**TRƯỜNG ĐẠI HỌC SÀI GÒN**

**KHOA CÔNG NGHỆ THÔNG TIN**

----- 🙡 🕮 🙣 -----



**ĐỒ ÁN MÔN HỌC:**

**CÁC CÔNG NGHỆ LẬP TRÌNH HIỆN ĐẠI**

**ĐỀ TÀI : ASP .NET CORE**

**Giáo viên hướng dẫn** : Phạm Thi Vương

**Nhóm sinh viên thực hiện**

3120410586 - Nguyễn Anh Tuấn

3120410264 - Vũ Trung Kiên

3120410005 - Phạm Quang Minh

Thành phố Hồ Chí Minh, 2024

# LỜI CẢM ƠN

Em xin bày tỏ lòng biết ơn sâu sắc tới thầy Phạm Thi Vương, người đã tận tình hướng dẫn và hỗ trợ em trong suốt quá trình thực hiện đề tài nghiên cứu về ASP.NET Core trong môn Các công nghệ lập trình hiện đại. Nhờ sự chỉ dẫn và tâm huyết của thầy, em không chỉ nắm vững kiến thức về ASP.NET Core mà còn hiểu rõ hơn về cách áp dụng công nghệ vào thực tế, qua đó phát triển kỹ năng và tư duy lập trình.

Bên cạnh đó, em xin chân thành cảm ơn Khoa Công nghệ Thông tin, Trường Đại học Sài Gòn, đã tạo điều kiện thuận lợi để em được học tập và nghiên cứu môn học này, mở ra cơ hội thực hiện đề tài có ý nghĩa và hữu ích cho sự nghiệp tương lai.

Em hy vọng sẽ tiếp tục nhận được sự quan tâm, chỉ dẫn của thầy và nhà trường trong các chặng đường học tập và phát triển sắp tới.

# MỤC LỤC

[LỜI CẢM ƠN 1](#_Toc182564484)

[MỤC LỤC 2](#_Toc182564485)

[DANH MỤC ẢNH 11](#_Toc182564486)

[DANH MỤC VIẾT TẮT 12](#_Toc182564487)

[Chương I. Giới thiệu 13](#_Toc182564488)

[1.1. Giới thiệu ASP.NET Core 13](#_Toc182564489)

[1.2. Lý do chọn đề tài 14](#_Toc182564490)

[Chương II. Tổng quan về ASP.NET Core 16](#_Toc182564491)

[2.1. ASP.NET Core là gì? 16](#_Toc182564492)

[2.1.1. ASP.NET Core là gì? 16](#_Toc182564493)

[2.1.2. .NET Core là gì? 16](#_Toc182564494)

[2.1.3. .NET Standard là gì? 17](#_Toc182564495)

[2.1.4. Sự khác nhau giữa .NET Core và .NET Framework 17](#_Toc182564496)

[2.2. Lịch sử phát triển của ASP.NET Core 17](#_Toc182564497)

[2.3. Các thành phần chính của ASP.NET 18](#_Toc182564498)

[2.4. So sánh ASP.NET với PHP 19](#_Toc182564499)

[2.5. Ứng dụng thực tế của ASP.NET 20](#_Toc182564500)

[2.6. Cấu trúc dự án ASP.NET Core 20](#_Toc182564501)

[2.6.1. Program.cs 20](#_Toc182564502)

[2.6.2. Startup.cs 21](#_Toc182564503)

[2.6.3. appsettings.json 21](#_Toc182564504)

[2.6.4. Controllers 21](#_Toc182564505)

[2.6.5. Views 22](#_Toc182564506)

[2.6.6. Models 22](#_Toc182564507)

[2.7. Xây dựng ứng dụng đầu tiên 22](#_Toc182564508)

[2.7.1. Cài đặt môi trường phát triển 22](#_Toc182564509)

[2.7.2. Tạo dự án ASP.NET Core Web Application (Phụ lục 1) 26](#_Toc182564510)

[Chương III. Kestrel – ASP.NET Core Web Server 29](#_Toc182564511)

[3.1. Kestrel là gì? 29](#_Toc182564512)

[3.2. Tại sao sử dụng Kestrel 29](#_Toc182564513)

[3.3. Sử dụng Kestrel 29](#_Toc182564514)

[3.4. Triển khai ứng dụng ASP.NET trên Server Linux với Kestrel Apache Nginx 31](#_Toc182564515)

[3.4.1. Cấu hình Kestrel trong ASP.NET core 31](#_Toc182564516)

[3.4.2. Publish ứng dụng ASP.NET core 33](#_Toc182564517)

[3.4.3. Giám sát ứng dụng ASP.NET với systemd 34](#_Toc182564518)

[Chương IV. Middleware và Request Pipeline trong ASP.NET Core [Phụ lục 2] 36](#_Toc182564519)

[4.1. Middleware là gì? 36](#_Toc182564520)

[4.2. Request Pipeline 36](#_Toc182564521)

[4.3. Tạo Middleware riêng trong ASP.NET Core 40](#_Toc182564522)

[4.4. Xây dựng Middleware triển khai từ IMiddleware 44](#_Toc182564523)

[4.4. Truyền dữ liệu giữa các Middleware 46](#_Toc182564524)

[Chương V. Làm việc với file cấu hình trong ASP.NET Core 47](#_Toc182564525)

[5.1. Cấu hình là gì 47](#_Toc182564526)

[5.2. Cấu hình trong ASP.NET Core 47](#_Toc182564527)

[5.3. Nguồn cấu hình 48](#_Toc182564528)

[5.4. Load cấu hình 48](#_Toc182564529)

[5.5. Đọc cấu hình 50](#_Toc182564530)

[5.6. Thêm file cấu hình theo ý muốn 52](#_Toc182564531)

[Chương VI. Sử dụng Static Files trong ASP.NET Core 53](#_Toc182564532)

[6.1. Sử dụng Static Files 53](#_Toc182564533)

[6.2. Tạo mới Static file 54](#_Toc182564534)

[Chương VII. MVC Design Pattern trong ASP.NET Core 56](#_Toc182564535)

[7.1. MVC Design Pattern trong ASP.NET Core 56](#_Toc182564536)

[7.2. Chia để trị (Separation of concerns) 56](#_Toc182564537)

[7.3. MVC Pattern làm việc trong ASP.NET Core ra sao? 58](#_Toc182564538)

[Chương VIII. Xây dựng ứng dụng ASP.NET Core MVC đầu tiên 60](#_Toc182564539)

[8.1. Tạo ứng dụng ASP.NET MVC đầu tiên 60](#_Toc182564540)

[8.2. Truy cập ứng dụng với HTTPS 62](#_Toc182564541)

[8.3. Route - Định tuyển trong ASP.NET MVC 62](#_Toc182564542)

[8.4. Tạo Controller trong ASP.NET MVC 64](#_Toc182564543)

[8.5. Tạo View trong ASP.NET MVC - Razor View Engine 68](#_Toc182564544)

[8.6. Truyền dữ liệu Controller tới View với ViewBag 70](#_Toc182564545)

[8.7. Truyền dữ liệu Controller tới View với Model 71](#_Toc182564546)

[8.8. Phát sinh controller, view, razor page, area 73](#_Toc182564547)

[Chương IX. Cơ bản về ASP.NET Core Controller 75](#_Toc182564548)

[9.1. Controller là gì? 75](#_Toc182564549)

[9.2. Thêm mới controller 75](#_Toc182564550)

[Chương X. Routing trong ASP.NET Core MVC 81](#_Toc182564551)

[10.1. Một vài định nghĩa 81](#_Toc182564552)

[10.1.1. Routing 81](#_Toc182564553)

[10.1.2. Route 82](#_Toc182564554)

[10.1.3. Route Collection 83](#_Toc182564555)

[10.1.4. Route Handler 83](#_Toc182564556)

[10.1.5. MVCRouteHandler 83](#_Toc182564557)

[10.1.6. Các cách cài đặt Route 83](#_Toc182564558)

[10.2. Routing mặc định của dự án MVC 84](#_Toc182564559)

[10.3. Routing trong ASP.NET Core MVC 85](#_Toc182564560)

[10.3.1. Conventional routing 85](#_Toc182564561)

[10.3.2. Attribute routing 86](#_Toc182564562)

[10.3.3. Route template 86](#_Toc182564563)

[10.3.4. Cấu trúc route template 86](#_Toc182564564)

[10.4. Conventional Routing 88](#_Toc182564565)

[10.5. Attribute Routing 90](#_Toc182564566)

[Chương XI. Route Constrains 96](#_Toc182564567)

[11.1. Route Constrains làm việc như thế nào? 96](#_Toc182564568)

[11.1.1. Inline Constraint 96](#_Toc182564569)

[11.1.2. Sử dụng tham số Constraint của MapRoute 96](#_Toc182564570)

[11.2. Sử dụng Constraints 97](#_Toc182564571)

[11.3. Danh sách của Route Constraint 98](#_Toc182564572)

[11.3.1. Constraints để kiểm tra kiểu dữ liệu 98](#_Toc182564573)

[11.3.2. Constraints để kiểm tra giá trị/ miền giá trị và độ dài 99](#_Toc182564574)

[Chương XII. Action Selectors & Action Verbs 100](#_Toc182564575)

[12.1. Action Selector là gì? 100](#_Toc182564576)

[12.1.1. Action Name 100](#_Toc182564577)

[12.1.2. Non Action 101](#_Toc182564578)

[12.1.3. Action Verbs 101](#_Toc182564579)

[Chương XIII. Action Result 104](#_Toc182564580)

[13.1. Action Result là gì? 104](#_Toc182564581)

[13.2. Sử dụng Action Result 105](#_Toc182564582)

[13.3. Các loại Action Result 107](#_Toc182564583)

[13.3.1. Trả về HTML 107](#_Toc182564584)

[13.3.2. Chuyển hướng người dùng 107](#_Toc182564585)

[13.3.3. FileResult 109](#_Toc182564586)

[13.3.4. Content Result 110](#_Toc182564587)

[13.3.5. Trả về lỗi và HTTP Code 111](#_Toc182564588)

[13.3.6. Kết quả liên quan đến bảo mật 113](#_Toc182564589)

[Chương XIV. View 115](#_Toc182564590)

[14.1. Định nghĩa? 115](#_Toc182564591)

[14.1.1. View là gì? 115](#_Toc182564592)

[14.1.2. Trách nhiệm của View 115](#_Toc182564593)

[14.1.3. Thực hành tạo View? [Phụ lục 3] 115](#_Toc182564594)

[14.2. ASP.NET Core tìm View như thế nào? 118](#_Toc182564595)

[Chương XV. Razor View Engine trong ASP.NET Core MVC 121](#_Toc182564596)

[15.1. Định nghĩa 121](#_Toc182564597)

[15.2. Cú pháp Razor 121](#_Toc182564598)

[15.2.1. Khối lệnh Razor 122](#_Toc182564599)

[15.2.2. Biểu thức Razor 122](#_Toc182564600)

[15.1.3. Hỗ trợ nhận diện cú pháp 123](#_Toc182564601)

[15.2.4. Các cú pháp cơ bản 124](#_Toc182564602)

[Chương XVI. Sử dụng Layouts và Section [Phụ lục 4] 129](#_Toc182564603)

[16.1. Layout của trang web 129](#_Toc182564604)

[16.2. Layout trong ASP.NET Core là gì? 130](#_Toc182564605)

[16.3. Layout 130](#_Toc182564606)

[16.4. Section 133](#_Toc182564607)

[Chương XVII. ViewBag và ViewData 136](#_Toc182564608)

[17.1. ViewData 136](#_Toc182564609)

[17.1.1. ViewData là gì? 136](#_Toc182564610)

[17.1.2. Sử dụng ViewData như thế nào? 136](#_Toc182564611)

[17.2. ViewBag 138](#_Toc182564612)

[17.2.1. ViewBag là gì? 138](#_Toc182564613)

[17.2.2. Sử dụng ViewBag 138](#_Toc182564614)

[17.3. So sánh ViewData và ViewBag 138](#_Toc182564615)

[Chương XVIII. Model và ViewModel 140](#_Toc182564616)

[18.1. Model là gì? 140](#_Toc182564617)

[18.1.1. Domain Model 140](#_Toc182564618)

[18.1.2. View Model 141](#_Toc182564619)

[18.1.3. Edit Model 142](#_Toc182564620)

[18.2. Model trong MVC Design Pattern 143](#_Toc182564621)

[18.3. Best practice với ViewModel 143](#_Toc182564622)

[18.4. Truyền dữ liệu từ Controller sang View 144](#_Toc182564623)

[Chương XIX. Xây dựng HTML Form [Phụ lục 5] 149](#_Toc182564624)

[19.1. Tạo mới project 149](#_Toc182564625)

[19.2. Cài đặt MVC Middleware 149](#_Toc182564626)

[19.3. Tạo một View model 149](#_Toc182564627)

[19.4. Tạo một Controller 150](#_Toc182564628)

[19.5. Tạo một View 151](#_Toc182564629)

[19.6. Index View 153](#_Toc182564630)

[Chương XX. Tag Helpers trong ASP.NET Core MVC 155](#_Toc182564631)

[20.1. Tag Helper là gì? 155](#_Toc182564632)

[20.2. Sử dụng Tag Helper? 155](#_Toc182564633)

[20.2.1. Form Tag Helper 157](#_Toc182564634)

[20.2.2. Label Tag Helper 158](#_Toc182564635)

[20.2.3. Input Tag Helper 158](#_Toc182564636)

[20.3. Lợi ích của Tag Helper 161](#_Toc182564637)

[Chương XXI. Cơ chế Model Binding: Truyền dữ liệu từ View lên Controller 164](#_Toc182564638)

[21.1. Model Binding là gì? 164](#_Toc182564639)

[21.2. Lấy dữ liệu từ Form Data trong Controller 164](#_Toc182564640)

[21.3. Cơ chế Model Binding? 165](#_Toc182564641)

[21.4. Khi không dùng model binding 168](#_Toc182564642)

[21.5. Các nguồn cho Model binding 169](#_Toc182564643)

[21.6. Điều khiển Binding Source 171](#_Toc182564644)

[21.6.1. [FromForm] 171](#_Toc182564645)

[21.6.2. [FromRoute] 171](#_Toc182564646)

[21.6.3. Binding query string sử dụng [FromQuery] 173](#_Toc182564647)

[21.6.4. Binding đến Request body sử dụng [FromBody] 173](#_Toc182564648)

[21.6.5. Binding từ Request Header sử dụng [FromHeader] 173](#_Toc182564649)

[21.6.6. Vô hiệu hoá binding với [BindNever] 173](#_Toc182564650)

[21.6.7. Bắt buộc Binding với [BindRequuired] 174](#_Toc182564651)

[Chương XXII. Validation 175](#_Toc182564652)

[22.1. Giới thiệu về Model Validation 175](#_Toc182564653)

[22.2. Model Validation làm việc ra sao? 176](#_Toc182564654)

[22.3. Cách sử dụng Validation Attributes 178](#_Toc182564655)

[22.3.1. Cập nhật Model với Data Annotaion 178](#_Toc182564656)

[22.3.2. Hiển thị danh sách lỗi Validations 178](#_Toc182564657)

[22.3.3. Kiểm tra ModelState.IsValid trong Controller Action 179](#_Toc182564658)

[22.4. Model State 180](#_Toc182564659)

[22.5. Validation Attributes 181](#_Toc182564660)

[22.6. Validation Tag Helper 184](#_Toc182564661)

[22.6.1. Validation Message Tag Helper 184](#_Toc182564662)

[22.6.2. Validation Summary Tag Helper 185](#_Toc182564663)

[22.7. Unobtrusive client-side validation 187](#_Toc182564664)

[22.7.1. Cách sử dụng unobtrusive client-side validation 187](#_Toc182564665)

[22.7.2. Chúng làm việc ra sao? 189](#_Toc182564666)

[Chương XXIII. Cơ chế Dependency Injection 190](#_Toc182564667)

[23.1. Giới thiệu về Dependency Injection 190](#_Toc182564668)

[23.2. Depedency Injection là gì? 191](#_Toc182564669)

[23.3. DI Container 191](#_Toc182564670)

[23.4. Ví dụ về hệ thống Dependency Injection [Phụ lục 6] 192](#_Toc182564671)

[23.5. Constructor Injection và Action Injection 197](#_Toc182564672)

[23.5.1. Constructor Injection 197](#_Toc182564673)

[23.5.2. Action Injection 197](#_Toc182564674)

[23.6. Lợi ích của Dependency Injection 198](#_Toc182564675)

[23.7. Vòng đời của Dependency Injection: Transient, Singleton và Scoped 198](#_Toc182564676)

[23.7.1. Quản lý vòng đời (lifetime) Service 198](#_Toc182564677)

[23.7.2. Project ví dụ [Phụ lục 7] 199](#_Toc182564678)

[TÀI LIỆU THAM KHẢO 208](#_Toc182564679)

[PHỤ LỤC 209](#_Toc182564680)

[Phụ lục 1: First web app 209](#_Toc182564681)

[Phụ lục 2: Middleware 209](#_Toc182564682)

[Phụ lục 3: 209](#_Toc182564683)

# DANH MỤC ẢNH

[Ảnh 1: Tải file cài đặt Visual Studio Community 24](#_Toc182564707)

[Ảnh 2: Thay đổi vị trí lưu Visual Studio 25](#_Toc182564708)

[Ảnh 3: Chọn template project 27](#_Toc182564709)

[Ảnh 4: Tạo project mới 28](#_Toc182564710)

[Ảnh 5: Web mẫu được tạo sẵn sau khi tạo project 28](#_Toc182564711)

[Ảnh 6: Sơ đồ các middleware trong pipeline 38](#_Toc182564712)

[Ảnh 7: MVC Design Pattern 57](#_Toc182564713)

[Ảnh 8: Khởi tạo dự án MVC 62](#_Toc182564714)

[Ảnh 9: Trang View đầu tiên 70](#_Toc182564715)

[Ảnh 10: Truyền dữ liệu Controller tới View với ViewBag 72](#_Toc182564716)

[Ảnh 11: Truyền dữ liệu Controller tới View với Model 74](#_Toc182564717)

[Ảnh 12: Thêm Controller 77](#_Toc182564718)

[Ảnh 13: Add Scaffold 78](#_Toc182564719)

[Ảnh 14: Add new controller 79](#_Toc182564720)

[Ảnh 15: Routing trong ASP.NET MVC Core 82](#_Toc182564721)

[Ảnh 16: Routing làm việc như thế nào? 83](#_Toc182564722)

[Ảnh 19: Cấu trúc website cơ bản 130](#_Toc182564723)

[Ảnh 20: Thêm \_Layout 131](#_Toc182564724)

[Ảnh 21: Index view có sử dụng \_Layout 133](#_Toc182564725)

[Ảnh 22: Cơ chế làm việc của Model binding 167](#_Toc182564726)

[Ảnh 23: Model Binding lấy dữ liệu từ form gửi đến Controller 168](#_Toc182564727)

[Ảnh 24: Model Validation 178](#_Toc182564728)

[Ảnh 25: Cách ProductService được inject vào HomeController 196](#_Toc182564729)

# 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

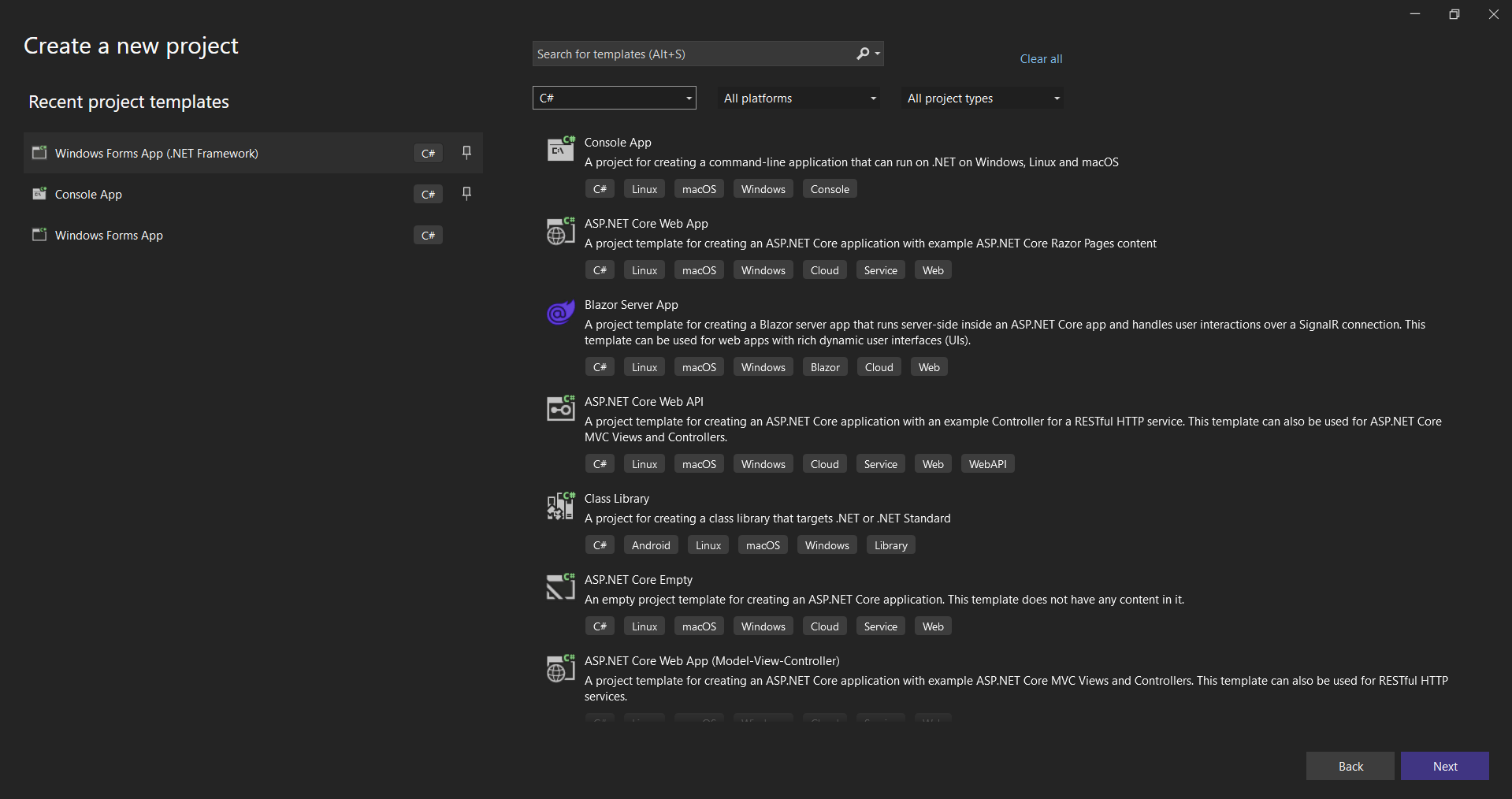
*sln* – Chỉnh sửa file .NET Core solution

*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

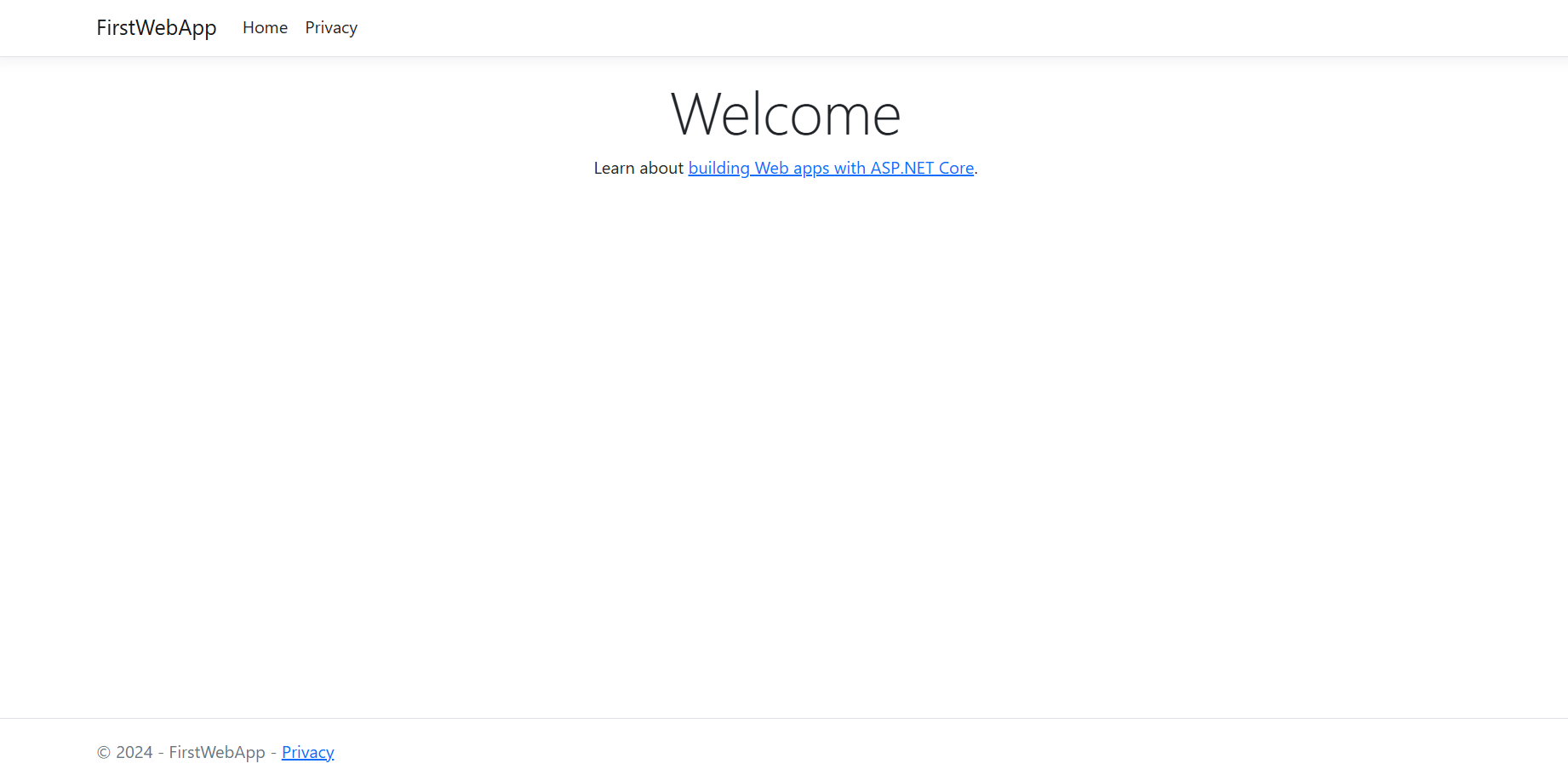
**A screenshot of a computer

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.

