

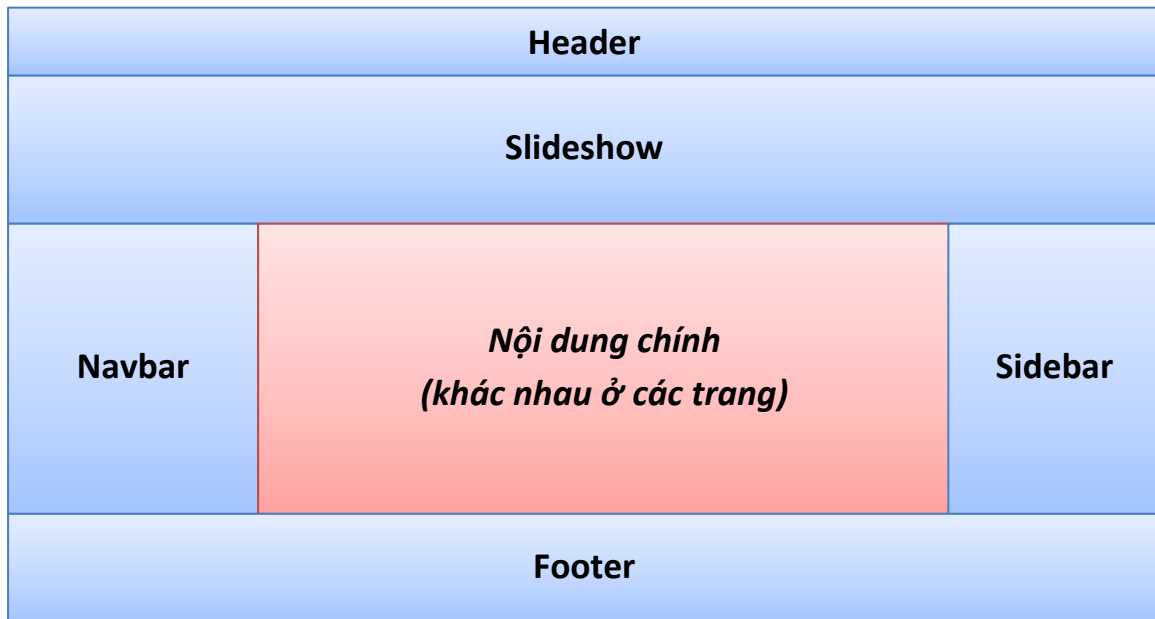
Buổi 2

Layout View, Tag Helper, Razor

I. Layout View trong ASP.NET Core MVC

1. Khái niệm

Để đảm bảo tính nhất quán, một ứng dụng web thông thường sẽ có 1 phần 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:



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*¹ 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, trong đó 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`.

Ví dụ: File `Views/Shared/_Layout.cshtml` của project Eshop:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    .....
  </head>
  <body>
    <header>
      .....
    </header>
```

¹ Từ nay gọi tắt là layout

```

<div class="container">
    <main role="main" class="pb-3">
        @RenderBody()
    </main>
</div>

<footer class="border-top footer text-muted">
    .....
</footer>
.....
@RenderSection("Scripts", required: false)
</body>
</html>

```

File `_ViewStart.cshtml` trong thư mục `Views` quy định layout mặc định cho các view, áp dụng cho tất cả các view ngang hàng hoặc thấp hơn file này. Tên layout có thể viết dưới dạng đường dẫn file (ví dụ: `~/Views/Shared/_Layout.cshtml`) hoặc dưới dạng tên ngắn gọn (ví dụ: `_Layout`):

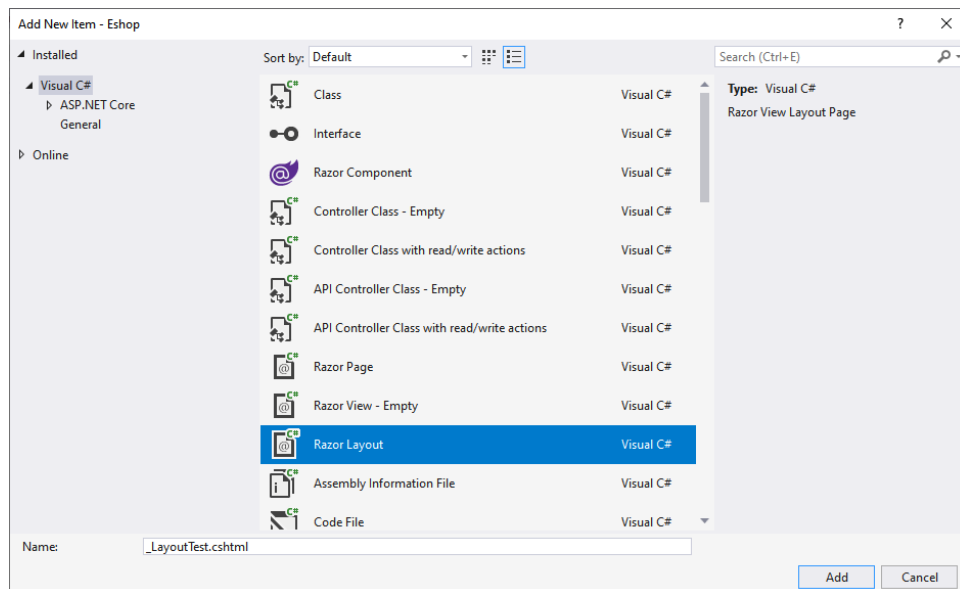
```

