ТЕХНОЛОГІЇ РОЗРОБКИ КРОСПЛАТФОРМЕНИХ ВЕБ-ДОДАТКІВ. КОНСПЕКТ ЛЕКЦІЙ.

Book · March 2024

CITATIONS
0

READS
188

1 author:

Пономарьов Ігор Володимирович
Дніпровський національний університет імені Олеся Гончара
40 PUBLICATIONS **4** CITATIONS

SEE PROFILE

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДНІПРОВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ОЛЕСЯ ГОНЧАРА

І.В. Пономарьов

ТЕХНОЛОГІЇ РОЗРОБКИ КРОСПЛАТФОРМЕНИХ ВЕБ-ДОДАТКІВ

КОНСПЕКТ ЛЕКЦІЙ

Дніпро

ДНУ

2024

УДК 004(075.8)

Рекомендовано Вченою радою факультету ФЕКС
Дніпровського національного університету імені Олеся Гончара
протокол № 57 від 14 листопада 2023 року

Рецензенти:
Корчинський Володимир Михайлович – д.т.н., проф., зав. каф. ТСМ Дніпровського національного університету ім. О. Гончара,
Прокоф'єв Тихон Анатолійович – к.ф.м.н., доц, доц. каф. КНІТ Дніпровського національного університету ім. О. Гончара.

Технології розробки кросплатформених веб-додатків. Конспект лекцій / І.В. Пономарьов – Дніпро: ДНУ, 2023. – 113 с., іл.

У конспекті лекцій розглянуті основні особливості технології розробки кросплатформених веб-додатків. Проводиться огляд роботи платформи ASP.NET Core, концепції патерну MVC. Детально описані визначення контролеру та його методів дій, подання з можливостями використання двигуна Razor, моделі, поняття маршрутизації. Для усіх, описаних понять, наведені прості та зрозумілі приклади.

Для студентів першого рівня вищої освіти галузі знань 12 «Інформаційні технології» та студентів споріднених галузей, наукових та інженерно-технічних працівників відповідного профілю.

Навчально-методичний посібник

Ігор Володимирович Пономарьов

ЗМІСТ

ВСТУП

Розробка додатку відразу під кілька платформ стикається з багатьма складнощами. Для створення додатку під конкретну платформу використовуються відповідні їй інструменти, середовища розробки та мови програмування. На розробку впливає і відмінність у підходах побудови графічного інтерфейсу та специфічні особливості платформи. Наявність великого діапазону платформ, засобів розробки та мов програмування збільшує як часові, так і грошові витрати на розробку.

Кросплатформеність надає можливість легко і просто створювати додатки для декількох платформ та дозволяє значно скоротити витрати на розробку та час випуску продуктів.

За допомогою сучасної технології ASP.NET Core створюються кросплатформені додатки які можуть працювати в операційних системах Windows, Linux і Mac OS. Фреймворк ASP.NET Core MVC є частиною платформи ASP.NET Core і застосовує патерн MVC. Розробка веб-додатків з використанням технології ASP.NET Core MVC забезпечує розподіл функціональності, що спрощує тестування, модифікацію та супровід додатків.

У курсі лекцій розглядаються сучасні концепції, поняття та методи розробки кросплатформених веб-додатків з застосуванням даної технології.

Курс лекцій вибірковій дисципліни «Технології розробки кросплатформених веб-додатків» сформовано на основі досвіду розробки веб-додатків та викладання принципів їх проєктування та реалізації.

Курс лекцій допомагає організувати самостійну роботу студента для вивчення і систематизації знань за темами, пов’язаними із поняттями та термінами, що використовуються в області розробки кросплатформених веб-додатків, концепції патерну MVC, системи маршрутизації.

1. Огляд технологій розробки кросплатформених додатків

1.1. Види додатків

Настільна програма розробляється під **конкретну апаратно-програмну платформу**. Вона оптимізована під конкретну **операційну систему (ОС)**, має доступ до **апаратної частини**, **встановлюється** на комп'ютері.

Термін **нативний додаток** (від англ. native - рідний) застосовується для мобільного додатка.

Переваги настільної програми:

- працює без виходу в мережу,
- безпека.

Недолік - несумісна з іншими операційними системами.

Веб-програма працює через веб-**браузер** на пристрої користувача. Це веб-сайт, який виглядає як настільна програма, але розміщується не на пристрої користувача.

Переваги веб-програми:

- може функціонувати на платформі з будь-якою ОС,
- не завантажується на комп'ютер,
- доступ до веб-програми з будь-якого пристрою.

Недолік - використання можливе тільки через Інтернет.

1.2. Технології розробки кросплатформених додатків

Технологія розробки кроссплатформенного додатка дозволяє розгортати та виконувати його на **різних операційних системах** та **різних веб-серверах**.

Веб-додаток - **клієнт-серверна** програма, в якій клієнт взаємодіє з сервером за допомогою **браузера**, а за сервер відповідає - **веб-сервер** (рис 1.1).

Логіка веб-додатка розподілена між сервером та клієнтом:

- зберігання даних здійснюється, переважно, на сервері,
- обмін інформацією відбувається через мережу.

Веб-сервер – це програмний продукт, який відповідає за розміщення в Інтернеті веб-додатків. Він зазвичай надає безліч взаємозалежних служб, таких як інтегрована безпека, FTP (File Transfer Protocol – протокол передачі файлів), поштовий обмін тощо.

Internet Information Services (IIS) для Windows® Server — це веб-сервер виробничого рівня від Microsoft, який має вбудовану підтримку веб-додатків класичного ASP та ASP.NET.

Веб-сервери для платформ, сумісних із UNIX: **Nginx** та **Apache**.

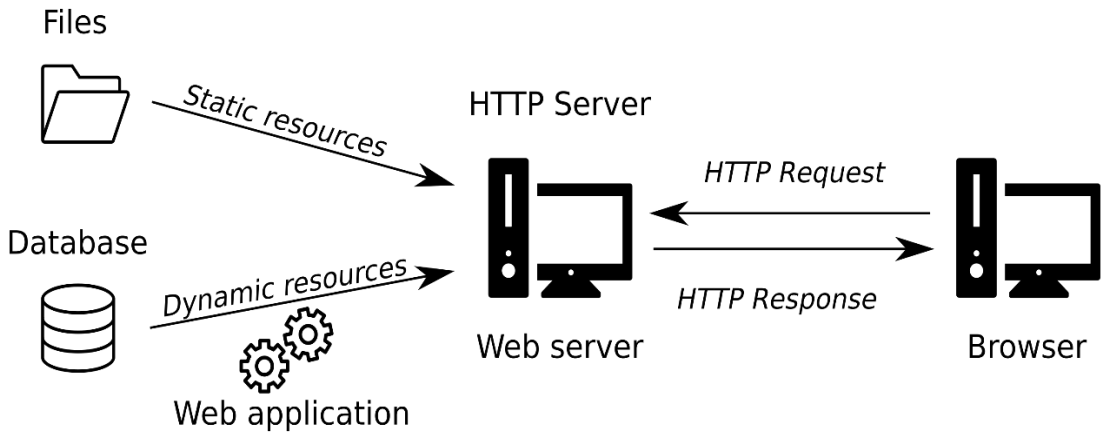


Рис. 1.1. Робота веб-додатка

Мови програмування для розробки веб-додатків поділяються на:

- **клієнтські** та
- **серверні.**

Для розробки сайту використовуються наступні мови програмування:

• **для реалізації графічного інтерфейсу користувача (GUI)**:

- HTML, CSS, JavaScript;

• для формування запитів, створення інтерактивного та незалежного від браузера інтерфейсу, роботи з даними та базами даних:

- PHP
- ASP.NET
- Python
- Java
- Ruby
- Perl і т.д.

Front-end та back-end розробка веб-додатку

Front-end розробка відноситься до будь-якого аспекту процесу розробки, який проявляється при завантаженні **сторінки в браузері** або має безпосереднє відношення до цього.

Наступні завдання зазвичай відносять до front-end розробки:

• графічний дизайн та підготовка зображень,

• дизайн інтерфейсу,

• інформаційний дизайн, тією мірою, якою він має відношення до досвіду **взаємодії користувача з сайтом;**

• верстка HTML-документів та таблиць стилів;

• JavaScript.

Back-end-розробка відноситься до програм та сценаріїв, які потай **виконуються на стороні сервера**, щоб зробити веб-сторінки динамічними та інтерактивними.

Наступні завдання належать до back-end розробки:

• інформаційний дизайн, тією мірою, якою він має відношення до **організації даних на сервері;**

• обробка форм;

• програмування баз даних;

• системи керування контентом;

• інші веб-програми, що працюють **на стороні сервера**, що використовують PHP, JSP, Ruby, ASP.NET, Java та інші мови програмування.

Рейтинг мов програмування

Рейтинг мов програмування та платформ розробки 2023 року приведений на рис. 1.2 (https://survey.stackoverflow.co/2023/#technology).

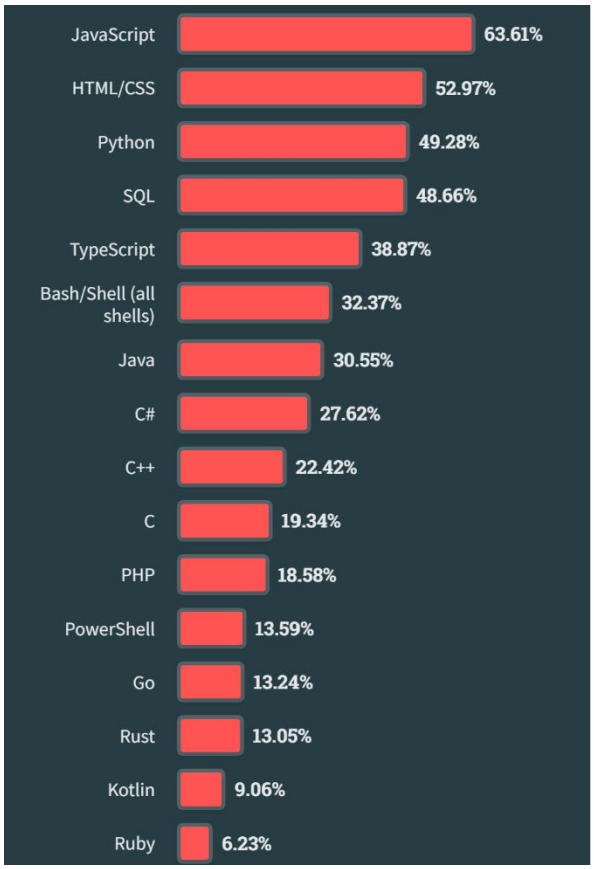


Рис. 1.2. Рейтинг мов програмування та платформ розробки

Український рейтинг використання мов програмування 2022 року приведений на рис. 1.3 (https://dou.ua/lenta/articles/language-rating-2022).

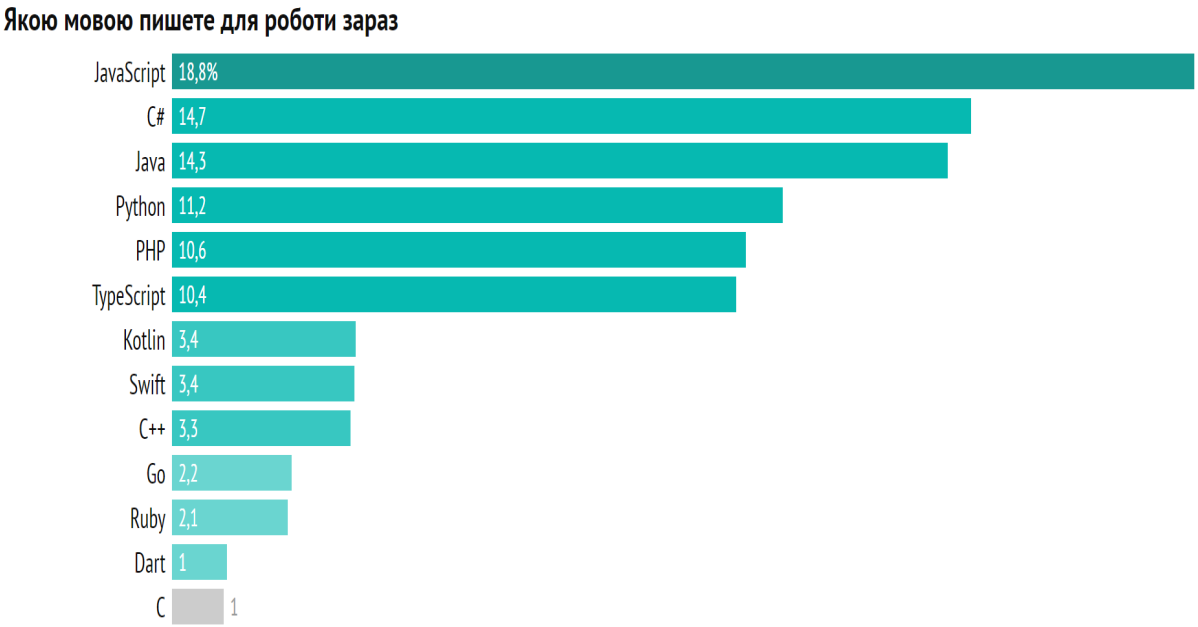


Рис. 1.3. Український рейтинг мов програмування

1.3. Платформи для створення веб-додатків розробки Microsoft

Active Server Pages активні серверні сторінки (Microsoft). **ASP** – це технологія **динамічного створення сторінок на стороні сервера**. Для реалізації програм ASP використовуються мови сценаріїв (VBScript або JScript).

ASP.NET (Active Server Pages для .NET) — платформа розробки веб-застосунків, до складу якої входить: веб-сервіси, програмна інфраструктура, модель програмування від компанії Майкрософт.

ASP.NET входить до складу платформи **.NET Framework\Core** і є розвитком більш старої технології Microsoft ASP.

Xamarin - це фреймворк для кросплатформної розробки мобільних додатків (iOS, Android, Windows Phone) з використанням мови C#.

.NET MAUI (Multi-platform App UI) - кроссплатформенний фреймворк для створення власних мобільних та настільних додатків за допомогою C# та XAML (Android, iOS, macOS, Windows, Tizen).

1.4. Платформи для створення веб-додатків розробки на базі JavaScript

Node або Node.js - програмна платформа, заснована на двигуні V8 (що транслює JavaScript в машинний код), що перетворює **JavaScript** з вузькоспеціалізованої мови на мову загального призначення. Node.js додає можливість JavaScript взаємодіяти з пристроями вводу-виводу через свій API (написаний на C++), підключати інші зовнішні бібліотеки, написані різними мовами, забезпечуючи виклики до них із JavaScript-коду. Node.js застосовується переважно **на сервері**, виконуючи роль веб-сервера, але є можливість розробляти на Node.js та десктопні віконні програми (за допомогою **NW.js**, **AppJS** або **Electron** для Linux, Windows та macOS).

AngularJS - **JavaScript-фреймворк** з відкритим вихідним кодом. Призначений для розробки односторінкових програм. Його мета – розширення браузерних додатків на основі **MVC-шаблону**, а також спрощення тестування та розробки. Фреймворк працює з **HTML**, що містить додаткові атрибути користувача, які описуються директивами, і пов'язує введення або виведення області сторінки з **моделлю**, що являє собою звичайні змінні JavaScript. Значення цих змінних задаються вручну або витягуються зі **статичних або динамічних даних JSON**.

React (іноді React.js або ReactJS) — **JavaScript-бібліотека** з відкритим вихідним кодом для розробки інтерфейсів користувача. React розробляється та підтримується Facebook, Instagram та спільнотою окремих розробників та корпорацій. React може використовуватися для розробки **односторінкових** та **мобільних додатків**. Його мета — надати високу швидкість, простоту та масштабованість.

1.5. Етапи розробки сайту

1. **Передпроектна підготовка**

1.1. Передпроектні дослідження

Ознайомлення з проектом, уточнення цілей та завдань. Вивчення бізнесу клієнта, визначення та аналіз цільової аудиторії. Аналіз конкурентів. Складання календарного плану робіт. Формування бюджету та робочої групи.

1.2. Розробка технічного завдання

Розробка та затвердження остаточного технічного завдання, що включає **вимоги до дизайну** та вимоги до технічної частини проекту. Розробка структури сайту (мапи сайту).

2. **Розробка та узгодження дизайну**

2.1. Дизайн-концепція сайту (креативний дизайн). Креативна ідея розробка основної графічної концепції дизайну сайту на прикладі головної сторінки. Адаптація елементів фірмового стилю клієнта для веб-сайту.

2.2. Технічний дизайн

Створення графічних шаблонів типових сторінок сайту на основі затвердженої концепції дизайну.

3. **Верстка**

Верстка HTML-сторінок сайту на основі затвердженого дизайну типових сторінок.

4. **Програмна частина проекту**

4.1. Інтеграція сайту із системою управління

Жоден сучасний сайт не обходиться без системи управління, тому що важлива не лише гарна зовнішня оболонка цього сайту, але й можливість зручної роботи з ним. Це особливо актуально для сайтів із розгалуженою структурою та великим обсягом даних. У цей етап входить: інтеграція із системою управління,

програмування, налаштування сервера, забезпечення безпеки проекту. Контроль якості.

4.2. Програмування, запуск проекту

На цьому етапі допрацьовується функціонал, що не включений до стандартного складу системи управління.

5. **Інформаційне наповнення сайту**

Інформаційне наповнення сайту необхідними фотографіями та контентом. З боку клієнта важливо підійти до цього етапу, заздалегідь підготувавши всю необхідну інформацію для сайту.

6. **Тестування сайту в Інтернеті**

Тестування працездатності сайту на наявність помилок, тестування HTML-сторінок на коректність роботи у різних браузерах (Internet Explorer, Chrome, Edge, Opera, Firefox, Safari).

7. **Здача сайту в експлуатацію**

Організація робіт з розміщення проекту в Інтернеті на **домені клієнта**. Фінальне тестування проекту. Навчання персоналу клієнта роботи із системою управління сайту.

8. Просування сайту

Комплекс дій для забезпечення високих позицій ресурсу в пошукових системах з метою підвищення відвідуваності сайту цільовою аудиторією.

1.6. Протокол HTTP

Веб-додаток завжди має на увазі існування як мінімум **двох комп'ютерів**, об'єднаних у мережу:

- на одному розгорнутий веб-сайт, а з

- іншого переглядаються дані за допомогою веб-браузера.

Під час розробки допускається, щоб ролі браузерного клієнта та веб-сервера, що обслуговує вміст, виконувались на одній машині.

Мережевий протокол, що з'єднує ці комп'ютери, називається **HTTP** (Hypertest Transfer Protocol - протокол передачі гіпертексту).

Гіпертекст - текст, що дозволяє не тільки послідовне прочитання, але і пов'язаний **покажчиками-посиланнями** з іншими текстами.

Цикл запит/відповідь HTTP

Коли клієнтська машина запускає веб-браузер, HTTP-запит виконується для доступу до певного ресурсу (зазвичай веб-сторінці) на віддаленій серверній машині.

HTTP є заснований на тексті протокол, побудований на базі стандартної парадигми **запит/відповідь**.

Універсальний покажчик ресурсу URL

Універсальний покажчик ресурсу URL (Universal Resource Locator) однозначно **ідентифікує будь-який ресурс у Інтернеті**. Саме такий рядок відображається в адресному полі браузера.

http://www.site.lviv.ua/documents/page.html або
http://213.82.46.1/documents/page.html

Універсальний покажчик ресурсу містить:

http://	**www.**	**site.lviv.ua/**	**documents/**	**page.html**
протокол	**служба**	**доменна адреса**	**шлях**	**файл**

• **протокол відповідної служби**. У цьому прикладі використано протокол http:// - протокол передачі гіпертексту;

• **назву служби**. У цьому прикладі це служба Веб – www;

• **доменну або IP-адресу**, яка однозначно ідентифікує веб-сервер у мережі Інтернет, на якому розміщено потрібний сайт чи інший ресурс;

• **шлях**, що складається з імен директорій, розділених символом "/" (слеш), послідовно відкриваючи як можна «добратися» до потрібної інформації. (В прикладі інформація знаходиться у директорії «documents»);

• **ім'я файлу**, який містить необхідну інформацію. (В прикладі інформація міститься у файлі page.html).

Цикл запит/відповідь HTTP

Наприклад, якщо здійснюється перехід до ресурсу **http://www.facebook.com**, то програмне забезпечення браузера покладається на веб-технологію служби доменних імен (Domain Name Service, **DNS**), яка перетворює запитаний **URL** на **IP-адресу** (32-бітове числове значення, що складається з чотирьох частин).

І в цей момент браузер відкриває **сокетне з'єднання** (зазвичай через **порт 80** для незахищених підключень) та посилає **HTTP-запит** для обробки на цільовому **сайті**.

Веб-сервер приймає вхідний запит HTTP і може обробити будь-які надіслані клієнтом вхідні значення (значення, введені в текстових полях, прапорці, перемикачі тощо), що формують правильну **відповідь HTTP** (рис 1.4).

Після цього **браузер** клієнтської сторони **візуалізує розмітку HTML**, відправлену веб-сервером.

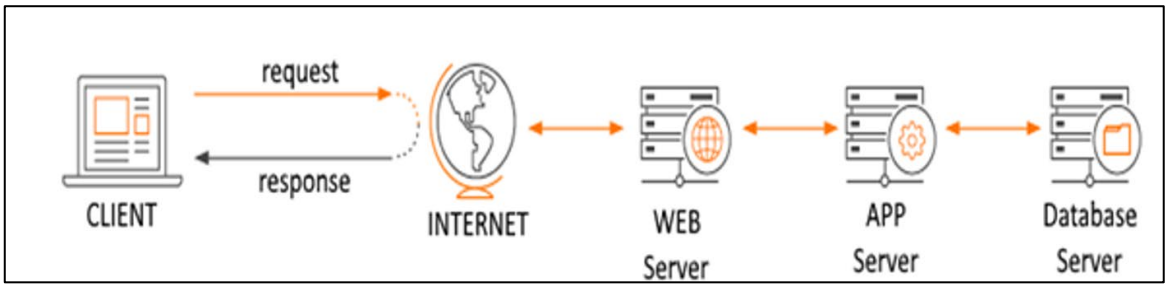


Рис. 1.4. Цикл запит/відповідь HTTP

1.7. Платформа ASP.NET Core

ASP.NET

Попередні версії ASP.NET були специфічними для операційної системи **Windows**, вимагаючи настільний комп'ютер з ОС Windows для написання веб-застосунків і **сервер Windows (Internet Information Services, IIS)** для їх розгортання та виконання.

ASP.NET Core

Компанія Microsoft зробила інфраструктуру ASP.NET Core **міжплатформною**, як щодо розробки, так і в плані розгортання. Продукт .NET Core доступний для різних платформ, включаючи **Linux** і **OS X/macOS** і буде переноситися на інші платформи.

Розробники додатків на платформі ASP.NET Core **незалежні від ОС та веб-серверів:**

• розробка та розгортання можливі на всіх основних ОС **Windows, Mac** та **Linux;**

• можна вільно вибрати різні **веб-сервера**:

  • **IIS,**

  • включені до платформи веб-сервери:

- **кросплатформовий Kestrel,**
- Web Listener;

• інші сервери:

- **Apache,**
- **Ngnix.**

Для розробки веб-додатків особливо добре підходить шаблон **ASP.NET Core MVC.**

Для успішного освоєння технології розробки кросплатформених додатків з допомогою шаблону ASP.NET Core MVC необхідна наявність знаннь у наступних технологіях.

1. Мова розмітки **HTML**, каскадні таблиці стилів **CSS** [3, Додаток А].
2. Мова програмування **C#** (асинхронне виконання, лямда-вирази) [1,2].
3. Об'єктно-орієнтоване програмування (**ООП**) на **C#** [1]:
   - інкапсуляція: клас (поля, властивості, методи, конструктори, методи доступу до членів класу),
   - успадкування (абстрактний клас, інтерфейс),
   - поліморфізм (перевантаження методів).
4. **Бази даних**, система управління базами даних (СУБД) SQL Server, мова запитів **SQL** [Додаток В].

1.8. Контрольні питання

1. Які є види додатків?
2. Що таке кроссплатформенний додаток?
3. Які особливості розробки веб-додатку?
4. Які платформи використовуються для створення веб-додатків?
5. Що таке протокол HTTP?
6. Які можливості платформи ASP.NET Core?

2. Концепція патерну MVC

Фреймворк ASP.NET Core MVC є частиною платформи ASP.NET Core, його відмінна риса – застосування патерну MVC.

Перевагою використання фрейморка ASP.NET Core MVC у порівнянні з "чистим" ASP.NET Core є те, що він **спрощує** в ряді ситуацій та сценаріїв **організацію та створення додатків**, особливо це стосується великих додатків.

Паттерн MVC з'явився ще наприкінці 1970-х років і в даний час застосовується у багатьох платформах та для різних мов програмування. Особливо популярний патерн MVC у веб-додатках.

Концепція патерну MVC передбачає поділ програми на **три компоненти**:

• **Модель (model)** описує дані, що використовуються в додатку, а також логіку, яка пов'язана безпосередньо з даними, наприклад, логіку валідації даних. Як правило, об'єкти моделей зберігаються у базі даних.

У MVC моделі представлені двома основними типами:

- моделі уявлень, які використовуються уявленнями для відображення та передачі даних, та
- моделі домену, що описують логіку управління даними.

Модель може містити дані, зберігати логіку управління цими даними. У той самий час модель не має містити логіку взаємодії з користувачем і не має визначати механізм обробки запиту. Крім того, модель не повинна містити логіку відображення даних у поданні.

• **Подання (view)** відповідають за візуальну частину або інтерфейс користувача, нерідко це HTML-сторінка, через яку користувач взаємодіє з додатком. Також уявлення може містити логіку, пов'язану з відображенням даних. У той же час, подання не повинно містити логіку обробки запиту користувача або управління даними.

• **Контролер (controller)** представляє центральний компонент MVC, який забезпечує зв'язок між користувачем та додатком, поданням та сховищем даних.

Він містить **логіку обробки запиту користувача**. Контролер **отримує дані**, що вводяться користувачем, і **обробляє їх**. І в залежності від результатів обробки надсилає користувачеві певне подання, наприклад, у вигляді уявлення, наповненого даними моделей.

Відносини між компонентами патерну описуються схемою приведеною на рис. 2.1.

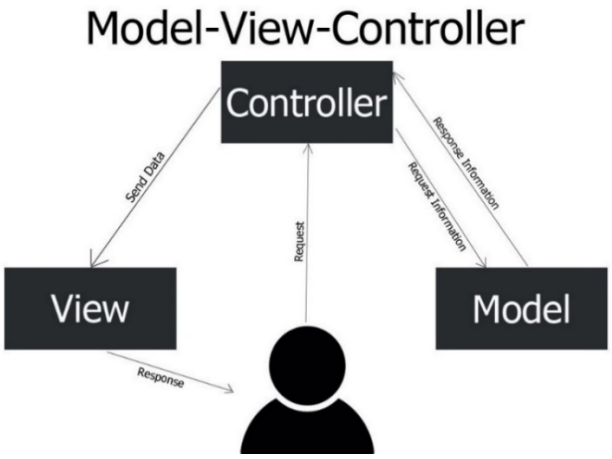


Рис. 2.1. Відносини між компонентами патерну MVC

У цій схемі модель є незалежним компонентом – будь-які зміни контролера чи подання ніяк не впливають на модель. Контролер та подання є відносно незалежними компонентами. Так, з подання можна звертатися до певного контролера, а з контролера генерувати подання, але при цьому нерідко їх можна змінювати незалежно один від одного (рис. 2.2).

Таке розмежування компонентів програми дозволяє реалізувати концепцію поділу відповідальності, коли кожен компонент відповідає за свою суворо окреслену сферу. У зв'язку із чим легше побудувати роботу над окремими компонентами. І завдяки цьому додаток легше розробляти, підтримувати та тестувати окремі компоненти.

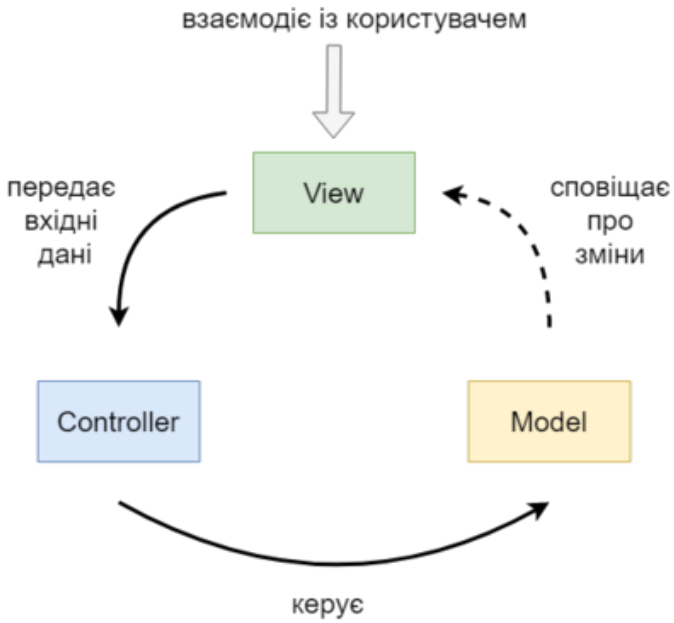


Рис. 2.2. Взаємодія між компонентами патерну MVC

Припустимо, якщо важлива візуальна частина чи фронтенд, то можна тестувати подання незалежно від контролера. Або можна зосередитися на бекенді і тестувати контролер.

2.1. Сервіси MVC

Функціональність MVC та її робота в додатку залежить від **сервісів**, що додаються.

Є ряд опцій з вбудовування сервісів, які використовуваються за необхідності, викликаючі наступні методи:

• **AddControllers()** дозволяє використовувати **контролери**, але без уявлень.

• **AddControllersWithViews()** додає лише ті сервіси фреймворку MVC, які дозволяють використовувати **контролери** та **подання** та пов'язану функціональність. При створенні проекту типу **ASP.NET Core Web App (Model-View-Controller**) використовується саме цей метод.

• **AddMvcCore()** додає лише **основні сервіси фреймворку MVC**, а всю додаткову функціональність, типу автентифікації та авторизації, валідації тощо, необхідно додавати самостійно.

• **AddMvc()** додає **всі сервіси фреймворку MVC** (у тому числі сервіси для роботи з автентифікацією та авторизацією, валідацією тощо).

2.2. Структура проекту шаблону ASP.NET Core Web App (Model-View-Controller)

Для створення проекту на ASP.NET Core MVC можна вибрати будь-який тип проекту на ASP.NET Core і вже додавати необхідні компоненти.

Для спрощення розробки, Visual Studio вже за промовчанням надає шаблон **ASP.NET Core Web App (Model-View-Controller)**. Структура створюваного проекту включає низку папок та файлів (рис. 2.3):

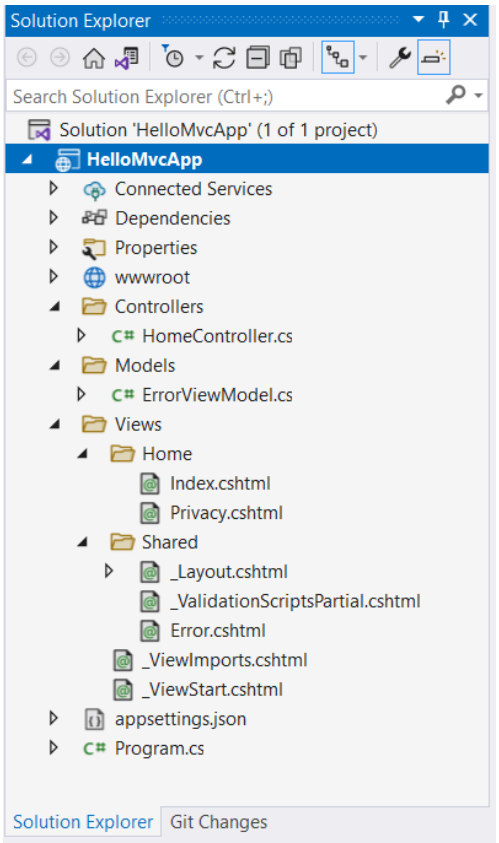


Рис. 2.3. Структура проекту ASP.NET Core Web App (Model-View-Controller)

• **Dependencies:** всі додані до проекту пакети та бібліотеки.
• **wwwroot**: цей вузол (на жорсткому диску йому відповідає однойменна папка) призначений для зберігання статичних файлів - зображень, скриптів javascript, css і т.д., які використовуються додатком.
• **Controllers**: папка для зберігання контролерів, які використовуються програмою. За замовчуванням тут вже є один контролер - **Homecontroller**.
• **Models**: каталог зберігання моделей. За замовчуванням створюється модель ErrorviewModel.
• **Views**: каталог для зберігання уявлень. Тут також за замовчуванням додаються ряд файлів уявлень.
• **appsettings.json**: зберігає конфігурацію програми.
• **Program.cs**: файл, який визначає вхідну точку в ASP.NET Core.

2.3. Контрольні питання

7. Яка концепція патерну MVC?
8. Які компоненти входять до патерну MVC?
9. Що таке подання (view)?
10. Яка структура створюваного проекту MVC?
11. Як у додатоку підключається підтримка контролерів та уявлень?
12. В якій папці проекту зберігаються подання?
13. Яке розширення мають файли подання?

3. Контролери

Основним елементом в архітектурі ASP.NET Core MVC є **контролер**. При надходженні запиту **система маршрутизації** вибирає **відповідний контролер** для обробки запиту і передає йому дані **запиту**. Контролер **обробляє ці дані** і відправляє назад результат обробки.

У ASP.NET Core MVC контролер представляє звичайний **клас C#**, який зазвичай успадковується від абстрактного базового класу Microsoft.AspNetCore.Mvc.**Controller**. **Контролер**, як і будь-який клас в C#, може мати **поля, властивості, методи**.

3.1. Дії контролера

Ключовим елементом контролера є його дії. Дії контролера - це **відкриті методи**, які можуть зіставлятися із запитами.

Наприклад, визначимо контролер HomeController з методом Index:

```
using Microsoft.AspNetCore.Mvc;
namespace MvcApp.Controllers
{
  public class HomeController : Controller
   {
        public string Index()
           {
                return "Hello DNU!";
           }
   }
}
```

*Метод Index має модифікатор **public** і тому може використовуватись для обробки запиту. Цей метод повертає рядок. А це означає, що при зверненні до цієї дії програма відправить у відповідь користувачеві цей рядок.*

Існують деякі умовності при використанні **контролерів**.

1. У проекті контролери розміщуються в каталозі **Controllers**. Однак це не обов'язково в принципі – можна додавати контролери до інших папок або навіть до кореня проекту.

2. Згідно з **умовами іменування, імена** контролерів зазвичай закінчуються суфіксом «**Controller**», в той час як інша частина цього суфікса вважається ім'ям контролера, наприклад, **HomeController.** Але в принципі ці умовності теж необов'язкові.

Але існують і обов'язкові умовності, які накладаються на контролери.

3. Зокрема, клас контролера повинен відповідати **принаймні одній** з наступних умов:

1) Клас контролера має суфікс «**Controller**»

```
public class HomeController
{
     //............
}
```

2) Клас контролера **успадковується від класу Controller**

```
public class Home : Controller
{
     //............
}
```

3) До класу контролера застосовується **атрибут [Controller]**

```
[Controller]
public class Home
{
     //................
}
```

3.2. Звернення до дій контролера

Зіставлення запиту з контролером та його дією відбувається завдяки **системі маршрутизації**.

У файлі Program.cs налаштовується **зіставлення запитів із контролерами**:

```
var builder = WebApplication.CreateBuilder(args);
builder.Services.AddControllers(); // додаємо підтримку контролерів
var app = builder.Build();
          // встановлюємо зіставлення маршрутів із контролерами
app.MapControllerRoute(
```

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

Метод app.MapControllerRoute() додає один маршрут з іменем default та шаблоном "{controller=Home}/{action=Index}/{id?}".

Цей шаблон встановлює **трисегментну** структуру рядка запиту:

```
controller/action/id
```

Спочатку йде **назва контролера**, потім **назва дії**, і далі може йти необов'язковий **параметр id**.

Щоб звернутися до контролера з веб-браузера, потрібно в адресному рядку набрати **адресу_сайту/ім'я_контролера/дія_контролера**.

*За запитом **адреси_сайту/Home/Index** (**localhost:xxxx/Home/Index**) система маршрутизації за замовчуванням викличе **метод Index контролера HomeController** для обробки вхідного запиту.*

3.3. Типи запитів

Методи можуть обслуговувати різні **типи запитів**. Для зазначення типу запиту HTTP треба застосувати до методу один із атрибутів:

[HttpGet], [HttpPost], [HttpPut], [HttpPatch], [HttpDelete] та [HttpHead].

Якщо атрибут явно не зазначений, то метод може обробляти всі типи запитів:

GET, POST, PUT, DELETE.

Наприклад:

```
using Microsoft.AspNetCore.Mvc;

namespace MvcApp.Controllers
{
    public class HomeController : Controller
    {
        [HttpGet]
        public string Index() => "Hello DNU!";

        [HttpPost]
        public string Hello() => "Hello ASP.NET";
    }
}
```

*У даному випадку метод Index обробляє лише **запити типу GET**, а метод Hello – запити типу **POST**.*

3.4. Передача даних до контролера через рядок запиту

Отримання даних через рядок запиту

Разом із запитом додаток може отримувати різноманітні дані. І щоб одержати ці дані, можливо використати різні способи.

Найпоширенішим способом вважається **застосування параметрів**. Визначення у методах контролера параметрів нічим не відрізняється від визначення параметрів у мові C#.

Наприклад, визначимо у контролері наступний метод:

```
using Microsoft.AspNetCore.Mvc;

namespace MvcApp.Controllers
{
    public class HomeController : Controller
    {
        public string Index(string name) => $"Your name: {name}";
    }
}
```

У цьому випадку метод Index отримує ззовні якийсь рядок через параметр name.

Передавати значення для параметрів можна різними способами.

При надсиланні **GET-запиту** значення можна передати **через рядок запиту**. Рядок запиту являє собою ту частину адреси, яка йде після знаку запитання **?** і являє собою набір параметрів, де кожен параметр відокремлюється від іншого знаком об'єднання **&**:

назва_ресурсу**?**параметр1=значення1**&**параметр2=значення2

Наприклад, в адресі:

```
https://localhost:7288/Home/Index?name=Bogdan&age=20
```

*частина **?name=Bogdan&age=20** являє собою рядок запиту, який містить два параметри: name і age. Значення параметра name – "Bogdan", а значення параметра age – 20.*

Наприклад, передаємо у вище визначений метод Index через рядок запиту дані для параметра name (рис. 3.1):

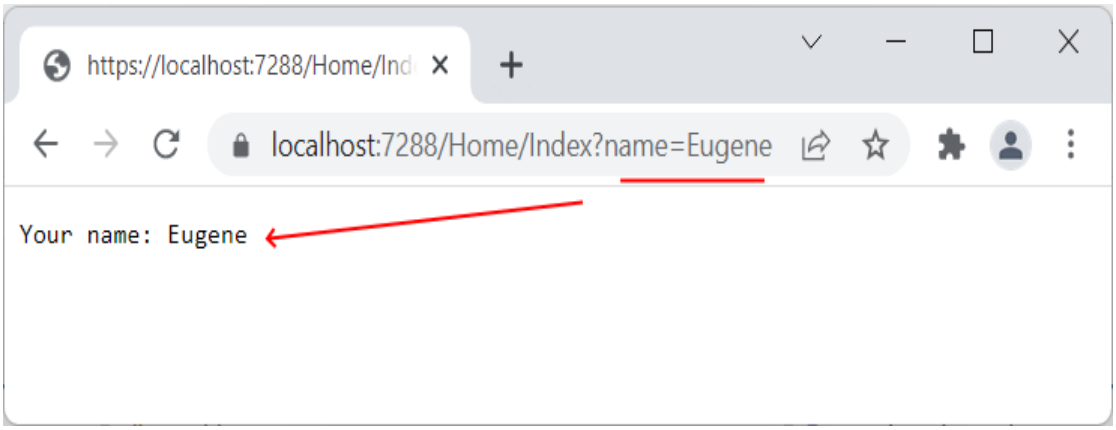


Рис. 3.1. Передача параметра методу контролера через рядок запиту

Тобто в даному випадку при зверненні до методу Index із запитом https://localhost:7288/Home/Index?name=Eugene параметру name передаватиметься значення "Eugene".

Система прив'язки MVC за промовчанням **зіставляє параметри запиту та параметри методу за назвою**. Тобто, якщо в рядку запиту йде параметр name, його значення буде передаватися саме параметру методу, який також називається name. При цьому має бути також **відповідність за типом,** тобто якщо параметр методу приймає числове значення, то через рядок запиту треба передавати для цього параметра число, а не рядок.

Так само можна передати значення для кількох параметрів.

Наприклад, змінимо метод Index наступним чином:

```
public string Index(string name, int age)
{
    return $"Name: {name}  Age: {age}";
}
```

У цьому випадку можна звернутися до дії контролера, набравши в адресному рядку https://localhost:7288/Home/Index?name=Tom&age=37 (рис. 3.2).

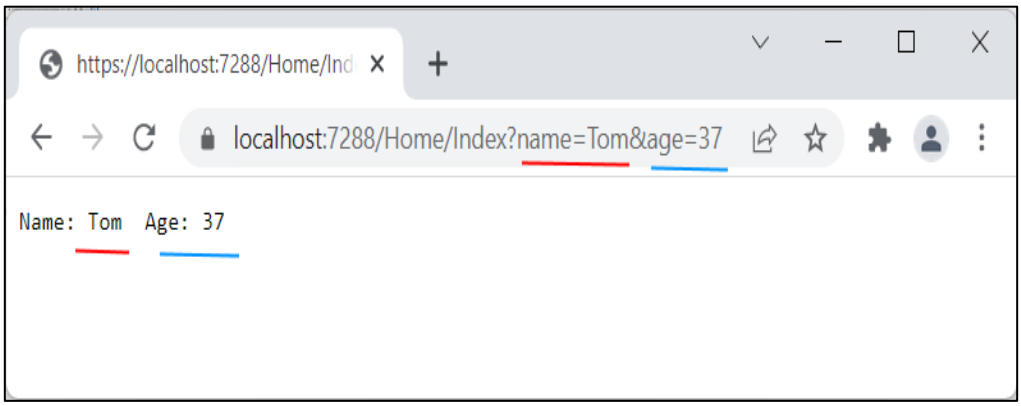


Рис. 3.2. Передача параметрів методу контролера через рядок запиту

Якщо в рядку запиту **не використовуємо параметри**, то для параметрів будуть передаватися **значення за замовчуванням**.

Наприклад, при надсиланні запиту https://localhost:7288/Home/Index параметри name і age будуть мати **значення за замовчуванням**. У такій ситуації можна надати **параметри за замовчуванням**, які працюватимуть, якщо через рядок запиту не передається жодних параметрів (рис. 3.3):

```
public string Index(string name = "Bob", int age = 33)
{
    return $"Name: {name}  Age: {age}";
}
```

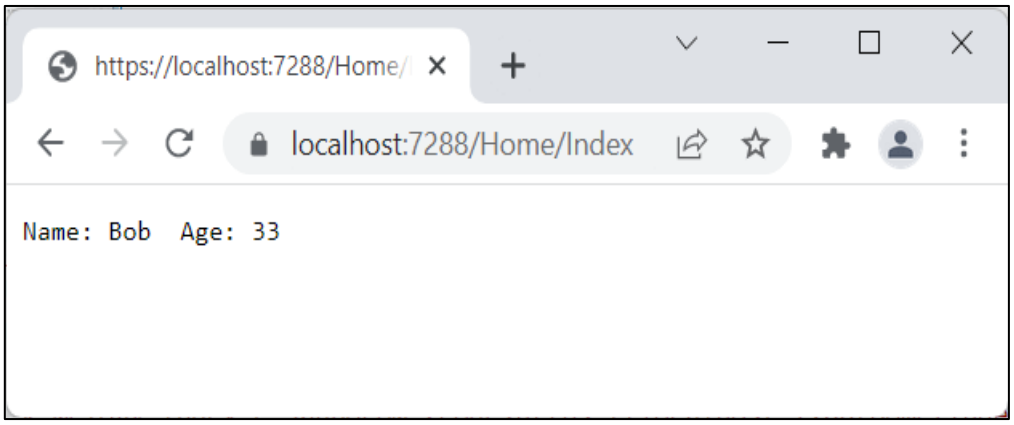


Рис. 3.3. Використання параметрів за замовчуванням методу контролера

Передача складних об'єктів

Рядок запиту використовується для передачі даних переважно примітивних типів, але можна отримувати й складніші об'єкти.

Наприклад, визначимо клас Person поруч із контролером HomeController:

```
using Microsoft.AspNetCore.Mvc;

namespace MvcApp.Controllers
{
```

```
    public class HomeController : Controller
    {
     public string Index(Person person)
      {
       return $"Person Name:{person.Name} Person Age:{person.Age}";
      }
    }
    public record class Person(string Name, int Age);
}
```

Клас Person визначає дві властивості: Name та Age. І в контролері HomeController метод Index приймає ***параметр типу Person****. У цьому випадку значення через рядок запиту передаються, як і в попередньому випадку.*

Параметри рядка запиту повинні бути **відповідні за ім'ям властивостям об'єкта**. Регістр букв назв параметрів при цьому не враховується (рис. 3.4):

Рис. 3.4. Передача параметрів методу контролера через рядок запиту

Передача масивів

Припустимо, що метод контролера приймає масив рядків-імен:

```
public class HomeController : Controller
{
    public string Index(string[] people)
    {
        string result = "";
```

```
            foreach (var person in people)
                result = $"{result}{person}; ";
            return result;
        }
    }
```

Для передачі даних методу через рядок запиту використовується назва параметра (рис. 3.5):

https://localhost:7288/Home/Index?people=Tom&people=Bob&people=Sam

Рис. 3.5. Передача параметрів методу контролера через рядок запиту

3.5. Передача даних у контролер через форми

Передача даних через форми у **запиті POST.**

POST-запити застосовуються так само широко, як і GET-запити. Як правило, такі запити надсилаються за **допомогою форм на веб-сторінці**. Основні принципи передачі даних є тими самими, що й у GET-запитах.

Для передачі POST-запитів визначимо контролер HomeController з двома методами Index:

```
using Microsoft.AspNetCore.Mvc;

namespace MvcApp.Controllers
```

```
{
  public class HomeController : Controller
    {
        [HttpGet]
     public async Task Index()
        {
            string content = @"<form method='post'>
                <label>Name:</label><br />
                <input name='name' /><br />
                <label>Age:</label><br />
                <input type='number' name='age' /><br />
                <input type='submit' value='Send' />
            </form>";
            Response.ContentType = "text/html;charset=utf-8";
            await Response.WriteAsync(content);
        }
        [HttpPost]
     public string Index(string name, int age) => $"{name}: {age}";
    }
}
```

Перший метод Index має **атрибут [HttpGet],** тому цей метод оброблятиме лише **запити GET**. Для спрощення прикладу у відповідь метод повертатиме HTML-код з формою введення даних (хоча звісно, для форми HTML можна було б визначити окрему HTML-сторінку або подання).

Ця форма містить два поля введення даних. Що важливо, перше поле має ім'я "name", яке задається за допомогою атрибуту "name":

```
<input name='name' />
```

Друге поле має ім'я "age":

```
<input type='number' name='age' />
```

Таким чином, при зверненні до методу користувач побачить у браузері форму введення даних.

При натисканні на кнопку Send введені дані будуть надсилатись на сервер.

Оскільки у елемента <form> не заданий атрибут action, який встановлює адресу, то введені дані відправляються на ту саму адресу (тобто, власне, методу з тим самим ім'ям – методу Index).

Але оскільки у форми встановлено **атрибут method='post'**, то дані будуть надсилатися у **запиті типу POST**. А запити даного типу опрацьовує другий метод Index:

```
[HttpPost]
public string Index(string name, int age) => $"{name}: {age}";
```

Щоб система могла **зв'язати параметри методу та дані форми**, необхідно, щоб **атрибути name у полів форми відповідали назвам параметрів**.

Тобто в даному випадку параметри методу Index називаються так само, як поля форми – name і age (рис. 3.6).

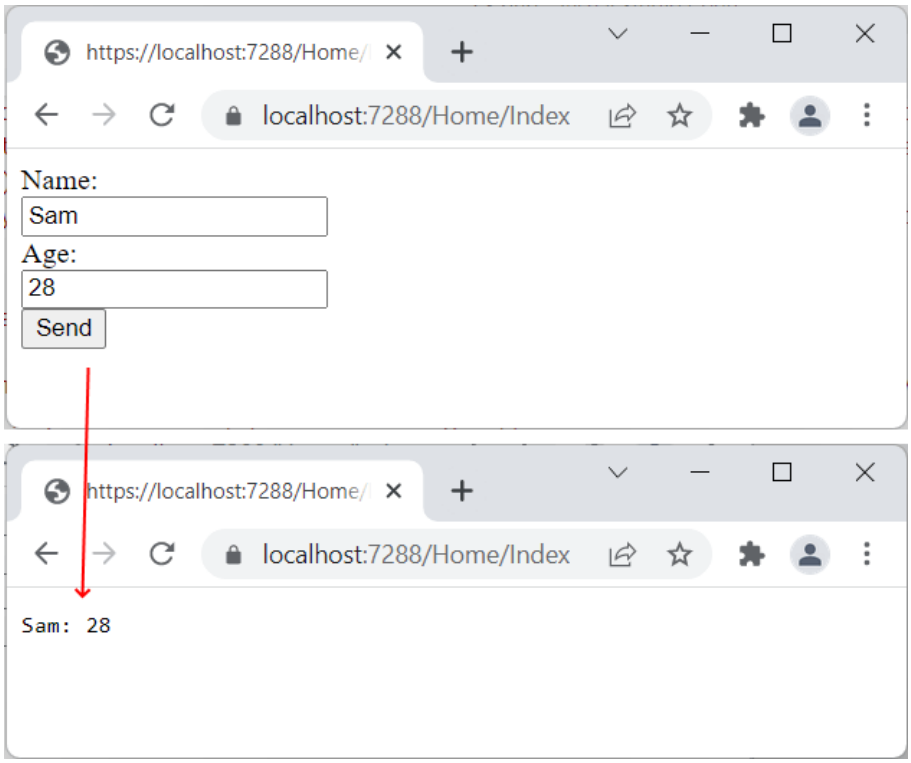


Рис. 3.6. Передача параметрів методу контролера через форму

Отримання складних об'єктів

При отриманні даних форм діють ті самі правила прив'язки, як і при отриманні параметрів рядка запиту. Тому з тієї ж форми можна отримати значення у вигляді складних об'єктів, у яких назви властивостей відповідають назвам полів форми:

```
using Microsoft.AspNetCore.Mvc;

namespace MvcApp.Controllers
{
 public class HomeController : Controller
 {
   public async Task Index()
        {
            string content = @"<form method='post'>
                <label>Name:</label><br />
                <input name='name' /><br />
                <label>Age:</label><br />
                <input type='number' name='age' /><br />
                <input type='submit' value='Send' />
            </form>";
            Response.ContentType = "text/html;charset=utf-8";
            await Response.WriteAsync(content);
        }
 [HttpPost]
 public string Index(Person person)=>$"{person.Name}: {person.Age}";
 }
 public record class Person(string Name, int Age);
}
```

В даному випадку поля форми name і age відповідні за назвою властивостям класу Person, тому замість поодиноких розрізнених значень отримуємо відправлену форму як **об'єкт Person**.

Отримання масивів

Для передачі масивів за допомогою форми треба створити набір однойменних полів, які називаються на ім'я масиву:

```
using Microsoft.AspNetCore.Mvc;

namespace MvcApp.Controllers
{
    public class HomeController : Controller
    {
        public async Task Index()
        {
            string form = @"<form method='post'>
                <p><input name='names' /></p>
                <p><input name='names' /></p>
                <p><input name='names' /></p>
                <input type='submit' value='Send' />
            </form>";
            Response.ContentType = "text/html;charset=utf-8";
            await Response.WriteAsync(form);
        }
        [HttpPost]
        public string Index(string[] names)
        {
            string result = "";
            foreach(string name in names)
            {
                result = $"{result} {name}";
            }
            return result;
        }
    }
}
```

В даному випадку на формі, що надсилається користувачу, розташовані три поля з ім'ям "names". У результаті при відправленні форми буде сформовано масив names із трьох елементів, який можна отримати у другому методі Index (рис. 3.7).

І також у елементів форми можна **явно вказати індекси**:

```
<form method='post'>
    <p><input name='names[0]' /></p>
    <p><input name='names[2]' /></p>
    <p><input name='names[1]' /></p>
    <input type='submit' value='Send' />
</form>
```

Також можна зовсім **обмежитися одними індексами**

```
<form method='post'>
    <p><input name='[0]' /></p>
    <p><input name='[2]' /></p>
    <p><input name='[1]' /></p>
    <input type='submit' value='Send' />
</form>
```

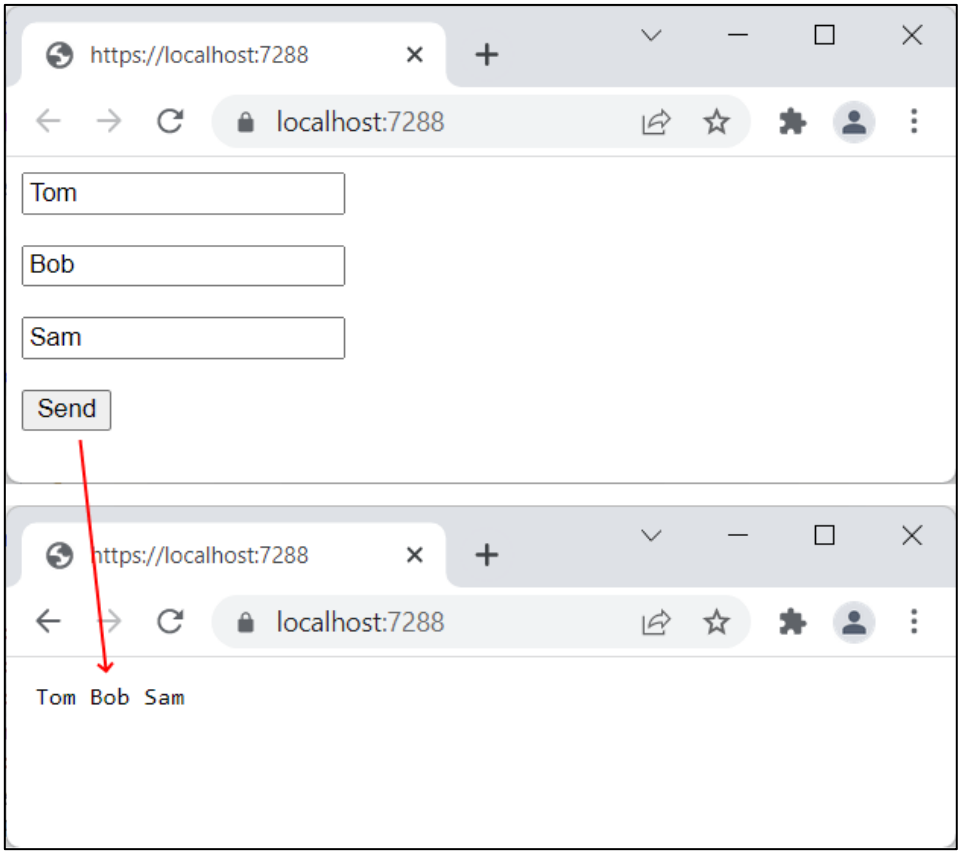


Рис. 3.7. Передача масивів за допомогою форми

3.6. Контрольні питання

1. Що таке контролер?
2. Які існують умовності при використанні контролерів?
3. Що таке дії контролера?
4. Як у додатоку підключається підтримка контролерів?
5. В якій папці проекту зберігаються контролери?
6. Яки типи запитів можуть обслуговувати методи контролерів?
7. Передача даних до контролера через рядок запиту

8. Передача даних до контролера через форми.
9. Обробка запитів типу GET
10. Обробка запитів типу POST.

4. Подання. Можливості движка подання Razor

Під час доступу до веб-додатку користувач очікує отримати веб-сторінку з деякими даними. MVC зазвичай використовує **подання**, які визначають **вигляд** додатку на основі яких потім формується веб-сторінка.

Подання у ASP.NET Core MVC - це **файли з розширенням .cshtml,** які містять код інтерфейсу користувача переважно в HTML, а також **конструкції Razor**, спеціального **двигуна подання**, що дозволяє переходити від HTML-коду до коду мови програмування C#.

Подання в проекті ASP.NET Core MVC зберегаються у папці **"Views"**.

Наприклад, проект типу **ASP.NET Core Web App (Model-View-Controller)** містить низку уявлень (рис. 4.1).

Подібний проект для зберігання подань у папці **«Views»** визначає певну структуру:

- як правило, **для кожного контролера** в проекті **створюється підкаталог** у папці Views, який називається на **ім'я контролера** і який зберігає подання, що використовуються методами даного контролера.

Так, за замовчуванням є контролер **HomeController** і для нього в папці Views є підкаталог **Home** з поданнями для методів контролера HomeController – в даному випадку це файли **Index.cshtml** та **Privacy.cshtml**.

- є папка **Shared**, яка зберігає **загальні уявлення для всіх контролерів**.

За замовчуванням це файли **_Layout.cshtml** (використовується як майстер-сторінка), **Error.cshtml** (використовуються для відображення помилок) та

_ValidationScripsPartial.cshtml (часткове подання, яке включає скрипти валідації форми).

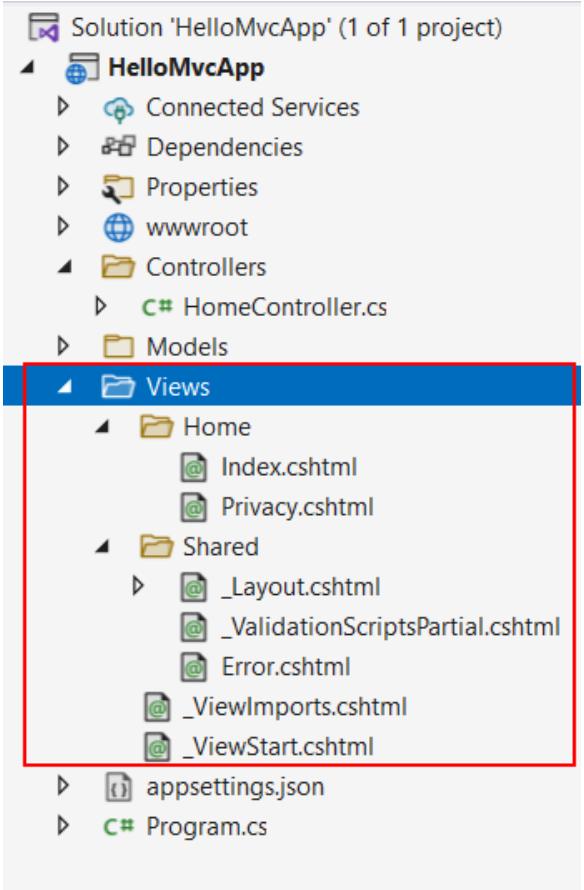


Рис. 4.1. Структура уявлення у проекті типу ASP.NET Core Web App (Model-View-Controller)

- у корені каталогу **Views** знаходяться два файли **_ViewImports.cshtml** та **_ViewStart.cshtml**. Ці файли містять код, який автоматично додається до всіх уявлень. **_ViewImports.cshtml** встановлює деякі спільні для всіх уявлень простори імен, а **_ViewStart.cshtml** встановлює спільну майстер-сторінку.

4.1. Об'єкт ViewResult

За роботу з поданнями відповідає об'єкт **ViewResult**. Він здійснює **рендеринг подання на веб-сторінку** і **повертає її у вигляді відповіді клієнту**. Щоб повернути об'єкт ViewResult, метод контролера викликає метод View:

```
public class HomeController : Controller
    {
        public IActionResult Index()
        {
           return View();
        }
    }
```

Виклик **методу View** повертає **об'єкт ViewResult**. Потім вже ViewResult здійснює рендеринг певного подання у відповідь. За замовчуванням контролер здійснює пошук подання у проекті за такими шляхами:

/Views/Ім'я_контролера/Ім'я_подання.cshtml/

Views/Shared/Ім'я_подання.cshtml

Метод View() має чотири перевантажені версії:

- **View():** для генерації відповіді використовується подання, яке на **ім'я** збігається з методом, що викликає.
- **View(string? viewName):** в метод передається **ім'я подання viewName**, що дозволяє перевизначити подання, що використовується за умовчанням.
- **View(object? model):** передає у подання **дані** у вигляді об'єкта model
- **View(string? viewName, object? model):** перевизначає **ім'я подання viewName** та передає у нього дані у вигляді об'єкта model.

Друга версія методу дозволяє перевизначити подання, що використовується. Якщо подання знаходиться в тій же папці, яка призначена для даного контролера, то до методу View() достатньо передати назву подання без розширення:

```
public class HomeController : Controller
  {
    public IActionResult Index()
     {
        return View("About");
     }
  }
```

У цьому випадку метод Index використовуватиме подання по шляху **Views/Home/About.cshtml**.

Якщо ж подання знаходиться в іншій папці, то треба передати повний шлях до подання:

```
public class HomeController : Controller
  {
   public IActionResult Index()
     {
        return View("~/Views/Some/About.cshtml");
     }
  }
```

В даному випадку передбачається, що подання розташовується по шляху **Views/Some/About.cshtml.**

4.2. Підключення функціоналу уявлень

Щоб програма могла використовувати подання, треба **підключити** відповідні **сервіси**.

Для підключення до додатку функціонала контролерів з поданнями в коді файлу **Program.cs** застосовується виклик методу **AddControllersWithViews()** (також можна застосовувати методи **AddMvcCore()** та **AddMvc()):**

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

                // додаємо підтримку контролерів з уявленнями
```

```
builder.Services.AddControllersWithViews();

var app = builder.Build();
          // встановлюємо зіставлення маршрутів із контролерами
app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");
app.Run();
```

Згідно з налаштуваннями за умовчанням, якщо назва подання не зазначена явно, то як подання буде використовуватися те, ім'я якого збігається з **ім'ям дії контролера**.

Наприклад, вищезазначена дія Index за замовчуванням буде шукати представлення ***Index.cshtml*** *в папці* **/Views/Home/.**

Приклад подання Index.cshtml:

```
<!DOCTYPE html>
<html>
<head>
    <title>ТРКД ДНУ. ASP.NET CORE MVC</title>
    <meta charset="utf-8" />
</head>
<body>
    <h2> Hello DNU!</h2>
</body>
</html>
```

Це подання нагадує **звичайну сторінку HTML**. Тут можна визначити всі стандартні елементи розмітки HTML, підключати стилі та скрипти.

Але повноцінною HTML-сторінкою подання все одно не є, тому що під час виконання ці подання компілюються у збірки і вже потім використовуються для генерації HTML-сторінок, які бачить користувач у своєму браузері.

І після запуску програми та звернення до **методу Index контролера Home** браузер відобразить веб-сторінку, яка буде згенерована на основі **подання Index.cshtml** (рис. 4.2).

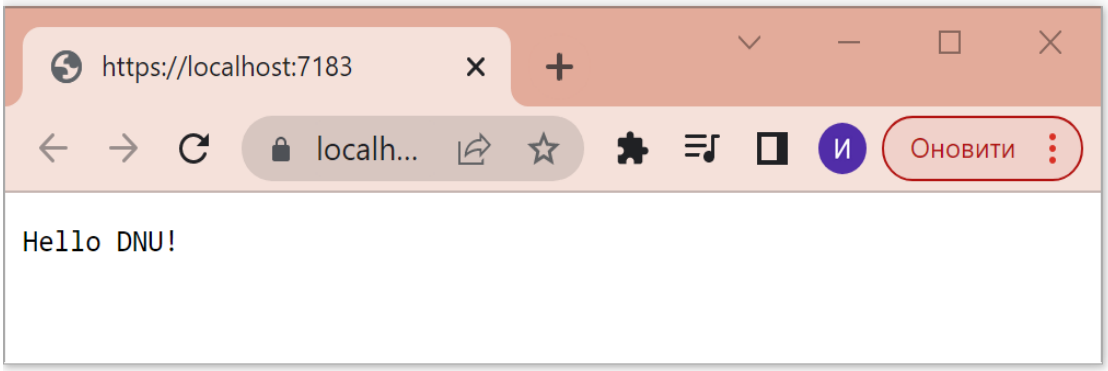


Рис. 4.2. Згенерована веб-сторінка

4.3. Двигун подання Razor

Подання в ASP.NET Core MVC може містити як стандартний код HTML, так й **вставки коду мовою програмування C#.** Для обробки коду, який містить елементи HTML, так і конструкції мови C#, застосовується **двигун подання**.

Насправді при виклику методу View контролер не робить рендеринг подання і не генерує розмітку HTML. Контролер лише готує дані та вибирає, яке подання треба повернути як об'єкт ViewResult. Потім вже об'єкт ViewResult звертається до движка подання для рендерингу подання у вихідну відповідь.

За замовчуванням в ASP.NET Core MVC застосовується один **двигун подання** - **Razor**.

Хоча за бажання можна також використовувати якісь інші сторонні движки або створити свій движок подання самостійно.

Мета движка подання Razor - визначити перехід від розмітки HTML до коду C#.

Синтаксис Razor простий - всі його конструкції передуються символом @ після якого відбувається перехід до коду C#.

Наприклад, визначимо таке подання:

```
<!DOCTYPE html>
```

```
<html>
<head>
<title> ТРКД ДНУ. ASP.NET CORE MVC </title>
<meta charset="utf-8" />
</head>
<body>
<h2>Time: @DateTime.Now.ToShortTimeString()</h2>
</body>
</html>
```

Тут замість виразу @DateTime.Now.ToShortTimeString() при рендерингу подання вставлятиметься поточний час (рис. 4.3):

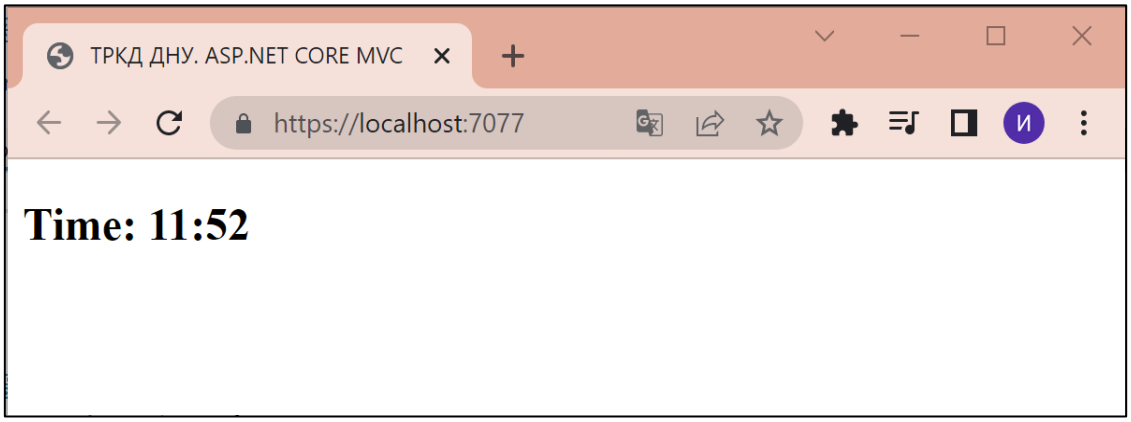


Рис. 4.3. Використання движка Razor для заголовку з поточним часом

4.4. Типи конструкцій Razor

Всі конструкції Razor можна умовно розділити на два види:

- **однорядкові вирази** та
- **блоки коду**.

Приклад застосування **однорядкових виразів**:

```
<p>Date: @DateTime.Now.ToLongDateString()</p>
```

В даному випадку використовується об'єкт DateTime та його метод ToLongDateString().

Ще один приклад:

```
<p>@(10 + 20)</p>
```

Оскільки перед дужками стоїть знак @, то вираз у дужках інтерпретуватиметься як вираз мовою C#. Тому браузер виведе число 30, а не "10 + 20".

Якщо створюємо код HTML, в якому символ @ присутній сам по собі, а не як частина синтаксису Razor, то щоб його відобразити, треба його дублювати:

```
<p>@@DateTime.Now =@DateTime.Now.ToLongDateString()</p>
```

Блоки коду можуть мати кілька виразів. Блок коду міститься у **фігурні дужки**, а кожен вираз завершується **точкою з комою** аналогічно блокам коду та виразам на C#:

```
@{
string head = "Hello DNU !";              // Визначаємо змінну head
string text = "ASP.NET Core Application"; // Визначаємо змінну text
}
<!DOCTYPE html>
<html>
 <head>
  <title> ТРКД ДНУ. ASP.NET CORE MVC </title>
  <meta charset = "utf-8" />
 </head>
 <body>
  <h2>@head</h2>   <!-- використовуємо змінну head -->
  <div>@text</div> <!-- використовуємо змінну text -->
 </body>
</html>
```

У блоках коду можна визначити звичайні змінні і потім використовувати їх у поданні (рис. 4.4).

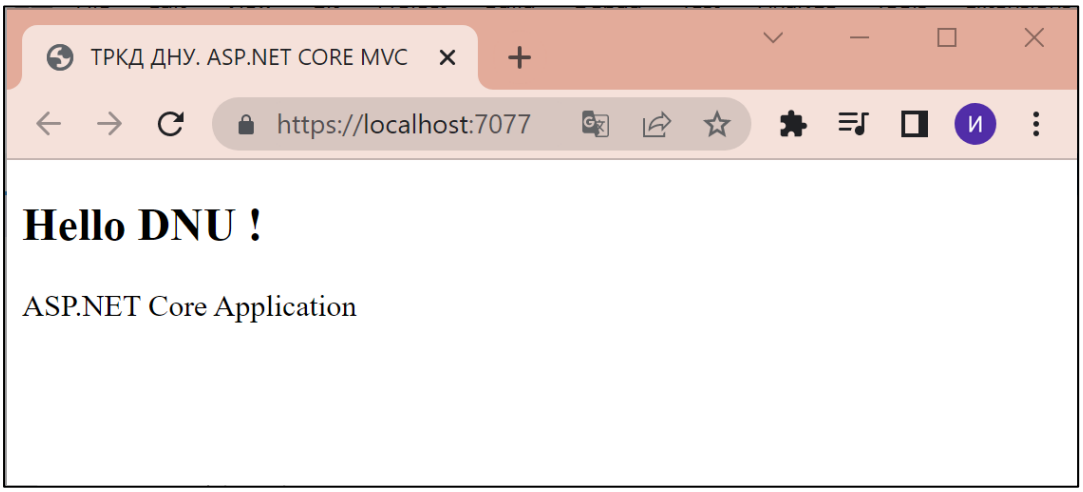


Рис. 4.4. Використання у поданні змінних

Якщо необхідно вивести **значення змінної** без будь-яких HTML-елементів, то використовують спеціальний **сніпет <text>:**

```
@{
  int i = 8;
  <text>@i</text>
}
<text>@(i+1)</text>
```

У Razor можуть використовуватись **коментарі**. Вони розташовуються між символами @* та *@:

@* текст коментаря *@

Умовні конструкції

Можна використовувати умовні конструкції if-else:

```
@{
   string morning = "Good Morning";
   string evening = "Good Evening";
   string hello = "Hello";
   int hour = DateTime.Now.Hour;
}
@if (hour < 12)
 {
   <h2>@morning</h2>
 }
else if (hour > 17)
 {
   <h2>@evening</h2>
 }
else
 {
   <h2>@hello</h2>
 }
```

Конструкція switch:

```
@{
   string language = "u kraine ";
}
@switch(language)
 {
   case "u kraine ":
     <h3> Привіт світ!</h3>
   break;
   case "german":
     <h3>Hallo Welt!</h3>
   break;
   default:
     <h3>Hello World!</h3>
   break;
 }
```

Цикли

Використовуються усі можливі цикли (рис. 4.5).

Цикл **for**:

```
@{
   string[] people = {"Taras", "Mykola", "Bogdan"};
}
<ul>
 @for (var i = 0; i < people.Length; i++)
```

```
  {
     <li>@people[i]</li>
  }
</ul>
```

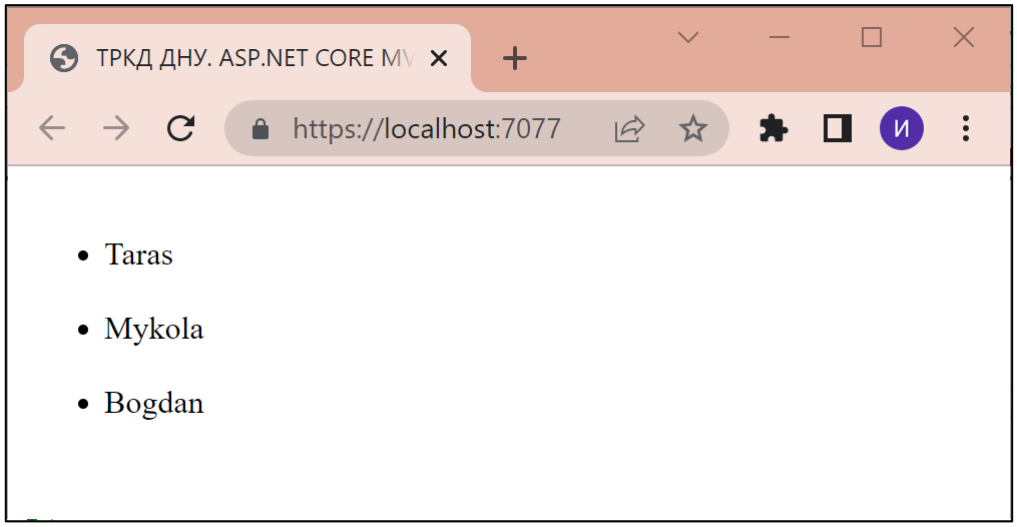


Рис. 4.5. Використання циклів у поданні

Цикл **foreach**:

```
@{
   string[] people = {"Taras", "Mykola", "Bogdan"};
}
<ul>
   @foreach (var person in people)
    {
      <li>@person</li>
   }
</ul>
```

Цикл **while**:

```
@{
   string[] people = {"Taras", "Mykola", "Bogdan"};

   var i = 0;
}
<ul>
 @while (i<people.Length)
   {
```

```
      <li>@people[i++]</li>
   }
</ul>
```

Цикл **do .. while**:

```
@{
   var i = 0;
}
<ul>
  @do
   {
     <li>@(i*i)</li>
   }
  while (i++<5);
</ul>
```

Конструкція try...catch

Конструкція try ... catch ... finally, як і в C#, дозволяє **обробити виняток**, який може виникнути під час виконання коду:

```
@try
 {
     throw new InvalidOperationException("Something wrong");
 }
catch (Exception ex)
 {
     <p>Exception: @ex.Message</p>
 }
finally
 {
     <p>finally</p>
 }
```

Якщо в блоці try викидається виняток, виконується блок catch. І в будь-якому випадку в кінці блоку try та catch виконується блок finaly.

Виведення тексту у блоці коду

Звичайний текст у блоці коду можливо вивести:

```
@{
   bool isEnabled = true;
}
@if (isEnabled)
{
   Hello DNU
}
```

У цьому випадку двигун Razor розглядатиме рядок "Hello DNU" як набір операторів мови C#, яких, звичайно, у C# немає, тому отримаємо помилку.

Щоб вивести текст як є в блоці коду, треба використовувати вираз **@:**:

```
@{
     bool isEnabled = true;
}
@if (isEnabled)
{
     @: Hello DNU
}
```

Функції

Директива **@functions** дозволяє визначити функції, які можуть бути використані в поданні (рис. 4.6).

Наприклад:

```
@functions
{
 public int Sum(int a, int b)
  {
     return a + b;
  }
 public int Square(int n) => n * n;
}
<p>Sum of 5 and 4:<b>@Sum(5,4)</b></p>
<p>Square of 4:<b>@Square(4)</b></p>
```

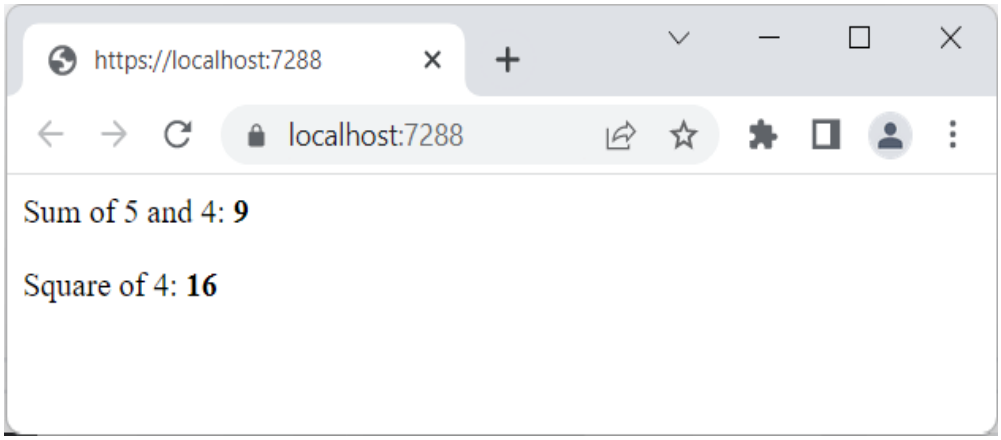


Рис. 4.6. Використання функцій у поданні

4.5. Передача даних у подання

Існують різні способи передачі даних з контролера у подання:

- об'єкт **ViewData,**
- об'єкт **ViewBag,**
- **модель подання.**

Об'єкт ViewData

Об'єкт **ViewData** представляє словник з пар ключ-значення:

```
public IActionResult Index()
  {
     ViewData["Message"] = "Hello DNU!";
     return View();
  }
```

Тут динамічно визначається об'єкт ViewData з ключем "Message" і значенням "Hello DNU!". При цьому як значення може бути будь-який об'єкт. Після визначення об'єкту можна його використовувати в поданні:

```
@{
   ViewData["Title"] = "ASP.NET Core MVC";
```

```
}
```

```
<h2>@ViewData["Title"]</h2>
<h3>@ViewData["Message"]</h3>
```

Не обов'язково встановлювати всі об'єкти в ViewData в контролері. Так, у даному випадку об'єкт із ключем "Title" встановлюється безпосередньо у поданні. У результаті в браузері побачимо значення обох елементів із ViewData (рис. 4.7).

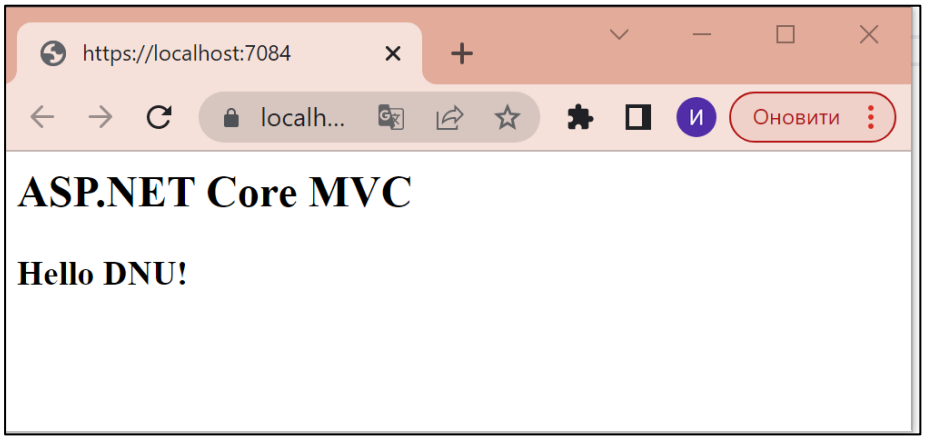


Рис. 4.7. Використання об'єктів ViewData у поданні

Об'єкт ViewBag

Об'єкт ViewBag багато в чому подібний до об'єкту ViewData. Він дозволяє визначити різні властивості та присвоїти їм будь-яке значення.

Перепишемо попередній приклад так:

```
public IActionResult Index()
{
      ViewBag.Message = "Hello DNU!";

      return View();
}
```

І так само змінемо подання:

```
@{
      ViewBag.Title = "ASP.NET Core MVC: ViewBag";
}
<h2>@ViewBag.Title</h2>
<h3>@ViewBag.Message</h3>
```

Властивості об'єкту ViewBag **визначаються динамічно**. Вони можуть містити як прості об'єкти типу string чи int, а й складні дані.

Наприклад, передамо до властивості об'єкту ViewBag список:

```
public IActionResult Index()
{
  ViewBag.People = new List<string> {"Taras","Mykola","Bogdan"};
  return View();
}
```

У поданні отримаємо цей список (рис. 4.8):

```
<h2>Count: @ViewBag.People.Count</h2>
<ul>
 @foreach(string person in ViewBag.People)
 {
      <li>@person</li>
 }
</ul>
```

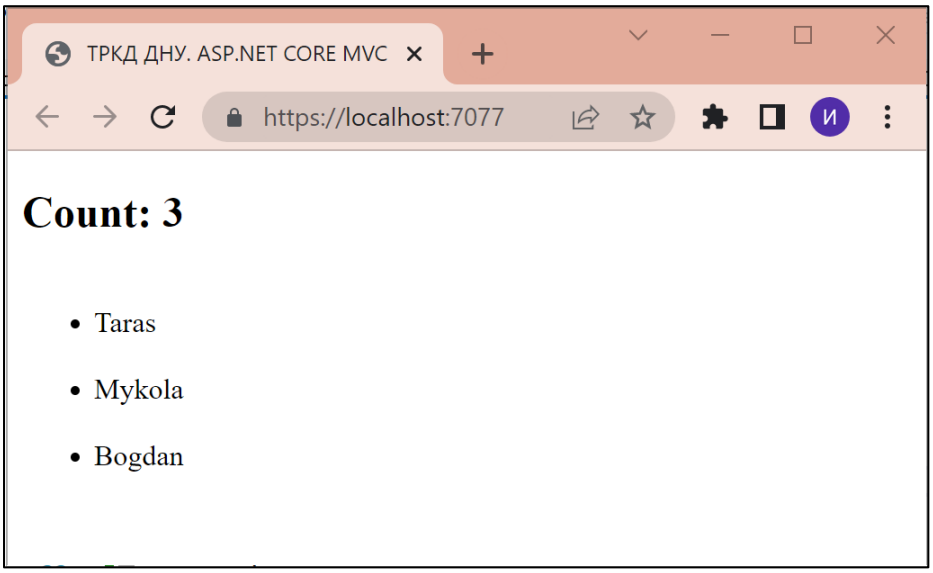


Рис. 4.8. Використання об'єктів ViewData у поданні

Модель подання

Модель подання є у багатьох випадках кращим способом передачі даних у подання. Для передачі даних у подання використовується одна з версій методу View():

```
public IActionResult Index()
{
  var people = new List<string> {"Taras", "Mykola", "Bogdan"};
  return View(people);
}
```

У метод View() передається список, тому моделлю подання Index.cshtml буде тип List<string> (або IEnumerable<string>). І тепер у поданні можна написати так:

```
@model List<string>

<h2>Count: @Model.Count</h2>
<ul>
@foreach(string person in Model)
 {
     <li>@person</li>
 }
</ul>
```

На самому початку подання за допомогою директиви **@model** встановлюється **модель подання. Тип моделі повинен збігатися з типом об'єкту**, який передається у метод View() в контролері.

Встановлення моделі вказує, що об'єкт **Model** тепер представлятиме об'єкт List<string> або список. І можна використовувати **Model** як список (рис. 4.9).

Подання, для яких визначено модель, ще називають **строго типізованими** або **strongly-typed views**.

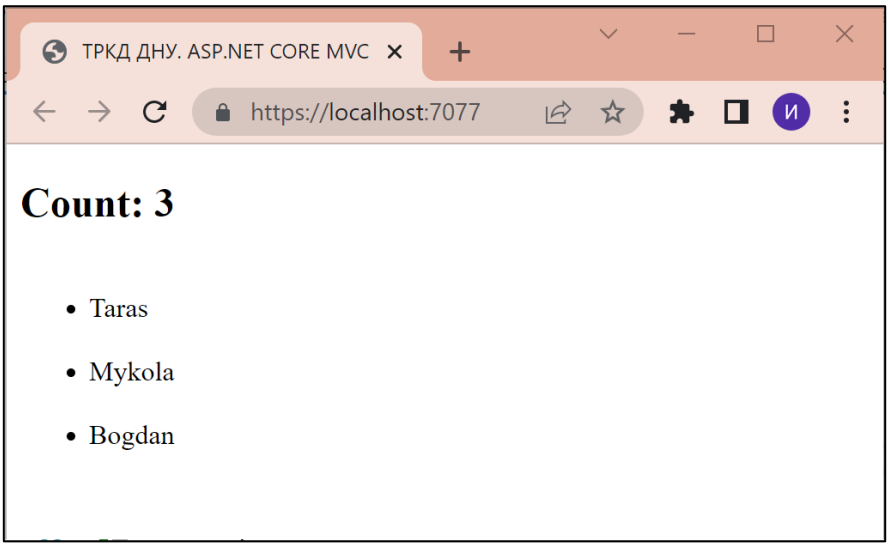


Рис. 4.9. Використання моделі подання для передачі даних у подання

4.6. Контрольні питання

1. Що таке подання?
2. Де в проекті розміщуються подання?
3. Як у додатку підключається підтримка подання?
4. Що робить спосіб View()? Які є перевантажені версії цього методу?
5. Що таке двигун уявлень Razor?
6. Можливості двигуна уявлень Razor.
7. Способи передачі даних із контролера у подання.

5. Система маршрутизації

Система маршрутизації зв'язує контролери MVC та їх дії з запитами. Можна використовувати різні методи, щоб включити маршрутизацію для контролерів MVC.

Наприклад, визначимо в файлі Program.cs наступний код:

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

          // Додавання підтримки контролерів з поданнями
builder.Services.AddControllersWithViews();

var app = builder.Build();

          // встановлюємо зіставлення маршрутів із контролерами

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

app.Run();
```

Тут для додавання маршруту, який зіставлятиметься з методами контролерів, у об'єкту IEndpointRouteBuilder (в якості якого виступає об'єкт WebApplication) викликається **метод MapControllerRoute().** Він визначає єдиний маршрут:

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

Параметр **name** визначає назву маршруту - вона довільна і у разі має значення "default". А параметр **pattern** визначає **шаблон маршруту**, з яким **зіставлятимуться вхідні маршрути**.

Шаблон маршруту використовує **три параметри**. Параметр "**controller**" буде зіставлятися на ім'я з одним із **контролерів** додатка, а параметр "**action**" - з **методом дії** цього контролера.

Наприклад, при запиті http://localhost:3456/Home/Index система вибере для обробки запиту контролер Home - ім'я контролера без префікса Controller та його дію Index.

Даний тип маршрутизації ще називають **Convention-Based Routing**, тобто маршрутизація, що базується на **умовностях у визначеннях маршрутів**.

5.1. Додавання маршрутів

Для додавання маршрутів до MVC застосовуються такі методи інтерфейсу **IEndpointRouteBuilder**:

• **MapControllerRoute()** визначає **довільний маршрут**:

MapControllerRoute(string name, string pattern, [object defaults = null], [object constraints = null], [object dataTokens = null])

Цей метод приймає параметри:

- name: назва машруту,
- pattern: шаблон машруту,
- defaults: значення параметрів маршрутів за замовчуванням,
- constraints: обмеження маршруту,
- dataTokens: визначення токенів маршруту.

Перші два параметри обов'язкові, інші необов'язкові.

- **MapDefaultControllerRoute()** визначає **стандартний маршрут**, фактично еквівалентний виклику:

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

Оскільки це визначення маршрута, що досить часто використовується, то для нього і був введений окремий метод.

- **MapAreaControllerRoute()** визначає маршрут, який також враховує область програми. Має такі параметри:

MapAreaControllerRoute(string name, string areaName, string pattern, [object defaults = null], [object constraints = null], [object dataTokens = null])

Обов'язковий параметр **areaName** дозволяє визначити область, з якою буде пов'язаний маршрут.

- **MapControllers()** зіставляє дії контролера із запитами, використовуючи маршрутизацію з урахуванням атрибутів.
- **MapFallbackToController()** визначає дію контролера, яка оброблятиме запит, якщо всі інші певні маршрути не відповідають запиту. Приймає ім'я контролера та його методу:

MapFallbackToController(string action, string controller)

5.2. Отримання параметрів маршрутів у контролері

У контролері можна отримати всі параметри маршруту, використовуючи об'єкт RouteData:

```
public class HomeController : Controller
{
    public string Index()
    {
        var controller = RouteData.Values["controller"];
        var action = RouteData.Values["action"];
        return $"controller: {controller}|action: {action}";
    }
}
```

5.3. Визначення маршрутів

Основою визначення маршруту служить його шаблон. На основі шаблону маршруту система маршрутизації визначає, чи оброблятиметься запит, а якщо буде, то яким контролером і яким методом контролера.

Наприклад, у файлі Program.cs задамо один маршрут:

```
var builder = WebApplication.CreateBuilder(args);
          // додаємо підтримку контролерів з уявленнями
builder.Services.AddControllersWithViews();
```

```
var app = builder.Build();
          // встановлюємо зіставлення маршрутів із контролерами

app.MapControllerRoute(
    name: "default",
    pattern: "{controller}/{action}");

app.Run();
```

Тут шаблон маршруту виглядає так:

```
"{controller}/{action}"
```

Шаблон маршруту може мати параметри. Параметри мають ім'я та визначаються у шаблоні маршруту всередині фігурних дужок:

{назва_параметра}

Наприклад, у шаблоні маршруту вище визначено два параметри: controller та action.

Коли приходить запит до додатку, система маршрутизації зіставляє запитаний шлях, якщо зіставлення пройшло успішно, то система маршрутизації використовує параметр "controller" для отримання імені контролера, а параметр "action" для визначення методу дії цього контролера, який оброблятиме запит.

Якщо в шаблоні маршруту йдуть кілька параметрів, то між ними повинен розташовуватися роздільник. Зазвичай як роздільник виступає **слєш** (/), який оформляє окремий сегмент на шляху запиту.

Тобто у прикладі показаному вище кожен окремий параметр маршруту відокремлений від іншого слешем, тому представлятиме окремий сегмент шляху запиту.

Наприклад, якщо прийшов запит на шляху "/home/index/", то перший сегмент "/home/" проектуватиметься на параметр "controller", тому буде обраний контролер Home. Другий сегмент - "/index/" буде проектуватись на параметр

"action", відповідно для обробки цього запиту буде обраний метод Index контролера Home.

Нехай визначено контролер HomeController з наступним кодом:

```
using Microsoft.AspNetCore.Mvc;

namespace MvcApp.Controllers
{
    public class HomeController : Controller
    {
        public string Index() => "Index Page";
        public string About() => "About Page";
    }
}
```

Таким чином, при запиті Home/Index запит буде оброблятися методом Index контролера Home, а запит Home/About - методом About того ж контролера (рис. 5.1).

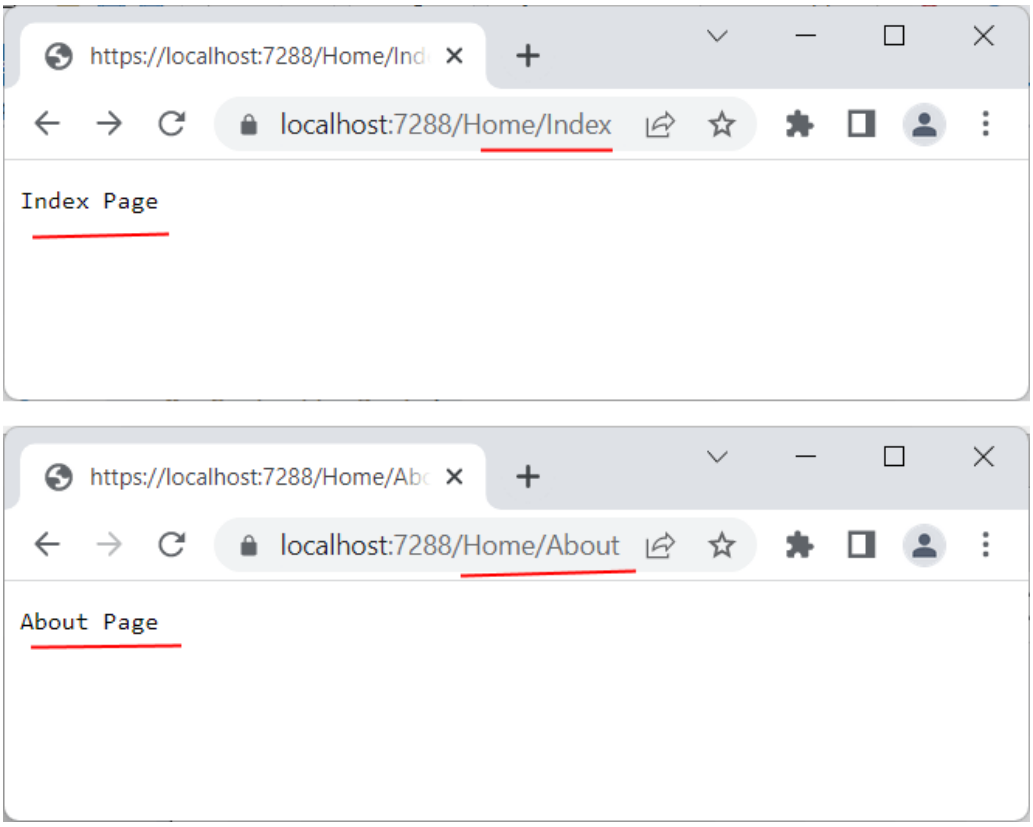


Рис. 5.1. Визначення маршруту

Шаблон маршруту може містити додаткові параметри, які можна отримати за допомогою параметрів методу.

Наприклад, визначимо такі маршрути:

```
var builder = WebApplication.CreateBuilder(args);
builder.Services.AddControllersWithViews();
 var app = builder.Build();
          // встановлюємо зіставлення маршрутів із контролерами

app.MapControllerRoute(
   name: "default",
   pattern: "{controller}/{action}/{id}");

app.MapControllerRoute(
   name: "name_age",
   pattern: "{controller}/{action}/{name}/{age}");

app.Run();
```

Тут перший маршрут – "default" приймає третій параметр – id, який розташовується у третьому сегменті рядка запиту.

Другий маршрут - "name_age" приймає додатково два параметри - name і age, що розташовуються відповідно у третьому та четвертому сегментах рядка запиту.

Для тестування маршрутів визначимо наступний контролер:

```
using Microsoft.AspNetCore.Mvc;

namespace MvcApp.Controllers
{
    public class HomeController : Controller
    {
        public string Index(int id) => $"Index Page. Id: {id}";
        public string About(string name, int age) => $"About Page.
                                  Name: {name}  Age: {age}";
    }
}
```

Якщо до додатку прийде запит типу "Home/Index/6", система маршрутизації зможе зіставити цей запит з першим маршрутом - маршрутом "default", який також містить три сегменти, і значення останнього сегмента - число 6 буде передано в метод Index через параметр id (рис. 5.2):

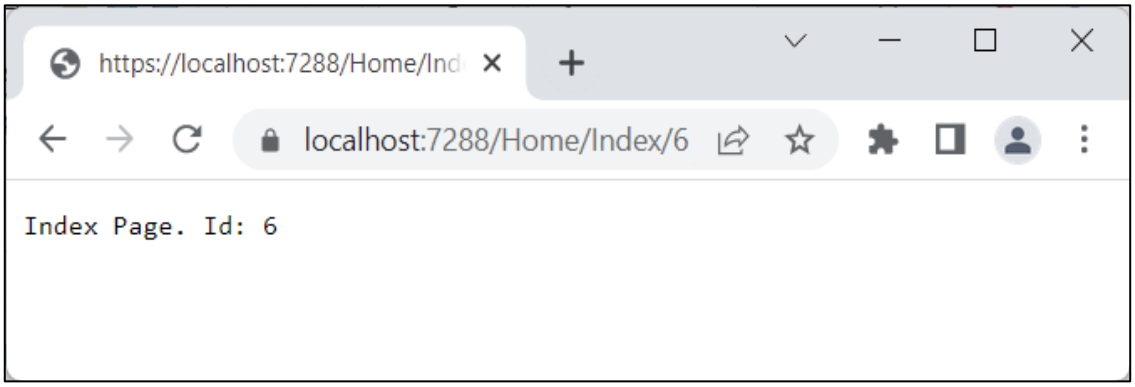


Рис. 5.2. Визначення маршруту з трьома сегментами

Якщо ж до додатку прийде запит із чотирьох сегментів типу "Home/About/Tom/37", то система маршрутизації зможе зіставити цей запит з другим маршрутом - маршрутом "name_age". Тоді значення третього сегмента буде передано у метод через параметр name, а значення четвертого сегмента - через параметр age (рис. 5.3):

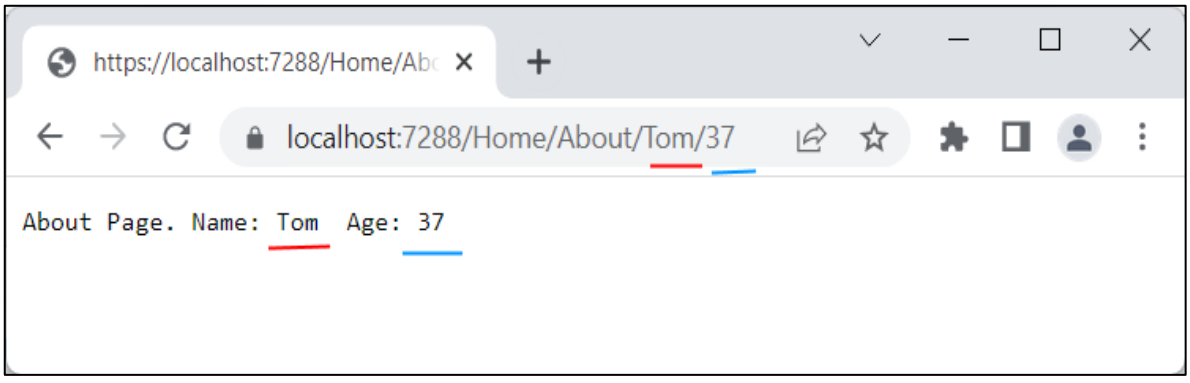


Рис. 5.3. Визначення маршруту з 4 сегментами

5.4. Статичні сегменти

Шаблони маршрутів можуть мати **статичні сегменти**, які пов'язуються з параметрами маршрутів.

Наприклад, визначимо наступний маршрут:

```
app.MapControllerRoute(
    name: "default",
    pattern: "api/{controller}/{action}/{id}");
```

В даному прикладі шаблон маршруту починається із статичного сегмента api. Таким чином, цьому маршруту будуть відповідати всі маршрути, що складаються з чотирьох сегментів, де перший сегмент дорівнює "api", наприклад, запит https://localhost:7288/api/Home/Index/6

Необов'язкові параметри маршрутів

Параметри маршруту можуть бути необов'язковими. Щоб визначити параметр як необов'язковий, після його назви вказується **знак питання** (**?**).

Наприклад, визначимо наступний маршрут:

```
app.MapControllerRoute(
    name: "default",
    pattern: "{controller}/{action}/{id?}");
```

У цьому шаблоні маршруту третій сегмент представляє параметр id, який позначений як необов'язковий. А це означає, що у запиті можна ігнорувати значення цього сегмента.

Наприклад, даний шаблон буде відповідати дво- та трисегментним запитам двом наступним URL:

/home/index

/home/index/23

Для тестування визначимо наступний контролер:

```
using Microsoft.AspNetCore.Mvc;

namespace MvcApp.Controllers
{
    public class HomeController : Controller
    {
        public string Index(int? id)
        {
            if(id is not null) return $"Product Id: {id}";

            return "Product List";
        }
    }
}
```

Якщо параметр id визначений, то метод Index повертає один рядок, якщо невизначений – інший (рис. 5.4):

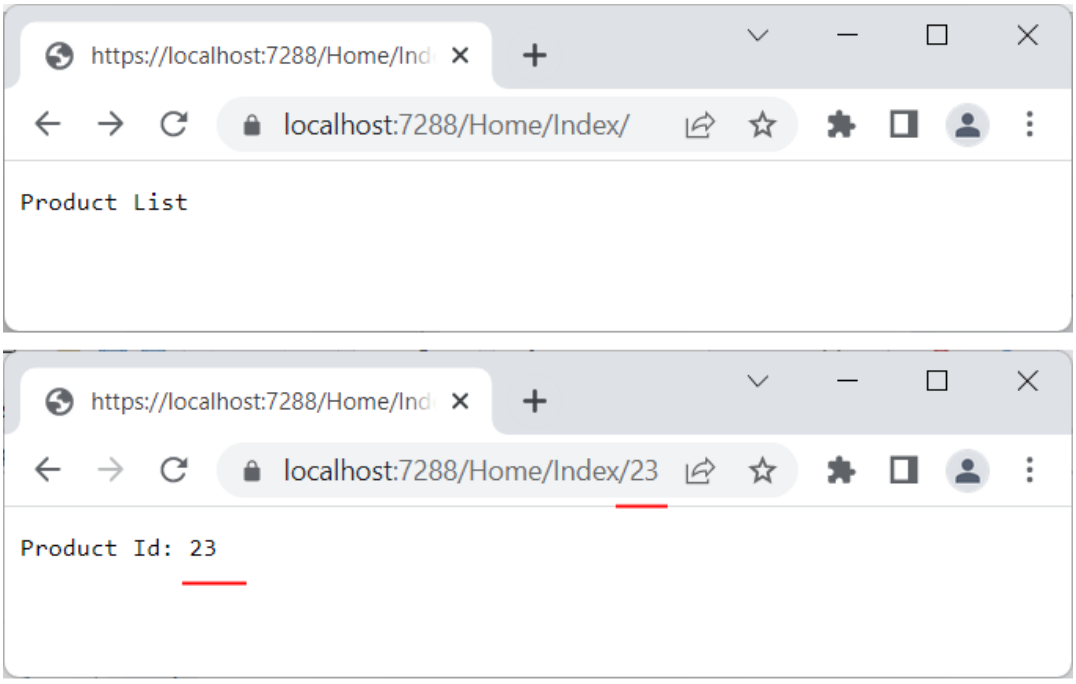


Рис. 5.4. Визначення маршруту з необов'язковим параметром

Необов'язкові параметри слід поміщати наприкінці шаблону маршруту, як у випадку з id у прикладі вище.

Тобто, наприклад, шаблон "{controller}/{action}/{id?}/{name}" не працюватиме коректно, якщо для параметра id не буде передано значення. А ось шаблон "{controller}/{action}/{name}/{id?}" працюватиме нормально.

5.5. Значення параметрів за замовчуванням

Параметрам маршруту можна призначити значення за промовчанням на випадок, якщо їм не передані значення.

Приклад.

```
var builder = WebApplication.CreateBuilder(args);
 builder.Services.AddControllersWithViews();
 var app = builder.Build();
          // встановлюємо зіставлення маршрутів із контролерами

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

app.Run();
```

Тут визначено шаблон маршруту, що складається із трьох параметрів. Параметр "controller" має значення за замовчанням "Home". Параметр "action" має значення за замовчанням "Index". Параметр "id" визначено як необов'язковий. У результаті за різних запитів отримаємо такі значения:

Запит	Параметри запиту
https://localhost:7084/	controller=Home action=Index
https://localhost:7084/Computer	controller=Computer action=Index

https://localhost:7084/ Computer /Show	controller=Computer action=Show
https://localhost:7084/ Computer /Show/5	controller=Computer action=Show id=5

Для встановлення значень за замовчуванням також можна застосовувати параметри defaults методу MapControllerRoute(). Цей параметр є об'єктом, властивості якого відповідають параметрам маршруту.

Наприклад, визначимо наступний маршрут:

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

app.MapControllerRoute(
    name: "default",
    pattern: "{action}",
    defaults: new {controller="Home", action="Index"});

app.Run();
```

Тут шаблон маршруту складається з одного сегмента, який відповідає параметру "action", тобто представляє дію контролера. А параметр defaults:

```
defaults: new {controller="Home", action="Index"}
```

встановлює, що як контролер за замовчуванням буде використовуватися контролер Home, а як дія - метод Index.

Наприклад, нехай є наступний HomeController:

```
public class HomeController : Controller
{
    public string Index() => "Index Page";
    public string About() => "About Page";
```

}

Запит типу https://localhost:7288/ буде оброблятися методом Index контролера Home, а запит https://localhost:7288/About - методом About (рис. 5.5).

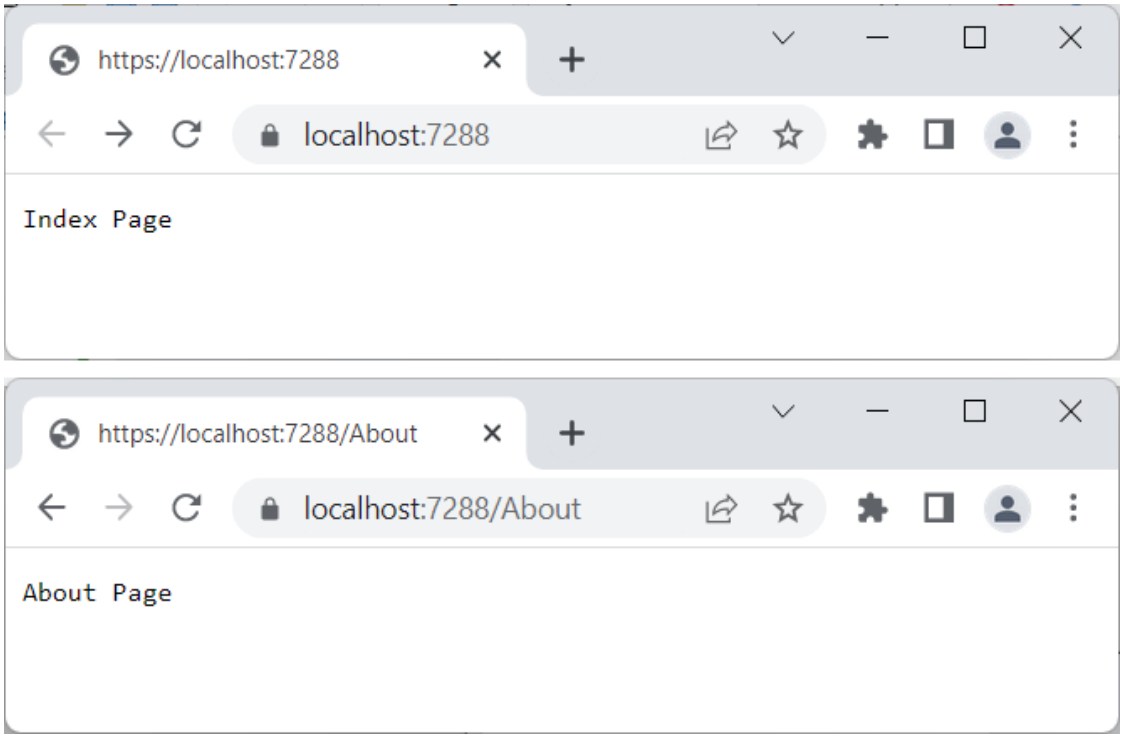


Рис. 5.5. Визначення значення параметрів маршруту за замовчанням

При використанні значень за замовчуванням можна не використовувати у шаблоні параметри маршруту.

Наприклад:

```
app.MapControllerRoute(
    name: "info",
    pattern: "contact/info",
    defaults: new {controller = "Home", action = "About"});
```

Запит https://localhost:7288/contact/info буде оброблятися методом About контролера Home.

5.6. Контрольні питання

1. Що таке маршрут?
2. Як додавати маршрут до веб-додатку?
3. Які методи додають маршрути до MVC?
4. Як визначається маршрут?
5. Що задає шаблон маршруту?
6. Які є параметри маршруту?
7. Як працює маршрут з значеннями параметрів за замовчуванням?

6. Визначення та застосування моделі

Однією з ключових складових шаблону MVC є **моделі**. Ключовим завданням моделей є опис структури і логіки використовуваних даних.

Зазвичай всі сутності, що використовуються в додатку, відокремлюються в окремі моделі, які описують структуру кожної сутності. Залежно від поставлених завдань і предметної області можна виділити різну кількість моделей в додатку.

Всі **моделі** форматуються як звичайні POCO-класи (plain-old CRL objects), тобто звичайні **класи в C#.**

Наприклад, якщо працюємо з даними користувача, то можна визначити наступну модель, яка представляє користувача:

```
public class Person
{
    public int Id { get; set; }
    public string Name { get; set; } = "";
    public int Age { get; set; }
}
```

Модель Person визначає ряд властивостей: унікальний ідентифікатор, ім'я користувача та вік.

Це класична **анемічна модель. Анемічна модель** не має поведінки і **зберігає** тільки стан **в якості властивостей**.

У C# для представлення таких моделей зручно використовувати **класи record**:

```
public record class Person(int Id, string Name, int Age);
```

Однак модель не обов'язково повинна складатися тільки з властивостей. Крім того, вона може мати **конструктор**, деякі **методи, поля**, в цілому, представлять стандартний клас на мові C#.

Моделі, які також **визначають поведінку**, на відміну від анемічних моделей, називаються «**жирними» моделями** (Rich Domain Model / Fat Model / Thick Model).

Наприклад, можемо до анемічної моделі додати деяку поведінку:

```
public class Person
{
    public int Id { get; set; }
    public string Name { get; set; }
    public int Age { get; set; }
    public Person(int id, string name, int age)
      {
           Id = id;
           Name = name;
           Age = age;
      }
    public string PrintInfo() => $"{Id}. {Name} ({Age})";
}
```

У додатку ASP.NET MVC Core моделі можна розділити за ступенем застосування на кілька груп:

• моделі, об'єкти яких зберігаються у спеціальних сховищах даних (наприклад, у **базах даних**, файлах xml тощо);

• **моделі подання**, які використовуються для передачі даних до подання або навпаки, для отримання даних із подання;

• **допоміжні моделі** для проміжних обчислень.

Як правило, для зберігання моделей у проекті створюється окрема папка **Models**. Моделі подання часто поміщаються в окрему папку, яка часто називається **ViewModels**.

Насправді це можуть бути каталоги з будь-якими назвами, можна поміщати моделі хоч у корінь проекту, але найпоширенішим стилем є назви Models і ViewModels.

6.1. Подключення Entity Framework Core

Entity Framework Core - це сучасний засіб відображення об’єктів і баз даних, що дозволяє автоматично **прив'язувати загальні класи C#** до **таблиць в базі даних**. Entity Framework Core працює з багатьма базами даних, включаючи базу даних SQL (локальна та Azure), SQLite, MySQL, PostgreSQL і Azure Cosmos DB.

Розглянемо роботу з базою даних MS SQL Server.

Для взаємодії з MS SQL Server через Entity Framework потрібен пакет **Microsoft.EntityFrameworkCore.SqlServer.** За замовчуванням він не присутній в проекті, тому його необхідно додати, наприклад, через **менеджер пакетів Nuget**:

Project -> Manage Nuget Packages … -> Browse (рис. 6.1).

Нехай маємо клас User:

```
public class User
{
    public int Id { get; set; }
    public string? Name { get; set; }      // Ім'я користувача
    public int Age { get; set; }           // вік користувача
```

}

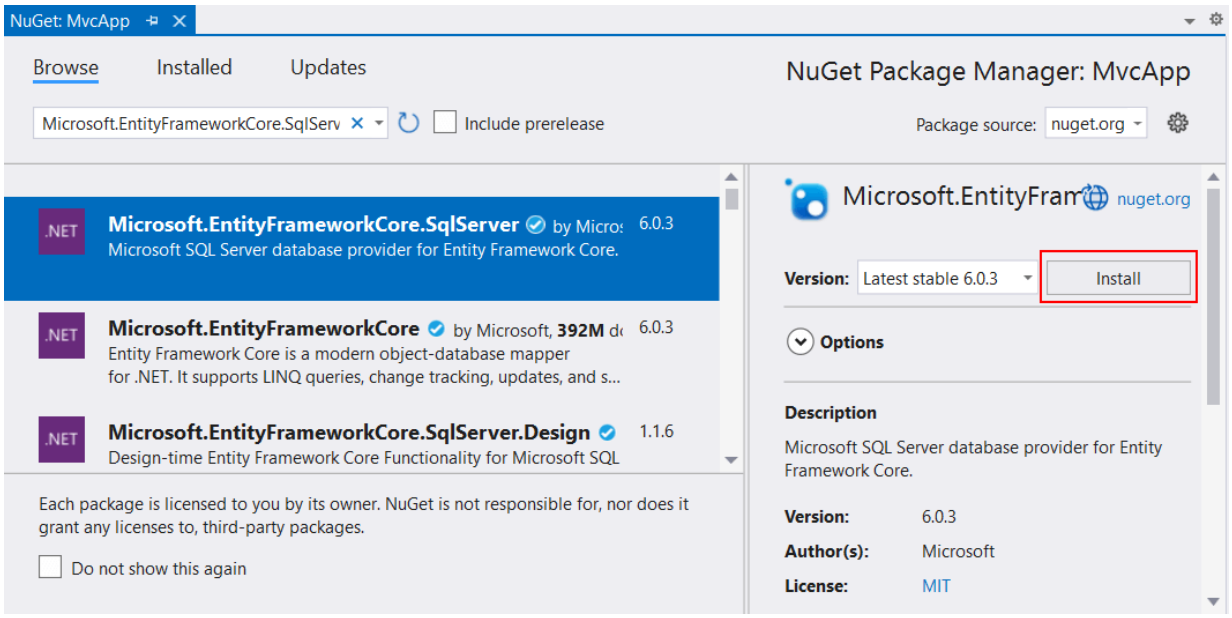


Рис. 6.1. Додавання пакету Microsoft.EntityFrameworkCore.SqlServer

Ця модель представляє об'єкти, які зберігатимуться у базі даних.

Щоб взаємодіяти з базою даних через Entity Framework, потрібен **контекст даних** - клас, успадкований від класу Microsoft.EntityFrameworkCore.**DbContext**.

Нехай це буде клас ApplicationContext:

```
using Microsoft.EntityFrameworkCore;
namespace MvcApp.Models
{
    public class ApplicationContext : DbContext
    {
      public DbSet<User> Users { get; set; } = null!;
      publicApplicationContext(DbContextOptions<ApplicationContext>
options) : base(options)
        {            // створюємо базу даних при першому зверненні
              Database.EnsureCreated();
        }
    }
}
```

Властивість DbSet є колекцією об'єктів, яка зіставляється з певною таблицею в базі даних.

За промовчанням назва властивості повинна відповідати **множині назві моделі** відповідно до правил англійської мови.

Тобто User - назва класу моделі представляє однину, а Users - множину.

Через **параметр options** в конструктор контексту даних передаватимуться **налаштування контексту.**

У конструкторі за допомогою виклику **Database.EnsureCreated()** за визначенням моделей буде **створюватись база даних** (якщо вона відсутня).

Щоб **підключатися до бази даних**, потрібно **встановити параметри підключення**. Для цього змінимо файл **appsettings.json**, додавши до нього визначення рядка підключення:

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

          // решта вмісту файлу
}
```

У цьому випадку використовуємо **спрощений двигун бази даних LocalDB**, який представляє легковагу версію SQL Server Express, призначену спеціально для розробки програм.

Додавання контексту даних як сервісу дозволить отримувати їх у **конструкторі контролера** через механізм застосування залежностей.

Необхідні налаштування у файлі **Program.cs**:

```
using Microsoft.EntityFrameworkCore;
using MvcApp.Models;       // простір імен класу ApplicationContext

var builder = WebApplication.CreateBuilder(args);

     // отримуємо рядок підключення з конфігураційного файлу
string connection =
builder.Configuration.GetConnectionString("DefaultConnection");

     // додаємо контекст ApplicationContext як сервіс у додаток
```

```
builder.Services.AddDbContext<ApplicationContext>(options =>
options.UseSqlServer(connection));

builder.Services.AddControllersWithViews();
var app = builder.Build();
app.MapDefaultControllerRoute();
app.Run();
```

6.2. Отримання контексту даних у контролері

Оскільки **контекст даних** додається **як сервіс**, то у конструкторі контролера отримуємо переданий контекст даних.

Для взаємодії з базою даних у контролері визначається **змінна контексту даних ApplicationContext db.**

```
public class HomeController : Controller
  {
      ApplicationContext db;
      public HomeController(ApplicationContext context)
      {
          db = context;
      }
  }
```

6.3. Додавання та виведення даних

Для додавання нового об'єкту у базу даних і виводу з неї всіх об'єктів визначимо в **контролері** три методи:

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using MvcApp.Models;

namespace MvcApp.Controllers
{
    public class HomeController : Controller
    {
        ApplicationContext db;
        public HomeController(ApplicationContext context)
        {
            db = context;
```

```
        }

        public async Task<IActionResult> Index()
        {
            return View(await db.Users.ToListAsync());
        }
        public IActionResult Create()
        {
            return View();
        }
        [HttpPost]
        public async Task<IActionResult> Create(User user)
        {
            db.Users.Add(user);
            await db.SaveChangesAsync();
            return RedirectToAction("Index");
        }
    }
}
```

У методі Index() за допомогою виклику **db.Users.ToListAsnc() отримуються об'єкти з бази даних**, створюється з них список та передається до подання.

А в Post-методі Create() за допомогою виклику **db.Users.Add()** для даних з об'єкта user формується **SQL-вираз INSERT**, а виклик **db.SaveChangesAsync()** виконує цей вираз, тим самим **додаючи дані до бази даних**.

Подання Create.cshtml:

```
@model MvcApp.Models.User

<h2> Додавання користувача </h2>
<form asp-action="create" asp-controller="home">
    <p>
        <label asp-for="Name"> Ім'я </label><br />
        <input type="text" asp-for="Name" />
    </p>
    <p>
        <label asp-for="Age">Вік</label><br />
        <input type="number" asp-for="Age" />
    </p>
    <p>
        <input type="submit" value=" Відправити " />
    </p>
</form>
```

Подання **Index.cshtml** відповідатиме за виведення об'єктів:

```
@model IEnumerable<MvcApp.Models.User>

<h2> Список користувачів </h2>
<p><a asp-action="Create"> Додати користувача </a></p>
<table class="table">
    <tr><th>Ім'я</th><th>Вік</th></tr>
    @foreach (var item in Model)
    {
        <tr>
            <td>@item.Name</td>
            <td>@item.Age</td>
        </tr>
    }
</table>
```

Подання **_ViewImports**.cshtml підключає **tag-хелпери** в уявлення:

```
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

Запустимо програму і звернемося до методу Create (рис. 6.2).

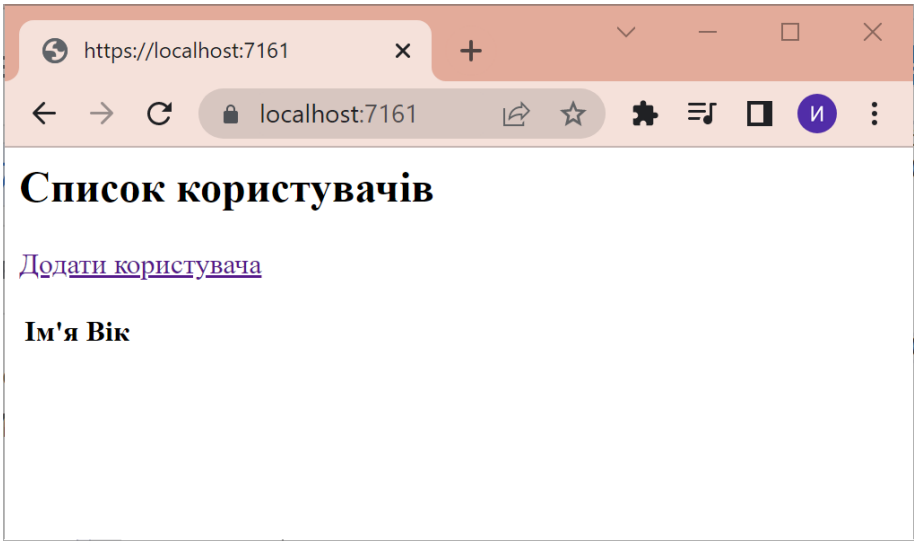


Рис. 6.2. Виконання методу Index()

Введемо у форму дані (рис. 6.3).

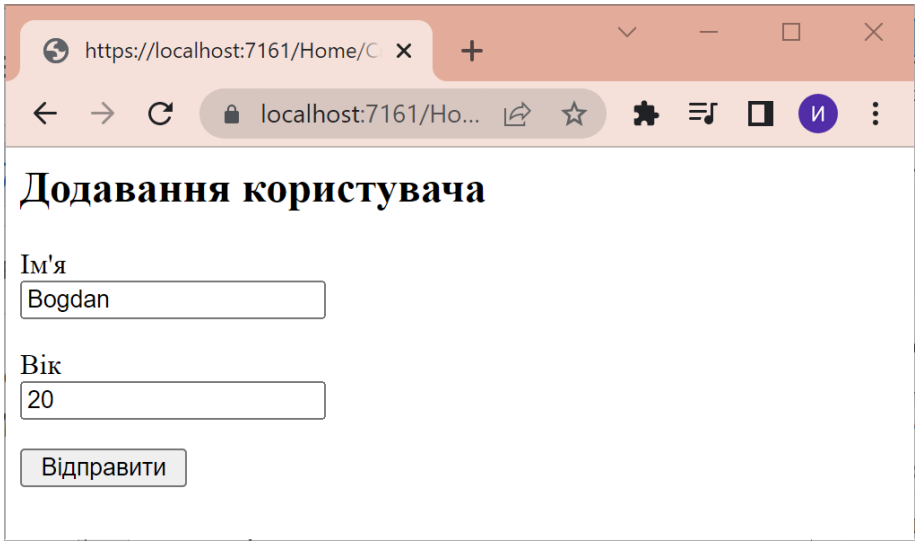


Рис. 6.3. Виконання методу Create()

Натиснемо на кнопку (рис. 6.4).

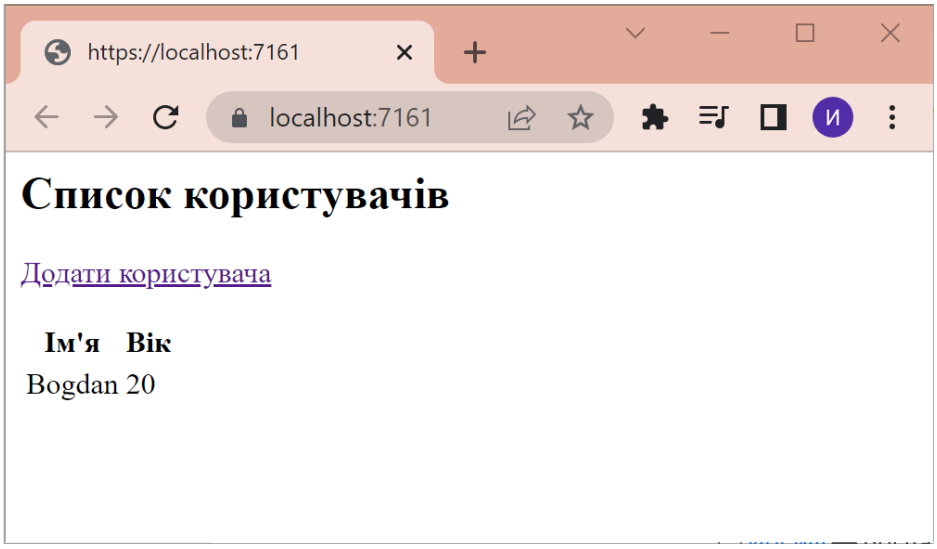


Рис. 6.4. Виконання методу Index()

Після виконання операцій з даними можна знайти автоматично згенеровану базу даних у вікні SQL Server Object Explorer і переглянути всі дані, які вона містить (рис. 6.5).

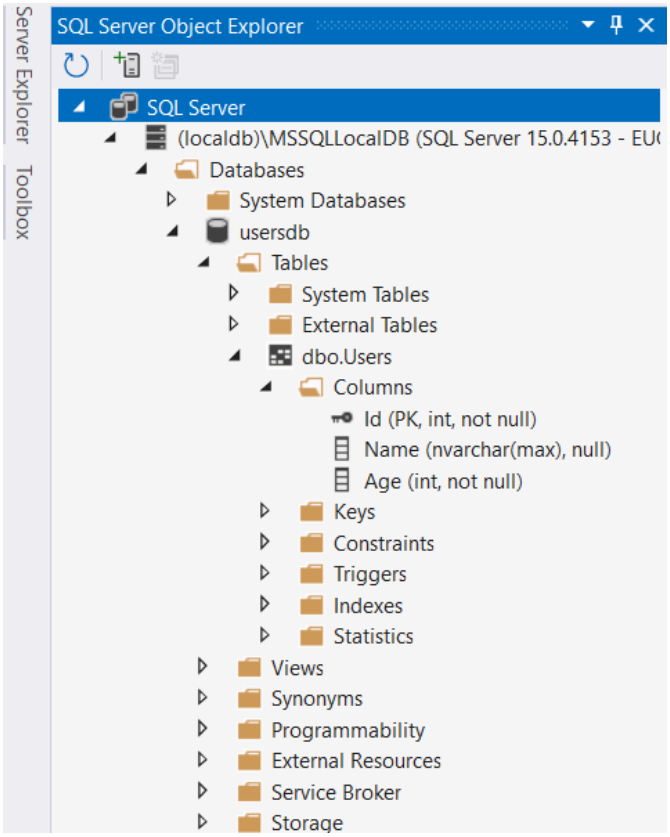


Рис. 6.5. Згенерована база даних у вікні SQL Server Object Explorer

6.4. Контрольні питання

1. Що таке модель?
2. Яка особливість анемічної моделі?
3. Які є групи моделей?
4. Де у проекті розміщуються моделі?
5. Для чого використовуються моделі подання?
6. Що таке Entity Framework Core?
7. Як до проекту підключити Entity Framework Core?
8. Як контролер взаємодіє з базою даних?

7. Майстер-сторінки layout

Майстер-сторінки або **макети** дають змогу вказати **єдиний шаблон для подань** і використовуються для створення узгодженого уніфікованого вигляду сайту. По суті, майстер-сторінки – це ті самі подання, які можуть містити інші подання.

Наприклад, на майстер-сторінці можна визначити меню, загальне для всіх інших уявлень, також підключити загальні стилі та сценарії. В результаті не доведеться вказувати шлях до файлів стилів на кожному окремому поданні, а потім змінювати його при необхідності. А спеціальні теги дозволяють вставляти інші подання в певному місці на майстер-сторінках.

Приклад. На майстер-сторінці **layout** визначається **заголовок (Header)** і **підвал (Footer)** сайту загальні для всіх інших уявлень.

Використання майстер-сторінки **layout** з поданням **View1.cshtml** показано на рис. 7.1.

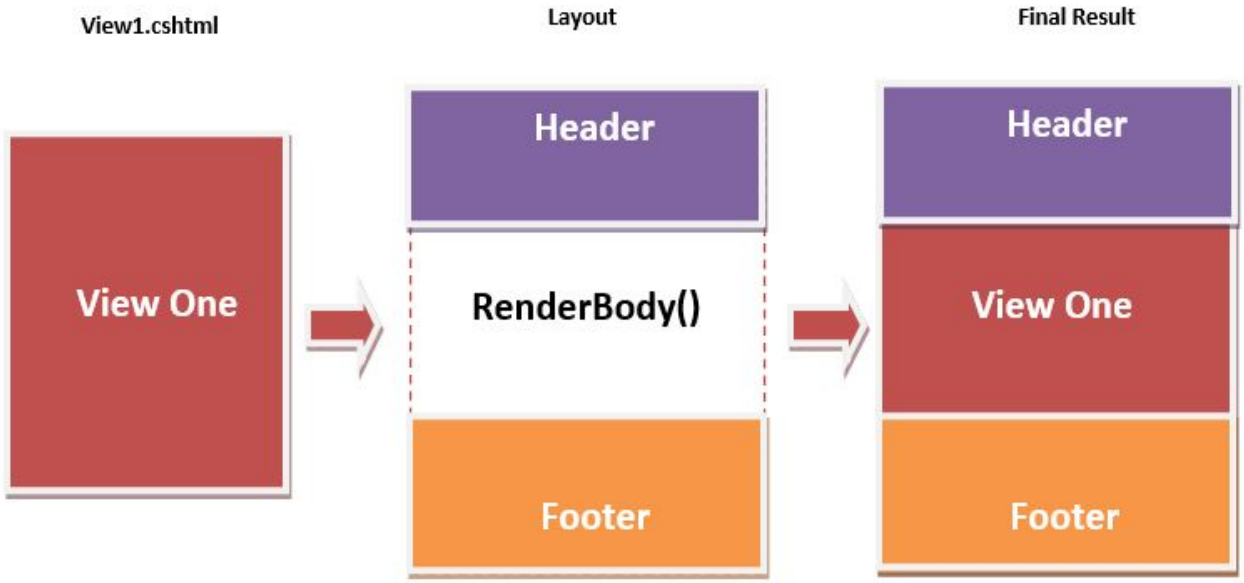


Рис. 7.1. Використання майстер-сторінки layout з поданням View1.cshtml

Використання майстер-сторінки **layout** з іншим поданням **View2.cshtml** показано на рис. 7.2.

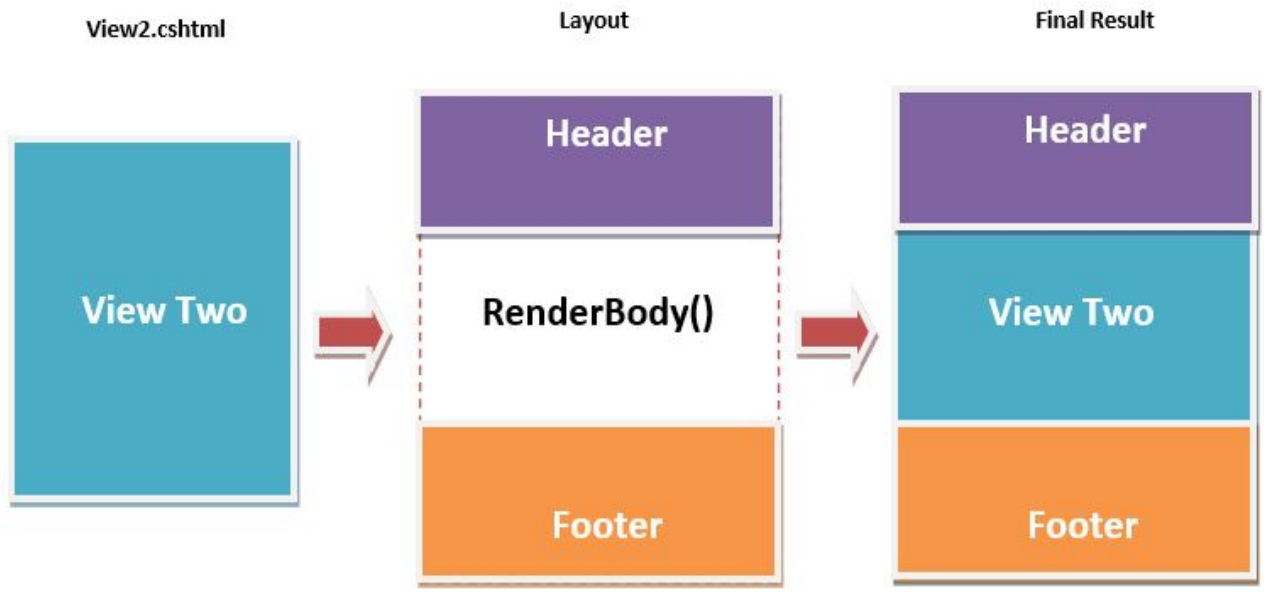


Рис. 7.2. Використання майстер-сторінки layout з поданням View2.cshtml

7.1. Визначення майстер-сторінки layout

Створемо майстер-сторінку **_Layout.cshtml** у каталозі **Views/Shared** використовуючи шаблон файлу **Razor Layout**:

```
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>DNU | @ViewBag.Title</title>
</head>
<body>
    <h2>@ViewBag.Title</h2>
    <div><a href="/Home/View1"> View1</a> | <a href="/Home/View2">
View2</a></div>
    <div>
        @RenderBody()
    </div>
</body>
</html>
```

Код майстер-сторінки нагадує повноцінну веб-сторінку: тут є основні теги <html>, <head>, <body> і так далі. Також тут можуть використовуватися конструкції Razor. Фактично це те саме подання. Наприклад, через вираз @ViewBag.Title з кожного окремого подання буде передаватися значення для заголовка веб-сторінки.

Головна ж відмінність від звичайних уявлень полягає у використанні **методу @RenderBody(),** який є **плейсхолдером** і на місце якого потім будуть підставлятися інші подання, що використовують цю майстер-сторінку. У результаті встановлюється для всіх уявлень веб-додатку **однаковий стиль оформлення.**

Створемо два подання: View1.cshtml та View2.cshtml.

Код подання **View1.cshtml:**

```
@{   ViewBag.Title = " View1";
     Layout = "/Views/Shared/_Layout.cshtml";
}
<h3> View1 Content</h3>
```

Значення **ViewBag.Title** застосовується на майстер-сторінці для виведення заголовка.

За допомогою **властивості Layout** встановлюється майстер-сторінка layout, що використовується.

В даному випадку це файл на шляху **/Views/Shared/_Layout.cshtml**

Код уявлення **View2.cshtml:**

```
@{   ViewBag.Title = " View2";
     Layout = "/Views/Shared/_Layout.cshtml";
}
<h3> View2 Content</h3>
```

Воно виглядає аналогічно уявленню Index.cshtml.

Ці уявлення викликаються **методами View1()** та **View2()** контролера HomeController:

```
public class HomeController : Controller
    {
        public IActionResult View1() => View();
        public IActionResult View2() => View();
    }
```

Запустимо проект і в браузері при зверненні до обох дій контролера **View1** та **View2** бачимо у браузері **веб-сторінки з однаковим стилем оформлення** (рис. 7.3).

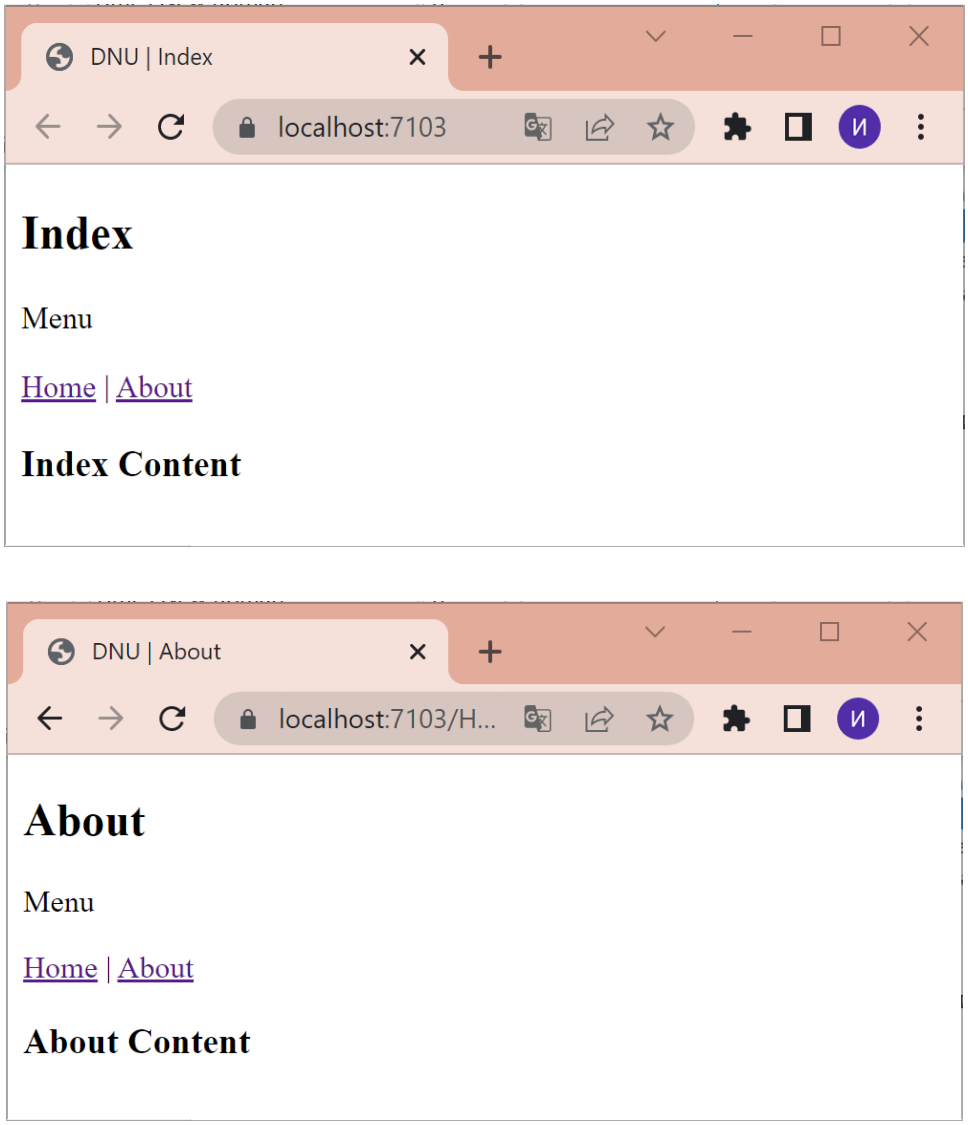


Рис. 7.3. Використання майстер-сторінки layout

При необхідності можливо визначати **різні майстер-сторінки**, наприклад, окремі майстер-сторінки для різних контролерів, і таким чином підключати їх до конкретних уявлень.

7.2. Файл _ViewStart.cshtml

У кожному поданні необхідно явно прописувати, яку **майстер-сторінку layout** буде застосовувати подання. Щоб спростити цю дію, можна використовувати файли **_ViewStart.cshtml.**

Код файлу **_ViewStart.cshtml** **додається на початок коду уявлень** під час їх запуску. При цьому файли уявлень, до яких застосовується _ViewStart.cshtml, повинні перебувати з цим файлом в **одному каталозі.**

Визначимо у файлі **_ViewStart.cshtml** наступний код:

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

У кожному поданні через синтаксис Razor є **властивість Layout**, яка зберігає **посилання на майстер-сторінку**.

Тут як майстер-сторінка встановлюється файл _Layout.cshtml. У цьому розширення можна використовувати.

Коли відбуватиметься **рендеринг подання**, то система шукатиме майстер сторінку _Layout за наступними шляхами:

/Views/[Назва_контролера]/_Layout.cshtml
/Views/Shared/_Layout.cshtml

Якщо в обох папках: і /Views/[Назва_контролера], і /Views/Shared/ є файл з однаковим ім'ям, наприклад, _Layout.cshtml, то до подання застосовується файл, який знаходиться з **ним в одній папці** як більш пріоритетний.

Таким чином, **можна визначити для уявлень кожного окремого контролера** або **уявлень**, що знаходяться в одній папці, свою **окрему майстер-сторінку**.

Після визначення файлу _ViewStart.cshtml можна видалити з уявлень View1.cshtml та View2.cshtml **підключення майстер-сторінки**. Наприклад, подання View1.cshtml буде виглядати наступним чином:

```
@{
    ViewBag.Title = "Index";
}
<h3>Index Content</h3>
```

7.3. Файл _ViewImports.cshtml

Файл **_ViewImports.cshtml** дозволяє **за умовчанням** підключити до подання деякий **функціонал.**

Приклад.

Нехай у проекті є певний **клас Person**:

```
namespace MvcApp
  {
     public record class Person(string Name, int Age);
  }
```

Щоб використовувати тип Person у поданні, необхідно **імпортувати** за допомогою **директиви using простір імен**, де цей тип визначено. Використуємо тип Person у поданні Index.cshtml:

```
@using MvcApp   @* Підключаємо простір імен класу Person *@
@{
    Layout = null;
    Person bogdan = new Person("Bogdan", 20);
```

```
}
<h2>Person Data</h2>
<h3>Name: @bogdan.Name</h3>
<h3>Age: @bogdan.Age</h3>
```

Якщо існує багато уявлень, де використовується той самий тип Person, необхідно визначити той самий вираз імпорту в усіх поданнях.

Це може створювати деякі незручності:

• повторюється той самий код;

• якщо простір імен зміниться, необхідно буде змінювати всі подання;

• можливо, захочемо підключити ще якісь простір імен, що ще збільшить роботу, якщо будуть якісь зміни.

За допомогою файлу **_ViewImports.cshtml** можна вирішити цю проблему.

Створемо файл _ViewImports.cshtml у каталозі Views використовуючи шаблон файлу Razor View Imports. Додамо до цього файлу підключення простору імен класу Person:

```
@using MvcApp   @* Підключаємо простір імен класу Person *@
```

У цьому випадку функціональність простору імен **MvcApp** буде **автоматично підключатися до всіх уявлень.**

І тоді можна **прибрати** з подання Index.cshtml рядок підключення простору імен.

Для **кожної групи поданнь** в одній папці можемо визначити **свій файл _ViewImports.cshtml**.

Наприклад, якщо необхідно окремо підключити ряд просторів імен лише у **подання контролера HomeController**, тоді треба додати файл _ViewImports.cshtml **в каталог Views/Home**. І в цьому випадку всі директиви та вирази з файлу **Views/Home/_ViewImports.cshtml** будуть застосовуватися до уявлень тільки з папки **Views/Home**. Крім того, до всіх уявлень у всіх папках продовжить застосовуватись **глобальний файл Views/_ViewImports.cshtml**.

7.4. Контрольні питання

1. Що таке майстер-сторінка?
2. Що робить метод RenderBody()?
3. Як підключити майстер-сторінку?
4. Де у проекті розміщується майстер-сторінка()?
5. Як використовується файл _ViewStart.cshtml?
6. Як використовується файл _ViewImports.cshtml?
7. Де у проекті розміщуються файли _ViewStart.cshtml та _ViewImports.cshtml?

8. Використання tag-хелперів

Для відображення контенту в поданні можна використовувати стандартні HTML-елементи, що дозволяють створювати блоки, списки, таблиці і т.д. Але крім власне HTML-елементів в ASP.NET Core MVC, для створення розмітки можна використовувати спеціальні **допоміжні методи** - **tag-**хелпери **(**є і HTML-хелпери).

Tag-хелпери - це **функціонал,** призначений для **генерації HTML-розмітки**. T**ag-хелпер**и використовуються в поданні і виглядають як звичайні HTML-елементи або атрибути, але коли додаток запущено, вони **обробляються движком Razor на стороні сервера** і в кінцевому підсумку перетворюються в **стандартні HTML-елементи**.

Tag-хелпери представляють більш зручний спосіб генерації HTML-елементів, ніж звичайні HTML-хелпери, тому що **tag-хелпери** багато в чому виглядають як звичайні HTML-елементи, **Visual Studio** має **вбудовану підтримку IntelliSense для tag-хелперів**.

Приклад використання tag-хелперов.

Нехай у папці Views/Home є подання Index.cshtml з наступним кодом:

```
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
<a asp-controller="Home" asp-action="Contacts">Контакти</a>
```

Спочатку у поданні йде директива addTagHelper

```
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

Перший параметр директиви вказує на tag-хелпери, які будуть доступні у поданні, а другий параметр визначає **бібліотеку хелперів**.

У прикладі директива використовує синтаксис підстановок - знак зірочки ("*") означає, що підключаються всі хелпери з бібліотеки Microsoft.AspNetCore.Mvc.TagHelpers.

Далі йде власне tag-хелпер:

```
<a asp-controller="Home" asp-action="Contacts">Контакти</a>
```

Даний хелпер створює посилання, для якого як контролер використовується Home, а як метод - Contacts.

У результаті при запуску проекту замість даного tag-хелпера буде сформовано гіперпосилання, після натискання на яке запит буде оброблятися методом Contacts контролера Home.

Видалення tag-хелперів

Директива **removeTagHelper** видаляє раніше додані tag-хелпери. Її застосування аналогічне:

```
@removeTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

Ця директива може бути корисною, якщо необхідно **обмежити** застосування хелперів в одному поданні або групі уявлень. Цю директиву можна також визначати у файлі _ViewImports.cshtml.

8.1. Тег-хелпер AnchorTagHelper

AnchorTagHelper представляє тег-хелпер, який дозволяє **створювати посилання**. Він може приймати низку спеціальних атрибутів:

• **asp-controller**: вказує на контролер, якому призначено запит;

• **asp-action**: вказує на дію контролера;

• **asp-area**: вказує на дію в області, де розташований контролер або сторінка RazorPage (якщо вони знаходяться в окремій області);

• **asp-page**: вказує на RazorPage, яка оброблятиме запит;

• **asp-page-handler**: вказує на обробник сторінки RazorPage, яка буде застосовуватись для обробки запиту;

• **asp-host**: вказує на домен сайту;

• **asp-protocol**: визначає протокол (http або https);

• **asp-route**: вказує назву маршруту;

• **asp-all-route-data**: встановлює набір значень параметрів;

• **asp-route-[назва параметра]:** визначає значення для певного параметра;

• **asp-fragment:** визначає ту частину хеш-посилання, яка йде після символу ґрат #. Наприклад, "paragraph2" у посиланні http://mysite.com/#paragraph2;

Атрибути asp-action та asp-controller

Посилання в ASP.NET Core з використанням tag-хелпера AnchorTagHelper:

```
<a asp-controller="Home" asp-action="About">Про сайт</a>
```

У разі використовується не елемент HTML <a />, саме хелпер **AnchorTagHelper**. Його атрибут **asp-controller** вказує на **назву контролера**, а **asp-action** визначає **дію**, до якої йтиме запит.

Якщо вказаний атрибут asp-action, але не вказаний asp-controller, то контролером використовується той контролер, який пов'язаний з поточним поданням.

Якщо необхідно встановити посилання на **дію контролера, який знаходиться в іншій області**, застосовується атрибут **asp-area**:

```
<a asp-controller="Home" asp-action="About" asp-area="Service">Про сайт</a>
```

У разі передбачається, що контролер Home перебуває у області Service.

Якщо, навпаки, в поданні, яке знаходиться в якійсь області, треба створити посилання на дію контролера, який не знаходиться у жодній області, то вказується **порожній атрибут**:

```
<a asp-controller="Home" asp-action="About" asp-area="">Про сайт</a>
```

Атрибути asp-host та asp-protocol

AnchorTagHelper **за замовчуванням** створює **локальне посилання**, якщо треба створити посилання на інший домен, то можемо застосувати атрибут **asp-host**:

```
<a asp-controller="Home" asp-action="About" asp-host="localhost.com" asp-protocol="https">Про сайт</a>
```

Цей елемент у результаті створює таке посилання: https://localhost.com/Home/About

Можна змінити стандартний протокол на https, використовуючи атрибут asp-protocol.

```
<a  asp-host="google.com" asp-protocol="https">Google</a>
```

```
<a asp-controller="doodles" asp-action="About" asp-host="google.com" asp-protocol="https"> Google About doodles</a>
```

Атрибути asp-route- та asp-all-route-data

Якщо метод приймає якісь параметри, які треба зазначити у посиланні:

```
public string GetPerson(int id) => $"Id: {id}";
```

можна використовувати атрибут **asp-route-:**

```
<a asp-controller="Home" asp-action="GetPerson" asp-route-id="2" >Person 2</a>
```

Якщо метод приймає кілька параметрів, наприклад:

```
public string GetPerson(int id, string name, int age) => $"id={id} name={name} age={age}";
```

то можна вказати кілька атрибутів asp-route-:

```
<a asp-controller="Home" asp-action="GetPerson" asp-route-id="2" asp-route-age="20" asp-route-name="Taras" >Person: 2 Taras 20</a>
```

Щоб не встановлювати всі параметри окремо, можна застосувати атрибут **asp-all-route-data**:

```
<a asp-controller="Home" asp-action="GetPerson" asp-all-route-data='new Dictionary<string,string>{{"id","2"},{"name","Taras"},{"age","20"}}'> Person: 2 Taras 20</a>
```

asp-all-route-data як значення приймає словник із параметрами та їх значеннями. В результаті буде генеруватися посилання, аналогічне попередньому.

Атрибут asp-route

За допомогою параметра **asp-route** можна **створити посилання на основі маршруту**. Наприклад, нехай є такі маршрути:

```
app.MapControllerRoute(
    name: "computer",
    pattern: "ComputerStore",
    defaults: new {controller = "Computer", action = "Index"});

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

Візьмемо перший маршрут на ім'я "**computer**":

```
<a asp-route="computer">Комп’ютери</a>
```

Такий тег створить таке посилання:

```
<a href="/ComputerStore">Комп’ютери</a>
```

8.2. Tag-хелпери форм

Окрема група tag-хелперів дозволяє створювати форми HTML та їх елементи. Тег-хелпери, що використовуються для створення форм, аналогічні відповідним елементам HTML за тим винятком, що вони **додають додаткову функціональність**.

Для створення форми використається клас **FormTagHelper**, представлений тегом form. Цей тег може приймати такі атрибути:

• **asp-controller**: вказує на контролер, якому призначено запит;

• **asp-action**: вказує на дію контролера;

• **asp-area**: вказує на назву області, в якій буде викликатись контролер для обробки форми;

• **asp-antiforgery**: якщо має значення true, то для цієї форми буде генеруватися маркер;

• **asp-route**: вказує назву маршруту;

• **asp-all-route-data**: встановлює набір значень параметрів;

• **asp-route-[назва параметра]:** визначає значення для певного параметра

• **asp-page**: вказує на сторінку RazorPage, яка оброблятиме запит;

• **asp-page-handler**: вказує на обробник сторінки RazorPage, який застосовується для обробки запиту;

• **asp-fragment**: вказує фрагмент, який додається до запитуваної адреси після символу #.

Приклад форми, що відправлятиме дані методу Create котроллера Home і для форми генеруватиметься маркер антипідробки antiforgery token:

```
<form asp-antiforgery="true" asp-action="Create" asp-controller="Home">
```

Решта тегів, які використовуються на формах, мають один загальний **атрибут asp-for**, який вказує, для якої **властивості моделі** створюється елемент.

Тег LabelTagHelper використовує тег label для створення мітки:

```
<label asp-for="Name"></label>
```

Тег InputTagHelper створює поле введення:

```
<input asp-for="Name" />
```

Тег TextAreaTagHelper використовується для створення багаторядкового текстового поля textarea:

```
<textarea asp-for="Name"></textarea>
```

Тег SelectTagHelper створює елемент списку:

```
<select asp-for="CompanyId" asp-items="ViewBag.Companies"></select>
```

Атрибут asp-items вказує на об'єкт IEnumerable<SelectListItem>, який буде використовуватись для заповнення списку.

При необхідності можна **вказати елемент**, який **відображатиметься за замовчуванням**:

```
<select asp-for="CompanyId" asp-items="ViewBag.Companies">
  <option selected="selected" disabled="disabled">Виберіть компанію
</option>
</select>
```

Робота з enum

Розглянемо прив'язку об'єкта select до перерахування.

Нехай визначено **тип enum** DayTime та модель DayTimeViewModel, яка зберігатиме вибране значення DayTime:

```
using System.ComponentModel.DataAnnotations;

public enum DayTime
{
     [Display(Name = "Ранок")]
     Morning,
     [Display(Name = "День")]
     Afternoon,
     [Display(Name = "Вечір")]
     Evening,
     [Display(Name = "Ніч")]
     Night
```

```
}
public class DayTimeViewModel
{
     public DayTime Period { get; set; }
}
```

У **контролері** визначимо пару методів, для відправки подання та отримання обраного значення:

```
public IActionResult Index() => View();
[HttpPost]
public string Index(DayTimeViewModel model) => model.Period.ToString();
```

В поданні **Index.cshtml** створемо список за допомогою **конструктора SelectList**:

```
@using MvcApp.Models
@model DayTimeViewModel

<form method="post">
    <div>
        <div>
                <label asp-for="Period"> Час доби </label>
                <select asp-for="Period" asp-items="@new
                SelectList (Enum.GetNames(typeof(DayTime)))">
               </select>
        </div>
        <div>
            <input type="submit" value="Save" />
        </div>
    </div>
</form>
```

У результаті вийде випадаючий список зі значеннями з перерахування (рис. 8.1):

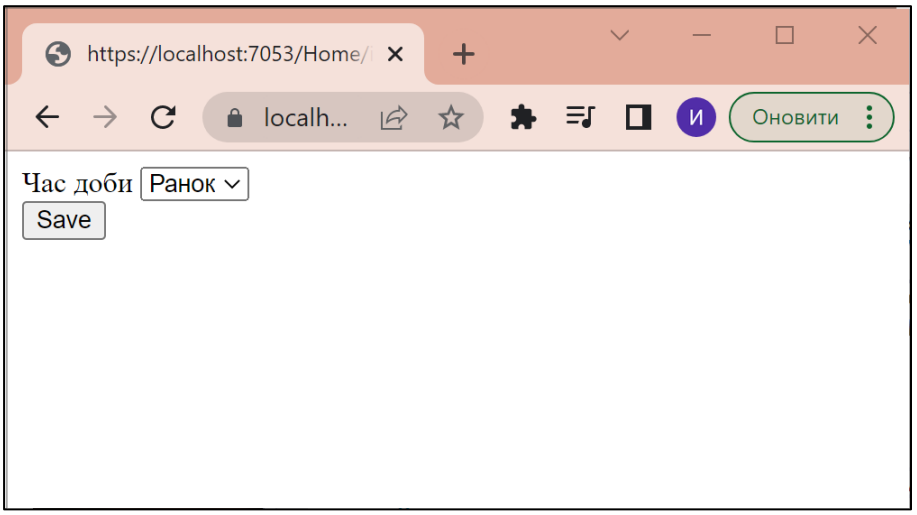


Рис. 8.1. Приклад роботи з тег-хелперами форм

8.3. Контрольні питання

1. Що таке tag-хелпери?
2. Як підключити бібліотеку tag-хелперів?
3. Який тег-хелпер дозволяє створювати посилання?
4. Який тег-хелпер дозволяє створювати форми?
5. Які атрибути у тег-хелпера FormTagHelper? Що вони роблять?
6. Які атрибути у тег-хелпера AnchorTagHelper? Що вони роблять?

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Матвєєва Н.О., Пономарьов І.В. Конспект лекцій з дисципліни «Системне програмування». - Дніпро: Вид-во «Ліра», 2022. – 96 с.
2. Матвєєва Н.О., Пономарьов І.В. Паралельне програмування на платформі .NET: навчальний посібник — Дніпро, 2023. — 238 с.
3. Wolf J. HTML and CSS. The Comprehensive Guide - Rheinwerk Computing, 2023. - 814 p.
4. Марк Дж. Прайс. C# 10 і .NET 6 – Сучасна міжплатформна розробка: створюйте програми, веб-сайти та служби за допомогою ASP.NET Core 6, Blazor і EF Core 6 за допомогою Visual Studio 2022 і Visual Studio Code, 2021, – 824 с.
5. Фрімен А. ASP.NET Core MVC з прикладами C# для професіоналів. 6-те видання - Видавництво Вільямс, 2017. - 992 с.
6. Lynn S. Software Architecture with ASP.NET 6 MVC and Visual Studio 2022. Third Edition – 2021. - 310 p.
7. Adam Freeman. Pro ASP.NET Core 6: Develop Cloud-Ready Web Applications Using MVC, Blazor, and Razor Pages. 9th Edition - Apress, 2022 – 1286p.
8. Overview of ASP.NET Core MVC [Electronic resource] – Mode of access: URL: https://learn.microsoft.com/en-us/aspnet/core/mvc/overview?view=aspnetcore-7.0 – Last access: 2022. – Title from the screen.
9. The complete ASP.NET Core MVC Tutorial [Electronic resource] – Mode of access: URL: https://asp.mvc-tutorial.com/ – Last access: 2022. – Title from the screen.
10. Get started with ASP.NET Core MVC [Electronic resource] – Mode of access: URL: https://learn.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/start-mvc?view=aspnetcore-7.0&tabs=visual-studio – Last access: 2023. – Title from the screen.

Додаток А. Презентація Основи HTML

ТЕХНОЛОГІЇ РОЗРОБКИ КРОСПЛАТФОРМЕНИХ ВЕБ-ДОДАТКІВ

2. Основи HTML

Елементи HTML

Документ HTML складається з **елементів**, а елементи складаються з **тегів**.
Як правило, елементи мають тег, що **відкриває** та **закриває**, які полягають у **кутові дужки**.
Наприклад:

```
<div>Текст елемента div</div>
```

Тут визначено елемент div, який має відкриваючий тег і тег, що закриває . Між цими тегами міститься вміст елемента div. В даному випадку як вміст виступає простий текст "Текст елементу div".

Елементи також можуть складатися з **одного тега**, наприклад елемент **
**, функція якого - перенесення рядка.

```
<div>Текст <br/> елемента div</div>
```

Такі елементи називають **пустими елементами**.
Слеш, що закриває, згідно з специфікацією необов'язковий, і рівнозначно використанню тега без слешу:

```
<br>
```

2

Атрибути HTML

Кожен **елемент** всередині тега може мати **атрибути**.

Наприклад:

<div **style="color:red;"**>Кнопка
<input **type="button" value="Натиснути"**>

Тут визначено два елементи: div та input. Елемент div має атрибут style. Після знака і в лапках пишеться значення атрибута: style="color:red;" - вказує, що колір тексту буде червоним.
Другий елемент - елемент input, що складається з одного тега, має два атрибути: type (вказує тип елемента - кнопка) і value (визначає текст кнопки).

Існують **глобальні** або **загальні для всіх елементів атрибути**, наприклад, style, а є *специфічні*, що застосовуються до певних елементів, наприклад, type.

Крім звичайних атрибутів, існують ще булеві або **логічні атрибути**. Подібні атрибути можуть не мати значення. Наприклад, у кнопки можна задати атрибут disabled:

<input type="button" value="Натиснути" **disabled**>

Атрибут disabled показує, що цей елемент вимкнено.

3

Створення документа HTML

Для створення документа HTML5 використовуються насамперед два елементи:

DOCTYPE та html.

Елемент DOCTYPE повідомляє веб-браузеру тип документа.

<!DOCTYPE html> вказує, що цей документ є документом html і що використовується версія мови розмітки html5.

А елемент **html** між своїм тегами, що відкриває і закриває, містить весь **вміст документа**.

Всередині елемента html можемо розмістити два інші елементи:

head та body.

Елемент **head** містить **метадані веб-сторінки** – **заголовок веб-сторінки**, тип кодування тощо, а також посилання на зовнішні ресурси – стилі, скрипти, якщо вони використовуються.

Елемент **body** власне визначає **вміст html-сторінки**.

4

Приклад створення документа HTML

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>Документ HTML5</title>
    </head>
    <body>
        <div> Зміст документа HTML5</div>
    </body>
</html>
```

В елементі **head** визначено два елементи:

- елемент title представляє заголовок сторінки
- елемент meta визначає метаінформацію сторінки.

Для коректного відображення символів бажано вказувати кодування. У цьому випадку за допомогою атрибуту charset="utf-8" вказуємо кодування utf-8, яке коректно відображатиме кириличні символи.

В елементі **title** заголовок "Документ HTML5" - саме така назва матиме **вкладка браузера**.

У межах елемента елемента body використовується лише один елемент – div, який оформляє блок. Вмістом цього блоку є простий рядок. Текст, визначений усередині елемента body - побачимо в основному полі браузера.

5

Елемент div

Елемент **div** служить для **структуризації контенту** на веб-сторінці, укладання вмісту в окремі блоки. Div створює блок, який за промовчанням розтягується по всій ширині браузера, а наступний після div елемент переноситься на новий рядок.

Наприклад:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>Документ HTML5</title>
    </head>
    <body>
        <div>Заголовок документу HTML5</div>
        <div>Текст документу HTML5</div>
    </body>
</html>
```

6

Параграфи

Параграфи створюються за допомогою тегів та , які містять деякий вміст. Кожен новий параграф розміщується на новому рядку.

Застосуємо параграфи:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>Документ HTML5</title>
    </head>
    <body>
        <div>Заголовок документу HTML5</div>
        <div>
            <p>Перший параграф</p>
            <p>Другий параграф</p>
        </div>
    </body>
</html>
```

Якщо в рамках одного параграфа нам треба перенести текст на інший рядок, то можемо скористатися елементом
:

```
<p>Перший рядок.<br/>Другий рядок.</p>
```

7

Заголовки

Елементи <h1>, <h2>, <h3>, <h4>, <h5> і <h6> служать створення заголовків різного рівня:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>Заголовки в HTML5</title>
    </head>
    <body>
        <h1>Заголовок першого рівня</h1>
        <h2>Заголовок другого рівня</h2>
        <h3>Заголовок третього рівня</h3>
        <h4>Заголовок четвертого рівня</h4>
        <h5>Заголовок п'ятого рівня</h5>
        <h6>Заголовок шостого рівня</h6>
    </body>
</html>
```

Заголовки виділяють шрифт жирним і за замовчуванням мають деякий розмір: від найбільшого <h1> до найдрібнішого <h6>. При визначенні заголовків слід враховувати, що на сторінці має бути лише **один заголовок першого рівня**, тобто **<h1>.** Він виконує роль **головного заголовка веб-сторінки**.

8

Форматування тексту

Ряд елементів html призначені для форматування текстового вмісту, наприклад, виділення жирним або курсивом і тому подібне.

Розглянемо ці елементи:

- **<b>**: виділяє текст жирним
- <del>: закреслює текст
- <i>: виділяє текст курсивом
- : виділяє текст курсивом, на відміну від тега <i> носить **логічне значення**, надає тексту, що виділяється, відтінок важливості
- <s>: закреслює текст
- <small>: робить текст трохи менше розміром, ніж оточуючий
- : виділяє текст жирним. На відміну від тега <b> призначено для логічного виділення, щоб показати важливість тексту. А <b> не носить характеру логічного виділення, виконує функції лише форматування
- <sub>: містить текст під рядком
- <sup>: містить текст над рядком
- <u>: підкреслює текст
- <ins>: визначає вставлений (або доданий) текст
- <mark>: виділяє текст кольором, надаючи йому відтінку важливості

9

Приклад форматування тексту

Застосуємо всі ці елементи форматування тексту:

```
<!DOCTYPE html>
<html>
    <head>
      <meta charset="utf-8">
      <title>Форматування тексту в HTML5</title>
    </head>
  <body>
    <p>Форматування у <mark>HTML5</mark></p>
    <p>Це <b>виділений</b> текст</p>
      <p>Це <strong>важливий</strong> текст</p>
    <p>Це <del>закреслений</del> текст</p>
    <p>Це <s>недійсний</s> текст</p>
    <p>Це <em>важливий</em> текст</p>
    <p>Це текст <i>курсивом</i> </p>
    <p>Це <ins>доданий</ins> текст</p>
    <p>Це <u>підкреслений</u> текст</p>
    <p>X<sub>i</sub> = Y<sup><small>2</small></sup> + Z<sup><small>2</small></sup></p>
  </body>
</html>
```

10

Робота із зображеннями

Для виведення зображень HTML використовується **елемент img**.
Цей елемент представляє нам два важливі **атрибути**:

• **src**: шлях до зображення. Це може бути відносний або абсолютний шлях у файловій системі або адреса в Інтернеті

• **alt**: опис тексту. Якщо браузер з якихось причин не може відобразити зображення (наприклад, якщо атрибут src некоректно заданий шлях), то браузер показує замість самої картинки даний текстовий опис.

Атрибут alt ще важливий тим, що пошукові системи текстового опису можуть індексувати зображення.

Наприклад, покладемо в ту ж папку, де у нас лежить файл index.html, якийсь файл зображення. І потім відобразимо його на веб-сторінці:

```
<!DOCTYPE html>
<html>
   <head>
      <meta charset="utf-8">
      <title>Тег img у HTML5</title>
   </head>
   <body>
      <img src= "img1.png" alt= «Тут має бути зображення" />
   </body>
</html>
```

Елемент img є порожнім елементом, тобто не містить тега, що закривається.

11

Посилання

Посилання, представлені елементом **<a>,** відіграють важливу роль - вони забезпечують **навігацію між окремими документами**.

Цей елемент має такі атрибути:

• **href**: визначає **адресу посилання**
• hreflang: вказує на мову документа, на яку веде це посилання
• media: визначає пристрій, для якого призначено посилання
• rel: визначає відношення між цим документом та ресурсом, на який веде посилання
• **target**: визначає, як документ за посиланням має відкриватися
• type: вказує на mime-тип ресурсу за посиланням.

Найбільш важливим атрибутом є href:

```
<!DOCTYPE html>
<html>
   <head>
      <meta charset="utf-8">
      <title>Посилання</title>
   </head>
   <body>
      <a href="content.html">Підручник з HTML5</a>
   </body>
</html>
```

12

Посилання (2)

Тут для посилання використовується ***відносний шлях*** *content.html. Тобто в* ***одній папці*** *з цим документом повинен знаходитись файл content.html, на який йтиме перехід по натисканню на посилання.*

Можна використовувати абсолютні шляхи з повною вказівкою адреси:

Підручник з HTML5

За замовчуванням ресурси, на які ведуть посилання, відкриваються у тому ж вікні. За допомогою атрибуту target можна перевизначити цю дію.

Наприклад, відкриття документа на посилання в новому вікні:

Підручник з HTML5

Значення **_blank** вказує браузеру, що ресурс потрібно відкрити у новій вкладці.

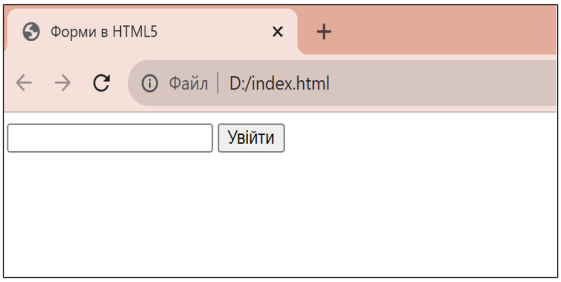
Помістивши всередину елемента <a> елемент <img>, можна зробити **посилання-зображення**:

```
<a href="index.html">
    <img src="cover.png" alt="HTML tutorial">
</a>
```

13

Форми

Форми в html представляють один із способів для введення та відправлення даних.
Усі поля форми розміщуються між тегами **<form>** та **</form>.**



Наприклад, створимо найпростішу форму:

```
<!DOCTYPE html>
<html>
   <head>
      <meta charset="utf-8">
      <title>Форми в HTML5</title>
   </head>
   <body>
      <form method="post" action="http://localhost:8080/login.php">
         <input name="login"/>
         <input type="submit" value="Увійти" />
      </form>
   </body>
</html>
```

Для налаштування форм елемента form визначені наступні атрибути:

- **method**: встановлює метод надсилання даних на сервер.

Допустимі два значення: **post** і **get**.
Значення **post** дозволяє передати дані на веб-сервер через спеціальні **заголовки**.
А значення **get** дозволяє надіслати дані через **рядок запиту**.

14

Форми (2)

• **action**: встановлює **адресу**, на яку передаються дані форми
• **enctype**: встановлює **тип даних**, що передаються.

Тип даних у свою чергу може набувати наступних значень:
- **application/x-www-form-urlencoded**: кодування даних, що надсилаються **за умовчанням**
- **multipart/form-data**: це кодування застосовується при надсиланні **файлів**
- **text/plain**: це кодування застосовується при надсиланні **простої текстової інформації**.

У цьому прикладі:

```
<form method="post" action="http://localhost:8080/login.php">
    <input name="login"/>
    <input type="submit" value="Увійти" />
</form>
```

у форми встановлено метод "post", тобто всі значення форми відправляються в тілі запиту, а адресою служить рядок http://localhost:8080/login.php. Адреса тут вказана випадковим чином.

Як правило, за вказаною адресою працює веб-сервер, який, використовуючи одну з технологій серверної сторони (PHP, NodeJS, ASP.NET тощо), може отримувати запити та повертати відповідь.

15

Елемент форми input

Форми складаються із певної кількості **елементів введення**. Всі елементи введення розміщуються між тегами <form> та </form>.

Найбільш поширеним елементом введення є елемент **input.** Однак реальна дія цього елемента залежить від того, яке значення встановлено у його **атрибуті type**.

А він може приймати такі основні значення:

- **text**: звичайне текстове поле
- password: теж текстове поле, тільки замість символів, що вводяться, відображаються зірочки, тому в основному використовується для введення пароля
- radio: радіо або перемикач. З групи радіокнопок можна вибрати лише одну
- checkbox: елемент прапорця, який може перебувати у відзначеному чи невідзначеному стані
- **submit**: кнопка надсилання форми
- color: поле для введення кольору
- date: поле для введення дати
- email: поле для введення адреси електронної пошти
- number: поле для введення чисел
- range: повзунок для вибору числа з деякого діапазону
- tel: поле для введення телефону
- url: поле для введення адреси url
- file: поле для вибору файлу, що відправляється
- image: створює кнопку у вигляді картинки …

16

Елементи форм

Крім елемента input у різних модифікаціях є ще невеликий **набір елементів**, які можна використовувати на формі:
- **button**: створює кнопку
- **select**: список, що випадає
- **label**: створює позначку, яка відображається поруч із полем введення
- **textarea**: багаторядкове текстове поле.

У всіх елементах введення можна встановити атрибути **name** та **value**. Ці атрибути мають важливе значення. За атрибутом **name ідентифікуємо поле введення**, а атрибут **value** дозволяє встановити **значення поля введення**.

Наприклад:

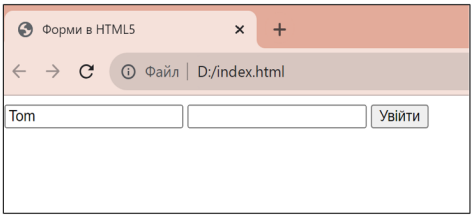
```
<!DOCTYPE html>
<html>
   <head>
      <meta charset="utf-8">
      <title>Форми в HTML5</title>
   </head>
   <body>
      <form method="get" action="index.html">
         <input type="text" name="login" value="Tom"/>
         <input type="password" name="password"/>
         <input type="submit" value="Увійти" />
      </form>
   </body>
</html>
```

Тут текстове поле має значення "Tom" (як зазначено в атрибуті value), тому під час завантаження веб-сторінки у цьому полі побачимо цей текст.

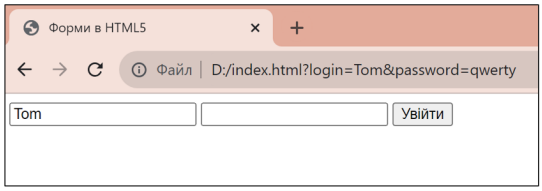
17

Елементи форм. Атрибути name та value

*Тут текстове поле має значення "**Tom**" (як зазначено в атрибуті **value**), тому під час завантаження веб-сторінки у цьому полі побачимо цей текст.*



Оскільки методом відправлення даних форми є метод "**get**", дані будуть надсилатися через **рядок запиту**. І при надсиланні побачимо введені дані у **рядку запиту**:



У рядку запиту нас цікавить наступний шматочок:

login=Tom&password=qwerty

Під час надсилання форми **браузер** з'єднує всі дані в **набір пар "ключ-значення"**.
У нашому випадку дві такі пари: login=Tom та password=qwerty.

Ключем у цих парах виступає **назва поля введення**, яке визначається атрибутом **name**, а **значенням** – власне те значення, яке введено у поле введення (або значення атрибута **value**).

Отримавши ці дані, **сервер** легко може дізнатися, які **значення поля вводу** були **введені користувачем**.

18

Елементи форм. Кнопки

Кнопки є елементом **button**. Вони мають широкі можливості щодо конфігурації.

Так, залежно від значення **атрибута type**, ми можемо створити різні типи кнопок:

- **submit**: кнопка, яка використовується для надсилання форми
- **reset**: кнопка скидання значень форми
- **button**: кнопка без спеціального призначення.

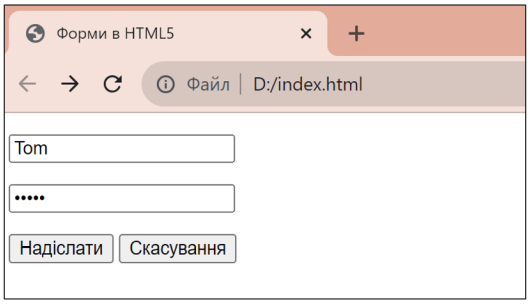
Якщо кнопка використовується для відправки форми, тобто у неї встановлений атрибут type="submit", то можемо задати ряд **додаткових атрибутів**:

- **form**: визначає форму, за якою закріплено кнопку відправлення
- **formaction:** встановлює адресу, на яку надсилається форма. Якщо елемент form заданий атрибут action, він перевизначається
- **formenctype**: встановлює формат надсилання даних. Якщо елемент form встановлений атрибут enctype, він перевизначається
- **formmethod**: встановлює метод відправлення форми (post чи get). Якщо елемент form встановлений атрибут method, він перевизначається.

19

Приклад форми з кнопками відправлення та скидання

Наприклад, визначимо на формі **кнопку відправлення** та **кнопку скидання**:




```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>Форми в HTML5</title>
    </head>
    <body>
        <form>
            <p><input type="text" name="login"/></p>
            <p><input type="password" name="password"/></p>
            <p>
                <button type="submit" formmethod="get" formaction="index.html">Надіслати</button>
                <button type="reset">Скасування</button>
            </p>
        </form>
    </body>
</html>
```

20

Таблиці

Для створення таблиць у HTML використовується елемент **table**.
Кожна таблиця між тегами <table> та </table> містить **рядки**, які представлені елементом **tr.** А кожен рядок між тегами <tr> та </tr> містить **осередки** у вигляді елементів **td**.

Створимо найпростішу таблицю:

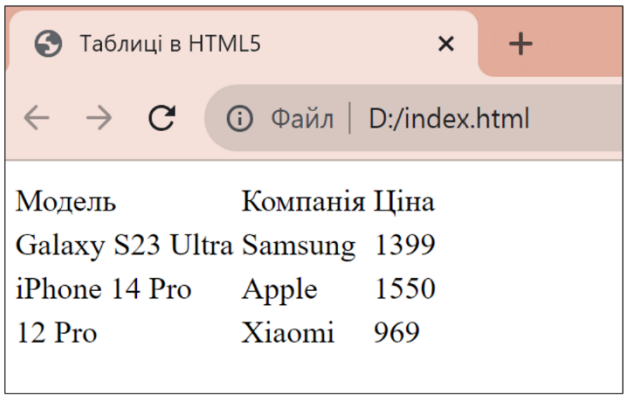
```
<!DOCTYPE html>
<html>
   <head>
      <meta charset="utf-8">
      <title>Таблиці в HTML5</title>
   </head>
   <body>
      <table>
         <tr>
            <td>Модель</td> <td>Компанія</td> <td>Ціна</td>
         </tr>
         <tr>
            <td>Galaxy S23 Ultra</td> <td>Samsung</td> <td>1399</td>
         </tr>
         <tr>
            <td>iPhone 14 Pro</td> <td>Apple</td> <td>1550</td>
         </tr>
         <tr>
            <td>12 Pro</td> <td>Xiaomi</td> <td>969</td>
         </tr>
      </table>
   </body>
</html>
```

21

Таблиці (2)

Тут у таблиці 4 рядки, і кожен рядок має три стовпці.

У цьому разі перший рядок виконує роль заголовка, інші три рядки власне є вмістом таблиці.



22

ТЕХНОЛОГІЇ РОЗРОБКИ КРОСПЛАТФОРМЕНИХ ВЕБ-ДОДАТКІВ

8. Введення в MS SQL Server

СУБД MS SQL Server

SQL Server є однією з найпопулярніших **систем управління базами даних** (СУБД). Ця СУБД підходить для різних проектів: від невеликих додатків до великих високонавантажених проектів.

SQL Server було створено компанією Microsoft (поточна версія 2022 року).

SQL Server довгий час був виключно системою керування базами даних для **Windows**, проте починаючи з версії 16 ця система доступна і на **Linux**.

SQL Server характеризується такими особливостями як:

- Продуктивність. SQL Server працює дуже швидко.
- Надійність та безпека. SQL Server надає шифрування даних.
- Простота.

З цієї СУБД щодо легко працювати та вести адміністрування.

Центральним аспектом у будь-якій СУБД є **база даних**.

База даних представляє **сховище даних**, організованих певним способом. Нерідко фізично база даних представляє файл на жорсткому диску, хоча така відповідність необов'язково.

Для зберігання та адміністрування баз даних застосовуються системи управління базами даних (database management system) або СУБД (DBMS).

2

Таблиці баз даних

Для організації баз даних MS SQL Server використовує **реляційну модель**. На сьогодні вона фактично є стандартом для організації баз даних.

Реляційна модель передбачає зберігання даних у вигляді **таблиць**, кожна з яких складається з рядків та стовпців.

Кожен **рядок** зберігає окремий **об'єкт**, а **стовпці** розміщуються **атрибути** цього об'єкта.

Для ідентифікації кожного рядка у межах таблиці застосовується **первинний ключ** (primary key). Як первинний ключ може виступати один або кілька стовпців. Використовуючи первинний ключ, можна посилатися на певний рядок у таблиці. Відповідно два рядки не можуть мати один і той же первинний ключ.

Через ключі одна таблиця може бути пов'язана з іншою, тобто між двома таблицями можуть бути організовані зв'язки. А сама таблиця може бути представлена у вигляді відношення ("relation").

Для взаємодії з базою даних використовується мова SQL (Structured Query Language). Клієнт (наприклад, зовнішня програма) надсилає **запит** мовою SQL за допомогою спеціального API. СУБД належним чином інтерпретує та виконує запит, а потім надсилає клієнту результат виконання.

Виробники СУБД використовують власні реалізації мови SQL, які трохи відрізняються один від одного.

Виділяються два різновиди мови SQL: PL-SQL та T-SQL. PL-SQL використовується у таких СУБД як **Oracle** та **MySQL.** **T-SQL** (Transact-SQL) застосовується у **SQL Server**.

3

Команди T-SQL

Залежно від завдання, яке виконує команда T-SQL, вона може належати до одного з таких типів:

• **DDL (Data Definition Language/Мова визначення даних**). До цього типу належать різні команди, які створюють базу даних, таблиці, індекси, процедури, що зберігаються і т.д. Загалом **визначають дані**.

Зокрема, до цього типу можна віднести такі команди:

- o **CREATE**: створює об'єкти бази даних (саму базу даних, таблиці, індекси тощо)
- o ALTER: змінює об'єкти бази даних
- o DROP: видаляє об'єкти бази даних
- o TRUNCATE: видаляє всі дані з таблиць

• **DML (Data Manipulation Language/Мова маніпуляції даними**). До цього типу відносять команди з вибору даних, їх оновлення, додавання, видалення - загалом усі команди, з допомогою яких можемо **управляти даними**. До цього типу належать такі команди:

- o **SELECT**: витягує дані з БД
- o **UPDATE**: оновлює дані
- o **INSERT**: додає нові дані
- o **DELETE**: видаляє дані

• **DCL (Data Control Language/Мова керування доступу до даних**). До цього типу відносять команди, які керують **правами доступу до даних**. Зокрема, це такі команди:

- o GRANT: надає права доступу до даних
- o REVOKE: відкликає права на доступ до даних

4

Встановлення MS SQL Server 2022 та SQL Server Management Studio

Встановлення MS SQL Server 2022

https://www.microsoft.com/nl-nl/sql-server/sql-server-downloads

Встановлення SQL Server Management Studio

Для зручного керування базами даних та різними опціями та налаштуваннями в MS SQL Server існує спеціальний засіб адміністрування, який називається **SQL Server Management Studio** (SSMS). Дану програму можна використовувати для створення баз даних та їх таблиць, написання та виконання запитів до баз даних, а також багато іншого.

https://learn.microsoft.com/en-us/sql/ssms/download-sql-server-management-studio-ssms?view=sql-server-ver16

Встановлення LocalDB

Крім повноцінного MS SQL Server у версіях Developer або **Express** існує полегшена легковажна версія SQL Server Express – движок LocalDB, який призначений спеціально для цілей розробки.

LocalDB може застосовуватися **для розробки** програм різними мовами програмування для **тестування їх роботи** з базою даних MS SQL Server, коли немає потреби у більшості можливостей стандартного MS SQL Server. І в цих умовах, звичайно, простіше встановити невеликий легковажний двигун, ніж повноцінний MS SQL Server.

Формально SQL LocalDB представляє компонент MS SQL Server Express, проте є кілька варіантів, як можна встановити LocalDB.

https://learn.microsoft.com/en-us/sql/database-engine/configure-windows/sql-server-express-localdb?view=sql-server-ver16

5

Бази даних

База даних представляє **сховище об'єктів**.

Основні з них:

• **Таблиці**: зберігають власне дані

• **Подання** (Views): вирази мови SQL, які повертають набір даних у вигляді таблиці

• **Процедури**, що зберігаються: виконують код на мові SQL по відношенню до даних до БД (наприклад, отримує дані або змінює їх)

• **Функції**: також код SQL, який виконує певне завдання

У SQL Server використовується два типи баз даних: системні та користувацькі.

Системні бази даних потрібні серверу SQL для коректної роботи.

А **бази даних користувача** створюються користувачами сервера і можуть зберігати будь-яку довільну інформацію. Їх можна змінювати та видаляти, створювати заново.

6

Створення таблиць

Ключовим об'єктом у базі даних є таблиці. Таблиці складаються з рядків та стовпців. Стовпці визначають тип інформації, що зберігається, а рядки містять значення цих стовпців.

Для створення бази даних використовується команда **CREATE DATABASE**.

CREATE DATABASE usersdb

Тим самим створюємо базу даних, яка називатиметься "usersdb".
Для створення таблиць використовується команда **CREATE TABLE**.
Наприклад, визначення найпростішої таблиці Customers:

```
CREATE TABLE Customers
(
   Id INT,
   Age INT,
   FirstName NVARCHAR(20),
   LastName NVARCHAR(20),
   Email VARCHAR(30),
   Phone VARCHAR(20)        )
```

7

Створення таблиць (2)

У цьому випадку таблиці Customers визначаються шість стовпців: Id, FirstName, LastName, Age, Email, Phone.

Перші два стовпці представляють ідентифікатор клієнта та його вік і мають тип INT, тобто зберігатимуть числові значення.

Наступні два стовпці представляють ім'я та прізвище клієнта і мають тип NVARCHAR(20), тобто представляють рядок UNICODE завдовжки трохи більше 20 символів.

Останні два стовпці Email і Phone представляють адресу електронної пошти та телефон клієнта і мають тип VARCHAR(30/20) - вони також зберігають рядок, але не кодування UNICODE.

За допомогою виразу **PRIMARY KEY** стовпець можна зробити первинним ключем.

```
CREATE  TABLE  Customers
(
   Id INT  PRIMARY  KEY,
   Age INT,
   FirstName NVARCHAR(20),
   LastName NVARCHAR(20),
   Email VARCHAR(30),
   Phone VARCHAR(20)      )
```

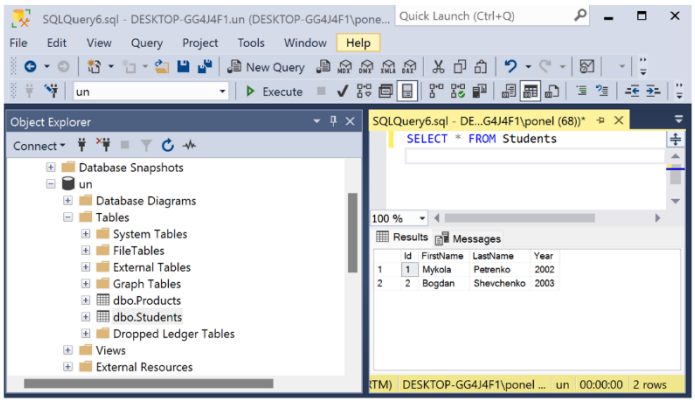
8

Приклад. Робота з таблицями у SQL Server Management Studio

Виконаємо запит до таблиці, зокрема отримаємо всі дані з неї.

База даних називається **un**, а таблиця – **dbo.Students**, тому для отримання даних з таблиці введемо наступний запит:

```
SELECT * FROM Students
```



Оператор SELECT дозволяє вибирати дані.

FROM вказує джерело, звідки брати дані.

Фактично цим запитом кажемо "ВИБРАТИ все з таблиці un.dbo.Students".

Для назви таблиці використовується повний її шлях із зазначенням бази даних та схеми.

9

Робота з даними таблиць. Команда INSERT

Для додавання даних застосовується команда INSERT, яка має такий формальний синтаксис:

INSERT [INTO] имя_таблицы [(список_стовпців)] **VALUES** (значення1, значення2, ... значення N)

Спочатку йде вираз **INSERT INTO**, потім у дужках можна вказати список стовпців через кому, в які треба додавати дані, і в кінці після слова VALUES дужках перераховують значення, що додаються для стовпців.

Наприклад, нехай раніше було створено таку базу даних:

```
CREATE  DATABASE  productsdb;
GO
USE productsdb;
CREATE  TABLE  Products
(   Id INT  IDENTITY PRIMARY  KEY,
   ProductName NVARCHAR(30) NOT  NULL,
   Manufacturer NVARCHAR(20) NOT  NULL,
   ProductCount INT  DEFAULT  0,
   Price MONEY NOT  NULL    )
```

Додамо до неї один рядок за допомогою команди **INSERT**:

```
INSERT Products VALUES ('iPhone 14', 'Apple', 5, 1520)
```

10

Команда INSERT

Варто враховувати, що значення стовпців у дужках після ключового слова **VALUES** передаються по **порядку їх оголошення**.

Наприклад, у виразі CREATE TABLE вище можна побачити, що першим стовпцем йде Id. Але оскільки йому заданий атрибут **IDENTITY**, то значення цього стовпця **автоматично генерується**, і його можна не вказувати.

Другий стовпець представляє ProductName, тому перше значення - рядок iPhone 14 буде передано саме цьому стовпцю.

Друге значення - рядок "Apple" буде передано третьому стовпцю Manufacturer і таке інше.

Тобто значення передаються стовпцям в такий спосіб:

- ProductName: 'iPhone 14’
- Manufacturer: 'Apple’
- ProductCount: 5
- Price: 1520

11

Команда SELECT

Команда SELECT використовується для отримання даних. У спрощеному вигляді він має такий синтаксис:

SELECT список_столбців FROM ім'я_таблиці

Наприклад, припустимо, що таблицю Products було створено раніше і до неї додано деякі вихідні дані:

```
CREATE  TABLE  Products
(  Id INT  IDENTITY PRIMARY  KEY,
   ProductName NVARCHAR(30) NOT  NULL,
   Manufacturer NVARCHAR(20) NOT  NULL,
   ProductCount INT  DEFAULT  0,
   Price MONEY NOT  NULL          );
INSERT  INTO  Products
VALUES
('iPhone 14', 'Apple', 3, 1500),
('iPhone 14 Pro', 'Apple', 2, 1560),
('iPhone 13', 'Apple', 5, 890),
('Galaxy 23', 'Samsung', 2, 1100),
('Galaxy 22', 'Samsung', 1, 800)
```

12

Команда SELECT (2)

Отримаємо всі об'єкти з цієї таблиці:

SELECT * FROM Products

Символ зірочка* вказує, що нам треба отримати **усі стовпці**.

Results Messages

	Id	ProductName	Manufacturer	ProductCount	Price
1	1	iPhone 14	Apple	3	1500,00
2	2	iPhone 14 Pro	Apple	2	1560,00
3	3	iPhone 13	Apple	5	890,00
4	4	Galaxy 23	Samsung	2	1100,00
5	5	Galaxy 22	Samsung	1	800,00

13

Оператор WHERE

Для **фільтрації** у команді SELECT застосовується оператор **WHERE**.
Після цього оператора ставиться умова, якій має відповідати рядок: **WHERE умова**

Якщо умова є **істинною**, то рядок потрапляє у результуючу **вибірку.** Як умову можна використовувати операції порівняння. Ці операції порівнюють два вирази.

У T-SQL можна застосовувати такі операції порівняння:

- =: порівняння на рівність (на відміну від сі-подібних мов у T-SQL для порівняння на рівність використовується один знак =)
- <>: порівняння на нерівність
- <: менше ніж
- >: більше ніж
- !<: не менше ніж
- !>: не більше ніж
- <=: менше ніж або дорівнює
- >=: більше ніж або дорівнює

Наприклад, знайдемо всі товари, виробником яких є компанія Samsung:

SELECT * FROM Products WHERE Manufacturer = 'Samsung'

14

Команди UPDATE та DELETE

Для зміни рядків у таблиці застосовується команда **UPDATE**.
Наприклад, збільшимо у всіх товарів ціну на 100:

```
UPDATE Products
SET Price = Price + 100
```

Використовуємо критерій і змінимо назву виробника з "Samsung" на "Samsung Inc.":

```
UPDATE Products
SET Manufacturer = 'Samsung Inc.'
WHERE Manufacturer = 'Samsung'
```

Для видалення застосовується команда **DELETE**:

DELETE [FROM] имя_таблицы WHERE умова_видалення

Наприклад, видалимо рядки, у яких id дорівнює 5:

```
DELETE Products
WHERE Id=5
```

Results Messages

	Id	ProductName	Manufacturer	ProductCount	Price
1	1	iPhone 14	Apple	3	1620,00
2	2	iPhone 14 Pro	Apple	2	1660,00
3	3	iPhone 13	Apple	5	990,00
4	4	Galaxy 23	Samsung Inc.	2	1200,00

15

Підписано до друку 26.03.2024. Формат 60x84/16.
Папір офсетний. Друк цифровий. Ум. друк. арк. 6,63.
Наклад 10 пр. Зам. № 52.

Видавництво та друкарня ПП «ЛІРА ЛТД»
Вул. Наукова, 5, м. Дніпро, 49107
Свідоцтво суб'єкта видавничої справи ДК № 6042 від
26.02.2018. dnipro.lira@gmail.com | +38 (067) 561-57-05 |
lira.dp.ua