ASP.NET Core Blazor cascading values and parameters

Article • 07/19/2024

This article explains how to flow data from an ancestor Razor component to descendent components.

Cascading values and parameters provide a convenient way to flow data down a component hierarchy from an ancestor component to any number of descendent components. Unlike Component parameters, cascading values and parameters don't require an attribute assignment for each descendent component where the data is consumed. Cascading values and parameters also allow components to coordinate with each other across a component hierarchy.

> ⓘ Note
>
> The code examples in this article adopt nullable reference types (NRTs) and .NET compiler null-state static analysis, which are supported in ASP.NET Core in .NET 6 or later. When targeting ASP.NET Core 5.0 or earlier, remove the null type designation (`?`) from the `CascadingType?`, `@ActiveTab?`, `RenderFragment?`, `ITab?`, `TabSet?`, and `string?` types in the article's examples.

Root-level cascading values

Root-level cascading values can be registered for the entire component hierarchy. Named cascading values and subscriptions for update notifications are supported.

The following class is used in this section's examples.

`Dalek.cs`:

C#

```
// "Dalek" ©Terry Nation https://www.imdb.com/name/nm0622334/
// "Doctor Who" ©BBC https://www.bbc.co.uk/programmes/b006q2x0

namespace BlazorSample;

public class Dalek
{
    public int Units { get; set; }
}
```

The following registrations are made in the app's `Program` file with AddCascadingValue:

- `Dalek` with a property value for `Units` is registered as a fixed cascading value.
- A second `Dalek` registration with a different property value for `Units` is named "`AlphaGroup`".

C#

```
builder.Services.AddCascadingValue(sp => new Dalek { Units = 123 });
builder.Services.AddCascadingValue("AlphaGroup", sp => new Dalek { Units = 456 });
```

The following `Daleks` component displays the cascaded values.

`Daleks.razor`:

```razor
@page "/daleks"

<PageTitle>Daleks</PageTitle>

<h1>Root-level Cascading Value Example</h1>

<ul>
    <li>Dalek Units: @Dalek?.Units</li>
    <li>Alpha Group Dalek Units: @AlphaGroupDalek?.Units</li>
</ul>

<p>
    Dalek© <a href="https://www.imdb.com/name/nm0622334/">Terry Nation</a><br>
    Doctor Who© <a href="https://www.bbc.co.uk/programmes/b006q2x0">BBC</a>
</p>

@code {
    [CascadingParameter]
    public Dalek? Dalek { get; set; }

    [CascadingParameter(Name = "AlphaGroup")]
    public Dalek? AlphaGroupDalek { get; set; }
}
```

In the following example, `Dalek` is registered as a cascading value using CascadingValueSource<T>, where `<T>` is the type. The `isFixed` flag indicates whether the value is fixed. If false, all recipients are subscribed for update notifications, which are issued by calling NotifyChangedAsync. Subscriptions create overhead and reduce performance, so set `isFixed` to `true` if the value doesn't change.

```C#
builder.Services.AddCascadingValue(sp =>
{
    var dalek = new Dalek { Units = 789 };
    var source = new CascadingValueSource<Dalek>(dalek, isFixed: false);
    return source;
});
```

> ⚠ **Warning**
>
> Registering a component type as a root-level cascading value doesn't register additional services for the type or permit service activation in the component.
>
> Treat required services separately from cascading values, registering them separately from the cascaded type.
>
> Avoid using AddCascadingValue to register a component type as a cascading value. Instead, wrap the `<Router>...</Router>` in the `Routes` component (`Components/Routes.razor`) with the component and adopt global interactive server-side rendering (interactive SSR). For an example, see the CascadingValue component section.

`CascadingValue` component

An ancestor component provides a cascading value using the Blazor framework's CascadingValue component, which wraps a subtree of a component hierarchy and supplies a single value to all of the components within its subtree.

The following example demonstrates the flow of theme information down the component hierarchy to provide a CSS style class to buttons in child components.

The following `ThemeInfo` C# class specifies the theme information.

> **Note**
>
> For the examples in this section, the app's namespace is `BlazorSample`. When experimenting with the code in your own sample app, change the app's namespace to your sample app's namespace.

`ThemeInfo.cs`:

C#

```
namespace BlazorSample;

public class ThemeInfo
{
    public string? ButtonClass { get; set; }
}
```

The following layout component specifies theme information (`ThemeInfo`) as a cascading value for all components that make up the layout body of the Body property. `ButtonClass` is assigned a value of btn-success, which is a Bootstrap button style. Any descendent component in the component hierarchy can use the `ButtonClass` property through the `ThemeInfo` cascading value.

`MainLayout.razor`:

razor

```
@inherits LayoutComponentBase

<div class="page">
    <div class="sidebar">
        <NavMenu />
    </div>

    <main>
        <div class="top-row px-4">
            <a href="https://learn.microsoft.com/aspnet/core/" target="_blank">About</a>
        </div>

        <CascadingValue Value="theme">
            <article class="content px-4">
                @Body
            </article>
        </CascadingValue>
    </main>
</div>

<div id="blazor-error-ui" data-nosnippet>
```

```
    An unhandled error has occurred.
    <a href="" class="reload">Reload</a>
    <a class="dismiss">✕</a>
</div>

@code {
    private ThemeInfo theme = new() { ButtonClass = "btn-success" };
}
```

Blazor Web Apps provide alternative approaches for cascading values that apply more broadly to the app than furnishing them via a single layout file:

- Wrap the markup of the `Routes` component in a CascadingValue component to specify the data as a cascading value for all of the app's components.

