ЗАТВЕРДЖЕНО

Науково-методичною радою
Державного університету
«Житомирська політехніка»

протокол від 19 травня 2026 р.
№ 4

МЕТОДИЧНІ РЕКОМЕНДАЦІЇ
для проведення лабораторних занять з навчальної дисципліни «Інтернет-програмування на ASP.NET (C#)»

для здобувачів вищої освіти освітнього ступеня «бакалавр»
спеціальності 122 «Комп’ютерні науки»
освітньо-професійна програма «Комп’ютерна графіка та розробка ігор»
факультет інформаційно-комп’ютерних технологій
кафедра комп’ютерних наук

Рекомендовано на засіданні кафедри комп’ютерних наук
___ _____ 20__ р.,
протокол № ___

Розробники: ст. викладач кафедри комп’ютерних наук БЕЙРАК Дмитро,
викладач кафедри комп’ютерних наук УКРАЇНЕЦЬ Микола

Житомир
2026

ЗМІСТ

Лабораторна робота №1

Тема: Вступ до ASP.NET Core

Мета: ознайомитися з основними принципами роботи .NET, навчитися налаштовувати середовище розробки та встановлювати необхідні компоненти, набути навичок створення рішень та проектів різних типів, набути навичок обробки запитів з використанням middleware.

Теоретичні відомості

ASP.NET — це популярна безкоштовна платформа з відкритим вихідним кодом від Microsoft для створення сучасних вебсайтів, веб-застосунків і API на основі Microsoft .NET. Вона дає можливість розробникам використовувати мови C# або F# для створення динамічних веб-сторінок і сервісів, застосовуючи різні підходи та технології, зокрема MVC, Razor Pages, Blazor і SignalR. Актуальна версія платформи — ASP.NET Core — є кросплатформеною та підтримує роботу на Windows, macOS і Linux.

Платформа орієнтована на сучасну веб-розробку та підходить для створення як звичайних веб-застосунків, так і високопродуктивних API та рішень із підтримкою реального часу (наприклад, через WebSockets). ASP.NET Core забезпечує високу швидкодію та масштабованість, що дозволяє використовувати її для проектів із великим навантаженням. Вона має модульну архітектуру, розвинену систему інструментів і бібліотек, підтримує Razor для генерації динамічного вмісту, MVC для чіткої структуризації коду, а також вбудовані механізми автентифікації, авторизації та захисту даних. Крім того, платформа добре інтегрується з хмарними сервісами, зокрема з Microsoft Azure.

Серед основних моделей і компонентів виділяють ASP.NET Core MVC, що реалізує шаблон «Модель–Представлення–Контролер» для створення складних інтерфейсів і API, Razor Pages — спрощений підхід, орієнтований на сторінки, а також Blazor — технологію для розробки інтерактивних

вебінтерфейсів із використанням C# замість JavaScript. ASP.NET Core є наступником класичного ASP.NET у складі .NET Framework і пропонує покращену продуктивність, гнучкість та сучасну архітектуру.

.NET CLI (dotnet) — це командний інтерфейс платформи Microsoft .NET, який дозволяє створювати проекти, керувати залежностями, компілювати код, запускати програми, публікувати застосунки. CLI працює в терміналі (Command Prompt, PowerShell, Bash тощо).

Для створення проекту використовується команда `dotnet new <тип_проекту>`. Найпоширеніші шаблони:

Таблиця 1.1 — Поширені шаблони проектів

Команда	Тип проекту
`dotnet new console`	Консольний застосунок
`dotnet new web`	ASP.NET Core Web
`dotnet new mvc`	ASP.NET Core MVC
`dotnet new classlib`	Бібліотека класів
`dotnet new sln`	Solution-файл

Створення проекту з назвою відбувається із зазначення відповідного параметру: dotnet new console -n MyApp. Після створення буде отримано наступну структуру:

- .csproj — файл конфігурації проекту
- Program.cs — точка входу
- папка obj — службові файли
- папка bin — скомпільовані файли

Так як зазвичай проекти .NET існують в рамках рішення (solution), створити його та додати до нього проект можна наступними командами: dotnet new sln -n MySolution та dotnet sln add MyApp/MyApp.csproj.

Наведемо перелік часто вживаних команд:

Таблиця 1.2 — Часто вживані команди dotnet

Команда	Пояснення
`dotnet sln list`	Перегляд проектів у solution
`dotnet restore`	Відновлення залежностей (завантажує NuGet-пакети, зазначені у .csproj)
`dotnet build -c Release`	Збірка (компіляція) проекту із зазначеною кофігурацією (за замовчуванням: Debug)
`dotnet run`	Збірка та запуск проекту
`dotnet add package Newtonsoft.Json`	Додавання NuGet-пакету
`dotnet remove package Newtonsoft.Json`	Видалення NuGet-пакету
`dotnet publish`	Публікація застосунку
`dotnet --version`	Перевірка версії .NET
`dotnet new list`	Список доступних для створення проекту шаблонів
`dotnet clean`	Очищення збірки
`dotnet test`	Запуск тестів
`dotnet --info`	Детальна інформація про SDK

NuGet — це менеджер пакетів для платформи Microsoft .NET, який використовується для підключення, оновлення та керування зовнішніми бібліотеками у проектах .NET. Він значно спрощує повторне використання коду та роботу із залежностями. Завдяки NuGet розробники можуть повторно використовувати вже створені рішення, що суттєво пришвидшує процес розробки та зменшує потребу писати код «з нуля».

Основною одиницею в NuGet є пакет — спеціальний файл формату .nupkg, який містить скомпільовані бібліотеки, метадані про версію, автора та опис, а також інформацію про залежності від інших пакетів. Коли розробник додає пакет до проекту, система автоматично завантажує всі необхідні компоненти та інтегрує їх у конфігураційний файл проекту. Це дозволяє уникнути ручного керування файлами бібліотек і мінімізує ризик помилок, пов’язаних із несумісністю версій.

Пакети NuGet зберігаються у спеціальних репозиторіях. Найбільшим публічним джерелом є nuget.org, де розміщено тисячі відкритих бібліотек для різноманітних завдань — від роботи з базами даних до реалізації складних алгоритмів або інтеграції з хмарними сервісами. Окрім публічних репозиторіїв, організації можуть створювати власні приватні сховища для розповсюдження внутрішніх бібліотек. Після завантаження пакети зберігаються у глобальному кеші на комп’ютері користувача, що дозволяє перевикористовувати їх без повторного завантаження.

Важливою особливістю NuGet є підтримка керування версіями. Розробник може вказати конкретну версію бібліотеки або задати діапазон допустимих версій. Це забезпечує стабільність проекту та контроль над оновленнями. У разі потреби пакет можна легко оновити або змінити його версію за допомогою командного інтерфейсу .NET або інтегрованих засобів середовищ розробки, таких як Microsoft Visual Studio чи Visual Studio Code.

Крім використання готових рішень, NuGet дає можливість створювати власні пакети. Це особливо важливо для командної розробки, коли спільні модулі або бібліотеки потрібно повторно використовувати в кількох проектах. Створений пакет можна поширювати всередині організації або опублікувати у відкритому доступі.

Middleware (проміжне програмне забезпечення) — це програмний компонент, який розташовується між клієнтським запитом і кінцевою

обробкою цього запиту застосунком. Іншими словами, middleware — це набір послідовних обробників, через які проходить кожен HTTP-запит і відповідь перед тим, як буде сформовано остаточний результат.

У середовищі ASP.NET Core middleware є ключовим елементом архітектури. Обробка запитів організована у вигляді конвеєра (pipeline), де кожен компонент виконує певну функцію: аналізує запит, може змінювати його, виконувати додаткову логіку або передавати далі наступному компоненту. Після обробки запиту формується відповідь, яка також проходить через цей конвеєр у зворотному порядку.

Основна ідея middleware полягає в модульності та розділенні відповідальності. Кожен компонент виконує лише одну конкретну задачу, наприклад аутентифікацію користувача, авторизацію доступу, логування запитів, обробку помилок, маршрутизацію, доповнення відповіді тощо. Завдяки такому підходу застосунок стає гнучким, а його функціональність можна легко розширювати або змінювати.

У Microsoft .NET middleware підключаються у певному порядку під час конфігурації застосунку. Порядок має велике значення, оскільки кожен компонент впливає на подальший процес обробки. Якщо, наприклад, спочатку не виконати перевірку аутентифікації, то захищені ресурси можуть стати доступними тим користувачам, що не повинні мати до них доступ. Правильна побудова конвеєра обробки запитів є важливою частиною проектування веб-застосунку.

Middleware можуть бути як вбудованими, так і власними (створеними розробниками). Платформа надає стандартні компоненти для роботи зі статичними файлами, обробки винятків, маршрутизації та забезпечення безпеки. Водночас розробник може створити власний middleware для реалізації специфічної бізнес-логіки або інтеграції з іншими сервісами.

Хід роботи

Завдання 1. Встановлення інтегрованого середовища розробки (IDE) та необхідних компонентів

Для роботи з ASP.NET Core найзручніше використовувати IDE Visual Studio. Наразі останньою версією є Visual Studio 2022. Для встановлення слід перейти за посиланням, прогорнути початковий банер та натиснути кнопку “Download Visual Studio”, випадаючого списку обрати **Community 2022** (рис. 1.1.)

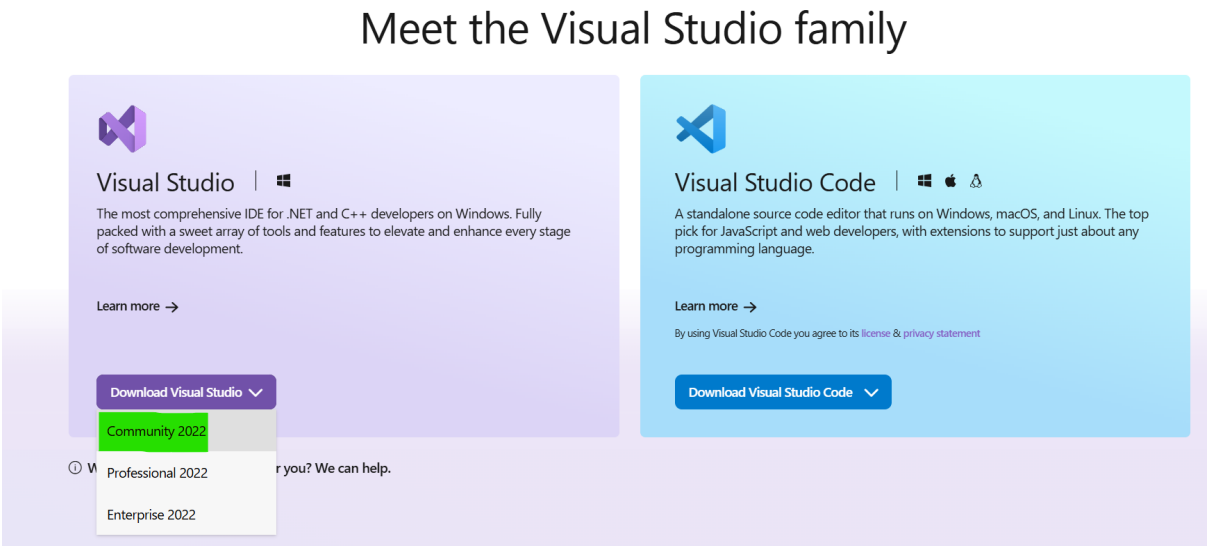

Рис. 1.1. Завантаження Visual Studio Installer

Тоді почнеться завантаження Visual Studio Installer. Після завершення слід запустити скачаний exe файл та дочекатися встановлення. Після завершення встановлення Visual Studio Installer, з’явиться можливість обрати необхідні компоненти. **Важливо: в розділі Web & Cloud обов’язково потрібно обрати компонент під назвою ASP.NET and web development (рис. 1.2.).** На наступному кроці інсталятор запропонує обрати для встановлення індивідуальні компоненти (вкладка Individual Components), на цьому кроці залишаємо все за замовчуванням.

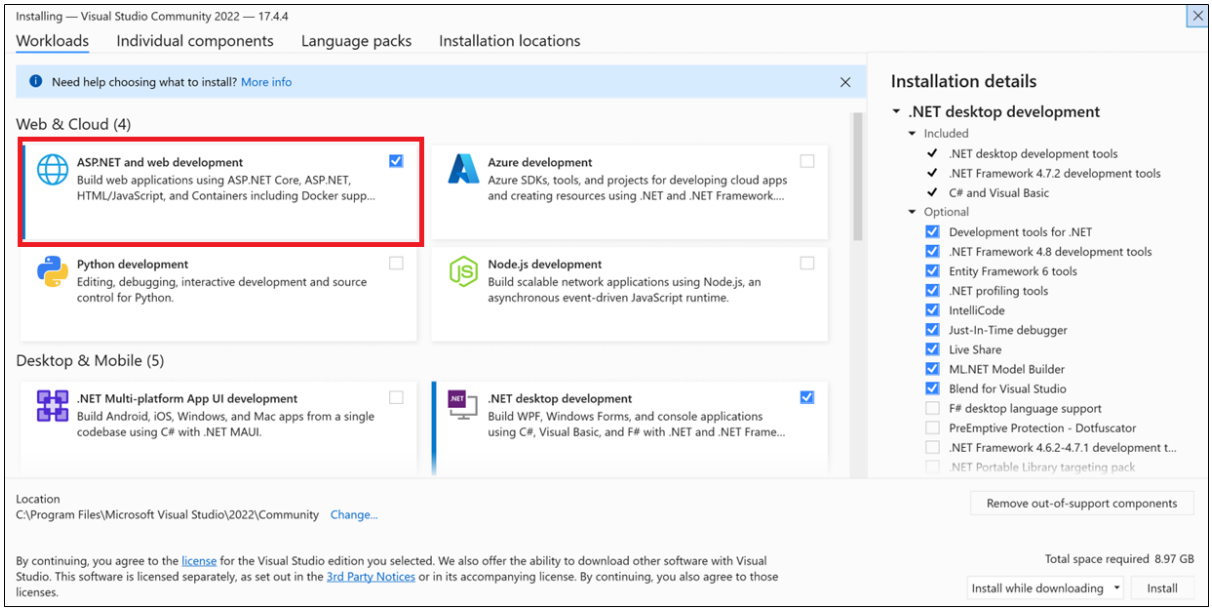

Рис. 1.2. Вибір необхідних компонентів

На вкладці Language Packs інсталятор дає можливість обрати необхідні мовні пакети. **Рекомендовано обирати English (рис. 1.3.)**.

