ASP.NET Core 3.1 框架與基礎服務

CHAPTER 4

ASP.NET Core 是一個跨平台、高效能、開源框架，用來建立現代化、雲端基礎、Internet 連結（指 IoT）的應用程式。然而在背後支撐整個框架運作的是眾多基礎服務，包括了 Hosting、Configuration 組態系統、相依性注入、Middleware、Routing、Environment、Logging 等服務，因此了解每個服務功用，服務如何調整與設定，便是本章要談的內容。

4-1 ASP.NET Core 框架簡介

以下從三個面向介紹 ASP.NET Core 框架，闡述其架構設計之改變，對開發與執行帶來的正面影響，以及最終產生的好處與優勢。

❖ 使用 ASP.NET Core 之利益

ASP.NET Core 是 ASP.NET 4.x 的重新設計，架構上變得更精簡與模組化，提供以下好處：

- 能在 Windows、macOS 和 Linux 上開發、建置與執行
- 整合現代化、client-side 框架與開發流程

- 用統一劇本建立 Web UI 和 Web APIs
- 一個雲端就緒、環境為基底的組態系統
- 內建 Dependency Injection 相依性注入
- 一個輕量級、高效能和模組化 HTTP Request Pipeline
- 新增 Razor Page 和 Blazor 專案開發模式
- 支援使用 gRPC 託管遠端程式呼叫（RPC）服務
- 能夠裝載到 Kestrel、IIS、HTTP.sys、Nginx、Apache、Docker，或在你自己的程序中自我裝載（self-host）
- 支援.NET Core Runtime 多版本並行（Side-by-side versioning）
- 可簡化現代化 Web 開發的工具
- Open-source 開源和以社群為中心

❖ 使用 ASP.NET Core 建立 Web UI 和 Web APIs

ASP.NET Core 在建立 Web UI 和 Web APIs 方面提供的功能有：

- MVC Pattern 能助你的 Web UI 和 Web APIs 更具有可測試性
- Razor Pages 是以 Page 為基礎的程式模型，使得建立 Web UI 更容易和更具生產性
- Razor Markup 為 Razor Pages 和 MVC Views 提供了更具生產力語法
- Tag Helpers 能夠讓 Server 端程式參與 Razor 檔中 HTML elements 元素的建立與轉譯（Rendering）
- 內建多種資料格式和內容協商，使你的 Web APIs 可以覆蓋廣泛的用戶端，包括瀏覽器和行動裝置
- Model Binding 自動將 HTTP Requests 對應到 Action 方法的參數
- Model Validation 自動執行 Client 端與 Server 端的驗證

❖ **Client 端開發（指 Front-End 前端開發）**

目前流行的前端框架像 Bootstrap、Angular、React，在.NET Core CLI 或 Visual Studio 專案樣板中都有內建支援，使得在開發這類前端程式時，可以得到很好的支援。到了 ASP.NET Core 3.0 時還新增 Blazor 框架支援，它是用 C#撰寫 Web UI 前端互動程式的一種新專案。

由以上幾個面向可體認到，.NET Core 的開放性、跨平台能力、高效能、前後端解決方案豐性，都是大大超越前一代。

4-2 ASP.NET Core Fundamentals 基礎服務概觀

若要理解 ASP.NET Core 框架全貌，從它的基礎服務與機制探索起，了解它提供哪些功能，這些服務又是如何交織運作，便能概要掌握其大體技術光譜，下面是 ASP.NET Core 框架的基礎服務大分類圖。

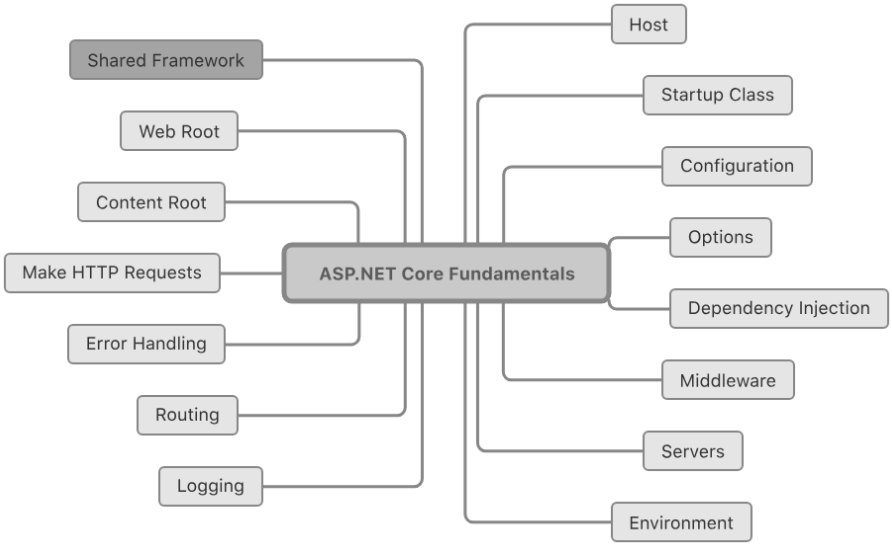


圖 4-1 ASP.NET Core Fundamental 基礎服務大分類

這些服務支撐起整個 ASP.NET Core 應用程式的運行，而服務之間也彼此協同與連動，下面說明每個服務概要功能：

- Host：裝載與執行.NET Core 應用程式的主機環境，它封裝了所有 App 資源，如 Server、Middleware、DI 和 Configuration，並實作 IHostedService
- Server：指 HTTP Server 或 Web Server 伺服器，用於監聽 HTTP 請求與回應的網頁伺服器
- Dependency Injection：相依性注入，亦稱 DI Container
- Middleware：在處理 HTTP 請求的管線中，包含一系列 Middleware 中介軟體元件
- Configuration 組態：ASP.NET Core 的組態框架，提供 Host 和 App 所需的組態存取系統
- Startup 類別：負責 Service 相依性注入和 Middleware 設定
- Options：是指 Options Pattern 選項模式，用類別來表示一組設定，.NET Core 中大量使用選項模式設定組態
- Environment：環境變數與機制，內建 Development、Staging 與 Production 三種環境
- Logging：資訊或事件的記錄機制
- Routing：自 ASP.NET 3.0 開始採用端點路由，它負責匹配與派送 HTTP 請求到應用程式執行端點
- Error Handling：負責錯誤處理的機制
- Make HTTP Request：是 IHttpClientFactory 實作，用於建立 HttpClient 實例
- Content Root：內容根目錄，代表專案目前所在的基底路徑
- Web Root：Web 根目錄，專案對外公開靜態資產的目錄
- Shared Framework：共享的框架組件

