APPENDIX

B

ASP.NET MVC 6 預覽

這本書才剛講完 ASP.NET MVC 5，很快的，MVC 6 已經出現在微軟的官方網頁，令人感嘆技術的快速演進，程式員也追得非常辛苦，不管是前端也好或是後端，快速發展的技術已經是資訊業界的常態了，所以走這一行的人要有所體認，將快速演進視為一種享受，因為再怎麼變，基礎知識還是那些，所以只要掌握基礎，它再怎麼變，也不會脫離基礎的技術原理，MVC 6 也是如此。

MVC 6 是 ASP.NET Core 平台的其中之一，要說到 MVC 6 最大的變化，其實也只是將 MVC 和 Web API 兩者整合，並且更新 ASP.NET Routing 的核心，讓 MVC 和 Web API 能適用於同一個 Routing API，而 MVC 和 Web API 同時共用 Model 以及 Controller (Web API 的 ApiController 併入 MVC 的 Controller 基礎類別)，也就是之後所有適合在 MVC 中的寫法，都能用於 Web API，反之亦然。Razor 仍然是 MVC 6 的 View 的首選指令碼架構，而 JSON 的生成則沒有太大變化。

在本書問世之時，ASP.NET 的新一代版本 vNext 正在原型階段開發中 (也是本書之前的 ASP.NET MVC 6 預覽的編寫背景)，於 Build 2015 開發人員研討會中命名為 ASP.NET 5 並釋出 Beta 4 的版本，至 2015/11/18 發布 RC1 版本，原本以為 ASP.NET 5 可以來得及於今年農曆過年前問世，但 2016 年 1 月微軟將 ASP.NET 5 改名為 ASP.NET Core (連同.NET Core 以及 Entity Framework Core 合稱 Core 家族)，同時還決定要將 ASP.NET 5 與.NET Core 的指令工具合併，而推遲了最終版本的發行時間，RC2 預期在 Build 2016 開發人員研討會中發布，會不會有 RC3 很難說，可以預見的是最終版本可能要等到今年第三甚至第四季才有可能與大家見面。

ASP.NET Core 是微軟由官方開發的第一個具跨平台能力 (Windows、Mac OSX、Linux) 的 Web 應用開發框架，研發初期是以代號 K (Project K) 為名，包含了執行期環境 (KRE)、版本管理工具 (KVM)、程式啟動工具 (K)、封裝管理工具 (KPM) 以及相關的函式庫，包含 ASP.NET MVC 6 以及 Entity Framework 7 (EF Core 的舊稱) 的開發環境，在 Project K 中一併開發，2015 年微軟將 Project K 定名為.NET Execution Environment，縮寫 DNX，其工具也由 K 改為 DNX，但工具的種類沒有太大變化，其實這時候就可以看出微軟賦與 ASP.NET Core 的定位，不只是跨平台，還要相容於 Mac 以及 Linux 的開發人員的習慣 (這兩個平台大多習慣使用指令)，對 Windows 環境的開發人員來說可能要花點時間重新適應這個改變。接著在 ASP.NET Core RC1 發行後，由於原本的 ASP.NET 5 的名稱可能會誤導產業界，認為 ASP.NET 5 是 ASP.NET 4.x 的升級版，因此微軟

決定將它改名為 ASP.NET Core，與.NET Core 的名稱同步，Entity Framework 7 也改名為 Entity Framework Core。

ASP.NET Core 核心設計上是採用 Open Web Interface for .NET (OWIN) 為概念發展，OWIN 強調以程式碼來定義系統功能，並一度在 ASP.NET MVC 5 列入其功能之一，後續的 Web API 與 SignalR 也使用了 OWIN，但並沒有引起太多開發人員的重視，其主因還是因為 Visual Studio 簡化了太多元件間參考定義的工作，若是要回歸由原始碼作業，反而會讓開發人員無法適應。但隨著微軟確定將 ASP.NET Core 開發為可跨平台的核心架構時，其專案參考系統也由 Visual Studio 為主的加入參考對話盒轉向到以 project.json 為主，使得開發人員不能再以 GUI 介面來加入元件參考，只能利用編輯 project.json 的方式加入，這時由程式碼加入功能的作法才慢慢的被開發人員所接受，雖然這在 Mac 以及 Linux 環境是再平常不過的事。

ASP.NET Core 不再使用 Web.config，而是改用 **project.json** 作為專案的主要組態檔[1]，project.json 定義了專案的基本資訊；專案使用的相依套件群 (Dependency Packages)；專案使用的特定作業系統框架套件 (Framework-Specific Packages)；啟動的指令與參數；建造的順序與資源的包含或排除等。

```
{
  "version": "1.0.0",
  "webroot": "wwwroot",
  "exclude": [
    "wwwroot"
  ],
  "dependencies": {
    "Kestrel": "1.0.0-beta4",
    "Microsoft.AspNet.Diagnostics": "1.0.0-beta4",
    "Microsoft.AspNet.Hosting": "1.0.0-beta4",
    "Microsoft.AspNet.Server.IIS": "1.0.0-beta4",
    "Microsoft.AspNet.Server.WebListener": "1.0.0-beta4",
    "Microsoft.AspNet.StaticFiles": "1.0.0-beta4"
  },
  "commands": {
     "web": "Microsoft.AspNet.Hosting --server Microsoft.AspNet.Server.WebListener
           --server.urls http://localhost:5000",
     "kestrel": "Microsoft.AspNet.Hosting --server Kestrel --server.urls
                http://localhost:5001",
     "gen": "Microsoft.Framework.CodeGeneration",
     "ef": "EntityFramework.Commands"
  },
  "frameworks": {
```

[1] https://github.com/aspnet/Home/wiki/Project.json-file

```
    "dnx451": { },
    "dnxcore50": { }
  }
}
```

雖然 ASP.NET Core 不再使用 Web.config，但如果你使用過 Visual Studio 2015 內的 ASP.NET Core 專案範本，或許會注意到裡面還是有 Web.config，不過它只有一個作用，即是在 IIS 環境中註冊 IIS Platform Handler，這是微軟為了能在 IIS 中代管 (Hosting) 原生 Web 平台引擎 (如 Java、node.js、Python 等) 所發展的工具，ASP.NET Core 的執行期使用的是具跨平台能力的 **Kestrel Server** (採用 libuv 開源專案所開發)，未來 IIS 雖然仍可以代管 ASP.NET Core 程式，但實際執行它的會是 Kestrel Server 內的 ASP.NET Core 核心，IIS 只會扮演將要求轉送出去的角色。

ASP.NET Core 也改變了定義功能的方式，採用**「以程式碼定義功能」**的架構，每個 ASP.NET Core 專案都會有一個 Startup.cs 檔，裡面會定義組態來源、程式功能以及需要注入的元件的程式碼。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNet.Builder;
using Microsoft.AspNet.Hosting;
using Microsoft.AspNet.Identity.EntityFramework;
using Microsoft.Data.Entity;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;
using WebApplication16.Models;
using WebApplication16.Services;

namespace WebApplication16
{
   public class Startup
   {
      public Startup(IHostingEnvironment env)
      {
         var builder = new ConfigurationBuilder()
            .AddJsonFile("appsettings.json")
            .AddJsonFile($"appsettings.{env.EnvironmentName}.json", optional: true);

         if (env.IsDevelopment())
         {
            builder.AddUserSecrets();
         }
```

```
        builder.AddEnvironmentVariables();
        Configuration = builder.Build();
    }

    public IConfigurationRoot Configuration { get; set; }

    public void ConfigureServices(IServiceCollection services)
    {
        // Add framework services.
        services.AddEntityFramework()
            .AddSqlServer()
            .AddDbContext<ApplicationDbContext>(options =>
                options.UseSqlServer(Configuration["Data:DefaultConnection
                                                   :ConnectionString"]));

        services.AddIdentity<ApplicationUser, IdentityRole>()
            .AddEntityFrameworkStores<ApplicationDbContext>()
            .AddDefaultTokenProviders();

        services.AddMvc();

        // Add application services.
        services.AddTransient<IEmailSender, AuthMessageSender>();
        services.AddTransient<ISmsSender, AuthMessageSender>();
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env,
                          ILoggerFactory loggerFactory)
    {
        loggerFactory.AddConsole(Configuration.GetSection("Logging"));
        loggerFactory.AddDebug();

        if (env.IsDevelopment())
        {
            app.UseBrowserLink();
            app.UseDeveloperExceptionPage();
            app.UseDatabaseErrorPage();
        }
        else
        {
            app.UseExceptionHandler("/Home/Error");


            try
            {
                using (var serviceScope = app.ApplicationServices.GetRequiredService
                                          <IServiceScopeFactory>()
                    .CreateScope())
                {
                    serviceScope.ServiceProvider.GetService<ApplicationDbContext>()
                        .Database.Migrate();
                }
            }
```

