# What is Razor?

Razor is a markup syntax for embedding server based code into web pages. The Razor syntax consists of Razor markup, C# and HTML. Files containing Razor code are typically dictated by a .cshtml file extension.

# Rendering HTML

In Razor, the default language is HTML. Therefore, rendering HTML is no different than in an HTML file[[1]](#footnote-1).

<p>Hello World</p>

Will result in <p>Hello World</p> being rendered in the final HTML output.

# Rendering Server Code

Razors supported server code language is C#. As stated in the [previous section](#_Rendering_HTML) the default language for Razor is HTML; this means in order to render C# output we must provide a transition to tell Razor we want to move out of the HTML syntax. Razor uses the @ symbol to dictate transitions. When transitioning from HTML Razor can transition into C# or it can transition into Razor specific markup. When an @ symbol is followed by a Razor [reserved keyword](TODO, LINK DOWN) it transitions into Razor specific markup, in all other circumstances it transitions into plain C#[[2]](#footnote-2). To render C# to the output you will typically make use of [implicit](#_Implicit_Expressions) and [explicit](#_Explicit_Expressions) expressions.

## Implicit Expressions

An implicit Razor expression consists of an @ symbol followed by a piece of C# code. To render the current date time within a tag you could do the following:

<p>@DateTime.Now</p>

This also works when rendering the output of methods:

<p>@DateTime.IsLeapYear(2016)</p>

This does not work when trying to render more complex logic:

<p>Last week this time (doesn't work): @DateTime.Now - TimeSpan.FromDays(7)</p>

Note that the ending TimeSpan.FromDays(7) is not highlighted because it’s being treated as HTML. With the exception of the C# await keyword implicit expressions must not contain spaces unles dictated by the calling of a method. For example we can intermingle spaces as long as the C# statement has a clear ending:

<p>@await DoSomething("hello", "world")</p>

## Explicit Expressions

An explicit Razor expression consists of an @ symbol with balanced parenthesis. To render last weeks’ time with an explicit expression you could do:

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

Any content within the yellow parenthesis is evaluated and rendered to the output.

## Expression Encoding

In both [implicit](#_Implicit_Expressions) and [explicit](#_Explicit_Expressions) expressions non-[IHtmlContent](TODO, LINK TO DOCS) content rendered is always HTML encoded. For instance, if we were to try and render HTML tags with an explicit expression it would render as encoded. Example:

@("<div>Hello World</div>") => &lt;div&gt;Hello World&lt;/div&gt;

To render HTML content without encoding you can utilize the Html.Raw method:

@Html.Raw("<div>Hello World</div>") => <div>Hello World</div>

# Code Blocks

Code blocks are how you write blocks of C# code in Razor; it includes one or more C# statements and is enclosed in braces. Unlike expressions C# code written in code blocks is not rendered to the output. The simplest form of code blocks is the @{...} syntax. For instance:

@{

var output = "Hello World";

}

<p>The rendered result: @output</p>

Would render:

<p>The rendered result: Hello World</p>

## Rendering markup in code blocks

In a code block the default language is C#; however, there are ways to transition back to rendering HTML.

### Implicit Transition

Upon typing HTML within a code block Razor will transition back into rendering HTML:

@{

var imCSharp = true;

<p>Now in HTML</p>

### }

### Explicit Delimited Transition

To define a sub-section of a code block that should render HTML surround the text to be rendered in <text> tags:

@{

/\* C# \*/<text>I'm HTML</text>/\* C# \*/

}

This approach is typically performed when you want to render HTML but don’t want to surround it with an HTML tag.

### Explicit Line Transition

To render an entire line inside of a code block, utilize the @: characters:

@{

/\* Still C# \*/@: <p>Hello World</p> /\* This is not C#, it's HTML \*/

## }

## Control Structures

Control structures are an extension of code blocks. All aspects of code blocks (transitioning to markup, inline C#) also apply to the following structures

### Conditionals @if, else if, else and @switch

The @if family of statements enable you to control when code runs:

@if (value % 2 == 0)

{

<p>The value was even</p>

}

Now if you were to add an else or an else if statement you do not provide an additional @ symbol:

@if (value % 2 == 0)

{

<p>The value was even</p>

}

else if (value == 1337)

