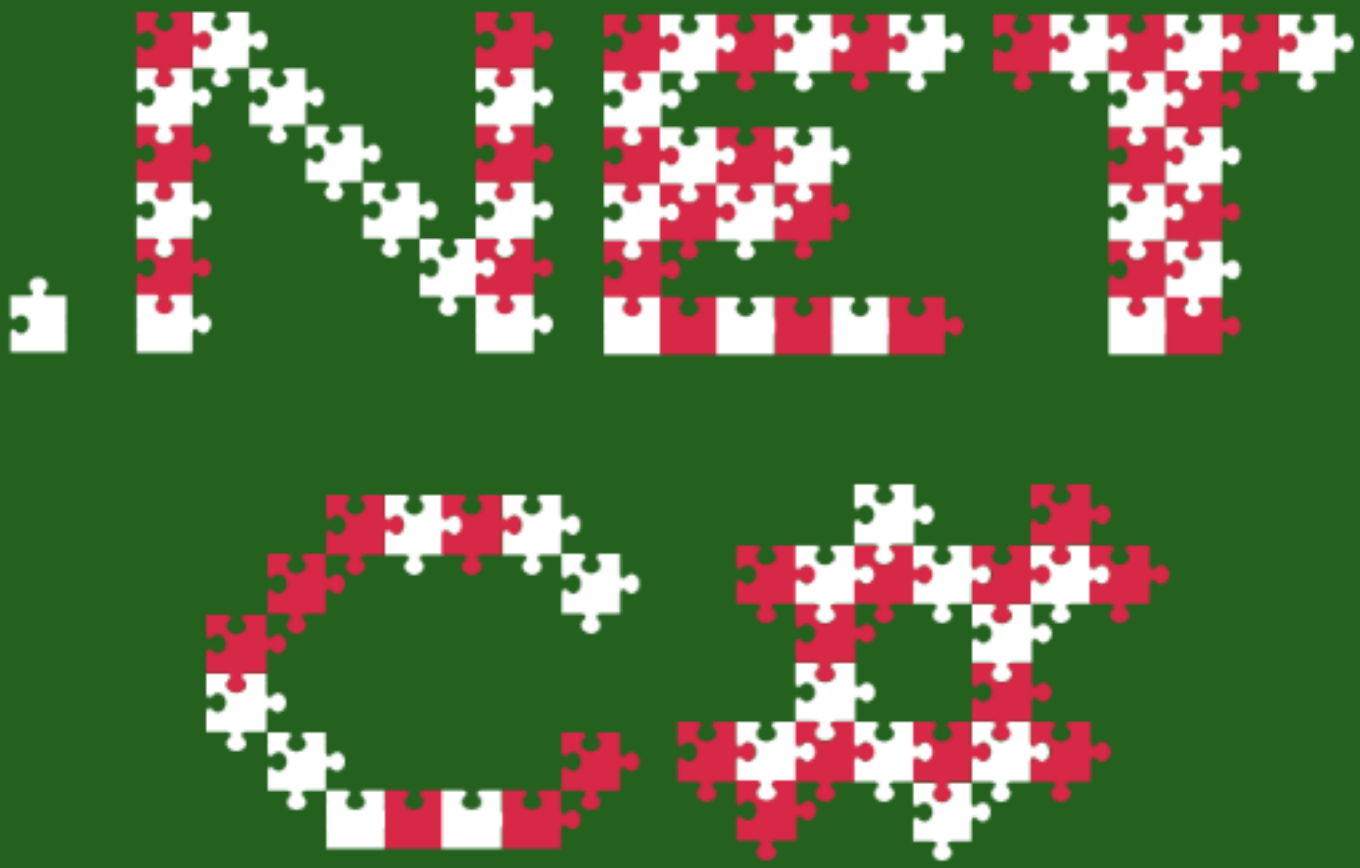
.NET 6 Blazor 快速體驗 Hands-on Lab

動手練習系列



Vulcan Lee 著

.NET 6.0 Blazor Server 開發快速體驗

Hands-On Lab 動手練習

Vulcan Lee

這本書的網址是 http://leanpub.com/NET6-Blazor-Quick-Overview

此版本發布於 2021-12-29

Contents

1. 關於本書 1

1.1 這本書能提供什麼 1

1.2 誰適合閱讀這本書 4

1.3 練習專案原始碼 4

2. 建立 Blazor Server Side 伺服器端的專案 5

2.1 安裝 Visual Studio 2022 5

2.2 Blazor UI 開發框架 7

2.3 建立 Blazor Server 專案 11

2.4 執行這個專案 16

2.5 ASP.NET Core Blazor 專案結構 19

2.5.1 Blazor Server 專案啟動 Program.cs 19

2.5.2 Blazor Server 啟動載入程序 _Host.cshtml 與 _Layout.cshtml . . 36

2.5.3 Blazor Server 框架進入點元件 App.razor 45

2.5.4 Blazor 框架預設版面配置元件 MainLayout.razor 46

2.5.5 功能選項切換導航面板元件 NavMenu.razor 48

2.5.6 依據路由資訊來顯示特定 Blazor 頁面元件 51

2.5.7 方便擴充的範本檔案 _Imports.razor 53

2.5.8 ASP.NET Core 設定 appsettings.json 54

2.5.9 Counter.razor 元件 54

2.5.10 FetchData.razor 元件 56

3. 版權頁 60

1. 關於本書

這是一本帶領對於 Blazor Server 開發技術有興趣的新手開發者，可以快速體驗這個微軟最新的網頁開發框架技術的開發過程，在這本書中，將不會講解枯澀的相關技術內容，而是設計一個應用情境，也就是一般常用的 CRUD (新增 Create, 查詢 Retrive, 更新 Update, 刪除 Delete) 的部落格文章應用程式需求開發，從無到有的帶領讀者透過 Visual Studio 2022 這個開發工具，設計出的 Blazor 伺服器端的這樣應用程式。

所以，購買本書的讀者，將會強烈建議讀者要跟著本書的內容，逐一進行專案的設計與練習，在每個練習階段，都會有不同要帶給讀者的學習方向；透過這樣的練習開發過程，就會明瞭 Blazor Server 這個開發框架技術究竟可以做到什麼樣的強大功能；這本書提供一個動手練習實作的說明操作步驟，體驗 Blazor 專案開發過程。因此，若您沒有這樣的興趣或者這樣的需求，請不要購買這本書。

因此，當讀者完成所有的專案練習開發，相信您對於 Blazor Server 這個優秀的開發框架必定有更清楚的認識，了解到不需要使用到繁雜的 JavaScript 程式語言，僅使用 Blazor 就可以做到那些網站設計上的功能。

1.1 這本書能提供什麼

在這本書裡面，將會提供 10 章的內容，分別是

- 建立 Blazor Server Side 伺服器端的專案

  了解如何透過 Visual Studio 2022 開發工具，開始建立一個伺服器端的 Blazor 專案，並且從這個 Blazor Server 專案範本所產生的檔案與相關內容進行了解 Blazor Server

運作方式與特色，這裡也會介紹該專案內建的兩個 Razor 元件 Component 設計方式。

- 使用 Entity Framework Core 存取資料庫

  在本書中所要設計的部落格文章管理紀錄，將會儲存在 Microsoft SQL Server 內，想要做到這樣的設計需求，最簡單的方式就是採用微軟提供的 Entity Framework Core 這套 ORM 套件，在這裡將會說明 Entity Framework Core 的基本架構與特色和核心運作機制，接著了解如何建立實體模型並且建立起一個資料庫與產生一些測試用的紀錄在資料表內；另外，針對資料庫處理要用到的 CRUD Create 新增、Retrive 查詢、Update 更新、Delete 刪除的計算方式，也會介紹如何使用 C# 程式碼來做到這些功能。

- 建立部落格文章頁面之 CRUD 功能

  現在將要開始實際進行建立一個 Blazor 頁面元件，在這裡設計出一個具有 CRUD 功能的互動網頁，更令人驚豔的是，所有的設計過程，完全僅使用到 C# 程式語言就可以完成這樣的設計工作，沒有用到任何一行 JavaScript 程式碼，因此，只要是身為 .NET C# 開發者，不論之前從事 Windows Forms、Web Forms、WPF、Xamarin 等等，將會透過 Blazor 這套 UI 開發框架幫助，成為一個全端 Web 開發者，整個學習過程相當的輕鬆與容易，無須花費大量精力與時間就可以學會 Blazor Server 開發技術。

- 使用 Bootstrap 強制回應 Modal 對話窗

  上一章所完成部落格文章管理頁面，是將資料表清單與紀錄編輯所用到的 HTML 透過元件內的布林型別屬性和資料綁定機制來動態變更瀏覽器上的 DOM 物件，如此做到切換網頁內容可以顯示不同操作畫面，在這一章中，將會使用 Bootstrap 5 所提供的強制回應對話窗樣式，讓紀錄表單編修的時候可以顯示一個流暢的對話窗，讓使用者在此對話窗中進行操作，形成一個流暢的操作體驗。

- 設計與使用修改與刪除 Blazor 元件

  上一章所完成部落格文章管理頁面，是將 HTML / CSS 都設計在同一個 Razor 元件檔案 (.razor) 內，現在需要使用 Blazor UI 開發框架所提供的一個絕佳功能，那就是可以把許多 UI 畫面，切割成為不同的 Razor 子元件，並且在 Blazor 路由頁面元件中參

考、使用這些子元件，讓這個網頁內容設計過程更加的容易與順暢，並且容易替換與維護。

- 將商業邏輯程式碼重構為 ViewModel

  到現在為止，所有的 UI 宣告標記 (HTML / CSS) 和商業邏輯程式碼 (C# 程式語言) 都寫在同一個 Razor 元件檔案 (.razor) 內，在此將要學習與使用關注點分離的設計方法，把網頁畫面的 UI 保留在 Razor 元件檔案內，而把商業邏輯的 C# 程式碼分離到 ViewModel 類別內，藉由透過 ASP.NET Core 所提供相依性注入服務機制，把 ViewModel 物件動態注入到 Razor 元件檔案內。

- 將資料庫存取程式碼分離出來

  成功的將 UI 與商業邏輯程式碼分離出來之後，接下來就是要把資料庫相關的程式碼，再度從 ViewModel 分離出來，讓 ViewModel 內要處理資料庫方面工作的時候，面對的是一個抽象介面，而在具體實作類別內，則是使用 Entity Framework Coew API 來存取資料庫，當然，這兩者之間還是要透過 ASP.NET Core 所提供相依性注入服務機制結合在一起。

- 設計使用者身分驗證與授權功能

  整個專案到現在僅剩下一個實務上經常會遇到的情境，那就是要能夠做到使用者的身分驗證與授權，在此將會使用 Cookie 來儲存使用者成功身分驗證之後的存取權杖，用來識別下次再度回到網站的時候，可以得知上次使用的使用者是哪位；另外也要控制僅有成功登入的使用者，可以使用剛剛設計的部落格文章管理頁面。

- 建立部落格文章 Web API

  最後，將要學習如何在 Blazor Server 專案內，啟用 Web API 程式設計能力，在此將會要來設計一個部落格文章管理的 RESTful Web API

- ASP.NET Core 6.0 Blazor Server 部署到 IIS

  現在整個 Blazor Server 專案已經完全開發完成，最後將會需要把這個專案部署到遠端的 Web 伺服器主機上，這篇文章將會要來說明如何進行部署到 Windows Server IIS 上相關工作。

1.2 誰適合閱讀這本書

對於任何想要學習 Blazor 這個開發技術，可以透過本書的開發練習說明，逐步了解到如何真正的設計出一個 Blazor 實用專案。

讀者必須具備 .NET / C# 的開發經驗、HTML / CSS 的基本認識即可，對於開發工具而言，本書是在 Windows 10 作業系統下，使用任何 Visual Studio 2022 的版本，就可以進行開發。

1.3 練習專案原始碼

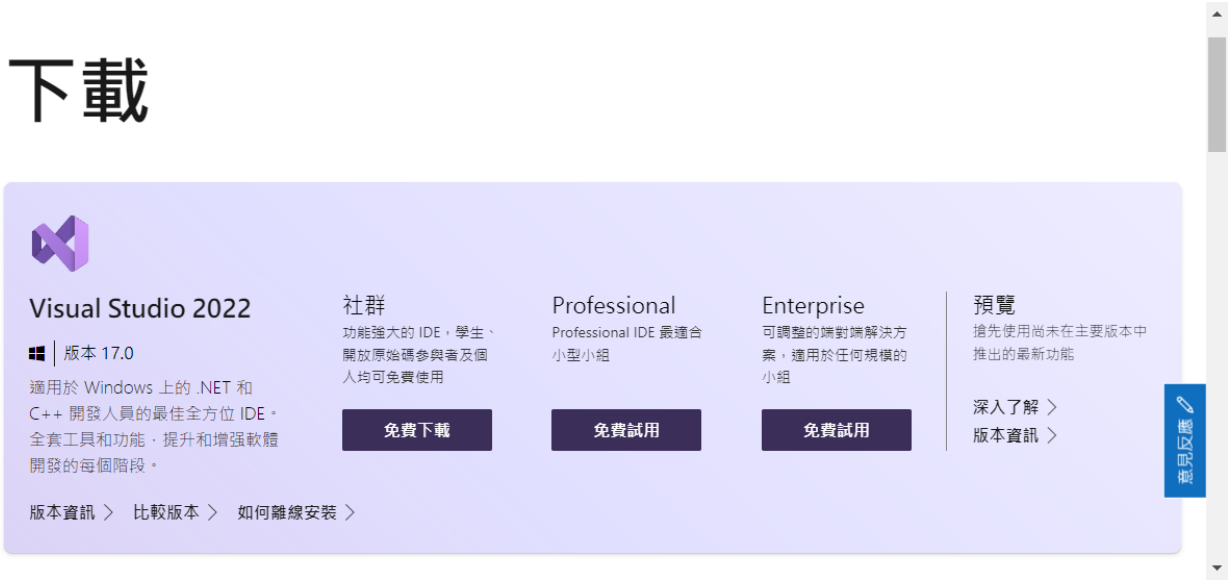
本電子書中的練習專案原始碼，可以從 https://github.com/vulcanlee/Blazor-Quick-Overview2.git[1] 取得

[1] https://github.com/vulcanlee/Blazor-Quick-Overview2.git

2. 建立 Blazor Server Side 伺服器端的專案

2.1 安裝 Visual Studio 2022

在要進行 ASP.NET Core Blazor Server 專案開發前，建議可以安裝 Visual Studio 2022 這套 IDE 開發工具，透過瀏覽器開啟這個網址 https://visualstudio.microsoft.com/zh-hant/downloads/ ，便可以下載這個 Visual Studio 2022 安裝程式，其中共有三個版本可以選擇 [社群] 、[Professional] 、[Enterprise]；這個 [社群] 版本的 Visual Studio 2022 為免費版本，也是可以完全進行 Blazor Server / Wasm 專案開發，因此，若不想付費購買其他版本的 Visual Studio 2022，那就請直接下載這個版本即可。



