# LỜI CẢM ƠN

# MỤC LỤC

# DANH MỤC ẢNH

Ảnh 1 12

# DANH MỤC VIẾT TẮT

|  |  |  |  |
| --- | --- | --- | --- |
| STT | Phần viết tắt | Phần viết đầy đủ | Giải nghĩa |
| 01 | MVC | Model-View-Controler |  |
| 02 | API |  |  |
|  | DI | Dependency Injection |  |
|  | HTTP |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

# Chương I. GIỚI THIỆU

## 1.1. GIỚI THIỆU ASP.NET Core

ASP.NET Core là một framework mã nguồn mở được phát triển bởi Microsoft, dùng để xây dựng các ứng dụng web, dịch vụ web và API. Đây là phiên bản tái cấu trúc của ASP.NET, với nhiều cải tiến về hiệu suất, khả năng mở rộng và tính linh hoạt.

**Các đặc điểm nổi bật:**

* **Đa nền tảng:** ASP.NET Core có thể chạy trên Windows, macOS và Linux, cho phép các nhà phát triển xây dựng ứng dụng mà không bị ràng buộc vào một hệ điều hành cụ thể.
* **Hiệu suất cao:** Với kiến trúc nhẹ và tối ưu hóa cho môi trường đám mây, ASP.NET Core mang lại hiệu suất xử lý tốt hơn so với các phiên bản trước.
* **Thiết kế mô-đun:** Các thành phần trong ASP.NET Core được thiết kế theo kiểu mô-đun, cho phép người dùng chọn lựa và thêm vào các thư viện cần thiết, giúp giảm kích thước ứng dụng và tăng tính hiệu quả.
* **Kiến trúc MVC:** ASP.NET Core hỗ trợ mô hình kiến trúc Model-View-Controller (MVC), giúp phân tách các thành phần trong ứng dụng, làm cho mã nguồn dễ quản lý và bảo trì hơn. MVC cho phép phát triển các ứng dụng web tương tác và dễ dàng mở rộng.
* **Hỗ trợ Dependency Injection:** ASP.NET Core tích hợp sẵn hỗ trợ Dependency Injection, giúp quản lý các phụ thuộc giữa các thành phần của ứng dụng một cách hiệu quả, nâng cao tính linh hoạt và khả năng kiểm thử.
* **Tích hợp với các công nghệ hiện đại:** Framework này hỗ trợ tích hợp với nhiều công nghệ hiện đại như Entity Framework Core cho việc truy xuất dữ liệu, SignalR cho ứng dụng thời gian thực, và Razor Pages để phát triển giao diện người dùng nhanh chóng.
* **Hỗ trợ tốt cho bảo mật:** ASP.NET Core cung cấp nhiều tính năng bảo mật mạnh mẽ như Xác thực (Authentication), Ủy quyền (Authorization), bảo vệ CSRF (Cross-Site Request Forgery) và XSS (Cross-Site Scripting). Điều này đảm bảo ứng dụng được bảo mật và an toàn hơn khi triển khai.
* **Hỗ trợ kiểm thử:** ASP.NET Core dễ dàng hỗ trợ việc kiểm thử ứng dụng (Unit Test và Integration Test), nhờ vào tính năng Dependency Injection, giúp tách biệt các lớp và kiểm soát được các phụ thuộc trong quá trình kiểm thử.
* **Cộng đồng và tài liệu phong phú:** ASP.NET Core được hỗ trợ bởi một cộng đồng lớn và tích cực, cung cấp nhiều tài liệu, hướng dẫn và thư viện mã nguồn mở, giúp các lập trình viên dễ dàng tìm hiểu và áp dụng.

**Ứng dụng thực tiễn:** ASP.NET Core được sử dụng rộng rãi trong việc phát triển các ứng dụng web doanh nghiệp, trang thương mại điện tử, API cho ứng dụng di động và nhiều loại ứng dụng khác, nhờ vào tính linh hoạt và hiệu suất của nó.

ASP.NET Core là một framework mạnh mẽ, linh hoạt và hiện đại, giúp các nhà phát triển xây dựng các ứng dụng web, dịch vụ API và ứng dụng đa nền tảng với hiệu suất cao. Kiến trúc nhẹ, mở rộng dễ dàng, và sự hỗ trợ từ cộng đồng là những lợi thế lớn mà nó mang lại.

## 1.2. LÝ DO CHỌN ĐỀ TÀI

Trong bối cảnh công nghệ ngày càng phát triển, việc xây dựng các ứng dụng web nhanh chóng, hiệu quả và linh hoạt là một yêu cầu thiết yếu. ASP.NET Core nổi bật như một framework đáp ứng tốt những yêu cầu này, giúp nhà phát triển tạo ra các ứng dụng chất lượng cao.

**ASP.NET Core được chọn làm đề tài vì có những đặc điểm nổi trội như sau:**

* **Đa nền tảng:** ASP.NET Core cho phép phát triển ứng dụng trên nhiều hệ điều hành khác nhau như Windows, macOS và Linux. Điều này không chỉ mang lại tính linh hoạt mà còn mở rộng cơ hội cho các nhà phát triển trong việc triển khai ứng dụng.
* **Hiệu suất và khả năng mở rộng:** Với kiến trúc tối ưu hóa và khả năng xử lý cao, ASP.NET Core là lựa chọn lý tưởng cho các ứng dụng yêu cầu hiệu suất tốt và khả năng mở rộng dễ dàng. Việc nghiên cứu framework này sẽ cung cấp những kiến thức cần thiết để xây dựng các ứng dụng có quy mô lớn.
* **Hỗ trợ công nghệ hiện đại:** ASP.NET Core tích hợp dễ dàng với các công nghệ và công cụ hiện đại như Entity Framework Core, SignalR, và các dịch vụ đám mây. Điều này giúp nhà phát triển dễ dàng triển khai các giải pháp phong phú và đa dạng, từ đó nâng cao chất lượng ứng dụng.
* **Tài liệu và cộng đồng phong phú:** Framework này được hỗ trợ bởi một cộng đồng lớn và rất nhiều tài liệu hướng dẫn, giúp các lập trình viên dễ dàng tiếp cận và học hỏi. Việc chọn đề tài này không chỉ mang lại kiến thức về kỹ thuật mà còn tạo cơ hội kết nối với cộng đồng phát triển.

Ngoài ra việc nghiên cứu và thực hành với ASP.NET Core sẽ giúp cải thiện kỹ năng lập trình, hiểu rõ hơn về kiến trúc ứng dụng web, và nâng cao khả năng giải quyết vấn đề trong phát triển phần mềm.

# Chương II. TỔNG QUAN VỀ ASP.NET CORE

## 2.1. ASP.NET Core là gì?

### 2.1.1. ASP.NET Core là gì?

ASP.NET Core là một tập hợp các thư viện chuẩn như một framework mã nguồn mở đa nền tảng *(cross-platform)* cho việc xây dựng những ứng dụng hiện tại dựa trên kết nối đám mây, giống như web apps, IoT và backend cho mobile.

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.

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 bao gồm các thành phần theo hướng module nhằm tối thiểu tài nguyên và chi phí phát triển.

### 2.1.2. .NET Core là gì?

.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.

### 2.1.3. .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.

### 2.1.4. 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 Framwork. Các tính năng như WebForms, Windows Forms, WPF chưa được đưa lên .NET Core. .NET Framwork chỉ chạy trên Windows trong khi .NET Core có thể chạy trên bất cứ nền tảng nào.

## 2.2. Lịch sử phát triển của ASP.NET Core

Phiên bản đầu tiên của ASP.NET được gọi là ASP.NET 1.0. Chương trình đã được giới thiệu cùng với . NET Framework 1.0 vào năm 2002. Kể từ đó, Microsoft đã liên tục cải tiến, phát triển ASP.NET.

ASP.NET Core có một số thay đổi kiến trúc lớn, đó là kết quả của việc học hỏi rất nhiều từ các framework module hóa khác. ASP.NET Core không còn dựa trên System.Web.dll nữa. Nó được dựa trên một tập hợp các gói, các module hay cũng được gọi là các Nuget packages.

**Những cải tiến nền tảng của ASP.NET Core:**

* Hợp nhất việc xây dựng web UI và web APIs
* Tích hợp những client-side frameworks hiện đại và những luồng phát triển
* Hệ thống cấu hình dựa trên môi trường đám mây thật sự
* Dependency injection được xây dựng sẵn
* HTTP request được tối ưu nhẹ hơn
* Có thể host trên IIS hoặc self-host trong process của riêng bạn
* Được xây dựng trên .NET Core, hỗ trợ thực sự app versioning
* Chuyển các thực thể, thành phần, module như những NuGet packages
* Những công cụ mới để đơn giản hóa quá trình phát triển web hiện đại
* Xây dựng và chạy đa nền tảng (Windows, Mac và Linux)
* Mã nguồn mở và tập trung vào cộng đồng

ASP.NET đã trở thành nền tảng công nghệ phát triển web phổ biến trong việc xây dựng ứng dụng web mạnh mẽ, tăng độ bảo mật và hiệu suất hoạt động trên Windows. ASP.NET đã được sử dụng trong nhiều dự án phát triển web quan trọng. Từ đó hình thành cộng đồng phát triển khổng lồ trong lĩnh vực công nghệ.

## 2.3. Các thành phần chính của ASP.NET

**Cấu trúc ASP.NET:**

* **Page (.aspx):** Đây là đơn vị cơ bản của một trang web ASP.NET. Mỗi trang có thể chứa mã HTML, CSS và mã lập trình để thể hiện nội dung và logic của trang.
* **Code-behind (.aspx.cs hoặc .aspx.vb):** Đây là file code phía sau của mỗi trang ASP.NET. Tiện ích chứa mã lập trình phục vụ cho trang tương ứng và được sử dụng để xử lý sự kiện và thao tác với dữ liệu.
* **Web.config:** Đây là file cấu hình chính cho ứng dụng ASP.NET. Công nghệ chứa thông tin về các cài đặt như chuỗi kết nối CSDL, cấu hình bảo mật và các thiết lập khác cho ứng dụng.

**Các thành phần của ASP.NET:**

* **Controls:** ASP.NET cung cấp một loạt các điều khiển web như textbox, Button, gridview, và nhiều hơn nữa. Các điều kiện này giúp xây dựng giao diện người dùng tương tác và thể hiện các dữ liệu.
* **Code-behind:** Mã lập trình phía sau mỗi trang được sử dụng để xử lý sự kiện, truy xuất và thực hiện các tác vụ logic phức tạp.
* **Data Access:** ASP.NET hỗ trợ nhiều phương pháp truy cập dữ liệu như ADO.NET, Entity Framework, LINQ để truy vấn và cập nhật cơ sở dữ liệu.
* **State Management:** ASP.NET cung cấp các cơ chế quản lý trạng thái như Viewstate, Session và Cookie để lưu trữ thông tin tạm thời hoặc duy trì trạng thái giữa các yêu cầu.
* **Authentication và Authorization:** ASP.NET cung cấp các cơ chế xác thực và phân quyền, cho phép ứng dụng kiểm tra danh tính người dùng và kiểm soát quyền truy cập vào các tài nguyên.
* **Caching:** ASP.NET hỗ trợ caching để tối ưu hóa hiệu suất ứng dụng bằng cách lưu trữ tạm thời dữ liệu hoặc mã được sử dụng thường xuyên để tránh thực hiện lại các tác vụ tốn kém.

## 2.4. So sánh ASP.NET với PHP

ASP.NET và PHP là hai ngôn ngữ lập trình phổ biến được sử dụng trong việc phát triển các ứng dụng web.

**Điểm giống nhau:**

* **Hoạt động đa nền tảng:** Cả ASP.NET và PHP đều có khả năng chạy trên đa nền tảng, bao gồm Windows, Linux và MacOS.
* **Hỗ trợ cộng đồng lớn:** Cả hai đều có cộng đồng phát triển lớn với nhiều tài liệu, thư viện và tài nguyên hỗ trợ đầy đủ.
* **Tích hợp với cơ sở dữ liệu:** ASP.NET và PHP đều mang đến nhiều cơ sở dữ liệu như Mysql, SQL Server và Oracle để lưu trữ và truy xuất dữ liệu.
* **Cung cấp các framework và thư viện mạnh mẽ:** Các tiện ích đều cung cấp các framework và thư viện giúp xây dựng ứng dụng web nhanh chóng và dễ dàng.

**Điểm khác nhau:**

* **Ngôn ngữ lập trình:** ASP.NET sử dụng C#, VB.NET và F# làm ngôn ngữ chính. Trong khi PHP được thiết kế đặc biệt cho lập trình web.
* **Độ phổ biến:** PHP được coi là ngôn ngữ lập trình web phổ biến nhất trên thế giới, ASP.NET thường được sử dụng trong các dự án và hệ thống phát triển bởi Microsoft.
* **Cấu trúc và phương pháp phát triển:** ASP.NET thường sử dụng cấu trúc MVC (Model-View-Controller) hoặc mô hình Web Forms. PHP có tích hợp linh hoạt và khả năng phát triển các ứng dụng web theo nhiều cách khác nhau.
* **Hiệu suất:** Trong một số trường hợp, ASP.NET có thể có hiệu suất tốt hơn khi xử lý các ứng dụng web lớn và tải trọng cao. PHP thường được sử dụng cho các ứng dụng web nhỏ hơn và yêu cầu tài nguyên ít hơn.

