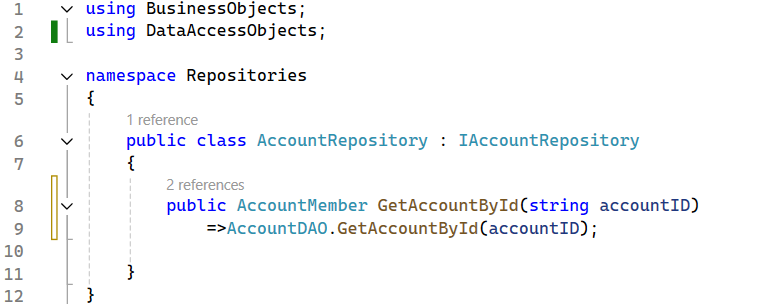
**Step 04**. Write codes for class **CategoryRepository.cs** as follows:



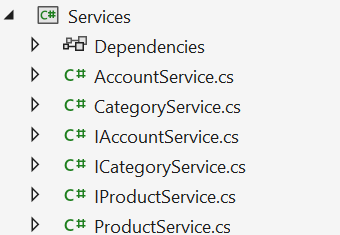
**Step 05**. Write codes for class **ProductRepository.cs** as follows:



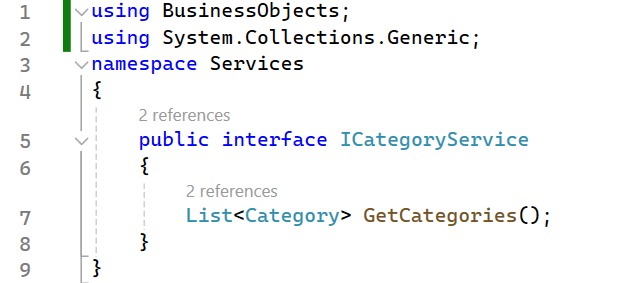
**Step 06**. Write codes for class **AccountRepository.cs** as follows:



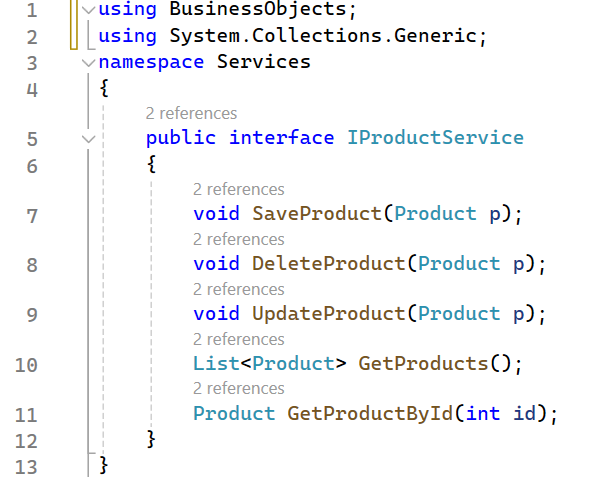
# Activity 05: Write codes for the Services project



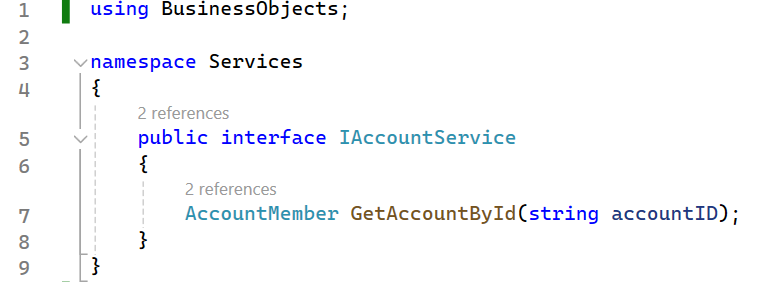
## **Step 01.** On the **Services** project, add an interface named **ICatergoryService.cs** and write codes as follows:



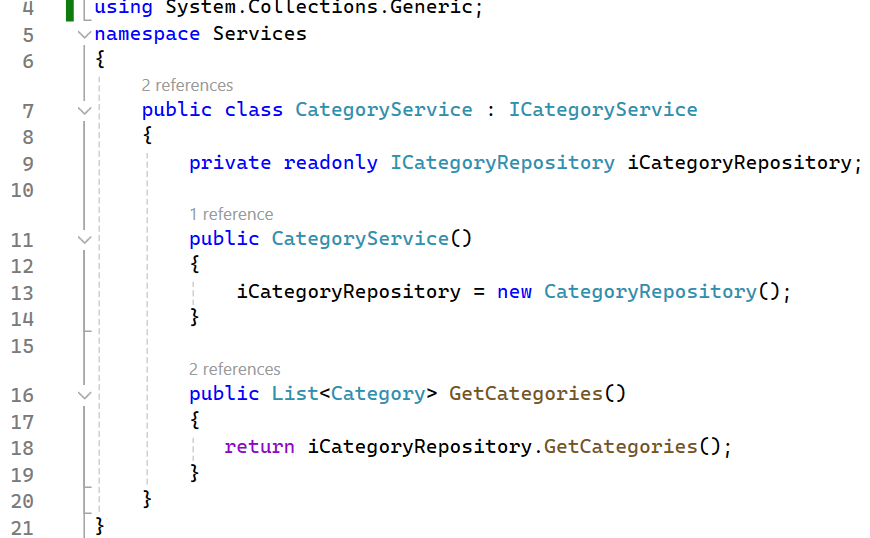
## **Step 02**. On the **Services** project, add an interface named **IProductService.cs** and write codes as follows:



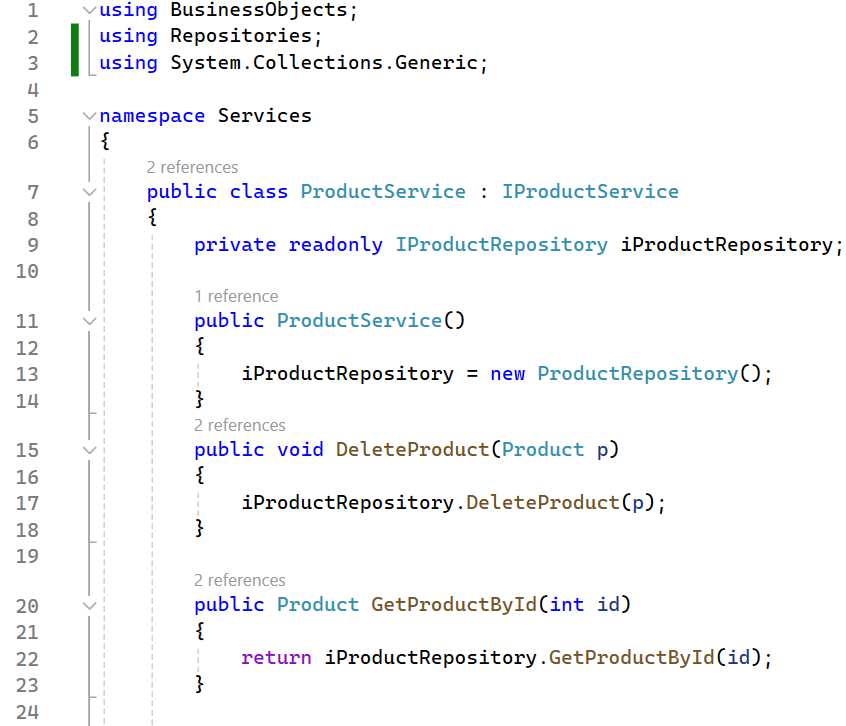
## **Step 03**. On the **Services** project, add an interface named **IAccountService.cs** and write codes as follows:

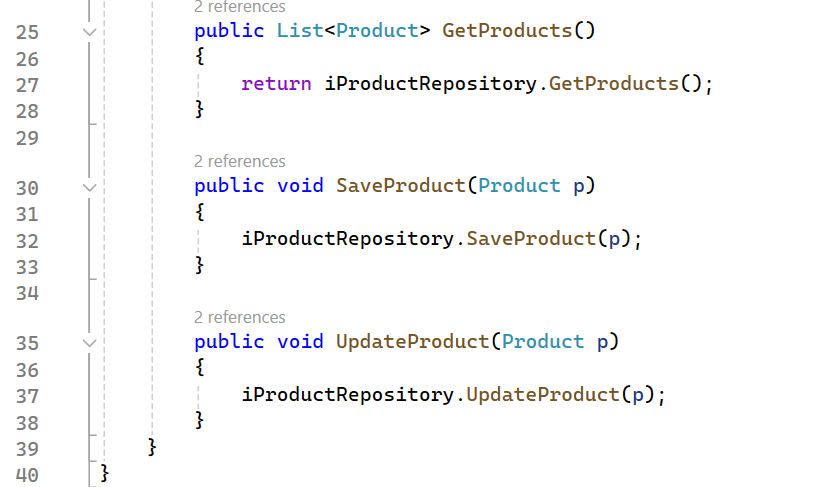


**Step 04**. Write codes for class **CategoryService.cs** as follows:

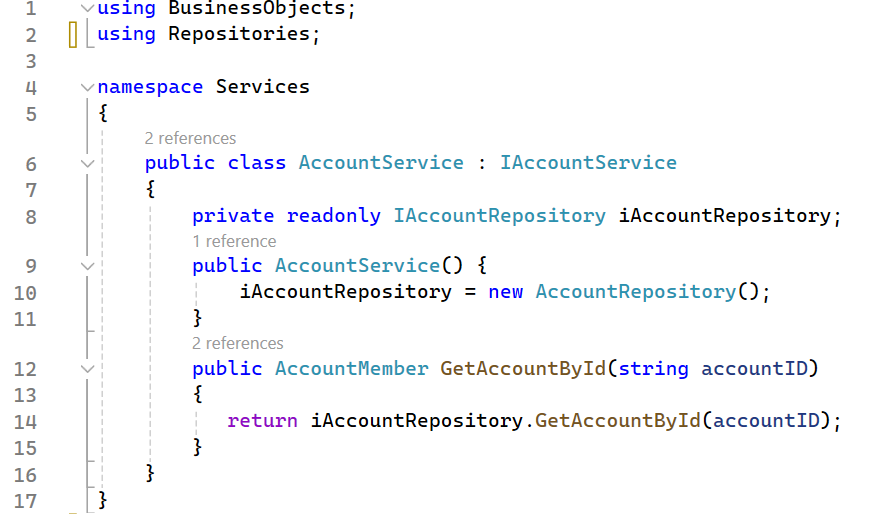


**Step 05**. Write codes for class **ProductService.cs** as follows:





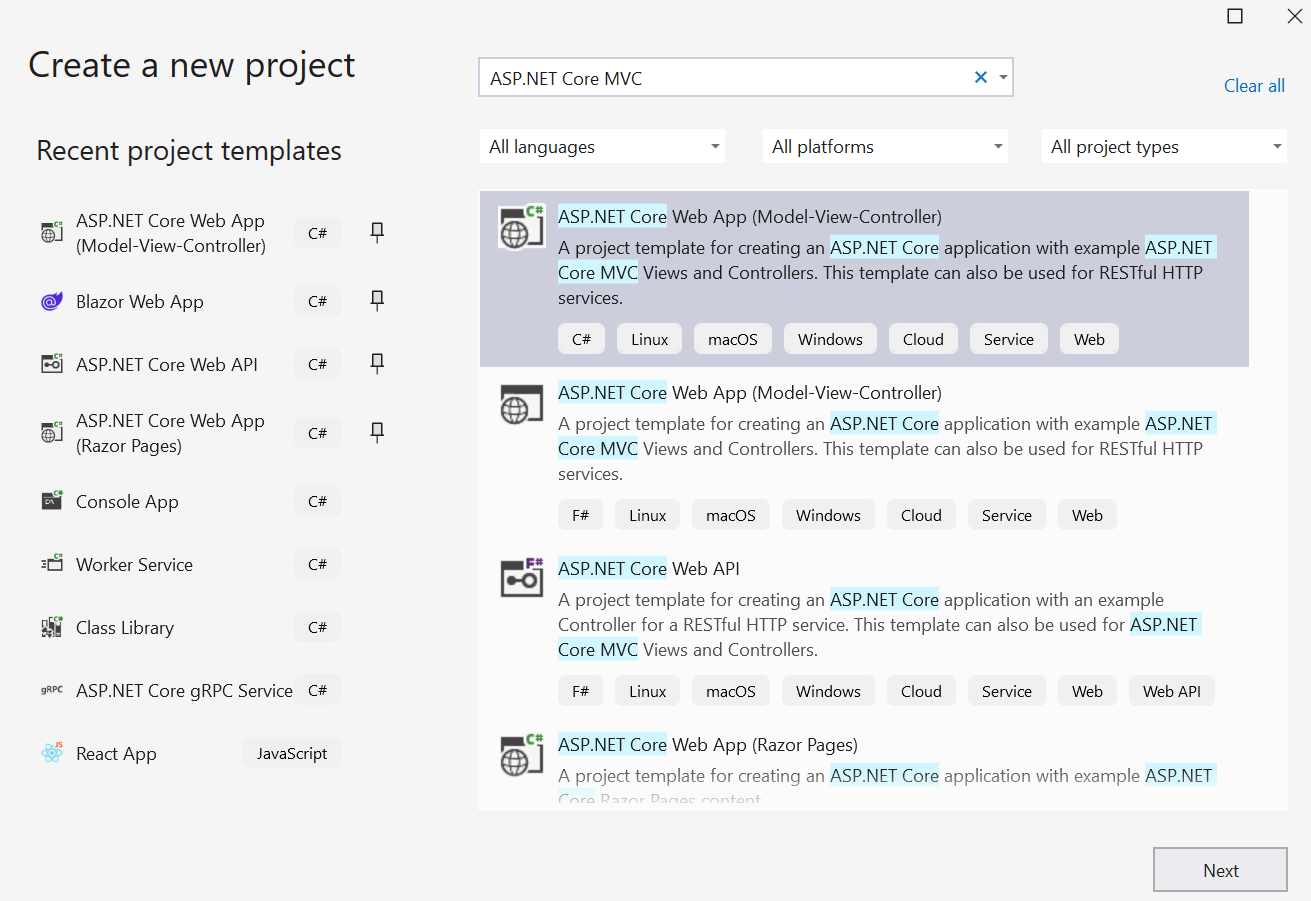
**Step 06**. Write codes for class **AccountService.cs** as follows:

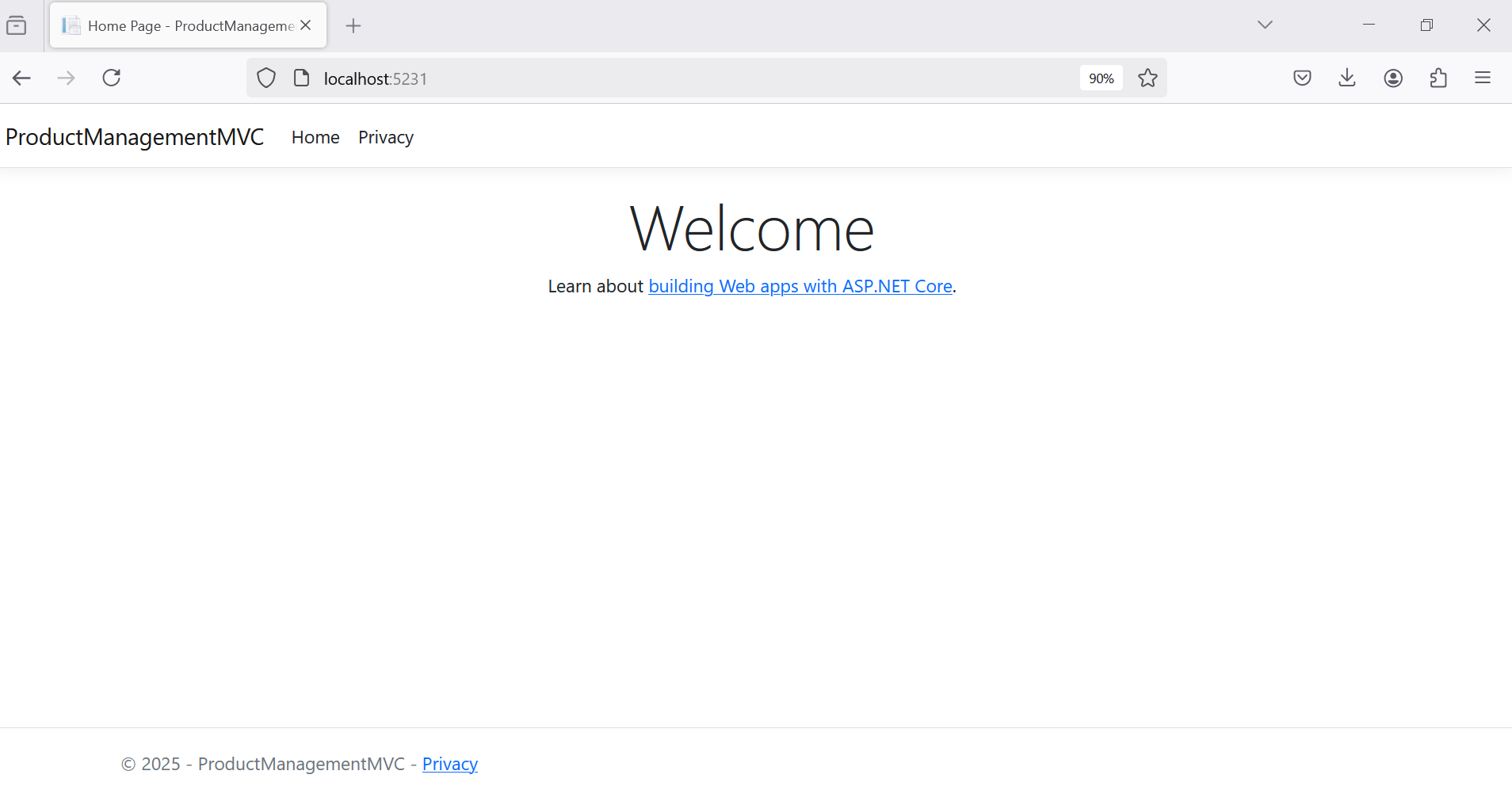


# Activity 06: Work with ASP.NET Core Web App (Model-View-Controller)

## **Step 01.** Create and run the **ASP.NET Core Web App (MVC)** project, the result as the following

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



****

## **Step 02**. Add connection string to **appsettings.json**.

{

"Logging": {

"LogLevel": {

"Default": "Information",

"Microsoft.AspNetCore": "Warning"

}

},

"AllowedHosts": "\*",

"ConnectionStrings": {

"MyStockDB": "Server=localhost;uid=sa;pwd=1234567890;database=MyStore;TrustServerCertificate=True"

}

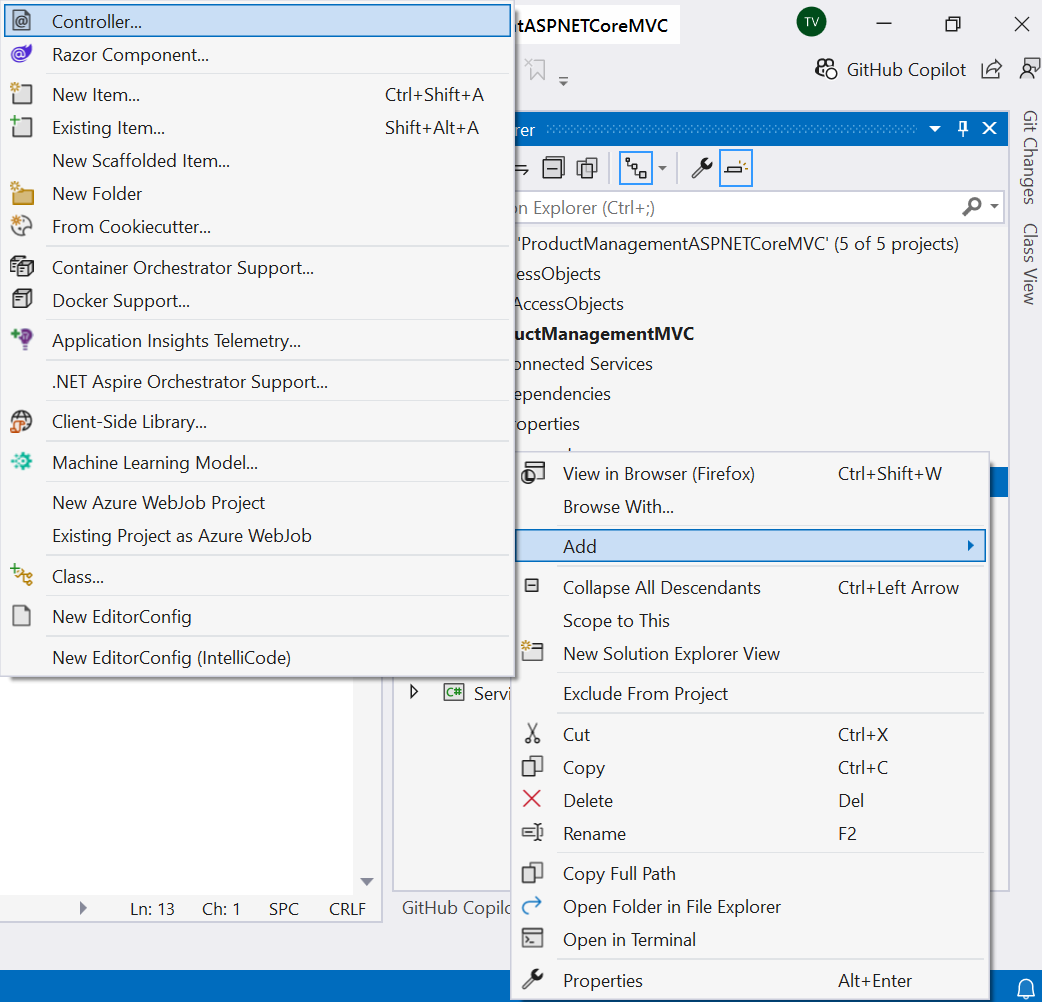
}

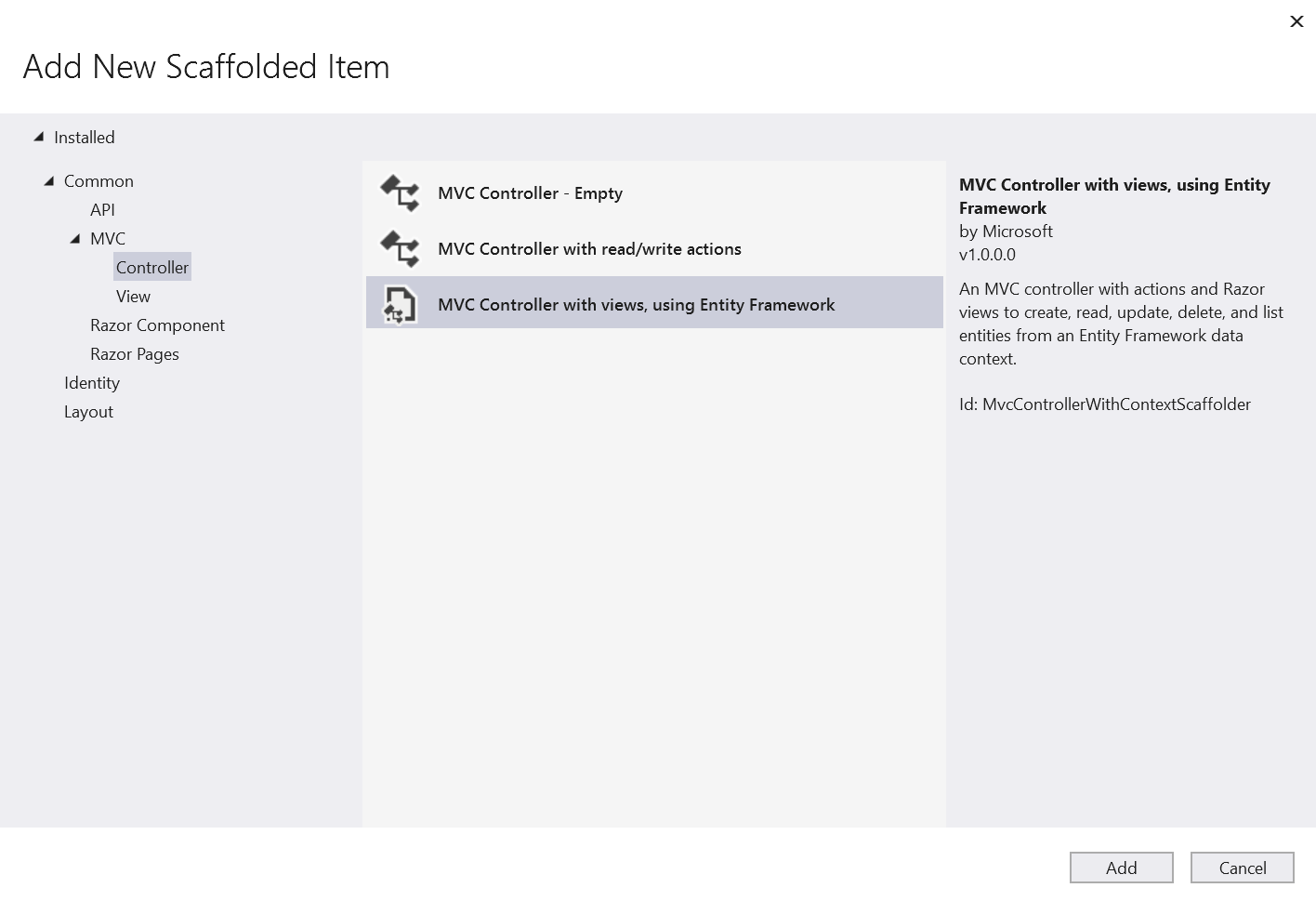
**Step 03.** Add **Business Objects** and **Services** projects as references for the ASP.NET Core Web App (MVC)

