



ASP.NET CORE CĂN BẢN

From: Bạch Ngọc Toàn <https://tedu.com.vn>



OCTOBER 29, 2019

MAKE BY: TOAN DAO KIEM

<http://tranvantoanblog.wordpress.com>

Contents

1. Giới thiệu về ASP.NET Core	3
2. Cài đặt và cấu hình môi trường phát triển ASP.NET Core	5
3. Dotnet CLI giao diện dòng lệnh trong ASP.NET Core	8
4. Bắt đầu khởi tạo ứng dụng ASP.NET Core	12
5. Cấu trúc dự án ASP.NET Core.....	16
6. Khởi động ứng dụng trong ASP.NET Core	25
7. Kestrel: Web Server dành cho ASP.NET Core	32
8. Middleware và Request Pipeline trong ASP.NET Core.....	35
9. Làm việc với file cấu hình trong ASP.NET Core	42
10. Sử dụng Static Files trong ASP.NET Core	49
11. MVC Design Pattern trong ASP.NET Core	52
12. Xây dựng ứng dụng ASP.NET Core MVC đầu tiên.....	55
13. Cơ bản về ASP.NET Core Controller	62
14. Cơ chế Routing trong ASP.NET Core	67
15. Attribute Routing trong ASP.NET Core	78
16. Route Constrains trong ASP.NET Core	82
17. Action Selectors & Action Verbs trong ASP.NET Core	87
18. Action Result trong ASP.NET Core	91
19. View trong ASP.NET Core.....	102
20. Razor View Engine trong ASP.NET Core MVC	109
21. Sử dụng Layouts và Section trong ASP.NET Core	119
22. ViewBag và ViewData trong ASP.NET Core.....	124
23. Model và ViewModel trong ASP.NET Core MVC.....	127
24. Truyền dữ liệu từ Controller sang View trong ASP.NET Core	132
25. Xây dựng HTML Form trong ASP.NET Core	136
26. Strongly Typed View trong ASP.NET Core.....	142
27. Tag Helpers trong ASP.NET Core MVC	147
28. Input Tag Helper trong ASP.NET Core.....	152
29. Environment Tag Helper trong ASP.NET Core	158
30. Cơ chế Model Binding: Truyền dữ liệu từ View lên Controller.....	162
31. Model Validation trong ASP.NET Core.....	171
32. Validation Tag Helper trong ASP.NET Core	180

33. Unobtrusive Client Validation trong ASP.NET Core	182
34. Cơ chế Dependency Injection trong ASP.NET Core	185
35. Vòng đời của Dependency Injection: Transient, Singleton và Scoped	193

Học ASP.NET Core căn bản

Nguồn: <https://tedu.com.vn/series/hoc-aspnet-core-can-ban.html>

Loạt bài viết hướng dẫn tự học lập trình ASP.NET Core căn bản cho tất cả mọi người yêu thích lập trình .NET.

1. Giới thiệu về ASP.NET Core

Bài viết này giới thiệu ngắn gọn về ASP.NET Core. ASP.NET Core là framework mới được xây dựng hướng tới tương thích đa nền tảng.

ASP.NET Core là gì?

ASP.NET Core là một tập hợp các thư viện chuẩn như một framework để xây dựng ứng dụng web.

ASP.NET Core không phải là phiên bản tiếp theo của ASP.NET. Nó là một cái tên mới được xây dựng từ đầu. Nó có một sự thay đổi lớn về kiến trúc và kết quả là nó gọn hơn, phân chia module tốt hơn.

ASP.NET Core có thể chạy trên cả .NET Core hoặc full .NET Framework.

What is .Net Core

.NET Core là môi trường thực thi. Nó được thiết kế lại hoàn toàn của .NET Framework. Mục tiêu chính của .NET Core là hỗ trợ phát triển ứng dụng đa nền tảng cho ứng dụng .NET. Nó được hỗ trợ trên Windows, Mac OS và Linux. .NET Core là một framework mã nguồn mở được xây dựng và phát triển bởi Microsoft và cộng đồng .NET trên [Github](#)

.NET Core là một tập con của Full .NET Framework. WebForms, Windows Forms, WPF không phải là một phần của .NET Core.

Nó cũng triển khai đặc điểm của .NET Standard.

.NET Standard là gì?

.NET Standard là một đặc tả chuẩn của .NET API hướng tới hỗ trợ trên tất cả các triển khai của nền tảng .NET. Nó định nghĩa một tập các quy tắc thống nhất cần thiết để hỗ trợ tất cả các ứng dụng trên nền .NET.

Bạn có thể tìm hiểu thêm về [.NET Standard](#) tại đây.

Sự khác nhau giữa .NET Core và .NET Framework

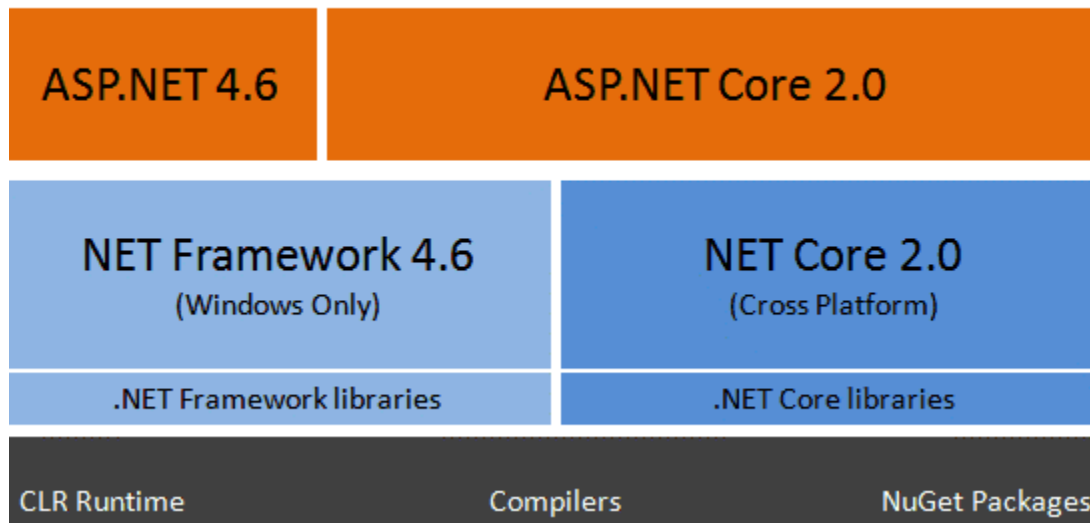
.NET Framework là môi trường cũ hơn và đã tồn tại trên Windows nhiều năm nay. .NET Core hỗ trợ các tính năng trong tập con của .NET Framework. Các tính năng như WebForms, Windows Forms, WPF chưa được đưa lên .NET Core. .NET Framework chỉ chạy trên Windows trong khi .NET Core có thể chạy trên bất cứ nền tảng nào.

ASP.NET Core

Từ Core trong ASP.NET Core rất dễ nhầm. Tên đó gợi nhớ việc ứng dụng ASP.NET Core chỉ có thể được build trên .NET Core, điều đó là sai.

Ứng dụng ASP.NET Core có thể được phát triển sử dụng .NET Core hoặc .NET Framework. Ứng dụng ASP.NET Core được xây dựng sử dụng .NET Core có thể chạy trên bất cứ hệ điều hành nào trong khi nếu được xây dựng trên .NET Framework chỉ có thể chạy trên Windows.

ASP.NET Core in .NET Ecosystem



Các đặc tính quan trọng của ASP.NET Core

Bạn có thể xây dựng và chạy ứng dụng ASP.NET đa nền tảng trên Windows, Mac và Linux (mã nguồn mở và cộng đồng phát triển)

ASP.NET Core hợp nhất ASP.NET MVC và ASP.NET Web API.

Có thể host trên IIS hoặc tự host.

Có sẵn Dependency Injection.

Dễ dàng tích hợp với các framework frontend như Angular, Knockout...

Hỗ trợ cấu hình cho nhiều môi trường.

Cơ chế HTTP Request pipeline mới.

Hỗ trợ quản lý phiên bản

Dùng chung toàn bộ Nuget Package.

Các nhánh của ASP.NET

Có hai nhánh của ASP.NET cho đến hiện tại

ASP.NET

Đây là phiên bản hiện tại của ASP.NET và nó cần .NET Framework để chạy, phiên bản hiện tại hoàn thiện là **4.6**

ASP.NET Core.

Là cách mới để xây dựng ứng dụng web. Nó có thể chạy cả trên .NET Framework và .NET Core. Phiên bản hiện tại đang là 2.2

Khác biệt quan trọng của ASP.NET và ASP.NET Core

ASP.NET	ASP.NET CORE
Phiên bản hiện tại 4.6	Phiên bản hiện tại 2.2
Nền tảng đã có từ lâu	Hoàn toàn được thiết kế mới
Chạy trên .NET Framework	Chạy trên cả .NET Core và .NET Framework
Chỉ trên Windows	Chạy trên tất cả các OS sử dụng .NET Core
Nền tảng ổn định với tính năng phong phú	Chưa hoàn chỉnh nhưng mong đợi sẽ hoàn chỉnh trong tương lai
WebForms được hỗ trợ	Không hỗ trợ WebForms
System.web.dll cồng kềnh	Nhỏ, nhẹ và module hóa
Bản quyền của Microsoft	ASP.NET Core là mã nguồn mở

Điều gì đã xảy ra với ASP.NET 5

ASP.NET Core trước đây có tên là ASP.NET vNext.

Sau đó đổi thành ASP.NET 5

Cuối cùng Microsoft đổi tên thành ASP.NET Core khi release bản 1.0.

2. Cài đặt và cấu hình môi trường phát triển ASP.NET Core

Bài viết này chúng ta sẽ xem qua việc cài đặt và cấu hình môi trường phát triển ASP.NET Core và có thể tiến thẳng đến phần phát triển ứng dụng với ASP.NET Core.

Cài đặt môi trường

Có vài yêu cầu cài đặt trước khi bạn phát triển ứng dụng với ASP.NET Core. Để bắt đầu bạn cần một IDE và Visual Studio 2017 là lựa chọn tốt nhất ở thời điểm hiện tại. Phiên bản Community Edition là miễn phí và bạn có thể sử dụng nó cho việc phát triển ứng dụng ASP.NET Core. Bạn cũng cần cài đặt .NET Core SDK.

Cài đặt Visual Studio 2017

Visual Studio là IDE (Integration Development Environment) là một sự lựa chọn khi bạn phát triển ứng dụng trên Windows.

Khi dùng ASP.NET Core 2.2, bạn cần sử dụng Visual Studio 2017 Update 15.9.12 hoặc cao hơn.

Bạn có thể download Visual Studio 2017 tại đây: <https://www.visualstudio.com/downloads/> nếu phiên bản hiện tại đang là Visual Studio 2019 Preview bạn có thể cài Visual Studio 2019. Hoặc download phiên bản 2017 tại đây: <https://visualstudio.microsoft.com/vs/older-downloads/>. Đến thời điểm hiện tại thì đã có Visual Studio 2019 Preview 2. Khi các bạn xem được bài viết này có thể đã có VS 2019 chính thức các bạn có thể dùng luôn.

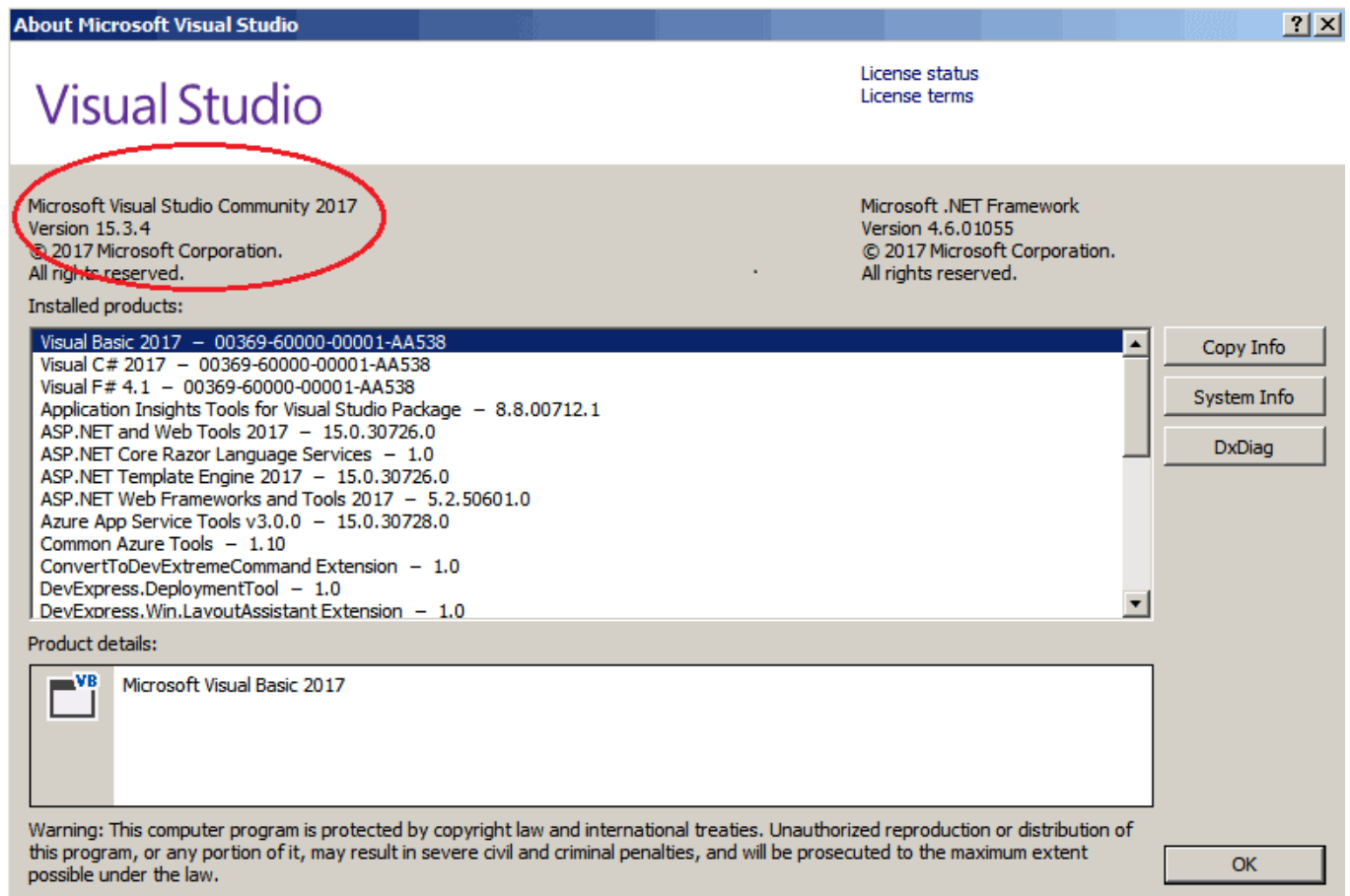
Bạn có thể sử dụng bất cứ phiên bản nào của Visual Studio 2017. Chúng ta sẽ sử dụng bản **Visual Studio Community edition** vì nó miễn phí cho mục đích cá nhân. Bạn có thể đọc thông tin bản quyền tại đây <https://www.visualstudio.com/vs/compare/>

Khi cài đặt Visual Studio chọn ASP.NET Web development Workload

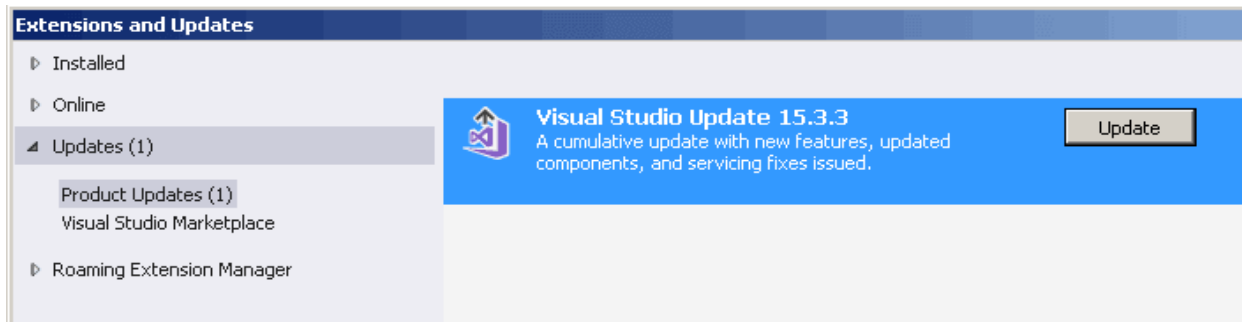
Nâng cấp Visual Studio 2017

Nếu Visual Studio đã có trên máy bạn, bạn có lẽ cần nâng cấp lên bản mới nhất

Bạn có thể kiểm tra phiên bản đang dùng tại menu Help >> About Microsoft Visual Studio trong IDE



Để nâng cấp lên bản mới nhất của Visual Studio bạn cần vào Tools -> Extensions và Updates Menu



Chọn Updates và click vào nút Update để update Visual Studio lên bản mới nhất

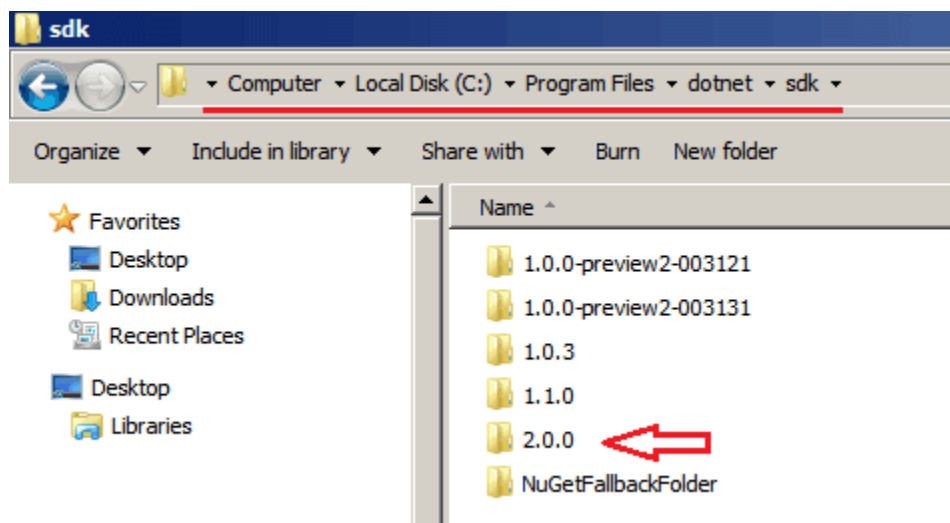
Cài đặt DOT NET Core SDK

Thường thì .NET Core SDK đi kèm Visual Studio nhưng để chắc chắn chúng ta vẫn cài đặt nó và download tại link dưới đây <https://www.microsoft.com/net/download/core>

Làm sao để tìm phiên bản .NET Core đã cài trên PC

Mở thư mục `C:\Program Files\dotnet\sdk` hoặc `C:\Program Files (x86)\dotnet\sdk`. Bạn có thể thấy tất cả phiên bản Dot net Core được cài đặt trên PC.

Bạn có thể kiểm tra phiên bản .NET Core Runtime tại `C:\Program Files\dotnet\shared\Microsoft.NETCore.App`



Bạn cũng có thể xem phiên bản .NET Core được cài đặt qua câu lệnh
`dotnet --version`


```
C:\Windows\system32\cmd.exe
C:\Users\...>
C:\Users\...>dotnet --version
2.0.0
C:\Users\...>
```

Bạn cũng có thể tìm đường dẫn của thư mục cài đặt dotnet core bằng câu lệnh:

```
where dotnet
```

Visual Studio Code

Visual Studio Code là một lựa chọn khác của IDE. Nó là lựa chọn tốt nếu bạn đang dùng MAC hoặc Linux.

Bạn cần cài đặt những thứ sau:

.NET Core SDK 2.2.300 SDK hoặc cao hơn từ <https://www.microsoft.com/net/core>

Visual Studio Code từ <https://code.visualstudio.com>

VS Code C# extension từ <https://marketplace.visualstudio.com/items?itemName=ms-vscode.csharp>

3. Dotnet CLI giao diện dòng lệnh trong ASP.NET Core

Công cụ giao diện dòng lệnh .NET CLI là công cụ mới cho việc phát triển ứng dụng .NET. Nó hỗ trợ đa nền tảng như Windows, MAC hoặc Linux. Trong bài viết này chúng ta sẽ tìm hiểu cách sử dụng .NET CLI để tạo ứng dụng ASP.NET Core đơn giản.

Download Dotnet CLI

Dot Net CLI được cài đặt như một phần của .NET Core SDK. Bạn có thể download tại đây [link](#)

CLI cài đặt mặc định đi kèm và có thể có nhiều phiên bản được cài đặt chung 1 máy.

Sử dụng Dotnet CLI

Cú pháp .NET CLI gồm 3 phần. Từ khóa chính, động từ, và các tham số

dotnet [verb] [arguments]

Tên từ khóa chính là “dotnet”

Động từ là lệnh mà bạn muốn thực thi. Lệnh thực hiện một hành động

Các tham số sẽ được pass vào để cung cấp thêm thông tin cho hành động.

Các lệnh thường dùng

Dưới đây là một số câu lệnh phổ biến thường dùng cho dotnet:

Lệnh	mô tả
new	Tạo mới project, file cấu hình hay solution.
restore	Tải về các thành phần phụ thuộc trong project.
build	Biên dịch dự án với các thành phần liên quan
publish	Đóng gói toàn bộ ứng dụng ra một thư mục.
run	Chạy source code mà không cần lệnh biên dịch.
test	Thực thi unit test
vstest	Thực thi unit test từ file cụ thể
pack	Đóng gói code vào NuGet package.
clean	Làm sạch output của dự án.
sln	Chỉnh sửa file .NET Core solution
help	Xem thêm trợ giúp
store	Lưu trữ các assembly cụ thể

Tạo mới dự án ASP.NET Core sử dụng dotnet CLI

Mở cửa sổ Command Prompt hoặc Windows Powershell và tạo thư mục tên "HelloWorld"

```
dotnet new
```

dotnet new là câu lệnh tạo mới project. Cú pháp như sau:

```
dotnet new <TEMPLATE> [--force] [-i|--install] [-lang|--language] [-n|--name] [-o|--output]
```

Các tham số

TEMPLATE

Tên mẫu dự án cần tạo

--force

Tham số này giúp tạo project ngay cả nếu thư mục đã có một project rồi nó vẫn tạo trên đó.

-i|--install <PATH|NUGET_ID>

Cài đặt một nguồn hoặc một template từ đường dẫn PATH hoặc NUGET_ID

-l|--list

Hiển thị danh sách template chỉ ra tên. Nếu bạn gọi lệnh dotnet new trên thư mục vừa được chạy, nó sẽ list ra các template có sẵn. Ví dụ thư mục hiện tại chứa một project nó sẽ không list ra tất cả các loại project template.

-lang|--language {C#|F#|VB}

Ngôn ngữ của template. Ngôn ngữ được chấp nhận tùy theo template.

-n|--name <OUTPUT_NAME>

Tên được tạo ra, nếu không chỉ ra tên, tên sẽ theo thư mục hiện tại.

-o|--output <OUTPUT_DIRECTORY>

Vị trí project được tạo ra, mặc định là thư mục hiện tại đang chạy câu lệnh

-h|--help

In ra trợ giúp cho câu lệnh

Danh sách các lệnh đầy đủ của **dotnet new** tại đây [here](#)

Tạo mới dự án dùng dotnet new

Câu lệnh dưới đây tạo mới project sử dụng **TEMPLATE**

dotnet new <TEMPLATE>

Bạn có thể xem danh sách template sử dụng câu lệnh:

dotnet new -l

```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows [Version 10.0.17134.677]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Users\toanbachn>dotnet new -l
Usage: new [options]

Options:
-h, --help                Displays help for this command.
-l, --list                Lists templates containing the specified name. If no name is specified, lists all templates.
-n, --name                The name for the output being created. If no name is specified, the name of the current directory is used.
-o, --output              Location to place the generated output.
-i, --install             Installs a source or a template pack.
-u, --uninstall           Uninstalls a source or a template pack.
--nuget-source            Specifies a NuGet source to use during install.
--type                   Filters templates based on available types. Predefined values are "project", "item" or "other".
--dry-run                Displays a summary of what would happen if the given command line were run if it would result in a template creation.
--force                  Forces content to be generated even if it would change existing files.
--lang, --language       Filters templates based on language and specifies the language of the template to create.

Templates
```

Short Name	Language	Tags
Console Application	console	[C#], F#, VB Common/Console
Class library	classlib	[C#], F#, VB Common/Library
Unit Test Project	mstest	[C#], F#, VB Test/MSTest
NUnit 3 Test Project	nunit	[C#], F#, VB Test/NUnit
NUnit 3 Test Item	nunit-test	[C#], F#, VB Test/NUnit
xUnit Test Project	xunit	[C#], F#, VB Test/xUnit
Razor Page	page	[C#] Web/ASP.NET
MVC ViewImports	viewimports	[C#] Web/ASP.NET
MVC ViewStart	viewstart	[C#] Web/ASP.NET
ASP.NET Core Empty	web	[C#], F# Web/Empty
ASP.NET Core Web App (Model-View-Controller)	mvc	[C#], F# Web/MVC
ASP.NET Core Web App	webapp	[C#] Web/MVC/Razor Pages
ASP.NET Core with Angular	angular	[C#] Web/MVC/SPA
ASP.NET Core with React.js	react	[C#] Web/MVC/SPA
ASP.NET Core with React.js and Redux	reactredux	[C#] Web/MVC/SPA
Razor Class Library	razorclasslib	[C#] Web/Razor/Library/Razor Class Library
ASP.NET Core Web API	webapi	[C#], F# Web/WebAPI
global.json file	globaljson	Config
NuGet Config	nugetconfig	Config
Web Config	webconfig	Config
Solution File	sln	Solution

```
C:\Users\toanbachn>
```

Danh sách template

TEMPLATE	mô tả
console	Console Application
classlib	Class library
mstest	Unit Test Project
xunit	xUnit Test Project
web	ASP.NET Core Empty
mvc	ASP.NET Core Web App (Model-View-Controller)
razor	ASP.NET Core Web App
angular	ASP.NET Core với Angular
react	ASP.NET Core với React.js
reactredux	ASP.NET Core với React.js và Redux
webapi	ASP.NET Core Web API

Để tạo một web trống ta sử dụng template **web**

```
dotnet new web
```

Restoring Dependencies với dotnet restore

Khi tạo mới project ta phải download các thành phần liên quan (dependencies) khi đó ta sử dụng dotnet restore

Dotnet restore

Sử dụng **-help** để nhận trợ giúp

```
dotnet restore --help
```

```
C:\Windows\system32\cmd.exe

E:\>md HelloWorld
E:\>Cd HelloWorld
E:\HelloWorld>
E:\HelloWorld>dotnet new web
The template "ASP.NET Core Empty" was created successfully.
This template contains technologies from parties other than Microsoft, see https://aka.ms/template-3pn for details.

Processing post-creation actions...
Running 'dotnet restore' on E:\HelloWorld\HelloWorld.csproj...
  Restoring packages for E:\HelloWorld\HelloWorld.csproj...
  Generating MSBuild file E:\HelloWorld\obj\HelloWorld.csproj.nuget.g.props.
  Generating MSBuild file E:\HelloWorld\obj\HelloWorld.csproj.nuget.g.targets.
  Restore completed in 3.35 sec for E:\HelloWorld\HelloWorld.csproj.

Restore succeeded.

E:\HelloWorld>dotnet restore
Restore completed in 59.07 ms for E:\HelloWorld\HelloWorld.csproj.
E:\HelloWorld>_
```

Chạy ứng dụng sử dụng dotnet run

Sử dụng `dotnet run` để chạy ứng dụng

Mở `localhost:5000/` trên trình duyệt bạn sẽ thấy dòng chữ "Hello World"

Kết luận

Chúng ta đã học cách sử dụng dotnet CLI (command-line interface). Công cụ này hữu dụng trên MAC /Linux nơi mà IDE như Visual Studio không có sẵn.

4. Bắt đầu khởi tạo ứng dụng ASP.NET Core

Bài viết này chúng ta sẽ tìm hiểu làm sao để build một ứng dụng web ASP.NET Core sử dụng Visual Studio 2017. Ứng dụng sẽ sử dụng một template trống và hiển thị dòng chữ "Hello World"

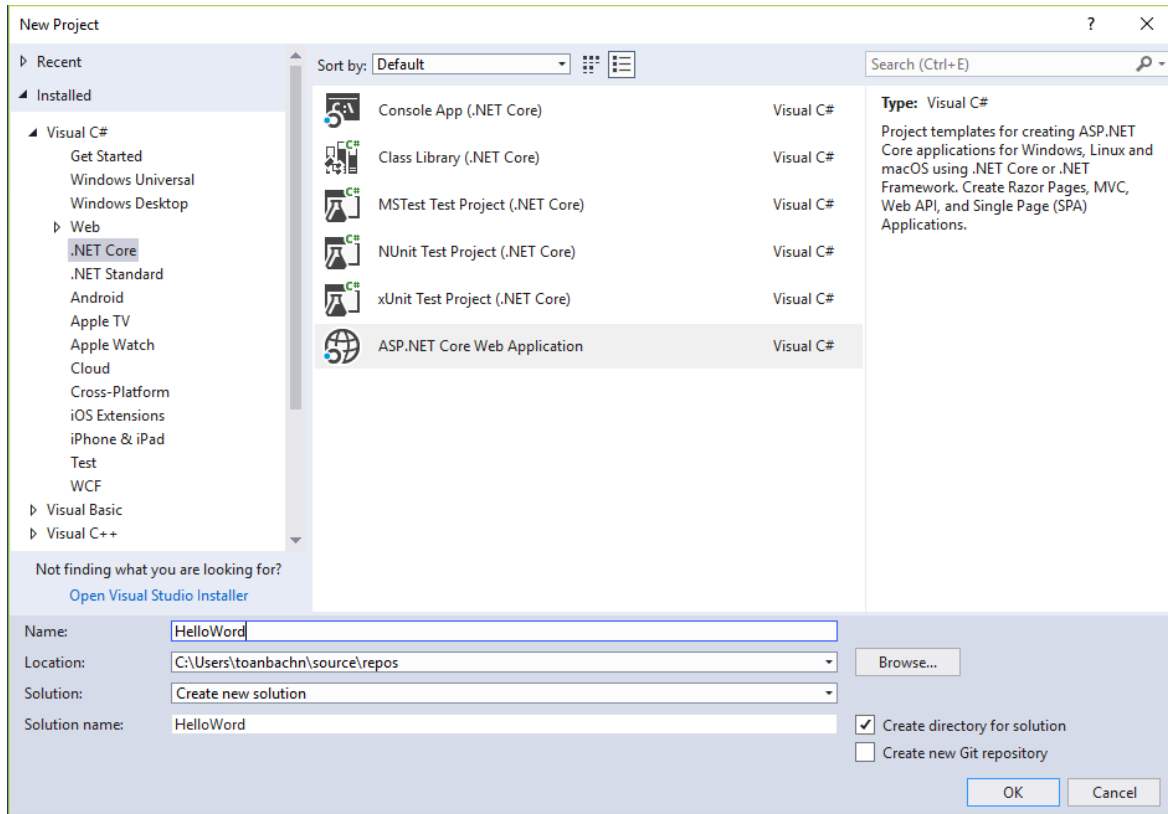
Tạo một ứng dụng

Khởi động Visual Studio 2017. Bạn cần bản 15.9.12 hoặc cao hơn. Nếu bạn chưa có nó có thể quay lại bài [cài đặt môi trường](#):

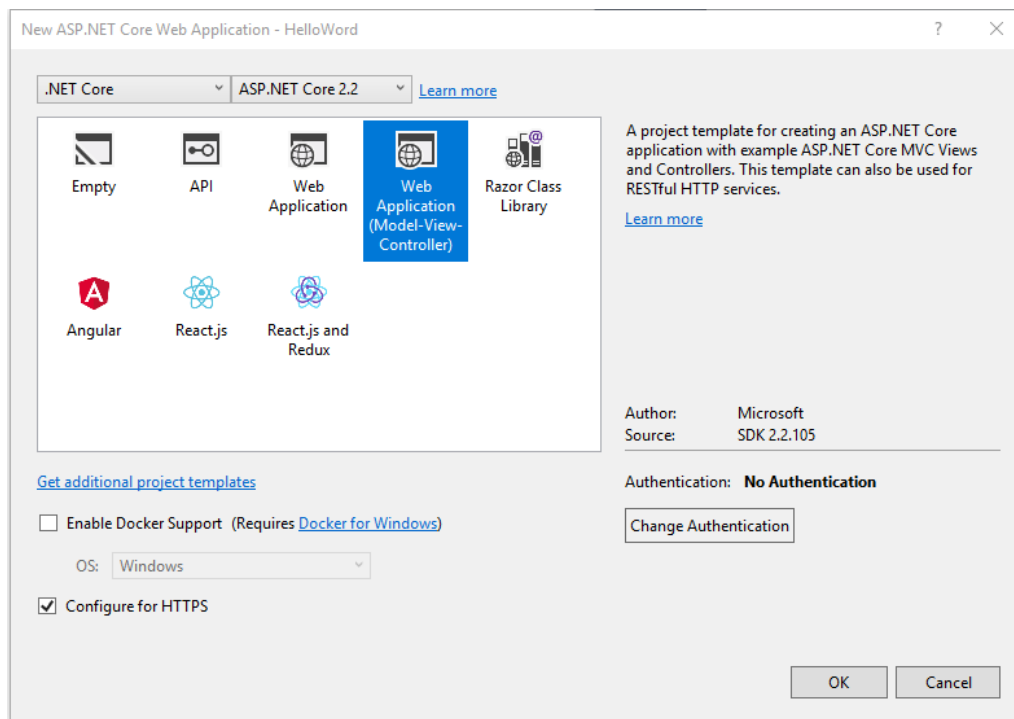
Vào menu Open -> File -> New -> Project

Chọn template ASP.NET Core Web Application. Template này nằm trong Visual C# -> .NET Core

Đặt tên project là `HelloWorld`.



Khi click nút OK, bạn sẽ sang dialog ASP.NET Core Web Application



Ở đây bạn có 1 số lựa chọn

Runtime

Dropdown đầu tiên là chọn môi trường, nó bao gồm 2 tùy chọn **.NET Core** & **.NET Framework**.

.NET Framework là framework cũ có nhiều tính năng nhưng chỉ giới hạn trên Windows

Tùy chọn số 2 là **.NET Core**, cho phép xây dựng ứng dụng chạy đa nền tảng

Chọn **.Net Core**

Phiên bản của ASP.NET

Dropdown 2 cho phép bạn chọn phiên bản của ASP.NET. Chúng ta có 3 lựa chọn ASP.NET Core 2.0, 2.1 và 2.2

Chọn **.NET Core 2.2**

Project Template

Tiếp theo chúng ta cần chọn Project Template. Có vài lựa chọn Angular, ReactJs template.

Chọn Empty

Docker Support

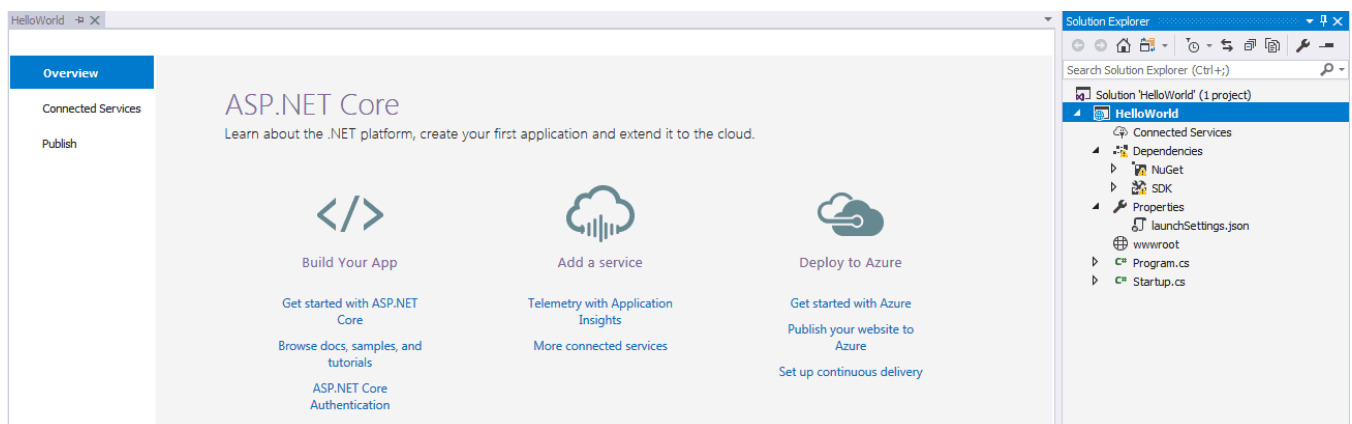
Docker support được thêm vào Visual Studio 2017 cho phép ứng dụng .NET chạy bên trong Docker container.

Tạm thời bỏ check nó đi

Authentication

Chúng ta có thể cài đặt individual, work or school hoặc windows authentication sử dụng tùy chọn này. Với Empty template, chỉ có lựa chọn là No Authentication. Không sao cứ để vậy sau chúng ta thêm Authentication bằng tay.

Click OK để tạo Project. Visual Studio tạo ra một project cơ bản nhỏ gọn nhất



Chạy ứng dụng

Nhấn F5 để chạy ứng dụng. Bạn sẽ thấy dòng chữ “Hello World” trên trình duyệt

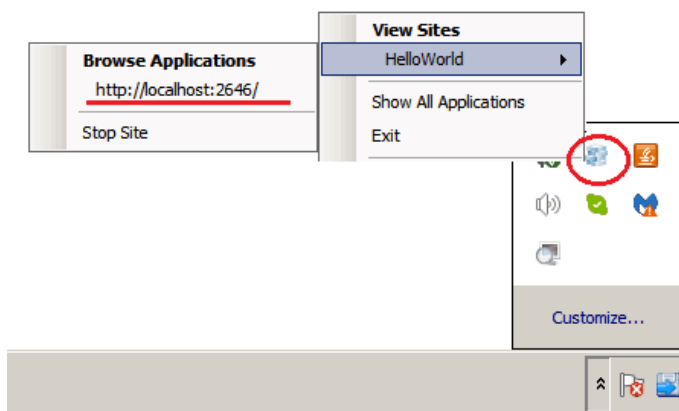
Bấm F5 để chạy ứng dụng ở mode Debug cho phép thay đổi ứng dụng khi nó đang chạy.

Bạn cũng có thể bấm Ctrl-F5, để chạy ứng dụng với mode không debug.

Chạy ứng dụng trên IIS Express

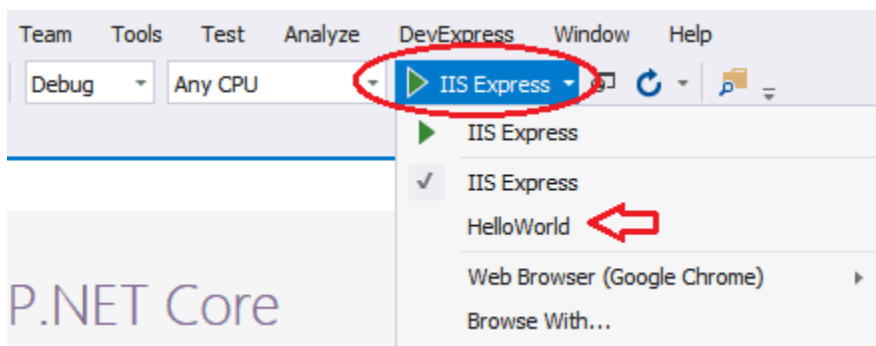
Visual Studio khởi động IIS Express và chạy ứng dụng của bạn. Nó chọn một cổng ngẫu nhiên để thực thi ứng dụng

Biểu tượng của IIS Express xuất hiện ở góc dưới bên phải theo hình dưới đây:



Chạy ứng dụng với dotnet CLI

Mặc định Visual Studio khởi động IIS Express để chạy ứng dụng. Bạn có thể đổi từ IIS Express sang HelloWorld (hoặc tên Project) từ thanh standard



Đổi nó thành HelloWorld và bấm F5. Ứng dụng sẽ chạy bằng Dotnet CLI.

Bạn cũng có thể mở thư mục chứa project (thư mục chứa file HelloWorld.csproj) và chạy lệnh **dotnet run** cũng ra kết quả tương tự

Kết luận

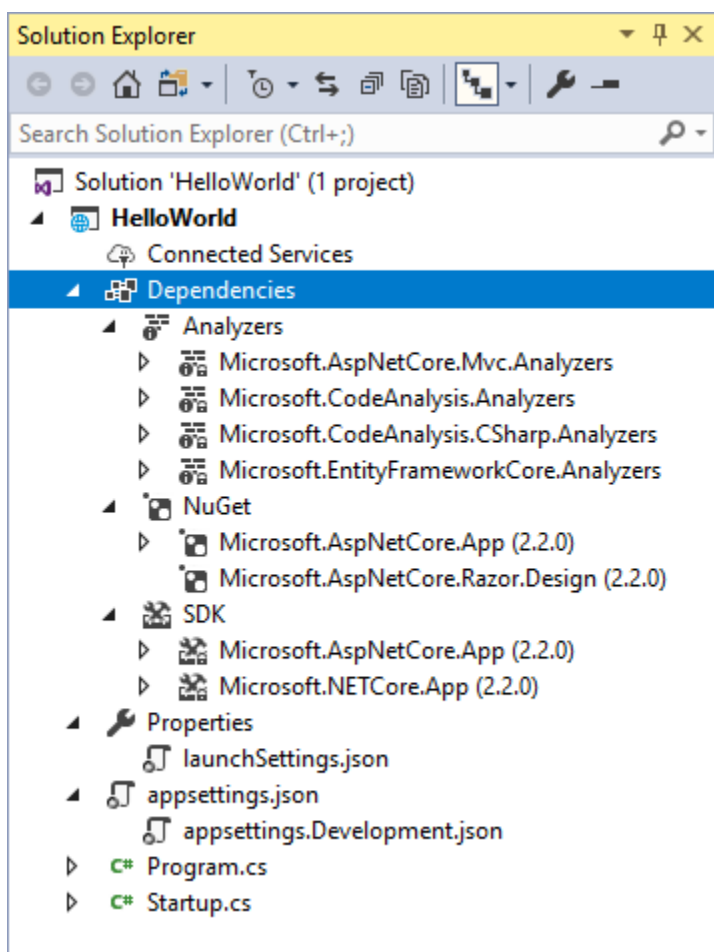
Chúng ta vừa tìm hiểu cách tạo một ứng dụng ASP.NET Core trong Visual Studio. Chúng ta cũng tìm hiểu cách chạy ứng dụng bằng IIS Express hoặc chạy với dotnet cli

5. Cấu trúc dự án ASP.NET Core

Bài viết này chúng ta sẽ tìm hiểu về cấu trúc solution một dự án ASP.NET Core với các thành phần được tổ chức trong đó.

Cấu trúc dự án ASP.NET Core

Sau khi tạo ra một solution ASP.NET Core từ bài trước sử dụng Empty template. Chúng ta sẽ được cấu trúc như sau:

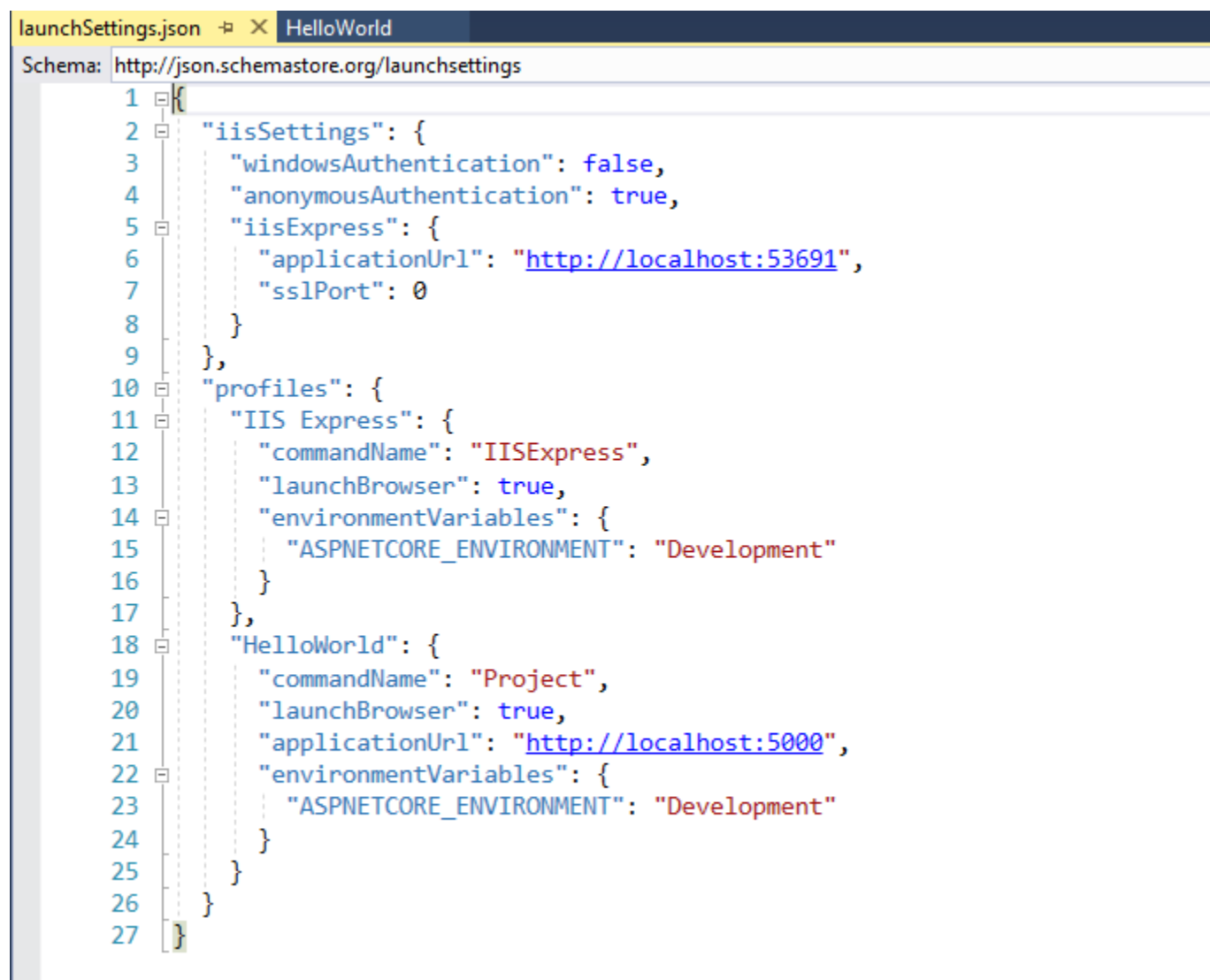


Solution chứa 3 folder là **Dependencies**, **Properties** và **wwwroot**. Thư mục **wwwroot** ở đây không được khởi tạo mặc định nhưng chúng ta có thể thêm mới bằng tay. Ngoài ra nó còn chứa 2 file rất quan trọng là **Program.cs** và **Startup.cs**.

Thư mục Properties

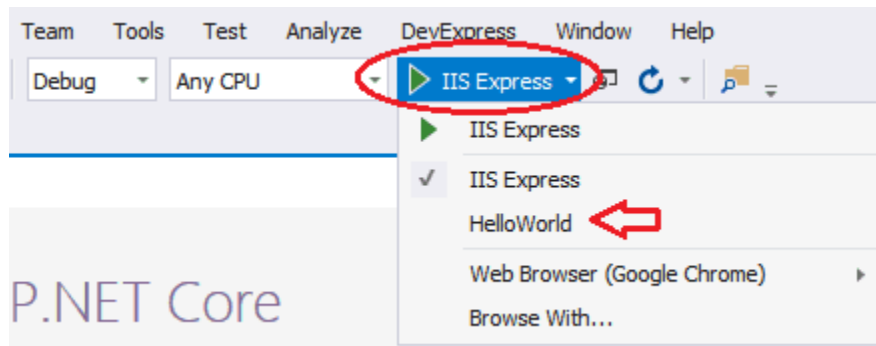
Thư mục Properties chứa một file gọi là **launchSettings.json**. File json này chứa tất cả thông tin cụ thể cài đặt project để có thể chạy được ứng dụng. Bạn sẽ thấy các profile debug, các biến môi trường được chỉ ra trong file này.

Có một điều chúng ta phải nhớ là file này chỉ sử dụng cho môi trường phát triển local của chúng ta thôi, khi publish lên server thì file này không cần thiết. Lúc đó mọi setting sẽ cần đặt trong file **appSettings.json**.



```
launchSettings.json x HelloWorld
Schema: http://json.schemastore.org/launchsettings
1 {
2   "iisSettings": {
3     "windowsAuthentication": false,
4     "anonymousAuthentication": true,
5     "iisExpress": {
6       "applicationUrl": "http://localhost:53691",
7       "sslPort": 0
8     }
9   },
10  "profiles": {
11    "IIS Express": {
12      "commandName": "IISExpress",
13      "launchBrowser": true,
14      "environmentVariables": {
15        "ASPNETCORE_ENVIRONMENT": "Development"
16      }
17    },
18    "HelloWorld": {
19      "commandName": "Project",
20      "launchBrowser": true,
21      "applicationUrl": "http://localhost:5000",
22      "environmentVariables": {
23        "ASPNETCORE_ENVIRONMENT": "Development"
24      }
25    }
26  }
27 }
```

Chúng ta tạm thời bỏ qua section “**iisSettings**” mà quan tâm đến section “**profiles**”. Nó là các profile cài đặt để chúng ta chạy ứng dụng. Ở đây có 2 profile là “**IIS Express**” và “**HelloWorld**” chính là tên của project luôn. Tương ứng với việc chúng ta chạy 2 mode là với IIS Express và Dotnet CLI ở bài trước. Chúng ta có thể xem lại bài trước để chọn chạy ở mode nào ở thanh công cụ của Visual Studio.



Ngoài ra chúng ta cũng thấy cả hai profile này đều có section “**ASPNETCORE_ENVIRONMENT**” được set mặc định là **Development**. Điều này sẽ giúp các bạn phân biệt các môi trường trong khi phát triển, testing và go live sản phẩm. Thường thì có **Development** là môi trường phát triển, **Staging** là môi trường test trước khi golive và **Production** là môi trường chạy thật. Nó giúp chúng ta có thể làm việc đa cấu hình cho các môi trường khác nhau.

wwwroot

Thư mục wwwroot này mới xuất hiện trên ASP.NET Core. Các file tĩnh như HTML, CSS, Javascript hay hình ảnh sẽ đặt trong này hoặc các thư mục con của nó.

Thư mục wwwroot được coi như thư mục root của website. Url của nó trở đến thư mục root sẽ ngay sau domain ví dụ: <http://yourDomainName.com/> sẽ trở đến thư mục **wwwroot**. Trong **wwwroot** ta có file image.png thì đường dẫn của nó sau khi publish sẽ là <http://yourDomainName.com/image.png>.

Tất cả các file được duyệt từ **wwwroot** hoặc folder con của nó. Chỉ đặt các file mà bạn muốn publish ra internet vào thư mục này. Các file trong các thư mục khác sẽ được khóa không thể truy cập từ bên ngoài trừ khi bạn cố tình cấu hình nó.

Toàn bộ code như C#, Razor file nên đặt ngoài thư mục này. Và để dùng được tính năng này chúng ta cần sử dụng tính năng **StaticFiles** trong ASP.NET Core mà chúng ta sẽ tìm hiểu sau.

Dependencies

Thư mục này chứa toàn bộ các thành phần phụ thuộc dùng cho ứng dụng, chúng ta gọi là **Dependencies**. Visual Studio sử dụng Nuget Packages để quản lý tất cả các dependencies phía server. Với Client tức là các thư viện JS thì chúng ta sử dụng **Client-side Libraries**. Đây là sự khác biệt so với phiên bản trước khi mà Nuget quản lý cả dependencies phía server và client.

Trong thư mục **Dependencies**, chúng ta có thư mục Nuget chứa các Nuget Packages đang sử dụng và thư mục SDK chứa **Microsoft.NETCore.App**. Đây là .NET Core Runtime sẽ sử dụng trong project.

Các file trong hệ thống

File project (**.csproj**) đã có nhiều thay đổi từ Visual Studio 2017. Nó đã đơn giản rất nhiều rồi. Có một sự tương quan trực tiếp giữa các file trong thư mục solution và những gì hiển thị trong **Solution Explorer** của Visual Studio. Các file bạn add vào solution folder nó cũng tự động trở thành một phần của project.

Để demo điều này, chúng ta sẽ add một file vào cấu trúc dự án trong File Explorer của Windows và xem xem chúng sẽ hiển thị trong cửa sổ Solution Explorer ngay lập tức ra sao.