Visual Studio 2022 下載網頁

完成下載安裝程式之後，請開啟與執行這個安裝程式，此時便會看到如下截圖，現在僅需要勾選 [ASP.NET 與網頁程式開發] 這個工作負載選項即可使用 ASP.NET Core，ASP.NET，HTML/JavaScript 及容器 (包含 Docker 支援) 建立 Web 應用程式及其開發環境會用到的工具。

Visual Studio 2022 安裝程式

2.2 Blazor UI 開發框架

Blazor 為 ASP.NET Core 下用於開發互動式網頁用戶端 UI 開發框架，當然還有 MVC[1] 與 Razor Page[2] 這兩種開發框架可以選擇。

[1] https://docs.microsoft.com/zh-tw/aspnet/core/mvc/overview?view=aspnetcore-6.0&WT.mc_id=DT-MVP-5002220

[2] https://docs.microsoft.com/zh-tw/aspnet/core/razor-pages/?view=aspnetcore-6.0&tabs=visual-studio&WT.mc_id=DT-MVP-5002220

Blazor 是個 UI 開發框架

很多人對於有許多誤解，其中一個就是 Blazor 僅僅是個 UI 開發框架，他負責處理網頁 UI 呈現與更新上的相關處理工作，透過資料綁定 Data Binding 機制與轉譯 Render 工作運作，可以達到即時更新瀏覽器端 DOM 內容，也就是可以在網頁上顯示最新的畫面。

對於其他關於 Web 應用程式上運作的機制與功能，就需要透過 ASP.NET Core 這套開發框架來進行設計，例如：使用者的身分驗證與授權、執行訊息日誌寫入功能、相依性入設計模式等等。

在 2019 年之前，想要開發 Web 專案，在 W3C[3] 組織下僅規畫三種語言制式語言可以使用，這是因為瀏覽器也僅能夠看得懂這三種語言所設計的網頁內容，這分別是 HTML / CSS / JavaScript，前兩者 HTML / CSS 是用於網頁視覺內容方面設計用的標記宣告語言，後者 JavaScript 則是可以在瀏覽器執行的唯一腳本 Script 程式碼；這也就是說，想要開發出 Web 應用程式，若是不懂得或者學會操作 JavaScript 這個腳本程式語言，是無法設計出動態網頁內容的 Web 應用程式。

這樣的限制條件對於許多 .NET 開發者而言，想要能夠開發出 Web 應用程式專案，就必須要再度額外學習 JavaScript 這套腳本程式語言，然而，絕大數的開發者並無法深入掌控 JavaScript 程式語言設計技能，只能夠在網路上找到適用範例程式碼或者別人寫好的套件，直接套用到自己的專案來使用；不幸的是，當遇到問題或者產生出錯誤臭蟲 Bug 的時候，因為學藝不精，就只能經常撞牆來解決問題。

這樣的情況終於等到 Blazor 的出現而有所改變，Blazor 這項技術是在 2019 年推出，當採用 Blazor 做為網站應用程式開發專案，對於網頁呈現部分，依舊使用 HTML / CSS 這兩個語言來進行宣告與設計，沒有任何改變；然而對於該應用程式專案商業邏輯部分，則可以擺脫以往只能夠唯一使用 JavaScript 腳本 Script 窘困情境，現在竟然可以 .NET / C# 程式語言來進行設計用戶端與伺服器端的商業邏輯，沒錯，你沒有看錯，就是使用 C# 程

[3] https://www.w3.org/

式語言，而在本書中所講解的動手練習開發的 Web 應用程式專案，就完全沒有寫任何一行 JavaScript 指令碼，就可以做到以往 Web 應用程式所做出的效果。

由於因為前端 Frontend 與後端 Backend 的應用程式都是使用 C# 程式語言來開發，所以對於用戶端 Client 與伺服器端 Server 開發出來的程式碼是可以共用的，因為可以透過建立類別庫 Class Library 來共用所設計出來的類別；這也說明了當在進行 Blazor 專案開發的時候，可以繼續使用 .NET 開發生態下所提供的各項 NuGet 套件產品，使得整體開發上可以更佳的輕鬆駕馭，這是因為所有的程式設計過程，就如同在其他 .NET 開發工具下所進行的程式開發與設計過程都是相同的，另外，這也代表可以享受到 .NET 生態系統下所提供的效能、可靠性和安全性帶來的好處。

這項特色對於身為 .NET 開發者而言，真是天大好消息，因為，對於已經孰悉 C# 程式語言的程式設計師來說，無須透過痛苦再度學習其他程式語言或開發框架，也無須經歷陡峭的學習曲線過程，便可以輕鬆地成為網站應用程式的全端開發工程師，沒錯，使用一套程式語言，就可以同時開發出前端與後端的 Web 應用程式。

這並不代表 Blazor 僅能夠使用 C# 程式語言來進行開發，在 Blazor 開發框架下，可以做到在 C# 程式語言內呼叫 JavaScript 函數腳本，當然也可以讓 JavaScript 腳本來呼叫 C# 內的靜態方法或者特定執行個體內的方法。

在 Blazor UI 開發框架下，對於整體 UI 的操作將是採用資料綁定 Data Binding 與 UI 轉譯 Render 機制來成動態網頁設計需求，在整體學習與應用上變成相當的容易與簡單，因為許多需要複雜處理機制與設計過程，微軟已經將其封裝在 Blazor UI 開發框架內了。

在要進行 Blazor 專案開發之前，要先了解到 Blazor 有兩種託管 Hosting 模式可以選擇，第一種是設計用來在 ASP.NET Core 後端伺服器上來運行，這樣的設計方式稱之為 Blazor Server 專案；第二種則是讓整個 Web 應用程式與 .NET CLR 直接在用戶端的瀏覽器來運行，是的，讀者你沒有看錯，C# 確實可以在瀏覽器上來運行，在 2019 年之前，想要開發網頁應用程式，JavaScript 腳本是一定需要學會的，這是因為在瀏覽器中，唯一可以使用的程式語言就只有 JavaScript，沒有其他的選擇，這樣的設計方式稱之為 Blazor

WebAssembly。

WebAssembly 的誕生與帶來 Web 應用程式開發一到曙光

在 2012 年開始就已經進行增強 JavaScript 執行效能與可以在 Web 應用程式中採用不同程式語言的開發研究，直到 2016 年推出第一套實驗性的產品，各家瀏覽器廠商就已經支援這樣的實驗性計畫功能，可以直接在瀏覽器上來執行，W3C 組織在 2019 年發布了 WebAssembly 作為瀏覽器上支援的第四種語言 (這四個語言分別是 HTML, CSS, JavaScript, WebAssembly)，而採用這樣方式來進行開發的專案就稱之為 Blazor WebAssembly 或者 Blazor Wasm 專案。

- Blazor Server

  若使用 Blazor Server 託管模式執行的時候，這個 Blazor Server 專案將會使用 ASP.NET Core 應用程式方式在伺服器上執行，若在瀏覽器的用戶端開啟該網站服務的端點 URL，此時將會透過 _Host.cshtml 這個 Razor 頁面 Page 做為啟動載入程序 Bootstrap，一旦 Blazor 系統準備完成，接下來的 UI 更新、事件處理及 JavaScript 呼叫會透過連接來處理 SignalR ，而所有用戶端的執行狀態內容都會儲存在 ASP.NET Core 伺服器端。
- Blazor Wasm (WebAssembly)

  對於使用 Blazor WebAssembly 託管模式執行的時候，這個 Blazor WebAssembly 將會透過 index.html 這個靜態宣告標記文件，做為啟動載入程序 Bootstrap，這裡將會把 Mono CLR 執行時期檔案 (這裡已經轉換成為 WebAssembly 語言) 載入到瀏覽器內，接著載入相關 .NET BCL 與這個專案的組件 Assembly 檔案 (通常為 .dll 副檔案名稱)，此時初始化工作就完成了，一旦 Blazor 系統準備完成，接下來應用程式會直接在瀏覽器 UI 執行緒上執行，而且 UI 更新、事件處理及 JavaScript 呼叫，也就是說，所有的程式碼就將在瀏覽器上運行。

無論選擇採用哪種託管模式來進行開發，對於 Blazor UI 框架下的應用程式和元件模型，都是採用相同的方式與觀念來進行開發。

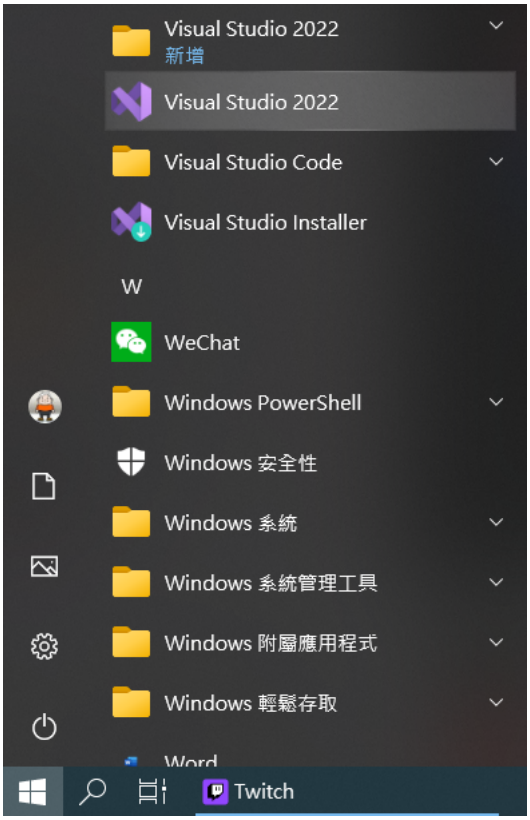
本書將是說明如何採用 Blazor Server 託管模式進行開發 Web 應用程式，接下來就來了解如何進行 Blazor Server 專案的開發做法與過程。

2.3 建立 Blazor Server 專案

首先，先要來學習如何透過 Visual Studio IDE 開發工具來建立一個 Blazor 伺服器端的專案

- 打開 Visual Studio 2022 開發工具

  在 Windows 10 作業系統的左下方點選搜尋功能圖示 (放大鏡圖片)，在搜尋文字輸入盒內輸入 `visual studio` 文字，此時，將會看到 [Visual Studio 2022 Current] 這個應用程式圖示，點選這個圖示便可以開啟 Visual Studio 2022 開發工具，或者可以從左下角點選開始視窗圖示，找到 Visual Studio 2022 應用程式。



Visual Studio 應用程式

本書範例程式碼將會採用 .NET 6.0 開發框建來進行設計，因此，建議直接將 Visual Studio 2022 + .NET 6 安裝到這台電腦上，方便進行練習；若尚未安裝 Visual Studio 2022，請參考前面說明內容。

- 當 [Visual Studio 2022] 對話窗出現之後，點選右下方的 [建立新的專案] 按鈕

現在要開始建立一個新的 Blazor Server 練習專案

Visual Studio 建立新的專案按鈕

- 在 [建立新專案] 對話窗內

- 設定上方的 [程式語言] 過濾條件為 [C#]

  如此，這裡專案清單範本就僅會顯示 C# 程式語言可以使用的專案範本
- 設定上方的 [專案類型] 過濾條件為 [Web]

  如此，這裡專案清單範本就僅會顯示與 Web 有關可用專案範本
- 找出 [Blazor Server 應用程式] 這個專案開發範本項目
- 點選這個專案開發範本

專案範本

- 請點選右下角 [下一步] 按鈕
- 出現 [設定新的專案] 對話窗
- 在 [專案名稱] 欄位輸入 `Blogger`

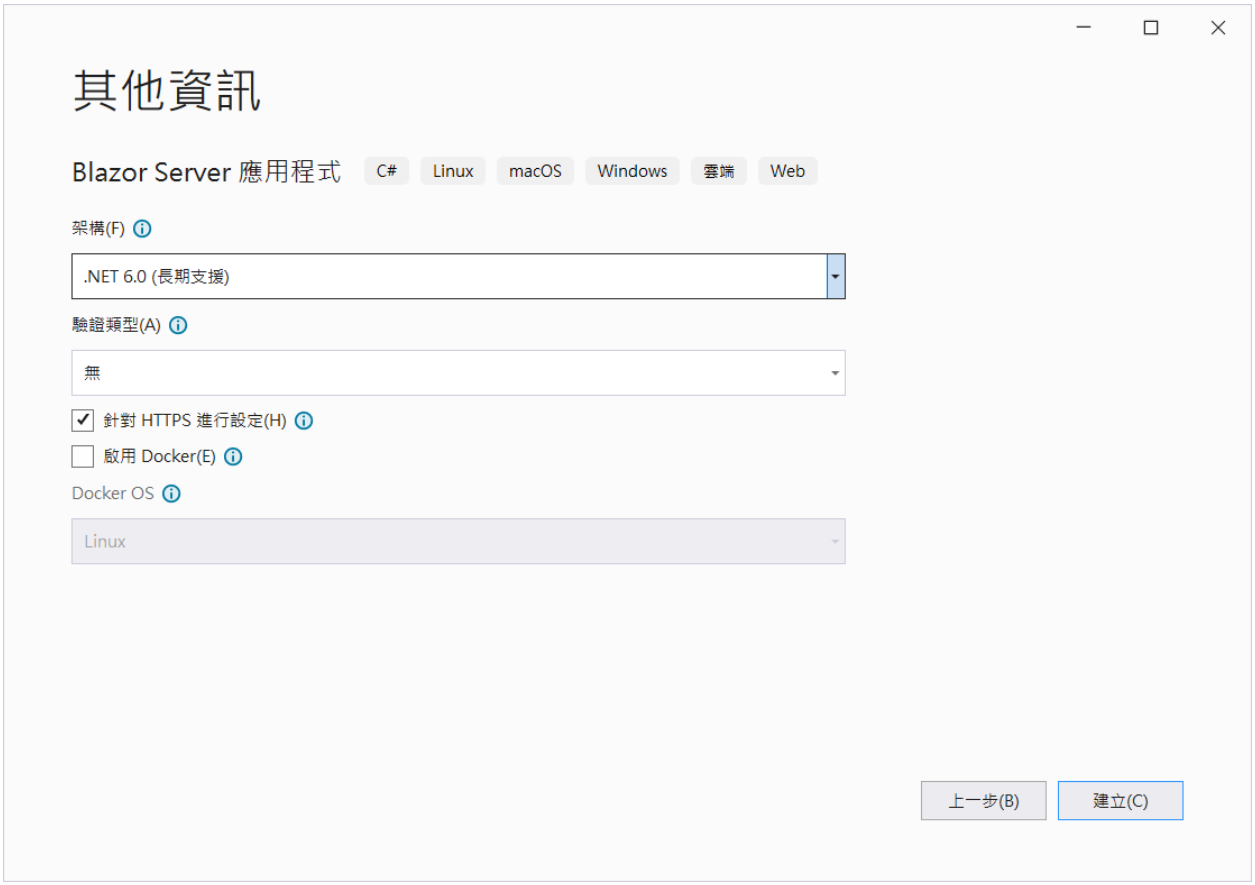
- 在 [位置] 欄位輸入你要儲存這個專案的路徑

設定新的專案

- 點選右下角 [下一步] 按鈕
- 接下來將會看到 [其他資訊] 對話窗

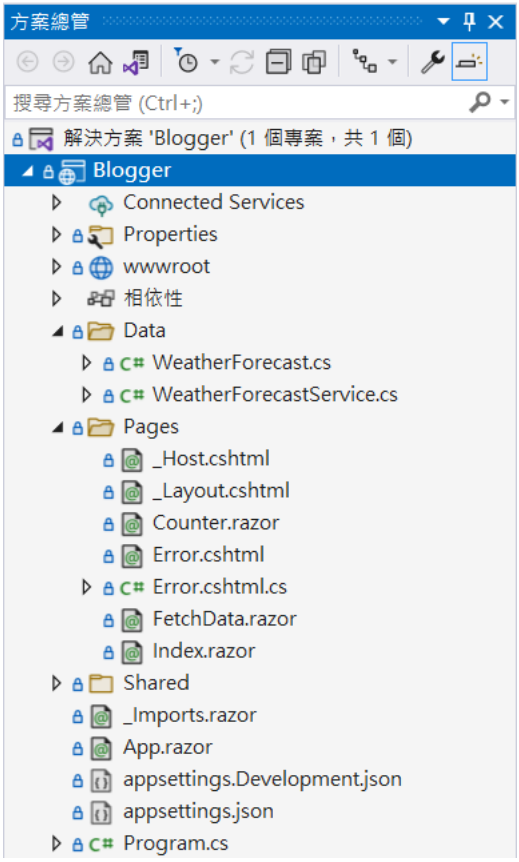
  在這個對話窗內，將可以額外進行這個專案的其他的設定
- 確認 [目標 Framework] 下拉選單欄位，設定為 [.NET 6.0 (長期支援)]