@{
    Layout = "_Layout";
}

```

Để tạo layout, chúng ta thực hiện các thao tác sau:

- Click chuột phải vào thư mục `Views/Shared` và chọn `Add → View...`
- Trong cửa sổ `Add New Scaffolded Item`, chọn `Razor View – Empty` và nhấn nút `Add`.
- Trong cửa sổ `Add New Item`, chọn `Razor Layout`, đặt tên cho layout (ví dụ: `_LayoutTest.cshtml`) và nhấn nút `Add`:



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

```

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

```

2. RenderBody() và RenderSection()

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

1 layout có thể chứa nhiều section. Phương thức `RenderSection()` nhận tham số là tên của section và 1 giá trị kiểu `bool` cho biết section này có bắt buộc phải có trong view hay không.

Ví dụ: Trong ví dụ trên, `_LayoutTest` có chứa 2 section được đặt tên là `Top` và `Bottom`, trong đó section `Top` bắt buộc phải có, còn section `Bottom` thì không.

3. 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... Layout mặc định cho các view được cài đặt trong file `_ViewStart.cshtml` (xem phần 1.1). Tuy nhiên, chúng ta có thể cài đặt cho view sử dụng 1 layout khác với mặc định bằng cách đặt câu lệnh sau vào đầu view:

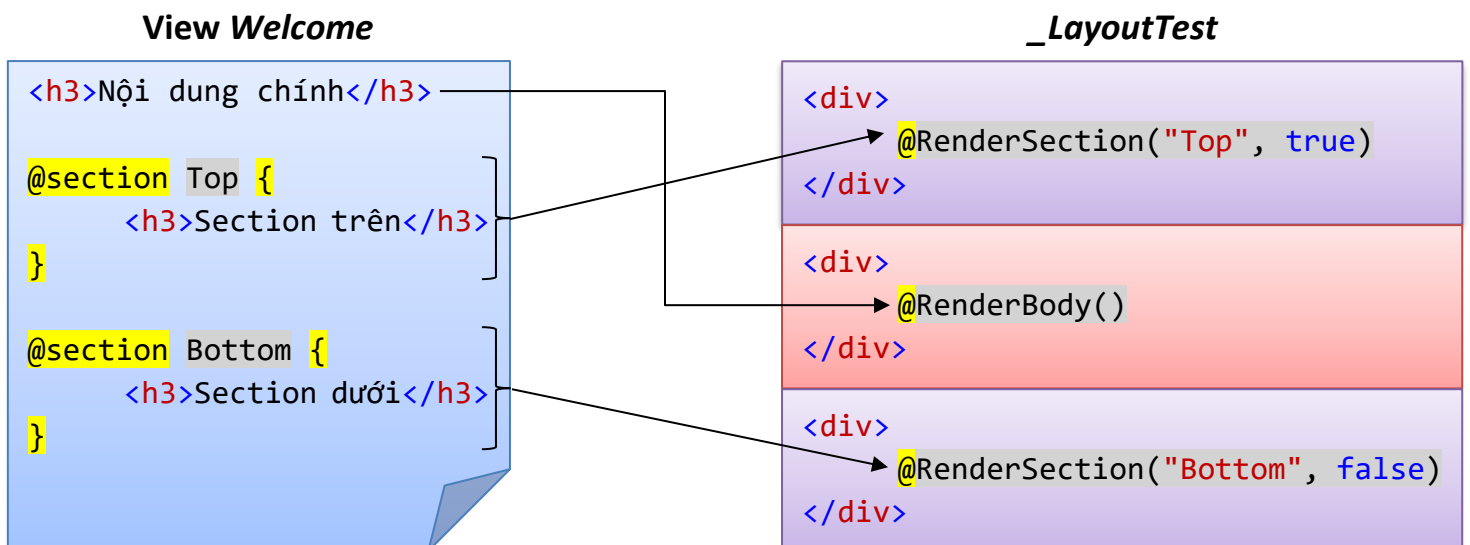
```
@{
    Layout = "tên Layout muốn dùng";
}
```

Giả sử có 1 view `Welcome` (tương ứng với action `Welcome` của controller `Accounts`). Trong view, nội dung của section phải được cài đặt bằng cú pháp Razor `@section` như sau:

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

<h3>Nội dung chính</h3>
@section Top {
    <h3>Section trên</h3>
}
@section Bottom {
    <h3>Section dưới</h3>
}
```

