Лабораторна робота №2

Тема: Робота з ASP.NET MVC. Конфігурація проекту. Відображення даних з БД на веб-сторінках.

Мета: Ознайомитися з архітектурою MVC, набути навичок створення головних компонентів архітектури MVC, навчитися конфігурувати підключення до БД та виконувати запити на читання даних, набути навичок роботи з представленнями (Views).

Хід роботи:

Завдання 1. Створення та конфігурація проекту

Створити проект, який реалізує архітектуру MVC, слідуючи інструкціям наведеним нижче.

Перш за все, ознайомтеся з темою вашого проекту, який ви розроблятимете протягом наступних лабораторних. Тема проекту визначається за номером по списку групи. Якщо вам не подобається тема, ви можете обрати свою, попередньо узгодивши її з викладачем. Варіанти наведено в таблиці 1.1.

7	Система для прокату транспорту	Користувачі можуть переглядати доступні транспорт (автомобілі, самокати, велосипеди тощо), робити бронювання та повертати транспортні засоби, а адміністратори керують автопарком.

Створіть проект послідовно виконавши наступні команди. Назви директорій та файлів повинні відповідати тематиці вашого проекту. Далі наведено приклади команд та коду для інтернет-магазину спортивних товарів SportsStore.

```
dotnet new globaljson --sdk-version 8.0 --output SportsSln/SportsStore
```

Ця команда створить global.json файл в директорії за шляхом SportsSln/SportsStore. Цей файл потрібен для уникнення проблем при роботі над одним проектом в команді. Він гарантує, що всі розробники, які працюють над проектом, використовують ту саму версію .NET SDK, запобігаючи проблемам із сумісністю.

Далі виконуємо команду:

```
dotnet new web --no-https --output SportsSln/SportsStore --framework net8.0
```

Ця команда створює новий проект типу ASP.NET Core minimal web API, опція --no-https відключає налаштування для HTTPS, це зручно для локальної розробки. Значення опції --output зі значенням SportsSln/SportsStore вказує, що проект слід створити в директорії за шляхом SportsSln/SportsStore. Опція --framework зі значенням net8.0 вказує, що проект має використовувати .NET8.

Наступна команда:

```
dotnet new sln -o SportsSln
```

Ця команда створює файл рішення (.sln) у каталозі SportsSln.

Далі виконуємо:

```
dotnet sln SportsSln add SportsSln/SportsStore
```

Ця команда додає проект SportsStore до файлу рішення SportsSln

Після виконання зазначених команд, отримаємо структуру файлів та директорій як на рисунку 1.1.

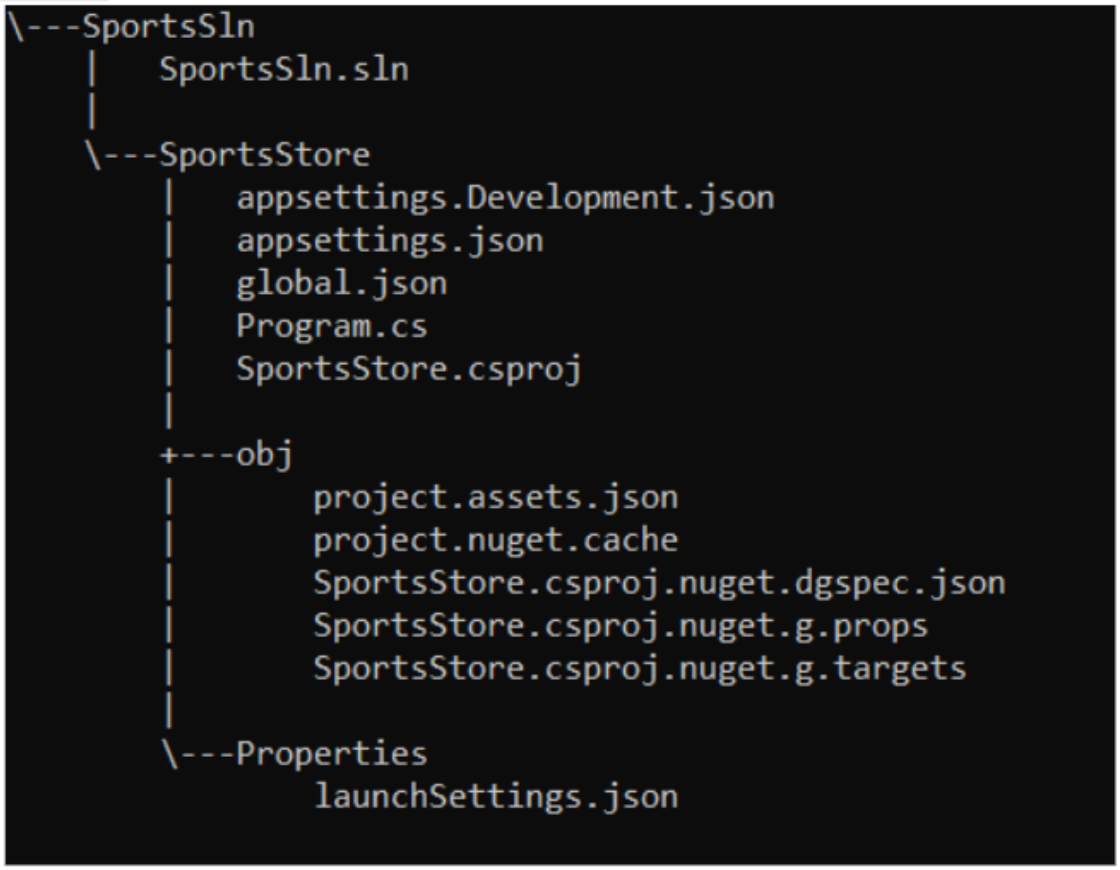


Рис. 1.1. Структура файлів і папок

Після цього, відкриваємо створений файл рішення в Visual Studio.

