Create and use ASP.NET Core Razor components

11/25/2020 • 24 minutes to read •

In this article

By Luke Latham, Daniel Roth, Scott Addie, and Tobias Bartsch

View or download sample code (how to download)

View or download sample code (how to download)

Blazor apps are built using *components*. A component is a self-contained chunk of user interface (UI), such as a page, dialog, or form. A component includes HTML markup and the processing logic required to inject data or respond to UI events. Components are flexible and lightweight. They can be nested, reused, and shared among projects.

Component classes

Components are implemented in Razor component files (`.razor`) using a combination of C# and HTML markup. A component in Blazor is formally referred to as a *Razor component*.

Razor syntax

Razor components in Blazor apps extensively use Razor syntax. If you aren't familiar with the Razor markup language, we recommend reading razor syntax reference for ASP.NET Core before proceeding.

When accessing the content on Razor syntax, pay special attention to the following sections:

- Directives: `@`-prefixed reserved keywords that typically change the way component markup is parsed or function.
- Directive attributes: `@`-prefixed reserved keywords that typically change the way component elements are parsed or function.

Names

A component's name must start with an uppercase character. For example, `MyCoolComponent.razor` is valid, and `myCoolComponent.razor` is invalid.

Routing

Routing in Blazor is achieved by providing a route template to each accessible component in the app. When a Razor file with an @page directive is compiled, the generated class is given a RouteAttribute specifying the route template. At runtime, the router looks for component classes with a RouteAttribute and renders whichever component has a route template that matches the requested URL. For more information, see ASP.NET Core Blazor routing.

razor Copy

```razor
@page "/ParentComponent"

...
```

Markup

The UI for a component is defined using HTML. Dynamic rendering logic (for example, loops, conditionals, expressions) is added using an embedded C# syntax called *Razor*. When an app is compiled, the HTML markup and C# rendering logic are converted into a component class. The name of the generated class matches the name of the file.

Members of the component class are defined in an @code block. In the @code block, component state (properties, fields) is specified with methods for event handling or for defining other component logic. More than one @code block is permissible.

Component members can be used as part of the component's rendering logic using C# expressions that start with `@`. For example, a C# field is rendered by prefixing `@` to the field name. The following example evaluates and renders:

- `headingFontStyle` to the CSS property value for `font-style`.
- `headingText` to the content of the `<h1>` element.

razor Copy

```razor
<h1 style="font-style:@headingFontStyle">@headingText</h1>
```

```
<h1 style="font-style:@headingFontStyle">@headingText</h1>

@code {
    private string headingFontStyle = "italic";
    private string headingText = "Put on your new Blazor!";
}
```

After the component is initially rendered, the component regenerates its render tree in response to events. Blazor then compares the new render tree against the previous one and applies any modifications to the browser's Document Object Model (DOM).

Components are ordinary C# classes and can be placed anywhere within a project. Components that produce webpages usually reside in the `Pages` folder. Non-page components are frequently placed in the `Shared` folder or a custom folder added to the project.

Namespaces

Typically, a component's namespace is derived from the app's root namespace and the component's location (folder) within the app. If the app's root namespace is `BlazorSample` and the `Counter` component resides in the `Pages` folder:

- The `Counter` component's namespace is `BlazorSample.Pages`.
- The fully qualified type name of the component is `BlazorSample.Pages.Counter`.

For custom folders that hold components, add a @using directive to the parent component or to the app's `_Imports.razor` file. The following example makes components in the `Components` folder available:

razor Copy

```razor
@using BlazorSample.Components
```

Components can also be referenced using their fully qualified names, which doesn't require the @using directive:

razor Copy

```razor
<BlazorSample.Components.MyComponent />
```

The namespace of a component authored with Razor is based on (in priority order):

- @namespace designation in Razor file (`.razor`) markup (`@namespace BlazorSample.MyNamespace`).
- The project's `RootNamespace` in the project file (`<RootNamespace>BlazorSample</RootNamespace>`).
- The project name, taken from the project file's file name (`.csproj`), and the path from the project root to the component. For example, the framework resolves `{PROJECT ROOT}/Pages/Index.razor` (`BlazorSample.csproj`) to the namespace `BlazorSample.Pages`. Components follow C# name binding rules. For the `Index` component in this example, the components in scope are all of the components:
  - In the same folder, `Pages`.
  - The components in the project's root that don't explicitly specify a different namespace.