  The following example cascades `ThemeInfo` data from the `Routes` component.

  `Routes.razor`:

  razor

  ```
  <CascadingValue Value="theme">
      <Router ...>
          ...
      </Router>
  </CascadingValue>

  @code {
      private ThemeInfo theme = new() { ButtonClass = "btn-success" };
  }
  ```

  > **Note**
  >
  > Wrapping the `Routes` component instance in the `App` component (`Components/App.razor`) with a CascadingValue component is ***not*** supported.

- Specify a *root-level cascading value* as a service by calling the AddCascadingValue extension method on the service collection builder.

  The following example cascades `ThemeInfo` data from the `Program` file.

  `Program.cs`

  C#

  ```
  builder.Services.AddCascadingValue(sp =>
      new ThemeInfo() { ButtonClass = "btn-primary" });
  ```

For more information, see the following sections of this article:

- Root-level cascading values
- Cascading values/parameters and render mode boundaries

`[CascadingParameter]` attribute

To make use of cascading values, descendent components declare cascading parameters using the [CascadingParameter] attribute. Cascading values are bound to cascading parameters **by type**. Cascading multiple values of the same type is covered in the Cascade multiple values section later in this article.

The following component binds the `ThemeInfo` cascading value to a cascading parameter, optionally using the same name of `ThemeInfo`. The parameter is used to set the CSS class for the **`Increment Counter (Themed)`** button.

`ThemedCounter.razor`:

razor

```razor
@page "/themed-counter"

<PageTitle>Themed Counter</PageTitle>

<h1>Themed Counter Example</h1>

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

<p>
    <button @onclick="IncrementCount">
        Increment Counter (Unthemed)
    </button>
</p>

<p>
    <button
        class="btn @(ThemeInfo is not null ? ThemeInfo.ButtonClass : string.Empty)"
        @onclick="IncrementCount">
        Increment Counter (Themed)
    </button>
</p>

@code {
    private int currentCount = 0;

    [CascadingParameter]
    protected ThemeInfo? ThemeInfo { get; set; }

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

Similar to a regular component parameter, components accepting a cascading parameter are rerendered when the cascading value is changed. For instance, configuring a different theme instance causes the `ThemedCounter` component from the CascadingValue component section to rerender.

`MainLayout.razor`:

razor

```razor
<main>
    <div class="top-row px-4">
        <a href="https://docs.microsoft.com/aspnet/" target="_blank">About</a>
    </div>
```

```
    <CascadingValue Value="theme">
        <article class="content px-4">
            @Body
        </article>
    </CascadingValue>
    <button @onclick="ChangeToDarkTheme">Dark mode</button>
</main>

@code {
    private ThemeInfo theme = new() { ButtonClass = "btn-success" };

    private void ChangeToDarkTheme()
    {
        theme = new() { ButtonClass = "btn-secondary" };
    }
}
```

CascadingValue<TValue>.IsFixed can be used to indicate that a cascading parameter doesn't change after initialization.

Cascading values/parameters and render mode boundaries

Cascading parameters don't pass data across render mode boundaries:

- Interactive sessions run in a different context than the pages that use static server-side rendering (static SSR). There's no requirement that the server producing the page is even the same machine that hosts some later Interactive Server session, including for WebAssembly components where the server is a different machine to the client. The benefit of static server-side rendering (static SSR) is to gain the full performance of pure stateless HTML rendering.
- State crossing the boundary between static and interactive rendering must be serializable. Components are arbitrary objects that reference a vast chain of other objects, including the renderer, the DI container, and every DI service instance. You must explicitly cause state to be serialized from static SSR to make it available in subsequent interactively-rendered components. Two approaches are adopted:
  - Via the Blazor framework, parameters passed across a static SSR to interactive rendering boundary are serialized automatically if they're JSON-serializable, or an error is thrown.
  - State stored in PersistentComponentState is serialized and recovered automatically if it's JSON-serializable, or an error is thrown.

Cascading parameters aren't JSON-serializable because the typical usage patterns for cascading parameters are somewhat like DI services. There are often platform-specific variants of cascading parameters, so it would be unhelpful to developers if the framework stopped developers from having server-interactive-specific versions or WebAssembly-specific versions. Also, many cascading parameter values in general aren't serializable, so it would be impractical to update existing apps if you had to stop using all nonserializable cascading parameter values.

Recommendations:

- If you need to make state available to all interactive components as a cascading parameter, we recommend using root-level cascading values. A factory pattern is available, and the app can emit updated values after app startup. Root-level cascading values are available to all components, including interactive components, since they're processed as DI services.
- For component library authors, you can create an extension method for library consumers similar to the following:

```
C#
```

```
builder.Services.AddLibraryCascadingParameters();
```

Instruct developers to call your extension method. This is a sound alternative to instructing them to add a `<RootComponent>` component in their `MainLayout` component.

Cascade multiple values

To cascade multiple values of the same type within the same subtree, provide a unique Name string to each CascadingValue component and their corresponding [CascadingParameter] attributes.

In the following example, two CascadingValue components cascade different instances of `CascadingType`:

razor

```
<CascadingValue Value="parentCascadeParameter1" Name="CascadeParam1">
    <CascadingValue Value="ParentCascadeParameter2" Name="CascadeParam2">
        ...
    </CascadingValue>
</CascadingValue>

@code {
    private CascadingType? parentCascadeParameter1;

    [Parameter]
    public CascadingType? ParentCascadeParameter2 { get; set; }
}
```

In a descendant component, the cascaded parameters receive their cascaded values from the ancestor component by Name:

razor

```
@code {
    [CascadingParameter(Name = "CascadeParam1")]
    protected CascadingType? ChildCascadeParameter1 { get; set; }

    [CascadingParameter(Name = "CascadeParam2")]
    protected CascadingType? ChildCascadeParameter2 { get; set; }
}
```

Pass data across a component hierarchy

Cascading parameters also enable components to pass data across a component hierarchy. Consider the following UI tab set example, where a tab set component maintains a series of individual tabs.

> **Note**
>
> For the examples in this section, the app's namespace is `BlazorSample`. When experimenting with the code in your own sample app, change the namespace to your sample app's namespace.

Create an `ITab` interface that tabs implement in a folder named `UIInterfaces`.

`UIInterfaces/ITab.cs`:

C#

```
using Microsoft.AspNetCore.Components;

namespace BlazorSample.UIInterfaces;

public interface ITab
{
    RenderFragment ChildContent { get; }
}
```

> **Note**
>
> For more information on RenderFragment, see ASP.NET Core Razor components.

The following `TabSet` component maintains a set of tabs. The tab set's `Tab` components, which are created later in this section, supply the list items (`<li>...</li>`) for the list (`<ul>...</ul>`).

Child `Tab` components aren't explicitly passed as parameters to the `TabSet`. Instead, the child `Tab` components are part of the child content of the `TabSet`. However, the `TabSet` still needs a reference each `Tab` component so that it can render the headers and the active tab. To enable this coordination without requiring additional code, the `TabSet` component *can provide itself as a cascading value* that is then picked up by the descendent `Tab` components.

`TabSet.razor`:

razor

```
@using BlazorSample.UIInterfaces

<!-- Display the tab headers -->

<CascadingValue Value="this">
    <ul class="nav nav-tabs">
        @ChildContent
    </ul>
</CascadingValue>

<!-- Display body for only the active tab -->

<div class="nav-tabs-body p-4">
    @ActiveTab?.ChildContent
</div>

@code {
    [Parameter]
    public RenderFragment? ChildContent { get; set; }

    public ITab? ActiveTab { get; private set; }

    public void AddTab(ITab tab)
    {
        if (ActiveTab is null)
        {
            SetActiveTab(tab);
        }
    }
```

```
    public void SetActiveTab(ITab tab)
    {
        if (ActiveTab != tab)
        {
            ActiveTab = tab;
            StateHasChanged();
        }
    }
}
```

Descendent `Tab` components capture the containing `TabSet` as a cascading parameter. The `Tab` components add themselves to the `TabSet` and coordinate to set the active tab.

`Tab.razor`:

razor

```
@using BlazorSample.UIInterfaces
@implements ITab

<li>
    <a @onclick="ActivateTab" class="nav-link @TitleCssClass" role="button">
        @Title
    </a>
</li>

@code {
    [CascadingParameter]
    public TabSet? ContainerTabSet { get; set; }

    [Parameter]
    public string? Title { get; set; }

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

    private string? TitleCssClass =>
        ContainerTabSet?.ActiveTab == this ? "active" : null;

    protected override void OnInitialized()
    {
        ContainerTabSet?.AddTab(this);
    }

    private void ActivateTab()
    {
        ContainerTabSet?.SetActiveTab(this);
    }
}
```

The following `ExampleTabSet` component uses the `TabSet` component, which contains three `Tab` components.

`ExampleTabSet.razor`:

razor

```
@page "/example-tab-set"

<TabSet>
    <Tab Title="First tab">
```

```
        <h4>Greetings from the first tab!</h4>

        <label>
            <input type="checkbox" @bind="showThirdTab" />
            Toggle third tab
        </label>
    </Tab>

    <Tab Title="Second tab">
        <h4>Hello from the second tab!</h4>
    </Tab>

    @if (showThirdTab)
    {
        <Tab Title="Third tab">
            <h4>Welcome to the disappearing third tab!</h4>
            <p>Toggle this tab from the first tab.</p>
        </Tab>
    }
</TabSet>

@code {
    private bool showThirdTab;
}
```

Additional resources

- Generic type support: Explicit generic types based on ancestor components
- State management: Factor out the state preservation to a common location