Mở notepad và dán đoạn code này vào:

```
namespace HelloWorld
```

```
{
```

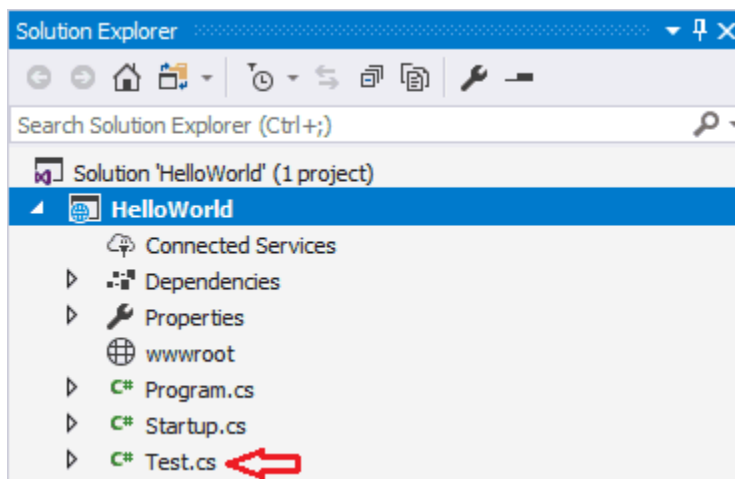
```
public class test
```

```
{
```

```
}
```

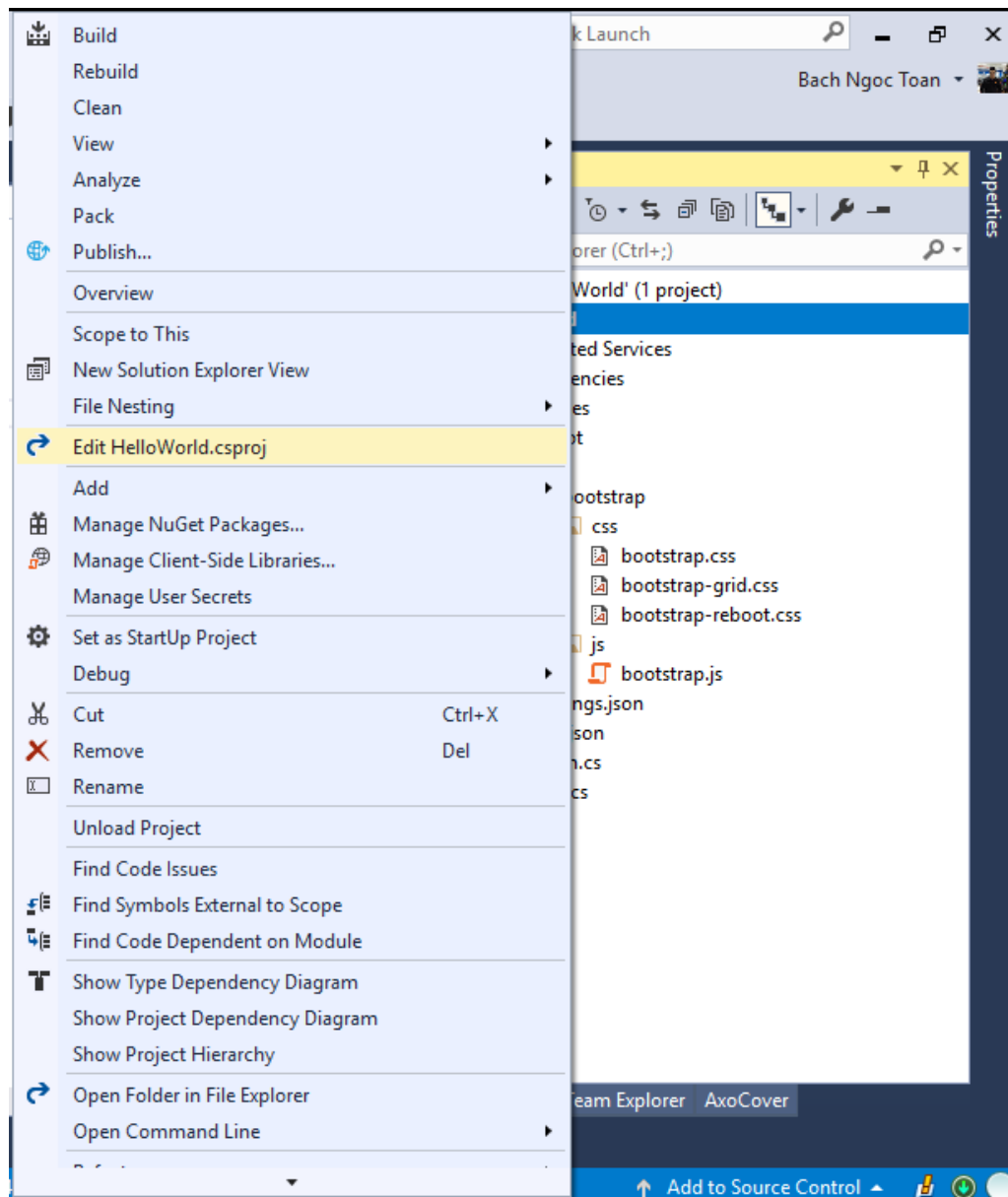
```
}
```

Sau đó lưu lại vào thư mục HelloWorld. File này sẽ tự động được thêm vào project và hiển thị lên.



Hệ thống Project mới

File project.json trong ASP.NET 1.0 đã bỏ đi từ 2.0. Các gói Nuget Packages sau khi cài đặt sẽ được liệt kê trong file .csproj. File .csproj có thể được mở trực tiếp từ Visual Studio. Chúng ta có thể click chuột phải vào project và chọn Edit .csproj file. Không cần phải unload project sau khi làm điều này.



File Project mới chứa một số các phần tử:

```
<Project Sdk="Microsoft.NET.Sdk.Web">
```

```
<PropertyGroup>
```

```
<TargetFramework>netcoreapp2.2</TargetFramework>
```

```
<AspNetCoreHostingModel>InProcess</AspNetCoreHostingModel>
```

```
</PropertyGroup>
```

```
<ItemGroup>
```

```
<PackageReference Include="Microsoft.AspNetCore.App" />
```

```
<PackageReference Include="Microsoft.AspNetCore.Razor.Design"  
Version="2.2.0" PrivateAssets="All" />
```

```
</ItemGroup>
```

```
</Project>
```

Trong đó có thể thấy TargetFramework đang là **netcoreapp2.2**. Phần tử ItemGroup sẽ chứa danh sách các Nuget Package, mặc định nó có **Microsoft.AspNetCore.App** và **Microsoft.AspNetCore.Razor.Design**. Còn thành phần **AspNetCoreHostingModel** chỉ ra cấu hình hosting, vấn đề này rất dài các bạn có thể tham khảo tại đây: <https://docs.microsoft.com/en-us/aspnet/core/host-and-deploy/aspnet-core-module?view=aspnetcore-2.2>

Bạn có thể update các giá trị trong file này sau đó save lại để thấy sự thay đổi. Như **TargetFramework** khi bạn muốn upgrade phiên bản .NET Core. Tương lai gần sẽ là .NET Core 3.0. Hay các bạn thêm các gói Nuget Packages mà không dùng Nuget Package Manage Tool mà thêm bằng tay qua đây. Với điều kiện phải chỉ đúng tên gói và phiên bản. Sau khi save xong thì Visual Studio sẽ tự tải về package cho bạn.

Nuget Packages

Có 3 cách để tải Nuget Package vào project:

Chỉnh sửa thẳng file .csproj

Sử dụng Nuget Package Manager

Qua lệnh Package Manager Console

Chỉnh sửa file .csproj

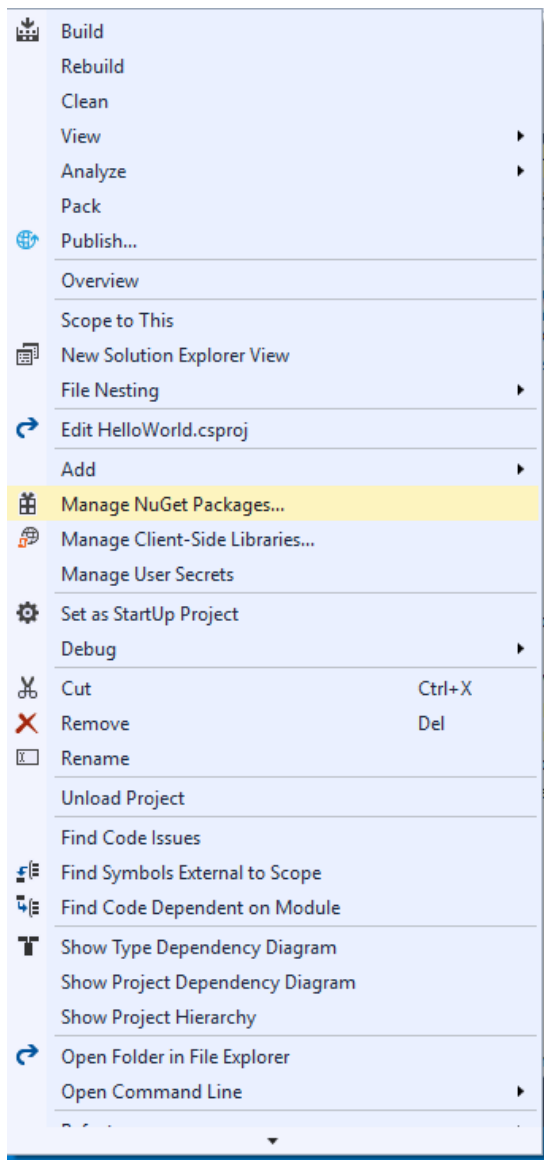
Chọn project và bấm chuột phải sau đó chọn Edit HelloWorld.csproj. Thêm đoạn code dưới đây vào phần ItemGroup và save lại.

```
<PackageReference Include="Microsoft.ApplicationInsights" Version="2.4.0" />
```



Nuget Package Manager sẽ tự download và cài đặt gói này về ngay sau khi bạn save và sẽ hiển thị ở ở mục Nuget. Dấu cảnh báo vàng sẽ tự mất khi nó được tải xong.

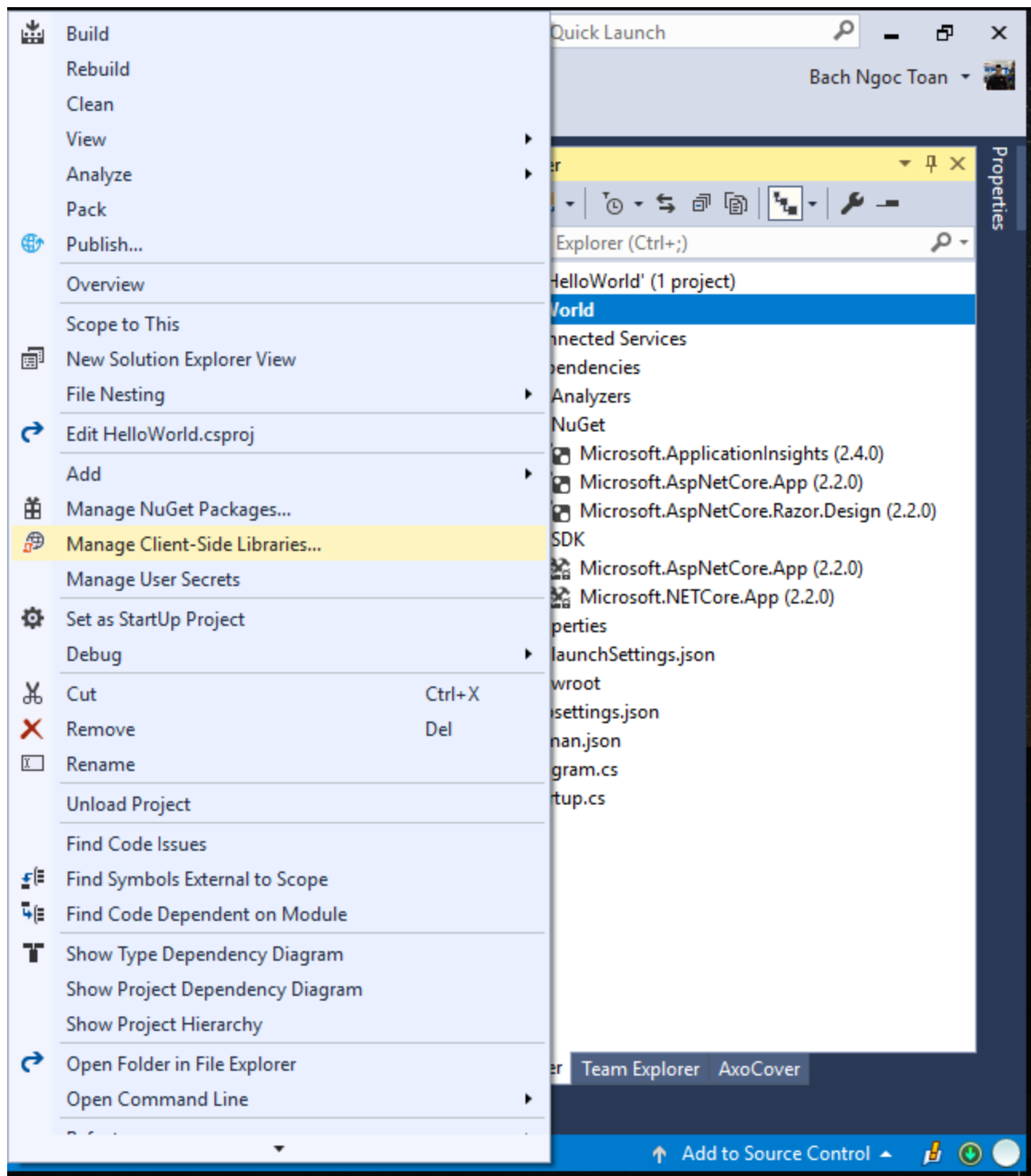
Sử dụng Nuget Package Manager



Chuột phải project chọn **Manage Nuget Packages** sau đó bạn có thể cài mới hay xóa bỏ các Nuget Packages bằng giao diện đơn giản. Nó cũng sẽ tự động update trong file .csproj.

Bower Packages

Chuột phải vào project và chọn **Manage Client-side Libraries** nó sẽ tạo ra file libman.json nơi bạn có thể thêm hay xóa các thư viện phía client. Nó sẽ tự tạo folder theo thuộc tính destination của từng thư viện.



Cái này rất hữu ích khi ứng dụng web chúng ta có các thư viện client như bootstrap, jquery...và chúng ta muốn theo dõi danh sách các gói, phiên bản của chúng. Ngoài ra có thể sử dụng các cách khác như NPM hay Yarn đều được. Mình thích dùng cách này vì nó có sẵn trong Visual Studio.

6. Khởi động ứng dụng trong ASP.NET Core

Trong bài viết này chúng ta sẽ tìm hiểu về 2 file Startup.cs và Program.cs. Program.cs tạo một webserver với phương thức Main còn Startup.cs chứa cấu hình các services và request pipeline trong ứng dụng.

Program.cs

Tất cả các ứng dụng .NET Core đều là các ứng dụng Console. Các kiểu ứng dụng khác nhau như là MVC, SPA...đều được xây dựng dựa trên ứng dụng console.

Ứng dụng Console luôn được chạy từ một file là Program.cs và phải chứa một hàm static void Main. Nó luôn được gọi khi ứng dụng khởi động.

Mở ứng dụng ở bài trước ra chúng ta sẽ xem file Program.cs. Class này chỉ có một method là Main

```
using System;

using System.Collections.Generic;

using System.IO;

using System.Linq;

using System.Threading.Tasks;

using Microsoft.AspNetCore;

using Microsoft.AspNetCore.Hosting;

using Microsoft.Extensions.Configuration;

using Microsoft.Extensions.Logging;


namespace HelloWorld
{
    public class Program
    {
        public static void Main(string[] args)
        {
            BuildWebHost(args).Run();
        }

        public static IWebHost BuildWebHost(string[] args) =>
            WebHost.CreateDefaultBuilder(args)
```

```

        .UseStartup<Startup>()

        .Build();
    }
}

```

Đây là điểm khởi đầu của ứng dụng (entry point). Phương thức chính này sẽ tạo một Host, build và run nó. Host này là một web server sẽ lắng nghe các HTTP Request.

```

using System.Threading.Tasks;
using Microsoft.AspNetCore;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Logging;

namespace HelloWorld
{
    public class Program
    {
        public static void Main(string[] args)
        {
            BuildWebHost(args).Run();
        }

        public static IWebHost BuildWebHost(string[] args) =>
        {
            WebHost.CreateDefaultBuilder(args)
                .UseStartup<Startup>()
                .Build();
        }
    }
}

```

3. Run it

1. Create a Host

2. Build it

Tạo một host

BuildWebHost

Phương thức static BuildWebHost là một phương thức tĩnh, nó cấu hình, build và trả về một đối tượng Host.

```

public static IWebHost BuildWebHost(string[] args) =>
{
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .Build();
}

```

Cú pháp này được biết đến là Expression-Bodied Function Members trong C# 6. Nó tương đương với function đầy đủ như sau:

```
public static IWebHost BuildWebHost(string[] args) {
    Return WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .Build();
}
```

Phương thức trên sử dụng class `WebHost`. Method **CreateDefaultBuilder** của class **WebHost** có trách nhiệm khởi tạo thể hiện **WebHostBuilder** với các cấu hình được yêu cầu.

CreateDefaultBuilder

Các công việc được thực hiện bởi **CreateDefaultBuilder** là:

Cấu hình Kestrel như là web server

Đặt thư mục gốc của ứng dụng sử dụng **Directory.GetCurrentDirectory()**

Load cấu hình từ

Appsettings.json

Appsettings.{Environment}.json

Load các cấu hình riêng (user secrets) khi chạy ứng dụng ở môi trường Development

Load các biến môi trường (Environment Variables)

Đọc các tham số truyền vào ứng dụng qua Command line argument

Bật logging

Tích hợp **Kestrel** chạy với IIS

Bạn có thể xem [mã nguồn](#) mở đầy đủ của WebHost tại đây:

```
public static IWebHostBuilder CreateDefaultBuilder(string[] args)
{
    var builder = new WebHostBuilder()
        .UseKestrel()
        .UseContentRoot(Directory.GetCurrentDirectory())
        .ConfigureAppConfiguration((hostingContext, config) =>
        {
            var env = hostingContext.HostingEnvironment;

            config.AddJsonFile("appsettings.json", optional: true,
reloadOnChange: true)
```

```

        .AddJsonFile($"appsettings.{env.EnvironmentName}.json",
optional: true, reloadOnChange: true);

        if (env.IsDevelopment())
        {
            var appAssembly = Assembly.Load(new
AssemblyName(env.ApplicationName));

            if (appAssembly != null)
            {
                config.AddUserSecrets(appAssembly, optional: true);
            }
        }

        config.AddEnvironmentVariables();

        if (args != null)
        {
            config.AddCommandLine(args);
        }
    })
    .ConfigureLogging((hostingContext, logging) =>
    {
        logging.AddConfiguration(hostingContext.Configuration.GetSection("Logging"));

        logging.AddConsole();
        logging.AddDebug();
    })
    .UseIISIntegration()
    .UseDefaultServiceProvider((context, options) =>
    {
        options.ValidateScopes =
context.HostingEnvironment.IsDevelopment();
    });

```

```
return builder;
```

```
}
```

Bạn có thể xem mã nguồn hàm đầy đủ các tính năng đó ở trên đây.

Kestrel

```
.UseKestrel()
```

Dòng code này thể hiện việc host sử dụng Kestrel web server. Kestrel là một HTTP web server lắng nghe các HTTP request. Server này cho phép ứng dụng của bạn chạy trên các hệ điều hành khác ngoài Windows.

Content Root

```
.UseContentRoot(Directory.GetCurrentDirectory())
```

Dòng này chỉ ra thư mục gốc của ứng dụng là thư mục hiện tại.

Load các file cấu hình

Load dữ liệu các file cấu hình từ:

Appsettings.json

Appsettings.{Environment}.json

Sử dụng các key cấu hình cho môi trường Development (user secrets)

Các biến môi trường (environment variable)

Command line arguments

```
.ConfigureAppConfiguration((hostingContext, config) =>
{
    var env = hostingContext.HostingEnvironment;

    config.AddJsonFile("appsettings.json", optional: true, reloadOnChange: true)
        .AddJsonFile($"appsettings.{env.EnvironmentName}.json", optional:
true, reloadOnChange: true);

    if (env.IsDevelopment())
    {
        var appAssembly = Assembly.Load(new AssemblyName(env.ApplicationName));

        if (appAssembly != null)
        {
            config.AddUserSecrets(appAssembly, optional: true);
        }
    }
}
```

```

    }
    config.AddEnvironmentVariables();
    if (args != null)
    {
        config.AddCommandLine(args);
    }
})

```

Bật logging

```

.ConfigureLogging((hostingContext, logging) =>
{
    logging.AddConfiguration(hostingContext.Configuration.GetSection("Logging"));
    logging.AddConsole();
    logging.AddDebug();
})

```

Đoạn này đọc cấu hình phần logging từ các file cấu hình cho việc xuất thông tin log ra console và ra cửa sổ debug.

Sử dụng tích hợp IIS

```

.UseIISIntegration()

```

Phương thức này tích hợp Kestrel với IIS

Default Service Provider

Đoạn này cài đặt DI Container có sẵn và bạn có thể tùy biến các hành động của nó

```

.UseDefaultServiceProvider((context, options) =>
{
    options.ValidateScopes = context.HostingEnvironment.IsDevelopment();
});

```

Và cuối cùng trả về một đối tượng builder.

```

return builder;

```

Build

Phương thức build của **WebHostBuilder** build lên một đối tượng **WebHost** để chạy ứng dụng

Run

Phương thức run của **WebHostBuilder** sẽ khởi động Host và bắt đầu lắng nghe HTTP Request.

Startup.cs

Chúng ta hãy nhìn dòng cấu hình của phần Program.cs. WebHost sẽ gọi phương thức UseStartup trước khi build lên host:

```
.UseStartup<Startup>()
```

Phương thức UseStartup nói cho Host biết là class nào sẽ gọi khi start host. Và Host sẽ tìm kiếm phần cài đặt trong 2 phương thức Configure và ConfigureServices.

Startup Class là gì?

Startup class là một class đơn giản không kế thừa hay triển khai bất cứ class hay interface nào. Nó có hai chức năng chính:

Cấu hình request pipeline để xử lý các request gửi đến ứng dụng.

Cấu hình các services cho phần dependency injection

ConfigureServices

Phương thức ConfigureServices cho phép chúng ta thêm hoặc đăng ký các service vào ứng dụng. Phần khác của ứng dụng có thể cần đến các service này cho dependency injection. Dependency Injection là một tính năng mới trong ASP.NET Core. Nó có rất nhiều lợi ích mà chúng ta sẽ có bài riêng để nói về nó:

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

Phương thức ConfigureServices cần các thể hiện của các service. Thể hiện của các service sẽ được inject (tiêm) vào phương thức ConfigureServices thông qua Dependency Injection.

Configure

Phương thức Configure cho phép bạn cấu hình HTTP Request Pipeline. HTTP Request Pipeline chỉ ra cách mà ứng dụng cần phải hồi đáp các HTTP Request. Các thành phần tạo nên request pipeline được gọi là middleware.

Phương thức Configure sẽ có dạng như sau:

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)  
{  
    if (env.IsDevelopment())  
    {  
        app.UseDeveloperExceptionPage();  
    }  
  
    app.Run(async (context) =>
```



```
{
    await context.Response.WriteAsync("Hello World!");
}
}
```

Phương thức configure cần thể hiện của `IApplicationBuilder` và `IHostingEnvironment`. Hai thể hiện này sẽ được inject vào `Configure` thông qua `Dependency Injector`.

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
```

Chúng ta sẽ thêm các middleware vào thể hiện của `IApplicationBuilder`. Đoạn tiếp theo sẽ kiểm tra nếu chúng ta đang ở môi trường `Development`, nếu có thì sẽ đăng ký middleware `DeveloperExceptionPage` sử dụng extension method `UseDeveloperExceptionPage`.

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

Tiếp theo chúng ta sẽ sử dụng method của ứng dụng để đăng ký middleware thứ 2. Middleware này sẽ ghi ra dòng chữ "Hello World" ra đối tượng response.

7. Kestrel: Web Server dành cho ASP.NET Core

Trong bài viết này chúng ta sẽ tìm hiểu về Kestrel, một web server dành cho ASP.NET Core. Cách mà chúng ta host ứng dụng trong ASP.NET Core đã thay đổi so với phiên bản ASP.NET cũ. Kestrel là cách mới để host ứng dụng ASP.NET Core. Nó chạy ứng dụng độc lập và hoàn toàn có thể tự host được.

Kestrel là gì?

Kestrel là một HTTP web server mã nguồn mở (open source), đa nền tảng (cross-platform), hướng sự kiện (event-driven) và bất đồng bộ (asynchronous I/O). Nó được phát triển để chạy ứng dụng ASP.NET Core trên bất cứ nền tảng nào. Nó được thêm vào mặc định trong ứng dụng ASP.NET Core.

Nó dựa trên [Luv](#), và Kestrel là một mã nguồn mở trên [Github](#)

Tại sao sử dụng Kestrel

Ứng dụng ASP.NET cũ thường dính chặt vào **IIS** (Internet Information Service). IIS là một web server với tất cả các tính năng đầy đủ cần có. Nó được phát triển từ khá lâu và rất trưởng thành,

nhưng nó cồng kềnh và nặng. Nó trở thành một trong những Web server tốt nhất ở thời điểm hiện tại nhưng nó cũng là một trong những thứ chậm nhất.

ASP.NET dính chặt vào IIS cũng là gánh nặng cho IIS.

Thiết kế mới của ứng dụng ASP.NET Core giờ đây hoàn toàn tách rời khỏi IIS. Điều này tạo cho ASP.NET Core có thể chạy trên bất cứ nền tảng nào. Nhưng nó vẫn có thể lắng nghe các HTTP Request và gửi lại response về cho client. Đó là Kestrel.

Sử dụng Kestrel

Kestrel chạy in-process trong ứng dụng ASP.NET Core. Vì thế nó chạy độc lập với môi trường. Kestrel web server nằm trong thư viện Microsoft.AspNetCore.Server.Kestrel.

Chúng ta hãy xem 2 class Program và Startup được giới thiệu trong bài: [Khởi động ứng dụng trong ASP.NET Core](#). Program class chứa một phương thức Main và là điểm khởi động của ứng dụng.

```
public class Program
{
    public static void Main(string[] args)
    {
        BuildWebHost(args).Run();
    }

    public static IWebHost BuildWebHost(string[] args) =>
    {
        WebHost.CreateDefaultBuilder(args)
            .UseStartup<Startup>()
            .Build();
    }
}
```

Phương thức **Main** gọi đến **CreateDefaultBuilder**, có trách nhiệm tạo một host cho ứng dụng. (Host là nơi chứa ứng dụng để chạy). **CreateDefaultBuilder** đăng ký **Kestrel** như là server sẽ sử dụng trong ứng dụng.

Có 2 cách để sử dụng Kestrel

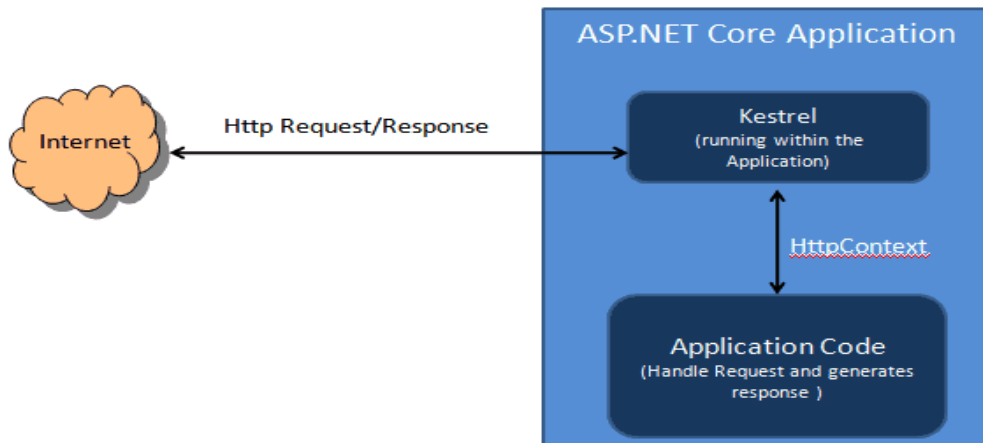
Tự host (Self Hosting)

Đăng sau một Web server khác

Tự host (Self Hosting)

Mô hình self hosting của ASP.NET Core là lắng nghe trực tiếp các HTTP Request từ internet như hình dưới đây:

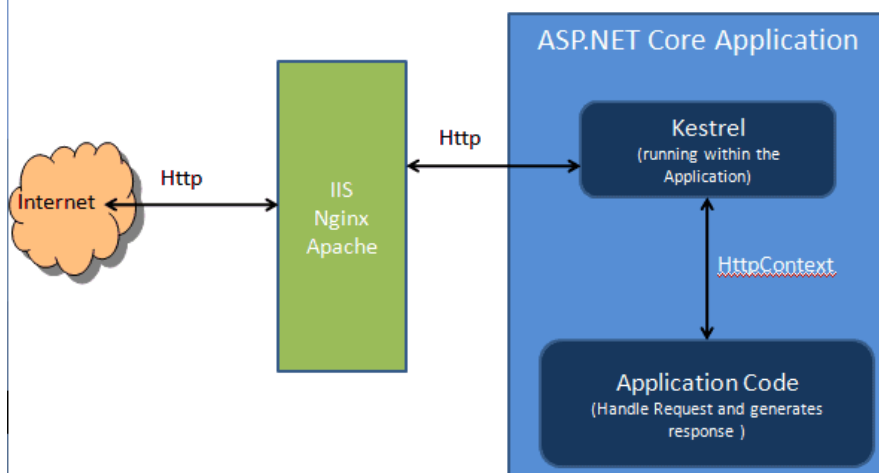
Self Hosting of ASP.NET Core Application



Đăng sau Web Server khác

Kestrel không phải là một web server đầy đủ tính năng. Nhưng nó nhanh. Nó không đủ mạnh để chạy Kestrel một mình trên môi trường **Production**. Nó được đề xuất chạy đằng sau một Web Server Fully Feature như IIS, Nginx hay Apache... Trong kịch bản này các web server khác như là một **reverse proxy server** (máy chủ ủy quyền). Các Reverse proxy server sẽ lấy HTTP Request từ internet và gửi sang Kestrel. IIS có thể lấy HTTP Request và thực hiện một số các xử lý như ghi log, lọc request, rewrite-url trước khi gửi sang cho Kestrel.

Hosting Model of ASP.NET Core Application



Có nhiều lý do tại sao chúng ta sử dụng mô hình này cho môi trường Production.

Bảo mật: Nó có thể hạn chế việc phơi ra ngoài những gì không cần thiết và nó cũng cho chúng ta các tùy chọn về cấu hình và bảo vệ ứng dụng.

Dễ dàng cài đặt cân bằng tải

Cài đặt SSL: Chỉ các máy chủ ủy quyền cần các chứng chỉ SSL, còn các server này có thể làm việc với ứng dụng của bạn trên mạng nội bộ sử dụng HTTP thuần.

Chia sẻ một IP với nhiều địa chỉ

Lọc các Request, ghi log và rewrite-URL...

Nó có thể restart ứng dụng nếu bị crash.

Phương thức CreateDefaultBuilder gọi UseIISIntegration, sẽ nói co ASP.NET rằng ứng dụng sử dụng IIS như là một reverse proxy server đứng trước Kestrel.

Thay thế cho Kestrel

Kestrel không phải là cách duy nhất để host ứng dụng ASP.NET Core. Có một web server khác triển khai có sẵn trên Windows là **HTTP.SYS**.

HTTP.sys là một HTTP Server dựa trên [Http.Sys kernel driver](#) và nó chỉ sử dụng trên Windows.

Các bạn có thể tìm hiểu thêm tại đây: [Cách sử dụng HTTP SYS](#)

Kết luận

Bài viết này chúng ta sẽ học về Kestrel server. Một In-process web server để chạy ASP.NET Core trên đa nền tảng.

8. Middleware và Request Pipeline trong ASP.NET Core

Bài viết này chúng ta sẽ học về Middleware và cách mà Request Pipeline làm việc với ứng dụng ASP.NET Core.

Request Pipeline

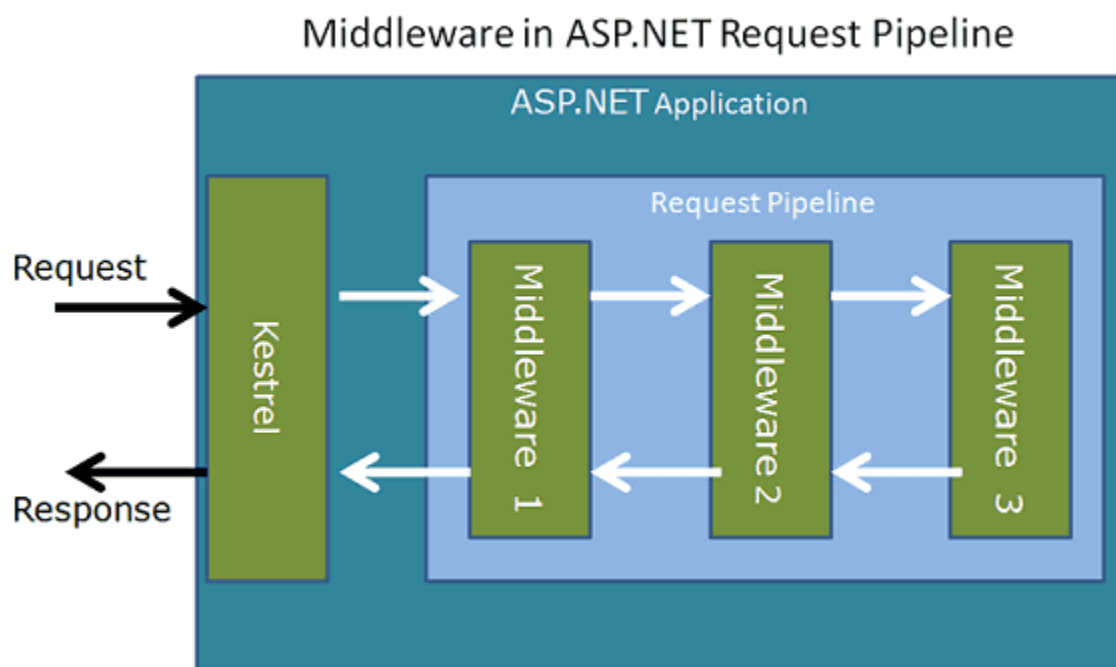
Request Pipeline là cơ chế bắt đầu khi các request bắt đầu được xử lý với một đối tượng Request đầu vào và kết thúc với đầu ra là một response. Pipeline chỉ ra cách mà ứng dụng phản hồi với HTTP Request. Request đến từ trình duyệt đi qua pipeline và quay trở lại khi xử lý xong để trả về client. Các thành phần đơn lẻ tạo nên pipeline này được gọi là middleware.

Middleware là gì?

Middleware là thành phần của phần mềm đóng vai trò tác động vào request pipeline (luồng request) để xử lý chúng và tạo ra response phản hồi lại client. Mỗi một tiến trình middleware thao tác với các request nhận được từ middleware trước nó. Nó cũng có thể quyết định gọi middleware tiếp theo trong pipeline hoặc trả về response cho middleware ngay trước nó. (ngắt pipeline).

Chúng làm việc ra sao

Hình dưới đây cho ta thấy Request Pipeline kết hợp với middleware làm việc trong ứng dụng ASP.NET Core.



Đầu tiên, HTTP Request đến (trực tiếp hoặc qua External web server) ứng dụng. Kestrel web server nhận lấy request và tạo một HttpContext và gán nó vào Middleware đầu tiên trong request pipeline.

Middleware đầu tiên sẽ nhận request, xử lý và gán nó cho middleware tiếp theo. Quá trình này tiếp diễn cho đến khi đi đến middleware cuối cùng. Tùy thuộc bạn muốn pipeline của bạn có bao nhiêu middleware.

Middleware cuối cùng sẽ trả request ngược lại cho middleware trước đó, và sẽ ngắt quá trình trong request pipeline.

Mỗi Middleware trong pipeline sẽ tuần tự có cơ hội thứ hai để kiểm tra lại request và điều chỉnh response trước khi được trả lại.

Cuối cùng, response sẽ đến Kestrel nó sẽ trả response về cho client. Bất cứ middleware nào trong request pipeline đều có thể ngắt request pipeline tại chỗ đó với chỉ một bước đơn giản là không gán request đó đi tiếp.

Quá trình này giống một cuộc thi chạy tiếp sức giữa các middleware phải không các bạn

Cấu hình Request Pipeline

Để bắt đầu sử dụng Middleware, chúng ta cần thêm nó vào Request Pipeline. Nó được thực hiện bởi phương thức **Configure** trong Startup class. Phương thức **Configure** sẽ nhận các thể hiện của **IApplicationBuilder**, sử dụng chúng để đăng ký các middleware của chúng ta. Mở ứng dụng Helloworld trong bài [Bắt đầu khởi tạo ứng dụng ASP.NET Core](#). Hoặc bạn có thể tạo một ứng dụng ASP.NET Core. Trong project template chọn **Empty Project** và chọn .NET Core, .NET Core 2.2.

Mở file **Startup** và tìm phương thức **Configure** sau đó thay đổi code như sau;

```

public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{

    app.Run(async (context) =>
    {
        await context.Response.WriteAsync("<div> Hello World from the
middleware 1 </div>");
    });

}

```

Chúng ta sử dụng phương thức **app.Run** để đăng ký middleware đầu tiên và hiển thị dòng chữ "Hello world from Middleware 1" khi nó thực thi.

Phương thức **app.Run** để lấy thể hiện của **HttpContext**. Bạn có thể sử dụng đối tượng **Response** từ **HttpContext** để viết thêm các thông tin vào **HttpResponse**.

Cấu hình Middleware với Use và Run

Sử dụng hai extension method **Use** và **Run** để cho phép chúng ta đăng ký Middleware nội bộ (inline middleware) vào request pipeline. Phương thức **Run** thêm vào một middleware ngắt. Phương thức **Use** thêm vào middleware, nó sẽ gọi middleware tiếp theo trong pipeline.

Giờ hãy thêm một middleware nữa sử dụng **app.Run**

```

public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{

    app.Use(async (context, next) =>
    {
        await context.Response.WriteAsync("<div> Hello World from the
middleware 1 </div>");
    });

    app.Run(async (context) =>
    {
        await context.Response.WriteAsync("<div> Hello World from the
middleware 2 </div>");
    });
}

```

```
});
```

```
}
```

Phương thức **app.Use** có nhận hai tham số, một là **HttpContext** và hai là một **RequestDelegate**, về cơ bản nó là tham chiếu tới middleware tiếp theo. Giờ hãy chạy code.

Thông điệp “Hello world from the middleware 1” xuất hiện trên trình duyệt. Thông điệp từ middleware thứ 2 không xuất hiện. Bởi vì trách nhiệm của nó gọi middleware tiếp theo. Chúng ta có thể gọi middleware tiếp theo bằng cách gọi phương thức **Invoke** của middleware tiếp theo:

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
```

```
{
```

```
    app.Use(async (context, next) =>
```

```
    {
```

```
        await context.Response.WriteAsync("<div> Hello World from the  
middleware 1 </div>");
```

```
        await next.Invoke();
```

```
    });
```

```
    app.Run(async (context) =>
```

```
    {
```

```
        await context.Response.WriteAsync("<div> Hello World from the  
middleware 2 </div>");
```

```
    });
```

```
}
```

Giờ chạy code bạn sẽ thấy cả hai dòng chữ. Tiếp theo chúng ta sẽ thêm một middleware nữa và thêm một dòng chữ theo sau khi gọi đến middleware tiếp theo

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
```

```
{
```

```
    app.Use(async (context, next) =>
```

```
    {
```

```

        await context.Response.WriteAsync("<div> Hello World from the
middleware 1 </div>");

        await next.Invoke();

        await context.Response.WriteAsync("<div> Returning from the
middleware 1 </div>");

    });

    app.Use(async (context, next) =>
    {

        await context.Response.WriteAsync("<div> Hello World from the
middleware 2 </div>");

        await next.Invoke();

        await context.Response.WriteAsync("<div> Returning from the
middleware 2 </div>");

    });

    app.Run(async (context) =>
    {

        await context.Response.WriteAsync("<div> Hello World from the
middleware 3 </div>");

    });

}

```

Chúng ta sẽ thấy rằng các middleware có cơ hội thứ 2 để sửa request trên đường quay lại.

Sắp xếp middleware

Middleware sẽ được thực thi theo thứ tự mà chúng ta thêm nó vào pipeline.

Custom Middleware

Trong phần trên chúng ta đã tạo các middleware nội bộ (inline middleware) sử dụng **app.Use** và **app.Run**. Cách khác là chúng ta tạo middleware bằng cách sử dụng các class. Class middleware không yêu cầu triển khai bất cứ interface nào hoặc kế thừa từ bất cứ class nào. Tuy nhiên nó có 2 quy tắc mà chúng ta phải tuân theo:

Class middleware phải khai báo public constructor và không static với ít nhất một tham số thuộc kiểu `RequestDelegate`. Đây chính là tham chiếu đến middleware tiếp theo trong pipeline. Khi bạn gọi **RequestDelegate** này thực tế là bạn đang gọi middleware kế tiếp trong pipeline.

Class middleware phải định nghĩa một method public tên là **Invoke** nhận một **HttpContext** và trả về một **Task**. Đây là phương thức được gọi khi request tới middleware.

Tạo custom middleware

Giờ chúng ta sẽ tạo một class đơn giản để demo. Tên của nó là SimpleMiddleware.

```
public class SimpleMiddleware
```

```
{  
  
    private readonly RequestDelegate _next;  
  
    public SimpleMiddleware(RequestDelegate next)  
    {  
        _next = next;  
    }  
  
    public async System.Threading.Tasks.Task Invoke(HttpContext context)  
    {  
        await context.Response.WriteAsync("<div> Hello from Simple Middleware  
</div>");  
        await _next(context);  
        await context.Response.WriteAsync("<div> Bye from Simple Middleware  
</div>");  
    }  
}
```

Đầu tiên trong constructor, chúng ta sẽ lấy tham chiếu của middleware tiếp theo trong pipeline. Chúng ta lưu nó trong biến local tên là _next

```
    public SimpleMiddleware(RequestDelegate next)  
    {  
        _next = next;  
    }
```

Tiếp theo chúng ta phải khai báo phương thức Invoke, nó sẽ nhận tham chiếu đến HttpContext. Chúng ta sẽ viết ra một vài dòng chữ vào response và sau đó gọi middleware tiếp theo sử dụng await _next(context).

```
public async System.Threading.Tasks.Task Invoke(HttpContext context)
```

```

{

    await context.Response.WriteAsync("<div> Hello from Simple Middleware
</div>");

    await _next(context);

    await context.Response.WriteAsync("<div> Bye from Simple Middleware
</div>");

}

```

Tiếp theo chúng ta sẽ cần đăng ký middleware trong request pipeline. Chúng ta có thể sử dụng phương thức `UseMiddleware` như dưới đây:

```
app.UseMiddleware<SimpleMiddleware>();
```

Copy đoạn trên vào phương thức **Configure**. Chạy code và bạn sẽ thấy dòng chữ xuất ra từ `SimpleMiddleware` hiển thị trên trình duyệt.

Extension Method

Chúng ta có thể tạo ra một extension method đơn giản để đăng ký middleware. Tạo một class khác có tên `SomeMiddlewareExtensions` và tạo một phương thức `UseSimpleMiddleware` như dưới:

```

public static class SomeMiddlewareExtensions
{
    public static IApplicationBuilder UseSimpleMiddleware(this
IApplicationBuilder builder)
    {
        return builder.UseMiddleware<SimpleMiddleware>();
    }
}

```

Giờ thay vì sử dụng:

```
app.UseMiddleware<SimpleMiddleware>();
```

Chúng ta sẽ sử dụng:

```
app.UseSimpleMiddleware();
```

để đăng ký middleware.

Tổng kết

Middleware là mã nguồn hay là các thành phần riêng lẻ để xử lý các request đến. Các middleware này đứng theo dãy với nhau gọi là request pipeline. Chúng ta tạo middleware và đăng ký nó trong phương thức Configure của class Startup.

9. Làm việc với file cấu hình trong ASP.NET Core

Cấu hình hệ thống đã thay đổi rất nhiều trong ASP.NET Core. ASP.NET Cũ sử dụng file Web.Config để lưu thông tin cấu hình. Hệ thống mới giờ đây cung cấp hẳn API để làm việc với cấu hình và hỗ trợ nhiều định dạng file như JSON, XML, INI, tham số đầu vào từ dòng lệnh (command line arguments), biến môi trường (environment variables)... Bài viết này chúng ta sẽ tìm hiểu về cách lưu trữ và đọc cấu hình từ file **appsettings.json**.

Cấu hình là gì?

Cấu hình là các tham số hoặc các cài đặt cụ thể cho ứng dụng. Các cài đặt này được lưu trữ riêng biệt trong code và trong các file độc lập. Nó giúp các developer và quản trị kiểm soát và dễ dàng thay đổi cách mà ứng dụng chạy.

Ví dụ: Connection Strings cần để kết nối đến cơ sở dữ liệu được lưu trong một file cấu hình. Bằng cách thay đổi chuỗi connection mà bạn có thể thay đổi tên cơ sở dữ liệu, vị trí... mà không cần thay đổi mã nguồn.

Ứng dụng ASP.NET sử dụng Web.Config để lưu trữ cấu hình. Ví dụ chuỗi kết nối được lưu như dưới đây:

```
<connectionStrings>
```

```
  <add connectionString="data source=ABC;Integrated Security=SSPI;  
    initial catalog=xyz" providerName="System.Data.SqlClient"/>
```

```
</connectionStrings>
```

Cấu hình trong ASP.NET Core

Cấu hình trong ứng dụng ASP.NET Core được lưu dưới dạng cặp tên-giá trị. Bạn có thể lưu chúng trong file JSON, XML hay INI.

Ví dụ dưới đây hiển thị một cấu hình JSON đơn giản:

```
{  
  "option1": "value1",  
  "option2": 2  
}
```

Cặp tên-giá trị này có thể được nhóm lại vào trong một cây đa cấp như dưới đây:

```
{
```

```

"subsection":
{
  "suboption1": "subvalue1",
  "suboption2": "subvalue2"
}
}

```

Đây cũng là điểm mới và mạnh hơn so với sử dụng Web.Config trước đây khi mà các cấu hình được nhóm lại từng nhóm và hỗ trợ nhiều cấp giúp quản lý cấu hình dễ dàng hơn. Ứng dụng ASP.NET Core đọc các file cấu hình từ lúc khởi động. Nó có thể được cấu hình để đọc trong lúc chạy nếu nó thay đổi.

Nguồn cấu hình

ASP.NET Core hỗ trợ đọc cấu hình từ các nguồn khác nhau và các định dạng khác nhau. Một vài nguồn được sử dụng phổ biến như:

Định dạng file (JSON, INI hoặc XML)

Command line Arguments (tham số dòng lệnh)

Environment variables (biến môi trường)

Custom Provider (cái này là tự tạo ra provider riêng theo ý muốn)

Load cấu hình

Mở Visual Studio 2017 và tạo một ứng dụng ASP.NET Core 2.2 **Empty Project**. Nếu bạn chưa biết hãy tham khảo bài viết [Bắt đầu khởi tạo ứng dụng ASP.NET Core](#).

Mở **Program.cs** nó sẽ có phương thức **Main**, đây là điểm khởi đầu của ứng dụng chúng ta cũng có bài viết nói đến nó ở [Khởi động ứng dụng trong ASP.NET Core](#)

```

public class Program
{
    public static void Main(string[] args)
    {
        BuildWebHost(args).Run();
    }

    public static IWebHost BuildWebHost(string[] args) =>
        WebHost.CreateDefaultBuilder(args)
            .UseStartup<Startup>()

```

```

        .Build();
    }

```

Phương thức **Main** gọi **CreateDefaultBuilder** là một helper class. Một trong các công việc mà **CreateDefaultBuilder** thực hiện là load cấu hình từ các nguồn:

```

.ConfigureAppConfiguration((hostingContext, config) => {

    var env = hostingContext.HostingEnvironment;

    config.AddJsonFile("appsettings.json", optional: true, reloadOnChange:
true)

        .AddJsonFile($"appsettings.{env.EnvironmentName}.json", optional:
true, reloadOnChange: true);

    if (env.IsDevelopment()) {
        var appAssembly = Assembly.Load(new AssemblyName(env.ApplicationName));
        if (appAssembly != null) {
            config.AddUserSecrets(appAssembly, optional: true);
        }
    }

    config.AddEnvironmentVariables();
    if (args != null) {
        config.AddCommandLine(args);
    }

})

```

ConfigureAppConfiguration

Phương thức **ConfigureAppConfiguration** nhận 2 tham số. Tham số đầu tiên là **HostingContext** là một thể hiện của **WebHostBuilderContext** và thứ hai là biến **config**, thể hiện của **IConfigurationBuilder**. **WebHostBuilderContext** đưa ra thuộc tính **HostingEnvironment** nó giúp chúng ta biết được chúng ta đang chạy ở môi trường **Development** (**IsDevelopment**), **Production** (**IsProduction**) hay **Staging** (**IsStaging**).

IConfigurationBuilder đưa ra một số phương thức để load file cấu hình.

Load cấu hình từ file JSON

Phương thức **AddJsonFile** load cấu hình từ file JSON.

```
config.AddJsonFile("appsettings.json", optional: true, reloadOnChange: true)
```

Tham số đầu tiên là tên của file JSON, đường dẫn tương đối đến **ContentRoot**.

Nếu tham số thứ 2 là **optional** là true, thì nó sẽ không sinh ra bất cứ lỗi gì nếu nó không tìm thấy file **appsettings.json**.

Tham số thứ 3 **reloadOnChange** nếu là true, nó sẽ load lại cấu hình nếu nội dung của file cấu hình thay đổi chúng ta không cần restart ứng dụng.

Dòng tiếp theo của mã nguồn sẽ load ra file json thứ 2 đúng với tên môi trường đang chạy: appsettings.{env.EnvironmentName}.json. Điều này giúp chúng ta có các cấu hình khác nhau cho mỗi môi trường như là appsettings.Development.json.

```
config.AddJsonFile("appsettings.json", optional: true, reloadOnChange: true)
```

Load từ user secrets

Đây là tính năng mới trong ASP.NET Core khi mà cấu hình được lưu trữ bên ngoài mã nguồn. Nó được gọi là user secrets và chỉ có thể áp dụng cho môi trường Development. Điều này hữu ích trong trường hợp bạn có những cấu hình chỉ muốn sử dụng trên máy dev thôi, và không muốn nó nằm trong mã nguồn khi đưa code lên source control chung.

```
if (env.IsDevelopment()) {  
    var appAssembly = Assembly.Load(new AssemblyName(env.ApplicationName));  
    if (appAssembly != null) {  
        config.AddUserSecrets(appAssembly, optional: true);  
    }  
}
```

Load cấu hình từ biến môi trường

Phương thức **AddEnvironmentVariables()** đọc tất cả các user và system environment variable trong hệ điều hành ra.

```
config.AddEnvironmentVariables();
```

Đọc từ command line arguments

Phương thức **AddCommandLine** đọc tất cả các tham số của dòng lệnh khi các bạn gọi ứng dụng bằng command line mà truyền các tham số.

```
if (args != null) {  
    config.AddCommandLine(args);  
}
```

Đọc cấu hình

Template mặc định là Empty không tạo ra file **appsettings.json**. Tạo file **appsetting.json** trong thư mục gốc của ứng dụng. Và thêm cấu hình ví dụ vào đây. Các key của file cấu hình là không phân biệt chữ hoa chữ thường.

```
{  
  "message" : "Hello from configuration"  
}
```

Sau đó hãy đọc thông số message này ra. Mở file **Startup.cs**. Using name space **using Microsoft.Extensions.Configuration;**

Để đọc cấu hình, chúng ta cần thể hiện của **IConfiguration**. Chúng ta có thể dùng dependency injection để lấy thể hiện của nó trong constructor của **Startup** class. Bạn có thể dùng kỹ thuật này tương tự với Controller.

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

Sau đó nhận giá trị với phương thức **Configuration.GetSection**:

```
await context.Response.WriteAsync("<div>" +  
Configuration.GetSection("message").Value + "</div>");
```

Đọc Connection String

Chuỗi kết nối đến cơ sở dữ liệu có thể được thêm vào dưới đây. Chuỗi kết nối MySQL và SQL được nhóm vào một section là **ConnectionStrings** hoặc một node nếu là XML

```
"ConnectionStrings" :  
{  
  "MySQLConnection": "This is MySQL Connection",  
  "SQLConnection": "This is SQL Server Connection"  
}
```

Mỗi một section hoặc 1 node được chia tách bằng dấu phẩy và giá trị của nó được lấy ra như sau:

```
await context.Response.WriteAsync("<div>" +  
Configuration.GetSection("ConnectionStrings:MySQLConnection").Value + "</div>");  
  
await context.Response.WriteAsync("<div>" +  
Configuration.GetSection("ConnectionStrings:SQLConnection").Value + "</div>");
```

Đọc mảng từ file cấu hình

Bạn có thể sử dụng JSON với mảng như sau:

```
"wizards": [  
  {  
    "Name": "Gandalf",  
    "Age": "1000"  
  },  
  {  
    "Name": "Harry",  
    "Age": "17"  
  }  
]
```

Để đọc được đoạn cấu hình trên chúng ta cần sử dụng index như là một phần được chia tách trong string bởi dấu hai chấm:

```
await context.Response.WriteAsync("<div>" +  
Configuration.GetSection("wizards:0:Name").Value + "</div>");
```

Command line

Hệ thống cấu hình có thể được load từ các tham số của dòng lệnh. Phương thức **CreateDefaultBuilder** tải các đối số của dòng lệnh sử dụng.

```
config.AddCommandLine(args);
```

Các đối số của dòng lệnh nên tuân theo các quy tắc cụ thể. Các đối số phải được truyền dạng cặp key-value. Mỗi cặp key value phải được cách nhau bởi dấu cách.

Mở file startup và đến phương thức Configure và thêm đoạn code:

```
await context.Response.WriteAsync("<div>" + Configuration.GetSection("Arg1").Value  
+ "</div>");
```

```
await context.Response.WriteAsync("<div>" + Configuration.GetSection("Arg2").Value  
+ "</div>");
```

Mở cửa sổ console và đến thư mục project sau đó gọi dotnet run với 2 tham số command line:

```
Dotnet run arg1=Hello arg2=World
```

Giá trị "Hello" và "World" được hiển thị trên trình duyệt với 2 dòng tách biệt.