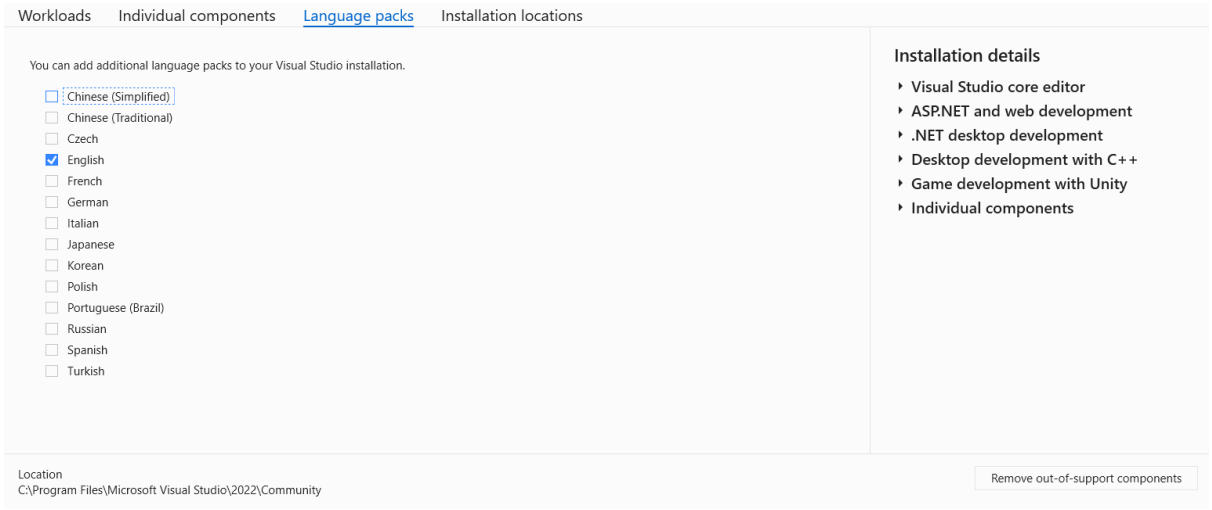

Рис. 1.3. Вибір мовних пакетів

На вкладці Installation locations за потреби обираємо директорію для встановлення Visual Studio, решту опцій можна залишити за замовчуванням (рис. 1.4).

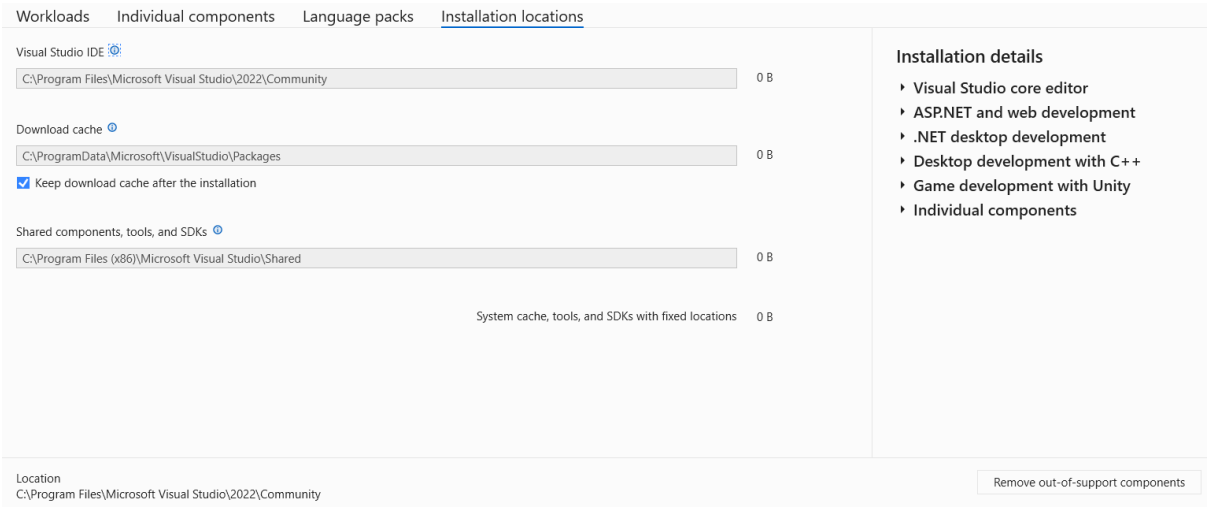

Рис. 1.4. Вибір директорії для встановлення

Після цього почнеться процес встановлення Visual Studio та всіх необхідних компонентів для розробки на ASP.NET Core.

За бажанням, можна налаштувати Visual Studio Code для роботи з .NET, детальна інструкція за посиланням.

Після завершення інсталяції, Visual Studio можна буде запустити будь-яким зручним способом, початкове вікно зображено на рисунку 1.5.

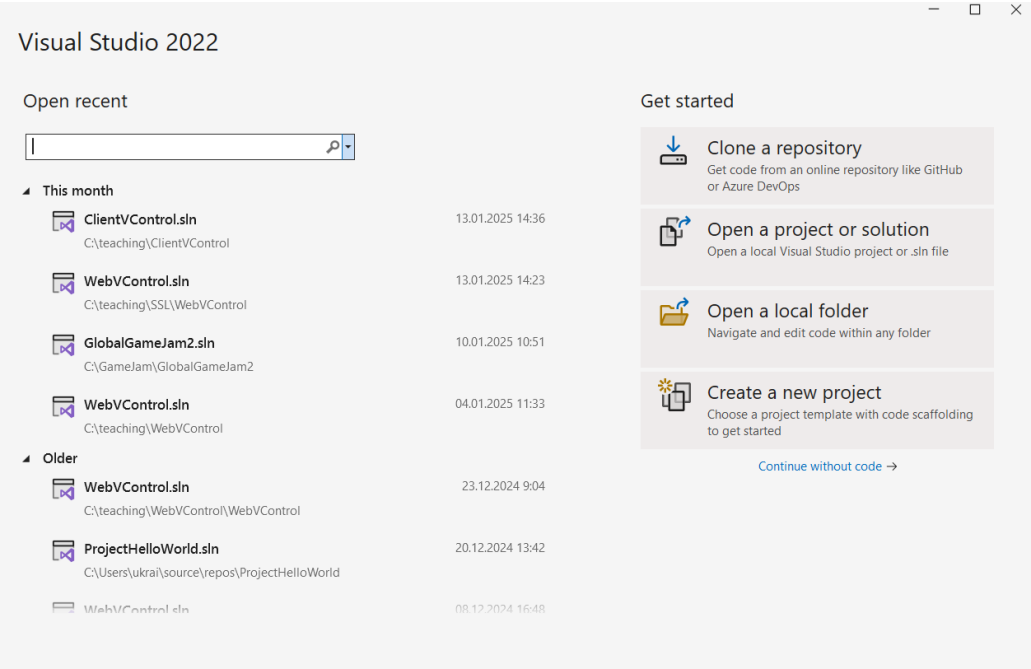

Рис. 1.5. Початкове вікно Visual Studio

Порядок дій, якщо Visual Studio вже встановлена на вашому ПК

Відкриваємо Visual Studio Installer зручним способом. Далі на інсталяції Visual Studio натискаємо кнопку “Modify” (рис. 1.6).

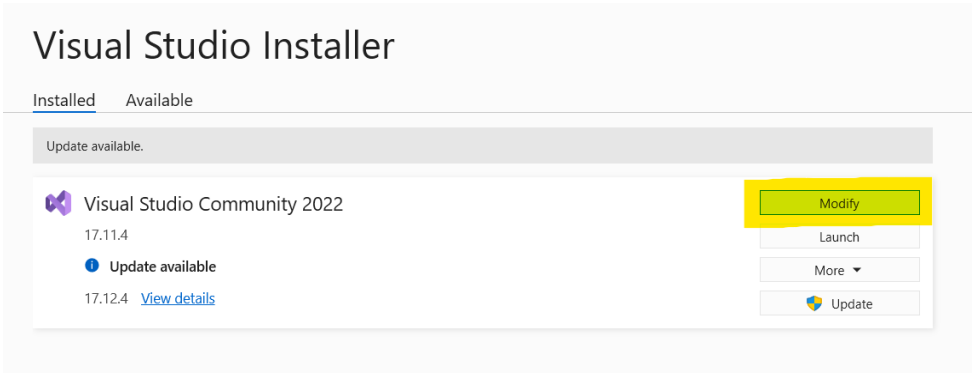

Рис. 1.6. Модифікація наявної інсталяції

Після цього відкриється вікно вибору компонентів, в розділі Web & Cloud обов’язково потрібно обрати компонент під назвою **ASP.NET and web development,** якщо він не вибраний (рис. 1.7).

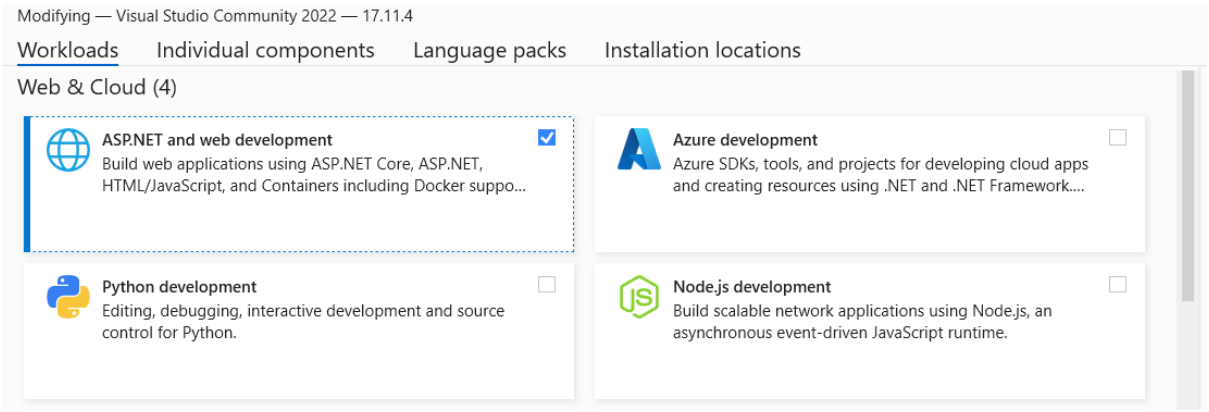

Рис. 1.7. Встановлення необхідних компонентів

Після цього в нижньому правому кутку натискаємо кнопку “Modify” і чекаємо завершення встановлення.

Завдання 2. Створення проектів

Частина 1.

1. Створіть проект для консольного додатку з назвою ConsoleToWeb з використанням dotnet CLI
2. Перетворіть створений консольний додаток на веб-додаток
3. Опишіть виконані кроки у звіті

Частина 2.

1. Створіть ASP.NET WebAPI проект без авторизації з назвою WebFromCli з використанням dotnet CLI
2. Реалізуйте GET-ендпоінт “/who”, який повертатиме у відповідь ваше ім’я та прізвище
3. Реалізуйте GET-ендпоінт “/time”, який повертатиме у відповідь поточний час на сервері
4. Наведіть лістинг реалізованих обробників у звіті

Частина 3.

1. Створіть ASP.NET MVC проект будь-яким зручним способом.
2. Реалізуйте контролер з назвою LabController
3. В створеному контролері реалізуйте обробник /info, який повертатиме View з даними про номер лабораторної роботи, тему, мету та ім’я та прізвище виконавця в табличному вигляді
4. Дані для відображення передати з контролера

Завдання 3. Робота з middleware

1. Ознайомтеся з поняттям middleware в ASP.NET
2. Створіть ASP.NET WebAPI проект з назвою MiddlewareSandbox
3. Реалізуйте наступні middleware
    a. Реалізуйте middleware, яке рахує кількість запитів до сервера і повертає це число у відповіді, наприклад: The amount of processed requests is X.
    b. Створіть middleware, яке аналізує параметри рядка запиту. Якщо в рядку запиту (query string) присутній параметр “custom”, то повертати у відповідь “You’ve hit a custom middleware!”, інакше пропускати запит далі. Приклад запиту зображено на рисунку 3.1.

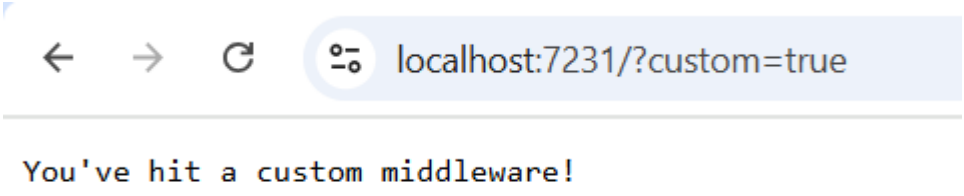

Рис. 3.1. Приклад виконання запиту

    c. Створіть middleware, яке логуватиме у консоль інформацію про всі запити. В логах відображати метод запиту (GET, POST тощо) та

його шлях (/user, /product тощо). Для перевірки POST та інших методів запиту можна скористатися Postman або іншою подібною утилітою.

d. Реалізуйте middleware, яке перевіряє наявність API-ключа у заголовках запиту. Для цього слід Перевіряти, чи є в запиті заголовок X-API-KEY. Якщо ключ неправильний або відсутній, повертати 403 Forbidden, не передавати запит для виконання далі. Якщо ключ в заголовку запиту співпадає з тим, який заданий на сервері, передавати запит далі. Для додавання відповідного запиту можна скористатися Postman або подібними утилітами.