Nhìn chung, ASP.NET và PHP đều là các công cụ mạnh mẽ để phát triển ứng dụng web. Sự lựa chọn giữa 2 ngôn ngữ này phụ thuộc vào yêu cầu dự án và sở thích cá nhân của người phát triển.

## 2.5. Ứng dụng thực tế của ASP.NET

**Xây dựng các ứng dụng ứng viên việc làm:** Chương trình ASP.NET có tích hợp thuận tiện với các công nghệ tìm kiếm và công cụ phân tích dữ liệu. Phần mềm cho phép phát triển các trang web tuyển dụng, quản lý hồ sơ ứng viên và tìm kiếm việc làm.

**Phát triển ứng dụng e-commerce:** ASP.NET cung cấp các công cụ và thư viện tiện ích để phát triển các ứng dụng e-commerce. Người dùng có thể tạo ra các trang web mua sắm trực tuyến, quản lý giỏ hàng, thanh toán và tích hợp các hệ thống thanh toán trực tuyến.

**Phát triển ứng dụng di động:** ASP.NET hỗ trợ việc phát triển ứng dụng di động thông qua các framework như Xamarin và ASP.NET Core. Từ đó mà lập trình viên dễ dàng xây dựng các ứng dụng di động đa nền tảng sử dụng mã nguồn chung.

**Một số ứng dụng khác:**

* **Phát triển các trang web tương tác:** ASP.NET cho phép lập trình viên tạo ra các trang web tương tác, cho phép người dùng tương tác với dữ liệu và chức năng trực tuyến.
* **Quản lý dữ liệu:** ASP.NET cung cấp các công cụ và thư viện để quản lý dữ liệu trong web ứng dụng. Người lập trình có thể thao tác với cơ sở dữ liệu.
* **Phát triển ứng dụng quản lý dự án:** ASP.NET cung cấp khả năng xây dựng các ứng dụng cho phép quản lý các công việc, lịch trình, nguồn lực và tiến độ dự án.

## 2.6. Cấu trúc dự án asp.net core

Trong một dự án ASP.NET Core MVC, các thành phần như: *Startup.cs, Program.cs, appsettings.json, Controllers, Views,* và *Models.*

### 2.6.1. Program.cs

Đây là điểm khởi đầu của ứng dụng ASP.NET Core. File Program.cs chứa phương thức *Main()* và sử dụng *WebHost.CreateDefaultBuilder()* để cấu hình máy chủ web cho ứng dụng.

**Các công việc chính:**

* Cấu hình máy chủ Kestrel (máy chủ web tích hợp của ASP.NET Core).
* Cấu hình logger để ghi nhận thông tin log.
* Chạy ứng dụng bằng cách gọi *.Build()* và *.Run()*

### 2.6.2. Startup.cs

Startup.cs là nơi cấu hình các dịch vụ và pipeline xử lý yêu cầu *(Request Pipeline)* của ứng dụng.

**Các phương thức chính:**

* *ConfigureServices(IServiceCollection services):* Nơi đăng ký các dịch vụ *(service)* mà ứng dụng cần, như Entity Framework, Identity, hoặc dịch vụ tùy chỉnh khác.
* *Configure(IApplicationBuilder app, IWebHostEnvironment env):* Cấu hình pipeline xử lý yêu cầu. Các middleware như UseRouting, UseEndpoints, và UseAuthorization được thêm ở đây để xử lý các yêu cầu.

### 2.6.3. appsettings.json

Đây là file cấu hình chính cho ứng dụng ASP.NET Core, nơi lưu các thiết lập như chuỗi kết nối cơ sở dữ liệu, cài đặt logging, và các cấu hình tùy chỉnh.

File này có thể chứa các cấu hình cho từng môi trường *(như appsettings.Development.json cho môi trường phát triển).*

Các giá trị từ *appsettings.json* được tải vào cấu hình bằng cách sử dụng *IConfiguration* trong ứng dụng và có thể truy cập dễ dàng từ các phần khác của ứng dụng.

### 2.6.4. Controllers

Thư mục *Controllers* chứa các lớp *controller*, là nơi xử lý các yêu cầu HTTP.

**Cách thức hoạt động:**

* Các *controller* sẽ nhận yêu cầu từ người dùng, tương tác với dữ liệu (thông qua *Models*), và trả về phản hồi (dữ liệu hoặc *view*).
* Mỗi *controller* có thể có nhiều action method tương ứng với các hành động khác nhau như GET, POST, PUT, và DELETE.

### 2.6.5. Views

*Views* là nơi lưu trữ các giao diện người dùng (UI) của ứng dụng.

**Cách thức hoạt động:**

* Các *view* là các tệp HTML động có thể kết hợp với dữ liệu từ *controller*, thường được viết bằng Razor *(.cshtml).*
* *Views* chứa các thư mục con theo tên của *controller* để tổ chức *view* cho từng *controller*, ngoài ra còn có các thư mục như *Shared* cho các *view* dùng chung.

### 2.6.6. Models

*Models* chứa các lớp định nghĩa dữ liệu và logic nghiệp vụ.

**Cách thức hoạt động:**

* *Models* đại diện cho dữ liệu của ứng dụng và các quy tắc nghiệp vụ liên quan.
* Các lớp *Model* thường bao gồm các thuộc tính đại diện cho dữ liệu và các phương thức để quản lý các quy tắc nghiệp vụ hoặc tương tác với cơ sở dữ liệu.

## 2.7. Xây dựng ứng dụng đầu tiên

### 2.7.1. Cài đặt môi trường phát triển

#### 2.7.1.1. Cài đặt Visual Studio

**Bước 1:** Vào trang <https://visualstudio.microsoft.com/downloads/>

Ở đây chúng ta chọn bản Community. Nhấn vào Free download để tải về. Bỏ qua bước này nếu bạn đã có Visual Studio cài đặt trong máy.

**A screenshot of a computer

Description automatically generated**

Ảnh 1: Tải file cài đặt Visual Studio Community

**Bước 2:** Mở file vừa mới tải về, sau đó chọn Run để bắt đầu cài đặt Visual Studio.

**Bước 3:** Chọn Continue và bạn chờ vài phút để tải file cài đặt về.

**Bước 4:** Chọn workloads cần thiết:

* ASP.NET và web development
* .NET Core cross-platform development

**Bước 5:** Chọn vị trí lưu file của Visual Studio. Sau đó bạn Install để bắt đầu quá trình tải và cài đặt.

**A screenshot of a computer

Description automatically generated**

Ảnh 2: Thay đổi vị trí lưu Visual Studio

**Bước 6:** Chờ Visual Studio tải và cài đặt các gói tài nguyên, quá trình này nhanh hay chậm tùy vào tốc độ tải của máy. Chọn Start after installation để khởi động Visual Studio sau khi tải xong.

**Bước 7:** Sau khi cài đặt xong thì mở phần mềm, chọn Create a new project để bắt đầu.

#### 2.6.1.2. Cài đặt .NET SDK

**Bước 1: Tải xuống .NET SDK**

* **Truy cập trang tải xuống:** Mở trình duyệt và truy cập trang chính thức của .NET tại <https://dotnet.microsoft.com/download>
* **Chọn phiên bản:** Chọn phiên bản .NET SDK mới nhất hoặc phù hợp với nhu cầu của bạn (chẳng hạn .NET 6 hoặc .NET 7).
* **Tải về:** Chọn hệ điều hành của bạn (Windows, macOS, hoặc Linux) và tải gói cài đặt tương ứng.

**Bước 2: Cài đặt .NET SDK trên hệ điều hành**

* **Trên Windows:**
  + Mở tệp .exe bạn vừa tải xuống và làm theo hướng dẫn trên màn hình để hoàn tất quá trình cài đặt.
  + Sau khi cài đặt, khởi động lại PC nếu cần thiết.
* **Trên macOS:**
  + Mở tệp .pkg đã tải về và làm theo các bước hướng dẫn để cài đặt.
  + Sau khi cài đặt, khởi động lại Mac nếu cần thiết.
* **Trên Linux:**
  + Các bước cài đặt trên Linux phụ thuộc vào bản phân phối bạn dùng (Ubuntu, Fedora, v.v.). Trang tải xuống của .NET sẽ có hướng dẫn cụ thể cho từng hệ điều hành.

**Bước 3: Kiểm tra cài đặt .NET SDK**

* Mở Command Prompt (trên Windows) hoặc Terminal (trên macOS/Linux).
* Nhập lệnh sau để kiểm tra phiên bản .NET SDK đã cài đặt: dotnet --version
* Nếu lệnh trên hiển thị phiên bản .NET, bạn đã cài đặt thành công .NET SDK.

#### 2.6.1.3. Cài đặt Dotnet CLI (tùy chọn)

Dot Net CLI được cài đặt như một phần của .NET Core SDK. Tải về Dot Net CLI tại <https://dotnet.microsoft.com/en-us/download>

**Các lệnh thường dùng:**

*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

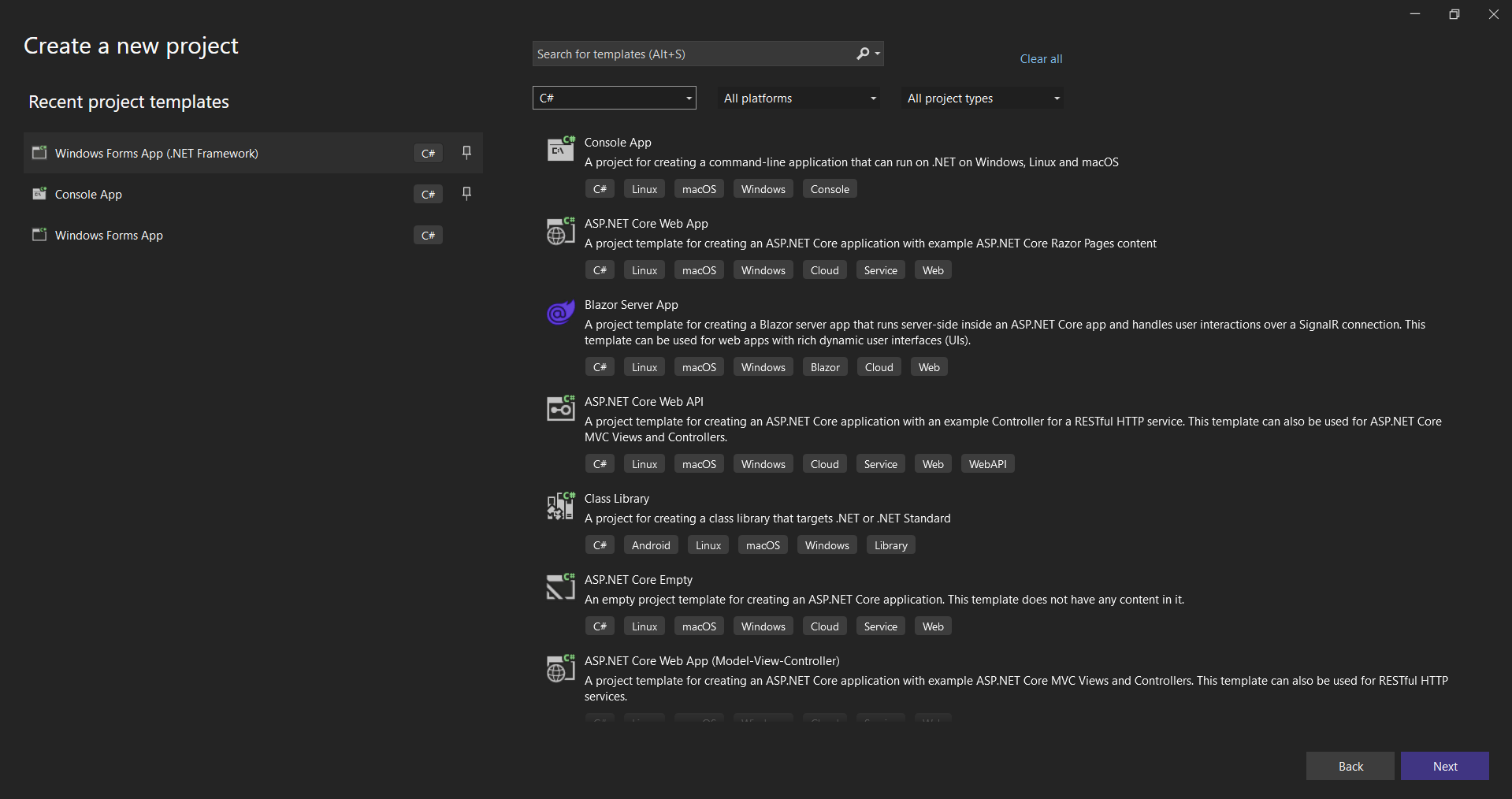
*sln* – Chỉnh sửa file .NET Core solution