Đọc biến môi trường

Phương thức **config.AddEnvironmentVariables()**; load biến môi trường vào bộ nhớ của configuration collection. Bạn có thể đọc như sau:

```
await context.Response.WriteAsync("<div>" + Configuration.GetSection("PATH").Value + "</div>");
```

Thêm file cấu hình theo ý muốn

Bạn cũng có thể thêm bất cứ file cấu hình tùy biến nào. Đây là một tính năng rất hay bạn có thể tạo ra các file cấu hình riêng cho từng module hoặc hệ thống. Nó giúp quản lý cấu hình dễ hơn.

Ví dụ tạo 1 file **test.json** trong thư mục gốc:

```
{  
  
  "testMessage": "Hello from test.json"  
  
}
```

Giờ bạn hãy mở file **Program.cs** ra và sửa code:

```
public static IWebHost BuildWebHost(string[] args) =>  
    WebHost.CreateDefaultBuilder(args)  
        .UseStartup<Startup>()  
        .ConfigureAppConfiguration((hostingContext, config) =>  
        {  
            config.AddJsonFile("Test.json", true, true);  
        })  
        .Build();
```

Giờ hệ thống cấu hình sẽ tự động thêm tất cả các cấu hình từ file test.json.

```
await context.Response.WriteAsync("<div>" +  
Configuration.GetSection("testMessage").Value + "</div>");
```

Thứ tự sắp xếp

Thứ tự file cấu hình nào sẽ được load trước sau nó phụ thuộc vào thằng nào được add vào sau sẽ được ghi đè thằng trước.

Tổng kết

Chúng ta đã tìm hiểu hệ thống quản lý cấu hình mới trong ASP.NET Core. ASP.NET Core Empty project đã cài đặt ứng dụng để đọc thông tin cấu hình từ appsettings.json, appsettings.{env.EnvironmentName}.json, user secrets, environment variable, và từ command line. Chúng ta có thể mở rộng bằng cách add thêm các file tùy biến mà chúng ta muốn.

10. Sử dụng Static Files trong ASP.NET Core

Bài viết này chúng ta sẽ học cách làm việc với Static File trong ASP.NET Core. ASP.NET Core có khả năng duyệt file tĩnh như là HTML, CSS, hình ảnh và Javascript trực tiếp từ client mà không đi qua MVC Middleware.

Sử dụng Static Files

Các file như HTML, CSS, ảnh, JavaScript được gọi là file tĩnh hay static files. Có hai cách để bạn có thể duyệt các file tĩnh này trong ASP.NET Core MVC. Trực tiếp hoặc thông qua Controller MVC. Bài này chúng ta sẽ học cách duyệt trực tiếp. Các file tĩnh đó nó có nội dung không thay đổi động khi người dùng request tới nó. Vì thế nó không cần phải tạo bất cứ yêu cầu nào thông qua MVC Middleware (hay cách khác là request pipeline) mà chỉ cần trả nguyên nội dung về thôi. ASP.NET Core cung cấp một Middleware có sẵn chỉ phục vụ việc này.

Tạo mới project

Mở Visual Studio 2017 và tạo mới một project sử dụng Empty Project. Tên của Project là StaticFiles. Bạn có thể tham khảo bài viết hướng dẫn [Bắt đầu khởi tạo ứng dụng ASP.NET Core](#). Chạy project và kiểm tra mọi thứ ok.

Static file middleware

Chúng ta cũng đã nói về [Middleware và Request Pipeline trong ASP.NET Core](#) trước đây. Để duyệt các static file chúng ta cần thêm StaticFiles Middleware vào. Middleware này có sẵn trong thư viện **Microsoft.AspNetCore.StaticFiles**. Nó không cần cài đặt gì vì nó là một phần của **Microsoft.AspNetCore.App**. Chúng ta có thể cấu hình ASP.NET Core để duyệt file tĩnh sử dụng extension method **UseStaticFiles**.

Mở file **Startup.cs** và đặt **app.UseStaticFiles()** trước **app.Run**:

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    app.UseStaticFiles();
    app.Run(async (context) =>
    {
        await context.Response.WriteAsync("Hello World!");
    });
}
```

Chú ý là **app.Run** là middleware ngắt. Nếu bạn đặt **UseStaticFiles** trước **app.Run** thì nó sẽ không được thực thi.

Static file ở đâu?

Theo quy tắc thì static file sẽ được lưu trong web root ở thư mục (**<content-root>/wwwroot**). Bạn có thể thay đổi nếu muốn

Content root

Content root là thư mục gốc của ứng dụng. Tất cả các file của ứng dụng như view, controller, pages, javascript hay bất cứ file nào thuộc về ứng dụng.

Content root tương tự như thư mục của ứng dụng để có thể thực thi hosting app.

Webroot

Web root thư mục trong content root nơi mà nó để các tài nguyên tĩnh như CSS, Javascript, hình ảnh..

Tạo mới static file

Để thêm static file, ví dụ bạn thêm test.html. Chọn thư mục **wwwroot**, click chuột phải và chọn AddAdd new Item. Chọn HTML Page và đặt tên nó là **test.html**.

```
<!DOCTYPE html>

<html>

<head>

    <meta charset="utf-8" />

    <title></title>

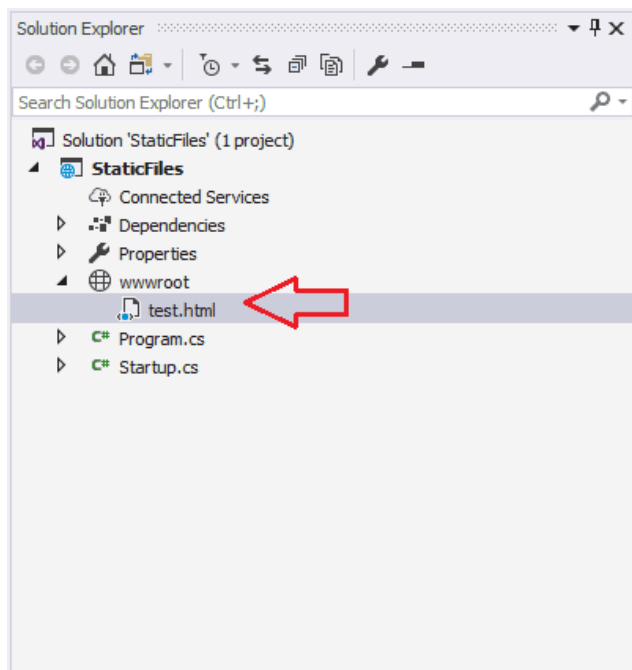
</head>

<body>

    <p> Hello world from test.html</p>

</body>

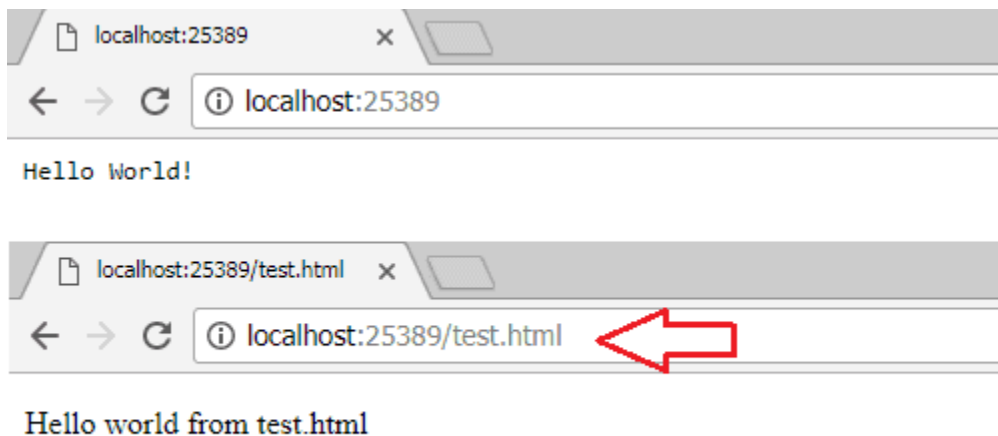
</html>
```



Bảo mật static file

Giờ hãy chạy project và nhìn thấy dòng chữ “Hello World”. Giờ bạn có thể chỉnh sửa URL thêm vào **test.html** và refresh bạn sẽ thấy dòng chữ “Hello world from test.html”

Static file middleware không kiểm tra việc người dùng có được quyền xem file hay không. Nếu bạn chỉ muốn những người có quyền truy cập file thì bạn nên lưu nó bên ngoài thư mục **wwwroot** và bạn có thể cho phép duyệt file theo quyền với Controller/Action trả về **FileResult**.



UseStaticFile là một middleware ngắt

Static file là một Middleware ngắt. Nếu file tìm thấy nó sẽ trả về file và ngắt request pipeline. Nó sẽ gọi middleware kế tiếp chỉ khi nó không tìm thấy file.

Làm sao để đặt Content rooth path và web rooth path

Như đã nói đến ở trên thì các static file được đặt ở web root. Chúng ta có thể đặt content root là thư mục hiện tại. Nó đã được cài đặt mặc định trong phương thức **CreateDefaultBuilder** của **Program.cs**. **CreateDefaultBuilder** là một helper class chứa các logic làm việc với cấu hình.

Nó cài đặt content root của ứng dụng bằng cách gọi extension method **UseContentRoot**:

```
.UseContentRoot(Directory.GetCurrentDirectory())
```

Theo quy tắc thì webroot được đặt cho thư mục **wwwroot**. Bạn có thể thay đổi bằng cách sử dụng phương thức **UseWebRoot**.

11. MVC Design Pattern trong ASP.NET Core

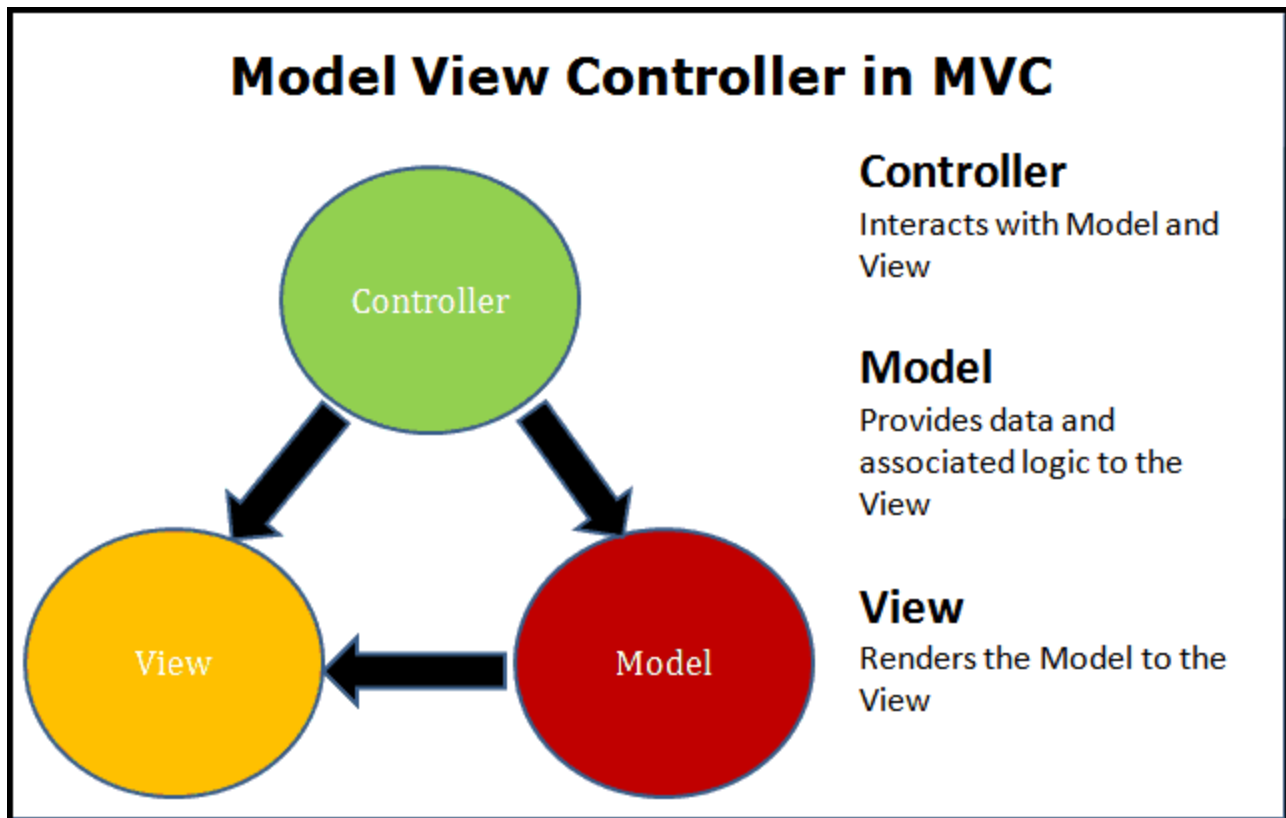
ASP.NET Core được dựa trên một design pattern phổ biến là MVC. Pattern MVC viết tắt của **Model-View-Controller**. Trong bài viết này chúng ta sẽ tìm hiểu MVC là gì và làm thế nào để sử dụng MVC pattern trong ASP.NET Core.

MVC Design Pattern trong ASP.NET Core

MVC Design Pattern là design pattern (mẫu thiết kế) phổ biến được sử dụng cho lớp trình bày (Presentation Layer). ASP.NET Core MVC tuân theo pattern này và nó là căn bản nhất để mọi thứ xoay quanh nó.

MVC là một pattern mạnh mẽ và biểu lộ rõ ràng kỹ thuật chia để trị (separating of concerns). Kiến trúc MVC chia ứng dụng ra làm 3 phần độc lập và duy nhất. Bao gồm layer **Model**, **View** và **Controller**.

Mỗi một layer trên có một trách nhiệm cụ thể. Model chứa dữ liệu. View tạo ra giao diện người dùng từ dữ liệu ở Model và tương tác với người dùng. View sẽ chuyển tương tác từ người dùng đến Controller, từ đó Controller sẽ build Model và cập nhật View.



Chia để trị (Separation of concerns)

Triết lý chia để trị có nghĩa là mỗi thành phần trong ứng dụng chỉ nên có một trách nhiệm cụ thể. Chúng sẽ độc lập với các thành phần khác nhiều nhất có thể. Nói cách khác, các thành phần nên giảm bớt sự phụ thuộc vào nhau. Ứng dụng được xây dựng dựa trên triết lý này sẽ dễ dàng kiểm thử, bảo trì và mở rộng.

MVC Pattern tuân theo triết lý chia để trị. Mỗi một thành phần trong 3-layer có thể được phát triển và kiểm thử độc lập với nhau để kết hợp thành một ứng dụng mạnh mẽ.

Điều quan trọng ở đây là MVC Pattern là một pattern cho tầng hiển thị. Nó chỉ giải quyết vấn đề làm sao và khi nào dữ liệu được hiển thị cho người dùng. Bạn cần sử dụng pattern này kết hợp với lớp truy cập dữ liệu (Data Access Layer) và lớp nghiệp vụ (Business Layer) ...để tạo ra một ứng dụng hoàn chỉnh.

Model

Model trình bày dữ liệu mà nó cần hiển thị cho khách hàng kết hợp với một số logic hiển thị. Nó là một đối tượng hoặc chỉ đơn giản là một class C# với các thuộc tính và phương thức. Model không nên phụ thuộc Controller hoặc View. Nó chỉ có trách nhiệm trong việc nắm dữ liệu. Class Model có thể được sử dụng lại.

View

View là một thành phần dùng để hiển thị dữ liệu trực quan từ dữ liệu thô trong Model. Nó có trách nhiệm lấy Model từ Controller sau đó tạo ra giao diện và hiển thị nó cho người dùng. View làm nhiệm vụ kết hợp giữa Model và dữ liệu của nó để hiển thị. Nó có thể cập nhật model và gửi ngược lại nó đến Controller để cập nhật vào cơ sở dữ liệu. View sẽ không bao giờ truy cập đến tầng Business Layer hoặc Data Access Layer.

Một view có trách nhiệm sau đây:

Có trách nhiệm tương tác với người dùng.

Render model cho người dùng.

Cho phép người dùng tương tác và gửi request cho Controller.

Bao gồm các trang HTML chuẩn, JavaScript và CSS.

Có khả năng render JSON, XML và trả về các kiểu khác.

Controller

Controller nhận các request. Nó sẽ xây dựng model và chọn view thích hợp để hiển thị chúng. Nó đứng giữa View và Model. Bạn có thể nghĩ nó là nhạc trưởng điều phối View và Model. Nó sẽ điều khiển toàn bộ quá trình làm việc của cả 3.

Controller không nên quá phức tạp và cồng kềnh trong code của bạn. Nó chỉ nên đóng vai trò điều phối. Còn công việc nghiệp vụ nên để cho Business Layer làm. (Ví dụ: Tầng data access layer (DAL) sẽ lấy dữ liệu, business layer sẽ thực thi các logic nghiệp vụ) để xây dựng nên model. Model sau đó sẽ được kết hợp với View để tạo ra giao diện.

Controller có một số trách nhiệm sau:

Xử lý các request đến từ người dùng.

Controller sau đó gửi request đến các service tương ứng trong Business Layer.

Gửi các model đến view để tạo giao diện.

Gửi các lỗi validation hoặc lỗi thực thi về View nếu có.

Controller không bao giờ truy cập đến tầng data.

Controller tương tác với cả model và view. Nó điều khiển luồng dữ liệu đi vào đối tượng Model và cập nhật view khi có dữ liệu thay đổi. Nó giữ View và Model tách biệt.

MVC Pattern làm việc trong ASP.NET Core ra sao?

Khi request được bắt đầu bằng việc người dùng làm gì đó, ví dụ là click vào một nút trên màn hình hoặc một đường link trên trình duyệt. Ví dụ: <http://localhost/Customer/List>

Request trên đến MVC Middleware sau đó đi qua Request Pipeline. Chúng ta có thể tham khảo bài viết: [Middleware và Request Pipeline trong ASP.NET Core](#). MVC Middleware nhận diện URL và quyết định xem controller nào sẽ được gọi. Quá trình mapping request vào controller gọi là **Routing**.

MVC Middleware gọi Controller và truyền các request đến Controller. Controller sẽ nhận các request này và quyết định xem sẽ làm gì với nó. Request có thể là thêm mới khách hàng hoặc là lấy ra danh sách khách hàng. Controller dựng lên Model tương ứng, nó sẽ gọi tầng business layer để hoàn thành công việc.

Controller sẽ truyền model tương ứng về View và để View xây dựng trang kết quả từ response. View sẽ tạo ra trang kết quả tương ứng. Response có thể là một HTML, XML, JSON hoặc một file để download. Sau đó nó sẽ gửi lại cho người dùng.

Vòng đời của Request hoàn thành và ứng dụng đợi các request tiếp theo, và như thế chúng ta lại có một chu trình mới khác.

Tổng kết

Bài viết này chúng ta đã tìm hiểu về MVC Design Pattern để áp dụng cho ứng dụng ASP.NET MVC Core. Chúng ta đã tìm hiểu 3 thành phần quan trọng trong MVC là View, Model và Controller và vai trò của từng thành phần trong ứng dụng. Cuối cùng chúng ta đã tìm hiểu MVC Pattern làm việc thế nào trong ứng dụng ASP.NET Core MVC.

12. Xây dựng ứng dụng ASP.NET Core MVC đầu tiên

Bài viết này chúng ta sẽ học cách xây dựng ứng dụng ASP.NET Core MVC từ đầu sử dụng Visual Studio. Bài trước chúng ta đã học về [MVC Design Pattern trong ASP.NET Core](#) và chúng làm việc ra sao trong ASP.NET MVC Core. Chúng ta sẽ tiếp tục khám phá kiến trúc MVC và tôi sẽ chỉ cho bạn làm sao để thêm middleware MVC vào ứng dụng. Sau đó sẽ là thêm mới Controller, View và Model vào ứng dụng MVC Core.

Xây dựng ứng dụng ASP.NET MVC đầu tiên

Mở **Visual Studio 2017** và tạo một ứng dụng **ASP.NET Core Empty project**. Tên project sẽ là **MVCCoreApp**. Bạn có thể tham khảo bài viết [Bắt đầu khởi tạo ứng dụng ASP.NET Core](#).

MVC Middleware

Đây là thứ đầu tiên bạn cần thêm vào ứng dụng MVC. Một middleware là một thành phần trực thuộc [request pipeline](#) làm nhiệm vụ xử lý request đến và tạo ra phản hồi (response). Chúng ta có bài viết nói riêng về phần này [Middleware và Request Pipeline trong ASP.NET Core](#).

Thêm MVC Service

Để project của chúng ta làm việc với MVC, đầu tiên chúng ta cần thêm các service liên quan đến MVC. Mở file **Startup.cs** và tìm đến phương thức **ConfigureServices** sau đó thêm extension method **AddMvc()**.

Phương thức **AddMvc()** sẽ thêm tất cả các service liên quan đến MVC như Authorization, RazorViewEngine, Tag Helper, Data Annotation, JsonFormatters, Cors... vào service collection. Sau đó các service này có thể được tiêm (injected) vào bất cứ đâu được yêu cầu sử dụng cơ chế Dependency Injection.

Các service được đăng ký trong phương thức **ConfigureServices** ở class **Startup**. Mở file **Startup.cs** và cập nhật như sau:

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



```
{ services.AddMvc();  
}
```

Thêm MVC Middleware

Tiếp theo chúng ta cần thêm middleware MVC vào request pipeline. Đơn giản chỉ cần gọi **app.UseMvcWithDefaultRoute()**. Middleware phải được đăng ký trong phương thức **Configure()** của Startup class.

UseMvcWithDefaultRoute là một extension method được đặt trong **Microsoft.AspNetCore.Mvc.Core.Package** này là một phần của **Microsoft.AspNetCore.App** đã được cài đặt. Các bạn không cần bận tâm nhé.

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

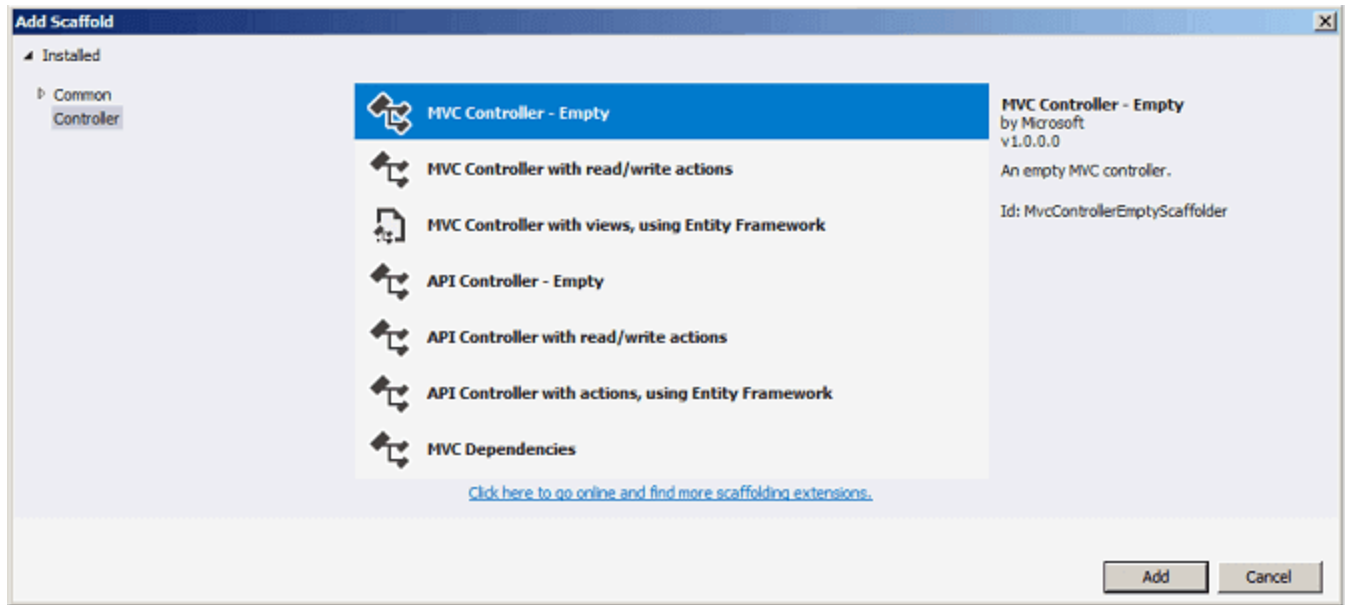
Phương thức **UseMvcWithDefaultRoute()** cài đặt MVC Middleware với route mặc định. Một tùy chọn khác là bạn có thể đăng ký MVC với phương thức **UseMvc()**. UseMvc yêu cầu bạn cài đặt một route. Chúng ta sẽ có bài riêng nói về routing này.

Thêm các controller

Controller trong MVC pattern có trách nhiệm nhận thông tin request và phản hồi lại cho người dùng. Nó sẽ xây dựng nên model và gán lại cho view để hiển thị.

Chuột phải vào Project và tạo một folder có tên **Controllers**. Chuột phải vào thư mục **Controllers** để chọn **Add Controller**.

Cửa sổ add controller hiện ra chúng ta chọn **MVC Controller – Empty**



Đặt tên cho controller mới là **HomeController**. Click **Add**. Giờ mở **HomeController** ra:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
namespace MVCCoreApp.Controllers
{
    public class HomeController : Controller
    {
        public IActionResult Index()
        {
            return View();
        }
    }
}
```

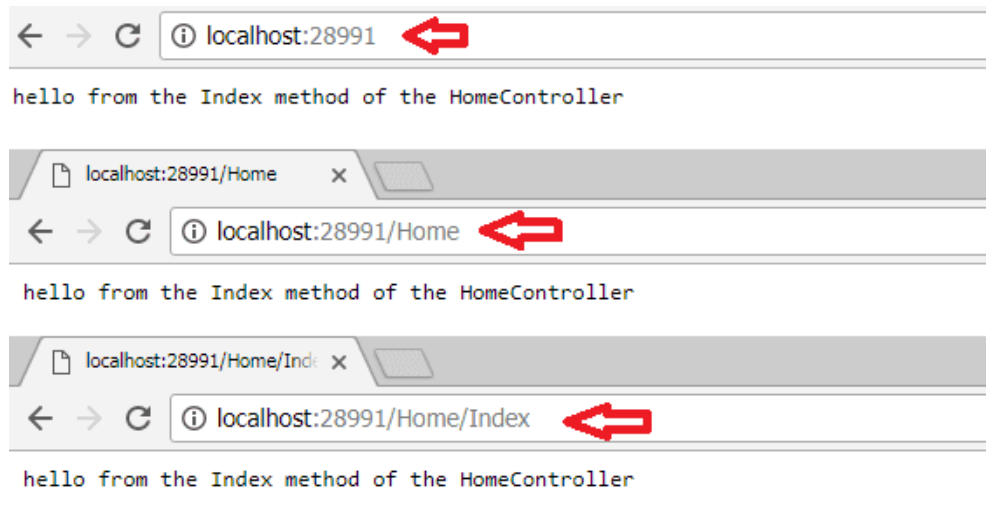
Bạn sẽ thấy **HomeController** kế thừa từ **Controller** và có một phương thức public là **Index**. Phương thức public này trong controller nó gọi là **Action Methods**.

Giờ bạn hãy thử thay đổi phương thức này và trả về một chuỗi như dưới đây:

```
public string Index()
{
    return "hello from the Index method of the HomeController";
}
```

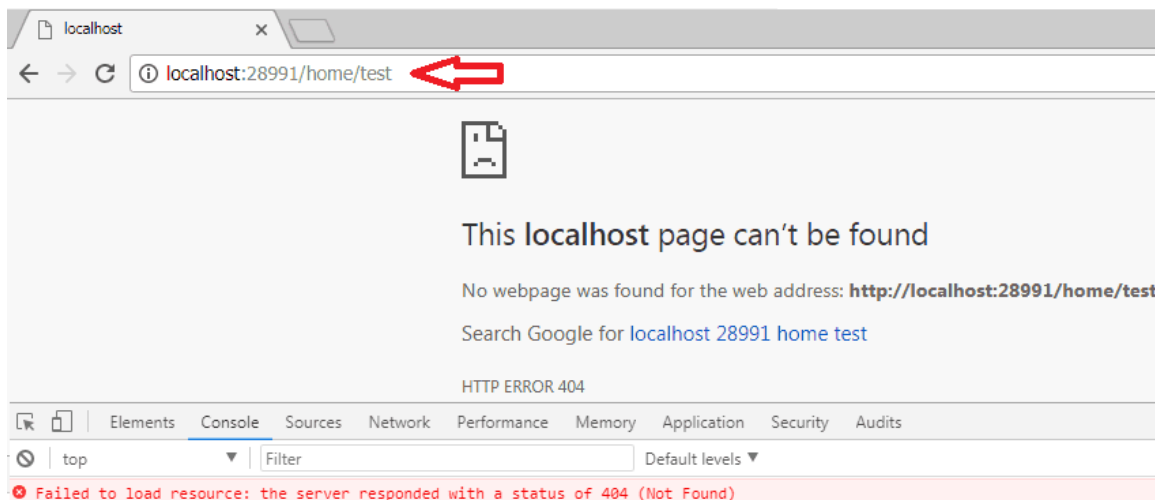
Chạy ứng dụng và bạn sẽ thấy dòng chữ “hello from the Index method of the HomeController” trên trình duyệt.

Sau đó thay đổi URL với /Home hoặc /Home/Index nó cũng trả về kết quả tương tự.



Khi bạn gõ **/Home/Index** trong trình duyệt, MVC sẽ tìm Controller bởi tên HomeController và tìm đến phương thức **Index** là một action method. Nó sẽ gọi Index và trả về kết quả lại cho user.

Giờ bạn hãy sửa thành **/Home/Test**. Nó sẽ ra kết quả là 404 trên trình duyệt. Điều này bởi vì không có phương thức nào tên là **Test** trong **HomeController** cả.



URL là / hoặc /Home cũng làm việc bởi vì MVC đã cấu hình tự động mặc định trở đến /Home/Index khi không chỉ ra nó.

ASP.NET MVC Core sử dụng cơ chế route để quyết định xem Controller và Action method nào sẽ được gọi. Chúng ta sẽ tìm hiểu trong bài Route trong ASP.NET Core.

Thêm View

View là thành phần hiển thị trực quan của model. Nó có trách nhiệm nhận thông tin model từ controller. Nó tạo ra giao diện từ model được gán và trình bày cho người dùng. Giờ chúng ta hãy thêm View vào ứng dụng.

Update Controller để trả về view

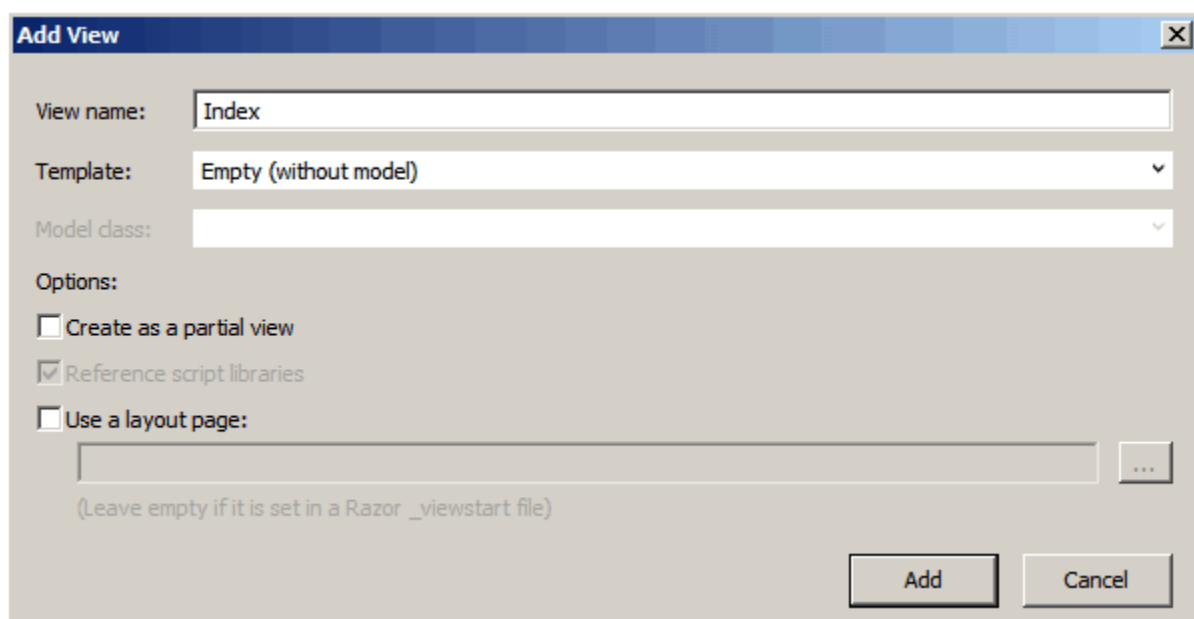
Vào HomeController sau đó đến phương thức Index. Thay đổi Index như sau:

```
public IActionResult Index()
{
    return View();
}
```

Chúng ta đã thay đổi phương thức Index để trả về một **IActionResult** thay vì một string. Một **ActionResult** là một kiểu trả về của **ActionMethod**. Phương thức View trả về một View tương ứng với Index.

Thêm View

Giờ hãy chuột phải vào Method Index và chọn Add View. Cửa sổ mở ra là Add View.



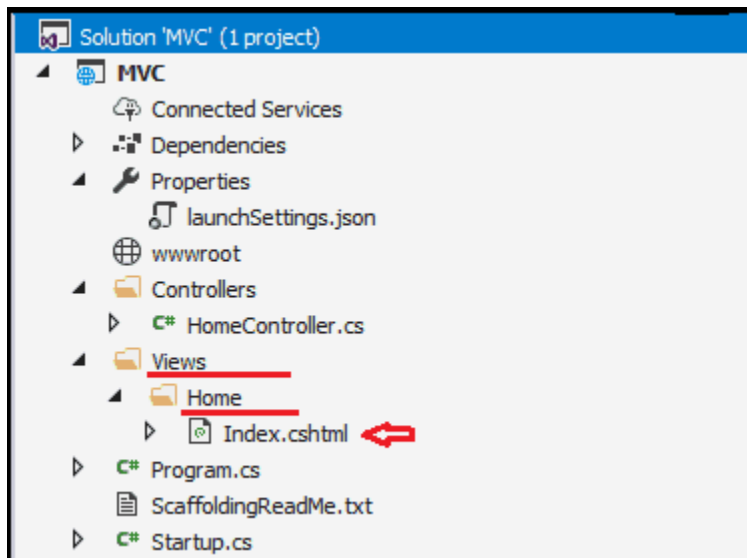
Để nguyên tên của nó là **Index. Template Empty (without model)**. Bỏ các lựa chọn là **Create as Partial Views** và **Use a Layout Page** như trên. Click nút **Add**.

Nó sẽ tạo ra một view trong thư mục **Views/Home**. Đây là quy tắc chuẩn của MVC Core khi mà tất cả các View sẽ nằm trong thư mục Views ngang hàng với thư mục **Controllers** và thư mục con Home chính là tên controller.

Views Folder

Theo quy tắc đó, tất cả các View sẽ nằm trong thư mục Views thuộc thư mục gốc. Mỗi Controller sẽ có một thư mục con bên trong Views nhưng không có hậu tố Controller. Vì thế HomeController sẽ có tương ứng một thư mục là Home trong thư mục Views.

Mỗi Action Method trong Controller sẽ được tạo ra một file view tương ứng giống tên của Action Method. Vì thế phương thức Index của **HomeController** sẽ có một file **Index.cshtml** trong thư mục Views/Home.



Giờ bạn hãy mở file **Index.cshtml** ra và thêm dòng chữ “<p>Hello from the View</p>” sau thẻ title như dưới đây:

```
@{
```

```
    Layout = null;
```

```
}
```

```
<!DOCTYPE html>
```

```
<html>
```

```
<head>
```

```

<meta name="viewport" content="width=device-width" />

<title>Index</title>

<p>Hello from the View</p>

</head>

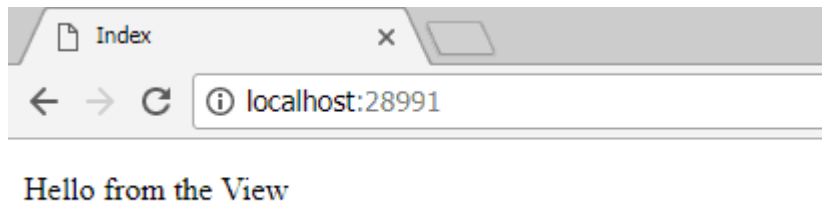
<body>

</body>

</html>

```

Giờ hãy chạy ứng dụng và nhìn kết quả:



MVC Controller và Action method gọi View bằng cách sử dụng phương thức **View()**. MVC sẽ tự tìm view tương ứng trong thư mục **View/<Controller>** và chọn view có tên **<ActionMethod>.cshtml**. Ngoài ra nó còn tìm cả trong thư mục “Shared” sau này chúng ta sẽ thấy.

Thêm model

Bạn quay lại HomeController và cập nhật:

```

public IActionResult Index()
{
    HomeModel message = new HomeModel();

    return View(message);
}

```

Chúng ta tạo ra một biến message và là thể hiện của class HomeModel.

```
HomeModel message = new HomeModel();
```

Sau đó gọi View và truyền vào biến đó:

```
return View(message)
```

Cập nhật View để nhận Model

```

@model MVCCoreApp.Models.HomeModel;

@{
    Layout = null; }

```

```
<!DOCTYPE html>

<html>

<head>

  <meta name="viewport" content="width=device-width" />

  <title>Index</title>

  <p> @Model.Message</p>

</head>

<body>

</body>

</html>
```

Giờ bạn hãy chạy app và xem kết quả Hello from Model



Tổng kết

Bài viết này chúng ta đã xây dựng ứng dụng ASP.NET Core từ đầu sử dụng Visual Studio. Chúng ta đã tìm hiểu làm sao để sử dụng MVC Middleware trong ứng dụng. Và cũng tìm hiểu cách thêm Controller sau đó trả về Hello message. Chúng ta cũng thêm Model và View vào để tạo ra một ứng dụng hoàn chỉnh theo Pattern MVC.

13. Cơ bản về ASP.NET Core Controller

Bài viết này chúng ta sẽ tìm hiểu khái niệm cơ bản của ASP.NET Core Controller. Bạn sẽ tìm hiểu Controller là gì và trách nhiệm của nó. Bạn cũng học cách thêm mới **Controller** và các **Action Method** trong controller cũng như các kiểu trả về.

Controller là gì?

Controller là thành phần đầu tiên nhận request từ người dùng. Khi người dùng truy cập URL qua trình duyệt, ASP.NET Core routing sẽ map request đó vào controller cụ thể.

Ví dụ: Request URL như sau: **http://localhost/Customer/List**

Trường hợp này, Controller có tên là **CustomerController** được gọi. Sau đó nó sẽ gọi đến Action method tên **List** rồi tạo ra response trả về cho user.

Trách nhiệm của Controller

Controller có 3 trách nhiệm chính:

Nhận request

Dựng model

Gửi trả response

Nhận request

Controller có trách nhiệm nhận request từ user.

Dựng model

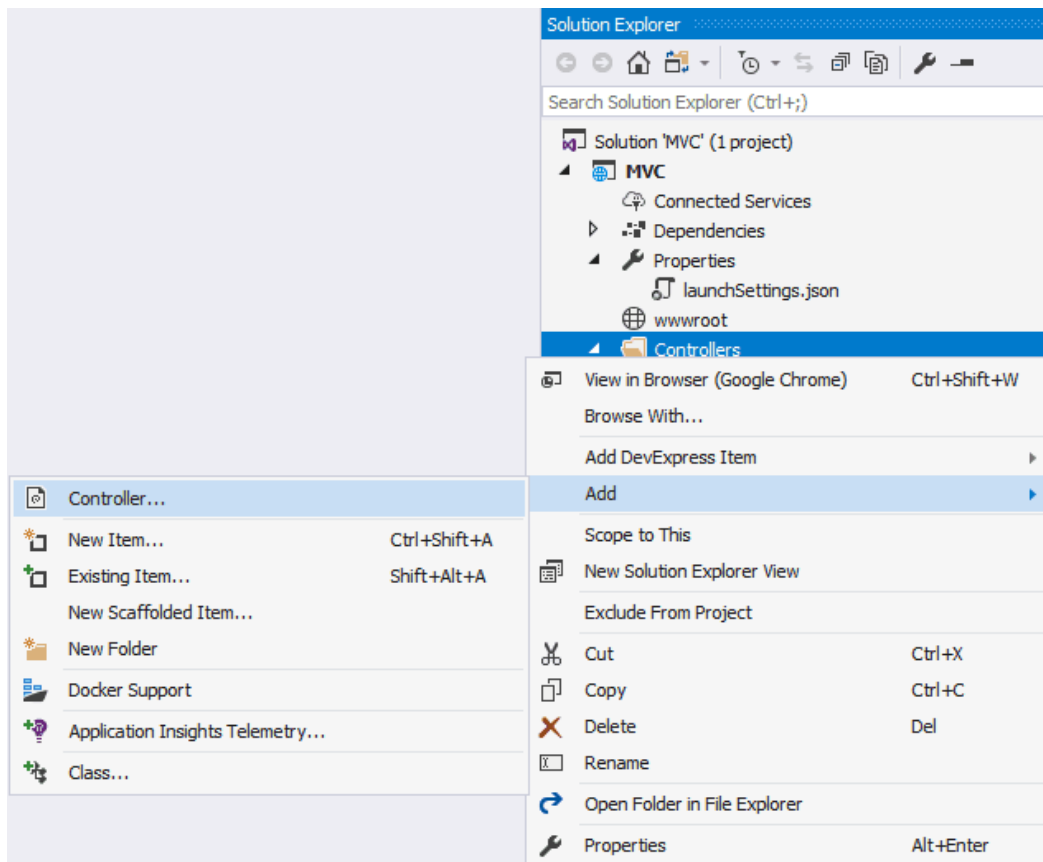
Controller Action method thực thi logic của ứng dụng và xây dựng nên model.

Gửi trả kết quả

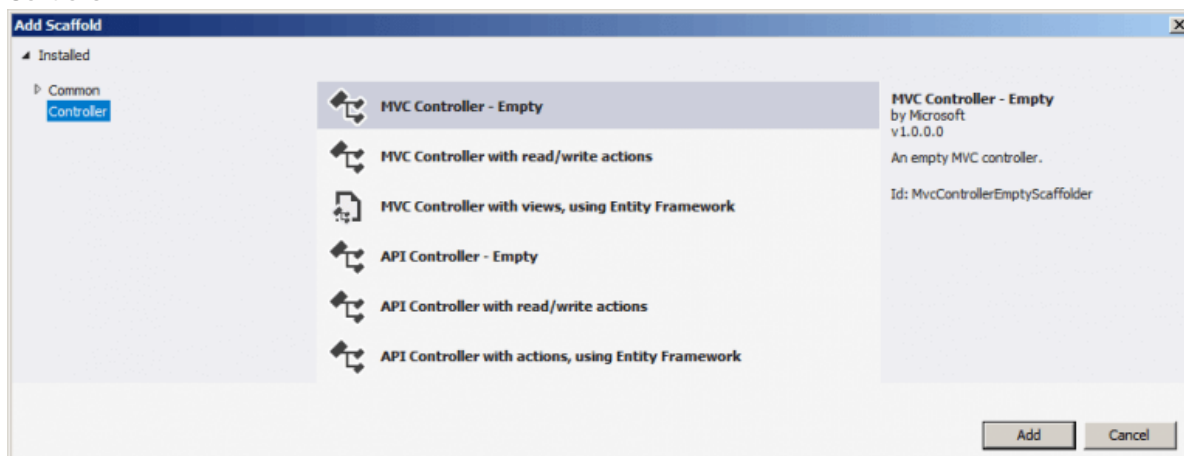
Cuối cùng nó sẽ trả về kết quả trong HTML, File, JSON, XML hoặc bất cứ định dạng nào về cho user.

Thêm mới controller

Controller không khác gì so với .NET Class thông thường. Chỉ có điều nó nằm trong thư mục **Controllers** là thư mục nằm trong thư mục gốc của ứng dụng. Bạn có thể tạo mới controller bằng cách chọn thư mục **Controllers** và chuột phải chọn **Add-> Controller**.



Sau đó bạn sẽ được nhìn thấy các tùy chọn khác nhau. Có 2 tùy chọn chính: MVC Controller và API Controller



Cả MVC và API controller đều kế thừa từ cùng class Controller nên nó không khác nhau nhiều. Ngoại trừ API controller nhằm mục đích trả về dữ liệu được định dạng cho client.

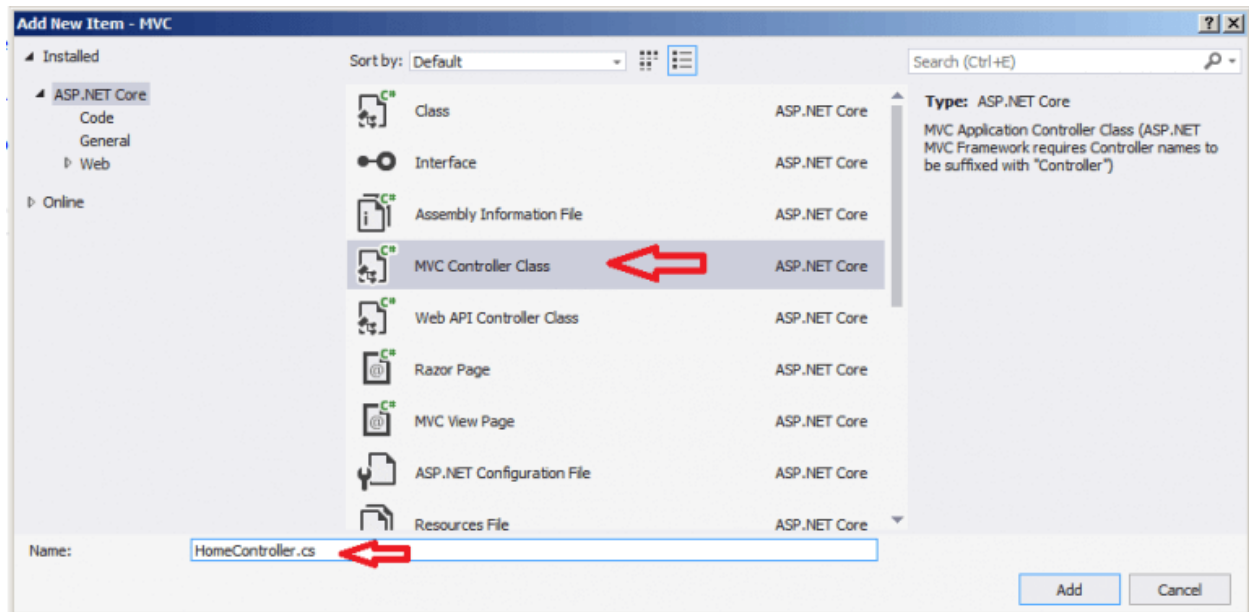
Chúng ta cũng có 3 tùy chọn cho mỗi loại trên:

Empty

With Read/Write Actions

With Views, using entity framework

Có cách khác tạo Controller là chuột phải vào thư mục **Controllers** chọn **Add new item** và chọn MVC Controller trong template. Nó sẽ tạo ra một Empty Controller.



Các controller trong ASP.NET Core kế thừa từ class **Controller** và nó lại kế thừa từ class **ControllerBaseClass**. 2 base class này cung cấp nhiều các phương thức hữu ích.

Controller class phải thỏa mãn ít nhất một trong các điều kiện sau:

Tên class phải có đuôi "Controller"

Class kế thừa từ một class cung phải có tên đuôi là "Controller"

Class được đặt thuộc tính [Controller]

Action Method

Bất cứ public method nào được chia ra ngoài bởi Controller đều được gọi là Action method. Action method get được gọi khi user gõ một URL trên trình duyệt.

Ví dụ: **http://localhost/Custom/List** sẽ gọi action method tên **List** trong **CustomerController**.

Action method sẽ gọi tầng service layer để phản hồi lại request. Service layer sẽ tương tác với database sử dụng tầng data access layer và map kết quả về model sau đó gửi lại cho Action method.

Action method sẽ gọi View với model để trả về kết quả cho người dùng.

Bất cứ public method trong controller có thể được gọi bởi bất cứ ai biết được URL của nó. Hãy cẩn thận khi bạn đặt một public method trong controller.

Khi tạo một Action method bạn cần nhớ những điều sau:

Action method phải là một public method

Action method không thể là static method hoặc một extension method.

Constructor, getter, setter không được sử dụng.

Các phương thức được kế thừa không được sử dụng như là một action method.

Action method không được chứa từ khóa ref hoặc out trên tham số.

Action method không được chứa thuộc tính [NonAction]

Action method không thể được nạp chồng (overloaded)

Routing

Routing module là một phần quan trọng của ASP.NET Core. Module này phân tích URL đến và gọi đúng Controller và action tương ứng với request.

Ví dụ URL được gọi là **http://localhost/Customers/List** thì đầu tiên nó sẽ đi qua routing module. Routing module sẽ phân tích URL này và chọn Controller tương ứng để nhận request. Sau đó nó sẽ tạo ra một thể hiện của Controller và gọi đúng action method.

Chúng ta sẽ thảo luận nó trong bài về routing trong ASP.NET Core. Routing có thể triển khai theo kiểu Convention based hoặc Attribute based. Routes được định nghĩa sử dụng convention hay attribute đều được. Routing module sử dụng Routes map để map request đến Controller Action. Routes có thể chỉ ra ActionName attribute đến tên Action method. Ngoài ra thì HTTP Action Verbs có thể được dùng để điều khiển theo HTTP request method. Hơn nữa, chúng ta có thể sử dụng Route Constrains để phân biệt giữa các route giống nhau.

Kiểu trả về

Action method có trách nhiệm trong việc nhận các request và tạo ra response trả về cho người dùng. Response có thể là một trang HTML, JSON, XML hay một file để download.

Một controller action có thể trả về bất cứ kiểu nào bạn muốn. Ví dụ action method dưới đây trả về một string.

```
public string Index() {  
    return "Hello from Index method of Home Controller";  
}
```

Tổng quan, một controller action sẽ trả về một cái gì đó gọi là Action Result. Mỗi một kiểu trả về như là HTML, JSON hay string thì đều là Action Result, nó kế thừa từ ActionResult class. ActionResult class là một lớp trừu tượng (abstract class).

Ví dụ để tạo ra một response dạng HTML thì chúng ta dùng ViewResult. Để tạo ra một string hoặc một văn bản thì chúng ta sử dụng ContentResult. Cả ViewResult và ContentResult đều kế thừa từ ActionResult.

ASP.NET Core MVC đã xây dựng sẵn vài kiểu trả về của action result bao gồm:

ViewResult - hiển thị HTML

EmptyResult - hiển thị không kết quả

RedirectResult - trình bày một lệnh chuyển hướng đến một URL mới

JsonResult - trình bày một JSON object được sử dụng cho ứng dụng AJAX.

JavaScriptResult - trả về một đoạn JavaScript

ContentResult - trả về một văn bản

FileContentResult - trả về một file có thể download dạng nhị phân

FilePathResult - trả về một file có thể download dạng đường dẫn

FileStreamResult - trả về một file có thể download dạng stream.

Tổng kết

Bài này chúng ta đã học về vài khái niệm căn bản của ASP.NET Core MVC Controller, controller action, và controller action result.

14. Cơ chế Routing trong ASP.NET Core

Một trong các thành phần quan trọng nhất của kiến trúc MVC là cơ chế routing (định tuyến). Nó là cơ chế quyết định xem Controller nào sẽ được xử lý request nào. Bài này chúng ta sẽ tìm hiểu Routing làm việc ra sao trong ứng dụng ASP.NET Core.

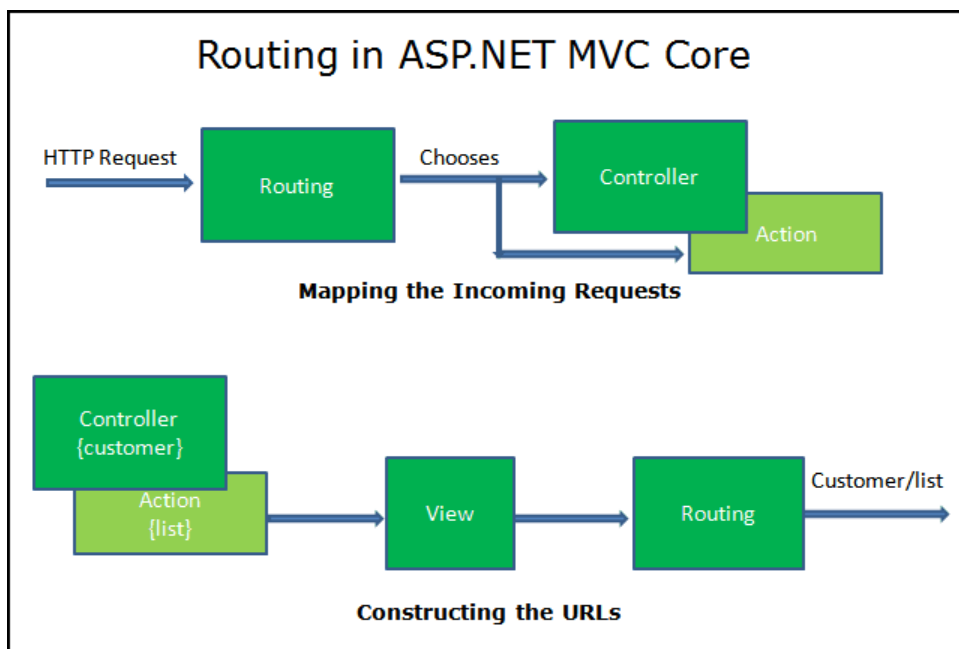
Routing là gì?

Routing là một quá trình khi ASP.NET Core xem xét các URL request gửi đến và "chỉ đường" cho nó đến Controller Actions. Nó cũng được sử dụng để tạo ra URL đầu ra. Quá trình này được đảm nhiệm bởi Routing Middleware. Routing Middleware có sẵn trong thư viện **Microsoft.AspNetCore.Routing**.