Питання для самоконтролю

1. Що являє собою платформа .NET? Чим досягається різноманітність мов і кросплатформеність додатків розроблених на .NET?
2. Якими були етапи розвитку технології ASP.NET? Чим відрізняється ASP.NET від ASP.NET Core?
3. Які типи додатків можна розробляти з використанням ASP.NET?
4. Що таке REST API?
5. Опишіть принцип роботи middleware.

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).

Теоретичні відомості

MVC (Model-View-Controller) – це архітектурний шаблон, який розділяє застосунок на три взаємопов’язані групи компонентів, кожна з яких відповідає за окремий аспект поведінки застосунку.

Модель представляє дані та бізнес-логіку застосунку. Клас моделі визначає форму даних – їх властивості, правила перевірки та зв’язки. У застосунку ASP.NET MVC моделі зазвичай є простими класами C# (часто званими POCO – Plain Old CLR Objects), властивості яких відповідають стовпцям у таблиці бази даних. Наприклад:

```
public class Product
{
    public int Id { get; set; }
    public string Name { get; set; }
    public decimal Price { get; set; }
}
```

Представлення відповідає за представлення даних користувачеві. В ASP.NET MVC представлення – це файли .cshtml, які використовують синтаксис Razor — комбінацію розмітки HTML та коду C#. Представлення отримують дані від контролера та відображають їх у HTML, який надсилається до браузера. Представлення повинно містити мінімальну логіку; його основне завдання — відображати надану йому інформацію.

Контролер діє як посередник між моделлю та представленням. Він отримує вхідні HTTP-запити, обробляє їх (що може включати читання або запис даних через моделі) та вибирає представлення для повернення у відповідь. Кожен публічний метод у контролері називається action-методом і зазвичай відповідає певній кінцевій точці (endpoint) URL-адреси. Наприклад:

```
public class ProductsController : Controller
{
    public IActionResult Index()
    {
        var products = _context.Products.ToList();
        return View(products);
    }
}
```

Загальний життєвий цикл запиту в MVC працює наступним чином: користувач надсилає HTTP-запит; система маршрутизації зіставляє цей запит з певним action-методом контролера; action-методо виконує необхідні операції (часто запитує базу даних через рівень моделі) та передає отримані дані до представлення; представлення відтворює HTML та повертає його до браузера користувача.

Entity Framework Core (EF Core) — це об'єктно-реляційний відображувач (Object-Relational Mapper, ORM), який дозволяє розробникам працювати з базою даних, використовуючи об'єкти C#, замість написання необроблених SQL-запитів. Налаштування підключення до бази даних включає кілька кроків.

Крок 1. Встановлення необхідних пакетів NuGet. Як мінімум, вам знадобиться Microsoft.EntityFrameworkCore та постачальник бази даних, такий як Microsoft.EntityFrameworkCore.SqlServer для SQL Server або Microsoft.EntityFrameworkCore.Sqlite для SQLite.

Крок 2. Визначення рядка підключення (connection string). Рядок підключення вказує, до якого сервера бази даних підключатися, назву бази даних та метод автентифікації. Він зберігається у файлі appsettings.json:

```
{
  "ConnectionStrings": {
    "DefaultConnection":
"Server=(localdb)\\mssqllocaldb;Database=MyAppDb;Trusted_Connection=True;"
  }
}
```

Крок 3. Реєстрація DbContext у контейнері впровадження залежностей (dependency injection). У Program.cs (або Startup.cs у старіших шаблонах проектів) контекст бази даних реєструється як служба, щоб його можна було впровадити в контролери:

```
builder.Services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(builder.Configuration.
                GetConnectionString("DefaultConnection")));
```

Клас DbContext – це центральний клас в EF Core, який керує підключенням до бази даних і зіставляє класи C# з таблицями бази даних. Сутності реєструються для зберігання шляхом оголошення властивостей DbSet<T> всередині класу, що успадковується від DbContext:

```
public class ApplicationDbContext : DbContext
{
    public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
        : base(options) { }

    public DbSet<Product> Products { get; set; }
    public DbSet<Category> Categories { get; set; }
}
```

Кожна властивість DbSet<T> повідомляє EF Core, що відповідний клас T має бути зіставлений з таблицею в базі даних. Назва властивості (Products, Categories) зазвичай стає назвою таблиці. У цьому файлі контексту реєструються всі сутності, які потрібно зберігати. Під час запуску міграцій EF Core фреймворк зчитує декларації DbSet для створення схеми бази даних.

Важливо розрізняти моделі (models) та моделі представлення (view models). Модель (її також називають доменною моделлю або сутністю) представляє реальний об'єкт в домені (предметній області) програми та зазвичай безпосередньо відображається в таблицю бази даних. Вона містить властивості, що відповідають стовпцям таблиці. Моделі пов'язані зі зберіганням даних та бізнес-правилами. Модель представлення – це клас, спеціально розроблений для передачі даних між контролером та представленням. Вона може містити властивості з однієї або кількох моделей, відформатовані або об'єднані таким чином, щоб це було зручно для певної сторінки.

Наприклад, для сторінки зі списком товарів можуть знадобитися не всі стовпці з таблиці «Товар», але, при цьому, може стати потрібною назва категорії з іншої, пов’язаної таблиці. Модель представлення вирішує цю проблему:

```
public class ProductListViewModel
{
    public string ProductName { get; set; }
    public decimal Price { get; set; }
    public string CategoryName { get; set; }
}
```

Використання моделей представлень запобігає атакам надмірного надсилання (коли користувач надсилає додаткові поля для зміни даних, до яких він не повинен мати доступу) та зберігає представлення відокремленими від схеми бази даних.

Рядок запиту – це частина URL-адреси, що йде після символу ? та містить пари ключ-значення (наприклад, /Products/Search?name=laptop&maxPrice=1000). В ASP.NET MVC значення рядка запиту автоматично прив’язуються до параметрів методу дії, якщо імена параметрів відповідають ключам рядка запиту:

```
public IActionResult Search(string name, decimal? maxPrice)
{
    var results = _context.Products
        .Where(p => p.Name.Contains(name))
        .Where(p => !maxPrice.HasValue || p.Price <= maxPrice)
        .ToList();
    return View(results);
}
```

Коли користувач переходить до /Products/Search?name=laptop&maxPrice=1000, фреймворк автоматично призначає значення "laptop" параметру name та 1000 параметру maxPrice. Цей механізм є частиною системи зв'язування моделей ASP.NET, яка також може зв'язувати значення з сегментів маршруту, полів форм та заголовків запитів.

У якості альтернативи, можна використати атрибут [FromQuery], щоб зробити джерело зв'язування явним, що особливо корисно, коли назва параметра має відрізнятися від ключа рядка запиту:

```
public IActionResult Search([FromQuery(Name = "q")] string searchTerm)
```

Щоб вказати, яку модель (або модель представлення) має отримати представлення Razor, у верхній частині файлу .cshtml розміщується директива @model:

```
@model IEnumerable<Product>

<h1>Product List</h1>
<ul>
    @foreach (var product in Model)
    {
        <li>@product.Name – @product.Price</li>
```

```
    }
</ul>
```

Директива @model (мала літера "m") оголошує тип даних, з якими працює представлення. Після оголошення властивість Model (велика літера "M") стає строго типізованим посиланням на дані, які контролер передав через метод View(). Це забезпечує підтримку IntelliSense в редакторі та перевірку типів під час компіляції, що робить код менш схильним до помилок порівняно з передачею даних через слабо типізовані механізми, такі як ViewBag або ViewData.

Тип у директиві @model має відповідати тому, що передає дія контролера. Якщо дія повертає View(productList), де productList є List<Product>, у представленні слід оголосити @model IEnumerable<Product> (або @model List<Product>).

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.

Таблиця 1.1. Тематика індивідуальних проектів

Номер за списком	Тема проекту	Короткий опис функціоналу
1	Веб-сайт для бронювання квитків на заходи	Користувачі можуть шукати події, бронювати квитки та керувати

		бронюванням, а адміністратори можуть додавати та змінювати події.
2	Система керування курсами	Студенти можуть реєструватися, переглядати та залишати курси, а викладачі (адміністратори) керують курсами.
3	Портал вакансій	Роботодавці можуть публікувати списки вакансій, а кандидати можуть відгукуватися на вакансії та керувати своїми відгуками.
4	Система запису на прийом до лікарні	Пацієнти можуть записатися на прийом до лікаря, переглядати розклад і отримувати нагадування. Лікарі та адміністратори керують доступністю лікарів.
5	Портал для опитувань	Користувачі можуть реєструватися та брати участь в опитуваннях, а адміністратори можуть створювати та аналізувати результати опитувань.
6	Веб-сайт для бронювання номерів готелю	Користувачі можуть переглядати доступні номери, робити бронювання та перевіряти історію бронювань, тоді як адміністратори керують номерами та цінами.
7	Система для прокату транспорту	Користувачі можуть переглядати доступні транспорт (автомобілі, самокати, велосипеди тощо),

		робити бронювання та повертати транспортні засоби, а адміністратори керують автопарком.
8	Веб-сайт бронювання квитків на кіносеанси	Користувачі переглядають фільми, які в прокаті, бронюють місця та керують бронюванням, а адміністратори керують розкладом фільмів, місцями і доступністю кінозалів.
9	Система управління волонтерськими зборами та благодійними заходами	Користувачі можуть переглядати, зареєструватися на участь у благодійних чи волонтерських заходах, робити донати на актуальні збори, адміністратори керують подіями та зборам.
10	Новинний портал	Користувачі можуть переглядати, додавати новини до обраного, залишати коментарі, адміністратори відповідають за керування новинами, модерацією коментарів.

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.

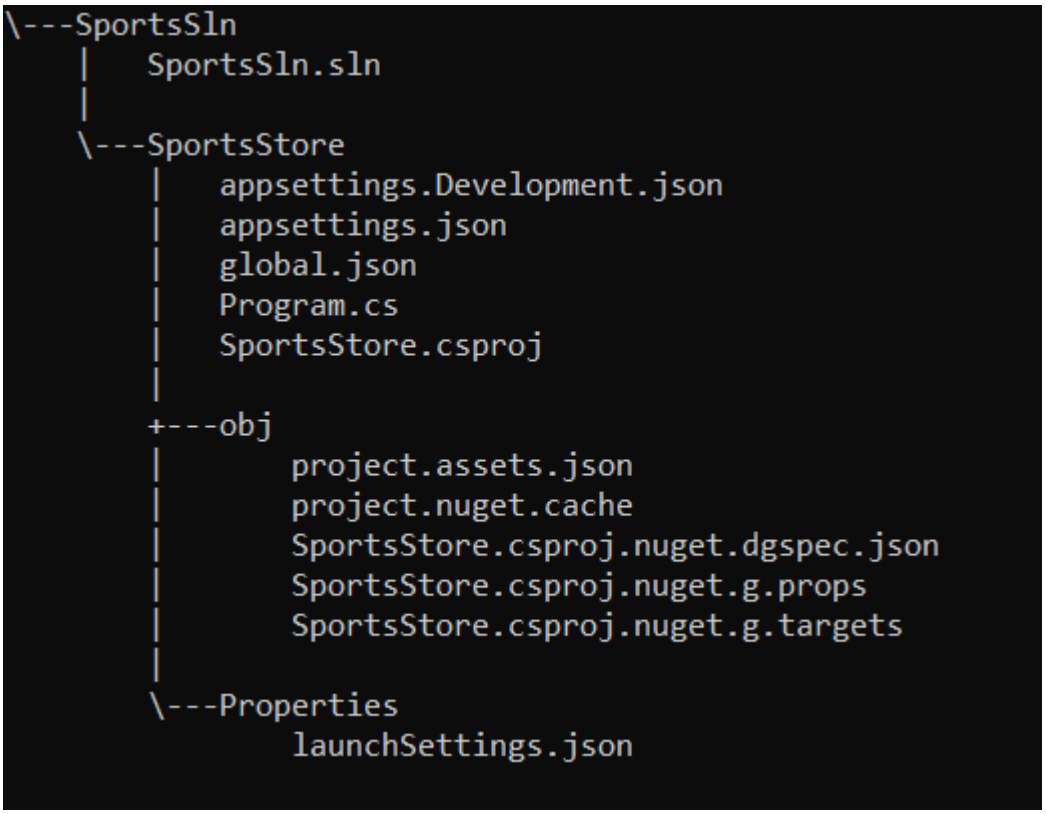

Рис. 1.1. Структура файлів і папок

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.