Note

The `global::` qualification isn't supported.

Importing components with aliased using statements (for example, `@using Foo = Bar`) isn't supported.

Partially qualified names aren't supported. For example, adding `@using BlazorSample` and referencing the `NavMenu` component (`NavMenu.razor`) with `<Shared.NavMenu></Shared.NavMenu>` isn't supported.

Partial class support

Razor components are generated as partial classes. Razor components are authored using either of the following approaches:

approaches.

- C# code is defined in an @code block with HTML markup and Razor code in a single file. Blazor templates define their Razor components using this approach.
- C# code is placed in a code-behind file defined as a partial class.

The following example shows the default `Counter` component with an @code block in an app generated from a Blazor template. HTML markup, Razor code, and C# code are in the same file:

`Pages/Counter.razor`:

razor Copy

```razor
@page "/counter"

<h1>Counter</h1>

<p>Current count: @currentCount</p>

<button class="btn btn-primary" @onclick="IncrementCount">Click me</button>

@code {
    private int currentCount = 0;

    void IncrementCount()
    {
        currentCount++;
    }
}
```

The `Counter` component can also be created using a code-behind file with a partial class:

`Pages/Counter.razor`:

razor Copy

```razor
@page "/counter"

<h1>Counter</h1>

<p>Current count: @currentCount</p>

<button class="btn btn-primary" @onclick="IncrementCount">Click me</button>
```

`Counter.razor.cs`:

C# Copy

```csharp
namespace BlazorSample.Pages
{
    public partial class Counter
    {
        private int currentCount = 0;

        void IncrementCount()
        {
            currentCount++;
        }
    }
}
```

Add any required namespaces to the partial class file as needed. Typical namespaces used by Razor components include:

C# Copy

```csharp
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Components;
using Microsoft.AspNetCore.Components.Authorization;
using Microsoft.AspNetCore.Components.Forms;
using Microsoft.AspNetCore.Components.Routing;
using Microsoft.AspNetCore.Components.Web;
```

Important

@using directives in the `_Imports.razor` file are only applied to Razor files (`.razor`), not C# files (`.cs`).

Specify a base class

The @inherits directive can be used to specify a base class for a component. The following example shows how a component can inherit a base class, `BlazorRocksBase`, to provide the component's properties and methods. The base class should derive from ComponentBase.

`Pages/BlazorRocks.razor`:

razor Copy

```razor
@page "/BlazorRocks"
@inherits BlazorRocksBase

<h1>@BlazorRocksText</h1>
```

`BlazorRocksBase.cs`:

C# Copy

```csharp
using Microsoft.AspNetCore.Components;

namespace BlazorSample
{
    public class BlazorRocksBase : ComponentBase
    {
        public string BlazorRocksText { get; set; } =
            "Blazor rocks the browser!";
    }
}
```

Use components

Components can include other components by declaring them using HTML element syntax. The markup for using a component looks like an HTML tag where the name of the tag is the component type.

The following markup in `Pages/Index.razor` renders a `HeadingComponent` instance:

razor Copy

```razor
<HeadingComponent />
```

`Components/HeadingComponent.razor`:

razor Copy

```razor
@using System.Globalization

<h1 style="font-style:@headingFontStyle">@headingText</h1>

<form>
    <div>
        <label class="form-check-label">
            <input type="checkbox" id="italicsCheck"
               @bind="italicsCheck" />
            Use italics
        </label>
    </div>

    <button type="button" class="btn btn-primary" @onclick="UpdateHeading">

        Update heading
    </button>
</form>

@code {
    private static TextInfo tinfo = CultureInfo.CurrentCulture.TextInfo;
    private string headingText =
        tinfo.ToTitleCase("welcome to blazor!");
    private string headingFontStyle = "normal";
    private bool italicsCheck = false;

    public void UpdateHeading()
    {
        headingFontStyle = italicsCheck ? "italic" : "normal";
    }
}
```

If a component contains an HTML element with an uppercase first letter that doesn't match a component name, a warning is emitted indicating that the element has an unexpected name. Adding an @using directive for the component's namespace makes the component available, which resolves the warning.

