Buổi 01

ASP.NET Core và mô hình MVC

I. Giới thiệu về ASP.NET Core

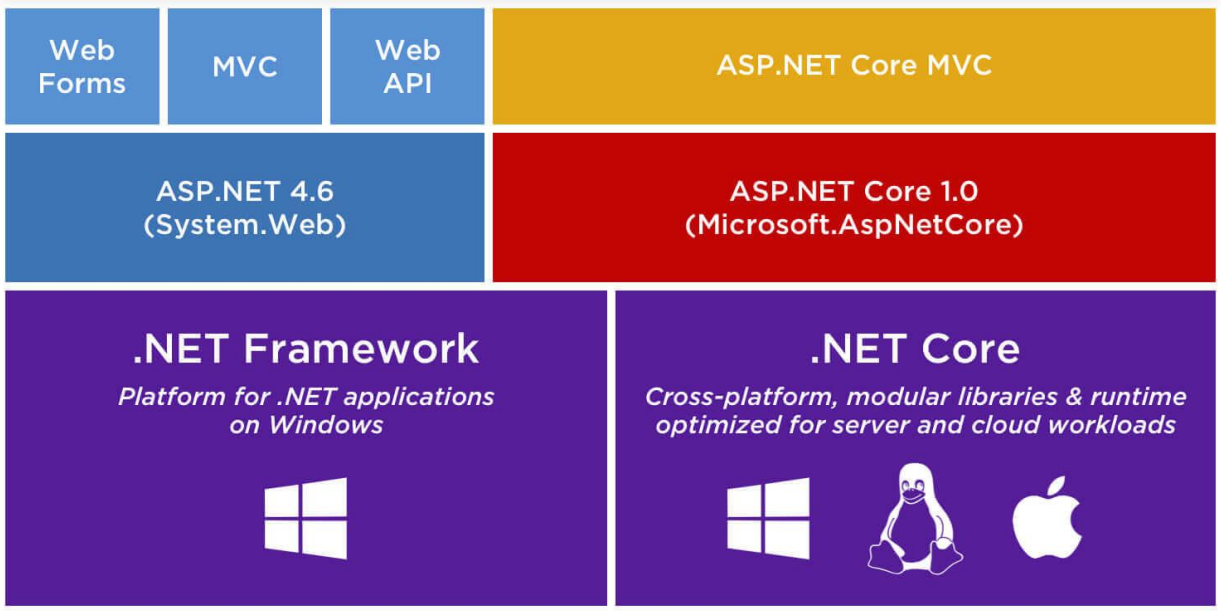
1. Tổng quan

ASP.NET Core là 1 framework mã nguồn mở (open-source) và đa nền tảng (cross-platform), dùng cho việc xây dựng các ứng dụng hiện đại và có thể tích hợp công nghệ điện toán đám mây, ví dụ các ứng dụng web, back-end cho ứng dụng mobile, IoT...

Ứng dụng ASP.NET Core có thể chạy trên phiên bản đầy đủ .NET Framework, trên Windows hoặc trên phiên bản .NET đa nền tảng.

ASP.NET Core bao gồm các thành phần theo hướng module nhằm giảm thiểu tài nguyên và chi phí phát triển nhưng vẫn giữ lại được sự mềm dẻo, linh hoạt. Những ứng dụng xây dựng bằng ASP.NET Core có thể chạy trên cả Windows, MacOS và Linux.

ASP.NET Core kế thừa từ ASP.NET nhưng có sự thay đổi lớn về kiến trúc do học hỏi từ các framework module hóa khác. ASP.NET Core không còn dựa trên System.Web.dll đồ sộ và nặng nề nữa mà dựa trên tập hợp các gói, các module được gọi là **NuGet Package**. Điều này cho phép lập trình viên tối ưu ứng dụng để chỉ bao gồm những thành phần cần thiết, giúp cho ứng dụng nhỏ gọn hơn, bảo mật chặt chẽ hơn, giảm sự phức tạp, tối ưu hiệu suất và giảm chi phí, thời gian cho việc phát triển.

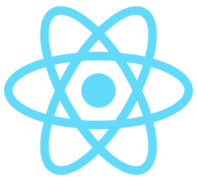


ASP.NET Core cũng được thiết kế để có thể tích hợp nhiều framework front-end, ví dụ:


jQuery


Bootstrap


AngularJS


React

2. Lịch sử phát triển

Phiên bản đầu tiên của ASP.NET Core phát hành vào năm 2016. Tại thời điểm đó, Microsoft đã dự định gọi tên framework này là ASP.NET 5. Tuy nhiên, để tránh nhầm lẫn với .NET Framework sẵn có, Microsoft đã quyết định sử dụng tên ASP.NET Core.

Kể từ phiên bản 5.0 trở đi, Microsoft đã loại bỏ từ "Core" khỏi tên framework, và chỉ đơn giản là .NET 5 (thay vì .NET Core 5.0).

Phiên bản	Ngày phát hành	Phiên bản Visual Studio hỗ trợ
.NET Core 1.0	27/06/2016	Visual Studio 2015 và 2017
.NET Core 1.1	18/11/2016	
.NET Core 2.0	14/08/2017	Visual Studio 2017
.NET Core 2.1	30/05/2018	
.NET Core 2.2	04/12/2018	Visual Studio 2017 v15.9 và 2019 v16.0 Preview 1
.NET Core 3.0	23/09/2019	Visual Studio 2017 và 2019
.NET Core 3.1	03/12/2019	Visual Studio 2019
.NET 5.0	10/11/2020	Visual Studio 2019 v16.8
.NET 6.0	08/11/2021	Visual Studio 2022
.NET 7.0	08/11/2022	Visual Studio 2022 v17.4
.NET 8.0	14/11/2023	Visual Studio 2022 v17.8

3. Các tính chất quan trọng

ASP.NET Core có các đặc điểm, tính năng quan trọng sau:

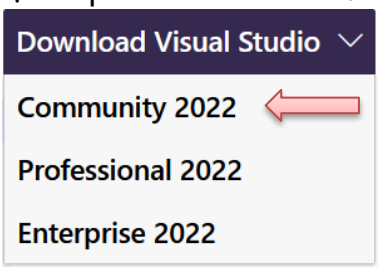
- Mã nguồn mở và được cộng đồng chung tay phát triển.
- Có thể xây dựng và chạy ứng dụng đa nền tảng trên Windows, MacOS và Linux.
- Module hóa bằng cách chỉ cung cấp phần cốt lõi (core). Lập trình viên có thể cài đặt thêm tính năng thông qua các gói bổ sung NuGet Package.
- Tối ưu cho việc xây dựng ứng dụng đám mây (cloud-based apps).
- Hợp nhất ASP.NET MVC và ASP.NET Web API thành mô hình lập trình duy nhất.

- Quá trình biên dịch diễn ra liên tục nhờ tính năng **Hot Reload**. Lập trình viên không cần phải biên dịch thủ công chương trình sau khi chỉnh sửa mã nguồn.
- Có thể lưu trữ trên IIS hoặc tại hosting tùy ý.
- Có sẵn Dependency Injection.
- Hỗ trợ cấu hình cho nhiều môi trường, nhiều phiên bản .NET trên cùng 1 máy.
- Cơ chế HTTP Request pipeline mới, gọn nhẹ hơn.

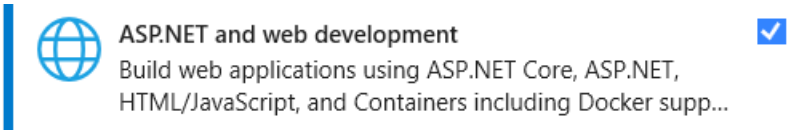
II. Công cụ và môi trường làm việc

Có thể dùng nhiều IDE[1] để phát triển ứng dụng .NET Core, ví dụ như Visual Studio, Visual Studio Code, hay thậm chí là Terminal[2] kết hợp với 1 text editor bất kỳ. Trong môn học này, **khuyến cáo** dùng **Visual Studio** phiên bản mới nhất (hiện tại là phiên bản 2022).

Địa chỉ tải Visual Studio (VS) chính thức từ Microsoft: https://visualstudio.microsoft.com/vs. Học viên nên chọn tải và cài đặt phiên bản **Community**, vì phiên bản này miễn phí cho mục đích sử dụng cá nhân và giáo dục.



Để có thể phát triển ứng dụng với ASP.NET Core, khi cài đặt VS, cần chọn workload **ASP.NET and web development**:



Ứng dụng ASP.NET Core có thể được xây dựng theo nhiều mô hình như:

- Razor Pages
- MVC
- Blazor

Trong môn học này, chúng ta sẽ xây dựng ứng dụng ASP.NET Core trên phiên bản **.NET 6.0** bằng mô hình **MVC** theo 2 hướng là **Web App** và **Web API**, kết hợp với việc tạo CSDL bằng Entity Framework và Code-First Migration[3].

Cách xây dựng bằng mô hình Razor Pages và Blazor được trình bày lần lượt trong Phụ lục 2 và Phụ lục 3.

---

[1] Integrated Development Environment – Môi trường phát triển tích hợp

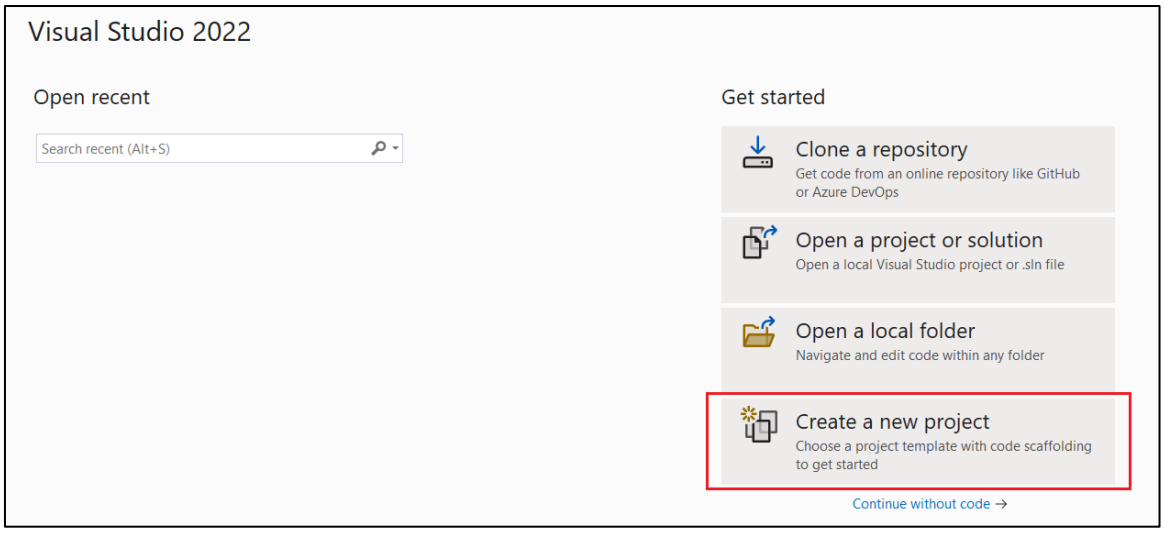
[2] Xem Phụ lục 1 – Xây dựng ứng dụng ASP.NET Core bằng Terminal

[3] Entity Framework và Code-First sẽ được trình bày ở buổi 3

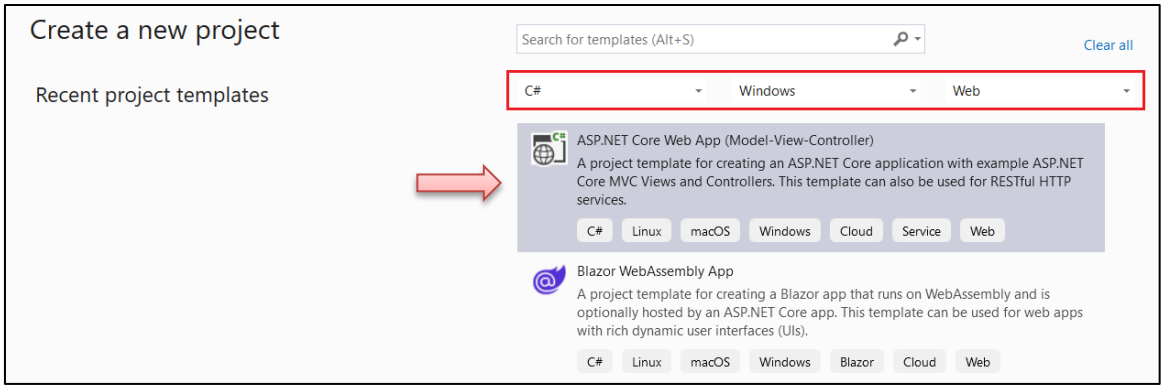
III. Tạo project ASP.NET Core MVC

Các bước để tạo 1 project ASP.NET Core MVC:

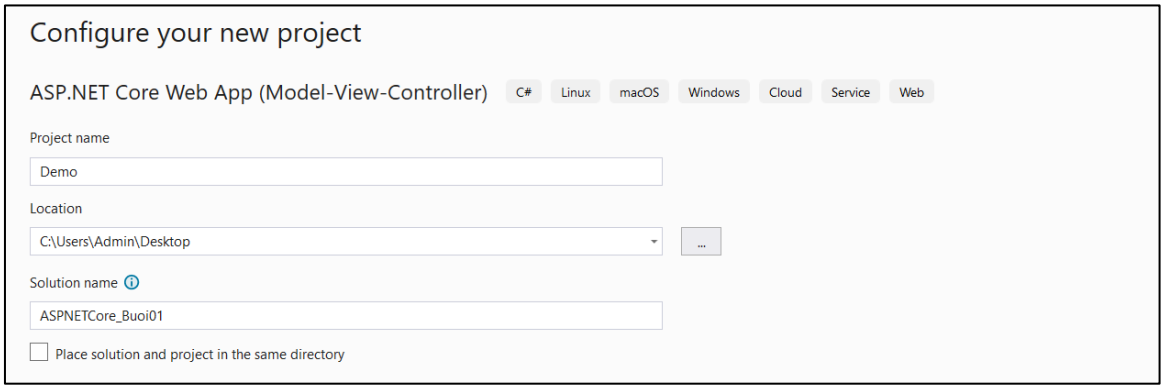
– Trong giao diện VS 2022, chọn **Create a new project**:



– Chọn loại project là **ASP.NET Core Web App (Model-View-Controller)**[1], sau đó nhấn **Next**. Có thể tìm nhanh bằng cách điền tên loại project vào khung tìm kiếm hoặc chọn ngôn ngữ, nền tảng từ filter như sau:

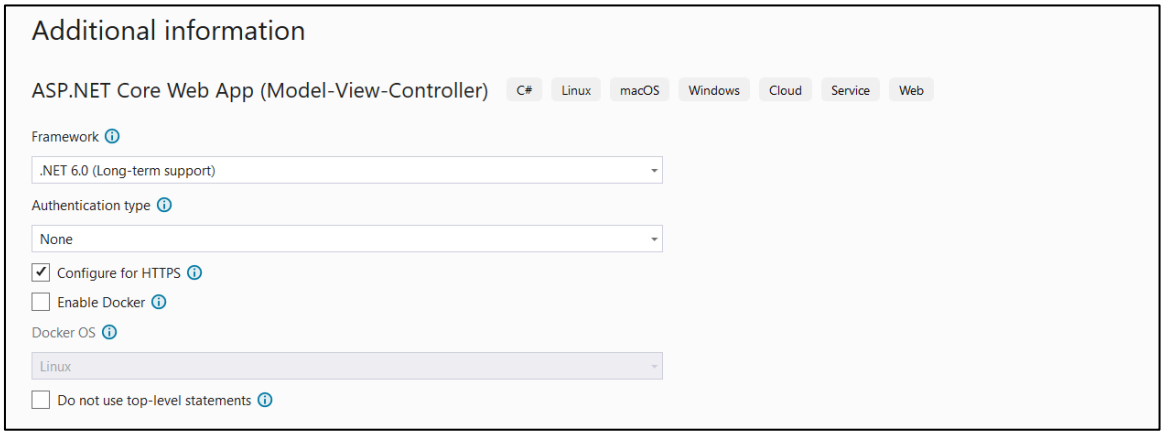


– Đặt tên cho project và solution (ví dụ như hình dưới: tên project là **Demo**, tên solution là **ASPNETCore_Buoi01**), chọn nơi lưu project, sau đó nhấn nút **Next**:

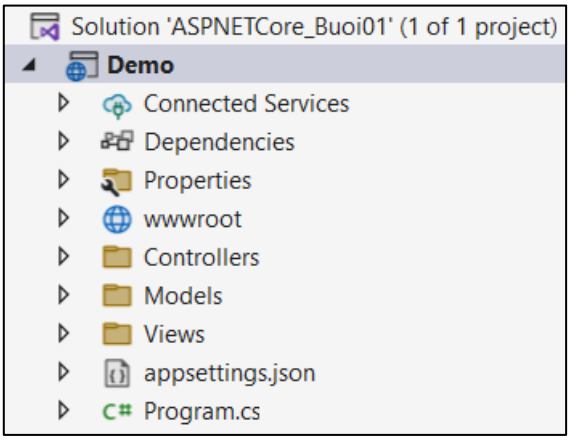


[1] **Lưu ý:** Tránh nhầm với project loại ASP.NET Web App (không có Model-View-Controller). Loại project này được viết theo mô hình Razor Pages.

– Chọn phiên bản .NET (mặc định ở VS 2022 là .NET 6.0). Các tùy chọn khác giữ nguyên mặc định như hình dưới, sau đó nhấn nút **Create**:



Cấu trúc 1 project ASP.NET Core MVC như sau:



– Thư mục **wwwroot** là thư mục gốc của website. Tất cả thành phần tĩnh như CSS, JavaScript, các tài nguyên (asset, resource) như hình ảnh, video, âm thanh… đều phải đặt trong thư mục này[1]. Project ASP.NET Core MVC đã có sẵn jQuery và Bootstrap nằm trong thư mục wwwroot/lib.

– Các thư mục **Controllers**, **Models**, **Views** để chứa các thành phần tương ứng của mô hình MVC. Mô hình này sẽ được giải thích ở phần IV.

– File **appsettings.json** chứa các cài đặt, thiết lập cho project, ví dụ như các giá trị toàn cục, chuỗi kết nối đến CSDL… Dữ liệu của file này được lưu trữ theo dạng JSON (tức là theo từng cặp key-value):

```
{
    "key1": "value1",
    "key2": "value2"
}
```

– File **Program.cs** là nơi chứa những đoạn lệnh để đăng ký Middleware[2], thiết lập các dịch vụ (Service), khởi chạy ứng dụng… Ở các phiên bản .NET 5.0 trở về trước, những công việc này nằm cả 2 file Program.cs và Startup.cs. Kể từ phiên bản .NET 6.0, Microsoft đã hợp nhất 2 file này lại thành 1 file Program.cs duy nhất.

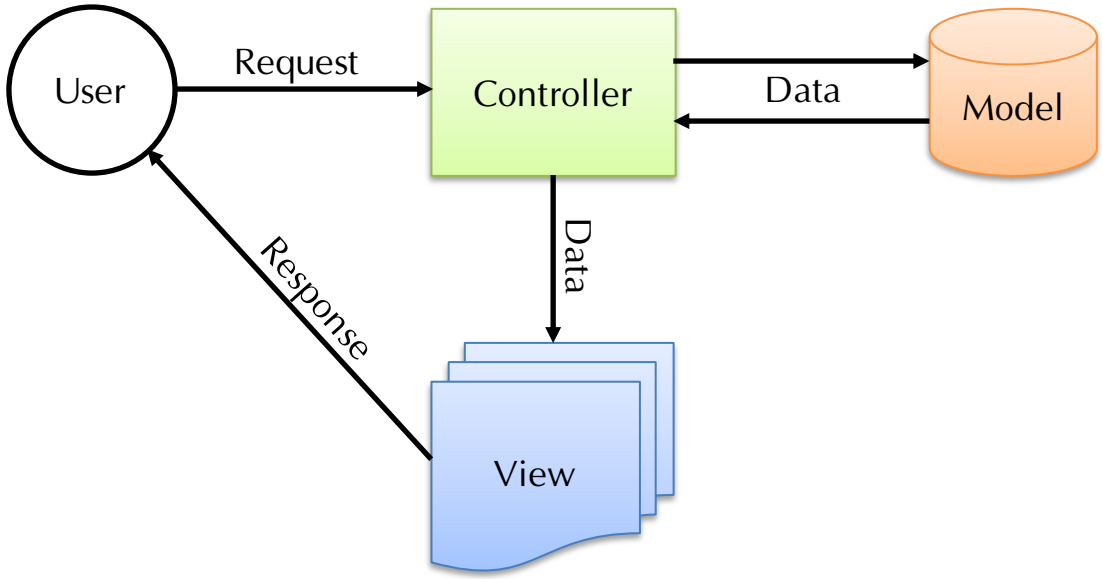
IV. Mô hình MVC

[1] Theo quy chuẩn của web, cần phải chia thư mục riêng phân loại cho các file này, ví dụ thư mục css, js, assets, images…

[2] Xem Phụ lục 4 – Middleware và Dependency Injection trong ASP.NET Core

1. Mô hình MVC trong ASP.NET Core

Mô hình MVC là 1 mô hình trong xây dựng phần mềm, ứng dụng, đặc biệt là với các ứng dụng web. Mô hình này chia ứng dụng thành 3 lớp logic như sau:



- **Model**: Chịu trách nhiệm xử lý logic cho dữ liệu: giao tiếp với CSDL, lưu trữ, truy vấn dữ liệu...
- **View**: Giao diện của ứng dụng. Dữ liệu sau khi được Controller xử lí sẽ hiển thị lên View để người dùng có thể xem được.
- **Controller**: Đóng vai trò trung gian giữa Model và View, quản lý và điều phối hoạt động của ứng dụng.

Cơ chế hoạt động của 1 ứng dụng web ASP.NET Core MVC như sau:

- Ứng dụng nhận **request** từ người dùng dưới dạng **route** và xử lý, chuyển yêu cầu này đến Controller.
- Controller xử lí yêu cầu. Nếu cần đến dữ liệu thì Controller sẽ gọi Model thực hiện.
- Dữ liệu từ Model sau khi đã được Controller xử lý sẽ được chuyển sang View.
- View hiển thị giao diện kèm theo dữ liệu cho người dùng.
- Kết quả hiển thị từ View sẽ được biên dịch sang HTML, CSS, JavaScript và gửi trả về cho trình duyệt web theo **response**.

Chi tiết về cơ chế hoạt động ở trên sẽ được trình bày cụ thể trong phần VIII.

2. Routing trong mô hình MVC

Về mặt bản chất, URL sẽ trỏ đến 1 tài nguyên trong mạng, cụ thể là 1 file (như file HTML, file PHP, file ASPX...), ví dụ:

http://www.someserver.com/index.html
http://www.otherserver.vn/data/register.php

Với cách này, cấu trúc thư mục, tên file... trên server sẽ bị lộ ra cho người dùng, và có thể bị lợi dụng cho mục đích xấu. Do đó, routing ra đời để khắc phục nhược điểm này.

ASP.NET Core MVC sử dụng middleware **Routing**[1] (định tuyến) để tiếp nhận các request từ người dùng thông qua giao thức HTTP và gửi chúng đến nơi xử lý request tương ứng.

Có 2 loại routing: Web App thường sử dụng **Conventional Routing**, còn Web API thường sử dụng **Attribute Routing**.

a) Convention Routing – Routing chuẩn

Convention Routing được định nghĩa trong file Program.cs của ứng dụng:

```
app.MapControllerRoute(
      name: "default",
      pattern: "{controller=Home}/{action=Index}/{id?}");
```

Đây là cấu trúc của 1 route mặc định của ASP.NET Core MVC. Route này quy định:

- Tên của route là **default**.
- Cú pháp route có dạng /tên_controller/tên_action/id_nếu_có
- Tên controller mặc định là **Home**, tức là nếu route không quy định controller, thì sẽ trỏ đến controller Home.
- Tên action mặc định là **Index**, tức là nếu route không quy định action, thì sẽ trỏ đến action Index.
- Tham số mặc định của URL là **id**, nghĩa là thay vì phải viết tham số theo dạng something?id=5 thì có thể viết theo dạng something/5.
- Dấu ? của tham số id cho biết tham số này không bắt buộc.

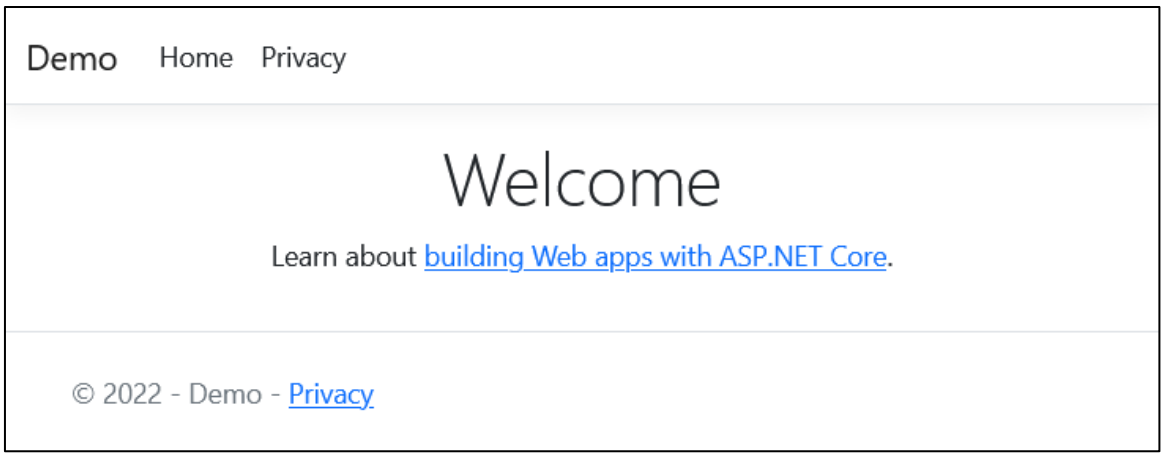
Ví dụ: Trong các ví dụ sau: http://www.eshop.com chính là domain của trang web hiện tại, và phần còn lại là route:

URL	Controller	Action	id
http://www.eshop.com/	Home	Index	-
http://www.eshop.com/**Home**			
http://www.eshop.com/**Home/Index**			
http://www.eshop.com/**Accounts**	Accounts	Index	-
http://www.eshop.com/**Accounts/Login**	Accounts	Login	-
http://www.eshop.com/**Products/Details/2**	Products	Details	2

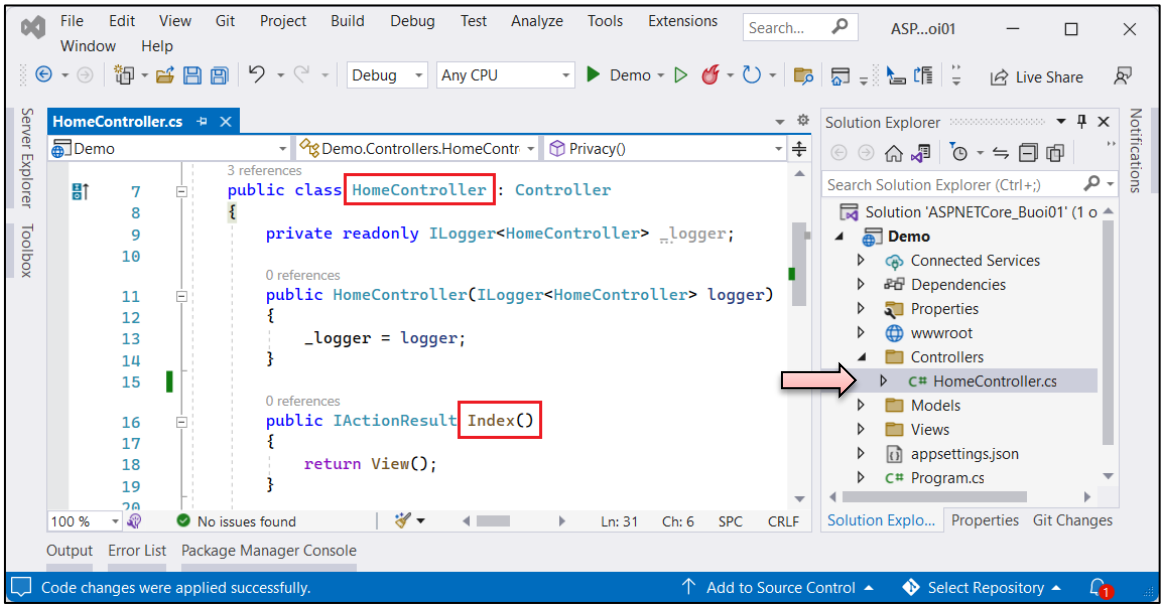
[1] Xem thêm về ASP.NET Core MVC Routing: https://docs.microsoft.com/en-us/aspnet/core/mvc/controllers/routing

Trong cùng 1 ứng dụng có thể định nghĩa nhiều Convention Routing, và route được coi là hợp lệ khi và chỉ khi tìm được controller và action đã có trong ứng dụng.

Khi biên dịch và chạy project **ASPNETCore_Buoi01/Demo** hiện tại, trình duyệt web sẽ hiển thị 1 trang web với URL là https://localhost:X, trong đó X là 1 port bất kỳ do server localhost tạo ra.



Đối chiếu với cú pháp route mặc định ở trên, URL này đang có route /Home/Index, tức là trỏ đến controller **Home** và action **Index**:



b) Attribute Routing – Routing theo thuộc tính

Thay vì định nghĩa trong file Program.cs, Attribute Routing sẽ định nghĩa route ngay tại controller và/hoặc action bằng chú thích [Route]. Cách này sẽ cho phép người dùng tùy biến linh hoạt hơn trong việc quy định route cho từng controller và từng action, thậm chí có thể đặt tên tùy chọn cho từng route.

Ví dụ: Tất cả ví dụ phía dưới đều minh họa trên controller **Home**:

- Quy định route đến action **Index** là /TrangChu và đặt tên route là **demo**:

```
[Route("TrangChu", Name = "demo")]
public IActionResult Index() { return View(); }
```

- Quy định route đến action **Index** là / hoặc là /TrangChu:

```
[Route("/")]
[Route("TrangChu")]
public IActionResult Index() { return View(); }
```

- Quy định route đến action **Index** là /TrangChu, kèm theo tham số mặc định là **p** (không bắt buộc), tức là /TrangChu/5 hoặc /TrangChu?p=5

```
[Route("TrangChu/{p?}")]
public IActionResult Index(int p) { return View(); }
```

- Quy định route đến action **Index** là /TrangChu kèm theo tham số mặc định là **p**, có giá trị mặc định là 10:

```
[Route("TrangChu/{p=10}")]
public IActionResult Index(int p) { return View(); }
```

- Quy định route đến action **Index** là /TrangChu/Index và route đến action **Privacy** là /TrangChu/DieuKhoan. Do 2 route này có cùng tiền tố là /TrangChu, nên thay vì định nghĩa theo cách thông thường ở từng action, chúng ta có thể định nghĩa phần tiền tố chung cho cả controller như sau:

```
[Route("TrangChu")]
public class HomeController : Controller
{
      [Route("Index")]
      public IActionResult Index() { return View(); }

      [Route("DieuKhoan")]
      public IActionResult Privacy() { return View(); }
}
```

- Quy định route đến action **Index** là /TrangChu/Index và route đến action **Privacy** là /DieuKhoan. Route này thì tương tự ví dụ ở trên, nhưng tại action có thể dùng ký tự ~ để định nghĩa lại route khác và không dùng tiền tố chung của controller.

```
[Route("TrangChu")]
public class HomeController : Controller
{
      [Route("Index")]
      public IActionResult Index() { return View(); }

      [Route("~/DieuKhoan")]
      public IActionResult Privacy() { return View();
      }
}
```

- Quy định route tương tự như Conventional Routing mặc định

```
[Route("{controller=Home}/{action=Index}/{id?}")]
public class HomeController : Controller { … }
```

V. Controller và View

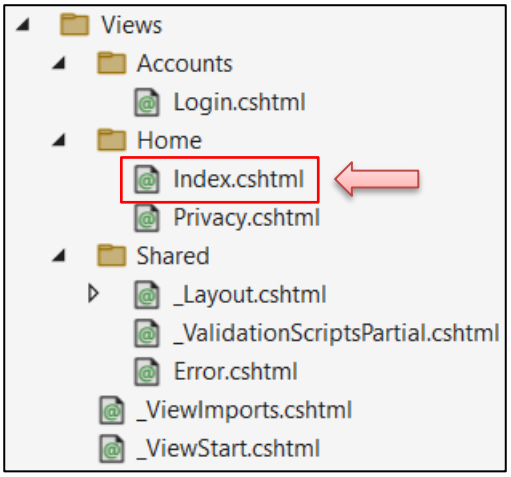
Với project **Demo** hiện tại, khi biên dịch project, route mặc định được gọi (/Home/Index), tương ứng với action **Index** của controller **Home**. Project này cũng chưa thao tác đến **Model**.

Đối với ASP.NET Core, **Controller** bản chất là 1 class, và **Action** là các phương thức của class đó. Phương thức `Index()` của class `HomeController` như sau:

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

Trong đó:

- Return type của phương thức này là `IActionResult`. Đây là 1 interface đại diện cho tất cả những kết quả mà 1 action có thể trả về cho client bằng response, ví dụ như trả về 1 view, trả về 1 yêu cầu chuyển hướng trang web, trả về 1 mã lỗi…
- Phương thức `View()` sẽ gọi view tương ứng với action hiện tại. Trong trường hợp này là Views/Home/Index.cshtml.



Tất cả các view nằm trong thư mục **Views**, trong đó gồm nhiều thư mục con tương ứng với các controller.

Ví dụ: Thư mục Home sẽ chứa các view tương ứng của controller **Home**.

Thư mục **Shared** là dùng chung cho ứng dụng, các view trong thư mục này có thể được gọi ở bất kỳ controller nào.

View được viết bằng HTML và C#, do đó phần mở rộng của nó là **.cshtml**.

Khi phương thức `View()` được gọi, đầu tiên ứng dụng sẽ tìm kiếm view cùng tên với action hiện tại trong thư mục của controller đó, sau đó là thư mục Shared.

Để gọi 1 view khác tên với action hiện tại:

- Nếu view cần gọi nằm chung trong thư mục của controller hiện tại, ví dụ action **Index** muốn gọi view Privacy.cshtml, ta viết câu lệnh như sau:

```
return View("Privacy"); // Lưu ý: không ghi phần mở rộng .cshtml
```

- Nếu view cần gọi nằm khác thư mục với controller hiện tại, ví dụ action **Index** muốn gọi view Login.cshtml thuộc thư mục Accounts, ta viết câu lệnh như sau:

```
return View("../Accounts/Login");
// HOẶC
return View("~/Views/Accounts/Login.cshtml");
```

Một số file khác trong thư mục **Views**:

- Shared/_Layout.cshtml: Dùng để tạo bố cục, giao diện chung cho ứng dụng.
- _ViewImports.cshtml: Dùng để khai báo **directive** chung cho tất cả các view (thay vì phải khai báo thủ công ở từng view).
- _ViewStart.cshtml: Dùng để cài đặt những câu lệnh sẽ được thực hiện trước khi mỗi view được chạy, ví dụ như cài đặt layout chung cho tất cả view.

Cả 3 file kể trên và khái niệm **directive** sẽ được trình bày cụ thể hơn ở buổi 2.

VI. ASP.NET Razor

Razor là 1 cú pháp lập trình ASP.NET được dùng để nhúng C# hoặc VB.NET trực tiếp vào mã nguồn HTML của trang web để tạo thành View hoặc Razor Pages, tương tự như PHP với thẻ `<?php … ?>` hoặc ASP.NET WebForm với thẻ `<% … %>`.

Razor được phát triển từ tháng 06/2010, phát hành tháng 01/2011 và hiện tại đã trở thành một phần của ASP.NET Core.

Razor sử dụng ký tự `@` để bắt đầu 1 đoạn code C#. Cú pháp của Razor như sau:

```
@{
    // Các câu lệnh theo cú pháp C#
}
```

Với các biểu thức hoặc giá trị biến, có thể lược bỏ cặp ngoặc `{ }` và dấu `;` cuối câu lệnh, ví dụ `@DateTime.Today` hoặc `@username` (với `username` là 1 biến C# đã khai báo).

Các trang web được viết theo cú pháp Razor có phần mở rộng là .cshtml, và sẽ được **Razor View Engine** biên dịch thành HTML trước khi gửi trả về trình duyệt web bằng response. Mặc định các View trong ASP.NET Core đều được viết theo cú pháp Razor.

