Створення веб-додатків у .NET за допомогою Blazor

Ahmad Mozaffar Mastering Blazor WebAssembly Copyright © 2023 Packt Publishing

Повний стек веб-розробок на C#

Створення сучасних веб-сайтів односторінкових програм сьогодні зазвичай означає написання JavaScript на клієнті та C# на сервері, коли ви використовуєте стек розробки Microsoft. Але з Blazor ви можете створювати все, використовуючи C#, і повторно використовувати знання та досвід, отримані з .NET. Перенесення існуючих додатків C#, таких як WinForms, до Інтернету не передбачає перекладу частини вашої логіки на JavaScript; ви можете повторно використати більшу частину цього коду, що призведе до зменшення тестування та помилок.

Blazor WebAssembly — це фреймворк, який дає змогу створювати односторінкові веб-додатки, які використовують C# на клієнті замість JavaScript. Він побудований на популярній і надійній структурі ASP.NET. Blazor WebAssembly не покладається на плагіни чи надбудови для використання C# у браузері. Потрібно лише, щоб браузер підтримував WebAssembly, що є у всіх сучасних браузерах.

Знайомство з Blazor WebAssembly

Blazor WebAssembly — це нова платформа односторінкових програм (SPA) Microsoft для створення інтерактивних веб-програм на .NET Framework. Оскільки Blazor WebAssembly створено на основі .NET Framework, він дозволяє запускати код C# як на клієнті, так і на сервері. Таким чином, замість того, щоб бути змушеними писати JavaScript на клієнті, тепер ми можемо використовувати C# всюди.

Перша браузерна війна

Швидко випускалися нові версії, і браузери почали додавати нові функції, такі як елементи <blink> і <marquee>. Це був початок першої війни браузерів, яка завдавала людям (особливо дизайнерам) головного болю, оскільки деякі розробники створювали сторінки з миготливими елементами керування 😊. Але розробникам також боліли голови через несумісність між

браузерами. Перша війна браузерів стосувалася того, щоб мати більше можливостей HTML, ніж конкуренти.

Але все це вже позаду з появою HTML5 і сучасних браузерів, таких як Google Chrome, Microsoft Edge, Firefox, Safari та Opera. HTML5 не лише визначає набір стандартних елементів HTML, але й визначає, як вони мають відображатися, що значно полегшує створення веб-сайту, який виглядає однаково в усіх сучасних браузерах. Потім, у 1995 році, Brendan Eich написав невелику мову програмування, відому як ECMAScript (спочатку вона називалася LiveScript). Її швидко охрестили JavaScript, оскільки її синтаксис був дуже схожий на Java.

Створення веб-сайту Ajax було серйозною справою, яку могли собі дозволити лише такі великі компанії, як Microsoft і Google. Незабаром це змінилося з появою таких бібліотек JavaScript, як jQuery та knockout.js (knockout також написав Стів Сандерсон, автор Blazor!). Сьогодні ми створюємо багатофункціональні веб-програми за допомогою Angular, React і Vue.js. Усі вони використовують JavaScript або мови вищого рівня, такі як TypeScript, які транспільуються в JavaScript.

Транспіляція візьме одну мову та перетворить її на іншу мову. Це дуже популярно у TypeScript, який дає вам сучасну високорівневу мову. Вам потрібен JavaScript, щоб запустити його в браузері, тому TypeScript буде «транспільовано» в JavaScript.

Друга браузерна війна

Це повертає нас до JavaScript і другої війни браузерів. Продуктивність JavaScript має першорядне значення в сучасних браузерах. Chrome, Edge, Firefox, Safari та Opera змагаються між собою, намагаючись переконати користувачів, що їхній веб-переглядач найшвидший, за допомогою крутих назв для свого механізму JavaScript, як-от V8 і Chakra. Ці механізми використовують новітні прийоми оптимізації, такі як компіляція Just-In-Time (JIT), де JavaScript перетворюється на рідний код, як показано на Рис. 1.

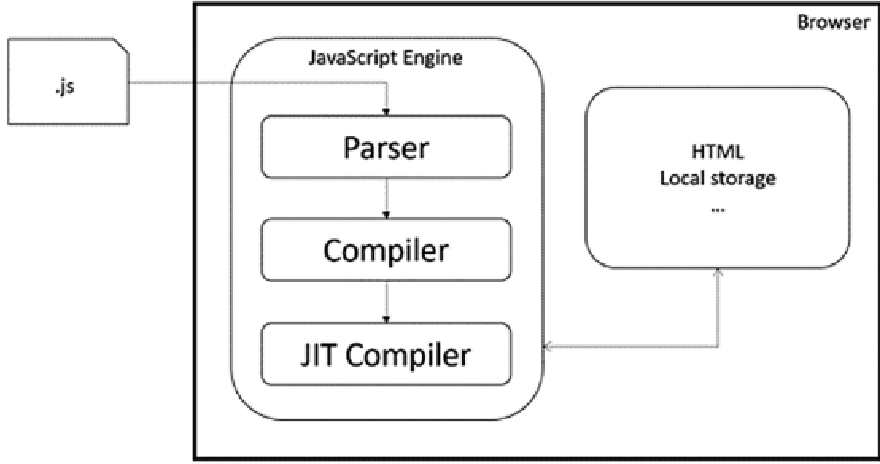


Рис. 1 – Процес виконання JavaScript

Цей процес потребує багато зусиль, оскільки JavaScript потрібно завантажити у браузер, де він аналізується, потім компілюється в байт-код, а потім «точно вчасно» перетворюється на рідний код. Отже, як ми можемо зробити цей процес ще швидшим?

Друга війна браузерів пов’язана з продуктивністю JavaScript.

Представляємо WebAssembly

WebAssembly дозволяє вам перенести аналіз і компіляцію на сервер, перш ніж ваші користувачі навіть відкриють свій браузер. За допомогою WebAssembly ви компілюєте свій код у форматі під назвою WASM (абревіатура від WebASseMbly), який завантажується браузером, де він компілюється за допомогою Just-In-Time у рідний код, як показано на Рис. 2.

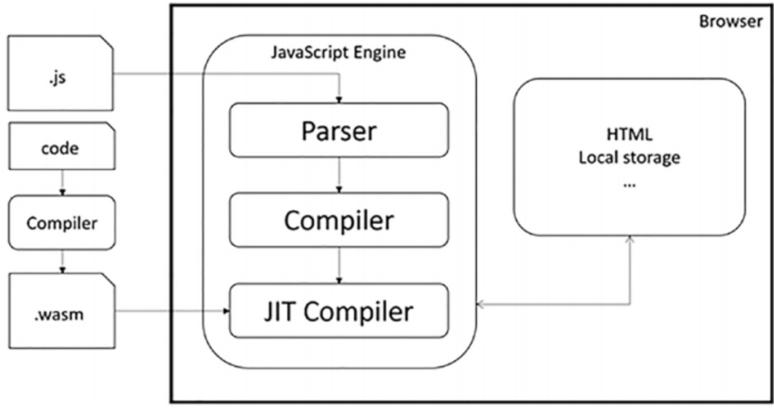


Рис. 2 – Процес виконання WebAssembly

Відкрийте браузер і відкрийте https://earth.google.com . Це має привести вас до програми Google Earth, написаної на WebAssembly, як показано на Рис.3. Трохи пограйте з цим, і ви побачите, що ця програма має чудову продуктивність, але початкове завантаження займає досить багато часу, оскільки потрібно завантажити весь код програми WASM.

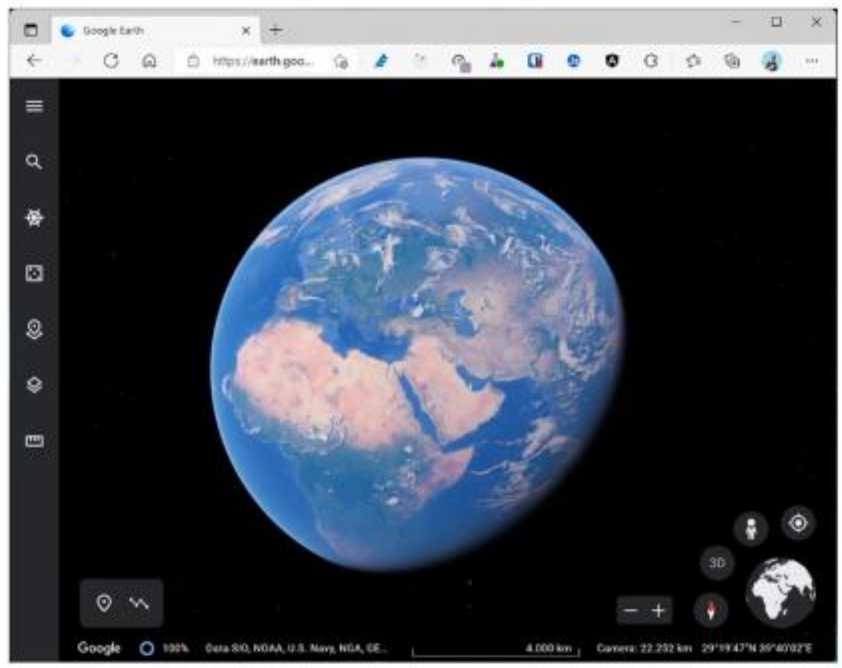


Рис. 3 – Google Earth у WebAssembly

Що таке WebAssembly? З офіційного сайту webassembly.org: WebAssembly (скорочено Wasm) — це двійковий формат інструкцій для стекової віртуальної машини. Wasm розроблений як переносна ціль (portable target) для компіляції мов високого рівня, таких як C/C++/Rust, що дозволяє розгортати клієнтські та серверні програми в Інтернеті.

Які браузери підтримують WebAssembly?

WebAssembly підтримується всіма основними браузерами: Chrome, Edge, Safari, Opera та Firefox, включаючи їх мобільні версії. Ви можете перевірити підтримку самостійно, відвідавши https://caniuse.com/?search=WASM .

