ASP Razor

Razor is the name of the MVC Framework view engine

A ***view engine*** processes content and looks for instructions, typically to insert dynamic content into the output sent to the browser

How Razor Works

- Razor converts CSHTML files into C# classes
- Compiles them
- And creates new instances each time a view is required to generate a result

The Razor Base Class

```
public class ASPV_Views_Home_Index_cshtml : RazorPage<string[]>
```

- The RazorPage class provides methods and properties that can be used in CSHTML files to access MVC features

Name	Description
Model	Returns the model data provided by the action method
ViewData	Returns a ViewDataDictionary object that provides access to other view data features
Layout	This property is used to specify a layout
ViewBag	Provides access to the view bag object
TempData	Provides access to the temp data
Context	Returns an HttpContext object that describes the current request and the response that is being prepared
User	Returns the profile of the user associated with this request
ViewContext	Returns a ViewContext object
RenderSection()	Is used to insert a section of content from the view into a layout
RenderBody()	Inserts all the content in a view that is not contained in a section into a layout
IsSectionDefined()	Is used to determine whether a view defines a section

Adding Dynamic Content to a Razor View

Technique	When to Use
Inline code	Use for small, self-contained pieces of view logic, such as if and foreach statements
Tag helpers	Used to generate attributes on HTML elements
Sections	For creating sections of content that will be inserted into layout at specific locations
Partial views	For sharing subsections of view markup between views. Cannot be used to perform business logic
View components	For creating reusable UI controls or widgets that need to contain business logic

Razor and C#

- Razor refers to the small set of conventions for how you embed C# code into a page
- For example, the convention of using **@** to mark code in the page and using **@{ }** to embed a code block is the Razor aspect of a page
  - Tag Helpers and Html Helpers are also considered to be part of Razor
- Razor syntax is used in both MVC view files and ASP.NET Razor Pages

- But you should not use Razor to perform business logic or manipulate your domain model objects in any way!

Implicit Razor expressions

- Start with @ followed by C# code
- Must not contain spaces
  - Unless the C# statement has a clear ending

Renders as you would expect:

```
<p>@DateTime.Now</p>
<p>@DateTime.IsLeapYear(2016)</p>
<p>@await DoSomething("hello", "world")</p>
```

Does not renders as you might expect because of the space:

```
<p>Last week: @DateTime.Now - TimeSpan.FromDays(7)</p>
```

$\Downarrow$

```
<p>Last week: 7/7/2016 4:39:52 PM - TimeSpan.FromDays(7)</p>
```

Explicit Razor expressions

- Explicit Razor expressions consist of an @ symbol with balanced parenthesis

```
<p>Last week: @(DateTime.Now - TimeSpan.FromDays(7))</p>
```

- You can use an explicit expression to concatenate text with an expression result:

```
@{
    var joe = new Person("Joe", 33);
}
<p>Age@(joe.Age)</p>
```

```
<p>Age33</p>
```

```
@{
    var joe = new Person("Joe", 33);
}
<p>Age@joe.Age</p>
```

```
<p>Age@joe.Age</p>
```

Razor interprets it as an email address

Expression encoding

- C# expressions that evaluate to a string are HTML encoded
- C# expressions that evaluate to IHtmlContent are rendered directly through IHtmlContent.WriteTo
- C# expressions that don't evaluate to a string or IHtmlContent are converted to a string by ToString and encoded before they're rendered

cshtml:

```
@("<span>Hello World</span>")
```

The generated html:

```
&lt;span&gt;Hello World&lt;/span&gt;
```

Rendered in the browser:

```
<span>Hello World</span>
```

AARHUS UNIVERSITY
SCHOOL OF ENGINEERING

No expression encoding

- When you don’t want your output encoded but rendered as HTML markup you can use HtmlHelper.Raw

```
@Html.Raw("<span>Hello World</span>")
```

⇩

```
<span>Hello World</span>
```

- **Warning**
  - Using HtmlHelper.Raw on unsanitized user input is a security risk!
  - User input might contain malicious JavaScript or other exploits
  - **Sanitizing user input is difficult so avoid using HtmlHelper.Raw with user input**

Implicit transitions

- The default language in a code block is C#, but you can transition back to HTML:

```
@{
    var inCSharp = true;
    <p>Now in HTML, was in C# @inCSharp</p>
}
```

Explicit delimited transition

- Surround the characters for rendering with the Razor <text> tag

```
@for (var i = 0; i < people.Length; i++)
{
    var person = people[i];
    <text>Name: @person.Name</text>
}
```

- Without an HTML or Razor tag, you receive a Razor runtime error
- To render the rest of an entire line as HTML inside a code block, use @:

```
@for (var i = 0; i < people.Length; i++)
{
    var person = people[i];
    @:Name: @person.Name
}
```