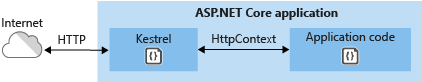
ASP.NET dính chặt vào IIS cũng là gánh nặng cho IIS.

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 [Phụ lục 2]

## 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:

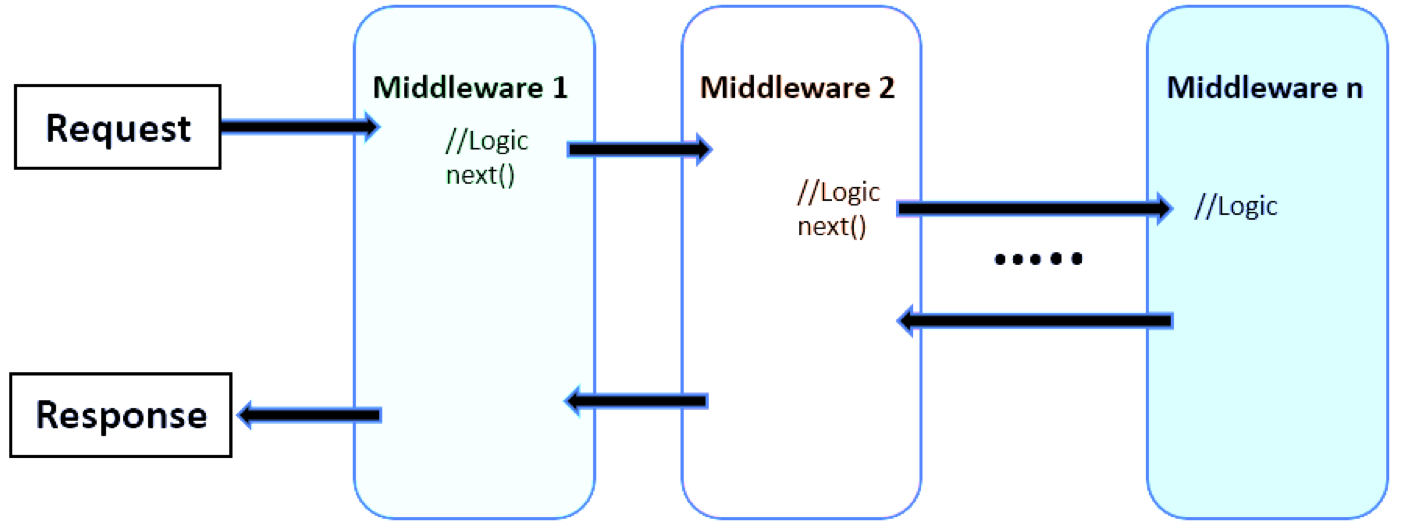
* Nhận một HTTP Request gửi đến và phát sinh ra HTTP Response để trả về
* Nhận một HTTP Request gửi đến, thi hành một số tác vụ (có thể là sửa đổi HTTP Request), sau đó chuyển đến một middleware khác.
* Nhận HTTP Response, sửa nó và chuyển đến một Middleware khác

## 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.

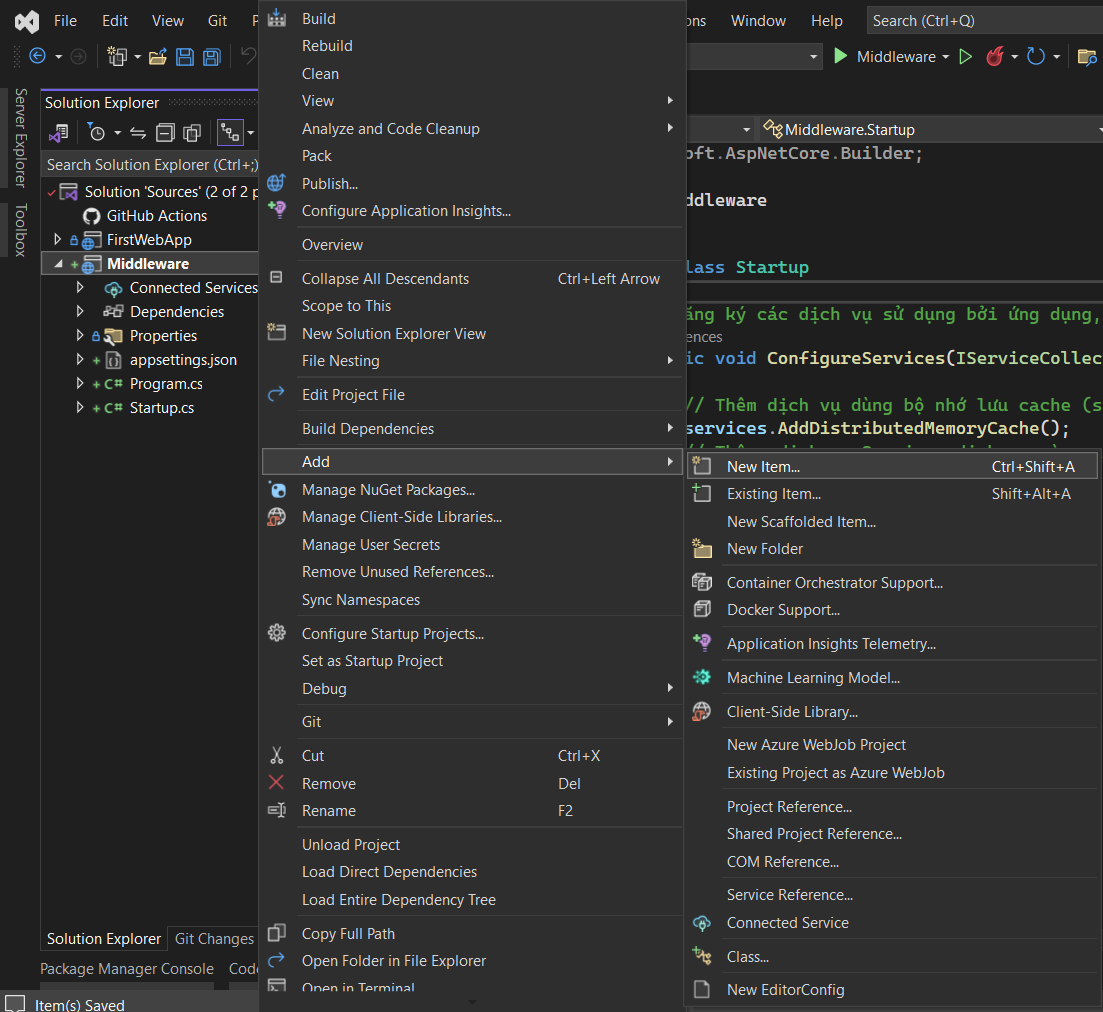


Ảnh 6: Sơ đồ các middleware trong pipeline

Để thực hành mở Visual Studio > Chọn ASP.NET Core Empty đặt tên project là Middleware.

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.

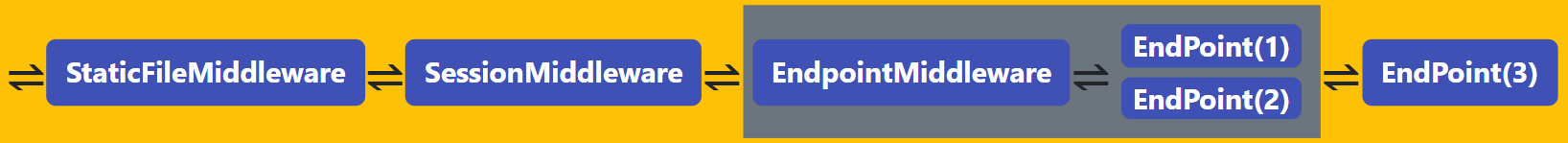
Thêm file Startup.cs nếu chưa có. Chuột phải vào tên dự án > Add > New Item > Start up class



Sửa code *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 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 theo 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(1) 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 dụng 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, put thì HttpContext đi qua các Middleware: *StaticFileMiddleware, SessionMiddleware, EndpointMiddleware, EndPoint(1)*

## 4.3. Tạo Middleware riêng trong ASP.NET Core

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**

Chuột phải vào tên dự án > Add > New Item > Middleware Class đặt tên class là *CheckAcessMiddleware*.

|  |
| --- |
| using Microsoft.AspNetCore.Builder;  using Microsoft.AspNetCore.Http;  using Middleware;  using System.Net.Http;  using System.Threading.Tasks;  namespace Middleware  {  public class CheckAcessMiddleware  {  // Lưu middlewware tiếp theo trong Pipeline  private readonly RequestDelegate \_next;  public CheckAcessMiddleware(RequestDelegate next)  {  \_next = next;  }  public Task Invoke(HttpContext httpContext){  if (httpContext.Request.Path == "/testxxx") {  Console.WriteLine("CheckAcessMiddleware: Cấm truy cập");  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  return \_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*.

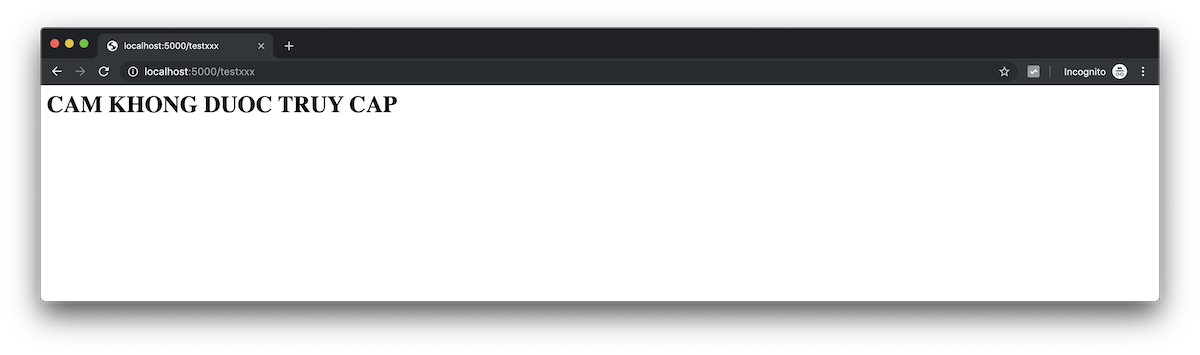
|  |
| --- |
| namespace Middleware.Middleware  {  public class CheckAcessMiddleware  {  /…  }  // Mở rộng cho IApplicationBuilder phương thức UseCheckAccess  public static class CheckAcessMiddlewareExtensions  {  public static IApplicationBuilder UseCheckAccess(this IApplicationBuilder builder)  {  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 IMiddleware

Bạn có thể triển khai interface 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.UseFront();**  // ... các Middleware khác  } |

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

## 4.4. Truyền dữ liệu giữa các Middleware

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:**

|  |
| --- |
| var 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**()) |

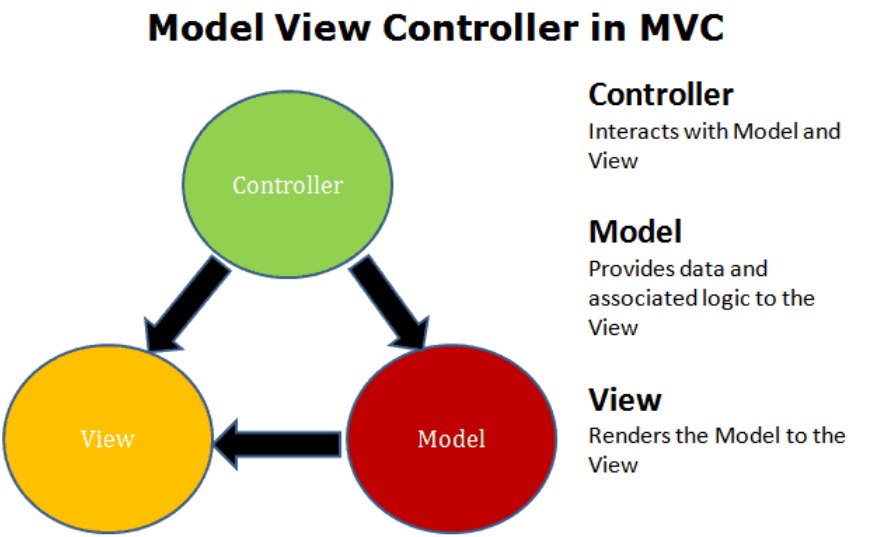
Theo quy tắc thì webroot được đặt cho thư mục wwwroot. Bạn có thể thay đổi bằng cách sử dụng phương thức UseWebRoot.

# 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

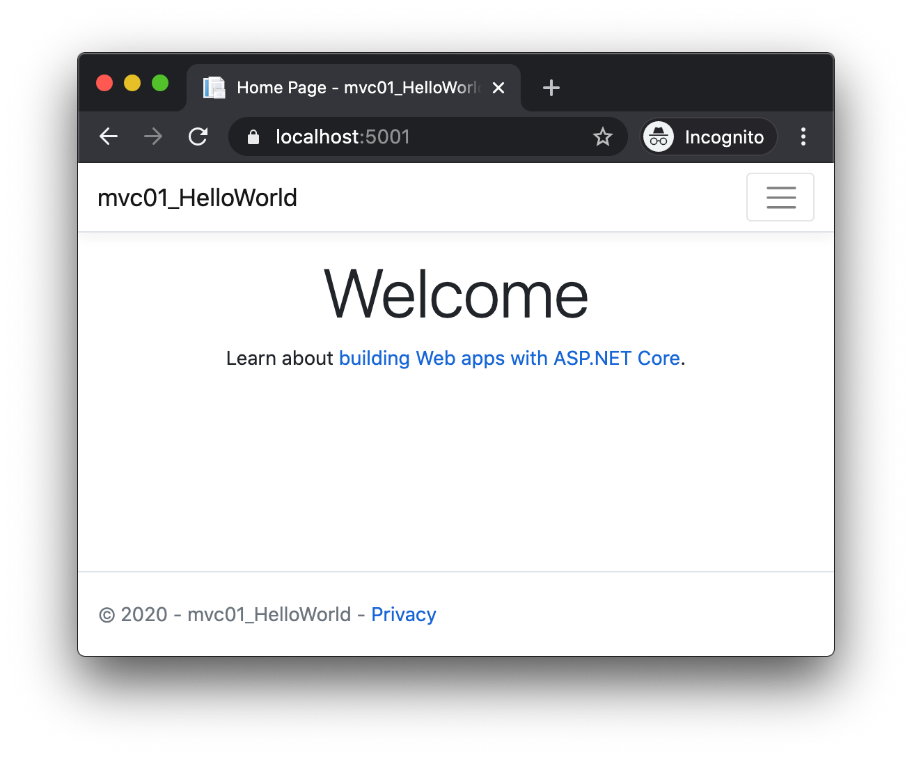
|  |
| --- |
| dotnet new mvc |

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.

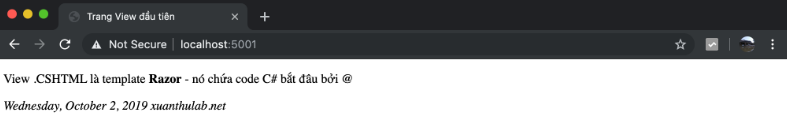
Khi một Action trả về đối tượng ViewResult (phương thức View của Controller tạo ra loại đối tượng này). Thì nó sẽ sử dụng View .cshtml và Razor làm việc để tạo nội dung HTML. Nó sẽ tìm đến thư mục Views, tìm đến thư mục cùng tên Controller ví dụ Home, sau đó là file .cshtml cùng tên với Action - file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

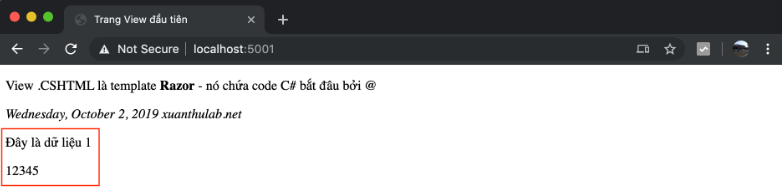
* Thiết lập Model cho View, khi gọi phương thức View của Controller. Trong View .cshtml sử dụng chỉ thị *@model* để thiết lập kiểu truyền tới cho Model của View
* Sử dụng property *ViewData* của View và Controller
* Sử dụng *ViewBag* của View và Controller
* Sử dụng TemData truyền dữ liệu giữa các trang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

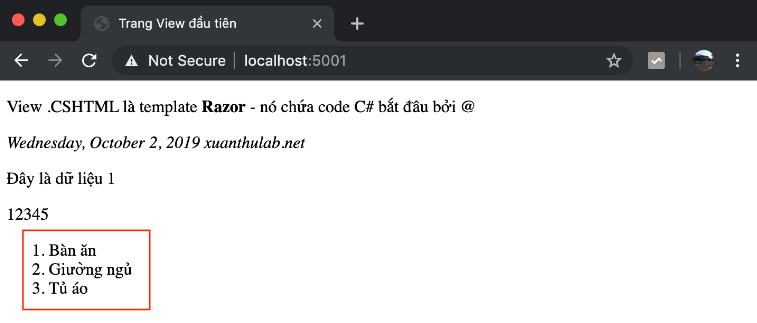
Lúc này, trong .phtml, Razor sẽ đọc được qua thuộc tính với tên Model - Ví dụ:

|  |
| --- |
| **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*

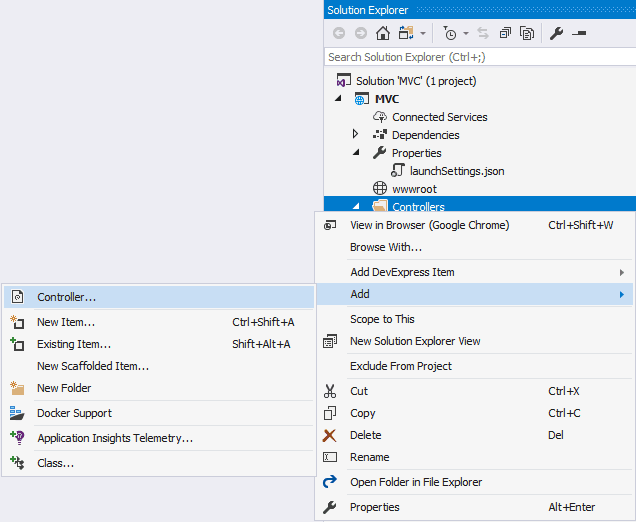
Trường hợp này, Controller có tên là CustomerController được gọi. Sau đó nó sẽ gọi đến Action metod tên List rồi tạo ra response trả về cho user.

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. Routing trong ASP.NET Core MVC

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 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?}

### 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.

### 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*.

### 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. Routing mặc định của dự án MVC

Trước mắt hãy cùng xem xét loại routing mặc định mà ASP.NET Core MVC sử dụng cho một project mới.

Khi cấu hình sử dụng Mvc Middleware, bạn bắt buộc phải chỉ định ít nhất một route template trong phương thức *Configure* của *Startup.cs* sử dụng phương thức *UseEndpoints* như sau:

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

*"{controller=Home}/{action=Index}/{id?}"* là route template mặc định ASP.NET Core sử dụng cho các dự án MVC. Đây là cơ chế conventional routing, áp dụng chung cho toàn bộ controller và action của ứng dụng.

Route template là chuỗi ký tự quy định hình thức của một Url hợp lệ.

Template mặc định của MVC quy định rằng phần path của một Url hợp lệ phải có dạng *{controller}/{action}hoặc {controller}/{action}/{id}*. Trong đó:

* {controller}, {action} và {id} được gọi là các placeholder hay route parameter. Placeholder {controller} và {action} được cơ chế routing của MVC sử dụng để xác định xem url này tương ứng với controller và action nào.
* Ký tự / phân chia chuỗi template thành các segment.
* Home và Index được gọi là giá trị mặc định tương ứng của controller và action.
* {id?} là không bắt buộc.

Đây là một template rất tổng quát.

Khi sử dụng template này, ASP.NET Core sẽ tách URL của truy vấn thành các segment. Hai segment đầu tiên cho phép rút ra thông tin về controller và action. Dựa trên thông tin này ASP.NET Core sẽ tìm kiếm controller và action phù hợp để thực thi.

Nếu có segment thứ 3, giá trị của segment này sẽ đưa vào một biến có tên id để truyền cho action. Segment thứ 3 không bắt buộc.

Lấy ví dụ truy vấn tới là Book/Delete/3. Cơ chế routing sẽ so khớp nó với mẫu {controller}/{action}/{id}. Quá trình so khớp này cho thấy controller là Book, action là Delete và id là 3. Từ thông tin về controlller và action, ASP.NET Core MVC sẽ kích hoạt đúng phương thức action phù hợp. Khi kích hoạt action này nó cũng đồng thời truyền giá trị id = 3 nếu action yêu cầu.

Khi bạn tạo một controller mới với các action trong đó, các action này tự động có URL phù hợp. Bạn không cần tự mình cấu hình router riêng. Khi nhìn thấy một Url, bạn có thể dự đoán được ngay action trong controller nào sẽ được gọi khi phát đi truy vấn.

## 10.3. Routing trong ASP.NET Core MVC

ASP.NET Core MVC cung cấp hai cơ chế routing: conventional routing và attribute routing.

### 10.3.1. Conventional routing

Conventional routing là loại routing được định nghĩa chung cho nhiều controller action dựa trên một số quy ước xác định. Bất kỳ controller và action nào tuân theo quy ước đều tự động tham gia vào routing tương ứng.

Trong cơ chế routing này, bạn xuất phát bằng cách định nghĩa một hoặc một vài route template trong phương thức Configure của lớp Startup.

Khi một truy vấn đến, ASP.NET Core MVC lần lượt so khớp Url của truy vấn với các template để xem Url đó phù hợp với template nào. Việc so sánh được thực hiện theo thứ tự khai báo của template: template nào khai báo trước sẽ so sánh trước. Nếu tìm thấy template phù hợp thì quá trình so khớp chấm dứt, kể cả khi vẫn còn template chưa so.

Conventional routing đưa ra khuôn mẫu Url và bạn xây dựng controller/action tuân thủ khuôn mẫu.

ASP.NET Core MVC bắt buộc phải xác định được tên controlller và action từ quá trình so khớp Url và route template.

Đây là cơ chế routing mặc định của ứng dụng ASP.NET Core MVC.

Conventional routing đơn giản, tiện lợi, dễ theo dõi và quản lý. Controller và action phải tuân theo quy ước thì mới áp dụng được cơ chế routing này.

### 10.3.2. Attribute routing

Attribute routing là loại routing được định nghĩa riêng rẽ cho từng action bằng cách đặt attribute [Route] trước mỗi action.

Attribute routing có ý tưởng ngược lại so với conventional routing. Trong attribute routing bạn xuất phát từ action và định nghĩa Url tương ứng cho nó.

Khi bắt đầu, ứng dụng duyệt qua tất cả các action (trong tất cả các controller) để xem action nào được đánh dấu [Route] và xây dựng một từ điển url (khóa) => action (giá trị). Khi truy vấn tới, ASP.NET Core MVC trích url và tìm khóa (url) tương ứng trong từ điển. Nếu tìm thấy, action tương ứng sẽ được thực thi.

Loại routing này linh hoạt hơn nhiều so với conventional routing song lại khó theo dõi và quản lý hơn. Khi sử dụng attribute routing, bạn cần tự mình quản lý một danh sách các routing riêng rẽ.

Phương pháp routing này hữu dụng hơn khi xây dựng Web API.

### 10.3.3. Route template

Mvc Middleware cho phép cung cấp và sử dụng đồng thời cả hai phương pháp routing và nhiều cấu trúc ánh xạ khác nhau. Mỗi cấu trúc ánh xạ được gọi là một router hoặc route template.

*Route template là những chuỗi ký tự định nghĩa cấu trúc cho các URL hợp lệ của ứng dụng.*

Dù sử dụng phương pháp nào, cuối cùng bạn cũng đều phải định nghĩa route template.

### 10.3.4. Cấu trúc route template

Cấu trúc route template là chung cho cả attribute và conventional routing.

**Segment và parameter**

Route template được phân chia thành segment. Mỗi segment có thể là:

* Một chuỗi ký tự “tĩnh” gọi là literal segment. Như trong my-route-1, hi chính là một literal segment. Literal segment yêu cầu phải hoàn toàn trùng khớp nhưng không phân biệt hoa thường. Trong template “hi/{fname}-{lname}”, segment đầu tiên bắt buộc phải là hi, Hi, HI, hay hI.
* Thành phần “động” gọi là placeholder hoặc parameter. Placeholder trong template được thể hiện trong cặp dấu {}. Đây là thành phần phức tạp và tạo ra sự linh hoạt của route template. Có hai loại placeholder: (1) loại do ASP.NET Core MVC định nghĩa sẵn và bắt buộc, bao gồm {controller} và {action}; (2) loại do bạn tự đặt ra.
* Tổ hợp của cả literal và route parameter. Đây chính là cách chúng ta tạo ra segment thứ hai của template “hi/{fname}-{lname}”.

Bạn bắt buộc phải có cách chỉ định {controller} và {action} khi thêm route template. Chỉ định này giúp cơ chế routing của ASP.NET Core MVC xác định controller và action xử lý truy vấn.

Nếu hai placeholder này không nằm sẵn trong chuỗi template, bạn phải cung cấp một object vô danh để chỉ định rõ giá trị của controller và action. Đây cũng là cách chúng ta sử dụng khi khai báo my-route-1.

Đối với parameter do bạn tự định nghĩa, bạn sử dụng nó như là cơ chế truyền dữ liệu từ Url vào action. Cơ chế routing có khả năng trích phần giá trị tương ứng của parameter và truyền vào làm tham số của action. Đây lại là một phần của cơ chế model binding mà chúng ta sẽ học ở một bài khác.