```
            catch { }
        }

        app.UseIISPlatformHandler(options =>
                                  options.AuthenticationDescriptions.Clear());

        app.UseStaticFiles();

        app.UseIdentity();


        app.UseMvc(routes =>
        {
            routes.MapRoute(
                name: "default",
                template: "{controller=Home}/{action=Index}/{id?}");
        });
    }

    // Entry point for the application.
    public static void Main(string[] args) => WebApplication.Run<Startup>(args);
  }
}
```

利用程式定義功能的作法，ASP.NET Core 可以不受到特定平台的限制，只需要在啟動時決定啟用哪些功能，沒有設定的就不使用，可大幅降低因為平台弱點或是誤關保護功能所造成的系統漏洞。只是對長久以來都依賴 GUI 設定平台的 Windows 環境開發人員而言，也是需要時間適應。程式定義功能的概念是源自 OWIN，而實作則是靠 ASP.NET Core 本身的發展的 **Dependency Injection** 架構提供，有了 Dependency Injection，應用程式可以實作自己的邏輯來取代現有的程式，甚至可以完全翻掉原有 MVC 應用程式的架構，並且能和目前主流的 Dependency Injection 服務 (例如 Autofac) 整合，使得 ASP.NET Core 應用程式的擴充能力變得更強，輕量化後的核心也方便應用程式的部署等等。

ASP.NET Core 的所有功能都是用 app.UseXXX()的方式加進應用程式，像是要用 MVC 就是 app.UseMVC()；要用 Browser Link 就用 app.UseBrowserLink()；要用靜態檔案功能就用 app.UseStaticFiles()；要用什麼服務就用 app.UseServices()加進去，這樣的風格和書中講解 ASP.NET Identity 2.0 的程式幾乎一樣。沒錯，這是因為 OWIN 的關係，ASP.NET Core 完全整合了 OWIN 架構，使得整個 ASP.NET Core 的功能都以 OWIN 的作法為主，這麼做的好處是 ASP.NET 不用再受限於 IIS，若是有開發者發展出給 Apache 的 OWIN Framework，那 ASP.NET 就能跑在 Apache Server 上，和 JSP 的 Tomcat 有異曲同工之妙，所以日後在非 IIS 的 Web 伺服器上看到能跑 ASP.NET，也不要太意外☺。

ASP.NET Core 在開發初期所累積的執行環境與引擎的經驗，為後續的.NET Core 開發提供了很好的基礎，**套件化相依管理 (Package-based Dependency Management)** 讓核心不再是一大包安裝檔，應用程式也不必再硬性要求相容於特定大包版本，只要應用程式需要，可自由選擇特定的版本，不需跟隨執行環境的大包版本，因此 ASP.NET Core 以及.NET Core 不會再有 GAC (Global Assembly Cache)，各應用程式可以在建造或是部署時使用.NET Core 的指令工具進行套件還原即可，可有效的降低應用程式部署大小，速度也會更快。

ASP.NET MVC 6 本身的改變並沒有很大，**原先 MVC 5 所用的技能在 MVC 6 基本上都沒什麼問題**，不過 Area 功能由 Routing API 以及[Area]特徵項取代，以往是要新增 Area，不過 MVC 6 只要修改 Routing 設定，在路徑中加入 {area}，以及在要使用 Area 的功能以[Area]標記即可：

```
app.UseMvc(routes =>
{
   routes.MapRoute(
      name: "AreasRoute",
      template: "{area}/{controller}/{action}");

   routes.MapRoute(
      name: "Default",
      template: "{controller}/{action}/{id?}",
      defaults: new { controller = "Home", action = "Index" });
});
```

```
namespace MyApp.Areas.Controllers
{
   [Area("Books")]
   public class HomeController : Controller
   {
      // Books/Home/Index
      public ActionResult Index()
      {
         return View();
      }
   }
}
```

MVC 6 中，Child View 由 **View Component** 取代，並且新增 HtmlHelper.Component 屬性，裡面提供 Invoke()，可呼叫程式中的 View Component：

```
[ViewComponent(Name = "MyViewComponent")]
public class SimpleViewComponent : ViewComponent
{
   public IViewComponentResult Invoke(int num)
   {
```

```
        var message = String.Format("The secret code is: {0}", num);
        return Result.Content(message);
    }
}
```

```
<html lang="en">
<head>
  <meta charset="utf-8" />
  <title>Index Page</title>
</head>
<body>
  <p>@Component.Invoke("MyViewComponent", 42)</p>
</body>
</html>
```

ASP.NET MVC 6 除了 View Component 之外，還新增了 **Tag Helper** 功能，允許開發人員在 View 中定義自己的 Tag，也允許擴充現有的 Tag，例如在`<form>`中加入自己的屬性值，或是編寫一個自己的 `<mytag></mytag>` 等，Tag Helper 可使用 C#來開發，以編譯好的二進位格式執行碼執行，和 Razor Helper 採直譯的方法完全不同，也有助於效能提升。

```
public class EmailTagHelper : TagHelper
{
    private const string EmailDomain = "contoso.com";

    // Can be passed via <email mail-to="..." />.
    // Pascal case gets translated into lower-kebab-case.
    public string MailTo { get; set; }

    public override void Process(TagHelperContext context, TagHelperOutput output)
    {
        output.TagName = "a";    // Replaces <email> with <a> tag

        var address = MailTo + "@" + EmailDomain;
        output.Attributes["href"] = "mailto:" + address;
        output.Content.SetContent(address);
    }
}
```

ASP.NET MVC 6 還有一個很棒的改變，就是**內建的相依注入能力**，Controller 只要在建構式中放置需要的介面，並在 Startup.cs 中註冊要使用的介面與實作對應，接下來的工作就由 ASP.NET Core 處理即可，無須自行呼叫。更棒的是，相依注入不但能用於 Controller，連 View 也可以使用。

```
using System.Linq;
using System.Threading.Tasks;
using ViewInjectSample.Interfaces;
```

```
namespace ViewInjectSample.Model.Services
{
   public class StatisticsService
   {
      private readonly IToDoItemRepository _toDoItemRepository;

      public StatisticsService(IToDoItemRepository toDoItemRepository)
      {
         _toDoItemRepository = toDoItemRepository;
      }

      public async Task<int> GetCount()
      {
         return await Task.FromResult(_toDoItemRepository.List().Count());
      }

      public async Task<int> GetCompletedCount()
      {
         return await Task.FromResult(
           _toDoItemRepository.List().Count(x => x.IsDone));
      }

      public async Task<double> GetAveragePriority()
      {
         if (_toDoItemRepository.List().Count() == 0)
         {
            return 0.0;
         }

         return await Task.FromResult(
           _toDoItemRepository.List().Average(x => x.Priority));
      }
   }
}
```

```
@using System.Threading.Tasks
@using ViewInjectSample.Model.Services
@model ViewInjectSample.Model.Profile
@inject ProfileOptionsService Options
<!DOCTYPE html>
<html>
<head>
   <title>Update Profile</title>
</head>
<body>
<div>
   <h1>Update Profile</h1>
   Name: @Html.TextBoxFor(m => m.Name)
   <br/>
   Gender: @Html.DropDownList("Gender",
         Options.ListGenders().Select(g =>
             new SelectListItem() { Text = g, Value = g }))
   <br/>
```

```
    State: @Html.DropDownListFor(m => m.State.Code,
          Options.ListStates().Select(s =>
              new SelectListItem() { Text = s.Name, Value = s.Code}))
    <br />

    Fav. Color: @Html.DropDownList("FavColor",
          Options.ListColors().Select(c =>
              new SelectListItem() { Text = c, Value = c }))
    </div>
</body>
</html>
```

ASP.NET Core 是一個劃時代的平台改變，它拋棄了以往 ASP.NET 肥大的組件，以 Core CLR 方式輕量化，以 Dependency Injection 讓平台有更大的彈性與擴充能力，以.NET Core 來降低組件相依性，並且盡可能的維持與 MVC 5 等原有版本的相容性，本文只是做個引子，未來 ASP.NET Core 還有更多可看性，且讓我們拭目以待。

Memo

本文所介紹的功能以 ASP.NET Core/.NET Core 於 2016 年 3 月份的版本為主，未來介面與功能仍有可能變化。