  選擇 [.NET 6.0 (長期支援)] 做為此專案的目標 Framework，將可以在此 Blazor Server 專案內使用最新的 Blazor 所提供的功能
- 這裡可以根據當時開發專案需要，自行決定是否有調整 Blazor 專案的其他特性，不過，在這裡將不需要做任何額外的設定
- 點選右下角的 [建立] 按鈕



其他資訊

- 請等待一段時間，這個 Blazor Server 專案已經建立完成
- 完成後的 Blazor Server 專案，將可以從 Visual Studio 2022 內的 [方案總管] 視窗內，看到這個 Blazor Server 專案的整個架構，這裡的架構資訊將會如同底下截圖



Blazor 方案結構

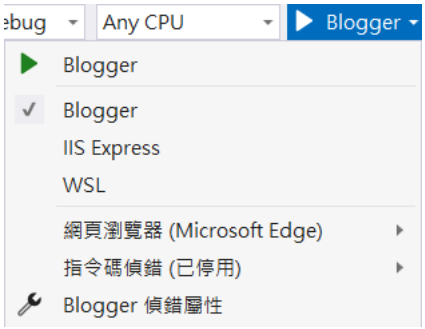
2.4 執行這個專案

在了解 Blazor Server 整體專案結構之前，先來把這個專案範本建置、啟動執行，看看 Blazor Server 預設範本提供那些功能。

- 在上方功能表的 [測試] 項目的下方，看到一個 [Blogger] 下拉選單

選擇啟動 Web Server 模式

- 點選該下拉選單右方的黑色三角形，一旦出現彈出功能表，可以看到 [IIS Express] 選項，若切換使用 [IIS Express] 選項，則表示要使用本機 IIS 方式來執行這個專案



使用哪種 Web Server 託管方式來執行這個專案

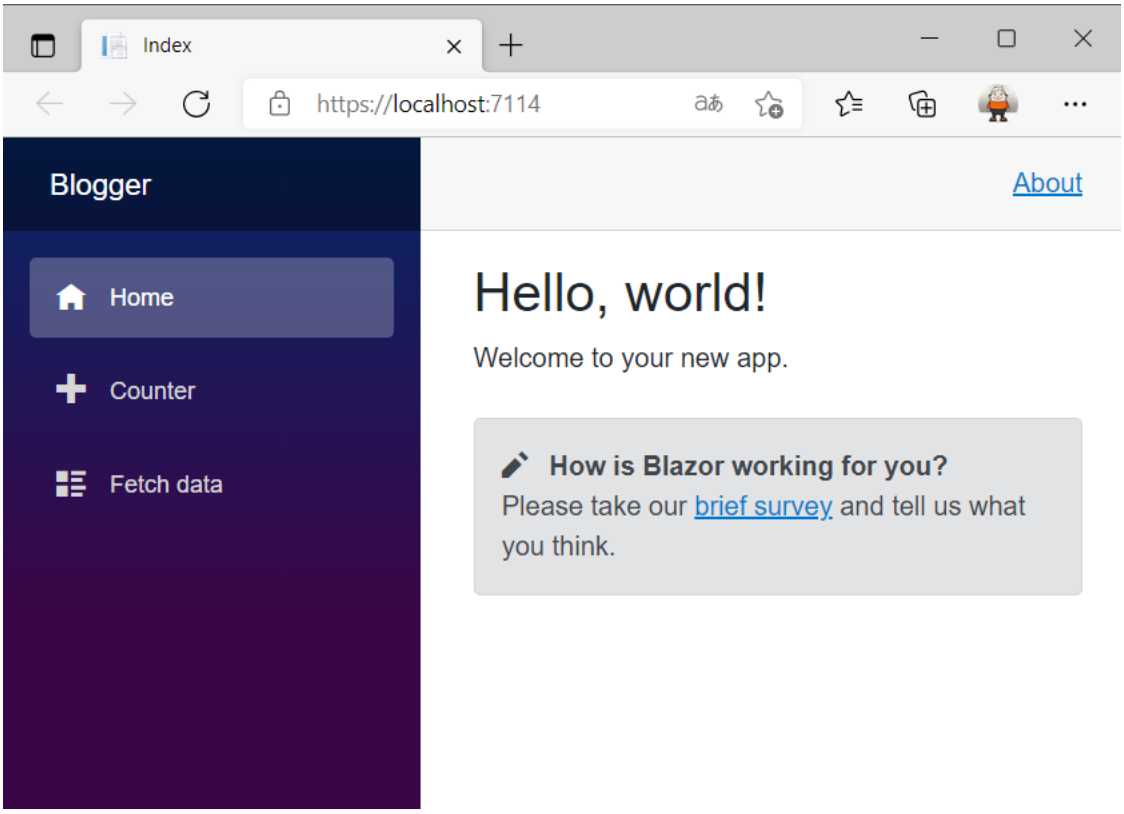
- 在這裡將會使用 Kestrel Web 服務來執行這個專案，因此，請維持 [Blogger] 這個選項

  建議使用預設的 Kestrel Web 服務來執行 Blazor Server 專案，因為，在開發階段，若有任何例外異常產生出來，可以隨時切換到 Blazor 專案執行日誌輸出主控台視窗，看到相關例外異常訊息與呼叫堆疊內容，以便可以快速找到發生問題原因與所在程式碼位置。

  所謂 Kestrel[4] 是適用於 ASP.NET Core 的跨平臺 Web 伺服器

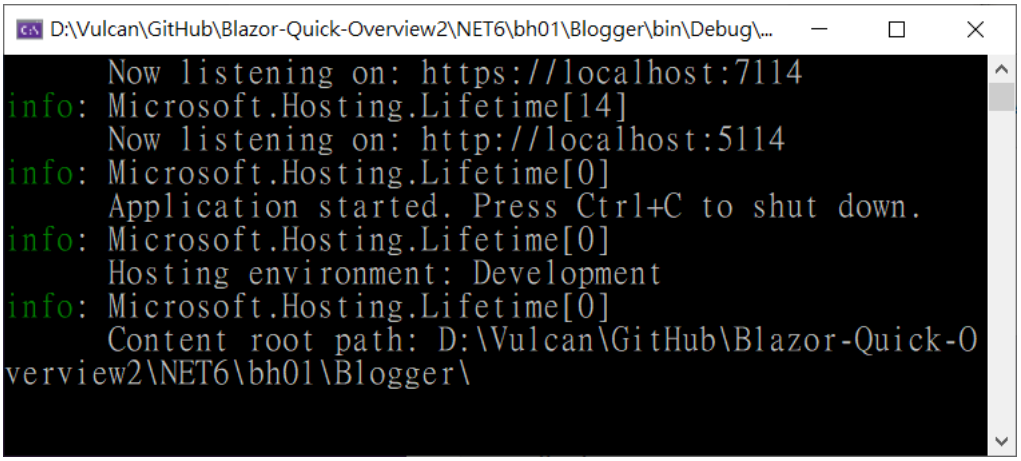
- 請點選工具列上方的綠色三角形，或者按下 F5 ，開始執行這個 Blazor 專案
- 此時，將會在瀏覽器上出現底下畫面

[4] https://docs.microsoft.com/zh-tw/aspnet/core/fundamentals/servers/kestrel?view=aspnetcore-6.0&WT.mc_id=DT-MVP-5002220



Blazor Server 專案第一次執行結果

- 另外，也會看到一個 [命令提示字元] 視窗出現，這裡可以看到整個 Blazor 專案執行的時候，相關的日誌訊息會寫到這個視窗內



Blazor 專案執行日誌輸出主控台視窗

2.5 ASP.NET Core Blazor 專案結構

現在要來介紹 Blazor Server 專案預設產生的相關檔案與其啟動過程。

2.5.1 Blazor Server 專案啟動 Program.cs

因為 Blazor Server 專案是建構於 ASP.NET Core 架構下，在 .NET Core / ASP.NET Core 的開發框架下，都是屬於一個 Console 主控台應用程式類型的專案，而這個檔案 [Program.cs] 為 ASP.NET Core 應用程式進入點 Entry Point，也就是當這個 Blazor Server 專案啟動之後，會先執行 [Program.cs] 檔案內的 [Main] 方法內的程式碼。

打開這個 [Program.cs] 檔案，將會看到如下的程式碼

```
using Blogger.Data;
using Microsoft.AspNetCore.Components;
using Microsoft.AspNetCore.Components.Web;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddRazorPages();
builder.Services.AddServerSideBlazor();
builder.Services.AddSingleton<WeatherForecastService>();

var app = builder.Build();

// Configure the HTTP request pipeline.
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    // The default HSTS value is 30 days. You may want to change this for production\
 scenarios, see https://aka.ms/aspnetcore-hsts.
    app.UseHsts();
}

app.UseHttpsRedirection();

app.UseStaticFiles();

app.UseRouting();


app.MapBlazorHub();
app.MapFallbackToPage("/_Host");

app.Run();
```

從 .NET 6 開始，使用 Console 產生範本的新專案會與舊版不同的程式碼，而在 Blazor Server 專案內也是相同的，更多這方面的資訊可以參考 新的 C# 範本會產生最上層的語句[5] 。在 .NET 6.0 下，對於原先在 .NET 5.0 之前的 Main() 方法，將進行簡化處理，將不會看到 Main() 這個方法，也就是說，不再需要寫入過多的程式碼，即可以做到 Main() 方

[5] https://docs.microsoft.com/zh-tw/dotnet/core/tutorials/top-level-templates?WT.mc_id=DT-MVP-5002220

法程式碼的效果。

在這裡來比對 .NET 5.0 與 .NET 6.0 所產生的 [Program.cs] 檔案有何差異，底下將會是在建立 Blazor Server 專案過程中，若在 [其他資訊] 對話窗內的 [目標 Framework] 下拉選單欄位中選擇了 [.NET 5.0 (目前)] 選項，此時所產生的專案內的 [Program.cs] 檔案將會如下

```
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Logging;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace Blogger
{
    public class Program
    {
        public static void Main(string[] args)
        {
            CreateHostBuilder(args).Build().Run();
        }

        public static IHostBuilder CreateHostBuilder(string[] args) =>
            Host.CreateDefaultBuilder(args)
                .ConfigureWebHostDefaults(webBuilder =>
                {
                    webBuilder.UseStartup<Startup>();
                });
    }
}
```

從這兩個 .NET 5.0 / .NET 6.0 版本的開發框架下所產生的 [Program.cs] 程式碼中進行比較，在 .NET 5.0 中，相關的服務 Service 設定與中介軟體 Middleware 設定，都會使

用 [Startup] 這個類別來進行設計，一旦呼叫了 [CreateHostBuilder] 方法將會產生一個 [IHostBuilder] 物件，該物件將會使用 [Startup] 類別內的程式碼進行調整這個 Web 應用程式該如何運作，這樣，此 ASP.NET Core Web 服務也就啟動了。

在 .NET 6.0 中，則是直接套用了 [新的 C# 範本會產生最上層的語句] ，不再使用了 [Startup] 這個類別，而是產生一個 [WebApplicationBuilder] 物件來進行該 Web 服務 Service 的註冊與相關運作參數的設定，接著透過這些設定內容，產生一個 [WebApplication] 物件，將會用來註冊需要用到的中介軟體 Middleware 服務到 HTTP 管道 Pipeline 內，完成之後就會將此 ASP.NET Core Web 服務啟動了。

接下來就來了解 .NET 6.0 架構下所產生的 [Program.cs] 做了哪些工作？

在 [Program.cs] 檔案內的第 5 行程式碼 `var builder = WebApplication.CreateBuilder(args);` ，就是這個 Blazor 專案所要執行的第一個敘述，這裡透過 WebApplication.CreateBuilder[6] 靜態方法將會取得一個 [WebApplicationBuilder] 物件，這表示了一個 Web 應用程式和服務的建立器執行個體。

有了這個 [WebApplicationBuilder] 物件，便可以透過該物件內所提供了許多屬性，進行用於進行該 Web 應用程式的運作行為設定，這裡會有幾個重要的屬性，例如：

- [Configuration] 屬性型別為 [ConfigurationManager] ，用於要撰寫之應用程式的設定提供者集合，當然，這個屬性也實作了 [IConfiguration] 介面，因此，可以存取該 ASP.NET Core 中的相關設定值內容。

ASP.NET Core 的設定

更多關於 ASP.NET Core 的設定資訊，可以參考 https://docs.microsoft.com/zh-tw/aspnet/core/fundamentals/configuration/?view=aspnetcore-6.0 此網頁介紹

[6] https://docs.microsoft.com/zh-tw/dotnet/api/microsoft.aspnetcore.builder.webapplication.createbuilder?view=aspnetcore-6.0&WT.mc_id=DT-MVP-5002220