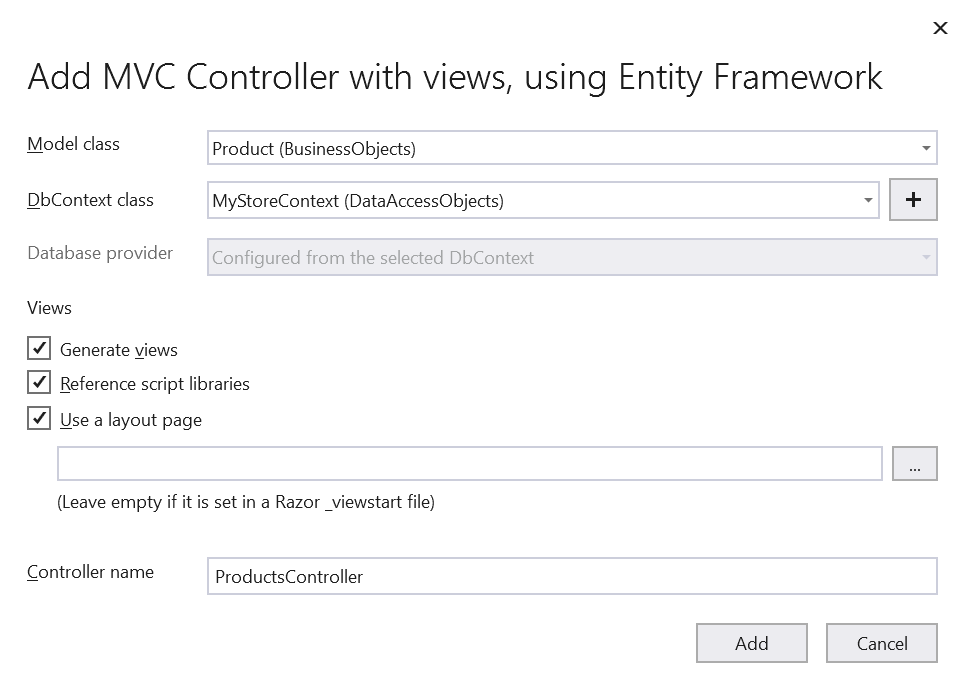
## **Step 04.** Create Controller

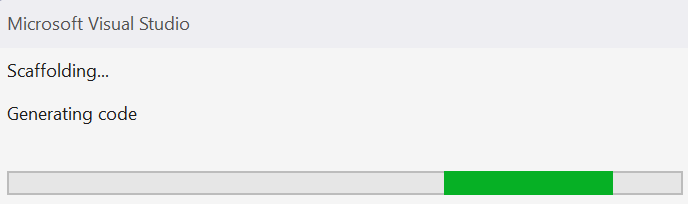
* Connect direct to the Data Access Layer (MyStoreContext.cs) to generate code
* Then
  + Add Dependency Injection
  + Change the code connects to the Service Layer

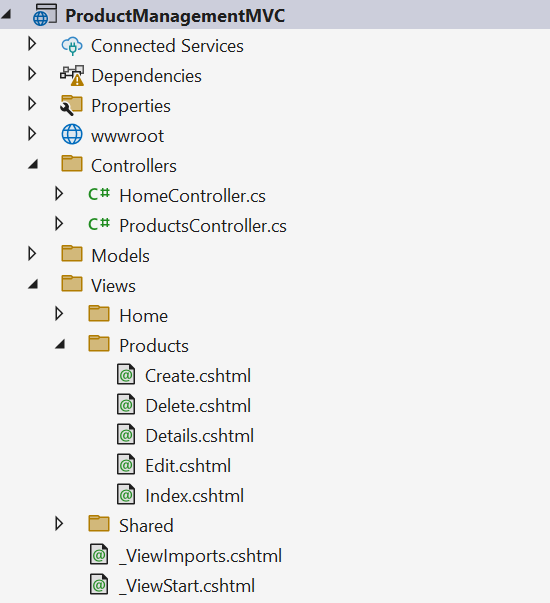
Add a Controller named ProductsController.cs





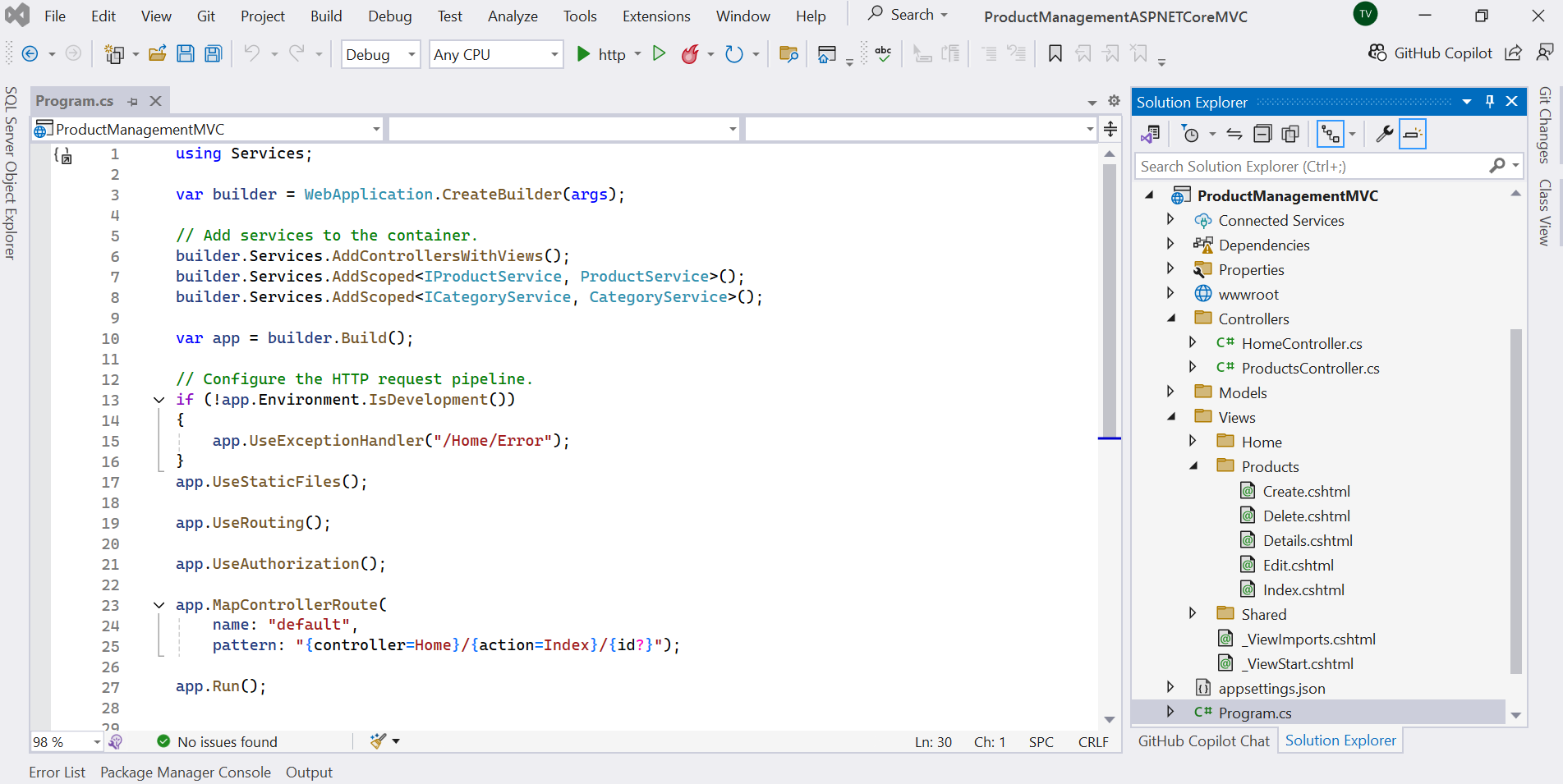




****

## **Step 05.** After generate the ProductsController.cs and View/Products/xxx.cshtml, change the code as the following

***Program.cs***



using Services;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.

