Bài giảng tích hợp:

ASP NET CORE

Faculty of IT

Email: smdat@hueic.edu.vn

Chương 3. ASP.NET Core MVC

3.1. Mẫu kiến trúc MVC (Model – View – Controller) trong ASP.NET Core

3.2. Dự án ASP.NET Core MVC

3.3. Controller trong ASP.NET Core MVC

3.4. Action và ActionResult trong ASP.NET Core MVC

3.5. View trong ASP.NET Core MVC, layout, viewstart, viewimports

3.6. Sử dụng LibMan – công cụ hỗ trợ tải thư viện

3.7. Routing trong ASP.NET Core MVC

3.8. Model binding trong ASP.NET Core MVC

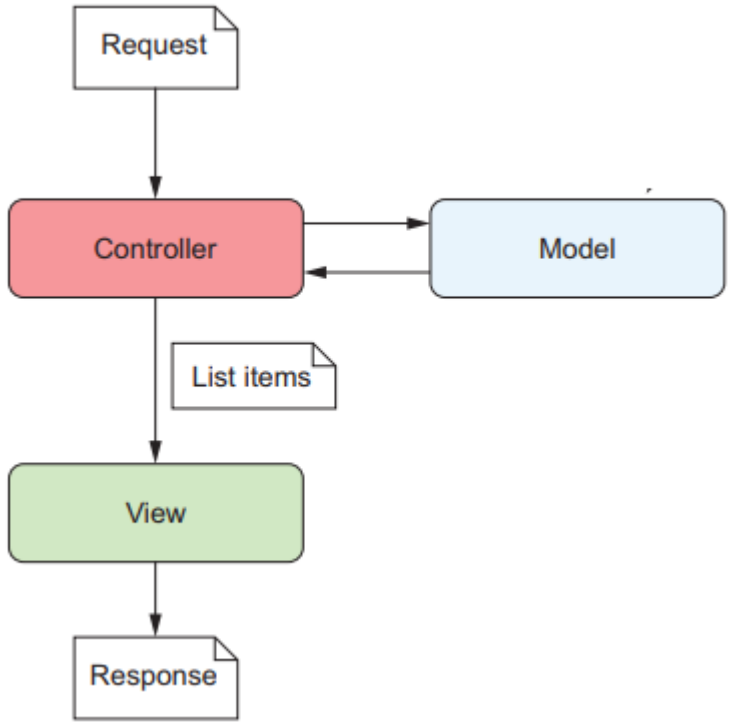
3.9. Thực hành

- **ASP.NET Core MVC** là tên gọi của framework trong ASP.NET Core thực thi mô hình kiến trúc **MVC**.
- **Framework** này giúp phát triển nhiều loại ứng dụng khác nhau, từ ứng dụng web truyền thống đến ứng dụng đơn trang hoặc **Web API**.
- **Mẫu kiến trúc MVC**
  - **MVC** là tên gọi tắt của **Model** – **View** – **Controller** – một *mẫu kiến trúc* (architectural pattern) lâu đời và rất phổ biến trong phát triển phần mềm.
  - Mẫu kiến trúc MVC được áp dụng rộng rãi trong ứng dụng web, desktop và mobile.
  - Trên thực tế, mẫu MVC nguyên bản vốn được xây dựng dành cho ứng dụng với giao diện đồ họa tương tự ứng dụng desktop chứ không phải cho ứng dụng web. Tuy nhiên, hiện nay mẫu kiến trúc này được sử dụng rộng rãi nhất trong các web framework.

Mẫu kiến trúc MVC

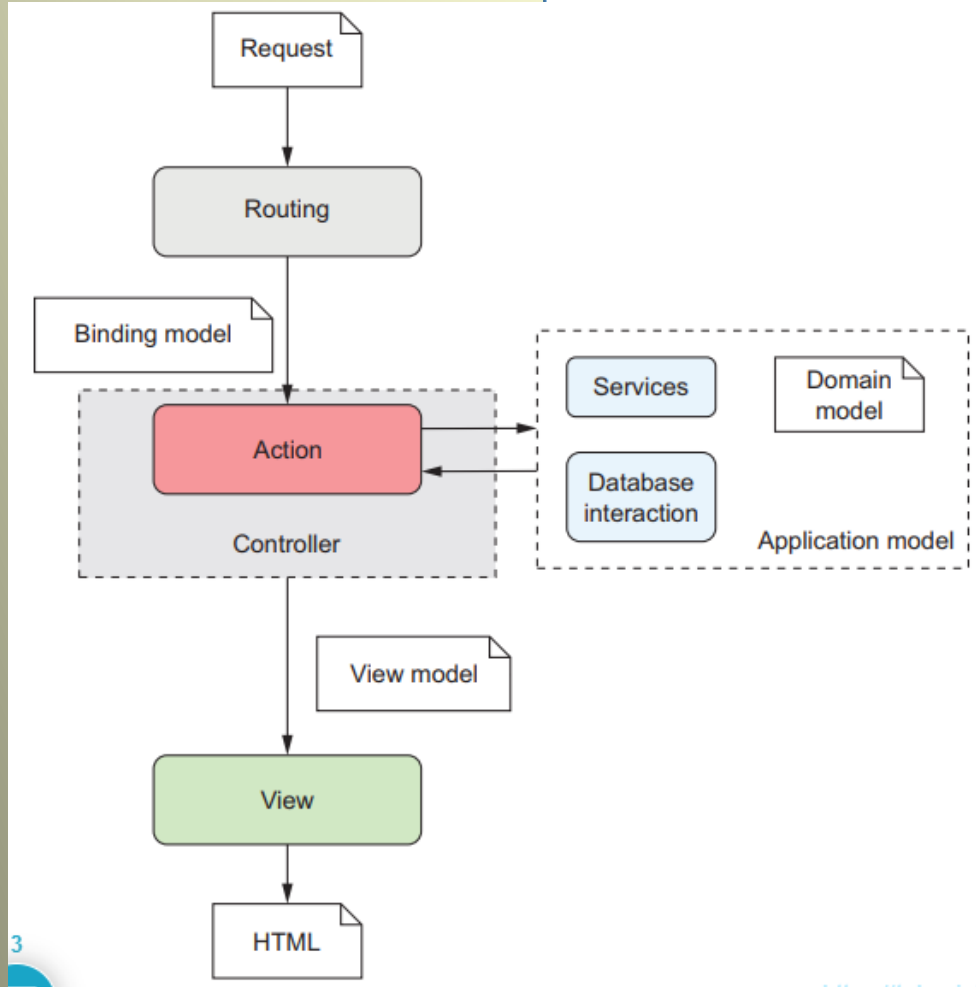
- **ASP.NET Core MVC** chỉ là một trong số các web framework vận dụng mẫu kiến trúc **MVC**.
- Mỗi ứng dụng hoặc framework diễn giải **MVC** theo cách riêng của mình và thường tập trung hơn vào một số khía cạnh của mẫu kiến trúc này. Ví dụ, mẫu **MVC** ứng dụng trên **Rails** hay **Spring** không hoàn toàn giống với **ASP.NET Core**, mặc dù đều có 3 thành phần cơ bản tương tự nhau.
- Tuy nhiên, dù vận dụng ở đâu và theo cách nào, mẫu này cũng hướng tới phân tách việc biểu diễn dữ liệu khỏi quản lý và xử lý dữ liệu. Để tạo ra sự phân tách này, mẫu **MVC** phân chia ứng dụng ra 3 thành phần với nhiệm vụ cơ bản như sau:
  - **Model** – dữ liệu và trạng thái của ứng dụng.
  - **View** – khuôn mẫu dành cho hiển thị dữ liệu.
  - **Controller** – cập nhật model và lựa chọn view.
- Mỗi thành phần của ứng dụng MVC chỉ chịu trách nhiệm cho một mảng duy nhất. Khi các thành phần kết hợp lại và tương tác với nhau sẽ tạo ra giao diện người dùng.

- **Tương tác giữa các thành phần trong kiến trúc MVC**
  - Thứ tự các sự kiện xảy ra khi ứng dụng phản ứng với tương tác/yêu cầu của người dùng như sau:
    - Controller nhận yêu cầu.
    - Controller lấy dữ liệu từ model, hoặc cập nhật dữ liệu của model
    - Controller lựa chọn view phù hợp cho việc hiển thị dữ liệu và chuyển dữ liệu của model sang cho view
    - View sử dụng dữ liệu của model để sinh ra giao diện.



- **Tương tác giữa các thành phần trong kiến trúc MVC**
  - Với cách mô tả **MVC** như trên, **controller** đảm nhiệm vai trò điểm tiếp nhận của tương tác. Người dùng trao đổi với **controller** để bắt đầu một tương tác.
  - Trong ứng dụng web, tương tác này chính là truy vấn **HTTP**. Như vậy khi một truy vấn **HTTP** tới ứng dụng, **controller** sẽ xử lý truy vấn. Tùy thuộc vào truy vấn, **controller** có thể thực hiện nhiều hoạt động khác nhau. Tuy nhiên các hoạt động này hầu như đều thực hiện trong sự kết hợp với **model**.
  - **Model** hoàn toàn độc lập với cách thức hiển thị của dữ liệu trên giao diện do **view** đảm nhiệm.
  - Sự độc lập giữa **model** và **view** giúp đơn giản hóa việc **test**. Thông thường rất khó để test giao diện. Khi không liên quan đến giao diện, thành phần **model** còn lại sẽ dễ dàng hơn để test.
  - Sự độc lập này cũng cho phép **controller** lựa chọn **view** cho phù hợp với yêu cầu. Tức là, với cùng một dữ liệu (**model**), **controller** có thể lựa chọn những cách hiển thị (**view**) khác nhau. Ví dụ, trong một ứng dụng web thông thường, **controller** sẽ lựa chọn **HTML view**. Nếu truy vấn đến từ một ứng dụng khác, **controller** có thể lựa chọn loại **view** đặc biệt ở dạng **JSON** hoặc **XML**.

3.1. Mẫu kiến trúc MVC (Model – View – Controller) trong ASP.NET Core

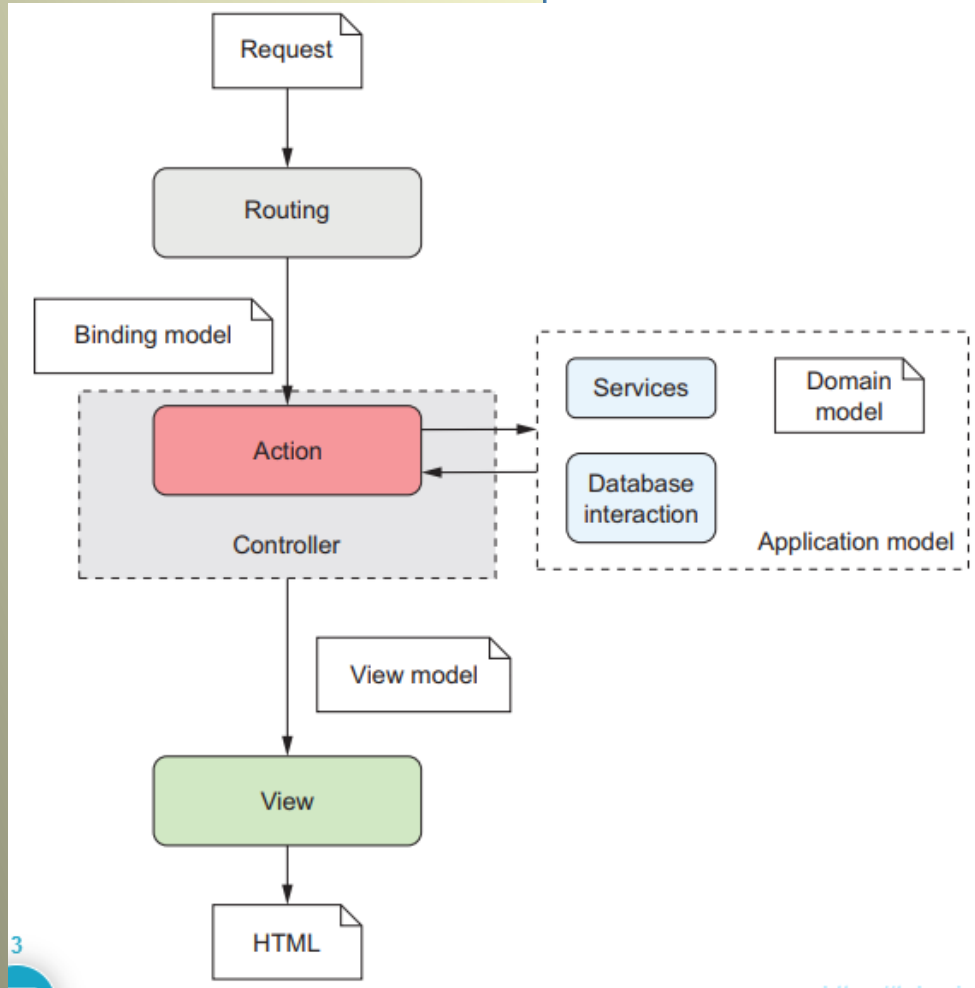


• MVC trong ASP.NET Core

- Đây là một chu kỳ xử lý từ lúc nhận được truy vấn **HTTP** cho đến khi sinh ra **HTML**.
- Truy vấn **HTTP** sẽ được cơ chế **routing** ánh xạ sang một phương thức xác định gọi là **action**.
- Trong **ASP.NET Core MVC**, **action** là phương thức được thực thi để đáp ứng lại một truy vấn.
- Để thực thi, action cần đến dữ liệu đầu vào chứa trong truy vấn **HTTP**. Dữ liệu được trích ra từ truy vấn thông qua cơ chế **model binding**
- **Binding model** là một object đóng vai trò "thùng chứa" dữ liệu trích xuất ra từ truy vấn để cung cấp cho **action**. **Binding model** là kết quả hoạt động của cơ chế **model binding** và là tham số đầu vào cho **action**.
- **Controller** trong **ASP.NET Core** là class chứa các **action** có quan hệ nhất định.
- **Action** khi thực thi sẽ tương tác với các thành phần còn lại của ứng dụng như các dịch vụ, cơ sở dữ liệu.
- Với cách tiếp cận **DDD** (**Domain-driven Design**), phần dữ liệu nghiệp vụ được thể hiện qua các domain model
- Tất cả các thành phần dịch vụ, domain model, v.v., được gọi chung là **application model**