Số lượng segment và placeholder trong mỗi route hoặc template không hạn chế, có thể bắt buộc hoặc tùy chọn, có thể chịu các giới hạn nhất định.

**Paramter tùy chọn và parameter bắt buộc**

Trong route template, mặc định mỗi parameter đều là bắt buộc. Nghĩa là nếu bạn đã đặt một parameter vào template, trong Url bắt buộc phải có phần tương ứng với parameter. Nếu thiếu, Url không phù hợp với template.

Nếu bạn muốn một parameter trở thành không bắt buộc, nghĩa là ở vị trí đó có thể có hoặc có thể không đặt giá trị gì, bạn đặt thêm ký tự ? ở sau tên parameter.

Ví dụ, trong route template mặc định “{controller=Home}/{action=Index}/{id?}”, {id?} là một parameter tùy chọn, trong khi {controller} và {action} là bắt buộc.

Khi này, cả hai Url Home/Index và Home/Index/3 đều phù hợp (với giả định rằng có HomeController và Index action).

**Giá trị mặc định của parameter**

Paramter có thể nhận giá trị mặc định bằng cách thêm =giá-trị vào sau tên paramter. Trong route mặc định “{controller=Home}/{action=Index}/{id?}”, controller có giá trị mặc định là Home, và action có giá trị mặc định là Index.

Sử dụng giá trị mặc định cho phép bỏ qua paramter tương ứng.

Ví dụ: Url /, /home và /home/index đều ánh xạ sang Index action của HomeController.

Theo đó, / tương ứng với bỏ qua cả controller, action và id; /home tương ứng với bỏ qua action và id

## 10.4. Conventional Routing

Conventional routing được định nghĩa trong phương thức *Configure* của *Startup.cs*:

|  |
| --- |
| public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {  if (env.IsDevelopment()) {  app.UseDeveloperExceptionPage();  }  app.UseRouting();  app.UseEndpoints(endpoints => {  endpoints.MapControllerRoute("default", "{controller = Home}/{action = Index}/{id?}");  });  } |

Mỗi template được định nghĩa bởi một lần gọi phương thức MapControllerRoute. Phương thức này tối thiểu chứa tên template và chuỗi template. Tên template vừa dùng để phân biệt các template, vừa được dùng trong html helper hoặc tag helper sinh URL.

Mặc định ASP.NET Core MVC định nghĩa sẵn cho bạn “default” template. Chúng ta đã giải thích ý nghĩa của default template ở phần trên. Bạn có thể định nghĩa bao nhiêu template tùy thích.

Quy trình hoạt động khi sử dụng conventional routing như sau:

1. Khi truy vấn tới, ASP.NET Core MVC sẽ trích ra url từ truy vấn.
2. Url sẽ được so khớp lần lượt với các template để xem nó phù hợp với template nào.
3. Nếu có nhiều route template, ASP.NET Core MVC sẽ so khớp lần lượt với từng template theo thứ tự khai báo cho đến khi tìm được route phù hợp.
4. Nếu tìm thấy template phù hợp, quá trình so khớp sẽ dừng lại, dù tiếp theo vẫn còn template nữa.
5. Dựa theo cấu trúc template, tên controller và action sẽ được trích ra từ Url.
6. Thực thi action tương ứng.

Bạn nên bố trí template cụ thể hơn ở trên, template tổng quát hơn ở dưới.

Giờ hãy điều chỉnh phương thức UseEnpoints như sau:

|  |
| --- |
| app.UseEndpoints(endpoints => {  endpoints.MapControllerRoute("my-route-1", "hi/{fname}-{lname}", new { controller = "Default", action = "Hi" });  endpoints.MapControllerRoute("my-route-2", "web/{action}/{controller}");  endpoints.MapControllerRoute("default", "{controller = Home}/{action = Index}/{id?}");  }); |

Bạn đã xây dựng hai route template mới. *“my-route-1”* ánh xạ những Url như *hi/Donald-Trump*, *hi/Vladimir-Putin* vào phương thức action *Hi* của *DefaultController*. *“my-route-2”* cho phép gọi đến *DefaultController.Hi()* qua Url *web/hi/default*.

Trong hai template mới, *my-route-2* tổng quát hơn vì nó áp dụng được với nhiều controller và action. *my-route-1* cụ thể hơn vì nó chỉ áp dụng duy nhất cho *DefaultController.Hi()*. Mức độ tổng quát của template được xác định qua số controller/action tiềm năng.

|  |
| --- |
| using Microsoft.AspNetCore.Mvc;  namespace WebApplication1.Controllers {  public class DefaultController : Controller {  public IActionResult Hi(string fname, string lname) {  return View((fname, lname));  }  }  } |

Trong template *my-route-1* có 2 segment: hi và {fname}-{lname}. Segment thứ nhất (hi) được gọi là literal segment. Segment thứ hai là tổ hợp của hai route parameter {fname} và {lname} phân tách bởi ký tự -.

Do từ chuỗi template không xác định được sẽ gọi controller và action nào, bạn cung cấp thêm một object thuộc kiểu vô danh, trong đó chỉ rõ controller sẽ luôn nhận giá trị Default, và action luôn nhận giá trị Hi. Object này đảm bảo rằng nếu Url tới phù hợp, action Hi của DefaultController sẽ luôn được gọi.

Cơ chế routing của ASP.NET Core so khớp Url tới xem nó trùng khớp với khuôn mẫu nào. Nếu hình thức url phù hợp với mẫu my-route-1 (/hi/xxx-yyy), chuỗi ký tự ở vị trí tương ứng fname và lname sẽ được tách ra và đặt vào trong một object dạng từ điển để truyền vào action.

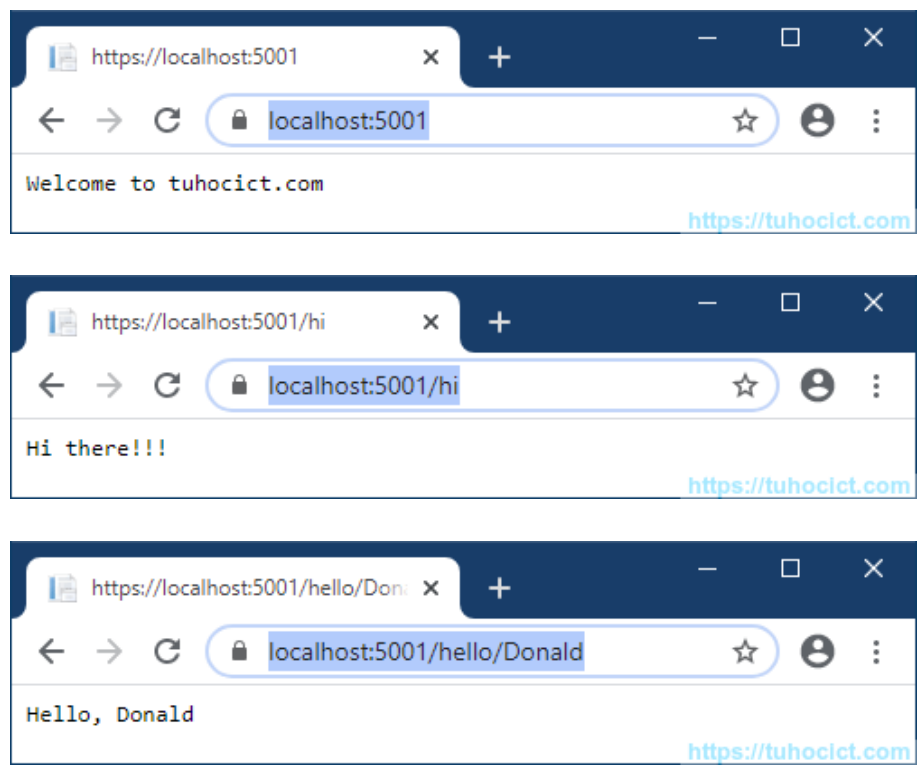
## 10.5. Attribute Routing

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.

Hãy xây dựng GreetingController với các action như sau:

|  |
| --- |
| using Microsoft.AspNetCore.Mvc;  namespace WebApplication1.Controllers {  public class GreetingController {  *[Route("")]*  public string Greeting() => $"Welcome to TuHocIct";  *[Route("hi")]*  public string Hi() => $"Hi there!!!";  *[Route("hello/{name}")]*  public string Hello(string name) => $"Hello, {name}";  }  } |

Chạy ứng dụng với các url /, /hi, /hello/Donald bạn sẽ thu được kết quả như sau:



3 action của GreetingController đều sử dụng attribute routing:

* Greeting tương ứng với root url /;
* Hi() tương ứng với url /hi;
* Hello(string name) tương ứng với url /hello/{name}.

Lưu ý: [Route("hi")] và [Route("/hi")] trong trường hợp này là tương đương nhau và cùng tương ứng với url /hi. Tuy nhiên, không phải lúc nào chúng cũng tương đương nhau. Nếu sử dụng cơ chế ghép nối url của attribute routing, template bắt đầu với ký tự “/” có ý nghĩa hoàn toàn khác biệt.

Có thể thấy việc định nghĩa attribute routing rất tự do. Tuy nhiên:

* Cần đảm bảo các template cho các action khác nhau không trùng lặp;
* Có thể định nghĩa nhiều template cho cùng một action.

Attribute routing cũng sử dụng placeholder/parameter trong template tương tự như conventional routing.

Cơ chế hoạt động của attribute routing khác biệt với conventional routing:

* Khi ứng dụng chạy, ASP.NET Core MVC sẽ scan qua tất cả các action. Nếu action nào được đánh dấu bằng [Route(…)] thì template tương ứng sẽ được trích ra và đặt làm khóa cho một từ điển url, còn action sẽ được sử dụng làm giá trị của mục từ điển tương ứng. Kết thúc quá trình scan sẽ thu được một từ điển bao gồm các cặp url => action.
* Khi truy vấn tới, url sẽ được trích ra và so sánh với các khóa có trong từ điển. Nếu trùng với khóa nào thì action tương ứng sẽ được thực thi.
* Nếu một action đã được thực thi, việc xử lý url kết thúc. Tất cả các conventional template bị bỏ qua.
* Nếu không tìm thấy khóa (url) tương ứng trong từ điển, ASP.NET Core MVC sẽ tiếp tục so khớp với các conventional template.

⇒ Attribute routing có độ ưu tiên cao hơn conventional routing.

Như vậy, mặc dù ASP.NET Core MVC cho phép sử dụng song song cả conventional và attribute routing, tuy nhiên nếu một action đã sử dụng attribute routing, nó sẽ không thể tham gia vào conventional routing được nữa.

Giờ hãy điều chỉnh GreetingController như sau:

|  |
| --- |
| *[Route("greeting")]*  public class GreetingController {  ...  } |

Bạn chỉ thêm attribute [Route(“greeting”)] vào trước GreetingController class. Nếu chạy ứng dụng, bạn sẽ phải dùng các Url sau:

* greeting/ cho Greeting() action
* greeting/hi cho Hi() action
* greeting/hello/{name} cho Hello(string name) action

Tức là url giờ được ghép từ template của controller và template của action. Đây là cơ chế ghép template của attribute routing.

Lưu ý về ký tự / đầu template: nếu một template bắt đầu bằng /, nó là một template tuyệt đối. Template tuyệt đối của action không ghép được với template của controller. Ví dụ [Route(“/hi”)] là một template tuyệt đối. Action với route này luôn luôn được truy xuất qua url /hi.

**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**(); |

### 13.3.6. Kết quả liên quan đến bảo mật

#### 13.3.6.1. SignInResult

*SignInResult* là kết quả của hành động đăng nhập. *SignInManager.SignInAsync* hoặc *PasswordSignInAsync* trả về một *SignInResult*. Nó có 4 thuộc tính là Succeeded, IsLockedOut, IsNotAllowed và RequiresTwoFactor.

#### 13.3.6.2. SignOutResult

*SignOutResult* là kết quả của hành động logout.

#### 13.3.6.3. ForbitResult

*ForbitResult* trả về lỗi 403 tức là người dùng không được cấp quyền để thực hiện một hành động nào đó trên tài nguyên nào đó. *ForbitResult* không có nghĩa là người dùng chưa chứng thực. Người dùng chưa chứng thực nên trả về *ChallengeResult* hoặc *UnAuthorisedResult*.

|  |
| --- |
| **return** new **ForbidResult**();  hoặc  **return** **Forbid**(); |

*Forbit* là phương thức của Controller base class trả về thể hiện của *ForbidResult*. Thay vào đó bạn có thể trả về Status Code:

|  |
| --- |
| **return** **StatusCode**(403); |

#### 13.3.6.4. ChallengeResult

*ChallengeResult* trả về khi chứng thực thất bại. Kết quả sẽ không gọi đến bất cứ middleware nào để tạo response. Ví dụ trả về lỗi 401 (Unauthorized) hoặc 403 (Forbidden) hoặc chuyển hướng người dùng đến trang đăng nhập.

#### 13.3.6.5. UnauthorizedResult

*UnauthorizedResult* trả về lỗi “401 – Unauthorized”. Controller sử dụng phương thức *Unauthorized* để trả về thể hiện của *UnauthorizedResult*.

# Chương XIV. View

## 14.1. Định nghĩa?

### 14.1.1. View là gì?

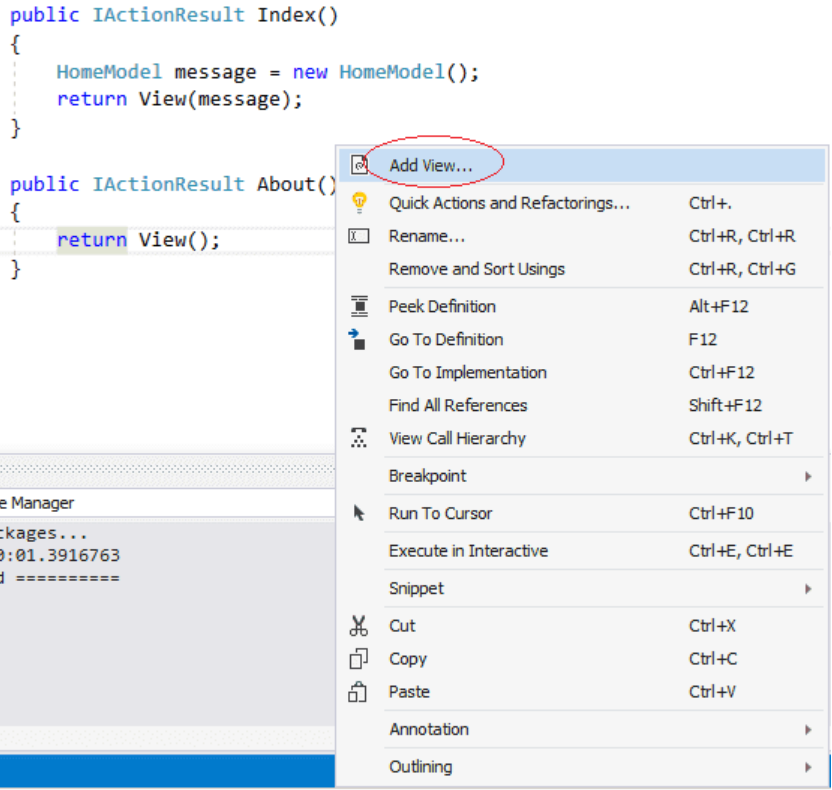
View có trách nhiệm tạo ra giao diện cho người dùng từ model. Controller trong ASP.NET Core sẽ nhận request sau đó thực thi với logic tương ứng với dữ liệu đầu vào từ request. Sau đó nó trả về Model cho View.

### 14.1.2. Trách nhiệm của View

Render ra giao diện và hiển thị model lên là trách nhiệm của View. View không nên chứa bất cứ logic nào và không được xử lý logic. View có thể sử dụng bất cứ định dạng nào để trả về cho user. Định dạng có thể là HTML, JSON, XML hay là bất cứ định dạng nào khác.

### 14.1.3. Thực hành tạo View? [Phụ lục 3]

Các method trong Controller trả về một Action Result (hoặc một class kế thừa từ *ActionResult*). *ActionResult* trả về HTML là một *ViewResult*. Phương thức View của Controller base class trả về một ViewResult. Chuột phải vào bất cứ đâu trong phương thức và chọn Add View.



Sau đó sẽ thấy cửa sổ Add MVC View như sau:

A screenshot of a computer

Description automatically generated

**Template:** Template có vài tùy chọn như Create, Delete, Details, Edit, List, Empty (without Model). Các template ở trên cần một model, ngoại trừ Empty Template. Visual Studio tạo ra một view để create, edit, delete...trên model được cung cấp phụ thuộc vào template mà bạn chọn. Mỗi Template trên đều có giải thích phía dưới.

* Create: Tạo một HTML Form để tạo mới một model. Nó tạo ra một label và một input cho mỗi thuộc tính trong model.
* Delete: Tạo một HTML Form cho việc xóa model. Nó tạo ra một label và giá trị heienj tại cho mỗi thuộc tính của model.
* Details: Tạo mới một view để hiển thị model. Nó tạo ra một label và giá trị của mỗi thuộc tính trong model.
* Edit: Nó tạo ra một HTML form để sửa model. Nó tạo ra một label và một input cho mỗi thuộc tính của model.
* Empty: Tạo ra một view trống
* List: Tạo ra một HTML Table hiển thị danh sách model. Nó tạo ra một cột cho mỗi thuộc tính của Model. Bạn cần trả về một IEnumerable<Model> cho View. View cũng chứa danh sách các hành động để thực hiện như create/edit/delete.

**Model class:** Dropdown hiển thị tất cả các Model class trong project. Tùy chọn này được loại bỏ nếu bạn chọn Empty project.

**Tạo một Partial View:** Tùy chọn này là Create as a Partial View. Partial View tạo một phần của view và không phải view hoàn chỉnh. *PartialViewResult* sử dụng model để tạo ra một phần của View. Kết quả của partial view nhìn giống hư một view bình thường ngoại trừ nó không có thẻ *<html>* hoặc không có thẻ *<head>* ở trên của View. Chọn tùy chọn này chỉ ra view của bạn sẽ tạo không phải view đầy đủ vì thế tùy chọn Layout sẽ bị bỏ đi.

**Thư viện Scripts:** Chọn tùy chọn này sẽ thêm *jquery.validate.min.js* và *jquery.validate.unobtrusive.min.json* vào thưu viện JavaScript. Các thư viện này cần thiết cho triển khai client-side validation. Các thư viện này đòi hỏi khi tạo view chứa một đầu vào cho dữ liệu như một Form Edit view hoặc một form Create.

**Use as a layout page:** Tùy chọn này cho phép bạn chọn một Layout page cho View. Layout page được dùng để chia sẻ các thành phần dùng chung trong trang của bạn và cung cấp một giao diện đồng nhất trong toàn bộ hệ thống.

Ví dụ chọn tên view là About, và template là Empty (without model) không chọn vào Create a partial view và Use a layout page. View được tạo sẽ nằm trong thư mục *Views/Home/About.cshtml*. Mở file *About.cshtml* ra và thêm dòng *<h1>About Us</h1>* vào sau thẻ tiêu đề:

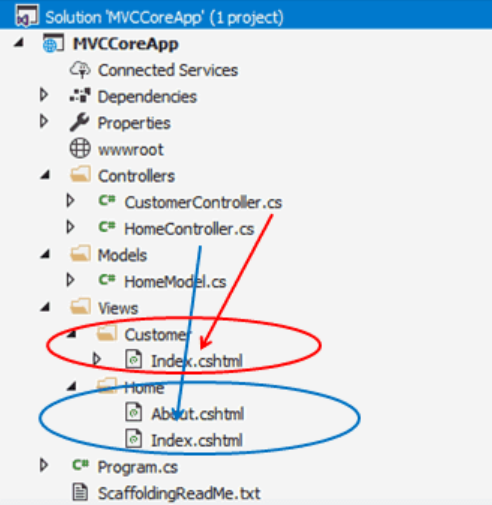
|  |
| --- |
| @{  Layout = null;  }    *<!DOCTYPE html>*  <**html**>  <**head**>  <**meta** name="viewport" content="width=device-width" />  <**title**>About</**title**>    <**h1**>About Us</**h1**>  </**head**>  <**body**>  </**body**>  </**html**> |

Chạy ứng dụng và bạn nhìn thấy dòng chữ "About us" trên ứng dụng.

**Vị trí của View**

Theo quy tắc thì tất cả View được lưu trong thư mục *Views* của ứng dụng. Mỗi Controller sẽ có một thư mục con trong thư mục *Views* với tên trùng tên controller nhưng sẽ không có hậu tố Controller. Vì thế sẽ có một *HomeController* thư mục tên *Home* trong thư mục *Views*.

Mỗi Action method trong *Controller* sẽ lấy một file cho nó, tên trùng với tên của Action method. Vì thế phương thức *Index* của *HomeController* có một file với tên *Index.cshtml* trong *Views/Home*.



## 14.2. ASP.NET Core tìm View như thế nào?

Project có một Controller tên là *HomeController*. Giờ hãy thêm *CustomerController* vào. Chọn thư mục Controllers chuột phải và add thêm Controller. Tên của nó là *CustomerController*.

|  |
| --- |
| using Microsoft.AspNetCore.Mvc;    **namespace** **MVCCoreApp.Controllers**  {  **public** **class** CustomerController : **Controller**  {  **public** **IActionResult** **Index**()  {  **return** **View**();  }  }  } |

Giờ chúng ta không tạo View thay vào đó chạy luôn URL /Customer. Chúng ta nhìn lỗi như sau:

A screenshot of a computer

Description automatically generated

Màn hình ở trên hiển thị chính xác lỗi khi ASP.NET Core MVC middleware tìm kiếm View trong 2 thư mục:

* */Views/Customer/Index.cshtml*
* */Views/Shared/Index.cshtml*

Đầu tiên là thư mục *Customer* với tên của controller. Nếu không thấy nó sẽ tìm tiếp trong thư mục Shared trước khi đưa ra lỗi *InvalidOperationException*.

Thư mục *Shared* là thư mục đặc biệt chứa *Views, Layouts* hoặc *Partial* view dùng chung cho nhiều view.

Mở *CustomerController* và click bất cứ chỗ nào trong *Index* method. Chuột phải chọn **Create an Empty View**. File *Index.cshtml* sẽ tự động tạo ra trong thư mục *Views/Customer*. Giờ hãy mở *Index.cshtml*:

|  |
| --- |
| @{  Layout = null;  }    *<!DOCTYPE html>*  <**html**>  <**head**>  <**meta** name="viewport" content="width=device-width" />  <**title**>Index</**title**>  <**h1**>Customer List</**h1**>  </**head**>  <**body**>  </**body**>  </**html**> |

Giờ hãy chạy ứng dụng và chúng ta không thấy lỗi này nữa.

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

## 15.1. Định nghĩa

**View Engine là gì?**

View Engine có trách nhiệm xuất ra HTML cuối cùng khi Controller gọi Action Method. Controller action method trả về các kiểu khác nhau của response. Nó gọi là Action Result. *ViewResult* là một ActionResult nó trả ra HTML Response. *ViewResult* được cung cấp bởi View Engine. Nó sẽ sản sinh ra HTML.

**Razor View Engine là gì?**

Razor View Engine là View Engine mặc định của ASP.NET Core. Nó lấy mã Razor trong file View và chuyển sang HTML response.

**Razor Markup**

Controller trong MVC gọi View bằng cách gán dữ liệu để tạo giao diện. View phải có khả năng xử lý dữ liệu và tạo response. Điều này được xử lý bằng cách dùng Razor, nó cho chúng ta sử dụng C# code trong file HTML. Razor View Engine xử lý các lệnh này và tạo ra HTML.

Các file chứa Razor có đuôi *.cshtml*. Cú pháp Razor thường ngắn hơn và đơn giản hơn cũng dễ học như C# hoặc VB. Visual Studio IntelliSense cũng hỗ trợ cú pháp Razor.

## 15.2. Cú pháp Razor

Razor sử dụng ký tự @ để chuyển HTML sang C#. Có 2 cách để khai báo:

* Sử dụng khối lệnh Razor
* Sử dụng Razor expression

Các biểu thức này được xử lý bởi Razor View Engine và được gán vào response.

### 15.2.1. Khối lệnh Razor

Khối lệnh Razor bắt đầu bởi @ và nằm trong cặp ngoặc nhọn. Bạn có thể sử dụng khối lệnh bất cứ đâu trong file. Một khối Razor có thể sử dụng để thao tác Model, khai báo biến, đặt thuộc tính của View. Tuy nhiên nó không nên sử dụng cho việc xử lý logic.

Ví dụ:

|  |
| --- |
| <**h3**>Code Block</**h3**>    @{  **var** greeting = "Welcome to our site!";  **var** weekDay = **DateTime.Now.DayOfWeek**;  }    @{  **var** cust = **new** **MVCCoreApp.Models.Customer**()  {  name = "Rahul Dravid",  address = "Bangalore"    };  } |

Đầu tiên chúng ta tạo một khối lệnh Razor bắt đầu với @ và cặp {}. Trong cặp ngoặc này chúng ta có một đoạn C# thông thường với lệnh khai báo hai biến greeting và weekDay

### 15.2.2. Biểu thức Razor

Biểu thức Razor bắt đầu với @ và theo sau là code C#. Biểu thức này có thể là ngầm định hoặc tường minh.