Parameters

Route parameters

Components can receive route parameters from the route template provided in the @page directive. The router uses route parameters to populate the corresponding component parameters.

Optional parameters are supported. In the following example, the `text` optional parameter assigns the value of the route segment to the component's `Text` property. If the segment isn't present, the value of `Text` is set to `fantastic` segment to the component's `Text` property. If the segment isn't present, the value of `Text` is set to `fantastic`.

`Pages/RouteParameter.razor`:

razor Copy

```razor
@page "/RouteParameter/{text?}"

<h1>Blazor is @Text!</h1>

@code {
    [Parameter]
    public string Text { get; set; }

    protected override void OnInitialized()
    {
        Text = Text ?? "fantastic";
    }
}
```

For information on catch-all route parameters (`{*pageRoute}`), which capture paths across multiple folder boundaries, see ASP.NET Core Blazor routing.

Component parameters

Components can have *component parameters*, which are defined using public simple or complex properties on the component class with the [Parameter] attribute. Use attributes to specify arguments for a component in markup.

`Components/ChildComponent.razor`:

razor Copy

```razor
<div class="panel panel-default">
    <div class="panel-heading">@Title</div>
    <div class="panel-body">@ChildContent</div>
    <button class="btn btn-primary" @onclick="OnClickCallback">
        Trigger a Parent component method
    </button>
</div>

@code {
    [Parameter]
    public string Title { get; set; }

    [Parameter]
    public RenderFragment ChildContent { get; set; }

    [Parameter]
    public EventCallback<MouseEventArgs> OnClickCallback { get; set; }
}
```

In the following example from the sample app, the `ParentComponent` sets the value of the `Title` property of the `ChildComponent`.

`Pages/ParentComponent.razor`:

razor Copy

```razor
@page "/ParentComponent"

<h1>Parent-child example</h1>

<ChildComponent Title="Panel Title from Parent"
                OnClickCallback="@ShowMessage">
    Content of the child component is supplied
    by the parent component.
</ChildComponent>
```

By convention, an attribute value that consists of C# code is assigned to a parameter using Razor's reserved @ symbol:

- Parent field or property: `Title="@{FIELD OR PROPERTY}`, where the placeholder `{FIELD OR PROPERTY}` is a C# field or property of the parent component.
- Result of a method: `Title="@{METHOD}"`, where the placeholder `{METHOD}` is a C# method of the parent component.
- Implicit or explicit expression: `Title="@({EXPRESSION})"`, where the placeholder `{EXPRESSION}` is a C# expression.

For more information, see razor syntax reference for ASP.NET Core.

⚠ **Warning**

Don't create components that write to their own *component parameters*, use a private field instead. For more information, see the **Overwritten parameters** section.

Child content

Components can set the content of another component. The assigning component provides the content between the tags that specify the receiving component.

In the following example, the `ChildComponent` has a `ChildContent` property that represents a RenderFragment, which represents a segment of UI to render. The value of `ChildContent` is positioned in the component's markup where the content should be rendered. The value of `ChildContent` is received from the parent component and rendered inside the Bootstrap panel's `panel-body`.

`Components/ChildComponent.razor`:

razor Copy

```razor
<div class="panel panel-default">
```

```
    <div class="panel-heading">@Title</div>
    <div class="panel-body">@ChildContent</div>

    <button class="btn btn-primary" @onclick="OnClickCallback">
        Trigger a Parent component method
    </button>
</div>

@code {
    [Parameter]
    public string Title { get; set; }

    [Parameter]
    public RenderFragment ChildContent { get; set; }

    [Parameter]
    public EventCallback<MouseEventArgs> OnClickCallback { get; set; }
}
```

Note

The property receiving the **RenderFragment** content must be named `ChildContent` by convention.