builder.Services.AddControllersWithViews();

builder.Services.AddScoped<IProductService, ProductService>();

builder.Services.AddScoped<ICategoryService, CategoryService>();

var app = builder.Build();

// Configure the HTTP request pipeline.

if (!app.Environment.IsDevelopment())

{

app.UseExceptionHandler("/Home/Error");

}

app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

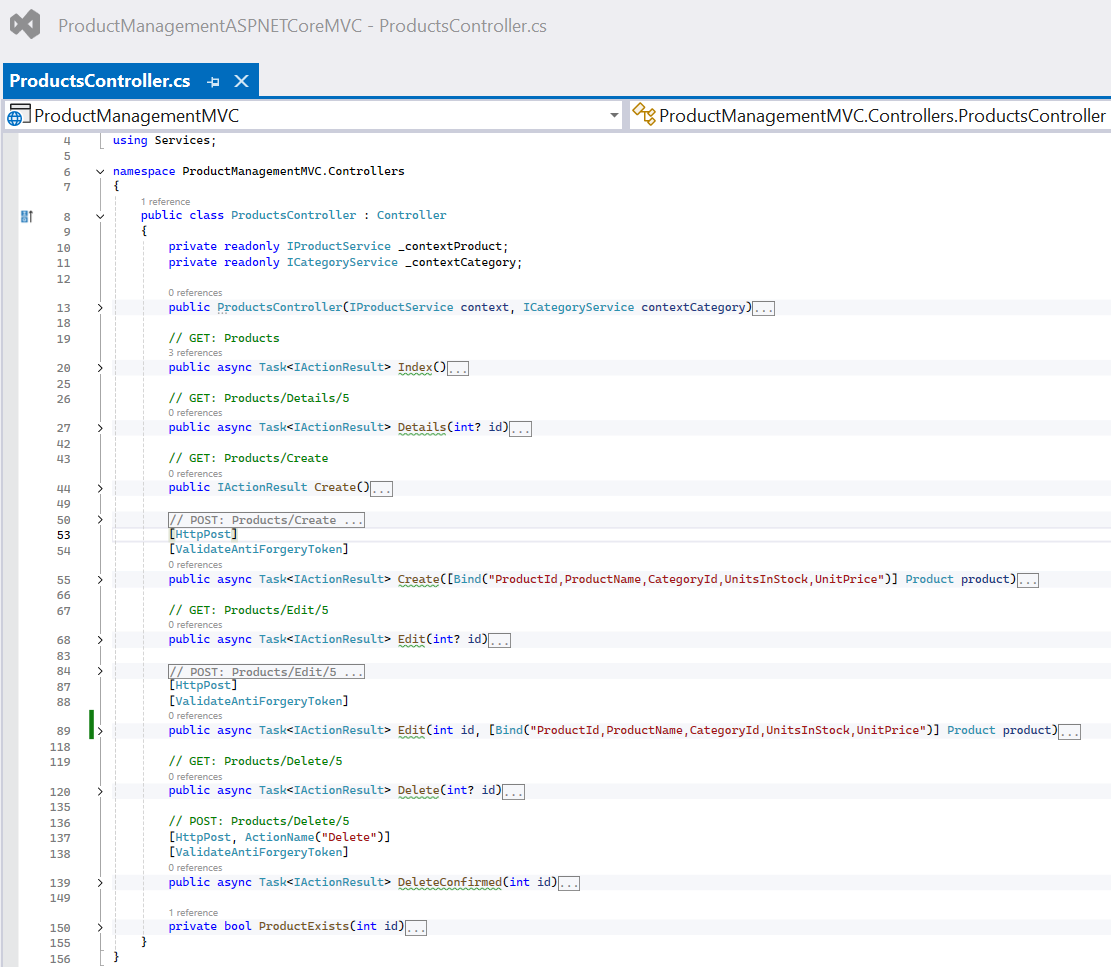
app.MapControllerRoute(

name: "default",

pattern: "{controller=Home}/{action=Index}/{id?}");

app.Run();

***ProductsController.cs***

******

*Use Dependency Injection for IProductService, ICategoryServiceFunction*

private readonly IProductService \_contextProduct;

private readonly ICategoryService \_contextCategory;

public ProductsController(IProductService contextProduct, ICategoryService contextCategory)

{

\_contextProduct = contextProduct;

\_contextCategory = contextCategory;

}

*Action method for get all information of products of ProductsController.cs (Method GET)*

public async Task<IActionResult> Index()

{

var myStoreContext = \_contextProduct.GetProducts();

return View(myStoreContext.ToList());

}

*Action method for get product by ID of ProductsController.cs (Method GET: Products/Detail/5)*

public async Task<IActionResult> Details(int? id)

{

if (id == null)

{

return NotFound();

}

var product = \_contextProduct.GetProductById((int)id);

if (product == null)

{

return NotFound();

}

return View(product);

}

*Action method for displaying the Create form of ProductsController.cs (Method GET: Products/Create)*

public IActionResult Create()

{

ViewData["CategoryId"] = new SelectList(\_contextCategory.GetCategories(), "CategoryId", "CategoryId");

return View();

}

*Action method for creating a new product of ProductsController.cs (Method POST: Products/Create)*

[HttpPost]

[ValidateAntiForgeryToken]

public async Task<IActionResult> Create([Bind("ProductId,ProductName,CategoryId,UnitsInStock,UnitPrice")] Product product)

{

if (ModelState.IsValid)

{

\_contextProduct.SaveProduct(product);

return RedirectToAction(nameof(Index));

}

ViewData["CategoryId"] = new SelectList(\_contextCategory.GetCategories(), "CategoryId", "CategoryId", product.CategoryId);

return View(product);

}

*Action method for opening a form to edit an existing product of ProductsController.cs (Method GET: Products/Edit/5)*

public async Task<IActionResult> Edit(int? id)

{

if (id == null)

{

return NotFound();

}

var product = \_contextProduct.GetProductById((int)id);

if (product == null)

{

return NotFound();

}

ViewData["CategoryId"] = new SelectList(\_contextCategory.GetCategories(), "CategoryId", "CategoryId", product.CategoryId);

return View(product);

}

*Action method for editing an existing product of ProductsController.cs (Method POST: Products/Edit/5)*

[HttpPost]

[ValidateAntiForgeryToken]

public async Task<IActionResult> Edit(int id, [Bind("ProductId,ProductName,CategoryId,UnitsInStock,UnitPrice")] Product product)

{

if (id != product.ProductId)

{

return NotFound();

}

if (ModelState.IsValid)

{

try

{

\_contextProduct.UpdateProduct(product);

}

catch (Exception)

{

if (!ProductExists(product.ProductId))

{

return NotFound();

}

else

{

throw;

}

}

return RedirectToAction(nameof(Index));

}

ViewData["CategoryId"] = new SelectList(\_contextCategory.GetCategories(), "CategoryId", "CategoryId", product.CategoryId);

return View(product);

}

*Action method for opening a confirmation to delete an existing product of ProductsController.cs (Method GET: Products/Delete/5)*

public async Task<IActionResult> Delete(int? id)

{

if (id == null)

{

return NotFound();

}

var product = \_contextProduct.GetProductById((int) id);

if (product == null)

{

return NotFound();

}

return View(product);

}

*Action method for deleting an existing product of ProductsController.cs (Method POST: Products/Delete/5)*

[HttpPost, ActionName("Delete")]