- [Environment] 屬性型別為 [IWebHostEnvironment] ，提供應用程式正在執行之 Web 主控環境的相關資訊，例如，取得 [IFileProvider] 或設定指向 WebRootPath 檔案路徑為何，或者得或設定包含 Web 應用程式內容檔案的目錄絕對路徑
- [Logging] 屬性型別為 [ILoggingBuilder] ，用來要撰寫之應用程式的記錄提供者集合，便可以加入第三方的日誌應用套件，方便管理日誌相關資訊。
- [Host] 屬性型別為 [ConfigureHostBuilder] ，用來設定主控制項專屬的 [IHost-Builder] 屬性，但不會建立；若要在設定之後建立，請呼叫 Build()。

[Services] 屬性型別為 [IServiceCollection] ，用於要撰寫之應用程式的服務集合，也就是說，透過該屬性便可以相關服務型別註冊到 ASP.NET Core 的 IoC / DI 相依性注入服務容器 Container 內。

找到第 7 行的註解 `// Add services to the container.` ，在其底下的 3 行敘述將會分別進行註冊 Blazor Server 應用專案會使用到的相關服務到相依性注入服務 DI / Dependency Injection Container 容器 Container 內；在 ASP.NET Core 專案內，內建了相依性服務注入服務，有了此功能，在整體專案開發與設計上，便可以依據 SOLID 原則內的 [DIP / Dependency Inversion Principle] 相依反轉原則來進行設計與撰寫程式碼，透過此設計原則所產生出來的程式碼將會具有 [低耦合] 、[高可維護姓] 、[方便進行測試] 等特性。

接下來是在第 8 行的敘述 `builder.Services.AddRazorPages();` 這裡將會註冊 Razor Page 系統會使用到的相關服務到 Blazor Server 相依性服務注入容器內，在 Blazor Server 專案下會必定需要使用到 ASP.NET Core Razor Page 功能的原因在於，需要透過 Razor Page 架構，動態產生與取得一個 HTML 宣告標記文件內容。

一旦瀏覽器讀取到此 HTML 文件之後，將會分析這個 HTML 文件內容，載入更多的 CSS / JavaScript 文件檔案，當然，其中一個最為重要的就是要執行 [_framework/blazor.server.js] 這個 JavaScript 程式碼腳本 Script，該程式碼將會啟動瀏覽器用戶端需要用到的相關渲染 / 轉譯 Render 會使用的 JavaScript 程式碼腳本與啟動需要與後端 ASP.NET Core 伺服器做雙向即時通訊的 SignalR 功能，因此，可以稱這個

Razor Page (也就是在 Pages 資料夾下方的 _Hosts.cshtml) 檔案為 Blazor Server 應用程式的啟動載入程序 Bootstrap。

若想要更進一步了解 [AddRazorPages] 這個方法做了哪些事情，可以參考 https://source.dot.net/#Microsoft.AspNetCore.Mvc.RazorPages/DependencyInjection/MvcRazorPa，底下將列出該延伸方法的程式碼內容

```
  internal static void AddRazorPagesServices(IServiceCollection services)
  {
    // Options
    services.TryAddEnumerable(
      ServiceDescriptor.Transient<IConfigureOptions<RazorViewEngineOptions>, R\
azorPagesRazorViewEngineOptionsSetup>());

    services.TryAddEnumerable(
      ServiceDescriptor.Transient<IConfigureOptions<RazorPagesOptions>, RazorP\
agesOptionsSetup>());

    // Routing
    services.TryAddEnumerable(ServiceDescriptor.Singleton<MatcherPolicy, Dynamic\
PageEndpointMatcherPolicy>());
    services.TryAddSingleton<DynamicPageEndpointSelectorCache>();
    services.TryAddSingleton<PageActionEndpointDataSourceIdProvider>();

    // Action description and invocation
    services.TryAddEnumerable(
      ServiceDescriptor.Singleton<IActionDescriptorProvider, CompiledPageActio\
nDescriptorProvider>());
    services.TryAddEnumerable(
      ServiceDescriptor.Singleton<IPageRouteModelProvider, CompiledPageRouteMo\
delProvider>());
    services.TryAddSingleton<PageActionEndpointDataSourceFactory>();
    services.TryAddEnumerable(ServiceDescriptor.Singleton<MatcherPolicy, Dynamic\
PageEndpointMatcherPolicy>());

    services.TryAddEnumerable(
      ServiceDescriptor.Singleton<IPageApplicationModelProvider, DefaultPageAp\
plicationModelProvider>());
    services.TryAddEnumerable(
```

```
            ServiceDescriptor.Singleton<IPageApplicationModelProvider, AutoValidateA\
ntiforgeryPageApplicationModelProvider>());
        services.TryAddEnumerable(
            ServiceDescriptor.Singleton<IPageApplicationModelProvider, Authorization\
PageApplicationModelProvider>());
        services.TryAddEnumerable(
            ServiceDescriptor.Singleton<IPageApplicationModelProvider, TempDataFilte\
rPageApplicationModelProvider>());
        services.TryAddEnumerable(
            ServiceDescriptor.Singleton<IPageApplicationModelProvider, ViewDataAttri\
butePageApplicationModelProvider>());
        services.TryAddEnumerable(
            ServiceDescriptor.Singleton<IPageApplicationModelProvider, ResponseCache\
FilterApplicationModelProvider>());

        services.TryAddSingleton<IPageApplicationModelPartsProvider, DefaultPageAppl\
icationModelPartsProvider>();

        services.TryAddEnumerable(
            ServiceDescriptor.Singleton<IActionInvokerProvider, PageActionInvokerPro\
vider>());
        services.TryAddEnumerable(
            ServiceDescriptor.Singleton<IRequestDelegateFactory, PageRequestDelegate\
Factory>());
        services.TryAddSingleton<PageActionInvokerCache>();

        // Page and Page model creation and activation
        services.TryAddSingleton<IPageModelActivatorProvider, DefaultPageModelActiva\
torProvider>();
        services.TryAddSingleton<IPageModelFactoryProvider, DefaultPageModelFactoryP\
rovider>();

        services.TryAddSingleton<IPageActivatorProvider, DefaultPageActivatorProvide\
r>();
        services.TryAddSingleton<IPageFactoryProvider, DefaultPageFactoryProvider>();

#pragma warning disable CS0618 // Type or member is obsolete
        services.TryAddSingleton<IPageLoader>(s => s.GetRequiredService<PageLoader>(\
));
#pragma warning restore CS0618 // Type or member is obsolete
        services.TryAddSingleton<PageLoader, DefaultPageLoader>();
        services.TryAddSingleton<IPageHandlerMethodSelector, DefaultPageHandlerMetho\
dSelector>();
```

```
    // Action executors
    services.TryAddSingleton<PageResultExecutor>();

    services.TryAddTransient<PageSaveTempDataPropertyFilter>();
}
```

在第 9 行程式碼為 `builder.Services.AddServerSideBlazor();` ，這行敘述將會把 Blazor Server 需要用到的相關服務註冊到 DI 容器內，例如，在這個方法內將會使用 `services.TryAddScoped<ProtectedLocalStorage>();` 與 `services.TryAddScoped<ProtectedSessionStorag` 兩個 ASP.NET Core 用於存取受保護的瀏覽器儲存體服務，前者使用瀏覽器端的 Window.localStorage 來進行取得或儲存資料，而後者則是使用 Window.sessionStorage 來進行取得或儲存資料；對於想要將資料載入或儲存至瀏覽器儲存體的任何元件中，可以在 Blazor 元件 Component 內使用指示詞 `@inject` ，便可以注入這個服務物件，例如：`@inject ProtectedLocalStorage BrowserStorage`。

> 根據 Mozilla Web API 官方文件 Window.localStorage[7] 與 Window.sessionStorage[8] 說明: localStorage 為一唯讀屬性, 此屬性允許您存取目前文件 (Document) 隸屬網域來源的 Storage 物件; 與 sessionStorage 不同的是其儲存資料的可存取範圍為跨瀏覽頁狀態 (Browser Sessions).localStorage 的應用與 sessionStorage 相似, 除了 localStorage 的儲存資料並無到期的限制, 而 sessionStorage 的儲存資料於目前瀏覽頁狀態結束的同時將一併被清除—也就是目前瀏覽器頁面被關閉的同時.

若想要更進一步了解 [AddServerSideBlazor] 這個方法做了哪些事情，可以參考 https://source.dot.net/#Microsoft.AspNetCore.Components.Server/DependencyInjection/Compon，底下將列出該延伸方法的程式碼內容

[7] https://developer.mozilla.org/zh-TW/docs/Web/API/Window/localStorage
[8] https://developer.mozilla.org/zh-TW/docs/Web/API/Window/sessionStorage

```
    public static IServerSideBlazorBuilder AddServerSideBlazor(this IServiceCollecti\
on services, Action<CircuitOptions>? configure = null)
    {
        var builder = new DefaultServerSideBlazorBuilder(services);

        services.AddDataProtection();

        services.TryAddScoped<ProtectedLocalStorage>();
        services.TryAddScoped<ProtectedSessionStorage>();

        // This call INTENTIONALLY uses the AddHubOptions on the SignalR builder, be\
cause it will merge
        // the global HubOptions before running the configure callback. We want to e\
nsure that happens
        // once. Our AddHubOptions method doesn't do this.
        //
        // We need to restrict the set of protocols used by default to our specializ\
ed one. Users have
        // the chance to modify options further via the builder.
        //
        // Other than the options, the things exposed by the SignalR builder aren't \
very meaningful in
        // the Server-Side Blazor context and thus aren't exposed.
        services.AddSignalR().AddHubOptions<ComponentHub>(options =>
        {
            options.SupportedProtocols.Clear();
            options.SupportedProtocols.Add(BlazorPackHubProtocol.ProtocolName);
        });

        // Register the Blazor specific hub protocol
        services.TryAddEnumerable(ServiceDescriptor.Singleton<IHubProtocol, BlazorPa\
ckHubProtocol>());

        // Here we add a bunch of services that don't vary in any way based on the
        // user's configuration. So even if the user has multiple independent server\
-side
        // Components entrypoints, this lot is the same and repeated registrations a\
re a no-op.
        services.TryAddEnumerable(ServiceDescriptor.Singleton<IPostConfigureOptions<\
StaticFileOptions>, ConfigureStaticFilesOptions>());
        services.TryAddSingleton<ICircuitFactory, CircuitFactory>();
        services.TryAddSingleton<IServerComponentDeserializer, ServerComponentDeseri\
alizer>();
```

```
        services.TryAddSingleton<ICircuitHandleRegistry, CircuitHandleRegistry>();
        services.TryAddSingleton<RootComponentTypeCache>();
        services.TryAddSingleton<ComponentParameterDeserializer>();
        services.TryAddSingleton<ComponentParametersTypeCache>();
        services.TryAddSingleton<CircuitIdFactory>();
        services.TryAddScoped<IErrorBoundaryLogger, RemoteErrorBoundaryLogger>();

        services.TryAddScoped(s => s.GetRequiredService<ICircuitAccessor>().Circuit);
        services.TryAddScoped<ICircuitAccessor, DefaultCircuitAccessor>();

        services.TryAddSingleton<CircuitRegistry>();

        // Standard blazor hosting services implementations
        //
        // These intentionally replace the non-interactive versions included in MVC.
        services.AddScoped<NavigationManager, RemoteNavigationManager>();
        services.AddScoped<IJSRuntime, RemoteJSRuntime>();
        services.AddScoped<INavigationInterception, RemoteNavigationInterception>();
        services.AddScoped<AuthenticationStateProvider, ServerAuthenticationStatePro\
vider>();

        services.TryAddEnumerable(ServiceDescriptor.Singleton<IConfigureOptions<Circ\
uitOptions>, CircuitOptionsJSInteropDetailedErrorsConfiguration>());
        services.TryAddEnumerable(ServiceDescriptor.Singleton<IConfigureOptions<Circ\
uitOptions>, CircuitOptionsJavaScriptInitializersConfiguration>());

        if (configure != null)
        {
            services.Configure(configure);
        }

        return builder;
    }
```

在第 10 行程式碼為 `builder.Services.AddSingleton<WeatherForecastService>();` ，這行敘述將會把在這個專案內自行設計的一個天氣預報服務註冊到 DI 容器內，而該服務在容器內生命週期為單一狀態，即一旦有用戶端要求注入該服務物件成功之後，不論何時任何用戶端再度注入這個服務物件，都會取得同一個服務物件，類似具有靜態物件的生命週期，一直到這個 ASP.NET Core 應用程式結束為止；如此，當要顯示天氣預報頁面的時

候，便可以透過注入該服務物件，取得最新的天氣預報資訊，並且透過 Blazor 資料綁定機制，顯示該服務物件得到的內容到瀏覽器的頁面上。

這個 [WeatherForecastService] 類別程式碼將位於 [Data] 資料夾的 [WeatherForecast-Service.cs] 檔案內，其程式碼如下所示

```
namespace Blogger.Data
{
  public class WeatherForecastService
  {
    private static readonly string[] Summaries = new[]
    {
    "Freezing", "Bracing", "Chilly", "Cool", "Mild", "Warm", "Balmy", "Hot", "Sw\
eltering", "Scorching"
  };

    public Task<WeatherForecast[]> GetForecastAsync(DateTime startDate)
    {
      return Task.FromResult(Enumerable.Range(1, 5).Select(index => new Weathe\
rForecast
      {
        Date = startDate.AddDays(index),
        TemperatureC = Random.Shared.Next(-20, 55),
        Summary = Summaries[Random.Shared.Next(Summaries.Length)]
      }).ToArray());
    }
  }
}
```

這個天氣預報服務物件之 [GetForecastAsync] 方法，將會產生出未來五天的隨機天氣預報資訊

對於第 12 行的敘述 `var app = builder.Build();` ，將會建立此 Web 應用程式和服務，該方法會回傳 [WebApplication] 型別的物件，該物件表示可以用於進行這個 Web 應用程式將會用到的 HTTP 管線 Pipeline 中介軟體 Middleware 有哪些和路由 Routing；這兩者

將會透過加入指定的中介軟體來做到，一旦一個 HTTP 請求 Request 來到這個 Web 應用程式，隨即會透過這裡所設定中介軟體 Middleware 來逐一進行檢查，若有違反該中介軟體規範者，將會提前送出 HTTP 回應 Response 到用戶端內。

找到第 14 行的註解 `// Configure the HTTP request pipeline.` ，在其之後的敘述，將會用來宣告這個 Web 應用程式會使用到那些 HTTP 管道 Pipeline 中介軟體，與要使用哪種路由設定方式。

第 15 行的 `if (!app.Environment.IsDevelopment())` 敘述將會檢查現在所啟動 Web 應用程式若不是在開發階段所執行的 (這裡指的是將會透過第12行的敘述 `var app = builder.Build();` 所取得之 [WebApplication] 型別的物件內屬性來得知)，若此條件為真，一旦 HTTP 請求發生了例外異常，將會攔截例外狀況、記錄這些例外狀況，接著將會顯示 [Pages] 資料夾內的 [Error.cshtml] Razor 頁面內容，底下將會是這個 Razor 頁面的宣告標記內容。