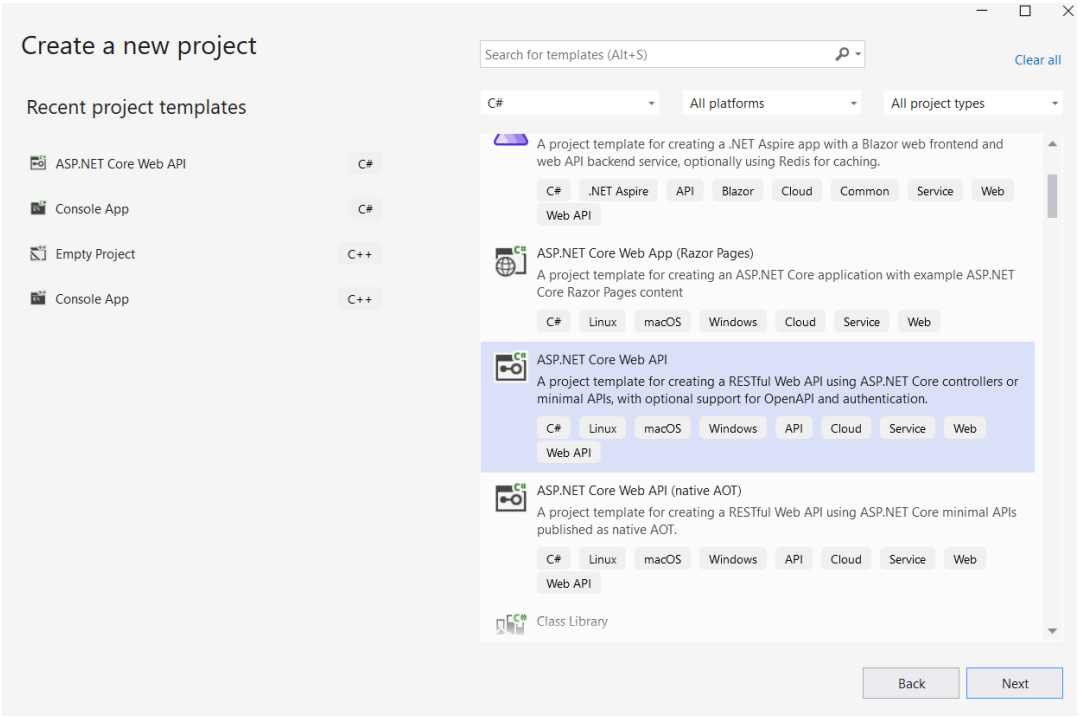

Рис. 1.2. Вибір шаблону проекту

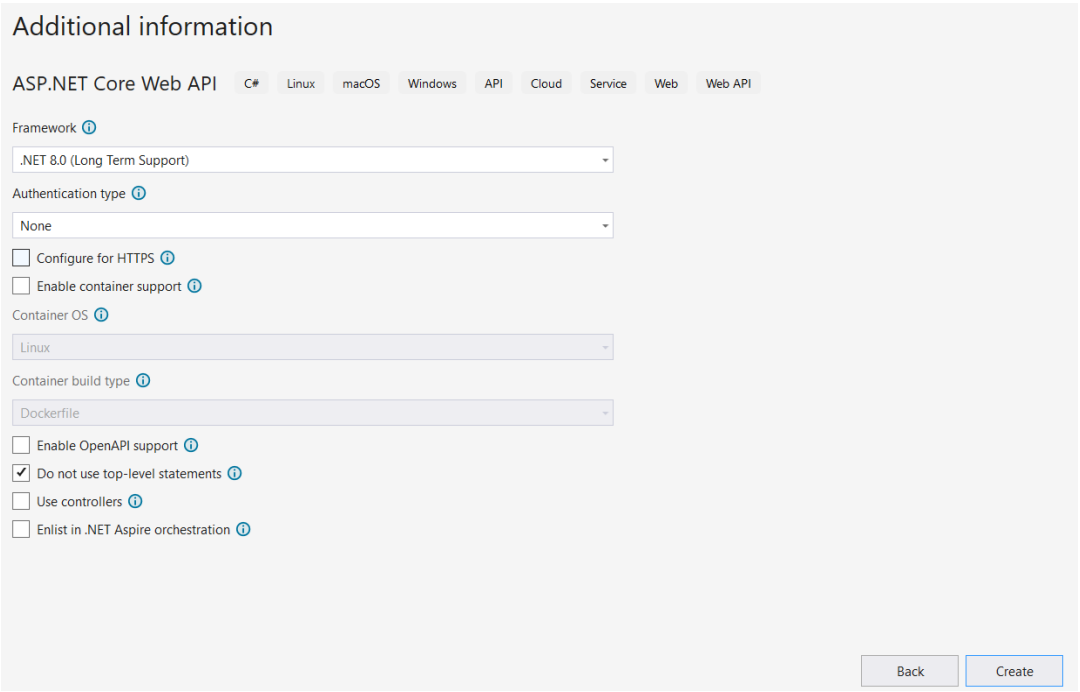

Рис. 1.3. Додаткові налаштування проекту

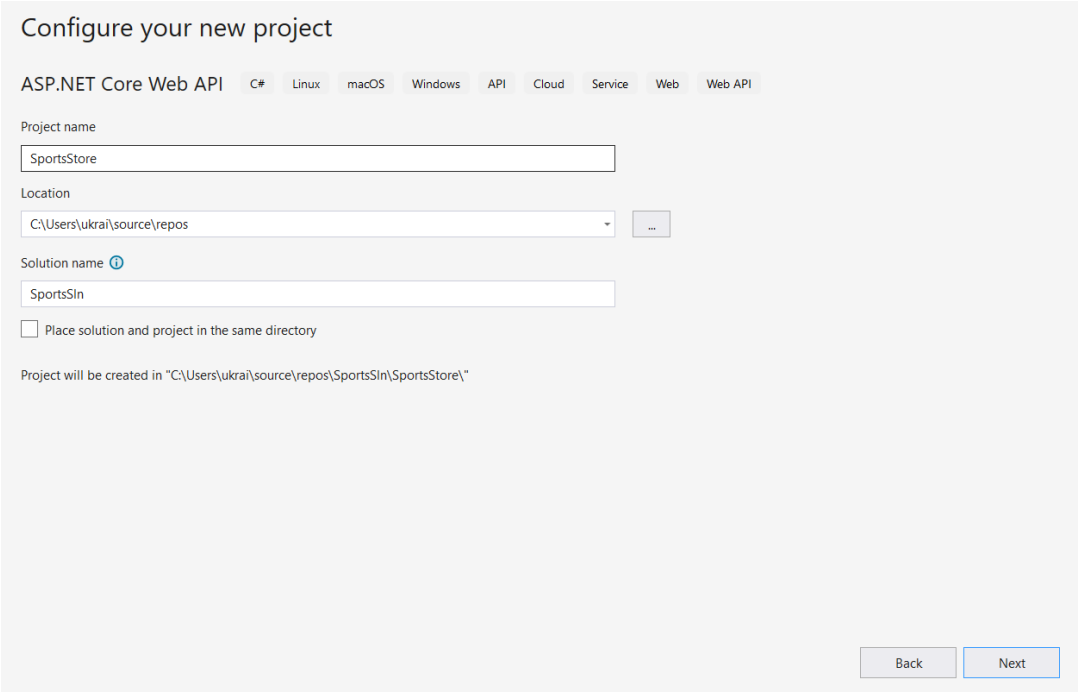

Рис. 1.4. Назва проекту та рішення

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.

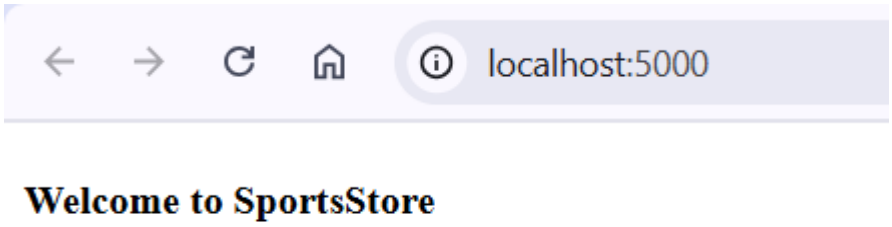

Рис. 1.5. Запущений проект

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 i` 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.).

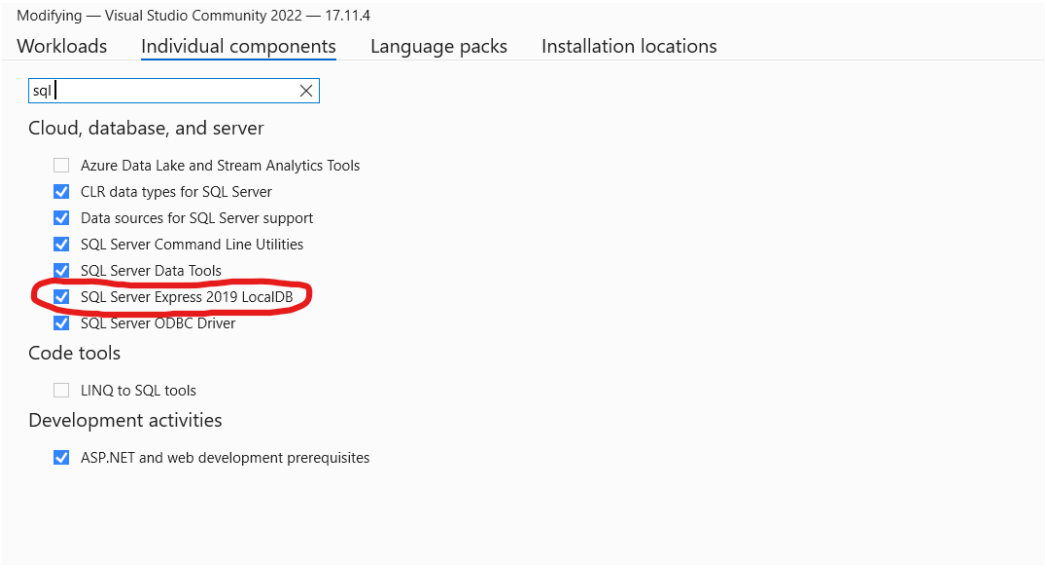

Рис. 2.1. Встановлення SQL Server LocalDB в VS Installer

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=tr
ue"
  }
}
```

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:SportsStoreCo
nnection"]);
});

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

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:SportsStoreCo
nnection"]);
});
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
```

Результат успішного виконання команди зображено на рисунку 2.2-2.3.

```
PS C:\teaching\ASP.NET\SportsSln\SportsStore> dotnet ef migrations add Initial
Build started...
Build succeeded.
Done. To undo this action, use 'ef migrations remove'
PS C:\teaching\ASP.NET\SportsSln\SportsStore>
```

Рис. 2.2. Успішне виконання команди

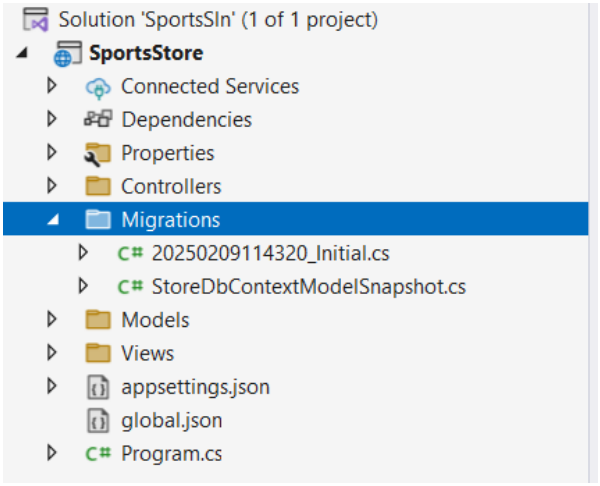

Рис. 2.3. Створена папка з міграціями

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:SportsStoreCo
nnection"]);
});
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.

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.

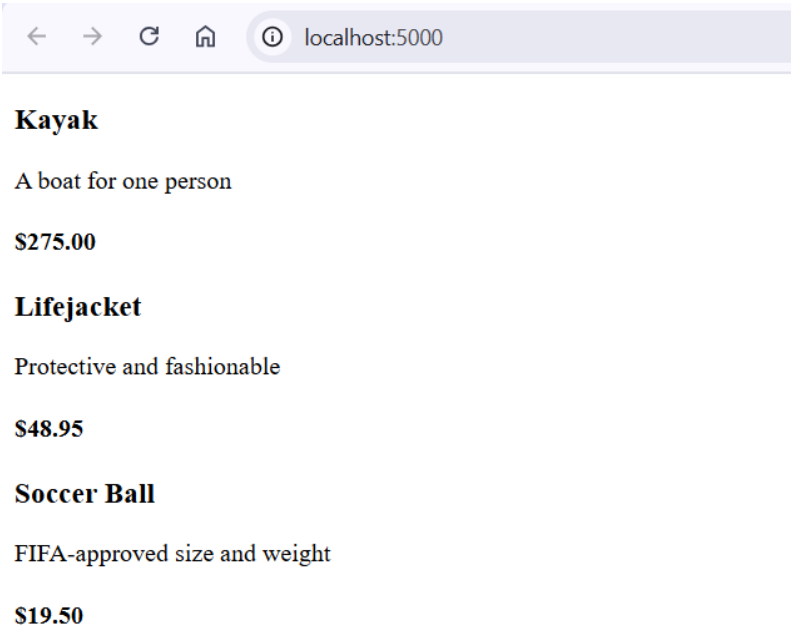

Рис. 3.1. Виведення даних на представлення

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.

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.

Завантажте вихідний код проекту та звіт до лабораторної роботи на корпоративний git-репозиторій.

Питання для самоконтролю

1. Опишіть паттерн MVC та його основні компоненти.
2. Як налаштовується підключення до БД за допомогою EF Core?

3. В якому файлі відбувається реєстрація сутностей для збереження в БД?
4. Що таке Model, а що таке View Model? Яка між ними різниця?
5. Як передати аргумент рядка запиту в Action-метод контролера?
6. Як визначити модель, з якою працюватиме представлення?

Лабораторна робота №3

Тема: Часткові представлення. Робота з View Components. Навігація. Фільтрація. Використання сесії.

Мета: отримати навички роботи з частковими представленнями, навчитися працювати з параметрами рядку запиту, набути навичок роботи з сесіями.

Теоретичні відомості

Часткове представлення (Partial View) – це представлення Razor (файл .cshtml), яке відображає частину HTML-контенту всередині іншого представлення. На відміну від звичайного представлення, часткове представлення не виконує повну дію контролера та не використовує сторінку макета – воно створює лише фрагмент розмітки, який вбудований у батьківське представлення.

Часткові представлення використовуються для розбиття великих, складних сторінок на менші, повторно використовувані компоненти. Типовими прикладами є меню навігації, картки продуктів, блоки коментарів та розділи спільних форм. Замість дублювання однієї й тієї ж розмітки на кількох сторінках, ви витягуєте її в часткове представлення та посилаєтеся на неї, де це необхідно.

