

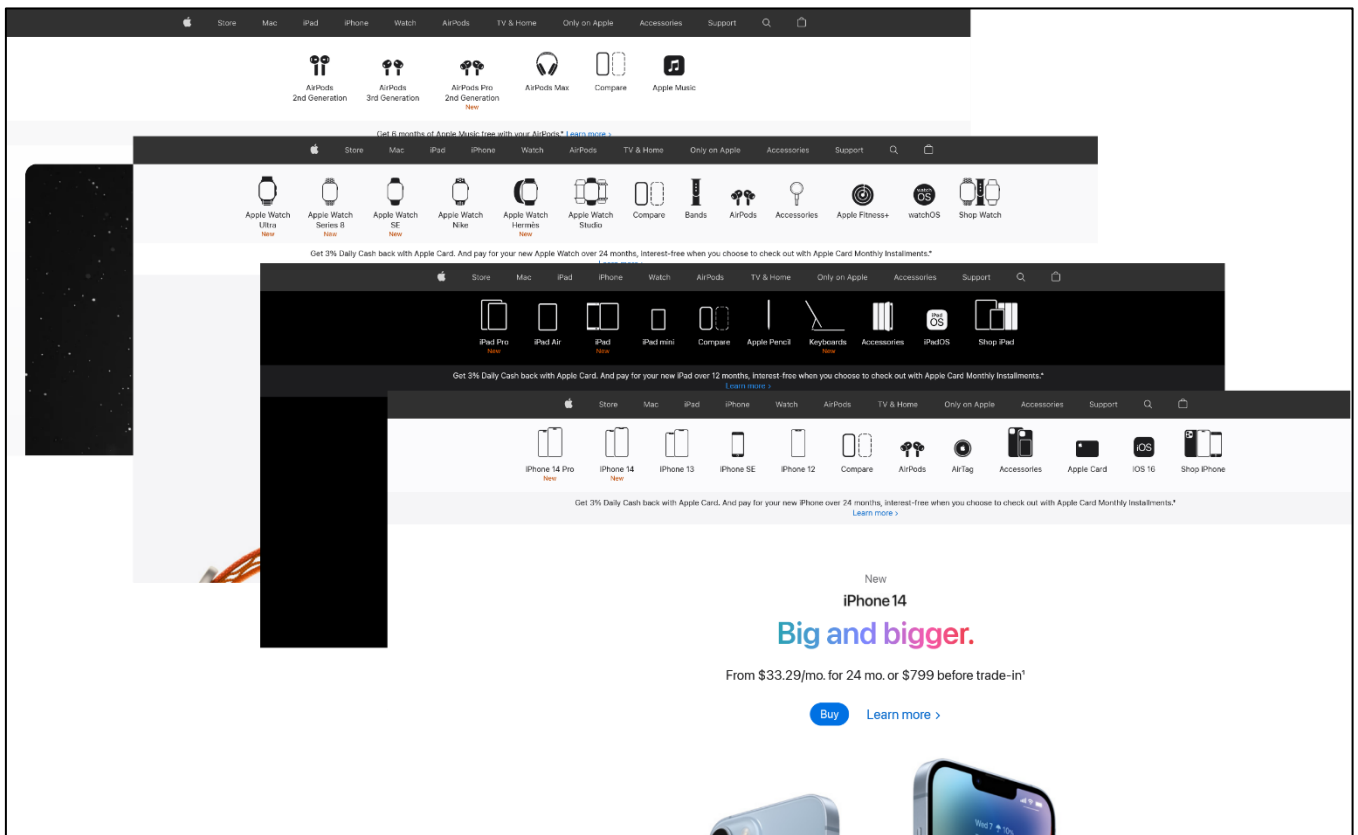
Buổi 2

Thiết kế và xử lý giao diện

I. Layout View

1. Khái niệm

Để đảm bảo tính nhất quán (consistency), một ứng dụng web thường sẽ có bố cục và giao diện giống nhau xuyên suốt tất cả các trang con, thông thường sẽ gồm những phần như Header, Slideshow, NavBar, Sidebar, Footer. Mỗi trang con sẽ lại có phần nội dung khác nhau, ví dụ:

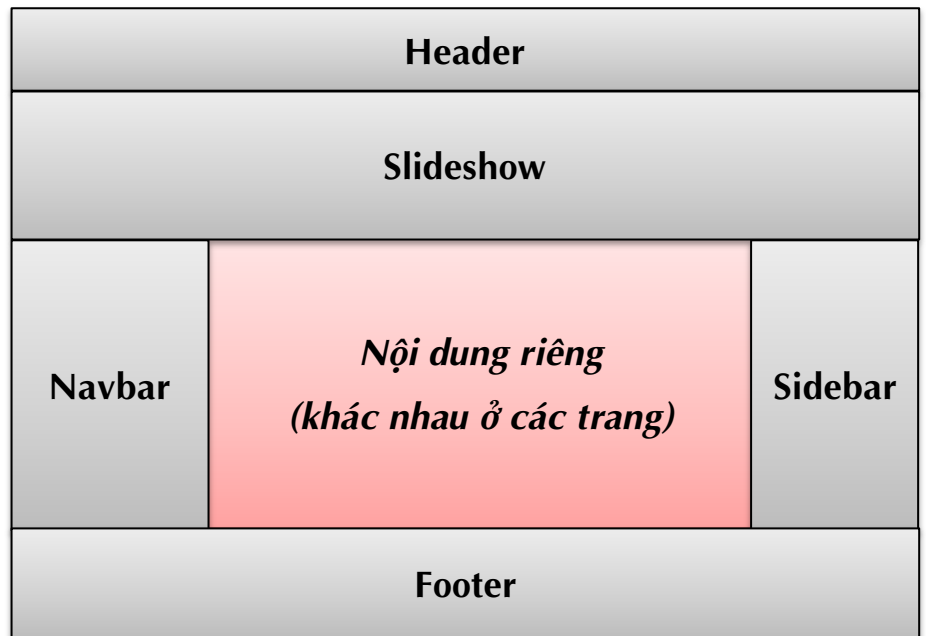


Trong ảnh trên, trang web Apple.com có bố cục và giao diện tương tự nhau ở các trang con, cụ thể là đều có 1 thanh NavBar chính ở trên cùng, dưới thanh NavBar này là 1 sub-menu phân loại sản phẩm. Chỉ có phần nội dung sản phẩm là khác nhau.

Để tránh việc lặp đi lặp lại code của những phần giao diện chung trên mỗi view, ASP.NET Core MVC cung cấp **Layout View** (gọi tắt là layout) giúp định nghĩa những thành phần chung đó. Giờ đây mỗi view chỉ cần chứa những đoạn code cho phần nội dung riêng, còn những phần giao diện chung sẽ được lấy từ layout này.

1 ứng dụng không nhất thiết phải có layout, và 1 ứng dụng cũng có thể có nhiều layout, và các view khác nhau có thể dùng những layout khác nhau.

Khi tạo 1 project ASP.NET Core MVC thì Visual Studio sẽ tự tạo 1 Layout View mặc định nằm trong file `_Layout.cshtml`. Do layout có thể được sử dụng bởi nhiều view khác nhau nên layout nên được đặt trong thư mục Views/Shared.