The `ParentComponent` in the sample app can provide content for rendering the `ChildComponent` by placing the content inside the `<ChildComponent>` tags.

`Pages/ParentComponent.razor`:

razor Copy

```
@page "/ParentComponent"

<h1>Parent-child example</h1>
```

```
<ChildComponent Title="Panel Title from Parent"
                OnClickCallback="@ShowMessage">
    Content of the child component is supplied
    by the parent component.
</ChildComponent>
```

Due to the way that Blazor renders child content, rendering components inside a `for` loop requires a local index variable if the incrementing loop variable is used in the child component's content:

razor Copy

```
@for (int c = 0; c < 10; c++)
{
    var current = c;
    <ChildComponent Title="@c">
        Child Content: Count: @current
    </ChildComponent>
}
```

Alternatively, use a `foreach` loop with Enumerable.Range:

razor Copy

```
@foreach(var c in Enumerable.Range(0,10))
{
    <ChildComponent Title="@c">
        Child Content: Count: @c
    </ChildComponent>
}
```

Attribute splatting and arbitrary parameters

Components can capture and render additional attributes in addition to the component's declared parameters. Additional attributes can be captured in a dictionary and then *splatted* onto an element when the component is rendered using the @attributes Razor directive. This scenario is useful when defining a component that produces a markup element that supports a variety of customizations. For example, it can be tedious to define attributes separately for an `<input>` that supports many parameters.

In the following example, the first `<input>` element (`id="useIndividualParams"`) uses individual component parameters, while the second `<input>` element (`id="useAttributesDict"`) uses attribute splatting:

razor

```razor
<input id="useIndividualParams"
       maxlength="@maxlength"
       placeholder="@placeholder"
       required="@required"
       size="@size" />

<input id="useAttributesDict"
       @attributes="InputAttributes" />

@code {
    public string maxlength = "10";
    public string placeholder = "Input placeholder text";
    public string required = "required";
    public string size = "50";

    public Dictionary<string, object> InputAttributes { get; set; } =
        new Dictionary<string, object>()
        {
            { "maxlength", "10" },
            { "placeholder", "Input placeholder text" },
            { "required", "required" },
            { "size", "50" }
        };
}
```

The type of the parameter must implement `IEnumerable<KeyValuePair<string, object>>` or `IReadOnlyDictionary<string, object>` with string keys.

The rendered `<input>` elements using both approaches is identical:

HTML Copy

```html
<input id="useIndividualParams"
       maxlength="10"
       placeholder="Input placeholder text"
       required="required"
       size="50">

<input id="useAttributesDict"
       maxlength="10"
       placeholder="Input placeholder text"
       required="required"
       size="50">
```

To accept arbitrary attributes, define a component parameter using the [Parameter] attribute with the CaptureUnmatchedValues property set to `true`:

razor Copy

```razor
@code {
    [Parameter(CaptureUnmatchedValues = true)]
    public Dictionary<string, object> InputAttributes { get; set; }
}
```

The CaptureUnmatchedValues property on [Parameter] allows the parameter to match all attributes that don't match any other parameter. A component can only define a single parameter with CaptureUnmatchedValues. The property type used with CaptureUnmatchedValues must be assignable from `Dictionary<string, object>` with string keys. `IEnumerable<KeyValuePair<string, object>>` or `IReadOnlyDictionary<string, object>` are also options in this scenario.

The position of @attributes relative to the position of element attributes is important. When @attributes are splatted on the element, the attributes are processed from right to left (last to first). Consider the following example of a component that consumes a `Child` component:

`ParentComponent.razor`:

razor Copy

```razor
<ChildComponent extra="10" />
```

`ChildComponent.razor`:

razor Copy

```razor
<div @attributes="AdditionalAttributes" extra="5" />

[Parameter(CaptureUnmatchedValues = true)]
public IDictionary<string, object> AdditionalAttributes { get; set; }
```

The `Child` component's `extra` attribute is set to the right of @attributes. The `Parent` component's rendered `<div>` contains `extra="5"` when passed through the additional attribute because the attributes are processed right to left (last to first):