*help* – Xem trợ giúp

*store* – Lưu trữ các assembly cụ thể

### 2.7.2. Tạo dự án ASP.NET Core Web Application (Phụ lục 1)

**Bước 1:** Mở Visual Studio > Chọn ASP.NET Core Web App

****

Ảnh 3: Chọn template project

**Bước 2:** Đặt tên Project và vị trí lưu sau đó bấm Next > Create

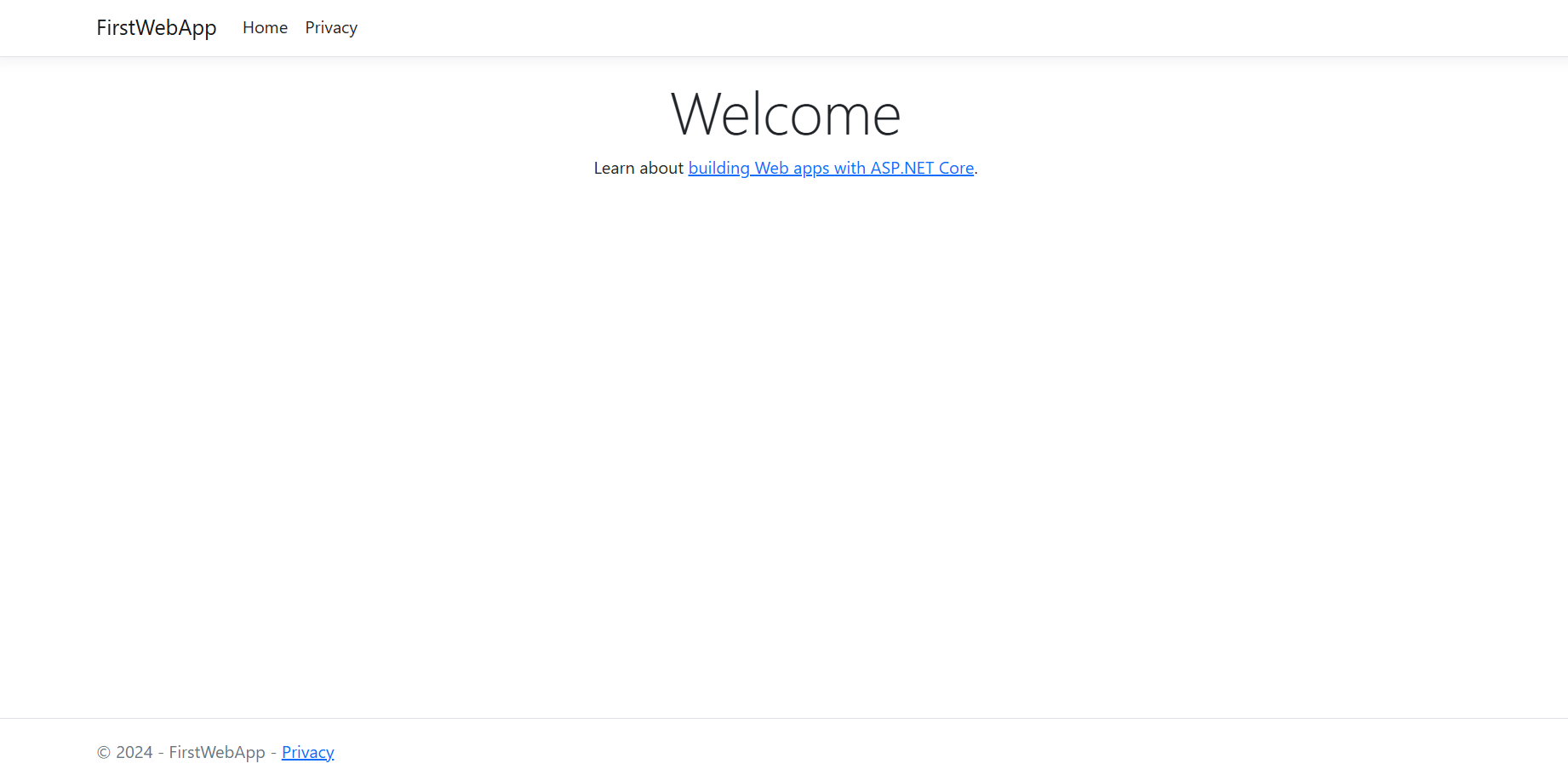
**A screenshot of a computer

Description automatically generated**

Ảnh 4: Tạo project mới

**Bước 3:** Bật tùy chọn *“Allow invalid certificates for resources loaded from localhost.”* trên trình duyệt của bạn.

**Bước 4:** Bấm nút chạy (F5 hoặc Ctrl+F5) chương trình và xem thành quả.



Ảnh 5: Web mẫu được tạo sẵn sau khi tạo project

Việc chạy ứng dụng với Ctrl-F5 *(non-debug)* cho phép bạn làm code thay đổi, lưu file, refresh trình duyệt và thấy ngay những thay đổi của code. Nhiều lập trình viên thích sử dụng chế độ non-debug để nhanh chóng chạy ứng dụng và xem những thay đổi. Điều này là tương tự với ngôn ngữ thông dịch như php, python hay ruby,..

# Chương III. Kestrel – ASP.NET Core Web Server

## 3.1. 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. Có nghĩa là khi bạn build ứng dụng asp.net, chạy nó thì sẽ tự động chạy web server Kestrel.

Github repo: <https://github.com/aspnet/KestrelHttpServer>

## 3.2. 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.

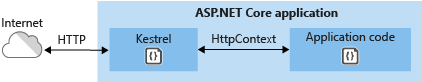
ASP.NET dính chặt vào IIS cũng là gánh nặng cho IIS.

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.

## 3.3. 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.

**Có 2 cách để sử dụng Kestrel:**

* **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:  
  
* **Đằng 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. A diagram of a computer

  Description automatically generated

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

1. 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.
2. Dễ dàng cài đặt cân bằng tải
3. 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 dùng HTTP thuần.
4. Chia sẻ một IP với nhiều địa chỉ
5. Lọc các Request, ghi log và rewrite-URL…
6. 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 đằng trước Kestrel.

## 3.4. Triển khai ứng dụng ASP.NET trên Server Linux với Kestrel Apache Nginx

Trong phần này, ta sẽ đi tìm hiểu một số cấu hình cơ bản để có thể áp dụng làm cơ sở chuẩn bị cho việc publish ứng dụng, triển khai ứng dụng trên môi trường product.

### 3.4.1. Cấu hình Kestrel trong ASP.NET core

Khi khởi tạo ứng dụng ASP.NET từ các template, bao giờ cũng có file Program.cs trong đó định nghĩa hàm *Main* - gọi *CreateHostBuilder* để chạy ứng dụng, code mặc định này đã sử dụng và chạy Kestrel.

|  |
| --- |
| **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.

Phụ thuộc vào môi trường chạy ứng dụng mà Kestrel chạy và lắng nghe ở các cổng khác nhau theo cấu hình của môi trường. Trong môi trường phát triển, thường nó sẽ lắng nghe ở cổng 5000 *(http)* và cổng 5001 *(https)*. Cổng này có thể thiết lập qua biến môi trường hệ thống ASPNETCORE\_URLS (ví dụ trong khi đóng gói vào docker có thiết lập biến môi trường ASPNETCORE\_URLS=https://+:443;http://+:80 thì Kestrel chạy và lắng nghe cổng 443 và 80)

Để tùy biến Kestrel với các thiết lập cụ thể sử dụng đối tượng *KestrelServerOptions* như sau:

|  |
| --- |
| **public static** IHostBuilder **CreateHostBuilder**(**string**[] args) =>  **Host.CreateDefaultBuilder**(args)  **.ConfigureWebHostDefaults**(webBuilder =>  {  **webBuilder.UseKestrel**(kestrelServerOptions => {  // Các thiết lập cho Kestrel tại đây  // sử dụng KestrelServerOptons để thiết lập  })  **webBuilder.UseStartup**<**Startup**>();  }); |

**Các thiết lập Kestrel:**

**Bind IP - thiết lập Kestrel lắng nghe trên cổng và IP:** Có thể sử dụng các phương thức *KestrelServerOptions.Listen*, *KestrelServerOptions.ListenAnyIP*, *KestrelServerOptions.ListenLocalhost* ví dụ:

|  |
| --- |
| **webBuilder.UseKestrel**(kestrelServerOptions => {  // Thiết lập lắng nghe trên cổng 8090 với IP bất kỳ  kestrelServerOptions.Listen(IPAddress.Any, 8090);  // Lắng nghe trên cổng 8091 trên server chạy ứng dụng  kestrelServerOptions.ListenLocalhost(8091);  kestrelServerOptions.Listen(IPAddress.Loopback, 8092, listenOptions => {  // Thiết lập sử dụng SSL - file xác thực SSL testCert.pfx  listenOptions.UseHttps("testCert.pfx", "testPassword");  });  }); |

**Thiết lập một số giới hạn của Kestrel với KestrelServerLimits:** Nếu muốn thay đổi các giới hạn mặc định thì có thể dùng, *KestrelServerLimits* với các thuộc tính đặt giới hạn, ví dụ thiết lập *keep-alive timeout*

|  |
| --- |
| **webBuilder.UseKestrel**(kestrelServerOptions => {  /...  kestrelServerOptions.Limits.KeepAliveTimeout = TimeSpan.FromMinutes(2);  }); |

**Tham khảo thêm các thiết lập khác tại:** kestrelserverlimits [https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.server.kestrel.core.kestrelserverlimits]

* **Thiết lập Bind IP cho Server:**

Có thể nhanh chóng dùng *webBuilder.UseUrls* để thiết lập IP và cổng của Kestrel, dùng địa chỉ Url để thiết lập Kestrel lắng nghe trên nó. Trong đó có chỉ ra giao thức (http, https), địa chỉ IP, và cổng. Ví dụ:

|  |
| --- |
| **public static** IHostBuilder **CreateHostBuilder**(**string**[] args) =>  **Host.CreateDefaultBuilder**(args)  .**ConfigureWebHostDefaults**(webBuilder =>  {  **webBuilder.UseUrls**("http://0.0.0.0:8090", "https://0.0.0.0:8091");  **webBuilder.UseStartup**<**Startup**>();  }); |

Sử dụng Kestrel phía sau các Http Server khác (Http Apache, Nginx, IIS) thì nên sử dụng cách này (*UseUrls*), và chỉ cần lắng nghe ở giao thức http, còn https được cấu hình ở server phía trước (*nginx, apache*).

|  |
| --- |
| **public static** IHostBuilder **CreateHostBuilder**(string[] args) =>  **Host.CreateDefaultBuilder**(args)  .**ConfigureWebHostDefaults**(webBuilder =>  {  // Chỉ nhận http (không https)  **webBuilder.UseUrls**("http://0.0.0.0:5000");  **webBuilder.UseStartup**<**Startup**>();  }); |

Ngoài ra cũng chú ý, để sử dụng chính xác Kestrel với nginx, apache cần thêm vào đầu *Startup.configure*

|  |
| --- |
| ...  // using Microsoft.AspNetCore.HttpOverrides;  **app.UseForwardedHeaders**(new **ForwardedHeadersOptions**  {  ForwardedHeaders = **ForwardedHeaders.XForwardedFor** **| ForwardedHeaders.XForwardedProto**  });  ... |

### 3.4.2. Publish ứng dụng ASP.NET core

Để biên dịch mã nguồn chạy ở môi trường Product, có thể thực hiện các lệnh như sau:

|  |
| --- |
| # phục hồi các dependency từ Nuget  dotnet restore  # build  dotnet build -c Release -o app/build  # publish  dotnet publish -c Release -o app/publish |

Kết quả là xuất ra ứng dụng ra thư mục app/publish, bạn dùng thư mục này để phân phối - triển khai chạy ứng dụng. Trong thư mục có file dll tên ứng dụng dùng để chạy ứng dụng.

|  |
| --- |
| dotnet tên-ứng-dụng.dll |

Chú ý, server chạy ứng dụng phải cài .NET Core SDK và .NET Core Runtime.

**Kèm thư mục khi publish**

Trong dự án có thể có các thư mục tài nguyên, ví dụ thư mục Uploads, nếu muốn thư mục này được copy vào publish thì trong file: .csproj thêm vào đoạn mã:

|  |
| --- |
| <**ItemGroup**>  <**Content** Include="Uploads\\*\*">  <**CopyToOutputDirectory**>PreserveNewest</**CopyToOutputDirectory**>  </Content>  </**ItemGroup**> |

### 3.4.3. Giám sát ứng dụng ASP.NET với systemd

Giả sử ứng dụng lưu trên Server Linux ở đường dẫn */home/userasp/mvcblog* (đã copy nội dung thư mục *app/publish* vào *mvcblog*), file chạy ứng dụng là *mvcblog.dll*, nó được publish và lắng nghe ở cổng http 8090 (cổng này do bạn đặt trong ứng dụng ở phần trên)

Bạn hoàn toàn có thể vào thư mục, chạy ứng dụng với lệnh:

|  |
| --- |
| cd /home/userasp/mvcblog/  dotnet mvcblog.dll |

Khi bạn chạy trực tiếp như vậy, nếu ứng dụng bị crash vì một sự cố nào đó - thì nó sẽ dừng hẳn và không được khởi động lại. Giải quyết trường hợp này hãy sử dụng systemd của Linux, giúp giám sát tình trạng ứng dụng, khởi động ứng dụng nếu chưa chạy hoặc bị crash.