**Implicit Razor Expressions**

Implicit Razor Expressions không cho phép dấu cách vì nó dùng để kết thúc biểu thức. Các biểu thức được xử lý bởi Razor View Engine và kết quả được thêm ngay vào vị trí nó đặt.

|  |
| --- |
| <**h3**>Code Expression</**h3**>    <**p**>**@greeting**</**p**>  <**p**>**@DateTime.Now**</**p**>  <**p**>Today is : **@WeekDay** thank you </**p**> |

Đoạn Razor dưới đây hiển thị tên và địa chỉ trong thẻ <p>

|  |
| --- |
| <**p**>Name : **@cust.name**</**p**>  <**p**>Address : **@cust.address**</**p**> |

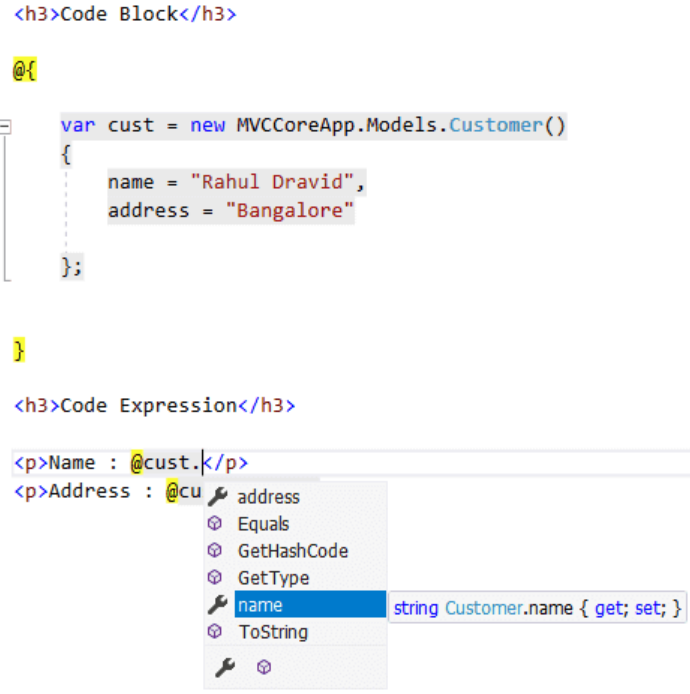
**Explicit Razor Expressions**

Explicit Razor Expressions bắt đầu bằng dấu @ và theo sau bởi cặp (). Bất cứ nội dung nào trong cặp ngoặc đơn này đều được xử lý bởi Razor và tạo ra output.

|  |
| --- |
| @{ **var** ISBNNo = "10001200"; }    <**p**>ISBN : **@ISBNNo**</**p**>  <**p**>ISBN**@(ISBNNo)**</**p**> |

### 15.1.3. Hỗ trợ nhận diện cú pháp

Hình dưới đây hiển thị cách Razor làm việc được sử dụng trong HTML và Intellisense hỗ trợ từ Visual Studio:



### 15.2.4. Các cú pháp cơ bản

**Sử dụng Directive**

Khai báo *@using* làm việc tương tự như C# vậy. Nó cho phép chúng ta import namespace. Ứng dụng có thể import namespace *MVCCoreApp.Models* theo cách sau:

|  |
| --- |
| @using MVCCoreApp.Models; |

Và bạn đơn giản là dùng nó var *cust = new Customer()* thay vì *var cust = new MVCCoreApp.Models.Customer()*

**Khai báo biến**

Các biến được khai báo sử dụng từ khóa var hoặc sử dụng kiểu dữ liệu C#. Các kiểu như int, float, decimal, bool, DateTime hay string có thể được dùng để lưu trữ kiểu dữ liệu tương ứng. Để khai báo biến bạn vẫn có thể dùng dấu @ trực tiếp:

|  |
| --- |
| <**h3**>Variables </**h3**>    <!-- Storing a string -->  @{ **var** message = "Welcome to our website"; }    <!-- Storing a date -->  @{ **DateTime** date = **DateTime.Now**; }    <**p**>@message</**p**>  <**p**> The current date is **@date**</**p**> |

**Comment**

Sử dụng @\*\*@ để đặt comment:

|  |
| --- |
| @\*This is comment\*@ |

**HTML trong khối lệnh**

Bất cứ HTML Element nào trong khối lệnh Razor được nhận dạng như bình thường:

|  |
| --- |
| @{  <**p**>Hello from the Code block</**p**>  } |

**Dòng lệnh đơn**

Bạn có thể xuất ra giá trị mà không cần thẻ HTML bằng dấu @:

|  |
| --- |
| @{  @:Hello from the Code block  } |

**Đa dòng lệnh**

Đối với nhiều dòng bạn cần phải có thẻ *<text></text>*

|  |
| --- |
| @{  <**text**>Hello from the multiline text </**text**>  } |

**If else**

Điều kiện *if* được dùng để tạo ra một nội dung dựa trên điều kiện đó:

|  |
| --- |
| @{**int** value = 200;}    @**if** (value > 100)  {  <**p**>Value is greater than 100.</**p**>  }  **else**  {  <**p**>Value is less than 100.</**p**>  } |

Hoặc:

|  |
| --- |
| @{  **var** value = 200;  **if** (value > 100)  {  <**p**>The value is greater than 100 </**p**>  }  **else**  {  <**p**>This value is less than 100.</**p**>  }  } |

Và lệnh *if else if*:

|  |
| --- |
| @**if** (SomeCondition)  {  }  **else** **if**(SomeOtherCondition)  {  }  **else**  {  } |

**Switch**

Một lệnh switch có thể thêm nội dung vào HTML trên một số điều kiện:

|  |
| --- |
| @**switch** (value)  {  **case** 0:  @: value is Zero  **break**;  **case** 100:  <**p**>Value is 100 </**p**>  **break**;  **case** 200:  <**p**>Value is @value </**p**>  **break**;  **case** 300:  <**text**>Value is 300</**text**>  **break**;  **default**:  <**p**>Invalid Value </**p**>  **break**;  } |

**Vòng lặp for**

Vòng lặp được sử dụng để lặp khối lệnh:

|  |
| --- |
| @**for** (**int** i = 0; i < 5; i++)  {  <**span**> @i </**span**>  } |

Trường hợp tốt nhất là dùng lặp qua các tập hợp đối tượng và hiển thị danh sách trong bảng:

|  |
| --- |
| @{  **var** custList = **new** **List**<Customer>()  {  **new** Customer() { name = "Rahul", address = "Bangalore" },  **new** Customer() { name = "Sachin", address = "Mumbai" }  };  }    <**table**>  <**thead**>  <**tr**><**td**>Name</**td**><**td**>Address</**td**></**tr**>  </**thead**>  @**foreach** (Customer custvar in custList)  {  <**tr**>  <**td**>@custvar.name</td>  <**td**>@custvar.address</td>  </**tr**>  }  </**table**> |

**Vòng lặp While**

|  |
| --- |
| <**h3**>While loop</**h3**>  @{  **var** r = 0;  **while** (r < 5)  {  r += 1;  <**span**> @r</**span**>  }  }    @{ **var** s = 0; }  @**while** (s < 5)  {  s += 1;  <**span**> @s</**span**>  }  } |

**Mã hóa HTML**

Tất cả biểu thức Razor được tự động mã hóa HTML

|  |
| --- |
| @{  **string** encodedMessage = "<**script**>alert('You are under cross-site script injection attack');</**script**>";  }  <**span**>@encodedMessage </**span**> |

Đoạn code trên sẽ không có kết quả ra một alert mà thay vào đó bạn sẽ thấy dòng chữ: *“<script>alert(‘You are under cross-site script injection attack’);</script>”* trên trình duyệt.

Razor sẽ tự động mã hóa < thành &lt; và > thành &gt;

Nếu muốn script thực thi, bạn bạn cần sử dụng @Html.Raw để int chuỗi không mã hóa ra.

|  |
| --- |
| <**span**>@**Html.Raw**(encodedMessage)</**span**> |

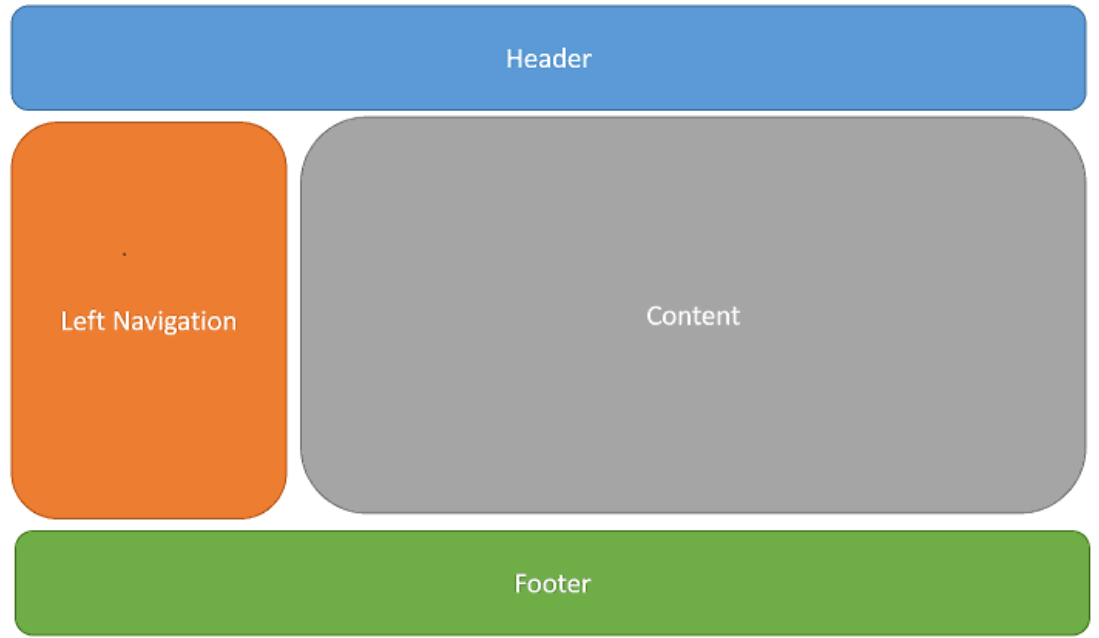
# Chương XVI. Sử dụng Layouts và Section [Phụ lục 4]

Layouts và Section trong ASP.NET MVC Core giúp chúng ta đảm bảo sự thống nhất giữa các trang trong toàn bộ các view của ứng dụng. Trong bài này chúng ta sẽ tìm hiểu cách tạo trang Layout và chia sẻ nó cho các view. Chúng ta sẽ sử dụng phương thức *RenderBody* để tạo ra View. Chúng ta cũng được biết làm thế nào định nghĩa các section và sử dụng nó với RenderSection. Cuối cùng chúng ta sẽ tìm hiểu về *\_ViewStart* trong việc định nghĩa Layout.

## 16.1. Layout của trang web

Thiết kế của hầu hết các website bao gồm một menu, một phần header, footer và phần thanh sidebar. Như bạn thấy đi từ một trang này đến trang kia trong website thì chỉ có nội dung ở giữa là thay đổi. Nhưng menu, header, footer hay màu sắc toàn cục vẫn giữ nguyên. Điều này giúp website có cái nhìn thống nhất.

Một ứng dụng web nhìn cơ bản sẽ như sau:



Ảnh 19: Cấu trúc website cơ bản

## 16.2. Layout trong ASP.NET Core là gì?

Views trong ASP.NET Core được tạo ra từ file *.cshtml* được đặt trong thư mục Views. Để giữ cho việc đồng nhất giữa các trang, chúng ta cần thêm header, footer và thanh điều hướng ở tất cả các view. Tuy nhiên điều này thường vướng víu và dễ sai sót đặc biệt nếu ta có nhiều view. Trang layout trong ASP.NET Core giúp chúng ta định nghĩa một giao diện có các phần tử dùng chung như header, footer, navigation menu trên trang ở một vị trí mà có thể dùng cho mọi nơi.

**Vị trí của trang Layout**

Theo quy ước của ASP.NET Core MVC thì các view được lưu trong thư mục Views. Mỗi Controller sẽ có một thư mục con trong thư mục Views tương ứng với tên trung với tên controller nhưng không có hậu tố "Controller". Tất cả các view được liên kết với controller trong thư mục này. Thư mục *Views* cũng có chứa thư mục tên là *Shared*, nơi chứa các view chia sẻ cho nhiều view khác dùng chung. Vì thế Layouts được đặt trong thư mục này: *Views/Shared*.

## 16.3. Layout

Tạo một thư mục tên là *Shared* bên trong thư mục *Views*. Chuột phải vào thư mục *Shared* chọn *Add View*. Và đặt tên View là *\_Layout*. Chọn Empty Template và không chọn vào cả 2 checkbox *Create as a partial view* và *Use a layout page* vì chính nó không phải partial view cũng không cần sử dụng một layout page nào vì nó là layout to nhất rồi. Trong tương lai vẫn có những layout là con của 1 layout khác chúng ta sẽ tìm hiểu sau.

A screenshot of a computer

Description automatically generated

Ảnh 20: Thêm \_Layout

Click vào *Add* để tạo View. Nó sẽ tạo ra file *\_Layout.cshtml* trong thư mục *Views/Shared*. Mở nó ra và dán đoạn nội dung này vào.

|  |
| --- |
| *<!DOCTYPE html>*    <**html**>  <**head**>  <**meta** name="viewport" content="width=device-width" />  <**title**>Layout Example</**title**>  </**head**>  <**body**>  <**div** id="header">  <**h1**>Layout example in HTML</**h1**>  </**div**>    <**div** id="content">  @**RenderBody**()  </**div**>    <**div** id="Footer">  <**p**>This is Footer</**p**>  </**div**>  </**body**>  </**html**> |

Đoạn HTML này có 3 phần là header, content và footer. Content là phần mà bạn muốn thêm nội dung các view khác vào.

**RenderBody**

*RenderBody* là một phương thức đặc biệt đánh dấu vị trí nơi mà các view sử dụng Layout này sẽ được đặt vào đó khi chạy. Cơ bản thì nó là một vùng định trước của các View sẽ hiển thị ở đó.

Chỉ có một *RenderBody* được gọi trong một trang Layout.

**Chỉ ra Layout được sử dụng trong View**

Thay đổi nội dung *Index* Action method trong *HomeController* như sau:

|  |
| --- |
| **public** **IActionResult** Index()  {  **return** View();  } |

Mở *Index.cshtml* ra từ *Views/Home* và copy dòng:

|  |
| --- |
| @{  Layout = "\_Layout";  }    <**p**>Here goes the content</**p**> |

Index view trở nên đơn giản hơn. Chúng ta sẽ loại bỏ các thẻ *<html>, <head>* và *<body>*. Các thẻ này giờ đây chuyển sang trang layout. Dòng đầu tiên của code là khối Razor, chúng ta định nghĩa trang layout sẽ được dùng cho view này. Chạy ứng dụng và nhìn kết quả.

A screen shot of a computer

Description automatically generated

Ảnh 21: Index view có sử dụng \_Layout

Giờ hãy thêm một Action method khác tên *AboutUs* trong *HomeController*.

|  |
| --- |
| **public** **IActionResult** AboutUs()  {  **return** View();  } |

Giờ hãy thêm một View và chọn Layout:

|  |
| --- |
| @{  Layout = "~/Views/Shared/\_Layout.cshtml";  }    <**h2**>About Us</**h2**> |

Giờ hãy chạy ứng dụng và vào URL: /Home/AboutUs. Bạn sẽ thấy nội dung đã thay đổi khi header và footer đã được thêm vào từ Layout



**Đường dẫn Layout**

Chúng ta sử dụng đường dẫn đầy đủ *"~/Views/Shared/\_Layout.cshtml"* trong view *AboutUs*, trong khi Index action lại sử dụng đường dẫn là *"\_Layout"*. Khi một đường dẫn partial được cung cấp cho Razor view engine tìm kiếm file layout trong thư mục *Views/{Controllers}* trước tiên, và theo sau đó là thư mục *Shared*. Quá trình này được gọi là tìm kiếm View.

## 16.4. Section

ASP.NET Core cho phép bạn tạo một hoặc nhiều section cho layout. Section cần một cái tên và một thuộc tính chỉ ra xem nó có bắt buộc phải có trong các view con không. Phương thức *RenderSection(string name, bool required:false)* định nghĩa tên section.

|  |
| --- |
| @**RenderSection**("footer", required: false) |

**Thêm Section vào Layout**

Mở layout ra và dán dòng lệnh sau vào sau nội dung:

|  |
| --- |
| <**div**>  @**RenderSection**("Test", required: false)  </**div**> |

Chúng ta định nghĩa một section tên *"Test"* với tham số required là false tức là không yêu cầu bắt buộc phải có section này trong view con.

**Định nghĩa Section trong view**

Giờ hãy vào trang View của *Index* và dán đoạn sau:

|  |
| --- |
| @{  Layout = "\_Layout";  }  <**p**>Here goes the content</**p**>    @**section Test**  {  <**p**>Test Section</**p**>  } |

Các section được định nghĩa với một khối lệnh Razor là *@section* bằng tên nó đi kèm. Chạy ứng dụng và kiểm tra xem các section đó có hiển thị sau content.

**Tạo section bắt buộc**

Thay đổi tham số *required:true* khi định nghĩa section trong layout như dưới đây:

|  |
| --- |
| @**RenderSection**("Test", required: true) |

Giờ hãy chạy ứng dụng và vào URL */Home/AboutUs* sau đó bạn nhìn thấy lỗi sau:

*InvalidOperationException: The layout page ‘/Views/Shared/\_Layout.cshtml’ cannot find the section ‘Test’ in the content page ‘/Views/Home/AboutUs.cshtml’.*

Điều này xảy ra vì chúng ta không định nghĩa section *"Test"* trong view *AboutUs.*

**\_ViewStart**

Trong ví dụ trên, chúng ta định nghĩa layout và dùng nó cho các view. Định nghĩa layout trong tất cả các view thật khó để bảo trì. Cách đơn giải nhất là định nghĩa layout trong *\_ViewStart.cshtml*.

Mục đích của file này là cài đặt giá trị mặc định cho các View khác trong folder và folder con của nó. Giờ hãy đến thư mục Views và tạo *\_ViewStart.cshtml* trong thư mục Views và dán đoạn sau:

|  |
| --- |
| @{  Layout = "\_Layout";  } |

Hãy bỏ hoặc comment đoạn code trên trong view *AboutUs*. Chạy ứng dụng và kiểm tra lại.

**Loại bỏ Layout từ view**

Vì chúng ta đã thêm file *\_ViewStart.cshtml* vào thư mục Views nên layout giờ đã có thể áp dụng cho tất cả các View. Trong trường hợp bạn không muốn sử dụng layout cho view nào bạn chỉ cần đặt giá trị Layout là null.

|  |
| --- |
| @{  Layout = null;  } |

hoặc đặt giá trị cho Layout là một Layout khác:

Bạn có thể tạo một file \_ViewStart tách biệt trong thư mục Controller, vậy là bạn có thể ghi đè file \_ViewStart từ thư mục cha.

# Chương XVII. ViewBag và ViewData

View cần lấy dữ liệu từ Controller. Một trong những cách truyền dữ liệu sang View là sử dụng đối tượng *ViewData* hoặc *ViewBag*. Bài này chúng ta sẽ tìm hiểu cách dùng *ViewBag* và *ViewData*.

## 17.1. ViewData

### 17.1.1. ViewData là gì?

*ViewData* là một thuộc tính của Controller base class, nó trả về một đối tượng *ViewDataDictionary*. *ViewDataDictionary* như tên của nó là một đối tượng dictionary cho phép lưu dữ liệu dạng key-value. Key phải là một chuỗi không phân biệt chữ hoa thường. Để truyền dữ liệu vào view bạn cần gán giá trị vào dictionary sử dụng key. Bạn có thể lưu bất kỳ số lượng key-value nào cần thiết trong *ViewData*.

*ViewData* truyền dữ liệu sang View từ Controller. Khi bạn gọi phương thức View trong Controller action, *ViewData* sẽ tự động gán vào View. Trong View bạn có thể truy cập giá trị được lưu trong *ViewData* cũng sử dụng key. Dữ liệu được lưu trong *ViewData* tồn tại chỉ trong request đó. Khi View được tạo xong cho client thì đối tượng *ViewData* đó cũng bị hủy.

### 17.1.2. Sử dụng ViewData như thế nào?

Trong ví dụ dưới đây nêu cách dùng *ViewData* trong Controller action:

|  |
| --- |
| **public IActionResult** SomeAction()  {  **ViewData**["Greeting"] = "Hello World!";  **return View**();  } |

Trong ví dụ trên, chúng ta thêm chữ "Hello World" vào *ViewData* sử dụng key là "Greeting". Bạn có thể gọi phương thức dưới đây trong View để nhận giá trị từ key "Greeting":

|  |
| --- |
| @**ViewData**["Greeting"] |

**Gán đối tượng cho ViewData**

Trong ví dụ trên, chúng ta lưu chuỗi dữ liệu vào *ViewData*. Chuỗi dữ liệu có thể được dùng trực tiếp mà không cần chuyển kiểu. Bạn có thể lưu bất cứ kiểu nào của dữu liệu như kiểu số nguyên, kiểu logic hay đối tượng trong *ViewData*.

Để sử dụng như một kiểu dữ liệu khác thì bạn cần phải chuyển kiểu giá trị từ *ViewData* sang kiểu tương ứng khi bạn dùng nó. Ví dụ dưới đây nêu cách dùng *ViewData* để gán đối tượng từ Controller sang View:

|  |
| --- |
| **public IActionResult** Index()  {  **ViewData**["Greeting"] = "Hello World";  **ViewData**["Product"] = **new** ProductModel()  {  ProductID=1,  Name = "Samsung galaxy Note",  Brand = "Samsung",  Price = 19000  };    **return View**();  } |

Trong View, chúng ta lấy sản phẩm từ *ViewData* và chuyển sang kiểu *ProductModel* và sử dụng nó:

|  |
| --- |
| @{  // Since Product isn't a string, it requires a cast.  var product = ViewData["Product"] as ProductModel;  }    @**ViewData**["Greeting"]!    @product.ProductID<br>  @product.Name<br>  @product.Brand<br>  @product.Price<br> |

Bạn có thể dùng *ViewData* để gán dữ liệu từ Controller vào View bao gồm cả Partial View và Layout.

## 17.2. ViewBag

### 17.2.1. ViewBag là gì?

Bạn đã thấy từ ví dụ trước chúng ta có thể lưu bất cứ thứ gì trong *ViewDataDictionary*, nhưng để truy cập được bất cứ thứ gì trong *ViewData* chúng ta cần phải chuyển kiểu dữ liệu. *ViewBag* sử dụng kiểu động (dynamic) mà chúng ta đã có trong phiên bản C# 4.0. Nó là một vỏ bọc của *ViewData* và cung cấp thuộc tính động cho *ViewData*.

### 17.2.2. Sử dụng ViewBag

*ViewBag* có thể tiện dụng hơn để làm việc mà không cần chuyển kiểu. Sử dụng *ViewBag* như sau:

|  |
| --- |
| **public IActionResult** SomeAction()  {  **ViewBag**.Greeting = "Hello";  **ViewBag**.Product = **new ProductModel**()  {  ProductID = 1,  Name = "Samsung galaxy Note",  Brand = "Samsung",  Price = 19000  };    **return View**();  } |

Sau đó bạn sử dụng trong View:

|  |
| --- |
| @**ViewBag**.Greeting World <br>    @**ViewBag**.product.ProductID<br>  @**ViewBag**.product.Name<br>  @**ViewBag**.product.Brand<br>  @**ViewBag**.product.Price<br> |

## 17.3. So sánh ViewData và ViewBag

Cả *ViewBag* và *ViewData* đều sử dụng *ViewDataDictionary* bên dưới. Vì thế bạn có thể sử dụng cả hai hoặc kết hợp chúng khi đọc hoặc ghi chúng. Ví dụ:

|  |
| --- |
| **ViewData**["Greeting"] = "Hello World"; |

Và nhận giá trị ở view sử dụng ViewBag:

|  |
| --- |
| @**ViewBag**.Greeting |

**Sự khác nhau giữa ViewData và ViewBag**

*ViewData* sử dụng cú pháp *Dictionary* để truy cập giá trị trong khi *ViewBag* sử dụng cú pháp truy cập giống truy cập thuộc tính của đối tượng. *ViewData* dẫn xuất từ *ViewDataDictionary*, nó có thuộc tính của dictionary như ContainsKey, Add, Remove và Clear.

*ViewBag* thì nhận từ *DynamicViewData* và nó cho phép tạo động các thuộc tính sử dụng dấu chấm *(@ViewBag.SomeKey = <giá trị>)* và không cần chuyển kiểu. Cú pháp của *ViewBag* giúp thêm giá trị nhanh hơn trong Controller và View.

*ViewData* cho phép sử dụng khoảng trắng trong Key vì nó là một chuỗi. Ví dụ *ViewData["Some Key With Namespace"]*. Nhưng *ViewBag* thì không thể.

*ViewData* cần phải chuyển kiểu dữ liệu khi không phải là một chuỗi. Nó cũng cần phải kiểm tra giá trị null để tránh lỗi. Check null đơn giản hơn trong *ViewBag*.

Ví dụ: *@ViewBag.Person?.Name*.

**Khi nào sử dụng ViewBag và ViewData**

*ViewData* và *ViewBag* đều là các tùy chọn hữu ích khi bạn muốn gán lượng dữ liệu nhỏ từ Controller sang View. Chọn cái nào thường hay tùy thuộc thói quen. Điểm yếu của cả *ViewBag* và *ViewData* là chúng giải quyết vấn đề động ở thời điểm runtime. Nó không kiểm tra kiểu ở lúc biên dịch (complie time). Vì thế tăng khả năng gây lỗi.

*ViewData* và *ViewBag* có thể truyền dữ liệu từ Controller sang View. Nó không thể truyền ngang từ Controller này sang Controller kia được.

# Chương XVIII. Model và ViewModel

Model có nghĩa rộng, nó là bất cứ cái gì tùy thuộc vào bạn muốn nó làm gì?. Trong ngữ cảnh của ASP.NET MVC thì Model có thể là một Domain Model, View Model hay là một Edit model.

## 18.1. Model là gì?

Model là tập hợp các đối tượng chứa dữ liệu của ứng dụng có thể chứa thêm cả các logic nữa. Model chia làm một số loại dựa trên công dụng và nơi chúng sử dụng. Có 3 loại mục đích chính:

* Domain Model
* View Model
* Edit Model

### 18.1.1. Domain Model

Một Domain Model thể hiện một đối tượng trong database. Domain model thường có một mối quan hệ 1-1 với một bảng trong cơ sở dữ liệu. Domain Model liên quan đến tầng truy cập dữ liệu (DAL) trong ứng dụng. Nó nhận từ cơ sở dữ liệu hoặc một nơi nào đó lưu dữ liệu bởi tầng truy cập dữ liệu. (DAL). Domain Model cũng được hiểu như entity model hay data model.