Існує кілька способів відображення часткового представлення. Рекомендований підхід в ASP.NET Core – це допоміжний тег <partial>:

```
<partial name="_ProductCard" model="product" />
```

Або ж можна скористатися допоміжним методом HTML:

```
@await Html.PartialAsync("_ProductCard", product)
```

За домовленістю, назви файлів часткового представлення починаються з символу підкреслення (наприклад, _ProductCard.cshtml), щоб візуально відрізнити їх від повних представлень. Часткове представлення може оголошувати власну директиву @model та отримувати дані з батьківського представлення. Наприклад:

```
@* _ProductCard.cshtml *@
@model Product

<div class="card">
    <h3>@Model.Name</h3>
    <p>Price: @Model.Price</p>
</div>
```

Батьківський вигляд потім передає екземпляр моделі під час виклику часткової функції:

```
@foreach (var product in Model.Products)
{
    <partial name="_ProductCard" model="product" />
}
```

Хоча часткові представлення ефективні для простих фрагментів, які можна повторно використовувати, вони мають обмеження: вони повністю залежать від батьківського представлення (або його контролера) для постачання даних. Компоненти представлення виправляють цей аспект, інкапсулюючи як логіку отримання даних, так і шаблон Razor для їх рендерингу в самостійну одиницю.

Компонент представлення складається з двох частин:

1. Клас, який успадковується від ViewComponent і містить метод InvokeAsync (або синхронний Invoke). Цей метод виконує будь-яку необхідну логіку, таку як запит до бази даних, і повертає результат представлення:

```
public class CategoryMenuViewComponent : ViewComponent
{
```

```
    private readonly ApplicationDbContext _context;

    public CategoryMenuViewComponent(ApplicationDbContext context)
    {
        _context = context;
    }

    public async Task<IViewComponentResult> InvokeAsync()
    {
        var categories = await _context.Categories.ToListAsync();
        return View(categories);
    }
}
```

2. Представлення Razor, розташоване за домовленістю за адресою Views/Shared/Components/{ComponentName}/Default.cshtml:

```
@model IEnumerable<Category>

<ul class="nav">
    @foreach (var category in Model)
    {
        <li><a href="/Products?categoryId=@category.Id">@category.Name</a></li>
    }
</ul>
```

Компонент View викликається з будь-якого представлення або макета за допомогою допоміжного елемента тегу <vc:> або методу Component.InvokeAsync:

```
<vc:category-menu />
@* or *@
@await Component.InvokeAsync("CategoryMenu")
```

Коли користувачі взаємодіють із веб-застосунком — натискають посилання, застосовують фільтри, переходять між сторінками — вибрані ними параметри часто закодовані в URL-адресі. ASP.NET Core надає два основні механізми для передачі таких аргументів діям контролера.

Параметри рядка запиту з'являються після символу ? в URL-адресі. Наприклад, у /Products/Index?categoryId=3&page=2 ключі categoryId та page містять вибрані користувачем параметри. Система зв'язування моделей

ASP.NET автоматично зіставляє ці ключі з параметрами методу дій з відповідними іменами:

```
public IActionResult Index(int? categoryId, int page = 1)
{
    var products = _context.Products.AsQueryable();

    if (categoryId.HasValue)
        products = products.Where(p => p.CategoryId == categoryId.Value);

    return View(products.Skip((page - 1) * 10).Take(10).ToList());
}
```

Параметри маршруту вбудовані безпосередньо в URL-шлях. Вони визначені в шаблоні маршруту та витягуються механізмом маршрутизації. Наприклад, для шаблону маршруту Products/Details/{id} запит до /Products/Details/5 пов'язує значення 5 з параметром id:

```
[Route("Products/Details/{id}")]
public IActionResult Details(int id)
{
    var product = _context.Products.Find(id);
    return View(product);
}
```

Для сценаріїв фільтрації, де користувачі вибирають категорії, цінові діапазони або пошукові терміни, параметри рядка запиту є звичайним вибором. Вони роблять стан фільтра видимим в URL-адресі, що означає, що користувач може додавати закладки, ділитися або повертатися до відфільтрованого представлення, не втрачаючи свого вибору.

Щоб створити такі URL-адреси в представленнях Razor, використовуйте помічники тегів asp-action, asp-controller та asp-route-*. Це генерує посилання типу /Products/Index?categoryId=3&page=1.

```
<a asp-action="Index"
   asp-controller="Products"
   asp-route-categoryId="@category.Id"
   asp-route-page="1">
```

```
    @category.Name
</a>
```

Так як HTTP — це протокол без збереження стану, кожен запит від браузера до сервера є незалежним, і сервер по суті не пам’ятає нічого про попередні запити від того самого користувача. Однак багатьом застосункам потрібно підтримувати стан користувача для кількох запитів, наприклад, відстежувати товари в кошику для покупок, запам’ятовувати бажану мову користувача або зберігати застосовані фільтри під час переходу користувача між сторінками.

Сеанс (або сесія, session) — це серверний механізм, який вирішує цю проблему. Коли сеанс встановлено, сервер створює область зберігання, пов’язану з унікальним ідентифікатором сеансу. Цей ідентифікатор надсилається браузеру як файл cookie. При кожному наступному запиті браузер надсилає файл cookie назад, дозволяючи серверу шукати відповідні дані сеансу.

Ключові характеристики сеансів в ASP.NET Core:

- Дані сеансу зберігаються на сервері (за замовчуванням у пам’яті, але їх можна налаштувати для використання розподіленого сховища, такого як Redis або SQL Server).
- Браузер зберігає лише ідентифікатор сеансу в файлі cookie — він ніколи не бачить фактичні дані сеансу.
- Сеанси мають тайм-аут (20 хвилин бездіяльності за замовчуванням). Якщо користувач не зробить жодних запитів протягом цього періоду, сеанс завершиться, а його дані будуть відкинуті.
- Значення сеансу зберігаються як байтові масиви. ASP.NET Core надає методи розширення для поширених типів: SetString / GetString та SetInt32 / GetInt32.

Увімкнення сеансів вимагає змін у двох місцях файлу Program.cs.

Крок 1. Зареєструйте служби сеансів у контейнері впровадження залежностей, викликавши AddSession (та AddDistributedMemoryCache, яка забезпечує резервне сховище в пам'яті):

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

builder.Services.AddDistributedMemoryCache();
builder.Services.AddSession(options =>
{
    options.IdleTimeout = TimeSpan.FromMinutes(30);
    options.Cookie.HttpOnly = true;
    options.Cookie.IsEssential = true;
});

builder.Services.AddControllersWithViews();
```

- IdleTimeout контролює, як довго сеанс залишається без активності.
- HttpOnly = true запобігає доступу клієнтського JavaScript до файлу cookie сеансу, що є найкращою практикою безпеки.
- IsEssential = true гарантує, що файл cookie сеансу надсилається, навіть якщо користувач не дав згоди на необов'язкові файли cookie.

Крок 2. Додайте проміжне програмне забезпечення сеансу до конвеєра запитів. Порядок має значення — UseSession має бути розміщено після UseRouting та перед MapControllerRoute:

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

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

app.Run();
```

Використання сесій у контролері. Після налаштування об'єкт HttpContext.Session доступний у будь-якій дії контролера:

```
public IActionResult ApplyFilter(int categoryId)
```

```
{
    HttpContext.Session.SetInt32("SelectedCategory", categoryId);
    return RedirectToAction("Index");
}

public IActionResult Index()
{
    int? categoryId = HttpContext.Session.GetInt32("SelectedCategory");

    var products = _context.Products.AsQueryable();
    if (categoryId.HasValue)
        products = products.Where(p => p.CategoryId == categoryId.Value);

    return View(products.ToList());
}
```

Для зберігання складних об'єктів (таких як список критеріїв фільтра) серіалізуйте їх у JSON перед збереженням та десеріалізуйте під час читання. Такий підхід дозволяє застосунку зберігати стан, специфічний для користувача, такий як застосовані фільтри, налаштування сортування або тимчасовий вибір, під час кількох навігацій сторінками в межах одного сеансу перегляду.

```
using System.Text.Json;

// Save
HttpContext.Session.SetString("Filters", JsonSerializer.Serialize(filterModel));

// Retrieve
var json = HttpContext.Session.GetString("Filters");
var filters = json != null ? JsonSerializer.Deserialize<FilterModel>(json) : new
FilterModel();
```

Хід роботи

Завдання 1. Створення часткового представлення (Partial View)

1. У папці Views/Shared створіть .cshtml файл для одиничного елементу зі списку сутностей (для SportsStore це буде ProductSummary.cshtml)
2. Перемістіть код для одиничного елементу з загального View Index.cshtm в створений файл.
3. Використайте створений Partial View в Index.cshtml за допомогою Tag Helper'a `<partial>`

Завдання 2. Навігація по категоріям

Реалізуйте навігацію за категорією (або за іншою ознакою, яка більше підходить за вашим варіантом), дотримуючись наступних кроків:

1. В папці проекту створіть папку Components, в ній створіть файл з назвою NavigationMenuViewComponent.cs та реалізуйте його наступним чином:

```
using Microsoft.AspNetCore.Mvc;

namespace SportsStore.Components
{
    public class NavigationMenuViewComponent : ViewComponent
    {
        public string Invoke()
        {
            return "Hello from the Nav View Component";
        }
    }
}
```

2. В загальному шаблоні (файл _Layout.cshtml) додайте розмітку для сайдбару. В доданій розмітці додайте виклик створеного NavigationMenuViewComponent. Структура вашої сторінки має бути схожою на ту, яка зображена на рисунку 2.1.

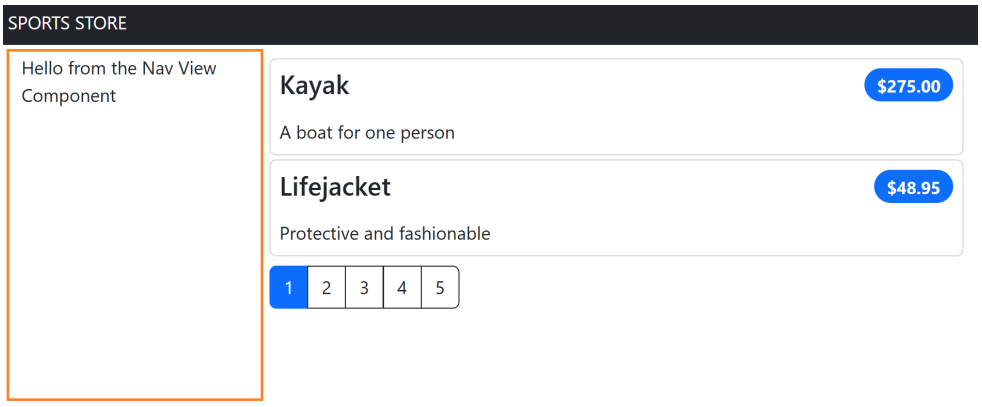

Рис. 2.1 Структура сторінки (кольором виділено сайдбар з ViewComponent)

3. Змініть реалізацію NavigationMenuViewComponent таким чином, щоб компонент отримував перелік категорій з БД і повертав результат типу IViewComponentResult.

4. В папці Views/Shared/Components/NavigationMenu створіть файл Default.cshtml. Детальніше про конвенції неймінгу, які використовує Razor для роботи з ViewComponents можна почитати тут.
5. В створеному файлі представлення реалізуйте виведення списку посилань, отриманих з ViewComponent'а.
6. У файлі Controllers/HomeController.cs в методі Index() реалізуйте підтримку фільтрації по категоріям.
7. Змініть ViewModel таким чином, щоб мати інформацію про обрану категорію на представленні. Це знадобиться для формування посилань на сторінки з урахуванням обраної категорії.
8. Реалізуйте виділення категорії, обраної користувачем. Для цього передавайте інформацію про обрану категорію з View Component в View.
9. Змініть реалізацію пейджингу таким чином, щоб він відображав кількість доступних сторінок відповідно до обраної категорії.

У результаті виконання завдання, ви маєте отримати сторінку схожу на ту, що зображена на рисунку 2.2.

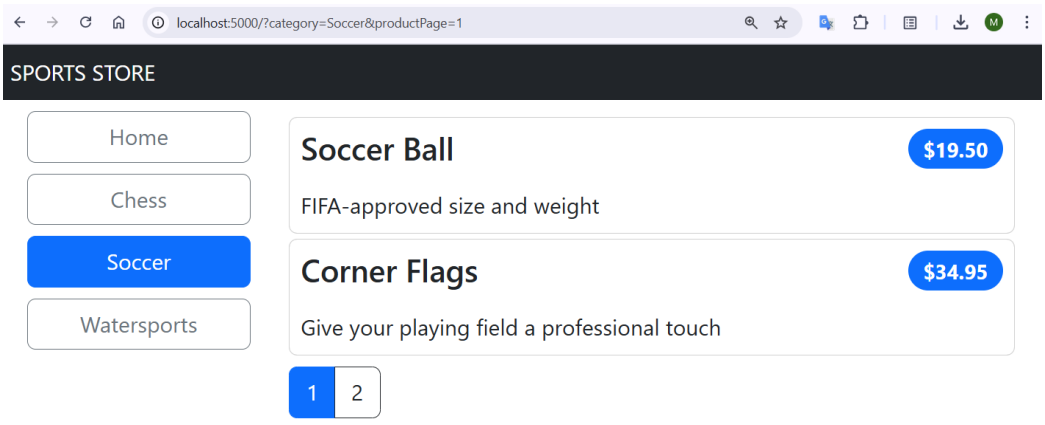

Рис. 2.2. Приклад сторінки

Завдання 3. Сесії

Реалізуйте функціональність відповідно до варіанту вашого проекту з використанням механізму сесії (табл. 3.1.)

Таблиця 3.1. Функціональності для реалізації

Номер за списком групи	**Назва проекту**	**Опис функціональності**
1	Веб-сайт для бронювання квитків на заходи	Використання сесії для тимчасового збереження вибраних квитків перед оплатою, щоб уникнути втрати вибору при навігації на інші сторінки.
2	Система керування курсами	Збереження вибраних курсів у сесії перед остаточною реєстрацією, щоб студент міг переглянути та змінити вибір.
3	Портал вакансій	Збереження незавершених заявок на вакансії у сесії, щоб кандидат міг продовжити заповнення форми після переривання.
4	Система запису на прийом до лікарні	Використання сесії для збереження деталей запису, щоб пацієнт міг продовжити оформлення після повернення.
5	Портал для опитувань	Збереження незавершених відповідей на опитування у сесії, щоб користувач міг завершити опитування пізніше.
6	Веб-сайт для бронювання номерів готелю	Тимчасове збереження вибраних номерів і деталей бронювання у сесії перед оплатою.