Tạo ra file dịch vụ trong thư mục */etc/systemd/system/,* ví dụ ứng dụng là *mvcblog*, tạo ra file */etc/systemd/system/mvcblog.service* (dùng lệnh vi), sau đó biên tập nội dung file này như sau:

|  |
| --- |
| [Unit]  Description=Ung dung ASP.NET MVC BLOG  [Service]  WorkingDirectory=/home/userasp/mvcblog  ExecStart=/usr/bin/dotnet /home/userasp/mvcblog/mvcblog.dll  Restart=always  # Khởi động lại ứng dụng sau 10 bị crash  RestartSec=10  KillSignal=SIGINT  SyslogIdentifier=asp-net-app  User=userasp  Environment=ASPNETCORE\_ENVIRONMENT=Production  [Install]  WantedBy=multi-user.target |

*/usr/bin/dotnet* là đường dẫn đầy đủ đến file binary của lệnh dotnet, có thể kiểm tra bằng lệnh:

|  |
| --- |
| which dotnet |

Thiết lập dịch vụ tự động chạy:

|  |
| --- |
| systemctl enable mvcblog |

Khởi chạy dịch vụ (ứng dụng):

|  |
| --- |
| systemctl start mvcblog |

Xem trạng thái

|  |
| --- |
| systemctl status mvcblog |

Lưu ý, ứng dụng ASP.NET chạy trên lắng nghe ở cổng 8090 hoặc cổng do bạn thiết lập, cổng này không public ra ngoài (không dùng firewall để mở). Cổng chỉ dùng cho các dịch vụ như apache, nginx chuyển hướng đến.

# Chương IV. Middleware và Request Pipeline trong ASP.NET Core

## 4.1. 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).*

Tóm lại, Middleware là một module code nó nhận yêu cầu gửi đến Request và trả về Response. Cụ thể trong ASP.NET Core, middlewarre có thể:

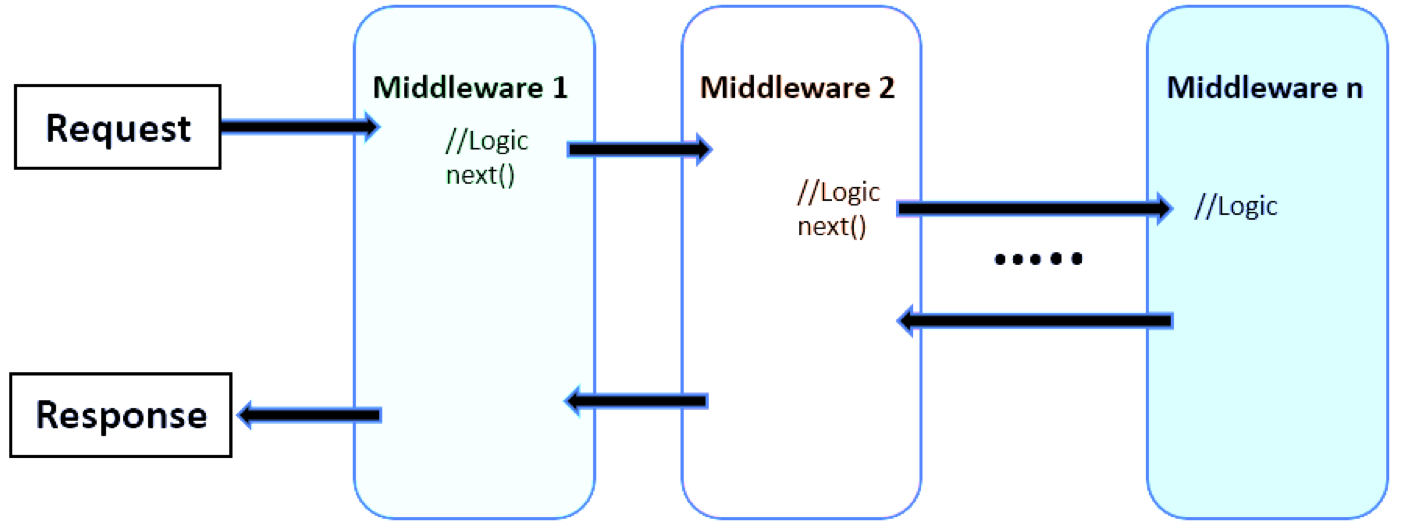
* Nhận một HTTP Request gửi đến và phát sinh ra HTTP Response để trả về
* Nhận một HTTP Request gửi đến, thi hành một số tác vụ (có thể là sửa đổi HTTP Request), sau đó chuyển đến một middleware khác.
* Nhận HTTP Response, sửa nó và chuyển đến một Middleware khác

## 4.2. 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.

Trong ứng dụng ASP.NET Core, các middlware kết nối lại với nhau thành một xích, middleware đầu tiên nhận HTTP Request, xử lý nó và có thể chuyển cho middleware tiếp theo hoặc trả về ngay HTTP Response. Chuỗi các middleware theo thứ tự như vậy gọi là pipeline.

Các middleware như là các dịch vụ nhỏ, đăng ký vào ứng dụng bằng cách sử dụng đối tượng IApplicationBuilder, sau đó ứng dụng căn sẽ xây dựng lên các pipeline (luồng xử lý) cho các truy vấn gửi đến.



Ảnh 6: Sơ đồ các middleware trong pipeline

Để thực hành, tạo ra ứng dụng web đơn giản với cách lệnh sau:

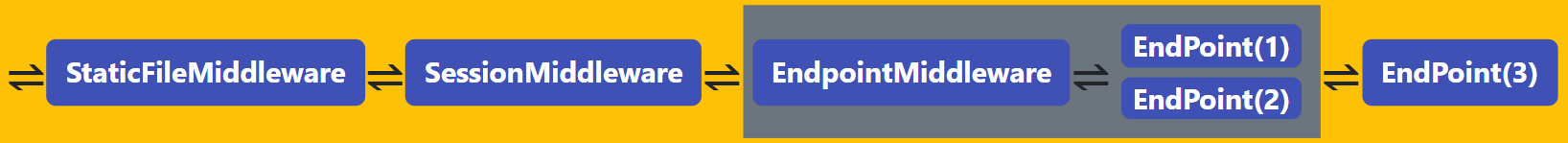
|  |
| --- |
| mkdir 02.middleware # tạo thư mục lưu code  cd 02.middleware # vào thư mục  dotnet new web # tạo dự án với template mẫu, đơn giản  code . # mở ra bằng Visual Studio Code, nhớ chọn .NET: Generate Assets for Build and Debug lần đầu mở ra |

Trong ứng dụng ASP.NET CORE, có sẵn một một loạt Middleware hoặc có thể xây dựng thêm để đưa vào một pipeline, luồng xử lý HttpRequest và trả về HttpResponse. Việc thêm Middleware vào pipeline được thực hiện ở phương thức Configure của lớp Startup. Các middleware trong pipeline xử lý các HTTP Message đều có dạng - nhận đầu vào là tham số kiểu HttpContext (có chứa HttpRequest và HttpRespone), thi hành code, sau đó chuyển đến cho middleware tiếp theo.

Ví dụ sửa mã *Startup.cs* có sử dụng một số Middleware như sau:

|  |
| --- |
| using System;  using System.Collections.Generic;  using System.Linq;  using System.Threading.Tasks;  using Microsoft.AspNetCore.Builder;  using Microsoft.AspNetCore.Hosting;  using Microsoft.AspNetCore.Http;  using Microsoft.Extensions.DependencyInjection;  using Microsoft.Extensions.Hosting;  **namespace \_02.middleware**{  **public class Startup** {  // Đăng ký các dịch vụ sử dụng bởi ứng dụng, services là một DI container  **public void ConfigureServices**(IServiceCollection services) {  // Thêm dịch vụ dùng bộ nhớ lưu cache (session sử dụng dịch vụ này)  **services.AddDistributedMemoryCache**();  // Thêm dịch vụ Session, dịch vụ này cung cấp Middleware  **services.AddSession**();  }  **public void Configure**(**IApplicationBuilder** app, **IWebHostEnvironment** env){  // Thêm StaticFileMiddleware - nếu Request là yêu cầu truy cập file tĩnh,  // Nó trả ngay về Response nội dung file và là điểm cuối pipeline, nếu khác  // nó gọi Middleware phía sau trong Pipeline  **app.UseStaticFiles**();  // Thêm SessionMiddleware: khôi phục, thiết lập - tạo ra session  // gán context.Session, sau đó chuyển gọi ngay middleware  // tiếp trong pipeline  **app.UseSession**();  // Thêm EndpointRoutingMiddleware: ánh xạ Request gọi đến Endpoint (Middleware cuối)  // phù hợp định nghĩa bởi EndpointMiddleware  **app.UseRouting**();  // app.UseEndpoint dùng để xây dựng các endpoint - điểm cuối của pipeline theo Url truy cập  **app.UseEndpoints**(endpoints =>  {  // EndPoint(2) khi truy vấn đến /Testpost với phương thức post hoặc put  **endpoints.MapMethods**("/Testpost" , **new** **string**[] {"post", "put"}, **async** **context** => {  **await context.Response.WriteAsync**("post/pust");  });  // EndPoint(2) - Middleware khi truy cập /Home với phương thức GET - nó làm Middleware cuối Pipeline  **endpoints.MapGet**("/Home", **async** context => {  **int**? count = **context.Session.GetInt32**("count");  count = (count != null) ? count + 1 : 1;  **context.Session.SetInt32**("count", count.Value);  **await context.Response.WriteAsync**($"Home page! {count}");  });  });  // EndPoint(3) app.Run tham số là hàm delegate tham số là HttpContex  // - nó tạo điểm cuối của pipeline.  **app.Run**(**async** context => {  context.Response.StatusCode = **StatusCodes.Status404NotFound**;  **await context.Response.WriteAsync**("Page not found");  });  }  }  } |

Với cấu hình trên khi chạy ứng có thể khởi tạo lên các pipleline để phân luồng đi của yêu cầu gửi đến (HttpContext), gồm có:



Với cấu hình trên, tùy thuộc vào truy vấn mà HttpContext có thể đi qua các Middleware: StaticFileMiddleware, SessionMiddleware, EndpointRoutingMiddleware, EndPoint(1), EndPoint (2), EndPoint (3), ví dụ:

* Nếu truy cập đến một file tĩnh (file chứa trong thư mục wwwroot), thì điểm cuối của pipeline là *StaticFileMiddleware* (các middleware khác không được sử dụng).
* Nếu truy cập địa chỉ /Home (http://localhost:5000/Home) thì HttpContext đi qua các Middleware: *StaticFileMiddleware, SessionMiddleware, EndpointMiddleware, EndPoint(2)*
* Nếu truy cập địa chỉ / thì HttpContext đi qua các Middleware: *StaticFileMiddleware, SessionMiddleware,EndpointMiddleware, EndPoint(3)*
* Nếu truy cập địa chỉ /Testpost với phương thức post, update thì HttpContext đi qua các Middleware: *StaticFileMiddleware, SessionMiddleware, EndpointMiddleware, EndPoint(1)*

## 4.3. Tạo Middleware riêng trong ASP.NET Core (Phụ lục 2)

Giờ ta sẽ thực hành tạo ra một Middleware như sau: tạo Middleware có tên là *CheckAcessMiddleware*, đăng ký Middleware này nằm sau *SessionMiddleware* trong pipeline của ứng dụng trên, *CheckAcessMiddleware* có chức năng - kiểm tra nếu truy cập là *URL /testxxx* thì không gọi Middleware tiếp theo mà thiết lập ngay HttpRespone hiện thị thông báo không được truy cập.

Còn nếu Url khác */testxxx* thì thêm vào HttpResponse một Header tên *throughCheckAcessMiddleware* với giá trị ngày tháng - để biết HttpRequest đã đi qua Middleware này sau đó gọi Middleware tiếp theo trong pipeline.

**Cấu trúc Middleware trong ASP.NET**

Một lớp (class) phù hợp là một Middleware trong ASP.NET nếu lớp đó có cấu trúc thỏa mãn những điều kiện sau:

* Có một phương thức khởi tạo public với tham số thứ nhất kiểu *RequestDelegate*, nếu có tham số thứ 2 thì các tham số tiếp theo này phải Inject được từ DI của hệ thống
* Phải có tổi thiểu một trong hai phương thức có tên *Invoke* hoặc *InvokeAsync* với tham số nhận là *HttpContext*, những phương thức này phải trả về *Task*. (Dùng *InvokeAsync* nếu muốn áp dụng kỹ thuật bất đồng bộ - nên làm)
* Trong *Invoke/InvokeAsync* bạn viết code xử lý tác vụ của Middleware, sau đó quyết định chuyển đến Middleware tiếp theo bằng cách gọi *RequestDeleage* đã truyền đến trong hàm tạo, hoặc không chuyến đến Middlware tiếp theo thì cần đảm bảo lúc này HttpResponse trrong HttpContext đã phù hợp để trả về cho Client.