Các phần nội dung từ view `Welcome` sẽ được đặt vào `_LayoutTest` như sau:



Khi view `Welcome` được gọi (thông qua action `Welcome` của controller `Accounts`), các section của view sẽ được đặt vào nơi gọi các phương thức `@RenderSection()` tương ứng. Toàn bộ phần còn lại của view không thuộc section nào sẽ được đặt vào nơi gọi phương thức `@RenderBody()`.

Điểm khác nhau giữa `@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 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: string.Section có bắt buộc không: boolean.

Ví dụ: Xem ví dụ 2.1.

II. Partial View

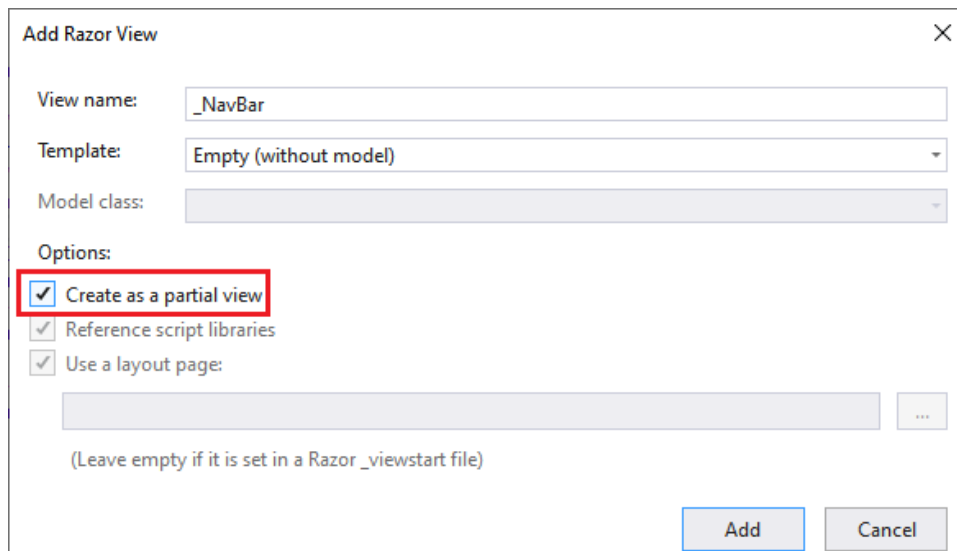
1. Khái niệm

Ở phần I chúng ta đã biết cách tạo giao diện chung cho nhiều trang web con bằng cách sử dụng *Layout View*, và 1 trang web có thể có nhiều layout. Tuy nhiên, các layout này cũng có thể có các phần giao diện chung, ví dụ: layout cho thành viên thường và layout cho admin có chung thanh *NavBar*. Để giảm bớt trùng lặp code khi có các thành phần giao diện chung giữa các view hoặc giữa các layout, chúng ta sử dụng *Partial View*.

Partial View là 1 phần của trang web có thể tái sử dụng được trong nhiều view/layout khác nhau, thậm chí sử dụng nhiều lần trong cùng 1 view/layout, do đó nó cũng nên được đặt trong thư mục *Views/Shared*. Bản thân Partial View được viết bởi code HTML và C#, do đó nó cũng được lưu trong file có phần mở rộng *.cshtml* tương tự như view và layout.