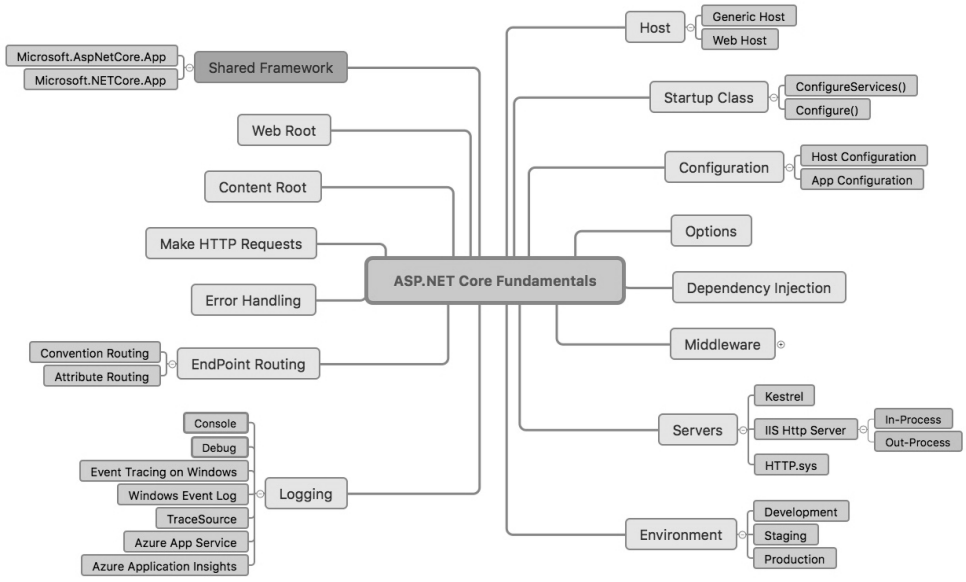


圖 4-2 ASP.NET Core Fundamental 基礎服務明細

以上 Fundamentals 服務如何影響 ASP.NET Core App？較為顯著的有：

- 掌控 ASP.NET Core App 系統運作
- 提供 Hosting 和 Web Server 組態設定
- 提供各種環境變數與組態值設定
- 提供多重環境組態設定：Development、Staging 和 Production
- 提供 DI 及 Middleware 設定
- 提供效能調校、Logging 等一堆功能

是故，開發人員若想全面掌握 ASP.NET Core，必需熟悉這些基礎服務知識與技巧，方能輕鬆駕馭。

4-3 重要基礎服務簡介

本節針對最為重要的基礎服務做介紹，讓您了解每個服務負責什麼功能，以及如何叫用這些服務。

4-3-1 ASP.NET Core 應用程式載入過程

下圖是 ASP.NET Core 應用程式以 dotnet run 執行，相關檔案載入過程，它有六個重要步驟，可與 ASP.NET Core 專案程式相對應，並以可目視及驗證的角度來論述，至於框架背景或底層不可視的執行過程，就暫不討論。

圖 4-3 ASP.NET Core App 主要執行與載入過程

六大過程說明：

1. dotnet run 是.NET Core CLI 命令工具，用來執行.NET Core 應用程式
2. dotnet run 執行時，會先載入 launchSettings.json 組態，此組態是供本機電腦環境使用，例如 IIS Express 或 Console App

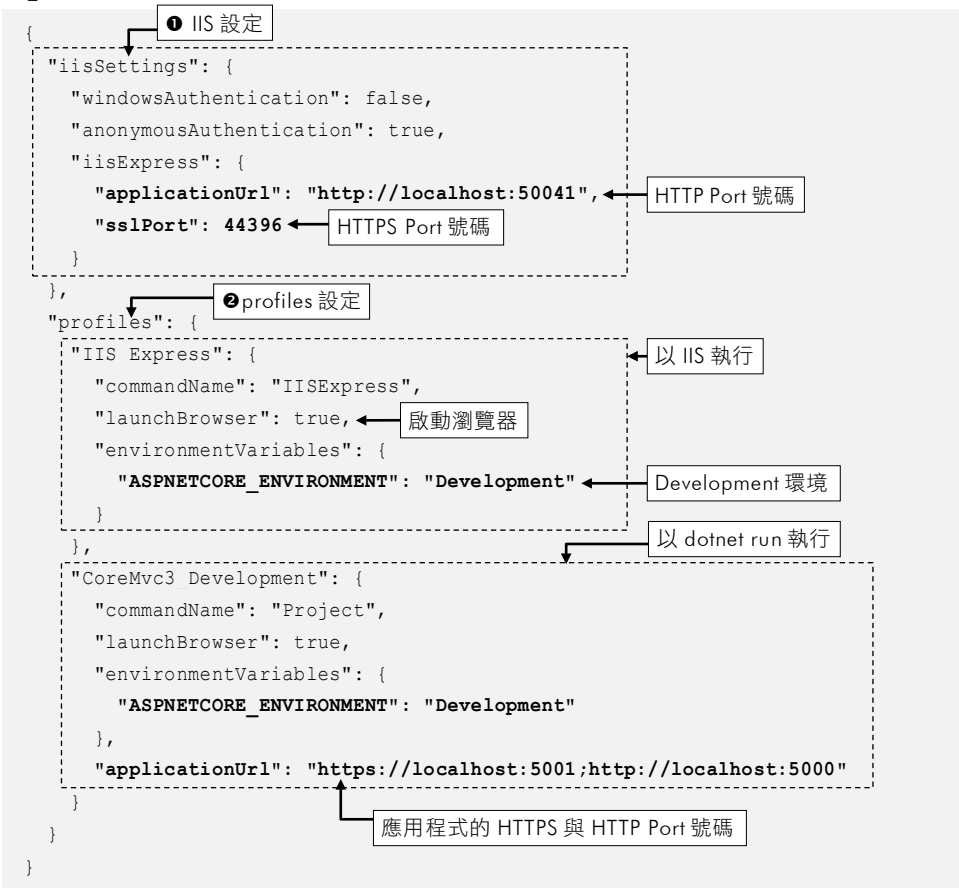
3. 再來執行 Program.cs，其 Main()是程式進入點，作用是建立 Generic Host，過程中還會載入 appsettings.json 與其他服務
4. 載入 appsettings.json 應用程式組態，預設有 Logging 和 AllowedHosts 組態，像若用到 EF Core 存取資料庫，其連線設定也是存放在此
5. 執行 Startup.cs，裡面包含 ConfigureServices 和 Configure 兩個重要方法：
   (1) 首先執行 ConfigureServices 方法，它是 DI Container（Service Container）相依性注入容器/Options Pattern 註冊的地方
   (2) 其次執行 Configure 方法，設定要載入哪些 Middleware 中介軟體
6. 在載入所有的組態和軟體服務後，呼叫 Build()和 Run()方法，建立 Host 主機，然後 Kestrel 就開始傾聽 HTTP 請求，並回應結果

那為何要解釋這六大執行過程，主要是讓你串起整個基礎框架服務的執行順序，理解它們是在什麼階段被載入執行，又扮演何種角色，以及彼此的關聯性。後續在說明個別服務功能時，才不會覺得是一群零散、各自為政的服務，同時在撰寫程式時，能更清楚什麼功能要在哪調整。

4-3-2 本機開發電腦環境組態檔 - launchSettings.json

當建立 ASP.NET Core 專案時，預設會有 launchSettings.json 和 appsettings.json 兩個組態檔，launchSettings.json 是本機開發電腦的環境組態檔，裡面分兩大類、三個區塊，第一類是 IIS 設定，第二類是 Profiles 設定。

Properties/launchSettings.json



此組態檔會決定專案執行與行為，例如用 IIS Express 或 Kestrel 網頁伺服器執行，是否要啟動瀏覽器、環境變數或應用程式監聽的 URL 網址，深入部分，在第 14 章組態檔會解釋其行為與作用。

TIP

launchSettings.json 僅供本機電腦使用，而不用於部署。

4-3-3 Program.cs - Main()建立 Host 主機

Program 的 Main()主要任務是建立 Host 主機：

Program.cs

```
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>();
            });
}
```

建立 Host 主機

使用 Startup 類別檔

以上所做工作有：

1. Main()方法調用 Host.CreateDefaultBuilder()，遵循 Builder Pattern 建立一個 Generic Host
2. Builder 有 Methods 方法定義 Web Server，例如 UseKestrel()方法，以及 UseStartup 方法指定 Startup 類別
3. 若有 IIS 的話，ASP.NET Core Web Host 會企圖在 IIS 上執行
4. 其他網頁伺服器 (例如 HTTP.sys) 則可透過叫用適當的擴充方法來使用
5. Build()和 Run()方法會建立 IWebHost 物件，它會裝載應用程式並監聽 HTTP Request 請求