**Bước 1: Tạo Middleware tên CheckAcessMiddleware**

Middleware/CheckAcessMiddleware.cs

|  |
| --- |
| using System;  using System.Collections.Generic;  using System.Linq;  using System.Threading.Tasks;  using Microsoft.AspNetCore.Hosting;  using Microsoft.AspNetCore.Http;  using Microsoft.Extensions.Configuration;  using Microsoft.Extensions.Hosting;  using Microsoft.Extensions.Logging;  **namespace \_02.middleware** {  **public class CheckAcessMiddleware** {  // Lưu middlewware tiếp theo trong Pipeline  **private readonly RequestDelegate \_next**;  **public CheckAcessMiddleware** (**RequestDelegate** next) => \_next = next;  **public async Task Invoke** (**HttpContext** httpContext) {  **if** (**httpContext.Request.Path** == "/testxxx") {  **Console.WriteLine** ("CheckAcessMiddleware: Cấm truy cập");  **await Task.Run** (  **async** () => {  **string** html = "<h1>CAM KHONG DUOC TRUY CAP</h1>";  httpContext.Response.StatusCode = **StatusCodes.Status403Forbidden**;  **await httpContext.Response.WriteAsync** (html);  }  );  } **else** {  // Thiết lập Header cho HttpResponse  **httpContext.Response.Headers.Add** ("throughCheckAcessMiddleware", new [] { DateTime.Now.ToString () });  **Console.WriteLine** ("CheckAcessMiddleware: Cho truy cập");  // Chuyển Middleware tiếp theo trong pipeline  **await \_next** (httpContext);  }  }  }  } |

**Bước 2: Đưa Middleware vào pipeline**

Như vậy đã có một Middleware theo đúng chuẩn để có thể đưa vào pipeline nhận HttpRequest của ASP.NET Core. Để đăng ký chỉ việc sử dụng phương thức *UseMiddleware* của *IapplicationBuilder*.

Ví dụ, tại phương thức *Configure* của *Startup*

|  |
| --- |
| **public void Configure**(**IApplicationBuilder** app, **IHostingEnvironment** env)  {  / ..  **app.UseStaticFiles**();  **app.UseSession**();  // Đưa Middleware vào pipeline - vị trí thứ 3  **app.UseMiddleware**<**CheckAcessMiddleware**>();  /..  } |

Tuy nhiên ta sẽ làm theo cách các Middleware mặc định được đăng ký, tức là thêm phương thức mở rộng vào *IApplicationBuilder*, ví dụ thêm vào nó phương thức *UseCheckAccess*.

Middleware/MyAppExtensions.cs

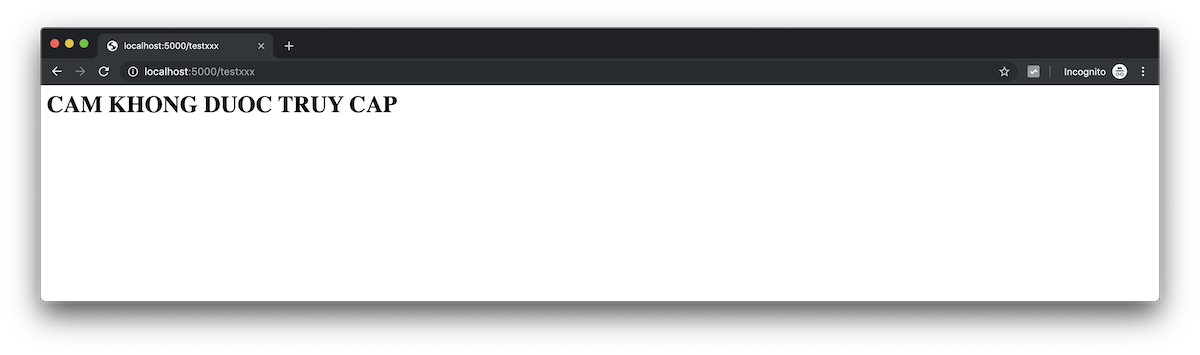
|  |
| --- |
| using Microsoft.AspNetCore.Builder;  **namespace \_02.middleware**{  **public static class MyAppExtensions**  {  // Mở rộng cho IApplicationBuilder phương thức UseCheckAccess  **public static IApplicationBuilder UseCheckAccess**(**this IApplicationBuilder** builder)  {  **return builder.UseMiddleware**<**CheckAcessMiddleware**>();  }  }  } |

Đã có phương thức mở rộng UseCheckAccess, để đăng ký Middleware vào pipeline thực hiện:

|  |
| --- |
| **public void Configure**(**IApplicationBuilder** app, **IHostingEnvironment** env)  {  / ..  **app.UseStaticFiles**(); // Đưa Middleware vào pipeline - vị trí đầu tiên  **app.UseSession**(); // Đưa Middleware vào pipeline - vị trí thứ 2  **app.UseCheckAccess**(); // Đưa Middleware vào pipeline - vị trí thứ 3  /..  } |

**Bước 3: Kiểm tra:**

Giờ truy cập vào địa chỉ /testxxx thì Request sau khi qua Middleware Session, đến Middleware *CheckAccess* thì bị chặn lại, các Middleware phía sau không được gọi.



Trong trường hợp, địa chỉ truy cập khác /testxxx, thì Request sau khi đi qua *CheckAccess* - thì Response có thêm một Header và chuyển các Request cho Middleware tiếp theo xử lý:

A screenshot of a computer

Description automatically generated

Header được thêm vào sau khi đi qua middleware *CheckAccess*

## 4.4. Xây dựng Middleware triển khai từ giao diện IMiddleware (Phụ lục 3)

Bạn có thể triển khai giao diện IMiddleware, trong đó chỉ việc xây dựng phương thức:

|  |
| --- |
| **Task InvokeAsync**(HttpContext context, RequestDelegate next) |

Ví dụ, xây dựng một Middleware - đặt tên *FrontMiddleware*, Middleware này không làm gì cả, nó chỉ in ra console thông báo cho biết Request đã đi qua nó - ta sẽ đặt Middleware này ở đầu pipeline, sau đó chuyển đến Middleware tiếp theo trong pipeline.

|  |
| --- |
| using System;  using System.Collections.Generic;  using System.Linq;  using System.Threading.Tasks;  using Microsoft.AspNetCore.Hosting;  using Microsoft.AspNetCore.Http;  using Microsoft.Extensions.Configuration;  using Microsoft.Extensions.Hosting;  using Microsoft.Extensions.Logging;  **namespace \_02.middleware** {  **public class FrontMiddleware : IMiddleware**  {  **public async Task InvokeAsync**(**HttpContext** context, **RequestDelegate** next)  {  **Console.Clear**();  **Console.WriteLine**("FrontMiddleware: " + context.Request.Path);  **await next**(context);  }  }  } |

Đối với loại Middleware này để sử dụng nó, phải đăng ký vào DI container trước (đăng ký dịch vụ) trong phươnng thức *ConfigureServices* của Startup

|  |
| --- |
| public void ConfigureServices(IServiceCollection services)  {  **services.AddTransient**<**FrontMiddleware**>();  //.. các đăg ký khác  } |

Sau đó thêm vào pipeline theo cách thông thường, khai báo đầu tiên trong Configure để nó đứng đầu pipeline:

|  |
| --- |
| public void Configure(IApplicationBuilder app, IWebHostEnvironment env)  {  **app.UseMiddleware**<**FrontMiddleware**>();  // ... các Middleware khác  } |

Chạy ứng dụng, để thấy HttpContext đi qua Middleware trên.

## 4.4. Truyền dữ liệu giữa các Middleware (Phụ lục 4)

Để truyền dữ liệu giữa các Middleware hãy sử dụng *HttpContext.Items*, nó là dữ liệu kiểu *IDictionary*, bạn có thể lưu dữ liệu vào trong nó ở Middleware này và ở Middleware khác đọc được.

**Lưu dữ liệu:**

|  |
| --- |
| **context.Items.Add**("dulieu1", "Data Object ..."); |

**Đọc dữ liệu:**

|  |
| --- |
| ar data = **context.Items**["dulieu1"]; |

# Chương V. 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)…

## 5.1. 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* để kết nối đến cơ sở dữ liệu được lưu trong một file cấu hình. Bằng cách thay đổi chuỗi connection 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**> |

## 5.2. Cấu hình trong ASP.NET Core

Cấu hình trong ứng dụng ASP.NET Core được lưu dưới dạng cặp key-value. Có thể lưu chúng trong file JSON, XML hay INI.

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.

## 5.3. 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 (tự tạo ra provider riêng theo ý muốn)

## 5.4. Load cấu hình

Tạo một ứng dụng ASP.NET Core Empty Project. Mở Program.cs nó sẽ có phương thức Main.

|  |
| --- |
| **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);  }  }) |

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.

**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);  } |

## 5.5. 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 không phân biệt chữ hoa chữ thường.

|  |
| --- |
| "**ConnectionStrings**" :  {  "**MySQLConnection**": "This is MySQL Connection",  "**SQLConnection**": "This is SQL Server Connection"  } |

Chuỗi kết nối MySQL và SQL được nhóm vào một section là ConnectionStrings hoặc một node nếu là XML. Mỗi một section hoặc 1 node được chia tách bằng dấu phẩy.

Sau đó hãy đọc thông số *ConnectionStrings* này ra. Mở file *Startup.cs*. 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.

|  |
| --- |
| **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**("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 dổ console và đến thư mục project sau đó gọi *dotnet run* với 2 tham số command line:

|  |
| --- |
| **Dotnet run** arg1=Hello arg2=World |

**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>"); |

## 5.6. 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>"); |

Nếu có nhiều cấu hình cùng lúc thì cấu hình được load sau sẽ ghi đè lên cấu hình trước.

# Chương VI. Sử dụng Static Files trong ASP.NET Core

ASP.NET Core có khả năng duyệt file tĩnh như là HTML, CSS, hình ảnh và Javacript trực tiếp từ client mà không đi qua MVC Middleware.

## 6.1. 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. 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. ASP.NET Core cung cấp một Middleware có sẵn chỉ phục vụ việc này.

**Static file middleware**

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.

**Vị trí đặt Static file**

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 chứa các tài nguyên tĩnh như CSS, Javascript, hình ảnh.

## 6.2. Tạo mới Static file

Để thêm static file, ví dụ bạn thêm test.html. Chọn thư mục wwwroot, click chuột phải và chọn Add > Add 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**()) |

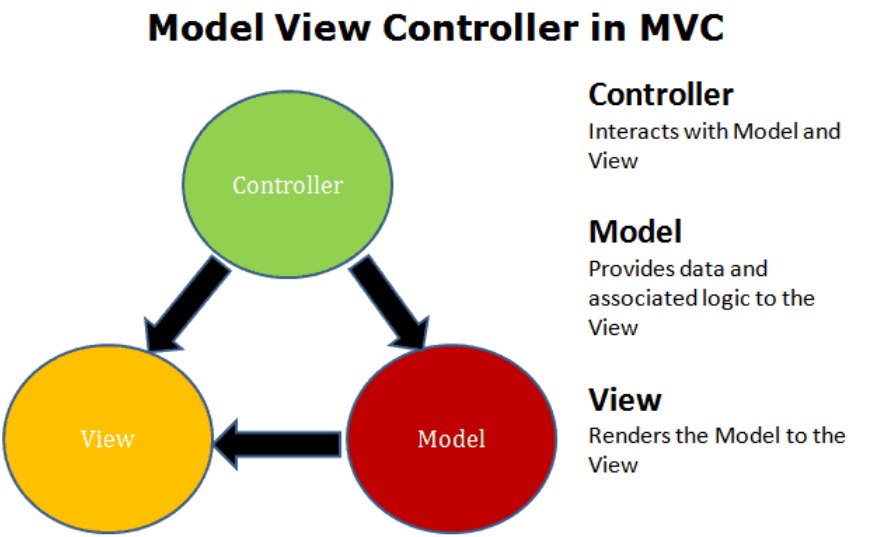
Theo quy tắc thì webroot được đặt cho thư mục wwwroot. Bạn có thể thay đổi bằng cách sử dụng phương thức UseWebRoot.

# Chương VII. MVC Design Pattern trong ASP.NET Core

## 7.1. MVC Design Pattern trong ASP.NET Core

MVC Design Pattern là design pattern phổ biến được sử dụng cho lớp trình bày (Presentation Layer). 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, 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.

****

Ảnh 7: MVC Design Pattern

## 7.2. 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 mộ 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 Bussiness 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 dư liệu đi vào đối tượng Model và cập nhật view khi có dữ liệu thay đổi. Nó giữ View và Model tách biệt.