Альтернативно, можна створити проект та рішення використовуючи графічний інтерфейс Visual Studio з такими ж налаштуваннями як і в зазначених вище командах. Процес створення проекту з використанням графічного інтерфейсу наведено на рисунках 1.2.-1.4.

Щоб налаштувати HTTP-порт, який ASP.NET Core буде використовувати для прослуховування HTTP-запитів, внесіть зміни, показані наступному лістингу 7.4, до файлу launchSettings.json у папці Properties.

```
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:5000",
      "sslPort": 0
    }
  },
  "profiles": {
    "SportsStore": {
      "commandName": "Project",
      "dotnetRunMessages": true,
      "launchBrowser": true,
      "applicationUrl": "http://localhost:5000",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
```

```
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}
```

Наступним кроком створіть у вашому проекті папки з назвами, вказаними в таблиці 1.2.

Таблиця 1.2.

Назва	Опис
Models	Ця папка міститиме модель даних і класи, які забезпечують доступ до даних у базі даних додатка.
Controllers	Ця папка міститиме класи контролерів, які обробляють HTTP-запити.
Views	Ця папка міститиме всі файли Razor, згруповані в окремі підпапки.
Views/Home	Ця папка міститиме файли Razor, які специфічні для контролера Home. Я створю її в розділі «Створення контролера і представлення».
Views/Shared	Ця папка міститиме файли Razor, які є спільними для всіх контролерів.

Перейдемо до налаштування сервісів та пайплайну обробки запитів. Файл Program.cs використовується для налаштування додатку ASP.NET Core. Модифікуйте його так, як показано у наступному лістингу для налаштування базового функціоналу додатку.

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllersWithViews();

var app = builder.Build();

app.UseStaticFiles();

app.MapDefaultControllerRoute();

app.Run();
```

Коротко по основним моментам цього коду:

1. builder.Services – Реєструє сервіси, які можуть використовуватися у всьому додатку через dependency injection механізм.
2. AddControllersWithViews – Налаштовує сервіси, необхідні для MVC і представлень Razor.

3. UseStaticFiles – Дозволяє обслуговувати статичний контент (CSS, JS, зображення) з папки wwwroot
4. MapDefaultControllerRoute – Реєструє маршрути MVC за допомогою стандартної конвенції для зіставлення запитів із контролерами та методами.

Перейдемо до налаштування рушія представлень (View Engine) Razor. Razor відповідає за обробку файлів перегляду з розширенням .cshtml для створення відповідей з готовим для відображення HTML. Потрібна певне початкове налаштування, Razor, щоб полегшити створення представлень. Додайте файл імпорту Razor View з назвою _ViewImports.cshtml у папку Views із вмістом, показаним у лістингу:

```
@using SportsStore.Models
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

Важливо: Visual Studio показуватиме помилку в рядку "@using НазваПроекту.Models", але в наступних кроках це буде виправлено.

Додайте файл Razor View під назвою _ViewStart.cshtml до папки Views із наступним вмістом:

```
@{
      Layout = "_Layout";
}
```

Файл Razor View Start повідомляє Razor використовувати файл макета в HTML, який він створює, зменшуючи кількість дублікатів у представленнях. Щоб створити представлення, додайте макет Razor під назвою _Layout.cshtml до папки Views/Shared із вмістом, показаним у лістингу:

```
<!DOCTYPE html>
<html>
<head>
      <meta name="viewport" content="width=device-width" />
      <title>SportsStore</title>
</head>
<body>
      <div>
            @RenderBody()
      </div>
</body>
</html>
```

Додайте файл класу з назвою HomeController.cs у папку /Controllers і імплементуйте в ньому наступний клас:

```
using Microsoft.AspNetCore.Mvc;

namespace SportsStore.Controllers
{
      public class HomeController : Controller
      {
            public IActionResult Index() => View();
      }
}
```

Метод Index() поки не робить нічого корисного і просто повертає результат виклику методу View, який успадковується від базового класу Controller. Цей результат вказує ASP.NET Core відобразити представлення за замовчуванням, пов'язане з методом Index(). Щоб створити представлення, додайте файл Razor

View під назвою Index.cshtml до папки Views/Home із вмістом, показаним у лістингу:

```
<h4>Welcome to SportsStore</h4>
```

Далі перейдемо до створення першої сутності моделі даних. У випадку SportsStore це модель продукту, для вашого проекту оберіть одну із базових сутностей для першої моделі – новина, подія, сеанс тощо. Створіть файл з відповідною назвою в папці Models, приклад моделі наведено в лістингу:

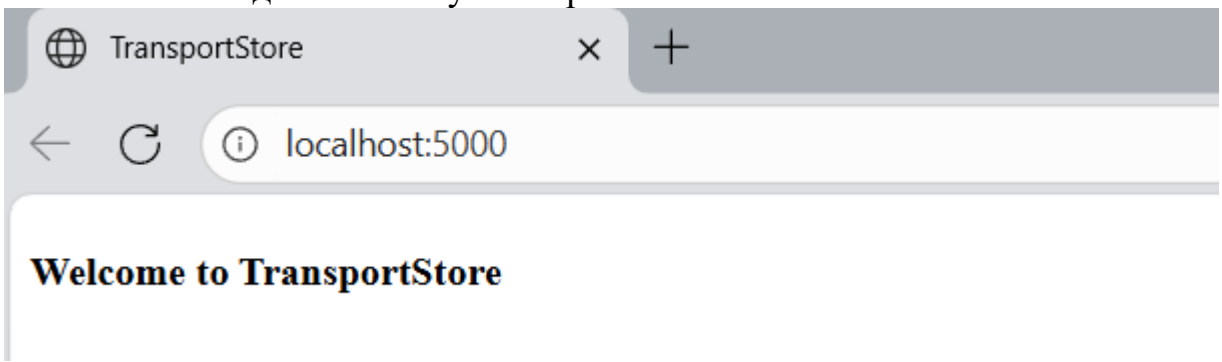
```
using System.ComponentModel.DataAnnotations.Schema;

namespace SportsStore.Models
{
      public class Product
      {
            public long? ProductID { get; set; }
            public string Name { get; set; } = String.Empty;
            public string Description { get; set; } = String.Empty;
            [Column(TypeName = "decimal(8, 2)")]
            public decimal Price { get; set; }
            public string Category { get; set; } = String.Empty;
      }
}
```

Після створення цього файлу, у представленні _ViewImports.cshtml повинна пропасти помилка.

Для властивості Price було використано атрибут Column для визначення типу даних SQL, який використовуватимуться для зберігання значень цієї властивості в БД. Не всі типи даних C# чітко корелюються з типами даних SQL, і цей атрибут гарантує, що база даних використовує правильний тип для даних.

Таким чином, виконавши всі попередні кроки, ми перетворили пустий WebAPI проект на проект, який реалізує паттерн MVC і повертає готові HTML-сторінки у відповіді на запити. Запустіть проект для перевірки працездатності. На рисунку 1.5. зображено результат запуску проекту SportsStore.

Виконав всі завдання та запустив проект:



Завдання 2. Робота з ORM

Налаштувати роботу з БД, використовуючи інструкції нижче.

Тепер наш проект містить деяке базове налаштування та може давати просту відповідь на запити. Тепер для реалізуємо роботу з даними. SportsStore зберігатиме свої дані в базі даних SQL Server LocalDB, доступ до якої здійснюється за допомогою Entity Framework Core. Entity Framework Core — це object-to-relational mapping (ORM) framework, і це найпоширеніший метод доступу до баз даних у проектах ASP.NET Core.

Для того щоб перевірити, чи встановлено у вас на ПК SQL Server LocalDB, можна виконати команду SqlLocalDB і в командному рядку. Якщо команда виконується без помилок, то SQL Server LocalDB встановлено. Також це можна перевірити через Visual Studio Installer. Для цього запустіть VS Installer, на поточній інсталяції натисніть “Modify” та перейдіть на вкладку “Individual Components”. В полі пошуку почніть вводити “SQL Server LocalDB” та перевірте результат. Якщо навпроти цієї опції стоїть галочка, то необхідний компонент встановлено, якщо ж ні – поставте галочку і проведіть встановлення цього компоненту (рис. 2.1.).
Наступним кроком буде додавання пакету Entity Framework Core до проекту. За допомогою командного рядка PowerShell виконайте команду, показану у наступному лістингу, у папці де знаходиться ваш проект (у випадку SportsStore це /SportsSln/SportsStore):

```
dotnet add package Microsoft.EntityFrameworkCore.Design
dotnet add package Microsoft.EntityFrameworkCore.SqlServer
```

Ці пакети встановлюють Entity Framework Core для роботи SQL Server. Entity Framework Core також потребує пакет інструментів, який включає інструменти командного рядка, необхідні для підготовки та створення баз даних для додатків ASP.NET Core. Для їхнього встановлення виконайте наступну команду (каталог, в якому вона виконуватиметься не важливий):

```
dotnet tool install --global dotnet-ef
```

Налаштування конфігурації, такі як рядки підключення до бази даних, зберігаються у файлах конфігурації JSON. Щоб визначити connection string до бази даних, яка буде використовуватися для збереження даних вашого проекту, додайте рядки, виділені у наступному лістингу, до файлу appsettings.json у теці проекту.

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings": {
"SportsStoreConnection":
"Server=(localdb)\\MSSQLLocalDB;Database=SportsStore;MultipleActiveResultSets=true
"
  }
}
```

Тут ми створюємо Connection string з назвою "SportsStoreConnection". В його значенні ми визначаємо підключення базу даних LocalDB під назвою SportsStore (Database=SportsStore;) та вмикаємо функцію множинного активного набору результатів (MultipleActiveResultSets=true), яка необхідна для деяких запитів до бази даних, що будуть зроблені додатком SportsStore з використанням Entity Framework Core. Оберіть назву бази даних у відповідності до тематики вашого проекту та вкажіть ці назви у вашій connection string.
Entity Framework Core надає доступ до бази даних через клас контексту. Для проекту SportsStore буде додано файл класу з ім'ям StoreDbContext.cs до папки Models. Оберіть назву класу контексту у відповідності до тематики вашого проекту. У цьому файлі визначимо клас, наведений в лістингу нижче.