[ValidateAntiForgeryToken]

public async Task<IActionResult> DeleteConfirmed(int id)

{

var product = \_contextProduct.GetProductById(id);

if (product != null)

{

\_contextProduct.DeleteProduct(product);

}

return RedirectToAction(nameof(Index));

}

*Function for checking if the product is existing or not - of ProductsController.cs*

private bool ProductExists(int id)

{

var tmp = \_contextProduct.GetProductById(id);

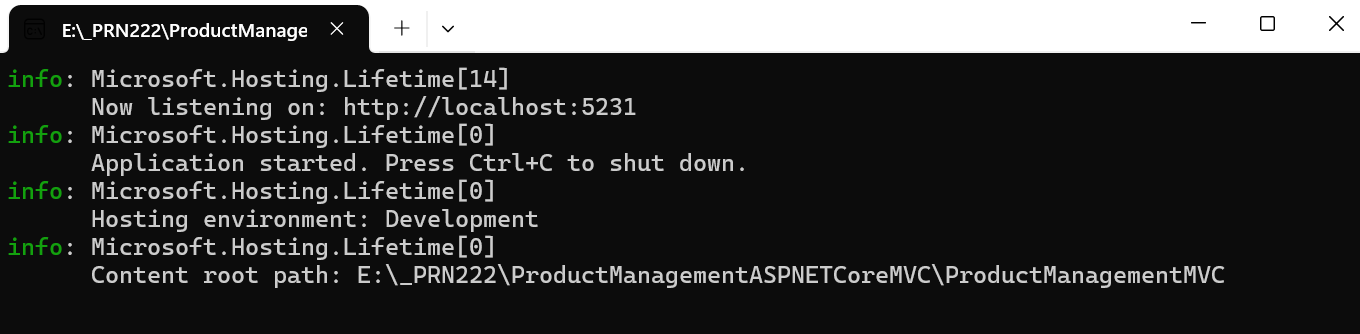
return (tmp != null)?true : false;

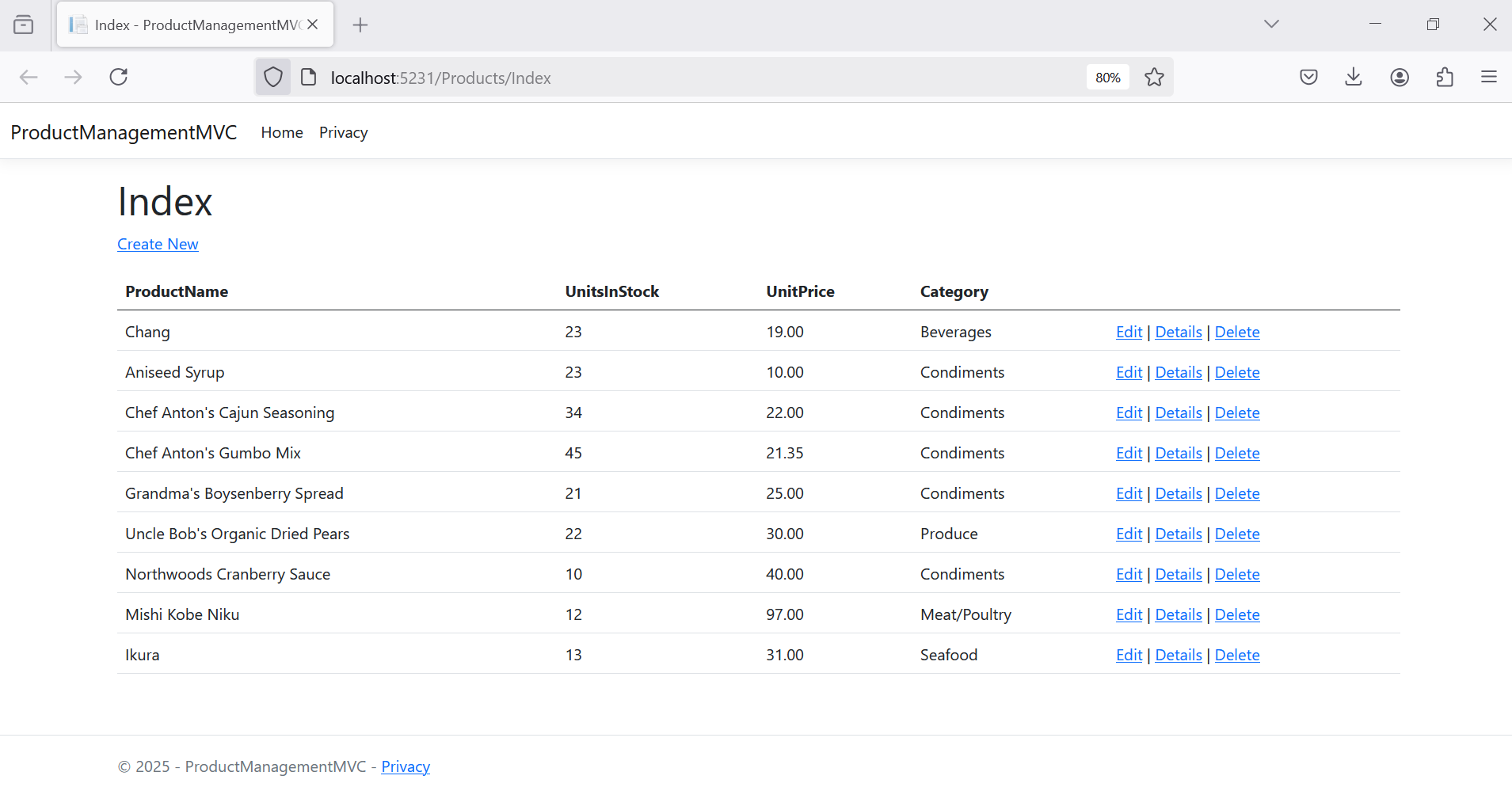
}

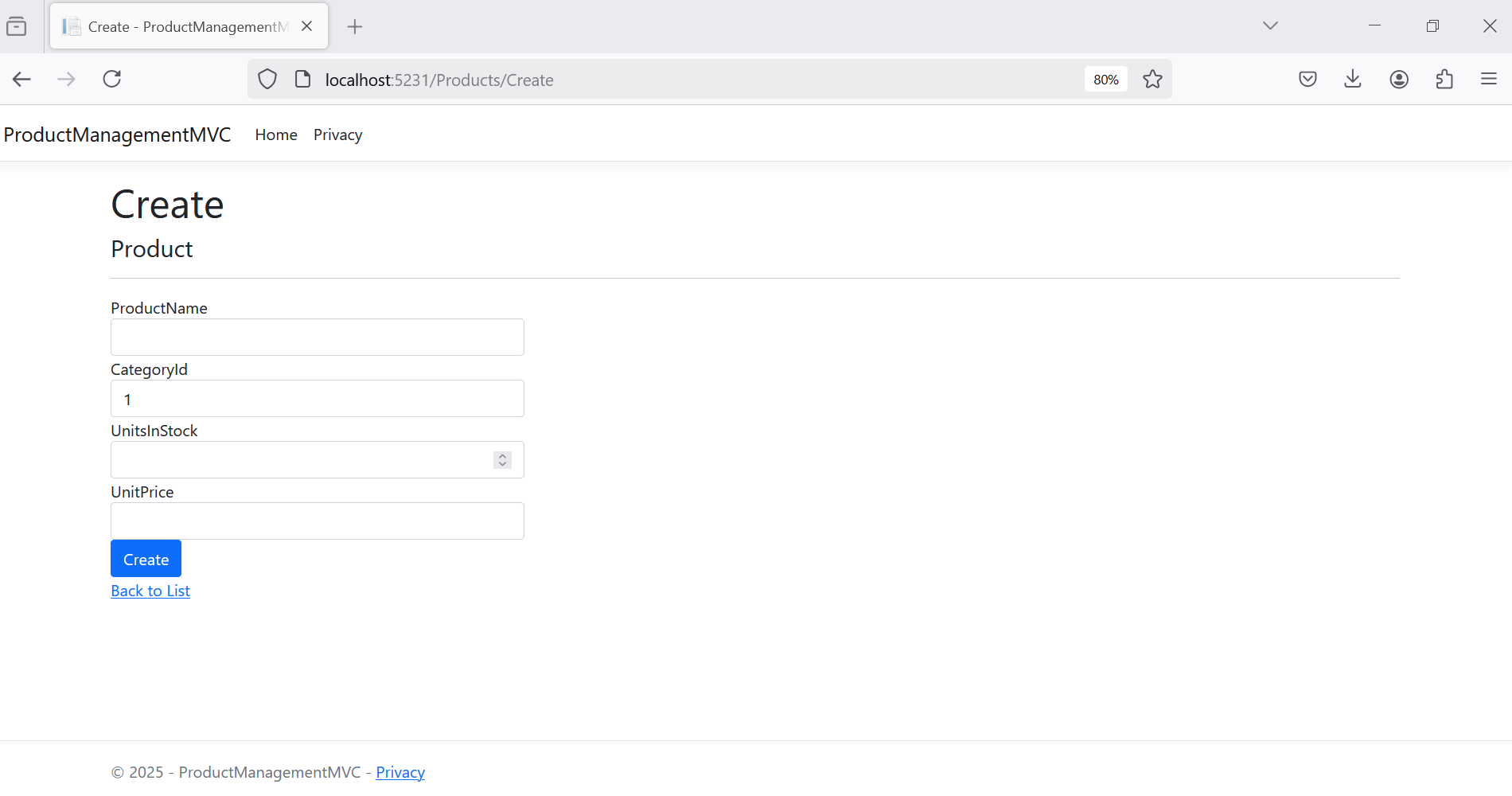
## **Step 06.** Change the View in the Razor View depend on your template.