Để tạo 1 *Partial View*, chúng ta thực hiện các thao tác sau:

- Click chuột phải vào thư mục *Views/Shared* và chọn *Add* → *View...*
- Trong cửa sổ *Add New Scaffolded Item*, chọn *Razor View* và nhấn nút *Add*.
- Trong cửa sổ *Add Razor View*, check vào mục *Create as a partial view*, đặt tên cho Partial View ở mục *View name* (ví dụ: *_NavBar*) và nhấn nút *Add*:



Giả sử chúng ta sẽ cài đặt NavBar vào Partial View này. Để cho đơn giản, chúng ta sẽ sử dụng NavBar của layout mặc định: Mở file *Views/Shared/_Layout.cshtml* và copy toàn bộ thẻ `<nav>` vào file *Views/Shared/_NavBar.cshtml*.

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 được viết dưới dạng đường dẫn file (ví dụ: *~/Views/Shared/_NavBar.cshtml*) hoặc dưới dạng tên ngắn gọn (ví dụ: *_NavBar*).

Do đó, nội dung thẻ `<header>` của file *_Layout.cshtml* sẽ được viết lại như sau:

```
<header>
    <partial name="_NavBar" />
</header>
```

2. Mục đích

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, đồng thời các thành phần này cũng sẽ được tách riêng, dễ quản lý và thay đổi sau này.
- 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 bố cục cho các view khác nhau → Nên sử dụng Layout View.
- Đoạn code phức tạp hoặc cần xử lý logic nhiều → Nên sử dụng view.

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 HTML nhưng được xử lý bởi *Razor View Engine* 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*...).

Ví dụ: Tạo 1 thẻ `<a>` trỏ tới *Accounts/Index* (controller *Accounts* và action *Index*):

```
<a asp-controller="Accounts" asp-action="Index">Danh sách tài khoản</a>
```

→ Thẻ này sẽ được Razor Engine biên dịch trước khi gửi xuống trình duyệt như sau:

```
<a href="/Accounts/Index">Danh sách tài khoản</a>
```

Có rất nhiều Tag Helper hỗ trợ viết các thẻ HTML². Trong khuôn khổ tài liệu buổi 2, chúng ta chỉ tìm hiểu những Tag Helper thường dùng nhất.

1. Anchor và Form Tag Helper

Thẻ `<a>` và `<form>` có 1 số Tag Helper sau:

- `asp-controller`: Tên controller đang trỏ đến. Nếu không có thì xem như đang trỏ đến controller hiện tại.
- `asp-action`: Tên action của controller đang trỏ đến. Nếu không có thì xem như đang gọi action mặc định. Hiện tại, action mặc định của controller là *Index* (được quy định trong file *Startup.cs*).

² Xem thêm về Tag Helper: <https://docs.microsoft.com/en-us/aspnet/core/mvc/views/tag-helpers/built-in/?view=aspnetcore-5.0>

- `asp-route-x`: Giá trị tham số duy nhất của route, với `x` là tên tham số. Hiện tại, tham số mặc định của route là `id` (được quy định trong file *Startup.cs*).

Ví dụ:

- Tạo liên kết đến trang thêm tài khoản:
`<a asp-controller="Accounts" asp-action="Create">Thêm`
- Tạo liên kết đến trang xem chi tiết tài khoản có ID là 5:
`<a asp-controller="Accounts" asp-action="Detail" asp-route-id="5">Chi tiết`
- Tạo form submit đến route *Accounts/Create*:
`<form asp-controller="Accounts" asp-action="Create">...</form>`

Ngoài ra, thẻ `<a>` còn hỗ trợ 1 số Tag Helper nâng cao³ như `asp-route`, `asp-all-route-data`...

2. Input và Label Tag Helper

Thẻ `<input>`⁴ và thẻ `<label>`⁵ đều có Tag Helper `asp-for` giúp tạo control tương ứng với 1 thuộc tính nào đó của View Model.