Ví dụ: Thực hiện các bước sau:

- Với project **Demo** hiện tại, mở file Views/Home/Index.cshtml. Mã nguồn hiện tại như sau:

```
@{
    ViewData["Title"] = "Home Page";
}
<div class="text-center">
    <h1 class="display-4">Welcome</h1>
    <p>Learn about <a href="…">building Web apps …</a>.</p>
</div>
```

- Sau thẻ `<p>…</p>` ở trên, lần lượt bổ sung thêm các câu lệnh sau:
  + Khai báo 1 biến:

```
@{
    string name = "Ricky";
```

```
}
```

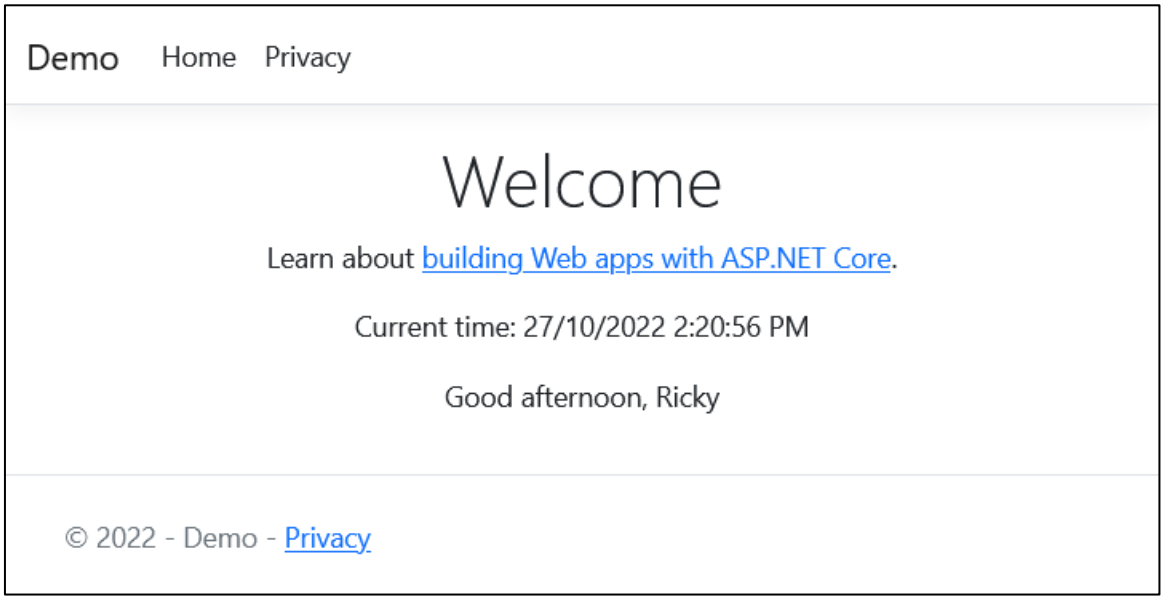
+ Xuất giá trị 1 biểu thức bằng thẻ HTML:

```
<p>Current time: @DateTime.Now</p>
```

+ Đoạn lệnh phức tạp:

```
@{
    if (DateTime.Now.Hour <= 12)
    {
        <p>Good morning, @name</p>
    }
    else
    {
        <p>Good afternoon, @name</p>
    }
}
```

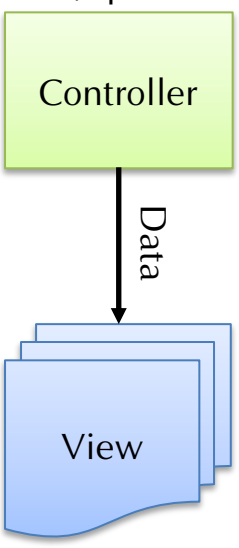
– Kết quả sau khi biên dịch project:



Lý thuyết về Razor sẽ được trình bày kỹ hơn ở buổi 2.

Lưu ý: Mặc dù ở VS 2022 và .NET 6.0, Microsoft đã không còn tô màu nền xám cho phần code C# của Razor như ở VS 2019 và .NET 5.0. Tuy nhiên, để dễ phân biệt phần mã nguồn HTML và mã nguồn C#, các ví dụ trong tài liệu môn học này vẫn sẽ tô màu nền xám cho phần code C# trong Razor.

VII. ViewData và ViewBag



Trong mô hình MVC, dữ liệu sau khi đã được xử lý, tính toán ở controller sẽ được chuyển sang **View** để hiển thị (xem ảnh bên).

ViewData và **ViewBag** được giới thiệu lần lượt ở MVC 1.0 và MVC 3.0, dùng để truyền dữ liệu từ controller sang view. Ở controller, chúng ta có thể gán giá trị cho ViewData/ViewBag, sau đó gọi lại giá trị này ở view.

Cả ViewData và ViewBag đều lưu dữ liệu theo kiểu **Dictionary**[1]. Tuy nhiên, chúng ta thao tác với ViewData thông qua từng cặp key-value, còn thao tác với ViewBag thông qua thuộc tính như 1 dynamic object.

Xét action **Index** của controller **Home**: action này đang gọi view Home/Index.cshtml:

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

Giả sử view **Index** nói trên cần hiển thị thông tin nhiệt độ và độ ẩm không khí. Trong 1 ứng dụng MVC thì thường những thông tin dạng này sẽ được tính toán ở controller, và giả sử đã tính ra nhiệt độ là 28°C và độ ẩm là 89%. Ta cần đưa 2 giá trị này sang view **Index** để hiển thị. Khi đó, chúng ta gán 2 giá trị này cho **ViewData** và/hoặc **ViewBag** như sau:

```
public IActionResult Index()
{
      ViewData["temperature"] = 28;        // Dùng ViewData
      ViewBag.humidity = 89;               // Dùng ViewBag
      return View();
}
```

Trong đó: `"temperature"` là key để truy cập ViewData, còn `humidity` là tên thuộc tính động (dynamic property) để truy cập ViewBag. Cả 2 đều do lập trình viên tự đặt.

Đối với view **Index**, chúng ta bổ sung các câu lệnh sau để gọi giá trị từ ViewData và ViewBag:

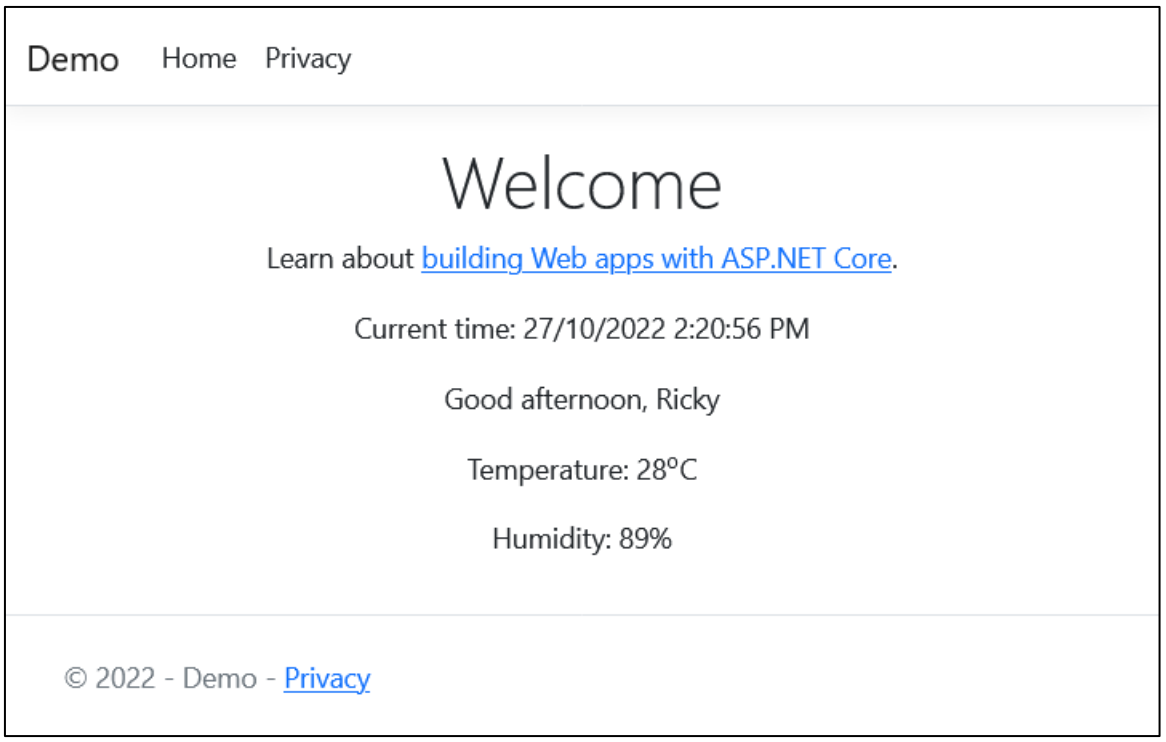
```
<p>Temperature: @ViewData["temperature"]<sup>o</sup>C</p>
<p>Humidity: @ViewBag.humidity%</p>
```

Do cả ViewData và ViewBag đều lưu dữ liệu chung vào 1 Dictionary, do đó có thể gán dữ liệu bằng ViewData nhưng lại lấy dữ liệu bằng ViewBag và ngược lại. Tại view **Index**, nếu thay thế 2 câu lệnh ở trên bằng 2 câu lệnh sau (đổi chỗ ViewBag và ViewData) vẫn sẽ cho kết quả như cũ:

```
<p>Temperature: @ViewBag.temperature<sup>o</sup>C</p>
<p>Humidity: @ViewData["humidity"]%</p>
```

[1] Xem thêm về Dictionary của C#: https://www.tutorialsteacher.com/csharp/csharp-dictionary

Kết quả sau khi biên dịch project:



So sánh ViewData và ViewBag:

	ViewData	**ViewBag**
Phiên bản ra mắt	MVC 1.0	MVC 3.0
Yêu cầu	.NET Framework 3.5	.NET Framework 4.0
Kiểu	Dictionary	Dynamic Object
Cách truy cập	Thông qua key-value	Thông qua dynamic property
Tốc độ thực thi	Nhanh hơn	Chậm hơn
Cần chuyển đổi kiểu dữ liệu (Type conversion)	Có, nếu dữ liệu lưu với cấu trúc phức tạp.	Không

VIII. Tạo 1 ứng dụng web MVC đơn giản

Thông thường, mô hình MVC dùng cho những ứng dụng lớn, có tương tác với CSDL. Tuy nhiên, trong tài liệu buổi 1 này, chúng ta sẽ không tương tác với CSDL mà chỉ minh họa trên dữ liệu có sẵn. Việc tương tác với CSDL sẽ được trình bày kể từ buổi 3.

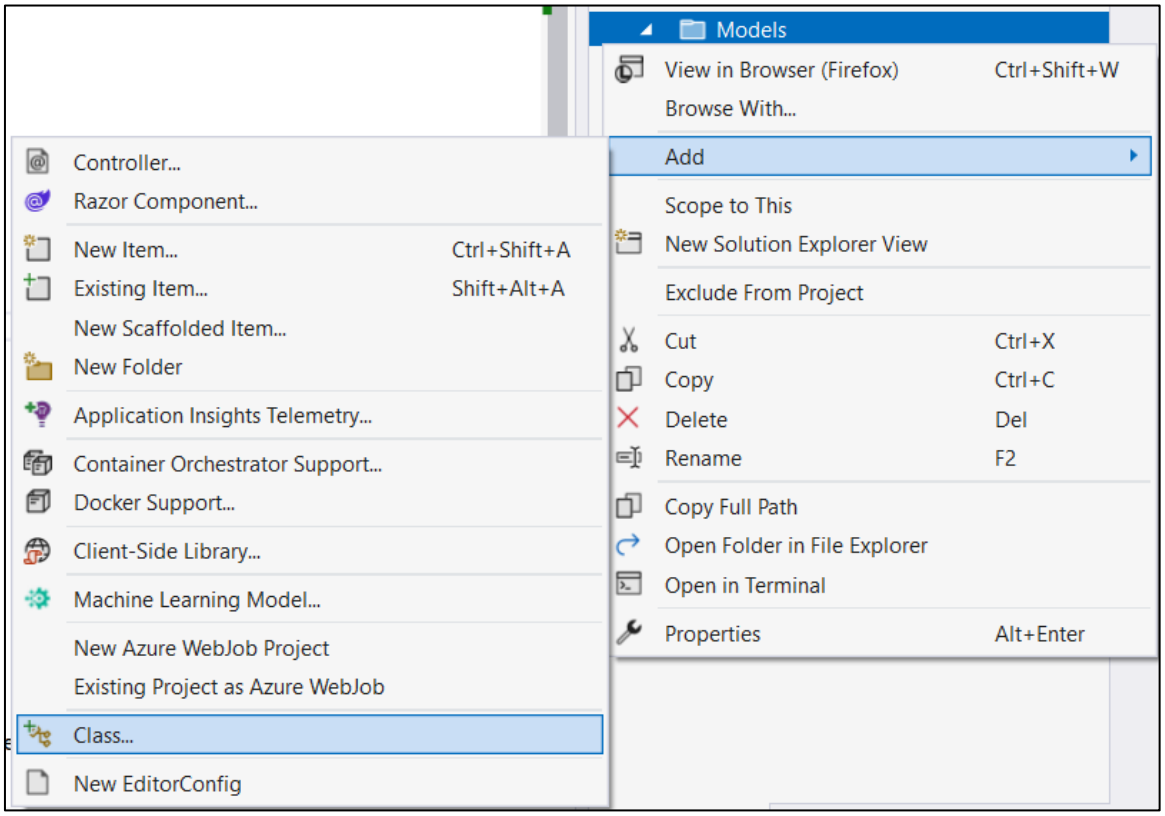
Trong ví dụ sau đây, chúng ta sẽ tạo 1 ứng dụng web hiển thị danh sách các sinh viên trong 1 lớp học thông qua route /Students/ListAll. Thông tin sinh viên gồm có ID (Mã SV), Name (Họ tên) và GPA (Điểm trung bình). Toàn bộ dữ liệu của sinh viên sẽ được tạo ra bằng code C#.

1. Tạo Model

Trong mô hình MVC, model dùng để định nghĩa các class mô phỏng lại các đối tượng cần thao tác trong ứng dụng (thường là các bảng trong CSDL). Các model này sẽ lưu trữ dữ liệu từ các bảng dưới dạng **object**.

Đối với ví dụ này, chúng ta cần tạo 1 class mô phỏng 1 sinh viên.

Để tạo model, click chuột phải vào thư mục **Models** và chọn **Add → Class...**:



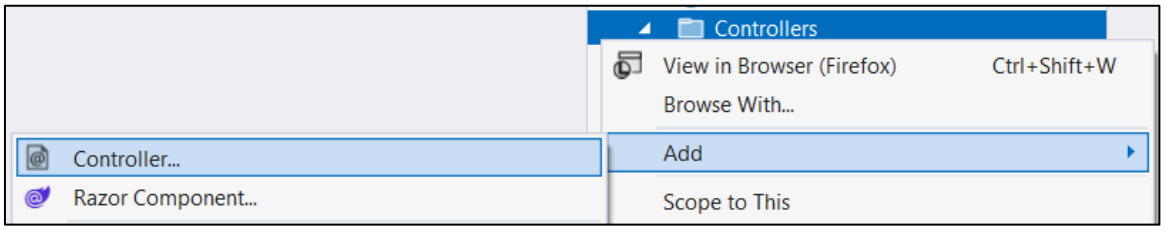
Đặt tên class là `Student` và nhấn nút **Add**. Cài đặt class này như sau:

```
public class Student
{
      public int Id { get; set; }
      public string Name { get; set; }
      public double GPA { get; set; }
}
```

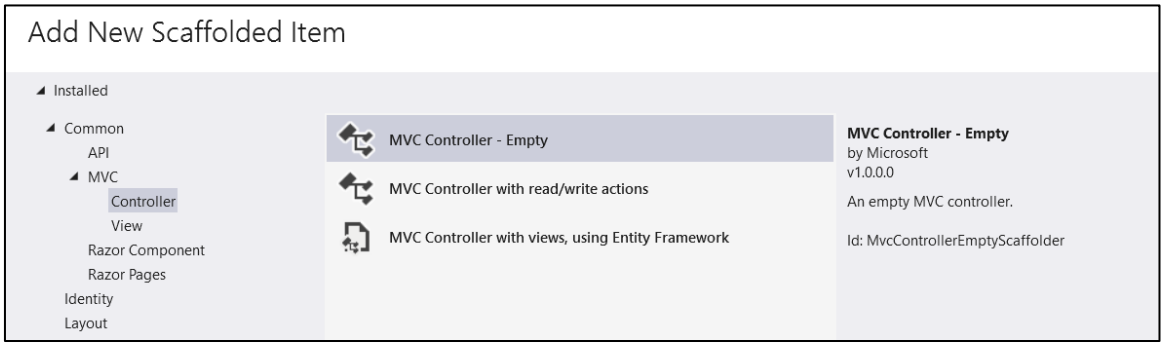
Lưu ý: Theo quy chuẩn đặt tên (Naming Convention) của C#, tên class là danh từ số ít.

2. Tạo Controller

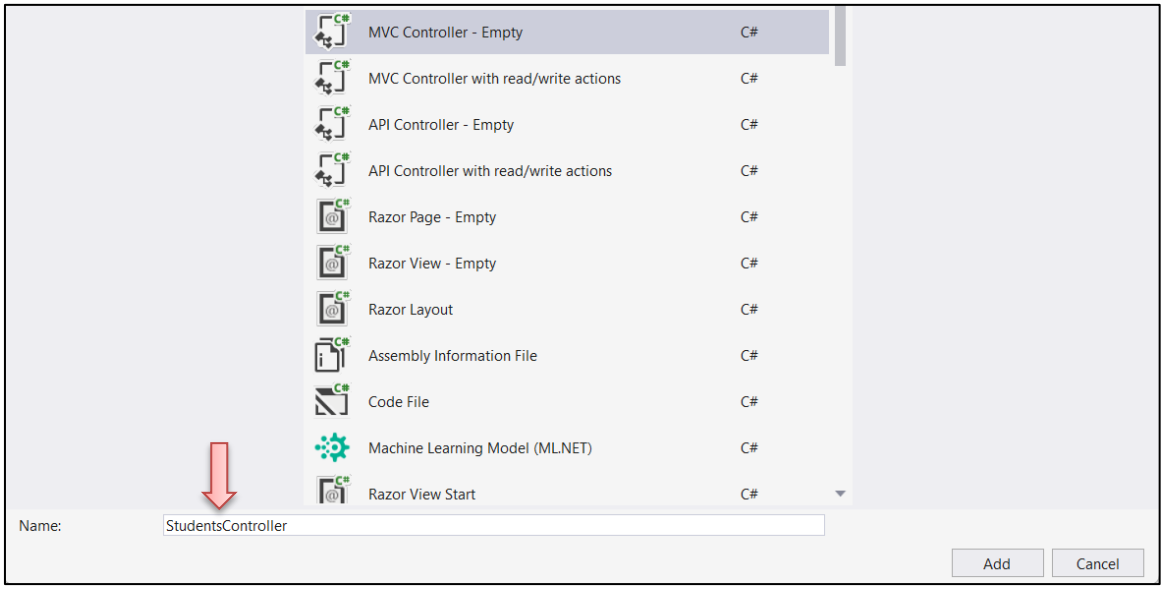
Để tạo controller, click chuột phải vào thư mục **Controllers** và chọn **Add → Controller...**:



Hộp thoại **Add New Scaffolded Item** liệt kê các loại controller. Chọn **MVC Controller – Empty** để tự viết mã nguồn, sau đó nhấn nút **Add**:



Hộp thoại **Add New Item** đã chọn sẵn **MVC Controller – Empty**. Đặt tên cho controller là `StudentsController`, sau đó nhấn nút **Add**:



Lưu ý: Theo quy chuẩn đặt tên của mô hình MVC, tên controller là 1 danh từ số nhiều. Tên class của controller là danh từ nói trên kết hợp với từ "Controller".

Class `StudentsController` được tạo ra, kế thừa từ class `Controller`, và có sẵn phương thức `Index()` tương ứng với action **Index**. Đối chiếu với route /Students/ListAll ở trên, ta cần tạo ra action mới có tên là **ListAll** như sau:

```
public class StudentsController : Controller
{
    public IActionResult Index()
    { … }

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

Trong action **ListAll**, ta sẽ viết code để tạo ra dữ liệu sinh viên (trên thực tế, dữ liệu sẽ được lấy từ CSDL). Do có nhiều sinh viên nên ta dùng kiểu `List`[1] của C# để lưu trữ toàn bộ sinh viên này. Sau khi đã có danh sách sinh viên, dùng ViewData/ViewBag để truyền danh sách này sang view:

```
public IActionResult ListAll()
{
    List<Student> students = new List<Student>();
    students.Add(new Student
    {
        Id = 1, Name = "John", GPA = 7.8
    });
    students.Add(new Student
    {
        Id = 2, Name = "Eve", GPA = 4.9
    });
    students.Add(new Student
    {
        Id = 3, Name = "Ricky", GPA = 10
    });
    ViewBag.Students = students;
    return View();
}
```

Thêm dữ liệu mẫu

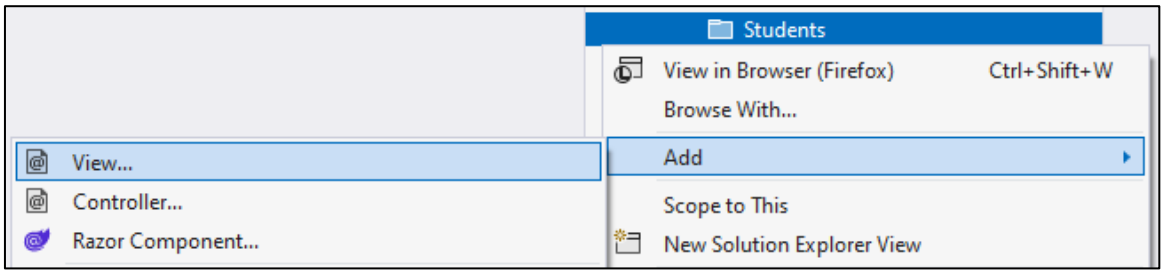
Truyền dữ liệu sang view bằng ViewBag

3. Tạo View

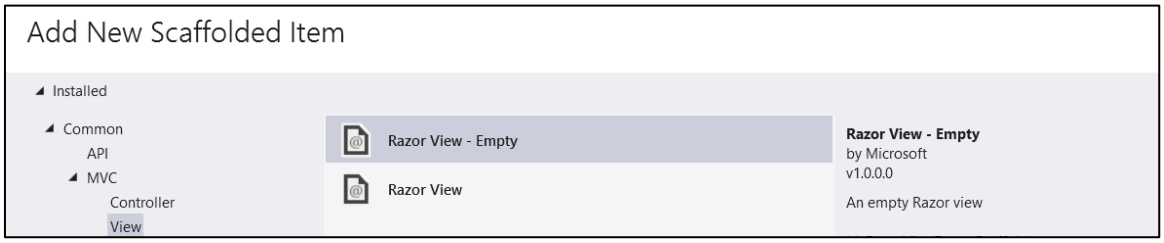
Trong mô hình MVC, mỗi controller sẽ có 1 thư mục cùng tên ở trong thư mục **Views**.

Click chuột phải vào thư mục **Views** và chọn **Add → New Folder**, đặt tên thư mục là **Students** (trùng tên với controller hiện tại).

Click chuột phải vào thư mục vừa tạo và chọn **Add → View...**:

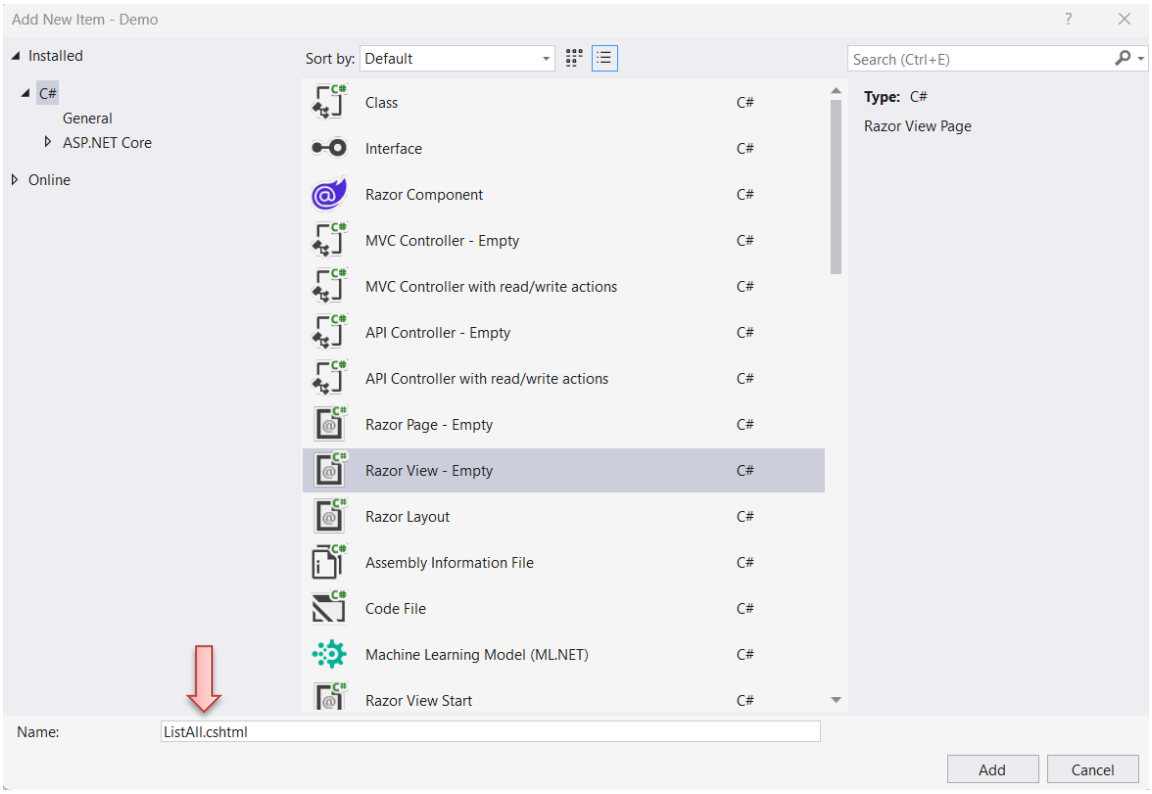


Hộp thoại **Add New Scaffolded Item**, chọn **Razor View – Empty** để tự viết mã nguồn, sau đó nhấn nút **Add**:



[1] Xem thêm về List của C#: https://www.tutorialsteacher.com/csharp/csharp-list

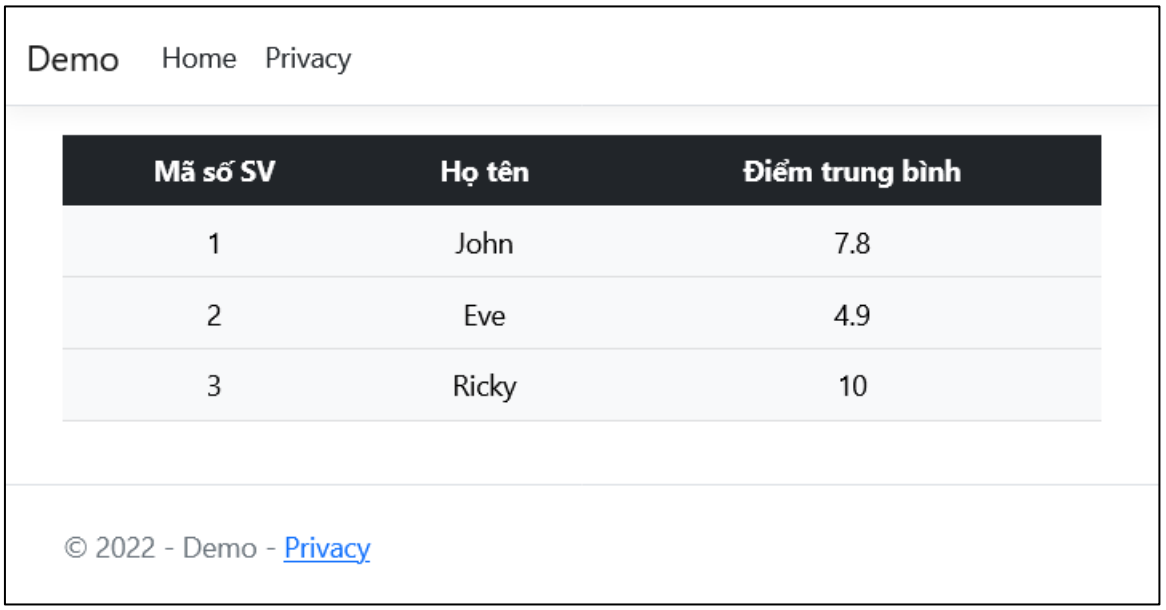
Hộp thoại **Add New Item** đã chọn sẵn **Razor View – Empty**. Đặt tên cho view này là **ListAll.cshtml**, sau đó nhấn nút **Add**:



View **ListAll** sẽ lấy danh sách tài khoản từ ViewBag và dùng vòng lặp để hiển thị thông tin từng sinh viên dưới dạng Table:

```
@{
      List<Student> students = ViewBag.Students;
}
<table class="table table-light text-center">
      <thead class=" table-dark">
            <tr>
                  <th>Mã số SV</th>
                  <th>Họ tên</th>
                  <th>Điểm trung bình</th>
            </tr>
      </thead>
      <tbody>
            @foreach (var item in students)
            {
                  <tr>
                        <td>@item.Id</td>
                        <td>@item.Name</td>
                        <td>@item.GPA</td>
                  </tr>
            }
      </tbody>
</table>
```

Kết quả sau khi biên dịch project và truy cập route /Student/ListAll:



4. Giải thích cách hoạt động của ứng dụng vừa tạo

Ứng dụng MVC ở trên sẽ hoạt động như sau:

- Khi người dùng truy cập route /Student/ListAll, **request** được gửi đến server.
- Server sẽ phân tách route này và chuyển yêu cầu đến controller **Students** và action **ListAll**. Do đó, phương thức `ListAll()` của class `StudentsController` sẽ được gọi.
- Phương thức `ListAll()` xử lý và cho ra danh sách sinh viên, sau đó truyền danh sách này vào ViewBag và gọi view tương ứng bằng câu lệnh `return View();`.
- View tương ứng là Students/ListAll.cshtml có code Razor để xử lý việc hiển thị dữ liệu bằng table.
- Cuối cùng, Razor View Engine sẽ biên dịch view nói trên thành mã HTML và gửi trả về trình duyệt web theo đường **response**.

Lưu ý: Với dữ liệu có cấu trúc phức tạp (trong ví dụ là `List<Student>`), nếu muốn dùng ViewData để truyền dữ liệu thì cần phải chuyển đổi kiểu dữ liệu như sau:

```
List<Student> students = ViewData["Students"] as List<Student>;
```

IX. ViewModel

Trong 1 ứng dụng web, có rất nhiều trang view được xây dựng để thao tác với 1 model nào đó. Ví dụ:

- /Students/ListAll: Hiển thị danh sách sinh viên – Model **Student**.
- /Students/Details: Hiển thị thông tin của 1 sinh viên – Model **Student**.
- /Accounts/Register: Đăng ký tài khoản mới – Model **Account**.
- /Products/Edit: Sửa 1 sản phẩm trong CSDL – Model **Product**.

Với những view dạng này, sẽ không có vấn đề gì nếu ta dùng ViewData/ViewBag để truyền dữ liệu từ controller sang view. Tuy nhiên, dữ liệu trong ViewData/ViewBag không

có kiểu cụ thể, do đó khi lấy dữ liệu phải trải qua bước chuyển đổi kiểu dữ liệu (type conversion) cũng như không tận dụng được các tính chất của lập trình hướng đối tượng và không tận dụng được các model đã tạo.

Do đó, MVC cung cấp **ViewModel**, cho phép lập trình viên khai báo và định nghĩa 1 model có kiểu dữ liệu cụ thể, dùng để chứa dữ liệu cho 1 view nào đó. Controller sẽ phát sinh dữ liệu cho ViewModel, sau đó truyền dữ liệu này sang view. View sẽ dùng dữ liệu từ ViewModel này để hiển thị.

Để khai báo **ViewModel** cho 1 view nào đó, viết câu lệnh sau vào đầu view.

```
@model <kiểu dữ liệu của ViewModel>
```

Ví dụ:

- Với ví dụ ở phần VIII, view **ListAll** hiển thị danh sách các sinh viên. View này cần dữ liệu ở dạng `List<Student>`, do đó ta bổ sung câu lệnh sau vào đầu view **ListAll**:

  ```
  @model List<Student>
  ```

- Giả sử có view **Details** dùng để hiển thị thông tin chi tiết của 1 sinh viên. Khi đó, view này cần dữ liệu ở dạng `Student`:

  ```
  @model Student
  ```

Lưu ý: Mỗi view chỉ có thể có **tối đa 1** ViewModel.

Dữ liệu của ViewModel có thể được truyền từ controller sang view bằng cách truyền tham số cho phương thức `View()` thay vì phải dùng ViewData/ViewBag:

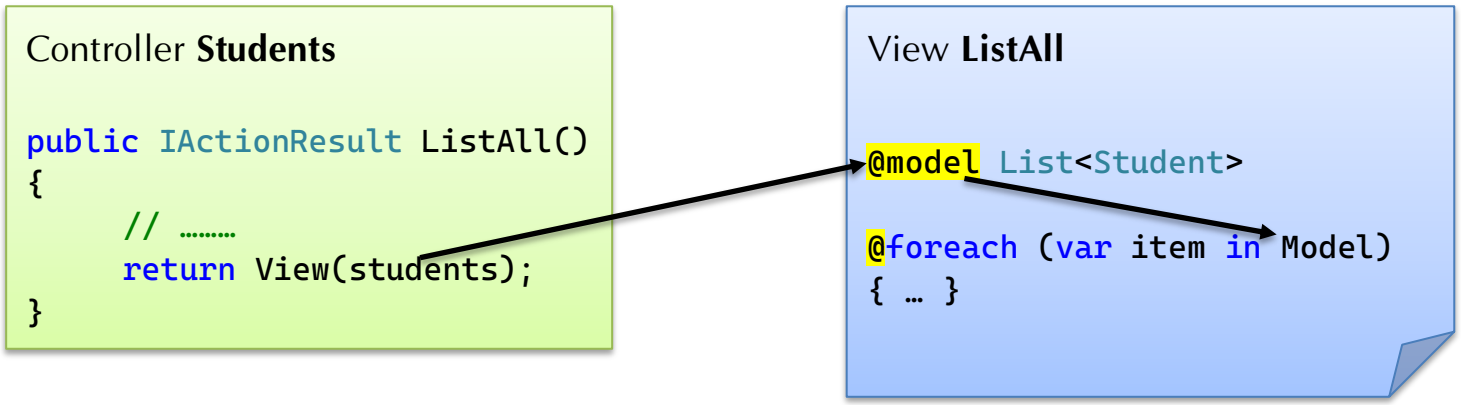
```
public IActionResult ListAll()
{
      // ……..
      ViewBag.Students = students;
      return View(students);
}
```

Khi thao tác với dữ liệu này ở view, chúng ta sẽ thao tác thông qua đối tượng `Model`:

```
@model List<Student>
@{
      List<Student> students = ViewBag.Students;
}

@foreach (var item in students)
@foreach (var item in Model)
{ … }
```

Minh họa việc sử dụng ViewModel:



Việc sử dụng ViewModel sẽ giúp code gọn gàng hơn vì không cần phải đưa dữ liệu vào ViewData/ViewBag và sau đó lại lấy dữ liệu ra, cũng như tận dụng được các tính năng như Tag Helper, Data Annotation, Data Validation[1]…

Lưu ý: ViewModel chỉ dùng để khai báo dữ liệu ở dạng model mà view dùng để hiển thị. Các dữ liệu khác muốn truyền từ controller sang view vẫn cần dùng ViewData/ViewBag.

Ngoài ra, với những ViewModel dạng tập hợp (Collection) như view **ListAll** ở trên, khuyến khích khai báo ở dạng `IEnumerable<T>`. Điều này giúp cho ViewModel có thể nhận nhiều loại collection khác nhau như `Array`, `List`, `Dictionary`… miễn là collection loại đó có triển khai interface `IEnumerable<T>`:

```
@model IEnumerable<Student>
```

[1] Tag Helper, Data Annotation, Data Validation sẽ được trình bày ở buổi 2, buổi 3 và buổi 6