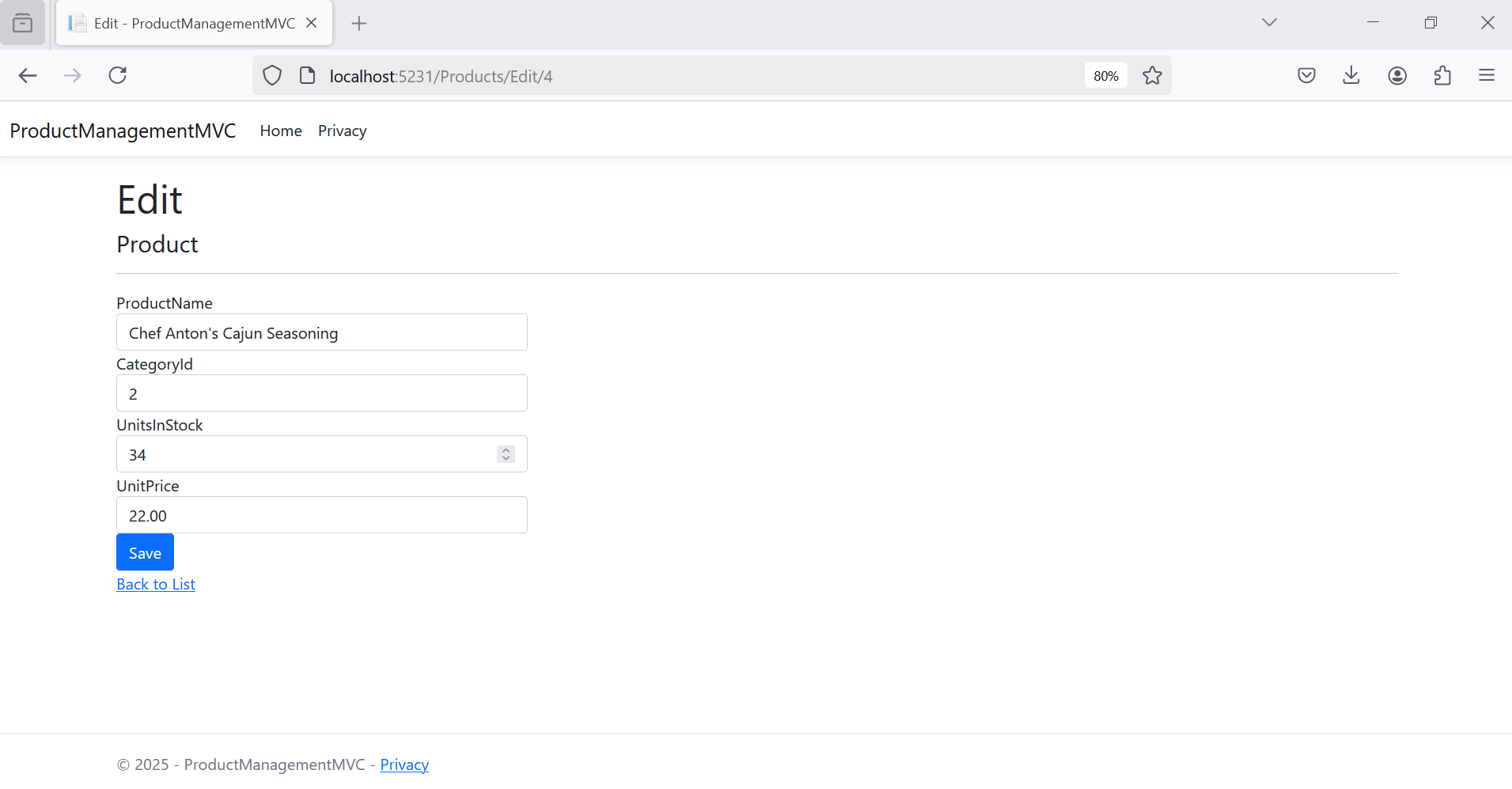
## **Step 07.** Clean ASP.NET Core Web App (Model-View-Controller) project, remove Entity Framework Core related packages.

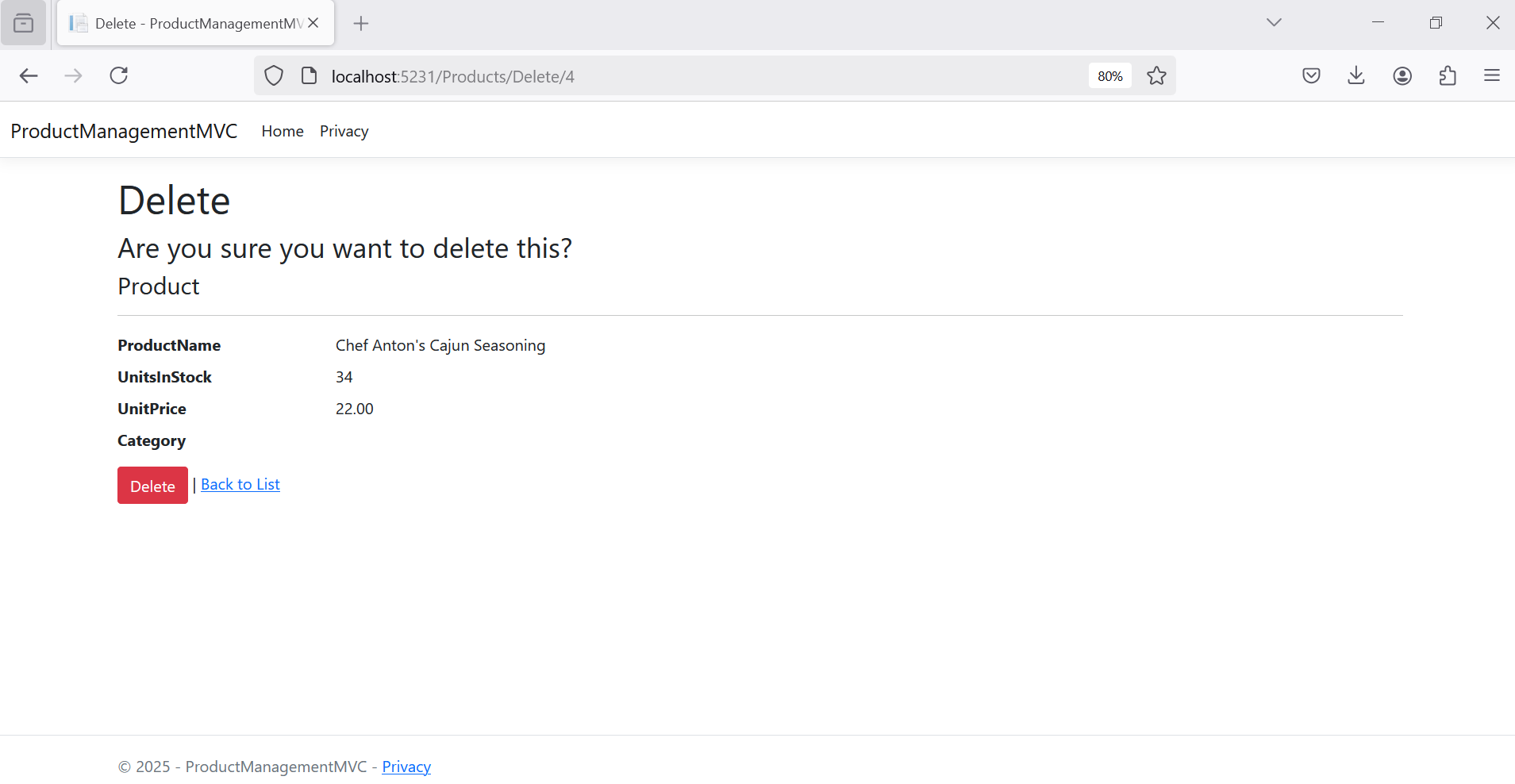
# Activity 07: Run the ASP.NET Core Web App (MVC) project and test all actions