@functions

- The @functions directive enables you to add function-level content to a view

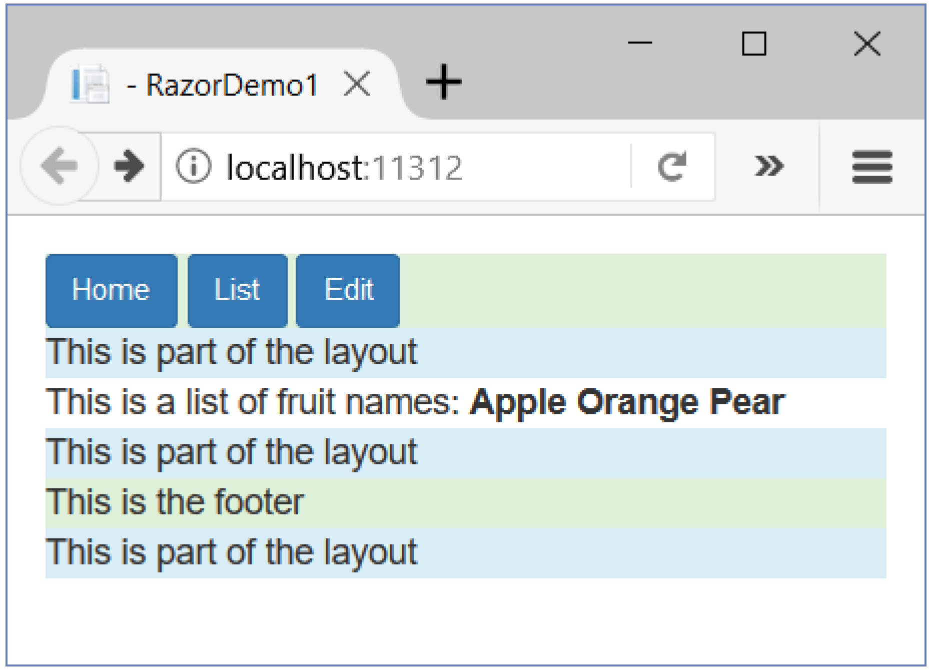
```
@functions {
    public string GetHello()
    {
        return "Hello";
    }
}

<div>From method: @GetHello()</div>
```

The generated Razor C# class

```
public class _Views_Home_Test_cshtml :
RazorPage<dynamic>
{
    // Functions placed between here
    public string GetHello()
    {
        return "Hello";
    }
    // And here.
public override async Task ExecuteAsync()
    {
        WriteLiteral("\r\n<div>From method: ");
        Write(GetHello());
        WriteLiteral("</div>\r\n");
    }
```

Sections

- Sections are defined in the view but applied in a layout with the **@RenderSection** expression


- RazorDemo1

localhost:11312

Home List Edit

This is part of the layout

This is a list of fruit names: **Apple Orange Pear**

This is part of the layout

This is the footer

This is part of the layout

_Layout.cshtml

```
<!DOCTYPE html>
<html>
<head>
<meta name="viewport" content="width=device-width" />
<title>@ViewBag.Title</title>
<link asp-href-include="lib/bootstrap/dist/css/*.min.css" rel="stylesheet" />
</head>
<body class="panel-body">
@RenderSection("Header")
<div class="bg-info">
This is part of the layout
</div>
@RenderBody()
<div class="bg-info">
This is part of the layout
</div>
@RenderSection("Footer")
<div class="bg-info">
This is part of the layout
</div>
</body>
</html>
```

```
@section Header {
<div class="bg-success">
@foreach (string str in new [] {"Home'
<a class="btn btn-sm btn-primary" asp
action="str">@str</a>
}
</div>
}
This is a list of fruit names:
@foreach (string name in Model) {
<span><b>@name</b></span>
}
@section Footer {
<div class="bg-success">
This is the footer
</div>
}
```

Optional Sections

- By default, a view has to contain all the sections for which there are RenderSection calls in the layout
- But you may define optional sections
  - which you do by passing an additional false argument to the RenderSection method

```
@RenderSection("scripts", false )
```

Partial Views

- Partial views are just regular CSHTML files

MyPartial.cshtml

```
<div class="bg-info">
<div>This is the message from the partial view.</div>
<a asp-action="Index">This is a link to the Index action</a>
</div>
```

List.cshtml

```
<body class="panel-body">
    This is the List View
    @Html.Partial("MyPartial")
</body>
```

Strongly Typed Partial Views

- The view model type is defined using the standard @model expression

MyStronglyTypedPartial.cshtml

```
@model IEnumerable<string>
<div class="bg-info">
  This is the message from the partial view.
  <ul>
    @foreach (string str in Model) {
      <li>@str</li>
    }
  </ul>
</div>
```

```
@model string[]
@{ Layout = null; }
<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-widt
  <title>Razor</title>
  <link asp-href-include="lib/bootstrap/dist/css/*
rel="stylesheet" />
</head>
<body class="panel-body">
  This is the List View
  @Html.Partial("MyStronglyTypedPartial", Model)
</body>
</html>
```

AARHUS UNIVERSITY
SCHOOL OF ENGINEERING

References & Links

- https://docs.microsoft.com/en-us/aspnet/core/mvc/views/razor