Routing có 2 trách nhiệm chính:

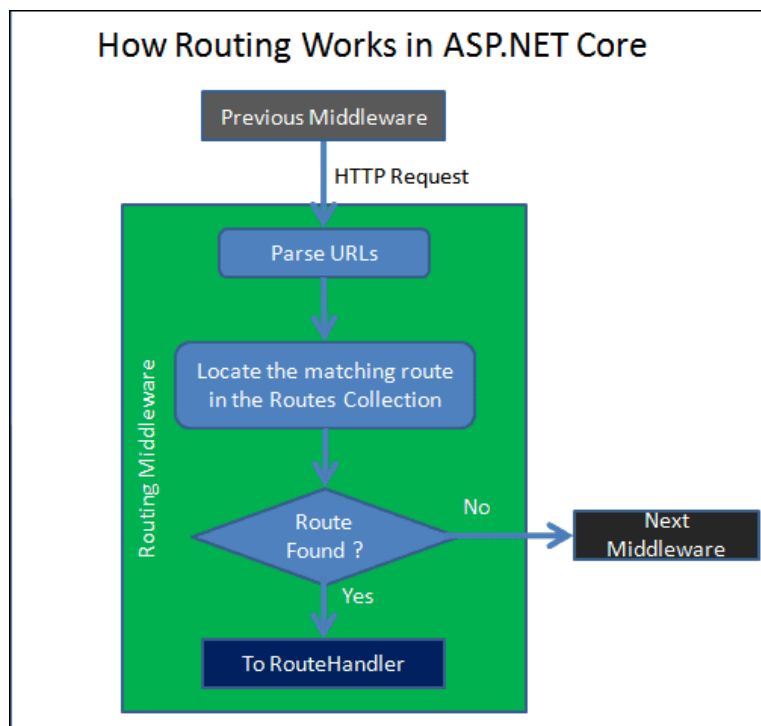
Nó map request đến vào Controller Action.

Tạo ra URL đầu ra tương ứng với Controller action.



Routing làm việc ra sao?

Mô hình dưới đây mô tả cách làm việc của cơ chế Routing trong ASP.NET Core:



Khi request đến thì Routing Middleware sẽ làm những việc sau đây:

Phân tích URL

Tìm kiếm xem có cái Route nào match trong **RouteCollection**

Nếu Route tìm thấy thì đẩy nó sang **RouteHandler**

Nếu không tìm thấy Route nào thì bỏ qua và gọi middleware tiếp theo

Route là gì?

Route tương tự như bản đồ. Chúng ta sử dụng bản đồ để đi đến điểm đích. Tương tự như thế, ứng dụng ASP.NET Core sử dụng Route để đến controller action.

Mỗi Route bao gồm các thông tin như tên, mẫu URL (URL pattern) hay còn gọi là template url, thông tin controller action mặc định và ràng buộc (constraints). URL pattern được so sánh với URL đến xem có đúng mẫu không. Một ví dụ của URL pattern là: `{controller=Home}/{action=Index}/{id?}`

Route được định nghĩa trong `Microsoft.AspNetCore.Routing`.

Route Collection là gì?

Route Collection là một tập hợp tất cả các Route trong ứng dụng. Một ứng dụng sẽ lưu một tập hợp các route ở một nơi duy nhất trong bộ nhớ. Các Route này sẽ thêm vào collection khi ứng dụng khởi động. Route Module sẽ tìm kiếm một Route match với URL request đến trong mỗi một Route của Route Collection. Route Collection được định nghĩa trong **Microsoft.AspNetCore.Routing**.

Route Handler là gì?

Route Handler là một thành phần quyết định sẽ làm gì với Route. Khi cơ chế routing tìm được một Route thích hợp cho một request đến, nó sẽ gọi đến **RouteHandler** và gửi route đó cho **RouteHandler** xử lý. Route Handler là class triển khai từ interface **IRouteHandler**. Trong ASP.NET Core thì Route được xử lý bởi **MvcRouteHandler**.

MvcRouteHandler

Đây là Route Handler mặc định của ASP.NET Core MVC Middleware. **MvcRouteHandler** được đăng ký khi đăng ký MVC Middleware. Bạn có thể ghi đè việc này bằng cách tự tạo cho mình một custom implementation của Route Handler.

MvcRouteHandler được định nghĩa trong namespace: **Microsoft.AspNetCore.Mvc**

MvcRouteHandler có trách nhiệm gọi Controller Factory, sau đó nó sẽ tạo ra một thể hiện của Controller được ghi trong Route. Controller sẽ được nhận và nó sẽ gọi một Action Method và tạo ra View. Vậy là hoàn thành request.

Làm sao để cài đặt Routes

Có hai cách khác nhau để cài đặt route:

Convention-based routing

Attribute routing

Convention-based routing

Convention based routing tạo ra Route dựa trên một loạt các quy tắc được định nghĩa trong file **Startup.cs**

Attribute routing

Tạo các Route dựa trên các attribute đặt trong Controller action. 2 hệ thống routing này có thể cùng tồn tại trong một hệ thống. Đầu tiên chúng ta sẽ tìm hiểu về Convention-based routing, còn Attribute based routing sẽ tìm hiểu sau.



Convention Based Routing

Convention based Routes được cấu hình trong phương thức Configure của Startup class. Routing được xử lý bởi Router Middleware. ASP.NET MVC thêm Routing Middleware vào request pipeline khi sử dụng **app.UseMvc()** hoặc **app.UseMvcWithDefaultRoute()**.

Phương thức **app.UseMvc** tạo ra một thể hiện của class **RouteBuilder**. **RouteBuilder** có một extension method là **MapRoute** cho phép chúng ta thêm **Route** vào Route Collection.

Routing engine được nhận một Route sử dụng API `routes.MapRoute`:

MapRoute Api

```
app.UseMvc(routes =>
{
    routes.MapRoute(
        "default",  Name of the Route
        "{controller=Home}/{action=Index}/{id?}");  URL Pattern
    });
```

Trong ví dụ trên, **MapRoute** tạo một route đơn lẻ nó có tên là default và với URL Pattern của route là **{controller=Home}/{action=Index}/{id?}**

URL Patterns

Mỗi Route phải chứa một URL Pattern. Pattern này sẽ được so sánh với URL request. Nếu pattern đúng với URL thì nó sẽ được sử dụng bởi hệ thống routing để xử lý URL đó. Mỗi một URL Pattern bao gồm một hoặc nhiều phần. Các phần chia tách bởi dấu gạch chéo.

Mỗi phần có thể là một hằng số (constant) hoặc một Route Parameter.

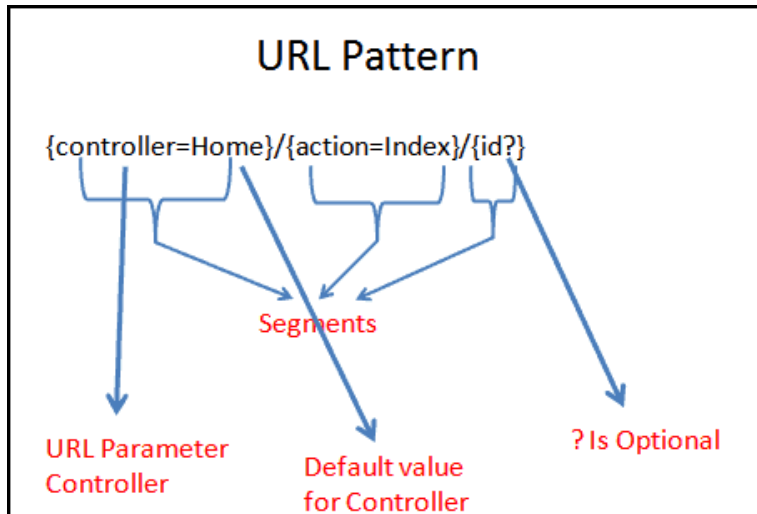
Route Parameter được bao bọc bởi một cặp dấu ngoặc nhọn ví dụ {controller}, {action}.

Route Parameter có thể có giá trị mặc định như **{controller=Home}** khi Home là giá trị mặc định của controller. Một dấu = sẽ gán giá trị cho tên parameter.

Bạn có thể có một thành phần dạng hằng số. Ví dụ: **admin/{controller=Home}/{action=Index}/{id?}**. Ở đây thì "admin" là một hằng tức là một chuỗi cố định phải tồn tại trên URL.

Dấu ? trong **{id?}** chỉ ra là tham số này không bắt buộc. Một dấu ? sau tên tham số chỉ ra tham số đó không yêu cầu phải có giá trị.

URL Pattern **{controller=Home}/{action=Index}/{id?}**. Đăng ký một route có thành phần đầu tiên trên URL là một controller, phần thứ 2 là Action method trong controller đó. Và phần cuối là dữ liệu thêm vào tên là id.

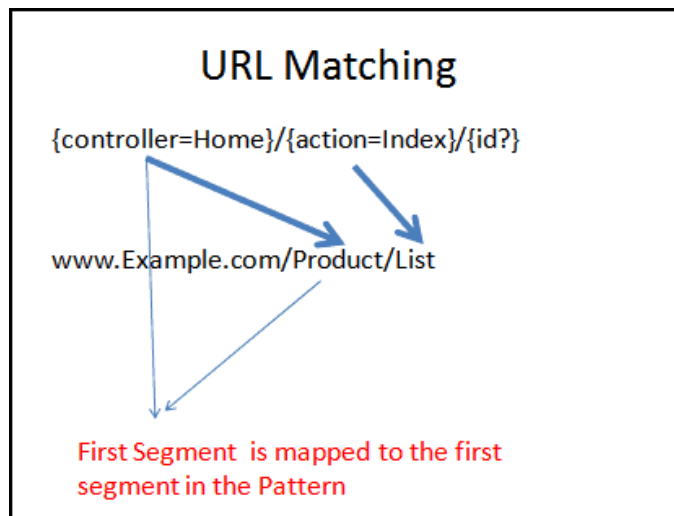


URL Matching

Mỗi phần trong URL request đến sẽ match tương ứng với thành phần của URL Pattern.

Route `{controller=Home}/{action=Index}/{id?}` có 3 thành phần. Phần cuối là tùy chọn. Xem xét ví dụ URL www.example.com/product/list thì URL này có 2 thành phần. URL này vẫn match với pattern ở trên vì phần thứ 3 không yêu cầu.

Routing Engine sẽ nhận diện `{controller}= Product & {action}= List`



URL www.example.com/product cũng match với URL pattern ở trên mặc dù nó chỉ có một thành phần. Vì phần cho action có giá trị mặc định là Index. Nếu không có thành phần tương ứng trong URL và thành phần đó có giá trị mặc định trong Pattern thì giá trị mặc định sẽ được chọn bởi Routing Engine.

Vì thế mà URL được nhận diện như là `{controller}=Product` and `{action}=Index`

Ví dụ www.example.com cũng được match với URL Pattern ở trên vì là thành phần đầu tiên controller cũng có giá trị mặc định là Home. URL này được nhận diện: `{controller}=Home` và `{action}=Index`

URL www.example.com/product/list/10 được nhận diện như là {controller}=Home,{action}=Index và{id}=10.

URL www.example.com/product/list/10/detail thì không đúng vì URL này có 4 thành phần trong khi URL Pattern lại chỉ có 3 thành phần.

Phiên bản trước của ASP.NET thì lại là match ngay cả nếu Controller Action method không tồn tại và ứng dụng trả về lỗi 404.

ASP.NET Core routing engine kiểm tra sự tồn tại của Controller và Action method cho mỗi route. Nếu không có controller và action method tương ứng trong ứng dụng thì cũng không được match ngay cả khi Route tồn tại.

Routeing trong Action

Hãy xây dựng một ứng dụng ASP.NET Core kiểm tra xem routing làm việc ra sao nhé.

Tạo một ứng dụng ASP.NET Core sử dụng .NET Core 2.2. Chọn Empty Project và đặt tên là MVCController.

Mở **Startup.cs** và mở phương thức **ConfigureServices** ra sau đó thêm MVC service như dưới:

```
public void ConfigureServices(IServiceCollection services){  
    services.AddMvc();  
}
```

Giờ hãy mở phương thức **Configure** ra trong **Startup.cs** và đăng ký MVC Middleware sử dụng **app.UseMvc**.

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env) {  
    if (env.IsDevelopment()) {  
        app.UseDeveloperExceptionPage();  
    }  
    app.UseMvc();  
    app.Run(async (context) => {  
        await context.Response.WriteAsync("Failed to Find Route");  
    });  
}
```

Phương thức **app.UseMvc** không được đăng ký bất cứ route nào. Mà **app.Run** lại là một Terminating Middleware (middleware ngắt) thì ví dụ trên sẽ hiển thị dòng "Failed to Find Route". Middleware này sẽ chỉ chạy khi **app.UseMvc** không tìm thấy bất cứ route nào.

Giờ hãy tạo thư mục **Controllers** trong thư mục gốc của project. Chuột phải vào thư mục Controllers chọn **Add Controller**. Chọn **"MVC - Controller Empty"**. Đặt tên là **HomeController**.

Giờ hãy mở **HomeController** ra và thay đổi phương thức **Index** như sau:

```
public string Index() {
    return "Hello from Index method of Home Controller";
}
```

Bạn có thể tham khảo bài viết [Xây dựng ứng dụng ASP.NET Core MVC đầu tiên](#) để tạo ứng dụng.

Chạy ứng dụng và gọi đến các URL như: `/`, `/Home`, `/Home/Index`.

Tất cả các trường hợp trên sẽ nhận dòng chữ "Fails to Find a Route". Vì nó không được đăng ký bất cứ route nào.

Đăng ký Route

Vào phương thức **Configure** của class **Startup** và thay đổi **app.UseMvc** như sau:

```
app.UseMvc(routes => {
    routes.MapRoute("default",
        "{controller}/{action}");
});
```

Chúng ta không chỉ ra bất cứ mặc định nào. Vì cả Controller và Action đều có trên URL:

URL ĐÚNG? NHẬN DIỆN

/	Không	
/Home	Không	
/Home/Index	Có	Controller=Home Action=Index

Giờ hãy thử route này:

```
app.UseMvc(routes => {
    routes.MapRoute("default",
        "{controller}/{action=index}");
});
```

Giờ chúng ta có giá trị mặc định trên URL Parameter:

URL ĐÚNG? NHẬN DIỆN

/	Không	
/Home	Có	Controller=Home Action=Index

URL ĐÚNG? NHẬN DIỆN

/Home/Index	Có	Controller=Home Action=Index
-------------	----	---------------------------------

Giờ hãy thêm mặc định cho Controller

```
routes.MapRoute("default", "{controller=Home}/{action=Index}");
```

URL CÓ ĐÚNG? NHẬN DIỆN

/	Có	Controller=Home Action=Index
/Home	Có	Controller=Home Action=Index
/Home/Index	Có	Controller=Home Action=Index

Giờ hãy thử route này:

```
routes.MapRoute("default", "{admin}/{controller=Home}/{action=Index}");
```

Chúng ta thêm Route Parameter admin cho route. Chú ý là Route Parameter được đóng bởi cặp ngoặc nhọn. Giờ hãy test lại URL:

URL CÓ ĐÚNG? NHẬN DIỆN

/	Không Không có mặc định cho admin. Vì thế phần này bắt buộc	
/Home	Có Thành phần đầu tiên match với admin	Admin=Home Controller=Home Action=Index
/Abc	Có	Admin=Abc Controller=Home Action=Index
/Home/Index	Không Admin=Home Controller=Index Không có controller nào là IndexController, vì thế không đúng	
/Xyz/Home	Có	Admin=Xyz Controller=Home Action=Index
/Admin/Home	Có	Admin=Admin Controller=Home Action=Index

Giờ hãy thử:

```
routes.MapRoute("default",
    "admin/{controller=Home}/{action=Index}");
});
```

Sự khác nhau giữa các route ở trên và route này là admin được định nghĩa dạng Constant (không có dấu ngoặc nhọn). Nghĩa là phần đầu tiên phải là từ "admin"

URL	CÓ ĐÚNG?	NHẬN DIỆN
/	Không, vì thành phần đầu tiên bắt buộc	
/Home	Không, vì thành phần đầu tiên phải chứa từ admin	
/Abc	Không, vì thành phần đầu tiên phải chứa từ admin	
/Admin	Có	Controller=Home Action=Index
/Admin/Home	Có	Controller=Home Action=Index

Các tham số cho Controller Action Method

Giờ hãy xem xét Route dưới đây. Đây là Route mặc định được đăng ký khi chúng ta sử dụng `app.UseMvcWithDefaultRoute`.

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

Nó có 3 thành phần và phần thứ 3 có tên là id, nó không bắt buộc. Phần id này có thể được gán như một tham số vào Controller action method.

Bất cứ tham số trên route nào trừ {controller} và {action} có thể được gán như các tham số vào action method.

Thay đổi phương thức Index của HomeController:

```
public string Index(string id) {
    if (id != null) {
        return "Received " + id.ToString();
    } else {
        return "Received nothing";
    }
}
```

Một request cho `"/Home/Index/10"` sẽ match với rouet trên và giá trị 10 sẽ được gán vào tham số id của action Index.

Route mặc định

Route mặc định có thể được chỉ ra bằng 2 cách. Cách đầu tiên là sử dụng dấu bằng (=) (`{controller=Home}`) như là các ví dụ trên.

Cách khác là sử dụng tham số thứ 3 của phương thức `MapRoute`.

```
routes.MapRoute("default",  
    "{controller}/{action}",  
    new { controller = "Home", action = "Index" });
```

Chúng ta tạo ra một thể hiện của một kiểu nặc danh, nó chứa các thuộc tính được trình bày trên URL. Các giá trị đó trở thành giá trị mặc định của URL Parameter.

Route trên tương tự như route `{controller=Home}/{action=Index}`.

Multiple Route

Trong ví dụ trên chúng ta chỉ sử dụng có 1 route. Chúng ta có thể cấu hình ASP.NET Core xử lý bất cứ route nào:

```
app.UseMvc(routes => {  
    routes.MapRoute("secure",  
        "secure",  
        new { Controller = "Admin", Action = "Index" });  
  
    routes.MapRoute("default",  
        "{controller=Home}/{action=Index}");  
});
```

Ví dụ trên có 2 route. Mỗi route phải có một tên duy nhất. Chúng ta đặt là "secure" và "default".

Route đầu tiên rất thú vị. Nó chỉ có một thành phần. Chúng ta cài đặt giá trị mặc định cho Controller và Action method trên route này. Controller và Action mặc định là phương thức Index của AdminController.

Tạo một AdminController.cs trong thư mục Controllers. Thay đổi nội dung phương thức Index:

```
public string Index() {  
    return "Hello from Index method of Admin Controller";  
}
```

Giờ chúng ta hãy thử chạy các URL sau:

URL	CÓ ĐÚNG?	NHẬN DIỆN
/	Có	Controller=Home Action=Index
/Secure	Có	Controller=Admin Action=Index
/Secure/Test	Không	
/Admin	Có Sẽ đến AdminController không qua rouet đầu tiên, những sẽ là route thứ 2	Controller=Admin Action=Index

Vấn đề thứ tự, Route trước sẽ được dùng

Thứ tự route nào được đăng ký rất quan trọng. URL Matching bắt đầu chạy từ trên xuống của tập Route Collection và tìm xem có Route nào match với URL không. Nó sẽ dừng lại khi tìm thấy Route match đầu tiên.

```
routes.MapRoute("Home",
    "{home}",
    new { Controller = "Home", Action = "Index" });
```

```
routes.MapRoute("secure",
    "secure",
    new { Controller = "Admin", Action = "Index" });
```

Route đầu tiên có URL Parameter {home} và match với tất cả. Route thứ 2 có chứa từ "secure" là một route cụ thể hơn.

Khi bạn dùng URL "/secure", nó không gọi đến AdminController nhưng thay vào đó lại là HomeController. Điều này bởi vì URL /secure match với route đầu tiên mất rồi.

Để làm cho route này chạy, hãy chuyển route secure lên trên Home Route.

```
routes.MapRoute("secure",
    "secure",
    new { Controller = "Admin", Action = "Index" });
```

```
routes.MapRoute("Home",
    "{home}",
    new { Controller = "Home", Action = "Index" });
```

Tổng kết

Bài viết này chúng ta đã học về cơ chế routing trong ASP.NET MVC Core. Chúng ta cần đăng ký Rouet sử dụng phương thức MapRoute của Routing Middleware. Routing Engine sau đó sẽ match URL đến với Route Collection và tìm ra một Route match với nó. Nó sẽ gọi RouteHandler (mặc định là MvcRouteHandler). MvcRouteHandler sau đó gọi Controller tương ứng trong Route.

15. Attribute Routing trong ASP.NET Core

Bài viết này chúng ta sẽ cùng tìm hiểu Attribute Routing trong ASP.NET Core. Bài trước chúng ta đã tìm hiểu về Routing và làm sao để ASP.NET Core map được request vào controller và action. Có 2 cách để bạn có thể cấu hình được routing trong ASP.NET Core.

Convention based Routing

Attribute-based Routing

Chúng ta đã tìm hiểu về Convention based Routing trong bài trước. Giờ chúng ta sẽ tìm hiểu về **Attribute based routing** nhé.

Attribute based routing là gì?

Attribute routing sử dụng các attribute để định nghĩa trực tiếp các route trong controller action. Attribute routing giúp bạn tăng sự tiện lợi hơn khi làm việc với URL trong ứng dụng.

Trong convention based routing thì tất cả các routing được cấu hình trong class **Startup**. Trong Attribute Routing thì lại cấu hình trong Controller.

Làm sao để cài đặt Attribute Routing

Routes được thêm vào sử dụng Route Attribute trong controller action. Attribute Route có 3 tham số, URL pattern, tên và thứ tự. Trong ví dụ dưới đây chúng ta thêm attribute route cho action method Index và gán "Home" làm URL Pattern:

```
public class HomeController: Controller{  
    [Route("Home")]  
    public string Index(){  
        return "Hello from Index method of Home Controller";  
    }  
}
```

Ví dụ trên, URL **/Home** gọi đến method **Index** của controller **Home**. Bởi vì Route attribute trả URL **/Home** đến method này.

Sử dụng Attribute Routing ra sao?

Mở project được tạo từ bài trước. Giờ bạn hãy mở **Startup.cs** và tìm phương thức **Configure**. Thay đổi code như sau:

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
```

```

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

    app.UseMvc();
}

```

Giờ chúng ta đã sử dụng **app.UseMvc()** để đăng ký MVC Middleware với không có route nào. Giờ hãy mở HomeController.cs. Sau đó thêm **[Route("Home")]** vào method Index:

```

[Route("Home")]

public string Index() {
    return "Hello from Index method of Home Controller";
}

```

Khi chúng ta gọi URL /Home thì Index method của HomeController được gọi:

URL	CÓ TÌM THẤY?	NHẬN DIỆN
/	Không	
/Home	Có	Controller=Home Action=Index
/Home/Index	Không	

Giờ thay đổi sang URL Pattern **"Home/Index"**:

```

[Route("Home/Index")]

public string Index() {
    return "Hello from Index method of Home Controller";
}

```

URL	TÌM THẤY?	NHẬN DIỆN
/	Không	
/Home	Không	
/Home/Index	Có	Controller=Home Action=Index

Thay đổi tên của Action Method

Chú ý rất quan trọng là URL Pattern không cần phải đúng với tên Controller và tên Action method. Ví dụ:

```

[Route("Say/Hello")]

```



```
public string Index(){
    return "Hello from Index method of Home Controller";
}
```

URL	TÌM THẤY?	NHẬN DIỆN
/	Không	
/Home	Không	
/Home/Index	Không	
/Say/Hello	Có	Controller=Home Action=Index

Nhiều Routes

Bạn cũng có thể đặt nhiều Route cho một Action method đơn sử dụng nhiều Routing Attribute như sau:

```
[Route("")]
[Route("Home")]
[Route("Home/Index")]
public string Index(){
    return "Hello from Index method of Home Controller";
}
```

URL	TÌM THẤY?	NHẬN DIỆN
/	Có	Controller=Home Action=Index
/Home	Có	Controller=Home Action=Index
/Home/Index	Có	Controller=Home Action=Index
/Say/Hello	Không	

Token Replacement

Attribute Routing dễ dàng hơn cho chúng ta bằng cách sử dụng tokens dạng như một biến thay thế. Ví dụ: [area], [controller] và [action]. Những token này sẽ được thay thế bởi các giá trị thực tế trong Route Collection:

```
[Route("")]
[Route("[controller]")]
```

```
[Route("[controller]/[action]")]
```

```
public string Index() {
```

```
    return "Hello from Index method of Home Controller";
```

```
}
```

Các token trong URL Pattern đảm bảo rằng tên Controller và tên Action nằm ở đó sẽ được đồng bộ với tên của class controller tự động.

Tham số gán vào Action method

Cũng giống như Convention based Routing, chúng ta có thể định nghĩa các URL Parameter tức là các tham số trên URL trong cặp ngoặc nhọn. Nó có thể được gán vào như là một tham số của Action method:

```
[Route("")]
```

```
[Route("[controller]")]
```

```
[Route("[controller]/[action]/{id?}")]
```

```
public string Index(string id) {
```

```
    if (id != null) {
```

```
        return "Received " + id.ToString();
```

```
    } else {
```

```
        return "Received nothing";
```

```
    }
```

```
}
```

Trong ví dụ trên, **id** là một tham số tùy chọn. Bất cứ giá trị nào được nhận được đều được truyền vào method **Index** như là một tham số.

Kết hợp các Route

Chúng ta có thể chỉ ra route attribute trong Controller class. Bất cứ URL Pattern nào được định nghĩa trên Controller thì đều được gán vào URL Pattern của Action method:

```
[Route("Home")]
```

```
public class HomeController : Controller {
```

```
    [Route("Index")]
```

```
    public string Index() {
```

```
        return "Hello from Index method of Home Controller";
```

```
    }
```

```
}
```

Đoạn code trên sẽ thỏa mãn Route **"Home/Index"**. Nếu Pattern bắt đầu với / thì nó được xem xét như đường dẫn tuyệt đối và không được kết hợp với URL Pattern.

Nó sẽ không làm việc cho URL **"Home/Index"**:

```
[Route("Home")]

public class HomeController : Controller {

    [Route("/Index")]

    public string Index() {

        return "Hello from Index method of Home Controller";

    }

}
```

Trộn Routing

Bạn cũng có thể sử dụng cả hai cơ chế convention-based routing và attribute based routing trong một project. Tuy nhiên nếu bạn định nghĩa attribute routing trong action thì convention-based routing sẽ không thể sử dụng trong action. Hay nói cách khác thì attribute-based routing sẽ được ưu tiên trước convention-based routing.

Action Verbs

Attribute route cũng có thể được dùng với HTTP Verb attribute như HttpGet, HttpPost...

```
[HttpGet("")]

[HttpGet("Home")]

[HttpGet("Home/Index")]

public string Index() {

    return "Hello from Index method of Home Controller";

}
```

Tổng kết

Chúng ta đã tìm hiểu cách thứ 2 để định nghĩa cơ chế routing cho ứng dụng.

16. Route Constrains trong ASP.NET Core

Route Constrains giúp chúng ta lọc và giới hạn các giá trị không mong muốn truyền vào controller action. Nó được kiểm tra bởi các ràng buộc áp dụng cho giá trị truyền vào URL Parameter. Ví dụ bạn muốn Route Engine của mình kiểm tra chính xác giá trị id truyền vào tham số **id** là kiểu số thay vì kiểu khác.

Route Constrains làm việc như thế nào?

Có 2 cách để bạn thêm Constrain vào URL Parameter:

Thêm trực tiếp vào URL Parameter

Sử dụng tham số Constrain trong phương thức MapRoute.

Inline Constraint được thêm vào URL Parameter sau dấu hai chấm (:):

```
routes.MapRoute("default", "{controller=Home}/{action=Index}/{id:int}");
```

Một khi Routing Engine tìm thấy một Route đúng với URL, nó sẽ gọi Route Constraint để kiểm tra mỗi thành phần của URL xem có thỏa mãn không? Các constraint (ràng buộc) này chỉ đơn giản là trả về kết quả có thỏa mãn hay không. Nếu không có nghĩa là tham số không được chấp nhận.

Route Constraints

Mở project ra, cái mà bạn đã tạo trong bài **Routing**. Vào phương thức **Configure** và dùng đoạn code dưới đây:

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

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

Giờ hãy copy đoạn code dưới đây vào method **Index** trong **HomeController**:

```
public class HomeController : Controller
{
    public string Index(int id) {
        return "I got " + id.ToString();
    }
}
```

Sau khi Request với URL là **"/Home/Index/10"** được tìm thấy trong route. Giá trị **10** sẽ được truyền vào như một tham số đến action method Index và bạn thấy dòng chữ "I got 10" trên trình duyệt.

Một request khác có URL `"/Home/Index/Test"` cũng đúng với route. Router thử chuyển giá trị `"Test"` sang giá trị `0` và truyền vào nó như một tham số đến Index action. Bạn sẽ nhận được dòng chữ `"I got 0"` trên trình duyệt.

Bạn có thể tránh điều này bằng cách sử dụng Route Constraint với tham số `id`.

Inline Constraint

Inline Constraint được thêm vào sau URL Parameter và được tách với dấu hai chấm. Ví dụ bạn có thể đảm bảo chỉ các giá trị số nguyên được chấp nhận:

```
routes.MapRoute("default", "{controller=Home}/{action=Index}/{id:int?}");
```

Ràng buộc `int` kiểm tra xem các giá trị của `id` phải chuyển được sang số nguyên. Thành phần `id` là tùy chọn không bắt buộc. Vì thế route sẽ match nếu như `id` không được chỉ ra, nhưng nếu một khi nó được chỉ ra thì nó phải là một giá trị số nguyên. Với `int` constraint thì Request `"/Home/Index/Test"` sẽ không match với route.

Sử dụng phương thức Constraint của MapRoute

Các constraints cũng được chỉ ra sử dụng các tham số constraints trong phương thức **MapRoute**. Để làm điều này bạn cần thêm namespace: **Microsoft.AspNetCore.Routing.Constraints**.

```
using Microsoft.AspNetCore.Routing.Constraints;
```

```
app.UseMvc(routes =>
{
    routes.MapRoute("default",
        "{controller}/{action}/{id}",
        new { controller = "Home", action = "Index" },
        new { id = new IntRouteConstraint() });
});
```

Chúng ta tạo một thể hiện của kiểu **Anonymous**, nó chứa các thuộc tính có tên giống như URL Parameter và các ràng buộc được áp dụng. Các thuộc tính này được gán cho thể hiện của class **Constraint**.

Trong ví dụ trên chúng ta tạo một class ẩn danh. Nó có thuộc tính `id`, match với `id` trong URL Parameter. Thể hiện của **IntRouteConstraint** được gán cho thuộc tính `id`.

Constraints trong Attribute Routing

Bạn cũng có thể đạt được điều này khi sử dụng Attribute routing:

```
[Route("Home/Index/{id:int}")]
public string Index(int id){
    return "I got " + id.ToString();}
```

Sử dụng Constraints ở đâu?

Constraints được sử dụng cho việc kiểm tra đầu vào nhưng đó không phải là lý do nó tồn tại. Kiểm tra đầu vào (input validation) không phải được xử lý bởi Route Constraints. Thay vào đó Controller phải kiểm tra đầu vào và gửi thông báo lỗi tương ứng cho người dùng. Nếu bạn sử dụng Route Constraints để kiểm tra dữ liệu đầu vào thì người dùng sẽ nhìn thấy lỗi 404 (Not Found).

Route constraints nên được sử dụng để hỗ trợ Route Engine phân biệt giữa 2 route tương tự. Ví dụ:

```
app.UseMvc(routes => {  
    routes.MapRoute("default",  
        "post/{id:int}",  
        new { controller = "Post", action = "PostsByID" });  
    routes.MapRoute("anotherRoute",  
        "post/{id:alpha}",  
        new { controller = "Post", action = "PostsByPostName" });  
});
```

Chúng ta có 2 Route nhìn giống nhau về pattern. Vì chúng đều là "post/{id:int}" và "post/{id:alpha}". Với hai kiểu dữ liệu là int và alpha (chỉ chấp nhận chữ cái), chúng ta có thể chỉ cho Routing Engine chọn **PostByID** Action method nếu giá trị là số và **PostByPostName** nếu giá trị id là kiểu chữ cái.

Danh sách của Route Constraint

Namespace **Microsoft.AspNetCore.Routing.Constraints** định nghĩa một tập các class có thể sử dụng để định nghĩa các constraint riêng lẻ.

Constraints để kiểm tra kiểu dữ liệu

Đây là các constraints để kiểm tra kiểu dữ liệu:

CONSTRAINT	CÚ PHÁP	CLASS	GHI CHÚ
int	{id:int}	IntRouteConstraint	Kiểm tra tham số chỉ được là số nguyên 32 bit.
alpha	{id:alpha}	AlphaRouteConstraint	Kiểm tra tham số chỉ được là ký tự thuộc bảng chữ cái tiếng anh A-Z
bool	{id:bool}	BoolRouteConstraint	Kiểm tra tham số chỉ được là giá trị logic true hoặc false.
datetime	{id:datetime}	DateTimeRouteConstraint	Kiểm tra tham số chỉ được là giá trị DateTime
decimal	{id:decimal}	DecimalRouteConstraint	Kiểm tra giá trị tham số chỉ được là kiểu thập phân
double	{id:double}	DoubleRouteConstraint	Cho phép giá trị tham số chỉ được là kiểu số thực 64 bit
float	{id:float}	FloatRouteConstraint	Cho phép tham số chỉ là kiểu số thực dấu chấm động

CONSTRAINT	CÚ PHÁP	CLASS	GHI CHÚ
guid	{id:guid}	GuidRouteConstraint	Chỉ cho phép kiểu Guid

Constraints để kiểm tra giá trị/ miền giá trị và độ dài:

CONSTRAINT	CÚ PHÁP	CLASS	GHI CHÚ
length(length)	{id:length(12)}	LengthRouteConstraint	Cho phép giá trị có độ dài trong khoảng
maxlength(value)	{id:maxlength(8)}	MaxLengthRouteConstraint	Cho phép giá trị có độ dài tối đa nằm trong dấu ngoặc đơn.
minlength(value)	{id:minlength(4)}	MinLengthRouteConstraint	Cho phép giá trị có độ dài tối thiểu nằm trong dấu ngoặc đơn.
range(min,max)	{id:range(18,120)}	RangeRouteConstraint	Cho phép giá trị nằm trong khoảng
min(value)	{id:min(18)}	MinRouteConstraint	Cho phép giá trị phải lớn hơn hoặc bằng giá trị trong ngoặc
max(value)	{id:max(120)}	MaxRouteConstraint	Cho phép giá trị phải nhỏ hơn hoặc bằng giá trị trong ngoặc

Constraints kiểm tra sử dụng Regular Expression

CONSTRAINT	CÚ PHÁP	CLASS	GHI CHÚ
regex(expression)	{ssn:regex(^\\d{{3}}-\\d{{2}}-\\d{{4}}\$)}	RegexRouteConstraint	Chứa giá trị thỏa mãn regular expression.

Ví dụ về Regular Express Route Constraint

Trong ví dụ dưới đây, chúng ta sử dụng biểu thức chính quy để hạn chế giá trị Year chỉ có 4 số:

```
app.UseMvc(routes => {
    routes.MapRoute("default",
        "{controller}/{action}/{year:regex(^\\d{{4}}$)}",
        new { controller = "Home", action = "Index" });
});
```

Cập nhật Index method của HomeController.cs:

```
public class HomeController : Controller {
    public string Index(int year) {
        return "Year = " + year.ToString();
    }
}
```

Một request với URL "Home/Index/2017" sẽ match với route nhưng **"/Home/Index/20"** thì không. Chú ý là biểu thức chính quy phải được thoát. Ví dụ: \, {, }, [,] cần được đặt nó gấp đôi lên để thoát tránh lỗi cho parameter.

Vì thế `^d{4}$` trở thành `^\d{4}$`.

Kết hợp các constraints

Nhiều constraints có thể được kết hợp bởi dấu hai chấm chia tách:

`"/{id:alpha:minlength(6)?}"`

Hoặc sử dụng phương thức **Constraints** trong **MapRoute**:

```
Using Microsoft.AspNetCore.Routing.CompositeRouteConstraint;

constraints: new {
    id = new CompositeRouteConstraint(
        new IRouteConstraint[] {
            new AlphaRouteConstraint(),
            new MinLengthRouteConstraint(6)
        }
    )
}
```

Tổng kết

Route Constraints rất hữu dụng trong việc phân biệt giữa các route giống nhau về URL Pattern. Nó giúp chúng ta giới hạn các giá trị không mong muốn truyền đến controller action.

17. Action Selectors & Action Verbs trong ASP.NET Core

Bài viết này chúng ta sẽ tìm hiểu thành phần Action Selector và vai trò của nó trong khi chọn controller action. Trong bài trước chúng ta có tìm hiểu về Routing trong ASP.NET Core. Chúng ta đã học về Routing Engine để chọn controller và action tương ứng để thực thi request. Các thành phần như Action selector, Action Name, Non-Action và Action Verb sẽ giúp chúng ta thêm công cụ điều khiển cho quá trình URL Matching Process.

Action Selector là gì?

Action selector là một attribute có thể được áp dụng cho controller action. Các attribute này giúp Routing Engine chọn đúng action method để xử lý request. ASP.NET Core bao gồm 3 kiểu Action Selector: **Action Name**, **Non Action**, **Action Verbs**

Action Name

ActionName attribute định nghĩa tên của một action. Routing engine sẽ sử dụng tên này thay vì tên phương thức để match với action name trong routing. Bạn sẽ dùng attribute này khi bạn muốn một alias cho tên phương thức:


```
[ActionName("Modify")]
```

```
public string Edit(){ return "Hello from Edit Method";}
```

Trong code phía trên, Action Name cho hành động Edit được thay đổi sang Modify sử dụng ActionName attribute. Chúng ta có thể không còn gọi phương thức này sử dụng tên Edit. Chúng ta phải sử dụng tên mới là "Modify" trong URL. Nhớ là bạn cũng có thể sử dụng route attribute để thay đổi Action Name:

```
public class HomeController : Controller
```

```
{
```

```
[Route("Home/Modify")]
```

```
public string Edit()
```

```
{
```

```
return "Hello from Edit Method";
```

```
}
```

```
}
```

Non Action

Public method trong class controller được gọi như Action methods. Các phương thức này có thể được gọi bởi bất cứ ai ở bất cứ đâu trên thế giới đơn giản chỉ cần gõ URL lên trình duyệt. Bạn có thể cho Routing Engine biết đó là một phương thức đặc biệt không phải là một Action method bằng cách đặt attribute NonAction như sau:

```
public class HomeController : Controller {
```

```
[NonAction]
```

```
public string Edit() {
```

```
return "Hello from Edit Method";
```

```
}
```

```
}
```

Action Verbs

Action verbs selector được sử dụng khi bạn muốn điều khiển action method dựa trên HTTP Request method. Điều này được đảm nhiệm sử dụng tập các attribute bởi MVC, ví dụ như HttpGet và HttpPost. Nó được gọi là Http Attributes.

Một số HTTP Verbs có sẵn trong ứng dụng ASP.NET Core. Chúng là GET, POST, PUT, DELETE, HEAD, OPTIONS, PATCH. Mỗi verbs này kết hợp với HTTP Method Attributes được định nghĩa trong namespace Microsoft.AspNetCore.Mvc.Routing.

Bạn có thể áp dụng các attribute này cho Controller action method. Khi khách hàng gửi request sử dụng một verb cụ thể, routing engine sẽ tìm controller action với một attribute tương ứng và gọi nó.

HTTP Attribute cho phép chúng ta định nghĩa 2 phương thức với cùng tên nhưng khác kiểu response với HTTP Verb khác nhau.

Ví dụ tốt nhất là phương thức Edit. Một phương thức Edit trả về một Get Request và tạo ra Edit Form. Phương thức khác nhận Post request và cập nhật database.

```
[HttpGet]

public ActionResult Edit(string id) {

    //Return the Edit Form

    return View(); }
```

```
[HttpPost]

public ActionResult Edit(Model model)

{

    //Update the database here

}
```

Gán routing value trong HTTP action verbs

Trong bài hướng dẫn trên Attribute routing chúng ta đã tìm hiểu làm sao để cấu hình route trong Route Attribute. Thay vào đó bạn có thể sử dụng HTTP Action verb để làm điều này:

```
[HttpGet("")]

[HttpGet("Home")]

[HttpGet("Home/Index")]

public string Index() {

    return "Hello from Index method of Home Controller";

}
```

HTTP Attribute

Dưới đây là danh sách HTTP Attributes

HttpGet

HttpGet attribute giới hạn Action method chỉ chấp nhận các request sử dụng GET verb. Các tham số trên URL (query string) tự động được thêm vào như các tham số. HttpGet được dùng để nhận một resource từ server.

```
[HttpGet]

public IEnumerable<product> GetAll()

{

    return db.Products.ToList();

}
```

```
}
```

```
[HttpGet("{id}")]
```

```
public IActionResult GetById(long id)
```

```
{
```

```
    var product= db.Products.Where(e => e.ProductID == id);
```

```
    return View(product);
```

```
}
```

HttpPost

HttpPost attribute giới hạn action method chấp nhận HTTP Request sử dụng Post verb. Post verb được dùng để tạo mới bản ghi.

```
[HttpPost]
```

```
public IActionResult Create(Product product)
```

```
{
```

```
    if (product == null)
```

```
    {
```

```
        return BadRequest();
```

```
    }
```

```
    db.Products.Add(product);
```

```
    db.SaveChanges();
```

```
    return RedirectToAction("Index");
```

```
}
```

HttpDelete

HttpDelete attribute giới hạn Action method chỉ chấp nhận HTTP Request sử dụng Delete verb. Delete verb được dùng để xóa tài nguyên đang tồn tại

HttpPut

HttpPut attribute giới hạn action method chỉ chấp nhận các HTTP Request sử dụng Put verb. Put verb được dùng để cập nhật hoặc tạo mới tài nguyên.

HttpHead

HttpHead attribute giới hạn action method chỉ chấp nhận HTTP Request sử dụng Head verb. Head verb được dùng để nhận các HTTP Header. Phương thức này được định danh cho GET ngoại trừ các server không trả về message body.

HttpOptions

HttpOptions attribute giới hạn action method chỉ chấp nhận các request sử dụng Options verb. Method này nhận thông tin về tùy chọn giao tiếp được hỗ trợ bởi web server.

HttpPatch

HttpPatch attribute giới hạn action method chỉ nhận các HTTP Request sử dụng Options verb. Method này sử dụng cho toàn bộ hoặc một phần việc cập nhật tài nguyên.

Sử dụng nhiều Action Verbs

AcceptVerbs attribute cho phép sử dụng nhiều action verb trên action method:

```
[AcceptVerbs(HttpVerbs.Get | HttpVerbs.Post)]
```

```
public ActionResult AboutUs() { return View(); }
```

Tổng kết

Action Name, Non Action và Action Verbs (Được biết đến như là Action Selector) cho chúng ta thêm công cụ điều khiển trong URL Matching Process. Action Verbs hoặc HTTP Attributes được dùng khi bạn muốn điều khiển Action method dựa trên HTTP Request method.

18. Action Result trong ASP.NET Core

Bài viết này chúng ta sẽ tìm hiểu làm sao để định dạng một response trả về trong Action method. ASP.NET Core cung cấp một tập các API gọi là Action Result để tạo ra các định dạng response cho từng nhu cầu. Hãy cùng tìm hiểu về Action Result là gì và sự khác nhau giữa các loại Action Result có sẵn nhé.

Action Result là gì?

Controller Action sẽ trả về kết quả cho client. Client có thể mong muốn một kết quả đơn giản như là chuỗi hay số nguyên hoặc một kết quả phức tạp như là JSON hay HTML view hoặc file để download.

Controller trong ASP.NET Core chỉ đơn giản là các class C#. Nó không cần phải kế thừa từ bất cứ base class nào. Nhưng ASP.NET Core cung cấp một class Controller nó kế thừa từ một ControllerBase class. Điều này giúp class cung cấp rất nhiều các method hữu ích, nó giúp Controller làm việc dễ dàng hơn. Vì thế thông thường các controller của chúng ta đều kế thừa từ ControllerBase class.

Controller Base class triển khai các loại Action Result khác nhau sẵn có giúp xây dựng các loại kết quả trả về khác nhau cho client. Ví dụ, ViewResult trả về một HTML response. Một RedirectResult chuyển hướng đến URL khác. Content Result trả về một chuỗi văn bản. Các kiểu trả về này được biết đến là Action Result.

ActionResult và IActionResult

IActionResult là một interface nó định nghĩa một khuôn mẫu cho toàn bộ các Action Result của một action method. ActionResult là một abstract base class triển khai interface IActionResult. Action result như ViewResult, PartialViewResult hay JsonResult...đều kế thừa từ ActionResult base class.

Tại sao lại dùng Action Result?

Không cần phải sử dụng Action Result trong ASP.NET Core. Các Controller không cần phải kế thừa từ Controller class. Bạn có thể thao tác trực tiếp với đối tượng HTTP Response trong Controller và quyết định xem trả về gì?

Ví dụ dưới đây mô tả cách inject đối tượng HttpContext vào Constructor của Controller và sau đó sử dụng nó để tạo ra Response:

```
public class HomeController {  
    HttpContext ctx;  
  
    public HomeController(IHttpContextAccessor _ctx) {  
        ctx = _ctx.HttpContext;  
    }  
  
    public async void Index() {  
        //Set Status Code  
        ctx.Response.StatusCode = 200;  
  
        //Set Content Type  
        ctx.Response.ContentType = "text/html";  
  
        //Create Response  
        ctx.Response.Headers.Add("SomeHeader", "Value");  
  
        byte[] content = Encoding.ASCII.GetBytes($"<html><body>Hello  
World</body></html>");  
  
        //Send it to the Client  
        await ctx.Response.Body.WriteAsync(content, 0, content.Length);  
    }  
}
```

Mặc dù cách tiếp cận này vẫn làm việc bình thường nhưng nó không phải là cách tốt nhất để tạo ra Response từ Controller. Xử lý tất cả các controller như thế này sẽ nhàm chán, dễ sinh lỗi và khó bảo trì. Action Result đóng gói tất cả các chi tiết vào trong nó. Nó có nhiều các tính năng hữu dụng giúp dễ dàng hơn trong việc tạo response.

Tạo ra Response trực tiếp từ Controller class như ví dụ trên. Nó sẽ làm khó cho Unit Test. Unit Test một Controller class sẽ cần phải mock các triển khai của Response object. Để test kết quả chúng ta lại cần chuyển nó sang HTML Response.

Action Result sẽ có tất cả những gì cần thiết để chạy Controller và Action method. Đối tượng context sẽ dễ dàng được giả lập thông qua base class của nó. Bạn không cần phải chuyển bất cứ kết quả sang HTML để kiểm tra kết quả của một action method. Bạn có thể kiểm tra ngay đối tượng ActionResult để đảm bảo rằng bạn nhận kết quả như mong muốn.

Làm sao để sử dụng ActionResult?

Như đã nhắc đến ở trên, ActionResult là một abstract class base mà triển khai IActionResult. Nó được định nghĩa trong namespace: Microsoft.AspNetCore.Mvc. ContentResult là một trong số các Action Result trả về một chuỗi văn bản:

```
public class HomeController : Controller {  
  
    public ContentResult Index()  
    {  
        ContentResult v = new ContentResult();  
        v.Content = "Hello World";  
        return v;  
    }  
}
```

Phương thức Index trả về một ContentResult. Đầu tiên chúng ta khởi tạo đối tượng của ContentResult sau đó gán giá trị "Hello World" cho thuộc tính Content của nó và trả về.

Chúng ta có một Helper method của Controller base class đó là View nó cũng làm việc tương tự nhưng code ngắn hơn nhiều:

```
public ContentResult Index()  
{  
    return Content("Hello");  
}
```

Phương thức Content cũng gọi ContentResult ngầm định. Hầu hết các Action Result đều có helper method được định nghĩa trong Controller base class. Thường thì các phương thức này có từ "Result". Ví dụ: Content cho ContentResult, View cho ViewResult.

Kiểu trả về

Phương thức Index ở trên trả về một ContentResult. Cách ưa chuộng hơn là sử dụng ActionResult như là một kiểu trả về:

```
public ActionResult Index()
```

```
{
    return Content("Hello");
}
```

Trả về ActionResult thay vì kiểu thực tế giúp chúng ta sử dụng bất cứ Action Result nào:

```
public ActionResult Index(int id)
{
    if (id==0) {
        return NotFound();
    }
    else {
        return Content("Hello");
    }
}
```

Phương thức Index trả về 2 kiểu Action Result là NotFoundResult và ContentResult phụ thuộc giá trị của tham số id.

Các loại Action Result

Có nhiều loại Action Result có sẵn trong Microsoft.AspNetCore.Mvc namespace. Nó được phân loại theo công năng sử dụng:

Trả về HTML

Chuyển hướng người dùng

Trả về file

Trả về nội dung văn bản

Trả về lỗi và HTTP Code

Kết quả liên quan đến bảo mật

Trả về HTML

Có 2 Action result trả về HTML Response. ViewResult và PartialViewResult.

ViewResult

Phương thức View() tìm kiếm View trong thư mục Views/<Controller> để tìm file .cshtml và chuyển nó cho Razor View Engine. Bạn có thể gán cho nó model dữ liệu. View sẽ trả về một ViewResult và kết quả là một HTML Response.

Mở HomeController và copy đoạn code sau:

```
public ActionResult Index()
```

```
{
    var movie = new Movie() { Name = "Avatar" };
    return View(movie);
}
```

Phương thức Index gọi View() sẽ trả về một ViewResult.

PartialViewResult

PartialView Result sử dụng model để tạo ra một phần của View. Chúng ta sử dụng ViewResult để tạo ra một view hoàn chỉnh còn PartialView trả về một phần của View. Kiểu trả về này hữu ích với Single Page Application (SPA) ki bạn muốn cập nhật một phần của View thông qua AJAX.

```
public ActionResult Index()
{
    var movie = new Movie() { Name = "Avatar" };
    return PartialView(movie);
}
```

Chuyển hướng người dùng

Redirect result được dùng khi bạn muốn chuyển hướng người dùng đến một URL khác. Có 4 loại redirect result có sẵn. RedirectResult, LocalRedirectResult, RedirectToActionResult và RedirectToRouteResult.

Mỗi một redirect này có thể trả về bất cứ mã trạng thái (status code) dưới đây:

[302 Found \(Chuyển tạm thời\)](#)

[301 Moved Permanently](#)

[307 Temporary Redirect](#)

[308 Permanent Redirect](#)

RedirectResult

RedirectResult sẽ trả về cho user bằng cách cung cấp đường dẫn tuyệt đối hoặc tương đối:

ACTION RESULT	CONTROLLER METHOD	STATUS CODE
RedirectResult	Redirect	302 Found (Temporarily moved)
	RedirectPermanent	301 Moved Permanently
	RedirectPermanentPreserveMethod	308 Permanent Redirect
	RedirectPreserveMethod	307 Temporary Redirect

Ví dụ:


```
Redirect("/Product/Index");
```

```
RedirectPermanent("/Product/Index");
```

```
RedirectPermanentPreserveMethod("/Product/Index");
```

```
RedirectPreserveMethod("/Product/Index");
```

Thay vào đó, bạn có thể sử dụng `RedirectResult` trực tiếp được định nghĩa trong `Microsoft.AspNetCore.Mvc`. Cú pháp là: **`RedirectResult(string url, bool permanent, bool preserveMethod)`**

```
return new RedirectResult(url: "/Product/Index", permanent: true, preserveMethod: true);
```

LocalRedirectResult

Action result này tương tự như `RedirectResult` nhưng chỉ khác một điều. Chỉ các local URL mới được chấp nhận. Nếu bạn cung cấp bất cứ một URL ngoài nào, phương thức này sẽ trả về một lỗi `InvalidOperationException`. Điều này tránh việc bị tấn công open redirect attack.

ACTION RESULT	CONTROLLER METHOD	STATUS CODE
LocalRedirectResult	LocalRedirect	302 Found (Temporarily moved)
	LocalRedirectPermanent	301 Moved Permanently
	LocalRedirectPermanentPreserveMethod	308 Permanent Redirect
	LocalRedirectPreserveMethod	307 Temporary Redirect

Ví dụ:

```
LocalRedirect("/Product/Index");
```

```
LocalRedirectPermanent("/Product/Index");
```

```
LocalRedirectPermanentPreserveMethod("/Product/Index");
```

```
LocalRedirectPreserveMethod("/Product/Index");
```

RedirectToActionResult

Action result này chuyển client đến một action và controller cụ thể. Nó nhận một tên Action method, một tên controller và các giá trị tham số:

ACTION RESULT	CONTROLLER METHOD	STATUS CODE
RedirectToActionResult	RedirectToAction	302 Found (Temporarily moved)
	RedirectToActionPermanent	301 Moved Permanently
	RedirectToActionPermanentPreserveMethod	308 Permanent Redirect

ACTION RESULT	CONTROLLER METHOD	STATUS CODE
	RedirectToActionPreserveMethod	307 Temporary Redirect

Ví dụ:

```
//Redirects to test action in AboutUsController
```

```
RedirectToAction(actionName: "Index", controllerName: "AboutUs");
```

```
//Redirects to Index action in the current Controller
```

```
RedirectToAction(actionName: "Index");
```

RedirectToRouteResult

Action result này chuyển khách hàng đến một route cụ thể. Nó nhận tên route, giá trị của route và chuyển chúng ta đến vị trí mà route cung cấp:

ACTION RESULT	CONTROLLER METHOD	STATUS CODE
RedirectToRouteResult	RedirectToRoute	302 Found (Temporarily moved)
	RedirectToRoutePermanent	301 Moved Permanently
	RedirectToRoutePermanentPreserveMethod	308 Permanent Redirect
	RedirectToRoutePreserveMethod	307 Temporary Redirect

Ví dụ:

```
// Redirect using route name
```

```
return RedirectToRoute("default");
```

```
//Redirect using Route Value
```

```
var routeValue = new RouteValueDictionary(new { action = "Index", controller = "Home" });
```

```
return RedirectToRoute(routeValue);
```

Trả về file

FileResult là một Action result sử dụng bởi Controller action để trả về file cho người dùng.