## 7.3. 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. 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.

# Chương VIII. Xây dựng ứng dụng ASP.NET Core MVC đầu tiên

## 8.1. Tạo ứng dụng ASP.NET MVC đầu tiên

MVC đó là một mô hình lập trình, cố gắng chia code thành 3 phần: *model* triển khai các mô hình dữ liệu, truy cập dữ liệu; *view* biểu diễn dữ liệu cho người dùng; *controller* nhận các yêu cầu browser gửi đến và trả về http response.

Trong NET Core 3.x tạo ứng dụng mẫu ASP.NET MVC nhanh từ template bằng lệnh

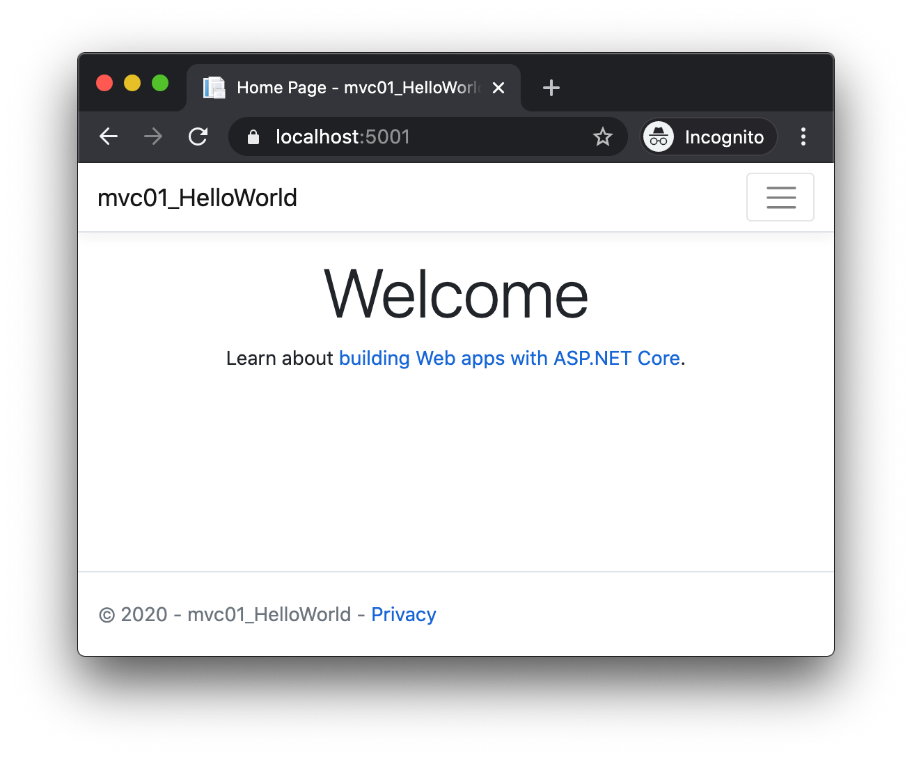
|  |
| --- |
| dotnet new mvc |

Hãy tạo ra một thư mục mẫu đặt tên là mvc01\_HelloWorld, vào thư mục đó, rồi gõ lệnh trên để tạo dự án thực hành.

Dự án mẫu tạo ra là cấu trúc cơ bản cho ứng dụng ASP.NET CORE MVC. Trong tổ chức thư mục dự án thấy có:

* Controllers : nơi chứa các lớp sẽ nhận yêu cầu gửi đến và trả về nội dung HTTP Response.
* Models: nơi chứa các lớp cung cấp, xử lý dữ liệu
* Views: chứa các file .cshtml để dựng HTML
* wwwroot: chứa các file tĩnh (hình ảnh, js, css ...). Trong thư mục này đã copy sẵn thư viện CSS Bootstrap và JS Jquery

Kết quả chạy trang web như hình sau:



Ảnh 8: Khởi tạo dự án MVC

Một số khái quát chung:

* Controller: Một lớp C# do bạn định nghĩa kế thừa từ lớp *Microsoft.AspNetCore.Mvc.Controller*, khi các truy vấn đến ứng dụng tùy thuộc vào cấu hình, căn cứ vào URL truy cập mà controller được khởi tạo và thiết lập vào nó (trong các property) tất cả thông tin của một reques, và một phương thức trong controller (một action) được thực thi để xử lý truy vấn.
* Action: Các action là những phương thức public (không là static, không overloaded) trong controller được gọi tự động tùy thuộc vào sự điều hướng của route trong ứng dụng (căn cứ vào URL).
* Action trả về các kiểu triển khai từ *IActionResult* như *ViewResult*, *EmptyResult*, *FileContentResult*, *RedirectResult*, *JsonResult*... đó là respone sẽ trả về cho browser.
* Tương ứng trong controller có sẵn các phương thức để trả về kiểu tương ứng trên như: View, Redirect, RedirectToAction, Content, File ..., một số Action có thể trả về không phải kiểu triển khai từ IActionResult thì nó tự động tạo ContentResult.
* Các request (truy vấn) được ánh xạ vào action thông qua routing. Các Action thường trả về bằng cách gọi phương thức *View()*, *View(model)*.
* View: Là các file .cshtml tích hợp sẵn cú pháp Razor (razor engine), (hãy xem về Razor Page để biết cú pháp viết trong view .cshtml) được tổ chức thành các thư mục cho từng Controller. Nếu controller tên là Home thì các view đặt trong thự mục /View/Home, tên các file view tương ứng tên Action của Controller, qua đó nó được dùng để dựng HTML.

## 8.2. Truy cập ứng dụng với HTTPS

Mở file *Startup.cs* ra, thấy mã sinh ra có đưa vào pipeline một Middleware là *HttpsRedirection* với lệnh *app.UseHttpsRedirection();*. Điều này có nghĩa ứng dụng tạo ra mặc định chỉ truy cập bởi https, sau này trên môi trường sản phẩm bạn cần cấu hình Web Server như Apache, Nginx, IIS để xác thực ssl. Tuy nhiên, ở môi trường phát triển bạn cần tự xác thực SSL - nếu không thì có lỗi truy cập. Công cụ dotnet hỗ trợ làm điều này, bạn thực hiện lệnh sau:

|  |
| --- |
| dotnet dev-certs https --clean  dotnet dev-certs https --trust |

Sau đó bạn đã có thể truy cập được đến [*https://localhost:5001*](https://localhost:5001)

## 8.3. Route - Định tuyển trong ASP.NET MVC

Routing *(định tuyến)* là quá quá trình xử lý thông qua nó ứng dụng xác định được cần thi hành các phương thức nào cho mỗi URL gửi đến. ASP.NET Core sử dụng một middleware để ánh xạ các URL tương ứng với phương thức nào của ứng dụng. Cụ thể hơn, Route xác định được phương thức (action) trong Controller - qua đó nó chuyển các thông tin Request đến và thi hành.

**IEndpointRouteBuilder.MapControllerRoute**

Cú pháp cơ bản đó là:

|  |
| --- |
| **MapControllerRoute**(string name, string pattern, object defaults = null, object constraints = null); |

*name* tên của route; *pattern* mẫu URL; *defaults* giá trị mặc định cho route; *constraints* là các ràng buộc.

Ví dụ:

|  |
| --- |
| /...  **app.UseRouting**();  // Tạo các Endpoint  **app.UseEndpoints**(endpoints =>  {  **endpoints.MapControllerRoute**(  **name**: "default",  **pattern**: "{controller=Home}/{action=Index}/{id?}");  });  /... |

Đoạn code trên đã tạo ra một Route ánh xạ URL vào Controller, mẫu này xác định trong pattern đó là URL với cấu trúc {controller=Home}/{action=Index}/{id?} nghĩa URL phù hợp mẫu có ba phần /controller/action/id phần thứ nhất tương ứng với tên Controller (tên lớp trong thư mục Controllers, ví dụ này là HomeController), phần thứ 2 biến action - là Action trong Controller (tên phương thức), phần thứ 3 là một tham số tên biến id (tham số này có dấu ? có nghĩa là có thể có hoặc không - tùy chọn)

Như vậy nếu truy cập địa chỉ /Home/Index thì có nghĩa sẽ tham chiếu đến lớp HomeController, gọi đến action (phương thức) Index của nó.

Tất nhiên, sẽ thêm được nhiều endpoint ánh xạ URL, khi truy vấn đến nó so sánh từ trên xuống, gặp cái nào phù hợp sẽ chuyển thi hành controller/action tương ứng. Ví dụ:

|  |
| --- |
| **app.UseEndpoints**(endpoints =>  {  **endpoints.MapControllerRoute**(  **name**: "default",  **pattern**: "{controller=Home}/{action=Index}/{id?}");  **endpoints.MapControllerRoute**(  **name**: "myroute",  **defaults**: new {controller="Home", action = "Index"},  **pattern**: "{title}-{id}.html");  }); |

Ở Route tên *myroute* phù hợp khi truy cập URL dạng */tieude-134.html*, ROUTE này không chỉ rõ Controller, Action trong pattern mà chỉ ra trong defaults như trên. Các Url phù hợp đều dẫn đến thi hành *HomeController::Index*

**Tạo ràng buộc trong Route – URL**

Với mỗi biến trong Route (như controller, action, id, title ...) có thể khống chế phải phù hợp với điều kiện nào đó, thường những điều kiện là biểu thức chính quy (RegEx - xem biểu thức chính quy). Ví dụ như Route có tên myroute sau đây:

|  |
| --- |
| /..  **app.UseEndpoints**(endpoints =>  {  **endpoints.MapControllerRoute**(  **name**: "default",  **pattern**: "{controller=Home}/{action=Index}/{id?}");  **endpoints.MapControllerRoute**(  **name**: "myroute", // đặt tên route  **defaults**: new {controller="Home", action = "Index"},  **constraints**: new {  id = @"\d+", // id phải có và phải là số  title = new RegexRouteConstraint(new Regex(@"^[a-z0-9-]\*$")) // title chỉ chứa số, chữ thường và ký hiệu -  },  **pattern**: "{title}-{id}.html");  });  /.. |

Bạn cũng có thể viết các dàng buộc trực tiếp trong pattern (xem thêm ở EndpointMiddleware đọc kỹ phần ràng buộc), các dàng buộc tham khảo tại route-constraint-reference ), ví dụ:

|  |
| --- |
| **endpoints.MapControllerRoute** (  **name**: "myroute", // đặt tên route  **defaults** : new { controller = "Home", action = "Index" },  **pattern**: "{title:alpha:maxlength(8)}-{id:int}.html");  // title chỉ chứa các chữ cái, dài tối đa 8, id là số nguyên |

## 8.4. Tạo Controller trong ASP.NET MVC

Controller có chức năng nhận Request gửi đến và trả về Response. ASP.NET MVC thì các Controller lưu trong thư mục *Controllers*, đồng thời các lớp controller đặt tên có hậu tố là Controller, ví dụ \*Controller, như trong ứng dụng đã tạo ra có sẵn lớp *HomeController*

Thông thường các controller kế thừa từ lớp *Controller*, sau đó khai báo các phương thức có thể được gọi khi Route ánh xạ đến gọi là Action.

|  |
| --- |
| **public class HomeController** : **Controller**  {  /...  **public IActionResult Index**()  {  **return View**();  }  /...  } |

**Các Action**

Controller có các phương thức gọi là Action, như trên action Index nó trả về IActionResult, thực tế nó có thể trả về bất kỳ kiểu dữ liệu nào. Ví dụ:

|  |
| --- |
| /...  **public string XinChao**() => "Xin chào ASP.NET Core 3.0";  /... |

Truy cập địa chỉ */Home/XinChao* thấy dòng text trên.

Trong Controller có sẵn các phương thức trả về các kiểu triển khai từ *IActionResult* như:

|  |  |
| --- | --- |
| **Kiểu trả về** | **Phương thức** |
| ContentResult | Content() |
| EmptyResult | new EmptyResult() |
| FileResult | File() |
| ForbidResult | Forbid() |
| JsonResult | Json() |
| LocalRedirectResult | LocalRedirect() |
| RedirectResult | Redirect() |
| RedirectToActionResult | RedirectToAction() |
| RedirectToPageResult | RedirectToRoute() |
| RedirectToRouteResult | RedirectToPage() |
| PartialViewResult | PartialView() |
| ViewComponentResult | ViewComponent() |
| StatusCodeResult | StatusCode() |
| ViewResult | View() |

**Action trả về ContentResult**

Lớp *ContentResult* là mở rộng của *ActionResult*, nó cho phép bạn tạo ra nội dung tài liệu trả về trong Action. Để tạo ra đối tượng *ContentResult* cần khởi tạo nó với ba tham số gồm: nội dung tài liệu, kiểu tài liệu và encoding của tài liệu.