File `_Layout.cshtml` mặc định có dạng như sau:

```
<!DOCTYPE html>
<html lang="en">
<head>
    .....
</head>
<body>
    <header>
        .....
    </header>
    <div class="container">
        <main role="main" class="pb-3">
            @RenderBody()
        </main>
    </div>
    <footer class="border-top footer text-muted">
        .....
    </footer>
    .....
    @await RenderSectionAsync("Scripts", required: false)
</body>
</html>
```

File `_ViewStart.cshtml` trong thư mục Views dùng để cài đặt những câu lệnh sẽ được thực hiện trước khi mỗi view được chạy. Hiện tại, file này chỉ có 1 dòng code quy định layout mặc định cho các view. Tên layout có thể viết dưới dạng tên ngắn gọn (ví dụ: `_Layout`) hoặc đường dẫn file (ví dụ: `~/Views/Shared/_Layout.cshtml`):

```
@{
```

```
Layout = "_Layout";
```

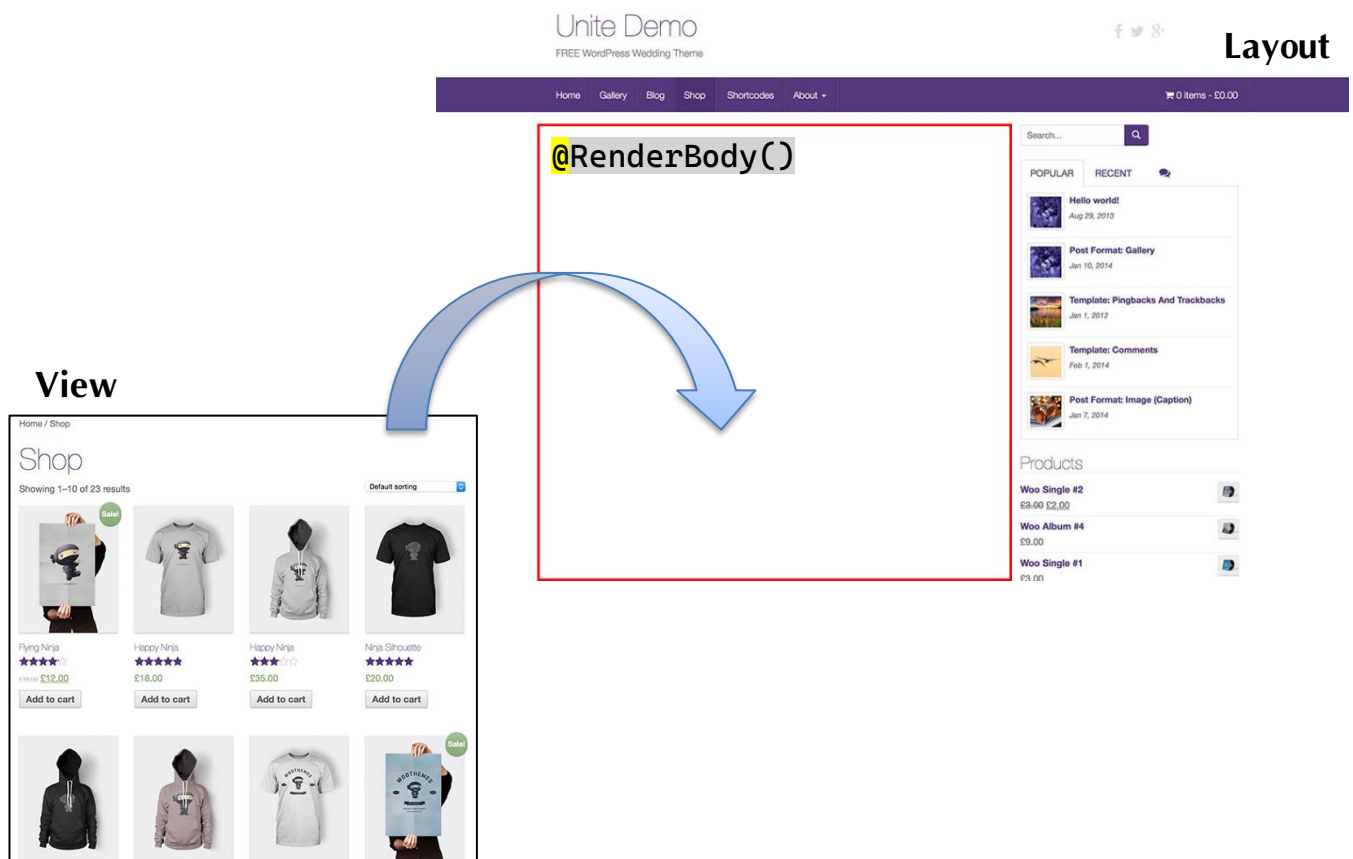
```
// HOẶC
```

```
Layout = "~/Views/Shared/_Layout.cshtml";
```

```
}
```

2. RenderBody() và RenderSection()

Phương thức `RenderBody()` đóng vai trò là **placeholder** cho phần nội dung riêng của các view. Các view có sử dụng layout này sẽ hiển thị nội dung tại nơi đang gọi phương thức `RenderBody()`. Trong 1 layout bắt buộc phải gọi phương thức `RenderBody()` đúng 1 lần.



Trong 1 layout, đôi khi nội dung từ view được đặt vào không phải 1 mà là nhiều vị trí khác nhau. Ví dụ trong ảnh trên, danh sách sản phẩm (mục Products) ở góc dưới của Layout có thể cũng là nội dung riêng của view.

Do đó, 1 layout có thể chứa nhiều **Section**. Section cũng đóng vai trò là **placeholder** để đặt nội dung từ view vào layout. Lập trình viên có thể khai báo nhiều section để chứa các nội dung khác từ view. Phương thức `RenderSection()` nhận 2 tham số là tên của section (tự chọn) và 1 giá trị kiểu **bool** cho biết section này có bắt buộc phải có trong view hay không.

Ví dụ: Trong layout mặc định nói trên có sẵn 1 section có tên là Scripts, và không bắt buộc, tức là view có thể có hoặc không có phần nội dung cho section này.

```
@RenderSection("Scripts", required: false)
```

Trong view, nếu muốn tạo nội dung cho section thì cần viết như sau:

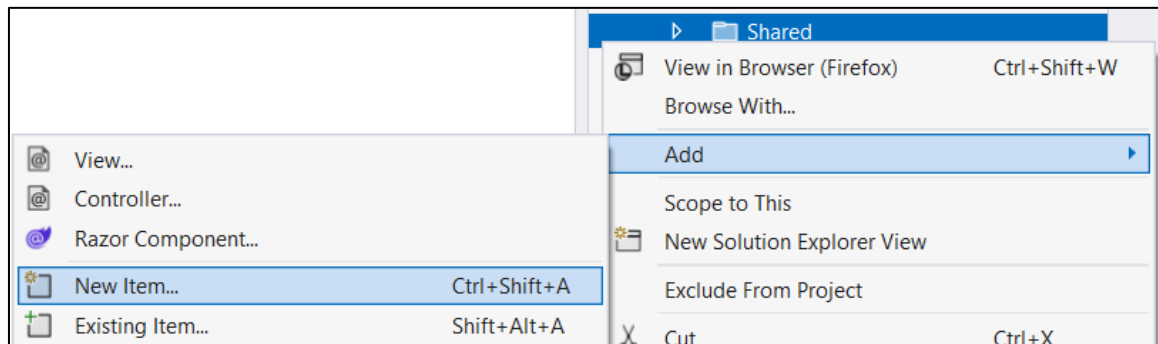
```
@section Tên_section {  
    // Nội dung muốn đặt vào section (HTML, Razor...)  
}
```

Những nội dung trong view không thuộc section nào thì tính là thuộc `RenderBody()`.

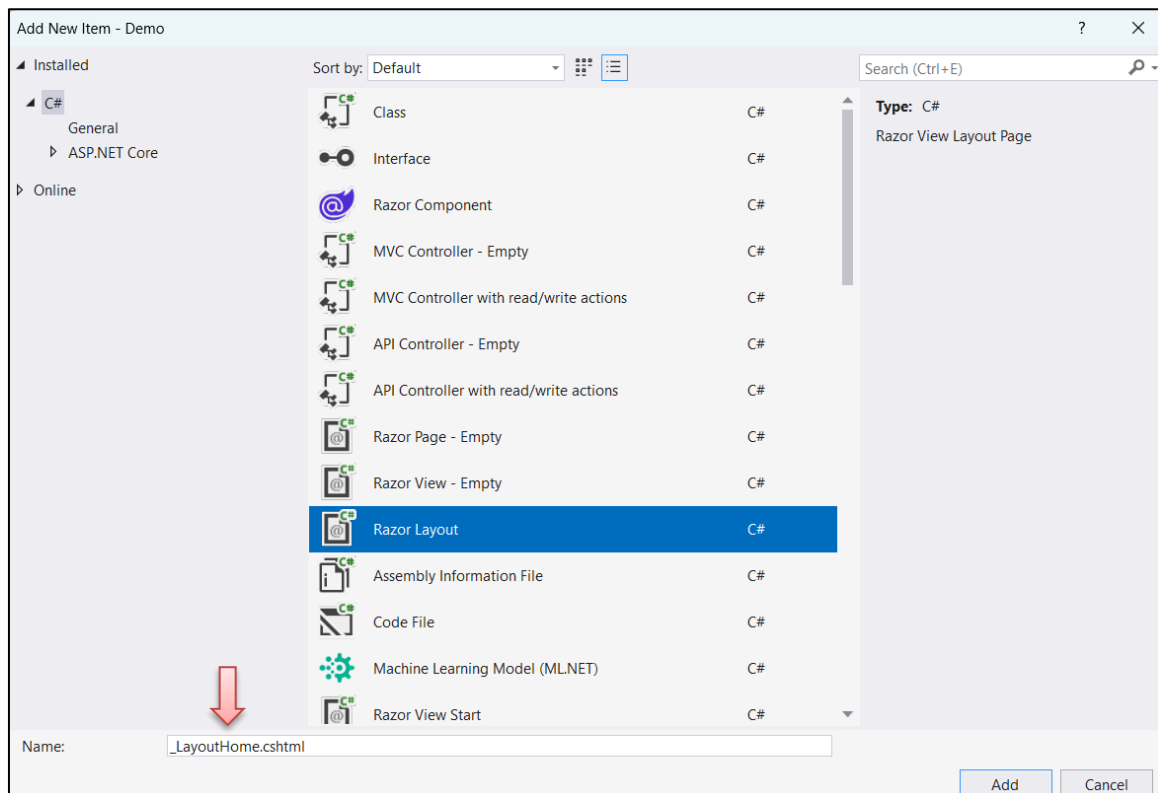
3. Tạo và sử dụng Layout View

Để tạo **Layout View**, chúng ta thực hiện các thao tác sau:

- Click chuột phải vào thư mục Views/Shared (hoặc thư mục muốn đặt layout này) và chọn **Add → New Item...**:



- Hộp thoại **Add New Item**, chọn **Razor Layout**, đặt tên cho file layout (ví dụ: _LayoutHome.cshtml) và nhấn nút **Add**:



Trong file _LayoutHome.cshtml, chúng ta viết thẻ `<body>` như sau:

```
<div>@RenderSection("Top", true)</div>  
<div>@RenderBody()</div>  
<div>@RenderSection("Bottom", false)</div>
```

Sửa lại file `_ViewStart.cshtml` để tất cả view đều dùng layout vừa tạo:

```
@{  
    Layout = "_LayoutHome";  
}
```

Tuy nhiên, khi biên dịch project sẽ gặp lỗi như sau:

```
InvalidOperationException: The layout page '/Views/Shared/_LayoutHome.cshtml'  
cannot find the section 'Top' in the content page '/Views/Home/Index.cshtml'.
```

Lỗi này là do **_LayoutHome** có định nghĩa 1 section **Top** bắt buộc, mà view **Index** đang truy cập thì chưa có nội dung cho section này. Do đó, ta bổ sung đoạn lệnh sau vào vị trí bất kỳ trong view **Index**:

```
@section Top {  
    <div>Đây là nội dung cho section Top</div>  
}
```

Kết quả sau khi biên dịch project:

Đây là nội dung cho section Top

Welcome

Learn about [building Web apps with ASP.NET Core](#).

Nội dung chính, đặt vào `RenderBody()`

4. Sử dụng nhiều Layout trong 1 ứng dụng

Trong 1 ứng dụng có thể có nhiều layout khác nhau, ví dụ layout cho thành viên thường, layout cho admin... Thậm chí, một vài view có thể sử dụng layout riêng. Layout mặc định cho các view được cài đặt trong file `_ViewStart.cshtml`, tuy nhiên, chúng ta có thể cài đặt cho view sử dụng 1 layout khác với mặc định bằng cách viết câu lệnh sau vào đầu view:

```
@{  
    Layout = "Tên_Layout_muốn_dùng";  
}
```

Ví dụ: Cài đặt layout mặc định là **_Layout**, và tạo 1 view dùng **_LayoutHome**:

- Tạo action **Contact** trong controller **Home** như sau:

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

- Tạo view **Contact** nằm trong thư mục `Views/Home`. View này sẽ dùng layout có tên là **_LayoutHome** vừa tạo ở phần 1.3 ở trên.

- Trong view **Contact**, viết các câu lệnh sau:

```
@{
    Layout = "_LayoutHome";
}

<h1>Nội dung chính</h1>

@section Top {
    <h2>Section trên</h2>
}

@section Bottom {
    <h3>Section dưới</h3>
}

<h4>Đây cũng là nội dung chính</h4>
```

- Kết quả sau khi biên dịch project và truy cập route /Home/Contact (xem ảnh bên).
- Các phần nội dung từ view **Contact** sẽ được đặt vào **_LayoutHome** như sau: Các section của view sẽ được đặt vào nơi gọi các phương thức **RenderSection()** tương ứng với tên section đó. Toàn bộ phần còn lại của view không thuộc section nào sẽ được đặt vào nơi gọi phương thức **RenderBody()**.

Section trên

Nội dung chính

Đây cũng là nội dung chính

Section dưới

View **Contact**

```
<h1>Nội dung chính</h1>

@section Top {
    <h2>Section trên</h2>
}

@section Bottom {
    <h3>Section dưới</h3>
}

<h4>Đây cũng là nội dung
chính</h4>
```

_LayoutHome

```
<div>
    @RenderSection("Top", true)
</div>

<div>
    @RenderBody()
</div>

<div>
    @RenderSection("Bottom",
        false)
</div>
```

5. So sánh `RenderBody()` và `RenderSection()`

| <code>RenderBody()</code> | <code>RenderSection()</code> |
|--|---|
| Trong layout bắt buộc phải có 1 và chỉ 1 lần gọi phương thức <code>RenderBody()</code> . | Trong layout có thể không có, có 1 hoặc có nhiều lần gọi phương thức <code>RenderSection()</code> . |
| Phương thức <code>RenderBody()</code> sẽ hiển thị tất cả nội dung của view mà không thuộc section nào. | Phương thức <code>RenderSection()</code> sẽ chỉ hiển thị phần nội dung của view được cài đặt trong section tương ứng. |
| Phương thức <code>RenderBody()</code> không kèm theo tham số. | Phương thức <code>RenderSection()</code> có kèm tham số: <ul style="list-style-type: none">Tên section: <code>string</code>.Section có bắt buộc không: <code>boolean</code>. |

