Razor syntax reference for ASP.NET Core

02/12/2020 • 18 minutes to read • +14

In this article

By Rick Anderson, Taylor Mullen, and Dan Vicarel

Razor is a markup syntax for embedding server-based code into webpages. The Razor syntax consists of Razor markup, C#, and HTML. Files containing Razor generally have a *.cshtml* file extension. Razor is also found in Razor components files (*.razor*).

Rendering HTML

The default Razor language is HTML. Rendering HTML from Razor markup is no different than rendering HTML from an HTML file. HTML markup in *.cshtml* Razor files is rendered by the server unchanged.

Razor syntax

Razor supports C# and uses the `@` symbol to transition from HTML to C#. Razor evaluates C# expressions and renders them in the HTML output.

When an `@` symbol is followed by a Razor reserved keyword, it transitions into Razor-specific markup. Otherwise, it transitions into plain C#.

To escape an `@` symbol in Razor markup, use a second `@` symbol:

CSHTML Copy

```
<p>@@Username</p>
```

The code is rendered in HTML with a single `@` symbol:

HTML Copy

```
<p>@Username</p>
```

HTML attributes and content containing email addresses don't treat the `@` symbol as a transition character. The email addresses in the following example are untouched by Razor parsing:

CSHTML Copy

```
<a href="mailto:Support@contoso.com">Support@contoso.com</a>
```

Implicit Razor expressions

Implicit Razor expressions start with `@` followed by C# code:

CSHTML Copy

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

With the exception of the C# `await` keyword, implicit expressions must not contain spaces. If the C# statement has a clear ending, spaces can be intermingled:

CSHTML Copy

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

Implicit expressions **cannot** contain C# generics, as the characters inside the brackets (`<>`) are interpreted as an HTML tag. The following code is **not** valid:

CSHTML Copy

```
<p>@GenericMethod<int>()</p>
```

The preceding code generates a compiler error similar to one of the following:

- The "int" element wasn't closed. All elements must be either self-closing or have a matching end tag.
- Cannot convert method group 'GenericMethod' to non-delegate type 'object'. Did you intend to invoke the method?`

Generic method calls must be wrapped in an explicit Razor expression or a Razor code block.

Explicit Razor expressions

Explicit Razor expressions consist of an `@` symbol with balanced parenthesis. To render last week's time, the following Razor markup is used:

CSHTML Copy

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

Any content within the `@()` parenthesis is evaluated and rendered to the output.

Implicit expressions, described in the previous section, generally can't contain spaces. In the following code, one week isn't subtracted from the current time:

CSHTML Copy

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

The code renders the following HTML:

HTML Copy

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

Explicit expressions can be used to concatenate text with an expression result:

CSHTML Copy

```
@{
    var joe = new Person("Joe", 33);
}

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

Without the explicit expression, `<p>Age@joe.Age</p>` is treated as an email address, and `<p>Age@joe.Age</p>` is rendered. When written as an explicit expression, `<p>Age33</p>` is rendered.

Explicit expressions can be used to render output from generic methods in *.cshtml* files. The following markup shows how to correct the error shown earlier caused by the brackets of a C# generic. The code is written as an explicit expression:

CSHTML Copy

```
<p>@(GenericMethod<int>())</p>
```

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 `IHtmlContent` are converted to a string by `ToString` and encoded before they're rendered.

CSHTML Copy

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

The preceding code renders the following HTML:

HTML Copy

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

The HTML is shown in the browser as plain text:

Hello World

`HtmlHelper.Raw` output isn't encoded but rendered as HTML markup.

> **⚠ 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. Avoid using `HtmlHelper.Raw` with user input.

CSHTML Copy

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

The code renders the following HTML:

HTML Copy

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

Razor code blocks

Razor code blocks start with `@` and are enclosed by `{}`. Unlike expressions, C# code inside code blocks isn't rendered. Code blocks and expressions in a view share the same scope and are defined in order:

CSHTML Copy

```
@{
    var quote = "The future depends on what you do today. - Mahatma Gandhi";
}

<p>@quote</p>

@{
    quote = "Hate cannot drive out hate, only love can do that. - Martin Luther King, Jr.";
}

<p>@quote</p>
```

The code renders the following HTML:

HTML Copy

```
<p>The future depends on what you do today. - Mahatma Gandhi</p>
<p>Hate cannot drive out hate, only love can do that. - Martin Luther King, Jr.</p>
```

In code blocks, declare local functions with markup to serve as templating methods:

CSHTML Copy

```
@{
    void RenderName(string name)
    {
        <p>Name: <strong>@name</strong></p>
    }

    RenderName("Mahatma Gandhi");
    RenderName("Martin Luther King, Jr.");
}
```

The code renders the following HTML:

HTML Copy

```
<p>Name: <strong>Mahatma Gandhi</strong></p>
<p>Name: <strong>Martin Luther King, Jr.</strong></p>
```

Implicit transitions

The default language in a code block is C#, but the Razor Page can transition back to HTML:

CSHTML Copy

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

Explicit delimited transition

To define a subsection of a code block that should render HTML, surround the characters for rendering with the Razor `<text>` tag:

CSHTML Copy

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

Use this approach to render HTML that isn't surrounded by an HTML tag. Without an HTML or Razor tag, a Razor runtime error occurs.

The `<text>` tag is useful to control whitespace when rendering content:

- Only the content between the `<text>` tag is rendered.
- No whitespace before or after the `<text>` tag appears in the HTML output.

Explicit line transition

To render the rest of an entire line as HTML inside a code block, use `@:` syntax:

CSHTML Copy

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

Without the `@:` in the code, a Razor runtime error is generated.

Extra `@` characters in a Razor file can cause compiler errors at statements later in the block. These compiler errors can be difficult to understand because the actual error occurs before the reported error. This error is common after combining multiple implicit/explicit expressions into a single code block.

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`

`@if` controls when code runs:

CSHTML Copy

```
@if (value % 2 == 0)
{
    <p>The value was even.</p>
}
```

`else` and `else if` don't require the `@` symbol:

CSHTML Copy

```
@if (value % 2 == 0)
{
    <p>The value was even.</p>
}
else if (value >= 1337)
{
    <p>The value is large.</p>
}
else
{
    <p>The value is odd and small.</p>
}
```

The following markup shows how to use a switch statement:

CSHTML Copy

```
@switch (value)
{
```

```
    case 1:
        <p>The value is 1!</p>
        break;
    case 1337:
        <p>Your number is 1337!</p>
        break;
    default:
        <p>Your number wasn't 1 or 1337.</p>
        break;
}
```

Looping `@for, @foreach, @while, and @do while`

Templated HTML can be rendered with looping control statements. To render a list of people:

CSHTML Copy

```
@{
    var people = new Person[]
    {
          new Person("Weston", 33),
          new Person("Johnathon", 41),
          ...
    };
}
```

The following looping statements are supported:

`@for`

CSHTML Copy

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

```
    var person = people[i];
    <p>Name: @person.Name</p>
    <p>Age: @person.Age</p>
}
```

`@foreach`

CSHTML Copy

```
@foreach (var person in people)
{
    <p>Name: @person.Name</p>
    <p>Age: @person.Age</p>
}
```

`@while`

CSHTML Copy

```
@{ var i = 0; }
@while (i < people.Length)
{
    var person = people[i];
    <p>Name: @person.Name</p>
    <p>Age: @person.Age</p>

    i++;
}
```

`@do while`

CSHTML Copy

```
@{ 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`

In C#, a `using` statement is used to ensure an object is disposed. In Razor, the same mechanism is used to create HTML Helpers that contain additional content. In the following code, HTML Helpers render a `<form>` tag with the `@using` statement:

CSHTML Copy

```
@using (Html.BeginForm())
{
    <div>
        Email: <input type="email" id="Email" value="">
        <button>Register</button>
    </div>
}
```

`@try, catch, finally`

Exception handling is similar to C#:

CSHTML Copy

```
@try
{
    throw new InvalidOperationException("You did something invalid.");
}
catch (Exception ex)
{
    <p>The exception message: @ex.Message</p>
}
finally
{
    <p>The finally statement.</p>
}
```

`@lock`

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

CSHTML Copy

```
@lock (SomeLock)
{
    // Do critical section work
}
```

Comments

Razor supports C# and HTML comments:

CSHTML Copy

```
@{
    /* C# comment */
```

```
    // Another C# comment
}
<!-- HTML comment -->
```

The code renders the following HTML:

HTML Copy

```
<!-- HTML comment -->
```

Razor comments are removed by the server before the webpage is rendered. Razor uses `@* *@` to delimit comments. The following code is commented out, so the server doesn't render any markup:

CSHTML Copy

```
@*
    @{
        /* C# comment */
        // Another C# comment
    }
    <!-- HTML comment -->
*@
```

Directives

Razor directives are represented by implicit expressions with reserved keywords following the `@` symbol. A directive typically changes the way a view is parsed or enables different functionality.

Understanding how Razor generates code for a view makes it easier to understand how directives work.

CSHTML Copy

```
@{
    var quote = "Getting old ain't for wimps! - Anonymous";
}

<div>Quote of the Day: @quote</div>
```

The code generates a class similar to the following:

C# Copy

```
public class _Views_Something_cshtml : RazorPage<dynamic>
{
    public override async Task ExecuteAsync()
    {
        var output = "Getting old ain't for wimps! - Anonymous";

        WriteLiteral("/r/n<div>Quote of the Day: ");
        Write(output);
        WriteLiteral("</div>");
    }
}
```

Later in this article, the section Inspect the Razor C# class generated for a view explains how to view this generated class.

`@attribute`

The `@attribute` directive adds the given attribute to the class of the generated page or view. The following example adds the `[Authorize]` attribute:

CSHTML Copy

```
@attribute [Authorize]
```

`@code`

This scenario only applies to Razor components (.razor).

The `@code` block enables a Razor component to add C# members (fields, properties, and methods) to a component:

razor Copy

```
@code {
    // C# members (fields, properties, and methods)
}
```

For Razor components, `@code` is an alias of @functions and recommended over `@functions`. More than one `@code` block is permissible.

`@functions`

The `@functions` directive enables adding C# members (fields, properties, and methods) to the generated class:

CSHTML Copy

```
@functions {
    // C# members (fields, properties, and methods)
}
```

In Razor components, use `@code` over `@functions` to add C# members.

For example:

CSHTML Copy

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

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

The code generates the following HTML markup:

HTML Copy

```
<div>From method: Hello</div>
```

The following code is the generated Razor C# class:

C# Copy

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc.Razor;

public class _Views_Home_Test_cshtml : RazorPage<dynamic>
{
    // Functions placed between here
    public string GetHello()
    {
        return "Hello";
    }
    // And here.
#pragma warning disable 1998
    public override async Task ExecuteAsync()
    {
        WriteLiteral("\r\n<div>From method: ");
        Write(GetHello());
```

```
        WriteLiteral("</div>\r\n");
    }
#pragma warning restore 1998
```

`@functions` methods serve as templating methods when they have markup:

CSHTML Copy

```
@{
    RenderName("Mahatma Gandhi");
    RenderName("Martin Luther King, Jr.");
}

@functions {
    private void RenderName(string name)
    {
        <p>Name: <strong>@name</strong></p>
    }
}
```

The code renders the following HTML:

HTML Copy

```
<p>Name: <strong>Mahatma Gandhi</strong></p>
<p>Name: <strong>Martin Luther King, Jr.</strong></p>
```

`@implements`

The `@implements` directive implements an interface for the generated class.

The following example implements System.IDisposable so that the Dispose method can be called:

CSHTML Copy

```
@implements IDisposable

<h1>Example</h1>

@functions {
    private bool _isDisposed;

    ...

    public void Dispose() => _isDisposed = true;
}
```

@inherits

The `@inherits` directive provides full control of the class the view inherits:

CSHTML Copy

```
@inherits TypeNameOfClassToInheritFrom
```

The following code is a custom Razor page type:

C# Copy

```
using Microsoft.AspNetCore.Mvc.Razor;

public abstract class CustomRazorPage<TModel> : RazorPage<TModel>
{
    public string CustomText { get; } =
        "Gardyloo! - A Scottish warning yelled from a window before dumping" +
        "a slop bucket on the street below.";
}
```

The `CustomText` is displayed in a view:

CSHTML Copy

```
@inherits CustomRazorPage<TModel>

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

The code renders the following HTML:

HTML Copy

```
<div>
    Custom text: Gardyloo! - A Scottish warning yelled from a window before dumping
    a slop bucket on the street below.
</div>
```

`@model` and `@inherits` can be used in the same view. `@inherits` can be in a *_ViewImports.cshtml* file that the view imports:

CSHTML Copy

```
@inherits CustomRazorPage<TModel>
```

The following code is an example of a strongly-typed view:

CSHTML Copy

```
@inherits CustomRazorPage<TModel>

<div>The Login Email: @Model.Email</div>
<div>Custom text: @CustomText</div>
```

If "rick@contoso.com" is passed in the model, the view generates the following HTML markup:

HTML Copy

```
<div>The Login Email: rick@contoso.com</div>
<div>
    Custom text: Gardyloo! - A Scottish warning yelled from a window before dumping
    a slop bucket on the street below.
</div>
```

`@inject`

The `@inject` directive enables the Razor Page to inject a service from the service container into a view. For more information, see Dependency injection into views.

`@layout`

This scenario only applies to Razor components (.razor).

The `@layout` directive specifies a layout for a Razor component. Layout components are used to avoid code duplication and inconsistency. For more information, see ASP.NET Core Blazor layouts.

`@model`

This scenario only applies to MVC views and Razor Pages (.cshtml).

The `@model` directive specifies the type of the model passed to a view or page:

CSHTML Copy

```
@model TypeNameOfModel
```

In an ASP.NET Core MVC or Razor Pages app created with individual user accounts, *Views/Account/Login.cshtml* contains the following model declaration:

CSHTML Copy

```
@model LoginViewModel
```

The class generated inherits from `RazorPage<dynamic>`:

C# Copy

```
public class _Views_Account_Login_cshtml : RazorPage<LoginViewModel>
```

Razor exposes a `Model` property for accessing the model passed to the view:

CSHTML Copy

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

The `@model` directive specifies the type of the `Model` property. The directive specifies the `T` in `RazorPage<T>` that the generated class that the view derives from. If the `@model` directive isn't specified, the `Model` property is of type `dynamic`. For more information, see Strongly typed models and the @model keyword.

`@namespace`

The `@namespace` directive:

- Sets the namespace of the class of the generated Razor page, MVC view, or Razor component.
- Sets the root derived namespaces of a pages, views, or components classes from the closest imports file in the directory tree, *_ViewImports.cshtml* (views or pages) or *_Imports.razor* (Razor components).

CSHTML Copy

```
@namespace Your.Namespace.Here
```

For the Razor Pages example shown in the following table:

- Each page imports *Pages/_ViewImports.cshtml*.
- *Pages/_ViewImports.cshtml* contains `@namespace Hello.World`.
- Each page has `Hello.World` as the root of it's namespace.

Page	Namespace
Pages/Index.cshtml	`Hello.World`
Pages/MorePages/Page.cshtml	`Hello.World.MorePages`
Pages/MorePages/EvenMorePages/Page.cshtml	`Hello.World.MorePages.EvenMorePages`

The preceding relationships apply to import files used with MVC views and Razor components.

When multiple import files have a `@namespace` directive, the file closest to the page, view, or component in the directory tree is used to set the root namespace.

If the *EvenMorePages* folder in the preceding example has an imports file with `@namespace Another.Planet` (or the *Pages/MorePages/EvenMorePages/Page.cshtml* file contains `@namespace Another.Planet`), the result is shown in the following table.

Page	Namespace
Pages/Index.cshtml	`Hello.World`
Pages/MorePages/Page.cshtml	`Hello.World.MorePages`
Pages/MorePages/EvenMorePages/Page.cshtml	`Another.Planet`

`@page`

The `@page` directive has different effects depending on the type of the file where it appears. The directive:

- In a *.cshtml* file indicates that the file is a Razor Page. For more information, see Custom routes and Introduction to Razor Pages in ASP.NET Core.
- Specifies that a Razor component should handle requests directly. For more information, see ASP.NET Core Blazor routing.

`@section`

This scenario only applies to MVC views and Razor Pages (.cshtml).

The `@section` directive is used in conjunction with MVC and Razor Pages layouts to enable views or pages to render content in different parts of the HTML page. For more information, see Layout in ASP.NET Core.

`@using`

The `@using` directive adds the C# `using` directive to the generated view:

CSHTML Copy

```
@using System.IO
@{
    var dir = Directory.GetCurrentDirectory();
}
<p>@dir</p>
```

In Razor components, `@using` also controls which components are in scope.

Directive attributes

Razor directive attributes are represented by implicit expressions with reserved keywords following the `@` symbol. A directive attribute typically changes the way an element is parsed or enables different functionality.

`@attributes`

This scenario only applies to Razor components (.razor).

`@attributes` allows a component to render non-declared attributes. For more information, see Create and use ASP.NET Core Razor components.

`@bind`

This scenario only applies to Razor components (.razor).

Data binding in components is accomplished with the `@bind` attribute. For more information, see ASP.NET Core Blazor data binding.

`@on{EVENT}`

This scenario only applies to Razor components (.razor).

Razor provides event handling features for components. For more information, see ASP.NET Core Blazor event handling.

`@on{EVENT}:preventDefault`

This scenario only applies to Razor components (.razor).

Prevents the default action for the event.

`@on{EVENT}:stopPropagation`

This scenario only applies to Razor components (.razor).

Stops event propagation for the event.

`@key`

This scenario only applies to Razor components (.razor).

The `@key` directive attribute causes the components diffing algorithm to guarantee preservation of elements or components based on the key's value. For more information, see Create and use ASP.NET Core Razor components.

`@ref`

This scenario only applies to Razor components (.razor).

Component references (`@ref`) provide a way to reference a component instance so that you can issue commands to that instance. For more information, see Create and use ASP.NET Core Razor components.

`@typeparam`

This scenario only applies to Razor components (.razor).

The `@typeparam` directive declares a generic type parameter for the generated component class. For more information, see ASP.NET Core Blazor templated components.

Templated Razor delegates

Razor templates allow you to define a UI snippet with the following format:

CSHTML Copy

```
@<tag>...</tag>
```

The following example illustrates how to specify a templated Razor delegate as a Func<T,TResult>. The dynamic type is specified for the parameter of the method that the delegate encapsulates. An object type is specified as the return value of the delegate. The template is used with a List<T> of `Pet` that has a `Name` property.

C# Copy

```
public class Pet
{
    public string Name { get; set; }
}
```

CSHTML Copy

```
@{
    Func<dynamic, object> petTemplate = @<p>You have a pet named <strong>@item.Name</strong>.</p>;
```

```
    var pets = new List<Pet>
    {
        new Pet { Name = "Rin Tin Tin" },
        new Pet { Name = "Mr. Bigglesworth" },
        new Pet { Name = "K-9" }
    };
}
```

The template is rendered with `pets` supplied by a `foreach` statement:

CSHTML Copy

```
@foreach (var pet in pets)
{
    @petTemplate(pet)
}
```

Rendered output:

HTML Copy

```
<p>You have a pet named <strong>Rin Tin Tin</strong>.</p>
<p>You have a pet named <strong>Mr. Bigglesworth</strong>.</p>
<p>You have a pet named <strong>K-9</strong>.</p>
```

You can also supply an inline Razor template as an argument to a method. In the following example, the `Repeat` method receives a Razor template. The method uses the template to produce HTML content with repeats of items supplied from a list:

CSHTML Copy

```
@using Microsoft.AspNetCore.Html
```

```
@functions {
    public static IHtmlContent Repeat(IEnumerable<dynamic> items, int times,
        Func<dynamic, IHtmlContent> template)
    {
        var html = new HtmlContentBuilder();

        foreach (var item in items)
        {
            for (var i = 0; i < times; i++)
            {
                html.AppendHtml(template(item));
            }
        }

        return html;
    }
}
```

Using the list of pets from the prior example, the `Repeat` method is called with:

- List<T> of `Pet`.
- Number of times to repeat each pet.
- Inline template to use for the list items of an unordered list.

CSHTML Copy

```
<ul>
    @Repeat(pets, 3, @<li>@item.Name</li>)
</ul>
```

Rendered output:

HTML Copy

```
<ul>
    <li>Rin Tin Tin</li>
    <li>Rin Tin Tin</li>
    <li>Rin Tin Tin</li>
    <li>Mr. Bigglesworth</li>
    <li>Mr. Bigglesworth</li>
    <li>Mr. Bigglesworth</li>
    <li>K-9</li>
    <li>K-9</li>
    <li>K-9</li>
</ul>
```

Tag Helpers

This scenario only applies to MVC views and Razor Pages (.cshtml).

There are three directives that pertain to Tag Helpers.

Directive	Function
@addTagHelper	Makes Tag Helpers available to a view.
@removeTagHelper	Removes Tag Helpers previously added from a view.
@tagHelperPrefix	Specifies a tag prefix to enable Tag Helper support and to make Tag Helper usage explicit.

Razor reserved keywords

Razor keywords

- `page` (Requires ASP.NET Core 2.1 or later)
- `namespace`
- `functions`
- `inherits`
- `model`
- `section`
- `helper` (Not currently supported by ASP.NET Core)

Razor keywords are escaped with `@(Razor Keyword)` (for example, `@(functions)`).

C# Razor keywords

- `case`
- `do`
- `default`
- `for`
- `foreach`
- `if`
- `else`
- `lock`
- `switch`
- `try`
- `catch`
- `finally`
- `using`
- `while`

C# Razor keywords must be double-escaped with `@(@C# Razor Keyword)` (for example, `@(@case)`). The first `@` escapes the Razor parser. The second `@` escapes the C# parser.

Reserved keywords not used by Razor

- `class`

Inspect the Razor C# class generated for a view

With .NET Core SDK 2.1 or later, the Razor SDK handles compilation of Razor files. When building a project, the Razor SDK generates an *obj/<build_configuration>/<target_framework_moniker>/Razor* directory in the project root. The directory structure within the *Razor* directory mirrors the project's directory structure.

Consider the following directory structure in an ASP.NET Core 2.1 Razor Pages project targeting .NET Core 2.1:

Copy

```
Areas/
  Admin/
    Pages/
      Index.cshtml
      Index.cshtml.cs
Pages/
  Shared/
    _Layout.cshtml
  _ViewImports.cshtml
  _ViewStart.cshtml
  Index.cshtml
  Index.cshtml.cs
```

Building the project in *Debug* configuration yields the following *obj* directory:

Copy

```
obj/
  Debug/
    netcoreapp2.1/
      Razor/
        Areas/
          Admin/
            Pages/
              Index.g.cshtml.cs
        Pages/
          Shared/
            _Layout.g.cshtml.cs
          _ViewImports.g.cshtml.cs
          _ViewStart.g.cshtml.cs
          Index.g.cshtml.cs
```

To view the generated class for *Pages/Index.cshtml*, open *obj/Debug/netcoreapp2.1/Razor/Pages/Index.g.cshtml.cs*.

View lookups and case sensitivity

The Razor view engine performs case-sensitive lookups for views. However, the actual lookup is determined by the underlying file system:

- File based source:
  - On operating systems with case insensitive file systems (for example, Windows), physical file provider lookups are case insensitive. For example, `return View("Test")` results in matches for */Views/Home/Test.cshtml*, */Views/home/test.cshtml*, and any other casing variant.
  - On case-sensitive file systems (for example, Linux, OSX, and with `EmbeddedFileProvider`), lookups are case-sensitive. For example, `return View("Test")` specifically matches */Views/Home/Test.cshtml*.
- Precompiled views: With ASP.NET Core 2.0 and later, looking up precompiled views is case insensitive on all operating systems. The behavior is identical to physical file provider's behavior on Windows. If two precompiled views differ only

in case, the result of lookup is non-deterministic.

Developers are encouraged to match the casing of file and directory names to the casing of:

- Area, controller, and action names.
- Razor Pages.

Matching case ensures the deployments find their views regardless of the underlying file system.

Additional resources

Introduction to ASP.NET Web Programming Using the Razor Syntax provides many samples of programming with Razor syntax.