FileResult

FileResult là một base class sử dụng để gửi file nhị phân về response. Nó là một abstract class được triển khai bởi FileContentResult, FileStreamResult, VirtualFileResult, và PhysicalFileResult. Các class này đảm nhiệm công việc gửi file về client.

FileContentResult

FileContentResult đọc một mảng byte và trả về như một file:

```
return new FileContentResult(byteArray, "application/pdf")
```

FileStreamResult

FileStreamResult đọc một luồng stream và trả về một file:

```
return new FileStreamResult(fileStream, "application/pdf");
```

VirtualFileResult

Action result này đọc nội dung của một file từ một đường dẫn tương đối của ứng dụng trên hosting và gửi nội dung về client dưới dạng 1 file:

```
return new VirtualFileResult("/path/filename", "application/pdf");
```

PhysicalFileResult

Action result này đọc nội dung của một file từ một vị trí vật lý và gửi nội dung về client như một file. Chú ý là đường dẫn phải là đường dẫn tuyệt đối:

```
return new PhysicalFileResult("c:/path/filename", "application/pdf");
```

Content Result

JsonResult

Action result này trả về dữ liệu được định dạng JSON. Nó chuyển một object sang JSON và trả nó về client:

```
public JsonResult About()  
{  
    return Json(object);  
}
```

or

```
public JsonResult About()  
{  
    return new JsonResult(object);  
}
```

ContentResult

ContentResult ghi một nội dung cụ thể trực tiếp vào response như một chuỗi định dạng văn bản thuần.

```
public ContentResult About()
{
    return Content("Hello World");
}
```

hoặc

```
public ContentResult About()
{
    return new ContentResult() { Content = "Hello World" };
}
```

EmptyResult

EmptyResult giống như tên của nó không chứa cái gì cả. Sử dụng nó khi bạn muốn thực thi một số logic trong controller nhưng không muốn trả về gì.

```
return new EmptyResult();
```

Trả về lỗi và HTTP Code

Loại Action result này được dùng trong Web API Controller. Kết quả sẽ được gửi về kèm HTTP Status Code. Một trong số chúng thì có thể gửi một đối tượng vào response.

StatusCodeResult

StatusCodeResult gửi kết quả và chỉ ra một HTTP Status code:

```
return StatusCode(200);
```

hoặc

```
return new StatusCodeResult(200);
```

ObjectResult

Action result này sẽ trả về một đối tượng kèm một HTTP Status Code là 200. Nó là một overload của method StatusCode

```
return StatusCode(200, new { message = "Hello" });
```

hoặc

```
return new ObjectResult(new { message = "Hello" });
```

OkResult

Action result này trả về nguyên chỉ có HTTP Status code 200:

```
return Ok();
```

hoặc

```
return new OkResult();
```

OkObjectResult

Action result này trả về một HTTP Status code 200:

```
return Ok(new {message="Hello"});
```

hoặc

```
return new OkObjectResult(new { message = "Not found" });
```

CreatedResult

CreatedResult sử dụng khi một tài nguyên được tạo ra sau request Post. Nó gửi trạng thái 201 về kèm đối tượng vừa được tạo:

```
return Created(new Uri("/Home/Index", UriKind.Relative), new {message="Hello World"});
```

hoặc

```
return new CreatedResult(new Uri("/Home/Index", UriKind.Relative), new { message = "Hello World" });
```

CreatedAtActionResult

Cái này tương tự CreatedResult nhưng nó nhận vào Controller và Action thay vì URL:

```
return CreatedAtAction("Index", new {message="Hello World"});
```

CreateAtRouteResult

Action Result này nhận vào giá trị route và tương tự như CreatedResult và CreatedAtActionResult

```
CreatedAtRoute("default", new { message = "Hello World" });
```

BadRequestResult

Action result này gửi về một HTTP Status code 400 cho client. Sử dụng response status code này khi chỉ ra cú pháp không đúng hoặc một request không được rõ ràng.

```
return BadRequest();
```

BadRequestObjectResult

Action result này tương tự BadRequestResult. Khác nhau là bạn có thể gửi về một ModelStateDictionary (chứa chi tiết lỗi) và cũng là status 400:

```
var modelState = new ModelStateDictionary();  
  
modelState.AddModelError("message", "errors found. Please rectify before  
continuing");  
  
return BadRequest(modelState);
```

Phương thức BadRequest có một overload thứ 2, trả về một BadRequestObjectResult

NotFoundResult

Action result này trả về lỗi HTTP 404 cho client:

```
return NotFound();
```

NotFoundObjectResult

Action result này tương tự như NotFoundResult nhưng trả về một đối tượng kèm lỗi 404. Overload thứ 2 của NotFound giúp nhận một đối tượng làm tham số để trả về NotFoundObjectResult.

```
return NotFound( new { message = "Not found" } );
```

UnsupportedMediaTypeResult

Action result này gửi về lỗi HTTP 415. Sử dụng action result này khi request với định dạng không được hỗ trợ bởi server.

```
return new UnsupportedMediaTypeResult();
```

NoContentResult

Action result này gửi lỗi HTTP 204 về. Sử dụng NoContentResult này khi request thành công nhưng không có nội dung nào được trả về

```
return NoContent();
```

Kết quả liên quan bảo mật

SignInResult

SignInResult là kết quả của hành động đăng nhập. SignInManager.SignInAsync hoặc PasswordSignInAsync trả về một SignInResult. Nó có 4 thuộc tính là Succeeded, IsLockedOut, IsNotAllowed và RequiresTwoFactor

Bạn có thể tham khảo về ASP.NET Identity Core.

SignOutResult

SignOutResult là kết quả của hành động logout

ForbitResult

ForbiddenResult trả về lỗi 403 tức là người dùng không được cấp quyền để thực hiện một hành động nào đó trên tài nguyên nào đó. ForbiddenResult không có nghĩa là người dùng chưa chứng thực. Người dùng chưa chứng thực nên trả về ChallengeResult hoặc UnauthorizedResult.

```
return new ForbiddenResult();
```

hoặc

```
return Forbidden();
```

Forbidden là phương thức của Controller base class trả về thể hiện của ForbiddenResult. Thay vào đó bạn có thể trả về Status Code:

```
return StatusCode(403);
```

ChallengeResult

ChallengeResult trả về khi chứng thực thất bại. Kết quả sẽ không gọi đến baatcuws middleware nào để tạo response. Ví dụ trả về lỗi 401 (Unauthorized) hoặc 403 (Forbidden) hoặc chuyển hướng người dùng đến trang đăng nhập.

UnauthorizedResult

UnauthorizedResult trả về lỗi “401 – Unauthorized”. Controller sử dụng phương thức Unauthorized để trả về thể hiện của UnauthorizedResult.

```
return Unauthorized();
```

hoặc

```
return new UnauthorizedResult();
```

Khác nhau giữa UnauthorizedResult và ChallengeResult là một thằng trả về Status Code còn 1 thằng là không làm gì với nó.

19. View trong ASP.NET Core

Bài viết này chúng ta sẽ tìm hiểu về Views trong ASP.NET Core. View là thành phần của MVC pattern nó có trách nhiệm hiển thị giao diện cho người dùng.

View là gì?

View có trách nhiệm tạo ra giao diện cho người dùng từ model. Controller trong ASP.NET Core sẽ nhận request sau đó thực thi với logic tương ứng với dữ liệu đầu vào từ request. Sau đó nó trả về Model cho View.

Tham khảo: Xây dựng ứng dụng ASP.NET Core MVC đầu tiên

Trách nhiệm của View

Render ra giao diện và hiển thị model lên là trách nhiệm của View. View không nên chứa bất cứ logic nào và không được xử lý logic. View có thể sử dụng bất cứ định dạng nào để trả về cho user. Định dạng có thể là HTML, JSON, XML hay là bất cứ định dạng nào khác.

Ví dụ

Mở project sau đó tạo một ứng dụng ASP.NET Core. Project có chứa một Controller và một Index view.

Tham khảo: Xây dựng ứng dụng ASP.NET Core MVC đầu tiên

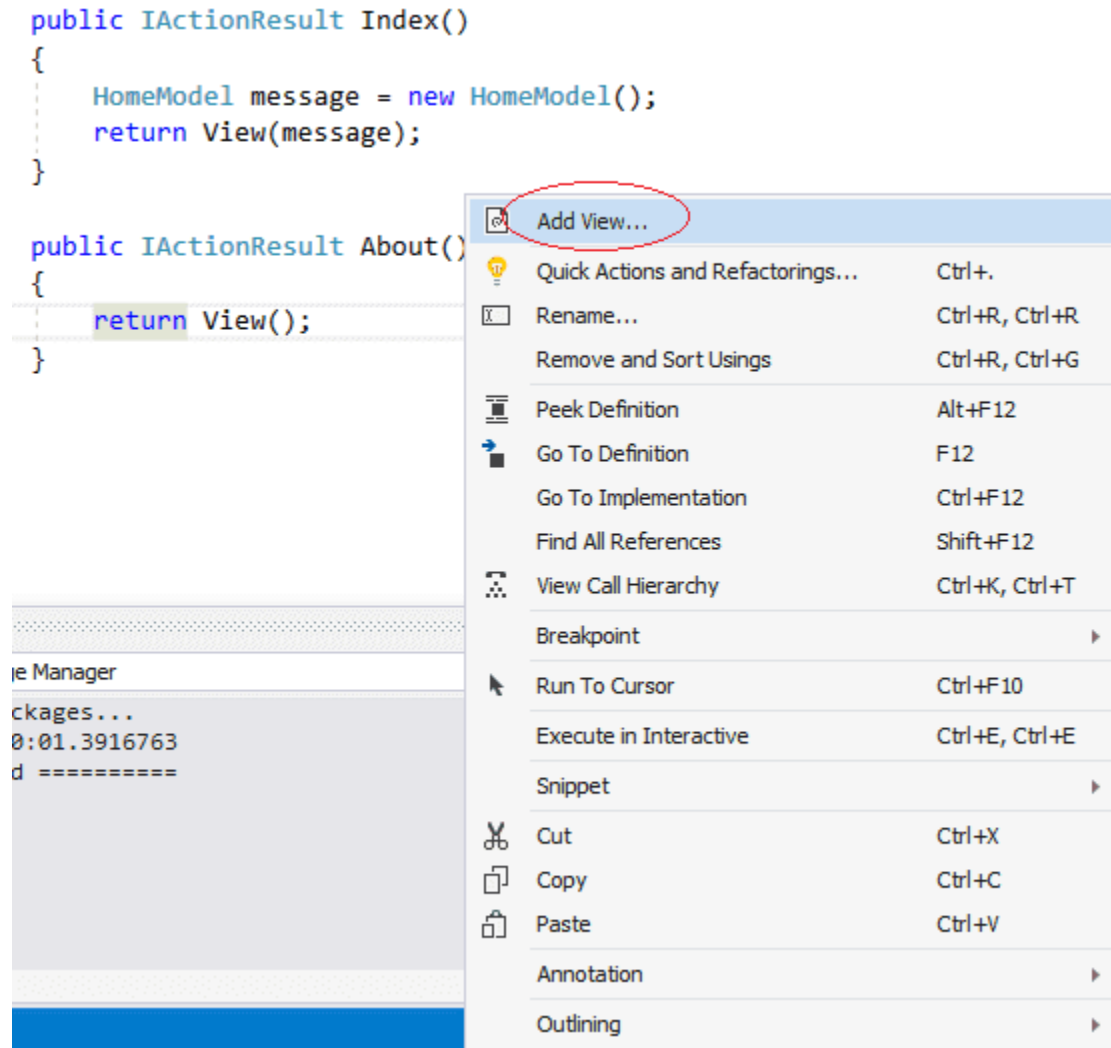
Làm sao để tạo View

Project ví dụ ở trên đã có một Index Action method. Giờ chúng ta sẽ tạo một Action method khác:

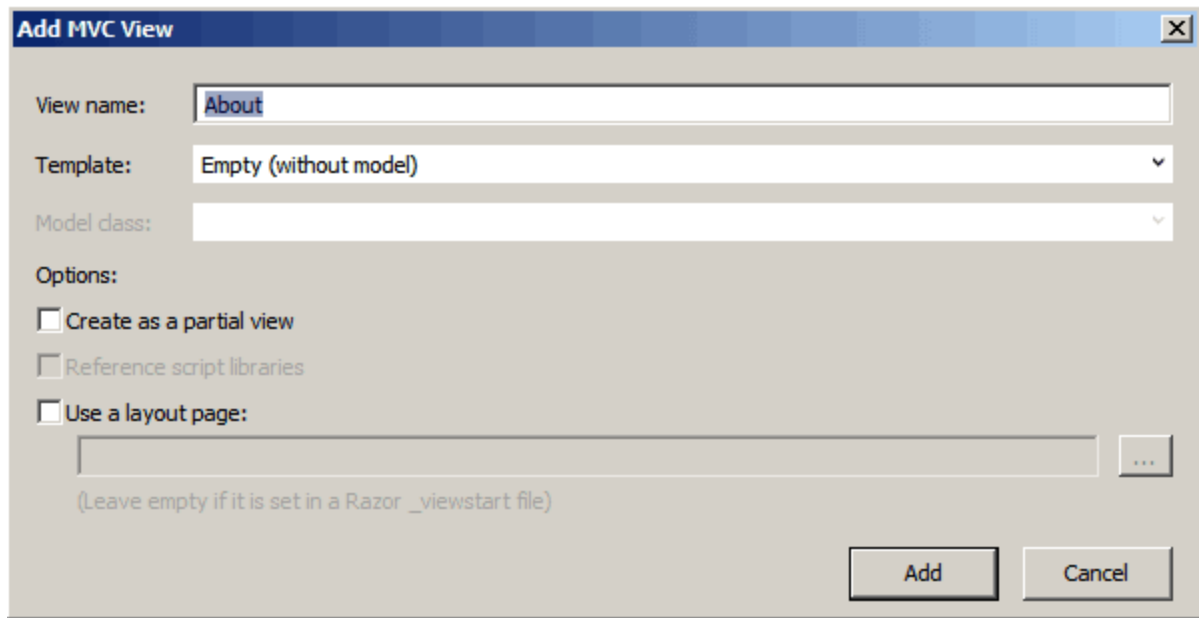
```
public IActionResult About()  
{  
    return View();  
}
```

Các method trong Controller trả về một Action **Result** (hoặc một class kế thừa từ ActionResult). Nên tham khảo: Action Result trong ASP.NET Core

ActionResult trả về HTML là một **ViewResult**. Phương thức View của Controller base class trả về một **ViewResult**. Chuột phải vào bất cứ đâu trong phương thức và chọn **Add View**



Sau đó sẽ thấy cửa sổ Add MVC View như sau:



Tên View

Nhập tên View tại đây. Theo quy tắc, tên phải cùng tên với Action Method name. Bạn có thể ghi đè phương thức này bằng cách đặt tên khác đi. Nếu bạn làm điều này thì bạn sẽ cần phải chỉ ra tên của View trong tham số đầu tiên của phương thức View: `View("ViewName")`

Template

Template có vài tùy chọn như Create, Delete, Details, Edit, List, Empty (without Model). Các template ở trên cần một model, ngoại trừ Empty Template. Visual Studio tạo ra một view để create, edit, delete...trên model được cung cấp phụ thuộc vào template mà bạn chọn. Mỗi Template trên đều có giải thích phía dưới.

Create

Tạo một HTML Form để tạo mới một model. Nó tạo ra một label và một input cho mỗi thuộc tính trong model.

Delete

Tạo một HTML Form cho việc xóa model. Nó tạo ra một label và giá trị hiện tại cho mỗi thuộc tính của model.

Details

Tạo mới một view để hiển thị model. Nó tạo ra một label và giá trị của mỗi thuộc tính trong model.

Edit

Nó tạo ra một HTML form để sửa model. Nó tạo ra một label và một input cho mỗi thuộc tính của model.

Empty

Tạo ra một view trống

List

Tạo ra một HTML Table hiển thị danh sách model. Nó tạo ra một cột cho mỗi thuộc tính của Model. Bạn cần trả về một `Enumerable<Model>` cho View. View cũng chứa danh sách các hành động để thực hiện như create/edit/delete.

Model class

Dropdown hiển thị tất cả các Model class trong project. Tùy chọn này được loại bỏ nếu bạn chọn Empty project.

Tạo một Partial View

Tùy chọn này là Create a Partial View. Partial View tạo một phần của view và không phải view hoàn chỉnh. PartialViewResult sử dụng model để tạo ra một phần của View. Kết quả của partial view nhìn giống hệt một view bình thường ngoại trừ nó không có thẻ `<html>` hoặc không có thẻ `<head>` ở trên của View. Chọn tùy chọn này chỉ ra view của bạn sẽ tạo không phải view đầy đủ vì thế tùy chọn Layout sẽ bị bỏ đi.

Thư viện Scripts

Chọn tùy chọn này sẽ thêm `jquery.validate.min.js` và `jquery.validate.unobtrusive.min.json` vào thư viện JavaScript. Các thư viện này cần thiết cho triển khai client-side validation. Các thư viện này đòi hỏi khi tạo view chứa một đầu vào cho dữ liệu như một Form Edit view hoặc một form Create.

Sử dụng Layout Page

Tùy chọn này cho phép bạn chọn một Layout page cho View. Layout page được dùng để chia sẻ các thành phần dùng chung trong trang của bạn và cung cấp một giao diện đồng nhất trong toàn bộ hệ thống.

Ví dụ chọn tên view là About, và template là Empty (without model) không chọn vào Create a partial view và Use a layout page. View được tạo sẽ nằm trong thư mục `Views/Home/About.cshtml`. Mở file About.cshtml ra và thêm dòng `<h1>About Us</h1>` vào sau thẻ tiêu đề:

```
@{
```

```
    Layout = null;
```

```
}
```

```
<!DOCTYPE html>
```

```
<html>
```

```
<head>
```

```
    <meta name="viewport" content="width=device-width" />
```

```
    <title>About</title>
```

```
<h1>About Us</h1>
</head>
<body>
</body>
</html>
```

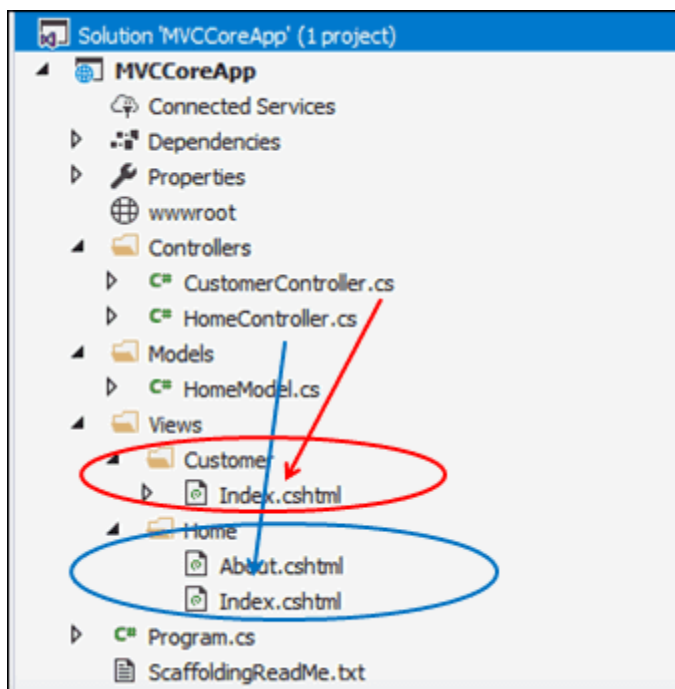
Chạy ứng dụng và bạn nhìn thấy dòng chữ "About us" trên ứng dụng.

Vị trí của View

Theo quy tắc thì tất cả View được lưu trong thư mục Views của ứng dụng. Mỗi Controller sẽ có một thư mục con trong thư mục Views với tên trùng tên controller nhưng sẽ không có hậu tố Controller. Vì thế sẽ có một **HomeController** thư mục tên Home trong thư mục Views.

Mỗi Action method trong Controller sẽ lấy một file cho nó, tên trùng với tên của Action method. Vì thế phương thức Index của HomeController có một file với tên **Index.cshtml** trong **Views/Home**.

Dưới đây là hình minh họa:



Mỗi Action method trong **HomeController** sẽ có một file View tương ứng như **About.cshtml** và **Index.cshtml** và lưu trong thư mục **Views/Home**. Tương tự như thế, action method của **CustomerController** sẽ có thư mục **Views/Customer**.

ASP.NET Core tìm View như thế nào?

Project Example có một Controller tên là **HomeController**. Giờ hãy thêm **CustomerController** vào. Chọn thư mục **Controllers** chuột phải và add thêm Controller. Tên của nó là **CustomerController**.

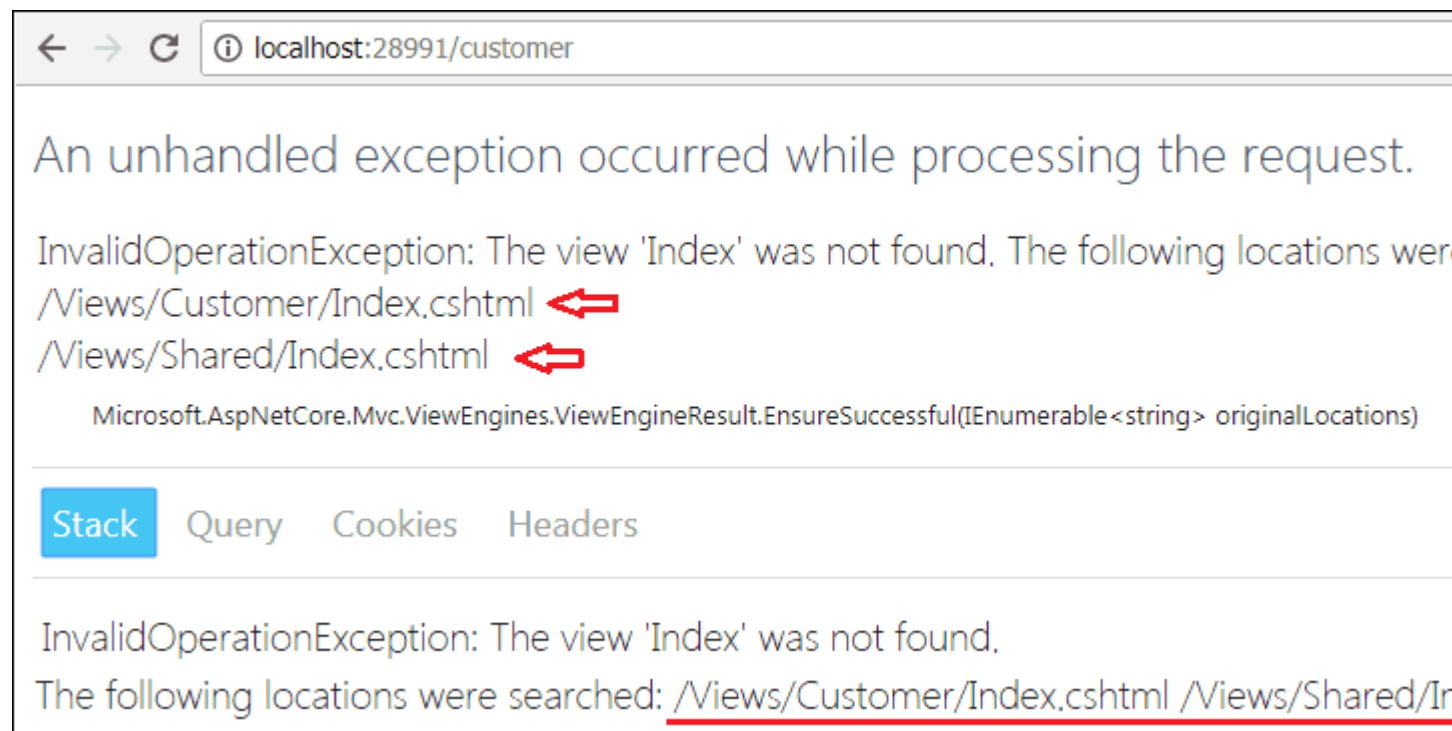
```

using Microsoft.AspNetCore.Mvc;

namespace MVCCoreApp.Controllers
{
    public class CustomerController : Controller
    {
        public IActionResult Index()
        {
            return View();
        }
    }
}

```

Giờ chúng ta không tạo View thay vào đó chạy luôn URL `/Customer`. Chúng ta nhìn lỗi như sau:



Màn hình ở trên hiển thị chính xác lỗi khi ASP.NET Core MVC middleware tìm kiếm View trong 2 thư mục:

/Views/Customer/Index.cshtml

/Views/Shared/Index.cshtml

Đầu tiên là thư mục Customer với tên của controller. Nếu không thấy nó sẽ tìm tiếp trong thư mục Shared trước khi đưa ra lỗi InvalidOperationException.

Thư mục Shared là thư mục đặc biệt chứa Views, Layouts hoặc Partial view dùng chung cho nhiều view.

Mở CustomerController và click bất cứ chỗ nào trong Index method. Chuột phải chọn Create an Empty View. File Index.cshtml sẽ tự động tạo ra trong thư mục **Views/Customer**. Giờ hãy mở Index.cshtml:

```
@{
```

```
    Layout = null;
```

```
}
```

```
<!DOCTYPE html>
```

```
<html>
```

```
<head>
```

```
    <meta name="viewport" content="width=device-width" />
```

```
    <title>Index</title>
```

```
    <h1>Customer List</h1>
```

```
</head>
```

```
<body>
```

```
</body>
```

```
</html>
```

Giờ hãy chạy ứng dụng và chúng ta không thấy lỗi này nữa.

20. Razor View Engine trong ASP.NET Core MVC

Trong bài này chúng ta sẽ đi dạo qua về Razor View Engine trong ASP.NET Core. Razor giúp dễ dàng nhúng C# code vào HTML và cung cấp khả năng tạo ra response động.

View Engine là gì?

View Engine có trách nhiệm xuất ra HTML cuối cùng khi Controller gọi Action Method. Controller action method trả về các kiểu khác nhau của response. Nó gọi là Action Result. ViewResult là một ActionResult nó trả ra HTML Response. ViewResult được cung cấp bởi View Engine. Nó sẽ sản sinh ra HTML.

Razor View Engine là gì?

Razor View Engine là View Engine mặc định của ASP.NET Core. Nó lấy mã Razor trong file View và chuyển sang HTML response.

Razor Markup

Controller trong MVC gọi View bằng cách gán dữ liệu để tạo giao diện. View phải có khả năng xử lý dữ liệu và tạo response. Điều này được xử lý bằng cách dùng Razor, nó cho chúng ta sử dụng C# code trong file HTML. Razor View Engine xử lý các lệnh này và tạo ra HTML.

Các file chứa Razor có đuôi .cshtml. Cú pháp Razor thường ngắn hơn và đơn giản hơn cũng dễ học như C# hoặc VB. Visual Studio IntelliSense cũng hỗ trợ cú pháp Razor.

Ví dụ

Để hiểu Razor làm việc ra sao? Hãy tạo Empty Project và thêm MVC Middleware vào. Bạn có thể tham khảo: Xây dựng ứng dụng ASP.NET Core MVC đầu tiên. Bạn có thể download mã nguồn từ Github.

Xem Index method của HomeController:

```
public IActionResult Index()
{
    return View();
}
```

Mở Index.cshtml và copy:

```
<h1>Razor Example</h1>
```

Chạy ứng dụng và bạn nhìn thấy ví dụ: Giờ bạn hãy mở thư mục Models ra và tạo một class Customer.cs

```
namespace MVCCoreApp.Models
{
    public class Customer
    {
        public string name { get; set; }
        public string address { get; set; }
    }
}
```

Cú pháp Razor

Razor sử dụng ký tự @ để chuyển HTML sang C#. Nos cos 2 cacsh deer khai baso:

Sử dụng Razor expression

Sử dụng khối lệnh Razor

Các biểu thức này được xử lý bởi Razor View Engine và được gán vào response.

Khối lệnh

Khối lệnh Razor bắt đầu bởi @ và nằm trong cặp ngoặc nhọn. Bạn có thể sử dụng khối lệnh bất cứ đâu trong file. Một khối Razor có thể sử dụng để thao tác Model, khai báo biến, đặt thuộc tính của View. Tuy nhiên nó không nên sử dụng cho việc xử lý logic.

Mở file Index.cshtml ra và copy đoạn này:

```
<h3>Code Block</h3>
```

```
@{  
    var greeting = "Welcome to our site!";  
    var weekDay = DateTime.Now.DayOfWeek;  
}
```

```
@{  
    var cust = new MVCCoreApp.Models.Customer()  
    {  
        name = "Rahul Dravid",  
        address = "Bangalore"  
    };  
}
```

Đầu tiên chúng ta tạo một khối lệnh Razor bắt đầu với @ và cặp {}. Trong cặp ngoặc này chúng ta có một đoạn C# thông thường với lệnh khai báo hai biến greeting và weekDay

Biểu thức Razor

Biểu thức Razor bắt đầu với @ và theo sau là code C#. Biểu thức này có thể là ngầm định hoặc tường minh.

Implicit Razor Expressions

Implicit Razor Expressions không cho phép dấu cách vì nó dùng để kết thúc biểu thức. Các biểu thức được xử lý bởi Razor View Engine và kết quả được thêm ngay vào vị trí nó đặt.

```
<h3>Code Expression</h3>
```



```
<p>@greeting</p>
```

```
<p>@DateTime.Now</p>
```

```
<p>Today is : @WeekDay thank you </p>
```

Đoạn Razor dưới đây hiển thị tên và địa chỉ trong thẻ <p>

```
<p>Name : @cust.name</p>
```

```
<p>Address : @cust.address</p>
```

Explicit Razor Expressions

Explicit Razor Expressions bắt đầu bằng dấu @ và theo sau bởi cặp (). Bất cứ nội dung nào trong cặp ngoặc đơn này đều được xử lý bởi Razor và tạo ra output.

```
@{ var ISBNNo = "10001200"; }
```

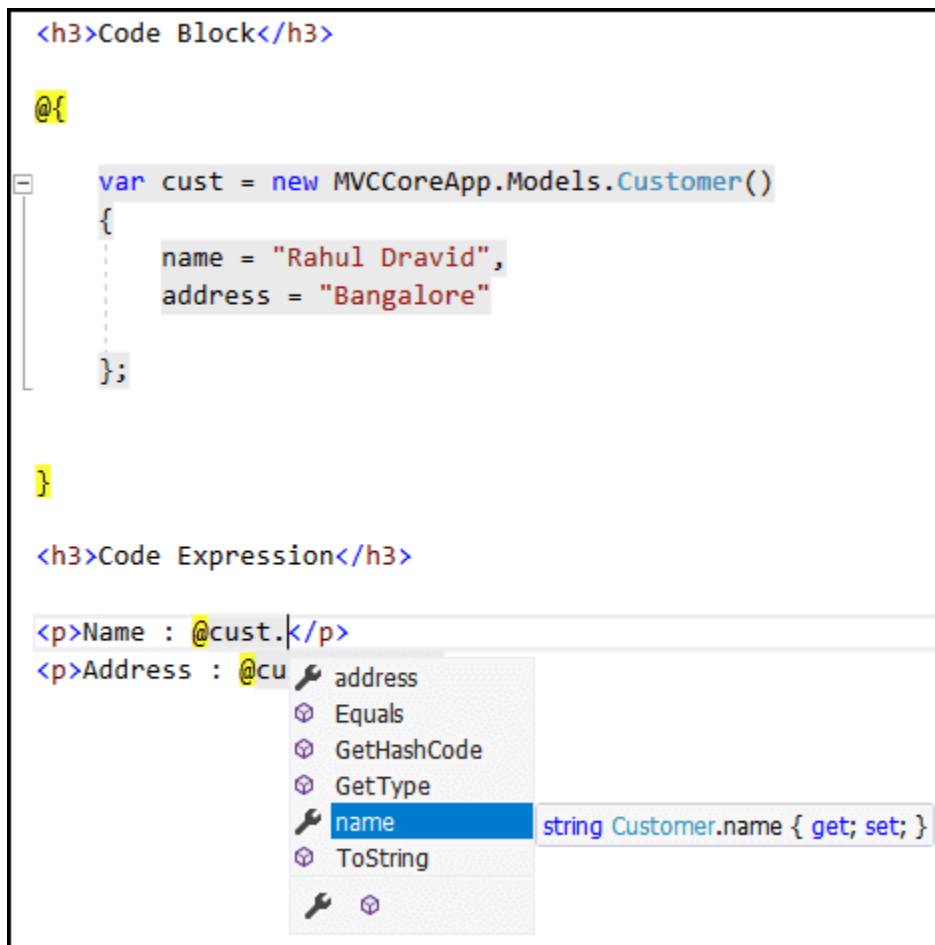
```
<p>ISBN : @ISBNNo</p>
```

```
@*//<p>ISBN@ISBNNo</p> //Does not work*@
```

```
<p>ISBN@(ISBNNo)</p>
```

Hỗ trợ nhận diện cú pháp

Hình dưới đây hiển thị cách Razor làm việc được sử dụng trong HTML và Intellisense hỗ trợ từ Visual Studio:



Sử dụng Directive

Khai báo `@using` làm việc tương tự như C# vậy. Nó cho phép chúng ta import namespace. Ứng dụng có thể import namespace `MVCCoreApp.Models` theo cách sau:

```
@using MVCCoreApp.Models;
```

Và bạn đơn giản là dùng nó `var cust=new Customer()` thay vì `var cust = new MVCCoreApp.Models.Customer()`

Khai báo biến

Các biến được khai báo sử dụng từ khóa `var` hoặc sử dụng kiểu dữ liệu C#. Các kiểu như `int`, `float`, `decimal`, `bool`, `DateTime` hay `string` có thể được dùng để lưu trữ kiểu dữ liệu tương ứng. Để khai báo biến bạn vẫn có thể dùng dấu `@` trực tiếp:

```
<h3>Variables </h3>
```

```
<!-- Storing a string -->
```

```
@{ var message = "Welcome to our website"; }
```

```
<!-- Storing a date -->
```

```
@{ DateTime date = DateTime.Now; }
```

```
<p>@message</p>
```

```
<p> The current date is @date</p>
```

Các chuỗi được bao trong một cặp nháy kép. Để sử dụng nháy kép trong chuỗi, bạn cần phải nhân đôi dấu nháy.

```
@{ var helloWorld = @"Hello ""World"""; }
```

```
<p>@helloWorld</p>
```

Tương tự như vậy thì dấu gạch chéo ngược cũng được làm như thế:

```
@{ var Path = @"C:\Windows\"; }<p>
```

```
The path is: @Path</p>
```

Bạn có thể in ký tự @ trong HTML bằng cách lặp 2 ký tự:

```
@{ var symbol = "You can print @ in html"; }
```

```
<p>The @@symbol is: @symbol</p>
```

Dấu @ trong email thì bạn có thể chỉ đích danh nó và không cần phải double:

```
<a href="mailto:admin@tektutorialsHub.com">admin@tektutorialsHub.com</a>
```

Comment

Sử dụng @**@ để đặt comment:

```
@*This is comment*@
```

HTML trong khối lệnh

Bất cứ HTML Element nào trong khối lệnh Razor được nhận dạng như bình thường:

```
@{
```

```
    <p>Hello from the Code block</p>
```

```
}
```

Dòng lệnh đơn

Bạn có thể xuất ra giá trị mà không cần thẻ HTML bằng dấu @:

```
@{
```

```
@:Hello from the Code block
```

```
}
```

Đa dòng lệnh

Đối với nhiều dòng bạn cần phải có thẻ `<text></text>`

```
@{
```

```
<text>Hello from the multiline text </text>
```

```
}
```

Lệnh điều kiện

Razor cho phép xử lý lệnh điều kiện như if hoặc switch.

If else

Điều kiện if được dùng để tạo ra một nội dung dựa trên điều kiện đó:

```
@{int value = 200;}
```

```
@if (value > 100)
```

```
{
```

```
<p>Value is greater than 100.</p>
```

```
}
```

```
else
```

```
{
```

```
<p>Value is less than 100.</p>
```

```
}
```

Hoặc đoạn code:

```
@{
```

```
var value = 200;
```

```
if (value > 100)
```

```
{
```

```
<p>The value is greater than 100 </p>
```

```
}
```

```
else
```

```

{
    <p>This value is less than 100.</p>
}
}

```

Và lệnh if else if:

```

@if (SomeCondition)
{
}
else if (SomeOtherCondition)
{
}
else
{
}

```

Switch

Một lệnh switch có thể thêm nội dung vào HTML trên một số điều kiện:

```

@switch (value)
{
    case 0:
        @: value is Zero
        break;
    case 100:
        <p>Value is 100 </p>
        break;
    case 200:
        <p>Value is @value </p>
        break;
    case 300:
        <text>Value is 300</text>
        break;
}

```

default:

```
<p>Invalid Value </p>
```

```
break;
```

```
}
```

Vòng lặp

Lặp foreach

Vòng lặp được sử dụng để lặp khối lệnh

```
@for (int i = 0; i < 5; i++)
```

```
{
```

```
<span> @i </span>
```

```
}
```

Trường hợp tốt nhất là dùng lặp qua các tập hợp đối tượng và hiển thị danh sách trong bảng:

```
@{
```

```
var custList = new List<Customer>()
```

```
{
```

```
new Customer() { name = "Rahul", address = "Bangalore" },
```

```
new Customer() { name = "Sachin", address = "Mumbai" }
```

```
};
```

```
}
```

```
<table>
```

```
<thead>
```

```
<tr><td>Name</td><td>Address</td></tr>
```

```
</thead>
```

```
@foreach (Customer custvar in custList)
```

```
{ <tr>
```

```
<td>@custvar.name</td>
```

```
<td>@custvar.address</td>
```

```
</tr>
```

```
}
```

```
</table>
```

Vòng lặp While

```
<h3>While loop</h3>
```

```
@{
```

```
    var r = 0;
```

```
    while (r < 5)
```

```
    {        r += 1;
```

```
        <span> @r</span>    }
```

```
}
```

```
@{ var s = 0; }
```

```
@while (s < 5)
```

```
{
```

```
    s += 1;
```

```
    <span> @s</span>
```

```
}
```

```
}
```

Mã hóa HTML

Tất cả biểu thức Razor được tự động mã hóa HTML

Ví dụ:

```
@{
```

```
    string encodedMessage = "<script>alert('You are under cross-site script  
injection attack');</script>";
```

```
}
```

```
<span>@encodedMessage </span>
```

Đoạn code trên sẽ không có kết quả ra một alert mà thay vào đó bạn sẽ thấy dòng chữ: "<script>alert('You are under cross-site script injection attack');</script>" trên trình duyệt.

Razor sẽ tự động mã hóa < thành < và > thành >

Nếu bạn nhìn thấy HTML Response sẽ như sau:

```
<span>&lt;script&gt;alert(&#x27; You are under cross-site script injection  
attack&#x27;);&lt;/script&gt; </span>
```

Nếu bạn muốn script thực thi, bạn cần sử dụng @Html.Raw để int chuỗi không mã hóa ra.

```
<span>@Html.Raw(encodedMessage)</span>
```

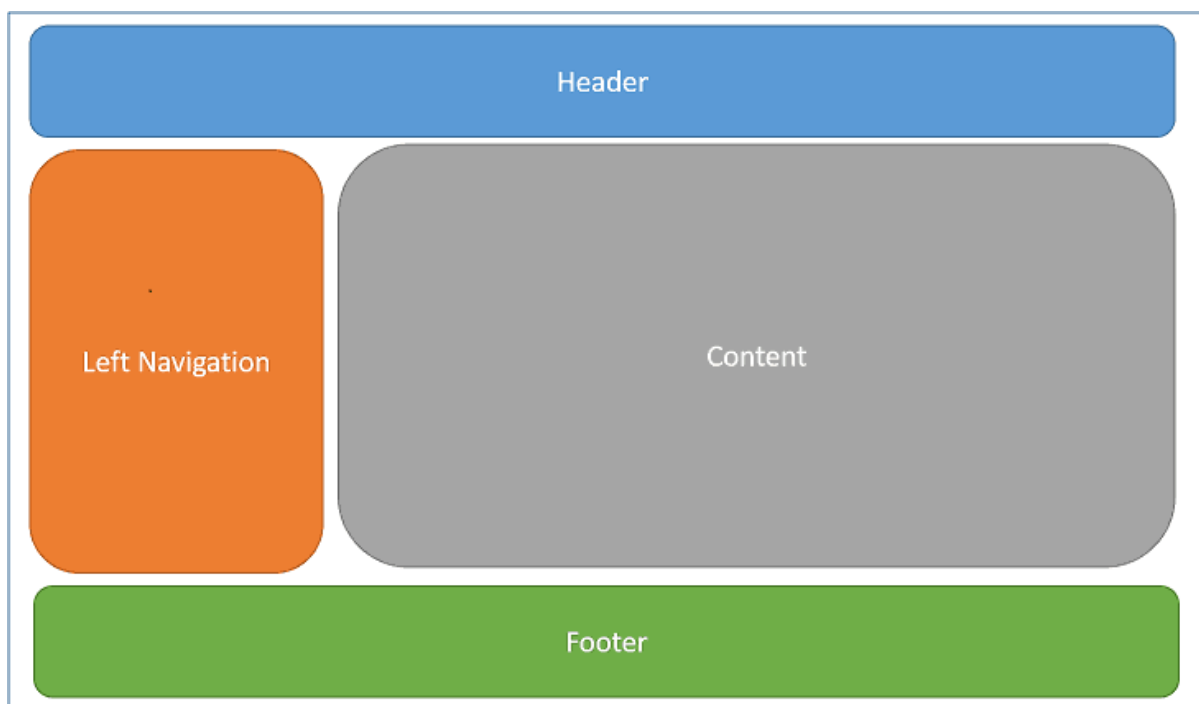
21.Sử dụng Layouts và Section trong ASP.NET Core

Layouts và Section trong ASP.NET MVC Core giúp chúng ta đảm bảo sự thống nhất giữa các trang trong toàn bộ các view của ứng dụng. Trong bài này chúng ta sẽ tìm hiểu cách tạo trang Layout và chia sẻ nó cho các view. Chúng ta sẽ sử dụng phương thức **RenderBody** để tạo ra View. Chúng ta cũng được biết làm thế nào định nghĩa các section và sử dụng nó với **RenderSection**. Cuối cùng chúng ta sẽ tìm hiểu về **_ViewStart** trong việc định nghĩa Layout.

Layout của trang web

Thiết kế của hầu hết các website bao gồm một menu, một phần header, footer và phần thanh sidebar. Như bạn thấy đi từ một trang này đến trang kia trong website thì chỉ có nội dung ở giữa là thay đổi. Nhưng menu, header, footer hay màu sắc toàn cục vẫn giữ nguyên. Điều này giúp website có cái nhìn thống nhất.

Một ứng dụng web nhìn cơ bản sẽ như sau:



Layout trong ASP.NET Core là gì?

Views trong ASP.NET Core được tạo ra từ file **.cshtml** được đặt trong thư mục **Views**. Để giữ cho việc đồng nhất giữa các trang, chúng ta cần thêm header, footer và thanh điều hướng ở tất cả các view. Tuy nhiên điều này thường vướng víu và dễ sai sót đặc biệt nếu ta có nhiều view. Trang layout trong ASP.NET Core giúp chúng ta định nghĩa một giao diện có các phần tử dùng chung như header, footer, navigation menu trên trang ở một vị trí mà có thể dùng cho mọi nơi.

Project ví dụ

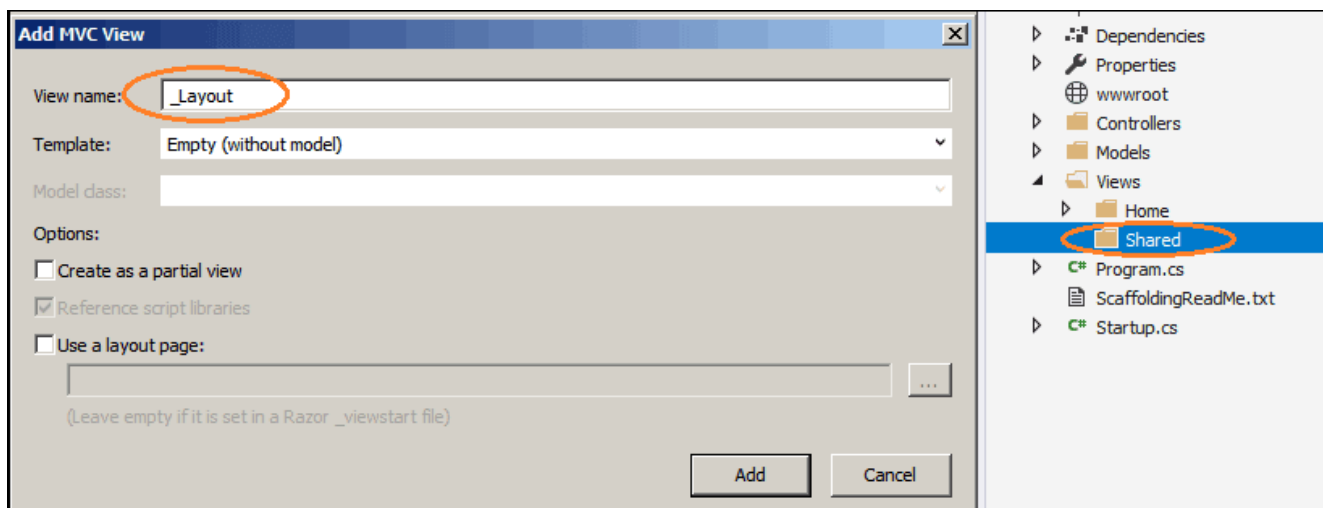
Mở một ứng dụng ASP.NET Core, chúng ta sử dụng kiến thức ở bài [Bắt đầu khởi tạo ứng dụng ASP.NET Core](#).

Vị trí của trang Layout

Theo quy ước của ASP.NET Core MVC thì các view được lưu trong thư mục Views. Mỗi Controller sẽ có một thư mục con trong thư mục Views tương ứng với tên trùng với tên controller nhưng không có hậu tố "**Controller**". Tất cả các view được liên kết với controller trong thư mục này. Thư mục **Views** cũng có chứa thư mục tên là **Shared**, nơi chứa các view chia sẻ cho nhiều view khác dùng chung. Vì thế Layouts được đặt trong thư mục này: **Views/Shared**.

Làm sao để tạo Layout

Tạo một thư mục tên là Shared bên trong thư mục Views. Chuột phải vào thư mục Shared chọn Add View. Và đặt tên View là **_Layout**. Chọn **Empty Template** và không chọn vào cả 2 checkbox **Create as a partial view** và **Use a layout page** vì chính thằng này không phải partial view cũng không cần sử dụng một layout page nào vì nó là layout to nhất rồi. Trong tương lai vẫn có những layout là con của 1 layout khác chúng ta sẽ tìm hiểu sau.



Click vào Add để tạo View. Nó sẽ tạo ra file **_Layout.cshtml** trong thư mục **Views/Shared**. Mở nó ra và dán đoạn nội dung này vào.

```
<!DOCTYPE html>
```

```
<html>
```

```
<head>
```

```
<meta name="viewport" content="width=device-width" />
```

```
<title>Layout Example</title>
```

```
</head>
```

```
<body>
```

```

<div id="header">
    <h1>Layout example in HTML</h1>
</div>
<div id="content">
    @RenderBody()
</div>
<div id="Footer">
    <p>This is Footer</p>
</div>
</body>
</html>

```

Đoạn HTML này có 3 phần là header, content và footer. Content là phần mà bạn muốn thêm nội dung các view khác vào.

RenderBody

RenderBody là một phương thức đặc biệt đánh dấu vị trí nơi mà các view sử dụng Layout này sẽ được đặt vào đó khi chạy. Cơ bản thì nó là một vùng định trước của các View sẽ hiển thị ở đó.

```
@RenderBody()
```

Chỉ có một **RenderBody** được gọi trong một trang Layout.

Chỉ ra Layout được sử dụng trong View

Thay đổi nội dung **Index** Action method trong **HomeController** như sau:

```

public IActionResult Index()
{
    return View();
}

```

Mở Index.cshtml ra từ Views/Home và copy dòng:

```

@{
    Layout = "_Layout";
}
<p>Here goes the content</p>

```

Index view trở nên đơn giản hơn. Chúng ta sẽ loại bỏ các thẻ <html>, <head> và <body> . Các thẻ này giờ đây chuyển sang trang layout. Dòng đầu tiên của code là khối Razor, chúng ta định nghĩa trang layout sẽ được dùng cho view này. Chạy ứng dụng và nhìn kết quả.



Giờ hãy thêm một Action method khác tên **AboutUs** trong **HomeController**.

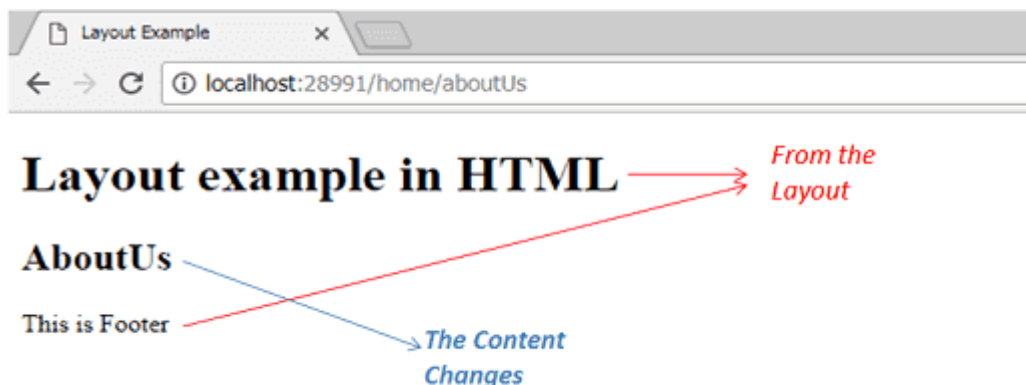
```
public IActionResult AboutUs()
{
    return View();
}
```

Giờ hãy thêm một View và chọn Layout:

```
@{
    Layout = "~/Views/Shared/_Layout.cshtml";
}

<h2>About Us</h2>
```

Giờ hãy chạy ứng dụng và vào URL: **/Home/AboutUs**. Bạn sẽ thấy nội dung đã thay đổi khi header và footer đã được thêm vào từ Layout



Đường dẫn Layout

Chúng ta sử dụng đường dẫn đầy đủ "**~/Views/Shared/_Layout.cshtml**" trong view **AboutUs**, trong khi Index action lại sử dụng đường dẫn là "**_Layout**". Khi một đường dẫn partial được cung cấp cho Razor view engine tìm kiếm file layout trong thư mục **Views/{Controllers}** trước tiên, và theo sau đó là thư mục Shared. Quá trình này được gọi là tìm kiếm View.

Section

ASP.NET Core cho phép bạn tạo một hoặc nhiều section cho layout. Section cần một cái tên và một thuộc tính chỉ ra xem nó có bắt buộc phải có trong các view con không. Phương thức **RenderSection(string name, bool required:false)** định nghĩa tên section.

```
@RenderSection("footer", required: false)
```

Thêm Section vào Layout

Mở layout ra và dán dòng lệnh sau vào sau nội dung:

```
<div> @RenderSection("Test", required: false) </div>
```

Chúng ta định nghĩa một section tên "**Test**" với tham số required là false tức là không yêu cầu bắt buộc phải có section này trong view con.

Định nghĩa Section trong view

Giờ hãy vào trang View của Index và dán đoạn sau:

```
@{ Layout = "_Layout"; }  
  
<p>Here goes the content</p>  
  
@section Test  
{ <p>Test Section</p> }
```

Các section được định nghĩa với một khối lệnh Razor là **@section** bằng tên nó đi kèm. Chạy ứng dụng và kiểm tra xem các section đó có hiển thị sau content.

Tạo section bắt buộc

Thay đổi tham số **required:true** khi định nghĩa section trong layout như dưới đây:

```
@RenderSection("Test", required: true)
```

Giờ hãy chạy ứng dụng và vào URL **/Home/AboutUs** sau đó bạn nhìn thấy lỗi sau:

InvalidOperationException: The layout page '**/Views/Shared/_Layout.cshtml**' cannot find the section '**Test**' in the content page '**/Views/Home/AboutUs.cshtml**'.

Điều này xảy ra vì chúng ta không định nghĩa section "**Test**" trong view AboutUs

_ViewStart

Trong ví dụ trên, chúng ta định nghĩa layout và dùng nó cho các view. Định nghĩa layout trong tất cả các view thật khó để bảo trì. Cách đơn giản nhất là định nghĩa layout trong **_ViewStart.cshtml**.

Mục đích của file này là cài đặt giá trị mặc định cho các View khác trong folder và folder con của nó. Giờ hãy đến thư mục Views và tạo **_ViewStart.cshtml** trong thư mục Views và dán đoạn sau:

```
@{ Layout = "_Layout"; }
```

Hãy bỏ hoặc comment đoạn code trên trong view **AboutUs**. Chạy ứng dụng và kiểm tra lại.

Loại bỏ Layout từ view

Vì chúng ta đã thêm file **_ViewStart.cshtml** vào thư mục Views nên layout giờ đã có thể áp dụng cho tất cả các View. Trong trường hợp bạn không muốn sử dụng layout cho view nào bạn chỉ cần đặt giá trị Layout là null.

```
@{ Layout = null; }
```

hoặc đặt giá trị cho Layout là một Layout khác:

```
@{ Layout = "_someOtherLayout"; }
```

Bạn có thể tạo một file **_ViewStart** tách biệt trong thư mục Controller, vậy là bạn có thể ghi đè file **_ViewStart** từ thư mục cha.