Ví dụ chúng ta có bảng Product như sau:



Nó được đại điện với Product Model:

|  |
| --- |
| **public class** Product  {  **public int** ProductId { get; set; }  **public string** Name { get; set; }    **public Decimal** Price { get; set; }  **public int** Rating { get; set; }    **public Brand** Brand { get; set; }  **public Supplier** Supplier { get; set; }  } |

### 18.1.2. View Model

*ViewModel* được tham chiếu đến các đối tượng chứa dữ liệu cần cho việc hiển thị cho người dùng. *ViewModel* liên quan đến tầng hiển thị của ứng dụng. Nó được định nghĩa dựa trên cách thức dữ liệu được hiển thị cho người dùng hơn là cách chúng được lưu trữ ra sao?

Ví dụ trong Product Model ở trên, người dùng cần hiển thị Brand Name và Supplier Name thay vì Brand ID và Supplier ID. Vì thế *ViewModel* trở thành:

|  |
| --- |
| **public class** ProductViewModel  {  **public int** ProductId { get; set; }  **public string** Name { get; set; }    **public Decimal** Price { get; set; }  **public int** Rating { get; set; }    **public string** BrandName { get; set; }  **public string** SupplierName { get; set; }    **public string** getRating()  {  **if** (Rating == 10)  {  **return** "\*\*\*\*\*";  }  **else if** (Rating >=8 )  {  **return** "\*\*\*\*";  }  **else if** (Rating >= 6)  {  **return** "\*\*\*";  }  **else if** (Rating >= 4)  {  **return** "\*\*";  }  **else**  {  **return** "\*";  }  }  } |

Chúng ta bỏ Brand và Supplier thay vào đó trả về Brand Name và Supplier Name. View Model cũng có thể có View liên quan đến logic như là hiển thị Rating cho người dùng. Bạn không nên đặt bất cứ logic nào khác ngoài logic hiển thị trên view vào View Model.

### 18.1.3. Edit Model

Edit Model hoặc Input Model đại điện dữ liệu cần để người dùng thay đổi hoặc thêm mới. Yêu cầu UI của Product cần chỉnh sửa khác với yêu cầu xem.

Ví dụ trong Product Model ở trên, người dùng cần hiển thị danh sách Brand và Supplier, trong khi thêm mới hay chỉnh sửa sản phẩm. Vì thế model trở thành:

|  |
| --- |
| **public class** ProductEditModel  {  **public int** ProductId { get; set; }    [**Required**(ErrorMessage = "Product Name is Required")]  [**Display**(Name = "Product Name")]  **public string** Name { get; set; }    **public Decimal** Price { get; set; }  **public int** Rating { get; set; }    **public List<Brand>** Brands { get; set; }  **public List<Supplier>** Suppliers { get; set; }    **public int** BrandID { get; set; }  **public int** SupplierID { get; set; }  } |

## 18.2. Model trong MVC Design Pattern

MVC Design Pattern là một pattern cho tầng hiển thị. Model trong MVC Design Pattern viết tắt của View Model và Edit Model. Hầu hết mọi người sử dụng từ View Model để chỉ cả: View Model và Edit Model.

**Lợi ích của View Model**

*ViewModel* rất hữu dụng khi bạn có một UI phức tạp, khi mà dữ liệu cần lấy ra từ vài Domain Model. Vì View Model được độc lập với Domain Model, nên việc này rất mềm dẻo và linh hoạt cho việc sử dụng nó. *ViewModel* giúp ứng dụng bảo mật hơn vì bạn không phải chìa các thuộc tính nhạy cảm và bí mật từ Domain Model ra như UserRole, IsAdmin...

## 18.3. Best practice với ViewModel

**Giữ Domain Model và View Model tách bạch**

Tránh sử dụng Domain model thay cho View Model. Bạn có thể chìa những thuộc tính nhạy cảm ra ngoài cho View. Domain Model thường gắn chặt vào database để sử dụng tầng DAL. Vì thế việc chìa các thuộc tính có thể sửa hay thêm mới vào database là nguy hiểm.

**Tạo Strongly Typed Views (View luôn có khai báo ViewModel tương ứng)**

Trong Strongly Typed Views, bạn hãy để View biết kiểu của viewModel được gán cho nó. Với strongly typed view, bạn sẽ có thể có sự trợ giúp gợi ý thuộc tính từ Visual Studio và dễ tìm lỗi nếu có trong quá trình phát triển.

**Sử dụng Data Annotation cho Validation**

Sử dụng Data Annotation để khai báo cho thuộc tính của viewModel và giúp tận dụng cơ chế client validation trong ASP.NET Core.

**Chỉ đặt các dữ liệu cần thiết trong ViewModel**

Giữ ViewModel nhỏ nhất có thể. Chỉ đặt các trường thực sự cần thiết cho việc hiển thị trong ViewModel.

**Sử dụng một Mapper để chuyển Model sang ViewModel**

Model nhận từ cơ sở dữ liệu cần được map sang ViewModel. Bạn có thể sử dụng AutoMapper để thực hiện điều này.

**ViewModel có thể chứa các logic chỉ cho view**

Về ý tưởng, ViewModel có thể chứa các dữ liệu và không có logic. Nhưng bạn có thể thêm một số logic đặc thù cho ViewModel.

**Sử dụng 1 ViewModel cho 1 View**

Tạo một ViewModel cho một View. Sẽ dễ bảo trì và dễ tìm lỗi.

**Đồng nhất**

Sử dụng ViewModel ngay cả cho các kịch bản đơn giản. Nó giúp dễ bảo trì và đảm bảo tính đồng nhất cho toàn ứng dụng.

## 18.4. Truyền dữ liệu từ Controller sang View

*ViewModel* tham chiếu đến các đối tượng chứa dữ liệu cần thiết để hiển thị cho người dùng. Nó có thể chứa dữ liệu từ một hoặc nhiều entity trong cơ sở dữ liệu. View Model nên chứa tất cả dữ liệu cần thiết để hiển thị cho người dùng.

*ViewModel* được gán từ Controller sang View bằng *ViewBag* hoặc *ViewData*. *ViewData* trả về một *ViewDataDictionary*, trong khi *ViewBag* chỉ là một lớp bao (wrapper) của *ViewData*, nó cung cấp thuộc tính động. *ViewDataDictionary* chứa một đối tượng dictionarychung và chứa một thuộc tính đặc biệt gọi là Model. Model cho phép chúng ta gán một *ViewModel* sang View. Thuộc tính Model này cho phép chúng ta tạo một strongly typed view.

Giờ chúng ta hãy tạo một View hiển thị một đối tượng Customer. Để làm điều này chúng ta cần tạo một Customer view model.

Chuột phải vào thư mục *Models* và chọn *Add>Class*. Tên của class là *Customer.cs*. Và có các thuộc tính sau:

|  |
| --- |
| **namespace MVCCoreApp.Models**  {  **public class** Customer  {  **public int** CustomerID { get; set; }  **public string** Name { get; set; }  **public string** Address { get; set; }    **public Customer()**  {  CustomerID = 1;  Name = "Rahul Dravid";  Address = "Bangalore";  }  }  } |

**Sử dụng ViewBag để truyền ViewModel về View**

Vào *Index* action method trong *HomeController*. Tạo một thuộc tính động tên *Customer* trong *ViewBag* và gán thể hiện của Customer như dưới đây:

|  |
| --- |
| **public IActionResult** Index()  {  **ViewBag**.Customer = **new** Customer();  **return View**();  } |

Bạn có thể tham chiếu đến đối tượng customer sử dụng *ViewBag.Customer*. Vì bạn đang sử dụng *ViewBag* sẽ không cần phải ép kiểu sang kiểu tương ứng.

|  |
| --- |
| @{  **Customer** customer = **ViewBag**.Customer;    <p>Id :@**customer**.CustomerID </p>  <p>Name :@**customer**.Name </p>  <p>Name :@**customer**.Address </p>  } |

Một hạn chế của cách trên là nó không kiểm tra kiểu lúc biên dịch. Nếu bạn sử dụng *ViewBag.Customers* thì nó cũng không phát hiện ra sai key lúc biên dịch mà phải đợi lúc chạy mới ra lỗi.

**Sử dụng thuộc tính Model để trả về ViewModel**

Như đã nhắc ở trên, *ViewDataDictionary* có một thuộc tính đặc biệt là Model, nó có thể được truy cập sử dụng ViewData. Chúng ta có thể gán Customer ViewModel về view sử dụng thuộc tính Model:

|  |
| --- |
| **public IActionResult** Method1()  {  **ViewData**.Model = **new** Customer();  **return View**();  } |

Bạn có thể truy cập Model trong View bằng cách sử dụng *ViewData.Model* hoặc Model (trả về ViewData.Model):

|  |
| --- |
| @{  Customer customer = Model;  //Or  //Customer customer =ViewData.Model;    <p>Id :@customer.CustomerID </p>  <p>Name :@customer.Name </p>  <p>Name :@customer.Address </p>  } |

Cách này cũng có hạn chế tương tự như cách trước. Nhưng sử dụng thuộc tính động định nghĩa trước Model cho chúng ta tùy chọn để tạo ra Strongly Typed View.

**Strongly Typed View là gì?**

View mà gắn vào một kiểu cụ thể của ViewModel thay vì một thuộc tính động gọi là strongly typed view. Trong ví dụ trên, chúng ta đang gắn Customer ViewModel vào View sử dụng ViewBag.Customer hoặc ViewData.Model. Trình biên dịch không biết gì về kiểu của model. Trong strongly typed view, chúng ta để cho View biết được kiểu của ViewModel được gán cho nó.

**Khai báo @model**

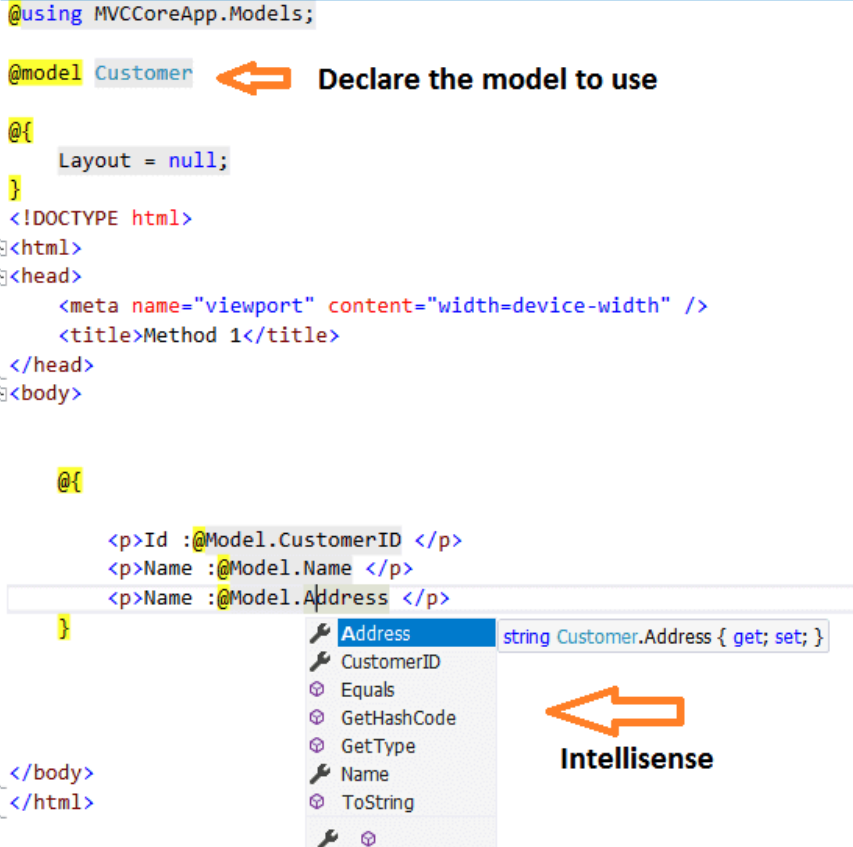
Strongly typed view được tạo sử dụng khai báo @model. Khai báo @model được đặt trên đầu của file view chỉ ra kiểu của ViewModel được gán.

|  |
| --- |
| @model Customer |

Sau đó tham chiếu trực tiếp đến model trong View:

|  |
| --- |
| @{  <p>Id :@Model.CustomerID </p>  <p>Name :@Model.Name </p>  <p>Name :@Model.Address </p>  } |

Với Strongly typed view bạn sẽ có thể có gợi ý từ Visual Studio và kiểm tra kiểu hay kiểm tra lỗi ở thời điểm biên dịch. Vì chỉ có 1 thuộc tính Model nên chỉ có 1 ViewModel được gán cho 1 view.



**Model và model**

Rất dễ nhầm Model và model. Khai báo model là khai báo được dùng để khai báo kiểu của *ViewModel*. Còn Model là một biến sử dụng để truy cập vào *ViewModel*. Kiểu của Model được khai báo tùy thuộc vào khai báo *@model* ở trên.

Trong thực tế, tất cả dữ liệu được gán vào View qua *ViewBag*. Khi bạn sử dụng khai báo model, Razor engine tạo ra một thuộc tính tên *Model*. *Model* sau đó trả về kiểu được khai báo.

Ví dụ chúng ta khai báo:

|  |
| --- |
| @model Customer |

Nó sẽ hiểu là:

|  |
| --- |
| Customer Model; |

**Cách đúng để gán ViewModel sang View**

Cách được khuyến nghị để gán dữ liệu từ *ViewModel* sang View là sử dụng phương thức View. Phương thức View nhận model như một tham số và tự động gán vào *ViewData.Model*.

|  |
| --- |
| **public IActionResult** CorrectWay()  {  **Customer** customer = **new** Customer();  **return View**(customer);  } |

# Chương XIX. Xây dựng HTML Form [Phụ lục 5]

## 19.1. Tạo mới project

Tạo mới một project tên là *ASPNetCoreForms*. Chọn mẫu là ASP.NET Core Web Application template. Template này sẽ có sẵn trong phần Visual C# -> .NET Core.

## 19.2. Cài đặt MVC Middleware

Mở file *Startup.cs* ra và cài đặt liên quan đến MVC theo như bài viết Xây dựng ứng dụng ASP.NET Core MVC đầu tiên.

|  |
| --- |
| public void ConfigureServices(IServiceCollection services)  {  services.AddMvc();  }    public void Configure(IApplicationBuilder app, IHostingEnvironment env)  {  if (env.IsDevelopment())  {  app.UseDeveloperExceptionPage();  }  app.UseMvcWithDefaultRoute();  } |

## 19.3. Tạo một View model

Chuột phải vào project tạo một thư mục là *Models*. Sau đó tạo một class tên *Product.cs* ở trong thư mục này:

|  |
| --- |
| namespace ASPNetCoreForms.Models  {  public class ProductEditModel  {  public int ID { get; set; }  public string Name { get; set; }  public decimal Rate { get; set; }  public int Rating { get; set; }  }    } |

Chú ý là chúng ta đặt tên *ProductEditModel* vì nó dùng để làm model cho form nhập dữ liệu.

## 19.4. Tạo một Controller

Bước tiếp theo là tạo một Controller. Tạo một thư mục tên Controllers vào thư mục gốc rồi click chuột phải vào nó chọn Add-> Controller. Chọn MVC Controller - Empty và click Add. Tên của Controller là HomeController. Thao tác này sẽ tạo controller với một Index action method. Chuột phải vào Index method chọn Add View để tạo một view Index.

**Tạo một Action Method**

Giờ hãy mở *HomeController.cs* và sử dụng lệnh using để thêm Models:

|  |
| --- |
| using ASPNetCoreForms.Models; |

Sau đó tạo một Action method:

|  |
| --- |
| *[HttpGet]*  public IActionResult Create()  {  return View();  }    *[HttpPost]*  public IActionResult Create(ProductEditModel model)  {  string message = "";    if (ModelState.IsValid)  {  message = "product " + model.Name + " Rate " + model.Rate.ToString() + " With Rating " + model.Rating.ToString() + " created successfully";  }  else  {  message = "Failed to create the product. Please try again";  }  return Content(message);  } |

Chúng ta đã định nghĩa 2 action method với tên là Create. Các action method này được đánh dấu với 2 Action verb là *HttpGet* và *HttpPost*. Bằng cách sử dụng Action verb chúng ta đảm bảo rằng action method đầu tiên tên Create sẽ chỉ xử lý các HTTP Get request và cái thứ 2 cũng tên Create nhưng sẽ chỉ xử lý ác request kiểu HTTP Post. Phương thức Create chấp nhận *ProductEditModel* làm đầu vào.

|  |
| --- |
| public IActionResult Create(ProductEditModel model) |

*ProductEditModel* chứa các giá trị khi form được submit. Nó là quá trình tự động map các giá trị từ form vào model. Quá trình này chạy ngầm và nó gọi là Model Binding.

Quá trình Model binding trong ASP.NET Core map dữ liệu từ HTTP Request vào tham số của Action method. Các tham số này có thể là kiểu đơn giản hoặc là một đối tượng phức tạp.

Tiếp theo chúng ta sử dụng *ModelState.IsValid* để kiểm tra nếu *ProductEditModel* được nhận có hợp lệ không?

|  |
| --- |
| if (ModelState.IsValid) |

Cơ chế ASP.NET Core Validation kiểm tra dữ liệu được submit lên từ form có bất cứ lỗi validation nào không. Nó sử dụng annotation được định nghĩa trong *ProductEditModel* để kiểm tra. Validation được thực hiện trước khi gọi action method.

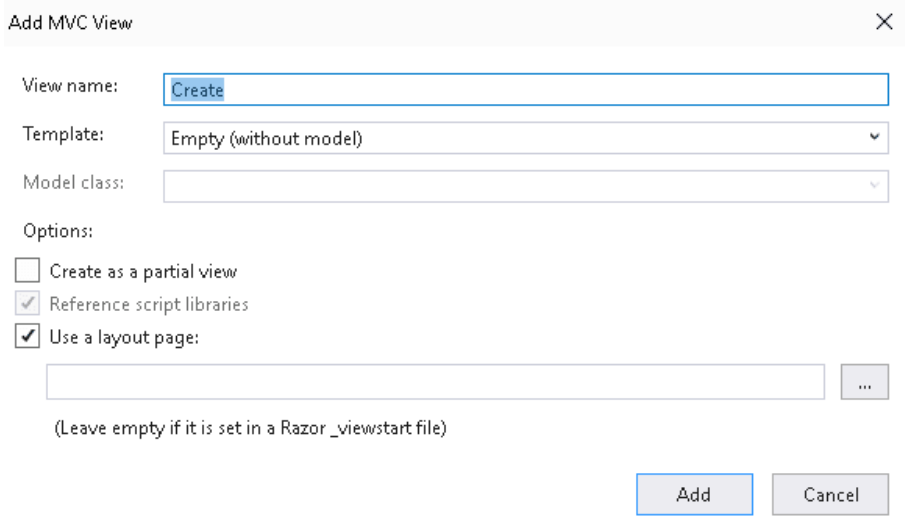
Cơ chế validation sẽ cập nhật đối tượng ModelState. Nếu không có lỗi nào thì thuộc tính IsValid sẽ được cập nhật là true. Nếu có bất cứ lỗi nào thì nó sẽ được update là false.

Cuối cùng chúng ta trả về thông tin lỗi cho client thôi qua ContentResult.

|  |
| --- |
| return Content(message); |

## 19.5. Tạo một View

Chọn phương thức muốn tạo sau đó click chuột phải và chọn Add View.



Nó sẽ tạo một file view tên là *Create.cshtml* trong thư mục Views.

**Tạo một form đơn giản**

Mở file *Create.cshtml* và thêm mới đoạn code:

|  |
| --- |
| @{  ViewData["Title"] = "Create";  }    <h2>Create</h2>    <form action="/home/create" method="post">  <label for="Name">Name</label>  <input type="text" name="Name" />    <label for="Rate">Rate</label>  <input type="text" name="Rate" />  <label for="Rating">Rating</label>  <input type="text" name="Rating" />  <input type="submit" name="submit" />  </form> |

Đây là một form đơn giản.

**Form Action**

Đầu tiên chúng ta định nghĩa một thẻ HTML Form, nó được sử dụng để xây dựng nên HTML Form. Thuộc tính action chỉ ra đường dẫn sẽ gọi khi người dùng submit form.

|  |
| --- |
| <form action="/home/create" method="post"> |

Tiếp theo chúng ta có một số các trường text cho phép nhập giá trị như Name, Rate, Rating của ProductEditModel.

|  |
| --- |
| <label for="Name">Name</label>  <input type="text" name="Name" />  <label for="Rating">Rating</label>  <input type="text" name="Rating" /> |

Điều quan trọng ở đây là tên của thuộc tính trong input phải trùng với tên thuộc tính của ProductEditModel. Cơ chế Model Binding sử dụng tên của file để map giá trị từ form vào ProductEditModel khi form được submit.

Dòng cuối sẽ là một submit button:

|  |
| --- |
| <input type="submit" name="submit" /> |

Giờ hãy mở file Index.cshtml

|  |
| --- |
| @{  ViewData["Title"] = "Index";  }    <h2>Index</h2>    <a href="Home/Create">Create</a> |

Chúng ta chỉ cần thêm link trỏ đến Create Action method:

|  |
| --- |
| <a href="Home/Create">Create</a> |

## 19.6. Index View

Mở file *HomeController.cs* ra và chọn Index action method sau đó chuột phải và chọn Add View để tạo Index view. Copy đoạn code dưới đây:

|  |
| --- |
| @{  ViewData["Title"] = "Index";  }  <h2>ASP.NET Core MVC - Forms</h2>  <a href="Home/Create">Create</a> |

Cuối cùng hãy chạy ứng dụng. Click nút Create sau đó nó sẽ mở một form. Nhập giá trị bên dưới và click submit.

A screenshot of a computer

Description automatically generated

Các giá trị trong form được submit thông qua HTTP Post đến /Home/Create action method.

ASP.NET Core Model Binder sẽ tạo một thể hiện của ProductEditModel sau đó map mỗi giá trị trong input field vào thuộc tính của ProductEditModel.

Giờ hãy tạo một giá trị không hợp lệ. Model Binder sẽ gặp lỗi khi map giá trị từ form vào thể hiện của ProductEditModel và ModelState.IsValid sẽ là false.

A screenshot of a computer

Description automatically generated

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

## 20.1. Tag Helper là gì?

Tag Helper giúp chúng ta viết phần tử HTML trong Razor sử dụng cú pháp thân thiện với HTML. Nó nhìn như là HTML chuẩn vậy nhưng code được xử lý bởi Razor Engine trên server và nó tận dụng đươc các ưu điểm của việc xử lý phía server.

Razor được tạo sử dụng Tag Helper nhìn như phần tử HTML thuần. Nó thao tác với các phần tử HTML như thêm mới phần tử HTML hay thay thế các nội dung có sẵn bằng một cái mới.

Ví dụ, sử dụng thẻ Form Tag Helper, chúng ta có thể tạo ra thẻ <form> như dưới đây. Với các thuộc tính asp-action và asp-controller của Form Tag Helper:

|  |
| --- |
| <form asp-action="create" asp-controller="home"> |

Sẽ được gen ra HTML:

|  |
| --- |
| <form action="/home/create" method="post"> |

**Mục đích của Tag Helpers**

Bạn có thể tạo form mà không cần dùng Tag Helper (hoặc HTML Helper) như bài trước. Tuy nhiên Tag Helper sẽ giúp tạo ra view HTML đơn giản hơn dựa trên dữ liệu từ Model gắn vào nó. Ví dụ Label Tag Helper sẽ tạo ra tiêu đề dựa trên attribute Data Annotation trong View Model. Tương tự như thế thì Input Tag Helper sẽ tạo ra id, name, type của phần tử HTML dựa trên kiểu dữ liệu của Model và thuộc tính Data Annotation.

## 20.2. Sử dụng Tag Helper?

ASP.NET Core Tag Helper nằm trong thư viện *Microsoft.AspNetCore.Mvc.TagHelpers* bạn cần import thư viện này để sử dụng Tag Helper.

**Thêm Tag Helper sử dụng @addTagHelper**

Để sử dụng Tag Helper bạn cần thêm khai báo *@addTagHelper* vào view, nơi mà bạn muốn sử dụng.

|  |
| --- |
| @addTagHelper \*, Microsoft.AspNetCore.Mvc.TagHelpers |

Đoạn code trên sử dụng wildcard ("\*") để chỉ ra tất cả Tag Helper được thêm vào từ thư viện *Microsoft.AspNetCore.Mvc.TagHelpers*.

**Thêm Tag Helper toàn bộ các View**

Thêm *@addTagHelper* vào một view nào đó chỉ có tác dụng trên view đó. Bạn có thể thêm *@addTagHelper* vào *\_ViewImports.cshtml* để sử dụng Tag Helper trên toàn bộ các view của ứng dụng.

**Bỏ Tag Helper**

Đoạn code dưới đây loại bỏ tất cả tag helper từ assembly *Microsoft.AspNetCore.Mvc.TagHelpers* từ một view cụ thể:

|  |
| --- |
| @removeTagHelper "\*, Microsoft.AspNetCore.Mvc.TagHelpers" |

**Thêm một số Tag Helper chỉ định**

Thay vì add tất cả tag helper thì bạn có thể chọn ra một số cái mà bạn muốn dùng thôi:

|  |
| --- |
| @addTagHelper "Microsoft.AspNetCore.Mvc.TagHelpers.InputTagHelper, Microsoft.AspNetCore.Mvc.TagHelpers" |

**Tắt Tag Helper với !**

Bằng cách sử dụng ký tự ! trước mỗi phần tử HTML, bạn sẽ có thể vô hiệu hóa tag helper cho phần tử đó:

|  |
| --- |
| <!label asp-for="Name"></!label> |

Tag helper label này được vô hiệu hóa với đoạn code trên. Bạn phải áp dụng ký tự ! cho cả thẻ đóng và thẻ mở.

**Sử dụng @tagHelperPrefix để bật Tag Helper**

Thay vì vô hiệu hóa tag helper sử dụng ký tự !, bạn có thể sử dụng *@tagHelperPrefix*

|  |
| --- |
| @tagHelperPrefix th: |

Giờ thì tiền tố th: phải được chỉ ra cho tất cả các tag helper trên view, để bật tag helper cho nó:

|  |
| --- |
| <th:label asp-for="Name"></th:label> //Tag helper is enabled  <label asp-for="Address"></label> //Tag helper is disabled |

**Ví dụ về Tag Helper**

Chúng ta xây dựng một Form đơn giản. Hãy dùng Tag Helper. Đầu tiên mở *HomeController.cs* ra và thay đổi action method *Create*.

|  |
| --- |
| *[HttpGet]*  public IActionResult Create()  {  ProductEditModel model = new ProductEditModel();  return View(model);  } |

Thể hiện của *ProductEditModel* được gán vào *View* để tạo Strongly Typed View. Mở *Create.csthml* từ thư mục */Views/Home*.