在 Blazor 元件間操作，但是卻發生產生例外異常的處置做法

當一個 HTTP 請求傳送來到 Web 應用程式，在處理該 HTTP 請求的時候，發生了拋出例外異常情況，將會顯示 [Error.cshtml] Razor 頁面內容；不過，當這個 Blazor Server Web 應用程式已經顯示在網頁上，接著在各個 Blazor 元件間進行操作，但是卻發生拋出例外異常的情況，此時，將不會使用 [Error.cshtml] Razor 頁面內容來顯示錯誤訊息。

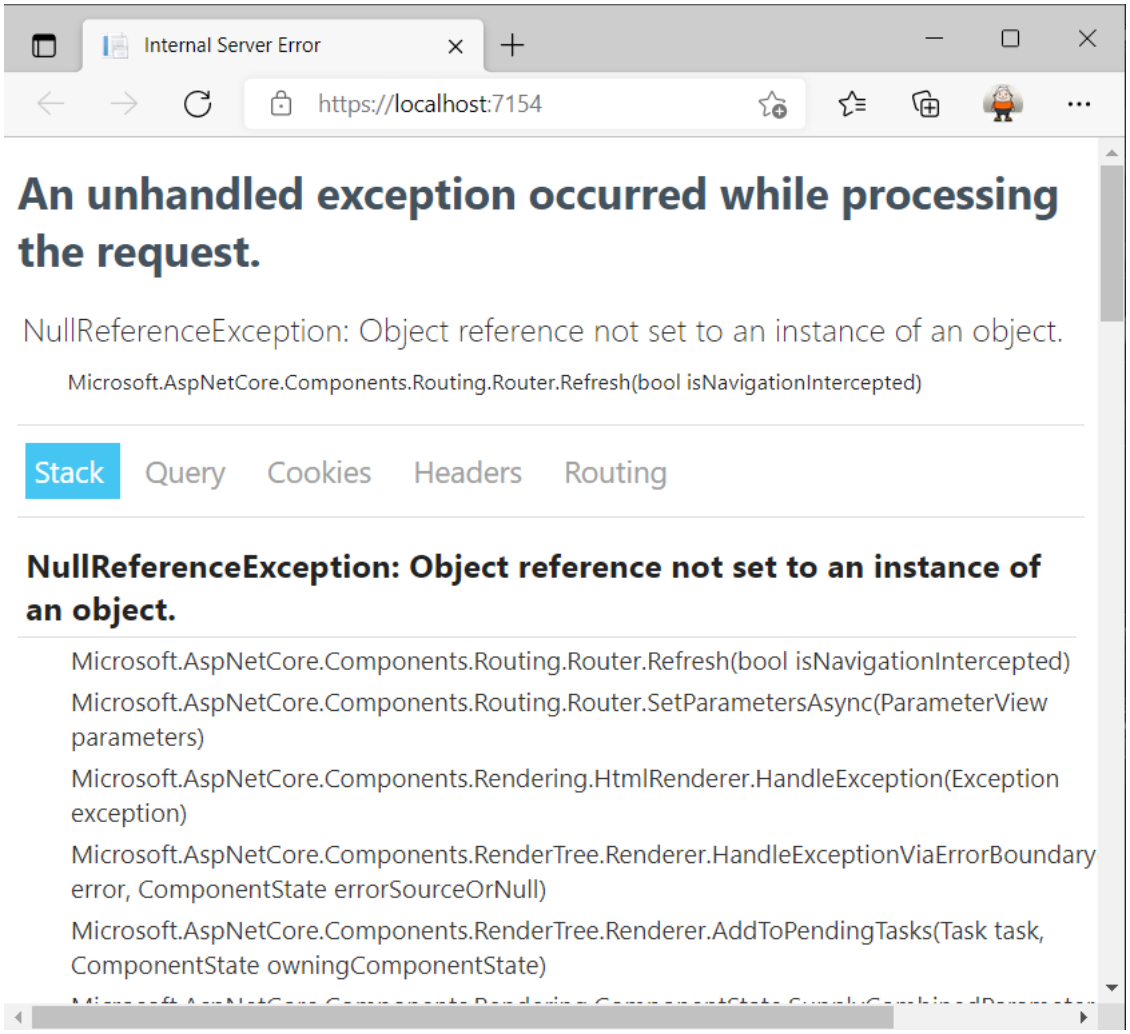
```
@page
@model Blogger.Pages.ErrorModel

<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0, maximum-sc\
ale=1.0, user-scalable=no" />
    <title>Error</title>
    <link href="~/css/bootstrap/bootstrap.min.css" rel="stylesheet" />
    <link href="~/css/site.css" rel="stylesheet" asp-append-version="true" />
</head>

<body>
    <div class="main">
        <div class="content px-4">
            <h1 class="text-danger">Error.</h1>
            <h2 class="text-danger">An error occurred while processing your request.\
</h2>

            @if (Model.ShowRequestId)
            {
                <p>
                    <strong>Request ID:</strong> <code>@Model.RequestId</code>
                </p>
            }

            <h3>Development Mode</h3>
            <p>
                Swapping to the <strong>Development</strong> environment displays de\
tailed information about the error that occurred.
            </p>
            <p>
                <strong>The Development environment shouldn't be enabled for deploye\
d applications.</strong>
                It can result in displaying sensitive information from exceptions to\
 end users.
                For local debugging, enable the <strong>Development</strong> environ\
ment by setting the <strong>ASPNETCORE_ENVIRONMENT</strong> environment variable to \
<strong>Development</strong>
                and restarting the app.
```

```
        </p>
      </div>
    </div>
</body>

</html>
```

現在要來看看當一個 HTTP 請求來到 ASP.NET Core Web 應用程式之後，卻發生了例外異常，對於開發階段 Development Stage 與正式產品階段 Production Stage 會看到什麼樣的畫面內容。

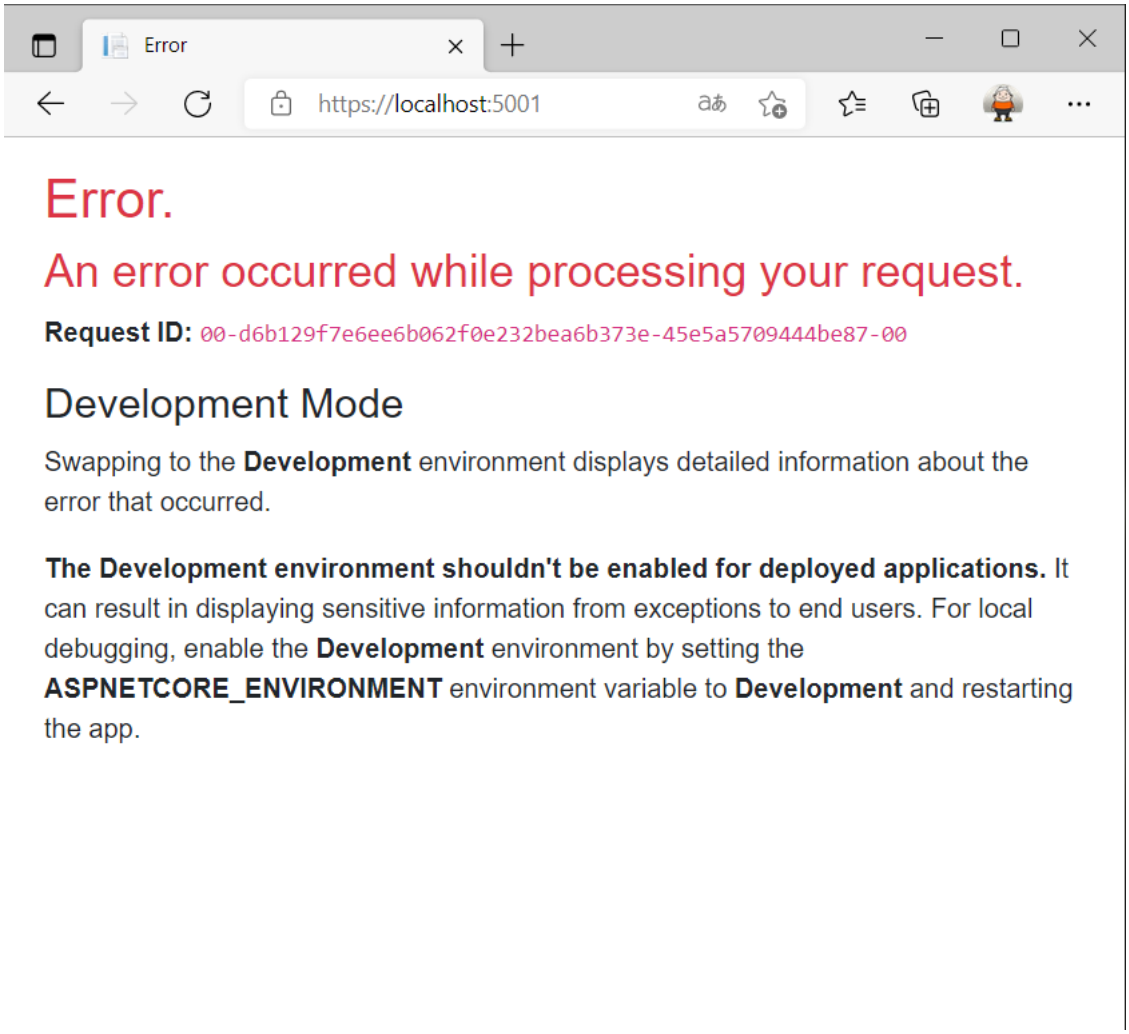
> 請務必要了解這裡所講解的內容，將會有助於保護該專案安全特性，避免專案內部程式碼資訊外洩到非開發者人員上，並且當此 Web 應用程式產生問題之後，可以在最短的時間內找到發生了什麼問題與找出是哪行程式碼造成此問題，並且可以進行修復。

當在開發階段 Development Stage ，產生一個 HTTP 請求到 ASP.NET Core 服務，若又產生了例外異常，這樣的情境將會造成看到這樣的底下螢幕截圖內容 (此時因為是在開發階段執行，因此將不會執行 `app.UseExceptionHandler("/Error");` 這個中介軟體所提供的功能)，從這個頁面內容中，可以輕鬆地看到此次發生的例外異常所造成的原因與呼叫堆疊內容，透過這些資訊，將會有助於開發者快速找出問題發生所在。



開發階段 Development Stage 例外異常

當在正式產品階段 Production Stage ，產生一個 HTTP 請求到 ASP.NET Core 服務，若又產生了例外異常，這樣的情境將會看到底下螢幕截圖內容；在該網頁上看不到任何關於此次問題發生了什麼錯誤訊息，也看不到例外異常呼叫堆疊內容，這樣可以保護到機密的資訊不會很輕易的外洩，然而，也造成當時發生了什麼問題，也無法進一步的追查窘境。



正式產品階段 Production Stage 例外異常

因此，若在正式產品發佈階段想要能夠查看到究竟發生了什麼問題產生了此次例外異常，需要透過第三方的日誌 Logger 套件來解決這樣需求，這些日誌套件可以將錯誤例外異常資訊寫入到伺服器上的本機檔案內或者資料庫與其他地方，如此，管理者便可以透過查看該日誌檔案來查看發生了什麼問題。

第 19 行的 `app.UseHsts();` 敘述將會使用 HTTP Strict 傳輸安全性通訊協定 (HSTS) 標頭傳送給用戶端，這裡的 Hsts 指的是 HTTP Strict Transport Security Protocol[9]，而 UseHsts

[9] https://cheatsheetseries.owasp.org/cheatsheets/HTTP_Strict_Transport_Security_Cheat_Sheet.html

這個方法將不會建議於開發階段中來使用。

第 22 行的 `app.UseHttpsRedirection();` 敘述將會在 ASP.NET Core 中強制使用 HTTPS，也就是說會將所有 HTTP 要求都重新導向至 HTTPS，因此，在進行 HTTP 封包傳輸的時候，將會進行 SSL 的加密保護，如此可以避免使用 HTTP 來傳送任何明碼機密資料被第三方惡意人員從網路上監聽截取。

第 24 行的 `app.UseStaticFiles();` 敘述將會宣告可以存取 Web 根目錄相關公用靜資源檔案，例如樣式表單 (.css)、JavaScript (.js)、影像 (.png、.jpg) 等這些資源，而在開發專案中，[wwwroot] 這個資料夾將會代表為 Web 根目錄。

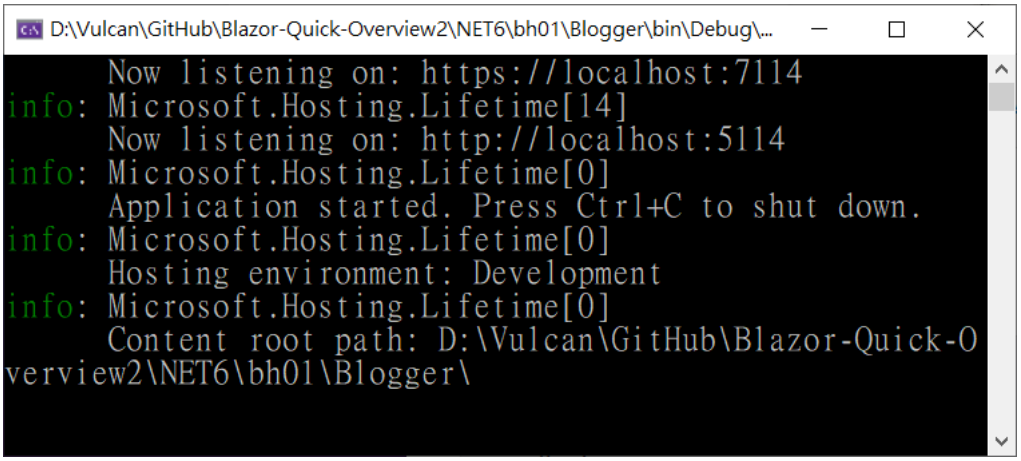
第 26 行的 `app.UseRouting();` 敘述表示將路由對應新增至中介軟體管線，當有新的服務端點請求產生的時候，將會透過這裡路由表格資訊決定要將這個服務端點請求送到哪個類別來處理。

第 26 行的 `app.MapBlazorHub();` 敘述將會把 SingalR 需要用到的 Hub 路徑新增到預設路由表中，並且啟動伺服器端的 SignalR 服務，如此，用戶端便可以與伺服器端透過 SignalR 服務進行雙向通訊，而 Blazor Server 開發框架中的資料綁定運作機制 (這包含了單向資料綁定、雙向資料綁定、事件綁定) 都會使用到 SignalR 服務，對於伺服器端需要使用者端的網頁內容需要要進行轉譯 Render 工作，更新瀏覽器上的 DOM 資訊，也會用到 SignalR 服務通知用戶端的瀏覽器來更新 DOM 到最新狀態。