{

<p>You are very leet.</p>

}

else

{

<p>The value was not leet and is odd.</p>

}

We can also use @switch statements:

@switch (value)

{

case 1:

<p>The value is 1!</p>

break;

case 1337:

<p>Your number is leet!</p>

break;

default:

<p>Your number was not 1 or leet.</p>

break;

}

### Looping @for, @foreach, @while

We can render templated HTML with looping control statements. For instance, if we wanted to render a list of people:

@{

var people = new Person[]

{

new Person("John", 33),

new Person("Doe", 41),

};

}

We could utilize the @for:

@for (var i = 0; i < people.Length; i++)

{

var person = people[i];

<p>Name: @person.Name</p>

<p>Age: @person.Age</p>

}

The @foreach:

@foreach (var person in people)

{

<p>Name: @person.Name</p>

<p>Age: @person.Age</p>

}

The @while

@{ var i = 0; }

@while (i < people.Length)

{

var person = people[i];

<p>Name: @person.Name</p>

<p>Age: @person.Age</p>

i++;

}

And finally the @do while

@{ var i = 0; }

@do

{

var person = people[i];

<p>Name: @person.Name</p>

<p>Age: @person.Age</p>

i++;

} while (i < people.Length)

### Compound @using

Compound using statements can be used to represent scoping. For instance, we can utilize [HTML helpers](LINK TO HTML HELPERS) to render a form tag with the @using statement:

@using (Html.BeginForm())

{

// Form content would go here

}

Note that you can also perform scope level actions like the above with [TagHelpers](LINK TO TAGHELPERS DOC).

### @try, catch, finally

To control exception handling you can utilize try-catch statements just like you would in C#:

@try

{

throw new InvalidOperationException("You did something invalid");

}

catch (Exception ex)

{

<p>The exception message: @ex.Message</p>

}

finally

{

// Do something

}

### @lock

Razor has the capability to protect critical sections with lock statements:

@lock (SomeLock)

{

// Do critical section work

}

# Comments

Since Razor supports both HTML and C# syntax it enables you to write comments for either or in their corresponding language. So C# and HTML comments work as they normally would. For instance:

@{

/\* Valid C# Comment \*/

}

<!-- Valid HTML comment -->

However, what happens if we wanted to comment out the entire above code segment? We can accomplish this with Razor’s commenting syntax @\*...\*@:

@\*

@{

/\* Valid C# Comment \*/

}

<!-- Valid HTML comment -->

\*@

# Directives

Razor directives are represented by implicit expressions with reserved keywords following the @ symbol. A directive will typically change the way a page is parsed or enable different functionality within your Razor page. At its core, a Razor page is just a generated C# file. A super simple example of what a Razor pages input generates behind the scenes is:

@{

var output = "Hello World";

}

<div>Output: @output</div>

Generates:

public class MyRazorPage : RazorPage<dynamic>

{

public override Task ExecuteAsync()

{

var output = "Hello World";

WriteLiteral("<div>Output: ");

Write(output);

WriteLiteral("</div>");

return Task.CompletedTask;

}

}

Understanding a small piece of how Razor works under the covers will make understanding the following directives easier.

## @using

The using directive will allow you to generate using statements in your Razor page. The directive can be used like any other C# using statement. For instance, doing:

@using System.Collections.Generic

Will end up rendering the using statement for your Razor page class:

using System.Collections.Generic;

By adding a using statement such as this one you’d gain the ability to use types from System.Collections.Generic without fully qualifying them.

## @model

The model directive allows you to specify the type of model your Razor page should use. Its syntax is @model <TypeNameOfModel>. In the class example provided in the [introduction to Directives](#_Directives) the class generated inherits from RazorPage<dynamic>. By adding an @model you control what’s inherited. Therefore, if you did:

@model LoginViewModel

Your Razor page would inherit from RazorPage<LoginViewModel>. This enables you to access the model in your Razor page via the Model property:

<div>The Login Email: @Model.Email</div>

Of course this is all dependent on you [passing in a model from your controller](LINK TO CONTROLLER ARTICLE).