7	Система для прокату транспорту	Використання сесій для збереження вибраних транспортних засобів і параметрів оренди перед оформленням.
8	Веб-сайт бронювання квитків на кіносеанси	Тимчасове збереження вибраних місць у сесії перед оформленням замовлення.
9	Система управління волонтерськими зборами та благодійними заходами	Використання сесій для збереження реєстрації на заходи або благодійних внесків перед остаточним підтвердженням.
10	Новинний портал	Збереження списку нещодавно переглянутих статей у сесії.

Приклад проекту з використанням сесії можна знайти за посиланням.

Питання для самоконтролю

1. Що таке Partial View?
2. Які можливості надають View Components? Чим вони відрізняються від звичайних View?
3. Як передавати аргументи адресного рядка в action-метод?
4. Що таке сесія?
5. Як налаштувати використання сесії в ASP.NET Core?

Лабораторна робота №4

Тема: CRUD-операції в EF Core. Валідація даних.

Мета: набути навичок роботи з ORM, навчитися реалізовувати операції створення, читання, оновлення та видалення даних з використанням EF Core, набути навичок роботи з механізмами валідації даних.

Теоретичні відомості

Об'єктно-реляційний маппер (ORM) усуває розрив між об'єктно-орієнтованим світом C# та реляційним світом баз даних. Entity Framework Core є стандартним ORM для .NET-застосунків. Порівняно з написанням “сирих” SQL-запитів, він пропонує кілька суттєвих переваг, таких як читабельність коду, захист від SQL-ін’єкцій, підтримка багатьох СУБД, відслідковування змін та міграції.

Об’єкт, що зберігається у базі даних та може бути унікально ідентифікованим, називається сутністю. Оновлення сутності в EF Core відбувається за простою схемою: отримання сутності, зміна її властивостей та збереження змін. Відстеження змін EF Core виявляє, які властивості були змінені, та генерує інструкцію UPDATE, яка змінює лише ці стовпці.

У наступному прикладі FindAsync(id) отримує сутність та приєднує її до трекера змін у стані «Unchanged» («без змін»). Коли властивостям присвоюються нові значення, трекер переміщує сутність у стан «Modified» («змінено»). Нарешті, SaveChangesAsync() генерує та виконує відповідну команду SQL UPDATE.

```
public async Task<IActionResult> Edit(int id, ProductViewModel model)
{
    var product = await _context.Products.FindAsync(id);
    if (product == null)
        return NotFound();

    product.Name = model.Name;
    product.Price = model.Price;
    product.CategoryId = model.CategoryId;
```

```
    await _context.SaveChangesAsync();
    return RedirectToAction("Index");
}
```

Альтернативний підхід використовується, коли сутність походить із зовнішнього джерела (наприклад, повідомлення форми) і не була завантажена з контексту. У цьому випадку вона приєднується явно та позначається як змінена:

```
_context.Products.Update(product);
await _context.SaveChangesAsync();
```

Метод Update повідомляє трекеру змін, що всі властивості сутності мають бути змінені, тому згенерований SQL-запит встановить усі стовпці, навіть ті, які фактично не змінилися. Перший підхід (завантаження, а потім зміна) зазвичай є кращим, оскільки він створює ефективніші запити та уникає випадкового перезапису одночасних змін.

EF Core не виконує команди бази даних негайно, коли ви додаєте, змінюєте або видаляєте сутності. Натомість усі зміни записуються в пам'ять засобом відстеження змін. Метод SaveChanges() (або його асинхронний аналог SaveChangesAsync()) – це єдина точка, в якій усі накопичені зміни скидаються в базу даних.

Коли викликається SaveChanges(), EF Core робить наступне:

1. Перевіряє засіб відстеження змін, щоб знайти всі сутності у станах Added, Modified або Deleted.

2. Генерує відповідні SQL-запити — INSERT, UPDATE або DELETE — для кожної відстежуваної зміни.

3. Огортає всі оператори в одну транзакцію. Це означає, що або всі зміни зберігаються в базі даних, або жодна з них не зберігається (якщо будь-який оператор завершується невдачею, весь пакет відкатується).

4. Оновлює стани сутностей — наприклад, щойно вставлені сутності отримують значення первинного ключа, згенеровані базою даних, і всі сутності повертаються до стану Unchanged.

```
// Multiple operations tracked in memory
_context.Products.Add(newProduct);
_context.Products.Remove(obsoleteProduct);
existingProduct.Price = 29.99m;

// One call sends all three operations to the database in a single transaction
await _context.SaveChangesAsync();
```

Поведінка пакетної обробки є одночасно оптимізацією продуктивності (менша кількість звернень до бази даних) та захистом цілісності даних (семантика «все або нічого» через транзакцію).

Анотації даних – це атрибути з простору імен System.ComponentModel.DataAnnotations, які розміщуються на властивостях моделі для оголошення правил перевірки. Вони виконують подвійну функцію в ASP.NET Core: EF Core використовує їх для формування схеми бази даних (наприклад, встановлення довжини стовпців та можливості використання значень null), а MVC-фреймворк використовує їх для перевірки введених користувачем даних, перш ніж вони потраплять до бази даних.

```
public class Product
{
    public int Id { get; set; }

    [Required(ErrorMessage = "Product name is required.")]
    [StringLength(200, MinimumLength = 2, ErrorMessage = "Name must be between 2
and 200 characters.")]
    public string Name { get; set; }

    [Range(0.01, 1_000_000, ErrorMessage = "Price must be between 0.01 and
1,000,000.")]
    public decimal Price { get; set; }

    [Required]
    [Display(Name = "Category")]
    public int CategoryId { get; set; }
}
```

У дії контролера властивість ModelState.IsValid перевіряється перед виконанням будь-яких операцій з базою даних. Якщо перевірка не вдається, дія повертає вигляд форми з повідомленнями про помилки, щоб користувач міг виправити введені дані:

```
[HttpPost]
public async Task<IActionResult> Create(Product product)
{
    if (!ModelState.IsValid)
        return View(product);

    _context.Products.Add(product);
    await _context.SaveChangesAsync();
    return RedirectToAction("Index");
}
```

На стороні перегляду, допоміжні засоби тегів, такі як asp-validation-for, відображають повідомлення про помилки, визначені в анотаціях:

```
<form asp-action="Create" method="post">
    <label asp-for="Name"></label>
    <input asp-for="Name" />
    <span asp-validation-for="Name" class="text-danger"></span>

    <label asp-for="Price"></label>
    <input asp-for="Price" />
    <span asp-validation-for="Price" class="text-danger"></span>

    <button type="submit">Create</button>
</form>
```

У наступній таблиці наведено найчастіше використовувані атрибути перевірки:

Атрибут	Призначення
`[Required]`	Властивість повинна мати значення (не може бути null або порожньою)

Атрибут	Призначення
	`[Required(ErrorMessage = "Name is required.")]`
`[StringLength]`	Обмежує максимальну (і за потреби мінімальну) довжину рядка `[StringLength(100, MinimumLength = 3)]`
`[Range]`	Обмежує числове значення заданим інтервалом `[Range(1, 999)]`
`[RegularExpression]`	Значення має відповідати заданому шаблону регулярного виразу `[RegularExpression(@"^\d{5}$", ErrorMessage = "Enter a 5-digit code.")]`
`[EmailAddress]`	Перевіряє, чи є рядок правильно сформованою адресою електронної пошти
`[Phone]`	Перевіряє формат номера телефону
`[Compare]`	Гарантує, що дві властивості мають однакове значення (наприклад, підтвердження пароля) `[Compare("Password", ErrorMessage = "Passwords do not match.")]`
`[Display]`	Встановлює зручну для користувача мітку для властивості у представленнях та повідомленнях про помилки `[Display(Name = "Unit Price")]`
`[DataType]`	Вказує тип даних (дата, валюта, пароль тощо) для відображення інтерфейсу користувача

Атрибут	Призначення
	`[DataType(DataType.Date)]`
`[Key]`	Позначає властивість як первинний ключ (використовується EF Core для створення схеми)
`[MaxLength]`	Встановлює максимальну довжину для масивів або рядків на рівні бази даних `[MaxLength(500)]`

Зазначені атрибути формують першу лінію захисту від невалідних даних. Вони є декларативними (правила видно безпосередньо на моделі), повторно використовуваними (та сама модель перевіряється в будь-якій формі, яка її використовує) та узгодженими (одні й ті самі правила застосовуються як на сервері, так і, з відповідними клієнтськими бібліотеками, у браузері ще до надсилання форми).

Хід роботи

Корисні посилання:

- CRUD-операції в ASP.NET з використанням EF Core
- Зв’язок “один-до-багатьох” в EF Core
- Валідація даних в ASP.NET з використанням DataAnnotations

Завдання 1.

Використовуючи підхід Code First, створити модель, яка разом з існуючою моделлю, утворюватиме зв’язок один-до-багатьох.

Завдання 2.

Реалізувати CRUD (Create, Read, Update, Delete) операції для обох моделей, враховуючи наступні вимоги:

- Реалізувати представлення з формами для операцій Create та Update
- Для виконання операції Read реалізувати окрему Details сторінку, на якій будуть виводитися всі дані про сутність.
- Перед виконанням операції Delete запитати користувача, підтвердження видалення запису.

Приклад реалізації методів репозиторію для виконання операцій з сутністю Product наведено в лістингу нижче:

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

                public void CreateProduct(Product p)
                {
                        context.Add(p);
                        context.SaveChanges();
                }

                public void DeleteProduct(Product p)
                {
                        context.Remove(p);
                        context.SaveChanges();
                }

                public void SaveProduct(Product p)
                {
                        context.SaveChanges();
                }
        }
}
```

Завдання 3.

Додати серверну валідацію для перевірки введених користувачем даних перед виконанням CRUD-операцій. Використати DataAnnotations для реалізації

перевірки введених даних, приклад використання DataAnnotations для сутності Products наведено в лістингу нижче:

```
using System.ComponentModel.DataAnnotations.Schema;
using System.ComponentModel.DataAnnotations;
namespace SportsStore.Models
{
        public class Product
        {
                public long? ProductID { get; set; }

                [Required(ErrorMessage = "Please enter a product name")]
                public string Name { get; set; } = String.Empty;

                [Required(ErrorMessage = "Please enter a description")]
                public string Description { get; set; } = String.Empty;

                [Required]
                [Range(0.01, double.MaxValue, ErrorMessage = "Please enter a
positive price")]
                [Column(TypeName = "decimal(8, 2)")]
                public decimal Price { get; set; }

                [Required(ErrorMessage = "Please specify a category")]
                public string Category { get; set; } = String.Empty;
        }
}
```

З прикладом реалізації CRUD-операцій та валідації можна ознайомитися за посиланням.

Питання для самоконтролю

1. Які переваги використання ORM, такого як EF Core, порівняно з виконанням чистих SQL-запитів?
2. Як ви реалізуєте оновлення наявної сутності в EF Core?
3. Яке призначення методу SaveChanges() в EF Core?
4. Як можна забезпечити дотримання правил валідації даних для властивостей сутності за допомогою анотацій даних?
5. Які атрибути анотацій даних найчастіше використовуються для валідації в EF Core?

Лабораторна робота №5

Тема: Аутентифікація та авторизація. Робота з Web API. JSON Web Token.

Мета: набути навичок роботи з механізмами аутентифікації та авторизації користувачів в ASP.NET, набути навичок роботи з Web API.

Теоретичні відомості

Терміни «аутентифікація» та «авторизація» часто використовуються разом, але вони представляють різні концепції безпеки.

Аутентифікація – це процес перевірки особи користувача. Коли користувач надає ім'я користувача та пароль, і система підтверджує дійсність цих облікових даних, користувач аутентифікований. Тепер система знає його особу.

Авторизація – це процес визначення того, що дозволено робити аутентифікованому користувачеві. Після того, як система дізнається, хто є користувачем, вона перевіряє, чи має цей користувач дозвіл на доступ до певного ресурсу або виконання певної дії. Наприклад, звичайний користувач може переглядати продукти, але лише адміністратор може їх видаляти.

В ASP.NET Core ці дві проблеми обробляються спеціалізованими компонентами проміжного програмного забезпечення, які додаються до конвеєра запитів у певному порядку:

```
app.UseAuthentication();  // Who are you?
app.UseAuthorization();   // Are you allowed to do this?
```

Порядок має значення: система повинна спочатку встановити особу користувача, перш ніж зможе оцінити його дозволи.

ASP.NET Core Identity — це система членства, яка забезпечує повну, готову до використання інфраструктуру для керування користувачами,

паролями, ролями та заявками. Вона обробляє хешування паролів, блокування облікових записів, підтвердження електронної пошти, двофакторну аутентифікацію та багато іншого — функціональність, реалізація якої з нуля була б складною та схильною до помилок.

Identity обертається навколо двох ключових класів сервісів:

- UserManager<IdentityUser> — надає методи для створення користувачів, пошуку користувачів за іменем або електронною поштою, перевірки паролів та керування даними користувачів. Це основний API для всіх операцій, пов'язаних з користувачами.
- SignInManager<IdentityUser> — керує процесом входу, включаючи перевірку пароля, створення файлів cookie та вихід. Його метод PasswordSignInAsync виконує перевірку облікових даних і, у разі успіху, видає файл cookie аутентифікації браузеру.

Обидва сервіси зареєстровані в контейнері впровадження залежностей і можуть бути введені в контролери:

```
public class AccountController : Controller
{
    private UserManager<IdentityUser> _userManager;
    private SignInManager<IdentityUser> _signInManager;

    public AccountController(UserManager<IdentityUser> userMgr,
                             SignInManager<IdentityUser> signInMgr)
    {
        _userManager = userMgr;
        _signInManager = signInMgr;
    }
}
```

Identity зберігає дані користувачів (облікові дані, ролі, заяви) у базі даних. Щоб налаштувати це, ви створюєте клас контексту, який успадковується від IdentityDbContext<IdentityUser> замість звичайного DbContext:

```
public class AppIdentityDbContext : IdentityDbContext<IdentityUser>
{
    public AppIdentityDbContext(DbContextOptions<AppIdentityDbContext> options)
```

```
        : base(options) { }
}
```

Цей базовий клас автоматично включає властивості DbSet для всіх таблиць Identity (Users, Roles, UserRoles, Claims тощо). База даних Identity зазвичай окрема від бази даних додатків, кожна з яких має власний рядок підключення в appsettings.json.

Identity реєструється в Program.cs з опціями, що контролюють складність пароля, вимоги до користувачів та політики блокування:

```
builder.Services.AddIdentity<IdentityUser, IdentityRole>(options =>
{
    options.Password.RequiredLength = 8;
    options.Password.RequireDigit = true;
    options.Password.RequireUppercase = true;
    options.User.RequireUniqueEmail = true;
})
.AddEntityFrameworkStores<AppIdentityDbContext>();
```

Після налаштування ви застосовуєте міграції EF Core для створення таблиць ідентифікації в базі даних.

Атрибут [Authorize] обмежує доступ до контролерів або окремих методів дій, щоб лише автентифіковані (і, за бажанням, спеціально авторизовані) користувачі могли отримати до них доступ.

Коли він розміщений на контролері, він застосовується до всіх дій у цьому контролері:

```
[Authorize]
public class OrderController : Controller { ... }
```

Коли його розміщують на певній дії, він обмежує лише цю кінцеву точку:

```
public class ProductController : Controller
{
    public IActionResult Index() { ... }          // accessible to everyone

    [Authorize]
```

```
    public IActionResult Edit(int id) { ... }      // requires authentication
}
```

Щоб виключити певну дію в межах захищеного контролера, використовуйте [AllowAnonymous]:

```
[Authorize]
public class AccountController : Controller
{
    [AllowAnonymous]
    public IActionResult Login() { ... }   // accessible without authentication

    [AllowAnonymous]
    public IActionResult Register() { ... }
}
```

Авторизація на основі ролей додає ще один рівень деталізації. Після призначення користувачів ролям (наприклад, «Адміністратор», «Користувач») через UserManager.AddToRoleAsync, ви можете обмежити доступ за ролями:

```
[Authorize(Roles = "Admin")]
public IActionResult DeleteProduct(int id) { ... }
```

У прикладі контролера входу ви можете помітити атрибут [ValidateAntiForgeryToken] у дії POST. Це захист від атак типу «підробка міжсайтового запиту» (CSRF), коли шкідливий веб-сайт обманом змушує браузер користувача надіслати форму до вашої програми.

ASP.NET Core генерує унікальний токен, який вбудовується в кожну форму (через допоміжний тег <form asp-action="...">) і зберігається в cookie. Коли форма надсилається, сервер перевіряє, чи обидва токени збігаються. Оскільки сайт зловмисника не може прочитати токен з cookie вашого домену, його підроблений запит буде відхилено.

Аутентифікація на основі файлів cookie добре працює для браузерних MVC-додатків, але веб-API часто обслуговують клієнтів, які не використовують браузер (мобільні додатки, інші сервіси), де файли cookie

непрактичні. Веб-токени JSON забезпечують механізм аутентифікації на основі токенів без стану, який добре підходить для API.

JWT працює наступним чином:

1. Клієнт надсилає запит на вхід з обліковими даними (ім'я користувача та пароль) до кінцевої точки аутентифікації.

2. Сервер перевіряє облікові дані. Якщо вони дійсні, він генерує JWT, що містить твердження про користувача (наприклад, ідентифікатор користувача, ім'я, ролі) та підписує його секретним ключем.

3. Сервер повертає токен клієнту.

4. Для всіх наступних запитів до захищених кінцевих точок клієнт включає токен у HTTP-заголовок Authorization: Authorization: Bearer <токен>.

5. Сервер перевіряє підпис та термін дії токена для кожного запиту. Якщо запит дійсний, він обробляється; інакше сервер повертає 401 Unauthorized.

Токен складається з трьох частин, закодованих у Base64, розділених крапками: заголовок (header), корисне навантаження (payload), підпис (signature). Заголовок визначає тип токена (JWT) та алгоритм підписання (наприклад, HS256). Корисне навантаження містить твердження: фрагменти інформації про користувача, такі як його ідентифікатор, ім'я користувача, ролі та час закінчення терміну дії токена. Підпис створюється шляхом підписання закодованого заголовка та корисного навантаження секретним ключем. Це гарантує, що токен не був підроблений.

Ключові характеристики JWT-аутентифікації:

- Відсутність стану. Серверу не потрібно зберігати дані сеансу. Вся необхідна інформація вбудована в сам токен. Це робить JWT ідеальним для розподілених систем та мікросервісів.

- Автономність. Токен містить ідентифікаційні дані та дозволи користувача, тому серверу не потрібно запитувати базу даних на кожен запит, щоб визначити, хто є користувачем.
- Кінцевий термін дії. Токени містять час закінчення терміну дії. Після закінчення терміну дії клієнт повинен повторно аутентифікуватися, щоб отримати новий токен.

Схема аутентифікації реєструється в Program.cs:

```
builder.Services.AddAuthentication(options =>
{
    options.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;
    options.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;
})
.AddJwtBearer(options =>
{
    options.TokenValidationParameters = new TokenValidationParameters
    {
        ValidateIssuer = true,
        ValidateAudience = true,
        ValidateLifetime = true,
        ValidateIssuerSigningKey = true,
        ValidIssuer = builder.Configuration["Jwt:Issuer"],
        ValidAudience = builder.Configuration["Jwt:Audience"],
        IssuerSigningKey = new SymmetricSecurityKey(
            Encoding.UTF8.GetBytes(builder.Configuration["Jwt:Key"]))
    };
});
```

Генерація токена після успішної аутентифікації:

```
private string GenerateJwtToken(IdentityUser user, IList<string> roles)
{
    var claims = new List<Claim>
    {
        new Claim(ClaimTypes.Name, user.UserName),
        new Claim(ClaimTypes.NameIdentifier, user.Id)
    };
    foreach (var role in roles)
        claims.Add(new Claim(ClaimTypes.Role, role));

    var                    key                    =                    new
SymmetricSecurityKey(Encoding.UTF8.GetBytes(_config["Jwt:Key"]));
    var        credentials        =        new        SigningCredentials(key,
SecurityAlgorithms.HmacSha256);

    var token = new JwtSecurityToken(
```

```
        issuer: _config["Jwt:Issuer"],
        audience: _config["Jwt:Audience"],
        claims: claims,
        expires: DateTime.UtcNow.AddHours(2),
        signingCredentials: credentials);

    return new JwtSecurityTokenHandler().WriteToken(token);
}
```

Згенерований рядок токена повертається клієнту в тілі відповіді. Клієнт зберігає його (наприклад, у локальному сховищі або в пам'яті) та додає до кожного наступного запиту API.

Хід роботи

Завдання 1.

Реалізувати реєстрацію, автентифікацію та авторизацію користувача з використанням пакету Microsoft Identity.

Для встановлення Identity, в папці з проектом слід виконати наступну команду:

```
dotnet add package Microsoft.AspNetCore.Identity.EntityFrameworkCore --version 8.0.0
```

Далі слід створити клас `AppIdentityDbContext` з наступним вмістом:

```
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore;
namespace SportsStore.Models {
 public class AppIdentityDbContext : IdentityDbContext<IdentityUser> {
 public AppIdentityDbContext(
 DbContextOptions<AppIdentityDbContext> options)
 : base(options) { }
 }
}
```

Далі слід додати нову ConnectionString в appsettings.json для бази даних, де зберігатимуться дані про користувачів:

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

Сконфігурувати Identity в файлі Program.cs, додавши наступні рядки:

```
…
using Microsoft.AspNetCore.Identity;

…
builder.Services.AddScoped();
builder.Services.AddScoped();
builder.Services.AddRazorPages();
builder.Services.AddDistributedMemoryCache();
builder.Services.AddSession();

…
builder.Services.AddDbContext(options => options.UseSqlServer(
builder.Configuration["ConnectionStrings:IdentityConnection"])
);
builder.Services.AddIdentity().AddEntityFrameworkStores();

var app = builder.Build();
app.UseStaticFiles();
app.UseSession();

app.UseAuthentication();
app.UseAuthorization();
```

Створіть міграцію для бази даних Identity наступною командою:

```
dotnet ef migrations add Initial --context AppIdentityDbContext
```

Та застосуйте її за допомогою наступної команди:

```
dotnet ef database update --context AppIdentityDbContext
```

Після конфігурації Identity, реалізуйте наступні функціональності:

1. Реєстрація користувача (приклад)
    a. встановити вимогу по унікальності електронної адреси користувача

b. встановити вимоги до пароля: не коротший за 8 символів, містить цифри та літери, обов'язково містить хоча б одну літеру в верхньому регістрі
c. реалізувати поле для підтвердження введеного паролю

2. Автентифікація користувача (Login)
3. Вихід користувача (Logout)
4. Особистий кабінет користувача, де він матиме змогу переглядати або змінювати дані про себе.

Розробіть всі необхідні форми з підтримкою серверної валідації та розробіть відповідний контроллер. Налаштуйте доступ до Action-методів контроллерів з використанням атрибуту [Authorize] – лише залогінений користувач повинен мати можливості виконувати дії на веб-сайті, для неавтентифікованого користувача має бути доступною лише реєстрація та логін. Приклад коду контролера для виконання логіну:

```
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc;
using SportsStore.Models.ViewModels;
namespace SportsStore.Controllers
{
        public class AccountController : Controller
        {
                private UserManager<IdentityUser> userManager;
                private SignInManager<IdentityUser> signInManager;
                public AccountController(UserManager<IdentityUser> userMgr,
                SignInManager<IdentityUser> signInMgr)
                {
                        userManager = userMgr;
                        signInManager = signInMgr;
                }
                public ViewResult Login(string returnUrl)
                {
                        return View(new LoginModel
                        {
                                Name = string.Empty,
                                Password = string.Empty,
                                ReturnUrl = returnUrl
                        });
                }
                [HttpPost]
                [ValidateAntiForgeryToken]
                public async Task<IActionResult> Login(LoginModel loginModel)
                {
```

```
                        if (ModelState.IsValid)
                        {
                                IdentityUser? user =
                                await
userManager.FindByNameAsync(loginModel.Name);
                    if (user != null)
                                {
                                        await signInManager.SignOutAsync();
                                        if                                  ((await
signInManager.PasswordSignInAsync(user,
                                        loginModel.Password,                false,
false)).Succeeded)
                                        {
                                                return
Redirect(loginModel?.ReturnUrl
                                                ?? "/Admin");
                                        }
                                }
                                ModelState.AddModelError("",  "Invalid  name  or
password");
                        }
                        return View(loginModel);
                }
                [Authorize]
                public async Task<RedirectResult> Logout(string returnUrl = "/")
                {
                        await signInManager.SignOutAsync();
                        return Redirect(returnUrl);
                }
        }
}
```

Завдання на додаткові бали (5 балів): реалізувати роботу з ролями користувачів.

Передбачити в проекті можливість реєстрації двох типів користувачів, наприклад пацієнт і лікар. Якщо тематика проекту не передбачає роботу з кількома типами користувачів, то реалізувати ролі звичайного користувача та адміністратора.

Завдання 2.

В рішенні створити новий WebAPI проект. Цей проект повинен реалізовувати всі функціональності, які наявні в MVC проекті у вигляді REST-ендпоїнтів. Для кращої організації і перевикористання коду слід перенести файли для роботи з БД (репозиторії, контексти, моделі) в окрему бібліотеку і

додати посилання на неї в MVC та WebAPI проектах. Встановіть необхідні залежності для проекту бібліотеки.

Після цього налаштуйте ConnectionString'и для створених БД для WebAPI проекту в файлі appsettings.json. Таким чином, після перенесення файлів для роботи з БД в окрему бібліотеку ми можемо використовувати всі необхідні сервіси в обох проектах.

Реалізуйте всі необхідні CRUD-операції в вашому WebAPI проекті, включаючи реєстрацію користувача. Перевірте працездатність ваших ендпоїнтів за допомогою Postman, Swagger або будь-яким іншим зручним інструментом.

Завдання 3.

Реалізуйте автентифікацію та авторизацію користувача з використанням механізму JSON Web Token (JWT). Вона повинна працювати наступним чином:

1. Клієнт відправляє запит на login-ендпоїнт зі своїм логіном та паролем.
2. Якщо користувач з таким логіном та паролем існує, тоді сервер повинен повернути згенерований JSON Web Token.
3. Для отримання доступу до дій, які вимагають авторизації, клієнт відправляє в запиті заголовок Authorization зі значенням Bearer TokenValue, де TokenValue – токен, який було згенеровано під час автентифікації.
4. Якщо токен валідний – сервер виконає запит, інакше поверне помилку 401 Unauthorized.

Приклад реалізації JWT аутентифікації з використанням Identity можна знайти за посиланням.

Питання для самоконтролю

1. Яка різниця між аутентифікацією та авторизацією? У якому порядку відповідні компоненти проміжного програмного забезпечення повинні бути зареєстровані в конвеєрі ASP.NET Core, і чому?
2. Які ролі UserManager та SignInManager в ASP.NET Core Identity?
3. Чим IdentityDbContext відрізняється від звичайного DbContext? Чому для даних Identity часто використовується окрема база даних?
4. Як атрибут [Authorize] контролює доступ до дій контролера? Як можна дозволити неавторизований доступ до певної дії в захищеному контролері?
5. Яке призначення атрибута [ValidateAntiForgeryToken] і який тип атаки він запобігає?
6. Опишіть структуру веб-токена JSON. Яку інформацію містить кожна з трьох його частин?
7. Поясніть процес аутентифікації JWT: як клієнт отримує токен і як він використовує токен для доступу до захищених кінцевих точок API?
8. Чому JWT вважається механізмом автентифікації без збереження стану? Яку перевагу це надає порівняно із сеансами на основі файлів cookie для веб-API?

Лабораторна робота №6

Тема: Розробка клієнтських додатків з використанням Blazor WebAssembly

Мета: отримати навички створення та інтеграції клієнтського додатку з RESTful Web API за допомогою Blazor WebAssembly.

Теоретичні відомості

Blazor має дві основні моделі виконання:

- Blazor Server – виконується на сервері. Blazor Server використовує JavaScript для отримання подій браузера, які пересилаються на ASP.NET Core сервер та обробляються за допомогою C# коду. Після обробки, оновлений стан додатку надсилається назад у браузер і відображається користувачеві.
- Blazor WebAssembly (Blazor WASM) – повністю виконується в браузері. Blazor WebAssembly не має залежності від сервера та виконує програму Blazor повністю в браузері. Результатом є клієнтський застосунок з доступом до всіх тих самих функцій Blazor Server, але без необхідності постійного HTTP-з’єднання до серверу.

Blazor WebAssembly — це фреймворк в екосистемі ASP.NET Core, який дозволяє розробникам створювати повноцінні інтерактивні веб-додатки за допомогою синтаксису C# і Razor. На відміну від традиційних односторінкових програм (SPA), які значною мірою покладаються на JavaScript фреймворки , Blazor WebAssembly переносить потужність екосистеми .NET безпосередньо в браузер.

В своїй основі Blazor WebAssembly використовує стандарт WebAssembly (Wasm) — бінарний формат інструкцій, який дозволяє коду, написаному

різними мовами, працювати з майже рідною швидкістю в сучасних веб-браузерах.

WebAssembly — це віртуальна машина, що працює в браузері. Мови високого рівня компілюються у формат асемблера низького рівня, який можна виконувати з продуктивністю, близькою до тієї, яка досягається під час виконання в звичайних умовах. WebAssembly надає доступ до API, які використовуються в JavaScript, що означає, що додатки WebAssembly можуть отримувати доступ до DOM, використовувати CSS та ініціювати асинхронні запити HTTP.

І WebAssembly, і Blazor WebAssembly ще на початку свого розвитку, і є деякі серйозні обмеження. WebAssembly — це нова технологія, яка підтримується лише останніми версіями браузерів. Використовувати WebAssembly неможливо, якщо проект потребує підтримки застарілих браузерів або навіть старіших версій сучасних браузерів. Додатки Blazor WebAssembly обмежені набором API, які надає браузер, що означає, що не всі функції .NET можна використовувати в програмі WebAssembly. Це не робить Blazor невигідним у порівнянні з фреймворками на стороні клієнта, такими як Angular, але це означає, що такі речі, як Entity Framework Core, недоступні, оскільки браузери обмежують додатки WebAssembly до надсилання HTTP-запитів.

Blazor WebAssembly використовує компоненти Razor, які є багаторазовими будівельними блоками інтерфейсу користувача. Кожен компонент інкапсулює логіку рендерингу, обробки подій і керування станом. Ця модель компонентів значною мірою спирається на сучасні фреймворки, такі як React або Angular, але з більш строгою, статично типізованою мовою.

Blazor WebAssembly використовується для створення односторінкових програм, які споживають дані з REST API. Припустимо, що в проекті WebAPI є наступний контролер:

```
[ApiController]
[Route("api/[controller]")]
public class StudentsController : ControllerBase
{
    private static List<Student> _students = new()
    {
        new Student { Id = 1, Name = "Alice", Age = 21, Major = "CS" },
        new Student { Id = 2, Name = "Bob", Age = 22, Major = "IT" }
    };

    [HttpGet]
    public ActionResult<List<Student>> Get() => _students;

    [HttpPost]
    public IActionResult Create(Student student)
    {
        student.Id = _students.Max(s => s.Id) + 1;
        _students.Add(student);
        return CreatedAtAction(nameof(Get), new { id = student.Id }, student);
    }

    [HttpPut("{id}")]
    public IActionResult Update(int id, Student student)
    {
        var existing = _students.FirstOrDefault(s => s.Id == id);
        if (existing is null) return NotFound();

        existing.Name = student.Name;
        existing.Age = student.Age;
        existing.Major = student.Major;
        return NoContent();
    }

    [HttpDelete("{id}")]
    public IActionResult Delete(int id)
    {
        var student = _students.FirstOrDefault(s => s.Id == id);
        if (student is null) return NotFound();

        _students.Remove(student);
        return NoContent();
    }
}
```

Щоб реалізувати компонент, який забезпечуватиме CRUD-операції для нашої сутності Students, ми повинні реалізувати компонент .razor із таким вмістом:

```
@page "/students"
@inject HttpClient Http

<h3>Students</h3>

@if (students == null)
{
    <p><em>Loading...</em></p>
}
else
{
    <ul>
        @foreach (var student in students)
        {
            <li>@student.Name - @student.Major (<button @onclick="() => 
DeleteStudent(student.Id)">Delete</button>)</li>
        }
    </ul>

    <h4>Add New Student</h4>
    <input @bind="newStudent.Name" placeholder="Name" />
    <input @bind="newStudent.Age" type="number" placeholder="Age" />
    <input @bind="newStudent.Major" placeholder="Major" />
    <button @onclick="AddStudent">Add</button>
}

@code {
    List<Student>? students;
    Student newStudent = new();

    protected override async Task OnInitializedAsync()
    {
        students = await 
Http.GetFromJsonAsync<List<Student>>("https://localhost:5001/api/students");
    }

    async Task AddStudent()
    {
        var response = await 
Http.PostAsJsonAsync("https://localhost:5001/api/students", newStudent);
        if (response.IsSuccessStatusCode)
        {
            students = await 
Http.GetFromJsonAsync<List<Student>>("https://localhost:5001/api/students");
            newStudent = new();
        }
    }

    async Task DeleteStudent(int id)
    {
        var response = await 
Http.DeleteAsync($"https://localhost:5001/api/students/{id}");
        if (response.IsSuccessStatusCode)
        {
            students = await 
Http.GetFromJsonAsync<List<Student>>("https://localhost:5001/api/students");
```

```
        }
    }
}
```

Загальний опис кожної частини цього компонента наведено в таблиці 1.

Таблиця 1. Блоки компонента Blazor

Блок	**Призначення**
Директива @page	Робить компонент доступним за URL-адресою (маршрутизація)
@inject	Впроваджує служби у компонент (наприклад, HttpClient)
HTML-розмітка	Визначає структуру та елементи інтерфейсу користувача
Блок @code	Містить стан компонента та логіку (методи, змінні, життєвий цикл компонента)
@bind	Забезпечує двостороннє зв’язування даних між інтерфейсом і властивостями C#
@onclick тощо	Обробка подій взаємодії користувача (наприклад, натискання кнопок, форми)

Модель Student визначена в проекті Blazor наступним чином:

```
public class Student
{
        public int Id { get; set; }
        public string Name { get; set; } = string.Empty;
        public int Age { get; set; }
        public string Major { get; set; } = string.Empty;
}
```

У проекті WebAPI має бути ввімкнено CORS, щоб можна було отримувати дані в клієнті Blazor. До файлу Program.cs проекту WebAPI слід додати наступні рядки:

```
app.UseCors(builder => builder
    .AllowAnyOrigin()
    .AllowAnyMethod()
    .AllowAnyHeader());
```

Також необхідно зареєструвати HttpClient у файлі Program.cs в проекті Blazor:

```
builder.Services.AddScoped(sp => new HttpClient { BaseAddress = new Uri("https://localhost:5001/") });
```

Хід роботи

Передумови:

Готовий проект Web API з лабораторної роботи №5.

Завдання 1.

Створіть у своєму рішенні проект Blazor WebAssembly, який взаємодіятиме з WebAPI. Розробіть сторінки та необхідні компоненти для виконання CRUD-операцій з існуючими сутностями. Додайте валідацію для створених форм.

Завдання 2.

Додайте автентифікацію та авторизацію до клієнта. Додайте сторінки та відповідні форми для реєстрації та входу. Реалізуйте автентифікацію на основі JWT. Приклад проекту, який реалізує автентифікацію та авторизацію за допомогою WebAPI та Blazor WebAssembly, можна знайти за наступним

посиланням. Докладніше з механізмами автентифікації та авторизації в Blazor WebAssembly можна ознайомитись тут.

Питання для самоконтролю

1. Що таке Blazor WebAssembly і чим він відрізняється від Blazor Server?
2. Чим компонент Razor відрізняється від стандартного класу C#?
3. Яка роль блоків @code у файлах Razor?
4. Що таке компонентна архітектура і чому вона корисна в Blazor?
5. Опишіть як ви використовували Web API із додатку Blazor WebAssembly.

Лабораторна робота №7

Тема: Створення веб-застосунків з підтримкою взаємодії в реальному часі з використанням SignalR

Мета: отримати практичний досвід інтеграції SignalR у веб-застосунок для забезпечення обміну даними в реальному часі за допомогою SignalR.

Теоретичні відомості

SignalR — це бібліотека для ASP.NET Core, яка дозволяє реалізувати обмін даними в реальному часі між сервером і клієнтами. Замість того, щоб клієнт постійно опитував сервер (polling) для отримання нових даних, SignalR забезпечує постійне двостороннє з’єднання, завдяки якому сервер самостійно надсилає оновлення клієнтам, щойно вони з’являються.

Бібліотека автоматично визначає найкращий спосіб з’єднання з-поміж доступних — WebSockets, Server-Sent Events або Long Polling, залежно від підтримки на стороні клієнта та сервера. Це дозволяє розробникам зосередитися на логіці застосунку, а не на реалізації транспорту повідомлень. Більш детальну інформацію про процес вибору способу транспортування можна знайти тут.

Основним елементом SignalR є Hub — клас, який обробляє виклики між сервером і клієнтами. За його допомогою клієнт може викликати методи на сервері, а сервер — викликати методи на підключених клієнтах.

Приклад простого хабу:

```
public class ChatHub : Hub
{
    public async Task SendMessage(string user, string message)
    {
        await Clients.All.SendAsync("ReceiveMessage", user, message);
    }
}
```

У цьому прикладі метод `SendMessage` приймає повідомлення від одного користувача й надсилає його всім клієнтам, викликаючи на них метод `ReceiveMessage`.

Щоб хаб працював, його потрібно зареєструвати в конфігурації застосунку (Program.cs):

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

app.MapHub<ChatHub>("/chathub");

app.Run();
```

На стороні клієнта (наприклад, на JavaScript) підключення до хабу виглядає так:

```
const connection = new signalR.HubConnectionBuilder()
    .withUrl("/chathub")
    .build();

connection.on("ReceiveMessage", (user, message) => {
    console.log(`${user}: ${message}`);
});

connection.start().catch(err => console.error(err.toString()));
```

У такій конфігурації користувач може надіслати повідомлення на сервер, і всі інші користувачі побачать його без оновлення сторінки.

Окрім базового функціоналу, SignalR підтримує такі корисні можливості, як:

- групи клієнтів (можна надсилати повідомлення лише певній групі),
- ідентифікація користувачів,
- обробка підключень і відключень (наприклад, вивід статусу онлайн/офлайн).

Це робить SignalR потужним інструментом для створення інтерактивних веб-застосунків у реальному часі.

Приклад реалізації ASP.NET проекту з використанням SignalR можна знайти за посиланням.

Хід роботи

Завдання.

Додайте SignalR до вашого MVC проекту розробленого під час попередніх лабораторних робіт. Реалізуйте функціональність, яка працюватиме в реальному часі, відповідно до тематики вашого проекту (табл. 1).

Таблиця 1. Функціональності для реалізації

Номер за списком	Тема проекту	Функціональність
1	Веб-сайт для бронювання квитків на заходи	Оновлення щодо наявності місць у режимі реального часу під час бронювання користувачами.
2	Система керування курсами	Миттєві сповіщення студентів про додавання нових курсів в бажаній категорії.
3	Портал вакансій	Миттєві сповіщення кандидатів про публікацію нової вакансії в бажаній категорії.
4	Система запису на прийом до лікарні	Оновлення доступних годин візиту у реальному часі, коли вони забронювані або скасовані.
5	Портал для опитувань	Оновлення результатів опитування у реальному часі.
6	Веб-сайт для бронювання номерів готелю	Оновлення інформації про доступність номерів у режимі реального часу.
7	Система для прокату транспорту	Миттєві оновлення щодо доступності транспортних засобів.

8	Веб-сайт бронювання квитків на кіносеанси	Оновлення схеми місць у режимі реального часу під час бронювання (блокування/розблокування місць у режимі реального часу).
9	Система управління волонтерськими зборами та благодійними заходами	Оновлення кількості реєстрацій та благодійних внесків у режимі реального часу.
10	Новинний портал	Відображення поставлених реакцій на публікації в реальному часі.

Опишіть додану функціональність у звіті.

Питання для самоконтролю

1. Що таке SignalR і яку проблему він вирішує?
2. Які методи транспортування підтримує SignalR і як він обирає метод для транспортування серед них?
3. Що таке Hub у SignalR і яку роль він виконує?
4. Як налаштувати SignalR Hub на сервері?
5. Як клієнти підключаються до SignalR Hub і отримують повідомлення?

Список рекомендованої літератури

1. ASP.NET documentation. URL: https://learn.microsoft.com/en-us/aspnet/core/
2. Lock A. ASP.NET Core in Action, Third Edition. Manning, 2023. 984 p.
3. Freeman A. Pro ASP.NET Core 7, Tenth Edition. Manning, 2023. 1256 p.
4. Malavasi A. Modern Full-Stack Web Development with ASP.NET Core: A project-based guide to building web applications with ASP.NET Core 9 and JavaScript frameworks. Packt, 2025. 446 p.
5. Mahdhi M. Mastering ASP.NET Core 10: Web Applications Made Easy with the Biggest Update Yet. Apress, 2025. 913 p.
6. ASP.NET Core. GitHub. URL: https://github.com/dotnet/aspnetcore