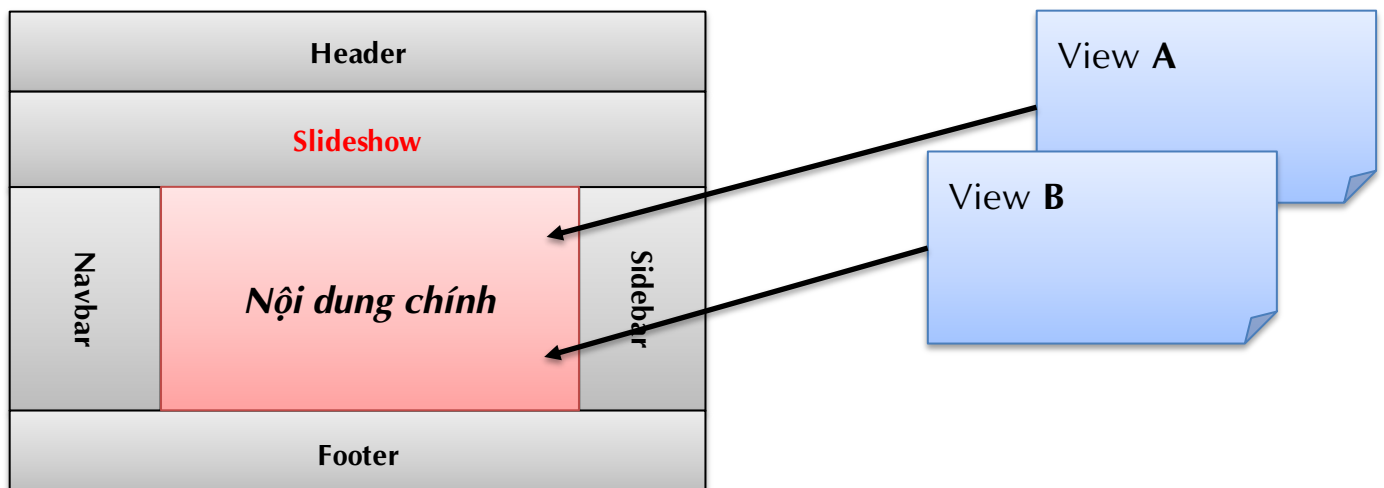
II. Partial View

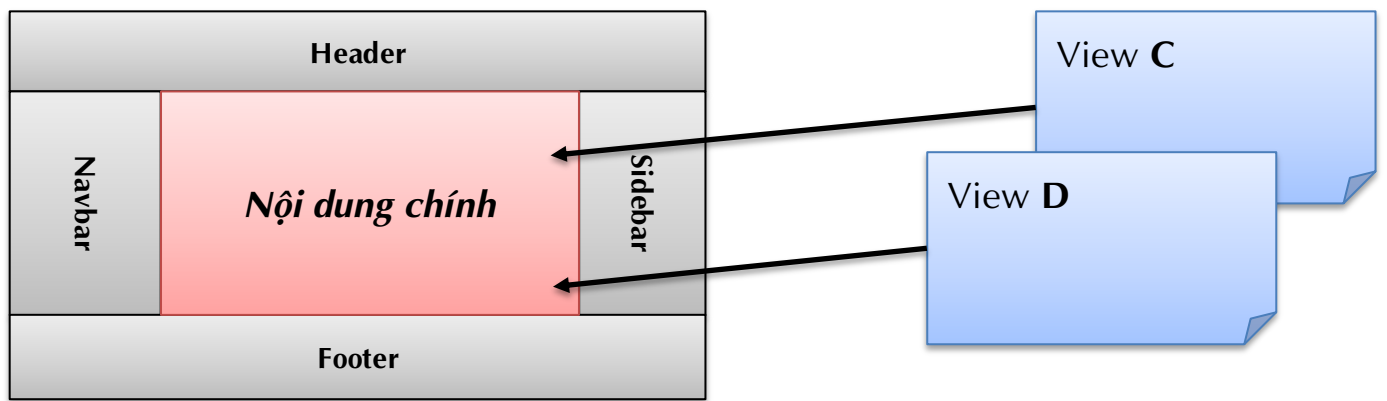
1. Khái niệm

Khi xây dựng giao diện, có những thành phần giao diện được sử dụng nhiều lần ở nhiều nơi khác nhau. Ví dụ:

- Nhiều view cùng sử dụng 1 slideshow giống nhau.
- Nhiều layout cùng sử dụng 1 navbar giống nhau.
- ...

Những trường hợp trên vẫn có thể giải quyết phần nào bằng Layout view, tuy nhiên sẽ gặp một số bất cập. Ví dụ: View **A**, **B** cùng có chung slideshow còn view **C**, **D** thì không, trong khi cả 4 view vẫn phải có layout tương tự nhau. Khi đó chúng ta phải tạo 1 layout có slideshow cho view **A**, **B** và 1 layout tương tự, không có slideshow cho view **C**, **D**:

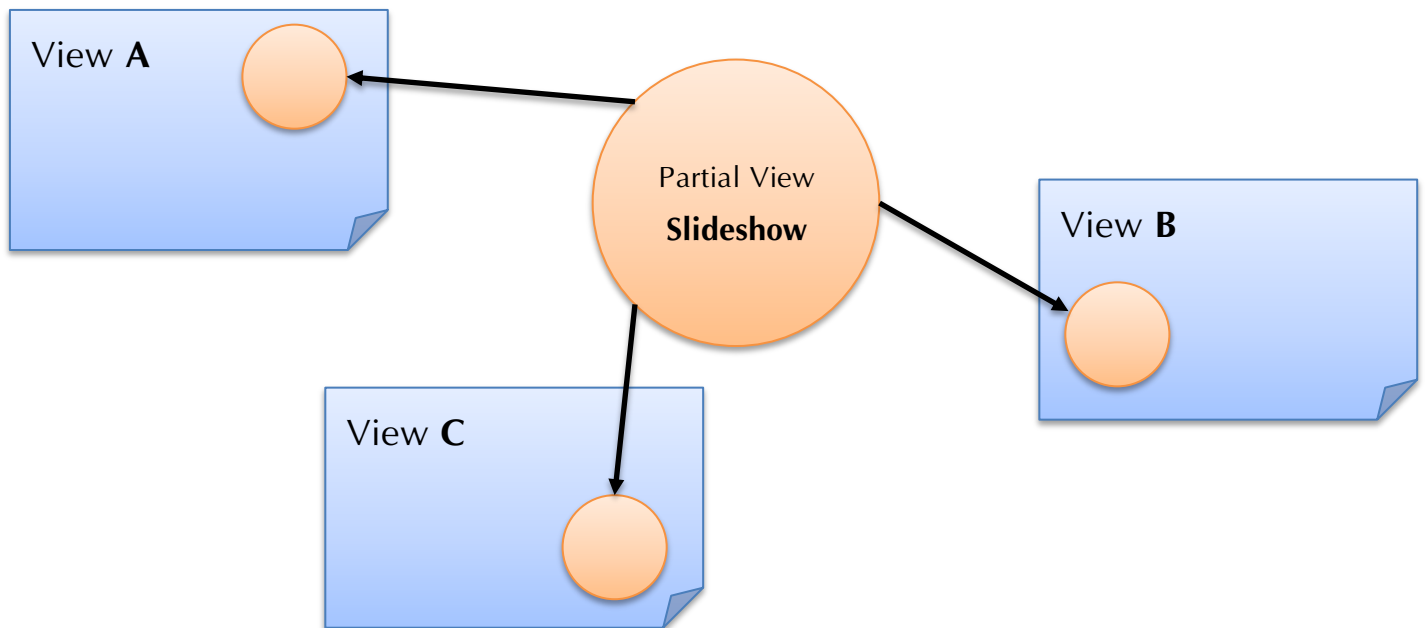




Việc xây dựng layout riêng như vậy khá bất tiện vì sẽ phát sinh ra nhiều layout khác nhau dẫn đến khó quản lý. Do đó, MVC cung cấp chức năng **Partial View**, cho phép tạo ra các thành phần giao diện có thể tái sử dụng được trên nhiều view hoặc layout khác nhau, giúp dễ quản lý, giảm được việc trùng lặp code.

Partial View là 1 file được viết bằng Razor tương tự như view, thường được đặt trong thư mục **Views/Shared** (để có thể dùng được trong tất cả các view/layout). Mã nguồn của Partial View là Razor, do đó phần mở rộng của nó cũng là **.cshtml** tương tự như view và layout.

Minh họa về công dụng của Partial View: tạo ra slideshow dùng được trong một số view:



Với cách dùng Partial View để tạo ra slideshow như vậy, nếu sau này cần chỉnh sửa code của slideshow, chỉ cần chỉnh sửa ở 1 chỗ duy nhất, đảm bảo code tuân thủ quy tắc **DRY**¹ trong xây dựng phần mềm.

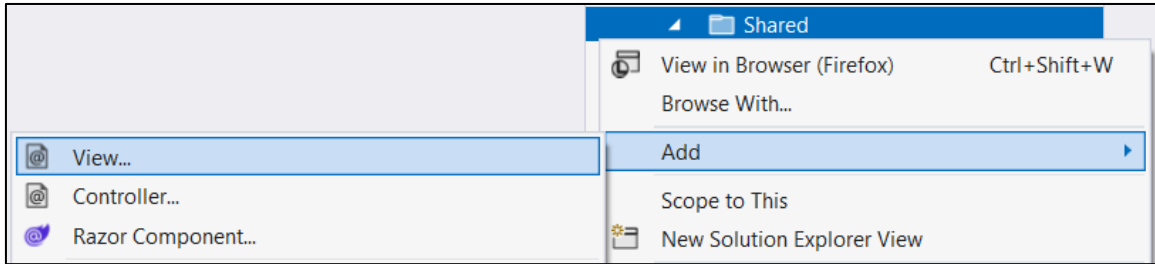
¹ DRY – Don't Repeat Yourself: Đây là 1 quy tắc trong xây dựng phần mềm, nhằm tới việc giảm sự lặp lại của code, bằng cách thay thế những đoạn lặp lại bằng hàm, phương thức... để có thể viết 1 lần, gọi nhiều lần. Quy tắc này giúp giảm sự trùng lặp về code, khiến code gọn gàng, dễ đọc, dễ bảo trì hơn. Xem thêm về quy tắc DRY:

https://en.wikipedia.org/wiki/Don%27t_repeat_yourself hoặc <https://www.softwareyoga.com/is-your-code-dry-or-wet/>

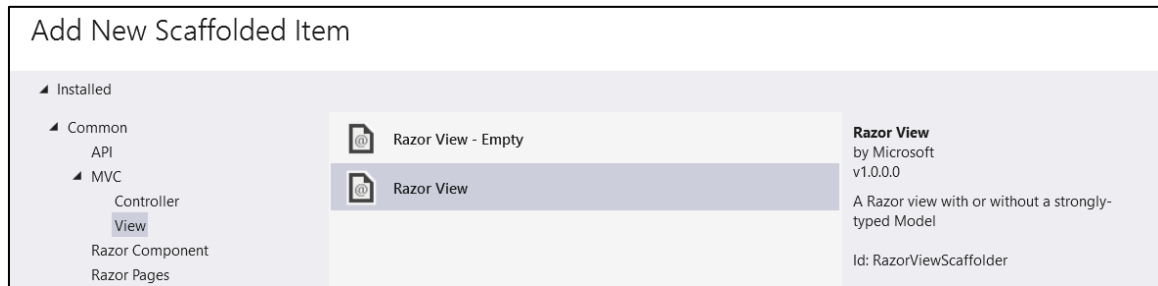
2. Tạo và sử dụng Partial View

Để tạo **Partial View**, chúng ta thực hiện các thao tác sau:

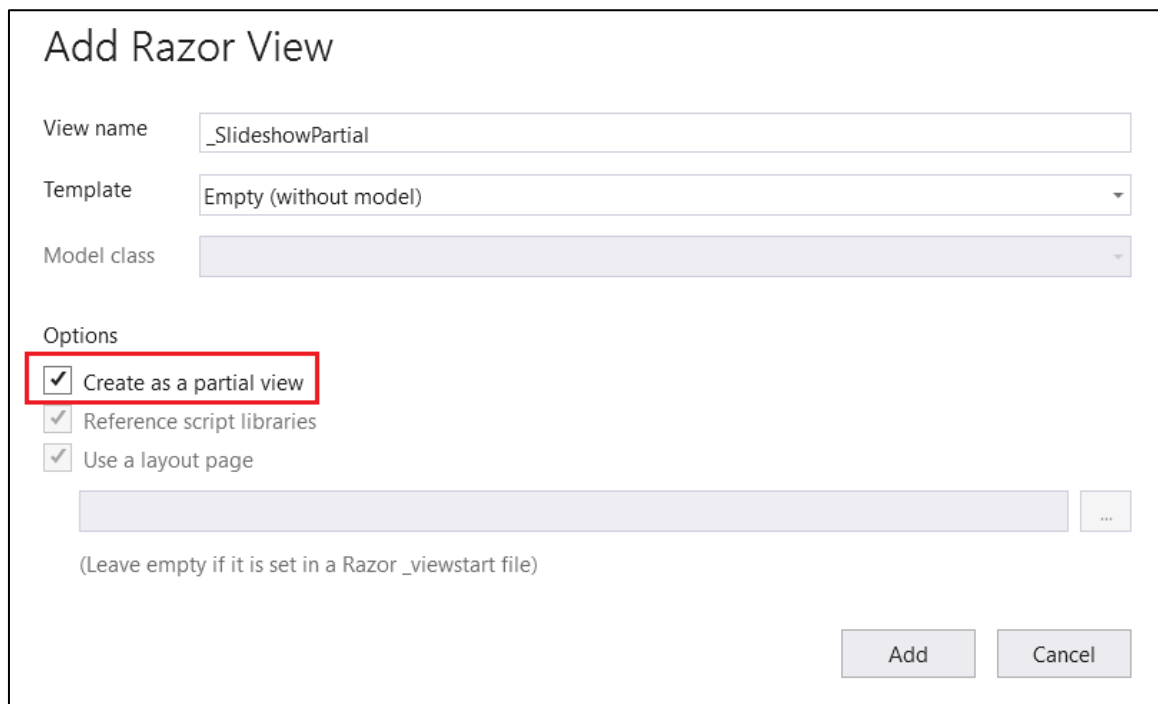
- Click chuột phải vào thư mục Views/Shared và chọn **Add → View...**:



- Hộp thoại **Add New Scaffolded Item**, chọn **Razor View** và nhấn nút **Add**:



- Hộp thoại **Add Razor View**, check vào mục **Create as a partial view**, đặt tên cho Partial View ở mục **View name** (ví dụ: **_SlideshowPartial¹**) và nhấn nút **Add**:



Ta sẽ cài đặt 1 slideshow đơn giản vào Partial View vừa tạo. Đoạn code để tạo ra 1 slideshow đơn giản từ Bootstrap có sẵn trong file **Slideshow.txt**. Copy toàn bộ đoạn code này và đặt vào file **_SlideshowPartial.cshtml** (lưu ý rằng slideshow ví dụ này chỉ có 1 ảnh).

¹ Khuyến khích đặt tên Partial View có ký tự **_** ở đầu để phân biệt với View, và có hậu tố "Partial" để phân biệt với Layout.

Để chèn 1 Partial View vào 1 view hoặc 1 layout, chúng ta dùng cú pháp sau:

```
<partial name="tên Partial View" />
```

Trong đó: Tên Partial View có thể viết dưới dạng tên ngắn gọn (ví dụ: **_SlideshowPartial**) hoặc đường dẫn file (ví dụ: ~/Views/Shared/_SlideshowPartial.cshtml).

Ví dụ: Để view **Index** và **Privacy** của controller **Home** sử dụng slideshow vừa tạo, bổ sung câu lệnh sau vào cả 2 view:

```
<partial name="_SlideshowPartial" />
```

3. Truyền model cho Partial View

Tương tự view, Partial View cũng có thể nhận dữ liệu dưới dạng ViewModel bằng cách thực hiện thao tác sau:

- Khai báo directive **@model** ở đầu Partial View.
- Truyền dữ liệu vào Partial View bằng thuộc tính **model** của thẻ **<partial />**.

Ví dụ: Cài đặt lại model **Student** và controller **Students** như ở buổi 1. Dùng action **Index** hiển thị danh sách sinh viên. Thông tin mỗi sinh viên hiển thị trong 1 thẻ **<div>**:

- Tạo Partial View trong thư mục Views/Shared, đặt tên là **_StudentInfoPartial**.
- Cài đặt cho Partial View này sử dụng ViewModel là **Student**, kèm theo thẻ **<div>** hiển thị thông tin sinh viên với giao diện tùy chọn:

```
@model Student

<div class="col border border-primary">
    <ul>
        <li>Mã SV: @Model.Id</li>
        <li>Họ tên: @Model.Name</li>
        <li>Điểm TB: @Model.GPA</li>
    </ul>
</div>
```

- View **Index** sử dụng Partial View vừa tạo để hiển thị từng sinh viên:

```
@model IEnumerable<Student>

<div class="row">
    @foreach (var item in Model)
    {
        <partial name="_StudentInfoPartial" model="item" />
    }
</div>
```

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

| | | |
|--|---|--|
| Demo Home Privacy | | |
| <ul style="list-style-type: none"> • Mã SV: 1 • Họ tên: John • Điểm TB: 7.8 | <ul style="list-style-type: none"> • Mã SV: 2 • Họ tên: Eve • Điểm TB: 4.9 | <ul style="list-style-type: none"> • Mã SV: 3 • Họ tên: Ricky • Điểm TB: 10 |
| © 2022 - Demo - Privacy | | |

Thuộc tính **model** của thẻ `<partial />` cũng có thể nhận vào 1 đối tượng theo dạng sau:

```
<partial name="_StudentInfoPartial"
    model='new Student { Id = 4, Name = "Peter", GPA = 6.5 }' />
```

4. Mục đích sử dụng của Partial View

Partial View thường được dùng để:

- Chia 1 view/layout thành nhiều phần nhỏ: Với 1 view/layout phức tạp, việc chia nhỏ thành các Partial View sẽ khiến view/layout chính ngắn gọn, dễ đọc hơn vì chỉ chứa code cấu trúc tổng thể, còn lại các thành phần sẽ được gọi từ Partial View.
- Giảm bớt việc trùng lặp code: Đối với những thành phần giao diện chung trên nhiều view/layout, việc cài đặt thành phần đó bằng Partial View sẽ giúp giảm trùng lặp code cũng như dễ thay đổi sau này.

Tuy nhiên, Partial View **không nên** dùng trong các trường hợp:

- Cài đặt các thành phần bố cục chung → Nên sử dụng **Layout View**.
- Đoạn code phức tạp hoặc cần xử lý logic nhiều (vì bản chất Partial View chỉ đơn giản là 1 file chứa code Razor, không đi kèm theo controller để xử lý logic) → Nên sử dụng **View Component**¹.

III. Tag Helper

Tag Helper cho phép lập trình viên viết các thẻ HTML trong Razor bằng cú pháp thân thiện với HTML. Cú pháp này tương tự như thuộc tính của thẻ HTML nhưng được xử lý bởi **Razor View Engine**² ở phía server, do đó tận dụng được các ưu điểm của việc xử lý ở phía server (ví dụ: có thể tương tác với model, hỗ trợ IntelliSense...).

¹ View Component sẽ được trình bày ở buổi 7

² Razor View Engine sẽ được trình bày ở phần IV.1

ASP.NET Core hỗ trợ sẵn rất nhiều Tag Helper¹. Trong buổi 2, chúng ta chỉ tìm hiểu những Tag Helper thường dùng nhất.

Visual Studio định dạng phần Tag Helper có **màu tím và in đậm**.

1. Anchor và Form

Thẻ `<a>` và `<form>` có 1 số Tag Helper hỗ trợ tạo liên kết đến các route trong ứng dụng hoặc các URL trên Internet (đối với thẻ `<a>`) hoặc quy định route sẽ nhận dữ liệu được submit từ form (đối với thẻ `<form>`):

- **asp-controller**: Tên controller đang trỏ đến. Nếu không có thì mặc định là controller hiện tại.
- **asp-action**: Tên action đang trỏ đến. Nếu không có thì xem như đang trỏ đến action mặc định. Hiện tại, action mặc định của mỗi controller là **Index** (được quy định tại phần routing² ở file **Program.cs**).
- **asp-route-x**: Giá trị tham số của route, với x là tên tham số. Hiện tại, tham số mặc định của route là **id**.
- **asp-route**: Tên của route³ đang trỏ đến.
- **asp-all-route-data**: Danh sách các tham số của route dưới dạng **Dictionary**.
- **asp-fragment**: Phần # ghi kèm theo route (thường dùng để tạo liên kết đến 1 phần tử có **id** nào đó trong DOM).
- **asp-protocol**: Giao thức dùng để truy cập route (http, https, ftp...).
- **asp-host**: Server/domain đang trỏ đến. Nếu không có thì xem như đang trỏ đến server/domain hiện tại.

Ví dụ: Trong các ví dụ dưới đây, giả sử các thẻ `<a>` đặt trong view **Home/Index** (tương ứng với controller **Home** và action **Index**). Các ví dụ về thẻ `<form>` sẽ tương tự thẻ `<a>`:

- Tạo liên kết đến route /Home/Privacy:

```
<a asp-controller="Home" asp-action="Privacy">...</a>  
<a asp-action="Privacy">...</a>
```
- Tạo liên kết đến route /Students/Index:

```
<a asp-controller="Students" asp-action="Index">...</a>  
<a asp-controller="Students">...</a>
```
- Tạo liên kết đến route có tên là **demo**:

¹ Xem thêm về Tag Helper <https://learn.microsoft.com/en-us/aspnet/core/mvc/views/tag-helpers/intro?view=aspnetcore-6.0>

² Xem lại tài liệu buổi 1, phần IV.2: Routing trong mô hình MVC

³ Xem lại tài liệu buổi 1, phần IV.2.b: Attribute Routing – Routing theo thuộc tính

- Tạo liên kết đến route /Students/Details/5 (hoặc /Students/Details?id=5):

```
<a asp-route="demo">...</a>
```
- Tạo liên kết đến route /Students/Details/5 (hoặc /Students/Details?id=5):

```
<a asp-controller="Students" asp-action="Details"
  asp-route-id="5">...</a>
```
- Tạo liên kết đến route /Students/Details/5?age=31:

```
<a asp-controller="Students" asp-action="Details"
  asp-route-id="5" asp-route-age="31">...</a>
```
- Tạo liên kết đến route /Students/Details/5?age=31&country=VN:

```
@{
    var routeData = new Dictionary<string, string>()
    {
        { "id", "5" }, { "age", "31" }, { "country", "VN" }
    };

    <a asp-controller="Students" asp-action="Details"
      asp-all-route-data="routeData">...</a>
```

- Tạo liên kết đến route ftp://www.eshop.com/Students/Index#list:

```
<a asp-protocol="ftp" asp-host="www.eshop.com"
  asp-controller="Students" asp-action="Index"
  asp-fragment="list">...</a>
```

2. Label và Input

Tag Helper `asp-for` giúp tạo thẻ `<label>`¹ và thẻ `<input>`² tương ứng với 1 thuộc tính X nào đó của ViewModel:

- Thẻ `<label>` được tạo ra sẽ có nội dung và thuộc tính `for` dựa trên X.
- Thẻ `<input>` được tạo ra sẽ có thuộc tính `id` và `name` dựa trên X, có thuộc tính `type` tùy thuộc vào kiểu dữ liệu của X.

Ví dụ: Sử dụng Tag Helper để tạo form thêm sinh viên theo các bước sau:

- Cài đặt action **Add** thuộc controller **Students**:

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

- Tạo view **Add** thuộc thư mục **Students** và quy định ViewModel:

```
@model Student
```

¹ Xem thêm về Tag Helper cho thẻ `<label>`: <https://learn.microsoft.com/en-us/aspnet/core/mvc/views/working-with-forms?view=aspnetcore-6.0#the-label-tag-helper>

² Xem thêm về Tag Helper cho thẻ `<input>`: <https://learn.microsoft.com/en-us/aspnet/core/mvc/views/working-with-forms?view=aspnetcore-6.0#the-input-tag-helper>

- Tạo form đăng nhập trong view **Add** như sau:

```
<form asp-controller="Students" asp-action="Add">
  <div class="form-group">
    <label asp-for="Id"></label>
    <input asp-for="Id" class="form-control" />
  </div>
  <div class="form-group">
    <label asp-for="Name"></label>
    <input asp-for="Name" class="form-control" />
  </div>
  <div class="form-group mb-2">
    <label asp-for="GPA"></label>
    <input asp-for="GPA" class="form-control" />
  </div>
  <button type="submit" class="btn btn-primary">Add</button>
</form>
```

Kết quả sau khi biên dịch project và truy cập route /Students/Add:

The screenshot shows a web application interface. At the top, there is a header with links: "Demo", "Home", and "Privacy". Below the header is a form titled "Add". The form contains three input fields: "Id" (a text box with a small dropdown arrow on the right), "Name" (a text box), and "GPA" (a text box). Below the input fields is a blue button labeled "Add". At the bottom of the page, there is a footer with the text "© 2022 - Demo - [Privacy](#)".

Trong ví dụ trên:

- Thẻ `<label>` dành cho thuộc tính `Name` sẽ có nội dung và thuộc tính `for` là `Name`:

```
<label asp-for="Name"></label>
// ↓ được biên dịch thành
<label for="Name">Name</label>
```

- Thẻ `<input>` dành cho thuộc tính `Id` sẽ có thuộc tính `id` và `name` là `Id` và thuộc tính `type` là `number` (vì thuộc tính `Id` của ViewModel có kiểu dữ liệu là `int`):

```
<input asp-for="Id" />
// ↓ được biên dịch thành
<input type="number" id="Id" name="Id" />
```

- Thẻ `<input>` dành cho thuộc tính **Name** sẽ có thuộc tính **id** và **name** là **Name** và thuộc tính **type** là **text** (vì thuộc tính **Name** của ViewModel có kiểu dữ liệu là **string**):

```
<input asp-for="Name" />
// ↓ được biên dịch thành
<input type="text" id="Name" name="Name" />
```

Ngoài ra, Tag Helper còn hỗ trợ tạo ra các thuộc tính kiểm tra dữ liệu (data validation) của HTML5 dựa trên các **Data Annotation**¹ của model.

