多奇·教育訓練

ASP.NET Core 5 開發實戰

新手上路篇

多奇數位創意有限公司

技術總監 黃保翕（Will 保哥）

https://blog.miniasp.com

Introducing ASP.NET Core framework

簡介 ASP.NET Core 框架

關於 ASP.NET Core 開發框架

- 跨平台、高效能、開放原始碼架構
- 主要用來開發現代化、雲端、網際網路連線應用程式
- 透過 ASP.NET Core 可以
  - 建置網站或 Web API
  - 使用任何你喜愛的開發工具 (Visual Studio, VSCode, Rider, …)
  - 執行在 Windows、macOS 或 Linux 平台
  - 使用 .NET Core 或 .NET Framework 皆可
  - 部署到雲端或公司內部主機
  - 完全免費

ASP.NET Core 明顯的改變

- 比較 ASP.NET MVC 5 與 ASP.NET Core 之間的專案架構差異
  - 沒有 `web.config` 檔案（**內建 Kestrel 超高速網站伺服器**）
  - 沒有 `global.asax` 檔案（**不再綁死 IIS 網站伺服器**）
  - 新版 ASP.NET 錯誤頁面（**不再有黃底紅字畫面**）
  - 所有的服務都改用 **DI**（**相依性注入**）方式提供
  - 真正的前後端分離
    - 預設使用 **wwwroot** 靜態檔案資料夾分離前後端檔案
- **RIP** (Rest in Peace)
  - ASP.NET Web Form
  - ASP.NET Web Services
  - WCF Services (僅支援 WCF 用戶端函式庫)

ASP.NET MVC 5 例外狀況畫面

Server Error in '/' Application.

錯誤發生

Description: An unhandled exception occurred during the execution of the current web request. Please review the stack trace for more information about the error and where it or

Exception Details: System.ArgumentException: 錯誤發生

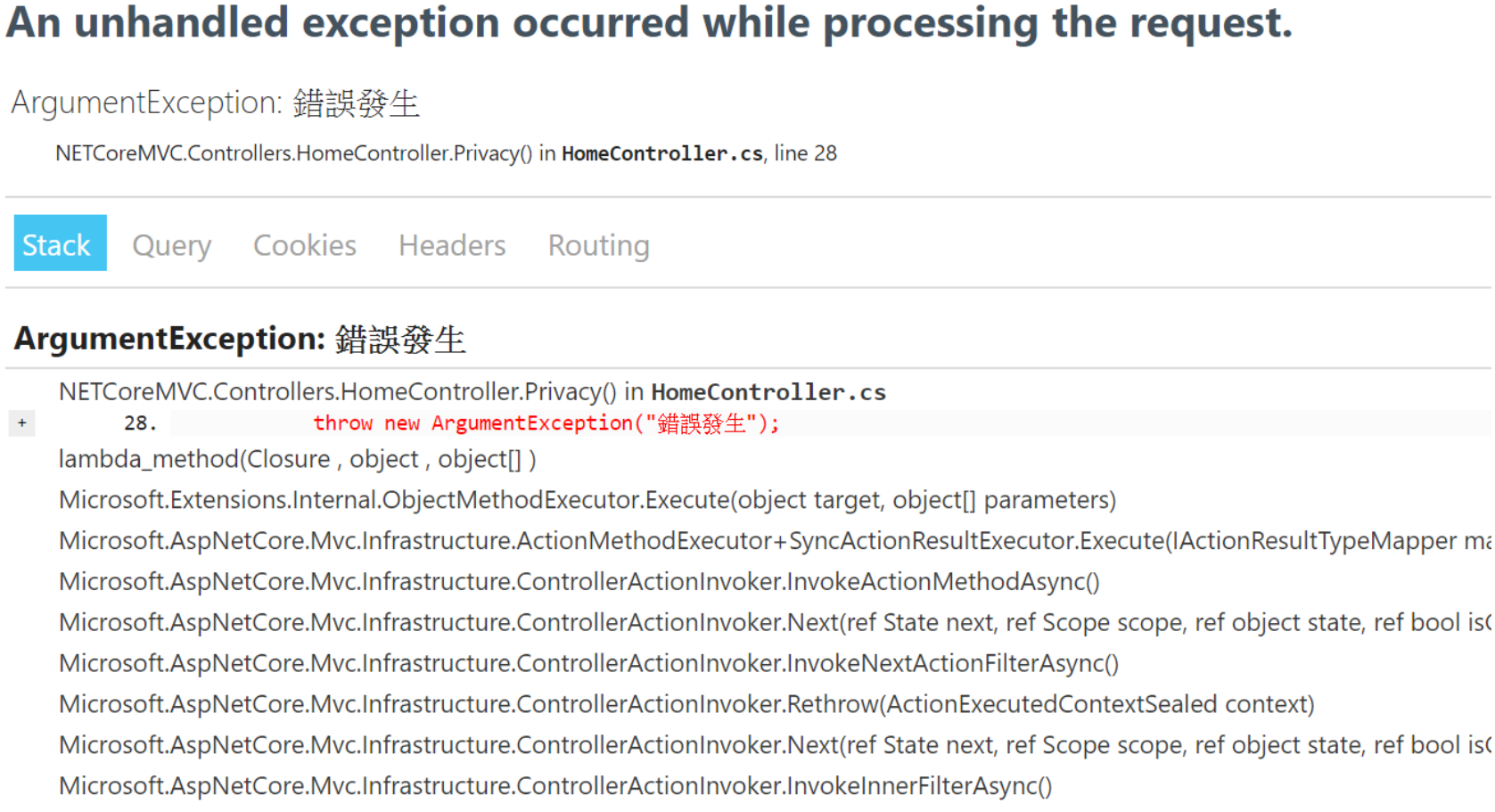
Source Error:

```
Line 17:        {
Line 18:            ViewBag.Message = "Your application description page.";
Line 19:            throw new ArgumentException("錯誤發生");
Line 20:            return View();
Line 21:        }
```