HTML Copy

```html
<div extra="5" />
```

In the following example, the order of `extra` and @attributes is reversed in the `Child` component's `<div>`:

`ParentComponent.razor`:

razor

```razor
<ChildComponent extra="10" />
```

`ChildComponent.razor`:

razor

```razor
<div extra="5" @attributes="AdditionalAttributes" />

[Parameter(CaptureUnmatchedValues = true)]
public IDictionary<string, object> AdditionalAttributes { get; set; }
```

The rendered `<div>` in the `Parent` component contains `extra="10"` when passed through the additional attribute:

HTML

```html
<div extra="10" />
```

Capture references to components

Component references provide a way to reference a component instance so that you can issue commands to that instance, such as `Show` or `Reset`. To capture a component reference:

- Add an @ref attribute to the child component.
- Define a field with the same type as the child component.

razor

```razor
<CustomLoginDialog @ref="loginDialog" ... />
```

```
@code {
    private CustomLoginDialog loginDialog;


    private void OnSomething()
    {
        loginDialog.Show();
    }
}
```

When the component is rendered, the `loginDialog` field is populated with the `MyLoginDialog` child component instance. You can then invoke .NET methods on the component instance.

Important

The `loginDialog` variable is only populated after the component is rendered and its output includes the `MyLoginDialog` element. Until the component is rendered, there's nothing to reference.

To manipulate components references after the component has finished rendering, use the **OnAfterRenderAsync or OnAfterRender methods**.

To use a reference variable with an event handler, use a lambda expression or assign the event handler delegate in the **OnAfterRenderAsync or OnAfterRender methods**. This ensures that the reference variable is assigned before the event handler is assigned.

razor Copy

```razor
<button type="button"
    @onclick="@(() => loginDialog.DoSomething())">Do Something</button>

<MyLoginDialog @ref="loginDialog" ... />

@code {
    private MyLoginDialog loginDialog;
}
```

To reference components in a loop, see Capture references to multiple similar child-components (dotnet/aspnetcore #13358).

While capturing component references use a similar syntax to capturing element references, it isn't a JavaScript interop feature. Component references aren't passed to JavaScript code. Component references are only used in .NET code.

Note

Do **not** use component references to mutate the state of child components. Instead, use normal declarative parameters to pass data to child components. Use of normal declarative parameters result in child components that rerender at the correct times automatically.

Synchronization context

Blazor uses a synchronization context (SynchronizationContext) to enforce a single logical thread of execution. A component's lifecycle methods and any event callbacks that are raised by Blazor are executed on the synchronization context.

Blazor Server's synchronization context attempts to emulate a single-threaded environment so that it closely matches the WebAssembly model in the browser, which is single threaded. At any given point in time, work is performed on exactly one thread, giving the impression of a single logical thread. No two operations execute concurrently.

Avoid thread-blocking calls

Generally, don't call the following methods. The following methods block the thread and thus block the app from resuming work until the underlying Task is complete:

- Result

- Wait
- WaitAny
- WaitAll
- Sleep
- GetResult

Invoke component methods externally to update state

In the event a component must be updated based on an external event, such as a timer or other notifications, use the `InvokeAsync` method, which dispatches to Blazor's synchronization context. For example, consider a *notifier service* that can notify any listening component of the updated state:

C#

```
public class NotifierService
{
    // Can be called from anywhere
    public async Task Update(string key, int value)
    {
        if (Notify != null)
        {
            await Notify.Invoke(key, value);
        }
    }

    public event Func<string, int, Task> Notify;
}
```

Register the `NotifierService`:

- In Blazor WebAssembly, register the service as singleton in `Program.Main`:

C# Copy

```
builder.Services.AddSingleton<NotifierService>();
```

- In Blazor Server, register the service as scoped in `Startup.ConfigureServices`:

C# Copy

```
services.AddScoped<NotifierService>();
```

Use the `NotifierService` to update a component:

razor Copy

```
@page "/"
@inject NotifierService Notifier
@implements IDisposable

<p>Last update: @lastNotification.key = @lastNotification.value</p>

@code {
    private (string key, int value) lastNotification;

    protected override void OnInitialized()
    {
        Notifier.Notify += OnNotify;
    }

    public async Task OnNotify(string key, int value)
    {
        await InvokeAsync(() =>
        {
            lastNotification = (key, value);
            StateHasChanged();
        });
    }
    public void Dispose()
    {
        Notifier.Notify -= OnNotify;
    }
}
```

In the preceding example, `NotifierService` invokes the component's `OnNotify` method outside of Blazor's synchronization context. `InvokeAsync` is used to switch to the correct context and queue a render.

Use @key to control the preservation of elements and components

When rendering a list of elements or components and the elements or components subsequently change, Blazor's diffing algorithm must decide which of the previous elements or components can be retained and how model objects should map to them. Normally, this process is automatic and can be ignored, but there are cases where you may want to control the process.

Consider the following example:

C# Copy

```csharp
@foreach (var person in People)
{
    <DetailsEditor Details="@person.Details" />
}

@code {
    [Parameter]
    public IEnumerable<Person> People { get; set; }
}
```

The contents of the `People` collection may change with inserted, deleted, or re-ordered entries. When the component rerenders, the `<DetailsEditor>` component may change to receive different `Details` parameter values. This may cause more complex rerendering than expected. In some cases, rerendering can lead to visible behavior differences, such as lost element focus.

The mapping process can be controlled with the @key directive attribute. @key causes the diffing algorithm to guarantee preservation of elements or components based on the key's value:

C# Copy

```
@foreach (var person in People)
{
    <DetailsEditor @key="person" Details="@person.Details" />
}

@code {
    [Parameter]
    public IEnumerable<Person> People { get; set; }
}
```

When the `People` collection changes, the diffing algorithm retains the association between `<DetailsEditor>` instances and `person` instances:

- If a `Person` is deleted from the `People` list, only the corresponding `<DetailsEditor>` instance is removed from the UI. Other instances are left unchanged.
- If a `Person` is inserted at some position in the list, one new `<DetailsEditor>` instance is inserted at that corresponding position. Other instances are left unchanged.
- If `Person` entries are re-ordered, the corresponding `<DetailsEditor>` instances are preserved and re-ordered in the UI.

In some scenarios, use of @key minimizes the complexity of rerendering and avoids potential issues with stateful parts of the DOM changing, such as focus position.

ⓘ **Important**

Keys are local to each container element or component. Keys aren't compared globally across the document.

When to use @key

Typically, it makes sense to use @key whenever a list is rendered (for example, in a foreach block) and a suitable value exists to define the @key.

You can also use @key to prevent Blazor from preserving an element or component subtree when an object changes:

razor

```razor
<div @key="currentPerson">
    ... content that depends on currentPerson ...
</div>
```

If `@currentPerson` changes, the @key attribute directive forces Blazor to discard the entire `<div>` and its descendants and rebuild the subtree within the UI with new elements and components. This can be useful if you need to guarantee that no UI state is preserved when `@currentPerson` changes.

When not to use @key

There's a performance cost when diffing with @key. The performance cost isn't large, but only specify @key if controlling the element or component preservation rules benefit the app.

Even if @key isn't used, Blazor preserves child element and component instances as much as possible. The only advantage to using @key is control over *how* model instances are mapped to the preserved component instances, instead of the diffing algorithm selecting the mapping.

What values to use for @key

Generally, it makes sense to supply one of the following kinds of value for @key:

- Model object instances (for example, a `Person` instance as in the earlier example). This ensures preservation based on object reference equality.
- Unique identifiers (for example, primary key values of type `int`, `string`, or `Guid`).

Ensure that values used for @key don't clash. If clashing values are detected within the same parent element, Blazor throws an exception because it can't deterministically map old elements or components to new elements or components. Only use distinct values, such as object instances or primary key values.

Overwritten parameters

The Blazor framework generally imposes safe parent-to-child parameter assignment:

- Parameters aren't overwritten unexpectedly.
- Side-effects are minimized. For example, additional renders are avoided because they may create infinite rendering loops.