Các kiểu dữ liệu của .NET và các loại thẻ `<input>` tương ứng:

| Kiểu dữ liệu .NET | Loại control | Input Type |
|--------------------------------|----------------|------------------------------------|
| <code>bool</code> | Checkbox | <code>type="checkbox"</code> |
| <code>string, double</code> | Textbox | <code>type="text"</code> |
| <code>DateTime</code> | DateTimePicker | <code>type="datetime-local"</code> |
| <code>byte, int, single</code> | Numeric | <code>type="number"</code> |

3. Select

Thẻ `<select>`² có Tag Helper `asp-items` cho phép quy định dữ liệu cho thẻ này. Dữ liệu này sẽ là 1 đối tượng thuộc kiểu `SelectList`.

Ví dụ: Tạo drop-down menu cho phép lựa chọn chuyên ngành.

- Thay đổi action **Add** của controller **Students** như sau:

```
public IActionResult Add()
{
    List<string> majors = new List<string> { "Information
Technology", "Mathematics", "English", "Chemistry" };
    ViewBag.Majors = new SelectList(majors);
    return View();
}
```

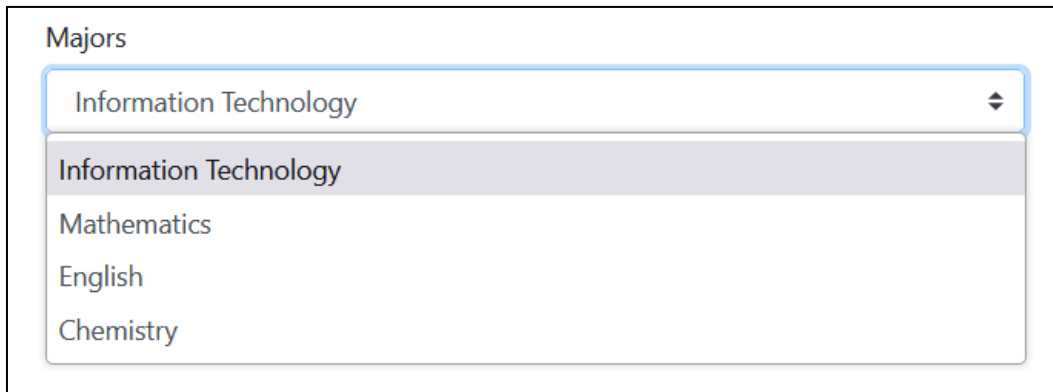
- Bổ sung thẻ `<select>` vào cuối form thêm sinh viên ở view **Add** như sau:

```
<div class="form-group">
    <label>Majors</label>
    <select asp-items="@ViewBag.Majors"
        class="custom-select"></select>
</div>
```

¹ Data Annotation – Chú thích dữ liệu sẽ được trình bày ở buổi 3

² Xem thêm về Tag Helper cho thẻ `<select>`: <https://learn.microsoft.com/en-us/aspnet/core/mvc/views/working-with-forms?view=aspnetcore-6.0#the-select-tag-helper>

Kết quả sau khi biên dịch project và truy cập route /Students/Add:



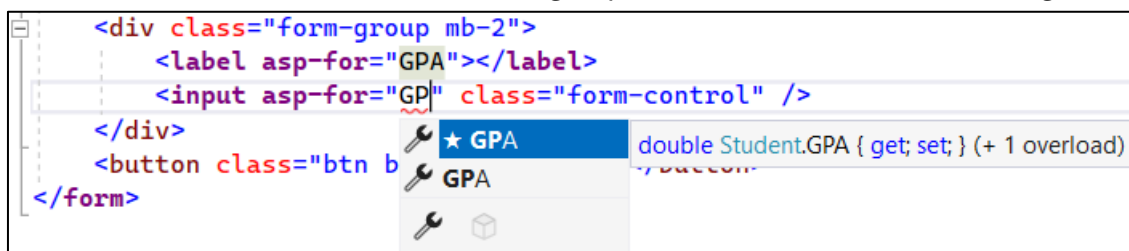
Trong ví dụ trên, dữ liệu dùng để đưa vào thẻ `<select>` là dữ liệu được tạo từ controller. Ở các buổi sau, sau khi đã học về cách tương tác với CSDL, ta sẽ biết cách lấy dữ liệu từ CSDL và đưa vào trong thẻ `<select>`.

4. Ưu điểm của Tag Helper

Tag Helper giúp lập trình viên thiết kế view nhanh hơn vì cú pháp của Tag Helper khá thân thiện với HTML. Chúng ta cũng có thể thêm các thuộc tính HTML cũng như CSS vào ngay trong thẻ HTML song song với việc sử dụng Tag Helper như sau:

```
<input asp-for="Id" placeholder="Student's ID" class="form-control" />
```

Tag Helper cũng hỗ trợ cơ chế gợi ý IntelliSense của Visual Studio. Trong ví dụ trên, khi tạo form thêm sinh viên, IntelliSense hỗ trợ gợi ý dựa trên ViewModel đang sử dụng:



Ngoài ra, do Tag Helper tạo thẻ `<input>` dựa trên model, nên khi kiểu dữ liệu của các thuộc tính của model thay đổi, thẻ `<input>` sẽ tự động thay đổi theo.

IV. Razor

1. Tổng quan

Ngôn ngữ HTML chỉ có khả năng định dạng văn bản chứ không thể thực hiện các phép toán logic và các câu lệnh điều khiển (rẽ nhánh, vòng lặp...). Mặc dù các ngôn ngữ lập trình back-end đều có khả năng tạo ra HTML để trả về cho trình duyệt web, nhưng phải sử dụng cú pháp riêng của ngôn ngữ đó, vốn cồng kềnh và không tiện lợi.

Do đó, người ta tạo ra những **View Engine** cùng ngôn ngữ đánh dấu (**markup language**) riêng của nó nhằm kết hợp ngôn ngữ lập trình back-end với HTML, biên dịch, tạo ra mã

nguồn HTML thuần và hiển thị trên trình duyệt web. Nói cách khác, View Engine có vai trò như chương trình dịch (compiler, interpreter) để chuyển đổi ngôn ngữ đó thành HTML.

Razor là 1 loại cú pháp/ngôn ngữ đánh dấu kết hợp giữa C# và HTML và đồng thời cũng là tên của 1 View Engine dùng để tạo ra HTML động trong ứng dụng ASP.NET Core. Razor được dùng trong Razor Pages, MVC và Blazor¹ với nhiệm vụ đánh dấu đâu là C#, đâu là HTML trong mã nguồn để **Razor View Engine** xử lý cho phù hợp. File mã nguồn Razor có phần mở rộng là **.cshtml**.

Ví dụ: Một đoạn code Razor đơn giản:

```
@{
    double math = 9.8;
    double physics = 5.6;
    double chemistry = 10;
}
<p>Current time: @DateTime.Now</p>
<p>GPA: @(math + physics + chemistry) / 3</p>
@if (math < 5)
{
    <p>You've FAILED at Math</p>
}
```

Nhờ có cú pháp Razor mà view engine có thể nhận ra đoạn code C# (được tô màu nền xám trong ví dụ trên) và đoạn code HTML (không có màu nền trong ví dụ trên) để xử lý và biên dịch cho đúng.

2. Các nguyên tắc cơ bản của Razor

Một số nguyên tắc cơ bản của Razor gồm có:

- Ngôn ngữ mặc định trong file .cshtml là HTML. Tức là khi không có gì đặc biệt trong thì Razor View Engine sẽ xem đoạn code đó là HTML.
- Ký tự @ yêu cầu Razor chuyển từ HTML sang C#. Ký tự này (và cả cặp ngoặc { } nếu đó là 1 khối lệnh) luôn được Razor tô màu nền vàng để dễ nhận biết.
- Cách viết chú thích² (comment) phụ thuộc vào ngôn ngữ. Code HTML thì chú thích bằng <!-- -->. Code C# thì chú thích bằng // hoặc /* */. Ngoài ra, Razor cho phép chú thích bằng @* *@ ở bất cứ đâu.

¹ Xem Phụ lục 3: ASP.NET Core – Blazor

² Với Visual Studio/Visual Studio Code, có thể dùng phím tắt Ctrl + K, C và Ctrl + K, U để chú thích và hủy bỏ chú thích.

- Razor có khả năng phân biệt ký tự @ trong địa chỉ email và sẽ không chuyển sang chế độ code C#. Ngoài ra, nếu muốn biểu diễn ký tự @, cần viết là @@.
- Kết quả dịch mã cuối cùng của Razor luôn là HTML.

3. Đoạn lệnh Razor

Đoạn lệnh Razor (Razor code block) là 1 đoạn lệnh với ngôn ngữ mặc định là C#, gồm 1 hoặc nhiều câu lệnh nằm trong vùng `@{ ... }`. Đây là nơi dùng để khai báo biến, hàm cục bộ cũng như viết các câu lệnh, các cấu trúc điều khiển C#. Hạn chế của code block là không thể khai báo kiểu dữ liệu mới bằng `class`, `struct`, `enum`...

Trên cùng 1 trang Razor có thể có nhiều code block. Các biến và hàm cục bộ được khai báo và định nghĩa trong 1 code block có thể được dùng trong bất kỳ code block hay biểu thức Razor nào trên cùng trang đó. Biến chỉ được phép sử dụng sau khi đã khai báo, còn hàm cục bộ có thể được gọi trước cả khi cài đặt hàm.

Ví dụ: Đoạn lệnh Razor đơn giản:

```
@{
    int x = 10, y = 4;
    int sum = x + y;
}
```

Các cấu trúc điều khiển trong Razor cũng được viết tương tự như trong C#, và có thể không cần dùng đến cặp ngoặc `{ }` nếu đoạn lệnh không chứa câu lệnh khác ngoài cấu trúc điều khiển này:

- Cấu trúc rẽ nhánh if và if-else:

```
@if (điều_kiện) { ... }
@if (điều_kiện) { ... } else { ... }
```

- Cấu trúc rẽ nhánh switch:

```
@switch (biểu_thức) {
    case giá_trị: ...
}
```

- Vòng lặp for và foreach:

```
@for (int i = 0; i < n; i++) { ... }
@foreach (var item in list) { ... }
```

- Vòng lặp while và do-while:

```
@while (điều_kiện) { ... }
@do { ... } while (điều_kiện);
```

- Cấu trúc try-catch:

```
@try { ... } catch (exception) { ... } finally { ... }
```

4. Biểu thức Razor

Biểu thức Razor (Razor expression) là 1 biểu thức C# mang 1 giá trị nào đó (có thể là 1 biến, 1 phép tính, 1 hàm trả về giá trị...). Biểu thức này có thể được chèn vào code HTML bằng ký tự `@` (không kèm theo cặp ngoặc `{ }`).

Các biểu thức không chứa khoảng trắng (ví dụ như biến, lời gọi hàm) và không chứa các ký tự gây nhầm lẫn với code HTML xung quanh được gọi là biểu thức ngầm định (**implicit expression**) và chỉ cần được viết sau ký tự `@`. Razor View Engine sẽ tự nhận biết khi nào kết thúc biểu thức.

Các biểu thức có chứa khoảng trắng (ví dụ như các phép tính) được gọi là biểu thức tường minh (**explicit expression**) và phải được đặt trong vùng `@(...)`. Khi 1 biểu thức không thể viết dưới dạng implicit thì buộc phải viết dưới dạng explicit.

Ví dụ: Một số biểu thức Razor đơn giản:

```
<p>The value of x is @x and the value of y is @y</p>
<p>The sum of 5 and 9 is @(5 + 9)</p>
<p>The square root of 2 is @Math.Sqrt(2)</p>
<p>Current time: @DateTime.Now</p>
```

5. Hàm cục bộ

Các hàm được viết trong 1 trang Razor được gọi là **hàm cục bộ (local function)**. Hàm cục bộ trong Razor được chia thành 2 loại:

a) Hàm trả về giá trị

Hàm này được viết tương tự như 1 hàm trả về giá trị trong C#. Hàm này có thể được gọi như 1 biểu thức (tức là chỉ cần ký tự `@`).

Ví dụ: Hàm tính tổng 2 số nguyên:

```
@{
    int x = 10, y = 4;
}
<p>The sum of x and y: @Add(x, y)</p>
@{
    int Add(int a, int b) { return a + b; }
}
```

b) Hàm không trả về giá trị

Hàm này có kiểu trả về là **void**, dùng để hiển thị 1 đoạn code HTML. Hàm này không thể được gọi như 1 biểu thức mà phải được gọi trong 1 code block (tức là phải dùng cú pháp `@{ ... }`).

Ví dụ: Hàm hiển thị thông tin của 1 sinh viên:

```
@{  
    void ShowStudentInfo(Student s)  
    {  
        <div class="user-info">  
            <p class="font-weight-bold">Student's name: @s.Name</p>  
            <p>GPA: @s.GPA</span>  
        </div>  
    }  
}  
<p>Gọi hàm:</p>  
@{ ShowStudentInfo(student); }
```

6. Chuyển đổi ngôn ngữ trong code block

Trong Razor, ngôn ngữ mặc định ở ngoài code block là HTML, ở trong code block là C#. Tuy nhiên, Razor vẫn cho phép chuyển đổi ngôn ngữ trong code block sang HTML (ví dụ khi muốn hiển thị 1 thẻ HTML khi đang ở giữa đoạn code C#).

Có 3 cách chuyển đổi ngôn ngữ trong code block:

a) Implicit transition

Với cách này, chỉ cần viết thẻ HTML, Razor View Engine sẽ tự hiểu từ đó trở đi là code HTML (cho đến khi gặp thẻ đóng tương ứng). Nhược điểm là cần phải có 1 thẻ HTML đi kèm với nội dung cần hiển thị.

Ví dụ:

```
@{  
    var username = "admin";  
    <h3>Good morning, <strong>@username</strong>. Nice day!</h3>  
}
```

b) Explicit delimited transition

Với cách này, cần sử dụng thẻ `<text></text>`. Đây không phải là thẻ HTML mà là 1 thẻ Razor, dùng để cho Razor View Engine biết là cần chuyển sang HTML. Chỉ có phần nội dung nằm giữa thẻ `<text></text>` được chuyển sang HTML (bất kể là bao nhiêu dòng), do đó kết quả hiển thị sẽ không cần phải đi kèm với 1 thẻ HTML như cách trên:

Ví dụ:

```
@{  
    var username = "admin";  
    <text>Good morning, @username.  
    Nice day!</text>  
}
```

c) Explicit line transition:

Dùng cú pháp `@:` để chuyển đổi phần nội dung từ đó đến hết dòng hiện tại sang HTML.

Ví dụ:

```
@{  
    var username = "admin";  
    @:Good morning, @username.  
}
```

V. Directive

Directive là 1 câu lệnh thường đặt ở đầu 1 trang Razor, dùng để quy định một số tùy chọn, thiết lập, và chỉ có tác dụng trên trang đó. Nếu muốn directive có tác dụng trên tất cả các view trong project hiện tại, cần viết directive vào file `_ViewImports.cshtml`.

Một số directive thường gặp:

- `@model`: Quy định ViewModel cho trang hiện tại.
- `@using`: Dùng để đưa **namespace** cần dùng vào trang web. Ví dụ: Nếu muốn sử dụng class `File`, cần viết directive `@using System.IO`.
- `@namespace`: Dùng để quy định namespace cho trang hiện tại.
- `@section`: Dùng để cài đặt nội dung cho section của layout (đã trình bày ở phần 1.2 ở trên).
- `@addTagHelper`: Dùng để thêm Tag Helper vào trang web.

File `_ViewImports.cshtml` mặc định có nội dung như sau:

```
@using Demo  
@using Demo.Models  
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```