Ví dụ, trả về một nội dung văn bản:

|  |
| --- |
| /...  **public IActionResult GetContent** () {  // mime - tham khảo https://en.wikipedia.org/wiki/Media\_type  **string** contentType = "text/plain";  **string** content = "Đây là nội dung file văn bản";  **return** **Content** (content, contentType, Encoding.UTF8);  }  /... |

**Action trả về JsonResult**

Phương thức Json của *Controller* trích xuất các thuộc tính của đối tượng và chuyển thành Json, Response trả về có header Content-Type *application/json*

|  |
| --- |
| /...  **public IActionResult** **GetJson**()  {  **return Json**(  **new** {  key1 = 100,  key2 = **new string**[] {"a", "b", "c"}  }  );  }  /... |

Truy cập địa chỉ */Home/GetJson* kết quả

|  |
| --- |
| {"key1":100,"key2":["a","b","c"]} |

**Chuyển hướng truy cập**

Có một số phương thức để chuyển hướng, thường có tiền tố Redirect, ví dụ vài phương thức như:

|  |
| --- |
| **public IActionResult TestRedirect**()  {  // Redirect 302 - chuyển hướng sang URL khác  **return Redirect**("https://github.com");    // Redirect 302 - Found  // return RedirectToRoute(new {controller="Home", action="Index"});    // Redirect 301 - chuyển hướng URL khác  // return RedirectPermanent("https://xuanthulab.net");  // Chuyển hướng sang Action khác  // return RedirectToAction("Index");  } |

**Trả về nội dung file**

Phương thức File, tùy ngữ cảnh nó trả về nội dung file.

|  |
| --- |
| /...  **public IActionResult FileAnh**()  {  **string** filepath = "logo.png";  **return** **File**(filepath, "image/png");  }  /... |

**Trả về 404 – NotFound**

|  |
| --- |
| /...  **public IActionResult KhongCoGi**()  {  **return NotFound**(  "Trang không thấy"  );  }  /... |

**Sử dụng tham số cho Action**

Có thể khai báo các Action có tham số, các tham số này được truyền thiết lập qua biến trong Route hay trong Url Query

|  |
| --- |
| /...  **public IActionResult Sum**(int x, int y)  {  **return Content**((x+y).ToString(), "text/plain", Encoding.UTF8);  }  /... |

Truy cập */Home/Sum?x=10&y=100* trả về nội dung 110

Trong lớp Controller còn có các thuộc tính truy cập nhiều loại thông tin rất đa dạng, những thuộc tính này sẽ tìm hiểu dần ở các phần sau như:

* HttpContext - thuộc tính này giúp lấy được ServiceProvider của ứng dụng.
* Request
* Response
* RouteData - dữ liệu Route
* ViewBag và ViewData - giúp chuyển dữ liệu đến View
* TempData - giúp chia sẻ dữ liệu giữa các phiên
* User

## 8.5. Tạo View trong ASP.NET MVC - Razor View Engine

View kết hợp với dữ liệu nó nhận được, phát sinh nội dung HTML trả về cho client. Các View (file .cshtml) được bố trí trong thư mục Views, nó là các template - là các file có phần mở rộng .cshtml, trong đó nó chứa mã HTML, đồng thời nhúng mã C# (server side script), những đoạn nhúng mã server side thì bắt đầu bằng ký hiệu @

View là các file .cshtml sử dụng cú pháp Razor, thực tế trong ứng dụng mô hình MVC có áp dụng lại cả các trang Razor Page hoàn chỉnh, nên bạn cần hiểu rõ về Razor Page sẽ được trình bày ở chương XVI.

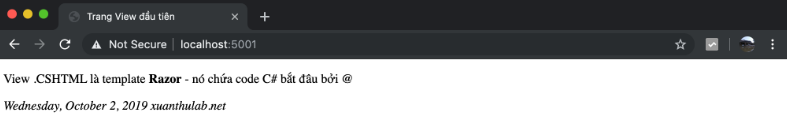
Khi một Action trả về đối tượng ViewResult (phương thức View của Controller tạo ra loại đối tượng này). Thì nó sẽ sử dụng View .cshtml và Razor làm việc để tạo nội dung HTML. Nó sẽ tìm đến thư mục Views, tìm đến thư mục cùng tên Controller ví dụ Home, sau đó là file .cshtml cùng tên với Action - file .cshtml tìm được sẽ sinh HTML.

Nếu vậy:

|  |
| --- |
| **public class HomeController** : **Controller**  {  /...  **public IActionResult Index**()  {  **return View**(); // Trả về ViewResult  }  /...  } |

Action trên sẽ sử dụng View tương ứng là *Views/Home/Index.cshtml*, hãy biên tập nội dung file thành:

|  |
| --- |
| @{  Layout = null;  }  <!**doctype** **html**>  <**html**>  <**head**>  <**title**>Trang View đầu tiên</**title**>  </**head**>  <**body**>  <**p**>View .CSHTML là template <**strong**>Razor</**strong**> - nó chứa code C# bắt đầu bởi @@</**p**>  <**p**><**i**>@DateTime.Now.ToLongDateString() xuanthulab.net </**i**></**p**>  </**body**>  </**html**> |



Ảnh 9: Trang View đầu tiên

Để chèn một dòng code server side, thì sử dụng @ sau đó là dòng code, Razor tự nhận biết được đoạn kết thúc hết code khi phát hiện thẻ HTML, như ví dụ trên

|  |
| --- |
| <p><i>@DateTime.Now.ToLongDateString() xuanthulab.net</i></p> |

Để mở khối thì sử dụng thêm { ... }

|  |
| --- |
| @{  //code C#  } |

**Thêm thư mục lưu trữ các View .cshtml**

Các View cho các Action mặc định lưu theo cấu trúc thư mục */View/ControllerName/ActionName.cshtml*, nếu muốn thay đổi hoặc thêm cấu trúc mới thì thiết lập cho RazorEngine ở *Startup.ConfigureService*, ví dụ:

|  |
| --- |
| // {0} - Action Name  // {1} - Controller Name  // {2} - Area Name  // RazorViewEngine.ViewExtension - .cshtml  **services.Configure**<**RazorViewEngineOptions**>(options => {  // Tìm thêm View ở /MyView/ControllerName/ActionName.cshtml  **options.ViewLocationFormats.Add**("/MyView/{1}/{0}" + **RazorViewEngine.ViewExtension**);  }); |

## 8.6. Truyền dữ liệu Controller tới View với ViewBag

Để truyền dữ liệu trong MVC, bạn có thể sử dụng:

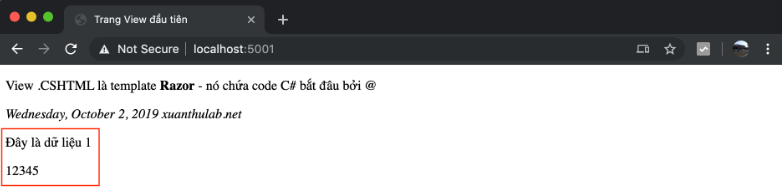
* Thiết lập Model cho View, khi gọi phương thức View của Controller. Trong View .cshtml sử dụng chỉ thị *@model* để thiết lập kiểu truyền tới cho Model của View
* Sử dụng property *ViewData* của View và Controller
* Sử dụng *ViewBag* của View và Controller
* Sử dụng TemData truyền dữ liệu giữa các trang (dùng cookie)

Để truyền dữ liệu từ Controller sang View, trong Controller có thể sử dụng thuộc tính ViewBag, đây là kiểu Dynamic, có thể thiết lập dữ liệu vào nó với cặp key/value

|  |
| --- |
| **public IActionResult Index()**  {  **ViewBag.dulieu1** = "Đây là dữ liệu 1"; //key=dulieu1, value = "Đây là dữ liệu 1"  **ViewBag.dulieu2** = 12345;  **return View**();  } |

Khi controller thiết lập dữ liệu qua ViewBag, thì ở View truy cập được thuộc tính này của Controller với ký hiệu @, đó là:

|  |
| --- |
| @{  Layout = null;  }  <!**doctype** **html**>  <**html**>  <**head**>  <**title**>Trang View đầu tiên</**title**>  </**head**>  <**body**>  <**p**>View .CSHTML là template <**strong**>Razor</**strong**> - nó chứa code C# bắt đầu bởi @@</**p**>  <**p**><**i**>@DateTime.Now.ToLongDateString()</**i**></**p**>  <**p**>@ViewBag.dulieu1</p>  <**p**>@ViewBag.dulieu2</p>  </**body**>  </**html**> |



Ảnh 10: Truyền dữ liệu Controller tới View với ViewBag

## 8.7. Truyền dữ liệu Controller tới View với Model

Trong Razor có một thuộc tính có tên Model thuộc tính này là một kiểu gereric (trong cú pháp ký hiệu nó là TModel, xem thêm Generic trong C# , TModel thực tế là kiểu gì được quyết định khi khởi tạo ViewResult).

Từ Controller, khi khởi tạo ViewResult với phương thức View, bạn có thể chỉ ra tham số chính là Model, ví dụ:

|  |
| --- |
| **return View**(ModelObject); // ModelObject là đối tượng bất kỳ - nó là Model trong Razor |

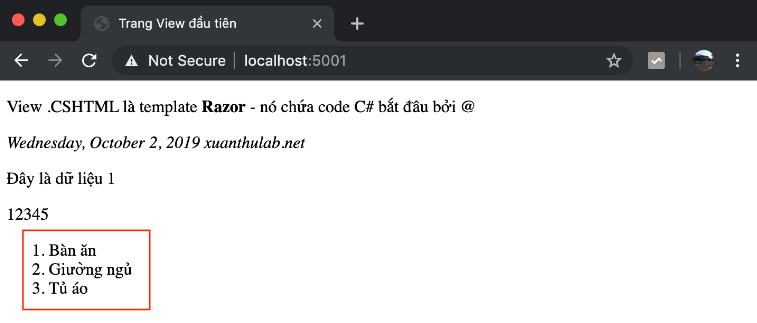
Lúc này, trong .phtml, Razor sẽ đọc được qua thuộc tính với tên Model - Ví dụ:

|  |
| --- |
| **public IActionResult Index**()  {  **ViewBag.dulieu1** = "Đây là dữ liệu 1";  **ViewBag.dulieu2** = 12345;  // Một danh sách tênn các sản phầm  **List<string>** sanpham = **new** **List**<**string**>()  {  "Bàn ăn",  "Giường ngủ",  "Tủ áo"  };  // sanpham gán vào Model tronng Razor  // Đã trả về ViewResult kèm dữ liệu  **return View**(sanpham);  } |

Lúc này ở View, Razor có thể truy cập được nó thông qua thuộc tính Model, nhưng ở phần đầu cần khai báo kiểu Model này với *@model List<string>*

|  |
| --- |
| **@model** **List**<**string**>    @{  Layout = null;  }  <!**doctype** **html**>  <**html**>  <**head**>  <**title**>Trang View đầu tiên</**title**>  </**head**>  <**body**>  <**p**>View .CSHTML là template <**strong**>Razor</**strong**> - nó chứa code C# bắt đầu bởi @@</**p**>  <**p**><**i**>**@DateTime.Now.ToLongDateString()** xuanthulab.net</**i**></**p**>  <**p>@ViewBag.dulieu1**</**p**>  <**p>@ViewBag.dulieu2**</**p**>  <**ol**>  **@foreach** (**var** sp in **Model**) {  <**li**>@sp</**li**>  }  </**ol**>  </**body**>  </**html**> |

*@model* là một chỉ thị của Razor, nó cho biết kiểu của Model, cú pháp khai báo là *@model Kiểu\_Của\_Model*



Ảnh 11: Truyền dữ liệu Controller tới View với Model

## 8.8. Phát sinh controller, view, razor page, area

*aspnet-codegenerator* là công cụ bổ sung vào lệnh dotnet nó trợ giúp phát sinh nhanh các cấu trúc thư mục chuẩn, tạo controller, view, razor page nhanh hơn, tránh những thao tác đơn giả bạn phải làm thủ công.

Đảm bảo cài đặt công cụ lệnh *dotnet aspnet-codegenerator*

|  |
| --- |
| dotnet tool install -g dotnet-aspnet-codegenerator  dotnet tool update -g dotnet-aspnet-codegenerator |

Thêm package

|  |
| --- |
| dotnet add package Microsoft.VisualStudio.Web.CodeGeneration.Design |

Gõ lệnh để xem hướng dẫn

|  |
| --- |
| dotnet aspnet-codegenerator -h |

**Tạo một Area**

Area là khu vực (thư mục) gom các Controller, Razor page thành nhóm. Để tạo ra cấu trúc thư mục cho một Area nào đó gõ lệnh:

|  |
| --- |
| dotnet aspnet-codegenerator NameArea |

Nó sẽ tạo thư mục *Areas/NameArea/* và trong nó là các thư mục con theo chuẩn như: Controller, View, Models...