# Activity 08: Session Management with ASP.NET Core Web App (Model-View-Controller)

## **Step 01.** Add the Session service in IoC Container of Kestrel Server, and then add the Middleware in HTTP request pipeline in Program.cs

using Services;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.

builder.Services.AddControllersWithViews();

builder.Services.AddScoped<IProductService, ProductService>();

builder.Services.AddScoped<ICategoryService, CategoryService>();

builder.Services.AddScoped<IAccountService, AccountService>();

builder.Services.AddSession(options =>

{

options.IdleTimeout = TimeSpan.FromMinutes(20); // Set session timeout

options.Cookie.HttpOnly = true; // For security

options.Cookie.IsEssential = true; // Ensure session cookie is always created

});

var app = builder.Build();

// Configure the HTTP request pipeline.

if (!app.Environment.IsDevelopment())

{

app.UseExceptionHandler("/Home/Error");

}

app.UseStaticFiles();

app.UseRouting();

app.UseSession();

app.UseAuthorization();

app.MapControllerRoute(

name: "default",

pattern: "{controller=Home}/{action=Index}/{id?}");

app.Run();

## **Step 02.** Create the AccountController and Login view

*Views/Account/Login.cshtml*

@model BusinessObjects.AccountMember

<h2>Login</h2>

<div class="row">

<div class="col-md-4">

<**form** **asp-action**="Login" method="post" **asp-controller**="Account">

<**div** **asp-validation-summary**="ModelOnly" class="text-danger"></**div**>

<div class="form-group">

<**label** **asp-for**="EmailAddress" class="control-label"></**label**>

<**input** **asp-for**="EmailAddress" class="form-control" />

<**span** **asp-validation-for**="EmailAddress" class="text-danger"></**span**>

</div>

<div class="form-group">

<**label** **asp-for**="MemberPassword" class="control-label"></**label**>

<**input** **asp-for**="MemberPassword" class="form-control" **type**="password" />

<**span** **asp-validation-for**="MemberPassword" class="text-danger"></**span**>

</div>

<div class="form-group">

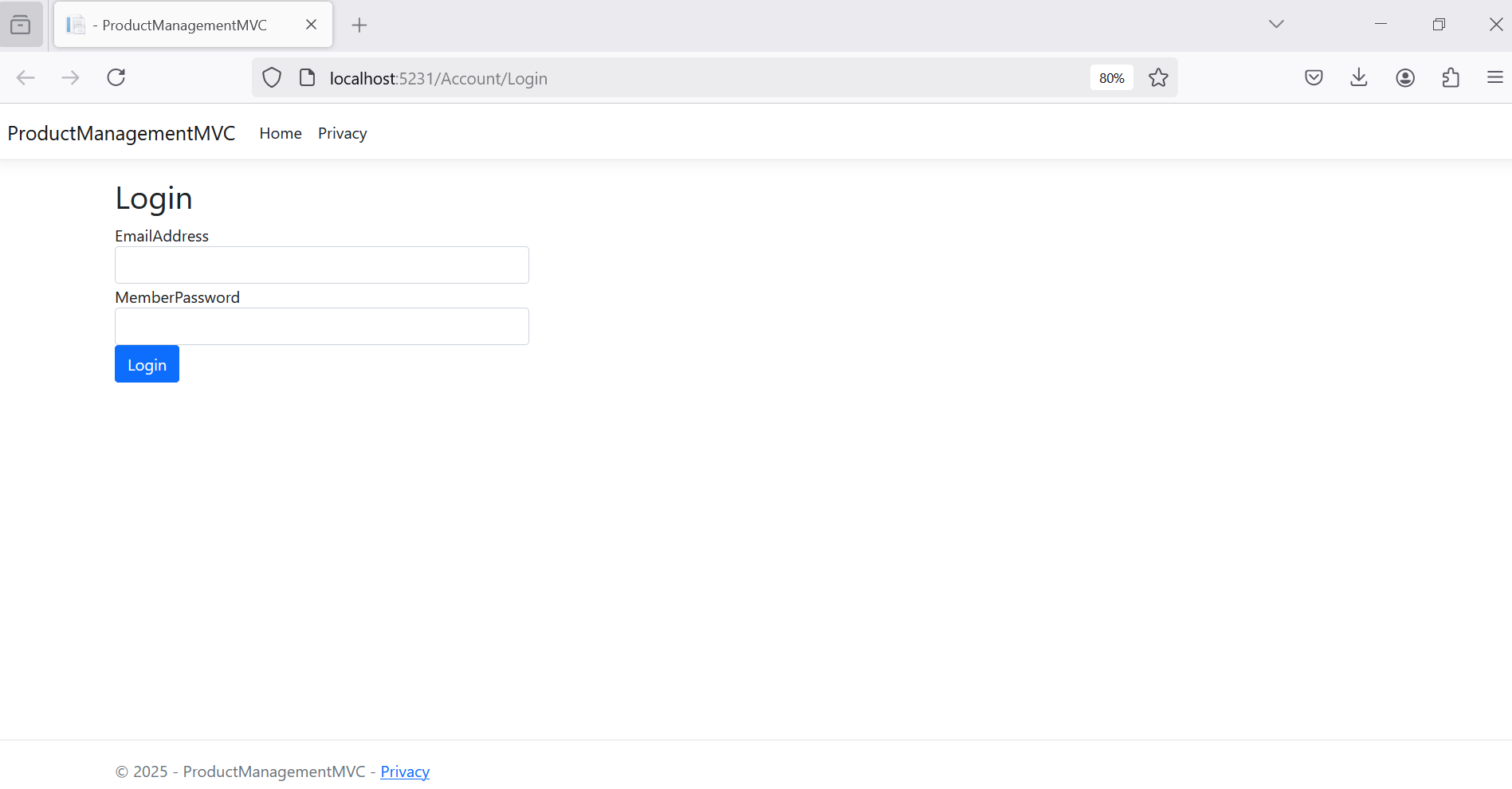
<input type="submit" value="Login" class="btn btn-primary" />

</div>

</**form**>

</div>

</div>



*Controllers/AccountController.cs*

public class AccountController : Controller

{

private readonly IAccountService \_accountService; // Inject your account service

public AccountController(IAccountService accountService)

{

\_accountService = accountService;

}

[HttpGet]

public IActionResult Login()

{

return View();

}

[HttpPost]

public IActionResult Login(AccountMember model)

{

if (ModelState.IsValid)

{

var user = \_accountService.GetAccountById(model.EmailAddress);

if (user != null && user.MemberPassword == model.MemberPassword)

{

// Store user information in session

HttpContext.Session.SetString("UserId", user.MemberId);

HttpContext.Session.SetString("Username", user.FullName);

return RedirectToAction("/Products/Index/", "Home"); // Redirect to home page

}

else

{

ModelState.AddModelError("", "Invalid username or password.");

}

}

return View(model);

}

public IActionResult Logout()

{

HttpContext.Session.Clear(); // Clear session data

return RedirectToAction("Login");

}

}

## **Step 03.** Accessing Session data in the ProductsController.cs.

*string userId = HttpContext.Session.GetString("UserId");*

*string username = HttpContext.Session.GetString("Username");*

public async Task<IActionResult> Index()

{

if (HttpContext.Session.GetString("UserId") == null)

{

// Redirect to the login page or display an error message

return RedirectToAction("Login", "Account");

}

var myStoreContext = \_contextProduct.GetProducts();

return View(myStoreContext.ToList());

}