### 20.2.1. Form Tag Helper

Form Tag Helper được bao bởi thẻ *<form>*. Form Tag Helper cung cấp một số thuộc tính phía server giúp chúng ta thao tác để tạo ra HTML. Một số thuộc tính đó là:

* *asp-controller*: Chỉ ra tên Controller sử dụng
* *asp-action*: Chỉ ra tên action method sử dụng
* *asp-area*: Chỉ ra tên Area sử dụng

Ví dụ:

|  |
| --- |
| <form asp-controller="Home" asp-action="Create"> |

Đoạn code trên sẽ biên dịch ra HTML thuần là:

|  |
| --- |
| <**form** **method**="post" **action**="/Home/Create">  <**input** **name**="\_\_RequestVerificationToken" **type**="hidden" **value**="CfDJ8PlIso5McDBOjgPkVg9O4mnNiAE8U0HkVlA9e-Mtc76u7fSjCnoy909Co49eGlbyJxpp-nYphF\_XkOrPo0tTGdygc2H8nCtZCcGURMZ9Uf01fPOg5jRARxTHXnb8N6yYADtdQSnJItXtYsir8GCWqZM" />  </**form**> |

Chú ý là Form Tag Helper nó sẽ tự động thêm Antiy-Forgery Token vào HTML tạo ra.

### 20.2.2. Label Tag Helper

Label Tag Helper được áp dụng cho phần tử label. Nó có một thuộc tính là asp-for. Sử dụng như sau:

|  |
| --- |
| <label asp-for="@Model.Name"></label> |

Nó sẽ dịch ra:

|  |
| --- |
| <label for="Name">Name</label> |

Tên của trường đó sẽ được lấy từ tên của thuộc tính trong Model hoặc từ Data Annotation của thuộc tính trong Model. Sử dụng từ khóa @Model là không bắt buộc. Bạn có thể chỉ ra tên thuộc tính trong Model luôn:

|  |
| --- |
| <label asp-for="Name"></label> |

Cái này bạn có thể đọc Strongly Typed View ở bài trước.

### 20.2.3. Input Tag Helper

Input tag helper áp dụng trên phần tử HTML là *<input>*. Nó chỉ có 2 thuộc tính server-side là:

* asp-for
* asp-format

**asp-for**

Thuộc tính này kết hợp với thuộc tính tương ứng trong View Model và tạo ra HTML dựa trên thông tin thuộc tính đó như kiểu, tên, và data annotation (chỉ dẫn trên thuộc tính của ViewModel).

|  |
| --- |
| <input asp-for=”<Expression Name>” /> |

Ví dụ:

Bạn có một model có thuộc tính Name:

|  |
| --- |
| public string Name { get; set; } |

Ở view sẽ có một phần tử input:

|  |
| --- |
| <input asp-for="Name" /><br /> |

Nó sẽ tạo ra một phần tử HTML như sau khi thực thi:

|  |
| --- |
| <input type="text" id="Name" name="Name" value="" /><br /> |

**Các thuộc tính của HTML Input**

Thuộc tính asp-for tự động tạo ra phần tử HTML dựa trên các thông số sau đây:

**(1) Kiểu chỉ ra trong HTML**

Type là thuộc tính được định nghĩa trong HTML và nó sẽ không bị ghi đè:

|  |
| --- |
| [EmailAddress]  public string Email { get; set; } |

View sẽ là:

|  |
| --- |
| <input type=”text” asp-for="Email" /><br /> |

HTML sẽ tạo ra một input text vì cái data annotation là [EmailAddress] nó không ảnh hưởng đến kết quả HTML tạo ra. Nó vẫn tạo ra như sau:

|  |
| --- |
| <input type="text" id="Email" name="Email" value="" /><br /> |

**(2) Dựa trên thuộc tính Data Annotaion**

Thuộc tính Data Annotation được áp dụng cho model để tạo ra thuộc tính type. Ví dụ ta có data annotation EmailAddress nó sẽ tự dịch ra type="email":

|  |
| --- |
| [EmailAddress]  public string Email { get; set; } |

Code view:

|  |
| --- |
| <input asp-for="Email" /><br /> |

HTML tạo ra sẽ là:

|  |
| --- |
| <input type="email" data-val="true" data-val-email="The Email field is not a valid e-mail address." id="Email" name="Email" value="" /><br /> |

Đây là danh sách các Data Annotation và các kiểu của input tạo ra tương ứng nếu sử dụng chúng:

|  |  |
| --- | --- |
| Attribute | Input Type |
| [EmailAddress] | type=”email” |
| [Url] | type=”url” |
| [HiddenInput] | type=”hidden” |
| [Phone] | type=”tel” |
| [DataType(DataType.Password)] | type=”password” |
| [DataType(DataType.Date)] | type=”date” |
| [DataType(DataType.Time)] | type=”time” |

**(3) Dựa trên .NET Type**

Nếu thuộc tính Data Annotation không được chỉ ra, thì Input Tag Helper sẽ sử dụng kiểu .NET của thộc tính đó để xác định kiểu của HTML tạo ra. Ví dụ:

Model:

|  |
| --- |
| public DateTime DateOfJoining { get; set; } |

View:

|  |
| --- |
| <input asp-for="DateOfJoining" /> |

HTML:

|  |
| --- |
| <input type="datetime-local" data-val="true" data-val-required="The DateOfJoining field is required." id="DateOfJoining" name="DateOfJoining" value="0001-01-01T00:00:00.000" /><br /> |

Đây là danh sách các kiểu .NET hay dùng và input tạo ra tương ứng:

|  |  |
| --- | --- |
| NET Type | Input Type |
| Bool | type=”checkbox” |
| String | type=”text” |
| DateTime | type=”datetime-local” |
| Byte, int, Single, Double | type=”number” |
| decimal, double, float | type=”text” |

**asp-format**

Thuộc tính asp-format định dạng chuỗi cho thuộc tính input. Ví dụ bạn format số với 2 số thập phân đằng sau:

|  |
| --- |
| public decimal Balance { get; set; } |
| <input asp-for="Balance" asp-format="{0:N2}" /><br /> |

HTML tạo ra:

|  |
| --- |
| <input type="text" data-val="true" data-val-number="The field Balance must be a number." data-val-required="The Balance field is required." id="Balance" name="Balance" value="0.00" /><br /> |

## 20.3. Lợi ích của Tag Helper

**Thân thiện với cú pháp HTML**

Tag Helper nhìn như là phần tử HTML chuẩn. Các Front end Developer không cần học cú pháp C# hay Razor để thêm các phần tử này vào View. Vì thế nó dễ dàng đạt được tính chất chia để trị. Bạn có thể dễ dàng thêm CSS hoặc bất cứ thuộc tính HTML nào vào Tag Helper như là với HTML.

**Được hỗ trợ bởi IntelliSense**

Tag Helper cung cấp sự hỗ trợ bởi cơ chế gợi ý thông minh của Visual Studio. Ví dụ dưới khi chúng ta thấy gợi ý cho label của thuộc tính Rating.

**Code sạch hơn**

Code sẽ sạch và rõ ràng hơn sử dụng HTML Helper cũ. Không cần phải sử dụng ký tự @ để chuyển giữa C# và HTML.

**Dễ mở rộng hơn**

ASP.NET Core MVC cung cấp nhiều tag helper có sẵn giúp chúng ta tạo view. Nhưng nếu không có tag helper nào phù hợp với nhu cầu. Bạn cũng có thể tạo ra Tag Helper của riêng mình bằng cách mở rộng các Tag Helper có sẵn. Trong tương lai mình sẽ có bài viết về việc này.

**Danh sách các Tag Helper có sẵn**

Thư viện Microsoft.AspNetCore.Mvc.TagHelpers chứa nhiều các Tag Helper có sẵn cho các công việc thường dùng như tạo form, validate form, label, link...

|  |  |  |
| --- | --- | --- |
| Tag Helper | Tạo ra | Thuộc tính |
| Form Tag Helper | <form> | asp-action, asp-all-route-data, asp-area, asp-controller, asp-fragment, asp-host, asp-page, asp-page-handler,asp-protocol,asp-route, asp-route- |
| Anchor Tag Helpers | <a> | asp-action, asp-all-route-data, asp-area, asp-controller, asp-Fragment, asp-host, asp-page, asp-page-handler, asp-Protocol, asp-route, asp-route- |
| Cache Tag Helper | <cache> | enabled1,expires-after2,expires-on3,expires-sliding4,priority5,vary-by6 |
| Environment Tag Helper | <environment> | names, include, exclude |
| Image Tag Helper | <img> | append-version |
| Input Tag Helper | <input> | for |
| Label Tag Helper | <label> | for |
| Link Tag Helper | <link> | href-include, href-exclude, fallback-href, fallback-href-include, fallback-href-exclude, fallback-test-class, fallback-test-value, fallback-test-property, fallback-test-value, append-version |
| Options Tag Helper | <select> | asp-for, asp-items |
| Partial Tag Helper | <partial> | name,model,for,view-data |
| Script Tag Helper | <script> | src-include, src-exclude, fallback-src, fallback-src-include, fallback-src-exclude fallback-test, append-version |
| Select Tag Helper | <select> | for, items |
| Textarea Tag Helper | <textarea> | for |
| Validation Message Tag Helper | <span> | validation-for |
| Validation Summary Tag Helper | <div> | validation-summary |

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

## 21.1. Model Binding là gì?

Model Binding là cơ chế map dữ liệu được gửi qua HTTP Request vào các tham số của action method trong Controller. HTTP Request có thể chứa dữ liệu từ nhiều định dạng. Dữ liệu có thể chứa trong HTML Form. Nó có thể là một phần của route value hoặc trên query string hay có thể là một body của request.

Cơ chế ASP.NET Core model binding cho phép chúng ta dễ dàng bind các giá trị này vào các tham số của action method. Các tham số này có thể là kiểu nguyên thủy hoặc kiểu đối tượng phức tạp.

## 21.2. Lấy dữ liệu từ Form Data trong Controller

Trong bài Tag Helper, chúng ta đã tạo một form cơ bản cho phép nhận một đối tượng Product. Khi người dùng click nút Submit thì dữ liệu sẽ được post lên phương thức Create trên Controller. Trong project đó chúng ta cũng tạo một ProductEditModel class chứa chi tiết của sản phẩm cần được tạo hoặc chỉnh sửa:

|  |
| --- |
| public class ProductEditModel  {  public int ID{ get; set; }  public string Name { get; set; }  public decimal Rate { get; set; }  public int Rating { get; set; }  } |

Một form được tạo chứa 3 field: Name, Rate và Rating:

|  |
| --- |
| <form action="/home/Create" method="post">  <label for="Name">Name</label>  <input type="text" name="Name" />  <label for="Rate">Rate</label>  <input type="text" name="Rate" />  <label for="Rating">Rating</label>  <input type="text" name="Rating" />  <input type="submit" name="submit" />  </form> |

Action method Create trong HomeController:

|  |
| --- |
| *[HttpPost]*  public IActionResult Create(ProductEditModel model)  {  string message = "";  if (ModelState.IsValid)  {  message = "product " + model.Name + " created successfully" ;  }  else  {  message = "Failed to create the product. Please try again";  }  return Content(message);  } |

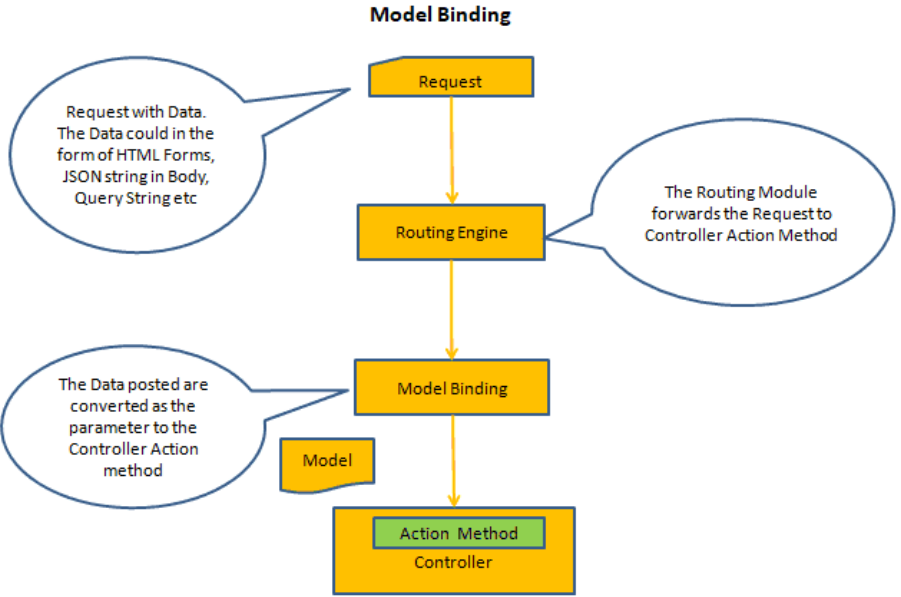
Một submit của form trên, các giá trị trong form sẽ tự động được map vào đối tượng *ProductEditModel* trong Action method của controller:

|  |
| --- |
| public IActionResult Create(ProductEditModel model) |

Cơ chế này tự động xảy ra đằng sau được gọi là Model Binding. Model Binder sẽ tìm các trường tương ứng giữa tham số *ProductEditModel* với trường trong form, route value hoặc query string...

## 21.3. Cơ chế Model Binding?

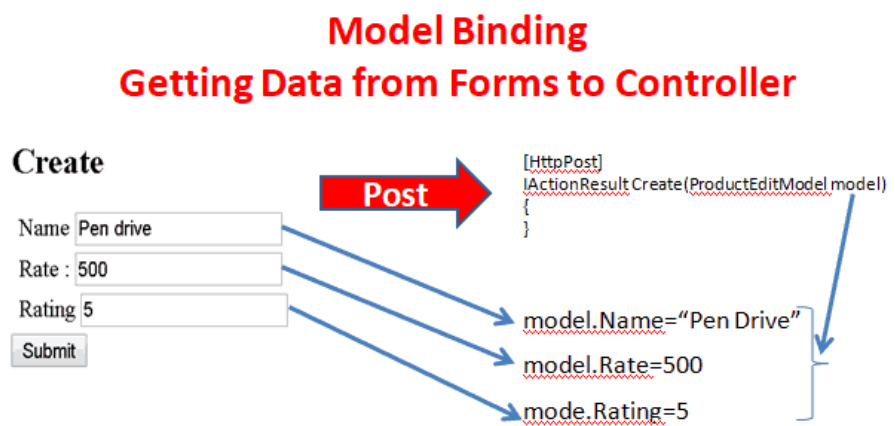
Hình dưới đây minh họa cơ chế làm việc của Model binding:



Ảnh 22: Cơ chế làm việc của Model binding

Khi người dùng click vào nút Submit thì một request Post được gửi lên server với Form Data, QueryString, Route Parameter... MVCRouteHandler của Routing Engine sẽ xử lý request đến và có trách nhiệm gọi action method tương ứng. Model Bindler sẽ được kích hoạt trước khi action method được gọi. Nó tìm dữ liệu thỏa mãn trong form data, querys tring và request parameter trong HTTP Request. Sau đso nó sẽ binding các giá trị vào tham số của action method qua tên.

Ví dụ, trường "name" trong form sẽ được map vào thuộc tính "Name" trong ProductEditModel. Rate trong form sẽ được map vào thuộc tính Rate...



Ảnh 23: Model Binding lấy dữ liệu từ form gửi đến Controller

Để Model binding làm việc đúng:

* Thuộc tính Name phải match với Request Data
* Các thuộc tính phải đặt public set

**Model Binder**

Model Binder có trách nhiệm gán dữ liệu vào các tham số của action method. Model Binder được tạo mởi model binder provider. Model binder phải được implement inteface *IModelBinderProvider*. Nghĩa là bạn có thể tạo một Model Binder của riêng mình hoặc mở rộng nó bằng cách triển khai inteface *IModelBinderProvider*. Custom model binder phải được đăng ký trong *ModelBinderProviders* trong *Startup.cs*.

|  |
| --- |
| services.AddMvc(options =>  {  options.ModelBinderProviders.Add(new CustomModelBinderProvider());  }); |

**ModelState**

Nếu Model binder không thành công trong việc bind dữ liệu từ Request vào thuộc tính model tương ứng, nó sẽ không đưa ra bất cứ thông báo lỗi nào. Nhưng nó sẽ update đối tượng *ModelState* với danh sách lỗi và set thuộc tính *IsValid* là false.

Vì thế kiểm tra *ModelState.IsValid* sẽ cho chúng ta thấy quá trình binding có thành công hay không.

Ví dụ: Trong ví dụ trên khi click nút submit mà không nhập bất cứ dữ liệu gì trên form thì kết quả sẽ ra validate thất bại và vì thế *ModelState.IsValid* sẽ là *false*.

## 21.4. Khi không dùng model binding

Trước khi chúng ta tìm hiểu sâu hơn về Model binding, chúng ta cần hiểu nếu không có model binding thì chúng ta sẽ truy cập đến dữ liệu từ request như thế nào? Xem lại code mà chúng ta đã tạo trong phần trước. Thêm mới action method *NoModelBinding*:

|  |
| --- |
| *[HttpPost]*  public IActionResult NoModelBinding()  {  ProductEditModel model = new ProductEditModel();  string message = "";  model.Name = Request.Form["Name"].ToString();  model.Rate = Convert.ToDecimal( Request.Form["Rate"]);  model.Rating =Convert.ToInt32( Request.Form["Rateing"]);  message = "product " + model.Name + " created successfully";  return Content(message);  } |

Và thay đổi thành:

|  |
| --- |
| <form action="/home/NoModelBinding" method="post"> |

**Truy cập trực tiếp đến query string**

Tương tự như thế, bạn có thể truy cập đến các giá trị trên query string sử dụng Request.Query. Lấy các giá trị trên query string.

Ví dụ, *Request.Query["id"].ToString()* trả về giá trị của id trên query string. Sử dụng *Request.QueryString.HasValue* sẽ cho bạn biết nếu có giá trị query string trên URL hiện tại hay không và Request.QueryString.Value sẽ trả về giá trị thô của query string.

**Truy cập trực tiếp đến Request Headers**

Tương tự như thế, bạn có thể sử dụng Request.Headers để truy cập các giá trị được gửi lên thông qua HTTP Header.

**Truy cập đến Route Data**

Để truy cập đến route bạn cần ghi đè phương thức *OnActionExecuting*:

|  |
| --- |
| using Microsoft.AspNetCore.Mvc.Filters;  public override void OnActionExecuting(ActionExecutingContext context)  {  string id = context.RouteData.Values["id"].ToString();  base.OnActionExecuting(context);  } |

Có thể thấy rằng có rất nhiều code để lấy giá trị được post lên HTTP Request. ASP.NET Core model binding sẽ làm giúp chúng ta các việc này mà dùng ít code hơn.

## 21.5. Các nguồn cho Model binding

Như đã nhắc đến trước đây, model binder có thể lấy dữ liệu từ rất nhiều nơi khác nhau. Đây là danh sách các nguồn dữ liệu theo thứ tự mà model binding sẽ tìm:

* HTML Form Value
* Route Value
* Query String

Model binder cũng có thể tìm dữ liệu từ các nguồn sau, nhưng chúng ta cần chỉ ra nguồn nào cần lấy một cách tường minh:

* Request Body
* Request Header
* Services

**Lấy dữ liệu từ Form và Query String**

Hãy thử binding action parameter với cả form và query string. Phương thức *FormAndQuery* sẽ như sau:

|  |
| --- |
| *[HttpGet]*  public IActionResult FormAndQuery()  {  return View();  }  *[HttpPost]*  public IActionResult FormAndQuery(string name,ProductEditModel model)  {  string message = "";  if (ModelState.IsValid)  {  message = "Query string "+ name + " product " + model.Name + " Rate " + model.Rate + " Rating " + model.Rating ;  }  else  {  message = "Failed to create the product. Please try again";  }  return Content(message);  } |

Chú ý rằng action method *FormAndQuery* có hai tham số name và *ProductEditModel*.

|  |
| --- |
| public IActionResult FormAndQuery(string name,ProductEditModel model) |

Tiếp theo, chúng ta tạo view *FormAndQuery* như sau:

|  |
| --- |
| <form action="/home/FormAndQuery/?name=Test" method="post">  <label for="Name">Name</label>  <input type="text" name="Name" />  <label for="Rate">Rate</label>  <input type="text" name="Rate" />  <label for="Rating">Rating</label>  <input type="text" name="Rating" />  <input type="submit" name="submit" />  </form> |

Form có tên trường là "name". Chúng ta cũng gửi "name=test" qua query string đến controller action:

|  |
| --- |
| <form action="/home/FormAndQuery/?name=Test" method="post"> |

Trong ví dụ trên, tham số "name" xuất hiện 2 lần như là một phần của query string. Khi form được submit, tham số "name" luôn map đến trường trong form chứ không phải query string. Bởi vì model binder luôn sử dụng thứ tự map dữ liệu theo thứ tự:

* Form Values
* Route Values
* Query Strings

Vì thế các giá trị trong form có trường name, tham số name luôn được gán giá trị. Chúng ta có thể thay đổi hành vi này bằng cách thêm thuộc tính [FromQuery]. Sử dụng [FromQuery] như sau:

|  |
| --- |
| public IActionResult FormAndQuery([FromQuery] string name,ProductEditModel model) |

Giờ nếu submit form thì tham số name sẽ lấy giá trị từ query string, trong khi *ProductEditModel* sẽ lấy giá trị từ form.

## 21.6. Điều khiển Binding Source

Trong ví dụ trước, chúng ta sử dụng *[FromQuery]* để bắt buộc model binder đổi hành vi mặc định và sử dụng query string làm nguồn cho binding. ASP.NET Core cung cấp cho chúng ta một số thuộc tính điều khiển và chọn nguồn nào sẽ được nhận khi binding: *[FromForm], [FromRoute], [FromQuery], [FromBody], [FromHeader], [FromServices]*

### 21.6.1. [FromForm]

*[FromForm]* ép model binder bind tham số vào các trường của HTML Form.

|  |
| --- |
| [HttpPost]  public IActionResult Create([FromForm] ProductEditModel model){  } |

### 21.6.2. [FromRoute]

*[FromRoute]* ép model binder bind tham số vào route data từ request.

Ví dụ: Tạo một action method FromRoute, cho phép một giá trị id và ProductEditModel. Chúng ta có 2 tham số id. MVC mặc định sẽ có tham số Id qua route và nó là tuỳ chọn. ProductEditModel cũng có thuộc tính id.

|  |
| --- |
| *[HttpGet]*  public IActionResult FromRoute()  {  return View();  }  *[HttpPost]*  public IActionResult FromRoute(string id, ProductEditModel model)  {  string message = "";  if (ModelState.IsValid)  {  message = "Route " + id + " Product id " + model.id + " product " + model.Name + " Rate " + model.Rate + " Rating " + model.Rating;  }  else  {  message = "Failed to create the product. Please try again";  }  return Content(message);  } |

Tạo ra view FromRoute

|  |
| --- |
| <form action="/home/FromRoute/Test" method="post">  <label for="id">id</label>  <input type="text" name="id" />  <label for="Name">Name</label>  <input type="text" name="Name" />  <label for="Rate">Rate</label>  <input type="text" name="Rate" />  <label for="Rating">Rating</label>  <input type="text" name="Rating" />  <input type="submit" name="submit" />  </form> |

Chú ý là chúng ta đang gọi controller action bằng cách sử dụng "test" như là giá trị route:

|  |
| --- |
| <form action="/home/FromRoute/Test" method="post"> |

Giờ khi submit form thì tham số id luôn map vào id từ form thay vì từ route value. Giờ hãy mở HomeController ra và áp dụng [FromRoute] trên tham số id.

|  |
| --- |
| public IActionResult FromRoute([FromRoute] string id, ProductEditModel model) |

Khi submit form thì id sẽ được nhận là "test"

### 21.6.3. Binding query string sử dụng [FromQuery]

*[FromQuery]* ép model binder bind giá trị vào tham số từ giá trị lấy từ query string.

### 21.6.4. Binding đến Request body sử dụng [FromBody]

*[FromBody]* ép model binder bind dữ liệu từ request body. Formatter sẽ chọn dựa trên content-type của request. Dữ liệu trong request body có nhiều format khác nhau như JSON, XML...Model binder sẽ tìm thuộc tính content-type trên header và chọn formatter để chuyển dữ liệu về kiểu mong muốn.

Ví dụ, khi content-type là 'application/json", model binder sử dụng *JsonInputFormatter* class để chuyển request body và map vào tham số.

### 21.6.5. Binding từ Request Header sử dụng [FromHeader]

[FromHeader] map các giá trị từ request header vào tham số của action:

|  |
| --- |
| public IActionResult FromHeader( [FromHeader] string Host)  {  return Content(Host);  } |

### 21.6.6. Vô hiệu hoá binding với [BindNever]

Để cho model binder biết rằng không bind cho thuộc tính nào đó.

|  |
| --- |
| [BindNever]  public int Rating { get; set; } |

Giờ thì model binder sẽ bỏ qua thuộc tính Rating ngay cả khi form có trường Rating.

### 21.6.7. Bắt buộc Binding với [BindRequuired]

Cái này ngược hẳn với [BindNever]. Trường nào được đánh dấu là BindRequired phải luôn hiển thị trên form và binding phải thực hiện nếu không thì *ModelState.IsValid* sẽ false.

# Chương XXII. Validation

Thường thì dữ liệu được nhập bởi người dùng sẽ không hợp lệ và không thể đưa vào cơ sở dữ liệu. Các dữ liệu được nhập có thể chứa một số lỗi chính tả hoặc người dùng cố tính nhập các dữ liệu không phù hợp. Vì thế chúng ta cần phải kiểm tra dữ liệu người dùng nhập vào trước khi lưu trữ vào cơ sở dữ liệu. ASP.NET Core cung cấp chó chúng ta thành phần gọi là Model Validator, nó dùng các attribute để kiểm tra dữ liệu trong model dễ dàng hơn. Chúng ta cũng tìm hiểu về ModelState và cách sử dụng nó.