A child component receives new parameter values that possibly overwrite existing values when the parent component rerenders. Accidentially overwriting parameter values in a child component often occurs when developing the component with one or more data-bound parameters and the developer writes directly to a parameter in the child:

- The child component is rendered with one or more parameter values from the parent component.
- The child writes directly to the value of a parameter.
- The parent component rerenders and overwrites the value of the child's parameter.

The potential for overwriting paramater values extends into the child component's property setters, too.

Our general guidance is not to create components that directly write to their own parameters.

Consider the following faulty `Expander` component that:

- Renders child content.
- Toggles showing child content with a component parameter (`Expanded`).
- The component writes directly to the `Expanded` parameter, which demonstrates the problem with overwritten parameters and should be avoided.

razor Copy

```razor
<div @onclick="Toggle" class="card bg-light mb-3" style="width:30rem">
    <div class="card-body">
        <h2 class="card-title">Toggle (<code>Expanded</code> = @Expanded)</h2>

        @if (Expanded)
        {
            <p class="card-text">@ChildContent</p>
        }
    </div>
</div>

@code {
    [Parameter]
    public bool Expanded { get; set; }

    [Parameter]
    public RenderFragment ChildContent { get; set; }

    private void Toggle()
    {
        Expanded = !Expanded;
    }
}
```

The `Expander` component is added to a parent component that may call StateHasChanged:

razor Copy

```
@page "/expander"

<Expander Expanded="true">
    Expander 1 content
</Expander>

<Expander Expanded="true" />

<button @onclick="StateHasChanged">
    Call StateHasChanged
</button>
```

Initially, the `Expander` components behave independently when their `Expanded` properties are toggled. The child components maintain their states as expected. When StateHasChanged is called in the parent, the `Expanded` parameter of the first child component is reset back to its initial value (`true`). The second `Expander` component's `Expanded` value isn't reset because no child content is rendered in the second component.

To maintain state in the preceding scenario, use a *private field* in the `Expander` component to maintain its toggled state.

The following revised `Expander` component:

- Accepts the `Expanded` component parameter value from the parent.
- Assigns the component parameter value to a *private field* (`expanded`) in the OnInitialized event.
- Uses the private field to maintain its internal toggle state, which demonstrates how to avoid writing directly to a parameter.

razor Copy

```
<div @onclick="Toggle" class="card bg-light mb-3" style="width:30rem">
    <div class="card-body">
```

```
        <h2 class="card-title">Toggle (<code>expanded</code> = @expanded)</h2>

        @if (expanded)
        {
            <p class="card-text">@ChildContent</p>
        }
    </div>
</div>

@code {
    private bool expanded;

    [Parameter]
    public bool Expanded { get; set; }

    [Parameter]
    public RenderFragment ChildContent { get; set; }

    protected override void OnInitialized()
    {
        expanded = Expanded;
    }

    private void Toggle()
    {
        expanded = !expanded;
    }
}
```

For additional information, see Blazor Two Way Binding Error (dotnet/aspnetcore #24599).

Apply an attribute

Attributes can be applied to Razor components with the @attribute directive. The following example applies the [Authorize] attribute to the component class:

razor Copy

```razor
@page "/"
@attribute [Authorize]
```

Conditional HTML element attributes

HTML element attributes are conditionally rendered based on the .NET value. If the value is `false` or `null`, the attribute isn't rendered. If the value is `true`, the attribute is rendered minimized.

In the following example, `IsCompleted` determines if `checked` is rendered in the element's markup:

razor Copy

```razor
<input type="checkbox" checked="@IsCompleted" />

@code {
    [Parameter]
    public bool IsCompleted { get; set; }
}
```

If `IsCompleted` is `true`, the check box is rendered as:

HTML Copy

```html
<input type="checkbox" checked />
```

If `IsCompleted` is `false`, the check box is rendered as:

HTML Copy

```
<input type="checkbox" />
```

For more information, see razor syntax reference for ASP.NET Core.

⚠ Warning

Some HTML attributes, such as **aria-pressed**, don't function properly when the .NET type is a `bool`. In those cases, use a `string` type instead of a `bool`.

Raw HTML

Strings are normally rendered using DOM text nodes, which means that any markup they may contain is ignored and treated as literal text. To render raw HTML, wrap the HTML content in a `MarkupString` value. The value is parsed as HTML or SVG and inserted into the DOM.

⚠ Warning

Rendering raw HTML constructed from any untrusted source is a **security risk** and should be avoided!

The following example shows using the `MarkupString` type to add a block of static HTML content to the rendered output of a component:

HTML Copy

```
@((MarkupString)myMarkup)

@code {
    private string myMarkup =
        "<p class='markup'>This is a <em>markup string</em>.</p>";
}
```

Razor templates

Render fragments can be defined using Razor template syntax. Razor templates are a way to define a UI snippet and assume the following format:

razor Copy

```razor
@<{HTML tag}>...</{HTML tag}>
```

The following example illustrates how to specify RenderFragment and RenderFragment<TValue> values and render templates directly in a component. Render fragments can also be passed as arguments to templated components.

razor Copy

```razor
@timeTemplate

@petTemplate(new Pet { Name = "Rex" })

@code {
    private RenderFragment timeTemplate = @<p>The time is @DateTime.Now.</p>;
    private RenderFragment<Pet> petTemplate = (pet) => @<p>Pet: @pet.Name</p>;

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

Rendered output of the preceding code:

HTML Copy

```html
<p>The time is 10/04/2018 01:26:52.</p>

<p>Pet: Rex</p>
```

Static assets

Blazor follows the convention of ASP.NET Core apps placing static assets under the project's web root (wwwroot) folder.

Use a base-relative path (`/`) to refer to the web root for a static asset. In the following example, `logo.png` is physically located in the `{PROJECT ROOT}/wwwroot/images` folder:

```razor
<img alt="Company logo" src="/images/logo.png" />
```

Razor components do **not** support tilde-slash notation (`~/`).

For information on setting an app's base path, see Host and deploy ASP.NET Core Blazor.

Tag Helpers aren't supported in components

Tag Helpers aren't supported in Razor components (`.razor` files). To provide Tag Helper-like functionality in Blazor, create a component with the same functionality as the Tag Helper and use the component instead.

Scalable Vector Graphics (SVG) images

Since Blazor renders HTML, browser-supported images, including Scalable Vector Graphics (SVG) images (`.svg`), are supported via the `<img>` tag:

HTML Copy

```html
<img alt="Example image" src="some-image.svg" />
```

Similarly, SVG images are supported in the CSS rules of a stylesheet file (`.css`):

css Copy

```css
.my-element {
    background-image: url("some-image.svg");
}
```

However, inline SVG markup isn't supported in all scenarios. If you place an `<svg>` tag directly into a component file (`.razor`), basic image rendering is supported but many advanced scenarios aren't yet supported. For example, `<use>` tags aren't currently respected, and @bind can't be used with some SVG tags. For more information, see SVG support in Blazor (dotnet/aspnetcore #18271).

Whitespace rendering behavior

Unless the @preservewhitespace directive is used with a value of `true`, extra whitespace is removed by default if:

- Leading or trailing within an element.
- Leading or trailing within a `RenderFragment` parameter. For example, child content passed to another component.
- It precedes or follows a C# code block, such as `@if` or `@foreach`.

Whitespace removal might affect the rendered output when using a CSS rule, such as `white-space: pre`. To disable this performance optimization and preserve the whitespace, take one of the following actions:

- Add the `@preservewhitespace true` directive at the top of the `.razor` file to apply the preference to a specific component.
- Add the `@preservewhitespace true` directive inside an `_Imports.razor` file to apply the preference to an entire subdirectory or the entire project.

In most cases, no action is required, as apps typically continue to behave normally (but faster). If stripping whitespace causes any problem for a particular component, use `@preservewhitespace true` in that component to disable this optimization.

Additional resources

- Threat mitigation guidance for ASP.NET Core Blazor Server: Includes guidance on building Blazor Server apps that must contend with resource exhaustion.