3.1. Mẫu kiến trúc MVC (Model – View – Controller) trong ASP.NET Core

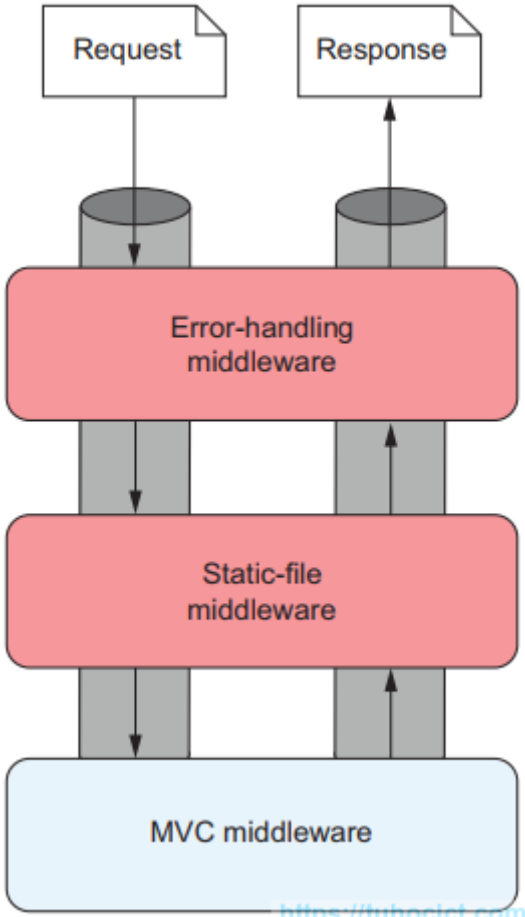
• MVC trong ASP.NET Core



- Tất cả các thành phần dịch vụ, domain model, v.v., được gọi chung là **application model**
- Quá trình tương tác này sẽ sinh ra dữ liệu phục vụ cho hiển thị, gọi là **view model**.
- **View model** là object đơn giản chứa dữ liệu cần thiết để sinh ra giao diện. Thông thường **view model** là một biến thể của dữ liệu lấy được từ **application model** cùng với dữ liệu phụ trợ cho hiển thị (như tiêu đề, phân trang, v.v.).
- **View** trong **ASP.NET Core MVC** là các trang **Razor** chứa loại mã hỗn hợp C# + HTML theo cú pháp Razor. Vì vậy người ta cũng thường gọi **view** trong **ASP.NET Core MVC** là **Razor view**. Kết quả xử lý của **Razor view** là **HTML**.

• MVC middleware trong ASP.NET Core

- Mô hình kiến trúc **MVC** trong **ASP.NET Core** được thực thi trong một **middleware** có tên là **MVC middleware**



- **MVC Middleware** đặt ở cuối chuỗi xử lý của **ASP.NET Core**. Toàn bộ code viết chính là để mở rộng **MVC Middleware** theo nhu cầu xử lý bài toán, như xây dựng các lớp **controller** riêng hoặc các **Razor view**.
- Tất cả các thành phần cần thiết cho một ứng dụng **MVC** như **routing**, **base controller class**, **model binding**, **Razor view engine**, v.v., đều được thực thi sẵn. Bạn có thể dễ dàng sử dụng, mở rộng, hoặc thậm chí thay thế các thành phần này.
- Để tiện lợi cho người lập trình, **MVC Middleware** thực hiện ***cấu hình dựa trên quy ước*** (configuaration over convention), như tên gọi của **controller**, vị trí lưu file của các **Razor view**, v.v..

- **Các loại model trong ASP.NET Core MVC**
  - Trong mô tả của kiến trúc **MVC** khái niệm **model** tương đối chung chung: thành phần chứa tất cả các loại dữ liệu và hoạt động không liên quan đến giao diện.
  - Cách mô tả **model** như vậy gây ra khó hiểu. Ít nhất chúng ta thực sự khó hình dung ra đâu là **model** trong một ứng dụng xây dựng theo kiến trúc **MVC**.
  - Trong **ASP.NET Core**, "**model**" được sử dụng để chỉ MỘT SỐ thành phần khác nhau, bao gồm: **binding model**, **application model**, **domain model**, **view model**.
  - Các thành phần này có vị trí và vai trò rất khác biệt nhau trong ứng dụng **ASP.NET Core MVC**.

- **Các loại model trong ASP.NET Core MVC**
  - **Binding model** là kết quả của việc trích xuất dữ liệu từ truy vấn **HTTP** (chứa trong **route data**, **query string**, **form**) và biến đổi thành **object** của **.NET**. **Binding model** có vai trò là tham số đầu vào cho **action**. Quá trình trích xuất và biến đổi dữ liệu này trong **ASP.NET Core** được gọi là **model binding**
  - **Domain model** là kết quả của phân tích dữ liệu theo mô hình **DDD**. Dù xây dựng ứng dụng theo kiến trúc nào đi chăng nữa thì cũng vẫn cần có **domain model**. **Domain model** không phải là đặc trưng của kiến trúc **MVC**.
  - **View model** là dữ liệu được **action** tạo ra và truyền sang cho **view**. **View model** có lẽ là thành phần gần gũi nhất với định nghĩa **M-model** trong **MVC**.
  - **View model** không "thuần nhất". Nghĩa là nó có thể là các loại model khác hoặc là tập hợp của nhiều loại dữ liệu khác nhau.
  - **Ví dụ**, **view model** có thể là: một danh sách các **object** của **domain model**; một **object** cụ thể của **domain model**; một bộ phận của một object domain model; kết hợp giữa domain model và thông tin phụ trợ như danh sách phân trang.
  - **View model** thường tạo ra từ các hoạt động liên quan đến cơ sở dữ liệu (thuộc về **application model**)

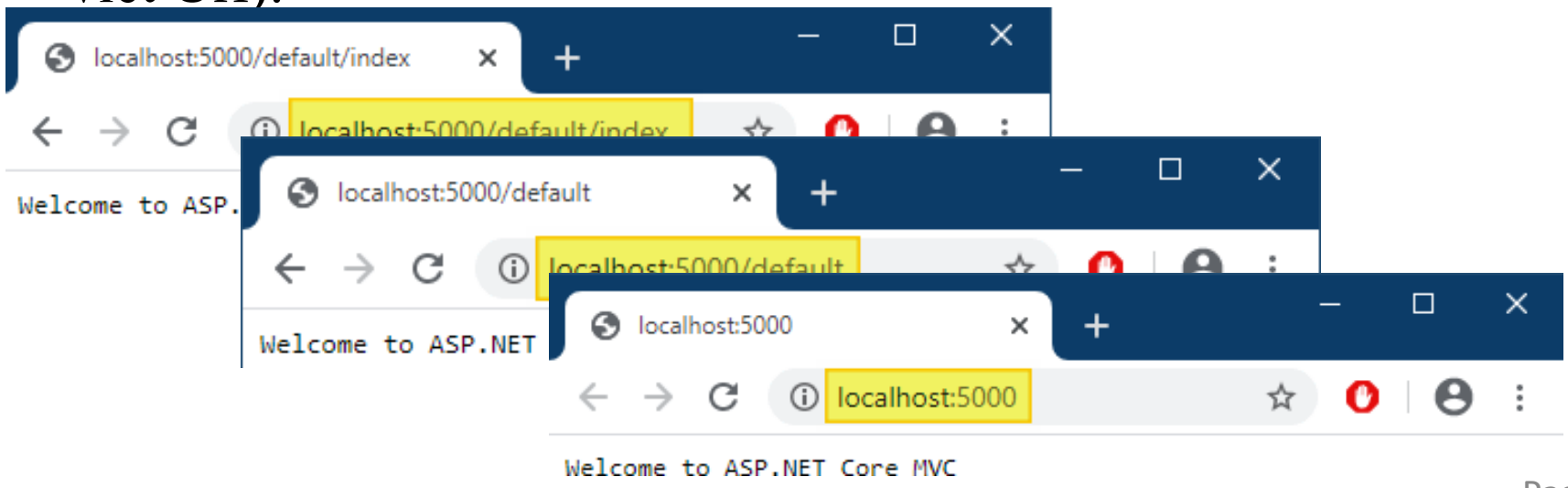
- **THỰC HÀNH 1:** Tự cấu hình dự án ASP.NET Core MVC
  - **Bước 1**: Tạo ra một dự án rỗng **ASP.NET Core**.
  - **Bước 2**: Mở file ***Startup.cs*** và viết code như sau:

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;

namespace WebApplication1 {
    public class Startup {
        public void ConfigureServices(IServiceCollection services) {
            services.AddControllersWithViews();
        }

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

            app.UseRouting();

            app.UseEndpoints(endpoints => {
                endpoints.MapControllerRoute("default", "{controller=Default}/{action=Index}/{id?}");
            });
        }
    }
}
```

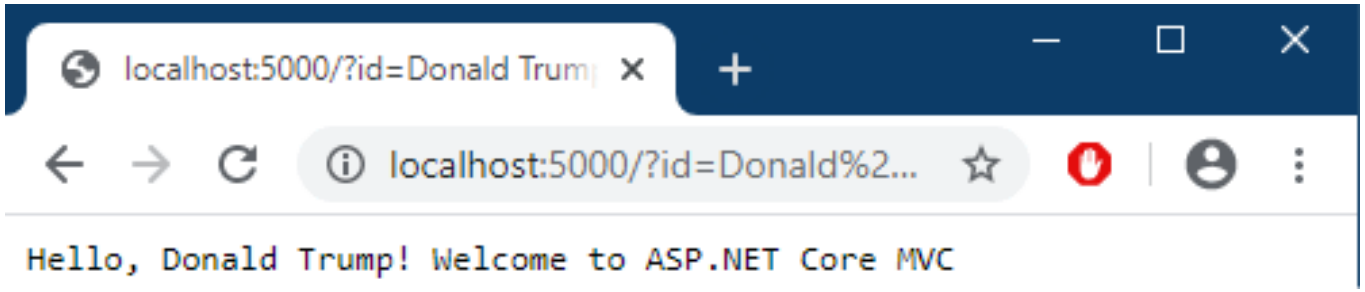
- **THỰC HÀNH 1:** Tự cấu hình dự án ASP.NET Core MVC
  - **Bước 3**: Tạo class **DefaultController** trong file cùng tên trực thuộc project và viết code như sau:

```
1.  namespace WebApplication1 {
2.      public class DefaultController : Controller {
3.          public string Index(string id) {
4.              if (string.IsNullOrEmpty(id))
5.                  return "Welcome to ASP.NET Core MVC";
6.              else
7.                  return $"Hello, {id}! Welcome to ASP.NET Core MVC";
8.          }
9.      }
10. }
```

  - Khi chạy chương trình sẽ thu được kết quả như sau (lưu ý cách viết Url):



- **THỰC HÀNH 1:** Tự cấu hình dự án ASP.NET Core MVC
  - **Bước 3**:
    - Nếu **path** có 2 **segment**, segment thứ nhất tương ứng với tên **controller**, segment thứ hai tương ứng với tên **action** trong **controller** đó. Như vậy ***/default/index*** tương ứng với gọi tới phương thức **Index** trong class ***DefaultController***.
    - Nếu không nhìn thấy **segment** thứ hai (tương đương với tên **action**), giá trị "**index**" sẽ được sử dụng. Nếu không nhìn thấy **segment** đầu, giá trị "**default**" sẽ được sử dụng. Như vậy action **Index** trong ***DefaultController*** tương ứng với url ***/default/index*** hoặc **/**. Nếu thêm chuỗi truy vấn ***?id=Donald Trump*** vào **Url** sẽ thu được kết quả như sau:



- **THỰC HÀNH 2:** Xây dựng View cho ASP.NET Core MVC
  - **Bước 1**: Xây dựng action **List** trong ***DefaultController*** như sau:

```
1.  public IActionResult List() {
2.      var model = new List<(string, string)> {
3.          ("Donald", "Trump"),
4.          ("Barack", "Obama"),
5.          ("George W.", "Bush")
6.      };
7.
8.      var view = View("/DefaultList.cshtml");
9.      view.ViewData.Model = model;
10.
11.     return view;
12. }
```

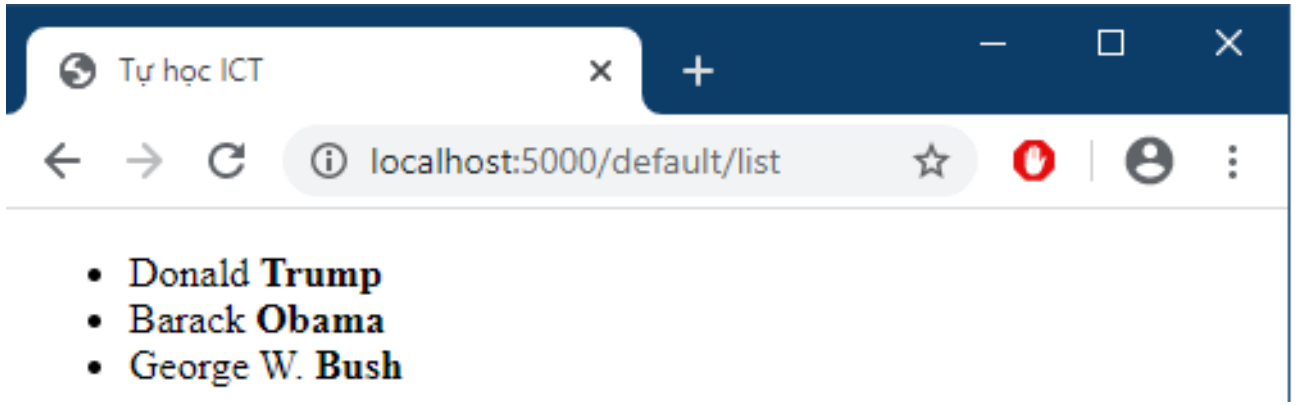
  - **Bước 2**: Tạo file ***DefaultList.cshtml*** trực thuộc dự án.

- **THỰC HÀNH 2:** Xây dựng View cho ASP.NET Core MVC
  - **Bước 3**: Viết code cho ***DefaultList.cshtml*** như sau:

```
@model List<(string firstName, string lastName)>

<!doctype html>

<html lang="en">
<head>
    <meta charset="utf-8">
    <title>Tự học ICT</title>
</head>

<body>
    <div>
        <ul>
            @foreach (var i in Model) {
                <li>@i.firstName <b>@i.lastName</b></li>
            }
        </ul>
    </div>
</body>
</html>
```

  - View trong **ASP.NET Core MVC** chỉ là một file **Razor** thông thường. Directive **@model** có tác dụng báo cho **Intellisense** biết kiểu của model để hỗ trợ nhắc code.
  - **Lưu ý: view** trong **MVC** không có **directive @page** (liên quan đến mẫu routing tới trang). Do vậy chỉ có thể sử dụng **view** kết hợp với **controller action** chứ không thể trực tiếp truy xuất view từ trình duyệt.

- **THỰC HÀNH 2:** Xây dựng View cho ASP.NET Core MVC
  - Chạy thử với url ***/default/list***, bạn thu được kết quả như sau:



(1) Model thực tế là bất kỳ dữ liệu nào cần hiển thị. Trong ví dụ trên, **model** là một danh sách các cặp giá trị (string, string)

(2) View là một file **Razor** với **directive @model** có tác dụng chỉ định kiểu cụ thể của **model** mà **view** đang cần hiển thị.

(3) File **view** được chỉ định qua lời gọi phương thức ***var view = View("/DefaultList.cshtml");*** Lối viết này báo rằng cần tìm file ***DefaultList.cshtml*** trong thư mục dự án.

(4) Giá trị của **model** được **action** truyền sang cho **view** thông qua object của **ViewData**: ***view.ViewData.Model = model;***

Có thể viết gộp (3) và (4) thành một lệnh duy nhất ***var view = View("/DefaultList.cshtml", model);***

Một số quy ước trong dự án ASP.NET Core MVC

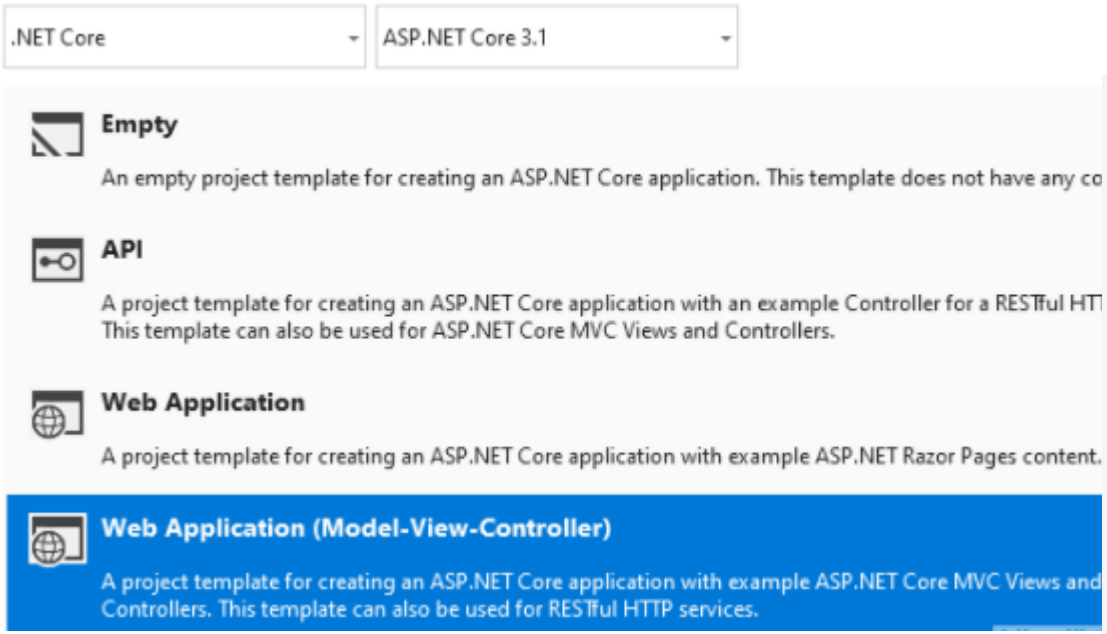
- (1) Các lớp controller nên đặt trong thư mục **Controllers** trực thuộc dự án
- (2) Tên lớp controller cần có hậu tố "**Controller**"
  - **ASP.NET Core MVC** yêu cầu các lớp **controller** phải thỏa mãn một trong hai yêu cầu:

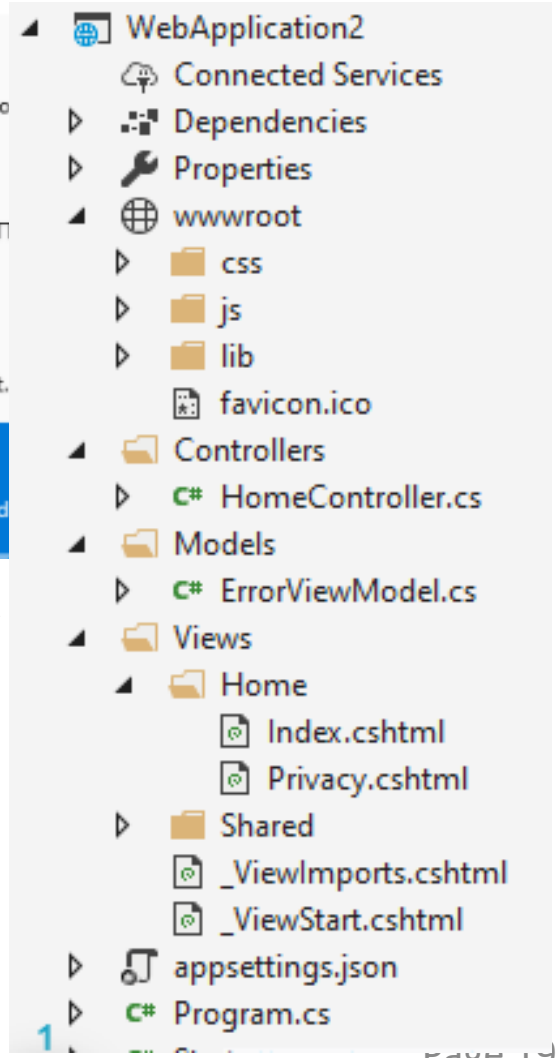
  (1) Kế thừa từ lớp cha **Controller** hoặc **ControllerBase**;

  (2) Nếu không phải lớp dẫn xuất của **Controller** (hoặc **ControllerBase**) thì tên gọi phải có hậu tố **Controller**.
- (3) Các file **Razor view** nên đặt trong thư mục **Views** trực thuộc dự án
  - Mặc định, **ASP.NET Core MVC** xác định file **view** như sau ***/Views/{Controller}/{Action}.cshtml***
  - **Ví dụ**, với action **List** trong controller **DefaultController**, **ASP.NET Core MVC** cho rằng file **Razor view** tương ứng sẽ là ***/Views/Default/List.cshtml***. Nếu không chỉ định bất kỳ thông tin gì khác về **view** (như tên hay đường dẫn), **ASP.NET Core MVC** sẽ tự tìm file tương ứng.

• Template cho dự án ASP.NET Core MVC

- Nếu sử dụng **Visual Studio**, mẫu dự án cho **MVC** được đặt tên là **Web Application (Model-View-Controller)**.





- Khi sử dụng template này bạn thu được một dự án như sau:

- **Controller trong ASP.NET Core MVC:** là class thỏa mãn các điều kiện sau:
  - (1) là loại class ***có thể khởi tạo được***, nghĩa là class có hàm tạo công khai (public constructor) + không static và không abstract.
  - (2) Có **tên gọi** kết thúc bằng "**Controller**", ví dụ ***HomeController***, ***DefaultController***.
  - (3) Là lớp **dẫn xuất** của **Controller** hoặc **ControllerBase**

  - Trong đó, điều kiện (2) và (3) không cần đồng thời thỏa mãn, nghĩa là nếu đã lớp dẫn xuất của **Controller** thì tên gọi không cần chứa hậu tố "**Controller**".
  - Nên xây dựng các lớp **controller** kế thừa từ lớp **Controller**. Lớp **Controller** cung cấp nhiều tiện ích giúp đơn giản hóa việc xây dựng các lớp **controller** cho ứng dụng. Một trong số những tiện ích quan trọng là các phương thức hỗ trợ trả kết quả (action result).
  - Lớp **ControllerBase** có vai trò tương tự như **Controller** nhưng không chứa các phương thức hỗ trợ **view**. Vì vậy **ControllerBase** thường dùng khi xây dựng **controller** trong **Web API**.

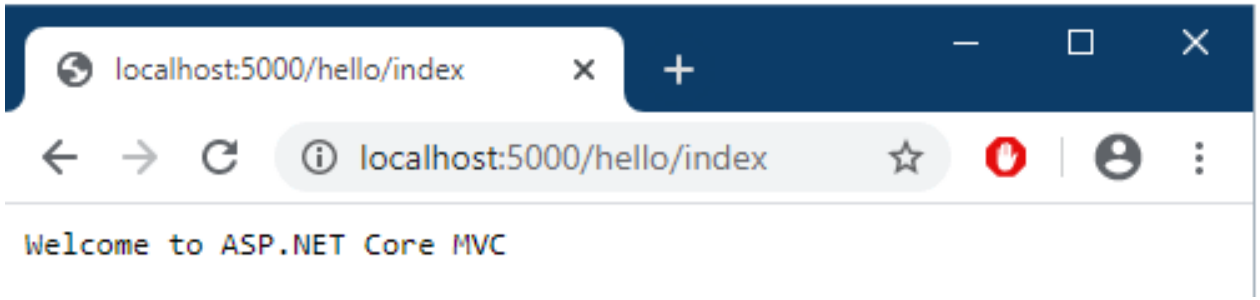
- **Tự xây dựng lớp controller**:
  - **Bước 1**: Cấu hình một dự án MVC mới từ một dự án rỗng sử dụng Startup class sau:.

```
1.  using Microsoft.AspNetCore.Builder;
2.  using Microsoft.AspNetCore.Hosting;
3.  using Microsoft.Extensions.DependencyInjection;
4.  using Microsoft.Extensions.Hosting;
5.
6.  namespace WebApplication1 {
7.      public class Startup {
8.          public void ConfigureServices(IServiceCollection services) {
9.              services.AddControllersWithViews();
10.         }
11.
12.         public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {
13.             if (env.IsDevelopment()) {
14.                 app.UseDeveloperExceptionPage();
15.             }
16.
17.             app.UseRouting();
18.
19.             app.UseEndpoints(endpoints => {
20.                 endpoints.MapControllerRoute("default", "{controller}/{action}");
21.             });
22.         }
23.     }
24. }
```

- **Tự xây dựng lớp controller:**
  - **Bước 2**: Xây dựng class **HelloController** trong file cùng tên trực thuộc dự án:

```
1.  namespace WebApplication1 {
2.      public class HelloController {
3.          public string Index() {
4.              return "Welcome to ASP.NET Core MVC";
5.          }
6.      }
7.  }
```

  - **Bước 3**: Chạy ứng dụng và nhập Url ***/hello/index***



  - Ví dụ trên xây dựng một class **controller** đơn giản nhất ***HelloController***. Class này có thể được sử dụng như một **controller** trong **ASP.NET Core MVC** vì tên gọi của nó chứa hậu tố **Controller**. Trong **HelloController** chỉ có một action **Index** trả về một chuỗi ký tự.

- **Sử dụng Razor view với controller trong ASP.NET Core MVC:**
  - **Bước 1.** Bổ sung phương thức sau vào **HelloController:**

```
1.  public ViewResult WithView() {
2.          var view = new ViewResult() { ViewName = "/HelloWithView.cshtml" };
3.          return view;
4.   }
```

  - **Bước 2**: Tạo file Razor ***HelloWithView.cshtml*** trực thuộc dự án và viết code như sau:

```
1.  <!doctype html>
2.
3.  <html lang="en">
4.  <head>
5.      <meta charset="utf-8">
6.      <title>Tự học ICT</title>
7.  </head>
8.
9.  <body>
10.     <h1>Hello world from Razor view!</h1>
11.     <p>It's @DateTime.Now.ToLongTimeString() now.</p>
12. </body>
13. </html>
```

- **Sử dụng Razor view với controller trong ASP.NET Core MVC:**
  - **Bước 3:** Chạy thử với Url ***/hello/withview*:**

  - Có thể thấy, để sử dụng Razor view cùng với **controller**, phương thức phải trả về kết quả có kiểu là **ViewResult**. **Object** của **ViewResult** chứa các thông tin cần thiết để **Mvc Middleware** tìm được file **Razor**. Khi tìm được file, **Mvc Middleware** sẽ gọi đến cơ chế **Razor** để chuyển **view** thành **HTML**.

- **Truyền view model từ controller sang view**:
  - Dữ liệu truyền từ **controller** sang **view** để hiển thị được gọi là **view model.**
  - **Bước 1**: Bổ sung các lệnh sau vào **HelloController**:

```
1.  private IModelMetadataProvider _provider;
2.  public ViewDataDictionary ViewData { get; set; }
3.  public ModelStateDictionary ModelState = new ModelStateDictionary();
4.  public HelloController(IModelMetadataProvider provider) {
5.      _provider = provider;
6.      ViewData = new ViewDataDictionary(_provider, ModelState);
7.  }
```

  - Đoạn code này khai báo thêm hai **property** cho **HelloController**: **ViewData** và **ModelState**. Để khởi tạo ViewData, bạn cần đến một object của **IModelMetadataProvider**. Object này được ASP.NET Core MVC tự sinh ra và truyền vào hàm tạo của controller thông qua cơ chế **Dependency Injection**.
  - Object **ViewData** có mhiệm vụ của nó là chứa dữ liệu có thể truy xuất được từ **view** cũng như **controller**.

- **Truyền view model từ controller sang view**:
  - **Bước 2:** Xây dựng phương thức **WithViewData** trong **HelloController**.

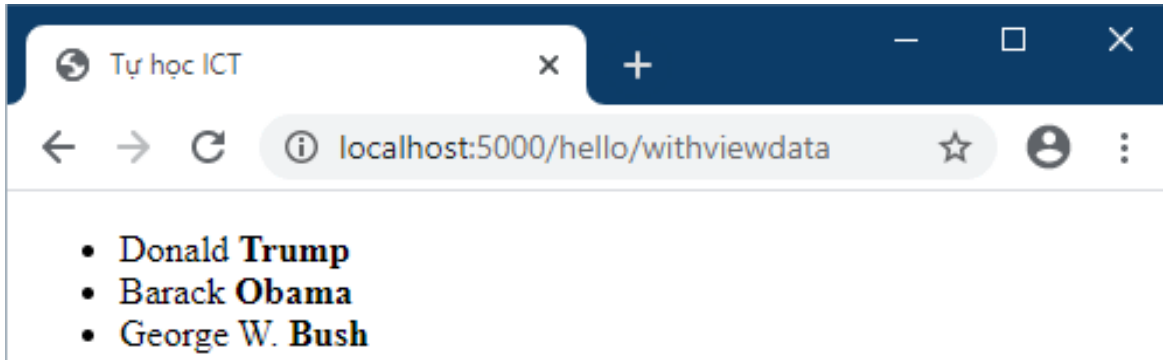
```
1.  public ViewResult WithViewData() {
2.      var model = new List<(string, string)> {
3.          ("Donald", "Trump"),
4.          ("Barack", "Obama"),
5.          ("George W.", "Bush")
6.      };
7.
8.      var view = new ViewResult() { ViewName = "/HelloWithViewData.cshtml" };
9.      view.ViewData = ViewData;
10.     view.ViewData.Model = model;
11.
12.     return view;
13. }
```

  - Trong object của **ViewResult** đã có sẵn khai báo property **ViewData**. **ViewData** của **view** được gán giá trị **ViewData** của **HelleController**. Do đó ở trong **view** có thể dễ dàng truy xuất các giá trị gán từ **controller**.
  - **Property Model** của **ViewData** có vai trò đặc biệt. **@Model** trong **view** chính là property **Model** của **ViewData**.

- **Truyền view model từ controller sang view**:
  - **Bước 3:** Xây dựng Razor view ***HelloWithViewData.cshtml*** như sau**:**

```
@model List<(string firstName, string lastName)>

<!doctype html>

<html lang="en">
<head>
    <meta charset="utf-8">
    <title>Tự học ICT</title>
</head>

<body>
    <div>
        <ul>
            @foreach (var i in Model) {
                <li>@i.firstName <b>@i.lastName</b></li>
            }
        </ul>
    </div>
</body>
</html>
```

  - Chạy ứng dụng với url /hello/withviewdata bạn thu được kết quả như sau:

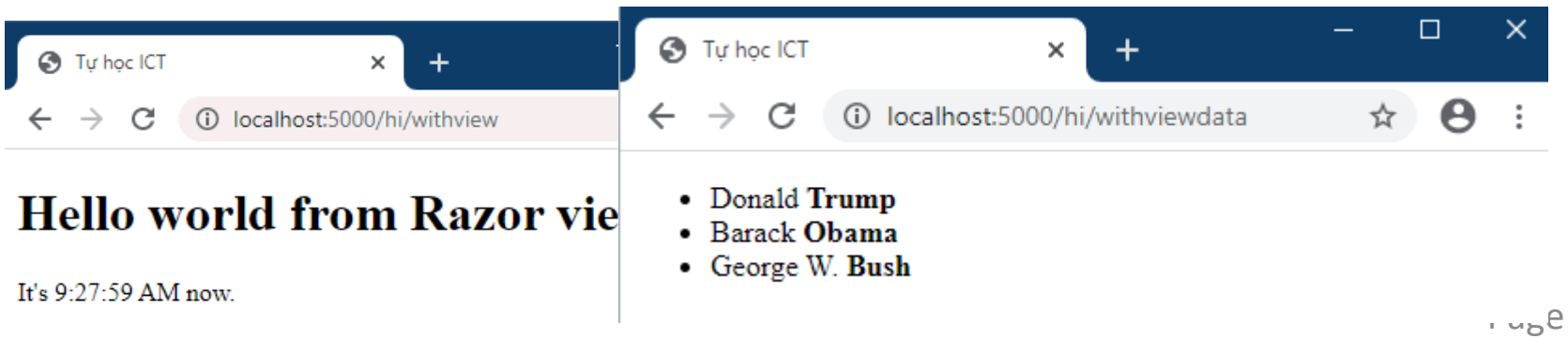


- **Lớp Controller trong ASP.NET Core MVC:**
  - Trong ví dụ trên cho thấy việc truyền dữ liệu từ **controller** sang **view** trong **ASP.NET Core MVC** mặc dù không phức tạp lắm nhưng cũng khá rắc rối.
  - Mỗi lần xây dựng **controller** mới sẽ phải lặp lại các thao tác tương tự để tạo ra **ViewData**, **ModelState**, **provider**. Khi cần chọn **view** và truyền **model**, sẽ phải lặp lại các thao tác khởi tạo và gán giá trị.
  - **ASP.NET Core MVC** tạo sẵn class **Controller** giúp giải quyết hết các vấn đề trên.

- **Lớp Controller trong ASP.NET Core MVC:**
  - Xây dựng **controller** mới **HiController** trong file tương ứng và viết code như sau:

```
1.  using System.Collections.Generic;
2.
3.  using Microsoft.AspNetCore.Mvc;
4.
5.  namespace WebApplication1 {
6.      public class HiController : Controller {
7.          public ViewResult WithView() {
8.              return View("/HelloWithView.cshtml");
9.          }
10.
11.         public ViewResult WithViewData() {
12.             var model = new List<(string, string)> {
13.                 ("Donald", "Trump"),
14.                 ("Barack", "Obama"),
15.                 ("George W.", "Bush")
16.             };
17.
18.             return View("/HelloWithViewData.cshtml", model);
19.         }
20.     }
21. }
```

  - Chạy ứng dụng với các Url tương ứng **/hi/withview** và **/hi/withviewdata**:



- **Action** trong **ASP.NET Core MVC** là những phương thức public của **controller** có thể được sử dụng để xử lý truy vấn.
- **Action result** là cách gọi chung của kết quả thực hiện của **action**. Kết quả trả về của mỗi **action** có thể được biểu diễn ở dạng giao diện **HTML**, ở dạng dữ liệu (**XML**, **JSON**) hoặc các dạng thức đặc biệt khác.
- **Action trong ASP.NET Core MVC**
  - Trong mô tả kiến trúc **MVC** tổng quát, **controller** được xem là điểm khởi đầu (entry point) của quá trình sinh giao diện. Tuy nhiên, trong **ASP.NET Core MVC**, phương thức có khả năng xử lý truy vấn trong lớp **controller** mới thực sự là điểm khởi đầu. Người ta gọi loại phương thức như vậy là **action.**
  - Trong **ASP.NET Core MVC**, mỗi URL tương ứng với một phương thức.
  - Khi một truy vấn đến server, **Mvc Middleware** căn cứ vào URL để xác định phương thức cần thực thi. Ví dụ, truy vấn chứa Url ***/home/index*** tương ứng với thực thi phương thức **Index** trong lớp **HomeController**.
  - Khi này, **Mvc Middleware** sẽ tìm kiếm trong assembly class **HomeController**, hoặc class con **Home** của **Controller**. Nếu tìm thấy, nó sẽ khởi tạo object của class này và tìm kiếm phương thức public **Index** trong đó. Nếu tìm thấy, nó sẽ thực thi phương thức **Index**.
  - **Index** là một phương thức **action** của **Home controller**.

- Bất kỳ phương thức public nào của lớp **controller** đều có thể trở thành phương thức **action**. Lớp **controller** có thể chứa không giới hạn các **action**. Mỗi action chịu các trách nhiệm sau:
  - Xác nhận tính chính xác của truy vấn
  - Thực thi các lệnh tương ứng với truy vấn
  - Lựa chọn loại phản hồi phù hợp.
- Trong 3 nhiệm vụ trên, nhiệm vụ "lựa chọn loại phản hồi" là bắt buộc đối với một action.
- Ở dạng đơn giản nhất, loại phản hồi có thể chỉ là một chuỗi ký tự như đã thấy trong phần thực hành 1 của bài học về dự án ASP.NET Core MVC
- Đối với ứng dụng web thông thường (trả HTML lại cho trình duyệt), action thường sẽ trả về một object thuộc kiểu **ActionResult** (hoặc **IActionResult**). **Mvc Middleware** sẽ sử dụng object này để sinh ra **HTML**.

- Lưu ý rằng, **action** không trực tiếp sinh ra **view**. Thay vào đó, **action** lựa chọn loại **view** phù hợp và chuẩn bị dữ liệu cho **view**. **Mvc Middleware** sử dụng **view engine** (Razor) để biến **view** (vốn là một template + dữ liệu) thành kết quả cuối cùng (**HTML**)..
- Một điểm lưu ý nữa là không nên thực hiện các loại tính toán nghiệp vụ trong action. Đây là một lỗi rất thường gặp ở người mới học lập trình.
- Thay vào đó, nên gọi dịch vụ phù hợp từ **application model**. **Ví dụ**, nếu cần tải dữ liệu từ cơ sở dữ liệu, không nên viết code xử lý cơ sở dữ liệu trực tiếp trong thân **action**. Thay vào đó, nên xây dựng một class riêng chuyên cho tương tác với cơ sở dữ liệu và gọi đến class này trong action.

Tham số cho action

- Một truy vấn có thể chứa tham số. Ví dụ, một truy vấn để lấy một sản phẩm xác định sẽ thường phải gửi kèm Id của sản phẩm tương ứng.
- Tham số của truy vấn có thể nằm trong URL (route data), nằm trog chuỗi truy vấn (query string), nằm trong đầu (header) hoặc thân truy vấn (request body).
- **Mvc Middleware** có khả năng trích giá trị tham số từ truy vấn và biến đổi (**convert**) sang kiểu tương ứng của **.NET**. Nếu **action** yêu cầu tham số, giá trị lấy từ truy vấn (sau khi convert) sẽ truyền sang cho **action**. Nếu **action** không cần tham số, giá trị trích ra sẽ không được sử dụng.
- Tham số của **action** có thể là các kiểu cơ sở của **.NET** (string, int, bool, v.v.), cũng có thể là kiểu phức tạp do người dùng định nghĩa (struct, class).
- Quá trình trích – biến đổi kiểu này được gọi là **model binding**. **Object** tạo ra trong quá trình **model binding** và đóng vai trò tham số đầu vào cho **action** được gọi là **binding model**
- Cơ chế **model binding** giúp đơn giản hóa việc tiếp nhận tham số từ truy vấn. Nói chung, nhờ **model binding**, bạn có thể khai báo phương thức **action** như một phương thức **C#** thông thường mà không cần quan tâm đến việc trích và biến đổi tham số từ truy vấn.

Các loại kết quả hoạt động của action

- Kết quả hoạt động của một **action** là đưa ra quyết định xem loại phản hồi nào cần trả lại và chuẩn bị dữ liệu tương ứng.
- Phản hồi cho một truy vấn **HTTP** có thể chứa **HTML** (sinh động từ khuôn mẫu), chứa dữ liệu (như JSON hay XML), là một hành động (điều hướng sang một URL khác), là thông báo lỗi (ví dụ 404 not found), chứa một file tĩnh, v.v..
- Tự bản thân **action** không thực hiện việc sinh phản hồi. Việc sinh phản hồi là nhiệm vụ của **Mvc Middleware**.
- **Ví dụ**, nếu **action** quyết định rằng cần sinh ra phản hồi chứa **HTML**, nó sẽ trả về object **ViewResult**. Trong object **ViewResult** chứa thông tin về file mẫu (**Razor view**) và dữ liệu (**view model**) cần thiết. **Mvc Middleware** sử dụng **object** này và kích hoạt **Razor view engine** để sinh ra **HTML** và đóng gói **HTML** đó vào gói tin phản hồi **HTTP**.
- Kết quả hoạt động của **action** được thể hiện qua kiểu dữ liệu trả về của phương thức.
- Trong trường hợp đơn giản nhất, kiểu dữ liệu trả về của **action** có thể chỉ là một chuỗi ký tự.
- Tuy nhiên, để thể hiện những kết quả hoạt động phức tạp hơn của **action**, **ASP.NET Core** tạo ra những kiểu trả về đặc biệt cho từng trường hợp. **Ví dụ**, kiểu trả về **ViewResult** thể hiện cần sinh ra **HTML** từ file **Razor**, **RedirectResult** thể hiện gói tin phản hồi ***302 redirect***, **NotFoundResult** thể hiện gói tin **404**.
- **ViewResult**, **RedirectResult**, hay **NotFoundResult** đều là các class thực thi một giao diện chung: IActionResult.

- **IActionResult và ActionResult**
  - Mọi class thể hiện kết quả hoạt động của action đều kế thừa từ lớp trừu tượng **ActionResult** hoặc thực thi giao diện **IActionResult**.
  - **ActionResult** và **IActionResult** là chung của **ASP.NET Core** chứ không phải tính năng riêng của **Razor Pages** hay **MVC**.
  - Trong Razor Pages **ActionResult** (**IActionResult**) không quá quan trọng.
  - Tuy nhiên trong **MVC**, gần như mọi **action** đều trả về một trong các kiểu dẫn xuất **ActionResult** hoặc kiểu thực thi **IActionResult**.
  - Khi mới làm quen với **MVC,** có thể phân vân xem nên sử dụng **ActionResult** hay **IActionResult** làm kiểu trả về của **action**. Thực tế là không có gì khác biệt nếu sử dụng các class **action result** do **ASP.NET Core** xây dựng sẵn. Các class này là hậu duệ của **ActionResult**, do vậy cũng đồng thời thực thi **IActionResult**.

- **IActionResult và ActionResult**
  - Dưới đây là một số class thường gặp nhất:
    - **ViewResult** – trả lại phản hồi chứa HTML. Đây là **action result** thường dùng nhất khi dùng **MVC** để viết các ứng dụng web **HTML** truyền thống
    - **RedirectResult** – trả lại phản hồi điều hướng trang **302**. Loại kết quả này dùng khi cần điều hướng người dùng tới một **URL** khác
    - **FileResult** – trả lại một file tĩnh trong phản hồi
    - **ContentResult** – trả lại loại phản hồi chỉ chứa một chuỗi ký tự
    - **StatusCodeResult** – trả lại một mã phản hồi (trong header)
    - **NotFoundResult** – trả lại ***mã 404 not found***.

- **Phương thức hỗ trợ của Controller**
  - Để đơn giản hóa hơn nữa việc trả về kết quả của **action**, lớp **Controller** đã xây dựng sẵn nhiều phương thức hỗ trợ.
  - Các phương thức hỗ trợ xây dựng sẵn trong lớp **Controller** được đặt tên theo class mà nó hỗ trợ nhưng loại bỏ đi hậu tố **Result**.
  - **Ví dụ**, phương thức **View** hỗ trợ trả lại kết quả thuộc loại **ViewResult**. Sử dụng phương thức **View** đơn giản hơn rất nhiều so với trực tiếp khởi tạo **ViewResult**.
  - Tương tự như vậy,
    - **RedirectResult** có phương thức hỗ trợ là **Redirect**()
    - **NotFoundResult** có phương thức hỗ trợ là **NotFound**
    - **FileResult** có phương thức hỗ trợ là **File**(),
    - v.v..

- **Phương thức hỗ trợ của Controller**
  - Giả sử trường hợp bạn muốn trả về một **view** sinh **HTML** có kiểu **ViewResult**, bạn cần viết code như sau:

```
1.  IActionResult Index(){
2.      var view = new ViewResult();
3.      view.ViewData = ViewData; // truyền ViewData của controller cho view
4.      view.ViewData.Model = ("Donald", "Trump"); // truyền view model sang cho view
5.      return view;
6.  }
```

  - Nếu sử dụng phương thức hỗ trợ **View**(), code của bạn giờ sẽ như sau:

```
1.  IActionResult Index(){
2.      return View(("Donald", "Trump"));
3.  }
```

  - Như vậy, **View** giúp chúng ta gộp lệnh khởi tạo **ViewResult**, truyền view **model**, và truyền **ViewData** vào một lệnh duy nhất.

- **Action overloading trong controller**
  - **Action** thực chất chỉ là một phương thức public bình thường trong **controller** class.
  - Trong class C# cho phép tình trạng có nhiều phương thức trùng tên nhưng khác danh sách tham số. Hiện tượng này trong C# được gọi là **method overloading** (nạp chồng phương thức). C# compiler sử dụng danh sách tham số để phân biệt các overload của cùng một phương thức.
  - Trong ASP.NET Core điều này vẫn đúng.
  - Tuy nhiên, nếu bạn xây dựng **controller** với hai **overload** của cùng một **action**, khi chạy ứng dụng và gọi tới **action** đó bạn sẽ gặp lỗi ***"AmbiguousMatchException: The request matched multiple endpoints."***
  - Hãy xây dựng controller như sau:

```
1.  namespace WebApplication1.Controllers {
2.      public class GreetingController {
3.          public string Hi() => $"Welcome to tuhocict.com";
4.          public string Hi(string name) => $"Welcome, {name}";
5.      }
6.  }
```

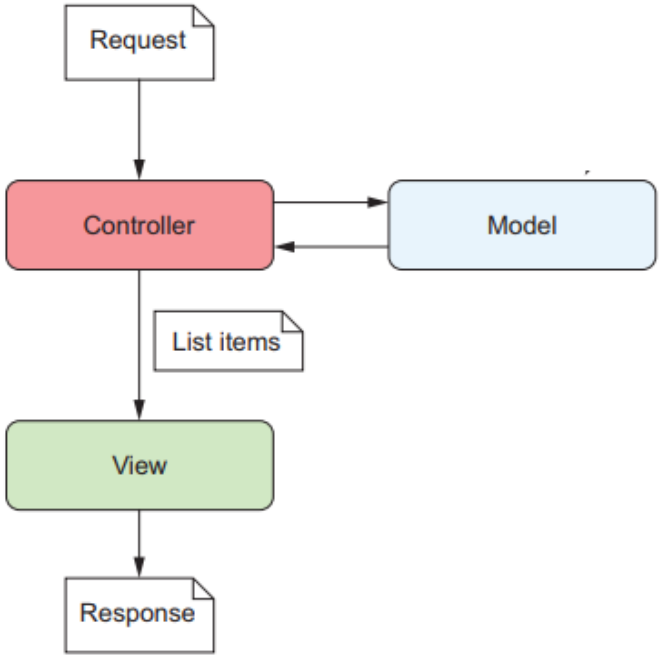
  - Chạy ứng dụng và nhập url ***/greeting/hi***, bạn sẽ gặp lỗi ***"AmbiguousMatchException: The request matched multiple endpoints."***

- **Action overloading trong controller**
  - Để sử dụng **method overloading** trong **controller class**, cần một loại kỹ thuật khác.
  - Nếu chỉ định loại truy vấn nào **action** sẽ xử lý, **ASP.NET Core** có thể phân biệt các **action** với nhau. **Ví dụ**, chỉ định phương thức **Hi**() xử lý truy vấn **GET**, **Hi(string)** xử lý truy vấn **POST**, **Hi(string, string)** xử lý **PUT**, **Hi(int)** xử lý **DELETE**
  - Khi một truy vấn tới, **ASP.NET Core** sử dụng thêm thông tin về loại truy vấn để lựa chọn **action**.
  - Vẫn ví dụ với **GreeingController**, giờ bạn thay đổi code như sau:

```
1.  using Microsoft.AspNetCore.Mvc;
2.
3.  namespace WebApplication1.Controllers {
4.      public class GreetingController {
5.          [HttpGet]
6.          public string Hi() => $"Welcome to tuhocict.com";
7.
8.          [HttpPost]
9.          public string Hi(string name) => $"Welcome, {name}";
10.     }
11. }
```

  - Để ý attribute **[HttpGet]** và **[HttpPost]** trước môi **action**. Các **attribute** này được gọi là **HTTP verb attribute**. Chúng có tác dụng giúp chỉ định loại truy vấn mà **action** này có thể xử lý.
  - Giờ nếu bạn chạy ứng dụng với url ***/greeting/hi*** sẽ không còn lỗi nữa. Khi này action **Hi()** sẽ được thực thi.

- Trong mô tả chung của mẫu kiến trúc MVC, view là thành phần chịu trách nhiệm sinh ra giao diện cho ứng dụng. Đây là khâu cuối cùng trong chuỗi xử lý yêu cầu của người dùng trước khi trả lại kết quả.
- Kiến trúc MVC qua sơ đồ minh họa dưới đây



- Tuy nhiên, mỗi framework khi thực thi MVC lại diễn đạt view và mối quan hệ giữa nó với các thành phần khác theo cách riêng. Chúng ta xem xét view trong quan hệ với các thành phần còn lại của MVC trong ASP.NET Core MVC.

• Làm việc với view trong ASP.NET Core MVC

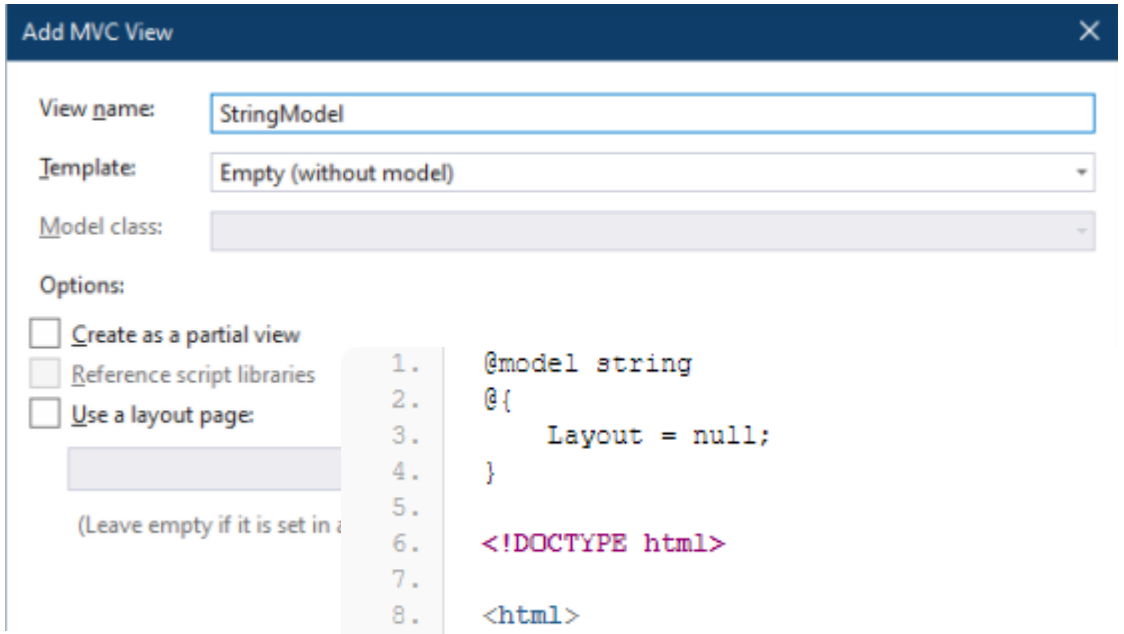
- **Bước 1**: Tạo một project rỗng **WebApplication1** và cấu hình để biến nó thành một **project MVC** sử dụng ***Startup*** class sau:

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;

namespace WebApplication1 {
    public class Startup {
        public void ConfigureServices(IServiceCollection services) {
            services.AddControllersWithViews();
        }

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

            app.UseRouting();

            app.UseEndpoints(endpoints => {
                endpoints.MapControllerRoute("default", "{controller}/{action}");
            });
        }
    }
}
```

- **Làm việc với view trong ASP.NET Core MVC**
  - **Bước 2**: Tạo thư mục **Controllers**. Trong thư mục này tạo class mới **HomeController** trong file ***HomeController.cs***.

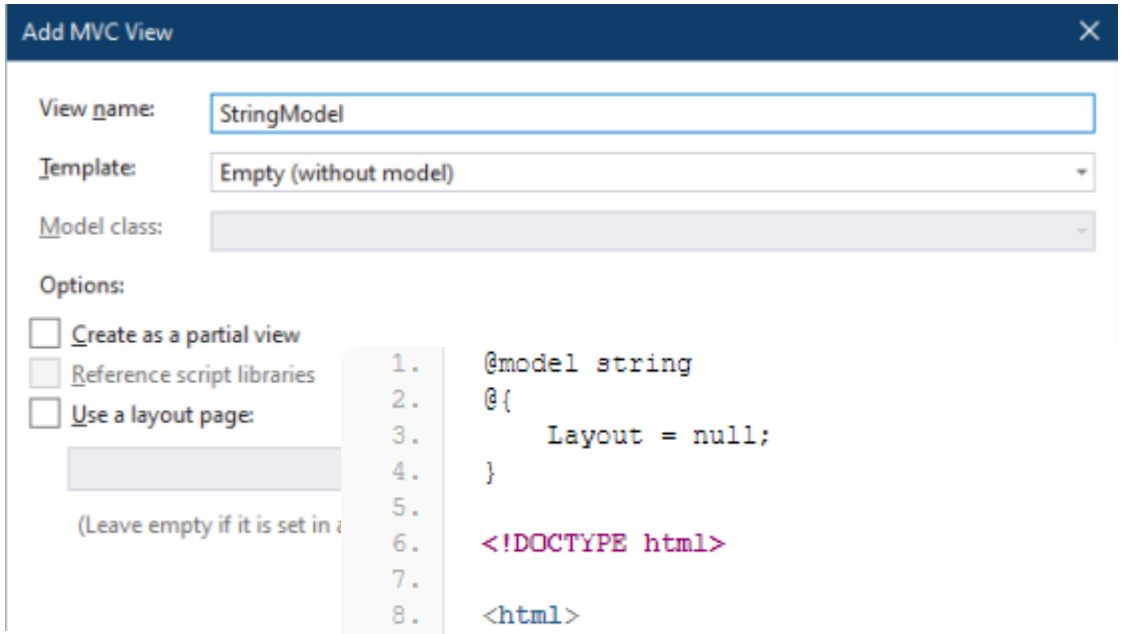
```
1.  using System;
2.  using Microsoft.AspNetCore.Mvc;
3.
4.  namespace WebApplication1.Controllers {
5.      public class HomeController : Controller {
6.          public IActionResult StringModel() {
7.              ViewData.Model = "Hello world from Home-Index action";
8.              var view = new ViewResult();
9.              view.ViewName = "/Views/Home/StringModel.cshtml";
10.             view.ViewData = ViewData;
11.             return view;
12.         }
13.
14.         public IActionResult TupleModel() {
15.             ViewData.Model = ("Donald", "Trump", new DateTime(1900, 12, 31));
16.             var view = new ViewResult();
17.             view.ViewData = ViewData;
18.             return view;
19.         }
20.     }
21. }
```