## 22.1. Giới thiệu về Model Validation

Form Data post dữ liệu lên Controller action tự động được map vào các tham số của action bởi Model Binder như chúng ta đã tìm hiểu ở bài trước. Model cần kiểm tra dữ liệu đầu vào xem có hợp lệ không. Quá trình kiểm tra này có thể được hoàn thành bởi client trước khi gửi lên server hoặc server kiểm tra sau khi nhận được từ client. Cơ chế validation phía client (client-side validation) rất quan trọng vì nó giúp tăng trải nghiệm người dùng khi kiểm tra dữ liệu mà không cần chờ đến server nhưng phía server lại nên đảm bảo một lần nữa để các dữ liệu không hợp lệ không thể đưa vào hệ thống.

**Sự quan trọng của client-side validation**

* Giúp tăng sự trải nghiệm
* Vì việc kiểm tra tiến hành phía trình duyệt client nên phản hồi nhanh hơn và gần như là ngay lập tức
* Tiết kiệm tài nguyên server như là băng thông bằng cách giảm truy vấn đến server.

**Sự quan trọng của server-side validation**

Client-side validation cung cấp trải nghiệm người dùng tốt hơn nhưng không tin cậy. Nó có thể lỗi do một trong các lý do sau:

* Javascript có thể bị tắt ở trình duyệt
* Người dùng có thể gửi trực tiếp dữ liệu đến người dùng mà không sử dụng ứng dụng hoặc sử dụng một số các trình chỉnh sửa request có hại.
* Khi Javascript có lỗi thì kết quả là dữ liệu được đưa vào hệ thống mà có thể không hợp lệ
* Vì thế điều quan trọng là kiểm tra dữ liệu phải được thực hiện ở cả phía server, ngay cả bạn đã validate ở phía client.

**Kiểm tra model một cách tường minh**

Một khi bạn nhận model trong controller, bạn có thể kiểm tra model đó bằng cách viết code như sau:

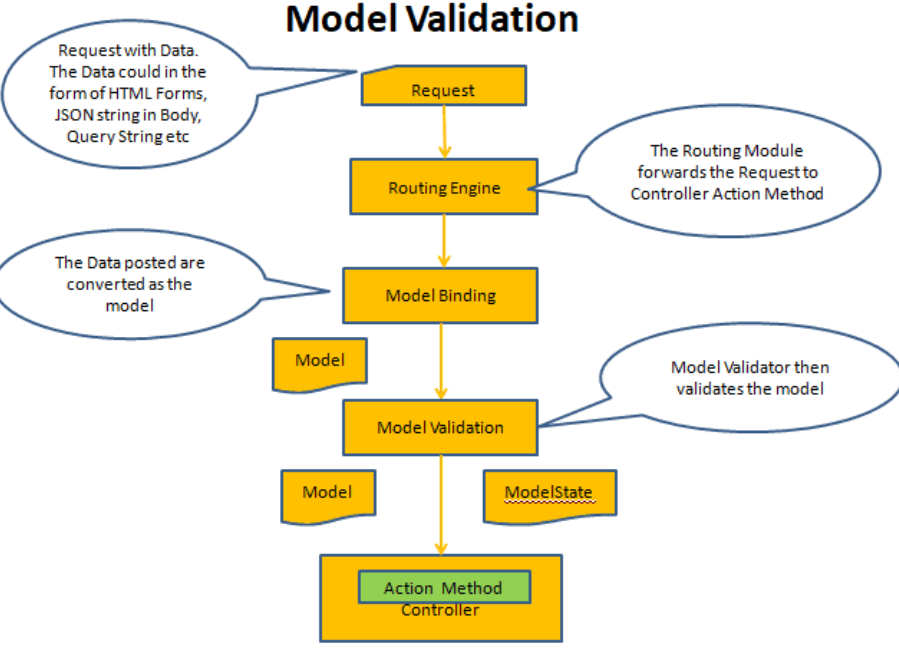
|  |
| --- |
| if (string.IsNullOrEmpty(model.Name))  {  //Validation Failed  //Send the list of errors to client  } |

Đoạn code trên đơn thuần chỉ kiểm tra xem thuộc tính name của model có rỗng hay null không. Đoạn code trên làm việc tốt, nhưng bạn sẽ phải làm nhiều lần tương tự nếu có nhiều thuộc tính trong model, khi đó bạn cần kiểm tra với các đoạn code giống nhau. Ở đây bạn cần xem làm sao để gửi các báo lỗi về client để hiển thị cho người dùng. Model Validator sẽ làm điều này cho bạn mà không cần viết các đoạn code thừa thãi.

## 22.2. Model Validation làm việc ra sao?

Chúng đã tìm hiểu cơ chế Model Binding làm việc ra sao trong ASP.NET Core. Vậy khi HTTP Request được đưa tới Model Binder thì nó sẽ được gọi trước khi truyền tham số vào Controller action. Model Binder sẽ không chỉ map dữ liệu vào action method mà nó còn kiểm tra chúng sử dụng Model Validator.

Model Validator chạy sau model binding và chạy một loạt các logic kiểm tra trên mỗi thuộc tính của model dựa trên các attribute bạn đặt cho các thuộc tính đó. Các attribute này gọi là Validation Attribute và chứa code sử dụng bởi Model Validator.



Ảnh 24: Model Validation

Tất cả các logic kiểm tra đều chạy phía server. ASP.NET Core có nhiều các attribute viết sẵn, bạn có thể thêm nó vào thuộc tính của model để cài đặt quy tắc kiểm tra. Các quy tắc này cũng được gọi là *DataAnnotations* và nằm trong namespace: *System.ComponentModel.DataAnnotations.* Bạn cần import nó vào ViewModel và sử dụng.

Ví dụ, bạn đặt trường bắt buộc phải nhập sẽ dùng *[Required]*

|  |
| --- |
| [Required]  Public string Name {get;set;} |

Model Binder không bắn ra bất cứ lỗi nào nếu quá trình kiểm tra không hợp lệ. Nhưng nó sẽ cập nhật đối tượng *ModelState* với danh sách lỗi và đặt thuộc tính *IsValid* là *false* trước khi gọi action method. Chúng ta cần kiểm tra *ModelState.IsValid* để biết xem quá trình kiểm tra có hợp lệ hay không để action method có thể trả về danh sách lỗi nếu cần.

|  |
| --- |
| if (ModelState.IsValid)  {  //Model is valid. Call Service layer for further processing of data  } else {  //Validation failed. Return the model to the user with the relevant error messages  } |

## 22.3. Cách sử dụng Validation Attributes

### 22.3.1. Cập nhật Model với Data Annotaion

Trong Model, thuộc tính thêm vào Data Annotation attribute được hiển thị bên dưới đây. Code sẽ được thêm vào attribute cho thuộc tính Name. Nó cũng được cài đặt thông báo trong trường hợp kiểm tra dữ liệu không hợp lệ với attribute đó:

|  |
| --- |
| [Required(AllowEmptyStrings =false,ErrorMessage ="Please enter the name")]  [StringLength(maximumLength:25,MinimumLength =10,ErrorMessage ="Length must be between 10 to 25")]  public string Name { get; set; } |

### 22.3.2. Hiển thị danh sách lỗi Validations

Trong view, sử dùng Validation Tag Helper để hiển thị lỗi cho người dùng. Tag helper asp-validation-summary giúp hiển thị danh sách các thông báo lỗi cho riêng Model validation trên form. Nó được gắn vào thẻ div và đặt trên cùng của form. asp-validation-for hiển thị danh sách lỗi cho thuộc tính Name ở phía bên phải của nó. Nó được gắn vào thẻ span và đặt ngay cạnh thuộc tính.

Validation tag helper thêm class field-validation-error và validation-summary-errors vào HTML nếu lỗi tìm thấy. Vì thế chúng a thêm các style CSS vào để hiển thị lỗi màu đỏ cho các class này. Bạn có thể đọc thêm phần Validation Tag Helper ở bài sau:

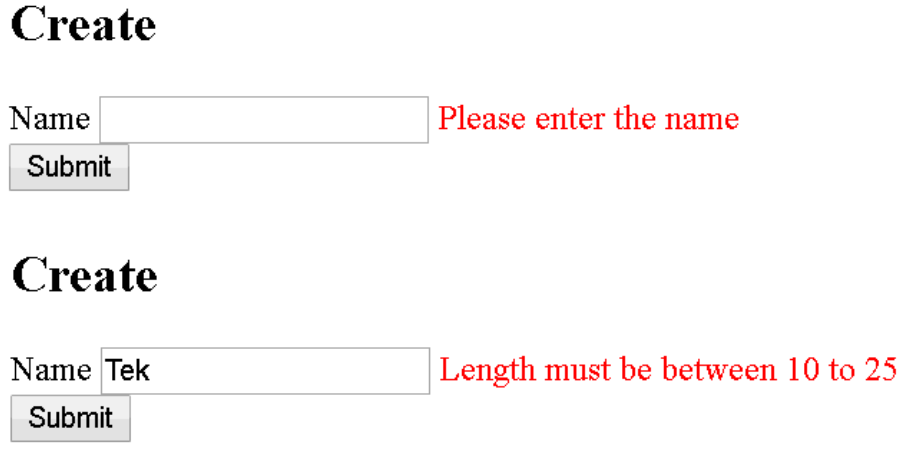
|  |
| --- |
| @addTagHelper \*, Microsoft.AspNetCore.Mvc.TagHelpers  @model ASPNetCoreForms.Models.ProductEditModel  @{  ViewData["Title"] = "Create";  }  <style>  .field-validation-error {  color: red  }  .validation-summary-errors {  color: red  }  </style>  <h2>Create</h2>  <form action="/home/create" method="post">  <div asp-validation-summary="ModelOnly">  <span>Please correct the following errors</span>  </div>  <label asp-for="Name">Name</label>  <input asp-for="Name" />  <span asp-validation-for="Name"></span>  <br />  <input type="submit" name="submit" />  </form> |

### 22.3.3. Kiểm tra ModelState.IsValid trong Controller Action

Cuối cùng trong Controller action method chúng ta kiểm tra nếu ModelState.IsValid nếu có bất cứ lỗi nào thì chúng ta trả về cho người dùng. Chúng ta trả về model để hiển thị các giá trị được nhập đồng thời trả kèm cả danh sách lỗi được hiển thị bởi Validation tag helper:

|  |
| --- |
| [HttpPost]  public IActionResult Create(ProductEditModel model)  {  string message = "";  if (ModelState.IsValid)  {  message = "product " + model.Name + " created successfully";  }  else  {  return View(model);  }  return Content(message);  } |

Đoạn code trên sẽ hiển thị như sau:



## 22.4. Model State

Model Binder cập nhật đối tượng ModelState với kết quả của Model Binding và validation. ModelState sẽ lưu chi tiết của các giá trị được cập nhật lên model và cả các thông tin lỗi trong quá trình validation xuất hiện trong mỗi thuộc tính. ModelState là thuộc tính của *ControllerBase* và thuộc kiểu *ModelStateDictionary*.

**Các phương thức của ModelState**

* *AddModelError*: Thêm một message lỗi cụ thể vào danh sách Errors với 1 key.
* *Clear*: Xoá toàn bộ các key và value trong thể hiện của *ModelStateDictionary*.
* *ClearValidationState(string)*: Xoá phần tử trong *ModelStateDictionary* đúng với key truyền vào.
* *GetFieldValidationState(string)*: Trả về *ModelValidationState* với key truyền vào.
* *TryAddModelError(string, Exception, ModelMetadata)*: Thêm một exception cụ thể vào Errors với một key.

**Sử dụng ModelState**

Lấy danh sách Errors: Lặp quá danh sách key và lấy danh sách Errors trong ModelState.

|  |
| --- |
| foreach (var modelStateKey in ModelState.Keys)  {  var modelStateVal = ModelState[modelStateKey];  foreach (var error in modelStateVal.Errors)  {  var key = modelStateKey;  var errorMessage = error.ErrorMessage;  }  } |

Thêm một Custom Error Message: Không phải tất cả các thông báo lỗi đều được đưa ra bởi Model Validator. Ví dụ bạn nhận một Product Model hợp lệ nhưng Product lại tồn tại trong cơ sở dữ liệu. Vậy trường hợp này bạn cần thêm vào một custom error message cho ModelState và gửi nó cho người dùng:

|  |
| --- |
| if (ModelState.IsValid) {  If (someService.IsProductExists(model)) {  ModelState.AddModelError("", “Product already exists”);  return View(model);  } else {  return View(model);  }  } |

## 22.5. Validation Attributes

*Namespace System.ComponentModel.DataAnnotaions* chứa một số các thuộc tính.

**CreditCard**

Kiểm tra thuộc tính có định dạng credit card.

|  |
| --- |
| [CreditCard(ErrorMessage ="Please enter a valid card No")]  public string creditCard { get; set; } |

**Compare**

So sánh giá trị với giá trị của thuộc tính khác.

Ví dụ, áp dụng vào trường hợp cần xác nhận mật khẩu. Tham số vào của attribute này là NewPassword. Nếu xác nhận mật khẩu không đúng với NewPassword thì quá trình kiểm tra không hợp lệ.

|  |
| --- |
| [Required(AllowEmptyStrings =false,ErrorMessage ="Please enter a valid password")]  public string NewPassword { get; set; }  [Compare(otherProperty:"NewPassword", ErrorMessage ="New & confirm password does not match")]  public string ConfirmPassword { get; set; } |

Cách khác là bạn có thể so sánh trong một phương thức get để kiểm tra:

|  |
| --- |
| [Required(AllowEmptyStrings = false, ErrorMessage = "Please enter a valid password")]  public string NewPassword { get; set; }  [Compare(otherProperty: "validatePassword", ErrorMessage = "New & confirm password does not match")]  public string ConfirmPassword { get; set; }  public string validatePassword  {  get  {  //Do Some calculations here  return this.NewPassword;  }  } |

**EmailAddress**

Kiểm tra thuộc tính có phải định dạng email không.

|  |
| --- |
| [EmailAddress(ErrorMessage ="Please enter a valid email")]  public string EmailID {get;set;} |

HTML được tạo ra:

|  |
| --- |
| <label for="EmailID">EmailID</label>  <input type="email" data-val="true" data-val-email="Please enter a valid email" id="EmailID" name="EmailID" value="" />  <span class="field-validation-valid" data-valmsg-for="EmailID" data-valmsg-replace="true"></span>  <br /> |

Như bạn thấy HTML, thuộc tính type của nó sẽ tạo ra là email. Trong trường hợp này thì trình duyệt sẽ hiển thị thông báo lỗi nếu có một email không hợp lệ.

**Phone**

Kiểm tra định dạng số điện thoại.

|  |
| --- |
| [Phone(ErrorMessage = "Please enter a valid Phone No")]  public string PhoneNo { get; set; } |
| <label asp-for="PhoneNo"></label>  <input asp-for="PhoneNo" />  <span asp-validation-for="PhoneNo"></span> |
| <label for="PhoneNo">PhoneNo</label>  <input type="tel" data-val="true" data-val-phone="Please enter a valid Phone No" id="PhoneNo" name="PhoneNo" value="" />  <span class="field-validation-valid" data-valmsg-for="PhoneNo" data-valmsg-replace="true"></span>  <br /> |

**Range**

Cho phép kiểm tra xem giá trị có nằm trong một khoảng cho trước không.

|  |
| --- |
| [Range(minimum:100,maximum:200, ErrorMessage = "Please enter a valid no between 100 & 200")]  public int Range { get; set; } |
| <label asp-for="Range"></label>  <input asp-for="Range" />  <span asp-validation-for="Range"></span> |
| <label for="Range">Range</label>  <input type="number" data-val="true" data-val-range="Please enter a valid no between 100 &amp; 200" data-val-range-max="200" data-val-range-min="100" data-val-required="The Range field is required." id="Range" name="Range" value="0" />  <span class="field-validation-valid" data-valmsg-for="Range" data-valmsg-replace="true"></span>  <br /> |

**RegularExpression**

Kiểm tra dữ liệu có khớp với một biểu thức chính quy không.

|  |
| --- |
| [RegularExpression(pattern: "^Mr\\..\*|^Mrs\\..\*|^Ms\\..\*|^Miss\\..\*", ErrorMessage ="Name must start with Mr./Mrs./Ms./Miss.")]  public string FullName { get; set; } |

**Required**

Yêu cầu một thuộc tính là bắt buộc. Các kiểu không chấp nhận giá trị null (decimal, int, float và DateTime) thường mặc định mà bắt buộc không cần dùng đến attribute này. Ứng dụng sẽ tự động kiểm tra phía server với các kiểu không cho phép null và tự đánh là Required.

**StringLength**

Kiểm tra xem chuỗi có nằm trong vùng độ dài cho phép.

|  |
| --- |
| [StringLength(MaximumLength: 25, MinimumLength = 10, ErrorMessage = "Length must be between 10 to 25")]  public string Name { get; set; } |

**Url**

Kiểm tra xem định dạng URL chuẩn không

|  |
| --- |
| [Url(ErrorMessage ="Please enter a valid URL")]  public string Url { get; set; } |
| <label asp-for="Url"></label>  <input asp-for="Url" />  <span asp-validation-for="Url"></span> |
| <label for="Url">Url</label>  <input type="url" data-val="true" data-val-url="Please enter a valid URL" id="Url" name="Url" value="" />  <span class="field-validation-valid" data-valmsg-for="Url" data-valmsg-replace="true"></span>  <br /> |

## 22.6. Validation Tag Helper

ASP.NET cung cấp các Tag Helper liên quan đến hiển thị validation message cho người dùng. Model Binder sẽ bind và kiểm tra dữ liệu nhận từ HTTP Request. Nó tạo một đối tượng ModelState chứa các danh sách thông báo lỗi được tạo ra bởi Model Binder. Validation Tag Helper sẽ tạo ra các phần tử HTML để hiển thị cho user trên view.

ASP.NET Core cung cấp 2 Tag Helper để hiển thị thông báo lỗi trên client:

* Validation Message Tag Helper
* Validation Summary Tag Helper

### 22.6.1. Validation Message Tag Helper

Validation Tag Helper hướng đến phần tử *<span>* và sử dụng nó để thiển thị một thông báo lỗi cho thuộc tính cụ thể.

|  |
| --- |
| [Required]  Public string name {get;set;} |
| <label asp-for="Name">Name</label>  <input asp-for="Name" />  <span asp-validation-for="Name"></span>  <br /> |

Đoạn HTML sau được tạo ra. Chú ý là phần tử *<span>* rỗng và class field-validation-valid được gắn cho nó.

|  |
| --- |
| <label for="Name">Name</label>  <input type="text" data-val="true" data-val-required="The Name field is required." id="Name" name="Name" value="" />  <span class="field-validation-valid" data-valmsg-for="Name" data-valmsg-replace="true"></span> |

Và khi bạn gửi form với một trường name bị rỗng. Đoạn HTML được tạo ra chú ý là phần tử <span> giờ đã có thông tin lỗi. Và class field-validation-error được gắn cho nó.

|  |
| --- |
| <label for="Name">Name</label>  <input type="text" class="input-validation-error" data-val="true" data-val-required="The Name field is required." id="Name" name="Name" value="" />  <span class="field-validation-error" data-valmsg-for="Name" data-valmsg-replace="true">The Name field is required.</span> |

Class dưới đây hiển thị màu đỏ cho class lỗi:

|  |
| --- |
| <style>  .field-validation-error { color:red }  </style> |

### 22.6.2. Validation Summary Tag Helper

Validation summary tag helper hướng đến phần tử <div> và dùng nó để hiển thị tất cả các thông tin lỗi trên form ở một nơi. Mỗi một thông báo lỗi được hiển thị với một danh sách không được sắp xếp. Bạn cần chỉ ra 3 giá trị cho asp-validation-summary tag helper. All, ModelOnly hoặc None.

#### 22.6.2.1. ASP Validation Summary All

Tuỳ chọn này hiển thị tất cả các thông báo lỗi.

|  |
| --- |
| <div asp-validation-summary="All"></div> |

Nó sẽ tạo ra đoạn HTML dưới đây khi không có lỗi nào được hiển thị.

|  |
| --- |
| <div class="validation-summary-valid" data-valmsg-summary="true">  <ul><li style="display:none"></li></ul>  </div> |

Và tạo ra HTML khi model lỗi:

|  |
| --- |
| <div class="validation-summary-errors" data-valmsg-summary="true">  <ul><li>The Name field is required.</li></ul>  </div> |

Để hiển thị tiêu đề

|  |
| --- |
| <div asp-validation-summary="All">  <span>Please correct the following errors</span>  </div> |

Và thêm style:

|  |
| --- |
| .validation-summary-valid span { display: none; } |

#### 22.6.2.2. ASP Validation Summary Model Only

Tuỳ chọn này hiển thị chỉ các lỗi liên quan đến validation mức Model. Các lỗi ở level thuộc tính sẽ không hiển thị. Sử dụng tuỳ chọn này nếu bạn đang hiển thị các lỗi validation liên quan đến thuộc tính sử dụng asp-validation-for.

|  |
| --- |
| <div asp-validation-summary="ModelOnly"></div> |

Điều lạ nữa là nó không tạo ra bất cứ HTML nào khác biệt, khi ModelState hợp lệ không giống như tuỳ chọn All.

Bạn có thể thêm lỗi mức độ Model trực tiếp trong Model sử dụng.

|  |
| --- |
| ModelState.AddModelError(string.Empty, "Errors occurred in the Model"); |

Đoạn HTML dưới được tạo ra khi lỗi xảy ra:

|  |
| --- |
| <div class="validation-summary-errors">  <ul><li>Errors occurred in the Model</li></ul>  </div> |

#### 22.6.2.3. ASP Validation Summary None

Tag Helper sẽ không hiển thị bất cứ lỗi nào, cái này giống như không có Tag Helper.

Chú ý là validation-summary-valid class được thêm vào khi model hợp lệ (không phải khi tuỳ chọn "All" được chọn) và nó được thay thế bởi validation-summary-errors khi model không hợp lệ.

Thêm các style để thay đổi màu chữ:

|  |
| --- |
| <style>  .validation-summary-errors { color:red }  </style> |

Bạn có thể hiển thị tiêu đề của thông báo lỗi:

|  |
| --- |
| <div asp-validation-summary="ModelOnly">  <span>Please correct the following errors</span>  </div> |

Tiêu đề được hiển thị chỉ khi lỗi được hiển thị.

## 22.7. Unobtrusive client-side validation

ASP.NET Core bao gồm các thư viện unobtrusive client-side validation, nó giúp dễ dàng thêm các client-side validation mà không cần viết code nhiều. Bên trên chúng ta sử dụng server-side validation, chúng ta cũng tìm hiểu cách sử dụng các attribute Annotation và được xử lý bởi Model Validator để kiểm tra Model. Thư viện unobtrusive client-side cũng sử dụng các attribute đó để kiểm tra các thuộc tính trên phía client. Sự khác nhau là nó sử dụng Javascript thay vì C#.

Unobtrusive (ngầm) validation sử dụng *jquery.validate.unobtrusive.js*. Thư viện này được xây dựng dựa trên *jquery.validate.js*, nó sử dụng Jquery. Vì thế chúng ta cần import tất cả nó vào view.

### 22.7.1. Cách sử dụng unobtrusive client-side validation

**Load các thư viện Javascript bắt buộc**

Đầu tiên chúng ta cần thêm JQuery, jquery.validate và jquery.validate.unobtrusive vào view. Các script này được thêm vào file Layout (\_Layout.cshtml) định nghĩa layout của ứng dụng và sử dụng chung cho tất cả các view.

|  |
| --- |
| <script src="https://ajax.aspnetcdn.com/ajax/jquery/jquery-2.2.0.min.js"></script>  <script src="https://ajax.aspnetcdn.com/ajax/jquery.validate/1.16.0/jquery.validate.min.js"></script>  <script src="https://ajax.aspnetcdn.com/ajax/jquery.validation.unobtrusive/3.2.6/jquery.validate.unobtrusive.min.js"></script> |

**Thêm Validation Attribute vào thuộc tính input**

Tiếp theo chúng ta cần thêm Validation Attribute vào thuộc tính model, cái này đã học trong bài trước. Các attribute này chúng ta thêm khi cài đặt server-side validation cũng được dùng luôn cho client-side validation. Trong view nhớ chỉ ra file layout sử dụng bằng cách:

|  |
| --- |
| @{  Layout = "\_Layout";  } |

Model

|  |
| --- |
| [Required(AllowEmptyStrings = false, ErrorMessage = "Please enter the name")]  [StringLength(maximumLength: 25, MinimumLength = 10, ErrorMessage = "Length must be between 10 to 25")]  public string Name { get; set; } |

View

|  |
| --- |
| <form asp-controller="home" asp-action="create" method="post">  <div asp-validation-summary="ModelOnly">  <span>Please correct the following errors</span>  </div>  <label asp-for="Name"></label>  <input asp-for="Name"/>  <span asp-validation-for="Name"></span>  <br />  <input type="submit" name="submit" />  </form> |

Controller

|  |
| --- |
| [HttpPost]  public IActionResult Create(ProductEditModel model, [FromQuery] string test)  {  string message = "";  if (ModelState.IsValid){  message = "product " + model.Name + " created successfully";  }  else {  return View(model);  }  return Content(message);  } |

Chúng ta không sửa bất cứ code server nào, ngoại trừ thêm các thư viện Javascript.

### 22.7.2. Chúng làm việc ra sao?

Các input tag helper tạo ra HTML như sau:

|  |
| --- |
| <label for="Name">Name</label>  <input type="text" data-val="true" data-val-length="Length must be between 10 to 25" data-val-length-max="25" data-val-length-min="10" data-val-required="Please enter the name" id="Name" name="Name" value="" />  <span class="field-validation-valid" data-valmsg-for="Name" data-valmsg-replace="true"></span>  <br /> |

Bạn có thể thấy các thuộc tính được thêm vào với ký tự bắt đầu là data-\*. Các thuộc tính data-\* là một phần của HTML 5 cho phép chúng ta thêm các thông tin mở rộng cho thẻ HTML.