前面曾提及 Host 是：”裝載與執行.NET Core 應用程式的主機環境，它封裝了所有 App 資源，如 Server、Middleware、DI 和 Configuration，並實作 IHostedService”

這樣的解釋對許多人來說可能還是很模糊，在此稍加說明，Host 是一個物件，封裝了前述種種相互依賴的服務，其目的只有一個，便是「生命週期管理」控制 App 應用程式啟動，以及優雅順利的關閉 Host 主機。

而 Program 中最重要的是 CreateDefaultBuilder()和 ConfigureWebHostDefaults()方法，它們執行了大量工作：

1. CreateDefaultBuilder()方法

本方法執行的工作有：

- 將 GetCurrentDirectory()方法回傳路徑設定為 Content Root
- 從❶DOTNET_"開頭的環境變數和❷Command-line 參數載入 Host 組態
- 從以下幾種來源載入 App 組態：
  - appsettings.json
  - appsettings.{Environment}.json
  - 應用程式在開發環境執行時的 Secret 管理員
  - 環境變數
  - Command-line 參數
- 載入以下 Logging 記錄提供者：
  - Console
  - Debug

- EventSource
- EventLog（限在 Windows 執行）

- 開發環境時，會啟用範圍驗證和相依性驗證

2. ConfigureWebHostDefaults()方法

本方法執行的工作有：

- 初始 WebHostBuilder 類別，在該類別建構函式中會載入 "ASPNETCORE_" 開頭的環境變數
- 使用 Kestrel 作為預設 Web Server，並使用 Host 組態提供者設定它
- 加入 Host 篩選 Middleware
- 加人 Routing 路由
- 啟用 IIS 整合

❖ 設定 Host 或 App 組態

在建立 Host 過程中，也可載入設定 Host 或 App 組態：

```
public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        //.UseContentRoot(Directory.GetCurrentDirectory())
        .ConfigureHostConfiguration(configHost =>
        {                                    設定 Host 組態
            //設定基底路徑
            configHost.SetBasePath(Directory.GetCurrentDirectory());

            //如想看見 SetBasePath()設定後路徑,可用下面程式抓取
            string contentRoot =
               ((Microsoft.Extensions.FileProviders.PhysicalFileProvider)configHost.
               Properties.Values.FirstOrDefault()).Root;

            configHost.AddJsonFile("hostsettings.json", optional: true);
            configHost.AddEnvironmentVariables(prefix: "PREFIX_");
            configHost.AddCommandLine(args);
```

```
        })                                    設定 App 組態
        .ConfigureAppConfiguration((hostingContext, configApp) =>
        {
            string path = Path.Combine(Directory.GetCurrentDirectory(), "ConfigFiles");
            //載入自訂 JSON 組態檔
            configApp.AddJsonFile(Path.Combine(path, "FutureCorp.json"),
             optional: true, reloadOnChange: true);
            //載入自訂 INI 組態檔
            configApp.AddIniFile(Path.Combine(path, "Mobile.ini"), true, true);
            //載入自訂 XML 組態檔
            configApp.AddXmlFile(Path.Combine(path, "Computer.xml"), true, true);
            //載入自訂 JSON 組態檔
            configApp.AddJsonFile(Path.Combine(path, "Device.json"), true, true);

            path = Path.Combine(Directory.GetCurrentDirectory(), "Configuration");
            configApp.AddJsonFile(Path.Combine(path, "Food.json"), true, true);
            configApp.AddInMemoryCollection(dictEmployees);
        })
        .ConfigureWebHostDefaults(webBuilder =>
        {
            webBuilder.UseStartup<Startup>();
            webBuilder.UseWebRoot(Directory.GetCurrentDirectory() + "/publicshare/");
        });
```

4-3-4 Startup.cs - 相依性注入與 Middleware 中介元件

Startup 類別負責：❶Service 相依性註冊和❷Middleware 設定，分別對映至 ConfigurationServices()和 Configure()兩個方法，而建構函式中是組態相依性注入，一般是 ConfigurationServices()方法中會存取組態。

Startup.cs

```
public class Startup
{                                          注入 Configuration 組態相依性
    public Startup(IConfiguration configuration)
    {
        Configuration = configuration;
    }
    public IConfiguration Configuration { get; }
```

```
                                                    1. DI Container 相依性注入容器
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddControllersWithViews();
        services.AddDbContext<DatabaseContext>(options =>
          options.UseSqlServer(Configuration.GetConnectionString("DatabaseContext")));
    }
                                                    2. 使用 Middleware 元件
    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
        …略
    }
}
```

以下概要說明兩個方法作用：

1. ConfigurationServices 方法

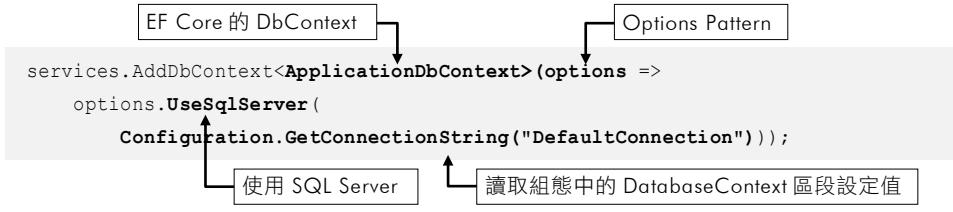
它是 DI 相依性注入容器（DI Container 或 Service Container），用來註冊服務介面與實作相依性的地方，ASP.NET Core 3.1 MVC 預設只有 AddControllersWithViews()一行程式（無 Identity 情況下 ）：

```
services.AddControllersWithViews();
```

但建立專案時，若選擇 Identity 驗證模式就會變成：

```
public void ConfigureServices(IServiceCollection services)
{                                                   EF Core
    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(
            Configuration.GetConnectionString("DefaultConnection")));
    services.AddDefaultIdentity<IdentityUser>(options =>
       options.SignIn.RequireConfirmedAccount = true)
        .AddEntityFrameworkStores<ApplicationDbContext>();
    services.AddControllersWithViews();             Identity 驗證
    services.AddRazorPages();
}
```

像 EF Core 區塊是為了註冊 DBContext 與 SQL Server 相關的設定：



同時，ConfigurationService 方法中也常調用 Configuration 或 Options Pattern 模式提供組態值。

2. Configure 方法

它是設定 HTTP 請求管線（Request Pipeline）使用哪些 Middleware 元件的地方，下面以 Use 開頭的方法就表示使用該 Middleware 元件。

```
                    端點路由
    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllerRoute(
            name: "default",
            pattern: "{controller=Home}/{action=Index}/{id?}");
        endpoints.MapRazorPages();
    });
}
```

❖ Middleware 元件運作方式與順序

HTTP 請求管線是由一系列 Middleware 元件組成，每個元件皆有不同的任務功用，並以非同步方式在 HttpContext 上作業，前一個元件會叫用下一個元件，或是終止請求的執行。

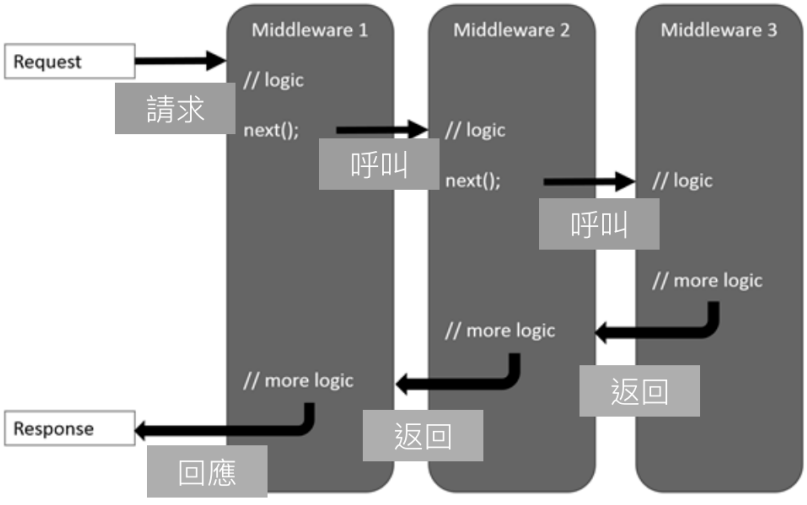


圖 4-4 HTTP 請求管線中 Middleware 元件運作流程

下圖是 ASP.NET Core 3.1 內建的 Middleware 元件。

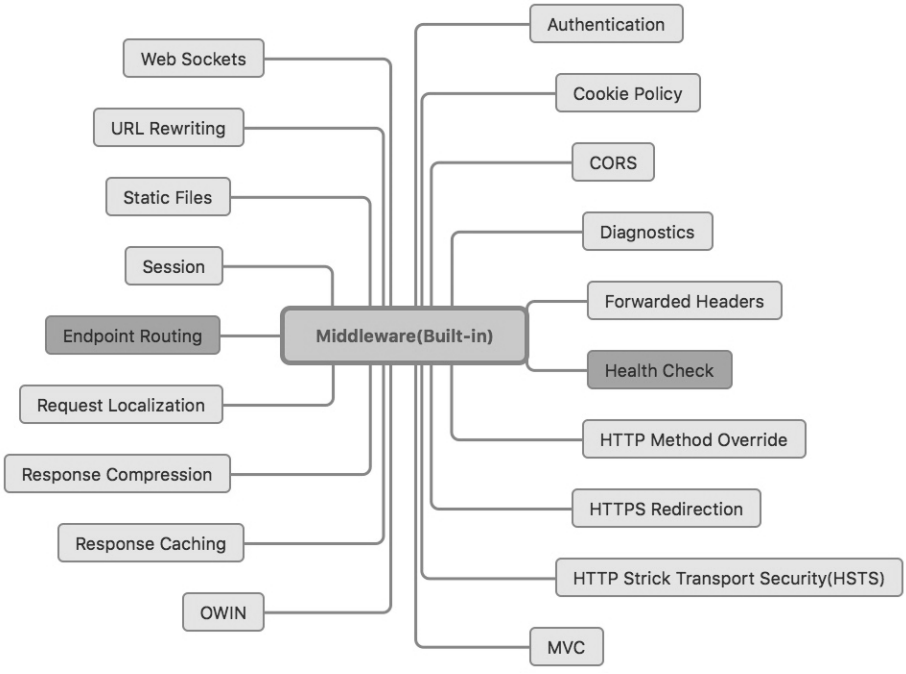


圖 4-5 ASP.NET Core 內建的 Middleware 元件

下表大略說明每個 Middleware 中介軟體元件之功用。

表 4-1 ASP.NET Core 3.1 內建的 Middleware 中介軟體元件

Middleware 元件	說明	順序
Authentication	提供驗證支援	在需要 HttpContext.User 之前。OAuth 回呼的終端機
Cookie Policy	追蹤使用者對用於儲存個人資訊的同意，並強制執行 Cookie 欄位的最低標準，例如 secure 和 SameSite	在發出 Cookie 的中介軟體之前。例如：驗證、工作階段、MVC (TempData)
CORS	設定跨原始來源（Cross-Origin）資源共用	在使用 CORS 的元件之前
Diagnostics	提供開發人員例外狀況頁面、例外狀況處理、狀態字碼頁，以及新應用程式的預設網頁的數個個別中介軟體	在產生錯誤的元件之前。 終端機的例外狀況，或為新的應用程式提供預設的網頁

Middleware 元件	說明	順序
Forwarded Headers	將設為 Proxy 的標頭轉送到目前請求	在使用更新欄位元件前。例如：scheme、host、client IP
Health Check	檢查 ASP.NET Core 應用程式及其相依性的健康狀態，例如檢查資料庫可用性	若某項要求與健康狀態檢查端點相符，則產生中止
HTTP Method Override	允許傳入的 POST 請求覆寫方法	在使用更新方法的元件之前
HTTPS Redirection	將所有 HTTP 請求都重新導向至 HTTPS	在使用 URL 的元件之前
HTTP Strict Transport Security (HSTS)	新增特殊的回應標頭的增強安全性中介軟體	在傳送回應前和修改請求的元件後。例如：轉送的標頭、URL Rewriting
MVC	使用 MVC/Razor Pages 處理請求	若請求符合路由則產生終止
OWIN	與基於 OWIN 的應用程式、服務器和 Middleware 的交互操作	若 OWIN 中介軟體完全處理請求則終止
Response Caching	提供快取回應的支援	在需要快取的元件之前
Response Compression	提供壓縮回應的支援	在需要壓縮的元件之前
Request Localization	提供當地語系化支援	在偵測當地語系化的元件之前
Endpoint Routing	定義並限制要求路由	匹配路由而終止
Session	提供管理使用者工作階段的支援	在需要工作階段的元件之前
Static Files	支援靜態檔案的提供和目錄瀏覽	若要求符合檔案則終止
URL Rewrite	提供重寫 URL 及重新導向要求的支援	在使用 URL 的元件之前
WebSockets	啟用 WebSockets 通訊協定	需要接受 WebSocket 請求的元件之前

每個 Middleware 元件皆負有不同任務，且有一定順序，例如 UseAuthentication 方法會在 UseAuthorization 之前做檢查。

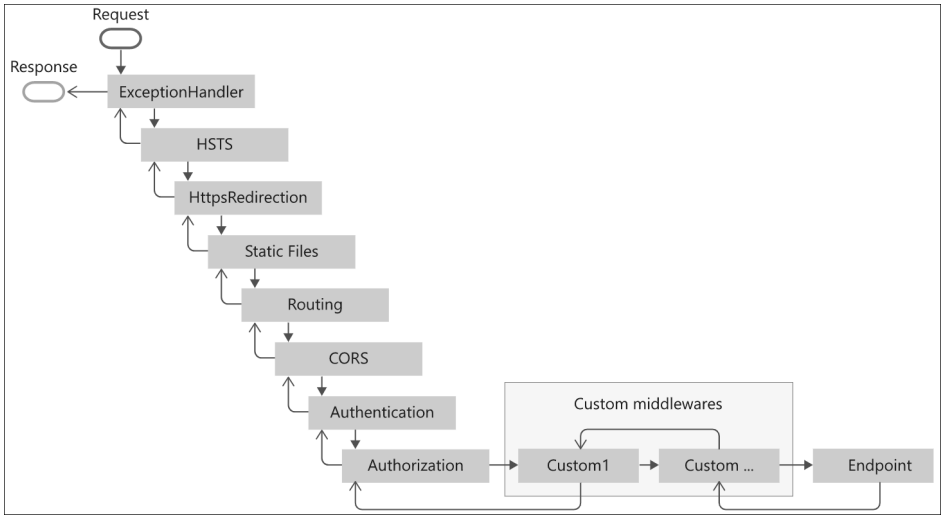


圖 4-6 Middleware 元件執行順序

4-3-5 Configuration 組態

ASP.NET Core 組態是基於 Key-Value Pairs 形式，組態提供者（Configuration Providers）從各種組態來源讀取資料後，再以 Key-Value 成對的方式儲存在組態系統中。

例如 ASP.NET Core 專案預設有 launchSettings.json 和 appsettings.json 兩個組態檔，前者是本機開發電腦環境組態檔，後者是給應用程式使用的組態檔，下面是 appsettings.json 組態內容。

appsettings.json

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings": {
    "DatabaseContext": "Server=(localdb)\\mssqllocaldb;Database=ProductDB;
      Trusted_Connection=True;MultipleActiveResultSets=true"
  },
  "Developer": {
    "Name": "聖殿祭司",
    "Email": "dotnetcool@gmail.com",
    "Website" :  "https://www.codemagic.com.tw"
  }
}
```

預設組態設定

資料庫連線字串設定

自訂組態設定

在新建 ASP.NET Core MVC 專案時，appsettings.json 僅有 Logging 和 AllowedHosts 兩區段，在此新增 ConnectionStrings 和 Developer 區段設定，並調用 IConfiguration 實例存取 ConnectionStrings 和 Developer 兩區段設定值。

Fundamental/ReadAppsettings.cshtml

```
@using Microsoft.Extensions.Configuration
@inject IConfiguration configuration
...
<p>DatabaseContext 資料庫連線 : @configuration["ConnectionStrings:DatabaseContext"]</p>

Developer 資訊如下:
<ul>
    <li>Name : @(configuration.GetValue<string>("Developer:Name"))</li>
    <li>Email: @(configuration.GetValue<string>("Developer:Email", "找不到 Email"))</li>
    <li>Website: @(configuration.GetSection("Developer:Website").Value)</li>
</ul>
```

讀取 ConnectionString 區段組態

讀取 Developer 區段組態

說明：以上用三種語法讀取組態值，至於實際語法為何如此，第 14 章有專門介紹，於此先不細述。

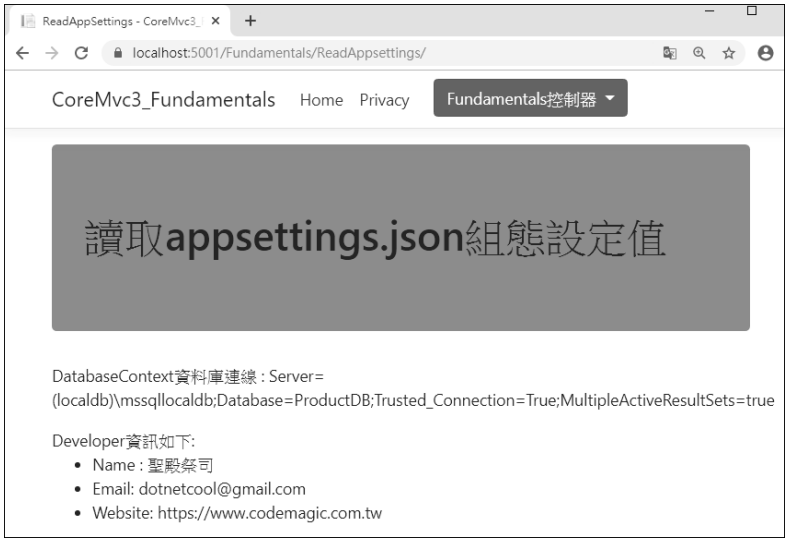


圖 4-7 讀取組態值

組態檔中若有中文設定值，請用 Visual Studio 另存成 UTF-8 編碼格式，否則會顯示亂碼。

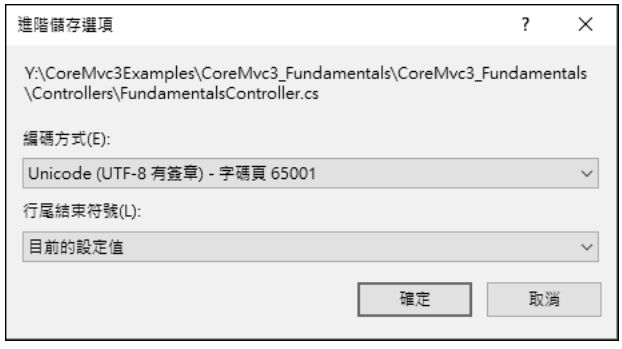


圖 4-8 以 UTF-8 編碼格式儲存

組態資料來源不僅支援 JSON 檔，完整支援如下，每種來源都有相對應的組態提供者負責讀取及解析：

- 環境變數
- 設定檔（JSON、XML、INI）
- 命令列參數
- 目錄檔案（Key-per-file）
- In-Memory .NET 物件
- Azure Key Vault
- Azure App Configuration
- 自訂 Provider

4-3-6 Options Pattern 選項模式

前面應用程式直接存取組態值會造成緊密耦合的問題，為避免這問題，ASP.NET Core 使用 Options Pattern 選項模式抽象化背後的組態系統，將組態資料先繫結到「類別」，應用程式再存取該類別，使得應用程式不直接相依組態系統。日後組態資料結構有任何的調整，應用程式也不必連動做修改，唯一需要調整的是 Options 類別的設定。

Options 類別有兩個要求：

1. 必須為非抽象類別
2. 建構函式必須為 public 且無參數

選項模式建立及使用步驟有四：

1. 建立組態資料，並載入組態系統中
2. 建立 Options 類別
3. 在 DI Container 中以 **services.Configure<T>**方法註冊 Options 類別
4. 在 Controller / View / Services 中存取 Options 類別

範例 4-1 透過 Options 選項模式讀取組態設定

用前面 appsettings.json 組態的 Developer 區段設定值為例，不直接存取組態，而是透過 Options 選項模式讀取組態設定：

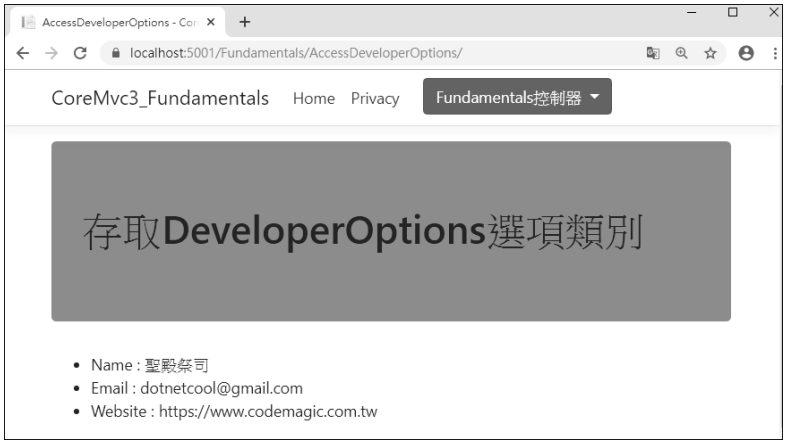


圖 4-9 透過 Options 選項類別讀取組態設定

step01 於 appsettings.json 組態中建立 Developer 開發者資料。

appsettings.json

```
  "Developer": {
    "Name": "聖殿祭司",
    "Email": "dotnetcool@gmail.com",
    "Website": "https://www.codemagic.com.tw"
  }
```

step02 建立 DeveloperOptions 類別，三個屬性名稱須與組態 Key 名稱對映。

Options/DeveloperOptions.cs

```
public class DeveloperOptions
{
    public string Name { get; set; }
    public string Email { get; set; }
    public string Website { get; set; }
}
```

step03 在 Startup 類別的 ConfigureServices 方法註冊 DeveloperOptions 類別，繫結其對應的組態設定。

Startup.cs

```
public void ConfigureServices(IServiceCollection services)
{
   ...
    //在 DI Container 中註冊 DeveloperOptions 類別
    services.Configure<DeveloperOptions>(options =>
      Configuration.GetSection("Developer").Bind(options));
}
```

DeveloperOptions 類別

讀取組態區段

將組態繫結至 DeveloperOptions 類別

step04 在 View 中注入 DeveloperOptions 選項類別實例。

Views/Fundamentals/AccessDeveloperOptions.cshtml

```
@using Microsoft.Extensions.Options
@inject IOptionsMonitor<DeveloperOptions> developerOptions
@{
    var devOptions = developerOptions.CurrentValue;
}
<ul>
    <li>Name : @devOptions.Name</li>
    <li>Email : @devOptions.Email</li>
    <li>Website : @devOptions.Website</li>
</ul>
```

IOptionMonitor<TOptions>

須呼叫 CurrentValue 屬性

讀取 Options 類別屬性

4-3-7 Environment 環境

ASP.NET Core 執行時會讀取 ASPNETCORE_ENVIRONMENT 環境變數，判斷它是 Development、Staging 或 Production 環境，依序代表開發、預備和生產環境，並依環境的不同而載入對映程式或設定。

在何處可看到 ASPNETCORE_ENVIRONMENT 環境變數？可開啟專案的 Properties/launchSettings.json，它是 Host 的組態檔，環境變數也是存放在此。

Properties/launchSettings.json

```
{
  "iisSettings": {
    …
  },
  "profiles": {
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "CoreMvc3_Fundamentals": {
      "commandName": "Project",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      },
      "applicationUrl": "https://localhost:5001;http://localhost:5000"
    },
    "CoreMvc3_Staging": {
      "commandName": "Project",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Staging"
      },
      "applicationUrl": "https://localhost:5001;http://localhost:5000"
    },
    "CoreMvc3_Production": {
      "commandName": "Project",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Production"
      },
      "applicationUrl": "https://localhost:5001;http://localhost:5000"
    }
  }
}
```

ASPNETCORE_ENVIRONMENT 環境變數

Development 開發環境

Development 開發環境

Staging 預備環境

Production 生產環境

TIP

若無環境變數設定，則預設為 Production。

環境變數也能在 Visual Studio 專案的【屬性】→【偵錯】中檢視與編輯。

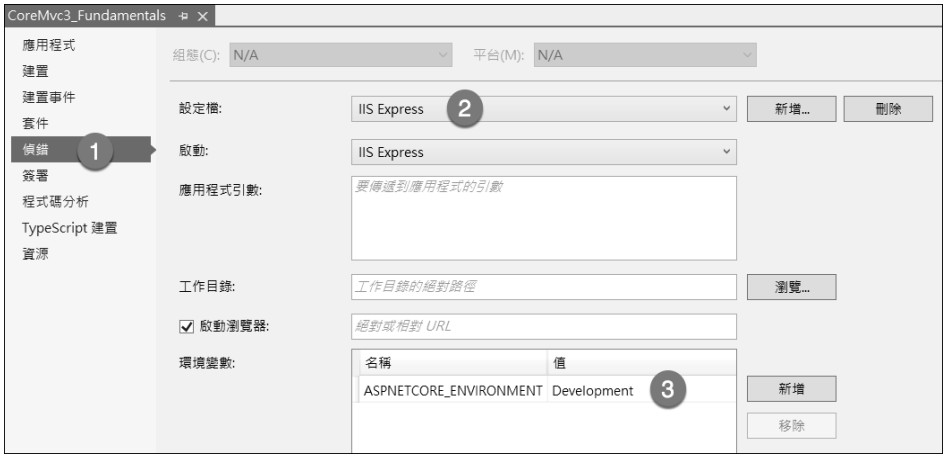


圖 4-10 在 Visual Studio 檢視專案的環境變數

有了環境變數，ASP.NET Core 是如何利用它來載入不同程式或設定？以下介紹幾種類型應用。

- Startup 類別的 Configure 方法

Startup 類別的 Configure 方法，判斷環境是否為 Development，進而決定使用不同的 Middleware 元件：

Startup.cs

```
public class Startup
{
    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
        if (env.IsDevelopment())
```

相依性注入

判斷是否為 Development 環境

```
        {
            app.UseDeveloperExceptionPage();
        }
        else
        {
            app.UseExceptionHandler("/Home/Error");
            app.UseHsts();
        }
        ...
    }
}
```

使用此 Middleware 元件

其他環境使用這兩個 Middleware 元件

說明：IsDevelopment()、IsStaging()和 IsProduction()三個方法可用來判斷是否為 Development、Staging 或 Production 環境，若成立會回傳 true。

也能將程式作以下改寫，二者意思相等：

```
if (env.IsDevelopment())
{
    app.UseDeveloperExceptionPage();
}

if (env.IsStaging() || env.IsProduction())
{
    app.UseExceptionHandler("/Home/Error");
    app.UseHsts();
}
```

✦ 在 Controller 及 View 中使用環境變數

View 若要使用環境變數，是在 Controller / Action 用 ViewData 傳遞給 View：

Controllers/FundamentalsController.cs

```
using Microsoft.AspNetCore.Hosting;
public class FundamentalsController : Controller
{
```

```
    private readonly IWebHostEnvironment _env;
    public FundamentalsController(IWebHostEnvironment env)
    {
        _env = env;
    }

    //顯示環境名稱
    public IActionResult EnvironmentName()
    {
        ViewData["EnvName"] = _env.EnvironmentName;
        return View();
    }
}
```

IWebHostEnvironment 相依性注入

讀取環境變數名稱

或技術上，也能在 View 直接注入 IWebHostEnvironment 相依性物件，而不必透過 ViewData 傳遞環境變數：

Views/Fundamentals/InjectEnvironment.cshtml

```
@inject Microsoft.AspNetCore.Hosting.IWebHostEnvironment env

@{
    //C# 8 switch expression
    string DisplayEnvironment(string envName) =>
        envName switch
        {
            "Development" => "開發環境",
            "Staging" => "預備環境",
            "Production" => "生產環境",
            _ => "其他環境"
        };
}

<p>目前環境是: @DisplayEnvironment(env.EnvironmentName)</p>
```

IWebHostEnvironment 相依性注入

將環境變數名稱傳遞給 local function

✦ Environment 標籤協助程式

使用 Environment 協助標籤的方便性在於，不需在 View 注入 IWebHostEnvironment 相依性物件，直接用<Environment>標籤宣告不

同環境變數所包含的區塊內容，便會依據目前的環境變數為何者，而轉譯輸出其包括的內容。

例如 View 宣告數個<environment>標籤，依照當前環境變數為何者，輸出對映的 HTML 和 JavaScript 程式區塊。例如為 Development，那麼就只輸出 Development 含括的 HTML 和 JavaScript 程式區塊：

Views/Fundamentals/EnvironmentTagHelper.cshtml

```
...
<environment include="Development">
    <h2><span class="badge badge-primary">Development 開發環境</span></h2>
</environment>

<environment include="Staging">
    <h2><span class="badge badge-danger">Staging 開發環境</span></h2>
</environment>

<environment include="Production">
    <h2><span class="badge badge-warning">Production 開發環境</span></h2>
</environment>

@section endJS
{
    <environment include="Development">
        <script>
            function Development() {
                alert("This is Development Environment.");
            }
        </script>
    </environment>

    <environment include="Staging">
        <script>
            function Staging() {
                alert("This is Staging Environment.");
            }
        </script>
    </environment>

    <environment include="Production">
```

```
        <script>
            function Production() {
                alert("This is Production Environment.");
            }
        </script>
    </environment>
}
```

4-3-8 Content Root 與 Web Root

ASP.NET Core 有兩個根目錄專有名詞：❶Conten Root 和❷Web Root，前者代表專案目前所在的基底路徑，後者是專案對外公開靜態資產的目錄，預設路徑為{content root}/wwwroot。

❖ Content Root 內容根目錄

Content Root 是 ASP.NET Core 應用程式內容的基底路徑，例如.exe、.dll、Razor Views、Razor Pages、靜態資產、組態檔皆以 Content Root 路徑為基準。那 Content Root 路徑是怎麼產生的？在 CreateDefaultBuilder 方法建立 Host 主機時，藉由 GetCurrentDirectory 方法回傳路徑，並將其設定到 Content Root 路徑。

Program.cs

```
.ConfigureHostConfiguration(configHost =>
   {
       //設定基底路徑
       configHost.SetBasePath(Directory.GetCurrentDirectory());

       //如想看見 SetBasePath()設定後路徑,可用下面程式抓取
       string contentRoot =
         ((Microsoft.Extensions.FileProviders.PhysicalFileProvider)configHost
           .Properties.Values.FirstOrDefault()).Root;
       ...
   }
```

以 CoreMvc3_Fundamentals 專案而言，Content Root 路徑外觀如下：

```
c:\CoreMvc3Examples\CoreMvc3_Fundamentals\CoreMvc3_Fundamentals
```

在 Controller 或 View 中讀取 Content Root 路徑資訊，是透過注入 IWebHostEnvironment 服務存取 ContentRootPath 屬性：

Controllers/RootPathController.cs

```
...
using Microsoft.AspNetCore.Hosting;
public class RootPathController : Controller
{
    private readonly IWebHostEnvironment _env;
    public RootPathController(IWebHostEnvironment env)
    {                                          ← IWebHostEnvironment 環境變數
        _env = env;
        string contetnRoot = env.ContentRootPath;
    }                                          ← 讀取 ContentRootPath 屬性

    //Content Root Path
    public IActionResult ContentRootPath()
    {
        ViewData["ContentRootPath"] = _env.ContentRootPath;
        return View();                         ← 讀取 ContentRootPath 屬性
    }
}
```

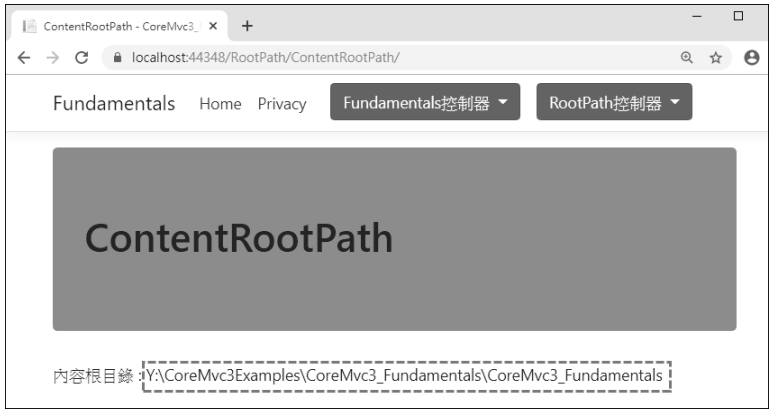


圖 4-11 讀取內容根目錄路徑

或用 dotnet run 執行專案，在終端機視窗也可看見 Content Root 路徑。

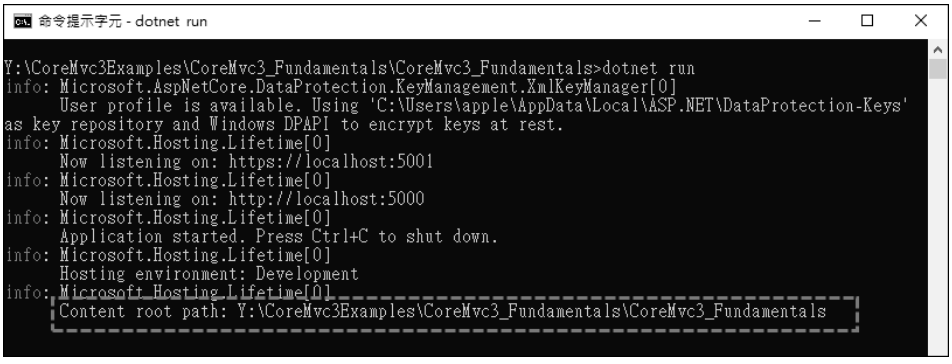


圖 4-12 用 dotnet run 執行顯示 Content Root 路徑

❖ Web Root 根目錄

Web Root 是指專案中包含公開資源的目錄，如 images、css、js、json 和 xml 等靜態檔，Web 根目錄預設路徑為{content root}/wwwroot。也就是 GetCurrentDirectory 方法回傳的路徑再補上「/wwwroot」就是 Web 根目錄路徑。

Web 根目錄 wwwroot 在 Visual Studio 中可直接看見，裡面皆為靜態資源檔，但凡要公開讓網路讀取的 images、css、js、json 或 xml 檔都是在此建立。

圖 4-13 Web 根目錄

❖ ContentRoot 和 WebRoot 路徑調整

一般情況下，ContentRoot 和 WebRoot 使用系統預設值就行了，但若想對 ContentRoot 和 WebRoot 路徑做調整，可在 CreateDefaultBuilder 方法中，用 UseContentRoot 和 UseWebRoot 方法指定路徑參數：

Program.cs

```
public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .UseContentRoot(Directory.GetCurrentDirectory())
        .ConfigureWebHostDefaults(webBuilder =>
        {
            webBuilder.UseStartup<Startup>();
            webBuilder.UseWebRoot(Directory.GetCurrentDirectory() + "/publicshare/");
        });
```

路徑參數

設定 ContentRoot 路徑

設定 WebRoot 路徑

路徑參數

特別是 Web 根目錄，如想將預設的 wwwroot 改用自訂的「publicshare」目錄作為網路公開服務，就可使用 UseWebRoot 方法指定「publicshare」路徑參數，這樣 Web 根目錄之路徑就會改變。但是相對的，所有 images、css、js、xml 檔也必須搬移至「publicshare」目錄才行，否則會讀不到對映的資源檔。

❖ 用 Middleware 設定靜態檔目錄

另一個跟 WebRoot 相關議題是，若在 WebRoot 之外有其他目錄存放著靜態資源檔，希望和 wwwroot 共同服務，那麼可用 Middleware 設定靜態檔目錄：

Startup.cs

```
...
app.UseStaticFiles(); //for the wwwroot folder

app.UseStaticFiles(new StaticFileOptions
{
```

```
    FileProvider = new
     PhysicalFileProvider(Path.Combine(Directory.GetCurrentDirectory(),
     "StaticFilesLibrary"))
});
```

或加用 RequestPath 屬性設定「/StaticFiles」目錄名稱：

```
app.UseStaticFiles(new StaticFileOptions
{
    FileProvider = new
     PhysicalFileProvider(Path.Combine(Directory.GetCurrentDirectory(),
     "StaticFilesLibrary")),
    RequestPath = "/StaticFiles"
});
```

以上兩種方式二擇一，第一種沒有指定 RequestPath，檔案請求路徑維持「～/…」，第二種指定了 RequestPath，View 中的<img>的 src 請求路徑須改成「～/StaticFiles/…」：

Views/RootPath/WebRootPath.cshtml

```
<img src="~/images/ferrari_small.jpg" alt="Ferrari" />
<br />
<img src="~/StaticFiles/images/ferrari_small.jpg" alt="Ferrari" />
```

4-3-9 Logging 記錄

ASP.NET Core 內建記錄資訊的 Logging API，亦可與第三方 Logging 提供者（Providers）搭配使用，內建提供者有：

- Console
- Debug
- Windows 平台的 Event Tracing
- Windows 平台的事件記錄
- TraceSource

- Azure App Service（需參考 Microsoft.Extensions.Logging.AzureAppServices 的 NuGet 套件）
- Azure Application Insights（需參考 Microsoft.Extensions.Logging.ApplicationInsights 的 NuGet 套件）

Logging 提供者會將 Log 記錄輸出或寫入到不同目的端，例如 Console 提供者會輸出 Log 記錄到 Console 中，事件記錄提供者就寫入事件檢視器，而 Azure Application Insights 儲存 Logs 在 Azure Application Insights 中。

ASP.NET Core 預設會加入 Console、Debug 和 Windows 平台的 Event Tracing 提供者，若想加入其他的，可在 Program 的 CreateHostBuilder 方法中用 ConfigureLogging 方法加入其他提供者：

Program.cs

```
using System.Diagnostics;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;
using Microsoft.Extensions.Logging.AzureAppServices;

public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .ConfigureLogging(loggingBuilder => {
            loggingBuilder.ClearProviders();
            loggingBuilder.AddConsole();
            loggingBuilder.AddDebug();
            loggingBuilder.AddEventSourceLogger();
            loggingBuilder.AddEventLog();
            loggingBuilder.AddTraceSource(new SourceSwitch("loggingSwitch","Verbose"),
              new TextWriterTraceListener("LoggingService.txt"));
            loggingBuilder.AddAzureWebAppDiagnostics();
            loggingBuilder.AddApplicationInsights();
        })
        .ConfigureWebHostDefaults(webBuilder =>
        {
            webBuilder.UseStartup<Startup>();
        });
```

設定 Logging 提供者

系統預設提供者

加入其他提供者

如加入多重 Logging 提供者，Logs 記錄能夠發送到多重目的作寫入，例如預設加入了三種提供者，那麼記錄資訊時，就會同時寫入到這三種目的端。

範例 4-2 在 Controller 控制器中使用 Logging 記錄資訊

在此於 Home 控制器/Index 動作方法使用 Logging 記錄資訊，步驟如下：

step01 在 Home 控制器建構函式注入 ILogger 相依性實例。

Controllers/HomeController.cs

```
...
using Microsoft.Extensions.Logging;

namespace CoreMvc3_Fundamentals.Controllers
{
    public class HomeController : Controller
    {
        private readonly ILogger<HomeController> _logger;
        public HomeController(ILogger<HomeController> logger)
        {
            _logger = logger;
        }
}
```

DI 相依性注入

指定 Category 分類名稱

說明：Category 分類名稱指定為控制器類別名稱，但其實可為任何字串。

step02 在 Index 動作方法以 Log 方法記錄資訊。

```
public IActionResult Index()
{
    EventId eventId = new EventId(1234, "我的記錄資訊");
    _logger.LogWarning(eventId, "Logging 記錄資訊- Home/Index 被呼叫");
    //以上亦可寫成以下一行
```

使用 LogWarning 方法記錄資訊

```
    _logger.LogWarning(1234, "Logging 記錄資訊- Home/Index 被呼叫");
                              事件 Id          記錄資訊
    return View();
}
```

說明：Logging 支援六種層級記錄方法：LogTrace、LogDebug、LogInformation、LogWarning、LogError、LogCritical。雖說每個方法都能使用，但因涉及系統 Loggin 預設組態層級關係，低於層級設定的記錄方法，資訊不會寫至記錄目的端，細節稍後再說明。

step03 按 F5 以 IIS Express 執行，瀏覽 Home/Index 網址，在 Visual Studio【偵錯】→【視窗】→【輸出】→顯示輸出來源「偵錯」，可看見 Log 記錄輸出。

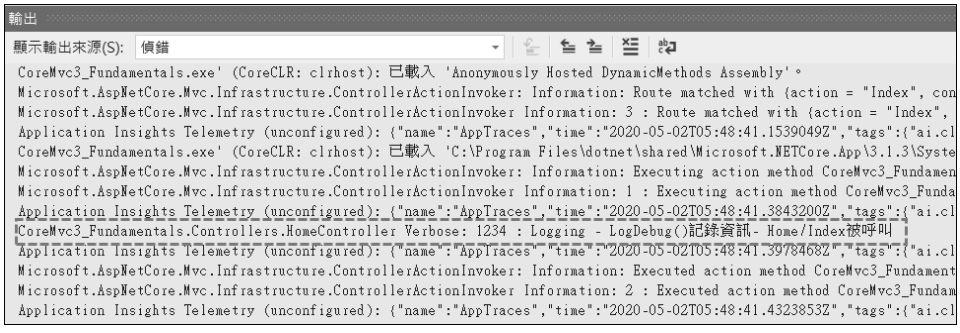


圖 4-14 在 Visual Studio 偵錯輸出中檢視 Log 記錄資訊

step04 另一種是用 dotnet run 執行，再瀏覽 Home/Index，於終端機視窗中亦可看到 Log 記錄輸出資訊。

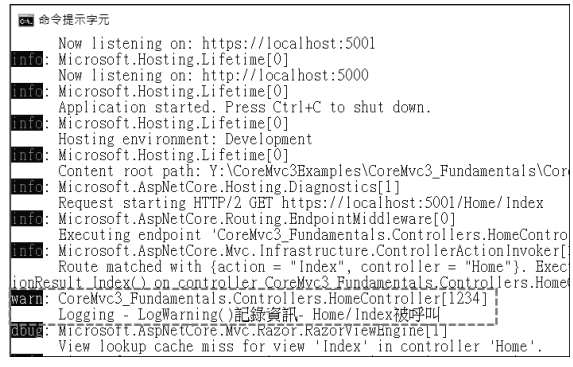


圖 4-15 在終端機視窗檢視 Log 記錄輸出資訊

step05 若有用 AddEventLog()方法加入事件記錄提供者，無論用哪種方式執行，皆會寫入 Windows 事件檢視器中。

圖 4-16 Windows 平台事件檢視器中的記錄資訊

❖ Log Level 記錄層級

每個 Log 記錄時皆會指定一個 LogLevel 列舉值，作用是指出記錄是嚴重或重要性程度，依重要性程度最高至最低，LogLevel 列舉值有以下幾種。

表 4-2 Log 記錄層級

等級	代碼	說明
None	6	不寫入 Log 訊息。指定 Logging 類別不應寫入任何的訊息。
Critical	5	記錄不可復原的應用程式，系統崩潰或災難性故障，需要立即關注。
Error	4	記錄當前執行流程因失敗而停止，而這個失敗是指當前的執行活動，但不是整個應用程式範圍的失敗。
Warning	3	記錄應用程式中的異常或非預事件外，但這些異常會不導致應用程式停止。

等級	代碼	說明
Information	2	用於記錄應用程式一般流程，而這些 Log 應該有一些長期價值。
Debug	1	在開發期間使用的互動式調查 Log 記錄。這些 Log 主要是包含 debugging 偵錯資訊，且沒有長期保存的價值。
Trace	0	此等級的 Log 包含大多數詳細資訊，這些訊息可能包含敏感的應用程式資料，這些訊息預設是 Disable 關閉，在 Production 生產環境中絕對不要開啟。

那 Log Level 記錄層級會如何影響程式？在 appsettings.Development.json 組態中，Logging 設定如下：

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Debug",
      "System": "Information",
      "Microsoft": "Information"
    }
  }
}
```

預設為 Debug 層級

以 ASP.NET Core 專案 Logging 的 Default 預設值為「Debug」情況下，必須需使用至少跟它一樣或較高等級的方法，Log 才會寫入至記錄目的端，例如 LogDebug、LogInformation、LogWarning、LogError、LogCritical 方法皆可，但用 LogTrace 方法就不做任何記錄了。

4-4 結論

本章介紹了每種服務運作原理，以及如何使用與調整服務。在理解個別服務原理後，再到串連起所有服務協作與運行順序後，腦中就會呈現井然有序的總體圖，此時便初步掌握 ASP.NET Core 系統開發火候，而不致落入零散無章，缺乏開發方向感的窘境。