22. ViewBag và ViewData trong ASP.NET Core

View cần lấy dữ liệu từ Controller. Một trong những cách truyền dữ liệu sang View là sử dụng đối tượng **ViewData** hoặc **ViewBag**. Bài này chúng ta sẽ tìm hiểu cách dùng **ViewBag** và **ViewData**.

ViewData là gì?

ViewData là một thuộc tính của **Controller base class**, nó trả về một đối tượng **ViewDataDictionary**. **ViewDataDictionary** như tên của nó là một đối tượng dictionary cho phép lưu dữ liệu dạng key-value. Key phải là một chuỗi không phân biệt chữ hoa thường. Để truyền dữ liệu vào view bạn cần gán giá trị vào dictionary sử dụng key. Bạn có thể lưu bất kỳ số lượng **key-value** nào cần thiết trong **ViewData**.

ViewData truyền dữ liệu sang **View** từ **Controller**. Khi bạn gọi phương thức **View** trong Controller action, **ViewData** sẽ tự động gán vào View. Trong View bạn có thể truy cập giá trị được lưu trong **ViewData** cũng sử dụng key. Dữ liệu được lưu trong ViewData tồn tại chỉ trong request đó. Khi View được tạo xong cho client thì đối tượng ViewData đó cũng bị hủy.

Sử dụng ViewData như thế nào?

Trong ví dụ dưới đây nêu cách dùng ViewData trong Controller action:

```
public IActionResult SomeAction()
{
    ViewData["Greeting"] = "Hello World!";

    return View();
}
```

Trong ví dụ trên, chúng ta thêm chữ **"Hello World"** vào ViewData sử dụng key là "Greeting". Bạn có thể gọi phương thức dưới đây trong View để nhận giá trị từ key "Greeting":

```
@ViewData["Greeting"]
```

Gán đối tượng cho ViewData

Trong ví dụ trên, chúng ta lưu chuỗi dữ liệu vào ViewData. Chuỗi dữ liệu có thể được dùng trực tiếp mà không cần chuyển kiểu. Bạn có thể lưu bất cứ kiểu nào của dữ liệu như kiểu số nguyên, kiểu logic hay đối tượng trong ViewData.

Để sử dụng như một kiểu dữ liệu khác thì bạn cần phải chuyển kiểu giá trị từ ViewData sang kiểu tương ứng khi bạn dùng nó. Ví dụ dưới đây nêu cách dùng ViewData để gán đối tượng từ Controller sang View:

```
public IActionResult Index()
{
    ViewData["Greeting"] = "Hello World";
    ViewData["Product"] = new ProductModel()
    {
        ProductID=1,
        Name = "Samsung galaxy Note",
        Brand = "Samsung",
        Price = 19000
    };

    return View();
}
```

Trong View, chúng ta lấy sản phẩm từ ViewData và chuyển sang kiểu **ProductModel** và sử dụng nó:

```
@{ // Since Product isn't a string, it requires a cast.
    var product = ViewData["Product"] as ProductModel; }

@ViewData["Greeting"]!
@product.ProductID<br>
@product.Name<br>
@product.Brand<br>
@product.Price<br>
```

Bạn có thể dùng ViewData để gán dữ liệu từ Controller vào View bao gồm cả Partial View và Layout.

ViewBag là gì?

Bạn đã thấy từ ví dụ trước chúng ta có thể lưu bất cứ thứ gì trong **ViewDataDictionary**, nhưng để truy cập được bất cứ thứ gì trong ViewData chúng ta cần phải chuyển kiểu dữ liệu.

ViewBag sử dụng kiểu động (dynamic) mà chúng ta đã có trong phiên bản C# 4.0. Nó là một vỏ bọc của ViewData và cung cấp thuộc tính động cho ViewData.

ViewBag có thể tiện dụng hơn để làm việc mà không cần chuyển kiểu. Sử dụng **ViewBag** như sau:

```
public IActionResult SomeAction()
{
    ViewBag.Greeting = "Hello";

    ViewBag.Product = new ProductModel()
    {
        ProductID = 1,
        Name = "Samsung galaxy Note",
        Brand = "Samsung",
        Price = 19000
    };

    return View();
}
```

Sau đó bạn sử dụng trong View:

```
@ViewBag.Greeting World <br>
```

```
@ViewBag.product.ProductID<br>
```

```
@ViewBag.product.Name<br>
```

```
@ViewBag.product.Brand<br>
```

```
@ViewBag.product.Price<br>
```

ViewBag và ViewData

Cả ViewBag và ViewData đều sử dụng **ViewDataDictionary** bên dưới. Vì thế bạn có thể sử dụng cả hai hoặc kết hợp chúng khi đọc hoặc ghi chúng. Ví dụ:

```
ViewData["Greeting"] = "Hello World";
```

Và nhận giá trị ở view sử dụng ViewBag:

```
@ViewBag.Greeting
```

Sự khác nhau giữa ViewData và ViewBag

ViewData sử dụng cú pháp Dictionary để truy cập giá trị trong khi **ViewBag** sử dụng cú pháp truy cập giống truy cập thuộc tính của đối tượng. ViewData dẫn xuất từ **ViewDataDictionary**, nó có thuộc tính của dictionary như **ContainsKey**, **Add**, **Remove** và **Clear**.

ViewBag thì nhận từ DynamicViewData và nó cho phép tạo động các thuộc tính sử dụng dấu chấm (@ViewBag.SomeKey = <giá trị>) và không cần chuyển kiểu. Cú pháp của ViewBag giúp thêm giá trị nhanh hơn trong Controller và view.

ViewData cho phép sử dụng khoảng trắng trong Key vì nó là một chuỗi. Ví dụ **ViewData["Some Key With Namespace"]**. Nhưng ViewBag thì không thể.

ViewData cần phải chuyển kiểu dữ liệu khi không phải là một chuỗi. Nó cũng cần phải kiểm tra giá trị null để tránh lỗi. Check null đơn giản hơn trong ViewBag. Ví dụ: @ViewBag.Person?.Name.

Khi nào sử dụng ViewBag và ViewData

ViewData và **ViewBag** đều là các tùy chọn hữu ích khi bạn muốn gán lượng dữ liệu nhỏ từ Controller sang View. Chọn cái nào thường hay tùy thuộc thói quen. Điểm yếu của cả **ViewBag** và **ViewData** là chúng giải quyết vấn đề động ở thời điểm runtime. Nó không kiểm tra kiểu ở lúc biên dịch (compile time). Vì thế tăng khả năng gây lỗi.

ViewData và **ViewBag** có thể truyền dữ liệu từ **Controller** sang **View**. Nó không thể truyền ngang từ Controller này sang Controller kia được.

23. Model và ViewModel trong ASP.NET Core MVC

Bài này mình sẽ kể cho bạn nghe tổng quan về Model trong ASP.NET Core. Model có nghĩa rộng, nó là bất cứ cái gì tùy thuộc vào bạn muốn nó làm gì?. Trong ngữ cảnh của ASP.NET MVC thì Model có thể là một Domain Model, View Model hay là một Edit model. Chúng ta sẽ cùng tìm hiểu các khái niệm này và cách sử dụng chúng trong bài viết nhé.

Model là gì?

Model là tập hợp các đối tượng chứa dữ liệu của ứng dụng có thể chứa thêm cả các logic nữa. Model chia làm một số loại dựa trên công dụng và nơi chúng sử dụng. Có 3 loại mục đích chính:

Domain Model

View Model

Edit Model

Domain Model là gì?

Một Domain Model thể hiện một đối tượng trong database. Domain model thường có một mối quan hệ 1-1 với một bảng trong cơ sở dữ liệu. Domain Model liên quan đến tầng truy cập dữ liệu (DAL) trong ứng dụng. Nó nhận từ cơ sở dữ liệu hoặc một nơi nào đó lưu dữ liệu bởi tầng truy cập dữ liệu (DAL). Domain Model cũng được hiểu như entity model hay data model.

Ví dụ về Domain Model

Ví dụ chúng ta có bảng Product như sau:

dbo.Product			
	Column Name	Data Type	Allow Nulls
🔑	ProductID	int	<input type="checkbox"/>
	Name	nvarchar(50)	<input type="checkbox"/>
	BrandID	int	<input type="checkbox"/>
	SupplierID	bigint	<input type="checkbox"/>
	Qty	decimal(10, 2)	<input type="checkbox"/>
	Price	decimal(10, 2)	<input type="checkbox"/>
▶	Rating	int	<input type="checkbox"/>
			<input type="checkbox"/>

Nó được đại diện với Product Model:

```
public class Product
{
    public int ProductId { get; set; }
    public string Name { get; set; }
    public Decimal Price { get; set; }
    public int Rating { get; set; }
    public Brand Brand { get; set; }
    public Supplier Supplier { get; set; }
}
```

ViewModel là gì?

ViewModel được tham chiếu đến các đối tượng chứa dữ liệu cần cho việc hiển thị cho người dùng. ViewModel liên quan đến tầng hiển thị của ứng dụng. Nó được định nghĩa dựa trên cách thức dữ liệu được hiển thị cho người dùng hơn là cách chúng được lưu trữ ra sao?

Ví dụ về ViewModel

Ví dụ trong Product Model ở trên, người dùng cần hiển thị Brand Name và Supplier Name thay vì Brand ID và Supplier ID. Vì thế ViewModel trở thành:

```
public class ProductViewModel
{
    public int ProductId { get; set; }
    public string Name { get; set; }
```

```

    public Decimal Price { get; set; }
    public int Rating { get; set; }
    public string BrandName { get; set; }
    public string SupplierName { get; set; }
    public string getRating()
    {
        if (Rating == 10)
        {
            return "*****";
        }
        else if (Rating >= 8 )
        {
            return "*****";
        }
        else if (Rating >= 6)
        {
            return "****";
        }
        else if (Rating >= 4)
        {
            return "***";
        }
        else
        {
            return "**";
        }
    }
}

```

Chúng ta bỏ Brand và Supplier thay vào đó trả về Brand Name và Supplier Name. View Model cũng có thể có View liên quan đến logic như là hiển thị Rating cho người dùng. Bạn không nên đặt bất cứ logic nào khác ngoài logic hiển thị trên view vào View Model.

Edit Model là gì?

Edit Model hoặc Input Model đại diện dữ liệu cần để người dùng thay đổi hoặc thêm mới. Yêu cầu UI của Product cần chỉnh sửa khác với yêu cầu xem.

Ví dụ về Edit Model

Ví dụ trong Product Model ở trên, người dùng cần hiển thị danh sách Brand và Supplier, trong khi thêm mới hay chỉnh sửa sản phẩm. Vì thế model trở thành:

```
public class ProductEditModel
{
    public int ProductId { get; set; }

    [Required(ErrorMessage = "Product Name is Required")]
    [Display(Name = "Product Name")]
    public string Name { get; set; }

    public Decimal Price { get; set; }
    public int Rating { get; set; }

    public List<Brand> Brands { get; set; }
    public List<Supplier> Suppliers { get; set; }

    public int BrandID { get; set; }
    public int SupplierID { get; set; }
}
```

Model trong MVC Design Pattern

MVC Design Pattern là một pattern cho tầng hiển thị. Model trong MVC Design Pattern viết tắt của View Model và Edit Model. Hầu hết mọi người sử dụng từ View Model để chỉ cả 2 thằng này: View Model và Edit Model.

Lợi ích của View Model

ViewModel rất hữu dụng khi bạn có một UI phức tạp, khi mà dữ liệu cần lấy ra từ vài Domain Model. Vì View Model được độc lập với Domain Model, nên việc này rất mềm dẻo và linh hoạt cho việc sử dụng nó. ViewModel giúp ứng dụng bảo mật hơn vì bạn không phải chia các thuộc tính nhạy cảm và bí mật từ Domain Model ra như UserRole, IsAdmin...

Best practice với ViewModel

Giữ Domain Model và View Model tách bạch

Tránh sử dụng Domain model thay cho View Model. Bạn có thể chia những thuộc tính nhạy cảm ra ngoài cho View. Domain Model thường gắn chặt vào database để sử dụng tầng DAL. Vì thế việc chia các thuộc tính có thể sửa hay thêm mới vào database là nguy hiểm.

Tạo Strongly Typed Views (View luôn có khai báo ViewModel tương ứng)

Trong Strongly Typed Views, bạn hãy để View biết kiểu của viewModel được gán cho nó. Với strongly typed view, bạn sẽ có thể có sự trợ giúp gợi ý thuộc tính từ Visual Studio và dễ tìm lỗi nếu có trong quá trình phát triển.

Sử dụng Data Annotation cho Validation

Sử dụng Data Annotation để khai báo cho thuộc tính của viewModel và giúp tận dụng cơ chế client validation trong ASP.NET Core.

Chỉ đặt các dữ liệu cần thiết trong ViewModel

Giữ ViewModel nhỏ nhất có thể. Chỉ đặt các trường thực sự cần thiết cho việc hiển thị trong ViewModel.

Sử dụng một Mapper để chuyển Model sang ViewModel

Model nhận từ cơ sở dữ liệu cần được map sang ViewModel. Bạn có thể sử dụng AutoMapper để thực hiện điều này.

ViewModel có thể chứa các logic chỉ cho view

Về ý tưởng, ViewModel có thể chứa các dữ liệu và không có logic. Nhưng bạn có thể thêm một số logic đặc thù cho ViewModel.

Sử dụng 1 ViewModel cho 1 View

Tạo một ViewModel cho một View. Sẽ dễ bảo trì và dễ tìm lỗi.

Đồng nhất

Sử dụng ViewModel ngay cả cho các kịch bản đơn giản. Nó giúp dễ bảo trì và đảm bảo tính đồng nhất cho toàn ứng dụng.

Tổng kết

Chúng ta đã tìm hiểu về Domain Model, View Model và Edit Model trong ứng dụng ASP.NET Core. Chúng ta cũng học được các best practice sử dụng View Model.

24. Truyền dữ liệu từ Controller sang View trong ASP.NET Core

Bài viết này chúng ta sẽ tìm hiểu làm sao để truyền dữ liệu từ Controller về View. Chúng ta nên tạo các ViewModel với tất cả những dữ liệu cần thiết sau đó trả nó về cho View. View Model có thể được gán cho View bằng cách tạo một thuộc tính động trong **ViewBag**. Nó có thể được gán sử dụng thuộc tính Model của ViewData. Thuộc tính Model cho phép chúng ta tạo một strongly typed View sử dụng khai báo **@model**.

Truyền ViewModel từ Controller sang View

ViewModel tham chiếu đến các đối tượng chứa dữ liệu cần thiết để hiển thị cho người dùng. Nó có thể chứa dữ liệu từ một hoặc nhiều entity trong cơ sở dữ liệu. View Model nên chứa tất cả dữ liệu cần thiết để hiển thị cho người dùng.

ViewModel được gán từ Controller sang View bằng ViewBag hoặc ViewData. ViewData trả về một ViewDataDictionary, trong khi ViewBag chỉ là một lớp bao (wrapper) của ViewData, nó cung cấp thuộc tính động. Chúng ta đã tìm hiểu trong bài về ViewBag và ViewData. ViewDataDictionary chứa một đối tượng dictionary chung và chứa một thuộc tính đặc biệt gọi là Model. Model cho phép chúng ta gán một ViewModel sang View. Thuộc tính Model này cho phép chúng ta tạo một strongly typed view.

Tạo mới Customer ViewModel

Giờ chúng ta hãy tạo một View hiển thị một đối tượng Customer. Để làm điều này chúng ta cần tạo một Customer view model.

Tạo một View Model

Chuột phải vào thư mục **Models** và chọn **Add-->Class**. Tên của class là **Customer.cs**. Và có các thuộc tính sau:

```
namespace MVCCoreApp.Models
{
    public class Customer
    {
        public int CustomerID { get; set; }
        public string Name { get; set; }
        public string Address { get; set; }

        public Customer()
        {
            CustomerID = 1;
        }
    }
}
```

```

        Name = "Rahul Dravid";
        Address = "Bangalore";
    }
}
}

```

Sử dụng ViewBag để truyền ViewModel về View

Vào **Index** action method trong **HomeController**. Tạo một thuộc tính động tên Customer trong **ViewBag** và gán thể hiện của Customer như dưới đây:

```

public IActionResult Index()
{
    ViewBag.Customer = new Customer();
    return View();
}

```

Bạn có thể tham chiếu đến đối tượng customer sử dụng **ViewBag.Customer**. Vì bạn đang sử dụng ViewBag sẽ không cần phải ép kiểu sang kiểu tương ứng.

```

@{
    Customer customer = ViewBag.Customer;

    <p>Id :@customer.CustomerID </p>

    <p>Name :@customer.Name </p>

    <p>Name :@customer.Address </p>
}

```

Một hạn chế của cách trên là nó không kiểm tra kiểu lúc biên dịch. Nếu bạn sử dụng **ViewBag.Customers** thì nó cũng không phát hiện ra sai key lúc biên dịch mà phải đợi lúc chạy mới ra lỗi.

Sử dụng thuộc tính Model để trả về ViewModel

Như đã nhắc ở trên, **ViewDataDictionary** có một thuộc tính đặc biệt là Model, nó có thể được truy cập sử dụng ViewData. Chúng ta có thể gán Customer ViewModel về view sử dụng thuộc tính Model:

```

public IActionResult Method1()
{
    ViewData.Model = new Customer();
    return View(); }

```

Bạn có thể truy cập Model trong View bằng cách sử dụng **ViewData.Model** hoặc **Model** (trả về **ViewData.Model**):

```
@{
    Customer customer = Model;

    //Or

    //Customer customer =ViewData.Model;

    <p>Id :@customer.CustomerID </p>

    <p>Name :@customer.Name </p>

    <p>Name :@customer.Address </p>
}
```

Cách này cũng có hạn chế tương tự như cách trước. Nhưng sử dụng thuộc tính động định nghĩa trước Model cho chúng ta tùy chọn để tạo ra Strongly Typed View.

Strongly Typed View là gì?

View mà gắn vào một kiểu cụ thể của ViewModel thay vì một thuộc tính động gọi là strongly typed view. Trong ví dụ trên, chúng ta đang gắn Customer ViewModel vào View sử dụng **ViewBag.Customer** hoặc **ViewData.Model**. Trình biên dịch không biết gì về kiểu của model. Trong strongly typed view, chúng ta để cho View biết được kiểu của ViewModel được gán cho nó.

Khai báo @model

Strongly typed view được tạo sử dụng khai báo **@model**. Khai báo **@model** được đặt trên đầu của file view chỉ ra kiểu của ViewModel được gán.

```
@model Customer
```

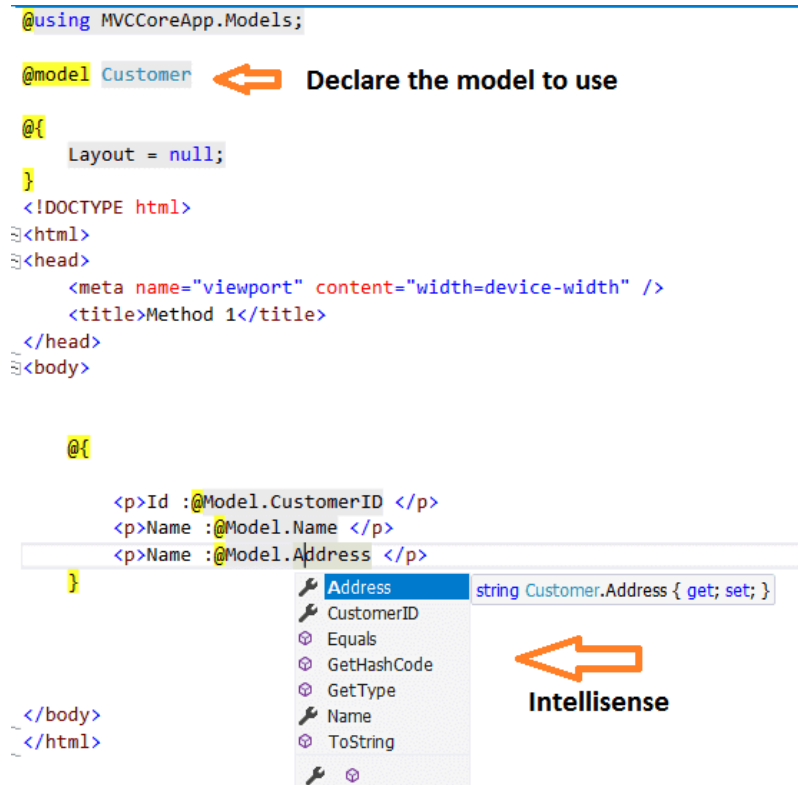
Sau đó tham chiếu trực tiếp đến model trong View:

```
@{
    <p>Id :@Model.CustomerID </p>

    <p>Name :@Model.Name </p>

    <p>Name :@Model.Address </p>
}
```

Với Strongly typed view bạn sẽ có thể có gợi ý từ Visual Studio và kiểm tra kiểu hay kiểm tra lỗi ở thời điểm biên dịch. Vì chỉ có 1 thuộc tính Model nên chỉ có 1 ViewModel được gán cho 1 view.



Model và model

Rất dễ nhầm Model và model.

Khai báo model là khai báo được dùng để khai báo kiểu của ViewModel. Còn Model là một biến sử dụng để truy cập vào ViewModel. Kiểu của Model được khai báo tùy thuộc vào khai báo @model ở trên.

Trong thực tế, tất cả dữ liệu được gán vào View qua ViewBag. Khi bạn sử dụng khai báo model, Razor engine tạo ra một thuộc tính tên Model. Model sau đó trả về kiểu được khai báo.

Ví dụ chúng ta khai báo:

```
@model Customer
```

Nó sẽ hiểu là:

```
Customer Model;
```

Cách đúng để gán ViewModel sang View

Cách được khuyến nghị để gán dữ liệu từ ViewModel sang View là sử dụng phương thức View. Phương thức View nhận model như một tham số và tự động gán vào ViewData.Model.

```
public IActionResult CorrectWay()
{
    Customer customer = new Customer();
```



```
return View(customer);
```

```
}
```

Tổng kết

Trong bài này chúng ta sẽ học cách gán dữ liệu từ Controller sang View sử dụng ViewModel. Bạn có thể sử dụng thuộc tính động của ViewBag hoặc sử dụng thuộc tính đặc biệt của ViewData là Model. Nếu bạn sử dụng thuộc tính Model, bạn có thể sử dụng khai báo `@model` để khai báo strongly typed view.

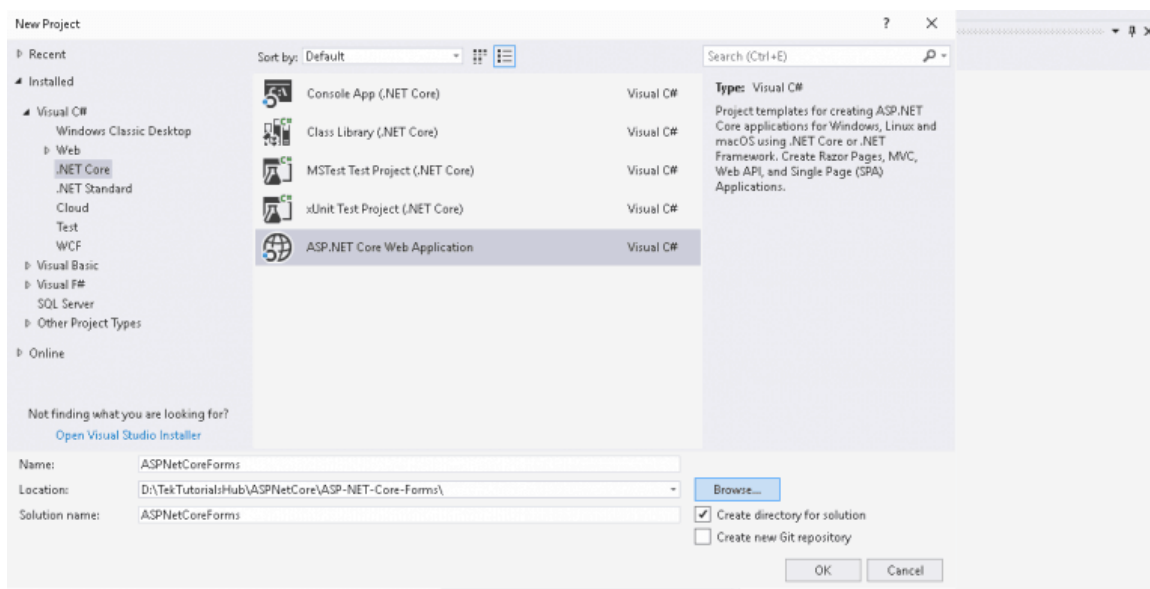
25. Xây dựng HTML Form trong ASP.NET Core

Bài này chúng ta sẽ cùng tạo một form rất cơ bản sử dụng ASP.NET Core. Một form là một đoạn HTML giúp người dùng có thể nhập được các thông tin trên trang web. Chúng ta xây dựng một form đơn giản sẽ cho phép nhập thông tin sản phẩm. Sau đó các bạn sẽ xem làm sao để một form nhận dữ liệu và gửi về máy chủ khi người dùng click nút submit.

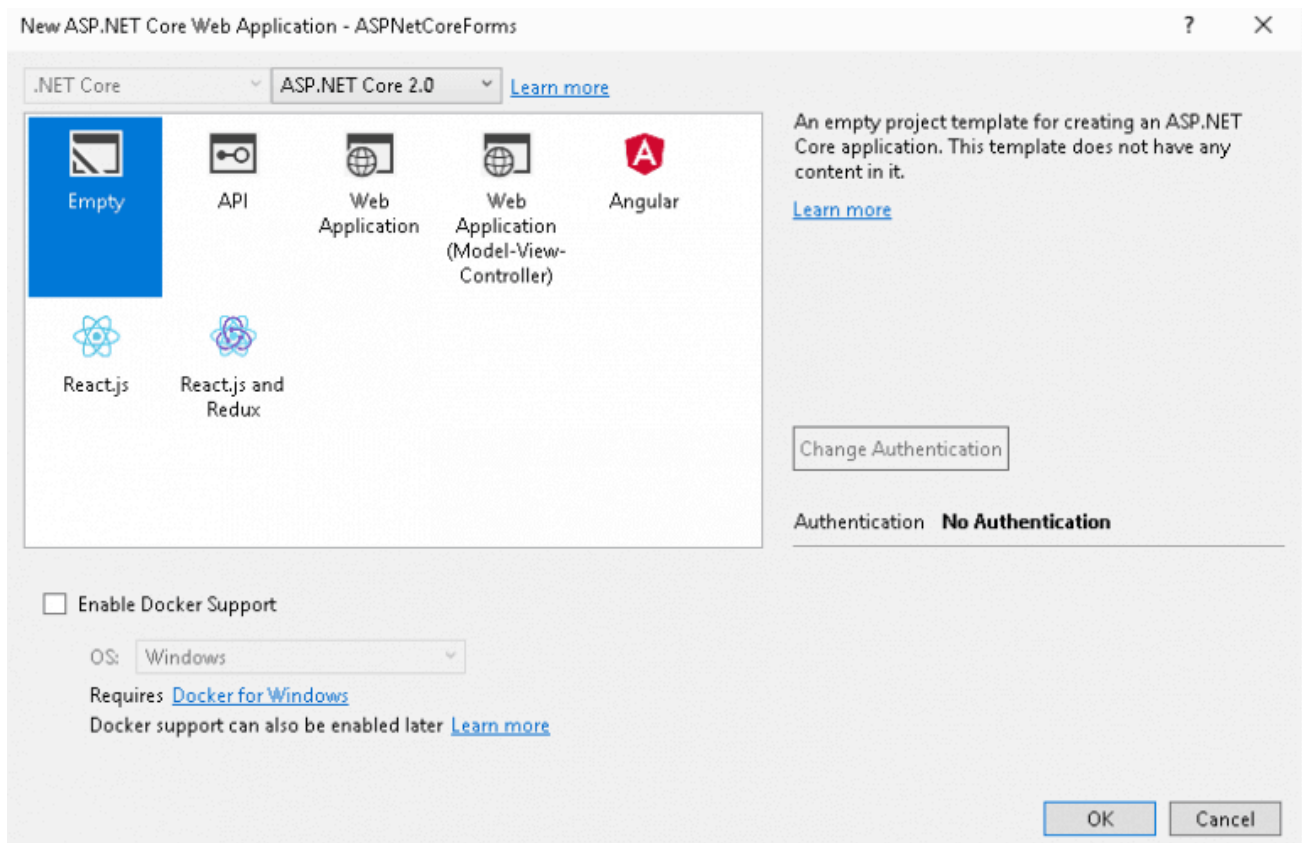
Nếu bạn là một người mới học ASP.NET Core thì hãy xem các bài hướng dẫn trước đây để nắm rõ.

Tạo mới project

Tạo mới một project tên là **ASPNetCoreForms**. Chọn mẫu là ASP.NET Core Web Application template. Template này sẽ có sẵn trong phần Visual C# -> .NET Core. Tên của project đặt là **ASPNetCoreForms**.



Tiếp theo chọn .NET Core và **ASP.NET Core 2.2** sau đó chọn **Empty template** và click **OK**.



Sau khi build project bạn sẽ thấy dòng chữ "Hello World" trên trình duyệt.

Cài đặt MVC Middleware

Mở file Startup.cs ra và cài đặt liên quan đến MVC theo như bài viết [Xây dựng ứng dụng ASP.NET Core MVC đầu tiên](#)

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc();
}

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

Tạo một View model

Chuột phải vào project tạo một thư mục là **Models**. Sau đó tạo một class tên **Product.cs** ở trong thư mục này:

```
namespace ASPNetCoreForms.Models {
```

```

public class ProductEditModel
{
    public int ID { get; set; }

    public string Name { get; set; }

    public decimal Rate { get; set; }

    public int Rating { get; set; }
}

```

Chú ý là chúng ta đặt tên **ProductEditModel** vì nó dùng để làm model cho form nhập dữ liệu. Bạn có thể đọc bài viết [View trong ASP.NET Core](#)

Tạo một Controller

Bước tiếp theo là tạo một Controller. Tạo một thư mục tên Controllers vào thư mục gốc rồi click chuột phải vào nó chọn Add-> Controller. Chọn MVC Controller - Empty và click Add. Tên của Controller là **HomeController**. Thao tác này sẽ tạo controller với một Index action method. Chuột phải vào Index method chọn Add View để tạo một view Index.

Tạo một Action Method

Giờ hãy mở **HomeController.cs** và sử dụng lệnh using để thêm Models:

```
using ASPNetCoreForms.Models;
```

Sau đó tạo một Action method:

```

[HttpGet]
public IActionResult Create()
{
    return View();
}

[HttpPost]
public IActionResult Create(ProductEditModel model)
{
    string message = "";

    if (ModelState.IsValid)
    {
        message = "product " + model.Name + " Rate " + model.Rate.ToString()
+ " With Rating " + model.Rating.ToString() + " created successfully";
    }

    Else { message = "Failed to create the product. Please try again"; }

    return Content(message);
}

```

Chúng ta đã định nghĩa 2 action method với tên là **Create**. Các action method này được đánh dấu với 2 Action verb là HttpGet và HttpPost. Bằng cách sử dụng Action verb chúng ta đảm bảo rằng action method đầu tiên tên Create sẽ chỉ xử lý các HTTP Get request và cái thứ 2 cũng tên Create

nhưng sẽ chỉ xử lý các request kiểu HTTP Post. Phương thức Create chấp nhận **ProductEditModel** làm đầu vào.

```
public IActionResult Create(ProductEditModel model)
```

ProductEditModel chứa các giá trị khi form được submit. Nó là quá trình tự động map các giá trị từ form vào model. Quá trình này chạy ngầm và nó gọi là Model Binding.

Quá trình Model binding trong ASP.NET Core map dữ liệu từ HTTP Request vào tham số của Action method. Các tham số này có thể là kiểu đơn giản hoặc là một đối tượng phức tạp.

Tiếp theo chúng ta sử dụng **ModelState.IsValid** để kiểm tra nếu ProductEditModel được nhận có hợp lệ không?

```
if (ModelState.IsValid)
```

Cơ chế ASP.NET Core Validation kiểm tra dữ liệu được submit lên từ form có bất cứ lỗi validation nào không. Nó sử dụng annotation được định nghĩa trong ProductEditModel để kiểm tra. Validation được thực hiện trước khi gọi action method.

Cơ chế validation sẽ cập nhật đối tượng ModelState. Nếu không có lỗi nào thì thuộc tính **IsValid** sẽ được cập nhật là true. Nếu có bất cứ lỗi nào thì nó sẽ được update là false.

Cuối cùng chúng ta trả về thông tin lỗi cho client thôi qua **ContentResult**.

```
return Content(message);
```

Tạo một View

Chọn phương thức muốn tạo sau đó click chuột phải và chọn Add View.

Add MVC View

View name:

Template:

Model class:

Options:

☐ Create as a partial view

☒ Reference script libraries

☒ Use a layout page:

...

(Leave empty if it is set in a Razor _viewstart file)

Add Cancel

Nó sẽ tạo một file view tên là **Create.cshtml** trong thư mục Views.

Tạo một form đơn giản

Mở file Create.cshtml và thêm mới đoạn code:

```
@{ ViewData["Title"] = "Create"; }

<h2>Create</h2>

<form action="/home/create" method="post">

    <label for="Name">Name</label>

    <input type="text" name="Name" />

    <label for="Rate">Rate</label>

    <input type="text" name="Rate" />

    <label for="Rating">Rating</label>

    <input type="text" name="Rating" />

    <input type="submit" name="submit" />

</form>
```

Đây là một form đơn giản.

Form Action

Đầu tiên chúng ta định nghĩa một thẻ HTML Form, nó được sử dụng để xây dựng nên HTML Form. Thuộc tính action chỉ ra đường dẫn sẽ gọi khi người dùng submit form.

```
<form action="/home/create" method="post">
```

Tiếp theo chúng ta có một số các trường text cho phép nhập giá trị như Name, Rate, Rating của ProductEditModel.

```
<label for="Name">Name</label>

<input type="text" name="Name" />

<label for="Rating">Rating</label>

<input type="text" name="Rating" />
```

Điều quan trọng ở đây là tên của thuộc tính trong input phải trùng với tên thuộc tính của ProductEditModel. Cơ chế **Model Binding** sử dụng tên của file để map giá trị từ form vào ProductEditModel khi form được submit.

Dòng cuối sẽ là một submit button:

```
<input type="submit" name="submit" />
```

Giờ hãy mở file Index.cshtml

```
@{ ViewData["Title"] = "Index"; }
```

```
<h2>Index</h2>
```

```
<a href="Home/Create">Create</a>
```

Chúng ta chỉ cần thêm link trở đến Create Action method:

```
<a href="Home/Create">Create</a>
```

Index View

Mở file HomeController.cs ra và chọn Index action method sau đó chuột phải và chọn Add View để tạo Index view. Copy đoạn code dưới đây:

```
@{ ViewData["Title"] = "Index"; }
```

```
<h2>ASP.NET Core MVC - Forms</h2>
```

```
<a href="Home/Create">Create</a>
```

Cuối cùng hãy chạy ứng dụng. Click nút Create sau đó nó sẽ mở một form. Nhập giá trị bên dưới và click submit.

A screenshot of a web browser's address bar. It shows navigation icons (back, forward, refresh) and the URL "localhost:54882/Home/Create".

Create

Name Rate Rating

A screenshot of a web browser's address bar. It shows navigation icons (back, forward, refresh) and the URL "localhost:54882/home/create".

product PEN DRIVE Rate 200 With Rating 5 created successfully

Các giá trị trong form được submit thông qua HTTP Post đến **/Home/Create** action method.

ASP.NET Core **Model Binder** sẽ tạo một thể hiện của ProductEditModel sau đó map mỗi giá trị trong input field vào thuộc tính của ProductEditModel.

Giờ hãy tạo một giá trị không hợp lệ. **Model Binder** sẽ gặp lỗi khi map giá trị từ form vào thể hiện của ProductEditModel và ModelState.IsValid sẽ là false.

The first screenshot shows a web browser at localhost:54882/Home/Create. The form has three input fields: 'Name' with the value 'PEN DRIVE', 'Rat' with the value 'AAA' (circled in red), and 'Rating' with the value '5'. A 'Submit' button is to the right. The second screenshot shows the same browser at localhost:54882/home/create, displaying a red error message: 'Failed to create the product. Please try again'.

Tổng kết

Đây là form rất đơn giản để các bạn hiểu được cách xây dựng form và submit dữ liệu lên action method.

26. Strongly Typed View trong ASP.NET Core

Trong bài viết trước mình đã hướng dẫn các bạn xây dựng một HTML Form đơn giản. Chúng ta dùng **ViewModel** nhưng không gán nó cho View. ViewModel có thể được gán từ controller sang View dùng **ViewBag** hoặc **ViewData**. ASP.NET Core cung cấp khả năng có thể gán strongly typed view model hoặc một object sang view. Cách tiếp cận này giúp chúng ta dễ dàng bảo trì code và phát hiện lỗi từ lúc biên dịch. Cơ chế **Scaffolding** trong Visual Studio có thể được dùng để tạo một View dựa trên ViewModel.

Strongly Typed View là gì?

View nào mà được kết hợp với một kiểu cụ thể của ViewModel thì được gọi là Strongly Typed View. Bằng cách chỉ ra model, Visual Studio cung cấp cơ chế gợi ý và kiểm tra kiểu lúc biên dịch. Chúng ta đã học cách truyền dữ liệu từ Controller sang View trước đây. Điều này thường được giải quyết với ViewBag hoặc ViewData. Trình biên dịch không biết gì về kiểu của model.

Trong Strongly typed view, chúng ta biết View sử dụng ViewModel nào sử dụng khai báo **@model**.

Khai báo @model

Strongly typed view được tạo sử dụng khai báo **@model**. ViewData có một thuộc tính đặc biệt gọi là Model. Nó là một đối tượng kiểu dynamic. Nó cho phép chúng ta sử dụng ViewData.Model.Prop. Sử dụng Model cách này không giúp Visual phát hiện và kiểm tra thuộc tính của model mà phải chờ lúc chạy chương trình mới phát hiện lỗi. Nhờ thế thì việc kiểm tra kiểu lúc biên dịch cũng không có.

Vấn đề ở trên đã được giải quyết bằng cách chỉ cho View biết kiểu của model sẽ được gắn vào ViewData.Model. Nó được giải quyết bằng khai báo @model, nó được đặt ở trên cùng của file View và chỉ ra kiểu của ViewModel được gắn.

Khi bạn sử dụng khai báo @model, Razor engine sẽ gán kiểu cho ViewData.Model. Thuộc tính Model sẽ trả về kiểu được khai báo.

Ví dụ:

```
@model ProductEditModel
```

Được hiểu thành:

```
ProductEditModel Model;
```

Vì thế nên kiểu của Model giờ được biết trước ở lúc biên dịch, chúng ta sẽ tận dụng được trình gọi ý cú pháp (IntelliSense) và kiểm tra kiểu lúc biên dịch.

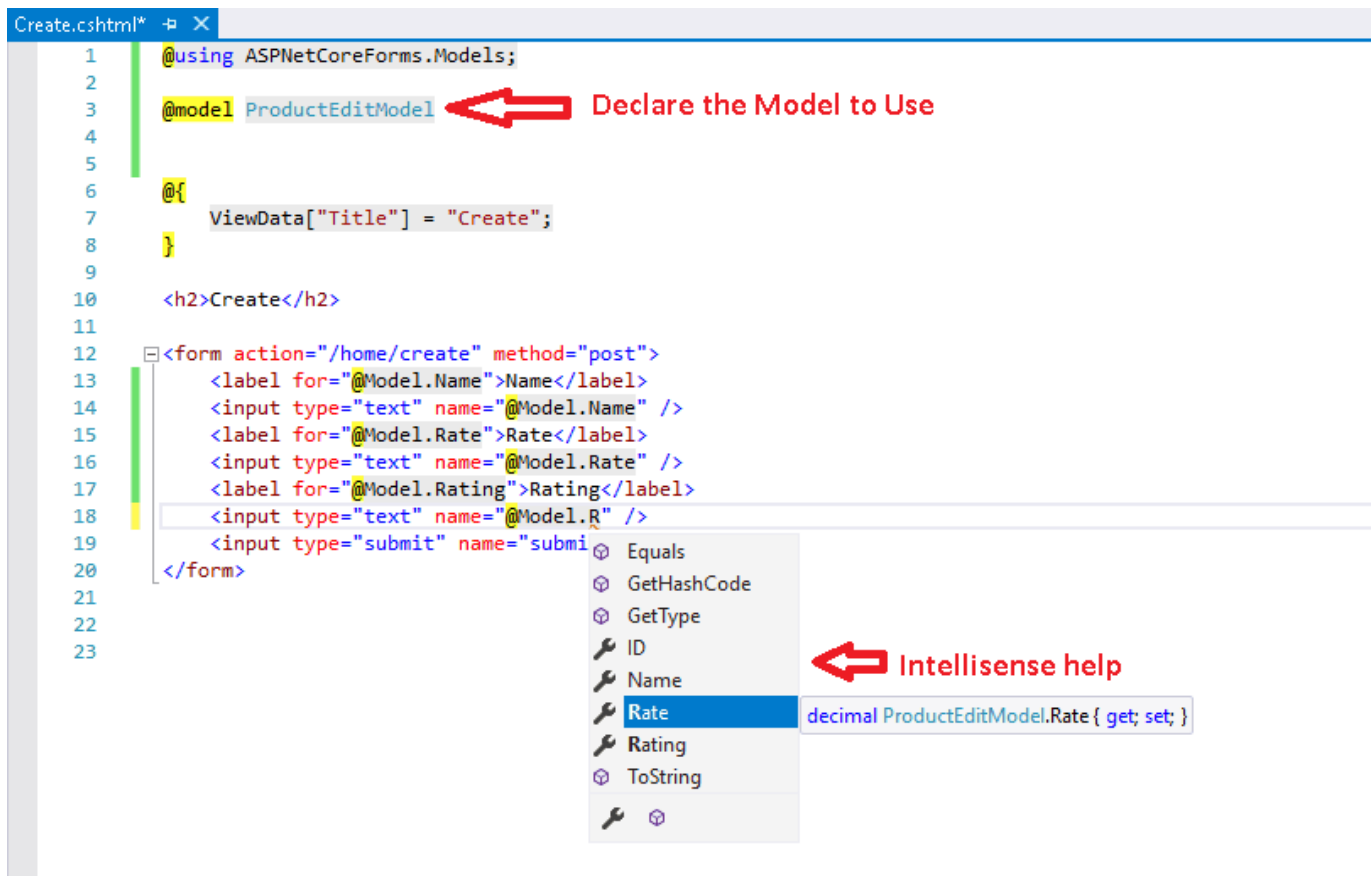
Ví dụ về Strongly Typed View

Chúng ta tiếp tục đi từ điểm cuối của bài trước. Cập nhật code để thêm ProductEditModel vào đầu của View:

```
@model ProductEditModel
```

Và bạn sẽ dùng thuộc tính Model để truy cập đến thuộc tính của nó:

```
<form action="/home/create" method="post">
    <label for="@Model.Name">Name</label>
    <input type="text" name="@Model.Name" />
    <label for="@Model.Rate">Rate</label>
    <input type="text" name="@Model.Rate" />
    <label for="@Model.Rating">Rating</label>
    <input type="text" name="@Model.Rating" />
    <input type="submit" name="submit" />
</form>
```

Gán Model từ Controller sang View

Bước tiếp theo chúng ta sẽ gán thể hiện của ProductEditModel sang View. Mở HomeController.cs và đến phương thức Create:

```
[HttpGet]
public IActionResult Create()
{
    ProductEditModel model = new ProductEditModel();
    return View(model);
}
```

Giờ hãy chạy ứng dụng

Lợi ích của Strongly Typed View

Hỗ trợ IntelliSense

Kiểm tra kiểu lúc biên dịch

Không phải chuyển kiểu

Vì thế chỉ có một thuộc tính Model, bạn chỉ có thể có 1 ViewModel trên View.

Model và model

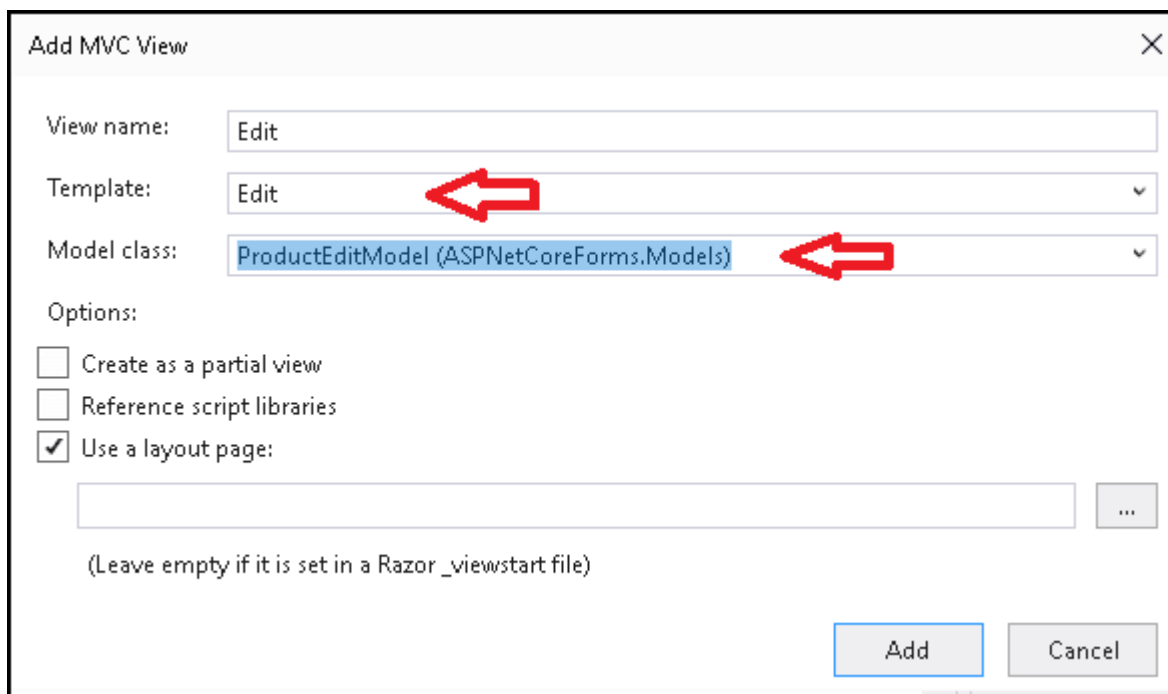
Rất dễ nhầm giữa Model và model. Khai báo model được dùng để khai báo kiểu của ViewModel. Còn Model là biến được dùng để truy cập vào ViewModel. Kiểu của Model được khai báo bằng từ khóa @model.

Sử dụng Scaffolding để tạo ra Strongly Typed View

Visual Studio Scaffolding System cho phép chúng ta tạo nhanh một view. Mở HomeController.cs và tạo một Action method là Edit.

```
[HttpGet]
public IActionResult Edit()
{
    ProductEditModel model = new ProductEditModel();
    return View(model);
}
```

Chọn template là Edit. Chọn Model class là ProductEditModel. Click vào Add để tạo ra một View. Sau đó mở View ra:



```
@model ASPNetCoreForms.Models.ProductEditModel
```

```
@{
```

```
    ViewData["Title"] = "Create";}
```

```

<h2>Create</h2>
<h4>ProductEditModel</h4>
<hr />
<div class="row">
  <div class="col-md-4">
    <form asp-action="Create" method="post">
      <div asp-validation-summary="ModelOnly" class="text-danger"></div>
      <div class="form-group">
        <label asp-for="ID" class="control-label"></label>
        <input asp-for="ID" class="form-control" />
        <span asp-validation-for="ID" class="text-danger"></span>
      </div>
      <div class="form-group">
        <label asp-for="Name" class="control-label"></label>
        <input asp-for="Name" class="form-control" />
        <span asp-validation-for="Name" class="text-danger"></span>
      </div>
      <div class="form-group">
        <label asp-for="Rate" class="control-label"></label>
        <input asp-for="Rate" class="form-control" />
        <span asp-validation-for="Rate" class="text-danger"></span>
      </div>
      <div class="form-group">
        <label asp-for="Rating" class="control-label"></label>
        <input asp-for="Rating" class="form-control" />
        <span asp-validation-for="Rating" class="text-danger"></span>
      </div>
      <div class="form-group">
        <input type="submit" value="Create" class="btn btn-default" />
      </div>
    </form>
  </div>
</div>

```

```
</form>
```

```
</div>
```

```
</div>
```

```
<div><a asp-action="Index">Back to List</a></div>
```

Tổng kết

Chúng ta đã kết hợp view vào ViewModel để tạo ra Strongly typed view. Chúng ta cũng học cách sử dụng Scaffolding để tạo ra Strongly typed view.

27. Tag Helpers trong ASP.NET Core MVC

Tag Helpers là tính năng mới của ASP.NET Core, nó giúp chúng ta thêm code phía server vào HTML dễ dàng. Trong bài này chúng ta sẽ sử dụng nó trong HTML Form mà chúng ta tạo trong Strongly Typed View trước.

Tag Helper là gì?

Tag Helper giúp chúng ta viết phần tử HTML trong Razor sử dụng cú pháp thân thiện với HTML. Nó nhìn như là HTML chuẩn vậy nhưng code được xử lý bởi Razor Engine trên server và nó tận dụng được các ưu điểm của việc xử lý phía server.

Razor được tạo sử dụng Tag Helper nhìn như phần tử HTML thuần. Nó thao tác với các phần tử HTML như thêm mới phần tử HTML hay thay thế các nội dung có sẵn bằng một cái mới.

Ví dụ, sử dụng thẻ Form Tag Helper, chúng ta có thể tạo ra thẻ <form> như dưới đây. Với các thuộc tính asp-action và asp-controller của Form Tag Helper:

```
<form asp-action="create" asp-controller="home">
```

Sẽ được gen ra HTML:

```
<form action="/home/create" method="post">
```

Mục đích của Tag Helpers

Bạn có thể tạo form mà không cần dùng Tag Helper (hoặc HTML Helper) như bài trước. Tuy nhiên Tag Helper sẽ giúp tạo ra view HTML đơn giản hơn dựa trên dữ liệu từ Model gắn vào nó. Ví dụ Label Tag Helper sẽ tạo ra tiêu đề dựa trên attribute Data Annotation trong View Model. Tương tự như thế thì Input Tag Helper sẽ tạo ra id, name, type của phần tử HTML dựa trên kiểu dữ liệu của Model và thuộc tính Data Annotation.

Sử dụng Tag Helper?

ASP.NET Core Tag Helper nằm trong thư viện **Microsoft.AspNetCore.Mvc.TagHelpers** bạn cần import thư viện này để sử dụng Tag Helper.

Thêm Tag Helper sử dụng @addTagHelper

Để sử dụng Tag Helper bạn cần thêm khai báo `@addTagHelper` vào view, nơi mà bạn muốn sử dụng.

```
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

Đoạn code trên sử dụng wildcard ("*") để chỉ ra tất cả Tag Helper được thêm vào từ thư viện `Microsoft.AspNetCore.Mvc.TagHelpers`.

Thêm Tag Helper toàn bộ các View

Thêm `@addTagHelper` vào một view nào đó chỉ có tác dụng trên view đó. Bạn có thể thêm `@addTagHelper` vào `_ViewImports.cshtml` để sử dụng Tag Helper trên toàn bộ các view của ứng dụng.

Bỏ Tag Helper

Đoạn code dưới đây loại bỏ tất cả tag helper từ assembly `Microsoft.AspNetCore.Mvc.TagHelpers` từ một view cụ thể:

```
@removeTagHelper "*", Microsoft.AspNetCore.Mvc.TagHelpers"
```

Thêm một số Tag Helper chỉ định

Thay vì add tất cả tag helper thì bạn có thể chọn ra một số cái mà bạn muốn dùng thôi:

```
@addTagHelper "Microsoft.AspNetCore.Mvc.TagHelpers.InputTagHelper, Microsoft.AspNetCore.Mvc.TagHelpers"
```

Tắt Tag Helper với !

Bằng cách sử dụng ký tự `!` trước mỗi phần tử HTML, bạn sẽ có thể vô hiệu hóa tag helper cho phần tử đó:

```
<!label asp-for="Name"></!label>
```

Tag helper label này được vô hiệu hóa với đoạn code trên. Bạn phải áp dụng ký tự `!` cho cả thẻ đóng và thẻ mở.

Sử dụng @tagHelperPrefix để bật Tag Helper

Thay vì vô hiệu hóa tag helper sử dụng ký tự `!`, bạn có thể sử dụng `@tagHelperPrefix`

```
@tagHelperPrefix th:
```

Giờ thì tiền tố `th:` phải được chỉ ra cho tất cả các tag helper trên view, để bật tag helper cho nó:

```
<th:label asp-for="Name"></th:label> //Tag helper is enabled
```

```
<label asp-for="Address"></label> //Tag helper is disabled
```

Ví dụ về Tag Helper

Chúng ta xây dựng một Form đơn giản như bài trước. Hãy dùng Tag Helper. Đầu tiên mở `HomeController.cs` ra và thay đổi action method `Create`.

```
[HttpGet]
```

```
public IActionResult Create()
{
    ProductEditModel model = new ProductEditModel();

    return View(model);
}
```

Thể hiện của **ProductEditModel** được gán vào View để tạo Strongly Typed View.
Mở **Create.csthml** từ thư mục **/Views/Home**.

Form Tag Helper

Form Tag Helper được bao bởi thẻ `<form>`. Form Tag Helper cung cấp một số thuộc tính phía server giúp chúng ta thao tác để tạo ra HTML. Một số thuộc tính đó là:

asp-controller: Chỉ ra tên Controller sử dụng

asp-action: Chỉ ra tên action method sử dụng

asp-area: Chỉ ra tên Area sử dụng

Ví dụ:

```
<form asp-controller="Home" asp-action="Create">
```

Đoạn code trên sẽ biên dịch ra HTML thuần là:

```
<form method="post" action="/Home/Create">

    <input name="__RequestVerificationToken" type="hidden"
value="CfDJ8PlIso5McDB0jgPkVg904mnNiAE8U0HkVlA9e-
Mtc76u7fSjCnoy909Co49eGlbyJxpp-
nYphF_XkOrPo0tTGdygc2H8nCtZCcGURMZ9Uf01fP0g5jRARxTHXnb8N6yYADtdQSnJIItXtYsir8G
CWqZM" />

</form>
```

Chú ý là Form Tag Helper nó sẽ tự động thêm Antiy-Forgery Token vào HTML tạo ra.

Label Tag Helper

Label Tag Helper được áp dụng cho phần tử label. Nó có một thuộc tính là asp-for. Sử dụng như sau:

```
<label asp-for="@Model.Name"></label>
```

Nó sẽ dịch ra:

```
<label for="Name">Name</label>
```

Tên của trường đó sẽ được lấy từ tên của thuộc tính trong Model hoặc từ Data Annotation của thuộc tính trong Model. Sử dụng từ khóa `@Model` là không bắt buộc. Bạn có thể chỉ ra tên thuộc tính trong Model luôn:

```
<label asp-for="Name"></label>
```

Cái này bạn có thể đọc Strongly Typed View ở bài trước.

Input Tag Helper

Tương tự, Input Tag Helper cũng được áp dụng cho phần tử input:

```
<input asp-for="Name" />
```

Nó sẽ dịch ra: `<input type="text" id="Name" name="Name" value="" />`

Thuộc tính type, id và name tự động lấy từ tên và kiểu dữ liệu của trường đó trong Model. Cuối cùng form nhìn như sau:

```
<form asp-controller="Home" asp-action="Create">
```

```
    <label asp-for="Name"></label>
```

```
    <input asp-for="Name" />
```

```
    <label asp-for="Rate"></label>
```

```
    <input asp-for="Rate" />
```

```
    <label asp-for="Rating"></label>
```

```
    <input asp-for="Rating" />
```

```
    <input type="submit" name="submit" />
```

```
</form>
```

Khi chạy nó sẽ tạo ra HTML như này:

```
<h2>Create</h2>
```

```
<form action="/Home/Create" method="post">
```

```
    <label for="Name">Name</label>
```

```
    <input type="text" id="Name" name="Name" value="" />
```

```
    <label for="Rate">Rate</label>
```

```
    <input type="text" data-val="true" data-val-number="The field Rate must  
be a number." data-val-required="The Rate field is required." id="Rate"  
name="Rate" value="0.00" />
```

```
    <label for="Rating">Rating</label>
```

```
    <input type="number" data-val="true" data-val-required="The Rating field  
is required." id="Rating" name="Rating" value="0" />
```

```
    <input type="submit" name="submit" />
```

```
    <input name="__RequestVerificationToken" type="hidden"  
value="CfDJ8PlIso5McDB0jgPkVg904mlyQIZDgGzfiyx_ZiUck9A5E6DqBMIPnDCjFTyw5As2AL  
JT34MG_lmaAanTwCeq1ugZ1r7w7qBsQCIGm07Zw1C6CFvJNj6y4kxrSq0PT0Lk7XXqPv9NDsTL7C -  
6aB85Mjo" />
```

```
</form>
```

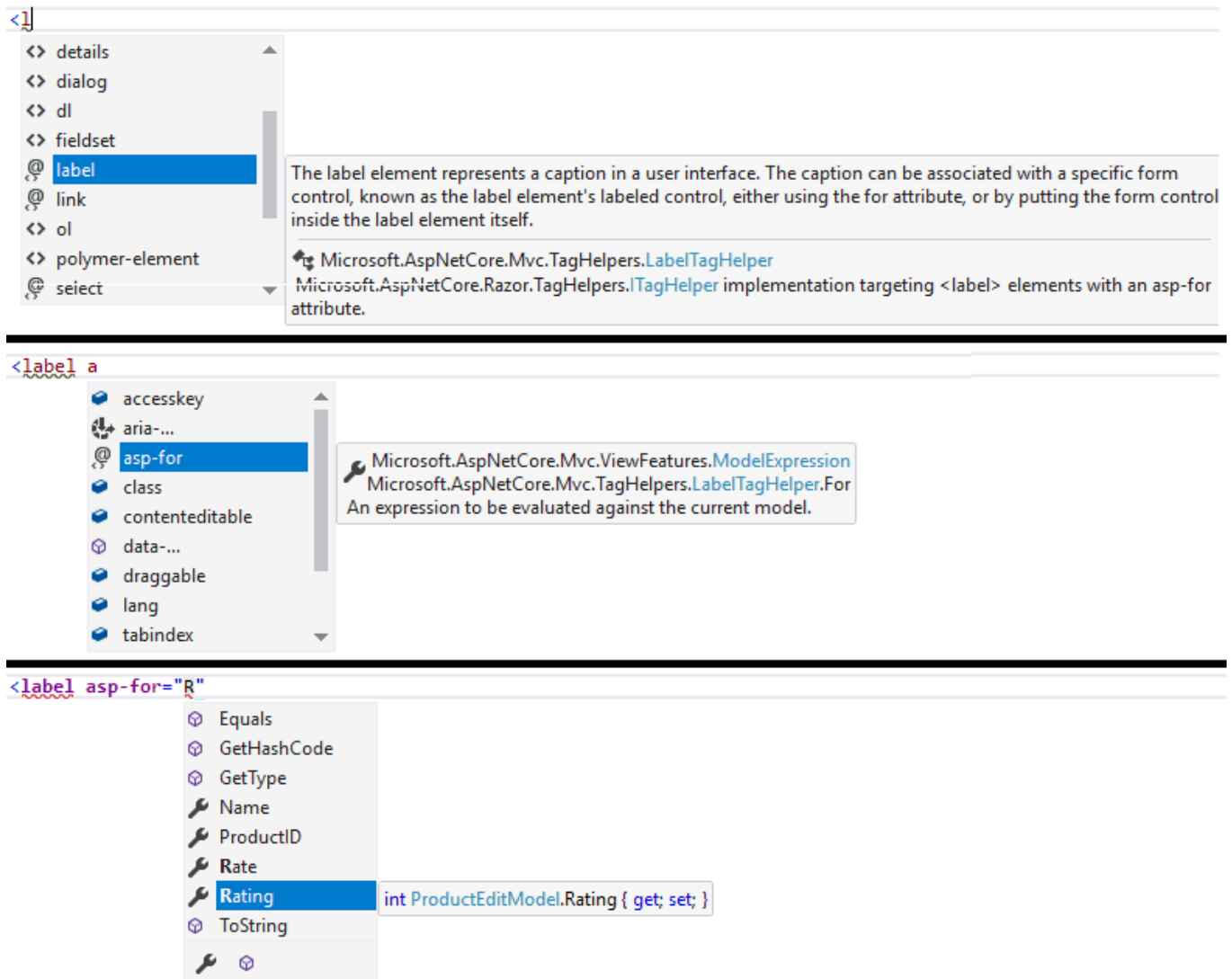
Lợi ích của Tag Helper

Thân thiện với cú pháp HTML

Tag Helper nhìn như là phần tử HTML chuẩn. Các Front end Developer không cần học cú pháp C# hay Razor để thêm các phần tử này vào View. Vì thế nó dễ dàng đạt được tính chất chia để trị. Bạn có thể dễ dàng thêm CSS hoặc bất cứ thuộc tính HTML nào vào Tag Helper như là với HTML.

Được hỗ trợ bởi IntelliSense

Tag Helper cung cấp sự hỗ trợ bởi cơ chế gợi ý thông minh của Visual Studio. Ví dụ dưới khi chúng ta thấy gợi ý cho label của thuộc tính Rating.



Code sạch hơn

Code sẽ sạch và rõ ràng hơn sử dụng HTML Helper cũ. Không cần phải sử dụng ký tự `@` để chuyển giữa C# và HTML.

Dễ mở rộng hơn

ASP.NET Core MVC cung cấp nhiều tag helper có sẵn giúp chúng ta tạo view. Nhưng nếu không có tag helper nào phù hợp với nhu cầu. Bạn cũng có thể tạo ra Tag Helper của riêng mình bằng cách mở rộng các Tag Helper có sẵn. Trong tương lai mình sẽ có bài viết về việc này.

Danh sách các Tag Helper có sẵn

Thư viện Microsoft.AspNetCore.Mvc.TagHelpers chứa nhiều các Tag Helper có sẵn cho các công việc thường dùng như tạo form, validate form, label, link...

TAG HELPER	TẠO RA	THUỘC TÍNH
Form Tag Helper	<form>	asp-action, asp-all-route-data, asp-area, asp-controller, asp-fragment, asp-host, asp-page, asp-page-handler, asp-protocol, asp-route, asp-route-
Anchor Tag Helpers	<a>	asp-action, asp-all-route-data, asp-area, asp-controller, asp-Fragment, asp-host, asp-page, asp-page-handler, asp-Protocol, asp-route, asp-route-
Cache Tag Helper	<cache>	enabled1, expires-after2, expires-on3, expires-sliding4, priority5, vary-by6
Environment Tag Helper	<environment>	names, include, exclude
Image Tag Helper		append-version
Input Tag Helper	<input>	for
Label Tag Helper	<label>	for
Link Tag Helper	<link>	href-include, href-exclude, fallback-href, fallback-href-include, fallback-href-exclude, fallback-test-class, fallback-test-value, fallback-test-property, fallback-test-value, append-version
Options Tag Helper	<select>	asp-for, asp-items
Partial Tag Helper	<partial>	name, model, for, view-data
Script Tag Helper	<script>	src-include, src-exclude, fallback-src, fallback-src-include, fallback-src-exclude, fallback-test, append-version
Select Tag Helper	<select>	for, items
Textarea Tag Helper	<textarea>	for
Validation Message Tag Helper		validation-for
Validation Summary Tag Helper	<div>	validation-summary

Chú ý: Tất cả các bài viết trên TEDU.COM.VN đều thuộc bản quyền TEDU, yêu cầu dẫn nguồn khi trích lại trên website khác.

28. Input Tag Helper trong ASP.NET Core

Input Tag Helper tạo ra phần tử HTML tương ứng với thuộc tính của Model là input. Thuộc tính model được kết hợp với tag helper sử dụng thuộc tính asp-for. Input tag helper tạo ra các thuộc tính

HTML tương ứng như type, name và id dựa trên các thông tin của thuộc tính model như kiểu dữ liệu và data annotation được áp dụng vào thuộc tính của ViewModel. ViewModel phải gắn vào một strongly typed view. Nó cũng cung cấp các validation cho thuộc tính giúp có thể hỗ trợ validate phía client. Thuộc tính asp-format giúp định dạng các dữ liệu nhập vào.

Input tag helper

Input tag helper áp dụng trên phần tử HTML là <input. Nó chỉ có 2 thuộc tính server-side là:

asp-for

asp-format

asp-for

Thuộc tính này kết hợp với thuộc tính tương ứng trong View Model và tạo ra HTML dựa trên thông tin thuộc tính đó như kiểu, tên, và data annotation (chỉ dẫn trên thuộc tính của ViewModel).

```
<input asp-for="<Expression Name>" />
```

Ví dụ về asp-for

Bạn có một model có thuộc tính Name:

```
public string Name { get; set; }
```

Ở view sẽ có một phần tử input:

```
<input asp-for="Name" /><br />
```

Nó sẽ tạo ra một phần tử HTML như sau khi thực thi:

```
<input type="text" id="Name" name="Name" value="" /><br />
```

Các thuộc tính của HTML Input

Thuộc tính asp-for tự động tạo ra phần tử HTML dựa trên các thông số sau đây:

Kiểu chỉ ra trong HTML

Data annotation được áp dụng cho thuộc tính model

Kiểu dữ liệu của thuộc tính đó trong .NET được dùng

Kiểu chỉ ra trong HTML

Type là thuộc tính được định nghĩa trong HTML và nó sẽ không bị ghi đè:

[EmailAddress]

```
public string Email { get; set; }
```

View sẽ là:

```
<input type="text" asp-for="Email" /><br />
```

HTML sẽ tạo ra một input text vì cái data annotation là [EmailAddress] nó không ảnh hưởng đến kết quả HTML tạo ra. Nó vẫn tạo ra như sau:

```
<input type="text" id="Email" name="Email" value="" /><br />
```

Dựa trên thuộc tính Data Annotation

Thuộc tính Data Annotation được áp dụng cho model để tạo ra thuộc tính type. Ví dụ ta có data annotation EmailAddress nó sẽ tự dịch ra type="email":

```
[EmailAddress]
```

```
public string Email { get; set; }
```

Code view:

```
<input asp-for="Email" /><br />
```

HTML tạo ra sẽ là:

```
<input type="email" data-val="true" data-val-email="The Email field is not a valid e-mail address." id="Email" name="Email" value="" /><br />
```

Đây là danh sách các Data Annotation và các kiểu của input tạo ra tương ứng nếu sử dụng chúng:

ATTRIBUTE	INPUT TYPE
[EmailAddress]	type="email"
[Url]	type="url"
[HiddenInput]	type="hidden"
[Phone]	type="tel"
[DataType(DataType.Password)]	type="password"
[DataType(DataType.Date)]	type="date"
[DataType(DataType.Time)]	type="time"

Dựa trên .NET Type

Nếu thuộc tính Data Annotation không được chỉ ra, thì Input Tag Helper sẽ sử dụng kiểu .NET của thuộc tính đó để xác định kiểu của HTML tạo ra. Ví dụ:

Model:

```
public DateTime DateOfJoining { get; set; }
```

View

```
<input asp-for="DateOfJoining" />
```

HTML:

```
<input type="datetime-local" data-val="true" data-val-required="The DateOfJoining field is required." id="DateOfJoining" name="DateOfJoining" value="0001-01-01T00:00:00.000" /><br />
```

Hoặc là kiểu boolean:

Model

```
public bool isActive { get; set; }
```

View:

```
<input asp-for="isActive" /><br />
```

HTML:

```
<input type="checkbox" data-val="true" data-val-required="The isActive field is required." id="isActive" name="isActive" value="true" /><br />
```

Đây là danh sách các kiểu .NET hay dùng và input tạo ra tương ứng:

NET TYPE	INPUT TYPE
Bool	type="checkbox"
String	type="text"
DateTime	type="datetime-local"
Byte, int, Single, Double	type="number"
decimal, double, float	type="text"

Thuộc tính id và name trong HTML

Thuộc tính id và name trong HTML nhận từ biểu thức tên của thuộc tính qua asp-for

[EmailAddress]

```
public string Email { get; set; }
```

```
<input type="text" asp-for="Email" /><br />
```

Nó sẽ tạo ra HTML như sau. Thuộc tính id và name giống như giá trị được điền trong asp-for

```
<input type="text" id="Email" name="Email" value="" /><br />
```

Các thuộc tính con

Hãy xem xét model person dưới đây có chứa thuộc tính Address và một mảng màu:

```
public class person
```

```
{
```

```
    public List<string> colors { get; set; }
```

```
    public Address Address { get; set; }
```

```
    public person()
```

```
    { Address = new Address();
```

```
      colors = new List<string>{"red","blue"};    }}
```

```
public class Address
{
    public string address1 { get; set; }
}
```

Bạn có thể tham chiếu đến address1:

```
<input asp-for="Address.address1" /><br />
```

Đoạn code ở trên tạo ra HTML như bên dưới. Chú ý rằng thuộc tính name và id giống với biểu thức trong asp-for nhé. Thuộc tính id sử dụng dấu gạch dưới thay cho dấu chấm:

```
<input type="text" id="Address_address1" name="Address.address1" value="" /><br />
```

Danh sách màu:

```
<input asp-for="colors[0]" /><br />
```

Nó sẽ tạo ra HTML và chỉ ra tên của phần tử dựa trên dấu [] và chỉ số ở trong

```
<input type="text" id="colors_0_" name="colors[0]" value="red" /><br />
```

Các thuộc tính Validation

Input Tag Helper cũng tạo ra các thuộc tính validate dữ liệu trong HTML. Các thuộc tính validate dữ liệu bắt đầu bằng data-val-*. Chúng chứa các thông tin validation như giá trị lớn nhất nhỏ nhất, bắt buộc nhập hay không, miền giá trị cho phép, biểu thức quy tắc hay thông báo lỗi...

ASP.NET Core Unobtrusive Client Validation framework sử dụng các thuộc tính data-val-* này để validate dữ liệu trên phía client. Input tag helper xác định các thuộc tính validation bằng cách phân tích các kiểu trong .NET và data annotation áp dụng cho thuộc tính model.

Ví dụ áp dụng thuộc tính [Required] cho kiểu string, thuộc tính data-val và data-val-required sẽ được thêm vào thẻ input HTML:

```
[Required]
```

```
public string Name { get; set; }
```

HTML tạo ra:

```
<input asp-for="@Model.Name" /><br />
```

```
<input type="text" data-val="true" data-val-required="The Name field is required." id="Name" name="Name" value="" /><br />
```

Thêm thuộc tính ErrorMessage vào Required:

```
[Required(ErrorMessage = "Please enter the name")]
```

```
public string Name { get; set; }
```

View:

```
<input asp-for="Name" /><br />
```

HTML tạo ra sẽ là:

```
<input type="text" data-val="true" data-val-required="Please enter the name" id="Name" name="Name" value="" /><br />
```

Kiểu DateTime tự động tạo ra data-val-required, ngay cả khi thuộc tính [Required] không được chỉ ra:

```
public DateTime DateOfJoining { get; set; }
```

```
<input asp-for="@Model.DateOfJoining" />
```

HTML tạo ra:

```
<input type="datetime-local" data-val="true" data-val-required="The DateOfJoining field is required." id="DateOfJoining" name="DateOfJoining" value="0001-01-01T00:00:00.000" /><br />
```

Bạn có thể bỏ việc này bằng cách đặt kiểu DateTime là nullable:

```
public DateTime? DateOfJoining { get; set; }
```

```
<label for="DateOfJoining">DateOfJoining</label>
```

HTML tạo ra:

```
<input type="datetime-local" id="DateOfJoining" name="DateOfJoining" value="" /><br />
```

Danh sách một số kiểu của data annotation sẽ tạo ra thuộc tính validation tương ứng:

ATTRIBUTE	MÔ TẢ
-----------	-------

Compare	Chỉ ra field khác cần so sánh giá trị với field này
MaxLength	Chỉ ra số ký tự dài nhất có thể được chấp nhận
MinLength	Chỉ ra số ký tự ít nhất có thể được chấp nhận
Range	Chỉ ra miền giá trị được chấp nhận
RegularExpression	Giá trị phải tuân theo regular expression
Remote	Bật tính năng validation phía client đến 1 server remote, ví dụ kiểm tra giá trị username có tồn tại hay không.
Required	Giá trị bắt buộc phải nhập. Chú ý là các kiểu DateTime hay kiểu số mặc định sẽ có cái này, bạn cần sử dụng nullable để bỏ nó đi
StringLength	Chỉ ra độ dài tối đa của chuỗi được chấp nhận

asp-format

Thuộc tính asp-format định dạng chuỗi cho thuộc tính input. Ví dụ bạn format số với 2 số thập phân đằng sau:

```
public decimal Balance { get; set; }
```

```
<input asp-for="Balance" asp-format="{0:N2}" /><br />
```

HTML tạo ra:

```
<input type="text" data-val="true" data-val-number="The field Balance must be a number." data-val-required="The Balance field is required." id="Balance" name="Balance" value="0.00" /><br />
```

Đây là [danh sách các format type](#) dùng cho as-format

29. Environment Tag Helper trong ASP.NET Core

Environment Tag Helper hỗ trợ tạo ra các nội dung phụ thuộc vào biến quy định môi trường trong ASP.NET Core. Bài này chúng ta sẽ tìm hiểu về nó nhé.

Environment Tag Helper

Environment Tag Helper áp dụng sử dụng thẻ `<environment>`. Nó xác định giá trị hiện tại của biến môi trường `ASPNETCORE_ENVIRONMENT` để render ra nội dung dựa trên giá trị của biến. Giá trị của `ASPNETCORE_ENVIRONMENT` được truy cập bởi `IHostingEnvironment.EnvironmentName`. Biến môi trường này đọc tại thời điểm ứng dụng khởi động và giá trị được đọc vào `IHostingEnvironment.EnvironmentName`. Theo quy ước thì có 3 giá trị tương ứng với 3 môi trường được hỗ trợ: **Development**, **Staging** và **Production**. Tuy nhiên bạn có thể đặt bất cứ tên môi trường nào tùy thích.

Thuộc tính Environment Tag Helper

Environment Tag Helper hỗ trợ 3 thuộc tính: **include**, **exclude**, **names**

include

Dấu hỏi được dùng để ngăn cách các giá trị là tên môi trường mà nội dung bên trong cặp thẻ sẽ tạo ra ở các môi trường đó.

```
<environment include="Development">
```

```
<h1>This only renders in the Development Environment</h1>
```

```
</environment>
```

```
<environment include="Staging,Production">
```

```
<h1>This only renders in the Staging and Production Environments!</h1>
```

```
</environment>
```

exclude

Một danh sách các tên môi trường cũng được chia tách trong giá trị của thuộc tính `exclude`. Thuộc tính này quy định nội dung được hiển thị ở tất cả các môi trường trừ các môi trường trong danh sách này. `Exclude` cho ASP.NET Core biết rằng sẽ không render ra nội dung nếu tên môi trường hiện tại đúng với môi trường đã chỉ ra:

```
<environment exclude="Development">
```

```
  <h1>This content is not rendered in the Development Environments!</h1>
```

```
</environment>
```

Cả `include` và `exclude` được áp dụng trùng một tên môi trường thì `exclude` luôn được ưu tiên.

names

Một dấu phẩy cũng được dùng để chia tách tên môi trường mà nội dung được render. Nếu tên môi trường hiện tại cũng nằm trong danh sách `exclude` thì nội dung vẫn không được render. Vì `exclude` được ưu tiên cao nhất.

```
<environment names="Development">
```

```
  <h1>This only renders in the Development Environment</h1>
```

```
</environment>
```

```
<environment names="Development,Production">
```

```
  <h1>This renders in Development and Production Environments</h1>
```

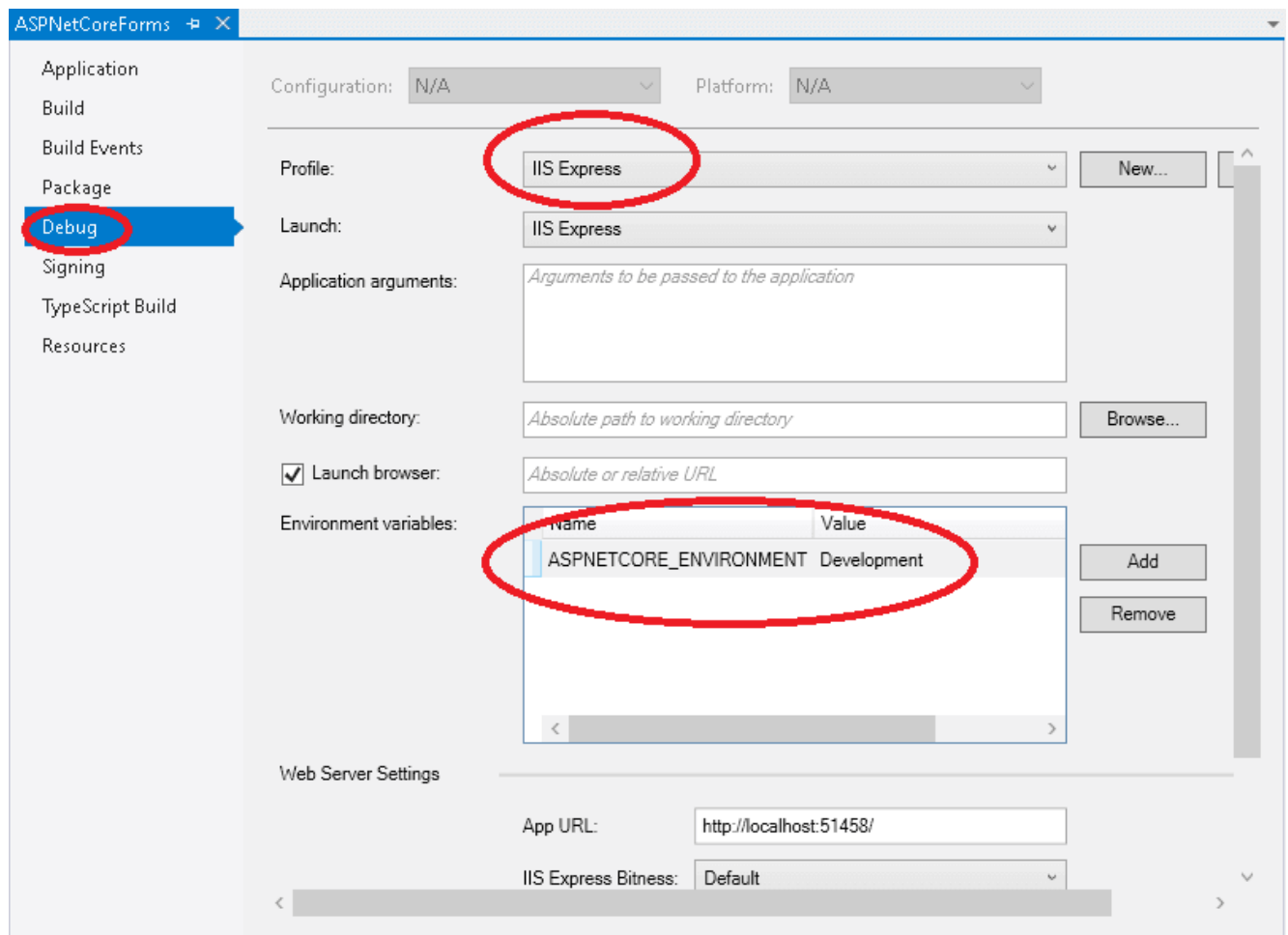
```
</environment>
```

Chỉ ra biến môi trường

Bạn có thể chỉ ra biến môi trường từ Properties của Project hoặc qua file `launchSettings.json`.

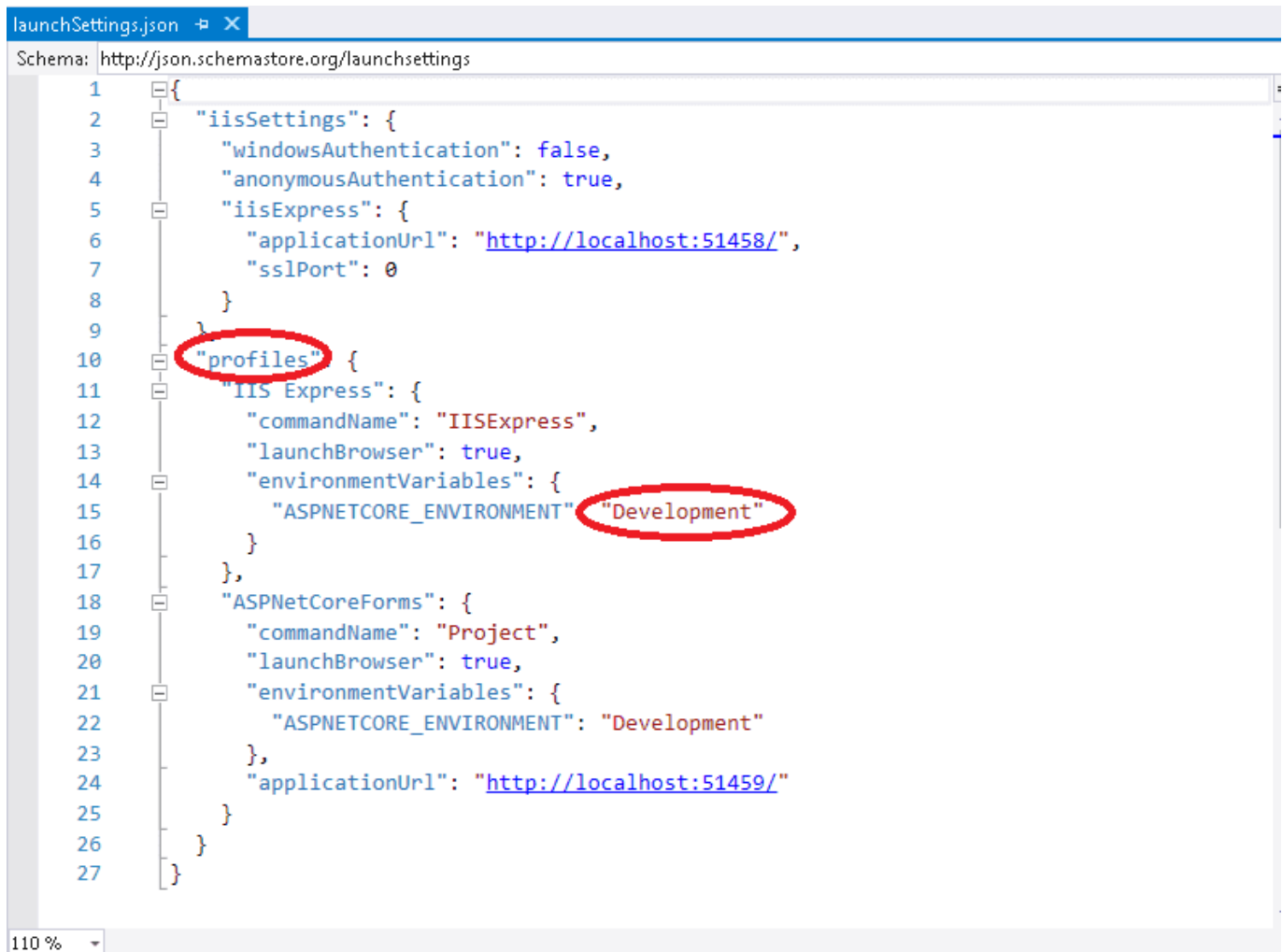
Project Properties

Mở Project Properties ra và chọn tab Debug. Trong phần tên biến môi trường chọn giá trị biến `ASPNETCORE_ENVIRONMENT` và thay đổi giá trị. Nếu tên biến là `ASPNETCORE_ENVIRONMENT` không tồn tại hãy click vào nút thêm mới.



launchSettings.json

Cách khác là thay đổi biến môi trường sử dụng **launchSettings.json** như dưới đây:



Ví dụ sử dụng Environment Tag Helper

Hầu hết các trường hợp sử dụng Environment Tag Helper là sử dụng để nhúng các file CSS hoặc JavaScript cho các môi trường khác nhau như **Development** hay **Production**. Ví dụ bạn cần load version full của CSS trong môi trường **Development** để debug tuy nhiên với **Production** bạn chỉ cần nhúng file đã bundle tức là file đã nén để tăng tốc độ và giảm băng thông truyền dữ liệu.

Ví dụ dưới đây cho phép nhúng file đã nén vào của **style.css** trên tất cả các môi trường trừ môi trường **Development**.

```
<environment include="Development">
```

```
  <link rel="stylesheet" href="~/css/style.css" />
```

```
</environment>
```

```
<environment exclude="Development">
```

```
  <link rel="stylesheet" href="~/css/style.min.css" />
```

```
</environment>
```

Exclude được ưu tiên

Nếu bạn sử dụng các thuộc tính include và exclude trong cùng 1 tag helper thì exclude luôn được ưu tiên. Ví dụ đoạn code dưới đây, môi trường Development là một giá trị quy định trong cả exclude và include. Trong trường hợp này, nội dung sẽ không được render ra cho môi trường Development do danh sách exclude được ghi đè danh sách include hoặc danh sách names.

```
<environment include="Development, Staging" exclude="Development">
    <h1>This will not show in Development or Production, but it will show in
    Staging.</h1>
</environment>
```

30. Cơ chế Model Binding: Truyền dữ liệu từ View lên Controller

Trong cơ chế **Model Binding** của ASP.NET Core chúng ta sẽ học cách làm sao để truyền dữ liệu từ View lên Controller. Chúng ta cũng sẽ tìm hiểu về **Model Binding** và cơ chế hoạt động của nó. ASP.NET Core cho phép chúng ta bind dữ liệu từ nhiều nguồn khác nhau như HTML Form sử dụng **[FromForm]**, từ giá trị route **[FromRoute]**, từ query string **[FromQuery]**, từ body của request **[FromBody]** và từ Header của request **[FromHeader]**.

Một số bài hướng dẫn trước đây có thể tham khảo:

Model và ViewModel trong ASP.NET Core MVC

Truyền dữ liệu từ Controller sang View trong ASP.NET Core

Xây dựng HTML Form trong ASP.NET Core

Strongly Typed View trong ASP.NET Core

Tag Helpers trong ASP.NET Core MVC

Model Binding là gì?

Model Binding là cơ chế map dữ liệu được gửi qua HTTP Request vào các tham số của action method trong Controller. HTTP Request có thể chứa dữ liệu từ nhiều định dạng. Dữ liệu có thể chứa trong HTML Form. Nó có thể là một phần của route value hoặc trên query string hay có thể là một body của request.

Cơ chế ASP.NET Core model binding cho phép chúng ta dễ dàng bind các giá trị này vào các tham số của action method. Các tham số này có thể là kiểu nguyên thủy hoặc kiểu đối tượng phức tạp.

Lấy dữ liệu từ Form Data trong Controller

Trong bài Tag Helper, chúng ta đã tạo một form cơ bản cho phép nhận một đối tượng **Product**. Khi người dùng click nút **Submit** thì dữ liệu sẽ được post lên phương thức **Create** trên Controller. Trong project đó chúng ta cũng tạo một **ProductEditModel** class chứa chi tiết của sản phẩm cần được tạo hoặc chỉnh sửa:

```
public class ProductEditModel
{
    public int ID{ get; set; }
    public string Name { get; set; }
    public decimal Rate { get; set; }
    public int Rating { get; set; }
}
```

Một form được tạo chứa 3 field: **Name**, **Rate** và **Rating**:

```
<form action="/home/Create" method="post">
    <label for="Name">Name</label>
    <input type="text" name="Name" />

    <label for="Rate">Rate</label>
    <input type="text" name="Rate" />
    <label for="Rating">Rating</label>
    <input type="text" name="Rating" />
    <input type="submit" name="submit" />
</form>
```

Action method **Create** trong **HomeController**:

```
[HttpPost]
public IActionResult Create(ProductEditModel model)
{
    string message = "";
    if (ModelState.IsValid)
    {
        message = "product " + model.Name + " created successfully" ;
    }
    else
    {
        message = "Failed to create the product. Please try again";
    }
}
```

```

    }

    return Content(message);
}

```

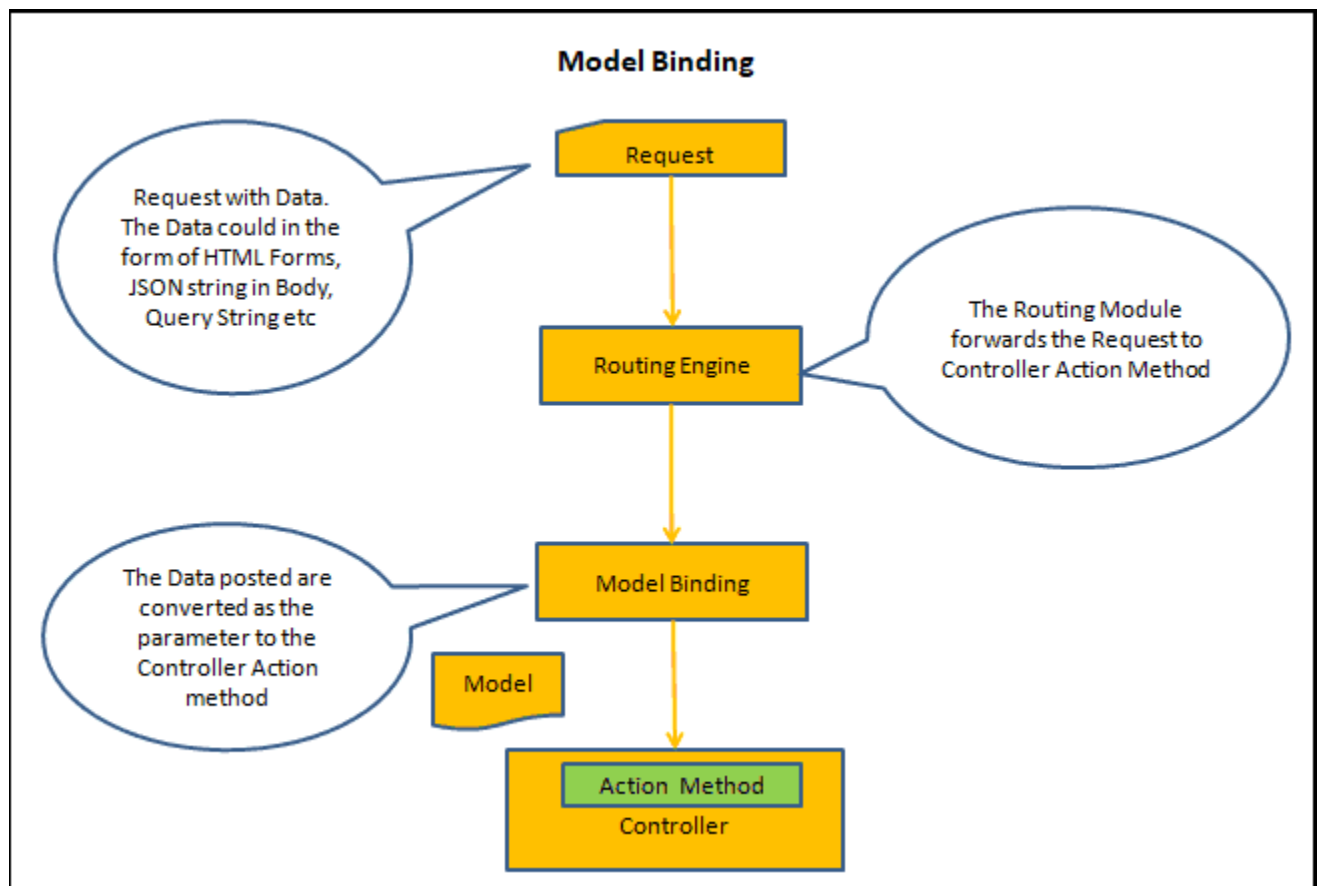
Một submit của form trên, các giá trị trong form sẽ tự động được map vào đối tượng **ProductEditModel** trong Action method của controller:

```
public IActionResult Create(ProductEditModel model)
```

Cơ chế này tự động xảy ra đằng sau được gọi là Model Binding. Model Binder sẽ tìm các trường tương ứng giữa tham số **ProductEditModel** với trường trong form, route value hoặc query string...

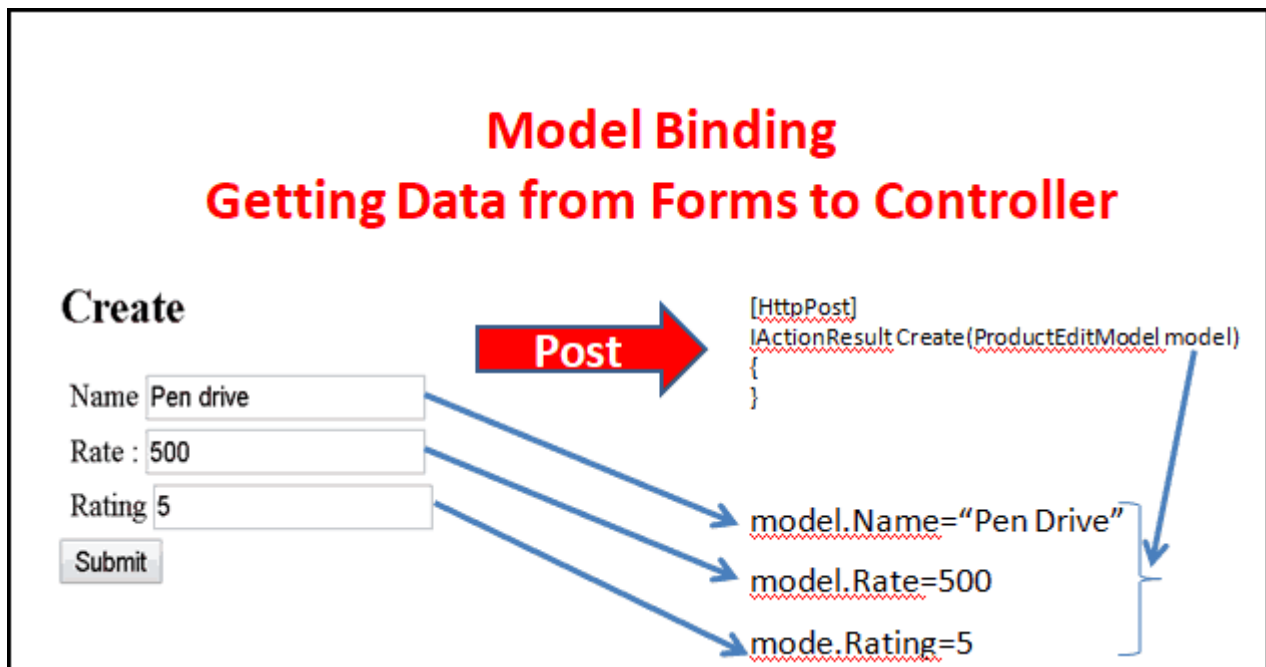
Cơ chế Model Binding làm việc như thế nào?

Hình dưới đây minh họa cơ chế làm việc của Model binding:



Khi người dùng click vào nút Submit thì một request Post được gửi lên server với Form Data, QueryString, Route Parameter...MVCRouteHandler của Routing Engine sẽ xử lý request đến và có trách nhiệm gọi action method tương ứng. Model Binder sẽ được kích hoạt trước khi action method được gọi. Nó tìm dữ liệu thỏa mãn trong form data, query string và request parameter trong HTTP Request. Sau đó nó sẽ binding các giá trị vào tham số của action method qua tên.

Ví dụ, trường "**name**" trong form sẽ được map vào thuộc tính "**Name**" trong **ProductEditModel**.
Rate trong form sẽ được map vào thuộc tính Rate...



Để Model binding làm việc đúng:

Thuộc tính Name phải match với Request Data

Các thuộc tính phải đặt public set

Model Binder

Model Binder có trách nhiệm gán dữ liệu vào các tham số của action method. Model Binder được tạo mỗi model binder provider. Model binder phải được implement interface **IModelBinderProvider**. Nghĩa là bạn có thể tạo một Model Binder của riêng mình hoặc mở rộng nó bằng cách triển khai interface **IModelBinderProvider**. Custom model binder phải được đăng ký trong **ModelBinderProviders** trong **Startup.cs**.

```
services.AddMvc(options =>
{
    options.ModelBinderProviders.Add(new CustomModelBinderProvider());
});
```

ModelState

Nếu Model binder không thành công trong việc bind dữ liệu từ Request vào thuộc tính model tương ứng, nó sẽ không đưa ra bất cứ thông báo lỗi nào. Nhưng nó sẽ update đối tượng ModelState với danh sách lỗi và set thuộc tính IsValid là false.

Vì thế kiểm tra **ModelState.IsValid** sẽ cho chúng ta thấy quá trình binding có thành công hay không.

Ví dụ: Trong ví dụ trên khi click nút **submit** mà không nhập bất cứ dữ liệu gì trên form thì kết quả sẽ ra validate thất bại và vì thế **ModelState.IsValid** sẽ là **false**.

Nếu không dùng model binding thì sao?

Trước khi chúng ta tìm hiểu sâu hơn về Model binding, chúng ta cần hiểu nếu không có model binding thì chúng ta sẽ truy cập đến dữ liệu từ request kiểu gì? Xem lại code mà chúng ta đã tạo trong phần trước. Thêm mới action method **NoModelBinding**:

```
[HttpPost]

public IActionResult NoModelBinding()
{
    ProductEditModel model = new ProductEditModel();
    string message = "";
    model.Name = Request.Form["Name"].ToString();
    model.Rate = Convert.ToDecimal( Request.Form["Rate"]);
    model.Rating =Convert.ToInt32( Request.Form["Rateing"]);
    message = "product " + model.Name + " created successfully";
    return Content(message);
}
```

Và thay đổi thành:

```
<form action="/home/NoModelBinding" method="post">
```

Truy cập trực tiếp đến query string

Tương tự như thế, bạn có thể truy cập đến các giá trị trên query string sử dụng **Request.Query**. Lấy các giá trị trên query string.

Ví dụ, **Request.Query["id"].ToString()** trả về giá trị của **id** trên query string. Sử dụng **Request.QueryString.HasValue** sẽ cho bạn biết nếu có giá trị query string trên URL hiện tại hay không và **Request.QueryString.Value** sẽ trả về giá trị thô của query string.

Truy cập trực tiếp đến Request Headers

Tương tự như thế, bạn có thể sử dụng **Request.Headers** để truy cập các giá trị được gửi lên thông qua HTTP Header.

Truy cập đến Route Data

Để truy cập đến route bạn cần ghi đè phương thức **OnActionExecuting**:

```
using Microsoft.AspNetCore.Mvc.Filters;

public override void OnActionExecuting(ActionExecutingContext context)
{

```

```
string id = context.RouteData.Values["id"].ToString();
```

```
base.OnActionExecuting(context); }
```

Bạn thấy rằng có rất nhiều code để lấy giá trị được post lên HTTP Request. ASP.NET Core model binding sẽ làm giúp bạn các việc này mà dùng ít code hơn.

Các nguồn cho Model binding

Như đã nhắc đến trước đây, model binder có thể lấy dữ liệu từ rất nhiều nơi khác nhau. Đây là danh sách các nguồn dữ liệu theo thứ tự mà model binding sẽ tìm:

HTML Form Value

Route Value

Query String

Model binder cũng có thể tìm dữ liệu từ các nguồn sau, nhưng chúng ta cần chỉ ra nguồn nào cần lấy một cách tường minh:

Request Body

Request Header

Services

Lấy dữ liệu từ Form và Query String

Hãy thử binding action parameter với cả form và query string. Phương thức **FormAndQuery** sẽ như sau:

```
[HttpGet]
```

```
public IActionResult FormAndQuery()
```

```
{
```

```
    return View();
```

```
}
```

```
[HttpPost]
```

```
public IActionResult FormAndQuery(string name, ProductEditModel model)
```

```
{    string message = "";
```

```
    if (ModelState.IsValid)
```

```
    {
```

```
        message = "Query string " + name + " product " + model.Name + " Rate " +  
model.Rate + " Rating " + model.Rating ;    }
```

```
    else
```

```
    {        message = "Failed to create the product. Please try again";    }
```



```
return Content(message); }
```

Chú ý rằng action method **FormAndQuery** có hai tham số **name** và **ProductEditModel**.

```
public IActionResult FormAndQuery(string name, ProductEditModel model)
```

Tiếp theo, chúng ta tạo view **FormAndQuery** như sau:

```
<form action="/home/FormAndQuery/?name=Test" method="post">
```

```
<label for="Name">Name</label>
```

```
<input type="text" name="Name" />
```

```
<label for="Rate">Rate</label>
```

```
<input type="text" name="Rate" />
```

```
<label for="Rating">Rating</label>
```

```
<input type="text" name="Rating" />
```

```
<input type="submit" name="submit" />
```

```
</form>
```

Form có tên trường là **"name"**. Chúng ta cũng gửi **"name=test"** qua query string đến controller action:

```
<form action="/home/FormAndQuery/?name=Test" method="post">
```

Trong ví dụ trên, tham số **"name"** xuất hiện 2 lần như là một phần của query string. Khi form được submit, tham số **"name"** luôn map đến trường trong form chứ không phải query string. Bởi vì model binder luôn sử dụng thứ tự map dữ liệu theo thứ tự:

Form Values

Route Values

Query Strings

Vì thế các giá trị trong form có trường **name**, tham số **name** luôn được gán giá trị. Chúng ta có thể thay đổi hành vi này bằng cách thêm thuộc tính **[FromQuery]**. Sử dụng **[FromQuery]** như sau:

```
public IActionResult FormAndQuery([FromQuery] string name, ProductEditModel model)
```

Giờ nếu submit form thì tham số **name** sẽ lấy giá trị từ query string, trong khi **ProductEditModel** sẽ lấy giá trị từ form.

Điều khiển Binding Source

Trong ví dụ trước, chúng ta sử dụng **[FromQuery]** để bắt buộc model binder đổi hành vi mặc định và sử dụng query string làm nguồn cho binding. ASP.NET Core cung cấp cho chúng ta một số thuộc tính điều khiển và chọn nguồn nào sẽ được nhận khi binding:

[FromForm], [FromRoute], [FromQuery], [FromBody], [FromHeader], [FromServices]

[FromForm]

[FromForm] ép model binder bind tham số vào các trường của HTML Form.

[HttpPost]

```
public IActionResult Create([FromForm] ProductEditModel model) { }
```

[FromRoute]

[FromRoute] ép model binder bind tham số vào route data từ request.

Ví dụ: Tạo một action method FromRoute, cho phép một giá trị **id** và **ProductEditModel**. Chúng ta có 2 tham số id. MVC mặc định sẽ có tham số Id qua route và nó là tùy chọn. **ProductEditModel** cũng có thuộc tính id.

[HttpGet]

```
public IActionResult FromRoute()  
{  
    return View();  
}
```

[HttpPost]

```
public IActionResult FromRoute(string id, ProductEditModel model)  
{  
    string message = "";  
    if (ModelState.IsValid)  
    {  
        message = "Route " + id + " Product id " + model.id + " product " +  
model.Name + " Rate " + model.Rate + " Rating " + model.Rating; }  
    else { message = "Failed to create the product. Please try again"; }  
    return Content(message);  
}
```

Tạo ra view **FromRoute**

```
<form action="/home/FromRoute/Test" method="post">  
    <label for="id">id</label>  
    <input type="text" name="id" />  
    <label for="Name">Name</label>  
    <input type="text" name="Name" />  
    <label for="Rate">Rate</label>
```

```



```

Chú ý là chúng ta đang gọi controller action bằng cách sử dụng **"test"** như là giá trị route:

```
<form action="/home/FromRoute/Test" method="post">
```

Giờ khi submit form thì tham số id luôn map vào id từ form thay vì từ route value. Giờ hãy mở **HomeController** ra và áp dụng [FromRoute] trên tham số **id**.