Ví dụ: Xem ví dụ 2.2. Tạo trang đăng nhập theo các thao tác sau:

- Cài đặt action *Login* thuộc controller *Accounts*:

```
public IActionResult Login()
{
    return View();
}
```
- Tạo view *Login* thuộc thư mục *Accounts*.
- Thêm dòng sau vào đầu view, cho biết view này dùng model *Account* đã tạo trước đó:
`@model Eshop.Models.Account`
- Tạo form đăng nhập như sau:

```
<form asp-controller="Accounts" asp-action="Login" method="post">
  <div class="form-group">
    <label asp-for="Username"></label>
    <input asp-for="Username" class="form-control"/>
  </div>
  <div class="form-group">
    <label asp-for="Password"></label>
    <input asp-for="Password" class="form-control"/>
  </div>
  <div class="form-group">
    <button type="submit" class="btn btn-primary">
      Đăng nhập
    </button>
  </div>
</form>
```

³ Xem thêm về Tag Helper nâng cao: <https://docs.microsoft.com/en-us/aspnet/core/mvc/views/tag-helpers/built-in/anchor-tag-helper?view=aspnetcore-5.0>

⁴ Xem thêm về Tag Helper cho thẻ `<input>`: <https://docs.microsoft.com/en-us/aspnet/core/mvc/views/working-with-forms?view=aspnetcore-5.0#the-input-tag-helper>

⁵ Xem thêm về Tag Helper cho thẻ `<label>`: <https://docs.microsoft.com/en-us/aspnet/core/mvc/views/working-with-forms?view=aspnetcore-5.0#the-label-tag-helper>

Tag Helper `asp-for` của thẻ `<input>` sẽ dựa vào kiểu dữ liệu của thuộc tính tương ứng của model để tạo ra control phù hợp. Trong ví dụ trên, thẻ `<input asp-for="Username">` sẽ tạo ra textbox vì thuộc tính `Username` của model `Account` có kiểu `string`.

Tương tự Tag Helper `asp-for` của thẻ `<label>` sẽ dựa vào tên thuộc tính tương ứng của model để tạo ra tiêu đề của control. Mặc định, tiêu đề control sẽ là tên thuộc tính tương ứng.

Kết quả sau khi biên dịch chương trình và chạy với route `~/Accounts/Login`:

Username

Password

Đăng nhập

Các kiểu dữ liệu của .NET và các dạng control tương ứng:

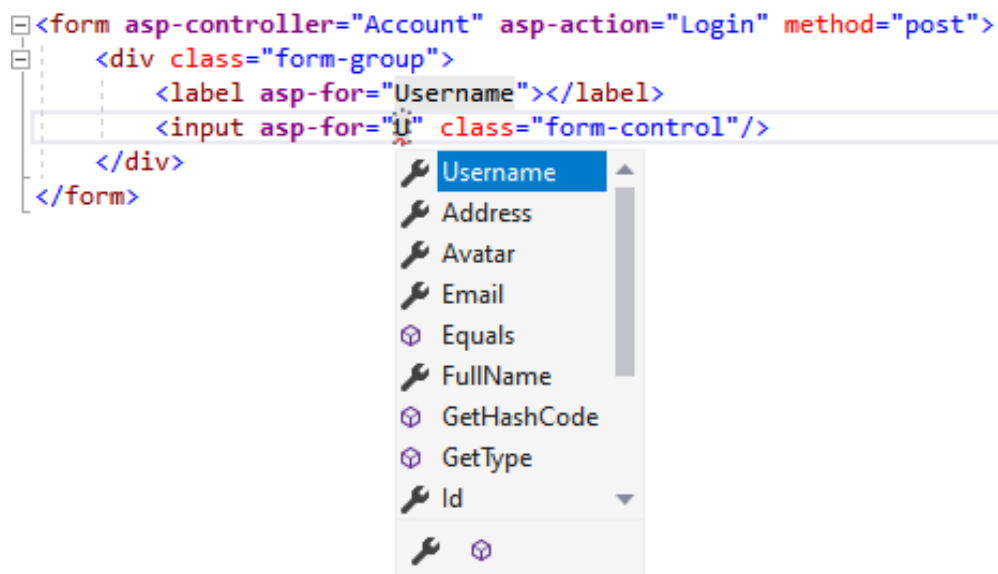
Kiểu dữ liệu .NET	Loại control	Input Type
Bool	Checkbox	<code>type="checkbox"</code>
String	Textbox	<code>type="text"</code>
DateTime	DateTimePicker	<code>type="datetime-local"</code>
Byte, Int, Single, Double	Numeric	<code>type="number"</code>