Розуміння анатомії проекту Blazor WebAssembly

Blazor став трендом у світі розробки програмного забезпечення завдяки своїй простоті, продуктивності та широкому спектру застосування в розробці веб-, мобільних і настільних додатків через .NET Multiplatform App UI (MAUI) Таким чином, розробникам корисно дізнатися про цю структуру та опанувати її.

Технічні вимоги

Щоб слідувати за нами, вам потрібно знати основні поняття мови програмування C# і основи веб-розробки.

Для процесу розробки знадобиться Visual Studio 2022 Community, яку можна безкоштовно завантажити для пристроїв Windows за адресою https://visualstudio.microsoft.com/vs/community , а для пристроїв macOS – за посиланням https://visualstudio.microsoft.com/vs /mac/ .

Крім того, ви можете використовувати Visual Studio Code для будь-якої ОС (Windows, macOS або Linux), завантаживши його з https://code.visualstudio.com/download .

Якщо ви не збираєтеся встановлювати Visual Studio 2022, переконайтеся, що пакет SDK .NET 7.0 доступний на вашому комп'ютері. Цей SDK зазвичай завантажується разом із Visual Studio 2022. Якщо у вас немає SDK, ви можете встановити його з https://dotnet.microsoft.com/en-us/download.

Створення вашого першого проекту Blazor WebAssembly

Давайте почнемо зі створення нашого проекту Blazor WebAssembly за допомогою .NET CLI, а потім зробимо те саме за допомогою Visual Studio 2022 і будемо будувати на основі цього для решти книги.

Використання .NET CLI

Щоб створити проект Blazor WebAssembly за допомогою .NET CLI, виконайте такі дії:

1. Відкрийте командний рядок у Windows або термінал у macOS і виконайте таку команду:

```
dotnet new blazorwasm -n BookStore
```

Попередня команда створить новий проект Blazor WebAssembly у папці під назвою BooksStore у каталозі, у якому ви виконали цю команду, і проект називатиметься BooksStore.

2. Ви можете створити свою програму за допомогою команди build після переходу до новоствореної папки таким чином:

```
cd BooksStore\
dotnet build
```

3. Далі ви можете запустити проект:

```
dotnet run
```

Під час запуску програми ви побачите дві URL-адреси в CLI: одну для https-посилання, а іншу для http-посилання. Перейдіть за посиланням https у вашому браузері, і ви зможете побачити остаточний результат програми Blazor WebAssembly за замовчуванням, яка була доступна з коробки .NET 7.0:

```
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: https://localhost:7166
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://localhost:5166
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: C:\AK\Mastering Blazor WebAssembly\Source Code\Mastering-Blazor-WebAssem
bly\Chapter_01\BooksStore\BooksStore
```

Використання Visual Studio 2022

Тепер давайте швидко розглянемо, як створити той самий проект Blazor WebAssembly за допомогою Visual Studio 2022:

1. Відкрийте Visual Studio 2022 і на початковій сторінці натисніть «Create a new project». Потім знайдіть шаблон Blazor WebAssembly.

2. Виберіть додаток Blazor WebAssembly:

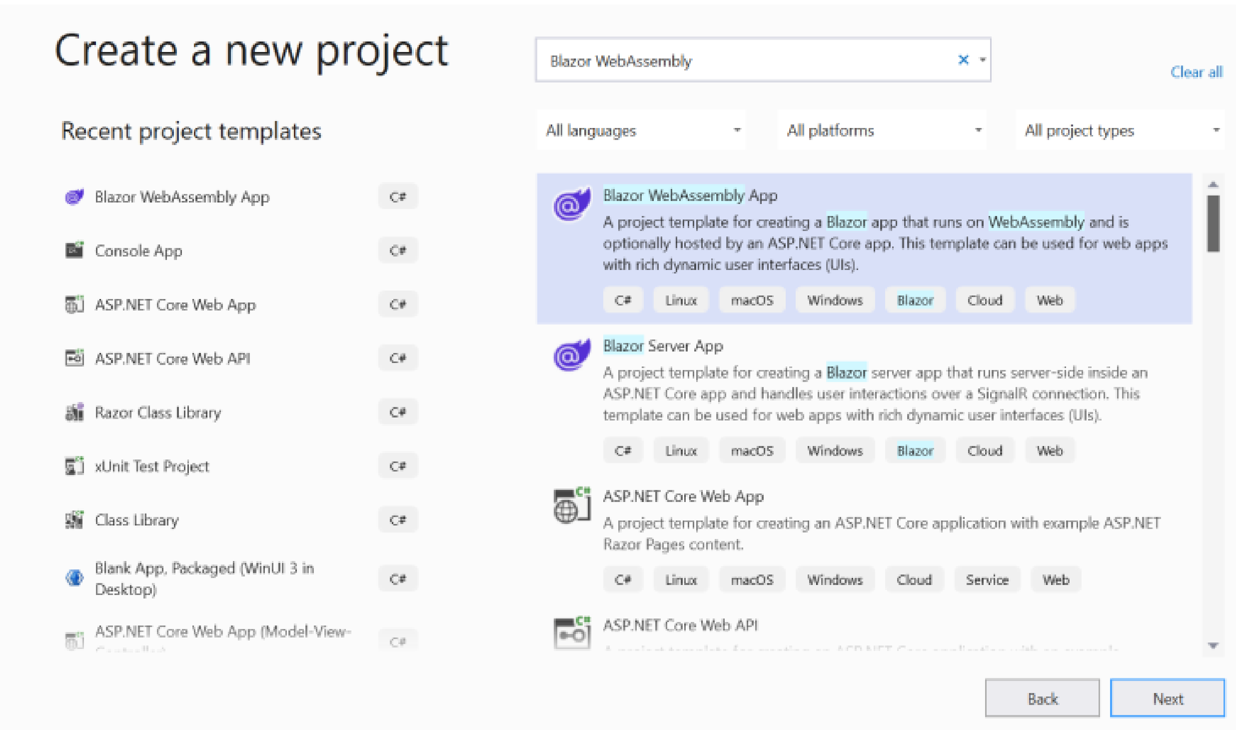


Рисунок 1.2 – Вибір шаблону проекту у VS 2022

3. Дайте своєму проекту назву та виберіть каталог, у якому ви хочете зберегти його на своїй машині:

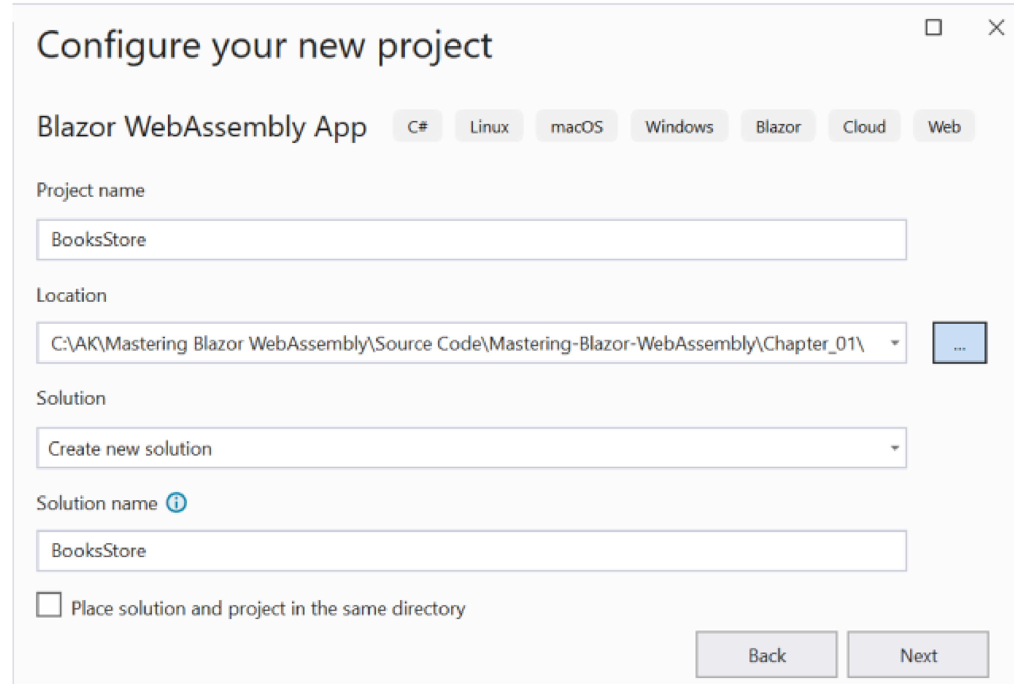


Рисунок 1.3 – Додавання назви та розташування проекту

4. Збережіть початкову конфігурацію, запропоновану Visual Studio. Натисніть Створити:

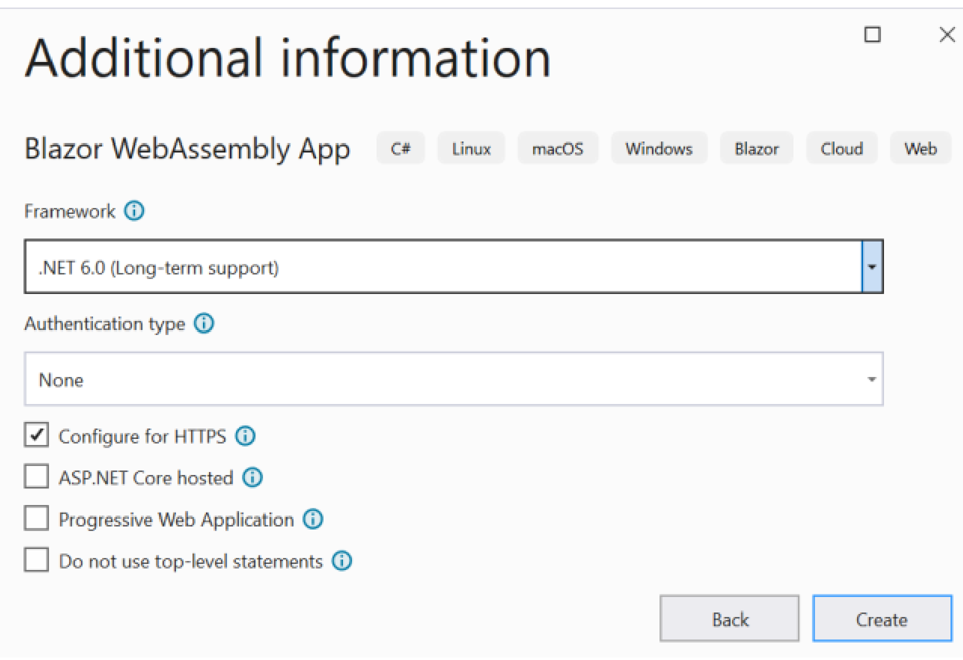


Рисунок 1.4 – Конфігурація проекту за замовчуванням у Visual Studio 2022

Проект буде створено, і ви побачите такі файли та папки на панелі VS Solution Explorer:

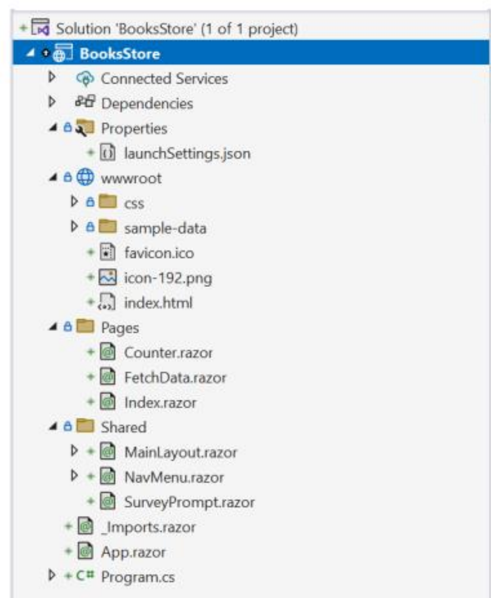


Рисунок 1.5 – Файли та папки проекту на панелі Solution Explorer

Натискання кнопки «Пуск» (F5) у вікні VS 2022 запустить проект і автоматично відкриє браузер, і ви побачите програму Blazor за умовчанням.

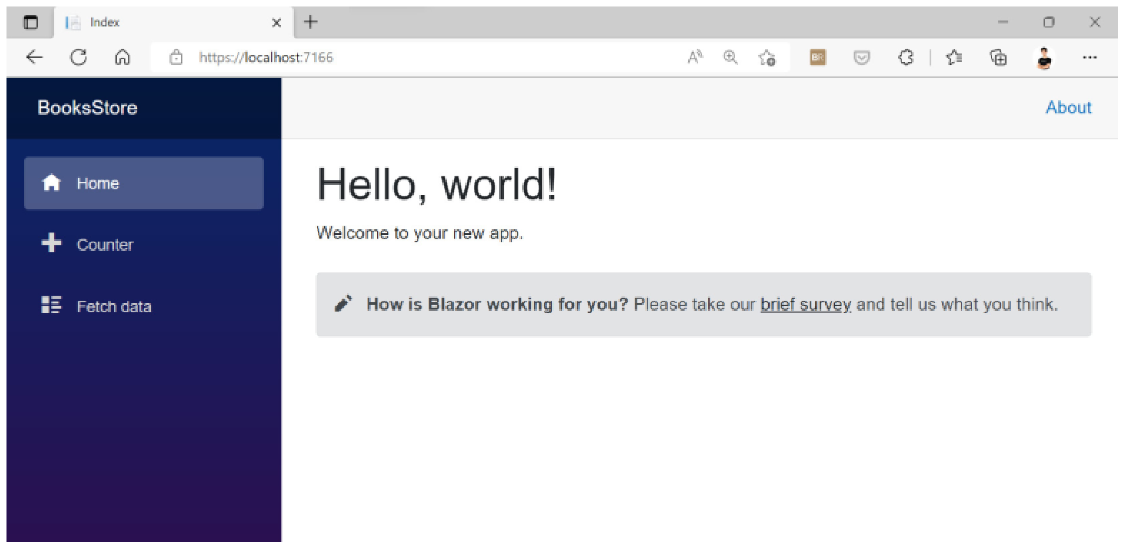


Рисунок 1.6 – За замовчуванням запущений проект Blazor WebAssembly

Щиро вітаю! Ви бачили знаменитий фіолетовий інтерфейс; тепер ми готові рухатися вперед і почати відкривати призначення кожного з цих файлів і папок.

Знайомство зі структурою проекту

Тепер у нас є робочий простір, який ми можемо використати, щоб розпочати створення нашого додатка, але перш ніж заглибитися глибше, давайте подивимось на структуру проекту, включаючи файли та папки, які ми бачимо. Розуміння того, що і чому для кожного файлу в Solution Explorer, є ключовим для завершення проекту, який ми будемо будувати.

Починаємо досліджувати!

Коренева папка wwwroot

Папка wwwroot вважається кореневою папкою вашого проекту. Корінь тут не означає корінь у вашому файловому провіднику, який містить файли Razor та інші папки. wwwroot — це місце, де знаходяться файли вашої програми, включаючи глобальні стилі CSS, основний HTML-файл проекту, index.html, JavaScript і ресурси (зображення, значки, статичні документи тощо). Тут ви також розміщуєте файли JSON, які представляють конфігурацію

вашої програми (ви прочитаєте більше про конфігурацію в наступному розділі цієї глави «Dependency injection in Blazor WebAssembly»).

За замовчуванням папка wwwroot містить наступне:

• Папка css: ця папка містить файли CSS Bootstrap і стандартний шрифт. Папка також містить файл app.css, який містить глобальні стилі програми.

• Папка зразкових даних: ця папка містить файл JSON із зразковими даними, що стосуються прогнозу погоди. Це використовується як джерело даних для сторінки Fetch data, про яку буде згадано незабаром.

• favicon.ico: це піктограма програми за замовчуванням, яку ви бачите у верхньому лівому куті вкладки браузера вашої програми, у списку улюблених сайтів у браузері та в деяких інших місцях.

• Icon-192.png: це файл PNG із значком Blazor, на який посилаються компоненти.

• index.html: назва цього файлу має бути вам знайома. У більшості випадків index.html є файлом HTML за замовчуванням, який відкривається під час доступу до статичного веб-сайту. Цей основний файл HTML містить нашу програму. Браузер завантажує цей файл і відкриває його з усіма його посиланнями на CSS і JavaScript, особливо blazor.webassembly.js. Тіло index.html містить, за замовчуванням, div з ідентифікатором програми (це фактично місце, де знаходиться весь інтерфейс користувача, який ви бачите в будь-якій програмі Blazor), а також інший div з ідентифікатором blazorerror-ui, який є простим дизайном div, який відображається, коли виникає необроблена виняткова ситуація (про це ви дізнаєтеся більше в Розділі 10, Обробка помилок у Blazor WebAssembly).

Розробляючи ваші програми, ви збираєтеся вносити зміни до цього файлу здебільшого, щоб додати посилання на стилі або деякі файли JavaScript далі в цій книзі.

Нижче наведено код для елементів тіла за замовчуванням (app div і blazor-error-ui div) у файлі index.html:

```
<html lang="en">
….
<body>
    <div id="app">Loading...</div>
```

```
    <div id="blazor-error-ui">
        An unhandled error has occurred.
        <a href="" class="reload">Reload</a>
        <a class="dismiss">✕ </a>
    </div>
    <script
        src="_framework/blazor.webassembly.js"></script>
</body>
</html>
```

Примітка

Термін односторінкова програма (SPA) здається трохи дивним, коли ви відкриваєте програму та починаєте переходити з однієї сторінки на іншу. Однак технічно SPA складаються з однієї HTML-сторінки, і Blazor або будь-який інший фреймворк JavaScript (ReactJS, Angular тощо) динамічно оновлює вміст цієї сторінки. Як ви бачили, у папці wwwroot у всій нашій програмі є лише одна HTML-сторінка, яка називається index.html.

Папка Pages

Pages — це папка за замовчуванням, куди Microsoft розміщує свої сторінки за замовчуванням, які постачаються разом із шаблоном. Ви можете використовувати Pages як контейнер для компонентів сторінки програми. Однак розміщувати компоненти в цій папці не обов’язково, її можна навіть видалити. Pages постачається з трьома зразками сторінок, які охоплюють різні концепції, такі як виклик спільного компонента, оновлення DOM за допомогою кнопки та отримання даних із джерела даних:

• Counter.razor: це компонент Razor, до якого ви можете отримати доступ через route / counter. Це також приклад виклику методу C# з кнопки HTML і прив’язки вмісту елемента HTML <p> до змінної C# для відображення результату.

• FetchData.razor: Ви можете отримати доступ до цього компонента, перейшовши до /fetch-data. Ця сторінка містить таблицю HTML, яка показує дані прогнозу погоди та демонструє, як отримати дані за допомогою HttpClient. Якщо ввести об’єкт і зробити HTTP-запит GET, щоб отримати дані JSON із файлу weather.json у папці wwwroot, дані відображаються в компоненті.

• Index.razor: це стандартний компонент, який ви побачите під час запуску проекту. Він показує вам, як викликати інші спільні компоненти, такі як SurveryPrompt, і передати йому параметр Title.

Shared папка

Як видно з назви, ця папка містить деякі спільні компоненти програми. Ви можете змінити цю структуру, але ми будемо дотримуватися такої самої структури в решті цієї книги та використовуватимемо цю папку для розміщення компонентів, які будуть спільними для всіх сторінок (макет програми, навігаційне меню, компоненти автентифікації тощо).). За замовчуванням шаблон Blazor містить три спільні компоненти:

• MainLayout.razor: містить типовий макет проекту та є особливим типом компонента. Він визначає, як виглядає програма, і розміщує компоненти вмісту (сторінки) у потрібному місці. За замовчуванням цей компонент посилається на компонент NavMenu та містить HTML-тег <main>, який відображає властивість @Body. Ця властивість містить вміст сторінок на основі навігаційного маршруту (більше про компоненти макета буде розглянуто в Розділі 3, Розробка розширених компонентів у Blazor).