第 27 行的 `app.MapFallbackToPage("/_Host");` 敘述將會宣告若所請求的服務端點，找不到合適的路由條件可以來執行，將會將此路由傳送至符合的 Razor 頁面端點，也就是在 [Pages] 資料夾下的 [_Host.cshtml] 檔案；一旦這個 Razor 頁面被執行之後，所產生的 HTML 內容回傳到用戶端的瀏覽器上，瀏覽器將會進行此次瀏覽器用戶端需要用到的相關 Blazor Server 功能的初始化與運行工作，故這個 Razor 頁面也可稱為 Blazor Server 用戶端的 Bootstrap 啟動載入程序。

第 31 行的 `app.Run();` 敘述將會開始啟動這個 Web 專案，並且開始監聽來自網路上的 HTTP 請求服務與進行 SignalR 雙向通訊工作，直到該專案被強制關閉；例如，當 Blazor

Server 啟動之後，將會出現如下圖的 Console 視窗，從這裡可以看到使用者可以在瀏覽器輸入 `https://localhost:7114` 或者 `http://localhost:5114` 服務端點，將會顯示這個 Blazor Server 首頁內容，而當在這個 Console 視窗下按下 [Ctrl] + [C] 將會終止這個 Web 應用程式執行。



Blazor 專案執行日誌輸出視窗

2.5.2 Blazor Server 啟動載入程序 _Host.cshtml 與 _Layout.cshtml

當在瀏覽器用戶端上輸入一個 Blazor Server 路由服務端點，例如 https://localhost:7114/ 或者 https://localhost:7114/counter 或者 https://localhost:7114/fetchdata ，將會透過 `app.MapFallbackToPage("/_Host");` 中介軟體導向到 [Pages] 資料夾下的 [_Host.cshtml] 這個 Razor 頁面上，這個 Razor 檢視頁面可以視為 Blazor UI 開發框架的 Bootstrap 啟動載入程序，而這個檔案內容如下

```
@page "/"
@namespace Blogger.Pages
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@{
    Layout = "_Layout";
}

<component type="typeof(App)" render-mode="ServerPrerendered" />
```

在這個 Razor 頁面內，宣告使用了 [_Layout.cshtml] 作為其產生 HTML 內容之版面配置設定，而在最後一行，則使用了 [component] 這個 HTML 標記項目 [Element] 用來顯示 Razor 元件 (也稱之為 Blazor 元件，不論是叫做哪個名稱，其副檔案名稱為 .razor)，並且將這個 [App.razor] 元件轉譯為靜態 HTML，並包含 Blazor 伺服器端應用程式的標記。底下將會轉譯出來的 HTML 文件內容

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <base href="/" />
    <link rel="stylesheet" href="css/bootstrap/bootstrap.min.css" />
    <link href="css/site.css" rel="stylesheet" />
    <link href="Blogger.styles.css" rel="stylesheet" />
    <!--Blazor:{"sequence":1,"type":"server","descriptor":"CfDJ8P\u002BU/GvukKFKm7iT\
BsxVwNv8Ckz/xGeomLUAtSidR53JBUrvocTXZQyM4mFTnjgAkqQi62oLGBxWUAai74RTL\u002BwdireZLhf\
tLi/K25yz6O/cVmUBiWMxsZwt6Ndy3//K3ZT5HTxfsWc0dn7CXD2/Mr1efCPwhX/OhswTkp6\u002Bxb1SJ3\
Rc4Th03z0vpDZtciQbw7rn/bWj9brFfgkDNRydlSnnfaMZ\u002BK2QzMD5IGp/ya2AYFNSAplxBizrSCNNG\
i1VIswOk52VdiQyfNeNslKcYv\u002B1Z6sSOU/sTUhU6AOQJMNzA7BK7M6NLFhm5NBZ3xpB4gURov1DErPy\
5b3w/EfPhCw9hM8bm6G77h/Fj6\u002BVlGda9vod3siE1LkC4lq6cn3i8qu/IJsxSRd3bjmHYpiFWX67V9D\
tgMhXQFUXSXgqtDqJ"}-->
</head>
<body>

<!--Blazor:{"sequence":0,"type":"server","prerenderId":"818a50c01e364713904f5253a350\
7f23","descriptor":"CfDJ8P\u002BU/GvukKFKm7iTBsxVwNuHZaqmfVL0ECVEaPHuT4AMUK6Mse3F9Zn\
6/\u002B89xhlCouDwjeshYsDgTo\u002BWezE7829Wkk/2qpoLJX8wRapmxojuTQuyAvse1josPLZek6pDn\
```

GkTy7Iq\u002BT8XxcxaVKgZx2WKfJFstalaoPZNMRqWk3C3TBhGvhGtLC6k81vhf3CvIypdx3nOvPYTGJVW\
84UQ14GVoOLdSsqBtrquXkU1LNOZqTFuqPeiBDaOgBzvP0mjQNyQq8R3lO\u002BoOC7gO3s6oUkdJsz0aiy\
QhX\u002BSJyIGWralIE8Lbe3BHRAR2DzV5Y7FXvZM\u002BjhauHM0HIWQt6inePQ="}-->

```
<div class="page" b-5q5yqf7nz1><div class="sidebar" b-5q5yqf7nz1><div class="top-row\
 ps-3 navbar navbar-dark" b-u0gkmwd8f6><div class="container-fluid" b-u0gkmwd8f6><a \
class="navbar-brand" href b-u0gkmwd8f6>Blogger</a>
      <button title="Navigation menu" class="navbar-toggler" b-u0gkmwd8f6><span cl\
ass="navbar-toggler-icon" b-u0gkmwd8f6></span></button></div></div>

<div class="collapse" b-u0gkmwd8f6><nav class="flex-column" b-u0gkmwd8f6><div class=\
"nav-item px-3" b-u0gkmwd8f6><a href="" class="nav-link active"><span class="oi oi-h\
ome" aria-hidden="true" b-u0gkmwd8f6></span> Home
          </a></div>
      <div class="nav-item px-3" b-u0gkmwd8f6><a href="counter" class="nav-link"><\
span class="oi oi-plus" aria-hidden="true" b-u0gkmwd8f6></span> Counter
          </a></div>
      <div class="nav-item px-3" b-u0gkmwd8f6><a href="fetchdata" class="nav-link"\
><span class="oi oi-list-rich" aria-hidden="true" b-u0gkmwd8f6></span> Fetch data
          </a></div></nav></div></div>

   <main b-5q5yqf7nz1><div class="top-row px-4" b-5q5yqf7nz1><a href="https://docs.\
microsoft.com/aspnet/" target="_blank" b-5q5yqf7nz1>About</a></div>

      <article class="content px-4" b-5q5yqf7nz1>

<h1>Hello, world!</h1>

Welcome to your new app.

<div class="alert alert-secondary mt-4"><span class="oi oi-pencil me-2" aria-hidden=\
"true"></span>
   <strong>How is Blazor working for you?</strong>

   <span class="text-nowrap">
      Please take our
      <a target="_blank" class="font-weight-bold link-dark" href="https://go.micro\
soft.com/fwlink/?linkid=2149017">brief survey</a></span>
   and tell us what you think.
</div></article></main></div>
      <!--Blazor:{"prerenderId":"818a50c01e364713904f5253a3507f23"}-->
```

```
    <div id="blazor-error-ui">

        An unhandled exception has occurred. See browser dev tools for details.

      <a href="" class="reload">Reload</a>
      <a class="dismiss">🗙</a>
    </div>

    <script src="_framework/blazor.server.js"></script>

<!-- Visual Studio Browser Link -->
<script type="text/javascript" src="https://localhost:44399/723cfb9340aa4d45bfa1527b\
98010c47/browserLink" async="async" id="__browserLink_initializationData" data-reque\
stId="94b4bdabc28446cd873a212b399225c4" data-requestMappingFromServer="False"></scri\
pt>
<!-- End Browser Link -->

<script src="/_framework/aspnetcore-browser-refresh.js"></script></body>
</html>
```

從上面產生出來的 HTML 文件內容可以看到有標示 `<!--Blazor: ... -->` 註解文字的地方，將會是 Blazor Server 產生出來並且使用在 Blazor 系統內，而在這個 `<div class="page" ... </div>` HTML 宣告標記，則是由 Blazor 元件所設計內容而自動產生出來的。

對於 [_Layout.cshtml] 檔案內容如下

```
@using Microsoft.AspNetCore.Components.Web
@namespace Blogger.Pages
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <base href="~/" />
    <link rel="stylesheet" href="css/bootstrap/bootstrap.min.css" />
    <link href="css/site.css" rel="stylesheet" />
    <link href="Blogger.styles.css" rel="stylesheet" />
    <component type="typeof(HeadOutlet)" render-mode="ServerPrerendered" />
</head>
<body>
    @RenderBody()

    <div id="blazor-error-ui">
        <environment include="Staging,Production">
            An error has occurred. This application may no longer respond until relo\
aded.
        </environment>
        <environment include="Development">
            An unhandled exception has occurred. See browser dev tools for details.
        </environment>
        <a href="" class="reload">Reload</a>
        <a class="dismiss">🗙</a>
    </div>

    <script src="_framework/blazor.server.js"></script>
</body>
</html>
```

在此文件中的 <head> 區段，將會預設載入 Bootstrap 5 CSS 樣式表單資源檔案，接著會載入這個專案自訂的 CSS 樣式表單 [site.css] 資源檔案，緊跟著會繼續載入這個 [Blogger.styles.css] 檔案，也就是 Blazor Server 專案內使用的 CSS 隔離組合[10] 功能資源檔案，其名稱的組合是採用這樣的設計：第一個 Blogger 名稱是此 .NET 組件名稱，預設

[10] https://docs.microsoft.com/zh-tw/aspnet/core/blazor/components/css-isolation?view=aspnetcore-6.0

為此專案名稱，底下將會是預設建立一個 Blazor Server 專案所產生的 CSS 隔離內容

```
/* _content/Blogger/Shared/MainLayout.razor.rz.scp.css */
.page[b-5q5yqf7nz1] {
    position: relative;
    display: flex;
    flex-direction: column;
}

main[b-5q5yqf7nz1] {
    flex: 1;
}

.sidebar[b-5q5yqf7nz1] {
    background-image: linear-gradient(180deg, rgb(5, 39, 103) 0%, #3a0647 70%);
}

.top-row[b-5q5yqf7nz1] {
    background-color: #f7f7f7;
    border-bottom: 1px solid #d6d5d5;
    justify-content: flex-end;
    height: 3.5rem;
    display: flex;
    align-items: center;
}

    .top-row[b-5q5yqf7nz1]  a, .top-row .btn-link[b-5q5yqf7nz1] {
        white-space: nowrap;
        margin-left: 1.5rem;
    }

    .top-row a:first-child[b-5q5yqf7nz1] {
        overflow: hidden;
        text-overflow: ellipsis;
    }

@media (max-width: 640.98px) {
    .top-row:not(.auth)[b-5q5yqf7nz1] {
        display: none;
    }
```

```
    .top-row.auth[b-5q5yqf7nz1] {
        justify-content: space-between;
    }

    .top-row a[b-5q5yqf7nz1], .top-row .btn-link[b-5q5yqf7nz1] {
        margin-left: 0;
    }
}

@media (min-width: 641px) {
    .page[b-5q5yqf7nz1] {
        flex-direction: row;
    }

    .sidebar[b-5q5yqf7nz1] {
        width: 250px;
        height: 100vh;
        position: sticky;
        top: 0;
    }

    .top-row[b-5q5yqf7nz1] {
        position: sticky;
        top: 0;
        z-index: 1;
    }

    .top-row[b-5q5yqf7nz1], article[b-5q5yqf7nz1] {
        padding-left: 2rem !important;
        padding-right: 1.5rem !important;
    }
}
/* _content/Blogger/Shared/NavMenu.razor.rz.scp.css */
.navbar-toggler[b-u0gkmwd8f6] {
    background-color: rgba(255, 255, 255, 0.1);
}

.top-row[b-u0gkmwd8f6] {
    height: 3.5rem;
    background-color: rgba(0,0,0,0.4);
}

.navbar-brand[b-u0gkmwd8f6] {
```

```
    font-size: 1.1rem;
}

.oi[b-u0gkmwd8f6] {
    width: 2rem;
    font-size: 1.1rem;
    vertical-align: text-top;
    top: -2px;
}

.nav-item[b-u0gkmwd8f6] {
    font-size: 0.9rem;
    padding-bottom: 0.5rem;
}

    .nav-item:first-of-type[b-u0gkmwd8f6] {
        padding-top: 1rem;
    }

    .nav-item:last-of-type[b-u0gkmwd8f6] {
        padding-bottom: 1rem;
    }

    .nav-item[b-u0gkmwd8f6]  a {
        color: #d7d7d7;
        border-radius: 4px;
        height: 3rem;
        display: flex;
        align-items: center;
        line-height: 3rem;
    }

.nav-item[b-u0gkmwd8f6]  a.active {
    background-color: rgba(255,255,255,0.25);
    color: white;
}

.nav-item[b-u0gkmwd8f6]  a:hover {
    background-color: rgba(255,255,255,0.1);
    color: white;
}

@media (min-width: 641px) {
```

```
    .navbar-toggler[b-u0gkmwd8f6] {
        display: none;
    }

    .collapse[b-u0gkmwd8f6] {
        /* Never collapse the sidebar for wide screens */
        display: block;
    }
}
```

下一個 `<component type="typeof(HeadOutlet)" render-mode="ServerPrerendered" />` 將會是宣告 Razor [HeadOutlet] 元件，其會呈現元件所提供 HeadContent 內容，這也就是說，在任何具有路由 Razor 元件中，使用了類似 `<PageTitle>Counter</PageTitle>` 這樣的用法，將會變更該瀏覽器的標題名稱，這樣的功能是在 .NET 6.0 平台下才有提供的。

在 `<body>` 區段，將會使用指示詞 [@RenderBody()] 用來顯示該 Razor 檢視的內容，也就是在 [_Host.cshtml] 文件中宣告的 `<component type="typeof(App)" render-mode="ServerPrerendered" />` 標記，也就是說，這裡將會要轉譯並顯示這個 Blazor 元件 (App.razor) 的內容，該元件位於 Blazor Server 專案的根目錄。

在其後的標記 `<div id="blazor-error-ui">...</div>` 將會宣告 Blazor 元件產生例外異常的時候，會將此處的內容顯示在螢幕上，讓使用者知道此 Web 應用程式拋出例外異常了。

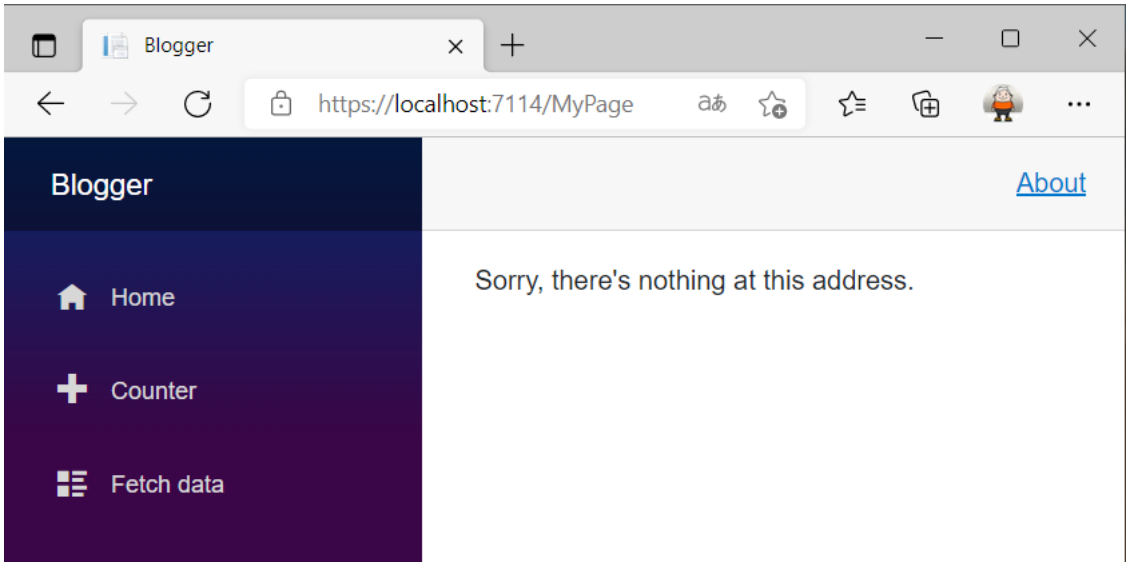
最後則是 Blazor Server 最重要的地方，那就是 `<script src="_framework/blazor.server.js"></script>` 用戶端必定要用到的 Blazor Server JavaScript 腳本，這個腳本會指向所建立的遠端 Balzor Server 透過 [MapBlazorHub] 所建立的 SignalR 端點，另外這個腳本也會提供存取網頁上 DOM 資訊的功能。一旦 SignalR 管道建立之後，這個瀏覽器的任何使用者互動所觸發 DOM 事件，將會透過 SignalR 通知遠端的 Blazor Server 主機，而在遠端的 Blazor Server 主機上產生了任何變更，也會透過 Blazor 轉譯機制，產生 DOM 的差異資訊，並通知瀏覽器來進行更新瀏覽器上 DOM 物件內容。

2.5.3 Blazor Server 框架進入點元件 App.razor

一旦 [_Host.cshtml] 載入啟動程式成功產生 HTML 文件並且載入到用戶端瀏覽器上，而且 Blazor Server 使用的 JavaScript 腳本也執行成功後，此時，就可以讓 Blazor 元件 Component 正常運作了，首先，第一個要轉譯 Render 的 Blazor 元件就是 [App.razor]，這個元件也稱之為 Blazor UI 框架下的進入點元件，底下是該元件的標記。

```
<Router AppAssembly="@typeof(App).Assembly">
    <Found Context="routeData">
        <RouteView RouteData="@routeData" DefaultLayout="@typeof(MainLayout)" />
        <FocusOnNavigate RouteData="@routeData" Selector="h1" />
    </Found>
    <NotFound>
        <PageTitle>Not found</PageTitle>
        <LayoutView Layout="@typeof(MainLayout)">
            <p role="alert">Sorry, there's nothing at this address.</p>
        </LayoutView>
    </NotFound>
</Router>
```

這個 Blazor 元件 [Router] ，將會根據要路由端點來決定要顯示什麼內容，若這個路由端點有定義在 Blazor 系統內 (只要在 Blazor 元件內有使用 [@page] 指示詞，就會有一筆路由資訊存在)，將會顯示 [Found] 這個元件，該元件內會透過 [RouteView] 這個 Blazor 元件來顯示指定的 Blazor 元件，並且該要顯示的 Blazor 元件將會套用 Blazor 系統內的 [MainLayout] 版面配置元件 (該元件位於 [Shared] 資料夾下)；然而，若指定的路由端點不存在指定的 Blazor 頁面元件，將會顯示 [NotFound] 這個 Blazor 元件，此時，螢幕上將會出現 [Sorry, there’s nothing at this address.] 文字訊息在網頁上。



指定的路由端點不存在

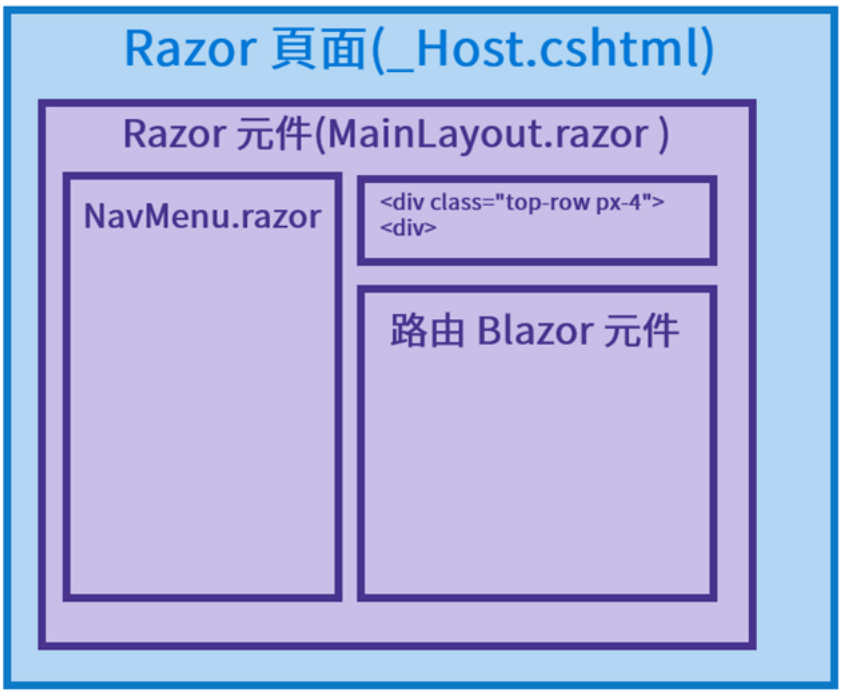
2.5.4 Blazor 框架預設版面配置元件 MainLayout.razor

每個可以路由 Blazor 元件，預設將會套用這個 [MainLayout.razor] 版面配置元件，打開 [Shared] 資料夾下的 [MainLayout.razor] 檔案，將會看到其檔案內容

```
@inherits LayoutComponentBase

<PageTitle>Blogger</PageTitle>

<div class="page">
  <div class="sidebar">
    <NavMenu />
  </div>

  <main>
    <div class="top-row px-4">
      <a href="https://docs.microsoft.com/aspnet/" target="_blank">About</a>
    </div>
```

```
      <article class="content px-4">
        @Body
      </article>
    </main>
</div>
```

在這個 [MainLayout.razor] 元件中，會在網頁的右方顯示這個 [NavMenu.razor] 系統的功能表清單 Razor 元件，在最上方將會顯示這個 `<a href="https://docs.microsoft.com/aspnet/" target="_blank">About</a>` HTML 標記，剩下的空間則會透過 `@Body` 來顯示所指定路由 Razor 元件內容。



MainLayout 元件形成的版面配置樣貌示意圖

2.5.5 功能選項切換導航面板元件 NavMenu.razor

接下來了解預設的功能表清單的設計方式，請打開 [Shared] 資料夾下的 [NavMenu.razor] 檔案，將會底下內容

```
<div class="top-row ps-3 navbar navbar-dark">
    <div class="container-fluid">
        <a class="navbar-brand" href="">Blogger</a>
        <button title="Navigation menu" class="navbar-toggler" @onclick="ToggleNavMe\
nu">
            <span class="navbar-toggler-icon"></span>
        </button>
    </div>
</div>

<div class="@NavMenuCssClass" @onclick="ToggleNavMenu">
    <nav class="flex-column">
        <div class="nav-item px-3">
            <NavLink class="nav-link" href="" Match="NavLinkMatch.All">
                <span class="oi oi-home" aria-hidden="true"></span> Home
            </NavLink>
        </div>
        <div class="nav-item px-3">
            <NavLink class="nav-link" href="counter">
                <span class="oi oi-plus" aria-hidden="true"></span> Counter
            </NavLink>
        </div>
        <div class="nav-item px-3">
            <NavLink class="nav-link" href="fetchdata">
                <span class="oi oi-list-rich" aria-hidden="true"></span> Fetch data
            </NavLink>
        </div>
    </nav>
</div>

@code {
    private bool collapseNavMenu = true;

    private string? NavMenuCssClass => collapseNavMenu ? "collapse" : null;
```

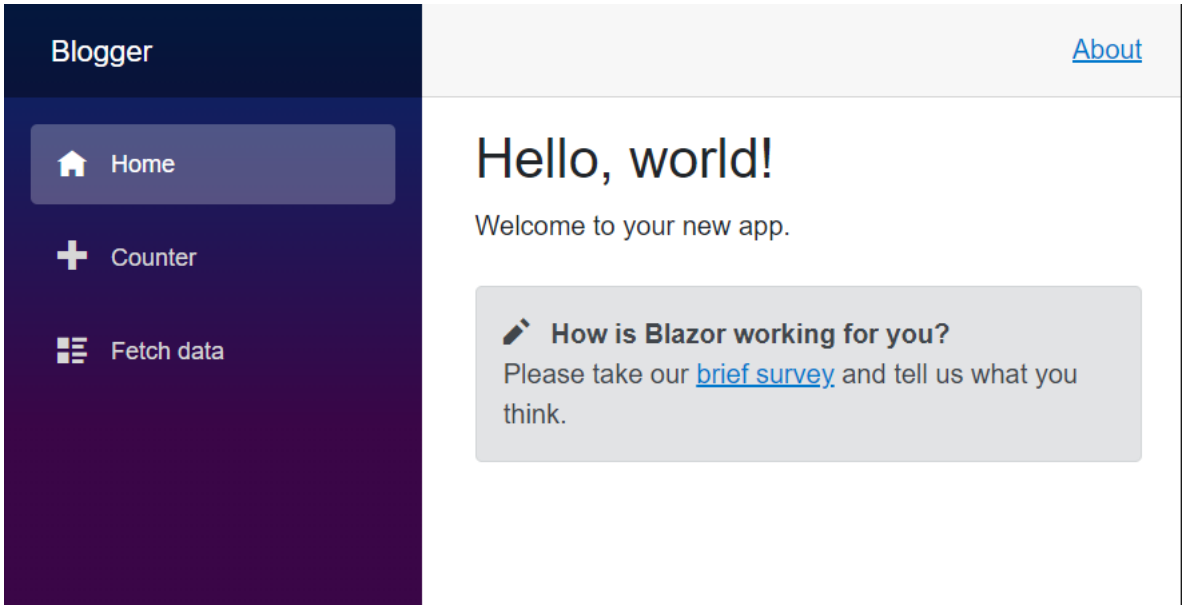
```
    private void ToggleNavMenu()
    {
        collapseNavMenu = !collapseNavMenu;
    }
}
```

對於每個功能表項目，將會透過 `<div class="nav-item px-3"> NavLink class="nav-link" href="counter"> <span class="oi oi-plus" aria-hidden="true"></span> Counter </NavLink> </div>` 宣告標記來運作，這裡使用 Blazor 內建的 [NavLink] 元件來做到切換頁面的功能，其中 [href] 元件參數將會用來指定當使用者點選這文字之後，要切換到哪個 Blazor 頁面元件，這裡切換頁面工作將會透過 SignalR 通道從瀏覽器用戶端傳送訊息到遠端 Blazor Server 上。

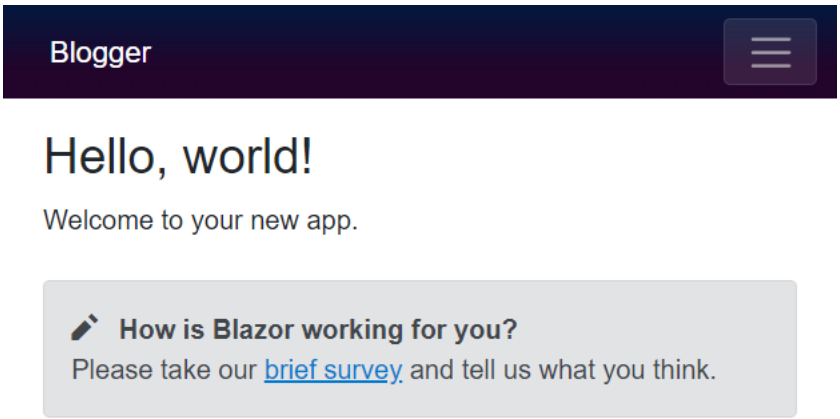
在 Blazor Server 內將會顯示指定的路由頁面元件，並且根據與現在顯示在網頁上的 DOM 資訊進行比較，找出差異 DOM 部分，接著將這些差異 DOM 資訊再度透過 SignalR 送至瀏覽器用戶端，此時，瀏覽器將會更新這些有差異的 DOM 物件，如此，就達到了更新網頁內容目的；特別要注意的是，剛剛所提到的過程，並不會透過 HTTP Request 請求來取得一個全新的 HTML 文件內容，接著由瀏覽器轉譯出這個網頁內容。

在該檔案的最下方將會有 [@code{...}] 指示詞，在這個區段內，可以為這個 Blazor 元件加入需要用到的商業邏輯程式碼，可喜可賀的是，在這裡若要設計相關商業邏輯需求，將不需要用到任何 JavaScript 程式碼，可以直接使用 C# 程式語言來進行設計，這對於身為 .NET C# 的開發者而言，真是太方便、太幸福了。

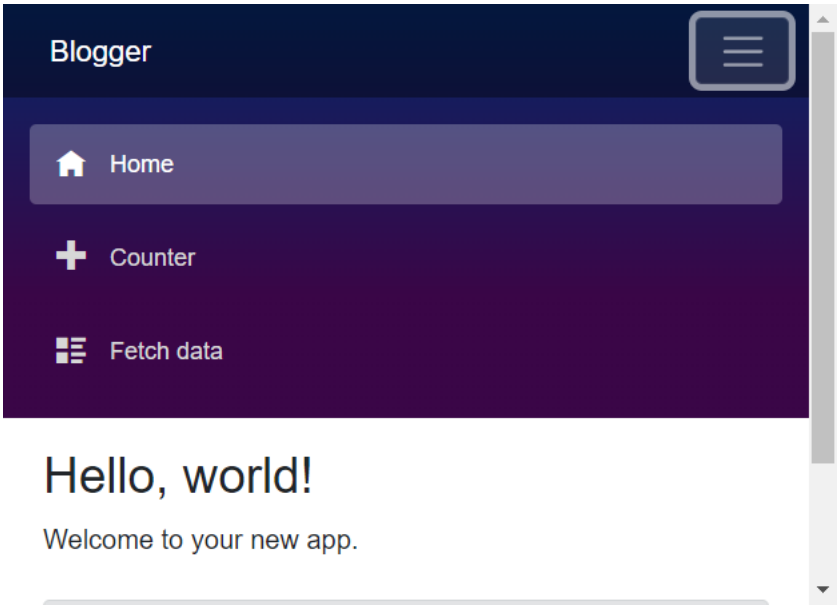
對於 Blazor 預設的專案將會提供網頁與手機模式的 RWD 樣式切換，一旦在手機上顯示這個網頁，該功能表清單將會需要透過左上方的漢堡按鈕來顯示出來。



在平板或者桌機下顯示的樣式



在行動裝置下顯示的樣式



在行動裝置下且點選漢堡按鈕顯示樣式

也就是說，當在手機模式下，使用者點選這個漢堡按鈕，將會觸發 [ToggleNavMenu()] 這個方法，這個方法將會變更其 [collapseNavMenu] 布林型別變數，true 變成 false，false 變成 true，接著透過資料綁定的機制，將 `private string? NavMenuCssClass => collapseNavMenu ? "collapse" : null;` 此屬性重新產生出最新的 [NavMenuCssClass] 屬性文字內容，而該 [NavMenuCssClass] 屬性值將會綁定到這個 `<div class="@NavMenuCssClass" @onclick="ToggleNavMenu">` HTML 標記上；一旦資料綁定機制運作完成之後，就可以動態的顯示或隱藏此 div 區段，該區段將會為功能表清單內容。

2.5.6 依據路由資訊來顯示特定 Blazor 頁面元件

若沒有指定任何路由參數，預設將會顯示 [Pages] 資料夾下的 [Index.razor] 頁面元件，其內容如下

```
@page "/"

<PageTitle>Index</PageTitle>

<h1>Hello, world!</h1>

Welcome to your new app.