- **Làm việc với view trong ASP.NET Core MVC**
  - **Bước 3**: Tạo thư mục **Views** trực thuộc dự án. Trong thư mục **Views** tạo thư mục **Home**.
  - Tạo file **view** bằng cách click phải chuột vào thư mục **Home**, chọn ***Add*** => ***View***. Đặt tên cho view là **StringModel**.




```
@model string
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Index</title>
</head>
<body>
    <h1>@Model</h1>
</body>
</html>
```

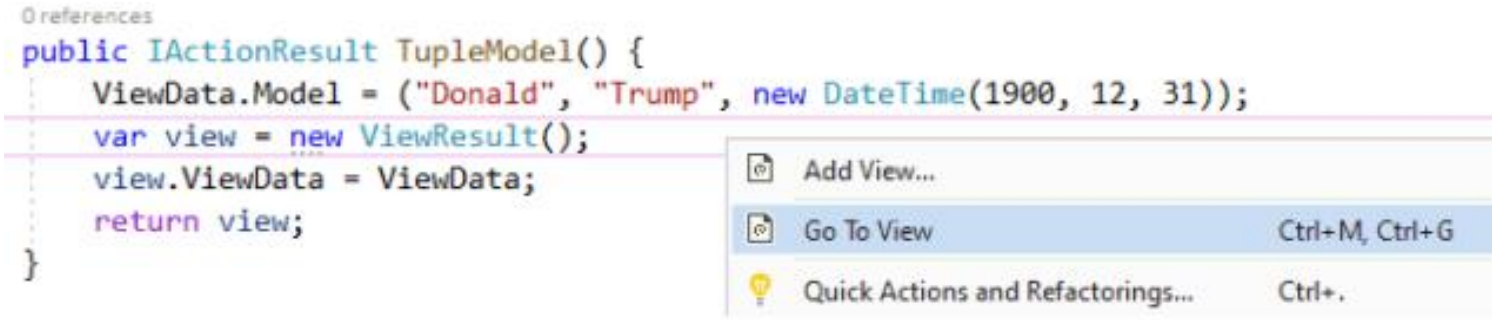
- **Làm việc với view trong ASP.NET Core MVC**
  - **Bước 3**: Tạo thư mục **Views** trực thuộc dự án. Trong thư mục **Views** tạo thư mục **Home**.
  - Tạo file **view** bằng cách click phải chuột vào thư mục **Home**, chọn ***Add*** => ***View***. Đặt tên cho view là **StringModel**.




```
1.  @model string
2.  @{
3.      Layout = null;
4.  }
5.
6.  <!DOCTYPE html>
7.
8.  <html>
9.  <head>
10.     <meta name="viewport" content="width=device-width" />
11.     <title>Index</title>
12. </head>
13. <body>
14.     <h1>@Model</h1>
15. </body>
16. </html>
```

• Làm việc với view trong ASP.NET Core MVC

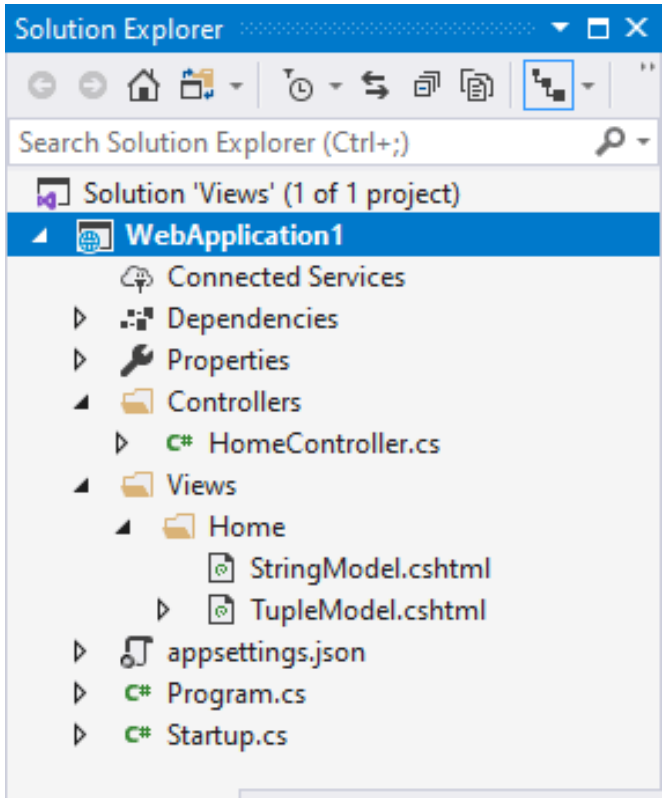
- **Bước 4**: click phải vào một dòng bất kỳ bên trong phương thức TupleModel và chọn Add View.



- Từ hộp thoại Add View tạo một view với có tên TupleModel.cshtml. Viết code cho TupleModel.cshtml như sau:

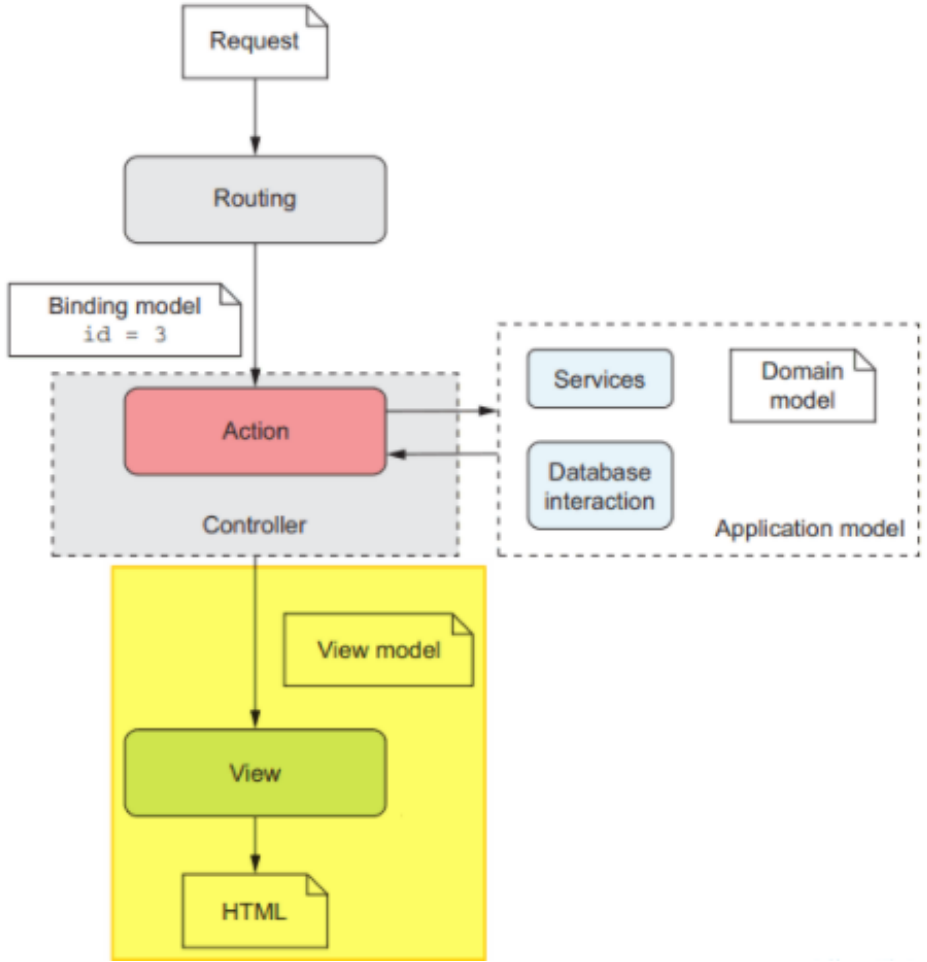
```
@model (string FirstName, string LastName, DateTime BirthDay)
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>TupleModel</title>
</head>
<body>
    <h1>Hello, @Model.FirstName <strong>@Model.LastName</strong></h1>
    <h2>Your birthday is @Model.BirthDay</h2>
</body>
</html>
```

• Làm việc với view trong ASP.NET Core MVC



- Dịch và chạy thử ứng dụng với hai URL
  - (1) /home/stringmodel
  - (2) /home/tuplemodel

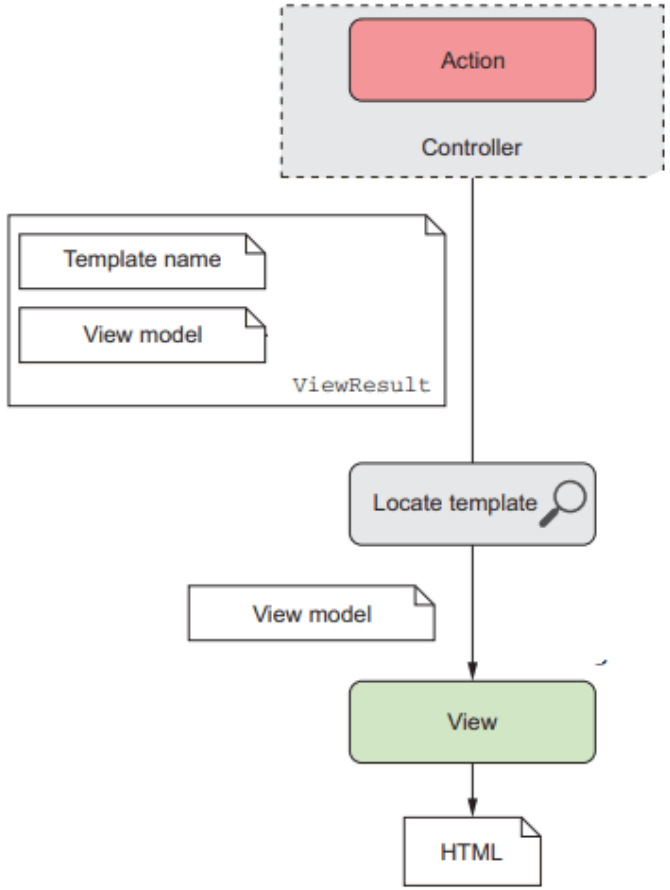
- **Quan hệ action – view**
  - Trong một ứng ụng web, giao diện ứng dụng được tạo ra bởi tổ hợp **HTML** (cho nội dung) + **CSS** (cho định dạng) + **JavaScript** (cho tương tác). Các phần của giao diện được đóng gói trong gói tin phản hồi **HTTP** để trả về cho trình duyệt.



- **Quan hệ action – view**
  - Trong bài học về **controller** và **action**, **action** chỉ định **view** để thể hiện dữ liệu, đồng thời cung cấp dữ liệu cần hiển thị.
  - Trong ví dụ trên chúng ta có **HomeController** với hai **action StringModel** và **TupleModel**. **StringModel** chỉ định rõ ràng là phải sử dụng file ***/Views/Home/StringModel.cshtml***. Đồng thời cung cấp dữ liệu là một chuỗi ký tự ***"Hello world ... "***
  - Tự bản thân **view** trong **MVC** không phải là giao diện. **View** là khuôn mẫu để tạo ra giao diện từ dữ liệu.
  - Để sinh ra giao diện, **view** cần biết dữ liệu sẽ được hiển thị. Dữ liệu này do **controller *action*** cung cấp. **Action** lấy dữ liệu từ việc tương tác với **model** (chính xác hơn là application model). Dữ liệu này có thể là một **object**, một danh sách **object**, v.v.. Để phân biệt, trong **ASP.NET Core MVC**, loại dữ liệu này được gọi là **view model**.

• Quan hệ action – view

- Để thể hiện thông tin, **view** cung cấp một khuôn mẫu (**template**) chứa những thông tin ***tĩnh*** (từ quá trình thiết kế) ghép nối với thông tin ***động*** (từ **view model**). Khuôn mẫu này trong **ASP.NET Core MVC** là các file **cshtml** được xây dựng bằng **Razor** – loại cú pháp đặc biệt kết hợp giữa HTML (cho thông tin tĩnh) và C# (cho thông tin động).
- Hãy để ý biến **Model** và **directive @model** trong các file **cshtml** ở trên. Đây là cách **View** trong **MVC** sử dụng **view model** do **controller action** cung cấp.



- **Directive @model** chỉ định kiểu của **Model**. **Model** là **property** chứa **view model** mà **action** truyền sang cho **view**.
- Qua ví dụ trên, **Model** sử dụng trên **Razor view** chính là **property ViewData.Model** nhận từ **Controller**. Mọi **view** đều tham chiếu tới **ViewData** của **Controller**. "**Model**" chỉ là một cách sử dụng tắt trên view của **ViewData.Model** cho tiện lợi.

- **File và thư mục cho view**
  - Mỗi view trong **ASP.NET Core MVC** thường tương ứng với một **controller action**, và mỗi **controller action** thường có một **view** tương ứng. Do vậy, giữa tên và đường dẫn của **view** với **controller action** có một quy ước xác định: tên gọi và đường dẫn của **view** cần tương ứng với cấu trúc ***/Views/{controller}/{action}.cshtml.***
  - **Ví dụ**, phương thức action **Index** của **HomeController** sẽ có **view** tương ứng là ***/Views/Home/Index.cshtml***. Và khi nhìn đến đường dẫn của file **cshtml** này ta cũng hình dung ra ngay nó là **view** tương ứng của **action** nào.
  - Trên thực tế, nếu bạn không chỉ định tên **view**, ASP.NET Core MVC sẽ làm như sau:
    - Tìm xem có thư mục ***/Views/{controller}*** hay không.
    - Nếu có thư mục này, tiếp tục tìm file ***{action}.cshtml*** trong đó. Nếu không thấy thư mục này thì sẽ tìm kiếm ngay ***{action}.cshtml*** trong ***/Views/Shared***.
    - Nếu không tìm thấy file trong ***/Views/{controller}*** sẽ tiếp tục tìm file ***{action}.cshtml*** trong thư mục ***/Views/Shared***.

- **Sử dụng phương thức hỗ trợ của Controller**
  - **Bước 1**: Bổ sung ba phương thức sau vào **HomeController**:

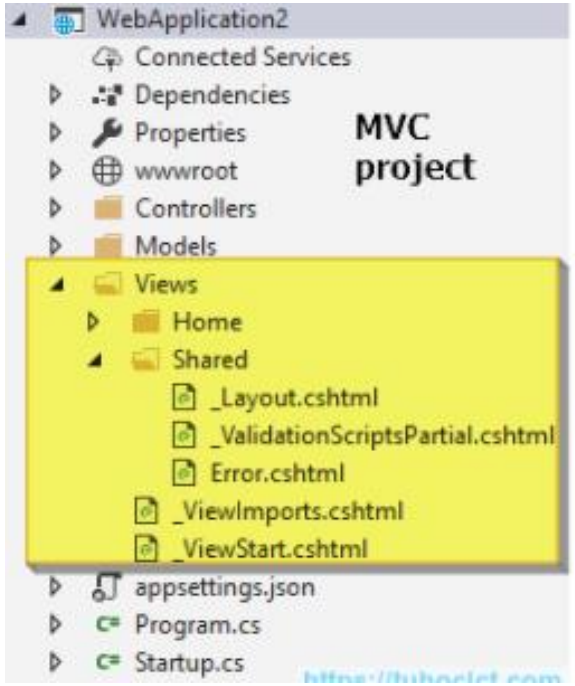
```
1.  public IActionResult ResultHelper() {
2.      return View();
3.  }
4.
5.  public IActionResult SpecifyView() {
6.      return View("ResultHelper");
7.  }
8.
9.  public IActionResult ReuseView() {
10.     var model = ("Putin", "Vladimir", new DateTime(1990, 5, 3));
11.     return View("TupleModel", model);
12. }
```

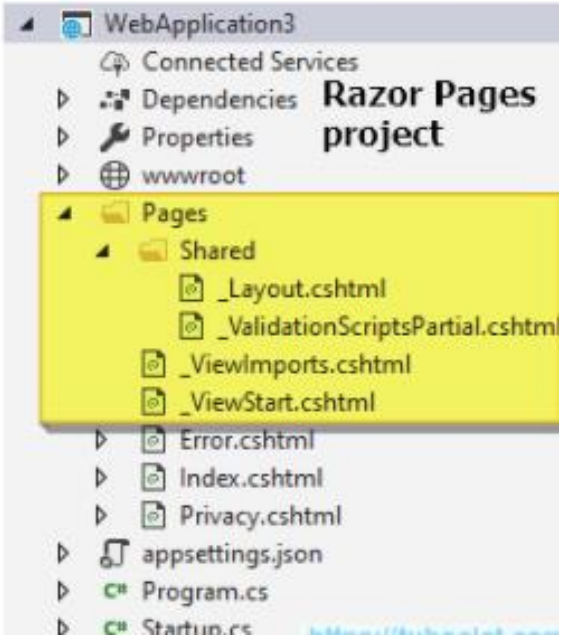
  - **Bước 2:** Sử dụng một trong những kỹ thuật đã biết để tạo **view** tương ứng với **action** này (***ResultHelper.cshtml***) và viết code như sau:

```
1.  @{
2.      Layout = null;
3.  }
4.
5.  <!DOCTYPE html>
6.
7.  <html>
8.  <head>
9.      <meta name="viewport" content="width=device-width" />
10.     <title>ResultHelper</title>
11. </head>
12. <body>
13.     <h1>This view was located automatically</h1>
14. </body>
15. </html>
```

  - Bạn có thể chạy thử ứng dụng với URL ***/home/resulthelper***, ***/home/specifyview*** và ***/home/reuseview***

- **Layout, view start, section, view imports trong MVC**
  - Do cùng sử dụng **Razor view engine**, **view** trong **MVC** hoạt động hoàn toàn giống như trang nội dung (**content page**) của **Razor Pages**. Nghĩa là cũng có thể sử dụng **layout**, **view start**, **section** hay **view imports** trong ứng dụng **MVC**
  - Sự khác biệt duy nhất là thư mục chứa **view** trong **MVC** là **Views**, còn trong **Razor Pages** trang nội dung được đặt trong thư mục **/Pages**. Tức là **/Views** trong **MVC** hoàn toàn tương đương với **/Pages** trong **Razor Pages**.





• Layout, view start, section, view imports trong MVC

- Template cho dự án MVC cũng đặt tên cho file layout mặc định là ***_Layout.cshtml*** và đặt nó trong thư mục ***/Views/Shared***
- **Ví dụ**, nếu không tìm thấy **view** cho **action** trong thư mục quy ước ***/Views/{controller}/{action}.cshtml***, **Razor view engine** sẽ tìm kiếm file ***{action}.cshtml*** trong thư mục ***/Views/Shared***.
- Trong layout có thể sử dụng **section** để chỉ định vị trí xuất nội dung với lệnh **@RenderSection**. Trong layout mặc định cũng đặt lệnh ***@RenderSection("Scripts", false)*** ở ngay trước khi đóng thẻ **</body>** giúp bạn chèn các khối **javascript** từ các **view** khi cần thiết.
- File ***_ViewStart.cshtml*** cho phép chạy một đoạn code **Razor** chung trước khi xử lý bất kỳ view nào. Do vậy **_ViewStart** được lợi dụng để chỉ định **layout** chung cho tất cả các **view**:

```
@{ Layout = "_Layout" }
```

- Lưu ý tên file bắt buộc phải là ***_ViewStart.cshtml***. Trong dự án có thể sử dụng nhiều ***_ViewStart*** khác nhau. ***Mỗi _ViewStart*** có tác dụng trong thư mục chứa nó và tất cả thư mục con. ***_ViewStart*** trong thư mục con sẽ có tác dụng mạnh hơn ***_ViewStart*** từ thư mục cha. ***_ViewStart*** trong thư mục **Views** sẽ có tác dụng với tất cả các **view**.

- **Routing** trong **ASP.NET Core** là quá trình ánh xạ (mapping) **truy vấn HTTP** với một đối tượng xử lý truy vấn đó (**handler**). Trong **MVC**, **handler** chính là phương thức **action** của **controller**. Tức là, **routing** trong **ASP.NET Core MVC** là cơ chế ánh xạ giữa truy vấn **HTTP** và **controller action**.
- Tức là, khi truy vấn tới, cơ chế **routing** xác định xem có **action** nào tương ứng với nó hay không để thực thi. Khi xây dựng một **controller** và các **action**, **routing** cũng xác định một tập hợp các **Url** hợp lệ tương ứng cho **controller** và **action** đó.
- ASP.NET Core MVC sử dụng hai cách thức khác nhau để chỉ định:
  - (1) sử dụng một **template** chung cho nhiều **controller action**, gọi là *conventional routing*
  - (2) chỉ định **Url** cụ thể cho từng **controller action** riêng biệt, gọi là *attribute routing*

Routing mặc định của dự án MVC

- 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**

```
1.  app.UseEndpoints(endpoints => {
2.      endpoints.MapControllerRoute(
3.          name: "default",
4.          pattern: "{controller=Home}/{action=Index}/{id?}");
5.  });
```

- **"{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.

- **Routing mặc định của dự án MVC**
  - 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.
  - 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.

Routing trong ASP.NET Core MVC

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

Routing trong ASP.NET Core MVC

- **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** 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**, cần tự mình quản lý một danh sách các **routing** riêng rẽ.
- Phương pháp **routing** này hữu dụng hơn khi xây dựng **Web API**.
- **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.

• Conventional routing trong ASP.NET Core MVC

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

```
ic 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**.
- Mặc định **ASP.NET Core MVC** định nghĩa sẵn cho bạn "**default**" **template**.
- Có thể định nghĩa bao nhiêu **template** tùy thích.

• Conventional routing trong ASP.NET Core MVC

- 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.
- Nên bố trí **template** cụ thể hơn ở trên, **template** tổng quát hơn ở dưới.
- 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?}");
});
```

- 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***

• Conventional routing trong ASP.NET Core MVC

- 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.

```
1.  using Microsoft.AspNetCore.Mvc;
2.
3.  namespace WebApplication1.Controllers {
4.      public class DefaultController : Controller {
5.          public IActionResult Hi(string fname, string lname) {
6.              return View((fname, lname));
7.          }
8.      }
9.  }
```

- 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, 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**.

- **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 (như **{lname}** và **{fname}** bên trên).
    - Tổ hợp của cả **literal** và **route parameter**. Đây chính là cách chúng ta tạo ra **segment** thứ hai của template "***hi/{fname}-{lname}***".
    - 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**, 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**.

• Cấu trúc route template

• 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 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**)

• Cấu trúc route template

• Giá trị mặc định của parameter

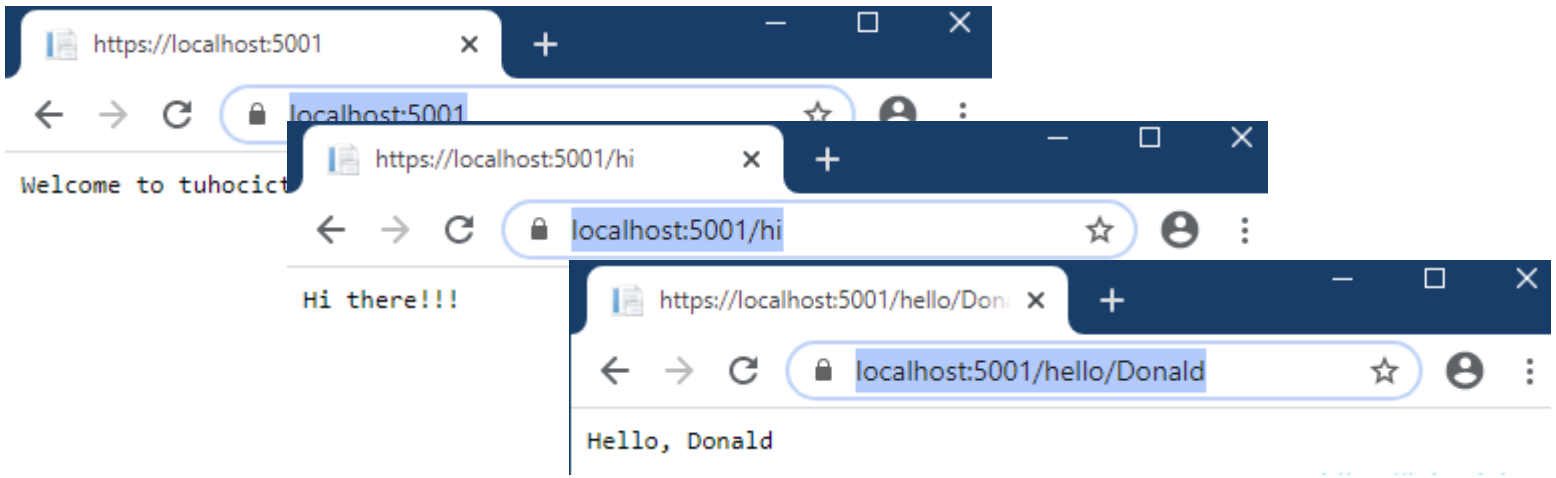
- **Paramter** có thể nhận giá trị mặc định bằng cách thêm =giá-trị vào sau tên **paramter**. Trong **route** mặc định ***"{controller=Home}/{action=Index}/{id?}"***, **controller** có giá trị mặc định là **Home**, và **action** có giá trị mặc định là **Index**.
- Sử dụng giá trị mặc định cho phép bỏ 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**

• Attribute routing trong ASP.NET Core MVC

- Hãy xây dựng **GreetingController** với các action như sau**:**

```
1.  using Microsoft.AspNetCore.Mvc;
2.
3.  namespace WebApplication1.Controllers {
4.      public class GreetingController {
5.          [Route("")]
6.          public string Greeting() => $"Welcome to TuHocIct";
7.
8.          [Route("hi")]
9.          public string Hi() => $"Hi there!!!";
10.
11.         [Route("hello/{name}")]
12.         public string Hello(string name) => $"Hello, {name}";
13.     }
14. }
```

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



- **Attribute routing trong ASP.NET Core MVC**
  - 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**.
  - Qua đây bạn có thể thấy, **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.