Thư viện Javascript unobtrusive đọc các thuộc tính data-val và thực hiện kiểm tra phía client trên trình duyệt khi người dùng submit form. Các validation này hoàn thành trước khi form được gửi qua HTTP Request. Nếu có validation nào lỗi thì request sẽ không được gửi.

Click F12 và mở cửa sổ Chrome Developer console và xem tab Network. Bạn có thể thấy không có request nào được gửi khi click nút Submit với các dữ liệu không hợp lệ.

# Chương XXIII. Cơ chế Dependency Injection

## 23.1. Giới thiệu về Dependency Injection

Dependency Injection (DI) giờ là một phần của ASP.NET Core. Tất cả các service của framework đều được inject khi chúng ta cần. Nhưng trước khi đi xa hơn, chúng ta cần hiểu tại sao cần dependency injection. Hãy xem ví dụ sau đây trong controller action method khi chúng ta muốn lấy danh sách sản phẩm ra từ *ProductService*.

|  |
| --- |
| **public class** HomeController : **Controller**  {  **public IActionResult** Index()  {  \_productService = **new ProductService**();  **return View**(\_productService.getAll());  }  } |

Action method *Index* có một phụ thuộc đến *ProductService*. Vì thế nó tạo ra một thể hiện và gọi phương thức phương thức *GetAll* để lấy danh sách sản phẩm. *ProductService* giờ đã bị gắn chặt (tightly couple) vào phương thức Index của *HomeController*.

Nếu chúng ta tạo *BetterProductService* và muốn thay thế *ProductService* cũ thì sao:

|  |
| --- |
| **public IActionResult** Index()  {  \_productService = **new BetterProductService**();  **return View**(\_productService.getAll());  } |

Chúng ta cần làm điều đó với tất cả các Controller, service có sử dụng *ProductService*. Nếu chúng ta muốn sử dụng *TestProductService* và chỉ muốn dùng nó cho mục đích test còn với môi trường Production thì sẽ là *ProductService*. Vậy là không dễ dàng.

Hãy xem xét trường hợp khi *ProductService* phụ thuộc vào service khác, sau đó service đó lại phụ thuộc service khác nữa. Nó không thường xuyên nhưng việc một chuỗi các phụ thuộc như thế chắc chắn có tồn tại trong thực tế.

Depedency Injection sẽ giải quyết tất cả các vấn đề này.

## 23.2. Depedency Injection là gì?

Dependency Injection (DI) là một design pattern khi một đối tượng không được tạo trong các thành phần phụ thuộc vào nó mà yêu cầu nó. Hãy thay đổi *HomeController* một chút:

|  |
| --- |
| **public class** HomeController : **Controller**  {  **private** **IProductService** \_productService;  **public** HomeController(**IProductService** productService)  {  \_productService = productService;  }  **public IActionResult** Index()  {  \_productService = new ProductService();  **return View**(\_productService.All());  }  } |

Sự khác nhau giữa đoạn code trên và đoạn code này là chúng ta không tạo ra thể hiện của *ProductService* trong Index action method. Chúng ta yêu cầu nó trong constructor của *HomeController*. Vấn đề đã được giải quyết chưa? Ai đó đã tạo thể hiện của *ProductService* và gán nó vào *HomeController*?

Đây là điểm mà ASP.NET Core Dependency Injection framework làm nhiệm vụ của nó. Trách nhiệm của nó là tạo ra thể hiện của *ProductService* và đối tượng này được gọi là DI Container hay Ioc Container.

Dependency Injection là một design pattern. Dependency injection framework triển khai design pattern này. Có nhiều framework như Autofac, Unity...bạn có thể sử dụng trong ASP.NET Core.

## 23.3. DI Container

DI Container là một đối tượng có trách nhiệm tạo các phụ thuộc (ProductService) và gán nó cho đối tượng yêu cầu (HomeController) nó.

**Làm thế nào DI Container biết đối tượng nào được tạo?**

Chúng ta cần cấu hình cho DI Container là class nào bạn muốn tạo. Chúng ta cần đặt trong class *Startup* với phương thức *ConfigureServices*.

|  |
| --- |
| public void ConfigureServices(IServiceCollection services)  {  services.AddMvc();  services.AddTransient<IProductService, ProductService>();  } |

Dòng thứ 2 sẽ đăng ký *ProductService* với service collection sử dụng phương thức *AddTransient*. Có 2 phương thức khác là *AddSingleton* và *AddScoped*.

## 23.4. Ví dụ về hệ thống Dependency Injection [Phụ lục 6]

Tạo một project ASP.NET Core sử dụng empty template và đặt tên nó là *DependencyInjection*. Thêm *HomeController* với *Index* method. Bạn có thể tham khảo các bài viết trước.

**Tạo ViewModel**

Tạo một folder Models và tạo ViewModel như sau:

|  |
| --- |
| public class ProductViewModel  {  public int Id { get; set; }  public string Name { get; internal set; }  } |

**Thêm mới service**

Tạo một thư mục Services và thêm mới một class với tên ProductService.cs

|  |
| --- |
| using DependencyInjection.Models;  using System.Collections.Generic;  namespace DependencyInjection.Service  {  public interface IProductService  {  List<ProductViewModel> getAll();  }  public class ProductService : IProductService  {  public List<ProductViewModel> getAll()  {  return new List<ProductViewModel>  {  new ProductViewModel {Id = 1, Name = "Pen Drive" },  new ProductViewModel {Id = 2, Name = "Memory Card" },  new ProductViewModel {Id = 3, Name = "Mobile Phone" },  new ProductViewModel {Id = 4, Name = "Tablet" },  new ProductViewModel {Id = 5, Name = "Desktop PC" } ,  };  }  }  } |

Đầu tiên chúng ta thêm interface *IProductService* và thêm *ProductService* để triển khai interface này. *ProductService* sẽ trả về một danh sách sản phẩm. Danh sách sản phẩm được fix cứng trong code. Trong thực tế thì nó sẽ lấy từ database.

**Sử dụng Service trong Controller**

Giờ hãy mở *HomeController*

|  |
| --- |
| using DependencyInjection.Service;  using Microsoft.AspNetCore.Mvc;  namespace DependencyInjection.Controllers  {  public class HomeController : Controller  {  private IProductService \_productService;  public HomeController(IProductService productService)  {  \_productService = productService;  }  public IActionResult Index()  {  return View(\_productService.getAll());  }  }  } |

Constructor của *HomeController* yêu cầu một thể hiện của *ProductService* và lưu trữ nó trong biến local tên là *\_productService*. Phương thức *Index* gọi view với danh sách sản phẩm được lấy ra từ phương thức *GetAll* của *ProductService*.

**View**

View chỉ hiển thị danh sách sản phẩm

|  |
| --- |
| @model List<DependencyInjection.Models.ProductViewModel>;  @{  ViewData["Title"] = "Index";  }  <h2>Index</h2>  @foreach (var product in Model)  {  <p>@product.Id @product.Name</p>  } |

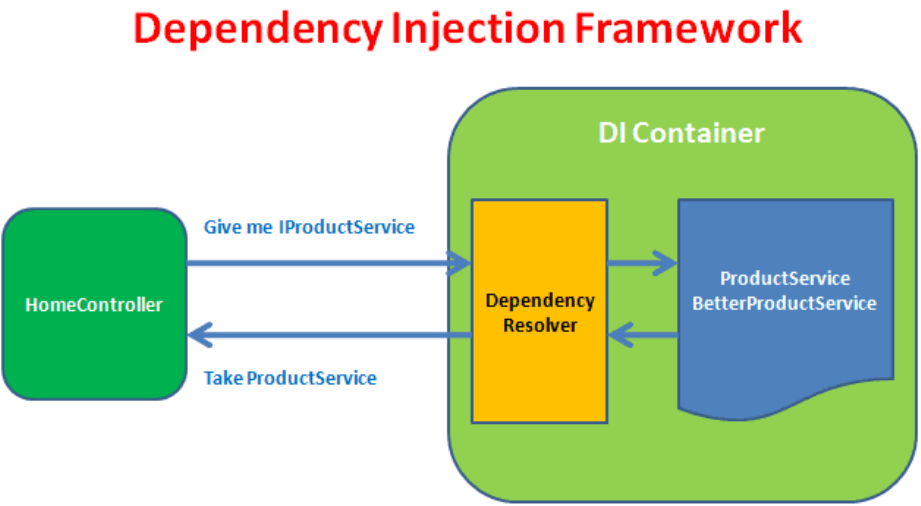
**Đăng ký service**

Bước cuối cùng là đăng ký service với Dependency Injection container. Mở *Startup.cs* và đến phương thức *ConfigureServices*. Nơi mà tất cả các service được cấu hình cho DI.

|  |
| --- |
| public void ConfigureServices(IServiceCollection services)  {  services.AddMvc();  services.AddTransient<IProductService, ProductService>();  } |

Giờ chúng ta sẽ đăng ký *ProductService* sử dụng phương thức *AddTransient*.

**Các thành phần phụ thuộc được inject như thế nào?**

****

Ảnh 25: Cách ProductService được inject vào HomeController

Khi một HomeController mới được yêu cầu, MVC sẽ yêu cầu DI framework cung cấp một thể hiện của HomeControlelr class. DI Container sẽ xem constructor của HomeController và xác định xem nó có những thành phần phụ thuộc nào (depedencies). Nó sẽ tìm các thành phần phụ thuộc trong danh sách được đăng ký của service collection và tìm service nào thoả mãn sau đó tạo thể hiện cho nó. Sau khi tạo HomeController và gán thể hiện của dependencies đó cho constructor.

**Tạo BetterProductService**

Giờ chúng ta sẽ tạo mới service khác là BetterProductService và muốn sử dụng nó thay vì ProductService

|  |
| --- |
| public class BetterProductService : IProductService  {  public List<ProductViewModel> getAll()  {  return new List<ProductViewModel>  {  new ProductViewModel {Id = 1, Name = "Television" },  new ProductViewModel {Id = 2, Name = "Refrigerator" },  new ProductViewModel {Id = 3, Name = "IPhone" },  new ProductViewModel {Id = 4, Name = "Laptop" },  };  }  } |

Vào *ConfigureServices* trong *Startup* thay đổi *ProductService* thành *BetterProductService*.

|  |
| --- |
| public void ConfigureServices(IServiceCollection services)  {  services.AddMvc();  services.AddTransient<IProductService, BetterProductService>();  } |

Bạn không phải vào mỗi controller hay service để thay đổi. Bạn có thể quản lý nó ở một nơi duy nhất. Tương tự bạn có thể làm với *TestProductService* sử dụng chúng để test.

**Thay đổi Service dựa trên environment**

Ví dụ, bạn có thể sử dụng môi trường và chuyển đổi giữa các service Trong constructor của Startup class yêu cầu IHostingEnvironment service:

|  |
| --- |
| IHostingEnvironment \_env;  public Startup(IHostingEnvironment env)  {  \_env = env;  } |

Tiếp theo trong ConfigureServices chúng ta cấu hình code với BetterProductService trong Production và ProductService cho các môi trường khác.

|  |
| --- |
| public void ConfigureServices(IServiceCollection services)  {  services.AddMvc();  if (\_env.IsProduction())  {  services.AddTransient<IProductService, BetterProductService>();  }  else  {  services.AddTransient<IProductService, ProductService>();  }  } |

Thay đổi môi trường chọn Project, chuột phải và chọn Properties. Chọn tab Debug. Thay đổi ASPNETCORE\_ENVIRONMENT thành môi trường gì mà bạn muốn.

## 23.5. Constructor Injection và Action Injection

Bạn có thể inject service vào Controller theo 2 cách:

* Constructor Injection
* Action Injection

### 23.5.1. Constructor Injection

Khi các thành phần được inject vào thông qua constructor sau đó cách này được gọi là constructor injection.

|  |
| --- |
| public class HomeController : Controller  {  private IProductService \_productService;  public HomeController(IProductService productService)  {  \_productService = productService;  }  public IActionResult Index()  {  return View(\_productService.All());  }  } |

### 23.5.2. Action Injection

Nếu các dependencies được inject qua một action method thì cách này gọi là Action Injection. Action Injection được thực hiện sử dụng [FromServices] attribute vào một tham số của action method. Sử dụng method này nếu service chỉ được sử dụng trong một action method thôi.

|  |
| --- |
| public class HomeController : Controller  {  public HomeController()  {  }  public IActionResult Index( [FromServices] IProductService productService)  {  return View(\_productService.All());  }  } |

## 23.6. Lợi ích của Dependency Injection

* Dependency injection giúp thực hiện kiến trúc lỏng lẻo (loose coupling) trong phần mềm.
* Code sẽ sạch và dễ đọc hơn
* Tăng khả năng có thể kiểm thử và bảo trì
* Cho phép bạn thay đổi triển khai mà không phải thay đổi quá nhiều code.

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

Hiểu về vòng đời của các service được tạo sử dụng Dependency Injection là rất quan trọng trước khi sử dụng chúng. Tạo các service mà không hiểu về sự khác nhau giữa Transient, Singleton và Scoped có thể làm hệ thống hoạt động không như mong muốn. Khi một service yêu cầu service khác thông qua DI, chúng ta biết rằng nó nhận một thể hiện mới hay một thể hiện đã tồn tại rất quan trọng. Vì thế việc nhận thức đúng vòng đời hay thời gian sống (lifetime) trong khi đăng ký service là việc rất quan trọng.

### 23.7.1. Quản lý vòng đời (lifetime) Service

Bất cứ khi nào chúng ta yêu cầu service, DI Container sẽ quyết định xem có tạo mới một thể hiện (instance) hay sử dụng lại thể hiện đã tạo từ trước đó. Vòng đời của Service phụ thuộc vào khi khởi tạo thể hiện và nó tồn tại bao lâu. Chúng ta định nghĩa vòng đời khi đăng ký service.

Có 3 mức độ vòng đời, bằng cách này chúng ta có thể quyết định mỗi service có vòng đời ra sao.

* Transient: Một thể hiện mới luôn được tạo, mỗi khi được yêu cầu.
* Scoped: Tạo một thể hiện mới cho tất cả các scope (Mỗi request là một scope). Trong scope thì service được dùng lại
* Singleton: Service được tạo chỉ một lần duy nhất.

### 23.7.2. Project ví dụ [Phụ lục 7]

Tạo một ứng dụng ASP.NET Core sử dụng Empty Template và gọi nó là DependencyInjection. Thêm HomeController với Index method. Bạn có thể theo bài viết trước.

**Tạo mới Service Interface**

Tạo mới service tên là SomeService trong thư mục Services. Thêm mới 3 interface. Mỗi interface tương ứng với một vòng đời trong ASP.NET Core. Interface thật đơn giản chúng chứa mỗi phương thức GetID trả về một Guid.

|  |
| --- |
| public interface ITransientService  {  Guid GetID();  }  public interface IScopedService  {  Guid GetID();  }  public interface ISingletonService  {  Guid GetID();  } |

**Tạo một Service**

Giờ chúng ta sẽ tạo một service đơn lẻ, triển khai cả 3 interface kia:

|  |
| --- |
| public class SomeService : ITransientService, IScopedService, ISingletonService  {  Guid id;  public SomeService()  {  id = Guid.NewGuid();  }  public Guid GetID()  {  return id;  }  } |

Service tạo ra một ID duy nhất và nó sẽ được khởi tạo và trả về id trong phương thức GetID. Giờ hãy xem chi tiết.

**Transient:**

Transient Service luôn tạo mới mỗi lần service được yêu cầu

**Đăng ký Transient Service**

Giờ bên trong phương thức ConfigureServices của Startup sẽ đăng ký SomeService thông qua ITransientService interface.

|  |
| --- |
| services.AddTransient<ITransientService, SomeService>(); |

**Inject vào Controller**

Mở HomeController ra và inject 2 thể hiện của SomeService như dưới đây:

|  |
| --- |
| public class HomeController : Controller  {  ITransientService \_transientService1;  ITransientService \_transientService2;  public HomeController(ITransientService transientService1,  ITransientService transientService2)  {  \_transientService1 = transientService1;  \_transientService2 = transientService2;  }  public IActionResult Index()  {  ViewBag.message1 ="First Instance " + \_transientService1.GetID().ToString();  ViewBag.message2 ="Second Instance "+ \_transientService2.GetID().ToString();  return View();  }  } |

Đầu tiên, chúng ta inject 2 thể hiện của service thông qua interface ITransientService trong constructor của HomeController.

|  |
| --- |
| public HomeController(ITransientService transientService1,ITransientService transientService2) |

Tiếp theo, chúng ta gọi phương thức GetID trong mỗi thể hiện và gán nó vào view sử dụng ViewBag.

|  |
| --- |
| ViewBag.message1 ="First Instance " + \_transientService1.GetID().ToString();  ViewBag.message2 ="Second Instance "+ \_transientService2.GetID().ToString(); |

**View**

Mở view và thêm code:

|  |
| --- |
| <h3>Transient Service</h3>  @ViewBag.message1  </br>  @ViewBag.message2 |

Chạy ứng dụng và bạn sẽ nhìn thấy 2 Guid khác nhau hiển thị trên màn hình. Bằng chứng là chúng ta đã nhận 2 thể hiện của Transient service.



**Scoped**

Các service với vòng đời Scoped được tạo chỉ một lần cho mỗi request (scope). Một thể hiện mới được tạo cho mỗi request và thể hiện được dùng lại trong request.

**Đăng ký Scoped Service**

Trong phương thức ConfigureServices đăng ký SomeService sử dụng phương thức AddScoped sử dụng IScopedService interface.

|  |
| --- |
| services.AddScoped<IScopedService, SomeService>(); |

**Inject scoped service vào Controller**

Tiếp theo inject service này vào Controller. Chúng ta đã có transient service được inject vào controller. Giờ không thay đổi gì nhưng thêm mới:

|  |
| --- |
| public class HomeController : Controller  {  ITransientService \_transientService1;  ITransientService \_transientService2;  IScopedService \_scopedService1;  IScopedService \_scopedService2;  public HomeController(ITransientService transientService1,  ITransientService transientService2,  IScopedService scopedService1,  IScopedService scopedService2)  {  \_transientService1 = transientService1;  \_transientService2 = transientService2;  \_scopedService1 = scopedService1;  \_scopedService2 = scopedService2;  }  public IActionResult Index()  {  ViewBag.message1 ="First Instance " + \_transientService1.GetID().ToString();  ViewBag.message2 ="Second Instance "+ \_transientService2.GetID().ToString();  ViewBag.message3 = "First Instance " + \_scopedService1.GetID().ToString();  ViewBag.message4 = "Second Instance " + \_scopedService2.GetID().ToString();  return View();  }  } |

Thêm vào action method 2 biến message3 và message4 cho scoped service.

**View**

Trong view thêm 3 dòng

|  |
| --- |
| <h3>Scoped Service</h3>  @ViewBag.message3  </br>  @ViewBag.message4 |

Chạy ứng dụng.

Thể hiện được tạo chỉ một lần cho mỗi request, đó là lý do chúng ta tạo ra 2 ID giống nhau. Giờ hãy refresh trình duyệt. ID đã thay đổi bởi vì một thể hiện mới được tạo ra cho mỗi request.

A screenshot of a chat

Description automatically generated

**Singleton**

Một thể hiện duy nhất của service được tạo khi nó được yêu cầu lần đầu. Sau đó mỗi một request sau này sẽ chỉ sử dụng cùng instance đó thôi. Request mới không tạo thể hiện mới mà nó được dùng lại.

**Đăng ký Singleton Service**

Singleton service được đăng ký sử dụng phương thức AddSingleton

|  |
| --- |
| services.AddSingleton<ISomeServiceService, SomeService>(); |

**Inject Singleton service vào Controller**

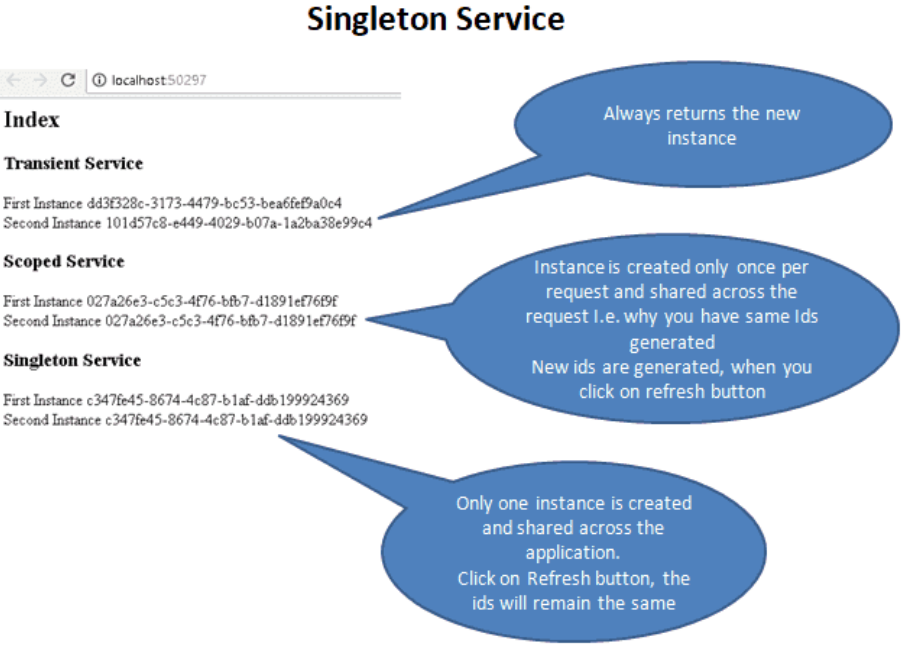
|  |
| --- |
| public class HomeController : Controller  {  ITransientService \_transientService1;  ITransientService \_transientService2;  IScopedService \_scopedService1;  IScopedService \_scopedService2;  ISingletonService \_singletonService1;  ISingletonService \_singletonService2;  public HomeController(ITransientService transientService1,  ITransientService transientService2,  IScopedService scopedService1,  IScopedService scopedService2,  ISingletonService singletonService1,  ISingletonService singletonService2)  {  \_transientService1 = transientService1;  \_transientService2 = transientService2;  \_scopedService1 = scopedService1;  \_scopedService2 = scopedService2;  \_singletonService1 = singletonService1;  \_singletonService2 = singletonService2;  }  public IActionResult Index()  {  ViewBag.message1 ="First Instance " + \_transientService1.GetID().ToString();  ViewBag.message2 ="Second Instance "+ \_transientService2.GetID().ToString();  ViewBag.message3 = "First Instance " + \_scopedService1.GetID().ToString();  ViewBag.message4 = "Second Instance " + \_scopedService2.GetID().ToString();  ViewBag.message5 = "First Instance " + \_singletonService1.GetID().ToString();  ViewBag.message6 = "Second Instance " + \_singletonService2.GetID().ToString();  return View();  }  } |

Đầu tiên chúng ta inject 6 thể hiện của SomeService. Hai thể hiện cho mỗi interface.

**View**

|  |
| --- |
| @{  ViewData["Title"] = "Index";  }  <h2>Index</h2>  <h3>Transient Service</h3>  @ViewBag.message1  </br>  @ViewBag.message2  <h3>Scoped Service</h3>  @ViewBag.message3  </br>  @ViewBag.message4  <h3>Singleton Service</h3>  @ViewBag.message5  </br>  @ViewBag.message6 |

Chạy ứng dụng và bạn thấy rằng các ID tạo ra từ Singleton service luôn giống nhau và sẽ không thay đổi ngay cả khi bạn refresh ứng dụng. Bạn có thể thấy trong hình dưới đây



**Vậy nên sử dụng cái nào?**

Transient service là cách an toàn nhất để tạo, vì bạn luôn tạo mới một thể hiện. Nhưng vì thế mà nó sẽ tạo mỗi lần bạn yêu cầu như vậy sẽ dùng nhiều bộ nhớ và tài nguyên hơn. Điều này có thể ảnh hưởng không tốt đến hiệu năng nếu quá nhiều thể hiện được tạo.

Sử dụng Transient Service phù hợp khi bạn muốn dùng cho các service nhẹ và nhỏ cũng như không có trạng thái.

Scoped service thì tốt hơn khi bạn muốn duy trì trạng thái trong một request.

Singleton được tạo chỉ một lần, nó không bị huỷ cho đến khi ứng dụng tắt. Bất cứ việc chiếm bộ nhớ nào với các service này đều tích lại theo thời gian và nó đầy lên. Nhưng cũng giúp chúng ta tiết kiệm bộ nhớ nếu xử lý tốt vì chúng được tạo chỉ 1 lần và sử dụng mọi nơi.

Sử dụng Singleton khi bạn cần duy trì trạng thái trong toàn hệ thống. Cấu hình, tham số của ứng dụng, các service logging, caching dữ liệu...là các ví dụ thường dùng Singleton.

**Inject service với các vòng đời khác nhau vào service khác**

Hãy cẩn thân, khi inject service vào service khác với các vòng đời khác nhau. Hãy xem ví dụ Singleton Service phụ thuộc vào một service khác được đăng ký với vòng đời là Transient.

Khi request đến lần đầu thì một instance của Singleton được tạo.

Khi request thứ 2 đến thì thể hiện này của Singleton được dùng lại. Và singleton này đã chứa một thể hiện của transient service. Vì thế nó không được tạo lại. Điều này vô hình chung đã chuyển transient service thành singleton service.

Các service với mức độ vòng đời thấp hơn được inject vào service có vòng đời cao hơn sẽ thay đổi service vòng đời thấp hơn thành vòng đời cao hơn. Điều này sẽ làm việc debug khó hơn và nó nên tránh. Từ thấp đến cao là Transient, Scoped và Singleton.

Vì thế hãy nhớ quy tắc:

* Không bao giờ inject Scoped & Transient service vào Singleton service
* Không bao giờ inject Transient Service vào Scope Service

# 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

## Phụ lục 1: First web app

Source code: <https://github.com/NT02IT/ASP.NET-CORE/tree/main/Sources/FirstWebApp>

## Phụ lục 2: Middleware

Source code: <https://github.com/NT02IT/ASP.NET-CORE/tree/main/Sources/Middleware>

## Phụ lục 3: