Хмельницький національний університет

Вебтехнології

Методичні рекомендації до лабораторних робіт для здобувачів першого (бакалаврського) рівня вищої освіти спеціальності 121- Інженерія програмного забезпечення

*Затверджено на засіданні*

*кафедри інженерії програмного*

*забезпечення.*

*Протокол № 4 від 12.12.2023 р.*

Хмельницький 2024

Вебтехнології. Методичні рекомендації до лабораторних робіт для здобувачів першого (бакалаврського) рівня вищої освіти спеціальності 121 «Інженерія програмного забезпечення» / О. М. Яшина, В.В. Мартинюк – Хмельницький: ХНУ, 2024. – 67 с.

Укладачі: Яшина О. М. канд. техн. наук, доцент, В.В. Мартинюк докт. техн. наук, проф.

Відповідальний за випуск: Бедратюк Л. П., доктор фіз.-мат. наук, проф.

**ЗМІСТ**

С.

**Вступ** 4

**1 Лабораторна робота №1** 7

**2 Лабораторна робота №2** 8

**3 Лабораторна робота №3** 11

**4 Лабораторна робота №4**. 13

**5 Лабораторна робота №5** 23

**6 Лабораторна робота №6** 31

**7 Лабораторна робота №7** 34

**8. Лабораторна робота №8**

**Перелік використаних джерел** 37

**Додаток А.** Шаблон титульної сторінки звіту 39

**ВСТУП**

Дисципліна «Вебтехнології» відноситься до обов’язкових освітніх компонентів у підготовці здобувачів першого бакалаврського рівня за освітньо-професійними програмами спеціальності 121 – Інженерія програмного забезпечення.

Мета дисципліни – опанування студентами базових знань з сучасних вебтехнологій та основ вебпрограмування за допомогою технології ASP.NET Core, використовуючи MVC, Razor Page, Blazor, а також засвоєння теоретичних і практичних знань із питань перетворення, передавання та використання інформації в глобальному просторі, розкриття значення інформаційних процесів у формуванні сучасної системно-інформаційної схеми інформатизованого світу.

Завдання лабораторного практикуму з дисципліни полягає в наданні здобувачам знань та практичних навичок з розробки та реалізації вебпроектів із застосуванням сучасних мов та засобів програмування.

Згідно із освітньою програмою, вивчення здобувачем дисципліни у повному обсязі має розвивати сприяє поглибленню та розширенню загальних та **професійних компетентностей та програмних результатів навчання**, зокрема: здатність застосовувати знання у практичних ситуаціях, а також дотримуватися специфікацій, стандартів, правил і рекомендацій в професійній галузі при реалізації процесів життєвого циклу; здатність обґрунтовано обирати та освоювати інструментарій з розробки та супроводження програмного забезпечення; аналізувати, цілеспрямовано шукати і вибирати необхідні для вирішення професійних завдань інформаційно-довідникові ресурси і знання з урахуванням сучасних досягнень науки і техніки; уміння вибирати та використовувати відповідну задачі методологію створення програмного забезпечення; знати і застосовувати на практиці фундаментальні концепції, парадигми і основні принципи функціонування мовних, інструментальних і обчислювальних засобів інженерії програмного забезпечення; мотивовано обирати мови програмування та технології розробки для розв’язання завдань створення і супроводження програмного забезпечення.

Оскільки технологія ASP.NET Core та ASP.NET Core MVC розроблена корпорацією Microsoft, то основоположним матеріалом для формування даних методичних рекомендацій є офіційна документація щодо розробки програмного забезпечення Microsoft [15].

**Загальні методичні поради.** Лабораторні роботи розроблені таким чином, щоб здобувач міг працювати одночасно над курсовим проектом. Тобто на початку навчання під час першої лабораторної роботи із наданого переліку здійснюється вибір теми курсового проекту і відповідно до цієї теми розробляються лабораторні роботи, які є частиною курсового проєкту.

Під час підготовки до лабораторної роботи студент повинен ознайомитися з її змістом, вивчити лекційний матеріал та відповідні розділи у джерелах, зазначених у списку літератури.

Звіт з лабораторної роботи має містити: назву, мету роботи, умову завдання відповідно до варіанту (обраної теми курсового проєкту), лістинг коду програми, результат виконання програми, відповіді на теоретичні запитання. Приклад титульної сторінки звіту наведено в додатку А.

Звіти з лабораторних робіт надсилаються викладачеві на електронну пошту або інші засоби комунікації (модульне середовище, месенджери тощо) до початку заняття (окрім першого заняття). У разі відсутності звіту здобувач не допускається до захисту виконаної лабораторної роботи.

Звіт з лабораторної роботи складає кожен студент індивідуально. Роботи виконуються згідно графіку виконання лабораторних робіт. Кожна наступна робота виконується після здачі попередньої роботи (теоретичний допуск до роботи, виконання роботи, обробка результатів роботи програми, здача звіту та захист лабораторної роботи).

Робота вважається захищеною, якщо викладач поставив за неї певний бал у електронний журнал групи, що знаходиться у системі «Електронний університет» Хмельницького національного університету.

Для якісного виконання лабораторних робіт здобувачами аудиторії, де проходять лабораторні роботи мають бути забезпечені комп'ютерною технікою, що має доступ до мережі Інтернет, а також мати відповідне програмне забезпечення (середовище Visual Studio).

Допуск до лабораторного практикуму включає проведення інструктажу з техніки безпеки та охорони праці, про що здобувачі підтверджують своїми підписами у спеціальному журналі, який знаходиться у завідуючого лабораторіями.

Здійснення оцінювання лабораторних робіт відбувається за інституційною чотирибальною шкалою: «відмінно», «добре», «задовільно», «незадовільно» і шкалою ЄКТС (А, В, С, Д, Е). Оцінки за лабораторні роботи є складовими підсумкової семестрової оцінки здобувача.

Оцінку “відмінно” студент отримує у разі, коли він чітко, послідовно та у повному обсязі виконав лабораторну роботу (розробив відповідну програму у середовищі програмування, виконав усі необхідні завдання, оформив правильно звіт), а при захисті лабораторної роботи показав грамотний, логічний виклад матеріалу, вірні і повні відповіді на запитання викладача.

Оцінка “добре” виставляється студенту у випадку, коли він виконав лабораторну роботу у повному обсязі, з дотриманням вимог, а при захисті демонструє тверде знання матеріалу роботи, грамотно і по суті викла­дає його, однак його робота містить несуттєві неточності у відповідях на питання.

Оцінки “задовільно” заслуговує студент, який виконав лабораторну роботу згідно із завданням, але припустився неточностей при його виконанні; при захисті виявив знання основного навчального матеріалу в обсязі, необ­хідному для подальшої професійної діяльності.

Оцінка “незадовільно” виставляється у випадку, коли студент неякісно виконав лабораторну роботу, а при захисті показав безсистемні знання, не вміє виділяти головне і другорядне, припускається помилок у визначенні понять, перекручує їх зміст, хаотично і невпевнено викла­дає матеріал, не може використовувати знання при вирішенні практич­них завдань.

**Лабораторна робота №1**

**Створення простого застосунку в ASP.NET Core MVC.**

**Мета.** Створити простий застосунок на Core MVC, що спрямований дати деяке початкове розуміння роботи MVC. Як шаблон проекту використовується Empty і проектується застосунок із самого початку.

* 1. **Теоретичні основи та рекомендації до виконання**

В основі платформи знаходиться шаблон **MVC** та передбачає взаємодію трьох компонентів: контролера **(controller),** моделі **(model)** і представлення **(view).**

Контролер **(controller)** представляє клас, з якого власне і починається робота програми. Цей клас забезпечує зв'язок між моделлю і представленням. Отримуючи дані, що вводяться користувачем, контролер виходячи з внутрішньої логіки при необхідності звертається до моделі і генерує відповідне представлення.

Представлення (**view)** - це візуальна частина або інтерфейс користувача застосунка (наприклад, **html**-сторінка, через яку користувач, що зайшов на сайт, взаємодіє з вебзастосунком).

Модель **(model)** – це сукупність класів, що описують логіку використовуваних даних.

**ASP.NET** - технологія створення вебзастосунків та вебсервісів від компанії **Microsoft**. Вона є складовою частиною платформи **Microsoft.NET** і розвитком старішої технології **ASP. ASP .NET** заснована на платформі. NET і використовує всі нові можливості, що надаються цією платформою.

Microsoft повністю перебудувала **ASP.NET**, грунтуючись на **Common Language Runtime (CLR),** яка є основою всіх застосунків. NET. При створенні застосунків на **ASP.NET** можливо писати код, використовуючи різні мови програмування, які підтримуються в **.NET Framework**, як комерційні (Visual **Basic, Visual C#, Visual C++, Visual J #** та ін), так і «відкриті**» (Python, Perl** та ін).

Вебзастосунок - це набір взаємопов'язаних файлів (**HTML, ASP, ASPX**, файлів зображень і т.п.), а також пов'язаних з ними компонентів (двійкових файлів **NET** або класичної **COM**), які розміщені на вебсервері.

Вебсервер - програмний продукт, на якому розміщаються **Web**-застосунки та який зазвичай забезпечує набір зв'язаних з вебзастосунками служб, таких як інтегровані засоби забезпечення безпеки, підтримка протоколу FTP, підтримка засобів передачі електронної пошти т. п. вебсервер (вебсервіси) від Microsoft називається Internet Information Services (**IIS**). На одному **IIS** може знаходитися безліч вебзастосунків. Кожен з цих вебзастосунків повинен розміщуватися в своєму віртуальному каталозі (**Virtual directory**). Віртуальному каталогу на вебсервері відповідає фізичний каталог на диску, в цьому фізичному каталозі будуть знаходитися файли, з яких складається вказаний вебзастосунок.

Структура документа **HTML**. **HTML** - це стандартна мова гіпертекстової розмітки, яка використовується для опису того, як текст, зображення, гіперпосилання і стандартні елементи графічного інтерфейсу будуть відображатися в веббраузері. Більшість сучасних середовищ розробки вебзастосунків (у тому числі **Microsoft Visual Studio**) дозволяють створювати вебсторінки, майже не звертаючись безпосередньо до самого коду **HTML** за допомогою спеціальних інтегрованих засобів розробки інтерфейсу, проте, тим не менш, розробник вебзастосунків повинен, безумовно, знати цю мову. Документ **HTML** зазвичай починається з набору тегів, в яких міститься загальна інформація про документ (заголовок, метадані файлу і т. п. ), за якою слідує саме тіло документа (тобто набір тексту, зображень, таблиць, гіперпосилань і т. д.). Теги **HTML** нечутливі до регістру.

**1.2 Приклади виконання завдання**

Для створення проекту на **ASP.NET Core MVC** можна вибрати будь-який тип проекту на **ASP.NET Core** і вже додавати необхідні компоненти. Однак для спрощення **Visual Studio** вже за замовчуванням надає шаблон **ASP.NET Core Web App** **(Model-View-Controller):**

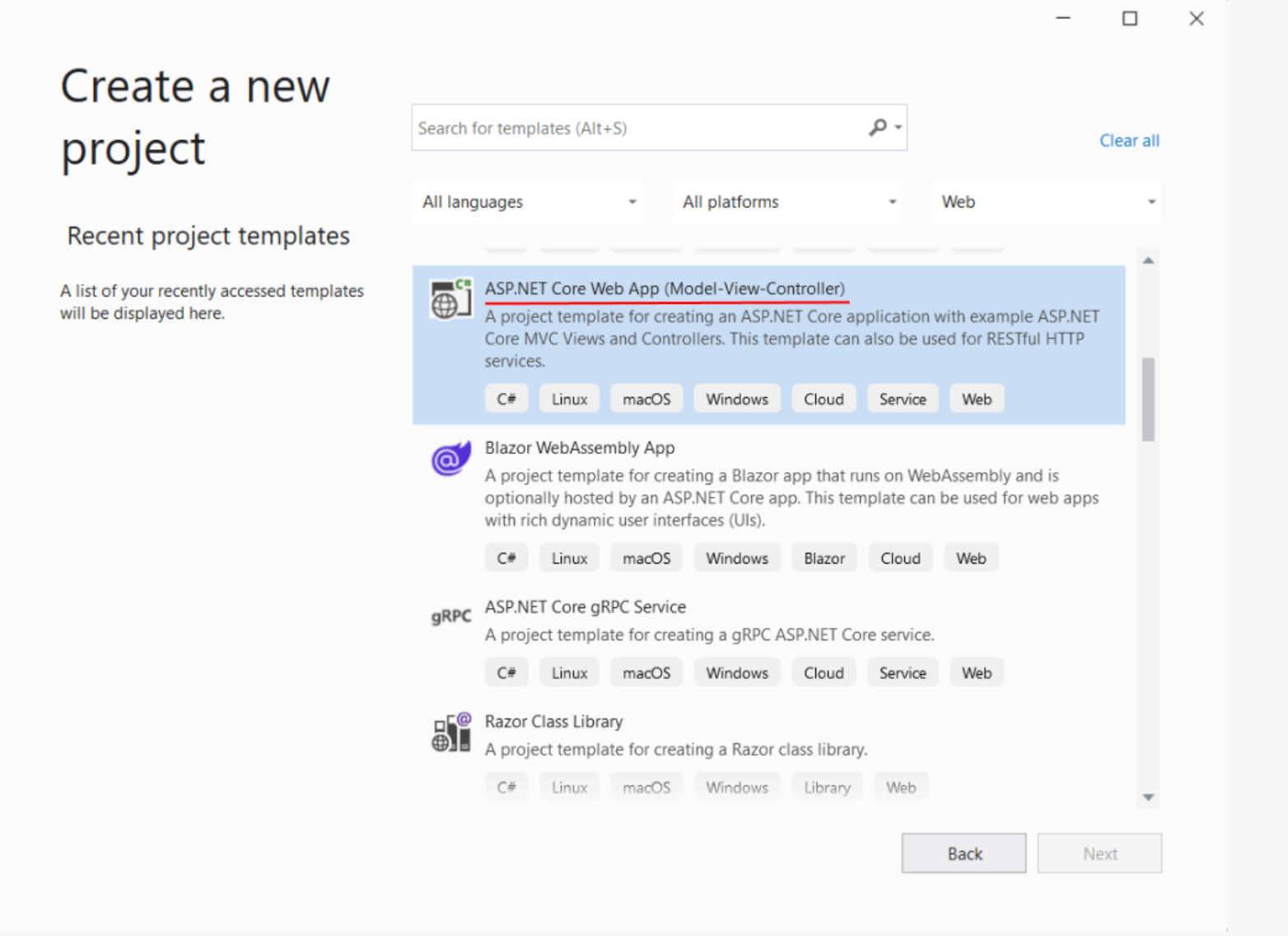


Рисунок 1.1 – Вибір шаблону проекту

Виберемо цей шаблон для створення проекту. Далі нам відкриється вікно для встановлення імені проекту. Припустимо, проект називатиметься **HelloMvcApp:**

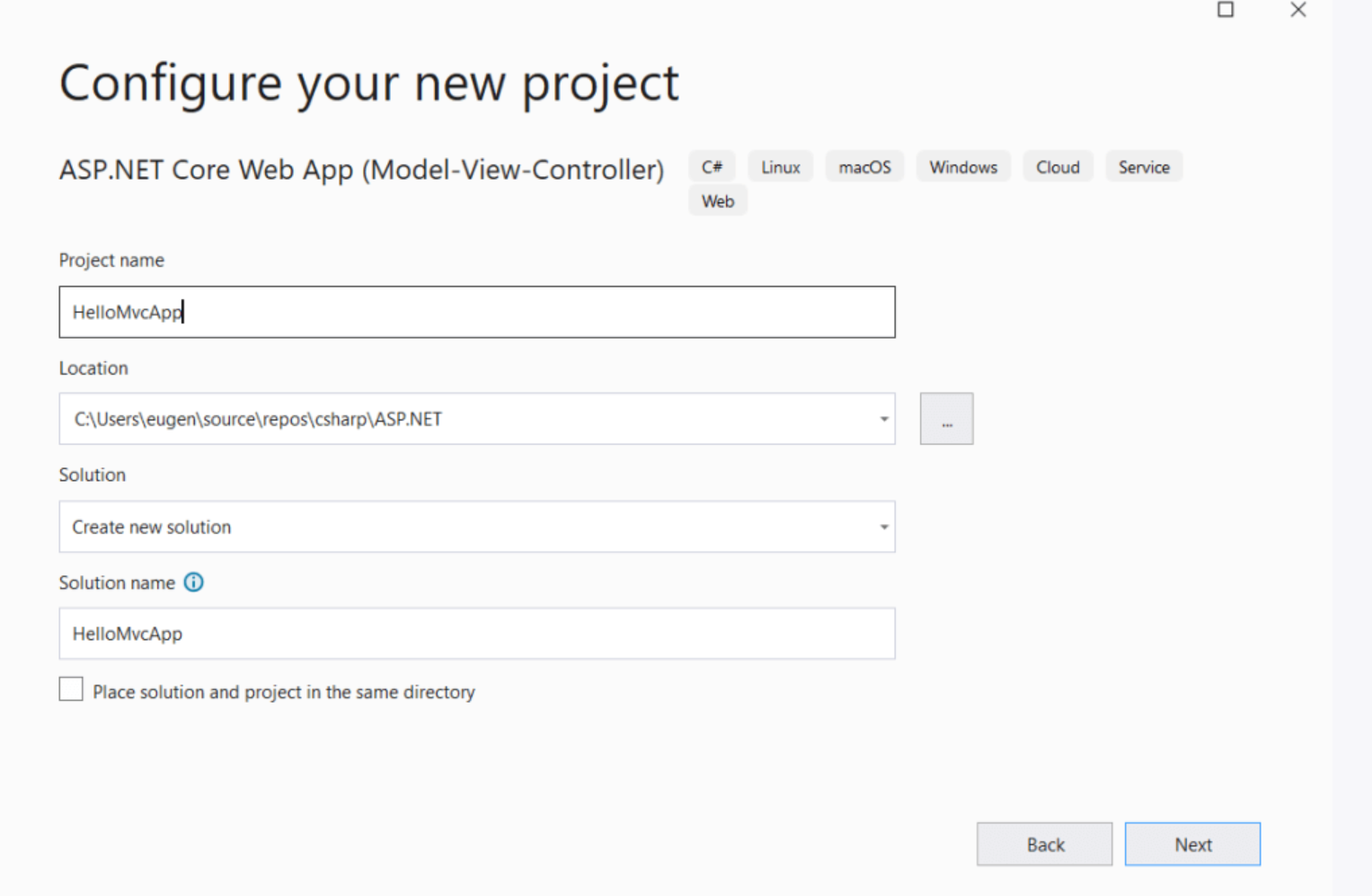


Рисунок 1.2 – Вибір стандартних налаштувань

Далі нам потрібно буде налаштувати стандартні налаштування для ASP.NET Core:

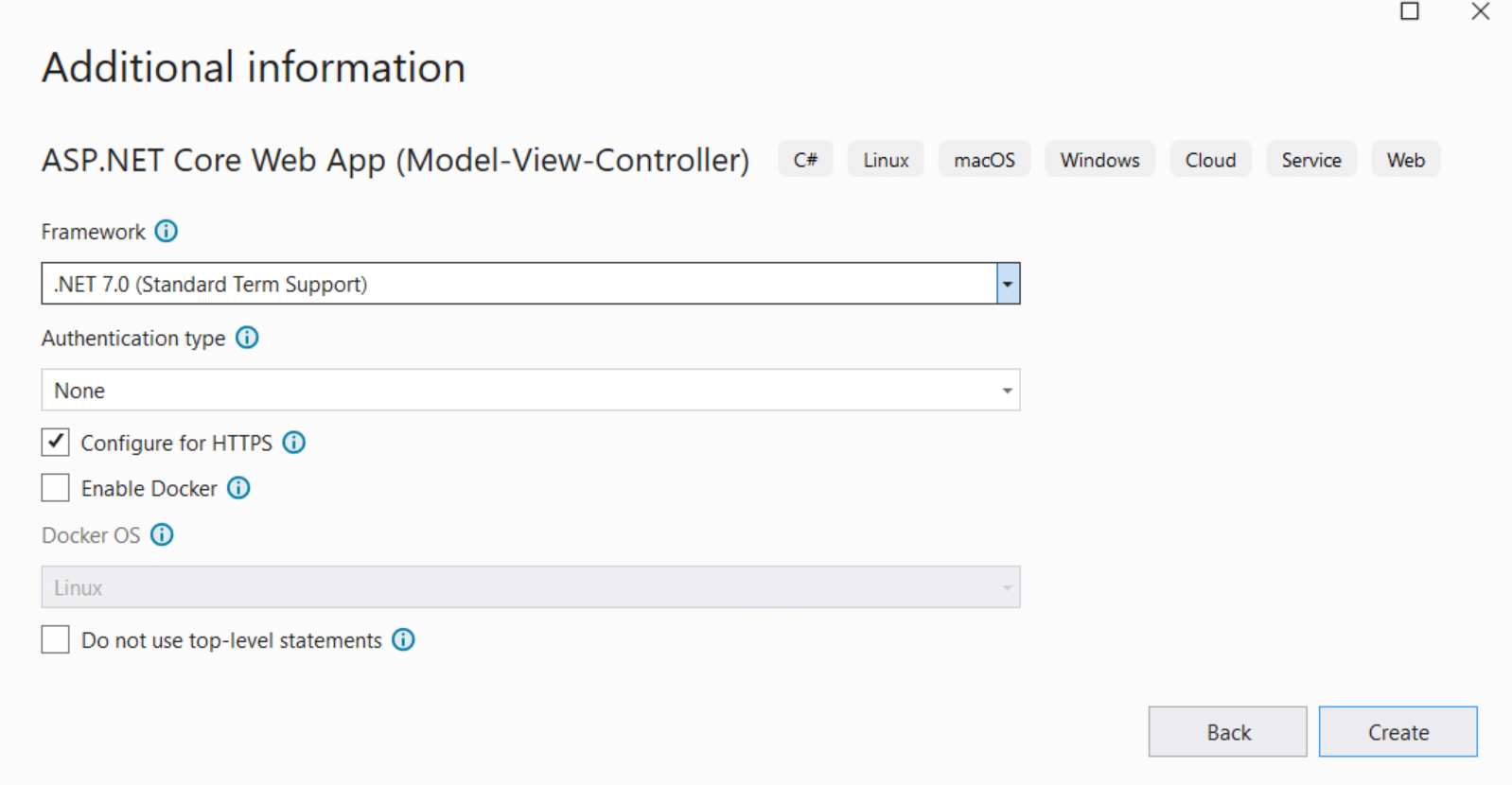


Рисунок 1.3 – Вибір необхідних параметрів

Залишимо всі параметри за замовчуванням і натисніть на **ОК**. І **Visual Studio** створить новий проект **MVC**.

Структура проекту буде аналогічна тій, що створює **.NET CLI:**

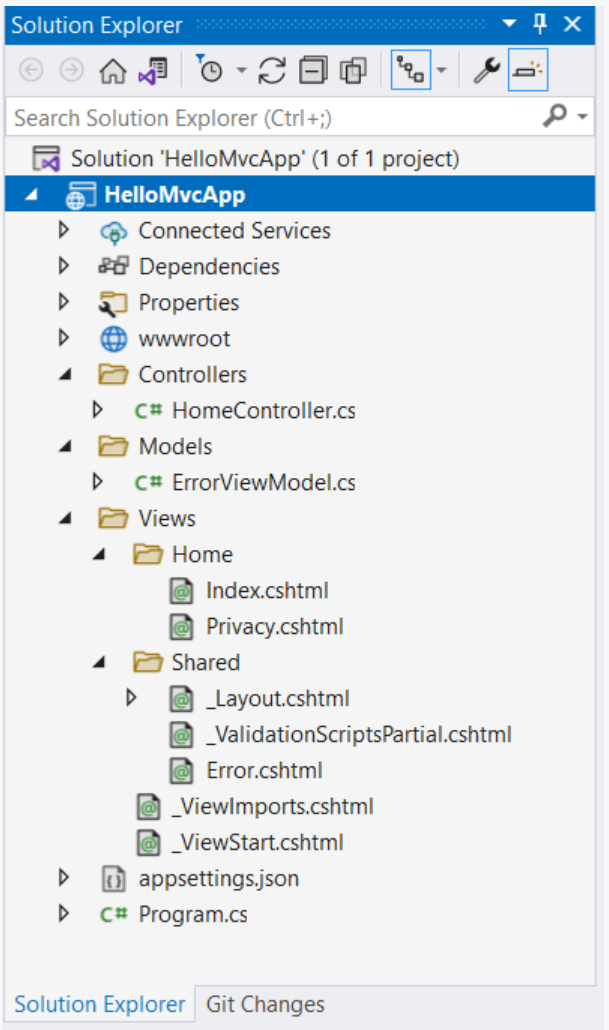


Рисунок 1.4 – Структура проекту

* **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 файлів і т.д., які використовуються застосунком.
* **Controllers**: папка для зберігання контролерів, які використовуються програмою. За замовчуванням тут вже є один контролер.
* **Models**: каталог для зберігання моделей. За замовчуванням створюється модель ErrorviewModel
* **Views**: каталог для зберігання представлень. Тут також за замовчуванням додаються ряд файлів - представлень
* **appsettings.json**: зберігає конфігурацію програми
* **Program.cs**: файл, який визначає вхідну точку в **ASP.NET Core**

Фактично ця та сама структура, що й у проекту типу **Empty** за винятком, що тут також додані за замовчуванням папки для ключових компонентів фреймворку **MVC**: контролерів і представлень. А також є додаткові вузли та файли для керування залежностями клієнтської частини програми.

І якщо ми запустимо проект на виконання, то спрацює запит до стандартного контролера - класу **HomeController**, який вибере для генерації відповіді потрібне представлення. І в результаті з представлення буде створена html-сторінка, яку ми побачимо у своєму веббраузері:

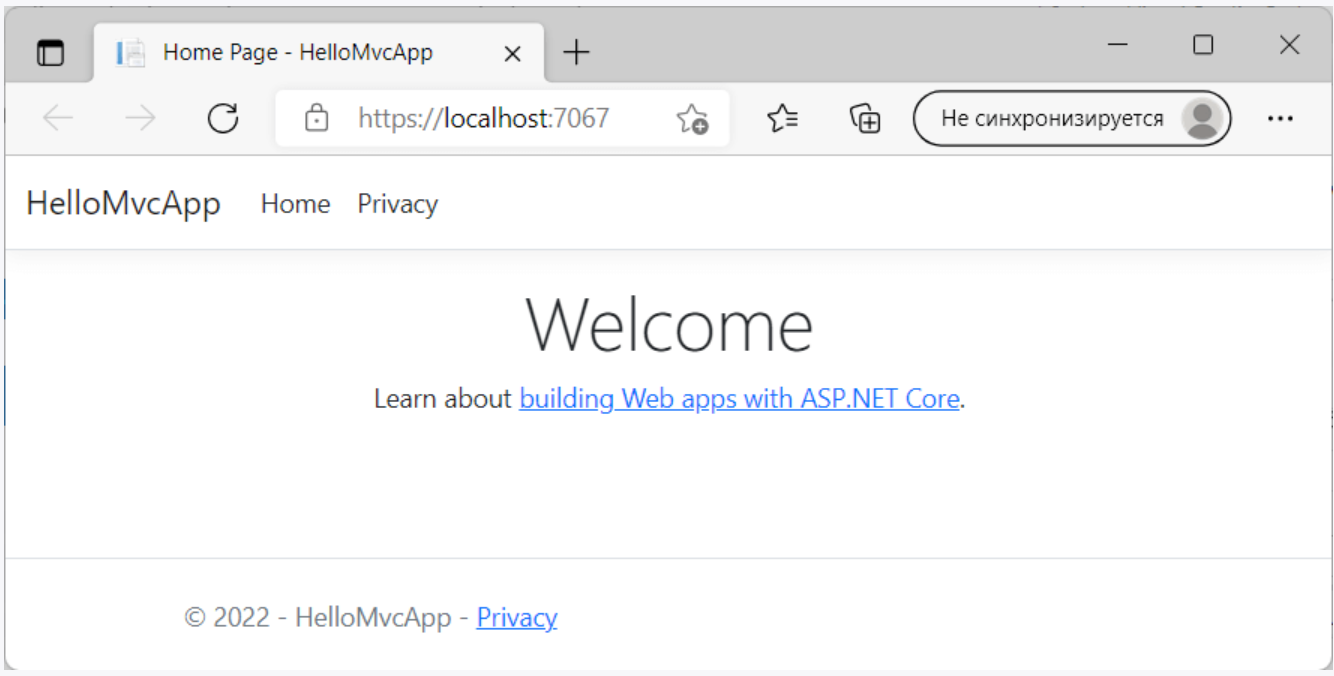


Рисунок 1.5 – Результат виконання проекту у браузері

**1.3 Завдання до лабораторної роботи**

Обрати із переліку запропонованих [4] тему курсового проекту. Розробити технічне завдання до проекту, а також створити простий застосунок в рамках першої лабораторної роботи із визначенням головної сторінки, на якій вказати дані про студента:

1. Прізвище та ім’я.
2. Номер групи та назва вузу.
3. Тему курсового проекту.
   1. **Контрольні запитання**
4. Що таке ASP.NET?
5. Що таке модель?
6. Що таке представлення?
7. Дати характеристику контролера.
8. З чого складається структура проекту?
9. Як здійснюється вибір шаблону проекту?
10. Що являє собою вебзастосунок?
11. Що таке вебсервер?

**Literature: [1, 5, 6, 14, 15]**

**Лабораторна робота №2**

**Додавання контролера, представлень та моделей до проєкту. Використання Create, Read, Update та Delete (CRUD)**

**Мета:** здійснити додавання необхідних контролерів, представлень та моделей до програми MVC ASP.NET Core.

**1.1 Теоретичні основи та рекомендації до виконання**

Архітектурна модель **MVC** розділяє застосунок на три основні компоненти: модель, представлення і контролер. За допомогою моделі **MVC** можна створювати застосунки, які зручніше тестувати та оновлювати порівняно з традиційними монолітними застосунками.

Застсоунки на основі моделі **MVC** включають наступне:

**Моделі (Models),** тобто класи, що представляють дані в застосунку. Класи моделі використовують логіку перевірки, яка дозволяє застосовувати бізнес-правила до цих даних. Зазвичай, об'єкти моделі витягують та зберігають стан моделі в базі даних. У подальшому прикладі модель **"Song"** витягує інформацію про пісні з бази даних та передає її в представлення або оновлює. Оновлені дані записуються в базу даних.

Представлення **(Views)** - це компоненти, які відображають користувацький інтерфейс застосунку. Зазвичай дані моделі відображаються в користувацькому інтерфейсі.

**Контролери -** класи, які:

* обробляють запити браузера;
* отримують дані моделі;
* викликають шаблони викликів представлення, які повертають відповідь.

У застосунку **MVC** представлення використовується лише для відображення інформації. Контролер обробляє та реагує на введення та взаємодію користувача. Наприклад, контролер обробляє сегменти URL-адреси та значення рядка запиту та передає ці значення моделі. Модель може використовувати ці значення для виконання запитів до бази даних.

Принципи цієї архітектури дозволяють ефективно впоратися з труднощами під час розробки застосунку, оскільки вони надають можливість працювати одночасно з одним аспектом реалізації, не торкаючись коду інших частин. Наприклад, можна змінювати код представлення незалежно від коду бізнес-логіки.

Ці принципи демонструються у прикладі, що поданий далі, застосунку для роботи з піснями. Проект **MVC** містить папки для контролерів та представлень.

* 1. **Приклади виконання завдання**

Далі буде розглянуто приклад застосунку для збереження та пошуку музичних творів.

Для того, щоб добавити контролер у застосунок необхідно виконати наступні дії.

В Solution Explorer натисніть правою кнопкою на **Controllers > Add > Controller** (рисунок 2.1).

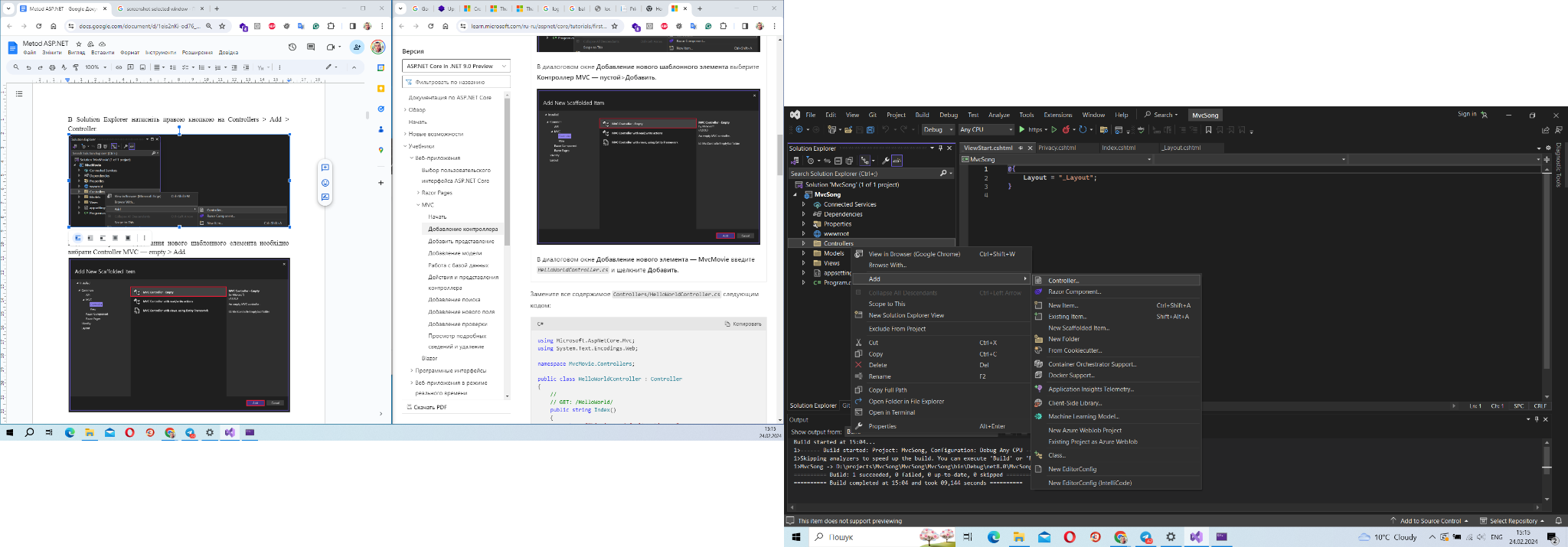


Рисунок 2.1 – Процедура добавлення контролера

В діалоговому вікні Додавання нового шаблонного елемента необхідно вибрати **Controller MVC - empty > Add.**

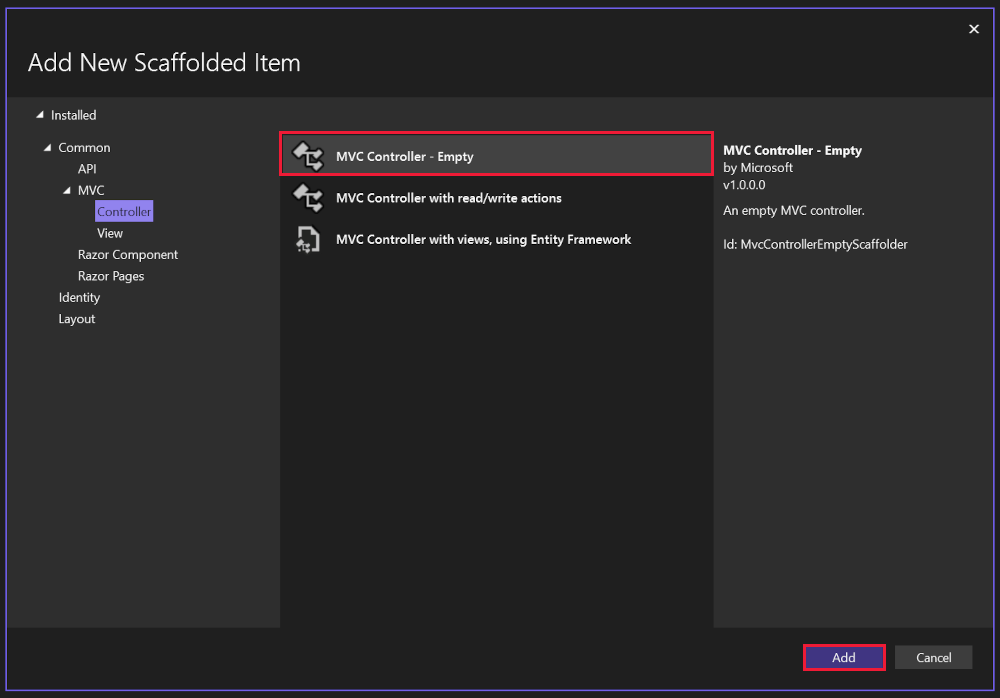
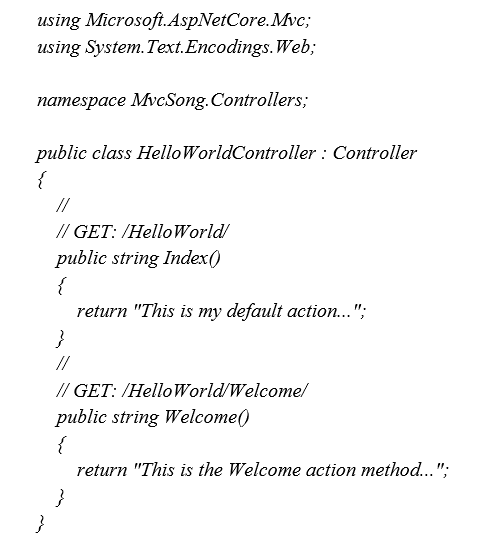


Рисунок 2.2 – Вибір порожнього шаблонного елемента

Далі новостворений файл необхідно заповнити наступним кодом:



Кожний публічний метод у контролері викликається як кінцева точка **HTTP**. У наведеному вище прикладі обидва методи повертають рядок. Зверніть увагу на коментарі перед кожним методом.

Кінцева точка HTTP:

Це URL-адрес, який є цільовим у вебзастосунку, наприклад, https://localhost:5001/HelloWorld.

Складається з:

* використовуваного протоколу: HTTPS;
* мережевого розташування вебсервера, включаючи порт TCP: localhost:5001;
* цільового універсального коду ресурсу (URI): HelloWorld.

У першому коментарі вказано, що цей метод HTTP GET викликається шляхом додавання /HelloWorld/ до базового **URL**-адреси.

У другому коментарі вказано, що цей метод **HTTP GET** викликається шляхом додавання **/HelloWorld/Welcome/** до **URL-**адреси. Далі використовується механізм формування шаблонів для створення методів **HTTP POST**, які оновлюють дані.

Додайте **/HelloWorld до URL запиту**. Метод Index поверне стрічку, що зображена на рисунку 2.3.

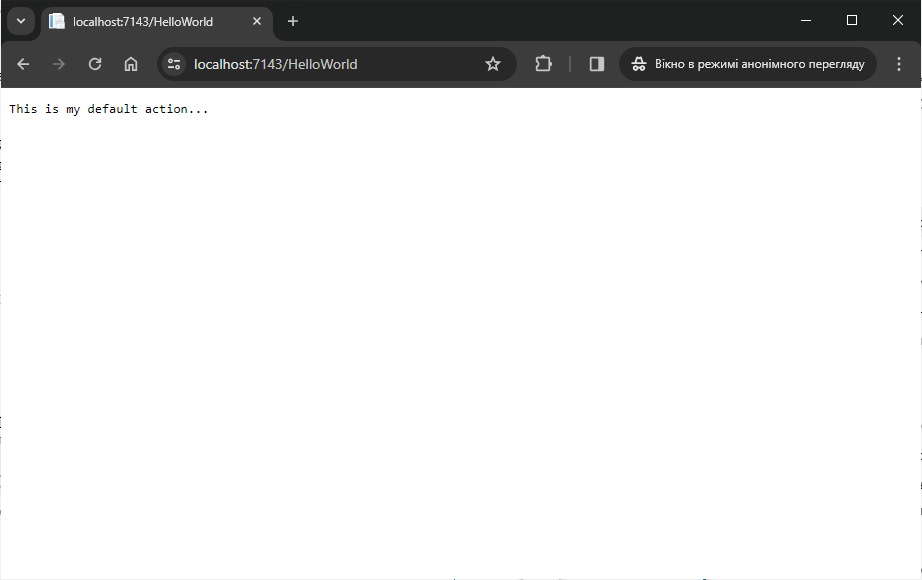


Рисунок 2.3 - URL запиту

**MVC** викликає класи контролера (і їхні методи дії) в залежності від вхідної **URL**-адреси. Логіка маршрутизації **URL**-адрес, яка використовується за замовчуванням у моделі **MVC**, визначає викликаний код на основі формату наступного вигляду:

**/[Controller]/[ActionName]/[Parameters]**

де:

**[Controller]** - назва контролера, який буде викликаний;

**[ActionName]** - назва методу дії в контролері, який буде викликаний;

**[Parameters]** - додаткові параметри, які можуть бути передані методу дії.

Формат маршрутизації заданий **у Program.cs** файлі (рисунок 2.4).

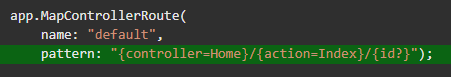


Рисунок 2.4 - Маршрутизація

Якщо при переході до застосунку не вказати сегменти **URL**-адреси, за замовчуванням встановлюються контролер **Home** та метод **Index**, які задані у вищезазначеному рядку шаблону. У попередніх сегментах **URL**-адрес:

* Перший сегмент URL-адреси визначає клас контролера, який буде виконуватися. Таким чином, **localhost:5001/HelloWorld** відповідає класу контролера **HelloWorld.**
* Друга частина сегмента URL-адреси визначає метод дії для класу. Таким чином, **localhost:5001/HelloWorld/Index** викликає метод **Index** класу **HelloWorldController.** Зверніть увагу, що у цьому випадку достатньо перейти за адресою **localhost:5001/HelloWorld,** і метод **Index** викликається за замовчуванням. Якщо ім'я методу, який викликається не вказано в запиті, для контролера викликається метод за замовчуванням - **Index**.
* У третій частині сегмента **URL**-адреси **(id)** вказуються дані маршруту.

Перейдіть за адресою **https://localhost:{PORT}/HelloWorld/Welcome. Замініть {PORT} на номер порту.**

Метод Welcome запускається і повертає рядок **"This is the Welcome action method..."**. Для цього **URL**-адреси вказані контролер **HelloWorld** та метод дії **Welcome**. Частина **URL**-адреси **[Parameters]** на даний момент ще не використовувалась.

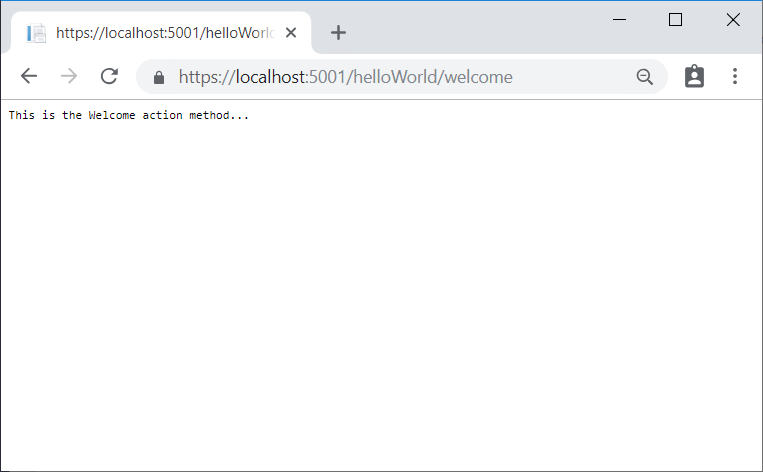


Рисунок 2.5 – Представлення із вказаним рядком **"This is the Welcome action method..."**

Змініть код, щоб передавати інформацію про параметри з **URL**-адреси в контролер. Наприклад, **/HelloWorld/Welcome?name=Rick&numtimes=4.**

Змініть метод **Welcome**, включивши два параметри, як показано в наступному коді.

*// GET: /HelloWorld/Welcome/*

*// Requires using System.Text.Encodings.Web;*

*public string Welcome(string name, int numTimes = 1)*

*{*

*return HtmlEncoder.Default.Encode($"Hello {name}, NumTimes is: {numTimes}");*

*}*

Попередній код:

* Використовує функцію необов'язкового параметра в **C#,** вказуючи, що параметр **numTimes** за замовчуванням приймає значення 1, якщо йому не передано значення.
* Використовує **HtmlEncoder.Default.Encode** для захисту застосунку від злонаміреного введення даних (наприклад, **JavaScript**).
* Використовує інтерпольовані рядки у вигляді **$"Hello {name},** **NumTimes is: {numTimes}".**

Запустіть застосунок та перейдіть за адресою **https://localhost:{PORT}/HelloWorld/Welcome?name=Rick&numtimes=4.** Замініть **{PORT}** на номер порта.

Використання різних значень для name та numtimes у **URL**-адресі (рисунок 2.6). Система прив'язки моделі **MVC** автоматично відповідає іменованими параметрами з рядка запиту з параметрами методу.

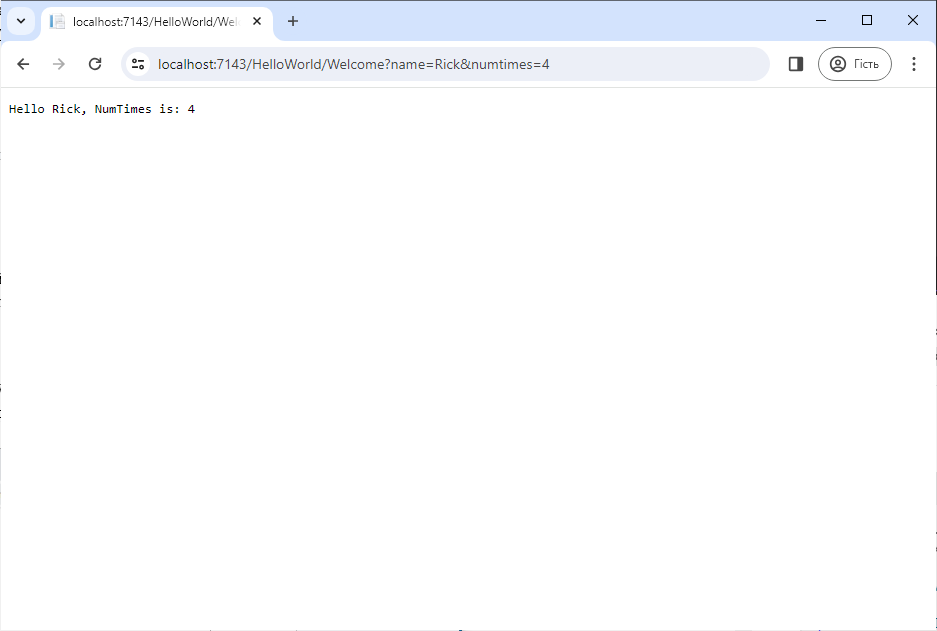


Рисунок 2.6 – Різні значення для name та numtimes у URL-адрес

**Додавання представлення до програми MVC ASP.NET Core**

Організовано інкапсулювати процес створення **HTML**-відповідей на клієнтському боці можна за допомогою файлів представлення **Razor**, шляхом зміни до класу **HelloWorldController** для використання.

Шаблони представлень створюються за допомогою **Razor**. Шаблони представлень на основі **Razor**:

* мають розширення файлу **.cshtml;**
* забезпечують зручний спосіб створення **HTML**-виведення за допомогою **C#.**

Зараз метод **Index** повертає рядок з повідомленням у класі контролера. У класі **HelloWorldController** необхідно замінити метод **Index** на наступний код:

*public IActionResult Index()*

*{*

*return View();*

*}*

Попередній код:

* викликає метод **View** контролера;
* використовує шаблон представлення для створення **HTML**-відповіді.

Методи контролера:

* називаються методами дій; наприклад, метод дії **Index** у попередньому коді;
* зазвичай повертають **IActionResult** або клас, який походить від **ActionResult**, а не простий тип, наприклад, string.

Щоб додати представлення необхідно зробити наступні дії:

Клацніть правою кнопкою миші на папку **Views**, а потім оберіть **Add> New** **Folder**. Назвіть папку **HelloWorld**.

Клацніть правою кнопкою миші на папку **Views/HelloWorld**, а потім оберіть **Add> New Item.**

У діалоговому вікні **"Add new item"** оберіть **"Show all templates".**

У вікні додавання нового елемента MvcSong виконайте наступні кроки.

* У полі пошуку в правому верхньому куті введіть view (представлення).
* Оберіть Представлення **Razor - empty**
* Збережіть значення поля **"Ім'я". Index.cshtml**
* Оберіть **Add**

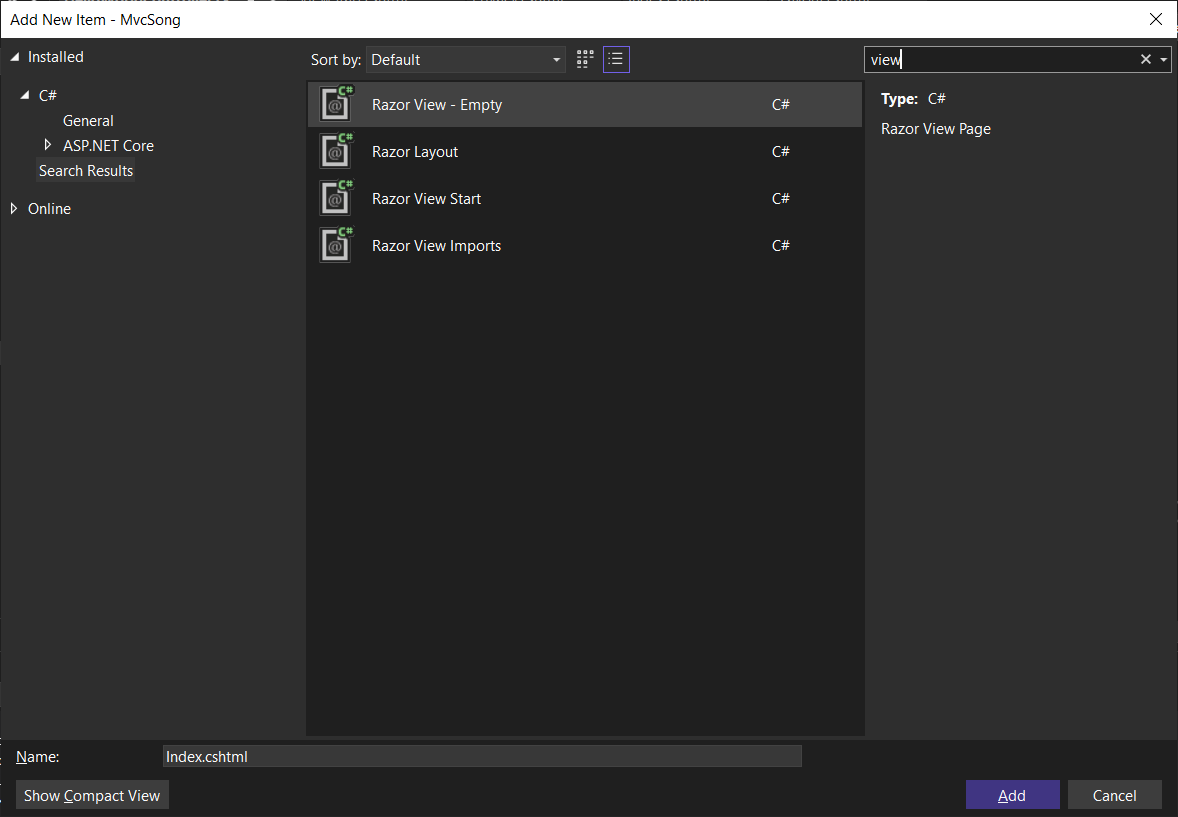


Рисунок 2.7 – Створення представлення

Замініть вміст файлу представлення **Views/HelloWorld/Index.cshtmlRazor** на наступний.

*@{*

*ViewData["Title"] = "Index";*

*}*

*<h2>Index</h2>*

*<p>Hello from our View Template!</p>*

Перейдіть за адресою **https://localhost:{PORT}/HelloWorld:**

* Метод **Index** у контролері **HelloWorldController** виконав оператор **return View();,** що вказує, що метод повинен використовувати файл шаблону представлення для відображення відповіді у браузері.
* Ім'я файлу шаблону представлення не вказано, за замовчуванням **MVC** використовує файл представлення за замовчуванням. Якщо ім'я файлу представлення не вказано, то відображається представлення за замовчуванням. У цьому прикладі представлення за замовчуванням має ту ж назву, що і метод дії **Index**. Використовується шаблон **/Views/HelloWorld/Index.cshtml** представлення.

На рисунку 2.8 показано рядок **"Hello from our View Template!",** який заданий у коді даного представлення.

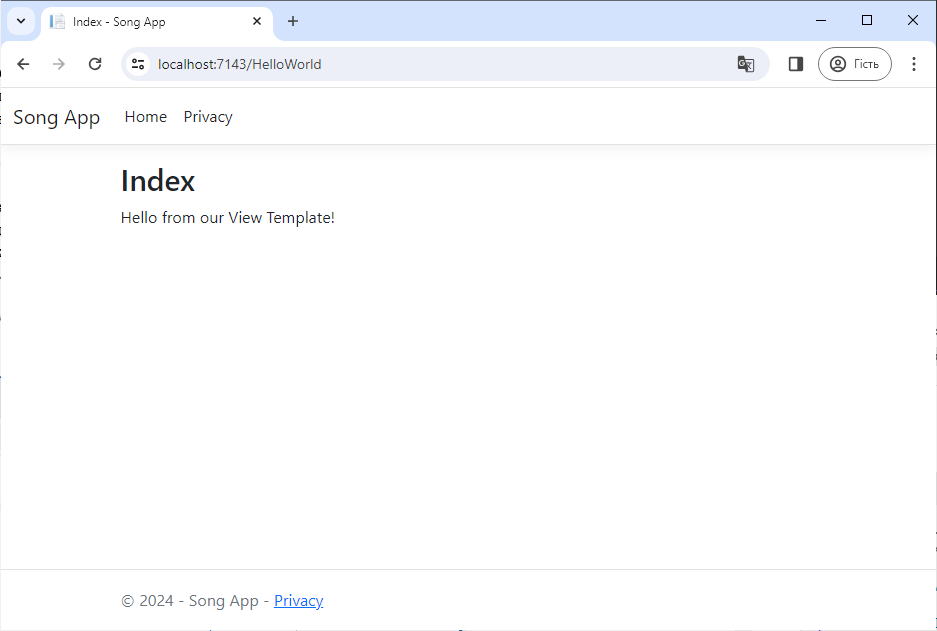


Рисунок 2.8 – Вигляд представлення

**Зміна представлень та сторінок макета**

Виберіть посилання в меню **(MvcSong, Home та Privacy).** Меню на кожній сторінці має однаковий макет. Макет меню реалізований у файлі **Views/Shared/\_Layout.cshtml.**

Відкрийте файл **Views/Shared/\_Layout.cshtml.**

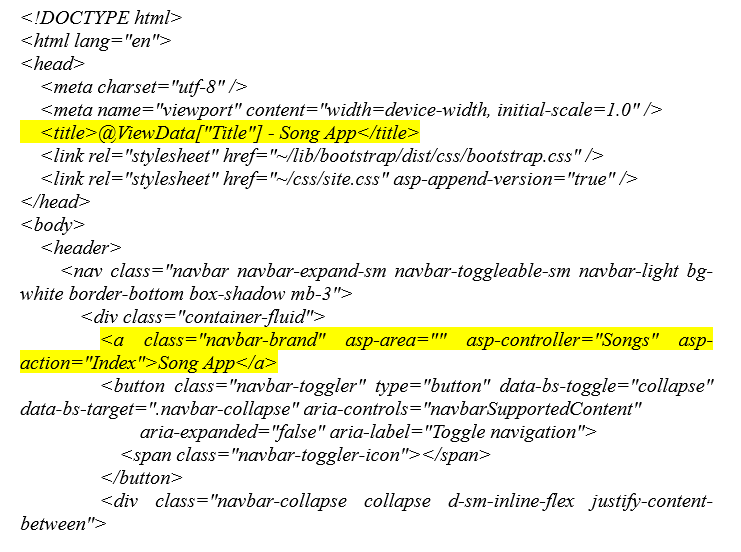
Шаблони макета дозволяють:

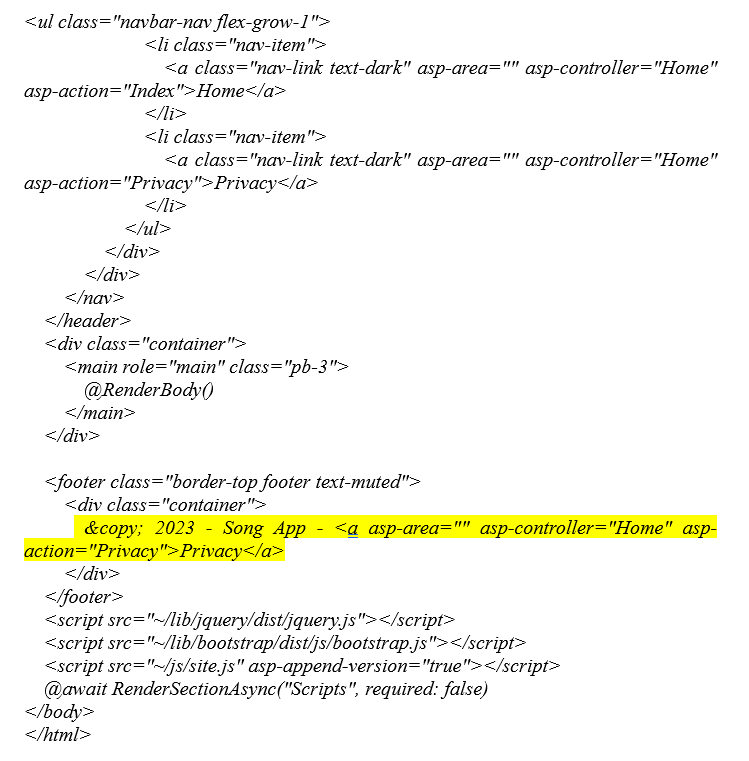
* вказати макет контейнера **HTML** сайту в одному місці;
* застосовувати макети контейнера **HTML** на декількох сторінках сайту.

Знайдіть рядок **@RenderBody(). RenderBody** - це заповнювач, в якому відображаються всі створювані сторінки для конкретних представлень, упаковані на сторінці макета. Наприклад, якщо клікнути посилання **Privacy**, представлення **Views/Home/Privacy.cshtml** відобразиться у методі **RenderBody**.

Зміна заголовка, нижнього колонтитула та посилання меню у файлі макету.

Змініть вміст файлу **Views/Shared/\_Layout.cshtml** наступним чином. Зміни виділені кольором:

**

**

Наведена вище розмітка вносить наступні зміни:

* заміна трьох входжень **MvcSong** на **Song App;**
* заміна елемента прив'язки **<a class="navbar-brand" asp-area="" asp-controller="Home" asp-action="Index">MvcSong</a>** на **<a class="navbar-brand" asp-controller="Songs" asp-action="Index">Song App</a>.**

У вище наведеній розмітці атрибут додаткової функції тега **прив'язки asp-area=""** та значення атрибута не були використані, оскільки цей застосунок не використовує області **(Areas).**

Примітка. Контролер **Songs** не був реалізований. На цьому етапі посилання **Song App** не працює.

Збережіть зміни та клацніть посилання **Privacy**.

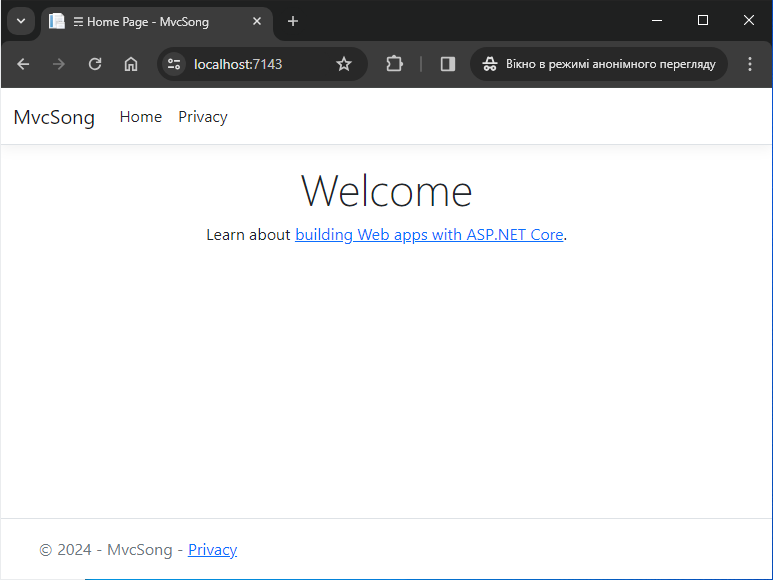


Рисунок 2.9 – Вигляд представлення зі змінами

Виберіть посилання **Home**.

Зверніть увагу, що в заголовку та тексті прив'язки також відображається **Song App.** Зробивши одну зміну в шаблоні макету, ми змінили заголовок та текст посилання на всіх сторінках сайту.

Перегляньте файл **Views/\_ViewStart.cshtml.**

*@{*

*Layout = "\_Layout";*

*}*

Файл **Views/\_ViewStart.cshtml** автоматично включається у кожне представлення **Views/Shared/\_Layout.cshtml.** Властивість **Layout** може встановлювати інше представлення макету або мати значення null, коли макет не використовується.

Відкрийте файл представлення **Views/HelloWorld/Index.cshtml.**

Змініть заголовок та елемент **<h2>,** як вказано нижче:

*@{*

*ViewData["Title"] = "Song List";*

*}*

*<h2>My Song List</h2>*

*<p>Hello from our View Template!</p>*

Заголовок та елемент **<h2>** трошки відрізняються, щоб ви могли побачити, який саме фрагмент коду змінює відображення.

**ViewData["Title"] = "Song List";** в наведеному вище коді присвоює значення "Song List" властивості Title словника **ViewData**. Властивість **Title** використовується у тегу **HTML <title>** на сторінці макету: **<title>@ViewData["Title"] - Song App</title>**

Збережіть зміни та перейдіть за посиланням **https://localhost:{PORT}/HelloWorld.**

Були змінені наступні елементи:

* Заголовок браузера.
* Основний заголовок.
* Додаткові заголовки.

Якщо зміни не відображаються у браузері, це може вказувати на кешування вмісту. У цьому випадку натискайте **CTRL+F5** у браузері для примусового завантаження відповіді від **сервера. Заголовок браузера створюється за допомогою атрибута ViewData["Title"],** який встановлюється у шаблоні представлення **Index.cshtml** та додаткового рядка **"- Song App",** доданого у файл макету.

Вміст шаблону **Index.cshtml** об'єднується з шаблоном **Views/Shared/\_Layout.cshtml.** У браузер відправляється одна **HTML**-відповідь. Шаблони макетів дозволяють легко вносити зміни, які застосовуються до всіх сторінок в застосунку.

**Передача даних з контролера в представлення**

Дії контролера викликаються у відповідь на вхідний запит **URL**-адреси. Код, який обробляє вхідні запити від браузера, додається до класу контролера. Контролер витягує дані з джерела даних та визначає тип відповіді, який буде відправлений у браузер. Контролер може використовувати шаблони представлень для створення та форматування відповіді **HTML**, що відправляється у браузер.

Контролер відповідає за надання даних, необхідних шаблону представлення для відображення відповіді.

Шаблони представлень не повинні:

* виконувати бізнес-логіку;
* безпосередньо взаємодіяти з базою даних.

Замість цього вони повинні працювати лише з даними, які контролер надає їм. Таке розподілення відповідальностей дозволяє забезпечити максимальну оптимізацію коду, а також його:

* очищення;
* тестування;
* обслуговування.

На даний момент метод **Welcome** у класі **HelloWorldController** приймає параметри **name** та **ID**, після чого виводить значення безпосередньо у браузер.

Замість відображення відповіді у вигляді рядка налаштуйте контролер для використання шаблону представлення. Шаблон представлення створює динамічну відповідь, для отримання якої потрібно передати відповідні дані з контролера у представлення. Для цього контролер може помістити динамічні дані (параметри), які потрібні шаблону представлення, в словник **ViewData**, до якого потім цей шаблон буде звертатися за динамічними даними.

У **HelloWorldController.cs** змініть метод **Welcome**, щоб додати значення **Message** та **NumTimes** у словник **ViewData**.

Словник **ViewData** представляє собою динамічний об'єкт, і це означає, що можна використовувати будь-який тип. Об'єкт **ViewData** не має визначених властивостей, доки не буде додано який-небудь елемент. Система прив'язки моделі **MVC** автоматично відповідає іменованим параметрам **name** та **numTimes** із рядка запиту з параметрами метода. Повний код **HelloWorldController**:

*using Microsoft.AspNetCore.Mvc;*

*using System.Text.Encodings.Web;*

*namespace MvcSong.Controllers;*

*public class HelloWorldController : Controller*

*{*

*public IActionResult Index()*

*{*

*return View();*

*}*

*public IActionResult Welcome(string name, int numTimes = 1)*

*{*

*ViewData["Message"] = "Hello " + name;*

*ViewData["NumTimes"] = numTimes;*

*return View();*

*}*

*}*

Об'єкт словника **ViewData** містить дані, які будуть передаватися в представлення.

Створіть шаблон привітання з ім'ям **Views/HelloWorld/Welcome.cshtml.**

У шаблоні представлення **Welcome.cshtml** створіть цикл, який відображатиме рядок **"Hello" NumTimes**. Замініть весь вміст **Views/HelloWorld/Welcome.cshtml** на наступний:

*@{*

*ViewData["Title"] = "Welcome";*

*}*

*<h2>Welcome</h2>*

*<ul>*

*@for (int i = 0; i < (int)ViewData["NumTimes"]!; i++)*

*{*

*<li>@ViewData["Message"]</li>*

*}*

*</ul>*

Збережіть зміни та перейдіть за наступним **URL**-адресою: **https://localhost:{PORT}/HelloWorld/Welcome?name=Rick&numtimes=4**

Дані витягуються з **URL**-адреси та передаються в контролер за допомогою засобу прив'язки моделі **MVC**. Контролер упаковує дані у словник **ViewData** та передає цей об'єкт у представлення. Представлення відображає дані у форматі **HTML** у браузері (рисунок 2.10).

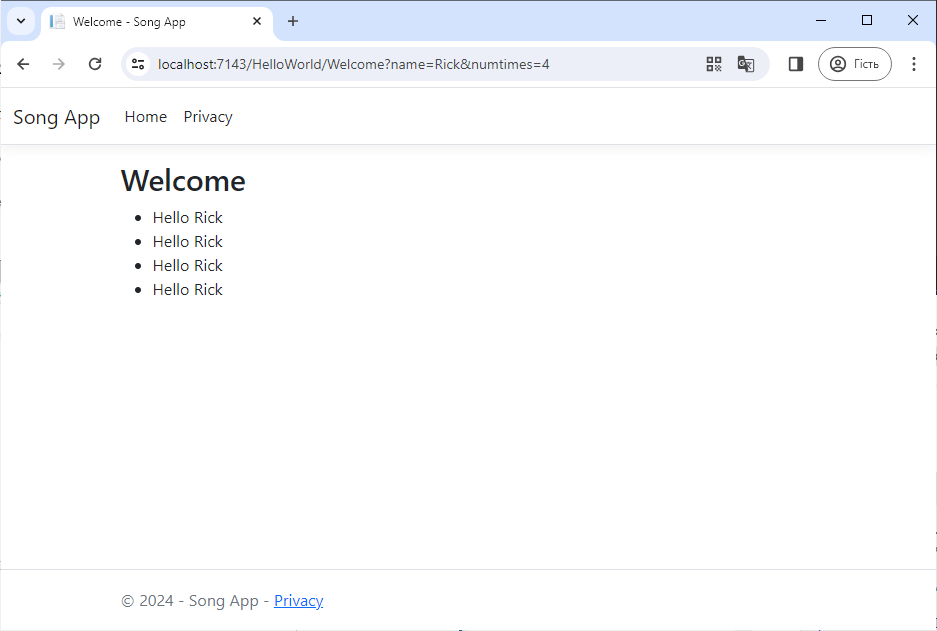


Рисунок 2.10 – Відображення даних у форматі HTML у браузері

**Додавання моделі до застосунку ASP.NET Core MVC**

Клацніть правою кнопкою миші на папку **Models** і оберіть **Add > Class.** Дайте файлу назву **Song.cs.**

**Models/Song.cs** Оновіть файл наступним кодом:

*using System.ComponentModel.DataAnnotations;*

*namespace MvcSong.Models;*

*public class Song*

*{*

*public int Id { get; set; }*

*public string Title { get; set; }*

*public string Author { get; set; }*

*[DataType(DataType.Date)]*

*public DateTime? ReleaseDate { get; set; }*

*public string? Genre { get; set; }*

*public int Duration { get; set; }*

*}*

Клас **Song** містить поле Id, яке база даних потребує як первинний ключ. Атрибут **DataType** для **ReleaseDate** вказує тип даних (**Date**). З цим атрибутом:

* Користувачеві не потрібно вводити відомості про час у поле дати.
* Відображається лише дата, а не час.

Знак питання після string вказує, що властивість допускає значення **NULL**.

**Створення шаблону сторінок для пісень**

Використовуйте інструмент шаблону для створення сторінок **Create, Read, Update** та **Delete (CRUD)** для моделі застосунку пісень.

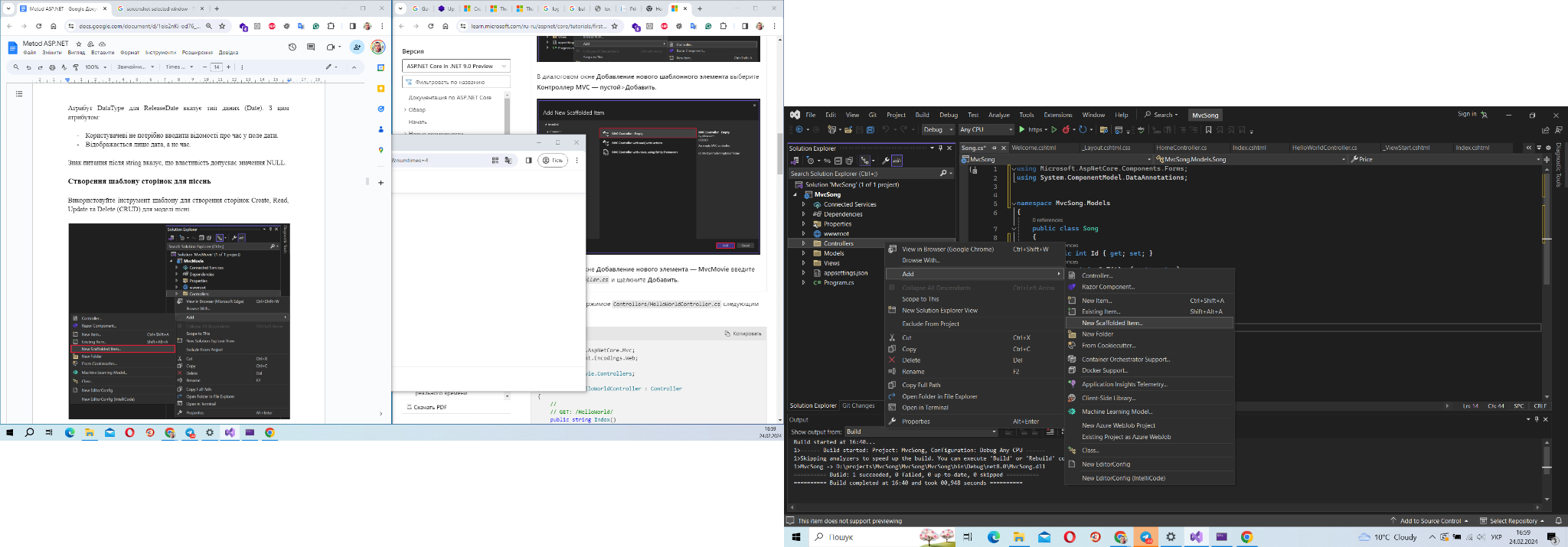


Рисунок 2.11 – Використання CRUD

В діалоговому вікні додавання нового шаблону елемента (рисунок 2.12):

* У лівій області оберіть **Installed > Common > MVC.**
* Оберіть контролер **MVC** з представленнями за допомогою **Entity Framework.**
* Оберіть **Add**.

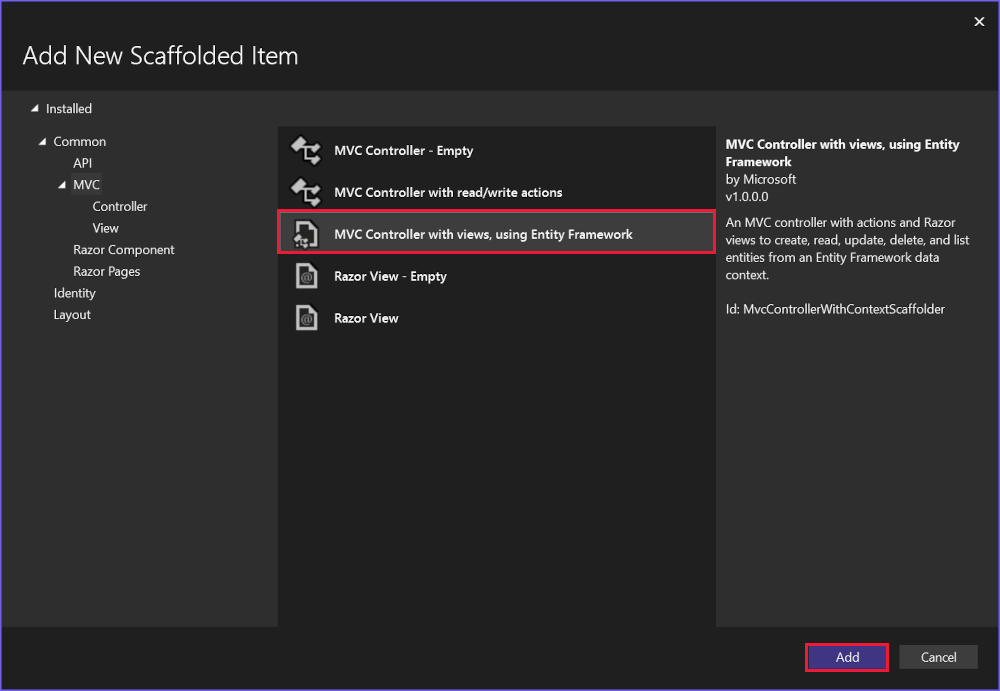


Рисунок 2.12 – Додавання нового шаблону елемента

У діалоговому вікні додавання контролера **MVC** з представленнями за допомогою Entity Framework виконайте наступне (рисунок 2.13):

* У списку **Клас моделі** оберіть **Пісня** (MvcSong.Models).
* У рядку **Клас** контексту даних клацніть знак плюса (+).
* У діалоговому вікні Додавання контексту даних створюється ім'я класу **MvcSong.Data.PagesSongContext.**
* Оберіть **Додати**.
* У списку **Постачальник бази даних** оберіть **SQL Server**.
* Параметри **Представлення** та **Ім'я контролера**: залиште значення за замовчуванням.
* Оберіть Додати.

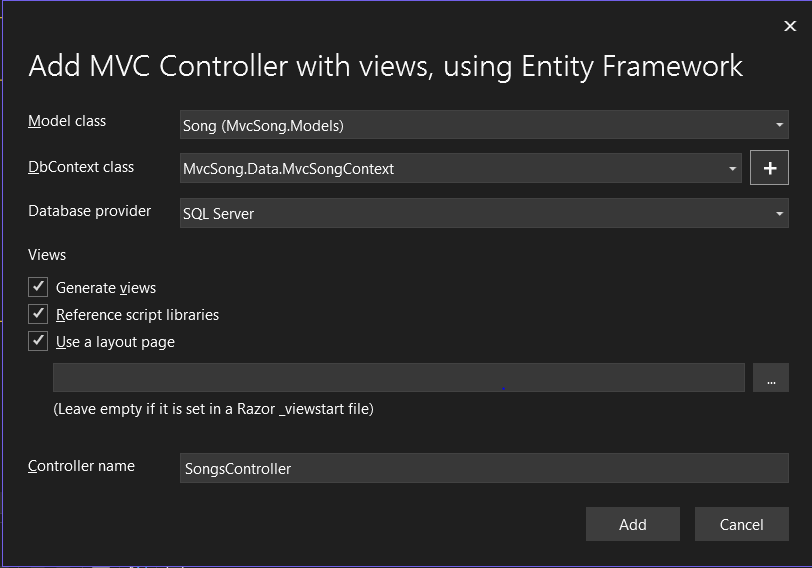


Рисунок 2.13 – Додавання шаблону моделі

Якщо виникне повідомлення про помилку, оберіть **"Додати ще раз",** щоб повторити спробу.

Шаблон додає наступні пакети:

* **Microsoft.EntityFrameworkCore.SqlServer**
* **Microsoft.EntityFrameworkCore.Tools**
* **Microsoft.VisualStudio.Web.CodeGeneration.Design**

При формуванні шаблонів створюються такі об'єкти:

* Контролер пісень: **Controllers/SongsController.cs**
* Razor-файли представлень для сторінок **Create, Delete, Details, Edit** та **Index: Views/Songs/\*.cshtml**
* Клас контексту для бази даних: **Data/MvcSongContext.cs**

Під час формування шаблонів виконуються такі дії:

* Вставляються необхідні посилання на пакети **Microsoft.EntityFrameworkCore.SqlServer** та **Microsoft.EntityFrameworkCore.Tools** в файл проекту **MvcSong.csproj.**
* Реєструється контекст бази даних у файлі **Program.cs.**
* Додається рядок підключення до бази даних у файл **appsettings.json**.

Автоматичне створення цих файлів і їх оновлення називається формуванням шаблонів.

Згенеровані сторінки наразі не можна використовувати, оскільки база даних ще не існує. Запустіть застосунок, щоб переконатися, що помилок немає.

**Перша міграція**

**Core** використовує механізм міграцій для створення та оновлення бази даних відповідно до моделі даних. Міграції представляють собою інструмент, який дозволяє автоматизовано керувати структурою бази даних на основі змін у моделі даних.

У меню Сервіс послідовно оберіть пункти **NuGet packet manager > package manager console.**

Введіть в консолі диспетчера пакетів (PMC) наступні команди:

*Add-Migration InitialCreate*

*Update-Database*

**Add-Migration InitialCreate:** створює файл міграції **Migrations/{timestamp}\_InitialCreate.cs**. Аргумент **InitialCreate** - це ім'я міграції. Можна використовувати будь-яке ім'я, але за звичаєм обирається ім'я, яке описує міграцію. Оскільки це перша міграція, створений клас містить код для створення схеми бази даних. Схема бази даних створюється на основі моделі, вказаної в класі MvcSongContext.

**Update-Database:** Оновлює базу даних до останньої міграції, створеної попередньою командою. Ця команда запускає метод **Up** у файлі **Migrations/{time-stamp}\_InitialCreate.cs**, який створює базу даних.

Якщо виникає помилка “SqlException: Cannot open database **"MvcSongContext-1" requested by the login. The login failed”,** то потрібно провести операцію **Update-Database.**

**Розгляд створеного класу контексту бази даних та реєстрації**

При цьому доступ до даних у **Core** здійснюється за допомогою моделі. Модель складається з класів сутностей та об'єкта контексту, який представляє сесію взаємодії з базою даних. Об'єкт контексту дозволяє виконувати запити та зберігати дані. Контекст бази даних успадковується від **Microsoft.EntityFrameworkCore.DbContext** і визначає сутності, які слід включити в модель даних.

Під час створення шаблонів генерується клас контексту бази даних **Data/MvcSongContext.cs:**

*using System;*

*using System.Collections.Generic;*

*using System.Linq;*

*using System.Threading.Tasks;*

*using Microsoft.EntityFrameworkCore;*

*using MvcSong.Models;*

*namespace MvcSong.Data*

*{*

*public class MvcSongContext : DbContext*

*{*

*public MvcSongContext (DbContextOptions<MvcSongContext> options)*

*: base(options)*

*{*

*}*

*public DbSet<MvcSong.Models.Song> Song { get; set; }*

*}*

*}*

Наведений вище код створює властивість **DbSet<Song>,** що представляє пісні в базі даних.

**Dependency Injection - внесення залежностей**

**ASP.NET Core** підтримує внесення залежностей. Сервіси, такі як контекст бази даних, реєструються в застосунку (в файлі **Program.cs**) з використанням механізму внесення залежностей. Ці служби потім надаються компонентам, які їх запитують через параметри конструктора.

У файлі **Controllers/SongsController.cs** використовується цей механізм внесення залежностей для передачі контексту бази даних **MvcSongContext** у конструктор контролера. Контекст бази даних використовується у кожному методі створення, читання, оновлення та видалення в цьому контролері.

При створенні шаблонів був створений наступний відокремлений код у файлі **Program.cs:**

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

*builder.Services.AddDbContext<MvcSongContext>(options =>*

*options.UseSqlServer(builder.Configuration.GetConnectionString("MvcSongContext")));*

Система конфігурації **ASP.NET Core** читає рядок підключення до контексту бази даних **MvcSongContext.**

**Перевірка створеного рядка підключення до бази даних.**

Під час створення шаблонів у файл **appsettings.json** було додано рядок підключення:

*{*

*"Logging": {*

*"LogLevel": {*

*"Default": "Information",*

*"Microsoft.AspNetCore": "Warning"*

*}*

*},*

*"AllowedHosts": "\*",*

*"ConnectionStrings": {*

*"MvcSongContext": "Data Source=MvcSongContext-ea7a4069-f366-4742-bd1c-3f753a804ce1.db"*

*}*

*}*

Під час локальної розробки система конфігурації **ASP.NET Core** зчитує ключ **ConnectionString** з файлу **appsettings.json**.

Клас **InitialCreate**. Файл міграції **Migrations/{timestamp}\_InitialCreate.cs:**

*using System;*

*using Microsoft.EntityFrameworkCore.Migrations;*

*#nullable disable*

*namespace MvcSong.Migrations*

*{*

*public partial class InitialCreate : Migration*

*{*

*protected override void Up(MigrationBuilder migrationBuilder)*

*{*

*migrationBuilder.CreateTable(*

*name: "Song",*

*columns: table => new*

*{*

*Id = table.Column<int>(type: "int", nullable: false)*

*.Annotation("SqlServer:Identity", "1, 1"),*

*Title = table.Column<string>(type: "nvarchar(max)", nullable: false),*

*Author = table.Column<string>(type: "nvarchar(max)", nullable: false),*

*ReleaseDate = table.Column<DateTime>(type: "datetime2", nullable: true),*

*Genre = table.Column<string>(type: "nvarchar(max)", nullable: true),*

*Duration = table.Column<int>(type: "int", nullable: false)*

*},*

*constraints: table =>*

*{*

*table.PrimaryKey("PK\_Song", x => x.Id);*

*});*

*}*

*protected override void Down(MigrationBuilder migrationBuilder)*

*{*

*migrationBuilder.DropTable(*

*name: "Song");*

*}*

*}*

*}*

У попередньому коді:

* метод **InitialCreate.Up** створює таблицю **Song** та налаштовує **Id** як первинний ключ;
* метод **InitialCreate.Down** скасовує зміни схеми, внесені під час міграції **Up.**

В контролері здійснюється внедрення залежностей. Відкрийте файл **Controllers/SongsController.cs** і перевірте конструктор:

*public class SongsController : Controller*

*{*

*private readonly MvcSongContext \_context;*

*public SongsController(MvcSongContext context)*

*{*

*\_context = context;*

*}*

Цей конструктор використовує введення залежностей для внесення контексту бази даних **(MvcSongContext)** в контролер. Контекст бази даних використовується в кожному методі створення, читання, оновлення та видалення в контролері.

Протестуйте сторінку, здійсніть введення та надішліть дані. Протестуйте сторінки редагування, інформації та видалення.

**Строго типізовані моделі та директива @model**

Раніше для передачі даних або об'єктів з контролера в представлення, було використано словник **ViewData.** **ViewData** - це динамічний об'єкт, який реалізує зручний механізм пізнього зв'язування для передачі інформації в представлення.

Модель **MVC** підтримує передачу строго типізованих об'єктів моделі в представлення. Такий підхід забезпечує перевірку коду під час компіляції. Механізм формування шаблонів передав строго типізовану модель у клас **SongsController** і представлення.

Ознайомтеся із створеним методом **Details** у файлі *Controllers/SongsController.cs:*

*// GET: Songs/Details/5*

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

*{*

*if (id == null)*

*{*

*return NotFound();*

*}*

*var Song = await \_context.Song*

*.FirstOrDefaultAsync(m => m.Id == id);*

*if (Song == null)*

*{*

*return NotFound();*

*}*

*return View(Song);*

*}*

Параметр **id** зазвичай передається як дані маршруту. Наприклад, **https://localhost:5001/Songs/details/1** вказує на:

* контролер **Songs** для першого сегмента **URL**-адреси;
* дію **details** для другого сегмента **URL**-адреси;
* значення 1 для **id** - останнього сегмента **URL**-адреси.

Параметр **id** можна передати за допомогою рядка запитань, як показано в наступному прикладі: <https://localhost:5001/Songs/details?id=1>.

Параметр **id** визначено як тип, який допускає значення **NULL (int?),** у випадку, якщо значення **id** не вказано.

Лямбда-вираз передається в метод **FirstOrDefaultAsync** для вибору записів пісень, які відповідають даним маршруту або значенню рядка запитань.

*var Song = await \_context.Song*

*.FirstOrDefaultAsync(m => m.Id == id);*

Якщо пісню знайдено, екземпляр моделі **Song** передається у представлення **Details**:

return View(Song);

Ознайомтеся з вмістом файлу **Views/Songs/Details.cshtml:**

*@model MvcSong.Models.Song*

*@{*

*ViewData["Title"] = "Details";*

*}*

*<h1>Details</h1>*

*<div>*

*<h4>Song</h4>*

*<hr />*

*<dl class="row">*

*<dt class = "col-sm-2">*

*@Html.DisplayNameFor(model => model.Title)*

*</dt>*

*<dd class = "col-sm-10">*

*@Html.DisplayFor(model => model.Title)*

*</dd>*

*<dt class = "col-sm-2">*

*@Html.DisplayNameFor(model => model.ReleaseDate)*

*</dt>*

*<dd class = "col-sm-10">*

*@Html.DisplayFor(model => model.ReleaseDate)*

*</dd>*

*<dt class = "col-sm-2">*

*@Html.DisplayNameFor(model => model.Genre)*

*</dt>*

*<dd class = "col-sm-10">*

*@Html.DisplayFor(model => model.Genre)*

*</dd>*

*<dt class = "col-sm-2">*

*@Html.DisplayNameFor(model => model.Price)*

*</dt>*

*<dd class = "col-sm-10">*

*@Html.DisplayFor(model => model.Price)*

*</dd>*

*</dl>*

*</div>*

*<div>*

*<a asp-action="Edit" asp-route-id="@Model.Id">Edit</a> |*

*<a asp-action="Index">Back to List</a>*

*</div>*

Оператор **@model** в початку файлу представлення визначає тип об'єкта, який буде очікувати представлення. При створенні контролера **"Song"** було включено наступне визначення **@model: @model MvcSong.Models.Song**.

Ця директива **@model** надає доступ до пісні, який контролер передав у представлення. Об'єкт **Model** є строго типізованим. Наприклад, у представленні **Details.cshtml** код передає кожне поле пісні в вбудовані функції **HTML** **DisplayNameFor** та **DisplayFor** зі строго типізованим об'єктом **Model**. Методи Create та Edit, а також відповідні представлення також передають об'єкт моделі Song.

Ознайомтеся з представленням **Index.cshtml** та методом **Index** в контролері **Songs**. Зверніть увагу на те, як у коді створюється об'єкт **List** при виклику метода **View**. Код передає список пісень з методу **Index** у представлення:

*// GET: Songs*

*public async Task<IActionResult> Index()*

*{*

*return View(await \_context.Song.ToListAsync());*

*}*

Код виводить інформацію про проблему, якщо властивість **Song** контексту даних має значення **NULL**.

Коли був створений контролер пісень, шаблон включав наступну інструкцію **@model** у верхній частині файлу **Index.cshtml:**

**@model IEnumerable<MvcSong.Models.Song>**

Ця директива **@model** надає доступ до списку пісень, який контролер передав у представлення, використовуючи строго типізований об'єкт **Model**. Наприклад, у представленні **Index.cshtml** код циклічно перебирає пісні за допомогою оператора **foreach** для строго типізованого об'єкта **Model**:

*@model IEnumerable<MvcSong.Models.Song>*

*@{*

*ViewData["Title"] = "Index";*

*}*

*<h1>Index</h1>*

*<p>*

*<a asp-action="Create">Create New</a>*

*</p>*

*<table class="table">*

*<thead>*

*<tr>*

*<th>*

*@Html.DisplayNameFor(model => model.Title)*

*</th>*

*<th>*

*@Html.DisplayNameFor(model => model.ReleaseDate)*

*</th>*

*<th>*

*@Html.DisplayNameFor(model => model.Genre)*

*</th>*

*<th>*

*@Html.DisplayNameFor(model => model.Price)*

*</th>*

*<th></th>*

*</tr>*

*</thead>*

*<tbody>*

*@foreach (var item in Model) {*

*<tr>*

*<td>*

*@Html.DisplayFor(modelItem => item.Title)*

*</td>*

*<td>*

*@Html.DisplayFor(modelItem => item.ReleaseDate)*

*</td>*

*<td>*

*@Html.DisplayFor(modelItem => item.Genre)*

*</td>*

*<td>*

*@Html.DisplayFor(modelItem => item.Price)*

*</td>*

*<td>*

*<a asp-action="Edit" asp-route-id="@item.Id">Edit</a> |*

*<a asp-action="Details" asp-route-id="@item.Id">Details</a> |*

*<a asp-action="Delete" asp-route-id="@item.Id">Delete</a>*

*</td>*

*</tr>*

*}*

*</tbody>*

*</table>*

Оскільки об'єкт **Model** є строго типізованим (так само, як і об'єкт **IEnumerable<Song>),** кожен елемент у циклі має тип **Song**. Окрім інших переваг, компілятор перевіряє типи, використовувані у коді.

**1.3 Завдання до лабораторної роботи**

Згідно із тематикою курсового проекту добавити всі необхідні контролери, представлення та моделі.

* 1. **Контрольні запитання**

1. Яким є об'єкт Model?
2. Що таке модель?
3. Що таке представлення?
4. Які способи добавлення контролерів існують?
5. Чого не повинні робити шаблони представлень?
6. Як створюються шаблони представлення?
7. Як змінити елементи у представленні?
8. Що таке Словник ViewData?
9. Які способи добавлення моделей існують?
10. Що таке CRUD?

**Literature: [1, 5, 6, 14, 15]**

**Лабораторна робота №3**

**Вибір користувацького інтерфейсу**

**Мета:** здійснити вибір тих функціональних можливостей, які слід об'єднати, щоб відповідати потребам користувацького вебінтерфейсу застосунку.

**1.1 Теоретичні основи та рекомендації до виконання**

Розробку користувацького інтерфейсу можна здійснювати завдяки таким технологіям:

1. **ASP.NET Core Blazor;**
2. **Razor Pages** в **ASP.NET Core;**
3. **ASP.NET Core MVC;**
4. **ASP.NET** основних односторінкових застосунків **(SPA)** з інтерфейсними платформами **JavaScript**;
5. Вибір гібридного рішення: **ASP.NET Core MVC** або **Razor Pages** у поєднанні з **Blazor**.

**1.Blazor** - це платформа вебінтерфейсу з повним стеком і рекомендується для більшості сценаріїв вебінтерфейсу.

Переваги використання **Blazor**:

* Модель повторно використовуваних компонентів.
* Ефективне промальовування компонентів.
* Гнучке відтворення компонентів із сервера або клієнта за допомогою **WebAssembly**.
* Створення розширених інтерактивних компонентів веб-інтерфейсу в **C#.**
* Відтворення компонентів статично із сервера.
* Поступово розширюйте відтворені сервером компоненти для плавнішої навігації та обробки форм і ввімкнення потокового відтворення.
* Загальний доступ до коду для загальної логіки на клієнті та сервері.
* Взаємодія з JavaScript.
* Інтеграція компонентів з наявними застосунками на основі **MVC**, **Razor Pages** або **JavaScript**.

**2.Razor Pages в ASP.NET Core**

Razor Pages - це модель на основі сторінок для створення відмальованого вебінтерфейсу сервера. Razor Користувацький інтерфейс сторінок динамічно відтворюється на сервері, щоб створити **HTML**-код і **CSS** сторінки у відповідь на запит браузера. Сторінка надходить на клієнт уже готова для перегляду. Підтримка **Razor Pages** здійснюється на базі **ASP.NET Core MVC.**

**Razor** Переваги сторінок:

* Швидке створення та оновлення користувацького інтерфейсу. Код сторінки зберігається на сторінці. Завдання, пов'язані з користувацьким інтерфейсом і бізнес-логікою, зберігаються окремо.
* Можливість виконати перевірку і масштабування великих застосунків.
* Упорядкування **ASP.NET** основних сторінок простіше, ніж **ASP.NET MVC**:
* Логіку і моделі уявлень можна зберігати разом у їхньому просторі імен і каталозі.
* Групи пов'язаних сторінок можуть зберігатися разом у власному просторі імен і каталозі.

**3.ASP.NET Core MVC.**

**ASP.NET Core MVC** відображає користувацький інтерфейс на сервері та використовує шаблон архітектури **Model-View-Controller (MVC).** Шаблон **MVC** відокремлює застосунок від трьох основних груп компонентів: моделей, уявлень і контролерів. Запити користувачів спрямовуються в контролер. Контролер відповідає за роботу з моделлю для виконання дій користувачів і отримання результатів запитів. Контролер вибирає подання для відображення користувачеві та надає йому всі необхідні дані моделі.

Переваги **ASP.NET Core MVC:**

Ґрунтується на масштабованій і продуманій моделі для створення великих вебзастосунків.

* Чіткий поділ завдань для максимальної гнучкості.
* Поділ обов'язків на основі шаблону **"Модель - представлення - контролер"** гарантує, що бізнес-модель можна легко розвивати, не зачіпаючи при цьому реалізацію можливостей нижчого рівня.

**4.ASP.NET основних односторінкових застосунків (SPA) з інтерфейсними платформами JavaScript.**

Є можливість створення клієнтської логіки для застосунків ASP.NET **Core** за допомогою популярних платформ **JavaScript**, таких як **Angular**, **React** і **Vue**. **ASP.NET Core** надає шаблони проектів для **Angular, React** і **Vue**, а також можна використовувати з іншими платформами **JavaScript**.

Переваги **SPA** в **ASP.NET Core SPA** з платформами JavaScript на застосунок до наведених вище переваг відтворення клієнта:

* Середовище виконання **JavaScript** уже надано разом із браузером.
* Велика спільнота і продумана екосистема.
* Створення клієнтської логіки для застосунків **ASP.NET Core** за допомогою популярних **JS платформ**, як-от **Angular**, **React** і **Vue**.

Недоліки:

* Потрібні додаткові мови програмування, платформи та засоби.
* Складно обмінюватися кодом, тому деяка логіка може дублюватися.

**5.Вибір гібридного рішення: ASP.NET Core MVC або Razor Pages у поєднанні з Blazor**

**MVC, Razor Pages** і **Blazor** є частиною платформи **ASP.NET Core** і розроблені для спільного використання. Компоненти **Razor** можна інтегрувати в застосунки **MVC** і **Razor Pages**. Одночасно з відтворенням сторінки або представлення можна виконувати попередню обробку компонентів.

Переваги **MVC** або **Razor Pages** у поєднанні з **Blazor**, які доповнюють звичайні переваги **MVC** і **Razor Pages**:

* попереднє промальовування виконує компоненти **Razor** на сервері та промальовує їх у вигляді представлення або сторінки, що покращує сприйняття та швидкість завантаження програми;
* додавання інтерактивності в наявні представлення (сторінки) за допомогою допоміжної функції тега компонента.
  1. **Приклади виконання завдання**

**Створення застосунку ASP.NET Core за допомогою Angular в Visual Studio.**

**Visual Studio** включає шаблони **ASP.NET Core Single Page Application** **(SPA)**, що підтримують **Angular** і **React**. Шаблони надають вбудовану папку клієнтського застосунку в проєктах **ASP.NET Core**, що містять базові файли та папки кожної платформи.

Метод, що описаний, можна використовувати для створення **ASP.NET** односторінкових застосунків core, які:

* розміщення клієнтського застосунку в окремому проекті поза проектом **ASP.NET Core**;
* створення клієнтського проекту на основі інтерфейсу **CLI** платформи, встановленого на комп'ютері.

Необхідні компоненти

Обов'язково встановіть такі компоненти:

**Visual Studio 2022** версії 17.8 або пізнішої версії зі встановленим робочим навантаженням **ASP.NET** і веброзробки. Перейдіть на сторінку завантажуваних матеріалів **Visual Studio**, щоб встановити її безкоштовно. Якщо потрібно встановити робоче навантаження і вже є **Visual Studio**, то необхідно перейти у розділ **Засоби>Отримати засоби та компоненти**..., після чого запуститься **Visual Studio Installer**. Далі необхідно вибрати робоче навантаження **ASP.NET** і розробка вебзастосунків, а потім натиснути Змінити. (npm (https://www.npmjs.com/), який входить до складу Node.js; CLI Angular (https://angular.io/cli). Версія може бути будь-якою на вибір).

Порядок створення інтерфейсного застосунку:

1. У вікні запуску (щоб відкрити його, клацніть **Файл>Вікно запуску**) виберіть **Створення нового проєкту**.
2. Знайдіть **Angular** у рядку пошуку у верхній частині, а потім виберіть **Angular** і **ASP.NET Core** (попередня версія) (рисунок 3.1)

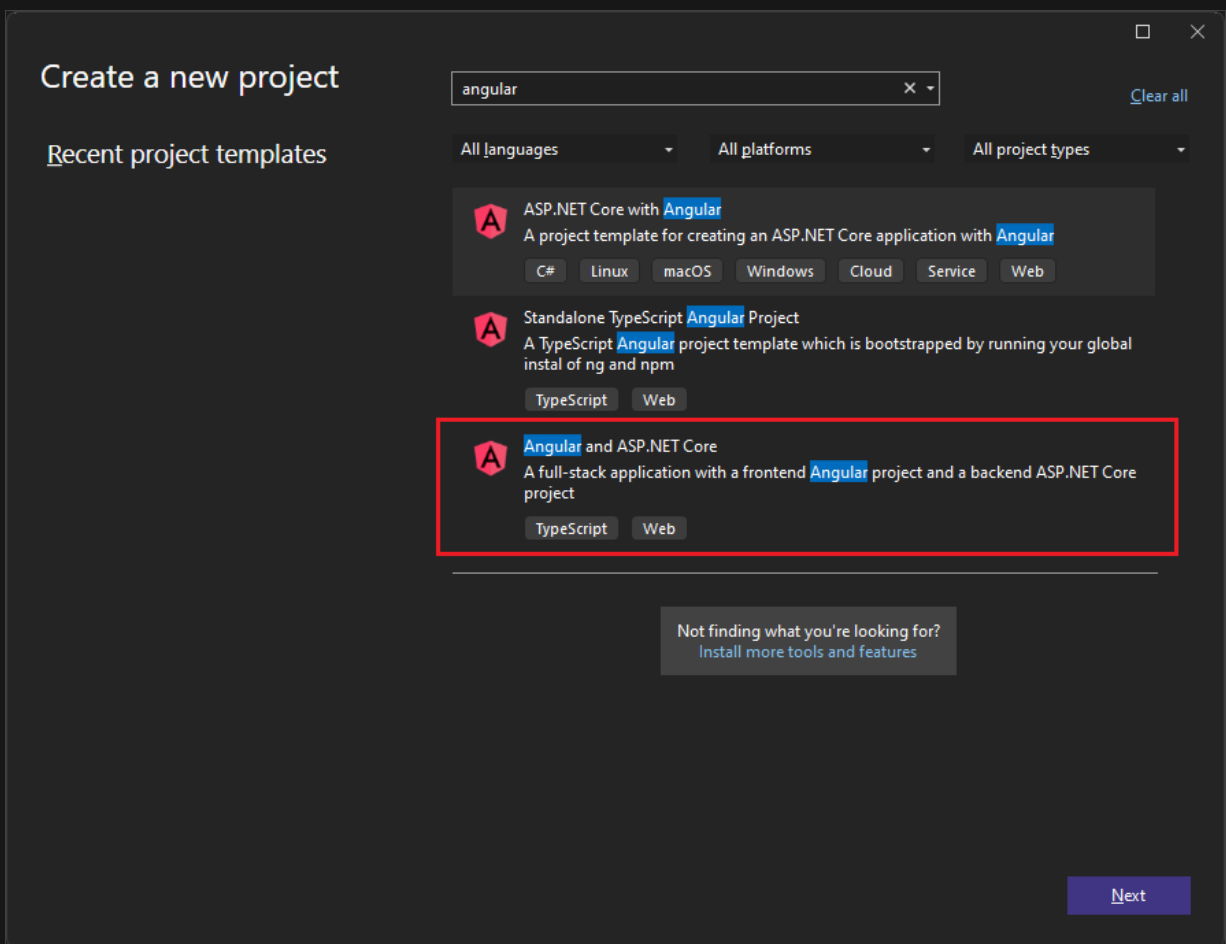


Рисунок 3.1 – Вікно створення нового проекту

1. Назвіть проєкт **AngularWithASP** та натисніть кнопку **"Створити".**

У порівнянні із автономним шаблоном Angular буде видно нові та змінені файли для інтеграції із **ASP.NET Core**, а саме:

aspnetcore-https.js;

proxy.conf.js;

package.json (змінений);

angular.json (змінений);

app.components.ts;

app.module.ts.

Оглядач рішень показано на рисунку 3.2.

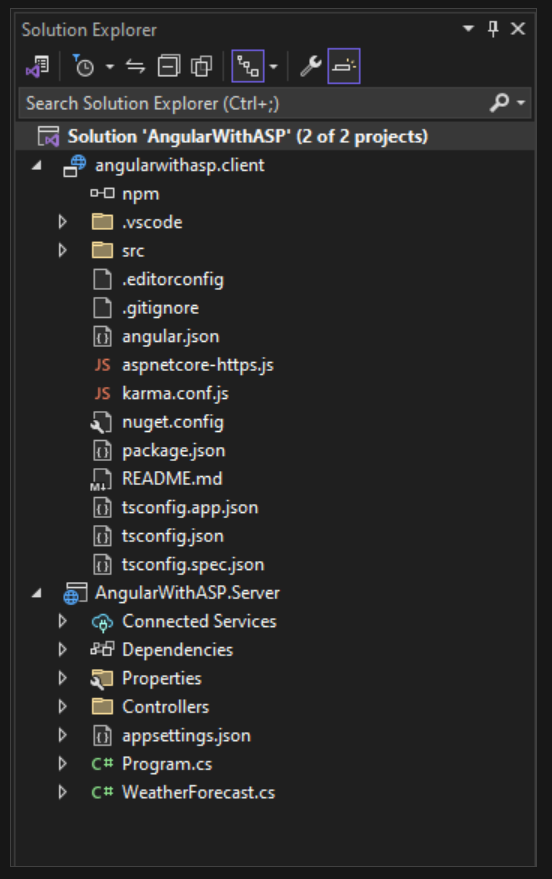


Рисунок 3.2 – Вигляд оглядача рішень

Далі необхідно здійснити задання властивостей проєкту. Це робиться наступним чином.

1.В Оглядачі рішень клацніть правою кнопкою миші проєкт **AngularWithASP.Server** і виберіть **"Властивості"** (рисунок 3.3).

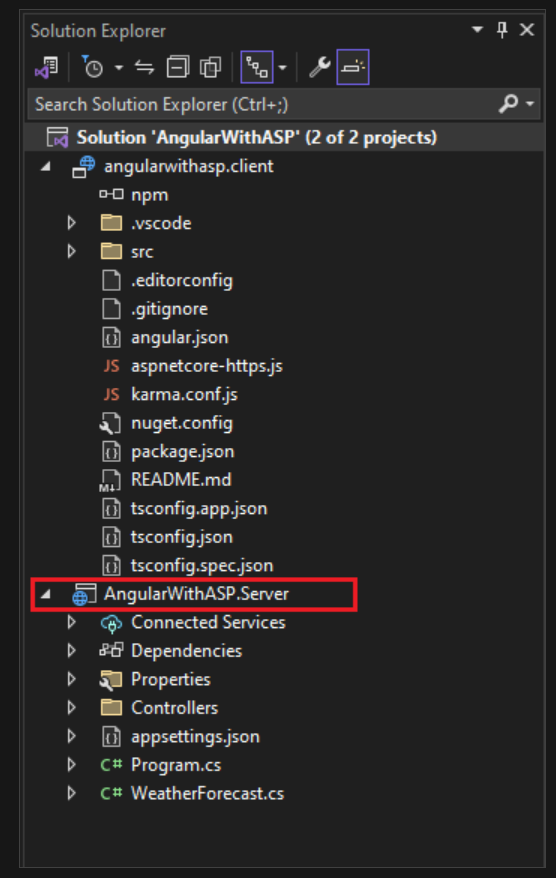


Рисунок 3.3 – Вибір пункту Властивості

На сторінці **"Властивості"** відкрийте вкладку «Налагодження» і виберіть пункт **«Відкрити профілі запуску налагодження»**. Перевірте параметр **«Запустити браузер»** для профілю з іменем проєкту **ASP.NET Core** (або **https**, якщо він присутній) (рисунок 3.4).

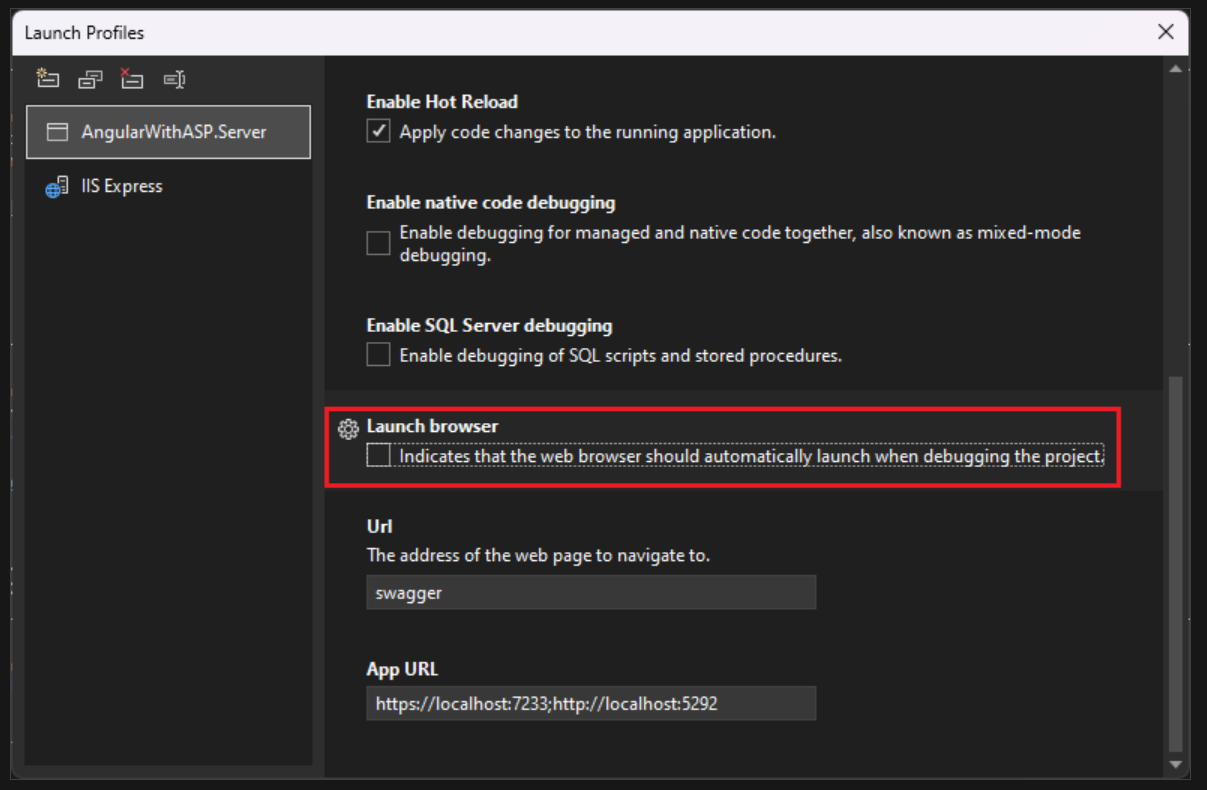


Рисунок 3.4 – Сторінка «Властивості»

**Примітка.** У **Visual Studio launch.json** зберігаються параметри запуску, пов'язані з кнопкою "Пуск" на панелі інструментів налагодження. **launch.json** повинен знаходитися в папці **VSCODE**.

***Запуск проєкту***

Натисніть клавішу **F5** або натисніть кнопку **"Пуск"** у верхній частині вікна, щоб запустити застосунок. Відображаються два командні рядки:

* проект, що виконується, **API ASP.NET Core;**
* виконання команди **ng start** в **Angular CLI.**

**Примітка.** Перевірте вихідні дані консолі для повідомлень. Наприклад, може бути повідомлення про оновлення **Node.js.**

1.В Оглядачі рішень клацніть правою кнопкою миші проєкт **AngularWithASP.Server** і виберіть **"Додати>посилання на проєкт".** Переконайтеся, що вибрано проект angularwithasp.client .

2.Виберіть **OK**.

3.Клацніть правою кнопкою миші проект **ASP.NET Core** і виберіть **"Змінити файл проекту".** Відкриється **CSPROJ**-файл проекту.

4.У **CSPROJ**-файлі переконайтеся, що посилання на проект містить **<ReferenceOutputAssembly>** елемент із заданим **false**-значенням.

Це посилання має мати такий вигляд:

*<ProjectReference Include="..\angularwithasp.client\angularwithasp.client.esproj">*

*<ReferenceOutputAssembly>false</ReferenceOutputAssembly>*

*</ProjectReference>*

5.Клацніть правою кнопкою миші проєкт **ASP.NET Core** і виберіть **"Перезавантажити проєкт",** якщо цей параметр доступний.

6.У Program.cs переконайтеся, що у файлі Program.cs присутній такий код.

*app.UseDefaultFiles();*

*app.UseStaticFiles();*

*// Configure the HTTP request pipeline.*

*if (app.Environment.IsDevelopment())*

*{*

*app.UseSwagger();*

*app.UseSwaggerUI();*

*}*

7.Щоб опублікувати, клацніть правою кнопкою миші проєкт **ASP.NET** **Core**, виберіть **"Опублікувати"**, а також виберіть параметри для зіставлення необхідного сценарію публікації, наприклад, **Azure**, публікації в папці тощо.

Процес публікації займає більше часу, ніж тільки проєкту **ASP.NET Core**, тому команда ***npm run build*** викликається під час публікації. Збірка Command виконується ***npm run build*** за замовчуванням.

**1.3 Завдання до лабораторної роботи**

Здійснити реалізацію користувацького інтерфейсу інтернет-магазину, онлайн-ресурсу, маркетплейсу або іншого вебзастосунку відповідно до тематики курсового проекту

**1.4 Контрольні запитання**

1. Які технології дозволяють здійснювати розробку користувацького інтерфейсу?
2. Що таке ASP.NET Core Blazor?
3. Як функціонує Razor Pages в ASP.NET Core?
4. Вказати переваги використання Blazor?
5. Вказати недоліки використання Blazor?
6. Як можна здійснити розробку користувацького інтерфейсу за допомогою Angular?
7. Охарактеризуйте основні переваги ASP.NET Core MVC?
8. Які переваги Razor Pages в ASP.NET Core?

**Literature: [1, 5, 6, 14, 15]**

**Лабораторна робота 4**

**Робота з базою даних в застосунках**

**Мета:** Навчитись добавляти базу даних у застосунки

**1.1 Теоретичні основи та рекомендації до виконання**

Об'єкт **MvcSongContex**t вирішує завдання підключення до бази даних і відображення об'єктів Song на записи бази даних. Контекст бази даних реєструється через контейнер інтеграцію залежностей у файлі **Program.cs:**

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

*builder.Services.AddDbContext<MvcSongContext>(options =>*

*options.UseSqlServer(builder.Configuration.GetConnectionString("MvcSongContext")));*

Система конфігурації ASP.NET Core зчитує ключ ConnectionString. Для розробки на локальному рівні вона отримує рядок підключення з файлу appsettings.json.

*"ConnectionStrings": {*

*"MvcSongContext": "Data Source=MvcSongContext-ea7a4069-f366-4742-bd1c-3f753a804ce1.db"*

*}*

Якщо застосунок розгортається на тестовому чи робочому сервері, можна встановити рядок підключення до робочого **SQL Server** за допомогою змінної середовища.

**SQL Server Express LocalDB**

LocalDB:

* це спрощена версія ядра **СУБД SQL Server Express**, яка встановлюється за замовчуванням разом із **Visual Studio.**
* Запускається за вимогою за допомогою рядка підключення.
* Використовується для розробки програм. Запускається в режимі користувача, тому конфігурація не є дуже складною.
* За замовчуванням створює файли **.mdf** в каталозі **C:/Users/{user}.**

**1.2 Приклади виконання завдання**

Перевірка бази даних. У меню Вид відкрийте браузер об'єктів **SQL Server (SSOX).**

Клацніть правою кнопкою миші конструктор переглядів таблиці **Song (dbo.Song) >** (Рисунки 4.1- 4.2).

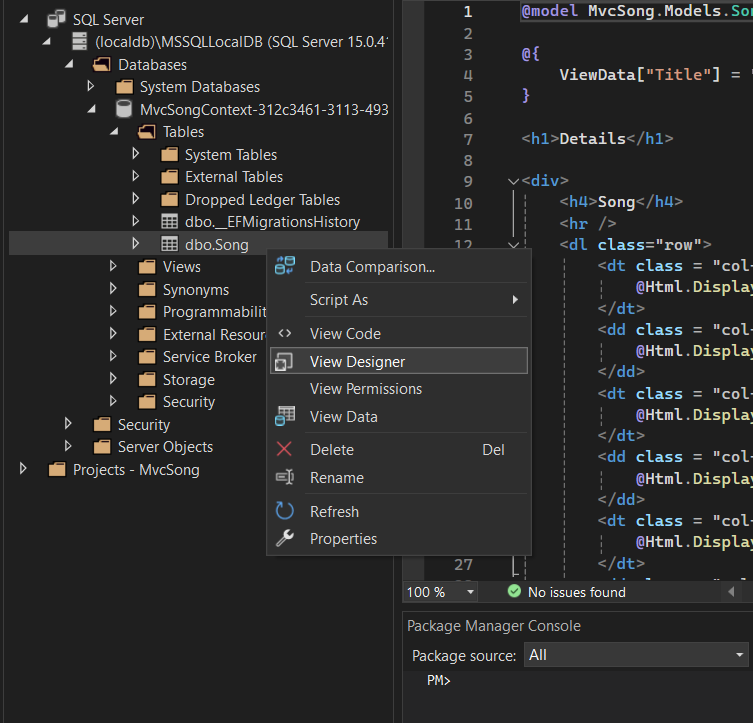


Рисунок 4.1 – Конструктор переглядів таблиці

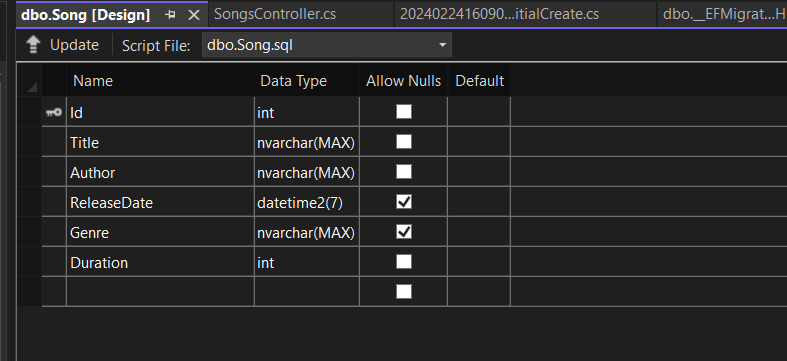


Рисунок 4.2 – Перегляд таблиць

Зверніть увагу на значок із зображенням ключа поруч з ID. За замовчуванням EF встановлює властивість з іменем ID в якості первинного ключа.

Клацніть правою кнопкою миші на таблицю **Song > View Data.**

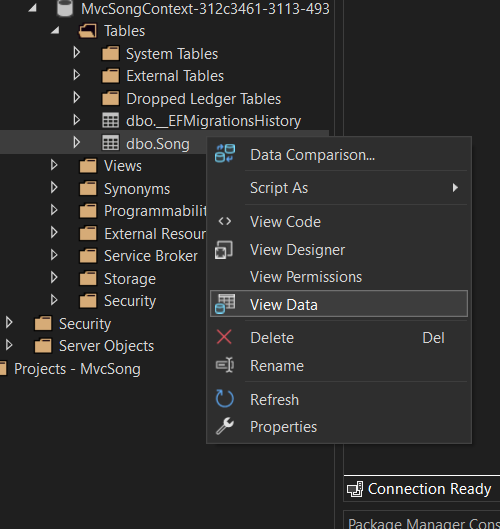


Рисунок 4.3 – Таблиця Song

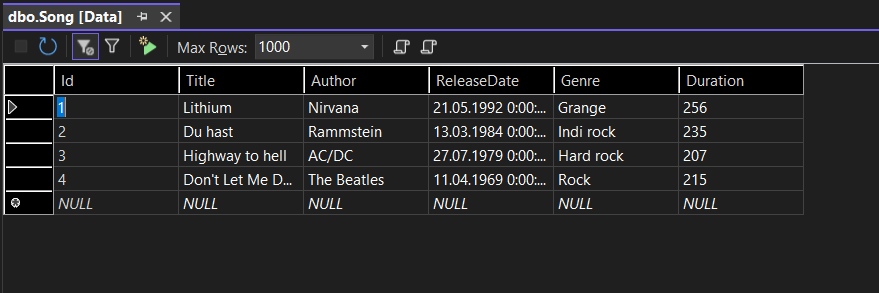


Рисунок 4.4 – Перегляд таблиці Song

Для заповнення бази даних створіть клас **SeedData** у папці **Models**. Замініть згенерований код на наступний:

*using Microsoft.EntityFrameworkCore;*

*using MvcSong.Data;*

*namespace MvcSong.Models*

*{*

*public static class SeedData*

*{*

*public static void Initialize(IServiceProvider serviceProvider)*

*{*

*using (var context = new MvcSongContext(*

*serviceProvider.GetRequiredService<*

*DbContextOptions<MvcSongContext>>()))*

*{*

*// Look for any Songs.*

*if (context.Song.Any())*

*{*

*return;*

*}*

*context.Song.AddRange(*

*new Song*

*{*

*Title = "Lithium",*

*Author = "Nirvana",*

*ReleaseDate = DateTime.Parse("1992-5-21"),*

*Genre = "Grange",*

*Duration = 256*

*},*

*new Song*

*{*

*Title = "Du hast",*

*Author = "Rammstein",*

*ReleaseDate = DateTime.Parse("1984-3-13"),*

*Genre = "Indi rock",*

*Duration = 235*

*},*

*new Song*

*{*

*Title = "Highway to hell",*

*Author = "AC/DC",*

*ReleaseDate = DateTime.Parse("1979-07-27"),*

*Genre = "Hard rock",*

*Duration = 207*

*},*

*new Song*

*{*

*Title = "Don't Let Me Down",*

Author = "The Beatles",

ReleaseDate = DateTime.Parse("1969-4-11"),

Genre = "Rock",

Duration = 215

}

);

context.SaveChanges();

}

}

}

}

Якщо в базі даних є пісні, повертається ініціалізатор заповнення, і пісні не додаються.

if (context.Song.Any())

*{*

*return; // DB has been seeded.*

*}*

*Для додавання ініціалізатора заповнення замініть код Program.cs наступним кодом:*

*using Microsoft.EntityFrameworkCore;*

*using Microsoft.Extensions.DependencyInjection;*

*using MvcSong.Data;*

*using MvcSong.Models;*

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

*builder.Services.AddDbContext<MvcSongContext>(options =>*

*options.UseSqlServer(builder.Configuration.GetConnectionString("MvcSongContext")));*

*builder.Services.AddControllersWithViews();*

*var app = builder.Build();*

*using (var scope = app.Services.CreateScope())*

*{*

*var services = scope.ServiceProvider;*

*SeedData.Initialize(services);*

*}*

*{*

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

*app.UseHsts();*

*}*

*app.UseHttpsRedirection();*

*app.UseStaticFiles();*

*app.UseRouting();*

*app.UseAuthorization();*

*app.MapControllerRoute(*

*name: "default",*

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

*app.Run();*

В застосунку буде відображатися представлення, зображене на рисунку 4.5:

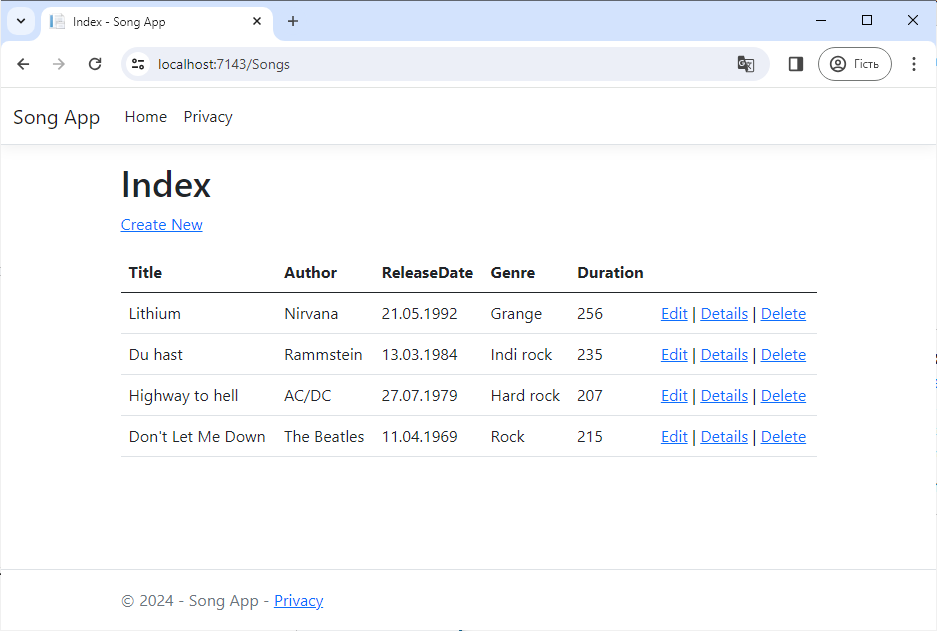


Рисунок 4.5 – Результат заповнення бази даних

**Методи та представлення контролера в ASP.NET Core**

Переважна частина застосунку готова, але ще потрібно вносити певні правки, наприклад, елемент **"ReleaseDate"** має містити два слова.

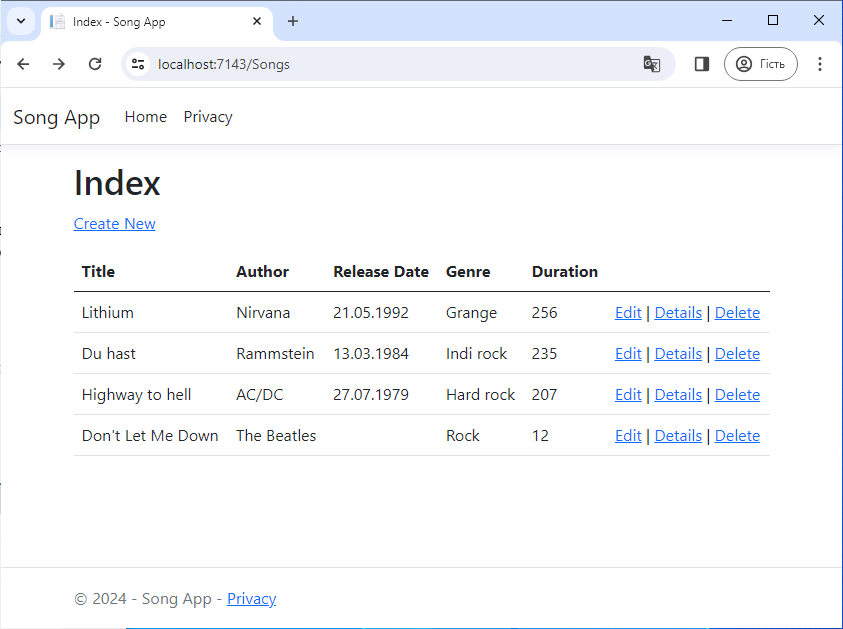


Рисунок 4.6 - Елемент **"ReleaseDate"**

Необхідно відкрити файл *Models/Song.cs* та додати виділені рядки:

*using Microsoft.AspNetCore.Components.Forms;*

*using System.ComponentModel.DataAnnotations;*

*namespace MvcSong.Models*

*{*

*public class Song*

*{*

*public int Id { get; set; }*

*public string? Title { get; set; }*

*public string? Author { get; set; }*

*[Display(Name = "Release Date")]*

*[DataType(DataType.Date)]*

*public DateTime? ReleaseDate { get; set; }*

*public string? Genre { get; set; }*

*public int Duration { get; set; }*

*}*

*}*

Атрибут **Display** визначає відображуване ім'я поля (у цьому випадку **"Release Date"** замість **"ReleaseDate").** Атрибут **DataType** визначає тип даних (**Date**), тому інформація про час, яка зберігається в полі, не відображається.

Перейдіть до контролера **Songs**. Наведіть вказівник миші на посилання **Edit** та утримуйте його на місці, щоб переглянути цільову **URL**-адресу.

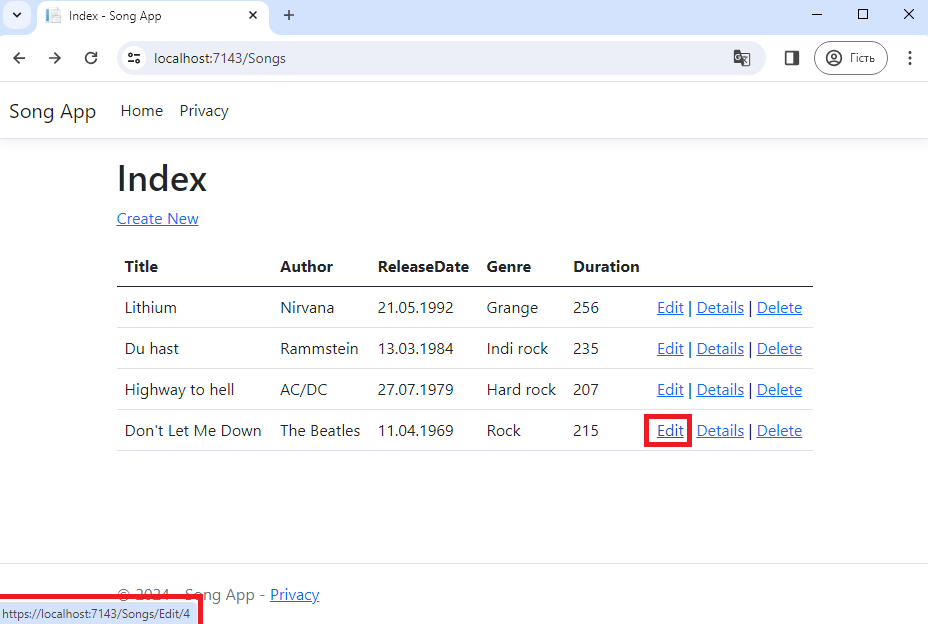


Рисунок 4.7 - Елемент "ReleaseDate", що складається з двох слів

Посилання **"Edit", "Details"** та **"Delete"** створюються за допомогою вбудованого елемента тегу прив'язки **Core MVC** в файлі **Views/Songs/Index.cshtml.**

*<a asp-action="Edit" asp-route-id="@item.Id">Edit</a> |*

*<a asp-action="Details" asp-route-id="@item.Id">Details</a> |*

*<a asp-action="Delete" asp-route-id="@item.Id">Delete</a>*

*</td>*

*</tr>*

Вбудовані функції тегів дозволяють серверному коду брати участь у створенні та відображенні **HTML**-елементів у файлах **Razor**. У вищезазначеному коді **AnchorTagHelper** динамічно створює значення атрибута **HTML** **href** на основі методу дії контролера та ідентифікатора маршруту. Для огляду створеної розмітки скористайтеся функцією перегляду вихідного коду або інструментами розробника у вашому обраному браузері. Нижче показана частина створеного **HTML**-коду:

<*td>*

*<a href="/Songs/Edit/4"> Edit </a> |*

*<a href="/Songs/Details/4"> Details </a> |*

*<a href="/Songs/Delete/4"> Delete </a>*

*</td>*

Формат для налаштування маршрутизації визначений у файлі **Program.cs:**

*app.MapControllerRoute(*

*name: "default",*

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

**ASP.NET Core** перетворює **https://localhost:5001/Songs/Edit/4** на запит методу **Edit** контролера **Songs** з параметром Id, який дорівнює 4. (Методи контролера також називаються методами дії.)

Вбудовані функції тегів є однією з найбільш популярних нововведень в **ASP.NET Core.**

Відкрийте контролер **Songs** і розгляньте два методи дії **Edit**. У наступному коді показано метод **HTTP GET Edit**, який витягує пісні та заповнює форму редагування, створену файлом **Edit.cshtml Razor.**

*// GET: Songs/Edit/5*

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

*{*

*if (id == null)*

*{*

*return NotFound();*

*}*

*var Song = await \_context.Song.FindAsync(id);*

*if (Song == null)*

*{*

*return NotFound();*

*}*

*return View(Song);*

*}*

Далі показано метод **HTTP POST Edit**, який обробляє надіслані значення пісень:

*[HttpPost]*

*[ValidateAntiForgeryToken]*

*public async Task<IActionResult> Edit(int id, [Bind("Id,Title,Author,ReleaseDate,Genre,Duration")] Song Song)*

*{*

*if (id != Song.Id)*

*{*

*return NotFound();*

*}*

*if (ModelState.IsValid)*

*{*

*try*

*{*

*\_context.Update(Song);*

*await \_context.SaveChangesAsync();*

*}*

*catch (DbUpdateConcurrencyException)*

*{*

*if (!SongExists(Song.Id))*

*{*

*return NotFound();*

*}*

*else*

*{*

*throw;*

*}*

*}*

*return RedirectToAction(nameof(Index));*

*}*

*return View(Song);*

*}*

Атрибут **[Bind]** є одним із способів захисту від надмірного передавання даних. Властивості слід включати лише в атрибут **[Bind],** який ви хочете змінити. **ViewModel** реалізує альтернативний підхід до захисту від надмірного передавання даних.

Зверніть увагу на другий метод дії **Edit**, перед яким стоїть атрибут **[HttpPost].**

Атрибут **HttpPost** вказує на те, що цей метод **Edit** може бути викликаний лише для запитів **POST**. Ви могли б додати атрибут **[HttpGet]** до першого методу редагування, але це необов'язково, оскільки значення **[HttpGet]** задається за замовчуванням.

Атрибут **ValidateAntiForgeryToken** використовується для захисту від підробки запитів разом із відповідним маркером безпеки, який створюється в файлі представлення редагування **(Views/Songs/Edit.cshtml)** із використанням допоміжної функції тегу **Form**.

*<form asp-action="Edit">*

Допоміжна функція тегу **Form** створює прихований маркер захисту від підробки, який повинен відповідати **[ValidateAntiForgeryToken]** аналогічному маркеру безпеки в методі Edit контролера **Songs**.

Метод **HttpGet Edit** приймає параметр **ID** пісні, виконує пошук пісні за допомогою методу **FindAsync** платформи Entity Framework і повертає вибрану пісню у представлення редагування. Якщо пісню не вдається знайти, виводиться помилка **NotFound** **(HTTP 404).**

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

*{*

*if (id == null)*

*{*

*return NotFound();*

*}*

*var Song = await \_context.Song.FindAsync(id);*

*if (Song == null)*

*{*

*return NotFound();*

*}*

*return View(Song);*

*}*

Якщо в представленні редагування використовується система створення шаблонів, вона аналізує клас Song та генерує код для відображення елементів **<label>** та **<input>** для кожної властивості класу. У наступному прикладі показане представлення редагування, створене системою створення шаблонів **Visual Studio:**

*@model MvcSong.Models.Song*

*@{*

*ViewData["Title"] = "Edit";*

*}*

*<h1>Edit</h1>*

*<h4>Song</h4>*

*<hr />*

*<div class="row">*

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

*<form asp-action="Edit">*

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

*<input type="hidden" asp-for="Id" />*

*<div class="form-group">*

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

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

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

*</div>*

*<div class="form-group">*

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

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

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

*</div>*

*<div class="form-group">*

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

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

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

*</div>*

*<div class="form-group">*

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

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

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

*</div>*

*<div class="form-group">*

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

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

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

*</div>*

*<div class="form-group">*

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

*</div>*

*</form>*

*</div>*

*</div>*

*<div>*

*<a asp-action="Index">Back to List</a>*

*</div>*

*@section Scripts {*

*@{await Html.RenderPartialAsync("\_ValidationScriptsPartial");}*

*}*

Зверніть увагу, що на початку файлу шаблону представлення є оператор **@model MvcSong.Models.Song**. **@model MvcSong.Models.Song** вказує, що для представлення потрібна модель представлення шаблону з типом **Song**.

Для оптимізації **HTML**-розмітки згенерований код використовує кілька вспоміжних функцій тегів. Допоміжна функція тега Label відображає ім'я поля **("Title",** **"ReleaseDate",** **"Genre"** або **"Price").** Допоміжна функція тегу **Input** відображає елемент **HTML <input>.** Вспоміжна функція тега Validation відображає будь-які повідомлення перевірки, пов'язані з вказаною властивістю.

Далі здійсніть запуск застосунку та перейдіть за **URL**-адресою **/Songs**. Клацніть посилання **Edit**. Перегляньте вихідний код сторінки у вікні браузера. Створений **HTML**-код для елемента **<form>** показано нижче.

*<form action="/Songs/Edit/4" method="post" novalidate="novalidate">*

*<input type="hidden" data-val="true" data-val-required="The Id field is required." id="Id" name="Id" value="4">*

*<div class="form-group">*

*<label class="control-label" for="Title">Title</label>*

*<input class="form-control" type="text" data-val="true" data-val-required="The Title field is required." id="Title" name="Title" value="Don't Let Me Down">*

*<span class="text-danger field-validation-valid" data-valmsg-for="Title" data-valmsg-replace="true"></span>*

*</div>*

*<div class="form-group">*

*<label class="control-label" for="Author">Author</label>*

*<input class="form-control" type="text" data-val="true" data-val-required="The Author field is required." id="Author" name="Author" value="The Beatles">*

*<span class="text-danger field-validation-valid" data-valmsg-for="Author" data-valmsg-replace="true"></span>*

*</div>*

*<div class="form-group">*

*<label class="control-label" for="ReleaseDate">ReleaseDate</label>*

*<input class="form-control" type="date" id="ReleaseDate" name="ReleaseDate" value="1969-04-11"><input name="\_\_Invariant" type="hidden" value="ReleaseDate">*

*<span class="text-danger field-validation-valid" data-valmsg-for="ReleaseDate" data-valmsg-replace="true"></span>*

*</div>*

*<div class="form-group">*

*<label class="control-label" for="Genre">Genre</label>*

*<input class="form-control" type="text" id="Genre" name="Genre" value="Rock">*

*<span class="text-danger field-validation-valid" data-valmsg-for="Genre" data-valmsg-replace="true"></span>*

*</div>*

*<div class="form-group">*

*<label class="control-label" for="Duration">Duration</label>*

*<input class="form-control" type="number" data-val="true" data-val-required="The Duration field is required." id="Duration" name="Duration" value="215"><input name="\_\_Invariant" type="hidden" value="Duration">*

*<span class="text-danger field-validation-valid" data-valmsg-for="Duration" data-valmsg-replace="true"></span>*

*</div>*

*<div class="form-group">*

*<input type="submit" value="Save" class="btn btn-primary">*

*</div>*

*<input name="\_\_RequestVerificationToken" type="hidden" value="CfDJ8A0eaDGVOh5GqjwgV9A\_cW8VZSHRYlj4-GCKXANafnlEH3\_luITPSsf16VeZtVuAUfcDGbka1rcpQ9VdLs13W7aX\_sbVklP3RntOal49DCcY9-i19485iA15UC6wyaQ2SErK6sOaM8q4wC8HsoipwZw"></form>*

Елементи **<input>** розташовані в елементі HTML <form>, атрибут action якого встановлює передачу даних за **URL**-адресою **/Songs/Edit/id.** Дані форми будуть передаватися на сервер при натисканні кнопки Save. У останньому рядку перед закриваючим елементом **</form>** відображається прихований маркер **XSRF**, створений допоміжною функцією тегу **Form**.

**Обробка запиту Post**

Наступний фрагмент коду показує версію методу дії **Edit**, призначеного для **[HttpPost]:**

*[HttpPost]*

*[ValidateAntiForgeryToken]*

*public async Task<IActionResult> Edit(int id, [Bind("Id,Title,Author,ReleaseDate,Genre,Duration")] Song song)*

*{*

*if (id != song.Id)*

*{*

*return NotFound();*

*}*

*if (ModelState.IsValid)*

*{*

*try*

*{*

*\_context.Update(song);*

*await \_context.SaveChangesAsync();*

*}*

*catch (DbUpdateConcurrencyException)*

*{*

*if (!SongExists(song.Id))*

*{*

*return NotFound();*

*}*

*else*

*{*

*throw;*

*}*

*}*

*return RedirectToAction(nameof(Index));*

*}*

*return View(song);*

*}*

Атрибут **[ValidateAntiForgeryToken]** перевіряє прихований маркер безпеки **XSRF**, який був створений генератором маркерів у допоміжній функції тегу **Form**.

Система прив'язки моделі приймає передані значення форми і створює об'єкт **Song**, який передається у якості параметра **Song**. Властивість **ModelState.IsValid** перевіряє, чи можна використовувати передані дані форми для зміни (редагування або оновлення) об'єкта **Song**. Допустимі дані зберігаються. Оновлені (змінені) дані пісні зберігаються в базі даних за допомогою виклику метода **SaveChangesAsync** у контексті бази даних. Після збереження даних код перенаправляє користувача на метод дії **Index** класу **SongsControlle**r, який відображає колекцію пісень з урахуванням щойно внесених змін.

Перед відправкою форми на сервер на клієнтській стороні перевіряється виконання всіх правил перевірки для полів. При виявленні помилок перевірки виводиться повідомлення про помилку, а форма не передається. Якщо **JavaScript** вимкнено, перевірка на клієнтській стороні не виконується. Однак, сервер виявить передані недопустимі значення, в результаті чого значення форми будуть відображені повторно з повідомленнями про помилки. Допоміжний елемент тегу перевірки у шаблоні **Views/Songs/Edit.cshtml** відображає відповідні повідомлення про помилки.

Всі методи **HttpGet** в контролері **Song** мають схожий шаблон. Вони отримують об'єкт пісні (або список об'єктів для методу Index) та передають об'єкт (модель) в представлення. Метод **Create** передає в представлення порожній об'єкт пісні **Create**. Всі методи, які створюють, редагують, видаляють або іншим чином змінюють дані, роблять це в перевантаженні методу **[HttpPost].** Зміна даних в методі **HTTP GET** пов'язана з загрозою безпеки. Зміна даних у методі **HTTP GET** також порушує рекомендації **HTTP** та шаблон архітектури **REST**, який вказує, що запити **GET** не повинні змінювати стан застосунку. Іншими словами, операція **GET** повинна виконуватися безпечним способом, тобто не мати побічних ефектів і не змінювати існуючі дані.

**1.3 Завдання до лабораторної роботи**

Відповідно до тематики курсового проекту здійснити реалізацію необхідних таблиць бази даних.

**1.4 Контрольні запитання**

1. Що таке база даних?
2. Яка роль атрибута ValidateAntiForgeryToken?
3. Що робить метод SaveChangesAsync?
4. Яка функція оператора @model?
5. Яка роль методу HttpGet Edit?
6. Для чого призначений атрибут Display?
7. Що таке вбудовані функції тегів та для чого вони слугують?
8. Що означає вираз @model MvcSong.Models.Song?

**Literature: [1, 5, 6, 14, 15]**

**Лабораторна робота 5**

**Додавання пошуку та полів у застосунок ASP.NET Core MVC**

**Мета:** Навчитись додавати пошук у створюваний проєкт, а також добавляти поля.

* 1. **Теоретичні основи та рекомендації до виконання**

**Фільтри дають змогу виконувати деякі дії до або після певної стадії обробки запиту. В ASP.NET Core MVC є такі типи фільтрів:**

Фільтри авторизації: визначають, чи авторизований користувач для виконання поточного запиту. Якщо користувач не авторизований для доступу до ресурсу, то фільтр завершує обробку запиту.

Фільтри ресурсів: виконуються після фільтрів авторизації. Його метод OnResourceExecuting() виконується до всіх інших фільтрів і до прив'язки моделі, а його метод OnResourceExecuted() виконується після всіх інших фільтрів.

Фільтри дій: застосовується тільки до дій контролера, запускається після фільтра ресурсів як до, так і після виконання методу контролера

Фільтри винятків: визначають дії щодо необроблених винятків

Фільтри результатів дій: фільтр застосовується до результатів методів контролера, виконується як до, так і після отримання результату

Разом усі ці типи фільтрів утворюють конвеєр фільтрів (filter pipeline), що вбудований у процес опрацювання запиту в MVC та який починає виконуватися після того, як інфраструктура MVC обрала метод контролера для опрацювання запиту. На різних етапах обробки запиту в цьому конвеєрі викликається відповідний фільтр.

Оновіть метод **Index** у файлі **Controllers/SongsController.cs**, використовуючи наступний код:

*public async Task<IActionResult> Index(string searchString)*

*{*

*if (\_context.Song == null)*

*{*

*return Problem("Entity set 'MvcSongContext.Song' is null.");*

*}*

*var Songs = from m in \_context.Song*

*select m;*

*if (!String.IsNullOrEmpty(searchString))*

*{*

*Songs = Songs.Where(s => s.Title!.Contains(searchString));*

*}*

*return View(await Songs.ToListAsync());*

*}*

Наступний рядок у методі Index створює LINQ-запит для вибору пісень:

var Songs = from m in \_context.Song

select m;

Цей запит лише визначається на цьому етапі і не виконується для бази даних.

Якщо параметр **searchString** містить рядок, запит пісень змінюється для фільтрації за значенням у рядку пошуку:

if (!String.IsNullOrEmpty(searchString))

{

Songs = Songs.Where(s => s.Title!.Contains(searchString));

}

Вищезазначений код **s => s.Title!.Contains(searchString**) представляє собою лямбда-вираз. Лямбда-вирази використовуються в запитах **LINQ** на основі методів як аргументи стандартних методів операторів запитів, таких як метод **Where** або **Contains** (використовується в вищезазначеному коді). Запити **LINQ** не виконуються, якщо їх визначено або змінено шляхом виклику методу, наприклад **Where**, **Contains** або **OrderBy**. Замість цього виконання запиту відкладається. Це означає, що обчислення виразу відкладається до тих пір, поки не буде виконано ітерацію його реалізованого значення або поки не буде викликаний метод **ToListAsync.**

Метод Contains виконується в базі даних, а не в наведеному вище коді C#. Регістр символів у запиті враховується в залежності від параметрів бази даних та сортування. У SQL Server метод **Contains** відповідає **SQL LIKE,** де регістр символів не враховується. У **SQLite** при параметрах сортування за замовчуванням регістр символів враховується.

Перейдіть за адресою **/Songs/Index.** Додайте до **URL**-адреси рядок запиту, наприклад, **?searchString=Hi**. Відображаються відфільтровані пісні (рисунок 5.1).

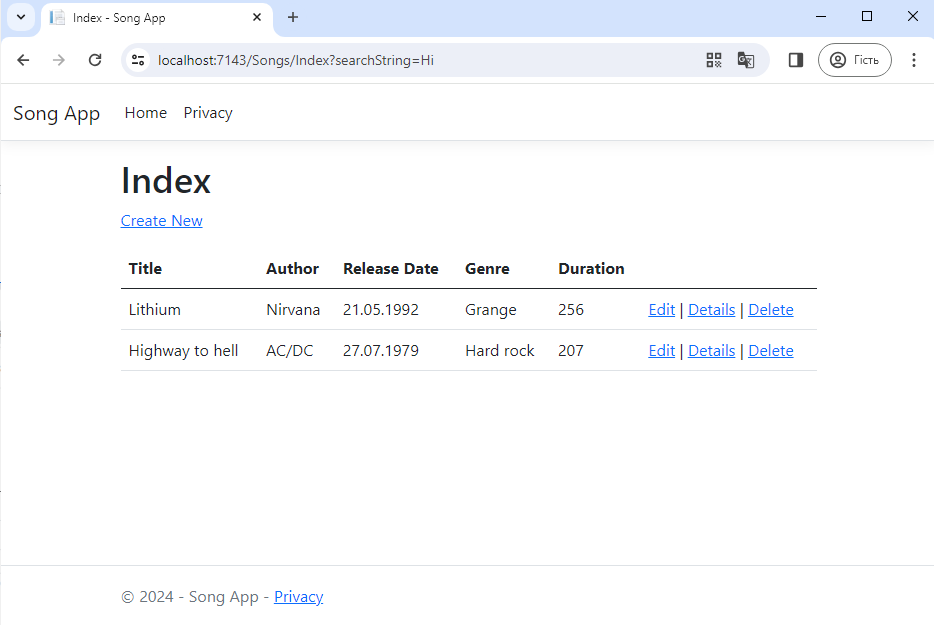


Рисунок 5.1 - Відображення відфільтрованих пісень

Якщо змінити сигнатуру метода **Index** на параметр з ім'ям id, цей параметр буде відповідати необов'язковому заповнювачу {id} для заданих за замовчуванням маршрутів у файлі **Program.cs.**

app.MapControllerRoute(

name: "default",

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

Змініть параметр на id, а всі входження searchString — на id.

Попередній метод Index:

public async Task<IActionResult> Index(string searchString)

{

if (\_context.Song == null)

{

return Problem("Entity set 'MvcSongContext.Song' is null.");

}

var Songs = from m in \_context.Song

select m;

if (!String.IsNullOrEmpty(searchString))

{

Songs = Songs.Where(s => s.Title!.Contains(searchString));

}

return View(await Songs.ToListAsync());

}

Оновлений метод Index:

public async Task<IActionResult> Index(string id)

{

if (\_context.Song == null)

{

return Problem("Entity set 'MvcSongContext.Song' is null.");

}

var Songs = from m in \_context.Song

select m;

if (!String.IsNullOrEmpty(id))

{

Songs = Songs.Where(s => s.Title!.Contains(id));

}

return View(await Songs.ToListAsync());

}

Тепер можна передати заголовок пошуку як дані маршруту (сегмент URL-адреси) замість значення рядка запиту (рисунок 5.2).

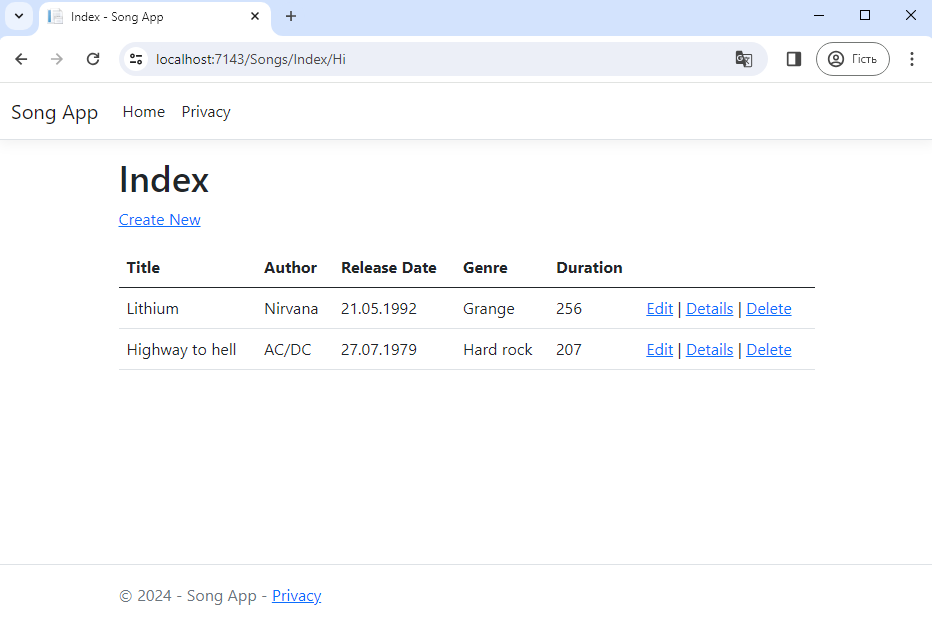


Рисунок 5.2 – Передача пошуку як даних маршруту

Тим не менше, ймовірно, користувачі не будуть змінювати **URL**-адреси кожного разу для пошуку пісень. Отже, потрібно додати елементи користувацького інтерфейсу для зручного фільтрування пісень. Якщо ви змінили сигнатуру методу **Index** для випробування передачі параметра ID з прив'язкою до маршруту, знову змініть її, щоб вона знову приймала параметр **searchString**:

public async Task<IActionResult> Index(string searchString)

{

if (\_context.Song == null)

{

return Problem("Entity set 'MvcSongContext.Song' is null.");

}

var Songs = from m in \_context.Song

select m;

if (!String.IsNullOrEmpty(searchString))

{

Songs = Songs.Where(s => s.Title!.Contains(searchString));

}

return View(await Songs.ToListAsync());

}

Відкрийте файл Views/Songs/Index.cshtml та додайте розмітку, виокремлену тегом <form> нижче:

@model IEnumerable<MvcSong.Models.Song>

@{

ViewData["Title"] = "Index";

}

<h1>Index</h1>

<p>

<a asp-action="Create">Create New</a>

</p>

<form asp-controller="Songs" asp-action="Index">

<p>

Title: <input type="text" name="SearchString" />

<input type="submit" value="Filter" />

</p>

</form>

<table class="table">

Тег **HTML <form>** використовує вспоміжну функцію тега **Form**, щоб при надсиланні форми рядок фільтра передавався в дію **Index** контролера **Songs**. Збережіть зміни та протестуйте фільтр (рисунок 5.3).

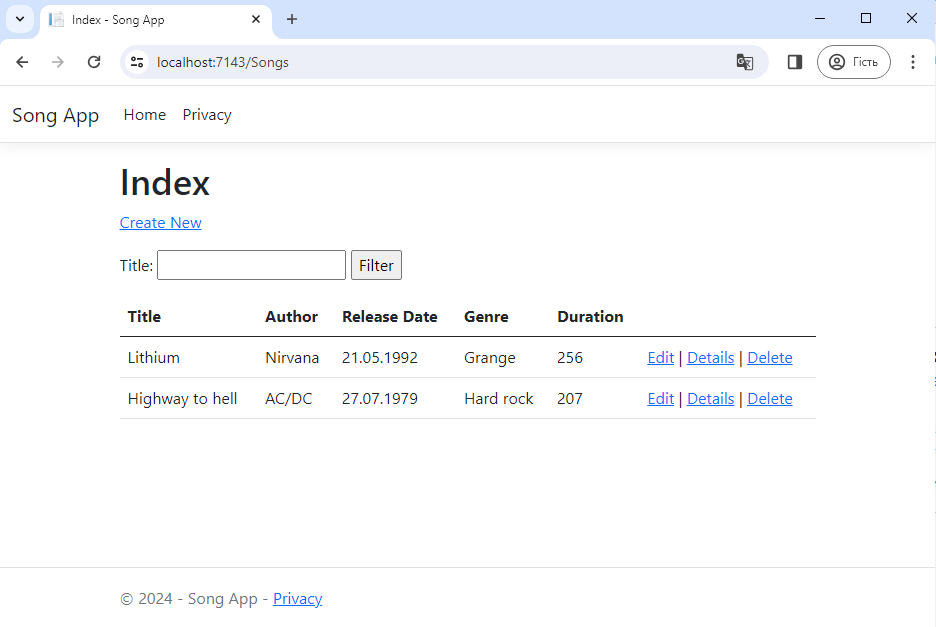


Рисунок 5.3 – Тестування фільтру

Навіть відзначений очікуванням, метод Index не включає перевантаження **[HttpPost].** Це не потрібно, оскільки метод не змінює стан застосунку і просто фільтрує дані.

Можна додати наступний метод [HttpPost] Index.

[HttpPost]

public string Index(string searchString, bool notUsed)

{

return "From [HttpPost]Index: filter on " + searchString;

}

Параметр **notUsed** використовується для створення перевантаження методу **Index**. При додаванні цього методу викликаючий метод дії буде відповідати методу **[HttpPost] Index**, і метод **[HttpPost] Index** буде мати вигляд, як показано на рисунку (Рисунок 5.4).

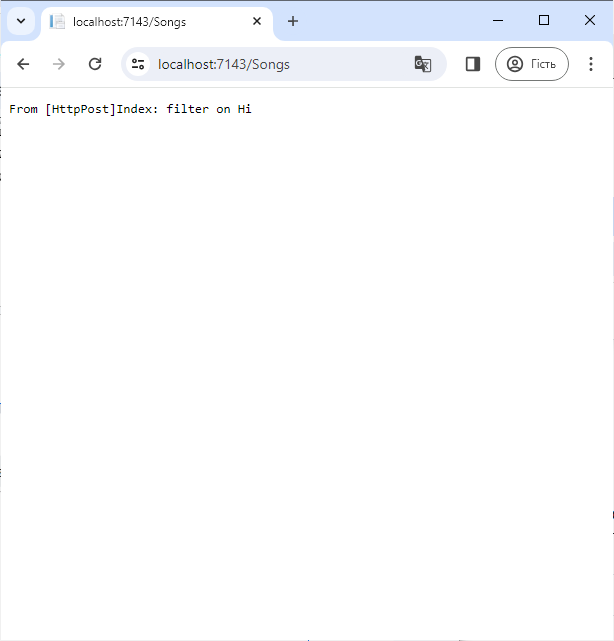


Рисунок 5.4 – Вигляд представлення

Проте при додаванні цієї версії **[HttpPost]** методу **Index** існує обмеження щодо загальної реалізації. Наприклад, вам може знадобитися додати конкретний пошук до вибраного чи відправити друзям посилання, за яким вони можуть переглянути схожий відфільтрований список пісень. Зверніть увагу, що **URL**-адреса запиту **HTTP POST** співпадає з **URL**-адресою запиту **GET** **(localhost:{ПОРТ}/Songs/Index)** - **URL**-адреса не містить інформації щодо пошуку. Дані рядка пошуку надсилаються на сервер у вигляді значення поля форми. Ви можете перевірити це за допомогою інструментів розробника браузера або Fiddler.

В тілі запиту відображається параметр пошуку та маркер **XSRF**. Допоміжна функція тега **Form** створює маркер захисту від підробки **XSRF.** В даному випадку, оскільки ми не змінюємо дані, перевірка маркера безпеки в методі контролера не потрібна.

Оскільки параметр пошуку знаходиться в тілі запиту, а не в **URL**-адресі, цю інформацію пошуку неможливо додати до закладок чи відкрити для загального доступу. Це можна виправити, вказавши, що запит має бути **HTTP GET** у файлі **Views/Songs/Index.cshtml.**

*@model IEnumerable<MvcSong.Models.Song>*

*@{*

*ViewData["Title"] = "Index";*

*}*

*<h1>Index</h1>*

*<p>*

*<a asp-action="Create">Create New</a>*

*</p>*

*<form asp-controller="Songs" asp-action="Index" method="get">*

*<p>*

*Title: <input type="text" name="SearchString" />*

*<input type="submit" value="Filter" />*

*</p>*

*</form>*

*<table class="table">*

Після відправки пошуку **URL**-адреса буде містити рядок пошукового запиту. Пошук також передається до методу **HttpGet Index**, навіть якщо визначено метод **HttpPost Index** (рисунок 5.5).

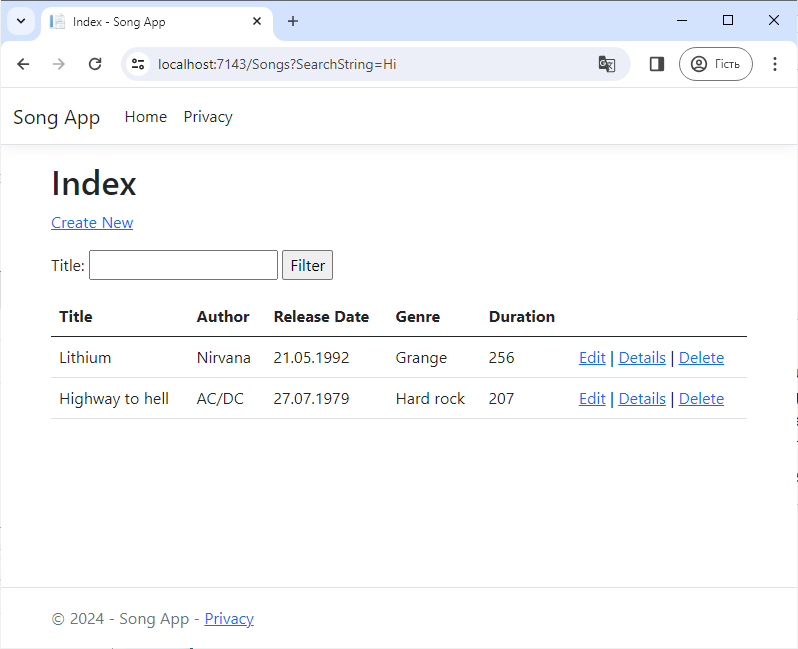


Рисунок 5.5 – Відображення результату

У наступному прикладі розмітки показано зміну тегу форми:

<form asp-controller="Songs" asp-action="Index" method="get">

**Додавання пошуку по жанру**

Додайте наступний клас SongGenreViewModel у папку Models:

*using Microsoft.AspNetCore.Mvc.Rendering;*

*using System.Collections.Generic;*

*namespace MvcSong.Models;*

*public class SongGenreViewModel*

*{*

*public List<Song>? Songs { get; set; }*

*public SelectList? Genres { get; set; }*

*public string? SongGenre { get; set; }*

*public string? SearchString { get; set; }*

*}*

Модель представлення пісень за жанром буде містити:

* Список пісень.
* Об'єкт SelectList із списком жанрів. Користувач може обрати жанр пісні із цього списку.
* Об'єкт SongGenre, що містить обраний жанр.
* SearchString, що містить текст, який користувачі вводять у поле пошуку.

Замініть метод Index у файлі SongsController.cs на наступний код:

*// GET: Songs*

*public async Task<IActionResult> Index(string SongGenre, string searchString)*

*{*

*if (\_context.Song == null)*

*{*

*return Problem("Entity set 'MvcSongContext.Song' is null.");*

*}*

*// Use LINQ to get list of genres.*

*IQueryable<string> genreQuery = from m in \_context.Song*

*orderby m.Genre*

*select m.Genre;*

*var Songs = from m in \_context.Song*

*select m;*

*if (!string.IsNullOrEmpty(searchString))*

*{*

*Songs = Songs.Where(s => s.Title!.Contains(searchString));*

*}*

*if (!string.IsNullOrEmpty(SongGenre))*

*{*

*Songs = Songs.Where(x => x.Genre == SongGenre);*

*}*

*var SongGenreVM = new SongGenreViewModel*

*{*

*Genres = new SelectList(await genreQuery.Distinct().ToListAsync()),*

*Songs = await Songs.ToListAsync()*

*};*

*return View(SongGenreVM);*

*}*

Наступний код визначає запит **LINQ**, який витягує всі жанри з бази даних.

*// Use LINQ to get list of genres.*

*IQueryable<string> genreQuery = from m in \_context.Song*

*orderby m.Genre*

*select m.Genre;*

Об'єкт **SelectList** із списком жанрів створюється шляхом проектування окремих жанрів (це необхідно для виключення повторюваних жанрів).

Коли користувач виконує пошук елемента, значення пошуку зберігається у полі пошуку.

Оновіть файл **Index.cshtml** у папці **Views/Songs/** наступним чином:

*@model MvcSong.Models.SongGenreViewModel*

*@{*

*ViewData["Title"] = "Index";*

*}*

*<h1>Index</h1>*

*<p>*

*<a asp-action="Create">Create New</a>*

*</p>*

*<form asp-controller="Songs" asp-action="Index" method="get">*

*<p>*

*<select asp-for="SongGenre" asp-items="Model.Genres">*

*<option value="">All</option>*

*</select>*

*Title: <input type="text" asp-for="SearchString" />*

*<input type="submit" value="Filter" />*

*</p>*

*</form>*

*<table class="table">*

*<thead>*

*<tr>*

*<th>*

*@Html.DisplayNameFor(model => model.Songs![0].Title)*

*</th>*

*<th>*

*@Html.DisplayNameFor(model => model.Songs![0].ReleaseDate)*

*</th>*

*<th>*

*@Html.DisplayNameFor(model => model.Songs![0].Genre)*

*</th>*

*<th>*

*@Html.DisplayNameFor(model => model.Songs![0].Price)*

*</th>*

*<th></th>*

*</tr>*

*</thead>*

*<tbody>*

*@foreach (var item in Model.Songs!)*

*{*

*<tr>*

*<td>*

*@Html.DisplayFor(modelItem => item.Title)*

*</td>*

*<td>*

*@Html.DisplayFor(modelItem => item.Author)*

*</td>*

*<td>*

*@Html.DisplayFor(modelItem => item.ReleaseDate)*

*</td>*

*<td>*

*@Html.DisplayFor(modelItem => item.Genre)*

*</td>*

*<td>*

*@Html.DisplayFor(modelItem => item.Duration)*

*</td>*

*<td>*

*<a asp-action="Edit" asp-route-id="@item.Id">Edit</a> |*

*<a asp-action="Details" asp-route-id="@item.Id">Details</a> |*

*<a asp-action="Delete" asp-route-id="@item.Id">Delete</a>*

*</td>*

*</tr>*

}

</tbody>

</table>

Перевірте лямбда-вираз, який використовується у наступному допоміжному **HTML**-методі:

*@Html.DisplayNameFor(model => model.Songs![0].Title)*

У попередньому коді допоміжний метод **HTML** **DisplayNameFor** перевіряє властивість **Title**, зазначену в лямбда-виразі, та визначає відображуване ім'я. Оскільки лямбда-вираз перевіряється, а не обчислюється, у випадку, якщо **model, model.Songs** або **model.Songs[0]** мають значення **null** або є порожніми, не відбувається порушення прав доступу. При обчисленні лямбда-виразу (наприклад, **@Html.DisplayFor(modelItem => item.Title**)) обчислюються значення властивостей моделі **model.Songs.** Після **!** використовується оператор, що дозволяє значення **null**, який вказує, що **Songs** не дорівнює **null**.

Перевірити роботу застосунку можна, виконавши пошук за жанром, за назвою пісні та обома цими параметрами:

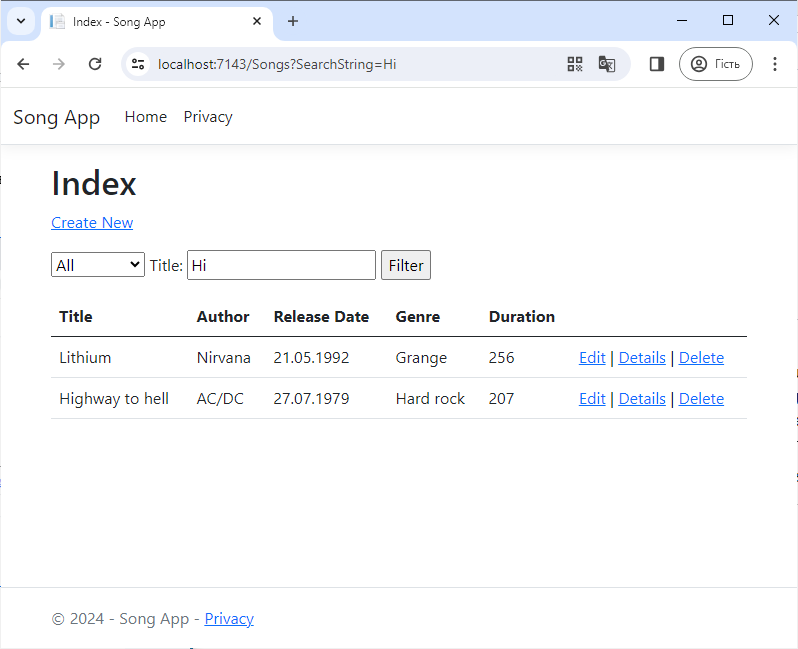


Рисунок 5.6 – Перевірка пошуку

**Додавання нового поля у додаток MVC ASP.NET**

Додавання властивості **"Rating"** у модель **"Song".** Додайте властивість **"Rating"** у файл **Models/Song.cs:**

*using Microsoft.AspNetCore.Components.Forms;*

*using System.ComponentModel.DataAnnotations;*

*namespace MvcSong.Models*

*{*

*public class Song*

*{*

*public int Id { get; set; }*

*public string? Title { get; set; }*

*public string? Author { get; set; }*

*[Display(Name = "Release Date")]*

*[DataType(DataType.Date)]*

*public DateTime? ReleaseDate { get; set; }*

*public string? Genre { get; set; }*

*public int Duration { get; set; }*

*public string? Rating { get; set; }*

*}*

*}*

Оскільки в класі Song було додано нове поле, необхідно оновити список прив'язки властивостей, включивши нову властивість. У файлі **SongsController.cs** оновіть атрибут **[Bind]** для методів дії **Create** та **Edit**, включивши властивість **Rating**:

[Bind("Id,Title,ReleaseDate,Genre,Price,Rating")]

Оновіть шаблони представлень для відображення, створення та редагування нової властивості **"Rating"** у браузері.

Змініть файл /Views/Songs/Index.cshtml та додайте поле "Rating":

<table class="table">

<thead>

<tr>

<th>

@Html.DisplayNameFor(model => model.Songs![0].Title)

</th>

<th>

@Html.DisplayNameFor(model => model.Songs![0].Author)

</th>

<th>

*@Html.DisplayNameFor(model => model.Songs![0].ReleaseDate)*

*</th>*

*<th>*

*@Html.DisplayNameFor(model => model.Songs![0].Genre)*

*</th>*

*<th>*

*@Html.DisplayNameFor(model => model.Songs![0].Duration)*

*</th>*

*<th>*

*@Html.DisplayNameFor(model => model.Songs![0].Rating)*

*</th>*

*<th></th>*

*</tr>*

*</thead>*

*<tbody>*

*@foreach (var item in Model.Songs!) {*

*<tr>*

*<td>*

*@Html.DisplayFor(modelItem => item.Title)*

*</td>*

*<td>*

*@Html.DisplayFor(modelItem => item.Author)*

*</td>*

*<td>*

*@Html.DisplayFor(modelItem => item.ReleaseDate)*

*</td>*

*<td>*

*@Html.DisplayFor(modelItem => item.Genre)*

*</td>*

*<td>*

*@Html.DisplayFor(modelItem => item.Duration)*

*</td>*

*<td>*

*@Html.DisplayFor(modelItem => item.Rating)*

*</td>*

*<td>*

*<a asp-action="Edit" asp-route-id="@item.Id">Edit</a> |*

*<a asp-action="Details" asp-route-id="@item.Id">Details</a> |*

*<a asp-action="Delete" asp-route-id="@item.Id">Delete</a>*

*</td>*

*</tr>*

*}*

*</tbody>*

*</table>*

Оновіть /Views/Songs/Create.cshtml, додавши поле "Rating".

*<div class="form-group">*

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

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

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

*</div>*

Оновіть інші шаблони.

Оновіть клас **SeedData** так, щоб він надавав значення нового стовпця. Нижче наведений приклад змін, які необхідно внести для кожного нового Song.

*new Song*

*{*

*Title = "Don't Let Me Down",*

*Author = "The Beatles",*

*ReleaseDate = DateTime.Parse("1969-4-11"),*

*Genre = "Rock",*

*Duration = 215,*

*Rating = "High",*

*}*

Для роботи застосунку необхідно оновити базу даних, включивши в неї нове поле. Якщо вона запущена, виникає наступне виключення **SqlException**:

SqlException: Invalid column name Rating.

Ця помилка пов'язана з тим, що оновлений клас моделі **Song** відрізняється від схеми таблиці **Song** в існуючій базі даних. (В таблиці бази даних відсутнє поле **Rating**.)

Цю помилку можна виправити кількома способами:

1. Ви можете автоматично видалити та повторно створити базу даних за допомогою **Entity Framework** на основі нової схеми класу моделі. Цей підхід зручний на ранніх етапах циклу розробки, коли всі дії виконуються з тестовою базою даних. У цьому випадку розвиток моделі та схеми бази даних відбувається одночасно. Недоліком такого підходу є втрата існуючих даних в базі, тому використання його в робочій базі даних є неможливим. При розробці застосунку часто використовується ініціалізатор для автоматичного заповнення бази тестовими даними. Це хороший підхід на ранніх етапах розробки та при використанні **SQLite**.

2. Можна явно змінити схему існуючої бази даних відповідно до нових класів моделі. Перевагою цього підходу є те, що зберігаються всі дані. Цю зміну можна виконати як вручну, так і за допомогою відповідного скрипта бази даних.

3. Ви можете оновити схему бази даних за допомогою **Code First Migrations.**

У меню **"Сервіс"** послідовно виберіть пункти **"Диспетчер пакетів NuGet" > "Консоль диспетчера пакетів".**

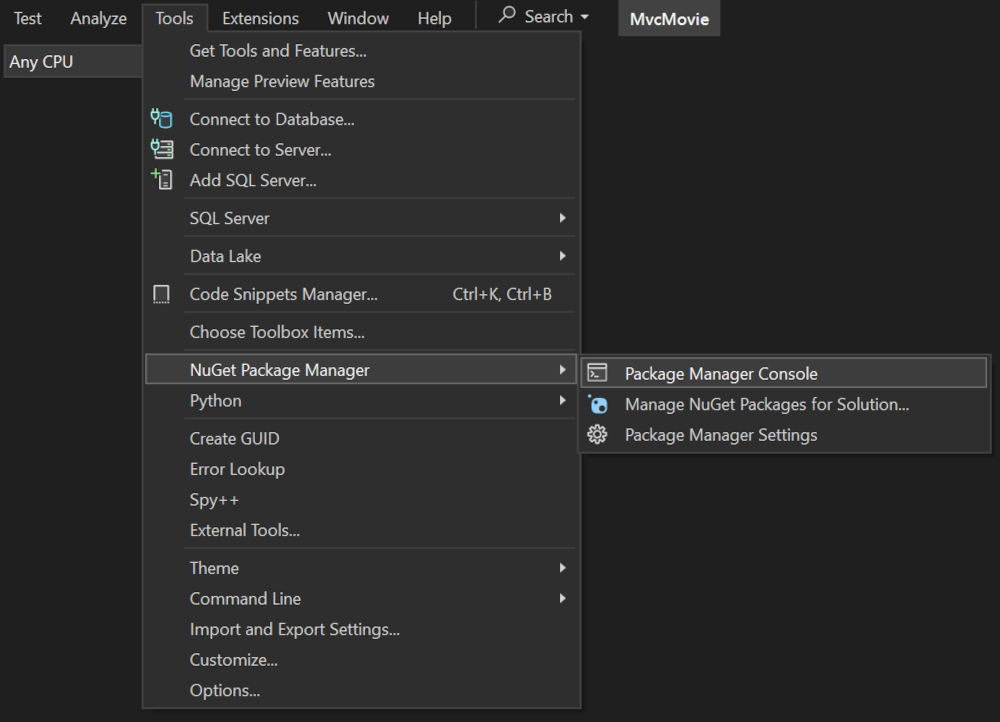


Рисунок 5.7 – Вибір "Диспетчера пакетів NuGet"

В консоль введіть наступні команди:

Add-Migration Rating

Update-Database

Команда **Add-Migration** вказує платформі міграції перевірити поточну модель **Song** з поточною схемою бази даних **Song** і створити необхідний код для перенесення бази даних до нової моделі.

Як ім'я файлу міграції використовується випадкове ім'я **"Rating".** Рекомендується присвоювати цьому файлу зрозуміле ім'я.

Якщо видалите всі записи з бази даних, при ініціалізації вона буде заповнена значеннями, і в неї буде включено поле **Rating**.

Запустіть застосунок і перевірте можливість створення, редагування та відображення пісень з використанням поля **Rating**.

**1.3 Завдання до лабораторної роботи**

Відповідно до тематики курсового проекту здійснити добавлення необхідних полів, а також реалізувати пошук за певними критеріями: товар, ціна, бренд, колір, назва, тип, локація, прізвище, номер телефону, адреса тощо.

* 1. **Контрольні запитання**

1. Як здійснити додавання нових полів до застосунку?
2. Що робить команда Add-Migration?
3. Як можна автоматично видалити та повторно створити базу даних за допомогою Entity Framework?
4. Чи можна явно змінити схему існуючої бази даних відповідно до нових класів моделі?
5. Що означає таке виключення SqlException?
6. Як оновити шаблони представлень для відображення, створення та редагування нової властивості у браузері?
7. Що таке маркер захисту від підробки XSRF?
8. Для чого використовується Code First Migrations.?

**Literature: [1, 5, 6, 14, 15]**

**Лабораторна робота 6**

**Авторизація на основі ролей.**

**Мета:** Розробити програмно у створюваному проєкті можливість здійснення авторизації відповідно до визначеної ролі.

**1.1 Теоретичні основи та рекомендації до виконання**

Відповідно до вимог кожному користувачу належить та чи інша роль (наприклад, зареєстрований, незареєстрований користувач, адміністратор тощо). Наприклад, Продавець може мати роль **AdministratorUser**, а зареєстрований користувач може мати лише роль **User**. Створення та управління цими ролями залежить від резервного сховища процесу авторизації. Ролі надаються розробнику через **IsInRole** метод класу **ClaimsPrincipal**. **AddRoles** необхідно додати до служби ролей.

Хоча ролі є твердженнями, не всі твердження є ролями. Залежно від видавця посвідчень роль може мати колекція користувачів, які можуть застосовувати твердження для учасників групи, а також фактичне твердження для посвідчення. Однак, твердження повинні бути відомостями про окремого користувача. Використання ролей для додавання тверджень користувачеві може заплутати межу між користувачем та їхніми окремими твердженнями. Ця плутанина полягає в тому, що шаблони **SPA** не призначені для ролей. Крім того, для організацій, що переносяться з локальної застарілої системи, поширення ролей протягом багатьох років може означати, що твердження ролі може бути занадто великим, щоб міститися в маркері, який можна використовувати **spAs**.

**1.2 Приклади виконання завдання**

1.Для додавання служб ролей до **Identity** необхідно зареєструвати служби **Program.cs** авторизації на основі ролей, викликавши **AddRoles** тип ролі в конфігурації застосунку **Identity**. Тип ролі в наступному прикладі: **IdentityRole**

*builder.Services.AddDefaultIdentity<IdentityUser>( ... )*

*.AddRoles<IdentityRole>()*

*...*

2.Додавання перевірки ролей. Перевірка авторизації на основі ролей:

Ролі є декларативними та вказують, який поточний користувач належить до тієї чи іншої ролі, щоб мати доступу до запитаного ресурсу. Застосовуються до Razor сторінок, контролерів або дій у контролері. Не вдається застосувати на Razor рівні обробника сторінки, їх необхідно застосувати до сторінки.

Наприклад, наступний код обмежує доступ до будь-яких дій **AdministrationController** для користувачів, які є членом **Administrator** ролі:

*[Authorize(Roles = "Administrator")]*

*public class AdministrationController : Controller*

*{*

*public IActionResult Index() =>*

*Content("Administrator");*

*}*

Кілька ролей можна вказати у вигляді розділеного комами списку:

*[Authorize(Roles = "HRManager,Finance")]*

*public class SalaryController : Controller*

*{*

*public IActionResult Payslip() =>*

*Content("HRManager || Finance");*

*}*

Доступно **SalaryController** тільки користувачам, які є членами **HRManager** ролі або **Finance** ролі.

При застосуванні декількох атрибутів доступ до користувача повинен бути членом усіх зазначених ролей. У наступному прикладі потрібно, щоб користувач був членом **PowerUserControlPanelUserі** ролі:

*[Authorize(Roles = "PowerUser")]*

*[Authorize(Roles = "ControlPanelUser")]*

*public class ControlPanelController : Controller*

*{*

*public IActionResult Index() =>*

*Content("PowerUser && ControlPanelUser");*

*}*

Доступ до дії можна обмежити, застосовуючи додаткові атрибути авторизації ролі на рівні дії:

*[Authorize(Roles = "Administrator, PowerUser")]*

*public class ControlAllPanelController : Controller*

*{*

*public IActionResult SetTime() =>*

*Content("Administrator || PowerUser");*

*[Authorize(Roles = "Administrator")]*

*public IActionResult ShutDown() =>*

*Content("Administrator only");*

*}*

На попередньому **ControlAllPanelController** контролері:

Administrator Члени ролі або **PowerUser** ролі можуть отримати доступ до контролера і **SetTime** дії.

Доступ до дії може отримати **ShutDown** тільки члени **Administrator** ролі.

Контролер може бути заблокований, але дозволити анонімний доступ до окремих дій, який не пройшов перевірку автентичності:

*[Authorize]*

*public class Control3PanelController : Controller*

*{*

*public IActionResult SetTime() =>*

*Content("[Authorize]");*

*[AllowAnonymous]*

*public IActionResult Login() =>*

*Content("[AllowAnonymous]");*

*}*

Для Razor Сторінок [Authorize] можна застосувати:

1. Використання згоди
2. Застосування до екземпляру [Authorize]PageModel:

*[Authorize(Policy = "RequireAdministratorRole")]*

*public class UpdateModel : PageModel*

*{*

*public IActionResult OnPost() =>*

*Content("OnPost RequireAdministratorRole");*

*}*

Важливо пам’ятати, що **Атрибути фільтра**, включно з **AuthorizeAttribute**, можуть застосовуватися тільки до **PageModel** і не можуть застосовуватися до певних методів обробника сторінок.

3.Перевірка ролей на основі політик

Вимоги до ролі також можна описати за допомогою синтаксису політики, де розробник реєструє політику під час запуску застосунку в межах конфігурації служби авторизації. Зазвичай це відбувається в **Program.cs** файлі (рисунок 2.1):

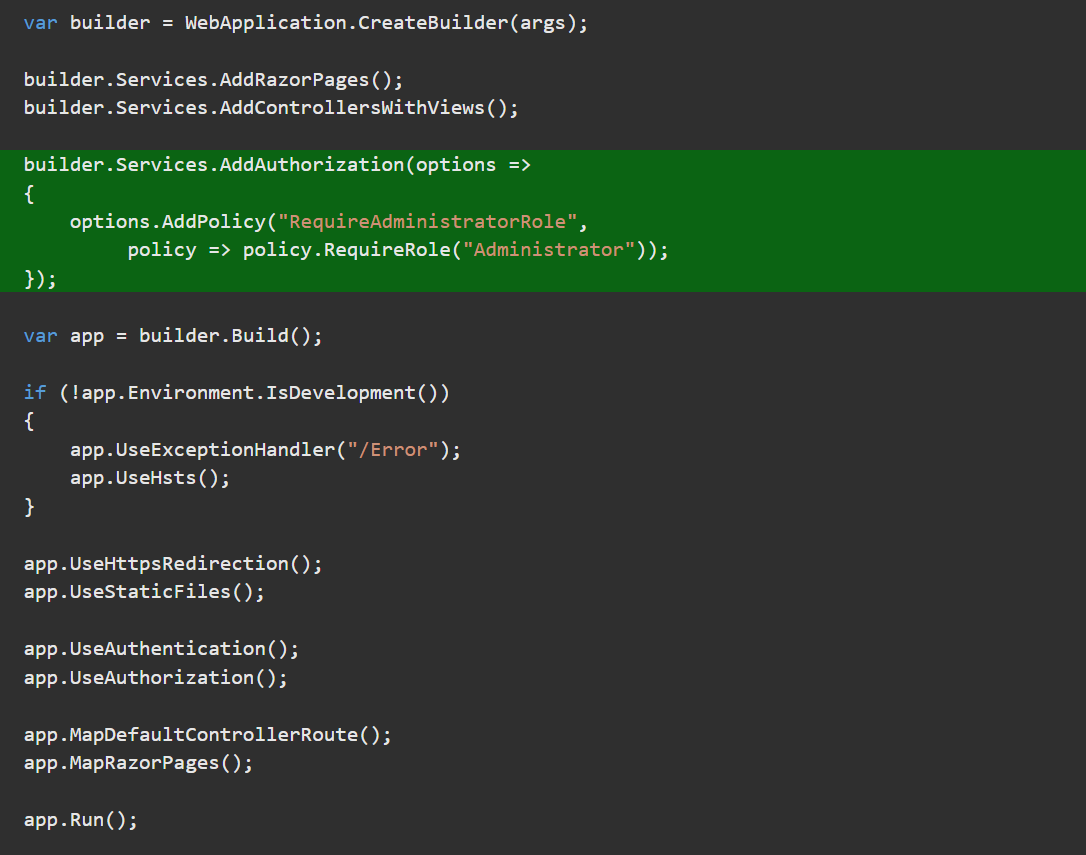
**

Рисунок 2.1 – Файл Program.cs

Політики застосвуються за допомогою **Policy** властивості атрибута **[Authorize]:**

*[Authorize(Policy = "RequireAdministratorRole")]*

*public IActionResult Shutdown()*

*{*

*return View();*

*}*

Щоб вказати кілька дозволених ролей в обов'язкових вимогах, потрібно вказати їх як параметри методу **RequireRole**:

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

*builder.Services.AddRazorPages();*

*builder.Services.AddControllersWithViews();*

*builder.Services.AddAuthorization(options =>*

*{*

*options.AddPolicy("ElevatedRights", policy =>*

*policy.RequireRole("Administrator", "PowerUser", "BackupAdministrator"));*

*});*

*var app = builder.Build();*

Наведений вище код дозволяє користувачам, які належать до ролей або **BackupAdministrator** ролям.**AdministratorPowerUser**.

**1.3 Завдання до лабораторної роботи**

Відповідно до тематики курсового проекту здійснити реалізації авторизації на основі наступних ролей:

1. Авторизований користувач.
2. Неавторизований користувач.
3. Адміністратор ресурсу.
4. Модератор відгуків та коментарів.

**1.4 Контрольні запитання**

1. Що таке авторизація?
2. Відмінність між авторизацією та аутентифікацією.
3. Яким чином можна описати вимоги до ролі?
4. В чому полягає суть розмежування за ролями?
5. Що таке роль в проекті?
6. Як можна обмежити доступ до дії за допомогою ролі?
7. Як здійснити додавання перевірки ролей?
8. Що необхідно зробити для додавання служб ролей?

**Literature: [1, 5, 6, 14, 15]**

**Лабораторна робота №7**

**Додавання можливостей валідації в застосунок MVC ASP.NET**

**Мета:** Здійснення перевірки валідації у створюваному проєкті

**1.1 Теоретичні основи та рекомендації до виконання**

Принцип **DRY** ("Don't Repeat Yourself" - не повторюйся) є одним із основоположних принципів розробки в моделі **MVC**. У моделі **ASP.NET Core MVC** рекомендується визначати функціональні можливості або поведінку лише один раз, а потім відображати їх в інших частинах застосунку. Це дозволяє мінімізувати обсяг коду, а також зменшує ризик помилок та спрощує його тестування та підтримку.

Яскравим прикладом застосування принципу "Не повторюйся" є підтримка валідації, яка реалізується в моделі **MVC** та на платформі **Entity Framework Core Code First.**

Правила валідації декларативно визначаються в одному місці (у класі моделі), а потім застосовуються в межах всього застосунку.

Атрибути валідації визначають поведінку для властивостей моделі, до яких вони застосовуються:

Атрибути **Required** і **MinimumLength** вказують, що властивість повинна мати значення. Тим не менш, для задоволення вимог валідації користувачу достатньо ввести пробіл.

Атрибут **RegularExpression** обмежує набір допустимих символів для введення. У вказаному вище коді для **Genre**:

* повинні використовуватися лише букви;
* перша буква повинна бути прописною;
* Пробіли допускаються, але цифри та спеціальні символи - ні.

Для атрибуту **RegularExpression Rating**:

* перший символ повинен бути прописною літерою;
* допускаються спеціальні символи та цифри, а також подальші пробіли. Значення **"PG-13"** допустиме для рейтингу, але неприпустиме для жанру.

Атрибут **StringLength** дозволяє вказати максимальну і, за потреби, мінімальну довжину рядкової властивості.

Типи значень (наприклад**, decimal, int, float, DateTime**) за своєю природою є обов'язковими і не вимагають атрибута **[Required].**

Наявність правил валідації, які автоматично застосовуються **ASP.NET Core**, допомагає підвищити рівень надійності застосунку. Це також гарантує, що у будь-якому випадку будуть виконані всі перевірки, і в базі даних не залишаться випадково пошкоджені дані.

**1.2 Приклади виконання завдання**

Додавання правил валідації до моделі «Пісні»

Простір імен **DataAnnotations** надає набір вбудованих атрибутів валідації, які декларативно застосовуються до класу або властивості. Крім того, **DataAnnotations** містить атрибути форматування (такі як **DataType**), які забезпечують форматування і не призначені для валідації.

Оновіть клас Song, щоб скористатися вбудованими атрибутами валідації **Required, StringLength, RegularExpression** та атрибутом форматування **DataType**.

*using System;*

*using System.ComponentModel.DataAnnotations;*

*using System.ComponentModel.DataAnnotations.Schema;*

*namespace MvcSong.Models;*

*public class Song*

*{*

*public int Id { get; set; }*

*[StringLength(60, MinimumLength = 3)]*

*[Required]*

*public string? Title { get; set; }*

*[Display(Name = "Release Date")]*

*[DataType(DataType.Date)]*

*public DateTime ReleaseDate { get; set; }*

*[Range(1, 100)]*

*[DataType(DataType.Currency)]*

*[Column(TypeName = "decimal(18, 2)")]*

*public decimal Price { get; set; }*

*[RegularExpression(@"^[A-Z]+[a-zA-Z\s]\*$")]*

*[Required]*

*[StringLength(30)]*

*public string? Genre { get; set; }*

*[RegularExpression(@"^[A-Z]+[a-zA-Z0-9""'\s-]\*$")]*

*[StringLength(5)]*

*[Required]*

*public string? Rating { get; set; }*

*}*

Запустіть застосунок та перейдіть до контролера пісень.

Клацніть посилання "Створити", щоб додати нову пісню. Введіть у форму які-небудь неприпустимі значення. Якщо функція перевірки jQuery на боці клієнта виявляє помилку, відповідна інформація відображається у відповідному повідомленні (рисунок 7.1).

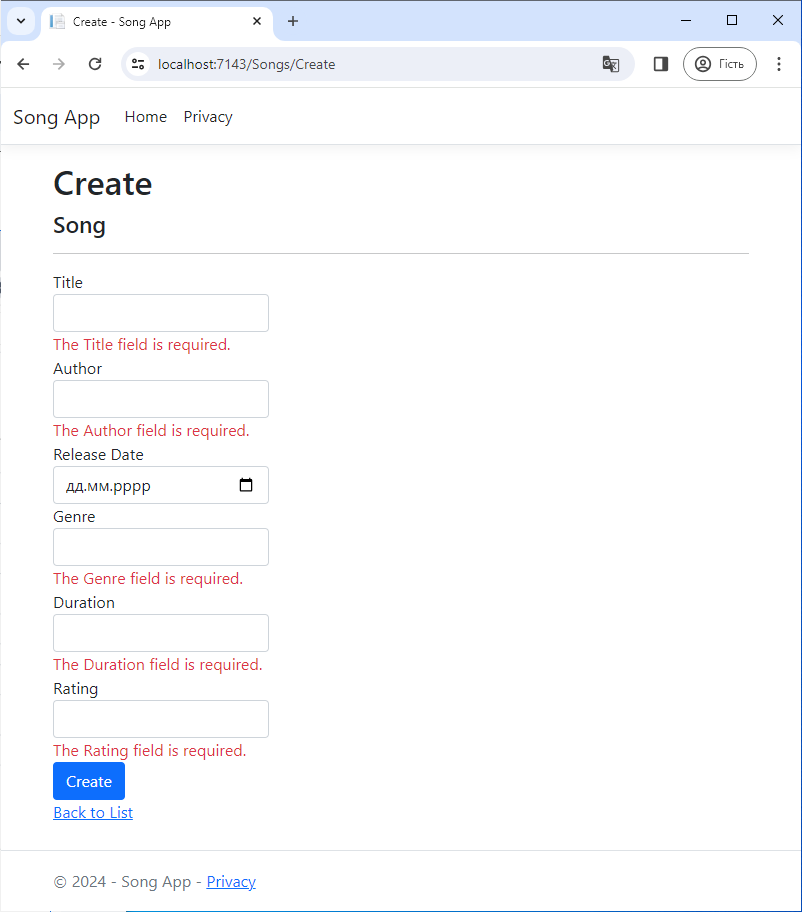


Рисунок 7.1 - Валідація

Зверніть увагу, що для кожного поля, яке містить неприпустиме значення, у формі автоматично відображається відповідне повідомлення про помилку перевірки. Ці помилки застосовуються як на боці клієнта (за допомогою **JavaScript і jQuery**), так і на боці сервера (якщо користувач вимикає **JavaScript**).

Важливою перевагою є те, що для реалізації цього інтерфейсу перевірки користувачеві не потрібно змінювати код у класі **SongsController** або у представленні **Create.cshtml.** В контролері та представленнях, створених в рамках цього керівництва, автоматично застосовуються правила перевірки, визначені застосуванням атрибутів до властивостей класу моделі **Song**. При перевірці за допомогою методу дії **Edit** застосовуються ті ж правила.

Дані форми надсилаються на сервер лише після виправлення всіх помилок перевірки на боці клієнта. Щоб перевірити це, встановіть точку зупинки в методі **HTTP Post** за допомогою інструменту **Fiddler** або засобів розробника **F12.**

Далі показано, як створюється інтерфейс перевірки без оновлення коду контролера чи представлень. У наступному прикладі коду є два методи **Create**.