- **Truyền dữ liệu cho controller action**
  - Hãy tạo một dự án rỗng và cấu hình sử dụng **Mvc Middleware** với **Startup** class sau:

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.DependencyInjection;

namespace WebApplication1 {
    public class Startup {
        public void ConfigureServices(IServiceCollection services) => services.AddControllersWithViews();

        public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {
            app.UseRouting();

            app.UseEndpoints(endpoints => {
                endpoints.MapControllerRoute("power", "power/{number?}/{power?}", new { controller = "App", action = "Power" });
                endpoints.MapControllerRoute("default", "{controller}/{action}/{id?}");
            });
        }
    }
}
```

  - Chú ý chúng ta thêm một **route template** mới: ***endpoints.MapControllerRoute("power", "power/{number?}/{power?}", new { controller = "App", action = "Power" });*** Route này chấp nhận các Url như **power/**, **power/2/4** và ánh xạ sang action **Power** của **AppController**

- **Truyền dữ liệu cho controller action**
  - Xây dựng AppController với action Power như sau:

```
1.  using System;
2.  using Microsoft.AspNetCore.Mvc;
3.
4.  namespace WebApplication1 {
5.      public class AppController : Controller {
6.          public IActionResult Power(int number, int power) {
7.              var res = Math.Pow(number, power);
8.              return Content($"{number} powered by {power} is {res}");
9.          }
10.     }
11. }
```

  - Để đơn giản, chúng ta không xây dựng **view**. Kết quả trả về chỉ là một chuỗi ký tự.
  - Giờ hãy chạy ứng dụng và thử nghiệm với các URL sau:
    1. ***/power*** (kết quả là 0 powered by 0 is 1)
    2. ***/power/2/4*** (2 powered by 4 is 16)
    3. ***/power?number=3&power=2*** (3 powered by 2 is 9)
    4. ***/power/2/4?number=3&power=2*** (2 powered by 4 is 16)

• Khái niệm model binding trong ASP.NET Core MVC

- Khi học về ***truy vấn HTTP*** ở những bài học đầu tiên về **ASP.NET Core,** đã biết rằng truy vấn **HTTP** thực sự chỉ là một chuỗi văn bản có định dạng.
- **Action** trong **ASP.NET Core MVC** là những phương thức **C#** thông thường. Như trong ví dụ trên, **action** là phương thức ***public IActionResult Power(int number, int power) { ... }***
- Điểm lưu ý ở **action** này là tham số đầu vào của nó là hai biến có kiểu ***int***. Không cần thực hiện bất kỳ thao tác gì liên quan đến truy vấn **HTTP** nhưng bạn vẫn có hai giá trị **int** để sử dụng trong **action**!
- **ASP.NET Core MVC** sử dụng một cơ chế có tên gọi là **model binding** để thực hiện công việc này. **Model binding** là cơ chế trích giá trị từ truy vấn **HTTP** và biến đổi thành các **object .NET** cung cấp cho **action**.
- **Model binding** cũng là điểm làm cho **ASP.NET Core MVC** trở nên thân thiện với lập trình viên. Không cần quan tâm về truy vấn **HTTP**. **ASP.NET Core MVC** thông qua **model binding** tự động trích dữ liệu phù hợp từ truy vấn **HTTP** để tạo **object** phù hợp cho **action**. Bạn chỉ cần tập trung vào xử lý logic của ứng dụng.

• Khái niệm model binding trong ASP.NET Core MVC

- Khi truy vấn **HTTP** tới, cơ chế **routing** sẽ xác định **action** cần thực thi. Dựa vào danh sách tham số của **action**, **model binding** tìm kiếm giá trị phù hợp trong truy vấn **HTTP**.
- Khi tìm kiếm được giá trị có tên trùng với tham số, **model binding** sẽ biến đổi giá trị (**convert**) từ **string** về kiểu của tham số.
- Sở dĩ **model binding** có thể tìm kiếm giá trị trong truy vấn là vì **HTTP** thường chứa dữ liệu ở dạng các cặp ***biến=giá trị***.
- **Ví dụ**: thường xuyên gặp lối viết như ***?id=5&category=book***. Đây là lối viết của **query string**. Trong phần thân của truy **POST** cũng chứa dữ liệu theo cách tương tự.
- Trường hợp dữ liệu nằm trong **segment** của **URL**, cơ chế **routing** sẽ làm nhiệm vụ so khớp **URL** với **route template**. Từ kết quả so khớp này sẽ rút ra giá trị tương ứng với **placeholder** (có vai trò tương tự tên biến).

- **Binding source**
  - Các vị trí có thể chứa giá trị trong một truy vấn **HTTP** được gọi là **binding source**.
  - Thông thường, trong một truy vấn **HTTP** có thể có 3 **binding source**: phần thân truy vấn **POST**, **URL**, **query string**.
  - Do dữ liệu tới có thể chứa ở nhiều **binding source**, **model binding** tìm kiếm theo thứ tự sau:
    - Dữ liệu chứa trong thân của truy vấn POST: những giá trị này được gọi là **form value** do nó được thu thập từ **HTML form**.
    - Dữ liệu chứa trong **URL**: ví dụ trong **URL** ***/product/get/3***, giá trị 3 là id của sản phẩm cần lấy. Giá trị gửi kèm **URL** như thế này có tên gọi là ***route data/value***. Để **URL** có thể mang **route data**, khi định nghĩa **route template** cần sử dụng **placeholder/route parameter**.
    - Dữ liệu chứa trong **query string** của **URL**: ví dụ trong Url ***/product/get?id=3***, id=3 là phần query string.
  - Nếu một giá trị trùng tên và nằm ở nhiều vị trí, cơ chế **model binding** sẽ lấy nó theo thứ tự ưu tiên: ***form value*** > ***route value*** > ***query string value.*** Tức là, **form value** có độ ưu tiên cao nhất, **query string value** có độ ưu tiên thấp nhất.

- **Binding với kiểu class**
  - **Model binding** hoạt động với cả ***kiểu cơ sở*** của **C#** cũng như kiểu do người dùng xây dựng (**class**, **struct**).
  - Tiếp tục với project thử nghiệm. Hãy tạo một class **Book** mới như sau:

```
1.  namespace WebApplication1 {
2.      public class Book {
3.          public int Id { get; set; }
4.          public string Title { get; set; }
5.          public string Authors { get; set; }
6.          public string Publisher { get; set; }
7.          public int Year { get; set; }
8.      }
9.  }
```

  - Có thể để file mã nguồn của class này ở đâu tùy ý. Để đơn giản, có thể đặt thẳng trong thư mục project.
  - Thêm action sau vào AppController:

```
public IActionResult CreateNewBook(Book book) {
    return Content($"{book.Title} by {book.Authors} ({book.Publisher}, {book.Year})");
}
```

  - Chạy thử với URL sau: ***/app/createnewbook?title=C%20programming&authors=Donald%20Trump&publisher=ICT&year=2020.***
  - Thu được kết quả

```
C programming by Donald Trump (ICT, 2020)
```

- **Binding với kiểu class**
  - Ví dụ trên cho thấy, cơ chế **model binding** không chỉ hoạt động với các kiểu cơ sở mà còn hoạt động với cả các kiểu class phức tạp hơn do người dùng định nghĩa.
  - Trong tình huống này ở **binding source** (ở đây là **query string**) phải sử dụng biến trùng tên với các **public property** của **class**. Trong **query string**, tên biến không phân biệt hoa/thường.
  - Khi đã xác định được **action** cần thực thi, cơ chế **binding** thử tạo **object** của class **Book** trước (vì **action** đòi hỏi tham số kiểu **Book**).
  - Khi khởi tạo xong **object**, cơ chế **binding** sẽ tìm qua tất cả các **public property** của **object**. Ứng với mỗi **public property**, nó sẽ tìm kiếm tham số có tên tương ứng trong các **binding source**. Nếu tìm thấy tham số phù hợp với **property**, nó sẽ trích giá trị tương ứng và gán vào cho **property**.
  - Quá trình tìm kiếm này tương tự như với tình huống **binding** với kiểu cơ sở.
  - Khi quá trình tìm kiếm kết thúc, **object** sẽ truyền vào cho **action** như thường lệ.

• Model binding với form

- **Form** có thể trả dữ liệu về **server** bằng truy vấn **GET** (mặc định) hoặc truy vấn **POST**.
- Trong trường hợp trả về bằng truy vấn **GET**, giá trị từ tất cả các điều khiển trên **form** mà được đặt tên (thiết lập qua **attribute *name*=…**) sẽ được tập hợp thành các cặp t***ên_điều_khiển=giá_trị*** trong **query string** của URL.
- Trong trường hợp trả về bằng truy vấn **POST**, các cặp t***ên_điều_khiển=giá_trị*** được đặt trong thân truy vấn.
- Dù là trả về theo truy vấn nào, cơ chế **model binding** đều có thể trích được dữ liệu tự động.

• Model binding với form

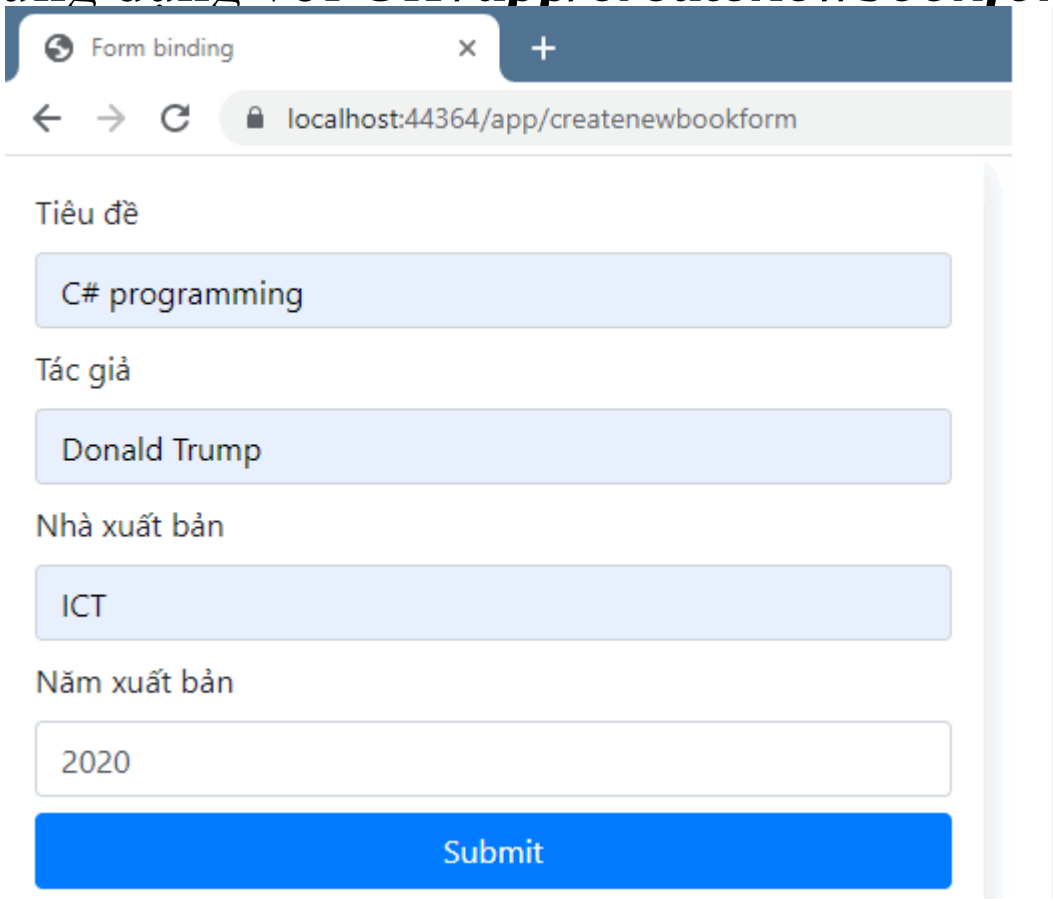
- **Ví dụ:** Tạo mới file ***Form.cshtml*** trong dự án (cho tiện lợi) và viết code như sau:

```
<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Form binding</title>
    <link href="https://stackpath.bootstrapcdn.com/bootstrap/4.5.0/css/bootstrap.min.css" rel="stylesheet" integrity="sha384-9aIt2nRpC1
</head>
<body>
    <form action="CreateNewBook" method="post" class="p-3 w-50 shadow">
        <label for="title" class="">Tiêu đề</label>
        <input name="title" type="text" class="form-control mb-2" id="title" />
        <label for="authors" class="">Tác giả</label>
        <input name="authors" type="text" class="form-control mb-2" id="authors" />
        <label for="publisher" class="">Nhà xuất bản</label>
        <input name="publisher" type="text" class="form-control mb-2" id="publisher" />
        <label for="year" class="">Năm xuất bản</label>
        <input name="year" type="number" class="form-control mb-2" id="year" />
        <input type="submit" value="Submit" class="btn btn-primary btn-block" />
    </form>
</body>
</html>
```

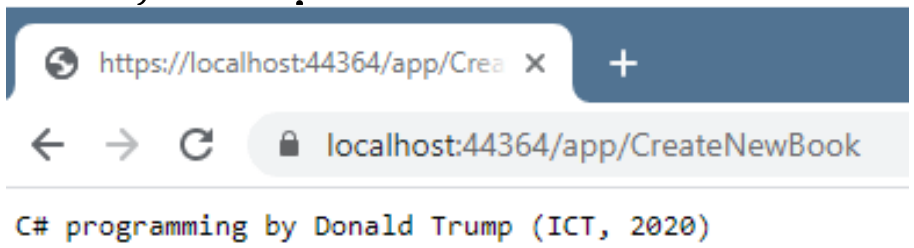
- Để ý thẻ ***<form action="CreateNewBook" method="post">***. Khi người dùng ấn nút **submit**, dữ liệu trả về cho action **CreateNewBook** mà chúng ta đã xây dựng từ ví dụ trước đó theo truy vấn **POST**.
- Các điều khiển trên **form** đều phải thiết lập attribute **name** với giá trị trùng với các **public property** của lớp **Book** (đã tạo trong ví dụ trước đó). Không có **name**, giá trị sẽ không được đóng gói để gửi về **server**.

• Model binding với form

- Chạy ứng dụng với Url ***/app/createnewbookform***



- Khi ấn **submit**, dữ liệu trả về action **CreateNewBook** xử lý như sau:



Thank you!