• NavMenu.razor: цей компонент містить пункти меню програми (набір URL-адрес для переходу між сторінками, згаданими в папці Pages).

• SuveryPrompt.razor: це простий компонент, який демонструє, як створити багаторазовий компонент, який приймає параметри. Індексна сторінка посилається на цей компонент і передає значення його параметру Title.

Файл _Imports.razor

У класах C# вам завжди потрібно використовувати класи, що існують у різних просторах імен. Отже, по суті, ми посилаємося на простір імен, оголошуючи оператор using у верхній частині файлу C#, щоб ми могли посилатися на необхідні класи та вкладені простори імен. Для цього існує файл _Imports.razor, і ви можете використовувати його для посилань на загальні простори імен у більшості ваших компонентів. Отже, немає необхідності додавати оператор using у верхній частині кожного файлу Razor для доступу до необхідного коду. За замовчуванням компонент посилається на деякі

загальні простори імен .NET, які ви використовуватимете у своїх компонентах, наприклад System.Net.Http і простір імен вашої збірки та спільної папки.

Нижче наведено код для файлу _imports.razor зі спільними посиланнями на простори імен:

```
@using System.Net.Http
…
@using Microsoft.JSInterop
@using BooksStore
@using BooksStore.Shared
```

Файл App.razor

Компонент програми є батьківським і головним компонентом програми Blazor. По суті, він визначає інфраструктуру, необхідну для функціонування вашої програми, наприклад компонент маршрутизатора, який відповідає URL-адресі, введеній у браузері, і відображає відповідний компонент у вашій програмі. Компонент програми також визначає компонент макета за замовчуванням (MainLayout, згаданий у спільній папці). Більше про компоненти макета буде розглянуто в частині 2, главі 3, Розробка розширених компонентів у Blazor. Крім того, компонент App містить розділ NotFound, який можна використовувати для відтворення певного вмісту, якщо запитану адресу не знайдено.

Файл Program.cs

У будь-якій програмі C# є точка входу, і Program.cs представляє точку входу вашої програми Blazor. Ця точка входу налаштовує хост Blazor WebAssembly і зіставляє компонент програми з відповідним div у файлі index.html (за замовчуванням div з ідентифікатором програми). Ми також використовуємо файл для реєстрації в контейнері DI будь-якої служби, яку нам потрібно використовувати протягом усього терміну служби програми.

Тепер ви зможете легко орієнтуватися в рішенні, зрозумівши, що робить кожен файл і чому він там; Наступним кроком у цій главі є розгляд концепції DI.

Впровадження залежностей (Dependency injection) у Blazor WebAssembly

Сучасна розробка програмного забезпечення пов’язана з масштабуванням, розділенням і тестуванням, тому, коли ви пишете код, ви повинні бути впевнені в його поведінці та надійності. Щоб досягти цієї мети, вам потрібно пам’ятати про принципи SOLID протягом усього шляху розробки. Принцип, представлений літерою D, є інверсією залежностей. В основному це стосується приховування реалізації ваших служб за інтерфейсами. Наприклад, для кожного класу обслуговування, який у вас є, ви можете створити інтерфейс, який містить методи цього класу, і змусити клас реалізовувати інтерфейс. Потім під час використання цієї служби ви посилаєтеся на інтерфейс замість безпосереднього класу. Ця стратегія відокремлює компоненти або фрагменти коду, які використовують службу, від її реалізації залежно від абстрактного рівня служби, яким є інтерфейс.

Все це дивовижно, але не вистачає однієї частини: як ініціалізувати об’єкт служби, який використовуватиметься в кількох місцях. Ось де DI вступає в гру. Це просто техніка, яка дозволяє вам реєструвати ваші служби, які використовуються в інших службах або компонентах, у централізованому контейнері, і цей контейнер відповідатиме за обслуговування цих об’єктів компонентам, яким вони потрібні.

Як працює ін’єкція залежностей

На наступній діаграмі показано потік класу ConsoleLoggingService:

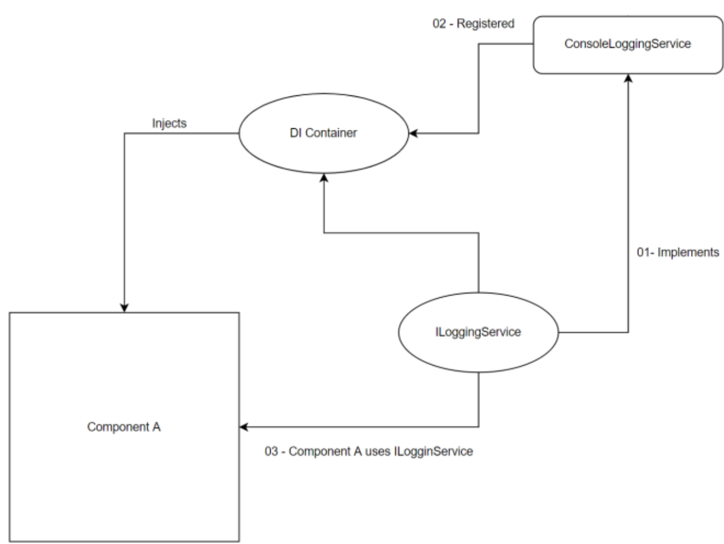


Рисунок 1.7 – Потік контейнера ін’єкції залежностей і пов’язаних служб

На попередньому малюнку клас ConsoleLoggingService реалізує інтерфейс ILoggingService. Потім у контейнері DI ми реєструємо екземпляр ILoggingService з новим об'єктом його реалізації: ConsoleLoggingService. Щоразу, коли компоненту A потрібна логіка журналювання, він використовує ILoggingService (рівень абстракції ConsoleLoggingService).

Замість того, щоб самостійно ініціалізувати новий екземпляр об'єкта, контейнер DI обслуговує ILoggingService для компонента A, що додає багато переваг, оскільки зберігає код розділеним. Крім того, логіку реалізації для повної служби можна змінити в будь-який час, не торкаючись компонента A, який залежить від базового інтерфейсу цієї служби. Наприклад, якщо ми хочемо зареєструвати дані на сервері, а не на консолі, щоб почати використовувати нову службу, ми можемо просто написати нову реалізацію ILoggingService і зареєструвати цей екземпляр у контейнері DI, не змінюючи нічого в компоненті A чи будь-якому іншому коді клієнта.

Використання ін'єкції залежностей у Blazor WebAssembly

Blazor WebAssembly поставляється з контейнером DI з коробки. Ви можете почати з реєстрації своїх служб у файлі Program.cs, а потім додати ці служби до клієнтських компонентів або служб.

Давайте запровадимо або напишемо нашу першу службу, ILoggingService, як згадувалося в попередньому розділі, з методом Log і створимо реалізацію для реєстрації повідомлень у вікні консолі браузера. Потім ми введемо цю службу в компонент FetchData і зареєструємо кількість погодних даних, отриманих компонентом FetchData. Щоб запровадити послугу, пройдіть наступні кроки:

1. У Solution Explorer клацніть правою кнопкою миші на проекті та натисніть **Add | New Folder**. Назвіть папку Services.

2. Клацніть правою кнопкою миші на новоствореній папці та виберіть **Add | New Item**.

3. У діалоговому вікні, що з'явиться, виберіть «**Interface**» як тип елемента та дайте йому назву ILoggingService. Потім натисніть **Add**.

4. Інтерфейс створено, і метою є створення методу Log. Як ви знаєте, в інтерфейсах ми визначаємо лише сигнатуру методу (його тип даних, що

повертається, і його параметри), тому ми визначимо цей метод наступним чином:

```
public interface ILoggingService
{
    void Log(string message);
}
```

5. Після створення інтерфейсу ми повинні створити реалізацію, яка реєструє це повідомлення на консолі, тому повторіть крок 2, але замість створення інтерфейсу давайте створимо клас і назвемо його ConsoleLoggingService.

6. Після створення класу давайте реалізуємо інтерфейс ILoggingService і запишемо логіку цього методу наступним чином:

```
public class ConsoleLoggingService : ILoggingService
{
    public void Log(string message)
    {
        Console.WriteLine(message);
    }
}
```

Метод Log викликає метод WriteLine у класі Console, але в Blazor WebAssembly метод Console.WriteLine не працює так само, як в інших програмах .NET, друкуючи рядок у програмі консолі. Він друкує рядок у вікні консолі в інструментах розробника браузера.

Ви можете отримати доступ до інструментів розробника у своєму браузері наступним чином:

Microsoft Edge: F12 у Windows або ⌘ + ⌥ + I для Mac

Інші браузери: Ctrl + Shift + I

Остання річ, щоб підготувати цю службу до впровадження в інші компоненти, це реєстрація її в контейнері DI.

7. Перейдіть до файлу Program.cs і зареєструйте службу методом AddScoped:

```
...
using BooksStore.Services;
var builder = WebAssemblyHostBuilder.CreateDefault(args);
```

```
...
builder.Services.AddScoped<ILoggingService,

ConsoleLoggingService>();
await builder.Build().RunAsync();
```

Наша перша служба зараз готова до використання та впроваджена в інші компоненти. У наведеному нижче прикладі показано, як додати цю службу в компонент FetchData і досягти необхідної цілі щодо реєстрації кількості елементів прогнозу погоди у вікні консолі браузера:

1. Відкрийте файл _Imports.razor і додайте оператор using до простору імен Services:

```
@using BooksStore.Services
```

Ми додаємо посилання на простір імен Services під час імпорту, оскільки ці служби здебільшого використовуватимуться в багатьох компонентах, тому нам не потрібно додавати цей оператор using у кожен компонент.