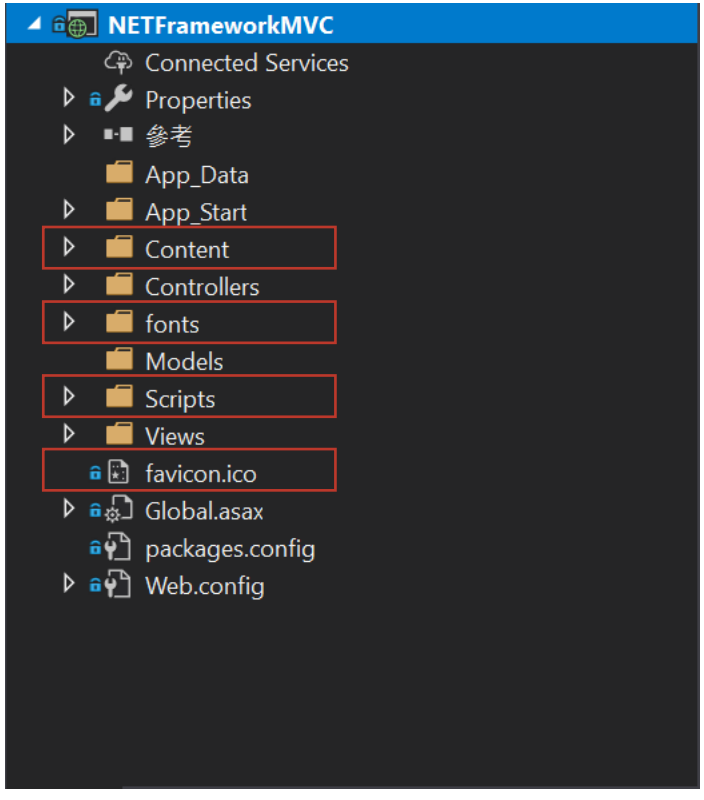
Source File: C:\Users\wakau\source\repos\CompareMvcSolution\NETFrameworkMVC\Controllers\HomeController.cs **Line:** 19

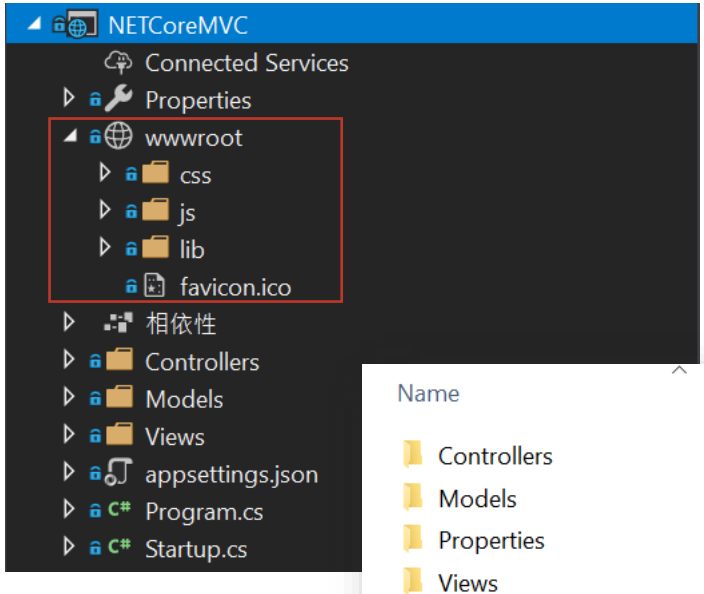
Stack Trace:

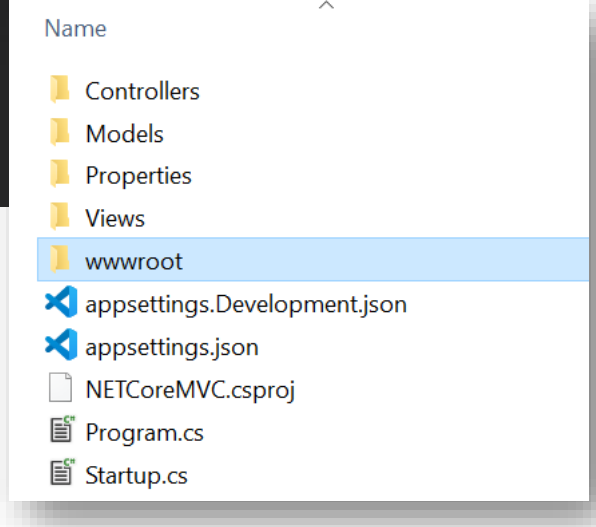
ASP.NET Core 5.0 例外狀況畫面



前後端分離檔案比較


NETFrameworkMVC
Connected Services
Properties
參考
App_Data
App_Start
Content
Controllers
fonts
Models
Scripts
Views
favicon.ico
Global.asax
packages.config
Web.config


NETCoreMVC
Connected Services
Properties
wwwroot
css
js
lib
favicon.ico
相依性
Controllers
Models
Views
appsettings.json
Program.cs
Startup.cs


Name
Controllers
Models
Properties
Views
wwwroot
appsettings.Development.json
appsettings.json
NETCoreMVC.csproj
Program.cs
Startup.cs

可以裝載 ASP.NET Core 的執行環境

- Kestrel
  - 內建於 ASP.NET Core 框架之中的簡易伺服器
  - 支援 HTTP/2、Negotiate、Kerberos、NTLM on Windows
- IIS
  - 網站一樣可以部署到企業現有的 IIS 網站伺服器中
  - 透過 IIS 的 ASP.NET Core Module 模組
- HTTP.sys
  - 一套僅限於 Windows 環境的 Web Server 實作
  - 支援 Windows 驗證、SNI SSL、HTTP/2 over TLS、回應快取、...
- Windows Service
- Nginx / Apache
- Docker

ASP.NET Core 包含哪些東西

- Web app
  - MVC / Tag Helpers / View components / Action Filters / Areas
  - Razor Pages / Razor syntax / Razor class libraries (RCL)
  - Blazor / Razor components class libraries
- Web API
- SignalR
- gRPC
- Identity
- *Entity Framework Core*

What's new in ASP.NET Core 5.0

What's new in ASP.NET Core 3.1

What's new in ASP.NET Core 3.0

What's new in ASP.NET Core 2.2

What's new in ASP.NET Core 2.1

What's new in ASP.NET Core 2.0

Creating new project and directory structures

建立新專案與認識目錄結構

使用 .NET Core CLI 建立專案範本

- 列出所有專案範本
  - **dotnet new -l**
  - **dotnet new -l --type=project -lang=C#**
  - **dotnet new -l --type=project -lang=VB**
  - **dotnet new -l --type=item**
  - **dotnet new -l --type=other**
- 建立不同的專案範本 (.NET Core 3.1)

  https://github.com/doggy8088/dotnet-new-templates
- 自訂 dotnet new 專案範本的重要觀念與範例

  https://blog.miniasp.com/post/2020/01/19/dotnet-new-template-how-to

ASP.NET Core Empty (web)

- 完全透過 ASP.NET Core 的 Endpoint Routing 輸出網頁
  - Understanding ASP.NET Core Endpoint Routing | aregcode

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment()) {
        app.UseDeveloperExceptionPage();
    }

    app.UseRouting();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapGet("/", async context =>
        {
            await context.Response.WriteAsync("Hello World!");
        });
    });
}
```

ASP.NET Core Web App (MVC)

- `Controllers` 控制器
- `Models` 模型
- `Views` 檢視頁面

- `wwwroot` 所有網站的**靜態資源**資料夾

ASP.NET Core Web App (Razor Pages)

- Pages　　所有 Razor 頁面
- Pages/Shared　　所有預設共用的 Razor 頁面
- Pages/_ViewStart.cshtml　　設定預設 Layout 的地方
- Pages/_ViewImports.cshtml　　設定預設 View 共用設定的地方

- wwwroot　　所有網站的**靜態資源**資料夾

- Properties/launchSettings.json
  - 透過 dotnet run 啟動時的設定檔

ASP.NET Core Web API

- Controllers 控制器

ASP.NET Core gRPC Service

- Protos　　Protocol Buffer Files
  - *.proto
  - https://grpc.io/
  - Introduction to gRPC on .NET Core

- Services　　gRPC 服務的實作類別

Blazor Server App

• Pages	所有頁面
• Pages/_Host.cshtml	預設首頁 (_Host.cshtml)
• Pages/*.razor	所有 Blazor 元件(頁面)
• Shared	共用 Blazor 元件的資料夾
• Data	預設的資料存取類別所在的資料夾
• wwwroot	所有網站的**靜態資源**資料夾

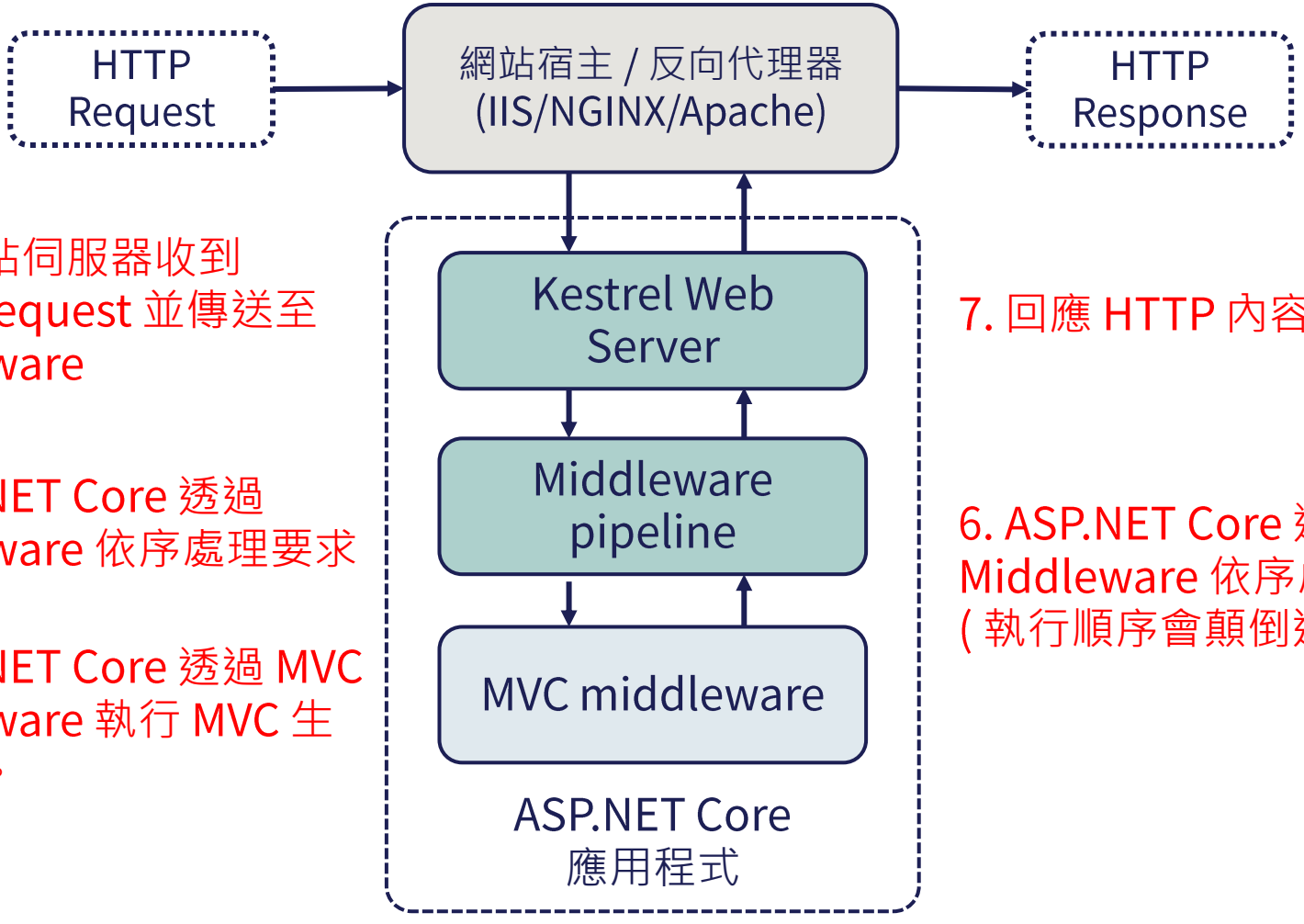
ASP.NET Core Execution Lifecycle

ASP.NET Core 執行生命週期

1. 使用者發出 HTTP 要求

2. 透過反向代理主機**轉送要求**到 ASP.NET Core 應用程式

8. 轉送 HTTP 回應給使用者



3. 由網站伺服器收到 HTTP Request 並傳送至 Middleware

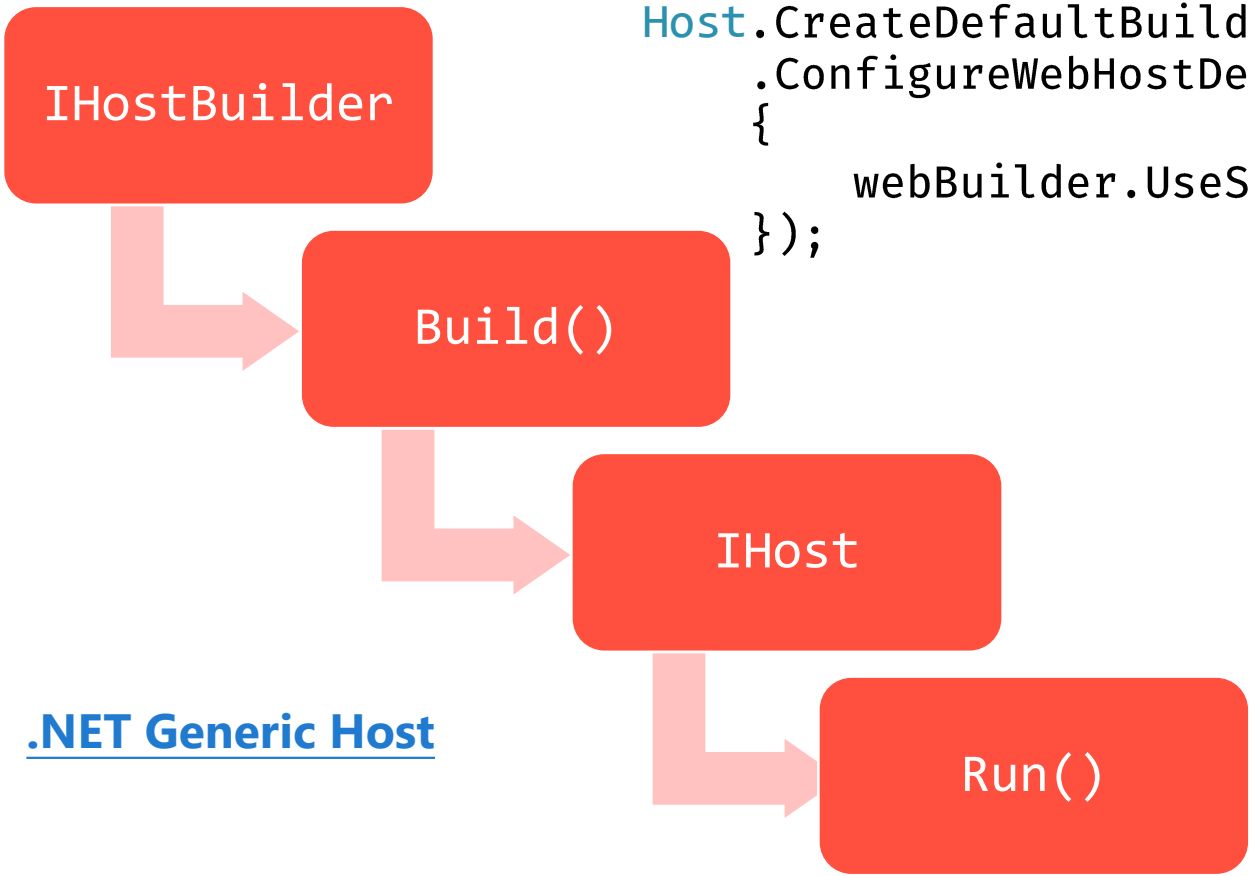
4. ASP.NET Core 透過 Middleware 依序處理要求

5. ASP.NET Core 透過 MVC Middleware 執行 MVC 生命週期。

7. 回應 HTTP 內容給用戶端

6. ASP.NET Core 透過 Middleware 依序處理要求 (執行順序會顛倒過來)

ASP.NET Core 的啟動流程




```
Host.CreateDefaultBuilder(args)
    .ConfigureWebHostDefaults(webBuilder =>
    {
        webBuilder.UseStartup<Startup>();
    });
```

.NET Generic Host

程式進入點：Program

- 負責 ASP.NET Core 的主要啟動流程 (包含預設定義)

```
10  namespace web2
11  {
12      public class Program
13      {
14          public static void Main(string[] args)
15          {
16              CreateHostBuilder(args).Build().Run();
17          }
18
19          public static IHostBuilder CreateHostBuilder(string[] args) ⇒
20              Host.CreateDefaultBuilder(args)
21                  .ConfigureWebHostDefaults(webBuilder ⇒
22                  {
23                      webBuilder.UseStartup<Startup>();
24                  });
25      }
26  }
```

Host.CreateDefaultBuilder(args)

- 此設計用到了 Builder Pattern (建造者模式)
- 查看 CreateDefaultBuilder(args) 原始碼
  - UseContentRoot
  - ConfigureHostConfiguration
  - ConfigureAppConfiguration
  - ConfigureLogging
  - UseDefaultServiceProvider

ConfigureWebHostDefaults()

- 查看 ConfigureWebHostDefaults() 原始碼
- 查看 WebHost.ConfigureWebDefaults() 原始碼
  - ConfigureAppConfiguration
  - UseKestrel
  - ConfigureServices
    - HostFilteringOptions / HostFilteringStartupFilter
    - ForwardedHeadersOptions / ForwardedHeadersStartupFilter
    - services.AddRouting()
  - UseIIS
  - UseIISIntegration

應用程式設定：Startup (建構式)

- 取得設定**服務**與**相依性注入**(DI)

```
public Startup(IConfiguration configuration)
{
    Configuration = configuration;
}

public IConfiguration Configuration { get; }
```

應用程式設定：Startup (服務與設定)

- 設定**服務**與**相依性注入**(DI)

```
public void ConfigureServices(IServiceCollection services)
{
}
```

- 設定 ASP.NET Core 的 HTTP 要求管線 (Middleware)

```
public void Configure(IApplicationBuilder app,
                      IWebHostingEnvironment env)
{
}
```

應用程式設定：Startup.cs

- **Startup()**
  - 建構式：通常只用來注入 `IConfiguration` 服務
- **ConfigureServices()**
  - 將應用程式所需的「**服務**」註冊到 **DI 容器**中
  - 此方法只會在應用程式啟動時執行一次！
- **Configure()**
  - **此方法用來設定 ASP.NET Core 如何回應用戶端要求**
  - 此方法一定要定義 ASP.NET Core 才能執行
  - 必須認識 Middleware Pipeline 的觀念
- ASP.NET Core 中的應用程式啟動

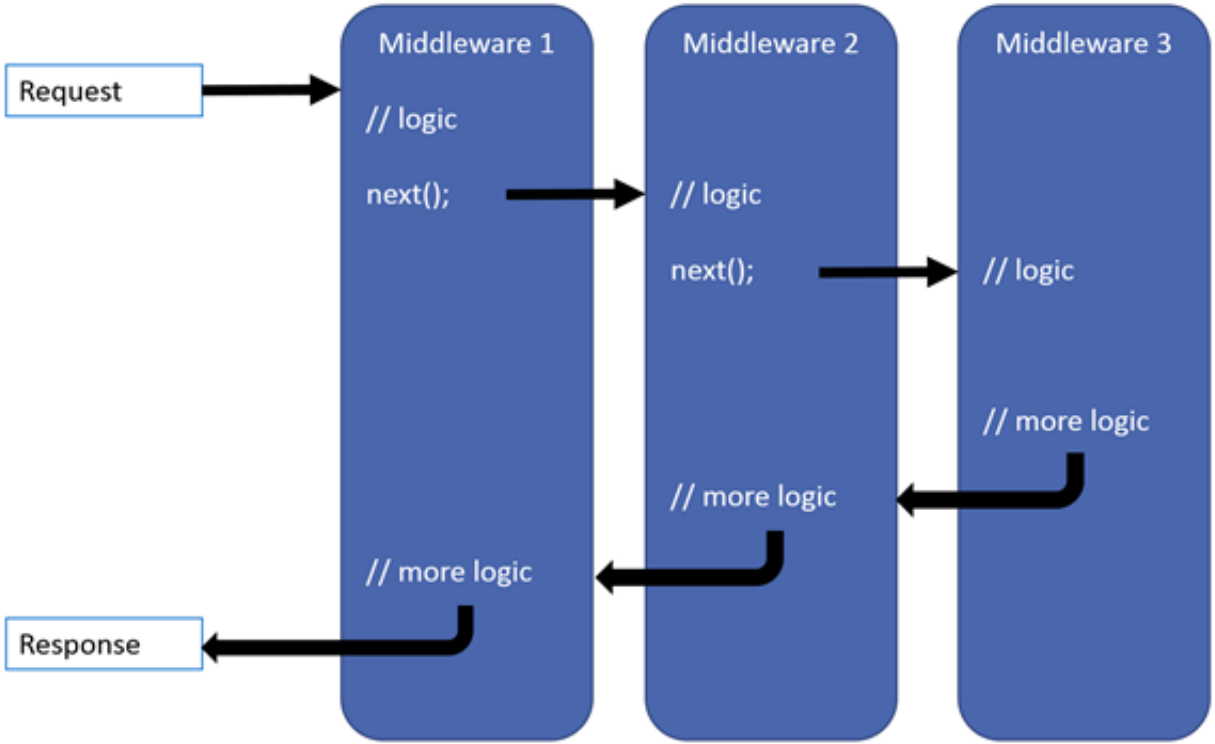
Middlewares

ASP.NET Core 的 Middleware 架構

認識 Middleware (中介軟體) 架構

- ASP.NET Core 中介軟體 (Middleware)
- Middleware 用來處理所有 HTTP **Requests** 與 **Responses**
- Middleware 的執行順序由 `Startup` 類別中的 `Configure` 方法決定
- 每一個 Middleware 元件可以選擇 (二選一)
  1. 將 Requests 傳給下一個 Middleware 元件
  2. 直接 Response 回用戶端
- 使用 IApplicationBuilder 擴充方法串接 Requests/Responses
  - Use
  - Run
  - Map

透過 Middleware 定義要求管線



定義 Middleware 執行順序的範例

- 定義在 Startup 類別的 Configure 方法中
  - Request 過程是**由上而下**的執行順序
  - Response 過程是**由下而上**的執行順序
  - **正確的定義 Middleware 順序非常重要！**
    這對 **安全性**、**執行效能** 與 **功能性** 都有非常密切的關係！

```
public void Configure(IApplicationBuilder app)
{
    app.UseHttpsRedirection();
    app.UseStaticFiles();
    app.UseRouting();
    app.UseAuthorization();
    app.UseEndpoints(endpoints => {
        endpoints.MapControllerRoute(name: "default",
            pattern: "{controller=Home}/{action=Index}/{id?}");
    });
}
```

示範 Middleware 執行順序對功能的影響

```
public void Configure(IApplicationBuilder app)
{
    // 強迫將 HTTP 全部轉向 HTTPS
    app.UseHttpsRedirection();

    // 服務靜態檔案傳輸
    app.UseStaticFiles();

    // 設定 HTTP 回應壓縮
    app.UseResponseCompression();

    // 設定授權檢查
    app.UseAuthorization();
```

靜態檔案無法使用 HTTP 回應壓縮
因為如果處理靜態檔案時，
這個中介層不會執行 **next();** 方法

ASP.NET Core MVC 將會
包含 HTTP 回應壓縮功能

自訂 Middleware - 使用 Run 擴充方法

- 最後一個 Middleware
  - 一般都不會直接跑 .Run() 方法
  - 通常都使用 `app.UseEndpoints();` 來建立應用程式必要的端點
- 自行寫 Run 的範例

```
public void Configure(IApplicationBuilder app)
{
    app.Run(async (context) => {
        await context.Response.WriteAsync("Hello World!");
    });
}
```

自訂 Middleware - 使用 Use 擴充方法

- 用自訂的程式碼邏輯來設定一層 Middleware
- 可以透過呼叫 `next();` 來決定是否進入下一個 Middleware

```
public void Configure(IApplicationBuilder app)
{
    app.Use(async (context, next) => {
        context.Response.ContentType = "text/plain";
        await context.Response.WriteAsync("Start\r\n");
        await next();
        await context.Response.WriteAsync("End\r\n");
    });

    app.Run(async (context) => {
        await context.Response.WriteAsync("Hello World.\r\n");
    });
}
```

自訂 Middleware - 使用 Map 擴充方法

- 用自訂的 "**路由**" 來設定獨立的 Pipeline

```
public void Configure(IApplicationBuilder app)
{
    app.Map("/map1", (app1) => {
        app1.Run(async (context) => {
            await context.Response.WriteAsync("Hello Map1.\r\n");
        });
    });

    app.Map("/map2", (app1) => {
        app1.Run(async (context) => {
            await context.Response.WriteAsync("Hello Map1.\r\n");
        });
    });

    app.Run(async (context) => {
        await context.Response.WriteAsync("Hello World.\r\n");
    });
}
```

app.UseEndpoints()

```
app.UseRouting();

app.UseEndpoints(endpoints =>
{
    endpoints.MapGet("/map1", async(context) => {
        await context.Response.WriteAsync("Hello Map1.\r\n");
    });

    endpoints.MapGet("/map2", async(context) => {
        await context.Response.WriteAsync("Hello Map2.\r\n");
    });

    endpoints.MapGet("/", async(context) => {
        await context.Response.WriteAsync("Hello World.\r\n");
    });

    endpoints.MapGet("/{*path}", async(context) => {
        string path = context.Request.RouteValues["all"].ToString();
        await context.Response.WriteAsync(path);
    });
});
```

Dependency Injection

ASP.NET Core 相依性注入

ASP.NET Core 大量使用 DI 設計架構

- 在 ASP.NET Core 已經內建許多可注入的"服務"
- 由 Startup.ConfigureServices() 負責設定 DI 容器
  - 需要的時候才加入服務集合 (IServiceCollection)

```
// This method gets called by the runtime.
// Use this method to add services to the container.
0 個參考 | 0 例外狀況
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<CookiePolicyOptions>(options =>
    {
        // This lambda determines whether user consent for non-essential
          cookies is needed for a given request.
        options.CheckConsentNeeded = context => true;
        options.MinimumSameSitePolicy = SameSiteMode.None;
    });

    services.AddMvc().SetCompatibilityVersion
      (CompatibilityVersion.Version_2_2);
}
```

GDPR

由 DI 容器控管服務的生命週期

- Transient (暫時性的實體)
  - **每次注入時**都會**建立全新物件**

```
services.AddTransient<IOperationTransient, Operation>();
```

- Scoped (具範圍的實體)
  - **每個 HTTP 要求**只會**共用一個物件**，並在**第一次注入時**建立物件

```
services.AddScoped<IMyDependency, MyDependency>();
```

- Singleton (單一實體)
  - 當**應用程式啟動**時，會在**第一次注入時**或在**註冊進 DI 容器時**建立物件

```
services.AddSingleton<IOperationSingleton, Operation>();
services.AddSingleton<IOperationSingletonInstance>(new Operation(Guid.Empty));
```

如何設計一個好的服務

- 服務本身最好也用 DI 注入其他服務實體
- 盡量不要用到靜態屬性或靜態方法 (Static Cling)
- 不要在服務類別中直接 new 出另一個服務的物件
- 依循 SOLID 原則進行設計 (**很重要**)
  - SRP **一個類別**應該**只有一個改變的理由**
    - 如果你習慣在服務中注入過多的服務，通常意味著 SRP 不佳
    - Razor Page 或 Controllers/Views 都應處理 UI 邏輯為主
  - OCP **軟體實體**應能**開放擴充**但**封閉修改**
  - LSP **子型別**必需**可替換**為他的**基底型別**
  - ISP **用戶端**不應該強迫相依於**沒用到的介面**
  - DIP **抽象**不應該相依於**細節**。而**細節**則應該**相依於抽象**

使用 DI 的注意事項

- 實作 IDisposable 應該注意 DI 的用法
  - 透過 ASP.NET Core 的 DI 容器會幫你管理**物件生命週期**
    - `services.AddScoped<Service1>();`
    - `services.AddSingleton<Service2>();`
    - `services.AddSingleton<ISomeService>(sp => { return new SomeServiceImplementation()); } // 工廠方法`
  - 如果是自己 new 出物件的，就不會自動回收物件！
    - `services.AddSingleton<Service3>(new Service3());`
    - `services.AddSingleton(new Service3());`
- 請確保 Singleton 服務必須擁有**執行緒安全** (Thread Safety)
- 在 **Middleware** 中**避免使用建構式注入**來存取 **Scoped** 服務
  - 建議實作注入在 Invoke 或 InvokeAsync 方法中

使用 DI 設計服務類別的最佳實務建議

- 避免儲存資料或設定在服務容器中。
  - 目的是為了減少服務與服務之間的相依性，也能降低錯誤
  - 盡量利用組態選項模式來共用服務之間會用到的組態資料
  - 避免透過一個服務來存取另一個服務的內容 (盡量啦)
- 盡可能的不要使用 **static** 宣告靜態類別/屬性/方法
  - 避免使用服務定位器模式 (Service Locator Pattern)
  - 避免使用 IServiceProvider.GetService 自行取得服務
- 避免以靜態方式存取 HttpContext 物件
  - 你可以在 Controllers 用、可以在 PageModel 裡面用，但就是不要在需要 DI 的 Service 類別中使用！

Built-in Middlewares

ASP.NET Core 內建的 Middlewares

ASP.NET Core 內建的 Middleware 列表

Middleware	主要功能
Authentication	提供身分驗證支援。在 HttpContext.User 之前執行。
Authorization	提供角色授權支援。緊接著 Authentication 之後執行。
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` 與 `SameSite` 等。
CORS	設定 CORS (跨來源資源共用)。 必須出現在所有需要使用 CORS 的 Middleware 之前。
Diagnostics	多種不同的 Middleware 負責處理例外狀況與顯示預設頁面等設定。
ForwardedHeaders	轉送 Proxy 標頭 (Headers) 到目前的 HTTP 要求
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。

ASP.NET Core 內建的 Middleware 列表

Middleware	主要功能
Header Propagation	設定透過 HttpClient 發出的要求必須包含特定 HTTP Headers 主要用來進行 Tracing (追蹤) 之用。
HTTP Method Override	允許傳入的 HTTP POST 覆寫成用其他的 HTTP Method 呼叫
HTTPS Redirection	將所有 HTTP 要求全部轉向到 HTTPS 連結
HSTS	提供 HSTS (HTTP Strict Transport Security) 支援 HSTS = HTTP 強制安全傳輸技術
MVC	處理 MVC/Razor 頁面要求
OWIN	以 OWIN 為基礎之應用程式、伺服器和中介軟體的 Interop

ASP.NET Core 內建的 Middleware 列表

Middleware	主要功能
Response Caching	提供 HTTP 回應快取支援
Response Compression	提供 HTTP 回應壓縮支援
Request Localization	提供多國語系 I18N 支援
Endpoint Routing	定義與限制要求的路由
SPA	提供 SPA 頁面的預設首頁支援
Session	提供 Session 功能支援
Static Files	提供靜態檔案存取與目錄瀏覽支援
URL Rewriting	提供 URL Rewrite (網址重寫與轉向)支援
WebSockets	啟用 WebSockets 通訊協定支援

聯絡資訊

The Will Will Web

網路世界的學習心得與技術分享

http://blog.miniasp.com/

Facebook

Will 保哥的技術交流中心

http://www.facebook.com/will.fans

Twitter

https://twitter.com/Will_Huang

多奇·教育訓練

THANK YOU!

Q&A