<SurveyPrompt Title="How is Blazor working for you?" />
```

這個元件使用了 Blazor 內建的 [PageTitle] 元件來更新該網頁的 Title 文字，如此，當切換到不同 Blazor 頁面元件的時候，將會在瀏覽器的標題上，看到此頁面元件的名稱。

在這個 Index 元件中，將會引用另外一個 [SurveyPrompt] 元件，這樣的設計方式為 Blazor UI 開發框架的主要設計精神，也就是說，對於採用 Blazor Server 做為網站服務的開發者而言，可以將一個頁面切割成為不同的 Razor 元件，接著在需要路由的 Razor 元件中將其組合起來，就可以形成一個新的複合元件內容，這樣的設計方式將大幅提升網頁的設計速度與品質，一旦需要替換或者更新某個功能或者畫面樣式的時候，也可以直接替換該 Blazor 元件就可以做到顯示出不同內容的目的。

從 [Shared] 資料夾內可以找到 [SurveyPrompt.razor] 元件檔案，底下將會是這個 Razor 元件的宣告標記文件內，該檔案內 C# 程式碼有建立一個 [Title] 屬性 Property，並且使用 `[Parameter]` 這個 C# 屬性 Attribute 來宣告這個 [Title] 屬性可以接收來自其他元件所傳送過來的參數；這樣的機制將會使得在 Blazor UI 開發框架下，具有不同的 Razor 元件可以自由傳遞參數物件能力。

```
<div class="alert alert-secondary mt-4">
    <span class="oi oi-pencil me-2" aria-hidden="true"></span>
    <strong>@Title</strong>

    <span class="text-nowrap">
        Please take our
        <a target="_blank" class="font-weight-bold" href="https://go.microsoft.com/f\
wlink/?linkid=2149017">brief survey</a>
    </span>
    and tell us what you think.
</div>

@code {
    // Demonstrates how a parent component can supply parameters
    [Parameter]
    public string? Title { get; set; }
}
```

2.5.7 方便擴充的範本檔案 _Imports.razor

在專案根目錄下的這個 [_Imports.razor] 檔案，可以在這裡加入一些指示詞，例如 @using 命名空間的指示詞，如此，在底下資料夾的每個 Blazor 元件內將會自動加入可以參考這些 @using 指示詞所指定的命名空間內的類別，底下將會是這個檔案內容，因此，有了這項功能，當在 Blazor 元件內要引用不同命名空間下的類別或者存取其 API 或者屬性，便無需在此 Blazor 元件內手動加入 [@using] 指示詞了。

```
@using System.Net.Http
@using Microsoft.AspNetCore.Authorization
@using Microsoft.AspNetCore.Components.Authorization
@using Microsoft.AspNetCore.Components.Forms
@using Microsoft.AspNetCore.Components.Routing
@using Microsoft.AspNetCore.Components.Web
@using Microsoft.AspNetCore.Components.Web.Virtualization
@using Microsoft.JSInterop
@using Blogger
@using Blogger.Shared
```

2.5.8 ASP.NET Core 設定 appsettings.json

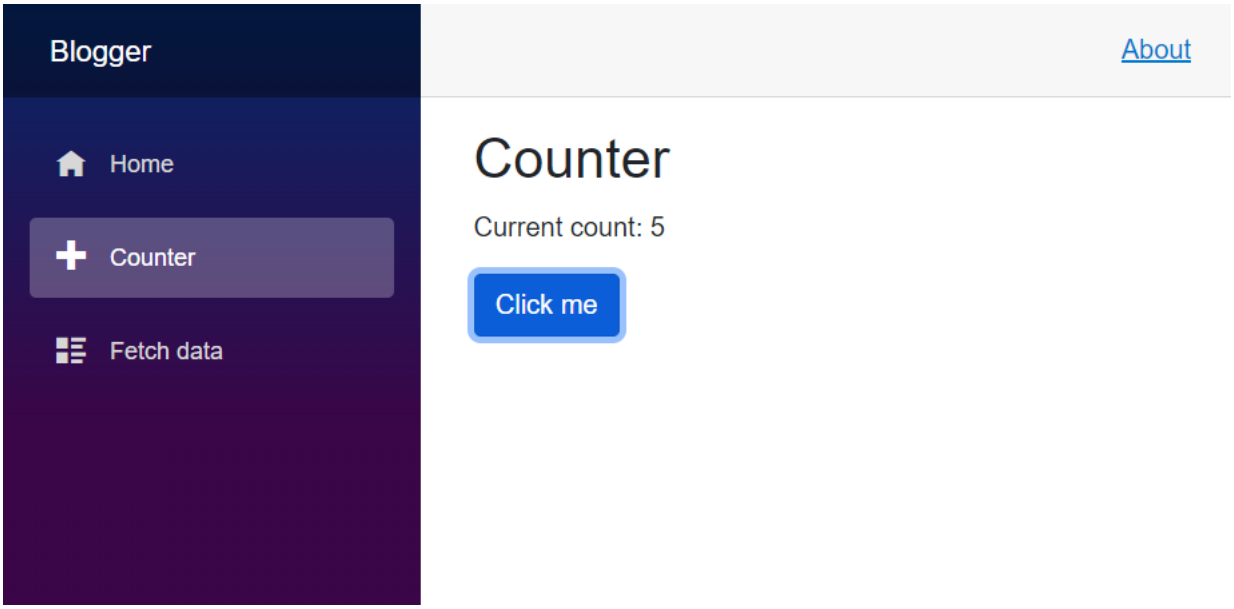
這個 [appsettings.json] 檔案將會提供應用程式佈設定內容，例如，可以在這裡宣告連線到資料庫的連線字串內容等等。

2.5.9 Counter.razor 元件

在這個預設 Blazor Server 專案範本中，將會產生兩個 Blazor 元件，這兩個元件都是可以透過在瀏覽器位址列上輸入特定服務端點，以便顯示此指定可路由 Blazor 元件，在這裡將會來解說這兩個 Blazor 元件是如何設計出來的，首先是計數器元件，當在瀏覽器位址列輸入 https://localhost:7114/counter 這個 URL，將會顯示 Counter.razor 元件在網頁上，其元件設計標記與程式碼如下

```
@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++;
    }
}
```

因此，切換到 [Counter] 這個 Blazor Server 專案範本內預設計數器元件，將會顯示如下結果畫面



Blazor 專案範本內預設計數器元件顯示結果

在這個計數器元件元件內，宣告一個欄位變數 [currentCount]，儲存了 [Click me] 按鈕被按下了幾次，在 `<button>` 這個標記中，使用了 `@onclick` 指示詞宣告了當使用者點選了這個按鈕，這就是所謂的事件綁定的用法，將會觸發後面所宣告的 C# 事件委派方法 IncrementCount ，一旦觸發了事件委派方法，將會將 [currentCount] 欄位變數加一，而這個欄位變數又在上方 HTML 文件中使用了單向資料綁定的方式來宣告，也就是 `<p role="status">Current count: @currentCount</p>` ，因此，一旦委派事件執行完畢之後，將會觸發整個網頁的轉譯工作，所以，網頁就會顯示出最新的 [currentCount] 欄位值在網頁上。

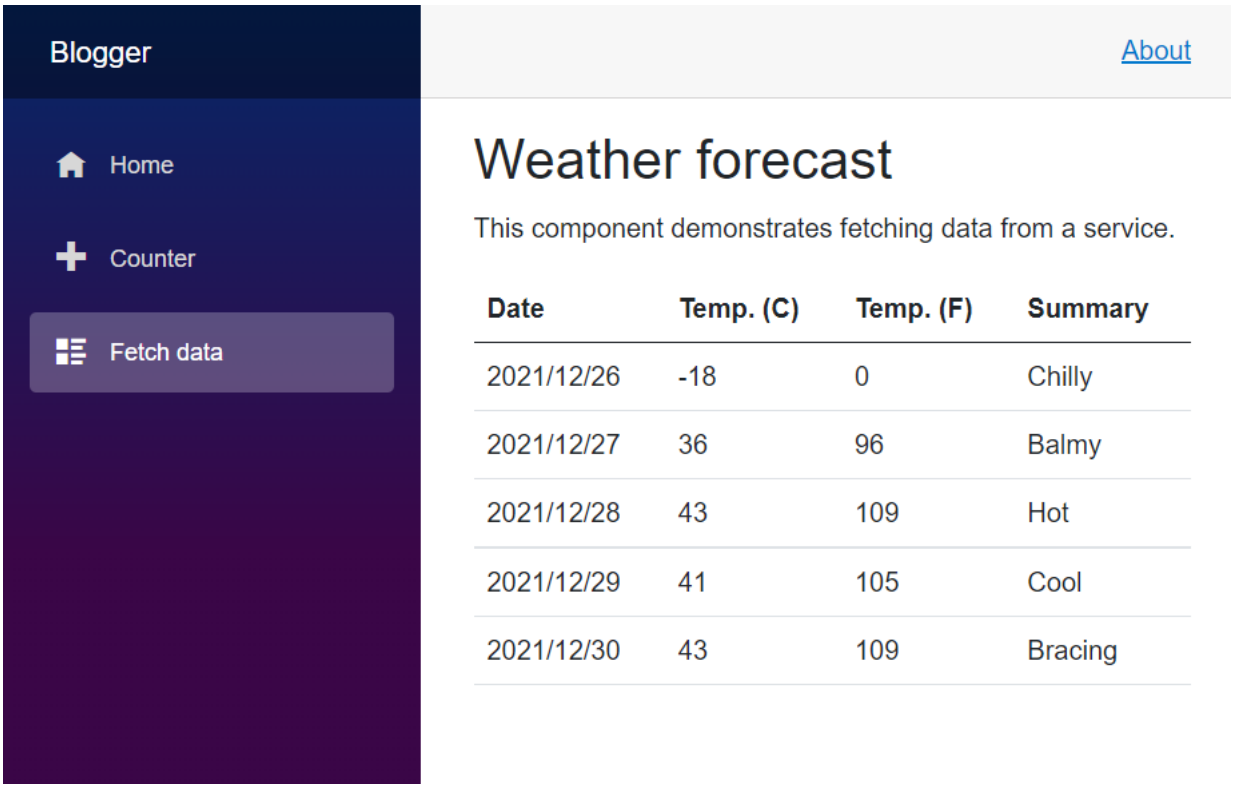
2.5.10 FetchData.razor 元件

接著是天氣預報元件，當在瀏覽器位址列輸入 https://localhost:7114/fetchdata 這個 URL，將會顯示 Counter.razor 元件在網頁上，其元件設計標記與程式碼如下

```
@page "/fetchdata"

<PageTitle>Weather forecast</PageTitle>

@using Blogger.Data
@inject WeatherForecastService ForecastService

<h1>Weather forecast</h1>

<p>This component demonstrates fetching data from a service.</p>

@if (forecasts == null)
{
    <p><em>Loading...</em></p>
}
else
{
    <table class="table">
        <thead>
            <tr>
                <th>Date</th>
                <th>Temp. (C)</th>
                <th>Temp. (F)</th>
                <th>Summary</th>
            </tr>
        </thead>
        <tbody>
            @foreach (var forecast in forecasts)
            {
                <tr>
                    <td>@forecast.Date.ToShortDateString()</td>
                    <td>@forecast.TemperatureC</td>
                    <td>@forecast.TemperatureF</td>
                    <td>@forecast.Summary</td>
                </tr>
            }
        </tbody>
    </table>
}

@code {
    private WeatherForecast[]? forecasts;
```

```
    protected override async Task OnInitializedAsync()
    {
        forecasts = await ForecastService.GetForecastAsync(DateTime.Now);
    }
}
```

因此，切換到 [Fetch data] 這個 Blazor Server 專案範本內預設天氣預報元件，將會顯示如下結果畫面



Blazor 專案範本內預設天氣預報元件顯示結果

在天氣預報元件內宣告了一個欄位 [forecasts] 集合物件，該變數將會儲存了最近五天的天氣預報資訊，在 `@code{}` 程式碼設計區段內，使用了這個 [OnInitializedAsync()] 元件生命週期事件，該事件會在這個天氣預報元件產生的時候，就會被觸發，一旦被觸發之後，將會透過相依性注入服務來注入一個 WeatherForecastService 型別物件 (這裡使用

了 `@inject WeatherForecastService ForecastService` 指示詞來宣告這個元件要注入一個型別為 WeatherForecastService 的物件到 ForecastService 變數內)。

一旦取得這個天氣預報服務的物件，便呼叫了 `await ForecastService.GetForecastAsync(DateTime.No` 方法來取得最近 5 天的天氣預報資訊；而在上方的 HTML 標記中，可以找到 `@foreach (var forecast in forecasts)` 這個指示詞宣告，當該網頁要進行轉譯工作的時候，將會把 forecasts 集合物件內的每一個物件，逐一顯示在網頁上，這也就是該天氣預報元件顯示在網頁上的最後結果。

3. 版權頁

.NET 6 Blazor 快速體驗 Hands-On Lab 動手練習

檔案格式：EPUB3、PDF、MOBI

版本： 1.0
日期： 2022.01

作者： Vulcan Lee 李進興