Відкрийте компонент FetchData у папці Pages і введіть ILoggingService за допомогою директиви @inject Razor у компоненті:

```
@page "/fetchdata"
...
@inject ILoggingService LoggingService
<PageTitle>Weather forecast</PageTitle>
<h1>Weather forecast</h1>
....
```

2. Наша служба готова до використання, і ми можемо викликати функцію Log з екземпляра об’єкта в коді C#, щоб зареєструвати кількість елементів таким чином:

```
....
         </tbody>
   </table>
}
@code {
    private WeatherForecast[]? forecasts;
    protected override async Task OnInitializedAsync()
    {
        forecasts = await
          Http.GetFromJsonAsync<WeatherForecast[]>
           ("sample-data/weather.json");
```

```
        LoggingService.Log($"Number of items retrieved
                              is {forecasts.Count()}");
} …
```

Якщо ви запустите проект і перейдете на сторінку Fetch data після отримання даних, ви зможете відкрити вікно консолі у своєму браузері, і ви побачите повідомлення Кількість отриманих елементів становить 5.

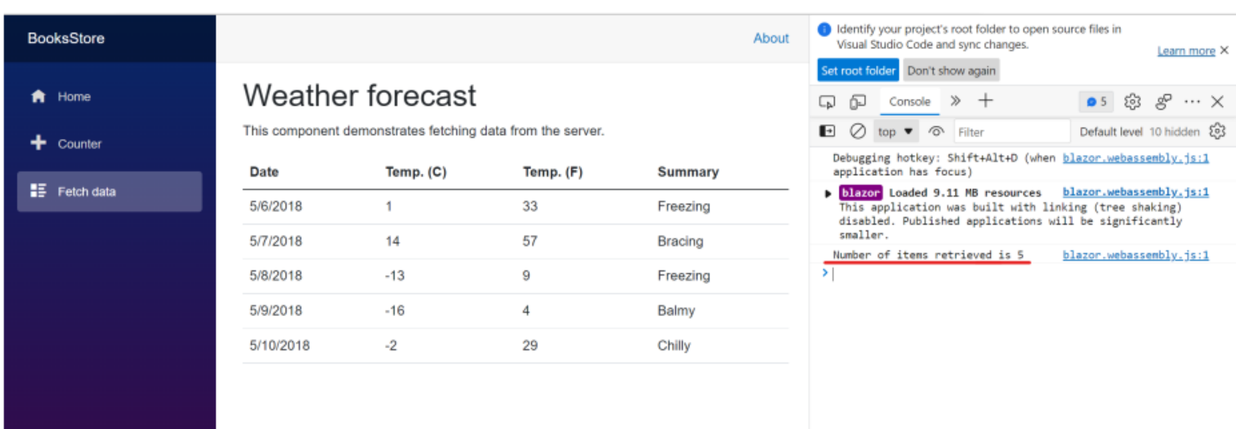


Рисунок 1.9 – Скріншот вікна консолі з надрукованим реченням

Порада

Щоб дізнатися більше про концепцію DI у Blazor, ви можете переглянути таке посилання: https://learn.microsoft.com/en-us/aspnet/core/blazor/fundamentals/dependency-injection?view=aspnetcore-7.0.

Тепер ми реалізували повний цикл, від створення абстрактного рівня сервісу до його реалізації, реєстрації сервісу в контейнері DI і, нарешті, впровадження та використання цього сервісу з окремого компонента.

Дізнавшись про DI та як він працює, ви зможете зрозуміти багато аспектів програми, наприклад написання зручних для обслуговування, відокремлених служб. Крім того, ви зможете використовувати такі вбудовані служби, як HttpClient для викликів API та службу IConfiguration, яку ми збираємося використовувати в наступному розділі для читання та використання конфігурацій програми.

Створення, зберігання та отримання конфігурацій програми

Конфігурації — це набір налаштувань, які ваша програма використовує протягом усього терміну служби. Конфігурації програми допомагають уникнути жорсткого кодування деяких значень у програмі, які можуть час від часу змінюватися. Крім того, процес дуже важливий, коли ви маєте справу з кількома середовищами. Наприклад, ви можете зберегти URL-адресу API, з яким спілкується ваша програма. Якщо на машині для розробників URL-адреса є локальним хостом, у робочому середовищі вона посилається на розміщений в Інтернеті API (більше про середовища буде розглянуто в наступному розділі).

Blazor WebAssembly вже підтримує конфігурацію з файлів JSON, які можна створити в папці wwwroot. Файл має називатися appsettings.json, і щоб мати файл конфігурації для кожного середовища, ви можете вказати далі, наприклад, appsettings.{ENVIRONMENT}.json або appsettings.Development.json.

Важливе зауваження.

Ви можете додати додаткові джерела для конфігурації, але файлів налаштувань програм достатньо, оскільки програма Blazor WebAssembly працює повністю на стороні клієнта. Підключення програми, наприклад, до служби Azure Key Vault означає, що рядок з'єднання зберігатиметься на клієнті та буде під загрозою легкого викриття.

Отже, для загальних конфігурацій appsettings є чудовим вибором. Інші види секретів настійно рекомендується зберігати на стороні сервера, і ми повинні розробити нашу програму таким чином, щоб клієнту не потрібно було їх отримувати або використовувати.

Ми створимо файл конфігурації та збережемо деякі налаштування, наприклад URL-адресу API, який ми будемо використовувати. Потім ми надрукуємо це значення за допомогою компонента Index. Давайте розпочнемо:

1. Клацніть правою кнопкою миші папку wwwroot і додайте новий елемент типу файлу JSON і назвіть його appsettings.json.

2. Додайте властивість під назвою «ApiUrl» і присвойте їй значення «http://localhost:44332/» таким чином:

```
{
    "ApiUrl": "http://localhost:44332"
}
```

Після збереження файлу ви можете отримати доступ до цього значення налаштування безпосередньо за допомогою служби IConfiguration, вставивши його в будь-який компонент або службу та посилаючись на індекс імені вашої властивості.

У наступному прикладі ми введемо службу IConfiguration у компонент Index і надрукуємо значення властивості ApiUrl:

1. Відкрийте файл Index.razor у папці Pages і вставте службу IConfiguration, як показано:

```
@page "/"
@inject IConfiguration Configuration
<PageTitle>Index</PageTitle>
...
```

2. Після введення служби ви готові отримати доступ до необхідного значення за допомогою індексатора екземпляра конфігурації. Ми збираємося додати тег <p> і відобразити таке: URL-адреса API: {API_URL_VALUE_IN_SETTINGS}:

```
...
<h1>Hello, world!</h1>
Welcome to your new app.
<p>Api Url: @Configuration["ApiUrl"]</p>
...
```

Після запуску програми ви повинні побачити такий результат у вікні браузера:

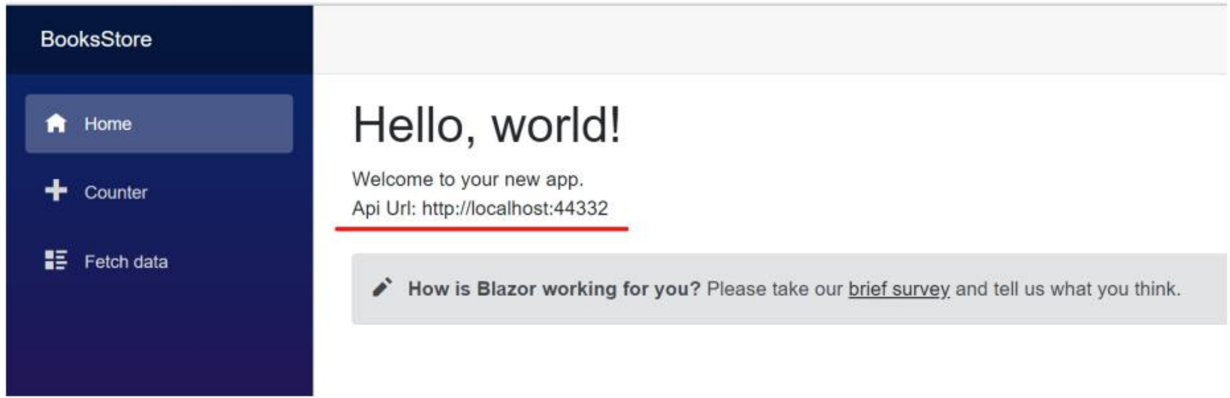


Рисунок 1.10 – Значення ApiUrl конфігурації з appsettings.json

Конфігурації важливі для вас у вашому навчанні, і під час розробки нашої реальної програми ми будемо використовувати їх час від часу. Після наведеного прикладу ви маєте чітко розуміти, як додавати конфігурації та як отримувати їх значення під час виконання програми.

У наступному розділі ми познайомимося з новими середовищами нашої програми та додамо новий файл конфігурації, який використовуватимемо лише під час запуску програми локально на машині розробника за допомогою команди dotnet run або шляхом налагодження програми з VS 22.

Керування прикладними середовищами

Розробка програмного забезпечення проходить кілька етапів. На початку ми можемо почати з чистої розробки та тестування, але після того, як ми почнемо розгортати нашу програму для виробництва, з’явиться багато проблем. Однією з цих проблем є наявність різних середовищ для роботи програми.

Наприклад, уявіть, що ваша програма спілкується з API, і цей API розроблено іншим розробником або командою. Програма розміщена в службі додатків Azure з двома слотами для розгортання (розробка/виробництво). Під час розробки ви збираєтеся використовувати URL-адресу API розробника, яка не вноситиме змін у робочу базу даних, а після внесення змін у робочому середовищі вам потрібно використовувати URL-адресу робочого слота.

У цьому сценарії буде важко змінити URL-адресу API, перш ніж надіслати її до робочої версії, а потім отримати її після завершення. Таким

чином, може бути допущено багато помилок, що призведе до розчарування процесу розробки. Саме тут вступає в дію керування середовищами, де ви можете писати код, відтворювати певний інтерфейс користувача або отримувати спеціальні конфігурації на основі середовища, у якому працює ваша програма, не роблячи занадто багато помилок, які затримують розробку та виробництво.

За замовчуванням, коли ви запускаєте програму Blazor через налагоджувач, вона запускається в середовищі розробки, а після публікації — у робочому середовищі.

У цьому розділі ми побачимо, як ми можемо налаштувати та отримати поточне середовище програми та отримати конфігурацію на основі цього.

Створення конфігураційного файлу на основі середовища

Щоб створити файл конфігурації, який використовуватиметься лише на етапі розробки, ви можете створити інший файл appsettings.json, але з назвою середовища перед розширенням, наприклад appsettings.Development.json. Щоб реалізувати це, зробіть наступні кроки:

1. Клацніть правою кнопкою миші папку wwwroot і виберіть Add | New Item. Виберіть файл JSON і назвіть його appsettings.Development.json.

2. Додайте ту саму властивість «ApiUrl», що існує в appsettings.json, і значення «Development» таким чином:

```
{
    "ApiUrl": "Development"
}
```

Коли ви запускаєте програму, ви помітите, що на екрані замість http://localhost:44332 відображатиметься значення Development. Ви повинні дотримуватися практики розділення налаштувань на основі середовища з самого початку, щоб це полегшило процес розробки.

Читання середовища всередині компонентів

У процесі розробки й у багатьох середовищах для вашої програми (наприклад, розробка, попередній перегляд і виробництво) вам потрібен

певний елемент інтерфейсу користувача або певна логіка, щоб програма працювала по-різному в різних середовищах. Основна реалізація для досягнення такого типу чуйності полягає в тому, щоб показувати певну функцію в попередньому перегляді, але приховувати її в робочій версії.

Щоб досягти згаданої поведінки, вам потрібно отримати значення поточного середовища у вашому коді та використати умову if, щоб визначити, як користувальницький інтерфейс або логіка поводяться в цьому середовищі.

У наступному прикладі ми збираємося використовувати IWebAssemblyHostEnvironment, вставивши його в компонент Index і видаливши компонент SurveyPrompt, якщо програма працює в середовищі розробки:

1. Відкрийте компонент Index у папці Pages і вставте службу IWebAssemblyHostEnvironment у компонент, але, звичайно, вам потрібно посилатися на простір імен, який містить службу, тобто Microsoft.AspNetCore.Components.WebAssembly.Hosting, як показує наступний фрагмент:

```
@page "/"
@using Microsoft.AspNetCore.Components.WebAssembly.Hosting
...
@inject IWebAssemblyHostEnvironment Host
<PageTitle>Index</PageTitle>
...
```

2. Наступним кроком є використання методу IsDevelopment, який повертає логічне значення, якщо поточне середовище дорівнює Development:

```
...
<p>Api Url: @Configuration["ApiUrl"]</p>
@if (!Host.IsDevelopment())
{
    <SurveyPrompt
      Title="How is Blazor working for you?" />
}
```

Після запуску програми на вашій машині в режимі налагодження компонент SuveryPrompt буде показано у робочому середовищі. Цей приклад демонструє чуйність функцій середовища, що дуже корисно.

Порада

IWebAssemblyHostEnvironment містить кілька методів, наприклад IsDevelopment(). Він також містить IsStaging() і IsProduction(). Ви також можете налаштувати назву за допомогою IsEnvionment("CUSTOM_ENVIORNEMNT_NAME").

Однією з тем, які потрібно розглянути, є явне налаштування поточного середовища. Ми розглянемо це в розділі 14 «Публікація програм Blazor WebAssembly» після публікації проекту, але ви можете піти глибше, прочитавши та дізнавшись більше про середовища на сторінці https://learn.microsoft.com/en-us/aspnet/core/blazor/fundamentals/environments?view=aspnetcore-7.0 .

Резюме

Протягом цього розділу ми ознайомилися з основами кожної програми Blazor WebAssembly, від налаштування інструментів розробки до створення першої програми за допомогою .NET CLI та Visual Studio 2022. Ми також відкрили анатомію та структуру програми визначаючи призначення кожного файлу та чому він там знаходиться. Ми розглянули DI, створили нашу першу службу, впровадили її в компонент і дослідили причини, чому це важливо для сучасної розробки програмного забезпечення.

Конфігурації є важливою частиною кожної програми. Щоб уникнути жорстко закодованих значень у вашому коді, потрібно враховувати середовище, у якому працює ваша програма. Ця додаткова увага може спростити процес розробки, особливо після публікації програми в робочій версії, коли інші функції ще розробляються. Управління середовищами та конфігураціями допомагає подолати невідповідності в розробці та керувати частиною, що розробляється.

У наступному розділі ви почнете реальні дії з Blazor, досліджуючи та розробляючи основи компонентів програми, які є основними будівельними блоками програми Blazor.

Розділ 2

Компоненти в Blazor

Компоненти є основними будівельними блоками кожної програми Blazor. Кожен компонент є самодостатньою частиною ***інтерфейсу користувача*** (UI) і пов'язаної з ним логіки та містить динамічну частину UI програми. Додаток Blazor — це, по суті, набір компонентів, розміщених разом, кожен з яких має власну відповідальність в інтерфейсі користувача, і взаємодія цих компонентів створює повну програму, яку ми прагнемо створити в цій книзі.

У цій главі ми збираємося зрозуміти концепцію компонентів з нуля, створимо наш перший компонент, а потім використаємо його. Ми відкриємо доступні механізми для передачі даних між компонентами, ввівши ***параметри***, ***каскадні параметри*** та ***зворотні виклики подій***.

Після цього ми розглянемо події життєвого циклу компонента та те, як ми можемо використовувати їх протягом життєвого циклу компонента. Нарешті, ми додамо деякі стилі CSS для кращого вигляду.

У цьому розділі ми розглянемо такі теми:

- Розуміння концепції компонентів
- Переміщення даних між компонентами
- Виявлення життєвого циклу компонента
- Стилізація компонентів за допомогою CSS

Технічні вимоги

Ви можете знайти вихідний код цієї глави за адресою https://github.com/PacktPublishing/Mastering-Blazor-WebAssembly/tree/main/Chapter_02.

Розуміння поняття компонентів

Додаток Blazor в основному складається з набору компонентів, які працюють разом для формування насичених динамічних додатків.

Компоненти Blazor формально відомі як компоненти Razor, у яких C# і HTML-код поєднані та використовується розширення файлу .razor. Компоненти можна вкладати один в одного, спільно використовувати та повторно використовувати в іншому проекті.

Розробляючи свій додаток Blazor, ви повинні думати про нього як про організацію, де кожен компонент є співробітником, відповідальним за певне завдання. Співпраця цих працівників є тим, що представляє робочий процес організації. Ця ж концепція стосується програми Blazor, у якій кожен компонент має мати чітку та конкретну роль в інтерфейсі користувача програми. У той же час кожен компонент повинен мати можливість отримувати відповідні дані від інших компонентів, щоб виконати свою частину роботи.

На наступному знімку екрана показано зразок програми для подкастів (Productive+, яку я розробив для відстеження використання свого часу), створеної за допомогою Blazor:

Рисунок 2.1 – Створена мною програма, яка розділена на невеликі компоненти, які складають повний інтерфейс користувача. Вихідний код для цього прикладу програми доступний за адресою https://github.com/aksoftware98/productivity-plus

На попередньому малюнку показано, як користувальницький інтерфейс конкретної програми містить кілька компонентів, які разом забезпечують розширений, багатий інтерфейс користувача. Сама сторінка також є компонентом із вкладеними компонентами, які містяться в ній. Нарешті, усі компоненти розміщуються разом у компоненті App.razor, який є коренем кожної програми.

Тепер, перш ніж почати створювати наш перший компонент, давайте розберемося з файлом Razor і синтаксисом, який ми використовуємо для створення компонентів.

Знайомство з Razor

Компоненти Blazor — це файли Razor, які дають нам надпотужний механізм для створення абсолютно динамічних та інтерактивних елементів інтерфейсу.

Файли-компоненти повинні мати назву, яка починається з великої літери, наприклад, AlertMessage.razor. Якщо натомість ви назвете компонент alertMessage.razor, ви зіткнетеся з помилкою під час компіляції, яка не дозволить компілятору створити програму.

Якщо ми подивимося на попередньо зібраний компонент Counter.razor, який постачається з новими проектами Blazor (компонент створює сторінку за замовчуванням, яка існує в кожній новій програмі Blazor і називається Counter) у наступному фрагменті коду, ми можемо помітити синтаксис компонента Razor на додаток до суміші коду C# і HTML:

```
@page "/counter"
<PageTitle>Counter</PageTitle>
<h1>Counter</h1>
<p role="status">Current count: @currentCount</p>
<button class="btn btn-primary" @onclick="IncrementCount">Click
me</
button>
```

```
@code {
    private int currentCount = 0;
    private void IncrementCount()
    {
       currentCount++;
    }
}
```

Як ви могли помітити, компонент Counter в основному написаний з використанням звичайних тегів HTML на додаток до фрагмента коду C#, який містить метод під назвою IncrementCount(), і приватний член під назвою currentCount.

У тегу абзацу, який постачається разом із компонентом, проект посилається на член C# private int у тексті, використовуючи @ char як префікс. Те саме стосується події onclick кнопки шляхом посилання на метод збільшення кількості цієї змінної.

Початок роботи зі створення компонентів за допомогою Razor є простим процесом, оскільки код не містить жодних складних розділів чи ініціалізацій; просто створіть свій інтерфейс користувача за допомогою HTML разом із кодом C#. Це справді так просто.

Однак перш ніж ми почнемо створювати наш перший компонент, давайте ознайомимося з синтаксисом і доступними способами написання виразів C# у поєднанні з кодом HTML. Окрім визначення деяких ключових слів, зарезервованих Razor, ми можемо визначити деякі специфікації для наших компонентів за допомогою директив Razor.

Синтаксис Razor

Давайте глибоко зануримося в синтаксис Razor і відкриємо доступні вирази Razor, які ми будемо використовувати для написання коду наших компонентів:

• ***Implicit Razor expressions*** (Неявні вирази Razor): ці вирази містять перші та найпростіші вирази для використання коду C# між тегами HTML: конструктори із символу @, за яким слідує член C#. Після компіляції проекту результат цього виразу буде відтворено та показано в HTML.

Нижче наведено приклад із нашого компонента Counter.razor:

<p role="status">Current count: @currentCount</p>

У попередньому коді ми можемо побачити вираз зі змінною currentCount у тегу абзацу.

Обчислення попереднього рядка полягає в тому, що значення змінної буде відображено в тегу HTML таким чином:

<p role="status">Current count: 0 </p>

• ***Explicit Razor expressions*** (Явні вирази Razor): синтаксис цих виразів дає вам більше можливостей для комбінування складних виразів C#. Вони використовують синтаксис символу @, за яким йдуть дві дужки (..), які містять вирази C#, які потрібно оцінити та відобразити.

У наведеному нижче прикладі показано оцінку змінної currentCount і визначення того, чи є вона парним числом як логічний результат (True, False):

<p>Is Even: @(currentCount % 2 == 0)</p>

Візуалізація попереднього рядка, якщо значення змінної currentCount дорівнює 4, створить такий HTML:

<p>Is Even: True </p>

• ***C# code blocks***: цей тип дозволяє писати блоки коду C# для вашого компонента Blazor, і код усередині не буде відтворено у HTML. Натомість код буде оголошено та виконано в складовому об’єкті.

Синтаксис блоків коду C# включає символ @ на додаток до ключового слова code, за яким слідують дві фігурні дужки { }, які містять код C#. Повертаючись до прикладу Counter.razor, ми можемо побачити, як блок коду використовується для створення змінної та функції для маніпулювання значенням цієї змінної наступним чином:

```
@code {
    private int currentCount = 0;
    private void IncrementCount()
    {

        currentCount++;
    }
}
```

Директиви Razor

За лаштунками кожен компонент Razor є класом C#, який генерується механізмом Razor після його аналізу. Razor має так звані директиви, які надають нам техніку маніпулювання процесом цього аналізу та надають певний контроль над деякими функціями чи специфікаціями цього згенерованого класу.

Razor має два типи директив, які поділяються залежно від способу їх застосування:

• ***Directives***: вони допомагають маніпулювати аналізом, специфікаціями та функціональністю компонента та застосовуються до компонента в цілому

• ***Directive attributes***: ці директиви застосовуються до деяких тегів і надають певні функції та маніпуляції з цим конкретним елементом

Давайте розглянемо деякі з найбільш часто використовуваних директив у компонентах Razor у наступній таблиці та визначимо призначення кожної з них:

Directive	**Usage**
@page	Робить компонент маршрутизируемим, реєструючи його в маршрутизаторі програми, і робить його доступним через певну URL-адресу за запитом у браузері через: @page "/counter" " Ми обговоримо це далі в Розділі 4, Навігація та маршрутизація.
@namespace	Дозволяє встановити явний простір імен для вашого компонента за допомогою: @namespace BooksStore.Client.Components За замовчуванням простір імен компонента — це ім’я проекту, за яким слідують імена папок, у яких він існує.
@inherits	Успадковує клас компонента від певного типу (реалізуючи успадкування для вашого компонента) через: @inherits MyBaseComponent
@inject	Впроваджує певну службу в компонент із DI контейнер через: @inject ILoggingService LoggingService

@code	Визначає блок коду C#:
	@code
	{
	    int currentCount = 0;
	    // More C# Code goes here
	}

@using	Посилається на простір імен у вашому компоненті за допомогою: @using System.Linq
@layout	Визначає макет, який використовуватиметься для вашого компонента: @layout BlankLayout Ми обговоримо це далі в розділі 3 «Розробка розширених компонентів у Blazor».
@bind	Реалізує прив'язку даних: <input @bind-Value="currentValue" /> Ми обговоримо це в розділі 3 «Розробка розширених компонентів у Blazor».
@implements	Реалізує інтерфейс: @implements IDisposable

Порада

Компоненти Razor підтримують часткові класи. Компоненти Razor — це в основному класи C#, тому ви можете мати HTML, розмітку та логіку C# в одному файлі або розмітку у файлі .razor, а код C# — в окремому файлі класу C# із назвою компонента та той самий базовий тип, але з ключовим словом partial, яке дозволяє вам розділити той самий клас C# на багато файлів.

Ця практика рекомендована, коли логіка C# компонента величезна, оскільки вона допомагає зберегти ваші файли невеликими та більш упорядкованими на додаток до використання повної потужності редактора C# у вашому IDE.

Створення вашого першого компонента Razor

Після ознайомлення з концепцією компонентів і вивчення основ файлу компонентів Blazor і його синтаксису настав час застосувати те, що ми навчилися, в життя.

Ми збираємося створити наш перший компонент, який буде карткою книги для відображення книги в нашій програмі. Ми почнемо з нуля, а потім додамо більше функціональних можливостей і стилів на ходу протягом усього розділу. Отже, почнемо:

1. Створіть файл компонента Razor у папці Shared, клацнувши папку правою кнопкою миші у Visual Studio. Виберіть Add | Razor component, дайте йому назву BookCard.razor і натисніть «Add».

Ми створили компонент у спільній папці Shared, оскільки це спільний компонент і використовуватиметься на кількох сторінках.

2. Після створення файлу ви готові почати писати свою першу розмітку.

3. Ми напишемо базовий дизайн HTML для картки, яка містить певну інформацію про книгу, і кнопку для додавання книги до кошика, як показано нижче:

```
<div style="border:1px solid black; padding:3px">
    <h6>Book 01</h6>
    <p>Author: Author Name</p>
    <p>Publishing date: 2022-07-02</p>
    <button style="width:100%">Add to Cart</button>
</div>
```

Тепер у нас є фіксований фрагмент HTML, але його можна повторно використовувати в багатьох компонентах, і саме цього ми збираємося досягти на останньому кроці.

4. Щоб відобразити компонент в іншому компоненті, все, що нам потрібно зробити, це вказати назву цього компонента в подвійних тегах <>. Щоб досягти цього в нашому прикладі, давайте відкриємо компонент Index.razor у папці Pages і посилаємося на нашу книжкову картку.

Оновлений файл має виглядати так:

```
...
<PageTitle>Index</PageTitle>
<h1>Hello, world!</h1>
Welcome to your new app.
<BookCard />
...
```

Тепер ми готові побачити наш перший багаторазовий компонент Razor, запустивши проект за допомогою кнопки «Пуск» у Visual Studio. Компонент Index автоматично відкриється як компонент за замовчуванням, і ми побачимо там картку нашої книги, як показано на наступному знімку екрана:

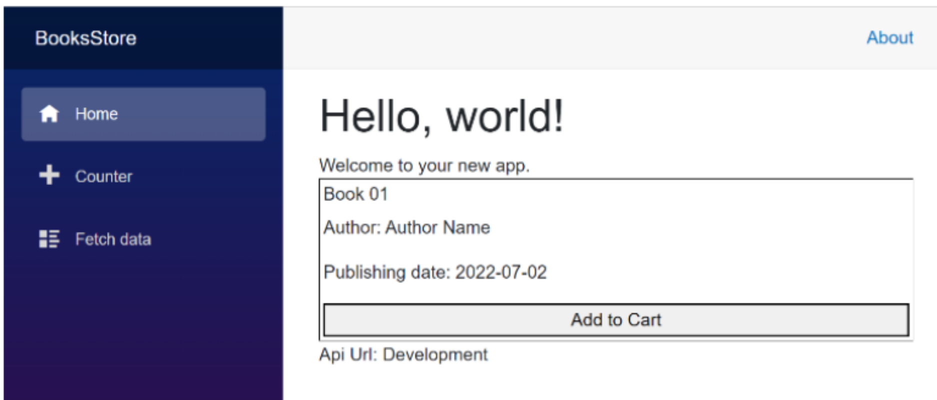


Рисунок 2.2 – Результат візуалізованого компонента BookCard на сторінці Index

Щиро вітаю! Ви успішно створили та використали свій перший компонент Razor після вивчення основ компонентів загалом і Razor зокрема. Тепер ми можемо зробити крок вперед, зробивши наш компонент трохи динамічним за допомогою різних методів для надсилання даних від батьківських компонентів до дочірніх і в протилежному напрямку, використовуючи параметри, каскадні параметри та зворотні виклики подій.

Переміщення даних між компонентами

Методи в C# працюють, приймаючи параметри для обробки та повертаючи інші типи даних після обробки. Компоненти Blazor працюють за такою ж концепцією. Оскільки кожен компонент представляє частину інтерфейсу користувача разом із своєю логікою, завжди існує необхідність надати компоненту певні дані, щоб він функціонував відповідно до встановленої логіки. Таким чином, функціонально компонент Blazor або рендерить дані, які вводяться певним чином, або реагує на основі встановленої логіки.

Компонент Blazor у деяких випадках потребує певних даних як вхідних даних або для візуалізації даних в інтерфейсі користувача, або для керування внутрішньою логікою на основі наданих значень. Крім того, у багатьох сценаріях компоненти мають можливість надсилати дані назад до батьківського. Ефективна співпраця компонентів програми шляхом спілкування один з одним дає нам чисту, добре функціонуючу та ефективну програму, яку ми намагаємося створити.

На щастя, Blazor надає нам потужні механізми для передачі даних між компонентами та в різних напрямках (від батьківських компонентів до дочірніх і навпаки).