## @inherits

The inherits directive is a lot like the model directive but it gives you full control of what class your Razor page inherits from. Its syntax is @inherits <TypeNameOfClassToInheritFrom>. For instance, let’s say we had the following custom Razor page type:

public abstract class CustomRazorPage : RazorPage

{

public string CustomText { get; } = "Hello World";

}

This would enable us to write the following Razor code:

@inherits CustomRazorPage

<div>Custom text: @CustomText</div>

This is the simplest way to utilize @inherits; however, we can also change this slightly to enable use of the previously mentioned [@model directive](#_@model). To do this, change your custom type to inherit from the generic Razor page like follows:

public abstract class CustomRazorPage<TModel> : RazorPage<TModel>

{

public string CustomText { get; } = "Hello World";

}

Now instead of having an @model and an @inherits statement in our Razor page we’d do the following:

@inherits CustomRazorPage<LoginViewModel>

<div>The Login Email: @Model.Email</div>

The reason we need to take this approach is because @model and @inherits cannot exist on the same page. You can however have an @model in a [\_ViewImports.cshtml file](LINK TO DOCS ON VIEWIMPORTS) above your Razor view. In that case your inherits would look like the following:

@inherits CustomRazorPage<TModel>

## @inject

FEEL FREE TO IGNORE ALL THIS AND LINK TO ANOTHER ARTICLE IF ONE IS WRITTEN. The inject directive enables you to inject a service from your [service container](LINK TO DI ARTICLE) into your Razor page for use. Its syntax is @inject <ServiceTypeName> <PropertyName>. Example:

@using Microsoft.AspNetCore.Hosting

@inject IHostingEnvironment HostingEnvironment

<div>The Application Name: @HostingEnvironment.ApplicationName</div>

Much like [@inherits](#_@inherits) you can also provide the TModel token if your service happens to depend on the current model type:

@inject IHtmlHelper<TModel> CustomHtmlHelper

@CustomHtmlHelper.Raw("<div>Hello World</div>")

An additional feature of @inject is that it also enables you to replace a few properties that are automatically injected into a Razor page. For instance, we could replace the Html property on our Razor page with the hosting environment:

@using Microsoft.AspNetCore.Hosting

@inject IHostingEnvironment Html

<div>The Application Name: @Html.ApplicationName</div>

If you inject a property that already exists on your Razor page and has been @injected before, or is one of the following properties:

* Html
* Json
* Component
* Url
* ModelExpressionProvider

Then your @inject statement will override the existence of that property.

## @functions

The functions directive enables you to add function level content to your Razor page. Its syntax is @functions { <C# Code> }. So by typing:

@functions {

public string GetHello()

{

return "Hello";

}

}

Your generated Razor page looks like:

public class MyRazorPage : RazorPage<dynamic>

{

// Functions places you between here

public string GetHello()

{

return "Hello";

}

// And here

public override Task ExecuteAsync()

{

// Generated code

return Task.CompletedTask;

}

}

This means you can then write:

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

## @section

The @section directive should be used in conjunction with an [layout page](LINK TO LAYOUT TUTORIAL) to enable views to render content in different parts of a final HTML page. Its syntax is @section <SectionName> { <Razor Code> }. For example:

@section Scripts {

<script src="~/js/site.js"></script>

}

This content would be rendered where the RenderSection(“Scripts”) method was located in the corresponding layout page.

## @addTagHelper

The addTagHelper directive is gone over extensively in the [Razor TagHelper article](LINK TO ADDTAGHELPER PORTION).

## @removeTagHelper

The removeTagHelper directive is gone over extensively in the [Razor TagHelper article](LINK TO REMOVETAGHELPER PORTION).

## @tagHelperPrefix

The tagHelperPrefix directive is gone over extensively in the [Razor TagHelper article](LINK TO TAGHELPERPREFIX PORTION).

# TagHelpers

TagHelpers are new in ASP.NET Core Razor 1.0 and are gone over extensively [here](LINK TO TAGHELPER ARTICLE).

THINGS STILL TO BE COVERED:

- Misc

o Expressions in HTML element attributes ( <input checked=”@true” /> )

o Markup templates (@<p>…</p>)

o Application Relative Path handling (~/)

1. Razor’s default mechanism for transitioning to C# is done by using the @ symbol. Therefore, HTML containing @ symbols may need to be escaped with a second @ symbol. Example: <p>@@Username</p> would render <p>@Username</p> in the final output. [↑](#footnote-ref-1)
2. HTML attributes containing email addresses don’t treat the @ symbol as a transition character. [↑](#footnote-ref-2)