```
using Microsoft.EntityFrameworkCore;
```

```
namespace SportsStore.Models
{
      public class StoreDbContext : DbContext
      {
            public   StoreDbContext(DbContextOptions<StoreDbContext>   options)   :
base(options) { }
            public DbSet<Product> Products => Set<Product>();
      }
}
```

Базовий клас DbContext надає доступ до базової функціональності ядра Entity Framework Core, а властивість Products забезпечить доступ до об'єктів Product у базі даних. Клас StoreDbContext є похідним від DbContext і додає властивості, які будуть використовуватися для читання та запису даних. Поки що маємо лише одну властивість, яка надаватиме доступ до об'єктів Product. Реалізуйте клас контексту відповідно до потреб вашого проекту.

Entity Framework Core має бути налаштований таким чином, щоб він знав тип бази даних, до якої він підключатиметься, який рядок підключення описує це підключення та який клас контексту представлятиме дані в базі даних. Лістинг нижче відображає необхідні зміни в Program.cs файлі для вищезгаданих налаштувань.

```
using Microsoft.EntityFrameworkCore;
using SportsStore.Models;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllersWithViews();

builder.Services.AddDbContext<StoreDbContext>(opts => {
      opts.UseSqlServer(builder.Configuration["ConnectionStrings:SportsStoreConnec
tion"]);
});

var app = builder.Build();

app.UseStaticFiles();

app.MapDefaultControllerRoute();

app.Run();
```

Інтерфейс IConfiguration надає доступ до системи конфігурації ASP.NET Core, яка включає в себе вміст файлу appsettings.json. Викликаючи builder.Configuration["ConnectionStrings:SportsStoreConnection")] ми звертаємось до властивості SportsStoreConnection об’єкту ConnectionStrings, який ми визначили раніше в файлі appsettings.json.

Entity Framework Core налаштовується за допомогою методу AddDbContext, який реєструє клас контексту бази даних і налаштовує підключення до бази даних. Метод UseSQLServer оголошує, що використовується SQL Server.

Наступним кроком буде створення інтерфейсу репозиторію та його реалізації. Паттерн репозиторію є одним з найпоширеніших, і він забезпечує зручний спосіб доступу до можливостей, представлених контекстним класом бази даних. Для реалізації репозиторію для SportsStore, створимо файл з назвою IStoreRepository.cs

(оберіть відповідну назву для вашого проекту) в папці Models і визначимо інтерфейс наступним чином:

```
namespace SportsStore.Models
{
      public interface IStoreRepository
      {
            IQueryable<Product> Products { get; }
      }
}
```

Цей інтерфейс використовує IQueryable<T>, щоб дозволити користувачеві отримати послідовність об'єктів Product. Інтерфейс IQueryable<T> є похідним від інтерфейсу IEnumerable<T> і представляє собою колекцію об'єктів, які можна запитувати, наприклад, об'єктів, керованих базою даних. Клас, який залежить від інтерфейсу IStoreRepository, може отримувати об'єкти Product без необхідності знати деталі того, як вони зберігаються або як клас реалізації буде їх отримує.

Щоб створити реалізацію інтерфейсу репозиторію, додамо файл класу з іменем EFStoreRepository.cs в папку Models і визначимо клас, показаний в лістингу:

```
namespace SportsStore.Models
{
      public class EFStoreRepository : IStoreRepository
      {
            private StoreDbContext context;
            public EFStoreRepository(StoreDbContext ctx)
            {
                  context = ctx;
            }
            public IQueryable<Product> Products => context.Products;
      }
}
```

На даний момент момент, реалізація репозиторію просто відображає властивість Products, визначену інтерфейсом IStoreRepository на властивість Products, визначену класом StoreDbContext. Властивість Products у контекстному класі повертає об'єкт DbSet<Product>, який реалізує інтерфейс IQueryable<T> і дозволяє легко реалізувати інтерфейс репозиторію при використанні Entity Framework Core.

ASP.NET Core підтримує сервіси, які дозволяють отримати доступ до об'єктів у всьому додатку. Однією з переваг сервісів є те, що вони дозволяють класам використовувати інтерфейси без необхідності знати, який клас реалізації використовується. Це означає, що компоненти програми можуть отримувати доступ до об'єктів, які реалізують інтерфейс IStoreRepository, не знаючи, що вони використовують клас реалізації EFStoreRepository. Це дозволяє легко змінювати клас реалізації, який використовується у додатку, без необхідності вносити зміни в окремі компоненти. Змінимо Program.cs таким чином, щоб створити сервіс для інтерфейсу IStoreRepository, який використовує EFStoreRepository як клас реалізації.

```
using Microsoft.EntityFrameworkCore;
using SportsStore.Models;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllersWithViews();
```

```
builder.Services.AddDbContext<StoreDbContext>(opts => {
      opts.UseSqlServer(builder.Configuration["ConnectionStrings:SportsStor
eConnection"]);
});

builder.Services.AddScoped<IStoreRepository, EFStoreRepository>();

var app = builder.Build();

app.UseStaticFiles();

app.MapDefaultControllerRoute();

app.Run();
```

Метод AddScoped створює сервіс таким чином, що кожен HTTP-запит отримує власний об'єкт репозиторію, що є типовим способом використання Entity Framework Core.

Entity Framework Core може генерувати схему бази даних, використовуючи класи моделі даних, за допомогою механізму міграцій. Коли ви розробляєте міграцію, Entity Framework Core створює клас C#, який містить команди SQL, необхідні для модифікації бази даних. Якщо вам потрібно змінити класи моделі, тоді ви можете створити нову міграцію, яка містить команди SQL, необхідні для відображення змін в базі даних. Таким чином, можна не турбуватися про ручне написання і тестування SQL-скриптів і ви можете зосередитися на класах моделі C# у додатку.

Команди Entity Framework Core виконуються з командного рядка. Відкрийте командний рядок PowerShell і виконайте команду, показану в лістингу нижче, у папці вашого проекту (для SportsStore це SportsSln/SportsStore), щоб створити клас міграції, який підготує базу даних до її першого використання.

```
dotnet ef migrations add Initial
```

Наповніть вашу БД початковими даними за зразком нижче.

Щоб наповнити БД даними одразу після створення, реалізуємо статичний клас SeedData в папці Models. Лістинг файлу SeedData.cs:

```
using Microsoft.EntityFrameworkCore;
using static System.Net.Mime.MediaTypeNames;

namespace SportsStore.Models
{
      public static class SeedData
      {
            public static void EnsurePopulated(IApplicationBuilder app)
            {
                  StoreDbContext context = app.ApplicationServices
                        .CreateScope().ServiceProvider
                        .GetRequiredService<StoreDbContext>();

                  if (context.Database.GetPendingMigrations().Any())
                  {
                        context.Database.Migrate();
                  }

                  if (!context.Products.Any())
                  {
                        context.Products.AddRange(
```

```
                    new Product
                    {
                          Name = "Kayak",
                          Description = "A boat for one person",
                          Category = "Watersports",
                          Price = 275
                    },
                    new Product
                    {
                          Name = "Lifejacket",
                          Description = "Protective and fashionable",
                          Category = "Watersports",
                          Price = 48.95m
                    },
                    new Product
                    {
                          Name = "Soccer Ball",
                          Description = "FIFA-approved size and weight",
                          Category = "Soccer",
                          Price = 19.50m
                    },
                    new Product
                    {
                          Name = "Corner Flags",
                          Description = "Give your playing field a professional
touch",
                          Category = "Soccer",
                          Price = 34.95m
                    },
                    new Product
                    {
                          Name = "Stadium",
                          Description = "Flat-packed 35,000-seat stadium",
                          Category = "Soccer",
                          Price = 79500
                    },
                    new Product
                    {
                          Name = "Thinking Cap",
                          Description = "Improve brain efficiency by 75%",
                          Category = "Chess",
                          Price = 16
                    },
                    new Product
                    {
                          Name = "Unsteady Chair",
                          Description = "Secretly give your opponent a
disadvantage",
                          Category = "Chess",
                          Price = 29.95m
                    },
                    new Product
                    {
                          Name = "Human Chess Board",
                          Description = "A fun game for the family",
                          Category = "Chess",
                          Price = 75
                    },
                    new Product
                    {
                          Name = "Bling-Bling King",
                          Description = "Gold-plated, diamond-studded King",
                          Category = "Chess",
```

```
                    Price = 1200
                });
                context.SaveChanges();
            }
        }
    }
}
```

Статичний метод EnsurePopulated отримує як аргумент IApplicationBuilder, який є інтерфейсом, що використовується у файлі Program.cs для реєстрації middleware для обробки HTTP-запитів. IApplicationBuilder також надає доступ до сервісів додатку, включаючи службу контексту бази даних Entity Framework Core.
Метод EnsurePopulated отримує об'єкт StoreDbContext через інтерфейс IApplicationBuilder і викликає метод Database.Migrate, якщо є невиконані міграції. Далі перевіряється кількість об'єктів типу Product в базі даних. Якщо в базі даних немає об'єктів, то база даних заповнюється колекцією об'єктів Product за допомогою методу AddRange, а потім записується в базу даних за допомогою методу SaveChanges.
Останнім кроком є заповнення бази даних при запуску додатку. Для цього додамо виклик методу EnsurePopulated в Program.cs, як як показано в лістингу:

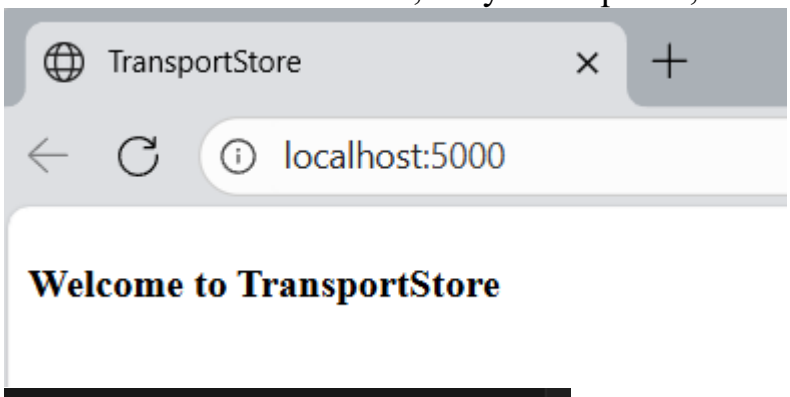
```
using Microsoft.EntityFrameworkCore;
using SportsStore.Models;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllersWithViews();

builder.Services.AddDbContext<StoreDbContext>(opts => {
     opts.UseSqlServer(builder.Configuration["ConnectionStrings:SportsStor
eConnection"]);
});

builder.Services.AddScoped<IStoreRepository, EFStoreRepository>();

var app = builder.Build();

app.UseStaticFiles();

app.MapDefaultControllerRoute();

SeedData.EnsurePopulated(app);

app.Run();
```

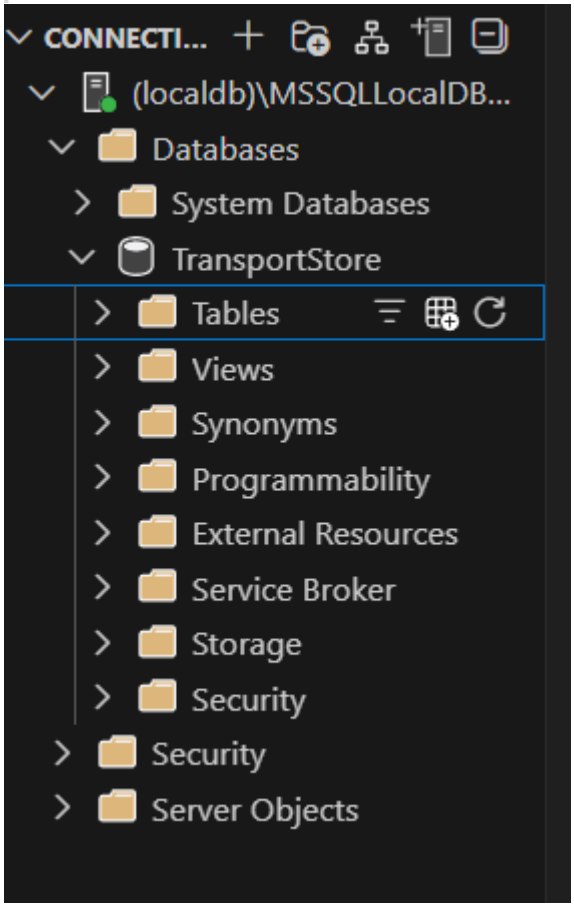
Якщо вам потрібно скинути базу даних, то виконайте цю команду в папці проекту замінивши значення оцпії --context на назву вашого файлу контексту:

```
dotnet ef database drop --force --context StoreDbContext
```

Після чого запустіть проект, і база даних буде перестворена і заповнена даними.

Після виконання вказівок, запустив проєкт, та згерерувалася база даних:





Завдання 3. Виведення даних на сторінці

Змініть контролер представлення для виведення даних з БД.

Для SportStore це виглядатиме наступним чином:

Лістинг файлу HomeController.cs

```
using Microsoft.AspNetCore.Mvc;
using SportsStore.Models;

namespace SportsStore.Controllers
{
      public class HomeController : Controller
      {
            private IStoreRepository repository;

            public HomeController(IStoreRepository repo)
            {
                  repository = repo;
```

```
            }

            public IActionResult Index() => View(repository.Products);
      }
}
```

Лістинг файлу Views/Home/Index.cshtml