Тепер настав час відкрити ці механізми та застосувати їх до компонента, який ми створили в попередньому розділі.

Параметри

Компонент Blazor може приймати параметри, як і будь-який метод C#, але визначення параметрів у Blazor зовсім інше. Параметри в компонентах Blazor — це властивості C#, прикрашені атрибутом [Parameter], і потім ми встановлюємо їхні значення, передаючи їх як атрибути під час виклику компонента.

Отже, давайте почнемо додавати деякі параметри до нашого компонента BookCard, щоб параметризувати його дані, приймаючи ці дані як параметри замість жорстко закодованих значень у HTML.

Щоб досягти цього, нам потрібно додати розділ @code до нашого компонента та визначити наступні параметри (Title, PublishingDate та Author), але властивості об’єкта (книги) можуть час від часу змінюватися під час циклу розробки програми. Тож замість того, щоб передавати кожну властивість як параметр, ми можемо створити клас Book, а потім передати об’єкт book компоненту.

Ця практика допоможе підтримувати ваші компоненти Blazor, оскільки коли ви додаєте нові властивості до своєї моделі, вам не потрібно додавати відповідний параметр до компонента або навіть до кількох компонентів, які використовують ту саму модель. Також немає необхідності додавати ці нові властивості як атрибути під час виклику компонента. Щоб досягти всього цього, нам потрібно зробити наступне:

1. Клацніть правою кнопкою миші на проекті, а потім виберіть Add | New Folder та дайте їй назву Models; папка буде містити моделі, які ми будемо використовувати в нашому проекті.

2. Додайте новий клас C# під назвою Book у папку Models, клацнувши папку правою кнопкою миші та вибравши Add | New Item із додаванням класу C# і присвоєнням йому імені Book.cs.

3. Далі нам потрібно додати властивості книги до створеного класу. Нам потрібні такі властивості: Title, PublishingDate і AuthorName, тому файл класу має виглядати так:

```
namespace BooksStore.Models;
public class Book
{
    public string? Title { get; set; }
    public string? AuthorName { get; set; }
    public DateTime PublishingDate { get; set; }
}
```

4. Перш ніж перейти до компонента BookCard, нам просто потрібно додати простір імен BooksStore.Models до файлу _Imports.razor, щоб нам не потрібно було додавати оператор using кожного разу, коли ми хочемо посилатися на модель:

```
...
@using BooksStore.Shared
@using BooksStore.Services
@using BooksStore.Models
```

5. Відкрийте файл BookCard.razor і модифікуйте його, щоб прийняти об’єкт Book як параметр, додавши директиву @code { } і параметр типу Book. Наступний фрагмент коду показує результат цього кроку:

```
     ...
     <button style="width:100%">Add to Cart</button>
</div>
@code
{
    [Parameter]
    public Book? Book { get; set; }
}
```

6. Після додавання параметра ми можемо посилатися на значення властивостей об’єкта Book за допомогою неявних виразів Razor у HTML через @ та ім’я властивості в об’єкті параметра (ми також будемо використовувати оператор із значенням nullable (?) просто у випадку, якщо значення параметра було нульовим), як показано в наступному фрагменті коду:

```
<div style="border:1px solid black; padding:3px">
    <h6>@Book?.Title</h6>
    <p>Author: @Book?.AuthorName</p>
```

```
    <p>Publishing date: @Book?.PublishingDate</p>
    <button style="width:100%">Add to Cart</button>
</div>
```

7. Останнім кроком є надання значення параметра Book компоненту під час його виклику. Поверніться до файлу Index.razor і додайте нову директиву @code {}, яка включатиме оголошення нового об’єкта Book:

```
...
@if (!Host.IsDevelopment())
{
    <SurveyPrompt Title="How is Blazor working for
       you?" />
}

@code
{
    private Book _firstBook = new Book
      {
        AuthorName = "John Smith",
        PublishingDate = new DateTime(2022, 08,
                                            01),
        Title = "Mastering Blazor WebAssembly"
      };
}
...
```

8. Змініть рядок, де він викликає компонент BookCard, передавши значення параметра Book за допомогою атрибута:

```
...
<BookCard Book="_firstBook" />
...
```

Отже, якщо ми запустимо проект прямо зараз, ми побачимо той самий компонент, але з даними, що відтворюються в порівнянні з попереднім. Ці дані передаються за допомогою параметрів під час виклику компонента, як показано на наступному знімку екрана:

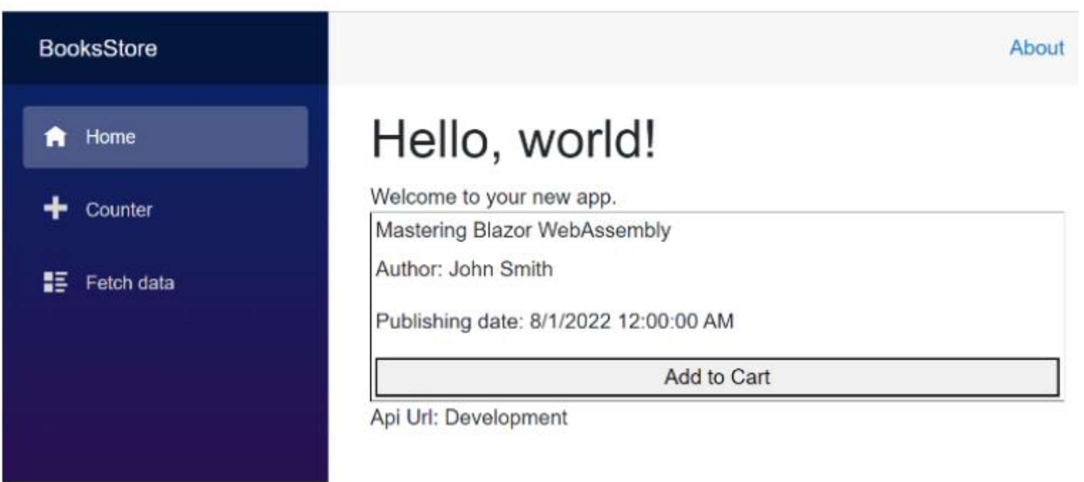


Рисунок 2.3 – Відтворений компонент після передачі параметрів

Той самий компонент тепер можна використовувати так само, як ми бачили в усьому додатку, і щоразу, коли вашому додатку потрібно показати цю частину інтерфейсу користувача. Параметр Book використовується для передачі даних, які потрібно відобразити в межах компонента, але зараз ми створимо інший параметр, який контролюватиме процес відтворення та функціональність компонента.

Другий параметр, який ми додамо, відповідатиме за показ або приховування кнопки «Додати до кошика», оскільки в деяких місцях програми нам просто потрібно бачити інформацію про книгу без необхідності додавати її до кошика для покупок.

Щоб досягти такої поведінки, ми створимо інший параметр типу Boolean під назвою WithButton і безпосередньо ініціалізуємо значення True у компоненті BookCard, як показано в наведеному нижче коді:

```
...
[Parameter]
public bool WithButton { get; set; } = true;
...
```

Тепер, після створення параметра, ми можемо обернути кнопку умовою if, щоб відобразити її, лише якщо значення цієї кнопки істинне, як у наступному фрагменті коду:

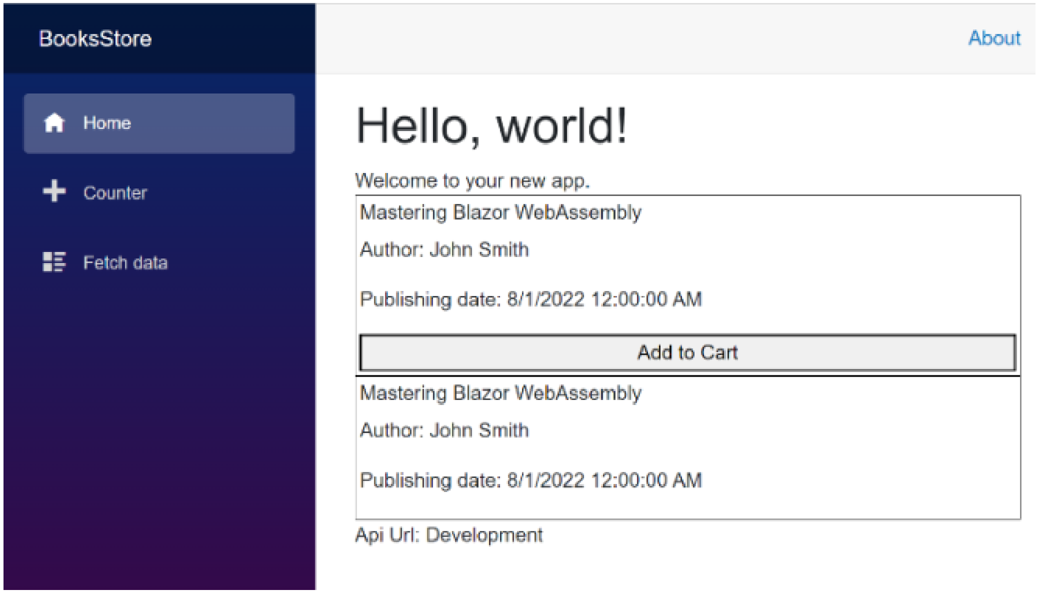
```
...
<p>Publishing date: @Book?.PublishingDate</p>
    @if (WithButton)
```

```
    {
        <button style="width:100%">Add to Cart</button>
    }
</div>
...
```

Тепер ми викличемо цей компонент ще раз у компоненті Index у файлі Index.razor, але надамо значення false для WithButton:

```
...
<BookCard Book="_firstBook" />
<BookCard Book="_firstBook" WithButton="false" />
<p>Api Url: @Configuration["ApiUrl"]</p>
...
```

Ви можете помітити, запустивши програму ще раз, що картка книги присутня двічі, але друга не містить кнопки «Add to Cart».



Малюнок 2.4 – Рендеринг двох книжкових карток, друга без кнопки