*// GET: Songs/Create*

*public IActionResult Create()*

*{*

*return View();*

*}*

*// POST: Songs/Create*

*// To protect from overposting attacks, enable the specific properties you want to bind to.*

*// For more details, see http://go.microsoft.com/fwlink/?LinkId=317598.*

*[HttpPost]*

*[ValidateAntiForgeryToken]*

*public async Task<IActionResult> Create([Bind("Id,Title,Author,ReleaseDate,Genre,Price,Rating")] Song Song)*

*{*

*if (ModelState.IsValid)*

*{*

*\_context.Add(Song);*

*await \_context.SaveChangesAsync();*

*return RedirectToAction(nameof(Index));*

*}*

*return View(Song);*

*}*

Перший метод **Create (HTTP GET)** відображає початкову форму створення. Друга версія **([HttpPost])** обробляє відправку форми. У другому методі **Create** (версія **[HttpPost])** викликається **ModelState.IsValid**, який визначає наявність помилок перевірки в пісні. При виклику цього методу оцінюються всі атрибути перевірки, які були застосовані до об'єкта. З наявністю помилок перевірки в об'єкті метод **Create** повторно відображає форму. Якщо помилок немає, метод зберігає нову пісню в базі даних. У цьому прикладі форма передається на сервер лише після виправлення всіх помилок перевірки, виявлених на стороні клієнта. Другий метод **Create** не викликається до тих пір, поки на стороні клієнта присутні помилки перевірки. При вимкненні **JavaScript** у браузері також вимикається перевірка на стороні клієнта. У цьому випадку ви можете перевірити метод **HTTP POST CreateModelState.IsValid**, який виявляє наявність помилок перевірки.

Можна встановити точку зупинки в метод **[HttpPost] Create** і переконатися, що його не викликається і дані форми не передаються, якщо на стороні клієнта є помилки перевірки. Якщо вимкнути **JavaScript** у браузері та надіслати форму із помилками, буде досягнута точка зупинки. Без **JavaScript** ви все одно отримаєте повну перевірку. На рисунку 7.2 показано, як вимкнути **JavaScript** у браузері **Firefox**.

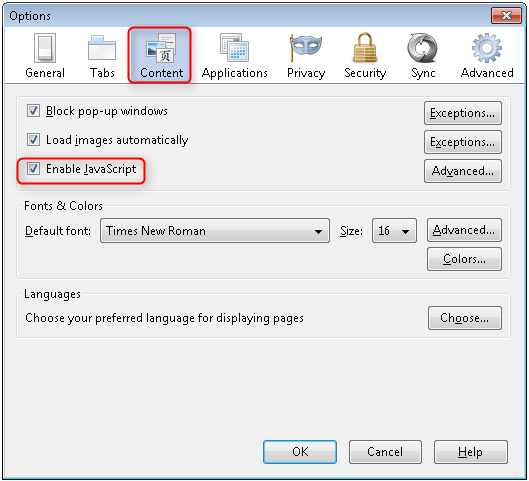


Рисунок 7.2 – Вимикання JavaScript у браузері Firefox

На рисунку 7.3 показано, як вимкнути JavaScript у браузері Chrome.

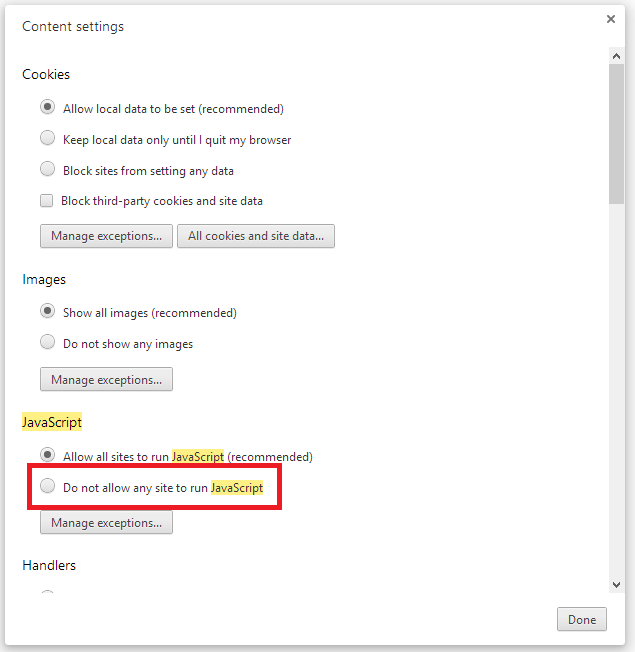


Рисунок 7.3 – Вимикання JavaScript у браузері Chrome.

Після вимкнення **JavaScript** передайте недопустимі дані та запустіть відлагодження у покроковому режимі (рисунок 7.4).

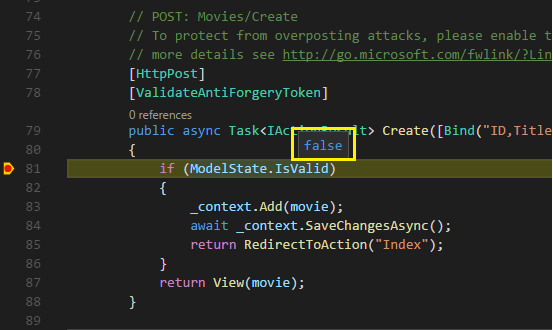


Рисунок 7.4 – Відлагодження у покроковому режимі

Частина розмітки представлення показана в наступному коді файлу **Create.cshtml:**

*<h4>Song</h4>*

*<hr />*

*<div class="row">*

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

*<form asp-action="Create">*

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

*<div class="form-group">*

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

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

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

*</div>*

*@\*Markup removed for brevity.\*@*

Вищезазначена розмітка використовується методами дій для відображення початкової форми та повторного виведення форми у випадку помилки.

Допоміжна функція тегу **Input** використовує атрибути **DataAnnotations** та створює **HTML**-атрибути, необхідні для перевірки **jQuery** на клієнтській стороні. Допоміжна функція тегу **Validation** виводить помилки перевірки.

Цей підхід зручний тим, що або контролер, або шаблон представлення **Create** нічого не знають про фактичні правила перевірки або виводяться повідомлення про помилки. Правила перевірки та рядки помилок вказуються лише в класі **Song**. Такі ж правила перевірки автоматично застосовуються до представлення **Edit** та будь-яких інших представлень моделі, які ви можете створювати або редагувати.

За необхідності можна змінити логіку перевірки в одному місці, додаючи атрибути перевірки до моделі (у цьому прикладі - до класу Song). Вам не доведеться турбуватися про невідповідність застосування правил в різних частинах застосунку, оскільки вся логіка перевірки буде визначена в одному місці і почне застосовуватися по всьому застосунку. Це дозволяє максимально оптимізувати код та забезпечує зручність його вдосконалення та підтримки. Крім того, таким чином ви повністю дотримуєтеся вимог принципу "Не повторюйся".

**Використання атрибутів DataType**

Відкрийте файл **Song.cs** та перевірте клас **Song**. У просторі імен **System.ComponentModel.DataAnnotations,** на доповнення до вбудованих атрибутів перевірки, надаються атрибути форматування. До полів з датою випуску та ціною вже застосовано значення переліку **DataType**. У наступному коді показані властивості **ReleaseDate** та **Price** з відповідним атрибутом **DataType**.

*[Display(Name = "Release Date")]*

*[DataType(DataType.Date)]*

*public DateTime ReleaseDate { get; set; }*

Атрибути **DataType** надають поради щодо форматування даних лише для перегляду, а також інші елементи та атрибути, наприклад, **<a>** для **URL**-адрес або **<a href="mailto:EmailAddress.com">** для електронної пошти. Для перевірки формату даних використовуйте атрибут **RegularExpression**. Атрибут **DataType** дозволяє вказати тип даних з більш точним визначенням, ніж вбудований тип бази даних, але не призначений для перевірки. У цьому випадку потрібно відстежувати лише дату, а не час. У переліку **DataType** є багато типів даних, таких як **Date**, **Time, PhoneNumber, Currency, EmailAddress** та інші. Атрибут **DataType** також забезпечує автоматичне надання функціоналу для певних типів у застосунку. Наприклад, для **DataType.EmailAddress** може бути створено посилання mailto:. Також в браузерах, що підтримують **HTML5**, може бути наданий селектор дати для **DataType.Date.** Атрибути DataType створюють атрибути **HTML 5 data**-, які використовуються браузерами з підтримкою **HTML 5.** Атрибути **DataType** не призначені для перевірки.

**DataType.Date** не визначає формат відображеної дати. За замовчуванням дані відображаються за форматами, встановленими в параметрах **CultureInfo** сервера.

За допомогою атрибута **DisplayFormat** можна явно вказати формат дати:

*[DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]*

*public DateTime ReleaseDate { get; set; }*

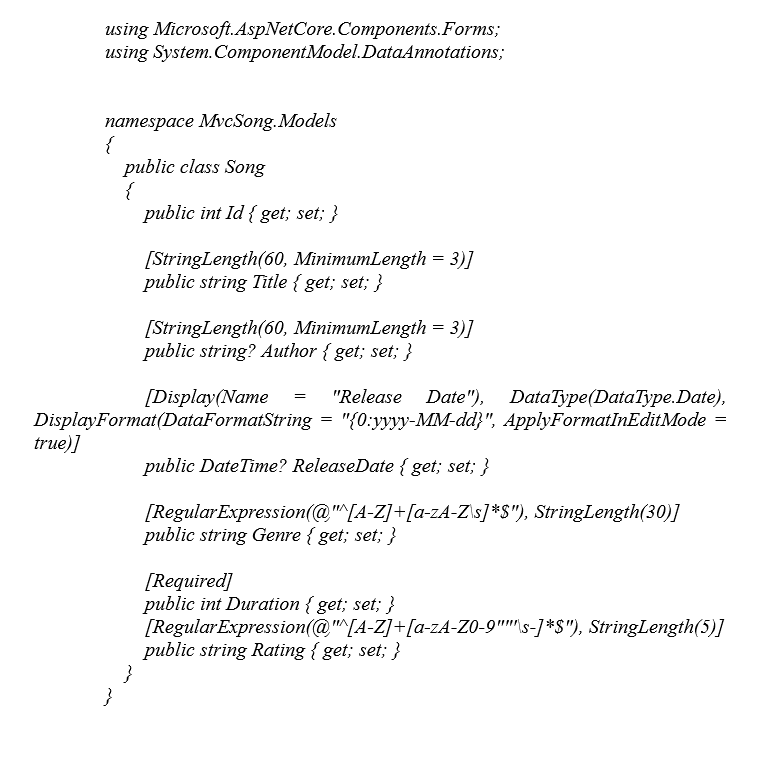
Параметр ApplyFormatInEditMode вказує, що формат також повинен застосовуватися при відображенні значення у текстовому полі для редагування. (У деяких випадках така поведінка може бути небажаною. Наприклад, у текстовому полі для редагування грошових значень зазвичай не потрібно відображати позначення валюти.)

Атрибут **DisplayFormat** може використовуватися окремо, але найчастіше рекомендується застосовувати його разом з атрибутом **DataType**. Атрибут **DataType** передає семантику даних (на відміну від способу їх виведення на екран) і надає наступні переваги порівняно з атрибутом **DisplayFormat**:

* Підтримка функцій **HTML5** в браузері (відображення елемента керування календарем, відповідного мовному стандарту символу валюти, посилань електронної пошти і т. д.).
* За замовчуванням формат відображення даних в браузері визначається відповідно до встановленого мовного стандарту.
* За допомогою атрибута **DataType** модель **MVC** може вибрати відповідний шаблон поля для відображення даних (**DisplayFormat** при окремому використанні базується на рядковому шаблоні).

Щоб використовувати атрибут **Range** з **DateTime**, слід вимкнути перевірку дат **jQuery**. Зазвичай рекомендується уникати компіляції моделей з фіксованими датами, тому атрибути **Range** та **DateTime** слід використовувати з великою обережністю.

У наступному коді продемонстровано об'єднання атрибутів в одному рядку:

**

**1.3 Завдання до лабораторної роботи**

Відповідно до тематики курсового проекту здійснити реалізацію валідації за відповідними атрибутами: датою, часом, автором, ім’ям, прізвищем, електронною поштою, мобільним телефоном, адресою тощо.

**1.4 Контрольні запитання**

1. Що таке валідація?
2. Назвати основні принципи валідації.
3. На що вказує параметр ApplyFormatInEditMode?
4. Яка роль атрибутів DataType?
5. Що визначають атрибути валідації?
6. Які є атрибути форматування?
7. Що відображає метод Create?
8. Для чого необхідний атрибут DisplayFormat?

**Literature: [1, 5, 6, 14, 15]**

**Лабораторна робота №8**

**Локалізація та мультимовність**

**Мета:** Здійснення локалізації та використання можливості мультимовності у створюваному проєкті

**1.1 Теоретичні основи та рекомендації до виконання**

Локалізація вебресурсу - переклад контенту та інтерфейсу сайту (кнопок, елементів навігації, форм реєстрації тощо) на мову цільової аудиторії у будь-якій країні. Простіше кажучи, це створення ще однієї мовної версії сайту.

Оптимально обраний шлях розподілу сайтів за мовним і локальним принципом - один з найважливіших чинників у просуванні, який значно впливає на успіх проекту в цілому, а також мінімізує тимчасові і фінансові ресурси на розробку і маркетинг. Це шлях до залучення додаткового трафіку, підвищення репутації, а також проявлення лояльності до клієнтів з різних регіонів і людям, принципових у питаннях мови. Крім того, геотаргетинг полегшує розуміння і зближує онлайн-магазин з покупцем. Існує безліч варіантів, як реалізувати локалізацію, але можна виділити кілька основних, які в більшості випадків є оптимальними [16].

Одним із завдань локалізації застосунку є перенесення локалізованого вмісту за допомогою коду, який спрощує заміну контенту для різних мов і регіональних параметрів.

**1.2 Приклади виконання завдання**

**Використання IStringLocalizer.**

**IStringLocalizer** та **IStringLocalizer<T>** створювалися для підвищення продуктивності під час розроблення локалізованих застосунків. **IStringLocalizer** використовує класи **ResourceManager** і **ResourceReader** для надання ресурсів, пов'язаних із певною мовою та регіональними параметрами, під час виконання. Цей інтерфейс має індексатор та інтерфейс **IEnumerable** для повернення локалізованих рядків. **IStringLocalizer** не вимагає зберігати рядки мовою за замовчуванням у файлі ресурсів. Можна розробляти застосунок, призначений для локалізації, не створюючи файли ресурсів на ранніх етапах розробки.

У наступному прикладі коду показано, як упакувати рядок **About Title** для локалізації.

*using Microsoft.AspNetCore.Mvc;*

*using Microsoft.Extensions.Localization;*

*namespace Localization.Controllers;*

*[Route("api/[controller]")]*

*public class AboutController : Controller*

*{*

*private readonly IStringLocalizer<AboutController> \_localizer;*

*public AboutController(IStringLocalizer<AboutController> localizer)*

*{*

*\_localizer = localizer;*

*}*

*[HttpGet]*

*public string Get()*

*{*

*return \_localizer["About Title"];*

*}*

*}*

У попередньому коді реалізація **IStringLocalizer<T>** отримана в результаті впровадження залежностей. Якщо локалізоване значення для рядка **"About Title"** не знайдено, повертається ключ індексатора, тобто рядок **"About Title"**.

Ви можете залишити літеральні рядки мовою за замовчуванням і укласти їх у засіб локалізації, щоб зосередитися на розробці програми. Ви розробляєте застосунок за допомогою мови за замовчуванням і готуєте його до кроку локалізації без першого створення файлу ресурсів за замовчуванням.

Ви також можете вибрати традиційний підхід і надати ключ для вилучення рядка мовою за замовчуванням. Для багатьох розробників новий робочий процес не має файлу **RESX** за замовчуванням і просто упаковує рядкові літерали, що може знизити витрати на локалізацію програми. Інші розробники віддають перевагу традиційному робочому потоку, оскільки його можна спростити для роботи з довгими строковими літералами та спростити оновлення локалізованих рядків.

**Використання IHtmlLocalizer.**

Використовуйте реалізацію **IHtmlLocalizer<TResource>** для ресурсів, що містять код **HTML**. **IHtmlLocalizer** **Html**-кодує аргументи, відформатовані в рядку ресурсу, але не кодує рядок ресурсу у форматі **HTML**. У наступному виділеному коді тільки значення name параметра закодовано у форматі **HTML**.

*using System;*

*using Microsoft.AspNetCore.Http;*

*using Microsoft.AspNetCore.Localization;*

*using Microsoft.AspNetCore.Mvc;*

*using Microsoft.AspNetCore.Mvc.Localization;*

*namespace Localization.Controllers;*

*public class SongController : Controller*

*{*

*private readonly IHtmlLocalizer<SongController> \_localizer;*

*public SongController(IHtmlLocalizer<SongController> localizer)*

*{*

*\_localizer = localizer;*

*}*

*public IActionResult Hello(string name)*

*{*

*ViewData["Message"] = \_localizer["<b>Hello</b><i> {0}</i>", name];*

*return View();*

*}*

**IStringLocalizerFactory**

На найнижчому рівні **IStringLocalizerFactory** можна отримати з впровадження залежностей:

public class TestController : Controller

*{*

*private readonly IStringLocalizer \_localizer;*

*private readonly IStringLocalizer \_localizer2;*

*public TestController(IStringLocalizerFactory factory)*

*{*

*var type = typeof(SharedResource);*

*var assemblyName = new AssemblyName(type.GetTypeInfo().Assembly.FullName);*

*\_localizer = factory.Create(type);*

*\_localizer2 = factory.Create("SharedResource", assemblyName.Name);*

*}*

*public IActionResult About()*

*{*

*ViewData["Message"] = \_localizer["Your application description page."]*

*+ " loc 2: " + \_localizer2["Your application description page."];*

*return View();*

*}*

Наведений вище код демонструє кожен із двох методів створення фабрики.

***Загальні ресурси***

Ви можете секціонування локалізованих рядків за контролером або областю, або лише за одним контейнером. У прикладі програми для **SharedResource** загальних ресурсів використовується клас маркера. Клас маркера ніколи не викликається:

*// Dummy class to group shared resources*

*namespace Localization;*

*public class SharedResource*

*{*

*}*

В наступному прикладі **InfoControllerSharedResource** використовуються локалізатори:

*public class InfoController : Controller*

*{*

*private readonly IStringLocalizer<InfoController> \_localizer;*

*private readonly IStringLocalizer<SharedResource> \_sharedLocalizer;*

*public InfoController(IStringLocalizer<InfoController> localizer,*

*IStringLocalizer<SharedResource> sharedLocalizer)*

*{*

*\_localizer = localizer;*

*\_sharedLocalizer = sharedLocalizer;*

*}*

*public string TestLoc()*

*{*

*string msg = "Shared resx: " + \_sharedLocalizer["Hello!"] +*

*" Info resx " + \_localizer["Hello!"];*

*return msg;*

*}*

***Локалізація представлення***

Служба **IViewLocalizer** надає локалізовані рядки для подання. Клас **ViewLocalizer** реалізує цей інтерфейс і знаходить розташування ресурсів за шляхом до файлу подання. У наступному прикладі коду демонструється використання реалізації **IViewLocalizer** за замовчуванням:

*@using Microsoft.AspNetCore.Mvc.Localization*

*@inject IViewLocalizer Localizer*

*@{*

*ViewData["Title"] = Localizer["About"];*

*}*

*<h2>@ViewData["Title"].</h2>*

*<h3>@ViewData["Message"]</h3>*

*<p>@Localizer["Use this area to provide additional information."]</p>*

Реалізація **IViewLocalizer** за замовчуванням знаходить файл ресурсів за ім'ям файлу подання. Можливості використовувати глобальний загальний файл ресурсів немає. ViewLocalizer реалізує локалізатор за допомогою *IHtmlLocalizer*, тому **Razor** не кодує локалізований рядок HTML. Рядки ресурсів можна параметризувати, а **IViewLocalizer** HTML-кодує параметри, але не рядок ресурсу. Розглянемо таку розмітку **Razor**:

*@Localizer["<i>Hello</i> <b>{0}!</b>", UserManager.GetUserName(User)]*

*Український файл ресурсов может содержать следующие значения:*

*Ключ Значення*

*<i>Hello</i> <b>{0}!</b> <i>Привіт</i> <b>{0} !</b>*

Перетворене для перегляду представлення міститиме розмітку **HTML** з файлу ресурсів. Як правило, тільки локалізований текст, а не **HTML**.

Щоб використовувати в поданні загальний файл ресурсів, впровадьте **IHtmlLocalizer<T**>:

*@using Microsoft.AspNetCore.Mvc.Localization*

*@using Localization.Services*

*@inject IViewLocalizer Localizer*

*@inject IHtmlLocalizer<SharedResource> SharedLocalizer*

*@{*

*ViewData["Title"] = Localizer["About"];*

*}*

*<h2>@ViewData["Title"].</h2>*

*<h1>@SharedLocalizer["Hello!"]</h1>*

***Локалізація DataAnnotations***

Повідомлення про помилки **DataAnnotations** локалізуються за допомогою **IStringLocalizer<T>.** У разі використання параметра **ResourcesPath = "Resources"** повідомлення про помилки в **RegisterViewModel** можуть зберігатися за одним із таких шляхів:

*Resources/ViewModels.Account.RegisterViewModel.fr.resx*

*Resources/ViewModels/Account/RegisterViewModel.fr.resx*

*using System.ComponentModel.DataAnnotations;*

*namespace Localization.ViewModels.Account;*

*public class RegisterViewModel*

*{*

*[Required(ErrorMessage = "The Email field is required.")]*

*[EmailAddress(ErrorMessage = "The Email field is not a valid email address.")]*

*[Display(Name = "Email")]*

*public string Email { get; set; }*

*[Required(ErrorMessage = "The Password field is required.")]*

*[StringLength(8, ErrorMessage = "The {0} must be at least {2} characters long.",*

*MinimumLength = 6)]*

*[DataType(DataType.Password)]*

*[Display(Name = "Password")]*

*public string Password { get; set; }*

*[DataType(DataType.Password)]*

*[Display(Name = "Confirm password")]*

*[Compare("Password", ErrorMessage =*

*"The password and confirmation password do not match.")]*

*public string ConfirmPassword { get; set; }*

*}*

Локалізовані атрибути, не пов'язані з перевіркою. Використання одного рядка ресурсу для кількох класів

У наступному коді показано, як можна використовувати один рядок ресурсу для атрибутів перевірки з декількома класами:

*services.AddMvc()*

*.AddDataAnnotationsLocalization(options => {*

*options.DataAnnotationLocalizerProvider = (type, factory) =>*

*factory.Create(typeof(SharedResource));*

*});*

У попередньому коді клас, який відповідає RESX-файлу, **SharedResource**, в якому зберігаються повідомлення перевірки. За допомогою цього підходу **DataAnnotations** використовує **SharedResource** лише ресурс, а не ресурс для кожного класу.

***Налаштування служб локалізації***

Служби локалізації налаштовуються в Program.cs:

*builder.Services.AddLocalization(options => options.ResourcesPath = "Resources");*

*builder.Services.AddMvc()*

*.AddViewLocalization(LanguageViewLocationExpanderFormat.Suffix)*

*.AddDataAnnotationsLocalization();*

**AddLocalization** додає служби локалізації в контейнер служб, включно з реалізаціями та **IStringLocalizer<T>IStringLocalizerFactory**. Наведений вище код також задає шлях до ресурсів "Ресурси".

Метод **AddViewLocalization** додає підтримку файлів локалізованих подань. У цьому прикладі локалізація подання заснована на суфіксі файлу подання. Наприклад, **fr** в **Index.fr.cshtml** файлі.

Метод **AddDataAnnotationsLocalization** додає підтримку локалізованих повідомлень перевірки DataAnnotations за допомогою абстракцій **IStringLocalizer**.

Примітка. Якщо буде неможливо вводити десяткові коми в полях для десяткових чисел, то для того аби забезпечити підтримку перевірки **jQuery** для інших мов, крім англійської, потрібно використовувати замість десяткової крапки кому (","), а для відображення даних у форматах для інших мов, окрім англійської, необхідно виконати дії, необхідні для глобалізації розроблюваної програми.

**1.3 Завдання до лабораторної роботи**

Відповідно до тематики курсового проекту здійснити локалізацію розроблюваного проєкту. Передбачити 2 мови.

**1.4 Контрольні запитання**

1. Що таке локалізаці?
2. Як здійснити локалізацію DataAnnotations?
3. Яким чином здійснити локалізацію представлення?
4. Використання IStringLocalizer.
5. Використання IHtmlLocalizer.

**Literature: [1, 5, 6, 14, 15, 16]**

**Список використаних джерел**

1. Модульне середовище для навчання. Доступ до ресурсу: https://msn.khmnu.edu.ua/course/view.php?id=8661.
2. Електронна бібліотека ХНУ. Доступ до ресурсу: <http://lib.khnu.km.ua/asp/php_f/p1age_lib.php>.
3. Репозитарій ХНУ. URL: http://elar.khnu.km.ua/jspui/?locale=uk.
4. Вебтехнології : методичні рекомендації до виконання курсового проєкту здобувачами першого (бакалаврського) рівня вищої освіти спеціальності 121 «Інженерія програмного забезпечення» / О. М. Яшина, В. В. Мартинюк. Хмельницький : ХНУ, 2024. 56 с. (англ., укр.).
5. What is front-end development. URL: <https://www.freecodecamp.org/news/front-end-developer-whatis-front-enddevelopment-explained-in-plain-english/> (дата звернення 01.01.2024).
6. Front-end development technologies concepts. URL: https://www.altexsoft.com/blog/front-enddevelopment-technologies-concepts/. 8. Beginners guide for back-end development. URL: <https://www.upwork.com/resources/beginners-guideback-end-development>. (дата звернення 01.01.2024).
7. Back-end architecture. URL: <https://www.codecademy.com/article/backend-architecture> (дата звернення 01.01.2024).
8. Development tools for web developers. URL: <https://www.geeksforgeeks.org/12-backenddevelopment-tools-for-webdevelopers/>
9. Який back-end потрібний вашому вебсайту? URL: <https://goldwebsolutions.com/uk/blog/yakijbackend-potribnij-vashomu-vebsajtu/> (дата звернення 01.01.2024).
10. Олівер Дік. Популярні Web-браузери. Енциклопедія користувача. – К.:DiaSoft, 2018. – 280 с.
11. Боуен Річ. Apache. Настільна книга адміністратора. – К.: DiaSoft, 2019. - 350 с.
12. Матвієнко, О.В. Internet-технології: проектування Webсторінки: Навчальний посібник для студентів вузів / О. В Матвієнко, І. Л Бородкіна. - 2-е вид., перероб. и доп. . - К.: ЦНЛ, 2019. - 154 c.
13. Куленко М.Я. Основи графічного дизайну : підручник для студентів вищих навч. закладів / Михайло Куленко; МОНУ; Київський нац. ун-т будівництва і архітектури. – 2-ге вид., виправл. та доп. – Київ : Кондор, 2020. – 492с.
14. Node.js vs .NET: What to Choose in 2023. URL: <https://www.techmagic.co/blog/node-js-vs-net-what-to-choose/>.
15. Документація ASP.NET Core. URL: <https://learn.microsoft.com/aspnet/core/security/authorization/roles?view=aspnetcore-9.0> (дата звернення 01.01.2024).
16. Мовні версії сайту: особливості локалізації. URL: <https://www.promodo.ua/blog/movni-versiyi-saytu-osoblivosti-lokalizaciyi> (дата звернення 01.02.2024).

Додаток А

Зразок титульної сторінки звіту з лабораторної роботи

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет інформаційних технологій

Кафедра інженерії програмного забезпечення

Лабораторна робота №

з дисципліни «Вебтехнології»

на тему: «Локалізація та мультимовність»

**Виконав:**

студент 3 курсу, групи ІПЗ-21-1 \_\_\_\_\_\_\_\_\_\_ Артем ГАЙДУЧЕНКО

(підпис) (ім’я, прізвище)

**Перевірив:** \_\_\_\_\_\_\_\_\_\_\_ Оксана ЯШИНА

(підпис) (ім’я, прізвище)

Хмельницький 2023