```
@model IQueryable<Product>

@foreach (var p in Model ?? Enumerable.Empty<Product>())
{
      <div>
            <h3>@p.Name</h3>
            @p.Description
            <h4>@p.Price.ToString("c")</h4>
      </div>
}
```

Після запуску проекту отримаємо результат зображений на рисунку 3.1.

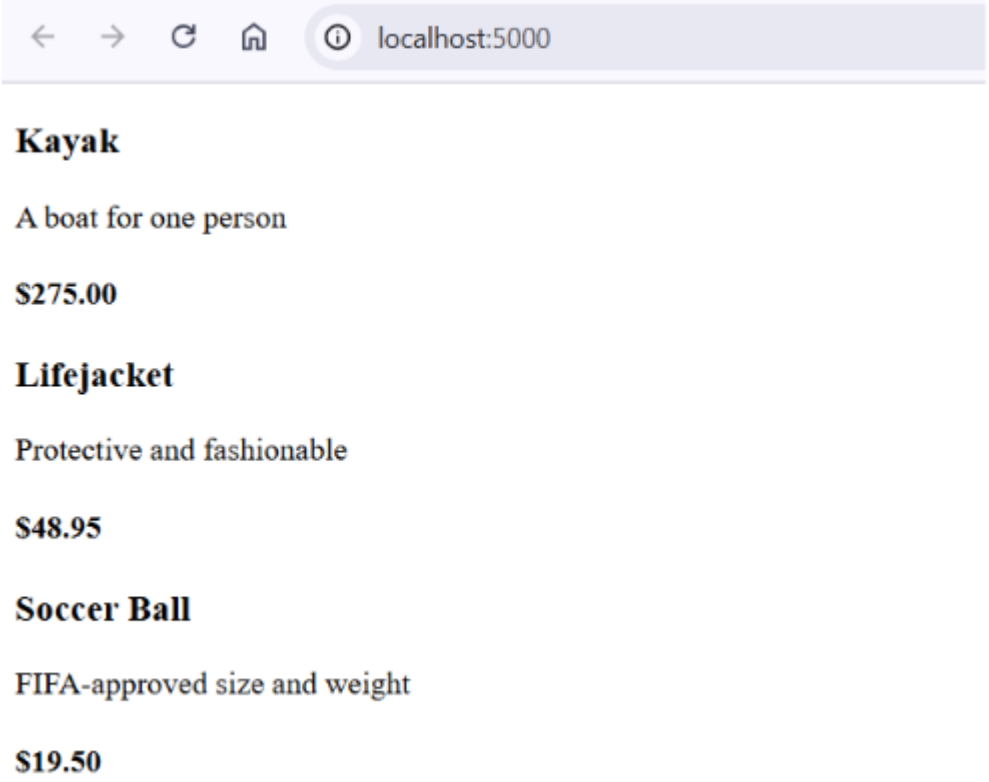
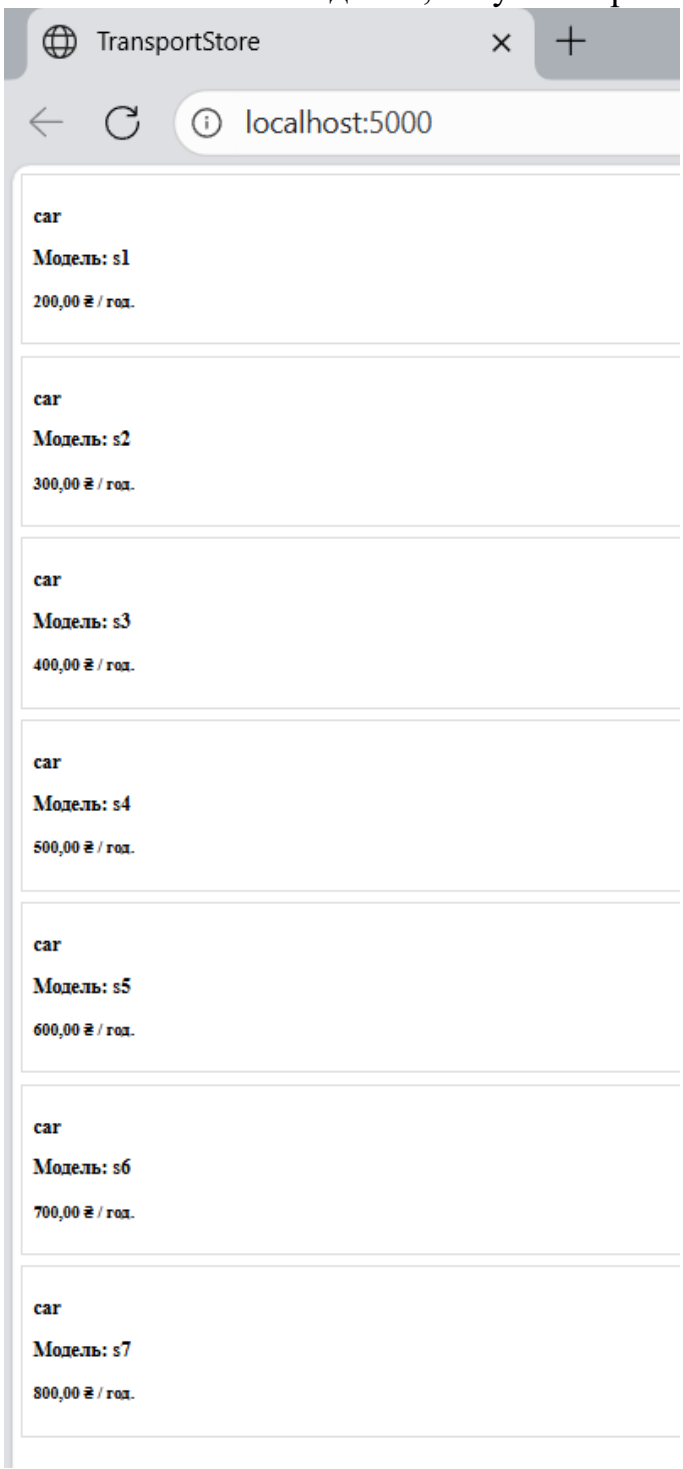


Рис. 3.1. Виведення даних на представлення

Після виконання завдання, запустив проєк та отримав такий результат:

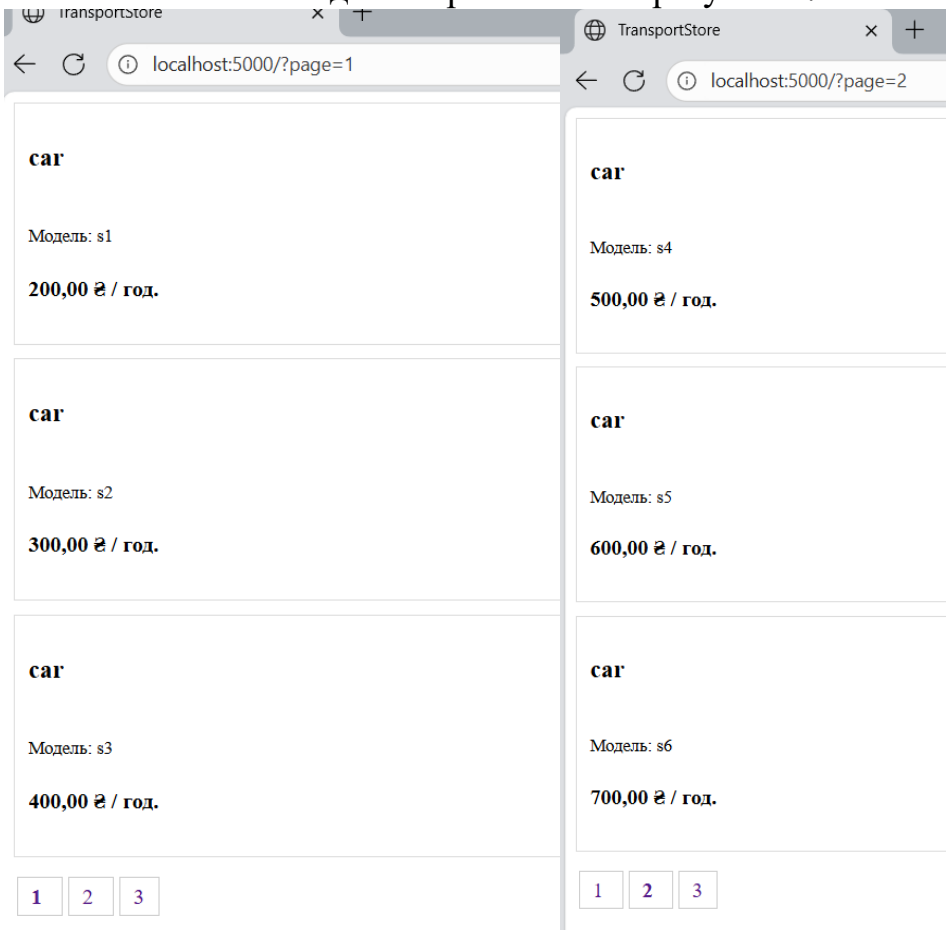


Завдання 4. Реалізуйте посторінковий вивід записів.

1. У папці Models створіть підпапку ViewModels. View Model – це сутність, створена спеціально для передачі даних між контролером та представленням, яка створюється під час роботи додатку і не зберігається в БД.
2. В папці ViewModels створіть клас PagingInfo для збереження даних про пейджинг (кількість записів на сторінці, загальна кількість записів, кількість сторінок тощо).

3. В папці ViewModels створіть View Model для посторінкового відображення списку сутностей. Вона повинна включати в собі колекцію сутностей та інформацію про пейджинг.
4. Додайте підтримку пейджингу в контролер. Для цього змініть метод Index таким чином, щоб він приймав номер поточної сторінку та реалізуйте в ньому запит до БД для отримання елементів саме для цієї сторінки. У представлення передавайте екземпляр створеної View Model на кроці номер 3.
5. Адаптуйте представлення для відображення списку продуктів з View Model.
6. Додайте на сторінку елемент для вибору поточної сторінки.

Після виконання завдань отримав такий результат:



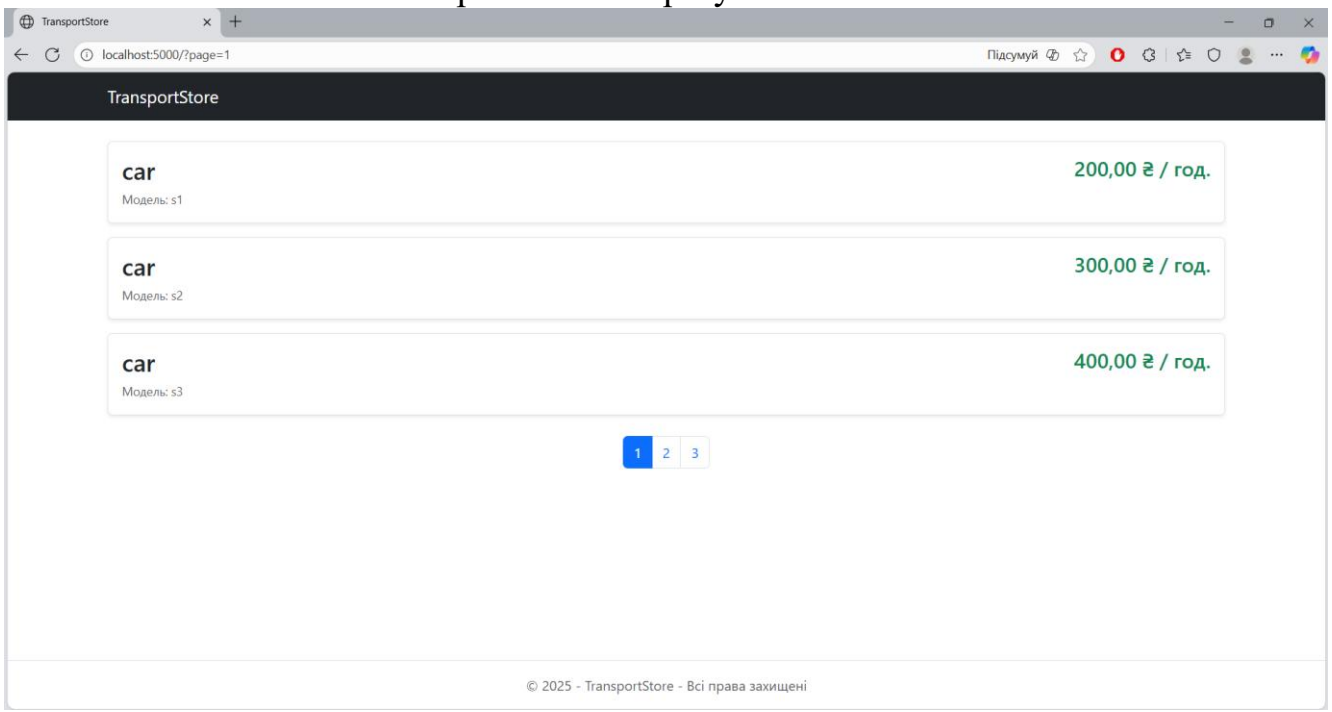
Завдання 5. Стилізація

1. Встановіть фреймворк Bootstrap, виконавши команди в папці проекту:

```
dotnet tool install --global Microsoft.Web.LibraryManager.Cli
libman init -p cdnjs
libman install bootstrap -d wwwroot/lib/bootstrap
```

2. Змініть базовий шаблон сторінки (_Layout.cshtml). Додайте елементи для хедера та футера.
3. Стилізуйте список сутностей та пейджинг.

Після виконнаня вказівок отримав такий результат:



Висновок: ознайомився з архітектурою MVC, набув навичок створення головних компонентів архітектури MVC, навчився конфігурувати підключення до БД та виконувати запити на читання даних, набув навичок роботи з представленнями (Views).