```
public IActionResult FromRoute([FromRoute] string id, ProductEditModel model)
```

Khi submit form thì id sẽ được nhận là **"test"**

Binding query string sử dụng [FromQuery]

[FromQuery] ép model binder bind giá trị vào tham số từ giá trị lấy từ query string.

Binding đến Request body sử dụng [FromBody]

[FromBody] ép model binder bind dữ liệu từ request body. Formatter sẽ chọn dựa trên **content-type** của request. Dữ liệu trong request body có nhiều format khác nhau như JSON, XML...Model binder sẽ tìm thuộc tính **content-type** trên header và chọn **formatter** để chuyển dữ liệu về kiểu mong muốn.

Ví dụ, khi content-type là **'application/json'**, model binder sử dụng **JsonInputFormatter** class để chuyển request body và map vào tham số.

Binding từ Request Header sử dụng [FromHeader]

[FromHeader] map các giá trị từ request header vào tham số của action:

```

public IActionResult FromHeader( [FromHeader] string Host)
{
    return Content(Host);
}

```

Vô hiệu hoá binding với [BindNever]

Để cho model binder biết rằng không bind cho thuộc tính nào đó.

[BindNever]

```
public int Rating { get; set; }
```

Giờ thì model binder sẽ bỏ qua thuộc tính Rating ngay cả khi form có trường Rating.

Bắt buộc Binding với [BindRequired]

Cái này ngược hẳn với [BindNever]. Trường nào được đánh dấu là BindRequired phải luôn hiển thị trên form và binding phải thực hiện nếu không thì ModelState.IsValid sẽ false.

31. Model Validation trong ASP.NET Core

Bài viết này chúng ta sẽ học về cơ chế Model Validation. Thường thì dữ liệu được nhập bởi người dùng sẽ không hợp lệ và không thể đưa vào cơ sở dữ liệu. Các dữ liệu được nhập có thể chứa một số lỗi chính tả hoặc người dùng cố tính nhập các dữ liệu không phù hợp. Vì thế chúng ta cần phải kiểm tra dữ liệu người dùng nhập vào trước khi lưu trữ vào cơ sở dữ liệu. ASP.NET Core cung cấp cho chúng ta thành phần gọi là Model Validator, nó dùng các attribute để kiểm tra dữ liệu trong model dễ dàng hơn. Chúng ta cũng tìm hiểu về ModelState và cách sử dụng nó.

Giới thiệu về Model Validation

Form Data post dữ liệu lên Controller action tự động được map vào các tham số của action bởi Model Binder như chúng ta đã tìm hiểu ở bài trước. Model cần kiểm tra dữ liệu đầu vào xem có hợp lệ không. Quá trình kiểm tra này có thể được hoàn thành bởi client trước khi gửi lên server hoặc server kiểm tra sau khi nhận được từ client. Cơ chế validation phía client (client-side validation) rất quan trọng vì nó giúp tăng trải nghiệm người dùng khi kiểm tra dữ liệu mà không cần chờ đến server nhưng phía server lại nên đảm bảo một lần nữa để các dữ liệu không hợp lệ không thể đưa vào hệ thống.

Sự quan trọng của client-side validation

Giúp tăng sự trải nghiệm

Vì việc kiểm tra tiến hành phía trình duyệt client nên phản hồi nhanh hơn và gần như là ngay lập tức

Tiết kiệm tài nguyên server như là băng thông bằng cách giảm truy vấn đến server.

Sự quan trọng của server-side validation

Client-side validation cung cấp trải nghiệm người dùng tốt hơn nhưng không tin cậy. Nó có thể lỗi do một trong các lý do sau:

Javascript có thể bị tắt ở trình duyệt

Người dùng có thể gửi trực tiếp dữ liệu đến người dùng mà không sử dụng ứng dụng hoặc sử dụng một số các trình chỉnh sửa request có hại.

Khi Javascript có lỗi thì kết quả là dữ liệu được đưa vào hệ thống mà có thể không hợp lệ

Vì thế điều quan trọng là kiểm tra dữ liệu phải được thực hiện ở cả phía server, ngay cả bạn đã validate ở phía client.

Kiểm tra model một cách tường minh

Một khi bạn nhận model trong controller, bạn có thể kiểm tra model đó bằng cách viết code như sau:

```
if (string.IsNullOrEmpty(model.Name))
{
    //Validation Failed

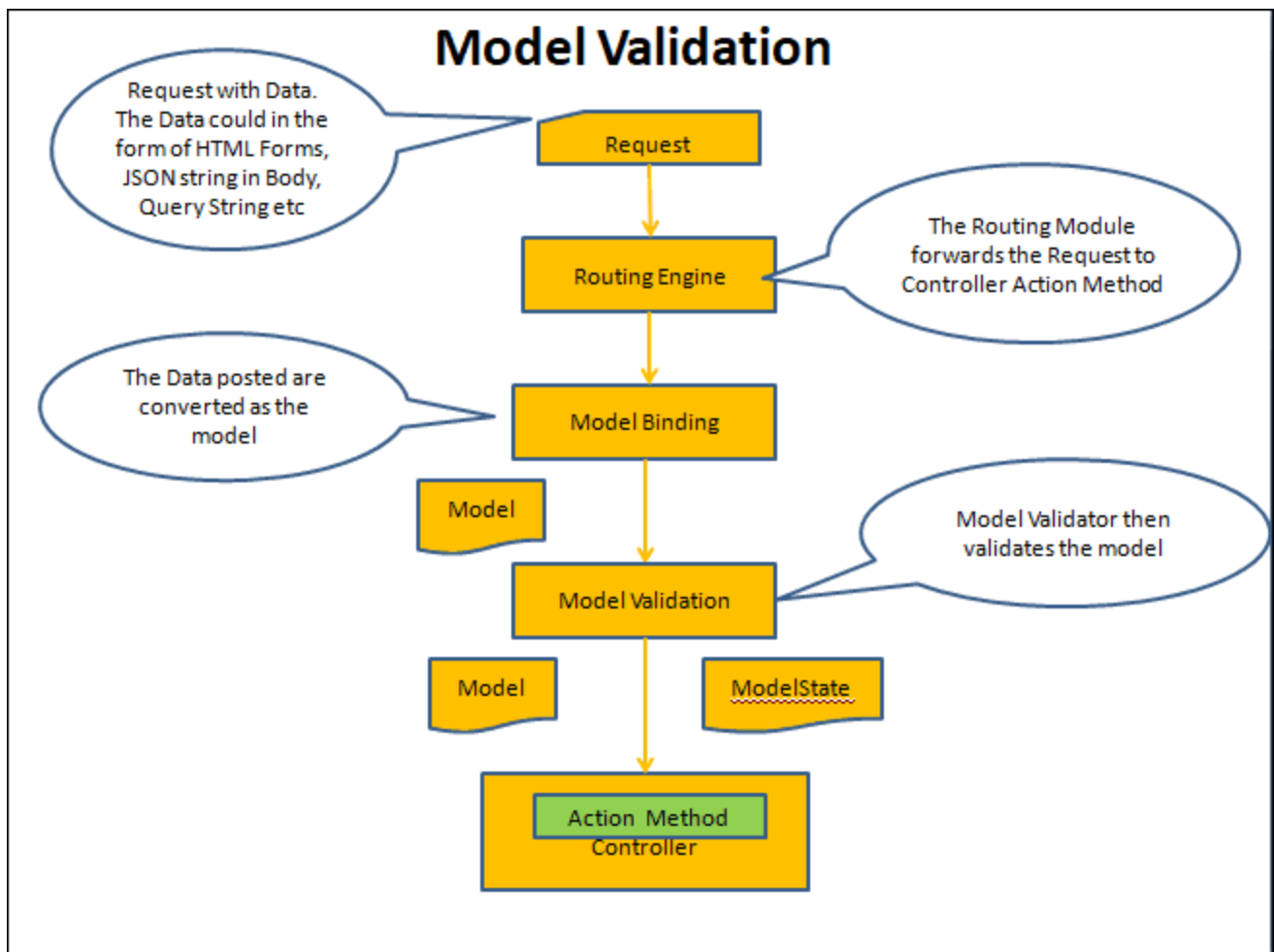
    //Send the list of errors to client }
```

Đoạn code trên đơn thuần chỉ kiểm tra xem thuộc tính name của model có rỗng hay null không. Đoạn code trên làm việc tốt, nhưng bạn sẽ phải làm nhiều lần tương tự nếu có nhiều thuộc tính trong model, khi đó bạn cần kiểm tra với các đoạn code giống nhau. Ở đây bạn cần xem làm sao để gửi các báo lỗi về client để hiển thị cho người dùng. Model Validator sẽ làm điều này cho bạn mà không cần viết các đoạn code thừa thãi.

Model Validation làm việc ra sao?

Chúng đã tìm hiểu cơ chế Model Binding làm việc ra sao trong ASP.NET Core rồi đúng không? Vậy khi HTTP Request được đưa tới Model Binder thì nó sẽ được gọi trước khi truyền tham số vào Controller action. Model Binder sẽ không chỉ map dữ liệu vào action method mà nó còn kiểm tra chúng sử dụng Model Validator.

Model Validator chạy sau model binding và chạy một loạt các logic kiểm tra trên mỗi thuộc tính của model dựa trên các attribute bạn đặt cho các thuộc tính đó. Các attribute này gọi là Validation Attribute và chứa code sử dụng bởi Model Validator.



Tất cả các logic kiểm tra đều chạy phía server. ASP.NET Core có nhiều các attribute viết sẵn, bạn có thể thêm nó vào thuộc tính của model để cài đặt quy tắc kiểm tra. Các quy tắc này cũng được gọi

là **DataAnnotations** và nằm trong namespace: **System.ComponentModel.DataAnnotations**. Bạn cần import nó vào **ViewModel** và sử dụng.

Ví dụ, bạn đặt trường bắt buộc phải nhập sẽ dùng **[Required]**

```
[Required]
```

```
Public string Name {get;set;}
```

Model Binder không bắn ra bất cứ lỗi nào nếu quá trình kiểm tra không hợp lệ. Nhưng nó sẽ cập nhật đối tượng **ModelState** với danh sách lỗi và đặt thuộc tính **IsValid** là **false** trước khi gọi action method. Chúng ta cần kiểm tra **ModelState.IsValid** để biết xem quá trình kiểm tra có hợp lệ hay không để action method có thể trả về danh sách lỗi nếu cần.

```
if (ModelState.IsValid)
```

```
{    //Model is valid. Call Service layer for further processing of data }  
else {    //Validation failed. Return the model to the user with the relevant  
error messages }
```

Cách sử dụng Validation Attributes

Cập nhật Model với Data Annotation

Trong Model, thuộc tính thêm vào Data Annotation attribute được hiển thị bên dưới đây. Code sẽ được thêm vào attribute cho thuộc tính **Name**. Nó cũng được cài đặt thông báo trong trường hợp kiểm tra dữ liệu không hợp lệ với attribute đó:

```
[Required(AllowEmptyStrings =false,ErrorMessage ="Please enter the name")]
```

```
[StringLength(maximumLength:25,MinimumLength =10,ErrorMessage ="Length must be  
between 10 to 25")]
```

```
public string Name { get; set; }
```

Hiển thị danh sách lỗi Validations

Trong view, sử dụng Validation Tag Helper để hiển thị lỗi cho người dùng. Tag helper **asp-validation-summary** giúp hiển thị danh sách các thông báo lỗi cho riêng Model validation trên form. Nó được gắn vào thẻ div và đặt trên cùng của form. **asp-validation-for** hiển thị danh sách lỗi cho thuộc tính **Name** ở phía bên phải của nó. Nó được gắn vào thẻ span và đặt ngay cạnh thuộc tính.

Validation tag helper thêm class **field-validation-error** và **validation-summary-errors** vào HTML nếu lỗi tìm thấy. Vì thế chúng ta thêm các style CSS vào để hiển thị lỗi màu đỏ cho các class này. Bạn có thể đọc thêm phần Validation Tag Helper ở bài sau:

```
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

```
@model ASPNetCoreForms.Models.ProductEditModel
```

```
@{    ViewData["Title"] = "Create"; }
```

```
<style>
```

```
.field-validation-error {
```

```

        color: red
    }

    .validation-summary-errors {
        color: red
    }
</style>
<h2>Create</h2>
<form action="/home/create" method="post">
    <div asp-validation-summary="ModelOnly">
        <span>Please correct the following errors</span>
    </div>
    <label asp-for="Name">Name</label>
    <input asp-for="Name" />
    <span asp-validation-for="Name"></span>
    <br />
    <input type="submit" name="submit" />
</form>

```

Kiểm tra ModelState.IsValid trong Controller Action

Cuối cùng trong Controller action method chúng ta kiểm tra nếu **ModelState.IsValid** nếu có bất cứ lỗi nào thì chúng ta trả về cho người dùng. Chúng ta trả về model để hiển thị các giá trị được nhập đồng thời trả kèm cả danh sách lỗi được hiển thị bởi Validation tag helper:

```

[HttpPost]
public IActionResult Create(ProductEditModel model)
{
    string message = "";
    if (ModelState.IsValid){
        message = "product " + model.Name + " created successfully"; }
    else { return View(model); }
    return Content(message); }

```

Đoạn code trên sẽ hiển thị như sau:

Create

Name

Please enter the name

Submit

Create

Name

Length must be between 10 to 25

Submit

ModelState

Model Binder cập nhật đối tượng ModelState với kết quả của Model Binding và validation. ModelState sẽ lưu chi tiết của các giá trị được cập nhật lên model và cả các thông tin lỗi trong quá trình validation xuất hiện trong mỗi thuộc tính. ModelState là thuộc tính của ControllerBase và thuộc kiểu **ModelStateDictionary**.

Các thuộc tính của ModelState

Các phương thức của ModelState

AddModelError

Thêm một message lỗi cụ thể vào danh sách Errors với 1 key.

Clear

Xoá toàn bộ các key và value trong thể hiện của **ModelStateDictionary**

ClearValidationState(string)

Xoá phần tử trong **ModelStateDictionary** đúng với key truyền vào

GetFieldValidationState(string)

Trả về **ModelValidationState** với key truyền vào

TryAddModelError(string, Exception, ModelMetadata)

Thêm một exception cụ thể vào Errors với một key

Ví dụ sử dụng ModelState

Lấy danh sách Errors

Lập quá danh sách key và lấy danh sách Errors trong ModelState


```

foreach (var modelStateKey in ModelState.Keys)
{
    var modelStateVal = ModelState[modelStateKey];
    foreach (var error in modelStateVal.Errors)
    {
        var key = modelStateKey;
        var errorMessage = error.ErrorMessage;
    }
}

```

Thêm một Custom Error Message

Không phải tất cả các thông báo lỗi đều được đưa ra bởi Model Validator. Ví dụ bạn nhận một Product Model hợp lệ nhưng Product lại tồn tại trong cơ sở dữ liệu. Vậy trường hợp này bạn cần thêm vào một custom error message cho ModelState và gửi nó cho người dùng:

```

if (ModelState.IsValid) {
    If (someService.IsProductExists(model)) {
        ModelState.AddModelError("", "Product already exists");
        return View(model);
    } else {
        return View(model);
    }
}

```

Danh sách Validation Attributes

Chúng ta nhìn 2 attribute **Required** và **StringLength** ở ví dụ trước.

Namespace **System.ComponentModel.DataAnnotations** chứa một số các thuộc tính. Đây là danh sách:

CreditCard

Kiểm tra thuộc tính có định dạng credit card

```

[CreditCard(ErrorMessage = "Please enter a valid card No")]
public string creditCard { get; set; }

```

Compare

So sánh giá trị với giá trị của thuộc tính khác

Ví dụ, áp dụng vào trường hợp cần xác nhận mật khẩu. Tham số vào của attribute này là **NewPassword**. Nếu xác nhận mật khẩu không đúng với **NewPassword** thì quá trình kiểm tra không hợp lệ.

```
[Required(AllowEmptyStrings = false, ErrorMessage = "Please enter a valid password")]
```

```
public string NewPassword { get; set; }
```

```
[Compare(otherProperty: "NewPassword", ErrorMessage = "New & confirm password does not match")]
```

```
public string ConfirmPassword { get; set; }
```

Cách khác là bạn có thể so sánh trong một phương thức get để kiểm tra:

```
[Required(AllowEmptyStrings = false, ErrorMessage = "Please enter a valid password")]
```

```
public string NewPassword { get; set; }
```

```
[Compare(otherProperty: "validatePassword", ErrorMessage = "New & confirm password does not match")]
```

```
public string ConfirmPassword { get; set; }
```

```
public string validatePassword
```

```
{
```

```
    get
```

```
    {
```

```
        //Do Some calculations here
```

```
        return this.NewPassword;
```

```
    }
```

```
}
```

EmailAddress

Kiểm tra thuộc tính có phải định dạng email không

```
[EmailAddress(ErrorMessage = "Please enter a valid email")]
```

```
public string EmailID {get;set;}
```

HTML được tạo ra:

```
<label for="EmailID">EmailID</label>
```

```
<input type="email" data-val="true" data-val-email="Please enter a valid email" id="EmailID" name="EmailID" value="" />

<span class="field-validation-valid" data-valmsg-for="EmailID" data-valmsg-replace="true"></span>

<br />
```

Như bạn thấy HTML, thuộc tính type của nó sẽ tạo ra là email. Trong trường hợp này thì trình duyệt sẽ hiển thị thông báo lỗi nếu có một email không hợp lệ.

Phone

Kiểm tra định dạng số điện thoại

```
[Phone(ErrorMessage = "Please enter a valid Phone No")]

public string PhoneNo { get; set; }

<label asp-for="PhoneNo"></label>

<input asp-for="PhoneNo" />

<span asp-validation-for="PhoneNo"></span>

<label for="PhoneNo">PhoneNo</label>

<input type="tel" data-val="true" data-val-phone="Please enter a valid Phone No" id="PhoneNo" name="PhoneNo" value="" />

<span class="field-validation-valid" data-valmsg-for="PhoneNo" data-valmsg-replace="true"></span>

<br />
```

Range

Cho phép kiểm tra xem giá trị có nằm trong một khoảng cho trước không

```
[Range(minimum:100,maximum:200, ErrorMessage = "Please enter a valid no between 100 & 200")]

public int Range { get; set; }

<label asp-for="Range"></label>

<input asp-for="Range" />

<span asp-validation-for="Range"></span>

<label for="Range">Range</label>

<input type="number" data-val="true" data-val-range="Please enter a valid no between 100 & 200" data-val-range-max="200" data-val-range-min="100" data-val-required="The Range field is required." id="Range" name="Range" value="0" />
```

```
<span class="field-validation-valid" data-valmsg-for="Range" data-valmsg-replace="true"></span>
```

```
<br />
```

RegularExpression

Kiểm tra dữ liệu có khớp với một biểu thức chính quy không

```
[RegularExpression(pattern: "Mr\\.|Mrs\\.|Ms\\.|Miss\\.|", ErrorMessage = "Name must start with Mr./Mrs./Ms./Miss.")]
```

```
public string FullName { get; set; }
```

Required

Yêu cầu một thuộc tính là bắt buộc. Các kiểu không chấp nhận giá trị null (decimal, int, float và DateTime) thường mặc định mà bắt buộc không cần dùng đến attribute này. Ứng dụng sẽ tự động kiểm tra phía server với các kiểu không cho phép null và tự đánh là Required.

StringLength

Kiểm tra xem chuỗi có nằm trong vùng độ dài cho phép

```
[StringLength(MaximumLength: 25, MinimumLength = 10, ErrorMessage = "Length must be between 10 to 25")]
```

```
public string Name { get; set; }
```

Url

Kiểm tra xem định dạng URL chuẩn không

```
[Url(ErrorMessage = "Please enter a valid URL")]
```

```
public string Url { get; set; }
```

```
<label asp-for="Url"></label>
```

```
<input asp-for="Url" />
```

```
<span asp-validation-for="Url"></span>
```

HTML tạo ra là

```
<label for="Url">Url</label>
```

```
<input type="url" data-val="true" data-val-url="Please enter a valid URL" id="Url" name="Url" value="" />
```

```
<span class="field-validation-valid" data-valmsg-for="Url" data-valmsg-replace="true"></span>
```

```
<br />
```

32. Validation Tag Helper trong ASP.NET Core

ASP.NET cung cấp các Tag Helper liên quan đến hiển thị validation message cho người dùng. Chúng ta đã tìm hiểu cách sử dụng server-side model validation trong bài trước. Model Binder sẽ bind và kiểm tra dữ liệu nhận từ HTTP Request. Nó tạo một đối tượng ModelState chứa các danh sách thông báo lỗi được tạo ra bởi Model Binder. Validation Tag Helper sẽ tạo ra các phần tử HTML để hiển thị cho user trên view.

Validation Tag Helpers

ASP.NET Core cung cấp 2 Tag Helper để hiển thị thông báo lỗi trên client:

Validation Message Tag Helper

Validation Summary Tag Helper

Validation Message Tag Helper

Validation Tag Helper hướng đến phần tử `` và sử dụng nó để hiển thị một thông báo lỗi cho thuộc tính cụ thể.

[Required]

```
Public string name {get;set;}
```

```
<label asp-for="Name">Name</label>
```

```
<input asp-for="Name" />
```

```
<span asp-validation-for="Name"></span>
```

```
<br />
```

Đoạn HTML sau được tạo ra. Chú ý là phần tử `` rỗng và class `field-validation-valid` được gán cho nó.

```
<label for="Name">Name</label>
```

```
<input type="text" data-val="true" data-val-required="The Name field is required." id="Name" name="Name" value="" />
```

```
<span class="field-validation-valid" data-valmsg-for="Name" data-valmsg-replace="true"></span>
```

Và khi bạn gửi form với một trường name bị rỗng. Đoạn HTML được tạo ra chú ý là phần tử `` giờ đã có thông tin lỗi. Và class `field-validation-error` được gán cho nó.

```
<label for="Name">Name</label>
```

```
<input type="text" class="input-validation-error" data-val="true" data-val-required="The Name field is required." id="Name" name="Name" value="" />
```

```
<span class="field-validation-error" data-valmsg-for="Name" data-valmsg-replace="true">The Name field is required.</span>
```

Class dưới đây hiển thị màu đỏ cho class lỗi:

```
<style>
    .field-validation-error { color:red }
</style>
```

Validation Summary Tag Helper

Validation summary tag helper hướng đến phần tử `<div>` và dùng nó để hiển thị tất cả các thông tin lỗi trên form ở một nơi. Mỗi một thông báo lỗi được hiển thị với một danh sách không được sắp xếp. Bạn cần chỉ ra 3 giá trị cho `asp-validation-summary` tag helper. **All**, **ModelOnly** hoặc **None**.

All

Tuỳ chọn này hiển thị tất cả các thông báo lỗi

```
<div asp-validation-summary="All"></div>
```

Nó sẽ tạo ra đoạn HTML dưới đây khi không có lỗi nào được hiển thị

```
<div class="validation-summary-valid" data-valmsg-summary="true">
    <ul><li style="display:none"></li></ul>
</div>
```

Và tạo ra HTML khi model lỗi:

```
<div class="validation-summary-errors" data-valmsg-summary="true">
    <ul><li>The Name field is required.</li></ul>
</div>
```

Để hiển thị tiêu đề

```
<div asp-validation-summary="All">
    <span>Please correct the following errors</span>
</div>
```

Và thêm style:

```
.validation-summary-valid span { display: none; }
```

ModelOnly

Tuỳ chọn này hiển thị chỉ các lỗi liên quan đến validation mức Model. Các lỗi ở level thuộc tính sẽ không hiển thị. Sử dụng tuỳ chọn này nếu bạn đang hiển thị các lỗi validation liên quan đến thuộc tính sử dụng `asp-validation-for`.

```
<div asp-validation-summary="ModelOnly"></div>
```

Điều lạ nữa là nó không tạo ra bất cứ HTML nào khác biệt, khi **ModelState** hợp lệ không giống như tùy chọn **All**.

Bạn có thể thêm lỗi mức độ Model trực tiếp trong Model sử dụng

```
ModelState.AddModelError(string.Empty, "Errors occurred in the Model");
```

Đoạn HTML dưới được tạo ra khi lỗi xảy ra:

```
<div class="validation-summary-errors">
    <ul><li>Errors occurred in the Model</li></ul>
</div>
```

None

Tag Helper sẽ không hiển thị bất cứ lỗi nào, cái này giống như không có Tag Helper.

Chú ý là **validation-summary-valid** class được thêm vào khi model hợp lệ (không phải khi tùy chọn **"All"** được chọn) và nó được thay thế bởi **validation-summary-errors** khi model không hợp lệ.

Thêm các style để thay đổi màu chữ:

```
<style>
    .validation-summary-errors { color:red }
</style>
```

Bạn có thể hiển thị tiêu đề của thông báo lỗi

```
<div asp-validation-summary="ModelOnly">
    <span>Please correct the following errors</span>
</div>
```

Tiêu đề được hiển thị chỉ khi lỗi được hiển thị.

33. Unobtrusive Client Validation trong ASP.NET Core

Trong bài viết này chúng ta sẽ thực hiện validation phía client sử dụng Javascript. Chúng ta đã học cách thực hiện server-side validation ở bài Model Validation. Cơ chế Unobtrusive client-side validation trong ASP.NET Core sử dụng cùng các attribute như với Model Validator.

Client-side validation

Hãy xem xét một ví dụ, khi bạn cần kiểm tra trường Name không được trống. Chúng ta dễ dàng làm với việc thêm attribute **Required** vào thuộc tính **Name** trong model.

[Required]

```
Public string Name {get;set;}
```

Model Validator sẽ thực hiện kiểm tra trên server và yêu cầu một request gửi tới server. Nó sẽ không tốt cho trải nghiệm người dùng vì phải tốn một khoảng thời gian chờ trước khi người dùng nhận thông báo lỗi từ server. Nó cũng tốn tài nguyên server cho một kiểm tra đơn giản.

Đây là trường hợp đơn giản nhất của client-side validation.

Client side validation có thể được thực hiện bằng nhiều cách:

Sử dụng các thư viện Javascript như JQuery validation và Javascript Unobtrusive

Sử dụng HTML 5 Validation

Viết Javascript của bạn

Unobtrusive client-side validation

ASP.NET Core bao gồm các thư viện unobtrusive client-side validation, nó giúp dễ dàng thêm các client-side validation mà không cần viết code nhiều. Trong bài trước chúng ta sử dụng server-side validation, chúng ta cũng tìm hiểu cách sử dụng các attribute Annotation và được xử lý bởi Model Validator để kiểm tra Model. Thư viện unobtrusive client-side cũng sử dụng các attribute đó để kiểm tra các thuộc tính trên phía client. Sự khác nhau là nó sử dụng Javascript thay vì C#.

Unobtrusive (ngầm) validation sử dụng `jquery.validate.unobtrusive.js`. Thư viện này được xây dựng dựa trên `jquery.validate.js`, nó sử dụng JQuery. Vì thế chúng ta cần import tất cả nó vào view:

Cách sử dụng unobtrusive client-side validation

Load các thư viện Javascript bắt buộc

Đầu tiên chúng ta cần thêm **JQuery**, **jquery.validate** và **jquery.validate.unobtrusive** vào view. Các script này được thêm vào file Layout (`_Layout.cshtml`) định nghĩa layout của ứng dụng và sử dụng chung cho tất cả các view.

```
<script src="https://ajax.aspnetcdn.com/ajax/jquery/jquery-2.2.0.min.js"></script>
```

```
<script  
src="https://ajax.aspnetcdn.com/ajax/jquery.validate/1.16.0/jquery.validate.min.js"></script>
```

```
<script  
src="https://ajax.aspnetcdn.com/ajax/jquery.validation.unobtrusive/3.2.6/jquery.validate.unobtrusive.min.js"></script>
```

Thêm Validation Attribute vào thuộc tính input

Tiếp theo chúng ta cần thêm Validation Attribute vào thuộc tính model, cái này đã học trong bài trước. Các attribute này chúng ta thêm khi cài đặt server-side validation cũng được dùng luôn cho client-side validation. Trong view nhớ chỉ ra file layout sử dụng bằng cách:

```
@{ Layout = "_Layout"; }
```


Model

```
[Required(AllowEmptyStrings = false, ErrorMessage = "Please enter the name")]
[StringLength(maximumLength: 25, MinimumLength = 10, ErrorMessage = "Length must be
between 10 to 25")]

public string Name { get; set; }
```

View

```
<form asp-controller="home" asp-action="create" method="post">
    <div asp-validation-summary="ModelOnly">
        <span>Please correct the following errors</span>
    </div>
    <label asp-for="Name"></label>
    <input asp-for="Name"/>
    <span asp-validation-for="Name"></span>
    <br />
    <input type="submit" name="submit" />
</form>
```

Controller

```
[HttpPost]

public IActionResult Create(ProductEditModel model, [FromQuery] string test)
{
    string message = "";
    if (ModelState.IsValid){
        message = "product " + model.Name + " created successfully";
    }
    else {
        return View(model);
    }
    return Content(message);
}
```

Chúng ta không sửa bất cứ code server nào, ngoại trừ thêm các thư viện Javascript

Chúng làm việc ra sao?

Các input tag helper tạo ra HTML như sau:

```
<label for="Name">Name</label>

<input type="text" data-val="true" data-val-length="Length must be between 10
to 25" data-val-length-max="25" data-val-length-min="10" data-val-
required="Please enter the name" id="Name" name="Name" value="" />

<span class="field-validation-valid" data-valmsg-for="Name" data-valmsg-
replace="true"></span>

<br />
```

Bạn có thể thấy các thuộc tính được thêm vào với ký tự bắt đầu là **data-***. Các thuộc tính **data-*** là một phần của HTML 5 cho phép chúng ta thêm các thông tin mở rộng cho thẻ HTML.

Thư viện Javascript unobtrusive đọc các thuộc tính **data-val** và thực hiện kiểm tra phía client trên trình duyệt khi người dùng submit form. Các validation này hoàn thành trước khi form được gửi qua HTTP Request. Nếu có validation nào lỗi thì request sẽ không được gửi.

Click **F12** và mở cửa sổ **Chrome Developer console** và xem tab **Network**. Bạn có thể thấy không có request nào được gửi khi click nút **Submit** với các dữ liệu không hợp lệ.

34. Cơ chế Dependency Injection trong ASP.NET Core

Dependency Injection giờ đã trở thành thành phần chính thức mặc định của ASP.NET Core. Nó giúp chúng ta đáp ứng tính chất lỏng lẻo (loosely couple), dễ đọc và bảo trì code. Trong bài viết này chúng ta sẽ học cơ bản về Dependency Injection trong việc xây dựng ứng dụng đơn giản.

Giới thiệu về Dependency Injection trong ASP.NET Core

Dependency Injection (DI) giờ là một phần của ASP.NET Core. Tất cả các service của framework đều được inject khi chúng ta cần. Nhưng trước khi đi xa hơn, chúng ta cần hiểu tại sao cần dependency injection. Hãy xem ví dụ sau đây trong controller action method khi chúng ta muốn lấy danh sách sản phẩm ra từ **ProductService**.

```
public class HomeController : Controller
{
    public IActionResult Index()
    {
        _productService = new ProductService();
        return View(_productService.getAll());
    }
}
```

Action method **Index** có một phụ thuộc đến **ProductService**. Vì thế nó tạo ra một thể hiện và gọi phương thức **GetAll** để lấy danh sách sản phẩm. **ProductService** giờ đã bị gắn chặt (tightly couple) vào phương thức **Index** của **HomeController**.

Nếu chúng ta tạo **BetterProductService** và muốn thay thế **ProductService** cũ thì sao:

```
public IActionResult Index()
{
    _productService = new BetterProductService();

    return View(_productService.getAll());
}
```

Chúng ta cần làm điều đó với tất cả các Controller, service có sử dụng **ProductService**. Nếu chúng ta muốn sử dụng **TestProductService** và chỉ muốn dùng nó cho mục đích test còn với môi trường Production thì sẽ là **ProductService**. Vậy là không dễ dàng.

Hãy xem xét trường hợp khi **ProductService** phụ thuộc vào service khác, sau đó service đó lại phụ thuộc service khác nữa. Nó không thường xuyên nhưng việc một chuỗi các phụ thuộc như thế chắc chắn có tồn tại trong thực tế.

Dependency Injection sẽ giải quyết tất cả các vấn đề này.

Dependency Injection là gì?

Dependency Injection (được biết là DI) là một design pattern khi một đối tượng không được tạo trong các thành phần phụ thuộc vào nó mà yêu cầu nó. Hãy thay đổi **HomeController** một chút:

```
public class HomeController : Controller
{
    private IProductService _productService;

    public HomeController(IProductService productService)
    {
        _productService = productService;
    }

    public IActionResult Index()
    {
        _productService = new ProductService();

        return View(_productService.All());
    }
}
```

Sự khác nhau giữa đoạn code trên và đoạn code này là chúng ta không tạo ra thể hiện của **ProductService** trong Index action method. Chúng ta yêu cầu nó trong constructor của **HomeController**. Vấn đề đã được giải quyết chưa? Ai đó đã tạo thể hiện của **ProductService** và gán nó vào **HomeController**?

Đây là điểm mà ASP.NET Core Dependency Injection framework làm nhiệm vụ của nó. Trách nhiệm của nó là tạo ra thể hiện của ProductService và đối tượng này được gọi là **DI Container** hay **IoC Container**.

Dependency Injection là một design pattern. Dependency injection framework triển khai design pattern này. Có nhiều framework như Autofac, Unity...bạn có thể sử dụng trong ASP.NET Core.

DI Container

DI Container là một đối tượng có trách nhiệm tạo các phụ thuộc (**ProductService**) và gán nó cho đối tượng yêu cầu (**HomeController**) nó.

Làm thế nào DI Container biết đối tượng nào được tạo?

Chúng ta cần cấu hình cho DI Container là class nào bạn muốn tạo. Chúng ta cần đặt trong class **Startup** với phương thức **ConfigureServices**.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc();

    services.AddTransient<IProductService, ProductService>();
}
```

Dòng thứ 2 sẽ đăng ký **ProductService** với service collection sử dụng phương thức **AddTransient**. Có 2 phương thức khác là **AddSingleton** và **AddScoped**. Có 3 phương thức định nghĩa vòng đời của service mà chúng ta sẽ thảo luận trong bài sau.

Ví dụ về hệ thống Dependency Injection

Tạo một project ASP.NET Core sử dụng empty template và đặt tên nó là **DependencyInjection**. Thêm HomeController với Index method. Bạn có thể tham khảo các bài viết trước.

Tạo ViewModel

Tạo một folder **Models** và tạo ViewModel như sau:

```
public class ProductViewModel
{
    public int Id { get; set; }
    public string Name { get; internal set; }
}
```

Thêm mới service

Tạo một thư mục **Services** và thêm mới một class với tên **ProductService.cs**

```
using DependencyInjection.Models;
using System.Collections.Generic;
namespace DependencyInjection.Service
{
    public interface IProductService
    {
        List<ProductViewModel> getAll();
    }
}
```

```

public class ProductService : IProductService
{
    public List<ProductViewModel> getAll()
    {
        return new List<ProductViewModel>
        {
            new ProductViewModel {Id = 1, Name = "Pen Drive" },
            new ProductViewModel {Id = 2, Name = "Memory Card" },
            new ProductViewModel {Id = 3, Name = "Mobile Phone" },
            new ProductViewModel {Id = 4, Name = "Tablet" },
            new ProductViewModel {Id = 5, Name = "Desktop PC" } ,
        };
    }
}

```

Đầu tiên chúng ta thêm interface **IProductService** và thêm **ProductService** để triển khai interface này. **ProductService** sẽ trả về một danh sách sản phẩm. Danh sách sản phẩm được fix cứng trong code. Trong thực tế thì nó sẽ lấy từ database.

Sử dụng Service trong Controller

Giờ hãy mở **HomeController**

```

using DependencyInjection.Service;
using Microsoft.AspNetCore.Mvc;
namespace DependencyInjection.Controllers
{
    public class HomeController : Controller
    {
        private IProductService _productService;

        public HomeController(IProductService productService)
        {
            _productService = productService;
        }

        public IActionResult Index() {

```

```

        return View(_productService.getAll());
    }
}
}

```

Constructor của **HomeController** yêu cầu một thể hiện của **ProductService** và lưu trữ nó trong biến local tên là **_productService**. Phương thức **Index** gọi view với danh sách sản phẩm được lấy ra từ phương thức **GetAll** của **ProductService**.

View

View chỉ hiển thị danh sách sản phẩm

```

@model List<DependencyInjection.Models.ProductViewModel>;

@{
    ViewData["Title"] = "Index";
}

<h2>Index</h2>

@foreach (var product in Model)
{
    <p>@product.Id @product.Name</p>
}

```

Đăng ký service

Bước cuối cùng là đăng ký service với Dependency Injection container. Mở **Startup.cs** và đến phương thức **ConfigureServices**. Nơi mà tất cả các service được cấu hình cho DI.

```

public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc();

    services.AddTransient<IProductService, ProductService>();
}

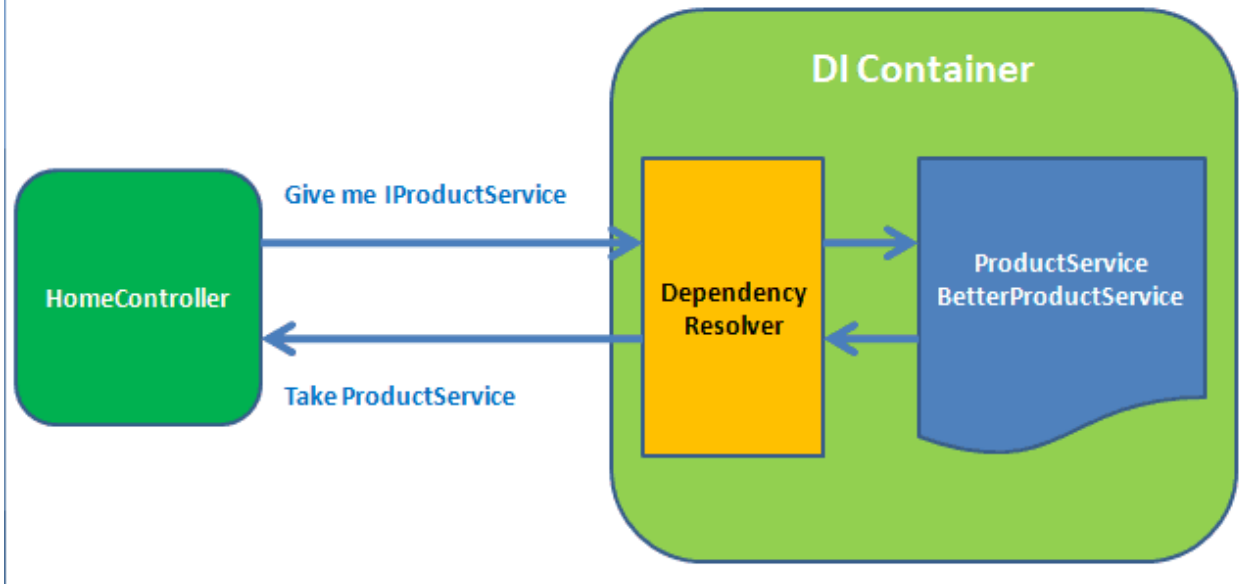
```

Giờ chúng ta sẽ đăng ký **ProductService** sử dụng phương thức **AddTransient**. Có 2 phương thức khác là **AddScoped** và **AddSingleton**. Chúng ta sẽ tìm hiểu sau nhé. Giờ hãy chạy ứng dụng và nhìn kết quả.

Các thành phần phụ thuộc được inject như thế nào?

Hình dưới đây mô tả cách mà **ProductService** được inject vào **HomeController**

Dependency Injection Framework



Khi một **HomeController** mới được yêu cầu, MVC sẽ yêu cầu DI framework cung cấp một thể hiện của **HomeController** class. DI Container sẽ xem constructor của **HomeController** và xác định xem nó có những thành phần phụ thuộc nào (dependencies). Nó sẽ tìm các thành phần phụ thuộc trong danh sách được đăng ký của service collection và tìm service nào thoả mãn sau đó tạo thể hiện cho nó. Sau khi tạo **HomeController** và gán thể hiện của dependencies đó cho constructor.

Tạo BetterProductService

Giờ chúng ta sẽ tạo mới service khác là **BetterProductService** và muốn sử dụng nó thay vì **ProductService**

```
public class BetterProductService : IProductService
```

```
{
```

```
    public List<ProductViewModel> getAll()
```

```
    {
```

```
        return new List<ProductViewModel>
```

```
        {
```

```
            new ProductViewModel {Id = 1, Name = "Television" },
```

```
            new ProductViewModel {Id = 2, Name = "Refrigerator" },
```

```
            new ProductViewModel {Id = 3, Name = "IPhone" },
```

```
            new ProductViewModel {Id = 4, Name = "Laptop" },
```

```
};

}

}
```

Tất cả bạn cần làm là vào **ConfigureServices** trong Startup thay đổi **ProductService** thành **BetterProductService**.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc();

    services.AddTransient<IProductService, BetterProductService>();
}
```

Bạn không phải vào mỗi controller hay service để thay đổi. Bạn có thể quản lý nó ở một nơi duy nhất. Tương tự bạn có thể làm với **TestProductService** sử dụng chúng để test.

Thay đổi Service dựa trên environment

Ví dụ, bạn có thể sử dụng môi trường và chuyển đổi giữa các service Trong constructor của **Startup** class yêu cầu **IHostingEnvironment** service:

```
IHostingEnvironment _env;

public Startup(IHostingEnvironment env)
{
    _env = env;
}
```

Tiếp theo trong **ConfigureServices** chúng ta cấu hình code với **BetterProductService** trong **Production** và **ProductService** cho các môi trường khác.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc();

    if (_env.IsProduction())
    {
        services.AddTransient<IProductService, BetterProductService>();
    }
    else
```



```

{
    services.AddTransient<IProductService, ProductService>();
}
}

```

Thay đổi môi trường chọn **Project**, chuột phải và chọn **Properties**. Chọn tab **Debug**. Thay đổi **ASPNETCORE_ENVIRONMENT** thành môi trường gì mà bạn muốn.

Constructor Injection và Action Injection

Bạn có thể inject service vào Controller theo 2 cách:

Constructor Injection

Action Injection

Constructor Injection

Khi các thành phần được inject vào thông qua constructor sau đó cách này được gọi là constructor injection

```

public class HomeController : Controller
{
    private IProductService _productService;

    public HomeController(IProductService productService)
    {
        _productService = productService;
    }

    public IActionResult Index()
    {
        return View(_productService.All());
    }
}

```

Action Injection

Nếu các dependencies được inject qua một action method thì cách này gọi là Action Injection. Action Injection được thực hiện sử dụng **[FromServices]** attribute vào một tham số của action method. Sử dụng method này nếu service chỉ được sử dụng trong một action method thôi.

```

public class HomeController : Controller {

```

```

public HomeController() { }

public IActionResult Index( [FromServices] IProductService
productService)

{

return View(_productService.All());

}

}

```

Lợi ích của Dependency Injection

Dependency injection giúp thực hiện kiến trúc lỏng lẻo (loose coupling) trong phần mềm.

Code sẽ sạch và dễ đọc hơn

Tăng khả năng có thể kiểm thử và bảo trì

Cho phép bạn thay đổi triển khai mà không phải thay đổi quá nhiều code.

35. Vòng đời của Dependency Injection: Transient, Singleton và Scoped

Hiểu về vòng đời của các service được tạo sử dụng Dependency Injection là rất quan trọng trước khi sử dụng chúng. Tạo các service mà không hiểu về sự khác nhau giữa **Transient**, **Singleton** và **Scoped** có thể làm hệ thống hoạt động không như mong muốn. Khi một service yêu cầu service khác thông qua DI, chúng ta biết rằng nó nhận một thể hiện mới hay một thể hiện đã tồn tại rất quan trọng. Vì thế việc nhận thức đúng vòng đời hay thời gian sống (lifetime) trong khi đăng ký service là việc rất quan trọng.

Quản lý vòng đời (lifetime) Service

Bất cứ khi nào chúng ta yêu cầu service, DI Container sẽ quyết định xem có tạo mới một thể hiện (instance) hay sử dụng lại thể hiện đã tạo từ trước đó. Vòng đời của Service phụ thuộc vào khi khởi tạo thể hiện và nó tồn tại bao lâu. Chúng ta định nghĩa vòng đời khi đăng ký service.

Chúng ta đã học cách sử dụng DI ở bài trước. Có 3 mức độ vòng đời, bằng cách này chúng ta có thể quyết định mỗi service có vòng đời ra sao.

Transient: Một thể hiện mới luôn được tạo, mỗi khi được yêu cầu.

Scoped: Tạo một thể hiện mới cho tất cả các scope (Mỗi request là một scope). Trong scope thì service được dùng lại

Singleton: Service được tạo chỉ một lần duy nhất.

Project ví dụ

Hãy tạo một project ví dụ để demo vòng đời của Service. Tạo một ứng dụng ASP.NET Core sử dụng **Empty Template** và gọi nó là **DependencyInjection**.

Thêm **HomeController** với **Index** method. Bạn có thể theo bài viết trước.

Tạo mới Service Interface

Tạo mới service tên là **SomeService** trong thư mục **Services**. Thêm mới 3 interface. Mỗi interface tương ứng với một vòng đời trong ASP.NET Core. Interface thật đơn giản chúng chứa mỗi phương thức **GetID** trả về một Guid.

```
public interface ITransientService
```

```
{ Guid GetID(); }
```

```
public interface IScopedService
```

```
{ Guid GetID(); }
```

```
public interface ISingletonService
```

```
{ Guid GetID(); }
```

Tạo một Service

Giờ chúng ta sẽ tạo một service đơn lẻ, triển khai cả 3 interface kia:

```
public class SomeService : ITransientService, IScopedService, ISingletonService
```

```
{ Guid id;
```

```
    public SomeService() {id = Guid.NewGuid(); }
```

```
    public Guid GetID() {return id; }
```

```
}
```

Service tạo ra một ID duy nhất và nó sẽ được khởi tạo và trả về id trong phương thức **GetID**. Giờ hãy xem chi tiết

Transient

Transient Service luôn tạo mới mỗi lần service được yêu cầu

Đăng ký Transient Service

Giờ bên trong phương thức **ConfigureServices** của **Startup** sẽ đăng ký **SomeService** thông qua **ITransientService** interface.

```
services.AddTransient<ITransientService, SomeService>();
```

Inject vào Controller

Mở **HomeController** ra và inject 2 thể hiện của **SomeService** như dưới đây:

```
public class HomeController : Controller
```

```
{
```

```
    ITransientService _transientService1;
```

```
    ITransientService _transientService2;
```

```

public HomeController(ITransientService transientService1,
                    ITransientService transientService2)
{
    _transientService1 = transientService1;
    _transientService2 = transientService2;
}

public IActionResult Index()
{
    ViewBag.message1 = "First Instance " +
    _transientService1.GetID().ToString();

    ViewBag.message2 = "Second Instance " +
    _transientService2.GetID().ToString();

    return View();
}
}

```

Đầu tiên, chúng ta inject 2 thể hiện của service thông qua interface **ITransientService** trong constructor của **HomeController**.

```

public HomeController(ITransientService transientService1, ITransientService
transientService2)

```

Tiếp theo, chúng ta gọi phương thức **GetID** trong mỗi thể hiện và gán nó vào view sử dụng **ViewBag**.

```

ViewBag.message1 = "First Instance " + _transientService1.GetID().ToString();
ViewBag.message2 = "Second Instance " + _transientService2.GetID().ToString();

```

View

Mở view và thêm code:

```

<h3>Transient Service</h3>

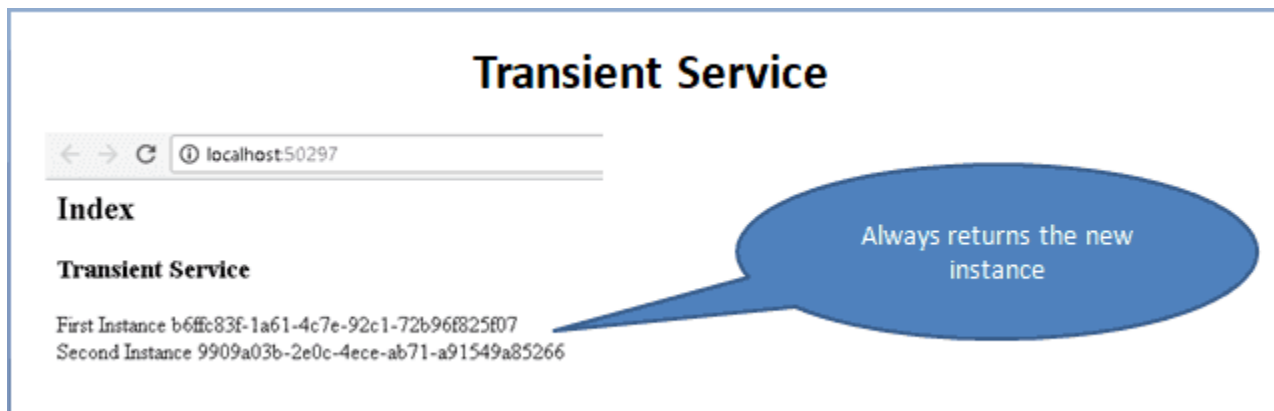
@ViewBag.message1

</br>

@ViewBag.message2

```

Chạy ứng dụng và bạn sẽ nhìn thấy 2 Guid khác nhau hiển thị trên màn hình. Bằng chứng là chúng ta đã nhận 2 thể hiện của Transient service.



Scoped

Các service với vòng đời Scoped được tạo chỉ một lần cho mỗi request (scope). Một thể hiện mới được tạo cho mỗi request và thể hiện được dùng lại trong request.

Đăng ký Scoped Service

Trong phương thức **ConfigureServices** đăng ký **SomeService** sử dụng phương thức **AddScoped** sử dụng **IScopedService** interface.

```
services.AddScoped<IScopedService, SomeService>();
```

Inject scoped service vào Controller

Tiếp theo inject service này vào Controller. Chúng ta đã có transient service được inject vào controller. Giờ không thay đổi gì nhưng thêm mới:

```
public class HomeController : Controller
{
    ITransientService _transientService1;
    ITransientService _transientService2;
    IScopedService _scopedService1;
    IScopedService _scopedService2;

    public HomeController(ITransientService transientService1,
                           ITransientService transientService2,
                           IScopedService scopedService1,
                           IScopedService scopedService2)
    {
        _transientService1 = transientService1;
        _transientService2 = transientService2;
        _scopedService1 = scopedService1;
    }
}
```

```

        _scopedService2 = scopedService2;
    }

    public IActionResult Index()
    {
        ViewBag.message1 = "First Instance " +
            _transientService1.GetID().ToString();

        ViewBag.message2 = "Second Instance " +
            _transientService2.GetID().ToString();

        ViewBag.message3 = "First Instance " + _scopedService1.GetID().ToString();
        ViewBag.message4 = "Second Instance " + _scopedService2.GetID().ToString();

        return View();
    }
}

```

Thêm vào action method 2 biến **message3** và **message4** cho scoped service.

View

Trong view thêm 3 dòng

```
<h3>Scoped Service</h3>
```

```
@ViewBag.message3
```

```
</br>
```

```
@ViewBag.message4
```

Chạy ứng dụng.

Thẻ hiển được tạo chỉ một lần cho mỗi request, đó là lý do chúng ta tạo ra 2 ID giống nhau. Giờ hãy refresh trình duyệt. ID đã thay đổi bởi vì một thẻ hiển mới được tạo ra cho mỗi request.

Scoped Service

The screenshot shows a web browser at `localhost:50297` displaying the following content:

```

Index

Transient Service

First Instance 2484a79a-9613-4fd0-b0ac-abfcc614b506
Second Instance 3843f43c-60c2-461b-bb63-f95c5b48201e

Scoped Service

First Instance 849c7453-6986-4785-ad09-84ad40f3a922
Second Instance 849c7453-6986-4785-ad09-84ad40f3a922
  
```

Callouts explain the behavior:

- Always returns the new instance**: Points to the Transient Service instances, which have different IDs.
- Instance is created only once per request and shared across the request i.e. why you have same Ids generated**: Points to the Scoped Service instances, which have the same ID.
- Now hit the browser refresh button. The id changes. i.e. because for every request new instance is generated**: Points to the Scoped Service instances, indicating that the ID changes upon a browser refresh.

Singleton

Một thể hiện duy nhất của service được tạo khi nó được yêu cầu lần đầu. Sau đó mỗi một request sau này sẽ chỉ sử dụng cùng instance đó thôi. Request mới không tạo thể hiện mới mà nó được dùng lại.

Đăng ký Singleton Service

Singleton service được đăng ký sử dụng phương thức **AddSingleton**

```
services.AddSingleton<ISomeServiceService, SomeService>();
```

Inject Singleton service vào Controller

```

public class HomeController : Controller
{
    ITransientService _transientService1;
    ITransientService _transientService2;
    IScopedService _scopedService1;
    IScopedService _scopedService2;
    ISingletonService _singletonService1;
    ISingletonService _singletonService2;
  
```

```

public HomeController(ITransientService transientService1,
                    ITransientService transientService2,
                    IScopedService scopedService1,
                    IScopedService scopedService2,
                    ISingletonService singletonService1,
                    ISingletonService singletonService2)
{
    _transientService1 = transientService1;
    _transientService2 = transientService2;
    _scopedService1 = scopedService1;
    _scopedService2 = scopedService2;
    _singletonService1 = singletonService1;
    _singletonService2 = singletonService2;
}

public IActionResult Index()
{
    ViewBag.message1 = "First Instance " +
    _transientService1.GetID().ToString();

    ViewBag.message2 = "Second Instance " +
    _transientService2.GetID().ToString();

    ViewBag.message3 = "First Instance " + _scopedService1.GetID().ToString();

    ViewBag.message4 = "Second Instance " +
    _scopedService2.GetID().ToString();

    ViewBag.message5 = "First Instance " +
    _singletonService1.GetID().ToString();

    ViewBag.message6 = "Second Instance " +
    _singletonService2.GetID().ToString();

    return View();
}
}

```

Đầu tiên chúng ta inject 6 thể hiện của **SomeService**. Hai thể hiện cho mỗi interface.

View


```
@{ ViewData["Title"] = "Index"; }
```

```
<h2>Index</h2>
```

```
<h3>Transient Service</h3>
```

```
@ViewBag.message1
```

```
</br>
```

```
@ViewBag.message2
```

```
<h3>Scoped Service</h3>
```

```
@ViewBag.message3
```

```
</br>
```

```
@ViewBag.message4
```

```
<h3>Singleton Service</h3>
```

```
@ViewBag.message5
```

```
</br>
```

```
@ViewBag.message6
```

Chạy ứng dụng và bạn thấy rằng các ID tạo ra từ Singleton service luôn giống nhau và sẽ không thay đổi ngay cả khi bạn refresh ứng dụng. Bạn có thể thấy trong hình dưới đây

Singleton Service

The screenshot shows a web browser at localhost:50297 displaying an 'Index' page. The page lists three service types: Transient Service, Scoped Service, and Singleton Service, each with two instance IDs. Callouts explain the behavior of each:

- Transient Service:** First Instance dd3f328c-3173-4479-bc53-bea6fef9a0c4, Second Instance 101d57c8-e449-4029-b07a-1a2ba38e99c4. Callout: "Always returns the new instance".
- Scoped Service:** First Instance 027a26e3-c5c3-4f76-bfb7-d1891ef76f9f, Second Instance 027a26e3-c5c3-4f76-bfb7-d1891ef76f9f. Callout: "Instance is created only once per request and shared across the request I.e. why you have same Ids generated. New ids are generated, when you click on refresh button".
- Singleton Service:** First Instance c347fe45-8674-4c87-b1af-ddb199924369, Second Instance c347fe45-8674-4c87-b1af-ddb199924369. Callout: "Only one instance is created and shared across the application. Click on Refresh button, the ids will remain the same".

Vậy nên sử dụng cái nào?

Transient service là cách an toàn nhất để tạo, vì bạn luôn tạo mới một thể hiện. Nhưng vì thế mà nó sẽ tạo mỗi lần bạn yêu cầu như vậy sẽ dùng nhiều bộ nhớ và tài nguyên hơn. Điều này có thể ảnh hưởng không tốt đến hiệu năng nếu quá nhiều thể hiện được tạo.

Sử dụng Transient Service phù hợp khi bạn muốn dùng cho các service nhẹ và nhỏ cũng như không có trạng thái.

Scoped service thì tốt hơn khi bạn muốn duy trì trạng thái trong một request.

Singleton được tạo chỉ một lần, nó không bị hủy cho đến khi ứng dụng tắt. Bất cứ việc chiếm bộ nhớ nào với các service này đều tích lại theo thời gian và nó đầy lên. Nhưng cũng giúp chúng ta tiết kiệm bộ nhớ nếu xử lý tốt vì chúng được tạo chỉ 1 lần và sử dụng mọi nơi.

Sử dụng Singleton khi bạn cần duy trì trạng thái trong toàn hệ thống. Cấu hình, tham số của ứng dụng, các service logging, caching dữ liệu...là các ví dụ thường dùng Singleton.

Inject service với các vòng đời khác nhau vào service khác

Hãy cẩn thận, khi inject service vào service khác với các vòng đời khác nhau. Hãy xem ví dụ Singleton Service phụ thuộc vào một service khác được đăng ký với vòng đời là Transient.

Khi request đến lần đầu thì một instance của Singleton được tạo.

Khi request thứ 2 đến thì thể hiện này của Singleton được dùng lại. Và singleton này đã chứa một thể hiện của transient service. Vì thế nó không được tạo lại. Điều này vô hình chung đã chuyển transient service thành singleton service.

Các service với mức độ vòng đời thấp hơn được inject vào service có vòng đời cao hơn sẽ thay đổi service vòng đời thấp hơn thành vòng đời cao hơn. Điều này sẽ làm việc debug khó hơn và nó nên tránh. Từ thấp đến cao là **Transient**, **Scoped** và **Singleton**.

Vì thế hãy nhớ quy tắc:

Không bao giờ inject Scoped & Transient service vào Singleton service

Không bao giờ inject Transient Service vào Scope Service