3. 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="Username" placeholder="Tên TK" class="form-control" />
```

Tag Helper cũng hỗ trợ cơ chế gợi ý IntelliSense của Visual Studio. Trong ví dụ trên, khi tạo form đăng nhập, IntelliSense hỗ trợ gợi ý dựa trên model đang sử dụng:



Ngoài ra, do Tag Helper giúp tạo form dựa trên model, nên khi kiểu dữ liệu của các thuộc tính của model thay đổi, form sẽ tự động thay đổi theo.

IV. Razor

1. Tổng quan về cú pháp Razor và Razor View Engine

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 để tạo ra mã nguồn HTML và hiển thị trên trình duyệt web. Nói cách khác, view engine có vai trò như chương trình dịch (compiler hoặc 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 Blazor⁶ với nhiệm vụ đánh dấu xem đâu là C#, đâu là HTML trong mã nguồn để view engine xử lý cho phù hợp. File mã nguồn Razor có phần mở rộng là *.cshtml*.

Ví dụ: Một đoạn mã nguồn 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) và đoạn code HTML (không có màu nền) 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 mã nguồn 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.
- 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 diễn tả ký tự @, cần viết là @@.
- Cách ghi chú thích⁷ (comment) phụ thuộc vào ngôn ngữ. Khi ở phần code HTML thì chú thích bằng <!-- -->, khi ở phần code C# thì chú thích bằng // hoặc /* */. Ngoài ra, Razor cho phép chú thích bằng @* *@ ở bất cứ đâu.
- Kết quả dịch mã cuối cùng của Razor luôn là HTML.

⁶ Xem Phụ lục #1: Blazor

⁷ Với Visual Studio/Visual Studio Code, có thể dùng phím tắt Ctrl + K, C / Ctrl + K, U để chú thích/hủy bỏ chú thích.

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 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: Một đoạn lệnh Razor đơn giản:

```
@{  
    int x = 10, y = 4;  
    int sum = x + y;  
}  
<p>The sum of x and y: @sum</p>
```

Các cấu trúc điều khiển trong Razor cũng được viết tương tự như trong C#, và có thể không cần dùng đến cặp ngoặc `{ }`:

- Cấu trúc rẽ nhánh `if` và `if-else`:

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

- Cấu trúc rẽ nhánh `switch`:

```
@switch (expression) {  
    case value: ...  
}
```

- Vòng lặp `for` và `foreach`:

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

- Vòng lặp `while` và `do-while`:

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

- 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 `{ }`).

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>
```

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 *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 *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.

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 tài khoản:

```
@{  
    void ShowAccountInfo(Account acc)  
    {  
        <div class="user-info">  
            <span class="font-weight-bold">@acc.FullName</span> -  
            <span>@acc.Address</span>  
        </div>  
    }  
}  
<p>Gọi hàm:</p>  
@{ ShowAccountInfo(acc); }
```

6. Chuyển đổi ngôn ngữ trong code block

Trong file 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 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 xuất nội dung kèm với 1 thẻ HTML.

Ví dụ:

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

c) Explicit line transition:

Với cách này, có thể sử dụng cú pháp @: để chuyển đổi 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 trang web trong project hiện tại, cần viết directive vào file *_ViewImports.cshtml*.

Một số directive thường gặp:

- **@model**: Quy định class được dùng để làm model cho trang web MVC. Directive này là bắt buộc khi trang hiện tại có dùng View Model.
- **@using**: Tương tự using của C#, dùng để đưa namespace của các class cần dùng vào trang web. Ví dụ: Nếu muốn sử dụng class *File*, cần viết directive **@using System.IO**.
- **@namespace**: Tương tự namespace của C#, dùng để quy định namespace cho trang web hiện tại.
- **@section**: Dùng để cài đặt nội dung section của layout (đã trình bày ở phần 1.3 ở trên).
- **@addTagHelper**: Dùng để thêm Tag Helper vào trang web.

VI. Bài tập

Dựa vào giao diện mẫu ở file *Template.rar*, xây dựng Layout View cho ứng dụng web. Khuyến khích chia nhỏ các thành phần giao diện thành các *Partial View*.

Tạo các model sau:

- Sản phẩm: gồm có tên sản phẩm, mô tả, giá tiền, hình ảnh.
- Tài khoản: gồm có tên đăng nhập, mật khẩu, email, SĐT, ngày sinh, trạng thái.

Sau đó, cài đặt các view sau:

1. Danh sách sản phẩm dưới dạng Grid (ít nhất 4 sản phẩm) với thông tin do SV tự chọn.
2. Thêm tài khoản: dựa vào model Tài khoản ở trên cùng với Tag Helper.
3. Danh sách tài khoản dưới dạng Table (ít nhất 4 tài khoản) với thông tin do SV tự chọn.