**Tạo ra Controller**

Tạo một controller tên First mã nguồn lưu ở thư mục Controllers

|  |
| --- |
| dotnet aspnet-codegenerator controller -name First -outDir Controllers |

Nó sẽ tạo ra lớp FirstController trong thư mục Controllers

**Tạo View**

Tạo ra View đặt tên Index, mẫu tạo là Empty, kết quả lưu tại Views/First sử dụng layout là \_Layout

|  |
| --- |
| dotnet aspnet-codegenerator view Index Empty -outDir Views/First -l \_Layout -f |

# Chương IX. Cơ bản về ASP.NET Core Controller

## 9.1. 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*

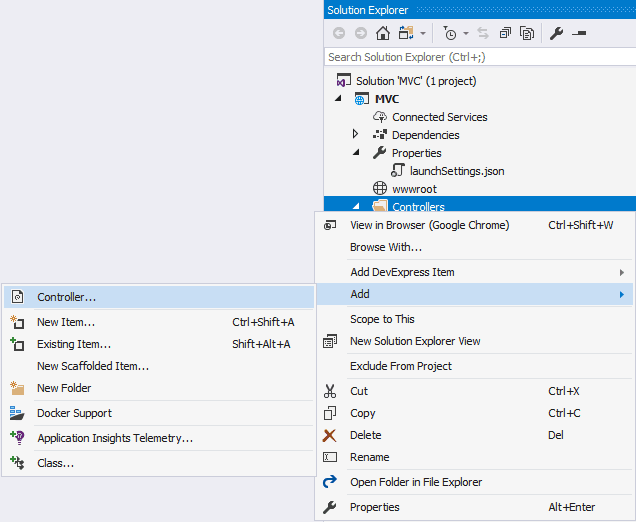
Trường hợp này, Controller có tên là CustomerController được gọi. Sau đó nó sẽ gọi đến Action metod tên List rồi tạo ra response trả về cho user.

Controller có 3 trách nhiệm chính:

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

## 9.2. 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.



Ảnh 12: Thêm Controller

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

* MVC Controller
* API Controller

A screenshot of a computer

Description automatically generated

Ảnh 13: Add Scaffold

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.

A screenshot of a computer

Description automatically generated

Ảnh 14: Add new 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 chìa ra ngoài bởi Controller đều được gọi là Action method. Action method get dduocj gọi khi user gõ một URL trên trình duyệt.

Ví dụ: http://localhost/Customer/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]
* Ation 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/Customer/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.

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 ddeuf 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.

# Chương X. 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.

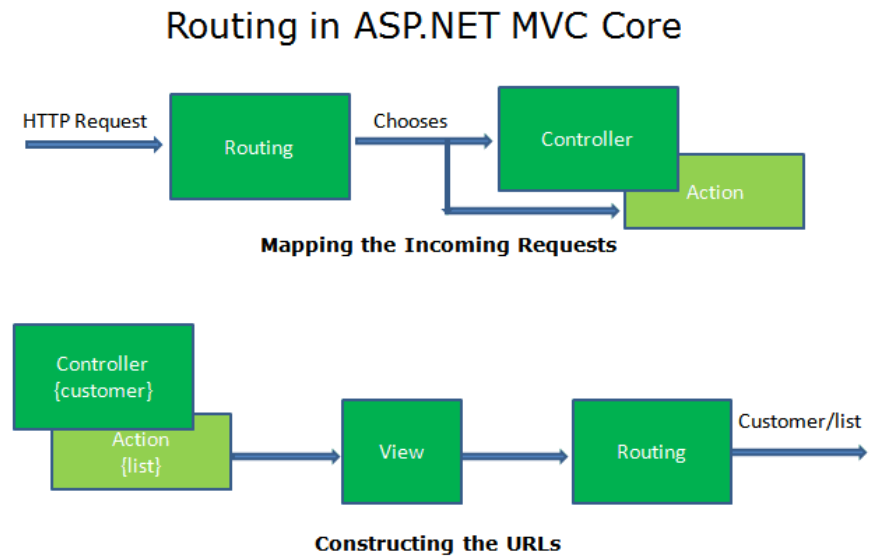
## 10.1. Một vài định nghĩa

### 10.1.1. Routing

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.



Ảnh 15: Routing trong ASP.NET MVC Core

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

A diagram of a computer program

Description automatically generated

Ảnh 16: Routing làm việc như thế nào?

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

### 10.1.2. Route

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*.

### 10.1.3. Route Collection

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*.

### 10.1.4. Route Handler

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*.

### 10.1.5. 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.

### 10.1.6. Các cách cài đặt Route

Có 2 cách khách nhau để cài đặt route:

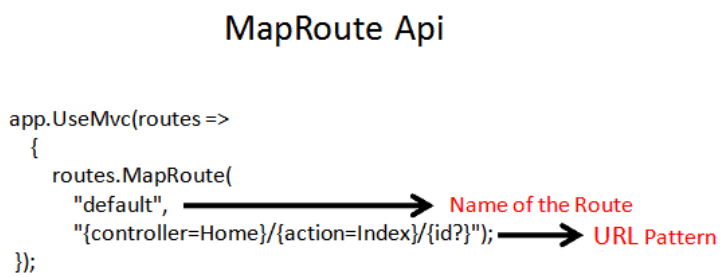
1. Convention-based routing: Convention based routing tạo ra Route dựa trên một loạt các quy tắc được định nghĩa trong file *Startup.cs*
2. Attribute routing:Tạo các Route dựa trên các attribute đặt trong Controller action. Hai hệ thống routing này có thể cùng tồn tại trong một hệ thống.

## 10.2. 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*:



Ảnh 17: MapRoute API

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 gọc 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.

A diagram of a diagram

Description automatically generated

Ảnh 18: URL Pattern

**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.

### Cài đặt Convention based routing [Phụ lục 5]

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

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

|  |
| --- |
| **public void ConfigureServices**(**IServiceCollection** services){  **services.AddMvc**();  } |

(3) 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.

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

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

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

(6) 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.

(7) Đăng ký Route

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

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

Đây là Route mặc định được đăng ký khi chúng ta sử dụng *app.UseMvcWithDefaultRoute*.

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.

(8) 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 route 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.

(9) 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";  } |

(10) Giờ chúng ta hãy thử chạy các URL sau: */, /Secure, /Secure/Test, /Admin*

**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.

## 10.3. Attribute Routing

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.

### Cài đặt Attribute Routing [Phụ lục 5]

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 trỏ URL */Home* đến method này.

(1) Mở project được tạo từ bài trước. Giờ 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**();  } |

(2) 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.

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

**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";  } |

**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 dược 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 gắn 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.

**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";  } |

# Chương XI. Route Constrains

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.

## 11.1. 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*.

### 11.1.1. Inline Constraint

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

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

Một khi Routeing 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.

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();  } |

### 11.1.2. Sử dụng tham số 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 nặc danh. Nó có thuộc tính id, match với id trong URL Parameter. Thể hiện của *IntRouteConstraint* được gán cho thuộc tính *id*.

## 11.2. Sử dụng Constraints

Constrains 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 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* Ation method nếu giá trị là số và *PostByPostName* nếu giá trị *id* là kiểu chữ cái.

**Kết hợp các constraints**

Nhiều constrains 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)  }) |

## 11.3. 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.

### 11.3.1. Constraints để kiểm tra kiểu dữ liệu

Đây là các constraints để kiểm tra kiểu dữ liệu:

|  |  |  |  |
| --- | --- | --- | --- |
| Constraints | 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 |
| **guid** | {id:guid} | GuidRouteConstraint | Chỉ cho phép kiểu Guid |

### 11.3.2. Constraints để kiểm tra giá trị/ miền giá trị và độ dài

|  |  |  |  |
| --- | --- | --- | --- |
| Constraints | 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 nhỏhơn hoặc bằng giá trị trong ngoặc |

Constraints kiểm tra sử dụng Regular Expression

|  |  |  |  |
| --- | --- | --- | --- |
| Constraints | Cú pháp | Class | Ghi chú |
| Regex  (expression) | {ssn:regex(^\\d{{3}}-\\d{{2}}-\\d{{4}}$)} | RegexRouteConstraint | Chứa giá trị thỏa mãn regular expression. |

# Chương XII. Action Selectors & Action Verbs

Chúng ta đã học về Routing Engine để chọn controller và action tướng 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.

## 12.1. 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

### 12.1.1. 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* thay vào đó 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";  }  } |

### 12.1.2. 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";  }  } |

### 12.1.3. Action Verbs

Action verbs selector được sử dụng khi bạn muốn điều kiển 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 ASP.NET Core: *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 client 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";  } |

**Danh sách HTTP Attributes**

|  |  |
| --- | --- |
| HTTP Attributes | Mô tả |
| HttpGet | HttpGet attribute giới hạn Action method chỉ cấp 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. |
| HttpPost | HttpPost attribute giới hạn action method chấp nhận HTTP Request sử dụng Post verb. Post verb được dùng để tạo mới bản ghi. |
| 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 serrver 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 | HttPatch 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();  } |

# Chương XIII. Action Result

## 13.1. 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 Controller 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.

**IActionResult và ActionResult**

*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 trự 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 trả 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.

## 13.2. Sử dụng Action Result

*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*.

## 13.3. Các loại Action Result

### 13.3.1. Trả về HTML

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

#### 13.3.1.1. 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*.

#### 13.3.1.2. 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) khi bạn muốn cập nhật một phần của View thông qua AJAX.

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

### 13.3.2. Chuyển hướng người dùng

Redriect 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 tạp thời)
* 301 Moved Permanently
* 307 Temporary Redirect
* 308 Permanent Redirect

#### 13.3.2.1. RedirectResult

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

|  |  |
| --- | --- |
| Controller Method | Status code |
| 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); |

#### 13.3.2.2. 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 tất công open redirect attack.

|  |  |
| --- | --- |
| Controller Method | Status code |
| 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"); |

#### 13.3.2.3. 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:

|  |  |
| --- | --- |
| Controller Method | Status code |
| RedirectToAction | 302 Found (Temporarily moved) |
| RedirectToActionPermanent | 301 Moved Permanently |
| RedirectToActionPermanentPreserveMethod | 308 Permanent Redirect |
| RedirectToActionPreserveMethod | 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); |

### 13.3.3. FileResult

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

#### 13.3.3.1. FileContentResult

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

|  |
| --- |
| **return new FileContentResult**(byteArray, "application/pdf") |

#### 13.3.3.2. FileStreamResult

FileStreamResult đọc một luồng stream và trả về một file:

|  |
| --- |
| **return new FileStreamResult**(fileStream, "application/pdf"); |

#### 13.3.3.3. 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"); |

#### 13.3.3.4. 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 dẫ phải là đường dẫn tuyệt đối:

|  |
| --- |
| **return new PhysicalFileResult**("c:/path/filename", "application/pdf"); |

### 13.3.4. Content Result

#### 13.3.4.1. 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);  } |

#### 13.3.4.2. 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");  }  or  **public ContentResult About**(){  **return new ContentResult**() { Content = "Hello World" };  } |

#### 13.3.4.3. 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**(); |

### 13.3.5. 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.

**13.3.5.1. StatusCodeResult**

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

|  |
| --- |
| **return StatusCode**(200);  hoặc  **return new StatusCodeResult**(200); |

**13.3.5.2. 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" }); |

**13.3.5.3. OkResult**

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

|  |
| --- |
| **return Ok**();  hoặc  **return new OkResult**(); |

**13.3.5.4. OkObjectResult**

Action result này trả về một HTTP Status code 200:

|  |
| --- |
| **return Ok**(**new** {message="Hello"});  hoặc  **return new OkObjectResult**(**new** { message = "Not found" }); |

**13.3.5.5. 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" }); |

**13.3.5.6. CreatedAtActionResult**

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

|  |
| --- |
| **return CreatedAtAction**("Index", **new** {message="Hello World"}); |

**13.3.5.7. CreateAtRouteResult**

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

|  |
| --- |
| **CreatedAtRoute**("default", **new** { mesage = "Hello World" }); |

**13.3.5.8. 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**() |

**13.3.5.9. 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*

**13.3.5.10. NotFoundResult**

Action result này trả về lỗi HTTP 404 cho client:

|  |
| --- |
| **return NotFound**(); |

**13.3.5.11. NotFoundObjectResult**

Action result này tương tự nuhw NotFoundResult nhưng trả về một đuối 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" }); |

**13.3.5.12. 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**(); |

**13.3.5.13. 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**(); |

# Chương XIV. View

# Chương XV. Razor View Engine trong ASP.NET Core MVC

# Chương XVI. Sử dụng Layouts và Section

# Chương XVII. ViewBag và ViewData

# Chương XVIII. Model và ViewModel

# Chương XIX. Truyền dữ liệu từ Controller sang View

# Chương XX. Xây dựng HTML Form

# Chương XXI. Strongly Typed View

# Chương XXII. Tag Helpers trong ASP.NET Core MVC

# Chương XXIII. Input Tag Helper

# Chương XIV. Environment Tag Helper

# Chương XXV. Cơ chế Model Binding: Truyền dữ liệu từ View lên Controller

# Chương XXVI. Model Validation

# Chương XXVII. Validation Tag Helper

# Chương XXVIII. Unobtrusive Client Validation

# Chương XXIX. Cơ chế Dependency Injection

# Chương XXX. Vòng đời của Dependency Injection: Transient, Singleton và Scoped

# TÀI LIỆU THAM KHẢO

[1] Học ASP.NET Core căn bản: <https://tedu.com.vn/series/hoc-aspnet-core-can-ban.html>

[2] ASP.NET Core là gì? <https://topdev.vn/blog/asp-net-core-la-gi/>

[3] C# cơ bản .NET Core: <https://xuanthulab.net/lap-trinh-c-co-ban/>

# PHỤ LỤC

**1. FirstWebApp Project:** Github: