```
@page "/on-params-set"
@page "/on-params-set/{StartDate:datetime}"

<PageTitle>On Parameters Set</PageTitle>

<h1>On Parameters Set Example</h1>

<p>
    Pass a datetime in the URI of the browser's address bar.
    For example, add <code>/1-1-2024</code> to the address.
</p>

<p>@message</p>

@code {
    private string? message;

    [Parameter]
    public DateTime StartDate { get; set; }

    protected override void OnParametersSet()
    {
        if (StartDate == default)
        {
            StartDate = DateTime.Now;

            message = $"No start date in URL. Default value applied " +
                $"(StartDate: {StartDate}).";
        }
        else
        {
            message = $"The start date in the URL was used " +
                $"(StartDate: {StartDate}).";
        }
    }
}
```

Asynchronous work when applying parameters and property values must occur during the OnParametersSetAsync lifecycle event:

C#

```
protected override async Task OnParametersSetAsync()
{
    await ...
}
```

If a custom base class is used with custom initialization logic, call OnParametersSetAsync on the base class:

C#

```
protected override async Task OnParametersSetAsync()
{
    await ...

    await base.OnParametersSetAsync();
}
```

It isn't necessary to call ComponentBase.OnParametersSetAsync unless a custom base class is used with custom logic. For more information, see the Base class lifecycle methods section.

If event handlers are provided in developer code, unhook them on disposal. For more information, see the Component disposal with IDisposable IAsyncDisposable section.

For more information on route parameters and constraints, see ASP.NET Core Blazor routing and navigation.

For an example of implementing `SetParametersAsync` manually to improve performance in some scenarios, see ASP.NET Core Blazor performance best practices.

After component render (`OnAfterRender{Async}`)

OnAfterRender and OnAfterRenderAsync are invoked after a component has rendered interactively and the UI has finished updating (for example, after elements are added to the browser DOM). Element and component references are populated at this point. Use this stage to perform additional initialization steps with the rendered content, such as JS interop calls that interact with the rendered DOM elements. The synchronous method is called prior to the asynchronous method.

These methods aren't invoked during prerendering or static server-side rendering (static SSR) on the server because those processes aren't attached to a live browser DOM and are already complete before the DOM is updated.

For OnAfterRenderAsync, the component doesn't automatically rerender after the completion of any returned `Task` to avoid an infinite render loop.

The `firstRender` parameter for OnAfterRender and OnAfterRenderAsync:

- Is set to `true` the first time that the component instance is rendered.
- Can be used to ensure that initialization work is only performed once.

`AfterRender.razor`:

razor

```razor
@page "/after-render"
@inject ILogger<AfterRender> Logger

<PageTitle>After Render</PageTitle>

<h1>After Render Example</h1>

<p>
    <button @onclick="HandleClick">Log information (and trigger a render)
</button>
</p>

<p>Study logged messages in the console.</p>

@code {
    protected override void OnAfterRender(bool firstRender) =>
        Logger.LogInformation("firstRender = {FirstRender}", firstRender);

    private void HandleClick() => Logger.LogInformation("HandleClick
called");
}
```

The `AfterRender.razor` sample produces following output to console when the page is loaded and the button is selected:

> OnAfterRender: firstRender = True
> HandleClick called
> OnAfterRender: firstRender = False

Asynchronous work immediately after rendering must occur during the OnAfterRenderAsync lifecycle event:

C#

```
protected override async Task OnAfterRenderAsync(bool firstRender)
{
    ...
}
```

If a custom base class is used with custom initialization logic, call OnAfterRenderAsync on the base class:

C#

```
protected override async Task OnAfterRenderAsync(bool firstRender)
{
    ...

    await base.OnAfterRenderAsync(firstRender);
}
```

It isn't necessary to call ComponentBase.OnAfterRenderAsync unless a custom base class is used with custom logic. For more information, see the Base class lifecycle methods section.

Even if you return a Task from OnAfterRenderAsync, the framework doesn't schedule a further render cycle for your component once that task completes. This is to avoid an infinite render loop. This is different from the other lifecycle methods, which schedule a further render cycle once a returned Task completes.

OnAfterRender and OnAfterRenderAsync *aren't called during the prerendering process on the server*. The methods are called when the component is rendered interactively after prerendering. When the app prerenders:

1. The component executes on the server to produce some static HTML markup in the HTTP response. During this phase, OnAfterRender and OnAfterRenderAsync aren't called.
2. When the Blazor script (`blazor.{server|webassembly|web}.js`) starts in the browser, the component is restarted in an interactive rendering mode. After a component is restarted, OnAfterRender and OnAfterRenderAsync **are** called because the app isn't in the prerendering phase any longer.

If event handlers are provided in developer code, unhook them on disposal. For more information, see the Component disposal with IDisposable IAsyncDisposable section.

Base class lifecycle methods

When overriding Blazor's lifecycle methods, it isn't necessary to call base class lifecycle methods for ComponentBase. However, a component should call an overridden base class lifecycle method in the following situations:

- When overriding ComponentBase.SetParametersAsync, `await base.SetParametersAsync(parameters);` is usually invoked because the base class method calls other lifecycle methods and triggers rendering in a complex fashion. For more information, see the When parameters are set (SetParametersAsync) section.

- If the base class method contains logic that must be executed. Library consumers usually call base class lifecycle methods when inheriting a base class because library base classes often have custom lifecycle logic to execute. If the app uses a base class from a library, consult the library's documentation for guidance.

In the following example, `base.OnInitialized();` is called to ensure that the base class's `OnInitialized` method is executed. Without the call, `BlazorRocksBase2.OnInitialized` doesn't execute.

`BlazorRocks2.razor`:

razor

```
@page "/blazor-rocks-2"
@inherits BlazorRocksBase2
@inject ILogger<BlazorRocks2> Logger

<PageTitle>Blazor Rocks!</PageTitle>

<h1>Blazor Rocks! Example 2</h1>

<p>
    @BlazorRocksText
</p>

@code {
    protected override void OnInitialized()
    {
        Logger.LogInformation("Initialization code of BlazorRocks2
executed!");

        base.OnInitialized();
    }
}
```

`BlazorRocksBase2.cs`:

C#

```
using Microsoft.AspNetCore.Components;

namespace BlazorSample;

public class BlazorRocksBase2 : ComponentBase
{
    [Inject]
    private ILogger<BlazorRocksBase2> Logger { get; set; } = default!;

    public string BlazorRocksText { get; set; } = "Blazor rocks the
browser!";
```

```
    protected override void OnInitialized() =>
        Logger.LogInformation("Initialization code of BlazorRocksBase2
executed!");
}
```

State changes (`StateHasChanged`)

StateHasChanged notifies the component that its state has changed. When applicable, calling StateHasChanged enqueues a rerender that occurs when the app's main thread is free.

StateHasChanged is called automatically for EventCallback methods. For more information on event callbacks, see ASP.NET Core Blazor event handling.

For more information on component rendering and when to call StateHasChanged, including when to invoke it with ComponentBase.InvokeAsync, see ASP.NET Core Razor component rendering.

Handle incomplete asynchronous actions at render

Asynchronous actions performed in lifecycle events might not have completed before the component is rendered. Objects might be `null` or incompletely populated with data while the lifecycle method is executing. Provide rendering logic to confirm that objects are initialized. Render placeholder UI elements (for example, a loading message) while objects are `null`.

In the following `Slow` component, OnInitializedAsync is overridden to asynchronously execute a long-running task. While `isLoading` is `true`, a loading message is displayed to the user. After the `Task` returned by OnInitializedAsync completes, the component is rerendered with the updated state, showing the "`Finished!`" message.

`Slow.razor`:

razor

```
@page "/slow"

<h2>Slow Component</h2>

@if (isLoading)
{
```

```
    <div><em>Loading...</em></div>
}
else
{
    <div>Finished!</div>
}

@code {
    private bool isLoading = true;

    protected override async Task OnInitializedAsync()
    {
        await LoadDataAsync();
        isLoading = false;
    }

    private Task LoadDataAsync()
    {
        return Task.Delay(10000);
    }
}
```

The preceding component uses an `isLoading` variable to display the loading message. A similar approach is used for a component that loads data into a collection and checks if the collection is `null` to present the loading message. The following example checks the `movies` collection for `null` to either display the loading message or display the collection of movies:

razor

```
@if (movies == null)
{
    <p><em>Loading...</em></p>
}
else
{
    @* display movies *@
}

@code {
    private Movies[]? movies;

    protected override async Task OnInitializedAsync()
    {
        movies = await GetMovies();
    }
}
```

Prerendering waits for *quiescence*, which means that a component doesn't render until all of the components in the render tree have finished rendering. This means that a

loading message doesn't display while a child component's OnInitializedAsync method is executing a long-running task during prerendering. To demonstrate this behavior, place the preceding `Slow` component into a test app's `Home` component:

```razor
@page "/"

<PageTitle>Home</PageTitle>

<h1>Hello, world!</h1>

Welcome to your new app.

<SlowComponent />
```

> **Note**
>
> Although the examples in this section discuss the **OnInitializedAsync** lifecycle method, other lifecycle methods that execute during prerendering may delay final rendering of a component. Only **OnAfterRender{Async}** isn't executed during prerendering and is immune to delays due to quiescence.

During prerendering, the `Home` component doesn't render until the `Slow` component is rendered, which takes ten seconds. The UI is blank during this ten-second period, and there's no loading message. After prerendering, the `Home` component renders, and the `Slow` component's loading message is displayed. After ten more seconds, the `Slow` component finally displays the finished message.

As the preceding demonstration illustrates, quiescence during prerendering results in a poor user experience. To improve the user experience, begin by implementing streaming rendering to avoid waiting for the asynchronous task to complete while prerendering.

Add the [StreamRendering] attribute to the `Slow` component (use `[StreamRendering(true)]` in .NET 8):

```razor
@attribute [StreamRendering]
```

When the `Home` component is prerendering, the `Slow` component is quickly rendered with its loading message. The `Home` component doesn't wait for ten seconds for the `Slow` component to finish rendering. However, the finished message displayed at the

end of prerendering is replaced by the loading message while the component finally renders, which is another ten-second delay. This occurs because the `Slow` component is rendering twice, along with `LoadDataAsync` executing twice. When a component is accessing resources, such as services and databases, double execution of service calls and database queries creates undesirable load on the app's resources.

To address the double rendering of the loading message and the re-execution of service and database calls, persist prerendered state with PersistentComponentState for final rendering of the component, as seen in the following updates to the `Slow` component:

razor

```
@page "/slow"
@attribute [StreamRendering]
@implements IDisposable
@inject PersistentComponentState ApplicationState

<h2>Slow Component</h2>

@if (data is null)
{
    <div><em>Loading...</em></div>
}
else
{
    <div>@data</div>
}

@code {
    private string? data;
    private PersistingComponentStateSubscription persistingSubscription;

    protected override async Task OnInitializedAsync()
    {
        persistingSubscription =
            ApplicationState.RegisterOnPersisting(PersistData);

        if (!ApplicationState.TryTakeFromJson<string>("data", out var
restored))
        {
            data = await LoadDataAsync();
        }
        else
        {
            data = restored!;
        }
    }

    private Task PersistData()
    {
        ApplicationState.PersistAsJson("data", data);
```

```
        return Task.CompletedTask;
    }

    private async Task<string> LoadDataAsync()
    {
        await Task.Delay(10000);
        return "Finished!";
    }

    void IDisposable.Dispose()
    {
        persistingSubscription.Dispose();
    }
}
```

By combining streaming rendering with persistent component state:

- Services and databases only require a single call for component initialization.
- Components render their UIs quickly with loading messages during long-running tasks for the best user experience.

For more information, see the following resources:

- ASP.NET Core Razor component rendering
- Prerender ASP.NET Core Razor components.

Handle errors

For information on handling errors during lifecycle method execution, see Handle errors in ASP.NET Core Blazor apps.

Stateful reconnection after prerendering

When prerendering on the server, a component is initially rendered statically as part of the page. Once the browser establishes a SignalR connection back to the server, the component is rendered *again* and interactive. If the OnInitialized{Async} lifecycle method for initializing the component is present, the method is executed *twice*:

- When the component is prerendered statically.
- After the server connection has been established.

This can result in a noticeable change in the data displayed in the UI when the component is finally rendered. To avoid this behavior, pass in an identifier to cache the state during prerendering and to retrieve the state after prerendering.

The following code demonstrates a `WeatherForecastService` that avoids the change in data display due to prerendering. The awaited Delay (`await Task.Delay(...)`) simulates a short delay before returning data from the `GetForecastAsync` method.

Add IMemoryCache services with AddMemoryCache on the service collection in the app's `Program` file:

C#

```
builder.Services.AddMemoryCache();
```

`WeatherForecastService.cs`:

C#

```
using Microsoft.Extensions.Caching.Memory;

namespace BlazorSample;

public class WeatherForecastService(IMemoryCache memoryCache)
{
    private static readonly string[] summaries =
    [
        "Freezing", "Bracing", "Chilly", "Cool", "Mild",
        "Warm", "Balmy", "Hot", "Sweltering", "Scorching"
    ];

    public IMemoryCache MemoryCache { get; } = memoryCache;

    public Task<WeatherForecast[]?> GetForecastAsync(DateOnly startDate)
    {
        return MemoryCache.GetOrCreateAsync(startDate, async e =>
        {
            e.SetOptions(new MemoryCacheEntryOptions
            {
                AbsoluteExpirationRelativeToNow =
                    TimeSpan.FromSeconds(30)
            });

            await Task.Delay(TimeSpan.FromSeconds(10));

            return Enumerable.Range(1, 5).Select(index => new
WeatherForecast
            {
                Date = startDate.AddDays(index),
                TemperatureC = Random.Shared.Next(-20, 55),
                Summary = summaries[Random.Shared.Next(summaries.Length)]
            }).ToArray();
        });
```

```
    }
}
```

For more information on the RenderMode, see ASP.NET Core Blazor SignalR guidance.

The content in this section focuses on Blazor Web Apps and stateful SignalR *reconnection*. To preserve state during the execution of initialization code while prerendering, see Prerender ASP.NET Core Razor components.

Prerendering with JavaScript interop

This section applies to server-side apps that prerender Razor components. Prerendering is covered in Prerender ASP.NET Core Razor components.

> **Note**
>
> Internal navigation for **interactive routing** in Blazor Web Apps doesn't involve requesting new page content from the server. Therefore, prerendering doesn't occur for internal page requests. If the app adopts interactive routing, perform a full page reload for component examples that demonstrate prerendering behavior. For more information, see **Prerender ASP.NET Core Razor components**.

During prerendering, calling into JavaScript (JS) isn't possible. The following example demonstrates how to use JS interop as part of a component's initialization logic in a way that's compatible with prerendering.

The following `scrollElementIntoView` function:

- Scrolls to the passed element with scrollIntoView.
- Returns the element's `top` property value from the getBoundingClientRect method.

JavaScript

```
window.scrollElementIntoView = (element) => {
  element.scrollIntoView();
  return element.getBoundingClientRect().top;
}
```

Where IJSRuntime.InvokeAsync calls the JS function in component code, the ElementReference is only used in OnAfterRenderAsync and not in any earlier lifecycle method because there's no HTML DOM element until after the component is rendered.

StateHasChanged (reference source) is called to enqueue rerendering of the component with the new state obtained from the JS interop call (for more information, see ASP.NET Core Razor component rendering). An infinite loop isn't created because StateHasChanged is only called when `scrollPosition` is `null`.

`PrerenderedInterop.razor`:

razor

```razor
@page "/prerendered-interop"
@using Microsoft.AspNetCore.Components
@using Microsoft.JSInterop
@inject IJSRuntime JS

<PageTitle>Prerendered Interop</PageTitle>

<h1>Prerendered Interop Example</h1>

<div @ref="divElement" style="margin-top:2000px">
    Set value via JS interop call: <strong>@scrollPosition</strong>
</div>

@code {
    private ElementReference divElement;
    private double? scrollPosition;

    protected override async Task OnAfterRenderAsync(bool firstRender)
    {
        if (firstRender && scrollPosition is null)
        {
            scrollPosition = await JS.InvokeAsync<double>(
                "scrollElementIntoView", divElement);

            StateHasChanged();
        }
    }
}
```

The preceding example pollutes the client with a global function. For a better approach in production apps, see JavaScript isolation in JavaScript modules.

Component disposal with `IDisposable` and `IAsyncDisposable`

If a component implements IDisposable or IAsyncDisposable, the framework calls for resource disposal when the component is removed from the UI. Don't rely on the exact timing of when these methods are executed. For example, IAsyncDisposable can be

triggered before or after an asynchronous Task awaited in OnInitalizedAsync is called or completes. Also, object disposal code shouldn't assume that objects created during initialization or other lifecycle methods exist.

Components shouldn't need to implement IDisposable and IAsyncDisposable simultaneously. If both are implemented, the framework only executes the asynchronous overload.

Developer code must ensure that IAsyncDisposable implementations don't take a long time to complete.

Disposal of JavaScript interop object references

Examples throughout the JavaScript (JS) interop articles demonstrate typical object disposal patterns:

- When calling JS from .NET, as described in Call JavaScript functions from .NET methods in ASP.NET Core Blazor, dispose any created IJSObjectReference/IJSInProcessObjectReference/JSObjectReference either from .NET or from JS to avoid leaking JS memory.
- When calling .NET from JS, as described in Call .NET methods from JavaScript functions in ASP.NET Core Blazor, dispose of a created DotNetObjectReference either from .NET or from JS to avoid leaking .NET memory.

JS interop object references are implemented as a map keyed by an identifier on the side of the JS interop call that creates the reference. When object disposal is initiated from either the .NET or JS side, Blazor removes the entry from the map, and the object can be garbage collected as long as no other strong reference to the object is present.

At a minimum, always dispose objects created on the .NET side to avoid leaking .NET managed memory.

DOM cleanup tasks during component disposal

For more information, see ASP.NET Core Blazor JavaScript interoperability (JS interop).

For guidance on JSDisconnectedException when a circuit is disconnected, see ASP.NET Core Blazor JavaScript interoperability (JS interop). For general JavaScript interop error handling guidance, see the *JavaScript interop* section in Handle errors in ASP.NET Core Blazor apps.

Synchronous `IDisposable`

For synchronous disposal tasks, use IDisposable.Dispose.

The following component:

- Implements IDisposable with the @implements Razor directive.
- Disposes of `obj`, which is a type that implements IDisposable.
- A null check is performed because `obj` is created in a lifecycle method (not shown).

razor

```
@implements IDisposable

...

@code {
    ...

    public void Dispose()
    {
        obj?.Dispose();
    }
}
```

If a single object requires disposal, a lambda can be used to dispose of the object when Dispose is called. The following example appears in the ASP.NET Core Razor component rendering article and demonstrates the use of a lambda expression for the disposal of a Timer.

`TimerDisposal1.razor`:

razor

```
@page "/timer-disposal-1"
@using System.Timers
@implements IDisposable

<PageTitle>Timer Disposal 1</PageTitle>

<h1>Timer Disposal Example 1</h1>

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

@code {
    private int currentCount = 0;
    private Timer timer = new(1000);

    protected override void OnInitialized()
    {
```

```
        timer.Elapsed += (sender, eventArgs) => OnTimerCallback();
        timer.Start();
    }

    private void OnTimerCallback()
    {
        _ = InvokeAsync(() =>
        {
            currentCount++;
            StateHasChanged();
        });
    }

    public void Dispose() => timer.Dispose();
}
```

> **Note**
>
> In the preceding example, the call to StateHasChanged is wrapped by a call to ComponentBase.InvokeAsync because the callback is invoked outside of Blazor's synchronization context. For more information, see ASP.NET Core Razor component rendering.

If the object is created in a lifecycle method, such as OnInitialized{Async}, check for `null` before calling `Dispose`.

`TimerDisposal2.razor`:

razor

```
@page "/timer-disposal-2"
@using System.Timers
@implements IDisposable

<PageTitle>Timer Disposal 2</PageTitle>

<h1>Timer Disposal Example 2</h1>

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

@code {
    private int currentCount = 0;
    private Timer? timer;

    protected override void OnInitialized()
    {
        timer = new Timer(1000);
        timer.Elapsed += (sender, eventArgs) => OnTimerCallback();
        timer.Start();
```

```
    }

    private void OnTimerCallback()
    {
        _ = InvokeAsync(() =>
        {
            currentCount++;
            StateHasChanged();
        });
    }

    public void Dispose() => timer?.Dispose();
}
```

For more information, see:

- Cleaning up unmanaged resources (.NET documentation)
- Null-conditional operators ?. and ?[]

Asynchronous `IAsyncDisposable`

For asynchronous disposal tasks, use IAsyncDisposable.DisposeAsync.

The following component:

- Implements IAsyncDisposable with the @implements Razor directive.
- Disposes of `obj`, which is an unmanaged type that implements IAsyncDisposable.
- A null check is performed because `obj` is created in a lifecycle method (not shown).

razor

```
@implements IAsyncDisposable

...

@code {
    ...

    public async ValueTask DisposeAsync()
    {
        if (obj is not null)
        {
            await obj.DisposeAsync();
        }
    }
}
```

For more information, see:

- Cleaning up unmanaged resources (.NET documentation)
- Null-conditional operators ?. and ?[]

Assignment of `null` to disposed objects

Usually, there's no need to assign `null` to disposed objects after calling Dispose/DisposeAsync. Rare cases for assigning `null` include the following:

- If the object's type is poorly implemented and doesn't tolerate repeat calls to Dispose/DisposeAsync, assign `null` after disposal to gracefully skip further calls to Dispose/DisposeAsync.
- If a long-lived process continues to hold a reference to a disposed object, assigning `null` allows the garbage collector to free the object in spite of the long-lived process holding a reference to it.

These are unusual scenarios. For objects that are implemented correctly and behave normally, there's no point in assigning `null` to disposed objects. In the rare cases where an object must be assigned `null`, we recommend documenting the reason and seeking a solution that prevents the need to assign `null`.

`StateHasChanged`

> ⓘ Note
>
> Calling StateHasChanged in `Dispose` and `DisposeAsync` isn't supported. StateHasChanged might be invoked as part of tearing down the renderer, so requesting UI updates at that point isn't supported.

Event handlers

Always unsubscribe event handlers from .NET events. The following Blazor form examples show how to unsubscribe an event handler in the `Dispose` method:

- Private field and lambda approach

```razor
@implements IDisposable
```

```
<EditForm ... EditContext="editContext" ...>
    ...
    <button type="submit" disabled="@formInvalid">Submit</button>
</EditForm>

@code {
    ...

    private EventHandler<FieldChangedEventArgs>? fieldChanged;

    protected override void OnInitialized()
    {
        editContext = new(model);

        fieldChanged = (_, __) =>
        {
            ...
        };

        editContext.OnFieldChanged += fieldChanged;
    }

    public void Dispose()
    {
        editContext.OnFieldChanged -= fieldChanged;
    }
}
```

- Private method approach

razor

```
@implements IDisposable

<EditForm ... EditContext="editContext" ...>
    ...
    <button type="submit" disabled="@formInvalid">Submit</button>
</EditForm>

@code {
    ...

    protected override void OnInitialized()
    {
        editContext = new(model);
        editContext.OnFieldChanged += HandleFieldChanged;
    }

    private void HandleFieldChanged(object sender,
FieldChangedEventArgs e)
    {
        ...
    }
```

```
    public void Dispose()
    {
        editContext.OnFieldChanged -= HandleFieldChanged;
    }
}
```

For more information, see the Component disposal with IDisposable and IAsyncDisposable section.

For more information on the EditForm component and forms, see ASP.NET Core Blazor forms overview and the other forms articles in the *Forms* node.

Anonymous functions, methods, and expressions

When anonymous functions, methods, or expressions, are used, it isn't necessary to implement IDisposable and unsubscribe delegates. However, failing to unsubscribe a delegate is a problem **when the object exposing the event outlives the lifetime of the component registering the delegate**. When this occurs, a memory leak results because the registered delegate keeps the original object alive. Therefore, only use the following approaches when you know that the event delegate disposes quickly. When in doubt about the lifetime of objects that require disposal, subscribe a delegate method and properly dispose the delegate as the earlier examples show.

- Anonymous lambda method approach (explicit disposal not required):

```
C#

private void HandleFieldChanged(object sender, FieldChangedEventArgs e)
{
    formInvalid = !editContext.Validate();
    StateHasChanged();
}

protected override void OnInitialized()
{
    editContext = new(starship);
    editContext.OnFieldChanged += (s, e) =>
HandleFieldChanged((editContext)s, e);
}
```

- Anonymous lambda expression approach (explicit disposal not required):

```
C#

private ValidationMessageStore? messageStore;
```

```
[CascadingParameter]
private EditContext? CurrentEditContext { get; set; }

protected override void OnInitialized()
{
    ...

    messageStore = new(CurrentEditContext);

    CurrentEditContext.OnValidationRequested += (s, e) =>
messageStore.Clear();
    CurrentEditContext.OnFieldChanged += (s, e) =>
        messageStore.Clear(e.FieldIdentifier);
}
```

The full example of the preceding code with anonymous lambda expressions appears in the ASP.NET Core Blazor forms validation article.

For more information, see Cleaning up unmanaged resources and the topics that follow it on implementing the `Dispose` and `DisposeAsync` methods.

Disposal during JS interop

Trap JSDisconnectedException in potential cases where loss of Blazor's SignalR circuit prevents JS interop calls and results an unhandled exception.

For more information, see the following resources:

- JavaScript isolation in JavaScript modules
- JavaScript interop calls without a circuit

Cancelable background work

Components often perform long-running background work, such as making network calls (HttpClient) and interacting with databases. It's desirable to stop the background work to conserve system resources in several situations. For example, background asynchronous operations don't automatically stop when a user navigates away from a component.

Other reasons why background work items might require cancellation include:

- An executing background task was started with faulty input data or processing parameters.
- The current set of executing background work items must be replaced with a new set of work items.

- The priority of currently executing tasks must be changed.
- The app must be shut down for server redeployment.
- Server resources become limited, necessitating the rescheduling of background work items.

To implement a cancelable background work pattern in a component:

- Use a CancellationTokenSource and CancellationToken.
- On disposal of the component and at any point cancellation is desired by manually canceling the token, call CancellationTokenSource.Cancel to signal that the background work should be cancelled.
- After the asynchronous call returns, call ThrowIfCancellationRequested on the token.

In the following example:

- `await Task.Delay(10000, cts.Token);` represents long-running asynchronous background work.
- `BackgroundResourceMethod` represents a long-running background method that shouldn't start if the `Resource` is disposed before the method is called.

`BackgroundWork.razor`:

razor

```
@page "/background-work"
@implements IDisposable
@inject ILogger<BackgroundWork> Logger

<PageTitle>Background Work</PageTitle>

<h1>Background Work Example</h1>

<p>
    <button @onclick="LongRunningWork">Trigger long running work</button>
    <button @onclick="Dispose">Trigger Disposal</button>
</p>
<p>Study logged messages in the console.</p>
<p>
    If you trigger disposal within 10 seconds of page load, the
    <code>BackgroundResourceMethod</code> isn't executed.
</p>
<p>
    If disposal occurs after <code>BackgroundResourceMethod</code> is called
but
    before action is taken on the resource, an
<code>ObjectDisposedException</code>
    is thrown by <code>BackgroundResourceMethod</code>, and the resource
isn't
```

```
    processed.
</p>

@code {
    private Resource resource = new();
    private CancellationTokenSource cts = new();
    private IList<string> messages = [];

    protected async Task LongRunningWork()
    {
        Logger.LogInformation("Long running work started");

        await Task.Delay(10000, cts.Token);

        cts.Token.ThrowIfCancellationRequested();
        resource.BackgroundResourceMethod(Logger);
    }

    public void Dispose()
    {
        Logger.LogInformation("Executing Dispose");

        if (!cts.IsCancellationRequested)
        {
            cts.Cancel();
        }

        cts?.Dispose();
        resource?.Dispose();
    }

    private class Resource : IDisposable
    {
        private bool disposed;

        public void BackgroundResourceMethod(ILogger<BackgroundWork> logger)
        {
            logger.LogInformation("BackgroundResourceMethod: Start method");

            if (disposed)
            {
                logger.LogInformation("BackgroundResourceMethod: Disposed");
                throw new ObjectDisposedException(nameof(Resource));
            }

            // Take action on the Resource

            logger.LogInformation("BackgroundResourceMethod: Action on
Resource");
        }

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

Blazor Server reconnection events

The component lifecycle events covered in this article operate separately from server-side reconnection event handlers. When the SignalR connection to the client is lost, only UI updates are interrupted. UI updates are resumed when the connection is re-established. For more information on circuit handler events and configuration, see ASP.NET Core Blazor SignalR guidance.

Additional resources

Handle caught exceptions outside of a Razor component's lifecycle

ASP.NET Core Razor component virtualization

Article • 10/18/2024

> **ⓘ Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the .NET 9 version of this article.

This article explains how to use component virtualization in ASP.NET Core Blazor apps.

Virtualization

Improve the perceived performance of component rendering using the Blazor framework's built-in virtualization support with the Virtualize<TItem> component. Virtualization is a technique for limiting UI rendering to just the parts that are currently visible. For example, virtualization is helpful when the app must render a long list of items and only a subset of items is required to be visible at any given time.

Use the Virtualize<TItem> component when:

- Rendering a set of data items in a loop.
- Most of the items aren't visible due to scrolling.
- The rendered items are the same size.

When the user scrolls to an arbitrary point in the Virtualize<TItem> component's list of items, the component calculates the visible items to show. Unseen items aren't rendered.

Without virtualization, a typical list might use a C# foreach loop to render each item in a list. In the following example:

- `allFlights` is a collection of airplane flights.
- The `FlightSummary` component displays details about each flight.
- The @key directive attribute preserves the relationship of each `FlightSummary` component to its rendered flight by the flight's `FlightId`.

razor

```razor
<div style="height:500px;overflow-y:scroll">
    @foreach (var flight in allFlights)
    {
        <FlightSummary @key="flight.FlightId" Details="@flight.Summary" />
    }
</div>
```

If the collection contains thousands of flights, rendering the flights takes a long time and users experience a noticeable UI lag. Most of the flights fall outside of the height of the `<div>` element, so most of them aren't seen.

Instead of rendering the entire list of flights at once, replace the foreach loop in the preceding example with the Virtualize<TItem> component:

- Specify `allFlights` as a fixed item source to Virtualize<TItem>.Items. Only the currently visible flights are rendered by the Virtualize<TItem> component.

  If a non-generic collection supplies the items, for example a collection of DataRow, follow the guidance in the Item provider delegate section to supply the items.

- Specify a context for each flight with the `Context` parameter. In the following example, `flight` is used as the context, which provides access to each flight's members.

razor

```razor
<div style="height:500px;overflow-y:scroll">
    <Virtualize Items="allFlights" Context="flight">
        <FlightSummary @key="flight.FlightId" Details="@flight.Summary" />
    </Virtualize>
</div>
```

If a context isn't specified with the `Context` parameter, use the value of `context` in the item content template to access each flight's members:

razor

```razor
<div style="height:500px;overflow-y:scroll">
    <Virtualize Items="allFlights">
        <FlightSummary @key="context.FlightId" Details="@context.Summary" />
    </Virtualize>
</div>
```

The Virtualize<TItem> component:

- Calculates the number of items to render based on the height of the container and the size of the rendered items.
- Recalculates and rerenders the items as the user scrolls.
- Only fetches the slice of records from an external API that correspond to the currently visible region, including overscan, when `ItemsProvider` is used instead of `Items` (see the Item provider delegate section).

The item content for the Virtualize<TItem> component can include:

- Plain HTML and Razor code, as the preceding example shows.
- One or more Razor components.
- A mix of HTML/Razor and Razor components.

Item provider delegate

If you don't want to load all of the items into memory or the collection isn't a generic ICollection<T>, you can specify an items provider delegate method to the component's Virtualize<TItem>.ItemsProvider parameter that asynchronously retrieves the requested items on demand. In the following example, the `LoadEmployees` method provides the items to the Virtualize<TItem> component:

razor

```
<Virtualize Context="employee" ItemsProvider="LoadEmployees">
    <p>
        @employee.FirstName @employee.LastName has the
        job title of @employee.JobTitle.
    </p>
</Virtualize>
```

The items provider receives an ItemsProviderRequest, which specifies the required number of items starting at a specific start index. The items provider then retrieves the requested items from a database or other service and returns them as an ItemsProviderResult<TItem> along with a count of the total items. The items provider can choose to retrieve the items with each request or cache them so that they're readily available.

A Virtualize<TItem> component can only accept **one item source** from its parameters, so don't attempt to simultaneously use an items provider and assign a collection to `Items`. If both are assigned, an InvalidOperationException is thrown when the component's parameters are set at runtime.

The following example loads employees from an `EmployeeService` (not shown):

C#

```
private async ValueTask<ItemsProviderResult<Employee>> LoadEmployees(
    ItemsProviderRequest request)
{
    var numEmployees = Math.Min(request.Count, totalEmployees -
request.StartIndex);
    var employees = await
EmployeesService.GetEmployeesAsync(request.StartIndex,
        numEmployees, request.CancellationToken);

    return new ItemsProviderResult<Employee>(employees, totalEmployees);
}
```

In the following example, a collection of DataRow is a non-generic collection, so an items provider delegate is used for virtualization:

razor

```
<Virtualize Context="row" ItemsProvider="GetRows">
    ...
</Virtualize>

@code{
    ...

    private ValueTask<ItemsProviderResult<DataRow>>
GetRows(ItemsProviderRequest request) =>
        new(new ItemsProviderResult<DataRow>(
            dataTable.Rows.OfType<DataRow>
().Skip(request.StartIndex).Take(request.Count),
            dataTable.Rows.Count));
}
```

Virtualize<TItem>.RefreshDataAsync instructs the component to rerequest data from its ItemsProvider. This is useful when external data changes. There's usually no need to call RefreshDataAsync when using Items.

RefreshDataAsync updates a Virtualize<TItem> component's data without causing a rerender. If RefreshDataAsync is invoked from a Blazor event handler or component lifecycle method, triggering a render isn't required because a render is automatically triggered at the end of the event handler or lifecycle method. If RefreshDataAsync is triggered separately from a background task or event, such as in the following `ForecastUpdated` delegate, call StateHasChanged to update the UI at the end of the background task or event:

C#

```
<Virtualize ... @ref="virtualizeComponent">
    ...
</Virtualize>

...

private Virtualize<FetchData>? virtualizeComponent;

protected override void OnInitialized()
{
    WeatherForecastSource.ForecastUpdated += async () =>
    {
        await InvokeAsync(async () =>
        {
            await virtualizeComponent?.RefreshDataAsync();
            StateHasChanged();
        });
    });
}
```

In the preceding example:

- RefreshDataAsync is called first to obtain new data for the Virtualize<TItem> component.
- `StateHasChanged` is called to rerender the component.

Placeholder

Because requesting items from a remote data source might take some time, you have the option to render a placeholder with item content:

- Use a Placeholder (`<Placeholder>...</Placeholder>`) to display content until the item data is available.
- Use Virtualize<TItem>.ItemContent to set the item template for the list.

razor

```
<Virtualize Context="employee" ItemsProvider="LoadEmployees">
    <ItemContent>
        <p>
            @employee.FirstName @employee.LastName has the
            job title of @employee.JobTitle.
        </p>
    </ItemContent>
    <Placeholder>
        <p>
            Loading&hellip;
        </p>
```

```
    </Placeholder>
</Virtualize>
```

Empty content

Use the EmptyContent parameter to supply content when the component has loaded and either Items is empty or ItemsProviderResult<TItem>.TotalItemCount is zero.

`EmptyContent.razor`:

razor

```razor
@page "/empty-content"

<PageTitle>Empty Content</PageTitle>

<h1>Empty Content Example</h1>

<Virtualize Items="stringList">
    <ItemContent>
        <p>
            @context
        </p>
    </ItemContent>
    <EmptyContent>
        <p>
            There are no strings to display.
        </p>
    </EmptyContent>
</Virtualize>

@code {
    private List<string>? stringList;

    protected override void OnInitialized() => stringList ??= new();
}
```

Change the `OnInitialized` method lambda to see the component display strings:

C#

```
protected override void OnInitialized() =>
    stringList ??= new() { "Here's a string!", "Here's another string!" };
```

Item size

The height of each item in pixels can be set with Virtualize<TItem>.ItemSize (default: 50). The following example changes the height of each item from the default of 50 pixels to 25 pixels:

```razor
<Virtualize Context="employee" Items="employees" ItemSize="25">
    ...
</Virtualize>
```

The Virtualize<TItem> component measures the rendering size (height) of individual items *after* the initial render occurs. Use ItemSize to provide an exact item size in advance to assist with accurate initial render performance and to ensure the correct scroll position for page reloads. If the default ItemSize causes some items to render outside of the currently visible view, a second rerender is triggered. To correctly maintain the browser's scroll position in a virtualized list, the initial render must be correct. If not, users might view the wrong items.

Overscan count

Virtualize<TItem>.OverscanCount determines how many additional items are rendered before and after the visible region. This setting helps to reduce the frequency of rendering during scrolling. However, higher values result in more elements rendered in the page (default: 3). The following example changes the overscan count from the default of three items to four items:

```razor
<Virtualize Context="employee" Items="employees" OverscanCount="4">
    ...
</Virtualize>
```

State changes

When making changes to items rendered by the Virtualize<TItem> component, call StateHasChanged to enqueue re-evaluation and rerendering of the component. For more information, see ASP.NET Core Razor component rendering.

Keyboard scroll support

To allow users to scroll virtualized content using their keyboard, ensure that the virtualized elements or scroll container itself is focusable. If you fail to take this step, keyboard scrolling doesn't work in Chromium-based browsers.

For example, you can use a `tabindex` attribute on the scroll container:

razor

```
<div style="height:500px; overflow-y:scroll" tabindex="-1">
    <Virtualize Items="allFlights">
        <div class="flight-info">...</div>
    </Virtualize>
</div>
```

To learn more about the meaning of `tabindex` value `-1`, `0`, or other values, see tabindex (MDN documentation).

Advanced styles and scroll detection

The Virtualize<TItem> component is only designed to support specific element layout mechanisms. To understand which element layouts work correctly, the following explains how `Virtualize` detects which elements should be visible for display in the correct place.

If your source code looks like the following:

razor

```
<div style="height:500px; overflow-y:scroll" tabindex="-1">
    <Virtualize Items="allFlights" ItemSize="100">
        <div class="flight-info">Flight @context.Id</div>
    </Virtualize>
</div>
```

At runtime, the Virtualize<TItem> component renders a DOM structure similar to the following:

HTML

```
<div style="height:500px; overflow-y:scroll" tabindex="-1">
    <div style="height:1100px"></div>
    <div class="flight-info">Flight 12</div>
    <div class="flight-info">Flight 13</div>
    <div class="flight-info">Flight 14</div>
    <div class="flight-info">Flight 15</div>
    <div class="flight-info">Flight 16</div>
```

```
    <div style="height:3400px"></div>
</div>
```

The actual number of rows rendered and the size of the spacers vary according to your styling and `Items` collection size. However, notice that there are spacer `div` elements injected before and after your content. These serve two purposes:

- To provide an offset before and after your content, causing currently-visible items to appear at the correct location in the scroll range and the scroll range itself to represent the total size of all content.
- To detect when the user is scrolling beyond the current visible range, meaning that different content must be rendered.

> ⓘ **Note**
>
> To learn how to control the spacer HTML element tag, see the **Control the spacer element tag name** section later in this article.

The spacer elements internally use an Intersection Observer to receive notification when they're becoming visible. `Virtualize` depends on receiving these events.

`Virtualize` works under the following conditions:

- **All rendered content items, including placeholder content, are of identical height.** This makes it possible to calculate which content corresponds to a given scroll position without first fetching every data item and rendering the data into a DOM element.

- **Both the spacers and the content rows are rendered in a single vertical stack with every item filling the entire horizontal width.** In typical use cases, `Virtualize` works with `div` elements. If you're using CSS to create a more advanced layout, bear in mind the following requirements:
  - Scroll container styling requires a `display` with any of the following values:
    - `block` (the default for a `div`).
    - `table-row-group` (the default for a `tbody`).
    - `flex` with `flex-direction` set to `column`. Ensure that immediate children of the Virtualize<TItem> component don't shrink under flex rules. For example, add `.mycontainer > div { flex-shrink: 0 }`.
  - Content row styling requires a `display` with either of the following values:
    - `block` (the default for a `div`).
    - `table-row` (the default for a `tr`).

- Don't use CSS to interfere with the layout for the spacer elements. The spacer elements have a `display` value of `block`, except if the parent is a table row group, in which case they default to `table-row`. Don't try to influence spacer element width or height, including by causing them to have a border or `content` pseudo-elements.

Any approach that stops the spacers and content elements from rendering as a single vertical stack, or causes the content items to vary in height, prevents correct functioning of the Virtualize<TItem> component.

Root-level virtualization

The Virtualize<TItem> component supports using the document itself as the scroll root, as an alternative to having some other element with `overflow-y: scroll`. In the following example, the `<html>` or `<body>` elements are styled in a component with `overflow-y: scroll`:

razor

```
<HeadContent>
    <style>
        html, body { overflow-y: scroll }
    </style>
</HeadContent>
```

Control the spacer element tag name

If the Virtualize<TItem> component is placed inside an element that requires a specific child tag name, SpacerElement allows you to obtain or set the virtualization spacer tag name. The default value is `div`. For the following example, the Virtualize<TItem> component renders inside a table body element (tbody), so the appropriate child element for a table row (tr) is set as the spacer.

`VirtualizedTable.razor`:

razor

```
@page "/virtualized-table"

<PageTitle>Virtualized Table</PageTitle>

<HeadContent>
    <style>
```

```
        html, body {
            overflow-y: scroll
        }
    </style>
</HeadContent>

<h1>Virtualized Table Example</h1>

<table id="virtualized-table">
    <thead style="position: sticky; top: 0; background-color: silver">
        <tr>
            <th>Item</th>
            <th>Another column</th>
        </tr>
    </thead>
    <tbody>
        <Virtualize Items="fixedItems" ItemSize="30" SpacerElement="tr">
            <tr @key="context" style="height: 30px;" id="row-@context">
                <td>Item @context</td>
                <td>Another value</td>
            </tr>
        </Virtualize>
    </tbody>
</table>

@code {
    private List<int> fixedItems = Enumerable.Range(0, 1000).ToList();
}
```

In the preceding example, the document root is used as the scroll container, so the `html` and `body` elements are styled with `overflow-y: scroll`. For more information, see the following resources:

- Root-level virtualization section
- Control head content in ASP.NET Core Blazor apps

ASP.NET Core Razor component rendering

Article • 10/18/2024

> **Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the .NET 9 version of this article.

This article explains Razor component rendering in ASP.NET Core Blazor apps, including when to call StateHasChanged to manually trigger a component to render.

Rendering conventions for `ComponentBase`

Components *must* render when they're first added to the component hierarchy by a parent component. This is the only time that a component must render. Components *may* render at other times according to their own logic and conventions.

Razor components inherit from the ComponentBase base class, which contains logic to trigger rerendering at the following times:

- After applying an updated set of parameters from a parent component.
- After applying an updated value for a cascading parameter.
- After notification of an event and invoking one of its own event handlers.
- After a call to its own StateHasChanged method (see ASP.NET Core Razor component lifecycle). For guidance on how to prevent overwriting child component parameters when StateHasChanged is called in a parent component, see Avoid overwriting parameters in ASP.NET Core Blazor.

Components inherited from ComponentBase skip rerenders due to parameter updates if either of the following are true:

- All of the parameters are from a set of known types† or any primitive type that hasn't changed since the previous set of parameters were set.

  †The Blazor framework uses a set of built-in rules and explicit parameter type checks for change detection. These rules and the types are subject to change at

any time. For more information, see the ChangeDetection API in the ASP.NET Core reference source.

> **Note**
>
> Documentation links to .NET reference source usually load the repository's default branch, which represents the current development for the next release of .NET. To select a tag for a specific release, use the **Switch branches or tags** dropdown list. For more information, see How to select a version tag of ASP.NET Core source code (dotnet/AspNetCore.Docs #26205).

- The override of the component's ShouldRender method returns `false` (the default `ComponentBase` implementation always returns `true`).

Control the rendering flow

In most cases, ComponentBase conventions result in the correct subset of component rerenders after an event occurs. Developers aren't usually required to provide manual logic to tell the framework which components to rerender and when to rerender them. The overall effect of the framework's conventions is that the component receiving an event rerenders itself, which recursively triggers rerendering of descendant components whose parameter values may have changed.

For more information on the performance implications of the framework's conventions and how to optimize an app's component hierarchy for rendering, see ASP.NET Core Blazor performance best practices.

Streaming rendering

Use *streaming rendering* with static server-side rendering (static SSR) or prerendering to stream content updates on the response stream and improve the user experience for components that perform long-running asynchronous tasks to fully render.

For example, consider a component that makes a long-running database query or web API call to render data when the page loads. Normally, asynchronous tasks executed as part of rendering a server-side component must complete before the rendered response is sent, which can delay loading the page. Any significant delay in rendering the page harms the user experience. To improve the user experience, streaming rendering initially renders the entire page quickly with placeholder content while asynchronous operations execute. After the operations are complete, the updated

content is sent to the client on the same response connection and patched into the DOM.

Streaming rendering requires the server to avoid buffering the output. The response data must flow to the client as the data is generated. For hosts that enforce buffering, streaming rendering degrades gracefully, and the page loads without streaming rendering.

To stream content updates when using static server-side rendering (static SSR) or prerendering, apply the `[StreamRendering(true)]` attribute to the component. Streaming rendering must be explicitly enabled because streamed updates may cause content on the page to shift. Components without the attribute automatically adopt streaming rendering if the parent component uses the feature. Pass `false` to the attribute in a child component to disable the feature at that point and further down the component subtree. The attribute is functional when applied to components supplied by a Razor class library.

The following example is based on the `Weather` component in an app created from the Blazor Web App project template. The call to Task.Delay simulates retrieving weather data asynchronously. The component initially renders placeholder content ("`Loading...`") without waiting for the asynchronous delay to complete. When the asynchronous delay completes and the weather data content is generated, the content is streamed to the response and patched into the weather forecast table.

`Weather.razor`:

razor

```razor
@page "/weather"
@attribute [StreamRendering(true)]

...

@if (forecasts == null)
{
    <p><em>Loading...</em></p>
}
else
{
    <table class="table">
        ...
        <tbody>
            @foreach (var forecast in forecasts)
            {
                <tr>
                    <td>@forecast.Date.ToShortDateString()</td>
                    <td>@forecast.TemperatureC</td>
```

```
                    <td>@forecast.TemperatureF</td>
                    <td>@forecast.Summary</td>
                </tr>
            }
        </tbody>
    </table>
}

@code {
    ...

    private WeatherForecast[]? forecasts;

    protected override async Task OnInitializedAsync()
    {
        await Task.Delay(500);

        ...

        forecasts = ...
    }
}
```

Suppress UI refreshing (`ShouldRender`)

ShouldRender is called each time a component is rendered. Override ShouldRender to manage UI refreshing. If the implementation returns `true`, the UI is refreshed.

Even if ShouldRender is overridden, the component is always initially rendered.

`ControlRender.razor`:

razor

```
@page "/control-render"

<PageTitle>Control Render</PageTitle>

<h1>Control Render Example</h1>

<label>
    <input type="checkbox" @bind="shouldRender" />
    Should Render?
</label>

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

<p>
    <button @onclick="IncrementCount">Click me</button>
</p>
```

```
@code {
    private int currentCount = 0;
    private bool shouldRender = true;

    protected override bool ShouldRender() => shouldRender;

    private void IncrementCount() => currentCount++;
}
```

For more information on performance best practices pertaining to ShouldRender, see ASP.NET Core Blazor performance best practices.

`StateHasChanged`

Calling StateHasChanged enqueues a rerender to occur when the app's main thread is free.

Components are enqueued for rendering, and they aren't enqueued again if there's already a pending rerender. If a component calls StateHasChanged five times in a row in a loop, the component only renders once. This behavior is encoded in ComponentBase, which checks first if it has queued a rerender before enqueuing an additional one.

A component can render multiple times during the same cycle, which commonly occurs when a component has children that interact with each other:

- A parent component renders several children.
- Child components render and trigger an update on the parent.
- A parent component rerenders with new state.

This design allows for StateHasChanged to be called when necessary without the risk of introducing unnecessary rendering. You can always take control of this behavior in individual components by implementing IComponent directly and manually handling when the component renders.

Consider the following `IncrementCount` method that increments a count, calls StateHasChanged, and increments the count again:

C#

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

Stepping through the code in the debugger, you might think that the count updates in the UI for the first `currentCount++` execution immediately after StateHasChanged is called. However, the UI doesn't show an updated count at that point due to the synchronous processing taking place for this method's execution. There's no opportunity for the renderer to render the component until after the event handler is finished. The UI displays increases for both `currentCount++` executions *in a single render*.

If you await something between the `currentCount++` lines, the awaited call gives the renderer a chance to render. This has led to some developers calling Delay with a one millisecond delay in their components to allow a render to occur, but we don't recommend arbitrarily slowing down an app to enqueue a render.

The best approach is to await Task.Yield, which forces the component to process code asynchronously and render during the current batch with a second render in a separate batch after the yielded task runs the continuation.

Consider the following revised `IncrementCount` method, which updates the UI twice because the render enqueued by StateHasChanged is performed when the task is yielded with the call to Task.Yield:

C#

```
private async Task IncrementCount()
{
    currentCount++;
    StateHasChanged();
    await Task.Yield();
    currentCount++;
}
```

Be careful not to call StateHasChanged unnecessarily, which is a common mistake that imposes unnecessary rendering costs. Code shouldn't need to call StateHasChanged when:

- Routinely handling events, whether synchronously or asynchronously, since ComponentBase triggers a render for most routine event handlers.
- Implementing typical lifecycle logic, such as OnInitialized or OnParametersSetAsync, whether synchronously or asynchronously, since ComponentBase triggers a render for typical lifecycle events.

However, it might make sense to call StateHasChanged in the cases described in the following sections of this article:

- An asynchronous handler involves multiple asynchronous phases

- Receiving a call from something external to the Blazor rendering and event handling system
- To render component outside the subtree that is rerendered by a particular event

An asynchronous handler involves multiple asynchronous phases

Due to the way that tasks are defined in .NET, a receiver of a Task can only observe its final completion, not intermediate asynchronous states. Therefore, ComponentBase can only trigger rerendering when the Task is first returned and when the Task finally completes. The framework can't know to rerender a component at other intermediate points, such as when an IAsyncEnumerable<T> returns data in a series of intermediate Tasks. If you want to rerender at intermediate points, call StateHasChanged at those points.

Consider the following `CounterState1` component, which updates the count four times each time the `IncrementCount` method executes:

- Automatic renders occur after the first and last increments of `currentCount`.
- Manual renders are triggered by calls to StateHasChanged when the framework doesn't automatically trigger rerenders at intermediate processing points where `currentCount` is incremented.

`CounterState1.razor`:

razor

```
@page "/counter-state-1"

<PageTitle>Counter State 1</PageTitle>

<h1>Counter State Example 1</h1>

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

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

@code {
    private int currentCount = 0;

    private async Task IncrementCount()
    {
```

```
        currentCount++;
        // Renders here automatically

        await Task.Delay(1000);
        currentCount++;
        StateHasChanged();

        await Task.Delay(1000);
        currentCount++;
        StateHasChanged();

        await Task.Delay(1000);
        currentCount++;
        // Renders here automatically
    }
}
```

Receiving a call from something external to the Blazor rendering and event handling system

ComponentBase only knows about its own lifecycle methods and Blazor-triggered events. ComponentBase doesn't know about other events that may occur in code. For example, any C# events raised by a custom data store are unknown to Blazor. In order for such events to trigger rerendering to display updated values in the UI, call StateHasChanged.

Consider the following `CounterState2` component that uses System.Timers.Timer to update a count at a regular interval and calls StateHasChanged to update the UI:

- `OnTimerCallback` runs outside of any Blazor-managed rendering flow or event notification. Therefore, `OnTimerCallback` must call StateHasChanged because Blazor isn't aware of the changes to `currentCount` in the callback.
- The component implements IDisposable, where the Timer is disposed when the framework calls the `Dispose` method. For more information, see ASP.NET Core Razor component lifecycle.

Because the callback is invoked outside of Blazor's synchronization context, the component must wrap the logic of `OnTimerCallback` in ComponentBase.InvokeAsync to move it onto the renderer's synchronization context. This is equivalent to marshalling to the UI thread in other UI frameworks. StateHasChanged can only be called from the renderer's synchronization context and throws an exception otherwise:

> System.InvalidOperationException: 'The current thread is not associated with the Dispatcher. Use InvokeAsync() to switch execution to the Dispatcher when triggering rendering or component state.'

`CounterState2.razor`:

razor

```razor
@page "/counter-state-2"
@using System.Timers
@implements IDisposable

<PageTitle>Counter State 2</PageTitle>

<h1>Counter State Example 2</h1>

<p>
    This counter demonstrates <code>Timer</code> disposal.
</p>

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

@code {
    private int currentCount = 0;
    private Timer timer = new(1000);

    protected override void OnInitialized()
    {
        timer.Elapsed += (sender, eventArgs) => OnTimerCallback();
        timer.Start();
    }

    private void OnTimerCallback()
    {
        _ = InvokeAsync(() =>
        {
            currentCount++;
            StateHasChanged();
        });
    }

    public void Dispose() => timer.Dispose();
}
```

To render a component outside the subtree that's rerendered by a particular event

The UI might involve:

1. Dispatching an event to one component.
2. Changing some state.

3. Rerendering a completely different component that isn't a descendant of the component receiving the event.

One way to deal with this scenario is to provide a *state management* class, often as a dependency injection (DI) service, injected into multiple components. When one component calls a method on the state manager, the state manager raises a C# event that's then received by an independent component.

For approaches to manage state, see the following resources:

- Bind across more than two components using data bindings.
- Pass data across a component hierarchy using cascading values and parameters.
- Server-side in-memory state container service (client-side equivalent) section of the *State management* article.

For the state manager approach, C# events are outside the Blazor rendering pipeline. Call StateHasChanged on other components you wish to rerender in response to the state manager's events.

The state manager approach is similar to the earlier case with System.Timers.Timer in the previous section. Since the execution call stack typically remains on the renderer's synchronization context, calling InvokeAsync isn't normally required. Calling InvokeAsync is only required if the logic escapes the synchronization context, such as calling ContinueWith on a Task or awaiting a Task with ConfigureAwait(false). For more information, see the Receiving a call from something external to the Blazor rendering and event handling system section.

WebAssembly loading progress indicator for Blazor Web Apps

A loading progress indicator isn't present in an app created from the Blazor Web App project template. A new loading progress indicator feature is planned for a future release of .NET. In the meantime, an app can adopt custom code to create a loading progress indicator. For more information, see ASP.NET Core Blazor startup.

ASP.NET Core Blazor templated components

Article • 10/18/2024

> **Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the .NET 9 version of this article.

By Robert Haken

This article explains how templated components can accept one or more UI templates as parameters, which can then be used as part of the component's rendering logic.

Templated components

Templated components are components that receive one or more UI templates as parameters, which can be utilized in the rendering logic of the component. By using templated components, you can create higher-level components that are more reusable. Examples include:

- A table component that allows a user to specify templates for the table's header, rows, and footer.
- A list component that allows a user to specify a template for rendering items in a list.
- A navigation bar component that allows a user to specify a template for start content and navigation links.

A templated component is defined by specifying one or more component parameters of type RenderFragment or RenderFragment<TValue>. A render fragment represents a segment of UI to render. RenderFragment<TValue> takes a type parameter that can be specified when the render fragment is invoked.

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

Often, templated components are generically typed, as the following `TemplatedNavBar` component (`TemplatedNavBar.razor`) demonstrates. The generic type (`<T>`) in the following example is used to render IReadOnlyList<T> values, which in this case is a list of pets for a component that displays a navigation bar with links to a pet detail component.

`TemplatedNavBar.razor`:

```
@typeparam TItem

<nav class="navbar navbar-expand navbar-light bg-light">
    <div class="container justify-content-start">
        @StartContent
        <div class="navbar-nav">
            @foreach (var item in Items)
            {
                @ItemTemplate(item)
            }
        </div>
    </div>
</nav>

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

    [Parameter, EditorRequired]
    public RenderFragment<TItem> ItemTemplate { get; set; } = default!;

    [Parameter, EditorRequired]
    public IReadOnlyList<TItem> Items { get; set; } = default!;
}
```

When using a templated component, the template parameters can be specified using child elements that match the names of the parameters. In the following example, `<StartContent>...</StartContent>` and `<ItemTemplate>...</ItemTemplate>` supply RenderFragment<TValue> templates for `StartContent` and `ItemTemplate` of the `TemplatedNavBar` component.

Specify the `Context` attribute on the component element when you want to specify the content parameter name for implicit child content (without any wrapping child element). In the following example, the `Context` attribute appears on the `TemplatedNavBar` element and applies to all RenderFragment<TValue> template parameters.

`Pets1.razor`:

```
@page "/pets-1"

<PageTitle>Pets 1</PageTitle>

<h1>Pets Example 1</h1>

<TemplatedNavBar Items="pets" Context="pet">
    <StartContent>
        <a href="/" class="navbar-brand">PetsApp</a>
    </StartContent>
    <ItemTemplate>
        <NavLink href="@($"/pet-detail/{pet.PetId}?ReturnUrl=%2Fpets-1")"
class="nav-link">
            @pet.Name
        </NavLink>
    </ItemTemplate>
</TemplatedNavBar>

@code {
    private List<Pet> pets = new()
    {
        new Pet { PetId = 1, Name = "Mr. Bigglesworth" },
        new Pet { PetId = 2, Name = "Salem Saberhagen" },
        new Pet { PetId = 3, Name = "K-9" }
    };

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

Alternatively, you can change the parameter name using the `Context` attribute on the RenderFragment<TValue> child element. In the following example, the `Context` is set on `ItemTemplate` rather than `TemplatedNavBar`.

`Pets2.razor`:

```
@page "/pets-2"

<PageTitle>Pets 2</PageTitle>

<h1>Pets Example 2</h1>

<TemplatedNavBar Items="pets">
    <StartContent>
        <a href="/" class="navbar-brand">PetsApp</a>
    </StartContent>
    <ItemTemplate Context="pet">
        <NavLink href="@($"/pet-detail/{pet.PetId}?ReturnUrl=%2Fpets-2")"
```

```
class="nav-link">
            @pet.Name
        </NavLink>
    </ItemTemplate>
</TemplatedNavBar>

@code {
    private List<Pet> pets = new()
    {
        new Pet { PetId = 1, Name = "Mr. Bigglesworth" },
        new Pet { PetId = 2, Name = "Salem Saberhagen" },
        new Pet { PetId = 3, Name = "K-9" }
    };

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

Component arguments of type RenderFragment<TValue> have an implicit parameter named `context`, which can be used. In the following example, `Context` isn't set. `@context.{PROPERTY}` supplies pet values to the template, where `{PROPERTY}` is a `Pet` property:

`Pets3.razor`:

```
@page "/pets-3"

<PageTitle>Pets 3</PageTitle>

<h1>Pets Example 3</h1>

<TemplatedNavBar Items="pets">
    <StartContent>
        <a href="/" class="navbar-brand">PetsApp</a>
    </StartContent>
    <ItemTemplate>
        <NavLink href="@($"/pet-detail/{context.PetId}?ReturnUrl=%2Fpets-
3")" class="nav-link">
            @context.Name
        </NavLink>
    </ItemTemplate>
</TemplatedNavBar>

@code {
    private List<Pet> pets = new()
    {
        new Pet { PetId = 1, Name = "Mr. Bigglesworth" },
        new Pet { PetId = 2, Name = "Salem Saberhagen" },
```

```
        new Pet { PetId = 3, Name = "K-9" }
    };

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

When using generic-typed components, the type parameter is inferred if possible. However, you can explicitly specify the type with an attribute that has a name matching the type parameter, which is `TItem` in the preceding example:

`Pets4.razor`:

```
@page "/pets-4"

<PageTitle>Pets 4</PageTitle>

<h1>Pets Example 4</h1>

<TemplatedNavBar Items="pets" Context="pet" TItem="Pet">
    <StartContent>
        <a href="/" class="navbar-brand">PetsApp</a>
    </StartContent>
    <ItemTemplate>
        <NavLink href="@($"/pet-detail/{pet.PetId}?ReturnUrl=%2Fpets-4")"
class="nav-link">
            @pet.Name
        </NavLink>
    </ItemTemplate>
</TemplatedNavBar>

@code {
    private List<Pet> pets = new()
    {
        new Pet { PetId = 2, Name = "Mr. Bigglesworth" },
        new Pet { PetId = 4, Name = "Salem Saberhagen" },
        new Pet { PetId = 7, Name = "K-9" }
    };

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

The example provided in the `TemplatedNavBar` component (`TemplatedNavBar.razor`) assumes that the `Items` collection doesn't change after the initial render; or that if it does change, maintaining the state of components/elements used in `ItemTemplate` isn't necessary. For templated components where such usage can't be anticipated, see the Preserve relationships with @key section.

Preserve relationships with `@key`

Templated components are often used to render collections of items, such as tables or lists. In such general scenarios, we can't assume that the user will avoid stateful components/elements in the item template definition or that there won't be additional changes to the `Items` collection. For such templated components, it's necessary to preserve the relationships with the `@key` directive attribute.

> ⓘ Note
>
> For more information on the `@key` directive attribute, see Retain element, component, and model relationships in ASP.NET Core Blazor.

The following `TableTemplate` component (`TableTemplate.razor`) demonstrates a templated component that preserves relationships with `@key`.

`TableTemplate.razor`:

```
@typeparam TItem

<table class="table">
    <thead>
        <tr>@TableHeader</tr>
    </thead>
    <tbody>
        @foreach (var item in Items)
        {
            <tr @key="@item">@RowTemplate(item)</tr>
        }
    </tbody>
</table>

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

    [Parameter, EditorRequired]
    public RenderFragment<TItem> RowTemplate { get; set; } = default!;
```

```
    [Parameter, EditorRequired]
    public IReadOnlyList<TItem> Items { get; set; } = default!;
}
```

Consider the following `Pets5` component (`Pets5.razor`), which demonstrates the importance of keying data to preserve model relationships. In the component, each iteration of adding a pet in OnAfterRenderAsync results in Blazor rerendering the `TableTemplate` component.

`Pets5.razor`:

```
@page "/pets-5"

<PageTitle>Pets 5</PageTitle>

<h1>Pets Example 5</h1>

<TableTemplate Items="pets" Context="pet">
    <TableHeader>
        <th>ID</th>
        <th>Name</th>
    </TableHeader>
    <RowTemplate>
        <td><input value="@pet.PetId" /></td>
        <td><input value="@pet.Name" /></td>
    </RowTemplate>
</TableTemplate>


@code {
    private List<Pet> pets = new()
    {
        new Pet { PetId = 1, Name = "Mr. Bigglesworth" },
        new Pet { PetId = 2, Name = "Salem Saberhagen" },
        new Pet { PetId = 3, Name = "K-9" }
    };

    protected override async Task OnAfterRenderAsync(bool firstRender)
    {
        // Insert new pet every 5 seconds
        if (pets.Count < 10)
        {
            await Task.Delay(5000);
            pets.Insert(0, new Pet() { PetId = pets.Count + 1, Name = "<new
pet>" });
            StateHasChanged();
        }
    }

    private class Pet
    {
```

```
        public int PetId { get; set; }
        public string? Name { get; set; }
    }
}
```

This demonstration allows you to:

- Select an `<input>` from among several rendered table rows.
- Study the behavior of the page's focus as the pets collection automatically grows.

Without using the `@key` directive attribute in the `TableTemplate` component, the page's focus remains on the same index position (row) of the table, causing the focus to shift each time a pet is added. To demonstrate this, remove the `@key` directive attribute and value, restart the app, and attempt to modify a field value as items are added.

Additional resources

- ASP.NET Core Blazor performance best practices
- Retain element, component, and model relationships in ASP.NET Core Blazor
- Blazor samples GitHub repository (dotnet/blazor-samples) (how to download)

ASP.NET Core Blazor CSS isolation

Article • 09/12/2024

> **Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the .NET 9 version of this article.

By Dave Brock

This article explains how CSS isolation scopes CSS to Razor components, which can simplify CSS and avoid collisions with other components or libraries.

Isolate CSS styles to individual pages, views, and components to reduce or avoid:

- Dependencies on global styles that can be challenging to maintain.
- Style conflicts in nested content.

Enable CSS isolation

To define component-specific styles, create a `.razor.css` file matching the name of the `.razor` file for the component in the same folder. The `.razor.css` file is a *scoped CSS file*.

For an `Example` component in an `Example.razor` file, create a file alongside the component named `Example.razor.css`. The `Example.razor.css` file must reside in the same folder as the `Example` component (`Example.razor`). The "`Example`" base name of the file is **not** case-sensitive.

`Example.razor`:

```razor
@page "/example"

<h1>Scoped CSS Example</h1>
```

`Example.razor.css`:

```css
h1 {
    color: brown;
    font-family: Tahoma, Geneva, Verdana, sans-serif;
}
```

The styles defined in `Example.razor.css` are only applied to the rendered output of the `Example` component. CSS isolation is applied to HTML elements in the matching Razor file. Any `h1` CSS declarations defined elsewhere in the app don't conflict with the `Example` component's styles.

> ⓘ Note
>
> In order to guarantee style isolation when bundling occurs, importing CSS in Razor code blocks isn't supported.

CSS isolation bundling

CSS isolation occurs at build time. Blazor rewrites CSS selectors to match markup rendered by the component. The rewritten CSS styles are bundled and produced as a static asset. The stylesheet is referenced inside the `<head>` tag (location of <head> content). The following `<link>` element is added to an app created from the Blazor project templates:

Blazor Web Apps:

```HTML
<link href="@Assets["{ASSEMBLY NAME}.styles.css"]" rel="stylesheet">
```

Standalone Blazor WebAssembly apps:

```HTML
<link href="{ASSEMBLY NAME}.styles.css" rel="stylesheet">
```

The `{ASSEMBLY NAME}` placeholder is the project's assembly name.

Within the bundled file, each component is associated with a scope identifier. For each styled component, an HTML attribute is appended with the format `b-{STRING}`, where

the `{STRING}` placeholder is a ten-character string generated by the framework. The identifier is unique for each app. In the rendered `Counter` component, Blazor appends a scope identifier to the `h1` element:

HTML

```
<h1 b-3xxtam6d07>
```

The `{ASSEMBLY NAME}.styles.css` file uses the scope identifier to group a style declaration with its component. The following example provides the style for the preceding `<h1>` element:

css

```
/* /Components/Pages/Counter.razor.rz.scp.css */
h1[b-3xxtam6d07] {
    color: brown;
}
```

At build time, a project bundle is created with the convention `obj/{CONFIGURATION}/{TARGET FRAMEWORK}/scopedcss/projectbundle/{ASSEMBLY NAME}.bundle.scp.css`, where the placeholders are:

- `{CONFIGURATION}`: The app's build configuration (for example, `Debug`, `Release`).
- `{TARGET FRAMEWORK}`: The target framework (for example, `net6.0`).
- `{ASSEMBLY NAME}`: The app's assembly name (for example, `BlazorSample`).

Child component support

CSS isolation only applies to the component you associate with the format `{COMPONENT NAME}.razor.css`, where the `{COMPONENT NAME}` placeholder is usually the component name. To apply changes to a child component, use the `::deep` pseudo-element to any descendant elements in the parent component's `.razor.css` file. The `::deep` pseudo-element selects elements that are *descendants* of an element's generated scope identifier.

The following example shows a parent component called `Parent` with a child component called `Child`.

`Parent.razor`:

razor

```razor
@page "/parent"

<div>
    <h1>Parent component</h1>

    <Child />
</div>
```

`Child.razor`:

razor

```razor
<h1>Child Component</h1>
```

Update the `h1` declaration in `Parent.razor.css` with the `::deep` pseudo-element to signify the `h1` style declaration must apply to the parent component and its children.

`Parent.razor.css`:

css

```css
::deep h1 {
    color: red;
}
```

The `h1` style now applies to the `Parent` and `Child` components without the need to create a separate scoped CSS file for the child component.

The `::deep` pseudo-element only works with descendant elements. The following markup applies the `h1` styles to components as expected. The parent component's scope identifier is applied to the `div` element, so the browser knows to inherit styles from the parent component.

`Parent.razor`:

razor

```razor
<div>
    <h1>Parent</h1>

    <Child />
</div>
```

However, excluding the `div` element removes the descendant relationship. In the following example, the style is **not** applied to the child component.

`Parent.razor`:

```razor
<h1>Parent</h1>

<Child />
```

The `::deep` pseudo-element affects where the scope attribute is applied to the rule. When you define a CSS rule in a scoped CSS file, the scope is applied to the right most element. For example: `div > a` is transformed to `div > a[b-{STRING}]`, where the `{STRING}` placeholder is a ten-character string generated by the framework (for example, `b-3xxtam6d07`). If you instead want the rule to apply to a different selector, the `::deep` pseudo-element allows you do so. For example, `div ::deep > a` is transformed to `div[b-{STRING}] > a` (for example, `div[b-3xxtam6d07] > a`).

The ability to attach the `::deep` pseudo-element to any HTML element allows you to create scoped CSS styles that affect elements rendered by other components when you can determine the structure of the rendered HTML tags. For a component that renders an hyperlink tag (`<a>`) inside another component, ensure the component is wrapped in a `div` (or any other element) and use the rule `::deep > a` to create a style that's only applied to that component when the parent component renders.

> **ⓘ Important**
>
> Scoped CSS only applies to ***HTML elements*** and not to Razor components or Tag Helpers, including elements with a Tag Helper applied, such as `<input asp-for="..." />`.

CSS preprocessor support

CSS preprocessors are useful for improving CSS development by utilizing features such as variables, nesting, modules, mixins, and inheritance. While CSS isolation doesn't natively support CSS preprocessors such as Sass or Less, integrating CSS preprocessors is seamless as long as preprocessor compilation occurs before Blazor rewrites the CSS selectors during the build process. Using Visual Studio for example, configure existing

preprocessor compilation as a **Before Build** task in the Visual Studio Task Runner Explorer.

Many third-party NuGet packages, such as AspNetCore.SassCompiler, can compile SASS/SCSS files at the beginning of the build process before CSS isolation occurs.

CSS isolation configuration

CSS isolation is designed to work out-of-the-box but provides configuration for some advanced scenarios, such as when there are dependencies on existing tools or workflows.

Customize scope identifier format

Scope identifiers use the format `b-{STRING}`, where the `{STRING}` placeholder is a ten-character string generated by the framework. To customize the scope identifier format, update the project file to a desired pattern:

XML

```
<ItemGroup>
  <None Update="Components/Pages/Example.razor.css" CssScope="custom-scope-identifier" />
</ItemGroup>
```

In the preceding example, the CSS generated for `Example.razor.css` changes its scope identifier from `b-{STRING}` to `custom-scope-identifier`.

Use scope identifiers to achieve inheritance with scoped CSS files. In the following project file example, a `BaseComponent.razor.css` file contains common styles across components. A `DerivedComponent.razor.css` file inherits these styles.

XML

```
<ItemGroup>
  <None Update="Components/Pages/BaseComponent.razor.css" CssScope="custom-scope-identifier" />
  <None Update="Components/Pages/DerivedComponent.razor.css" CssScope="custom-scope-identifier" />
</ItemGroup>
```

Use the wildcard (`*`) operator to share scope identifiers across multiple files:

XML

```
<ItemGroup>
  <None Update="Components/Pages/*.razor.css" CssScope="custom-scope-
identifier" />
</ItemGroup>
```

Change base path for static web assets

The `scoped.styles.css` file is generated at the root of the app. In the project file, use the <StaticWebAssetBasePath> property to change the default path. The following example places the `scoped.styles.css` file, and the rest of the app's assets, at the `_content` path:

XML

```
<PropertyGroup>
  <StaticWebAssetBasePath>_content/$(PackageId)</StaticWebAssetBasePath>
</PropertyGroup>
```

Disable automatic bundling

To opt out of how Blazor publishes and loads scoped files at runtime, use the `DisableScopedCssBundling` property. When using this property, it means other tools or processes are responsible for taking the isolated CSS files from the `obj` directory and publishing and loading them at runtime:

XML

```
<PropertyGroup>
  <DisableScopedCssBundling>true</DisableScopedCssBundling>
</PropertyGroup>
```

Disable CSS isolation

Disable CSS isolation for a project by setting the `<ScopedCssEnabled>` property to `false` in the app's project file:

XML

```
<ScopedCssEnabled>false</ScopedCssEnabled>
```

Razor class library (RCL) support

Isolated styles for components in a NuGet package or Razor class library (RCL) are automatically bundled:

- The app uses CSS imports to reference the RCL's bundled styles. For a class library named `ClassLib` and a Blazor app with a `BlazorSample.styles.css` stylesheet, the RCL's stylesheet is imported at the top of the app's stylesheet:

  ```
  css
  ```

  ```
  @import '_content/ClassLib/ClassLib.bundle.scp.css';
  ```

- The RCL's bundled styles aren't published as a static web asset of the app that consumes the styles.

For more information on RCLs, see the following articles:

- Consume ASP.NET Core Razor components from a Razor class library (RCL)
- Reusable Razor UI in class libraries with ASP.NET Core

Additional resources

- Razor Pages CSS isolation
- MVC CSS isolation

Dynamically-rendered ASP.NET Core Razor components

Article • 11/06/2024

> **ⓘ Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the .NET 9 version of this article.

By Dave Brock

Use the built-in DynamicComponent component to render components by type.

Dynamic components

A DynamicComponent is useful for rendering components without iterating through possible types or using conditional logic. For example, DynamicComponent can render a component based on a user selection from a dropdown list.

In the following example:

- `componentType` specifies the type.
- `parameters` specifies component parameters to pass to the `componentType` component.

razor

```
<DynamicComponent Type="componentType" Parameters="parameters" />

@code {
    private Type componentType = ...;
    private IDictionary<string, object> parameters = ...;
}
```

For more information on passing parameter values, see the Pass parameters section later in this article.

Example

In the following example, a Razor component renders a component based on the user's selection from a dropdown list of four possible values.

Expand table

User spaceflight carrier selection	Shared Razor component to render
Rocket Lab®	`RocketLab.razor`
SpaceX®	`SpaceX.razor`
ULA®	`UnitedLaunchAlliance.razor`
Virgin Galactic®	`VirginGalactic.razor`

`RocketLab.razor`:

razor

```razor
<h2>Rocket Lab®</h2>

<p>
    Rocket Lab is a registered trademark of
    <a href="https://www.rocketlabusa.com/">Rocket Lab USA Inc.</a>
</p>
```

`SpaceX.razor`:

razor

```razor
<h2>SpaceX®</h2>

<p>
    SpaceX is a registered trademark of
    <a href="https://www.spacex.com/">Space Exploration Technologies Corp.
</a>
</p>
```

`UnitedLaunchAlliance.razor`:

razor

```razor
<h2>United Launch Alliance®</h2>

<p>
    United Launch Alliance and ULA are registered trademarks of
```

```
    <a href="https://www.ulalaunch.com/">United Launch Alliance, LLC</a>.
</p>
```

`VirginGalactic.razor`:

razor

```
<h2>Virgin Galactic®</h2>

<p>
    Virgin Galactic is a registered trademark of
    <a href="https://www.virgingalactic.com/">Galactic Enterprises, LLC</a>.
</p>
```

`DynamicComponent1.razor`:

razor

```
@page "/dynamic-component-1"

<PageTitle>Dynamic Component 1</PageTitle>

<h1>Dynamic Component Example 1</h1>

<p>
    <label>
        Select your transport:
        <select @onchange="OnDropdownChange">
            <option value="">Select a value</option>
            @foreach (var entry in components.Keys)
            {
                <option value="@entry">@entry</option>
            }
        </select>
    </label>
</p>

@if (selectedType is not null)
{
    <div class="border border-primary my-1 p-1">
        <DynamicComponent Type="selectedType" />
    </div>
}

@code {
    private readonly Dictionary<string, Type> components = new()
    {
        ["Rocket Lab"] = typeof(RocketLab),
        ["SpaceX"] = typeof(SpaceX),
        ["ULA"] = typeof(UnitedLaunchAlliance),
        ["Virgin Galactic"] = typeof(VirginGalactic)
```

```
    };
    private Type? selectedType;

    private void OnDropdownChange(ChangeEventArgs e)
    {
        if ((e.Value is string dropdownValue) &&
            !String.IsNullOrWhiteSpace(dropdownValue))
        {
            selectedType = components[dropdownValue];
        }
        else
        {
            selectedType = null;
        }
    }
}
```

In the preceding example:

- An Dictionary<TKey,TValue> is used to manage components to be displayed.
- Names serve as the dictionary keys and are provided as selection options.
- Component types are stored as dictionary values using the typeof operator.

Pass parameters

If dynamically-rendered components have component parameters, pass them into the DynamicComponent as an `IDictionary<string, object>`. The `string` is the name of the parameter, and the `object` is the parameter's value.

The following example configures a component metadata object (`ComponentMetadata`) to supply parameter values to dynamically-rendered components based on the type name. The example is just one of several approaches that you can adopt. Parameter data can also be provided from a web API, a database, or a method. The only requirement is that the approach returns an `IDictionary<string, object>`.

`ComponentMetadata.cs`:

C#

```
namespace BlazorSample;

public class ComponentMetadata
{
    public required Type Type { get; init; }
    public required string Name { get; init; }
    public Dictionary<string, object> Parameters { get; } = [];
}
```

The following `RocketLabWithWindowSeat` component (`RocketLabWithWindowSeat.razor`) has been updated from the preceding example to include a component parameter named `WindowSeat` to specify if the passenger prefers a window seat on their flight:

`RocketLabWithWindowSeat.razor`:

```razor
<h2>Rocket Lab®</h2>

<p>
    User selected a window seat: @WindowSeat
</p>

<p>
    Rocket Lab is a trademark of
    <a href="https://www.rocketlabusa.com/">Rocket Lab USA Inc.</a>
</p>

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

In the following example:

- Only the `RocketLabWithWindowSeat` component's parameter for a window seat (`WindowSeat`) receives the value of the **`Window Seat`** checkbox.
- Component names are used as dictionary keys using the nameof operator, which returns component names as constant strings.
- The dynamically-rendered components are shared components:
  - Shown in this article section: `RocketLabWithWindowSeat` (`RocketLabWithWindowSeat.razor`)
  - Components shown in the Example section earlier in this article:
    - `SpaceX` (`SpaceX.razor`)
    - `UnitedLaunchAlliance` (`UnitedLaunchAlliance.razor`)
    - `VirginGalactic` (`VirginGalactic.razor`)

`DynamicComponent2.razor`:

```razor
@page "/dynamic-component-2"

<PageTitle>Dynamic Component 2</PageTitle>

<h1>Dynamic Component Example 2</h1>
```

```
<p>
    <label>
        <input type="checkbox" @bind="windowSeat"
            @bind:after="HandleWindowSeatChanged" />
        Window Seat (Rocket Lab only)
    </label>
</p>

<p>
    <label>
        Select your transport:
        <select @onchange="OnDropdownChange">
            <option value="">Select a value</option>
            @foreach (var c in components)
            {
                <option value="@c.Key">@c.Value.Name</option>
            }
        </select>
    </label>
</p>

@if (selectedComponent is not null)
{
    <div class="border border-primary my-1 p-1">
        <DynamicComponent Type="selectedComponent.Type"
            Parameters="selectedComponent.Parameters" />
    </div>
}

@code {
    private Dictionary<string, ComponentMetadata> components =
        new()
        {
            [nameof(RocketLabWithWindowSeat)] = new ComponentMetadata()
            {
                Type = typeof(RocketLabWithWindowSeat),
                Name = "Rocket Lab with Window Seat",
                Parameters = { [nameof(RocketLabWithWindowSeat.WindowSeat)]
= false }
            },
            [nameof(VirginGalactic)] = new ComponentMetadata()
            {
                Type = typeof(VirginGalactic),
                Name = "Virgin Galactic"
            },
            [nameof(UnitedLaunchAlliance)] = new ComponentMetadata()
            {
                Type = typeof(UnitedLaunchAlliance),
                Name = "ULA"
            },
            [nameof(SpaceX)] = new ComponentMetadata()
            {
                Type = typeof(SpaceX),
                Name = "SpaceX"
```

```
            }
        };
    private ComponentMetadata? selectedComponent;
    private bool windowSeat;

    private void HandleWindowSeatChanged()
    {
        components[nameof(RocketLabWithWindowSeat)]
            .Parameters[nameof(RocketLabWithWindowSeat.WindowSeat)] =
windowSeat;
    }

    private void OnDropdownChange(ChangeEventArgs e)
    {
        if ((e.Value is string dropdownValue) &&
!String.IsNullOrWhiteSpace(dropdownValue))
        {
            selectedComponent = components[dropdownValue];
        }
        else
        {
            selectedComponent = null;
        }
    }
}
```

Event callbacks (`EventCallback`)

Event callbacks (EventCallback) can be passed to a DynamicComponent in its parameter dictionary.

`ComponentMetadata.cs`:

C#

```
namespace BlazorSample;

public class ComponentMetadata
{
    public required Type Type { get; init; }
    public required string Name { get; init; }
    public Dictionary<string, object> Parameters { get; } = [];
}
```

Implement an event callback parameter (EventCallback) within each dynamically-rendered component.

`RocketLab2.razor`:

razor

```razor
<h2>Rocket Lab®</h2>

<p>
    Rocket Lab is a registered trademark of
    <a href="https://www.rocketlabusa.com/">Rocket Lab USA Inc.</a>
</p>

<button @onclick="OnClickCallback">
    Trigger a Parent component method
</button>

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

`SpaceX2.razor`:

razor

```razor
<h2>SpaceX®</h2>

<p>
    SpaceX is a registered trademark of
    <a href="https://www.spacex.com/">Space Exploration Technologies Corp.
</a>
</p>

<button @onclick="OnClickCallback">
    Trigger a Parent component method
</button>

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

`UnitedLaunchAlliance2.razor`:

razor

```razor
<h2>United Launch Alliance®</h2>

<p>
    United Launch Alliance and ULA are registered trademarks of
    <a href="https://www.ulalaunch.com/">United Launch Alliance, LLC</a>.
</p>

<button @onclick="OnClickCallback">
```

```
    Trigger a Parent component method
</button>

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

`VirginGalactic2.razor`:

razor

```
<h2>Virgin Galactic®</h2>

<p>
    Virgin Galactic is a registered trademark of
    <a href="https://www.virgingalactic.com/">Galactic Enterprises, LLC</a>.
</p>

<button @onclick="OnClickCallback">
    Trigger a Parent component method
</button>

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

In the following parent component example, the `ShowDTMessage` method assigns a string with the current time to `message`, and the value of `message` is rendered.

The parent component passes the callback method, `ShowDTMessage` in the parameter dictionary:

- The `string` key is the callback method's name, `OnClickCallback`.
- The `object` value is created by EventCallbackFactory.Create for the parent callback method, `ShowDTMessage`. Note that the this keyword isn't supported in C# field initialization, so a C# property is used for the parameter dictionary.

`DynamicComponent3.razor`:

razor

```
@page "/dynamic-component-3"

<PageTitle>Dynamic Component 3</PageTitle>

<h1>Dynamic Component Example 3</h1>
```

```
<p>
    <label>
        Select your transport:
        <select @onchange="OnDropdownChange">
            <option value="">Select a value</option>
            @foreach (var c in Components)
            {
                <option value="@c.Key">@c.Value.Name</option>
            }
        </select>
    </label>
</p>

@if (selectedComponent is not null)
{
    <div class="border border-primary my-1 p-1">
        <DynamicComponent Type="selectedComponent.Type"
            Parameters="selectedComponent.Parameters" />
    </div>
}

<p>
    @message
</p>

@code {
    private ComponentMetadata? selectedComponent;
    private string? message;

    private Dictionary<string, ComponentMetadata> Components =>
        new()
        {
            [nameof(RocketLab2)] = new ComponentMetadata()
            {
                Type = typeof(RocketLab2),
                Name = "Rocket Lab",
                Parameters = { [nameof(RocketLab2.OnClickCallback)] =
                    EventCallback.Factory.Create<MouseEventArgs>(this,
ShowDTMessage) }
            },
            [nameof(VirginGalactic2)] = new ComponentMetadata()
            {
                Type = typeof(VirginGalactic2),
                Name = "Virgin Galactic",
                Parameters = { [nameof(VirginGalactic2.OnClickCallback)] =
                    EventCallback.Factory.Create<MouseEventArgs>(this,
ShowDTMessage) }
            },
            [nameof(UnitedLaunchAlliance2)] = new ComponentMetadata()
            {
                Type = typeof(UnitedLaunchAlliance2),
                Name = "ULA",
                Parameters = {
[nameof(UnitedLaunchAlliance2.OnClickCallback)] =
                    EventCallback.Factory.Create<MouseEventArgs>(this,
```

```
ShowDTMessage) }
            },
            [nameof(SpaceX2)] = new ComponentMetadata()
            {
                Type = typeof(SpaceX2),
                Name = "SpaceX",
                Parameters = { [nameof(SpaceX2.OnClickCallback)] =
                    EventCallback.Factory.Create<MouseEventArgs>(this,
ShowDTMessage) }
            }
        };

    private void OnDropdownChange(ChangeEventArgs e)
    {
        if ((e.Value is string dropdownValue) &&
!String.IsNullOrWhiteSpace(dropdownValue))
        {
            selectedComponent = Components[dropdownValue];
        }
        else
        {
            selectedComponent = null;
        }
    }

    private void ShowDTMessage(MouseEventArgs e) =>
        message = $"The current DT is: {DateTime.Now}.";
}
```

Avoid catch-all parameters

Avoid the use of catch-all parameters. If catch-all parameters are used, every explicit parameter on DynamicComponent effectively is a reserved word that you can't pass to a dynamic child. Any new parameters passed to DynamicComponent are a breaking change, as they start shadowing child component parameters that happen to have the same name. It's unlikely that the caller always knows a fixed set of parameter names to pass to all possible dynamic children.

Access the dynamically-created component instance

Use the Instance property to access the dynamically-created component instance.

Create an interface to describe the dynamically-created component instance with any methods and properties that you need to access from the parent component when the

component is dynamically loaded. The following example specifies a `Log` method for implementation in components.

`Interfaces/ILoggable.cs`:

C#

```
using Microsoft.AspNetCore.Components;

namespace BlazorSample.Interfaces;

public interface ILoggable : IComponent
{
    public void Log();
}
```

Each component definition implements the interface. The following example is a modified Rocket Lab® component from the Example section that logs a string via its `Log` method.

`RocketLab3.razor`:

razor

```
@using BlazorSample.Interfaces
@implements ILoggable
@inject ILogger<RocketLab3> Logger

<h2>Rocket Lab®</h2>

<p>
    Rocket Lab is a registered trademark of
    <a href="https://www.rocketlabusa.com/">Rocket Lab USA Inc.</a>
</p>

@code {
    public void Log() => Logger.LogInformation("Woot! I logged this call!");
}
```

The remaining three shared components (`VirginGalactic3`, `UnitedLaunchAlliance3`, `SpaceX3`) receive similar treatment:

- The following directives are added to the components, where the `{COMPONENT TYPE}` placeholder is the component type:

  razor

```
@using BlazorSample.Interfaces
@implements ILoggable
@inject ILogger<{COMPONENT TYPE}> Logger
```

- Each component implements a `Log` method. The log category written by the logger includes the fully-qualified name of the component type when LogInformation is called:

  razor

  ```
  @code {
      public void Log()
      {
          Logger.LogInformation("Woot! I logged this call!");
      }
  }
  ```

The parent component casts the dynamically-loaded component instance as an `ILoggable` to access members of the interface. In the following example, the loaded component's `Log` method is called when a button is selected in the UI:

razor

```
...
@using BlazorSample.Interfaces

...

<DynamicComponent Type="..." @ref="dc" />

...

<button @onclick="LogFromLoadedComponent">Log from loaded component</button>

@code {
    private DynamicComponent? dc;

    ...

    private void LogFromLoadedComponent() => (dc?.Instance as
ILoggable)?.Log();
}
```

For a working demonstration of the preceding example, see the DynamicComponent4 component in the Blazor sample app.

Trademarks

Rocket Lab is a registered trademark of Rocket Lab USA Inc. SpaceX is a registered trademark of Space Exploration Technologies Corp. United Launch Alliance and ULA are registered trademarks of United Launch Alliance, LLC. Virgin Galactic is a registered trademark of Galactic Enterprises, LLC.

Additional resources

- ASP.NET Core Blazor event handling
- DynamicComponent

ASP.NET Core Blazor `QuickGrid` component

Article • 12/05/2024

> **Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the .NET 9 version of this article.

The QuickGrid component is a Razor component for quickly and efficiently displaying data in tabular form. `QuickGrid` provides a simple and convenient data grid component for common grid rendering scenarios and serves as a reference architecture and performance baseline for building data grid components. `QuickGrid` is highly optimized and uses advanced techniques to achieve optimal rendering performance.

Package

Add a package reference for the Microsoft.AspNetCore.Components.QuickGrid package.

> **Note**
>
> For guidance on adding packages to .NET apps, see the articles under *Install and manage packages* at Package consumption workflow (NuGet documentation). Confirm correct package versions at NuGet.org.

Sample app

For various `QuickGrid` demonstrations, see the **QuickGrid for Blazor** sample app. The demo site is hosted on GitHub Pages. The site loads fast thanks to static prerendering using the community-maintained BlazorWasmPrerendering.Build GitHub project.

`QuickGrid` implementation

To implement a `QuickGrid` component:

- Specify tags for the `QuickGrid` component in Razor markup (`<QuickGrid>...</QuickGrid>`).
- Name a queryable source of data for the grid. Use ***either*** of the following data sources:
  - Items: A nullable `IQueryable<TGridItem>`, where `TGridItem` is the type of data represented by each row in the grid.
  - ItemsProvider: A callback that supplies data for the grid.
- Class: An optional CSS class name. If provided, the class name is included in the `class` attribute of the rendered table.
- Theme: A theme name (default value: `default`). This affects which styling rules match the table.
- Virtualize: If true, the grid is rendered with virtualization. This is normally used in conjunction with scrolling and causes the grid to fetch and render only the data around the current scroll viewport. This can greatly improve the performance when scrolling through large data sets. If you use Virtualize, you should supply a value for ItemSize and must ensure that every row renders with a constant height. Generally, it's preferable not to use Virtualize if the amount of data rendered is small or if you're using pagination.
- ItemSize: Only applicable when using Virtualize. ItemSize defines an expected height in pixels for each row, allowing the virtualization mechanism to fetch the correct number of items to match the display size and to ensure accurate scrolling.
- ItemKey: Optionally defines a value for `@key` on each rendered row. Typically, this is used to specify a unique identifier, such as a primary key value, for each data item. This allows the grid to preserve the association between row elements and data items based on their unique identifiers, even when the `TGridItem` instances are replaced by new copies (for example, after a new query against the underlying data store). If not set, the `@key` is the `TGridItem` instance.
- OverscanCount: Defines how many additional items to render before and after the visible region to reduce rendering frequency during scrolling. While higher values can improve scroll smoothness by rendering more items off-screen, a higher value can also result in an increase in initial load times. Finding a balance based on your data set size and user experience requirements is recommended. The default value is 3. Only available when using Virtualize.
- Pagination: Optionally links this `TGridItem` instance with a PaginationState model, causing the grid to fetch and render only the current page of data. This is normally used in conjunction with a Paginator component or some other UI logic that displays and updates the supplied PaginationState instance.

- In the `QuickGrid` child content (RenderFragment), specify PropertyColumn<TGridItem,TProp>s, which represent `TGridItem` columns whose cells display values:
  - Property: Defines the value to be displayed in this column's cells.
  - Format: Optionally specifies a format string for the value. Using Format requires the `TProp` type to implement IFormattable.
  - Sortable: Indicates whether the data should be sortable by this column. The default value may vary according to the column type. For example, a TemplateColumn<TGridItem> is sorted if any SortBy parameter is specified.
  - InitialSortDirection: Indicates the sort direction if IsDefaultSortColumn is `true`.
  - IsDefaultSortColumn: Indicates whether this column should be sorted by default.
  - PlaceholderTemplate: If specified, virtualized grids use this template to render cells whose data hasn't been loaded.
  - HeaderTemplate: An optional template for this column's header cell. If not specified, the default header template includes the Title, along with any applicable sort indicators and options buttons.
  - Title: Title text for the column. The title is rendered automatically if HeaderTemplate isn't used.

For example, add the following component to render a grid.

For Blazor Web Apps, the `QuickGrid` component must adopt an interactive render mode to enable interactive features, such as paging and sorting.

`PromotionGrid.razor`:

razor

```
@page "/promotion-grid"
@using Microsoft.AspNetCore.Components.QuickGrid

<PageTitle>Promotion Grid</PageTitle>

<h1>Promotion Grid Example</h1>

<QuickGrid Items="people">
    <PropertyColumn Property="@(p => p.PersonId)" Sortable="true" />
    <PropertyColumn Property="@(p => p.Name)" Sortable="true" />
    <PropertyColumn Property="@(p => p.PromotionDate)" Format="yyyy-MM-dd"
Sortable="true" />
</QuickGrid>

@code {
    private record Person(int PersonId, string Name, DateOnly
PromotionDate);
```

```
    private IQueryable<Person> people = new[]
    {
        new Person(10895, "Jean Martin", new DateOnly(1985, 3, 16)),
        new Person(10944, "António Langa", new DateOnly(1991, 12, 1)),
        new Person(11203, "Julie Smith", new DateOnly(1958, 10, 10)),
        new Person(11205, "Nur Sari", new DateOnly(1922, 4, 27)),
        new Person(11898, "Jose Hernandez", new DateOnly(2011, 5, 3)),
        new Person(12130, "Kenji Sato", new DateOnly(2004, 1, 9)),
    }.AsQueryable();
}
```

Access the component in a browser at the relative path `/promotion-grid`.

There aren't current plans to extend `QuickGrid` with features that full-blown commercial grids tend to offer, for example, hierarchical rows, drag-to-reorder columns, or Excel-like range selections. If you require advanced features that you don't wish to develop on your own, continue using third-party grids.

Sort by column

The `QuickGrid` component can sort items by columns. In Blazor Web Apps, sorting requires the component to adopt an interactive render mode.

Add `Sortable="true"` (Sortable) to the PropertyColumn<TGridItem,TProp> tag:

razor

```
<PropertyColumn Property="..." Sortable="true" />
```

In the running app, sort the `QuickGrid` column by selecting the rendered column title.

Page items with a `Paginator` component

The `QuickGrid` component can page data from the data source. In Blazor Web Apps, paging requires the component to adopt an interactive render mode.

Add a PaginationState instance to the component's `@code` block. Set the ItemsPerPage to the number of items to display per page. In the following example, the instance is named `pagination`, and ten items per page is set:

C#

```
PaginationState pagination = new PaginationState { ItemsPerPage = 10 };
```

Set the `QuickGrid` component's Pagination property to `pagination`:

razor

```
<QuickGrid Items="..." Pagination="pagination">
```

To provide a UI for pagination, add a Paginator component above or below the `QuickGrid` component. Set the Paginator.State to `pagination`:

razor

```
<Paginator State="pagination" />
```

In the running app, page through the items using a rendered `Paginator` component.

QuickGrid renders additional empty rows to fill in the final page of data when used with a `Paginator` component. In .NET 9 or later, empty data cells (`<td></td>`) are added to the empty rows. The empty rows are intended to facilitate rendering the QuickGrid with stable row height and styling across all pages.

Apply row styles

Apply styles to rows using CSS isolation, which can include styling empty rows for `QuickGrid` components that page data with a Paginator component.

Wrap the `QuickGrid` component in a wrapper block element, for example a `<div>`:

diff

```
+ <div>
    <QuickGrid ...>
        ...
    </QuickGrid>
+ </div>
```

Apply a row style with the `::deep` pseudo-element. In the following example, row height is set to `2em`, including for empty data rows.

`{COMPONENT}.razor.css`:

```
```

```css
::deep tr {
    height: 2em;
}
```

Alternatively, use the following CSS styling approach:

- Display row cells populated with data.
- Don't display empty row cells, which avoids empty row cell borders from rendering per Bootstrap styling.

`{COMPONENT}.razor.css`:

```css
::deep tr:has(> td:not(:empty)) > td {
    display: table-cell;
}

::deep td:empty {
    display: none;
}
```

For more information on using `::deep` pseudo-elements with CSS isolation, see ASP.NET Core Blazor CSS isolation.

Custom attributes and styles

QuickGrid also supports passing custom attributes and style classes (Class) to the rendered table element:

```razor
<QuickGrid Items="..." custom-attribute="value" Class="custom-class">
```

Entity Framework Core (EF Core) data source

Use the factory pattern to resolve an EF Core database context that provides data to a `QuickGrid` component. For more information on why the factory pattern is recommended, see ASP.NET Core Blazor with Entity Framework Core (EF Core).

A database context factory (IDbContextFactory<TContext>) is injected into the component with the `@inject` directive. The factory approach requires disposal of the database context, so the component implements the IAsyncDisposable interface with the `@implements` directive. The item provider for the `QuickGrid` component is a `DbSet<T>` obtained from the created database context (CreateDbContext) of the injected database context factory.

QuickGrid recognizes EF-supplied IQueryable instances and knows how to resolve queries asynchronously for efficiency.

Add a package reference for the Microsoft.AspNetCore.Components.QuickGrid.EntityFrameworkAdapter NuGet package.

> **Note**
>
> For guidance on adding packages to .NET apps, see the articles under *Install and manage packages* at Package consumption workflow (NuGet documentation). Confirm correct package versions at NuGet.org.

Call AddQuickGridEntityFrameworkAdapter on the service collection in the `Program` file to register an EF-aware IAsyncQueryExecutor implementation:

C#

```
builder.Services.AddQuickGridEntityFrameworkAdapter();
```

The following example uses an `ExampleTable` DbSet<TEntity> (table) from a `AppDbContext` database context (`context`) as the data source for a `QuickGrid` component:

razor

```
@using Microsoft.AspNetCore.Components.QuickGrid
@using Microsoft.EntityFrameworkCore
@implements IAsyncDisposable
@inject IDbContextFactory<AppDbContext> DbFactory

...

<QuickGrid ... Items="context.ExampleTable" ...>
    ...
</QuickGrid>

@code {
```

```
    private AppDbContext context = default!;

    protected override void OnInitialized()
    {
        context = DbFactory.CreateDbContext();
    }

    public async ValueTask DisposeAsync() => await context.DisposeAsync();
}
```

In the code block (`@code`) of the preceding example:

- The `context` field holds the database context, typed as an `AppDbContext`.
- The `OnInitialized` lifecycle method assigns a new database context (CreateDbContext) to the `context` field from the injected factory (`DbFactory`).
- The asynchronous `DisposeAsync` method disposes of the database context when the component is disposed.

You may also use any EF-supported LINQ operator to filter the data before passing it to the Items parameter.

The following example filters movies by a movie title entered in a search box. The database context is `BlazorWebAppMoviesContext`, and the model is `Movie`. The movie's `Title` property is used for the filter operation.

razor

```
@using Microsoft.AspNetCore.Components.QuickGrid
@using Microsoft.EntityFrameworkCore
@implements IAsyncDisposable
@inject IDbContextFactory<BlazorWebAppMoviesContext> DbFactory

...

<p>
    <input type="search" @bind="titleFilter" @bind:event="oninput" />
</p>

<QuickGrid ... Items="FilteredMovies" ...>
    ...
</QuickGrid>

@code {
    private string titleFilter = string.Empty;
    private BlazorWebAppMoviesContext context = default!;

    protected override void OnInitialized()
    {
        context = DbFactory.CreateDbContext();
```

```
    }

    private IQueryable<Movie> FilteredMovies =>
        context.Movie.Where(m => m.Title!.Contains(titleFilter));

    public async ValueTask DisposeAsync() => await context.DisposeAsync();
}
```

For a working example, see the following resources:

- Build a Blazor movie database app tutorial
- Blazor movie database sample app: Select the latest version folder in the repository. The sample folder for the tutorial's project is named `BlazorWebAppMovies`.

Display name support

A column title can be assigned using ColumnBase<TGridItem>.Title in the PropertyColumn<TGridItem,TProp>'s tag. In the following movie example, the column is given the name "`Release Date`" for the column's movie release date data:

razor

```
<PropertyColumn Property="movie => movie.ReleaseDate" Title="Release Date"
/>
```

However, managing column titles (names) from bound model properties is usually a better choice for maintaining an app. A model can control the display name of a property with the [Display] attribute. In the following example, the model specifies a movie release date display name of "`Release Date`" for its `ReleaseDate` property:

C#

```
[Display(Name = "Release Date")]
public DateTime ReleaseDate { get; set; }
```

To enable the `QuickGrid` component to use the DisplayAttribute.Name, subclass PropertyColumn<TGridItem,TProp> either in the component or in a separate class:

C#

```
public class DisplayNameColumn<TGridItem, TProp> : PropertyColumn<TGridItem,
TProp>
{
    protected override void OnParametersSet()
```

```
    {
        if (Title is null && Property.Body is MemberExpression
memberExpression)
        {
            var memberInfo = memberExpression.Member;
            Title =
                memberInfo.GetCustomAttribute<DisplayNameAttribute>
().DisplayName ??
                memberInfo.GetCustomAttribute<DisplayAttribute>().Name ??
                memberInfo.Name;
        }

        base.OnParametersSet();
    }
}
```

Use the subclass in the `QuickGrid` component. In the following example, the preceding `DisplayNameColumn` is used. The name "`Release Date`" is provided by the [Display] attribute in the model, so there's no need to specify a Title:

razor

```
<DisplayNameColumn Property="movie => movie.ReleaseDate" />
```

The [DisplayName] attribute is also supported:

C#

```
[DisplayName("Release Date")]
public DateTime ReleaseDate { get; set; }
```

However, the `[Display]` attribute is recommended because it makes additional properties available. For example, the `[Display]` attribute offers the ability to assign a resource type for localization.

Remote data

In Blazor WebAssembly apps, fetching data from a JSON-based web API on a server is a common requirement. To fetch only the data that's required for the current page/viewport of data and apply sorting or filtering rules on the server, use the ItemsProvider parameter.

ItemsProvider can also be used in a server-side Blazor app if the app is required to query an external endpoint or in other cases where the requirements aren't covered by an IQueryable.

Supply a callback matching the GridItemsProvider<TGridItem> delegate type, where `TGridItem` is the type of data displayed in the grid. The callback is given a parameter of type GridItemsProviderRequest<TGridItem>, which specifies the start index, maximum row count, and sort order of data to return. In addition to returning the matching items, a total item count (`totalItemCount`) is also required for paging and virtualization to function correctly.

The following example obtains data from the public OpenFDA Food Enforcement database.

The GridItemsProvider<TGridItem> converts the GridItemsProviderRequest<TGridItem> into a query against the OpenFDA database. Query parameters are translated into the particular URL format supported by the external JSON API. It's only possible to perform sorting and filtering via sorting and filtering that's supported by the external API. The OpenFDA endpoint doesn't support sorting, so none of the columns are marked as sortable. However, it does support skipping records (`skip` parameter) and limiting the return of records (`limit` parameter), so the component can enable virtualization and scroll quickly through tens of thousands of records.

`FoodRecalls.razor`:

razor

```razor
@page "/food-recalls"
@inject HttpClient Http
@inject NavigationManager Navigation

<PageTitle>Food Recalls</PageTitle>

<h1>OpenFDA Food Recalls</h1>

<div class="grid" tabindex="-1">
    <QuickGrid ItemsProvider="@foodRecallProvider" Virtualize="true">
        <PropertyColumn Title="ID" Property="@(c => c.Event_Id)" />
        <PropertyColumn Property="@(c => c.State)" />
        <PropertyColumn Property="@(c => c.City)" />
        <PropertyColumn Title="Company" Property="@(c => c.Recalling_Firm)"
/>
        <PropertyColumn Property="@(c => c.Status)" />
    </QuickGrid>
</div>

<p>Total: <strong>@numResults results found</strong></p>

@code {
    private GridItemsProvider<FoodRecall>? foodRecallProvider;
    private int numResults;
```

```
    protected override async Task OnInitializedAsync()
    {
        foodRecallProvider = async req =>
        {
            var url = Navigation.GetUriWithQueryParameters(
                "https://api.fda.gov/food/enforcement.json",
                new Dictionary<string, object?>
            {
                { "skip", req.StartIndex },
                { "limit", req.Count },
            });

            var response = await
Http.GetFromJsonAsync<FoodRecallQueryResult>(
                url, req.CancellationToken);

            return GridItemsProviderResult.From(
                items: response!.Results,
                totalItemCount: response!.Meta.Results.Total);
        };

        numResults = (await Http.GetFromJsonAsync<FoodRecallQueryResult>(

"https://api.fda.gov/food/enforcement.json"))!.Meta.Results.Total;
    }
}
```

For more information on calling web APIs, see Call a web API from an ASP.NET Core Blazor app.

`QuickGrid` scaffolder

The `QuickGrid` scaffolder scaffolds Razor components with `QuickGrid` to display data from a database.

The scaffolder generates basic Create, Read, Update, and Delete (CRUD) pages based on an Entity Framework Core data model. You can scaffold individual pages or all of the CRUD pages. You select the model class and the `DbContext`, optionally creating a new `DbContext` if needed.

The scaffolded Razor components are added to the project's in a generated folder named after the model class. The generated `Index` component uses a `QuickGrid` component to display the data. Customize the generated components as needed and enable interactivity to take advantage of interactive features, such as paging, sorting and filtering.

The components produced by the scaffolder require server-side rendering (SSR), so they aren't supported when running on WebAssembly.

Visual Studio

Right-click on the `Components/Pages` folder and select **Add** > **New Scaffolded Item**.

With the **Add New Scaffold Item** dialog open to **Installed** > **Common** > **Blazor** > **Razor Component**, select **Razor Components using Entity Framework (CRUD)**. Select the **Add** button.

CRUD is an acronym for Create, Read, Update, and Delete. The scaffolder produces create, edit, delete, details, and index components for the app.

Complete the **Add Razor Components using Entity Framework (CRUD)** dialog:

- The **Template** dropdown list includes other templates for specifically creating create, edit, delete, details, and list components. This dropdown list comes in handy when you only need to create a specific type of component scaffolded to a model class. Leave the **Template** dropdown list set to **CRUD** to scaffold a full set of components.
- In the **Model class** dropdown list, select the model class. A folder is created for the generated components from the model name (if the model class is named `Movie`, the folder is automatically named `MoviePages`).
- For **DbContext class**, select an existing database context or select the + (plus sign) button and **Add Data Context** modal dialog to add a new database context.
- After the model dialog closes, the **Database provider** dropdown list defaults to **SQL Server**. You can select the appropriate provider for the database that you're using. The options include SQL Server, SQLite, PostgreSQL, and Azure Cosmos DB.
- Select **Add**.

For an example use of the `QuickGrid` scaffolder, see Build a Blazor movie database app (Overview).

Integrate ASP.NET Core Razor components with MVC or Razor Pages

Article • 11/18/2024

> **ⓘ Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the **.NET 9 version of this article**.

Razor components can be integrated into Razor Pages or MVC apps. When the page or view is rendered, components can be prerendered at the same time.

> **ⓘ Important**
>
> Framework changes across ASP.NET Core releases led to different sets of instructions in this article. Before using this article's guidance, confirm that the document version selector at the top of this article matches the version of ASP.NET Core that you intend to use for your app.

Prerendering can improve Search Engine Optimization (SEO) by rendering content for the initial HTTP response that search engines can use to calculate page rank.

After configuring the project, use the guidance in the following sections depending on the project's requirements:

- For components that are directly routable from user requests. Follow this guidance when visitors should be able to make an HTTP request in their browser for a component with an @page directive.
  - Use routable components in a Razor Pages app
  - Use routable components in an MVC app
- For components that aren't directly routable from user requests, see the Render components from a page or view section. Follow this guidance when the app embeds components into existing pages or views with the Component Tag Helper.

Configuration

Use the following guidance to integrate Razor components into pages or views of an existing Razor Pages or MVC app.

1. Add an imports file to the root folder of the project with the following content. Change the `{APP NAMESPACE}` placeholder to the namespace of the project.

   `_Imports.razor`:

   ```razor
   @using System.Net.Http
   @using Microsoft.AspNetCore.Authorization
   @using Microsoft.AspNetCore.Components.Authorization
   @using Microsoft.AspNetCore.Components.Forms
   @using Microsoft.AspNetCore.Components.Routing
   @using Microsoft.AspNetCore.Components.Web
   @using Microsoft.AspNetCore.Components.Web.Virtualization
   @using Microsoft.JSInterop
   @using {APP NAMESPACE}
   ```

2. In the project's layout file (`Pages/Shared/_Layout.cshtml` in Razor Pages apps or `Views/Shared/_Layout.cshtml` in MVC apps):

   - Add the following `<base>` tag and HeadOutlet component Tag Helper to the `<head>` element:

     ```CSHTML
     <base href="~/" />
     <component
     type="typeof(Microsoft.AspNetCore.Components.Web.HeadOutlet)"
         render-mode="ServerPrerendered" />
     ```

     The `href` value (the *app base path*) in the preceding example assumes that the app resides at the root URL path (`/`). If the app is a sub-application, follow the guidance in the *App base path* section of the Host and deploy ASP.NET Core Blazor article.

     The HeadOutlet component is used to render head (`<head>`) content for page titles (PageTitle component) and other head elements (HeadContent component) set by Razor components. For more information, see Control head content in ASP.NET Core Blazor apps.

   - Add a `<script>` tag for the `blazor.server.js` script immediately before the `Scripts` render section (`@await RenderSectionAsync(...)`):

HTML

```
<script src="_framework/blazor.server.js"></script>
```

The framework adds the `blazor.server.js` script to the app. There's no need to manually add a `blazor.server.js` script file to the app.

> **Note**
>
> Typically, the layout loads via a `_ViewStart.cshtml` file.

3. Register the Blazor Server services in `Program.cs` where services are registered:

C#

```
builder.Services.AddServerSideBlazor();
```

4. Add the Blazor Hub endpoint to the endpoints of `Program.cs` where routes are mapped. Place the following line after the call to `MapRazorPages` (Razor Pages) or `MapControllerRoute` (MVC):

C#

```
app.MapBlazorHub();
```

5. Integrate components into any page or view. For example, add a `Counter` component to the project's `Shared` folder.

   `Pages/Shared/Counter.razor` (Razor Pages) or `Views/Shared/Counter.razor` (MVC):

razor

```
<h1>Counter</h1>

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

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

@code {
    private int currentCount = 0;

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

```
    }
}
```

Razor Pages:

In the project's `Index` page of a Razor Pages app, add the `Counter` component's namespace and embed the component into the page. When the `Index` page loads, the `Counter` component is prerendered in the page. In the following example, replace the `{APP NAMESPACE}` placeholder with the project's namespace.

`Pages/Index.cshtml`:

CSHTML

```
@page
@using {APP NAMESPACE}.Pages.Shared
@model IndexModel
@{
    ViewData["Title"] = "Home page";
}

<component type="typeof(Counter)" render-mode="ServerPrerendered" />
```

MVC:

In the project's `Index` view of an MVC app, add the `Counter` component's namespace and embed the component into the view. When the `Index` view loads, the `Counter` component is prerendered in the page. In the following example, replace the `{APP NAMESPACE}` placeholder with the project's namespace.

`Views/Home/Index.cshtml`:

CSHTML

```
@using {APP NAMESPACE}.Views.Shared
@{
    ViewData["Title"] = "Home Page";
}

<component type="typeof(Counter)" render-mode="ServerPrerendered" />
```

For more information, see the Render components from a page or view section.

Use routable components in a Razor Pages app

This section pertains to adding components that are directly routable from user requests.

To support routable Razor components in Razor Pages apps:

1. Follow the guidance in the Configuration section.
2. Add an `App` component to the project root with the following content.

   `App.razor`:

   ```razor
   @using Microsoft.AspNetCore.Components.Routing

   <Router AppAssembly="typeof(App).Assembly">
       <Found Context="routeData">
           <RouteView RouteData="routeData" />
       </Found>
       <NotFound>
           <PageTitle>Not found</PageTitle>
           <p role="alert">Sorry, there's nothing at this address.</p>
       </NotFound>
   </Router>
   ```

3. Add a `_Host` page to the project with the following content. Replace the `{APP NAMESPACE}` placeholder with the app's namespace.

   `Pages/_Host.cshtml`:

   ```CSHTML
   @page
   @addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers

   <component type="typeof(App)" render-mode="ServerPrerendered" />
   ```

   > **Note**
   >
   > The preceding example assumes that the HeadOutlet component and Blazor script (`_framework/blazor.server.js`) are rendered by the app's layout. For more information, see the Configuration section.

   RenderMode configures whether the `App` component:

   - Is prerendered into the page.

- Is rendered as static HTML on the page or if it includes the necessary information to bootstrap a Blazor app from the user agent.

For more information on the Component Tag Helper, including passing parameters and RenderMode configuration, see Component Tag Helper in ASP.NET Core.

4. In the `Program.cs` endpoints, add a low-priority route for the `_Host` page as the last endpoint:

```
C#

app.MapFallbackToPage("/_Host");
```

5. Add routable components to the project. The following example is a `RoutableCounter` component based on the `Counter` component in the Blazor project templates.

`Pages/RoutableCounter.razor`:

```
razor

@page "/routable-counter"

<PageTitle>Routable Counter</PageTitle>

<h1>Routable Counter</h1>

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

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

@code {
    private int currentCount = 0;

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

6. Run the project and navigate to the routable `RoutableCounter` component at `/routable-counter`.

For more information on namespaces, see the Component namespaces section.

Use routable components in an MVC app

This section pertains to adding components that are directly routable from user requests.

To support routable Razor components in MVC apps:

1. Follow the guidance in the Configuration section.
2. Add an `App` component to the project root with the following content.

   `App.razor`:

   ```razor
   @using Microsoft.AspNetCore.Components.Routing

   <Router AppAssembly="typeof(App).Assembly">
       <Found Context="routeData">
           <RouteView RouteData="routeData" />
       </Found>
       <NotFound>
           <PageTitle>Not found</PageTitle>
           <p role="alert">Sorry, there's nothing at this address.</p>
       </NotFound>
   </Router>
   ```

3. Add a `_Host` view to the project with the following content. Replace the `{APP NAMESPACE}` placeholder with the app's namespace.

   `Views/Home/_Host.cshtml`:

   ```CSHTML
   @addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers

   <component type="typeof(App)" render-mode="ServerPrerendered" />
   ```

   > **Note**
   >
   > The preceding example assumes that the HeadOutlet component and Blazor script (`_framework/blazor.server.js`) are rendered by the app's layout. For more information, see the Configuration section.

   RenderMode configures whether the `App` component:

- Is prerendered into the page.
- Is rendered as static HTML on the page or if it includes the necessary information to bootstrap a Blazor app from the user agent.

For more information on the Component Tag Helper, including passing parameters and RenderMode configuration, see Component Tag Helper in ASP.NET Core.

4. Add an action to the Home controller.

`Controllers/HomeController.cs`:

```
C#

public IActionResult Blazor()
{
    return View("_Host");
}
```

5. In the `Program.cs` endpoints, add a low-priority route for the controller action that returns the `_Host` view:

```
C#

app.MapFallbackToController("Blazor", "Home");
```

6. Create a `Pages` folder in the MVC app and add routable components. The following example is a `RoutableCounter` component based on the `Counter` component in the Blazor project templates.

`Pages/RoutableCounter.razor`:

```
razor

@page "/routable-counter"

<PageTitle>Routable Counter</PageTitle>

<h1>Routable Counter</h1>

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

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

@code {
    private int currentCount = 0;
```

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

7. Run the project and navigate to the routable `RoutableCounter` component at `/routable-counter`.

For more information on namespaces, see the Component namespaces section.

Render components from a page or view

This section pertains to adding components to pages or views, where the components aren't directly routable from user requests.

To render a component from a page or view, use the Component Tag Helper.

Render stateful interactive components

Stateful interactive components can be added to a Razor page or view.

When the page or view renders:

- The component is prerendered with the page or view.
- The initial component state used for prerendering is lost.
- New component state is created when the SignalR connection is established.

The following Razor page renders a `Counter` component:

CSHTML

```
<h1>Razor Page</h1>

<component type="typeof(Counter)" render-mode="ServerPrerendered"
    param-InitialValue="InitialValue" />

@functions {
    [BindProperty(SupportsGet=true)]
    public int InitialValue { get; set; }
}
```

For more information, see Component Tag Helper in ASP.NET Core.

Render noninteractive components

In the following Razor page, the `Counter` component is statically rendered with an initial value that's specified using a form. Since the component is statically rendered, the component isn't interactive:

CSHTML

```
<h1>Razor Page</h1>

<form>
    <input type="number" asp-for="InitialValue" />
    <button type="submit">Set initial value</button>
</form>

<component type="typeof(Counter)" render-mode="Static"
    param-InitialValue="InitialValue" />

@functions {
    [BindProperty(SupportsGet=true)]
    public int InitialValue { get; set; }
}
```

For more information, see Component Tag Helper in ASP.NET Core.

Component namespaces

When using a custom folder to hold the project's Razor components, add the namespace representing the folder to either the page/view or to the `_ViewImports.cshtml` file. In the following example:

- Components are stored in the `Components` folder of the project.
- The `{APP NAMESPACE}` placeholder is the project's namespace. `Components` represents the name of the folder.

CSHTML

```
@using {APP NAMESPACE}.Components
```

The `_ViewImports.cshtml` file is located in the `Pages` folder of a Razor Pages app or the `Views` folder of an MVC app.

For more information, see ASP.NET Core Razor components.

Persist prerendered state

Without persisting prerendered state, state used during prerendering is lost and must be recreated when the app is fully loaded. If any state is setup asynchronously, the UI may flicker as the prerendered UI is replaced with temporary placeholders and then fully rendered again.

To persist state for prerendered components, use the Persist Component State Tag Helper (reference source). Add the Tag Helper's tag, `<persist-component-state />`, inside the closing `</body>` tag of the `_Host` page in an app that prerenders components.

> **Note**
>
> Documentation links to .NET reference source usually load the repository's default branch, which represents the current development for the next release of .NET. To select a tag for a specific release, use the **Switch branches or tags** dropdown list. For more information, see How to select a version tag of ASP.NET Core source code (dotnet/AspNetCore.Docs #26205).

In `Pages/_Host.cshtml` of Blazor apps that are `ServerPrerendered` in a Blazor Server app:

CSHTML

```
<body>
    ...

    <persist-component-state />
</body>
```

Decide what state to persist using the PersistentComponentState service. PersistentComponentState.RegisterOnPersisting registers a callback to persist the component state before the app is paused. The state is retrieved when the application resumes.

In the following example:

- The `{TYPE}` placeholder represents the type of data to persist (for example, `WeatherForecast[]`).
- The `{TOKEN}` placeholder is a state identifier string (for example, `fetchdata`).

razor

```
@implements IDisposable
@inject PersistentComponentState ApplicationState
```

```
...

@code {
    private {TYPE} data;
    private PersistingComponentStateSubscription persistingSubscription;

    protected override async Task OnInitializedAsync()
    {
        persistingSubscription =
            ApplicationState.RegisterOnPersisting(PersistData);

        if (!ApplicationState.TryTakeFromJson<{TYPE}>(
            "{TOKEN}", out var restored))
        {
            data = await ...;
        }
        else
        {
            data = restored!;
        }
    }

    private Task PersistData()
    {
        ApplicationState.PersistAsJson("{TOKEN}", data);

        return Task.CompletedTask;
    }

    void IDisposable.Dispose()
    {
        persistingSubscription.Dispose();
    }
}
```

The following example is an updated version of the `FetchData` component based on the Blazor project template. The `WeatherForecastPreserveState` component persists weather forecast state during prerendering and then retrieves the state to initialize the component. The Persist Component State Tag Helper persists the component state after all component invocations.

`Pages/WeatherForecastPreserveState.razor`:

razor

```
@page "/weather-forecast-preserve-state"
@using BlazorSample.Shared
@implements IDisposable
@inject IWeatherForecastService WeatherForecastService
@inject PersistentComponentState ApplicationState
```

```
<PageTitle>Weather Forecast</PageTitle>

<h1>Weather forecast</h1>

<p>This component demonstrates fetching data from the server.</p>

@if (forecasts == null)
{
    <p><em>Loading...</em></p>
}
else
{
    <table class="table">
        <thead>
            <tr>
                <th>Date</th>
                <th>Temp. (C)</th>
                <th>Temp. (F)</th>
                <th>Summary</th>
            </tr>
        </thead>
        <tbody>
            @foreach (var forecast in forecasts)
            {
                <tr>
                    <td>@forecast.Date.ToShortDateString()</td>
                    <td>@forecast.TemperatureC</td>
                    <td>@forecast.TemperatureF</td>
                    <td>@forecast.Summary</td>
                </tr>
            }
        </tbody>
    </table>
}

@code {
    private WeatherForecast[] forecasts = Array.Empty<WeatherForecast>();
    private PersistingComponentStateSubscription persistingSubscription;

    protected override async Task OnInitializedAsync()
    {
        persistingSubscription =
            ApplicationState.RegisterOnPersisting(PersistForecasts);

        if (!ApplicationState.TryTakeFromJson<WeatherForecast[]>(
            "fetchdata", out var restored))
        {
            forecasts =
                await
WeatherForecastService.GetForecastAsync(DateOnly.FromDateTime(DateTime.Now))
;
        }
        else
        {
            forecasts = restored!;
```

```
        }
    }

    private Task PersistForecasts()
    {
        ApplicationState.PersistAsJson("fetchdata", forecasts);

        return Task.CompletedTask;
    }

    void IDisposable.Dispose()
    {
        persistingSubscription.Dispose();
    }
}
```

By initializing components with the same state used during prerendering, any expensive initialization steps are only executed once. The rendered UI also matches the prerendered UI, so no flicker occurs in the browser.

The persisted prerendered state is transferred to the client, where it's used to restore the component state. ASP.NET Core Data Protection ensures that the data is transferred securely in Blazor Server apps.

Prerendered state size and SignalR message size limit

A large prerendered state size may exceed Blazor's SignalR circuit message size limit, which results in the following:

- The SignalR circuit fails to initialize with an error on the client: Circuit host not initialized.
- The reconnection UI on the client appears when the circuit fails. Recovery isn't possible.

To resolve the problem, use ***either*** of the following approaches:

- Reduce the amount of data that you are putting into the prerendered state.
- Increase the SignalR message size limit. ***WARNING***: Increasing the limit may increase the risk of Denial of Service (DoS) attacks.

Additional Blazor Server resources

- State management: Handle prerendering

- Razor component lifecycle subjects that pertain to prerendering
  - Component initialization (OnInitialized{Async})
  - After component render (OnAfterRender{Async})
  - Stateful reconnection after prerendering
  - Prerendering with JavaScript interop
- Authentication and authorization: General aspects
- Handle Errors: Prerendering
- Host and deploy: Blazor Server
- Threat mitigation: Cross-site scripting (XSS)
- OnNavigateAsync is executed *twice* when prerendering: Handle asynchronous navigation events with OnNavigateAsync

Consume ASP.NET Core Razor components from a Razor class library (RCL)

Article • 09/12/2024

> **Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the **.NET 9 version of this article**.

Components can be shared in a Razor class library (RCL) across projects. Include components and static assets in an app from:

- Another project in the solution.
- A referenced .NET library.
- A NuGet package.

Just as components are regular .NET types, components provided by an RCL are normal .NET assemblies.

Create an RCL

Visual Studio

1. Create a new project.
2. In the **Create a new project** dialog, select **Razor Class Library** from the list of ASP.NET Core project templates. Select **Next**.
3. In the **Configure your new project** dialog, provide a project name in the **Project name** field. Examples in this topic use the project name `ComponentLibrary`. Select **Next**.
4. In the **Additional information** dialog, don't select **Support pages and views**. Select **Create**.
5. Add the RCL to a solution:
   a. Open the solution.
   b. Right-click the solution in **Solution Explorer**. Select **Add** > **Existing Project**.

c. Navigate to the RCL's project file.
d. Select the RCL's project file (`.csproj`).

6. Add a reference to the RCL from the app:
   a. Right-click the app project. Select **Add** > **Project Reference**.
   b. Select the RCL project. Select **OK**.

Consume a Razor component from an RCL

To consume components from an RCL in another project, use either of the following approaches:

- Use the full component type name, which includes the RCL's namespace.
- Individual components can be added by name without the RCL's namespace if Razor's @using directive declares the RCL's namespace. Use the following approaches:
  - Add the `@using` directive to individual components.
  - include the `@using` directive in the top-level `_Imports.razor` file to make the library's components available to an entire project. Add the directive to an `_Imports.razor` file at any level to apply the namespace to a single component or set of components within a folder. When an `_Imports.razor` file is used, individual components don't require an `@using` directive for the RCL's namespace.

In the following examples, `ComponentLibrary` is an RCL containing the `Component1` component. The `Component1` component is an example component automatically added to an RCL created from the RCL project template that isn't created to support pages and views.

`Component1.razor` in the `ComponentLibrary` RCL:

razor

```
<div class="my-component">
    This component is defined in the <strong>ComponentLibrary</strong>
package.
</div>
```

In the app that consumes the RCL, reference the `Component1` component using its namespace, as the following example shows.

`ConsumeComponent1.razor`:

razor

```razor
@page "/consume-component-1"

<h1>Consume component (full namespace example)</h1>

<ComponentLibrary.Component1 />
```

Alternatively, add a @using directive and use the component without its namespace. The following `@using` directive can also appear in any `_Imports.razor` file in or above the current folder.

`ConsumeComponent2.razor`:

razor

```razor
@page "/consume-component-2"
@using ComponentLibrary

<h1>Consume component (<code>@@using</code> example)</h1>

<Component1 />
```

For library components that use CSS isolation, the component styles are automatically made available to the consuming app. There's no need to manually link or import the library's individual component stylesheets or its bundled CSS file in the app that consumes the library. The app uses CSS imports to reference the RCL's bundled styles. The bundled styles aren't published as a static web asset of the app that consumes the library. For a class library named `ClassLib` and a Blazor app with a `BlazorSample.styles.css` stylesheet, the RCL's stylesheet is imported at the top of the app's stylesheet automatically at build time:

css

```css
@import '_content/ClassLib/ClassLib.bundle.scp.css';
```

For the preceding examples, `Component1`'s stylesheet (`Component1.razor.css`) is bundled automatically.

`Component1.razor.css` in the `ComponentLibrary` RCL:

css

```css
.my-component {
    border: 2px dashed red;
```

```
    padding: 1em;
    margin: 1em 0;
    background-image: url('background.png');
}
```

The background image is also included from the RCL project template and resides in the `wwwroot` folder of the RCL.

`wwwroot/background.png` in the `ComponentLibrary` RCL:

To provide additional library component styles from stylesheets in the library's `wwwroot` folder, add stylesheet `<link>` tags to the RCL's consumer, as the next example demonstrates.

> **ⓘ Important**
>
> Generally, library components use CSS isolation to bundle and provide component styles. Component styles that rely upon CSS isolation are automatically made available to the app that uses the RCL. There's no need to manually link or import the library's individual component stylesheets or its bundled CSS file in the app that consumes the library. The following example is for providing global stylesheets *outside of CSS isolation*, which usually isn't a requirement for typical apps that consume RCLs.

The following background image is used in the next example. If you implement the example shown in this section, right-click the image to save it locally.

`wwwroot/extra-background.png` in the `ComponentLibrary` RCL:

Add a new stylesheet to the RCL with an `extra-style` class.

`wwwroot/additionalStyles.css` in the `ComponentLibrary` RCL:

```
css

.extra-style {
    border: 2px dashed blue;
    padding: 1em;
    margin: 1em 0;
```

```
    background-image: url('extra-background.png');
}
```

Add a component to the RCL that uses the `extra-style` class.

`ExtraStyles.razor` in the `ComponentLibrary` RCL:

razor

```
<div class="extra-style">
    <p>
        This component is defined in the <strong>ComponentLibrary</strong>
package.
    </p>
</div>
```

Add a page to the app that uses the `ExtraStyles` component from the RCL.

`ConsumeComponent3.razor`:

razor

```
@page "/consume-component-3"
@using ComponentLibrary

<h1>Consume component (<code>additionalStyles.css</code> example)</h1>

<ExtraStyles />
```

Link to the library's stylesheet in the app's `<head>` markup (location of <head> content):

Blazor Web Apps:

HTML

```
<link href="@Assets["_content/ComponentLibrary/additionalStyles.css"]"
rel="stylesheet">
```

Standalone Blazor WebAssembly apps:

HTML

```
<link href="_content/ComponentLibrary/additionalStyles.css"
rel="stylesheet">
```

Make routable components available from the RCL

To make routable components in the RCL available for direct requests, the RCL's assembly must be disclosed to the app's router.

Open the app's `App` component (`App.razor`). Assign an Assembly collection to the AdditionalAssemblies parameter of the Router component to include the RCL's assembly. In the following example, the `ComponentLibrary.Component1` component is used to discover the RCL's assembly.

razor

```
AdditionalAssemblies="new[] { typeof(ComponentLibrary.Component1).Assembly
}"
```

For more information, see ASP.NET Core Blazor routing and navigation.

Create an RCL with static assets in the `wwwroot` folder

An RCL's static assets are available to any app that consumes the library.

Place static assets in the `wwwroot` folder of the RCL and reference the static assets with the following path in the app: `_content/{PACKAGE ID}/{PATH AND FILE NAME}`. The `{PACKAGE ID}` placeholder is the library's package ID. The package ID defaults to the project's assembly name if `<PackageId>` isn't specified in the project file. The `{PATH AND FILE NAME}` placeholder is path and file name under `wwwroot`. This path format is also used in the app for static assets supplied by NuGet packages added to the RCL.

The following example demonstrates the use of RCL static assets with an RCL named `ComponentLibrary` and a Blazor app that consumes the RCL. The app has a project reference for the `ComponentLibrary` RCL.

The following Jeep® image is used in this section's example. If you implement the example shown in this section, right-click the image to save it locally.

`wwwroot/jeep-yj.png` in the `ComponentLibrary` RCL:

Add the following `JeepYJ` component to the RCL.

`JeepYJ.razor` in the `ComponentLibrary` RCL:

razor

```
<h3>ComponentLibrary.JeepYJ</h3>

<p>
    <img alt="Jeep YJ&reg;" src="_content/ComponentLibrary/jeep-yj.png" />
</p>
```

Add the following `Jeep` component to the app that consumes the `ComponentLibrary` RCL. The `Jeep` component uses:

- The Jeep YJ® image from the `ComponentLibrary` RCL's `wwwroot` folder.
- The `JeepYJ` component from the RCL.

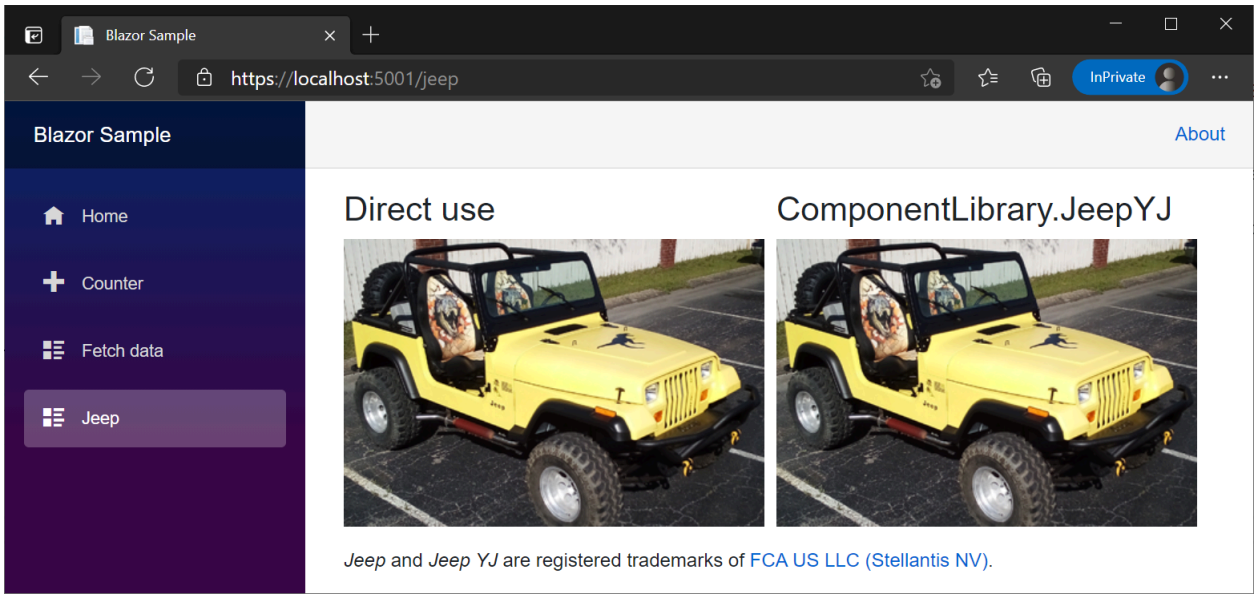
`Jeep.razor`:

razor

```
@page "/jeep"
@using ComponentLibrary

<div style="float:left;margin-right:10px">
    <h3>Direct use</h3>

    <p>
        <img alt="Jeep YJ&reg;" src="_content/ComponentLibrary/jeep-yj.png"
/>
    </p>
</div>

<JeepYJ />

<p>
```

```
    <em>Jeep</em> and <em>Jeep YJ</em> are registered trademarks of
    <a href="https://www.stellantis.com">FCA US LLC (Stellantis NV)</a>.
</p>
```

Rendered `Jeep` component:



For more information, see Reusable Razor UI in class libraries with ASP.NET Core.

Create an RCL with JavaScript files collocated with components

Collocation of JavaScript (JS) files for Razor components is a convenient way to organize scripts in an app.

Razor components of Blazor apps collocate JS files using the `.razor.js` extension and are publicly addressable using the path to the file in the project:

`{PATH}/{COMPONENT}.razor.js`

- The `{PATH}` placeholder is the path to the component.
- The `{COMPONENT}` placeholder is the component.

When the app is published, the framework automatically moves the script to the web root. Scripts are moved to `bin/Release/{TARGET FRAMEWORK MONIKER}/publish/wwwroot/{PATH}/{COMPONENT}.razor.js`, where the placeholders are:

- `{TARGET FRAMEWORK MONIKER}` is the Target Framework Moniker (TFM).
- `{PATH}` is the path to the component.
- `{COMPONENT}` is the component name.

No change is required to the script's relative URL, as Blazor takes care of placing the JS file in published static assets for you.

This section and the following examples are primarily focused on explaining JS file collocation. The first example demonstrates a collocated JS file with an ordinary JS function. The second example demonstrates the use of a module to load a function, which is the recommended approach for most production apps. Calling JS from .NET is fully covered in Call JavaScript functions from .NET methods in ASP.NET Core Blazor, where there are further explanations of the Blazor JS API with additional examples. Component disposal, which is present in the second example, is covered in ASP.NET Core Razor component lifecycle.

The following `JsCollocation1` component loads a script via a HeadContent component and calls a JS function with IJSRuntime.InvokeAsync. The `{PATH}` placeholder is the path to the component.

> **ⓘ Important**
>
> If you use the following code for a demonstration in a test app, change the `{PATH}` placeholder to the path of the component (example: `Components/Pages` in .NET 8 or later or `Pages` in .NET 7 or earlier). In a Blazor Web App (.NET 8 or later), the component requires an interactive render mode applied either globally to the app or to the component definition.

Add the following script after the Blazor script (location of the Blazor start script):

HTML

```
<script src="{PATH}/JsCollocation1.razor.js"></script>
```

`JsCollocation1` component (`{PATH}/JsCollocation1.razor`):

razor

```
@page "/js-collocation-1"
@inject IJSRuntime JS

<PageTitle>JS Collocation 1</PageTitle>

<h1>JS Collocation Example 1</h1>

<button @onclick="ShowPrompt">Call showPrompt1</button>

@if (!string.IsNullOrEmpty(result))
```

```
{
    <p>
        Hello @result!
    </p>
}

@code {
    private string? result;

    public async void ShowPrompt()
    {
        result = await JS.InvokeAsync<string>(
            "showPrompt1", "What's your name?");
        StateHasChanged();
    }
}
```

The collocated JS file is placed next to the `JsCollocation1` component file with the file name `JsCollocation1.razor.js`. In the `JsCollocation1` component, the script is referenced at the path of the collocated file. In the following example, the `showPrompt1` function accepts the user's name from a Window prompt() and returns it to the `JsCollocation1` component for display.

`{PATH}/JsCollocation1.razor.js`:

JavaScript

```
function showPrompt1(message) {
  return prompt(message, 'Type your name here');
}
```

The preceding approach isn't recommended for general use in production apps because the approach pollutes the client with global functions. A better approach for production apps is to use JS modules. The same general principles apply to loading a JS module from a collocated JS file, as the next example demonstrates.

The following `JsCollocation2` component's `OnAfterRenderAsync` method loads a JS module into `module`, which is an IJSObjectReference of the component class. `module` is used to call the `showPrompt2` function. The `{PATH}` placeholder is the path to the component.

> **ⓘ Important**
>
> If you use the following code for a demonstration in a test app, change the `{PATH}` placeholder to the path of the component. In a Blazor Web App (.NET 8 or later),

the component requires an interactive render mode applied either globally to the app or to the component definition.

`JsCollocation2` component (`{PATH}/JsCollocation2.razor`):

razor

```
@page "/js-collocation-2"
@implements IAsyncDisposable
@inject IJSRuntime JS

<PageTitle>JS Collocation 2</PageTitle>

<h1>JS Collocation Example 2</h1>

<button @onclick="ShowPrompt">Call showPrompt2</button>

@if (!string.IsNullOrEmpty(result))
{
    <p>
        Hello @result!
    </p>
}

@code {
    private IJSObjectReference? module;
    private string? result;

    protected async override Task OnAfterRenderAsync(bool firstRender)
    {
        if (firstRender)
        {
            /*
                Change the {PATH} placeholder in the next line to the path
of
                the collocated JS file in the app. Examples:

                ./Components/Pages/JsCollocation2.razor.js (.NET 8 or later)
                ./Pages/JsCollocation2.razor.js (.NET 7 or earlier)
            */
            module = await JS.InvokeAsync<IJSObjectReference>("import",
                "./{PATH}/JsCollocation2.razor.js");
        }
    }

    public async void ShowPrompt()
    {
        if (module is not null)
        {
            result = await module.InvokeAsync<string>(
                "showPrompt2", "What's your name?");
            StateHasChanged();
```

```
        }
    }

    async ValueTask IAsyncDisposable.DisposeAsync()
    {
        if (module is not null)
        {
            await module.DisposeAsync();
        }
    }
}
```

`{PATH}/JsCollocation2.razor.js`:

JavaScript

```
export function showPrompt2(message) {
  return prompt(message, 'Type your name here');
}
```

Use of scripts and modules for collocated JS in a Razor class library (RCL) is only supported for Blazor's JS interop mechanism based on the IJSRuntime interface. If you're implementing JavaScript [JSImport]/[JSExport] interop, see JavaScript JSImport/JSExport interop with ASP.NET Core Blazor.

For scripts or modules provided by a Razor class library (RCL) using IJSRuntime-based JS interop, the following path is used:

`./_content/{PACKAGE ID}/{PATH}/{COMPONENT}.{EXTENSION}.js`

- The path segment for the current directory (`./`) is required in order to create the correct static asset path to the JS file.
- The `{PACKAGE ID}` placeholder is the RCL's package identifier (or library name for a class library referenced by the app).
- The `{PATH}` placeholder is the path to the component. If a Razor component is located at the root of the RCL, the path segment isn't included.
- The `{COMPONENT}` placeholder is the component name.
- The `{EXTENSION}` placeholder matches the extension of component, either `razor` or `cshtml`.

In the following Blazor app example:

- The RCL's package identifier is `AppJS`.
- A module's scripts are loaded for the `JsCollocation3` component (`JsCollocation3.razor`).

- The `JsCollocation3` component is in the `Components/Pages` folder of the RCL.

C#

```
module = await JS.InvokeAsync<IJSObjectReference>("import",
    "./_content/AppJS/Components/Pages/JsCollocation3.razor.js");
```

Client-side browser compatibility analyzer

Client-side apps target the full .NET API surface area, but not all .NET APIs are supported on WebAssembly due to browser sandbox constraints. Unsupported APIs throw PlatformNotSupportedException when running on WebAssembly. A platform compatibility analyzer warns the developer when the app uses APIs that aren't supported by the app's target platforms. For client-side apps, this means checking that APIs are supported in browsers. Annotating .NET framework APIs for the compatibility analyzer is an on-going process, so not all .NET framework API is currently annotated.

Blazor Web Apps that enable Interactive WebAssembly components, Blazor WebAssembly apps, and RCL projects *automatically* enable browser compatibility checks by adding `browser` as a supported platform with the `SupportedPlatform` MSBuild item. Library developers can manually add the `SupportedPlatform` item to a library's project file to enable the feature:

XML

```
<ItemGroup>
  <SupportedPlatform Include="browser" />
</ItemGroup>
```

When authoring a library, indicate that a particular API isn't supported in browsers by specifying `browser` to UnsupportedOSPlatformAttribute:

C#

```
using System.Runtime.Versioning;

...

[UnsupportedOSPlatform("browser")]
private static string GetLoggingDirectory()
{
    ...
}
```

For more information, see Annotating APIs as unsupported on specific platforms (dotnet/designs GitHub repository).

JavaScript isolation in JavaScript modules

Blazor enables JavaScript isolation in standard JavaScript modules. JavaScript isolation provides the following benefits:

- Imported JavaScript no longer pollutes the global namespace.
- Consumers of the library and components aren't required to manually import the related JavaScript.

For more information, see Call JavaScript functions from .NET methods in ASP.NET Core Blazor.

Avoid trimming JavaScript-invokable .NET methods

Runtime relinking trims class instance JavaScript-invokable .NET methods unless they're explicitly preserved. For more information, see Call .NET methods from JavaScript functions in ASP.NET Core Blazor.

Build, pack, and ship to NuGet

Because Razor class libraries that contain Razor components are standard .NET libraries, packing and shipping them to NuGet is no different from packing and shipping any library to NuGet. Packing is performed using the dotnet pack command in a command shell:

.NET CLI

```
dotnet pack
```

Upload the package to NuGet using the dotnet nuget push command in a command shell.

Trademarks

Jeep and *Jeep YJ* are registered trademarks of FCA US LLC (Stellantis NV).

Additional resources

- Reusable Razor UI in class libraries with ASP.NET Core
- Use ASP.NET Core APIs in a class library
- Add an XML Intermediate Language (IL) Trimmer configuration file to a library
- CSS isolation support with Razor class libraries

ASP.NET Core Razor class libraries (RCLs) with static server-side rendering (static SSR)

Article • 09/27/2024

This article provides guidance for component library authors considering support for static server-side rendering (static SSR).

Blazor encourages the development of an ecosystem of open-source and commercial component libraries, formally called *Razor class libraries (RCLs)*. Developers might also create reusable components for sharing components privately across apps within their own companies. Ideally, components are developed for compatibility with as many hosting models and rendering modes as possible. Static SSR introduces additional restrictions that can be more challenging to support than interactive rendering modes.

Understand the capabilities and restrictions of static SSR

Static SSR is a mode in which components run when the server handles an incoming HTTP request. Blazor renders the component as HTML, which is included in the response. Once the response is sent, the server-side component and renderer state is discarded, so all that remains is HTML in the browser.

The benefit of this mode is cheaper, more scalable hosting, because no ongoing server resources are required to hold component state, no ongoing connection must be maintained between browser and server, and no WebAssembly payload is required in the browser.

All existing components can still be used with static SSR. However, the cost of this mode is that event handlers, such as `@onclick`†, can't be run for the following reasons:

- There's no .NET code in the browser to run them.
- The server has immediately discarded any component and renderer state that would be needed to execute event handlers or to rerender the same component instances.

†There's a special exception for the `@onsubmit` event handler for forms, which is always functional, regardless of render mode.

This is equivalent to how components behave during prerendering, before a Blazor circuit or the .NET WebAssembly runtime is started.

For components whose only role is to produce read-only DOM content, these behaviors for static SSR are completely sufficient. However, library authors must consider what approach to take when including interactive components in their libraries.

Options for component authors

There are three main approaches:

- **Don't use interactive behaviors** (Basic)

  For components whose only role is to produce read-only DOM content, the developer isn't required to take any special action. These components naturally work with any render mode.

  Examples:
  - A "user card" component that loads data corresponding to a person and renders it in a stylized UI with a photo, job title, and other details.
  - A "video" component that acts as a wrapper around the HTML `<video>` element, making it more convenient to use in a Razor component.

- **Require interactive rendering** (Basic)

  You can choose to require that your component is only used with interactive rendering. This limits the applicability of your component, but means that you may freely rely on arbitrary event handlers. Even then, you should still avoid declaring a specific `@rendermode` and permit the app author who consumes your library to select one.

  Examples:
  - A video editing component in which users may splice and re-order segments of video. Even if there was a way to represent these edit operations with plain HTML buttons and form posts, the user experience would be unacceptable without true interactivity.
  - A collaborative document editor that must show the activities of other users in real time.

- **Use interactive behaviors, but design for static SSR and progressive enhancement** (Advanced)

Many interactive behaviors can be implemented using only HTML capabilities. With a good understanding of HTML and CSS, you can often produce a useful baseline of functionality that works with static SSR. You can still declare event handlers that implement more advanced, optional behaviors, which only work in interactive render modes.

Examples:

- A grid component. Under static SSR, the component may only support displaying data and navigating between pages (implemented with `<a>` links). When used with interactive rendering, the component may add live sorting and filtering.
- A tabset component. As long as navigation between tabs is achieved using `<a>` links and state is held only in URL query parameters, the component can work without `@onclick`.
- An advanced file upload component. Under static SSR, the component may behave as a native `<input type=file>`. When used with interactive rendering, the component could also display upload progress.
- A stock ticker. Under static SSR, the component may display the stock quote at the time the HTML was rendered. When used with interactive rendering, the component may then update the stock price in real time.

For any of these strategies, there's also the option of implementing interactive features with JavaScript.

To choose among these approaches, reusable Razor component authors must make a cost/benefit tradeoff. Your component is more useful and has a broader potential user base if it supports all render modes, including static SSR. However, it takes more work to design and implement a component that supports and takes best advantage of each render mode.

When to use the `@rendermode` directive

In most cases, reusable component authors should **not** specify a render mode, even when interactivity is required. This is because the component author doesn't know whether the app enables support for InteractiveServer, InteractiveWebAssembly, or both with InteractiveAuto. By not specifying a `@rendermode`, the component author leaves the choice to the app developer.

Even if the component author thinks that interactivity is required, there may still be cases where an app author considers it sufficient to use static SSR alone. For example, a map component with drag and zoom interactivity may seem to require interactivity.

However, some scenarios may only call for rendering a static map image and avoiding drag/zoom features.

The only reason why a reusable component author should use the `@rendermode` directive on their component is if the implementation is fundamentally coupled to one specific render mode and would certainly cause an error if used in a different mode. Consider a component with a core purpose of interacting directly with the host OS using Windows or Linux-specific APIs. It might be impossible to use such a component on WebAssembly. If so, it's reasonable to declare `@rendermode InteractiveServer` for the component.

Streaming rendering

Reusable Razor components are free to declare `@attribute [StreamRendering]` for streaming rendering ([StreamRendering] attribute API). This results in incremental UI updates during static SSR. Since the same data-loading patterns produce incremental UI updates during interactive rendering, regardless of the presence of the `[StreamRendering]` attribute, the component can behave correctly in all cases. Even in cases where streaming static SSR is suppressed on the server, the component still renders its correct final state.

Using links across render modes

Reusable Razor components may use links and enhanced navigation. HTML `<a>` tags should produce equivalent behaviors with or without an interactive Router component and whether or not enhanced navigation is enabled/disabled at an ancestor level in the DOM.

Using forms across render modes

Reusable Razor components may include forms (either `<form>` or `<EditForm>`), as these can be implemented to work equivalently across both static and interactive render modes.

Consider the following example:

razor

```
<EditForm Enhance FormName="NewProduct" Model="Model"
OnValidSubmit="SaveProduct">
    <DataAnnotationsValidator />
```

```
    <ValidationSummary />

    <p><label>Name: <InputText @bind-Value="Item.Name" /></label></p>

    <button type="submit">Submit</button>
</EditForm>

@code {
    [SupplyParameterFromForm]
    private Product? Model { get; set; }

    protected override void OnInitialized() => Model ??= new();

    private async Task Save()
    {
        ...
    }
}
```

The Enhance, FormName, and SupplyParameterFromFormAttribute APIs are only used during static SSR and are ignored during interactive rendering. The form works correctly during both interactive and static SSR.

Avoid APIs that are specific to static SSR

HttpContext is only available during static SSR, so don't rely on HttpContext when creating components that work across render modes. The HttpContext API doesn't make sense to use during interactive rendering either because there's no active HTTP request in flight at those times. It's meaningless to think about setting a HTTP status code or writing to the HTTP response.

Reusable components are free to receive an HttpContext when available, as follows:

C#

```
[CascadingParameter]
public HttpContext? Context { get; set; }
```

The value is `null` during interactive rendering and is only set during static SSR.

In many cases, there are better alternatives than using HttpContext. If you need to know the current URL or to perform a redirection, the APIs on NavigationManager work with all render modes. If you need to know the user's authentication state, use Blazor's AuthenticationStateProvider service (AuthenticationStateProvider documentation) over using HttpContext.User.

Use Razor components in JavaScript apps and SPA frameworks

Article • 11/06/2024

> **Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the .NET 9 version of this article.

This article covers how to render Razor components from JavaScript, use Blazor custom elements, and generate Angular and React components.

> **Note**
>
> We recommend using the `blazor.server.js` (Blazor Server) and `blazor.webassembly.js` (Blazor WebAssembly) scripts when integrating Razor components into an existing JavaScript app until better support for the `blazor.web.js` (Blazor Web App) script is added in the future. For more information, see RegisterCustomElement stopped working in Blazor 8 (dotnet/aspnetcore #53920).

Angular sample apps

- CustomElementsBlazorSample (Blazor Server) (javiercn/CustomElementsBlazorSample, branch: blazor-server): Blazor Server is supported in .NET 8/9. To migrate this .NET 7 sample, see Migrate from ASP.NET Core 7.0 to 8.0 and Migrate from ASP.NET Core in .NET 8 to ASP.NET Core in .NET 9.
- CustomElementsBlazorSample (Blazor WebAssembly) (javiercn/CustomElementsBlazorSample, branch: blazor-wasm): To migrate this .NET 7 sample, see Migrate from ASP.NET Core 7.0 to 8.0 and Migrate from ASP.NET Core in .NET 8 to ASP.NET Core in .NET 9.

Render Razor components from JavaScript

Razor components can be dynamically-rendered from JavaScript (JS) for existing JS apps.

The example in this section renders the following Razor component into a page via JS.

`Quote.razor`:

razor

```razor
<div class="m-5 p-5">
    <h2>Quote</h2>
    <p>@Text</p>
</div>

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

In the `Program` file, add the namespace for the location of the component.

Call RegisterForJavaScript on the app's root component collection to register a Razor component as a root component for JS rendering.

RegisterForJavaScript includes an overload that accepts the name of a JS function that executes initialization logic (`javaScriptInitializer`). The JS function is called once per component registration immediately after the Blazor app starts and before any components are rendered. This function can be used for integration with JS technologies, such as HTML custom elements or a JS-based SPA framework.

One or more initializer functions can be created and called by different component registrations. The typical use case is to reuse the same initializer function for multiple components, which is expected if the initializer function is configuring integration with custom elements or another JS-based SPA framework.

> **ⓘ Important**
>
> Don't confuse the `javaScriptInitializer` parameter of RegisterForJavaScript with JavaScript initializers. The name of the parameter and the JS initializers feature is coincidental.

The following example demonstrates the dynamic registration of the preceding `Quote` component with "`quote`" as the identifier.

- In a Blazor Server app, modify the call to AddServerSideBlazor in the `Program` file:

  C#

  ```
  builder.Services.AddServerSideBlazor(options =>
  {
      options.RootComponents.RegisterForJavaScript<Quote>(identifier:
  "quote",
          javaScriptInitializer: "initializeComponent");
  });
  ```

- In a Blazor WebAssembly app, call RegisterForJavaScript on RootComponents in the client-side `Program` file:

  C#

  ```
  builder.RootComponents.RegisterForJavaScript<Quote>(identifier:
  "quote",
      javaScriptInitializer: "initializeComponent");
  ```

Attach the initializer function with `name` and `parameters` function parameters to the `window` object. For demonstration purposes, the following `initializeComponent` function logs the name and parameters of the registered component.

`wwwroot/jsComponentInitializers.js`:

JavaScript

```
window.initializeComponent = (name, parameters) => {
  console.log({ name: name, parameters: parameters });
}
```

Render the component from JS into a container element using the registered identifier, passing component parameters as needed.

In the following example:

- The `Quote` component (`quote` identifier) is rendered into the `quoteContainer` element when the `showQuote` function is called.
- A quote string is passed to the component's `Text` parameter.

`wwwroot/scripts.js`:

JavaScript

```
window.showQuote = async () => {
  let targetElement = document.getElementById('quoteContainer');
  await Blazor.rootComponents.add(targetElement, 'quote',
  {
    text: "Crow: I have my doubts that this movie is actually 'starring' " +
      "anybody. More like, 'camera is generally pointed at.'"
  });
}

const btn = document.querySelector("#showQuoteBtn");
btn.addEventListener("click", showQuote);
```

After the Blazor script is loaded, load the preceding scripts into the JS app:

HTML

```
<script src="_framework/{BLAZOR SCRIPT}"></script>
<script src="jsComponentInitializers.js"></script>
<script src="scripts.js"></script>
```

In the preceding example, the `{BLAZOR SCRIPT}` placeholder is the Blazor script.

In HTML, place the target container element (`quoteContainer`). For the demonstration in this section, a button triggers rendering the `Quote` component by calling the `showQuote` JS function:

HTML

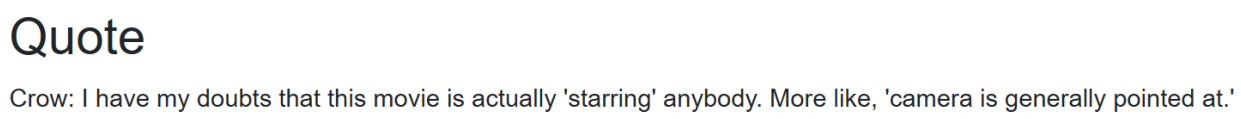
```
<button id="showQuoteBtn">Show Quote</button>

<div id="quoteContainer"></div>
```

On initialization before any components are rendered, the browser's developer tools console logs the `Quote` component's identifier (`name`) and parameters (`parameters`) when `initializeComponent` is called:

Console

```
Object { name: "quote", parameters: (1) […] }
  name: "quote"
  parameters: Array [ {…} ]
    0: Object { name: "Text", type: "string" }
    length: 1
```

When the **Show Quote** button is selected, the `Quote` component is rendered with the quote stored in `Text` displayed:



Quote ©1988-1999 Satellite of Love LLC: *Mystery Science Theater 3000* (Trace Beaulieu (Crow))

> **Note**
>
> `rootComponents.add` returns an instance of the component. Call `dispose` on the instance to release it:
>
> JavaScript
>
> ```
> const rootComponent = await window.Blazor.rootComponents.add(...);
>
> ...
>
> rootComponent.dispose();
> ```

The preceding example dynamically renders the root component when the `showQuote()` JS function is called. To render a root component into a container element when Blazor starts, use a JavaScript initializer to render the component, as the following example demonstrates.

The following example builds on the preceding example, using the `Quote` component, the root component registration in the `Program` file, and the initialization of `jsComponentInitializers.js`. The `showQuote()` function (and the `script.js` file) aren't used.

In HTML, place the target container element, `quoteContainer2` for this example:

HTML

```
<div id="quoteContainer2"></div>
```

Using a JavaScript initializer, add the root component to the target container element.

`wwwroot/{PACKAGE ID/ASSEMBLY NAME}.lib.module.js`:

JavaScript

```
export function afterStarted(blazor) {
  let targetElement = document.getElementById('quoteContainer2');
  blazor.rootComponents.add(targetElement, 'quote',
    {
      text: "Crow: I have my doubts that this movie is actually 'starring' "
+
          "anybody. More like, 'camera is generally pointed at.'"
    });
}
```

> **Note**
>
> For the call to `rootComponents.add`, use the `blazor` parameter (lowercase `b`) provided by the Blazor start event. Although the registration is valid when using the `Blazor` object (uppercase `B`), the preferred approach is to use the parameter.

For an advanced example with additional features, see the example in the `BasicTestApp` of the ASP.NET Core reference source (`dotnet/aspnetcore` GitHub repository):

- JavaScriptRootComponents.razor
- wwwroot/js/jsRootComponentInitializers.js
- wwwroot/index.html

> **Note**
>
> Documentation links to .NET reference source usually load the repository's default branch, which represents the current development for the next release of .NET. To select a tag for a specific release, use the **Switch branches or tags** dropdown list. For more information, see How to select a version tag of ASP.NET Core source code (dotnet/AspNetCore.Docs #26205).

Blazor custom elements

Use Blazor custom elements to dynamically render Razor components from other SPA frameworks, such as Angular or React.

Blazor custom elements:

- Use standard HTML interfaces to implement custom HTML elements.
- Eliminate the need to manually manage the state and lifecycle of root Razor components using JavaScript APIs.

- Are useful for gradually introducing Razor components into existing projects written in other SPA frameworks.

Custom elements don't support child content or templated components.

Element name

Per the HTML specification, custom element tag names must adopt kebab case:

❌ `mycounter`
❌ `MY-COUNTER`
❌ `MyCounter`
✔ `my-counter`
✔ `my-cool-counter`

Package

Add a package reference for Microsoft.AspNetCore.Components.CustomElements to the app's project file.

> **Note**
>
> For guidance on adding packages to .NET apps, see the articles under *Install and manage packages* at Package consumption workflow (NuGet documentation). Confirm correct package versions at NuGet.org.

Example component

The following examples are based on the `Counter` component from the Blazor project template.

`Counter.razor`:

razor

```razor
@page "/counter"

<PageTitle>Counter</PageTitle>

<h1>Counter</h1>

<p role="status">Current count: @currentCount</p>
```

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

@code {
    private int currentCount = 0;

    private void IncrementCount() => currentCount++;
}
```

Blazor Server registration

Take the following steps to register a root component as a custom element in a Blazor Server app.

Add the Microsoft.AspNetCore.Components.Web namespace to the top of the `Program` file:

C#

```
using Microsoft.AspNetCore.Components.Web;
```

Add a namespace for the app's components. In the following example, the app's namespace is `BlazorSample` and the components are located in the `Pages` folder:

C#

```
using BlazorSample.Pages;
```

Modify the call to AddServerSideBlazor. Specify the custom element with RegisterCustomElement on the RootComponents circuit option. The following example registers the `Counter` component with the custom HTML element `my-counter`:

C#

```
builder.Services.AddServerSideBlazor(options =>
{
    options.RootComponents.RegisterCustomElement<Counter>("my-counter");
});
```

Blazor WebAssembly registration

Take the following steps to register a root component as a custom element in a Blazor WebAssembly app.

Add the Microsoft.AspNetCore.Components.Web namespace to the top of the `Program` file:

```
C#

using Microsoft.AspNetCore.Components.Web;
```

Add a namespace for the app's components. In the following example, the app's namespace is `BlazorSample` and the components are located in the `Pages` folder:

```
C#

using BlazorSample.Pages;
```

Call RegisterCustomElement on RootComponents. The following example registers the `Counter` component with the custom HTML element `my-counter`:

```
C#

builder.RootComponents.RegisterCustomElement<Counter>("my-counter");
```

Use the registered custom element

Use the custom element with any web framework. For example, the preceding `my-counter` custom HTML element that renders the app's `Counter` component is used in a React app with the following markup:

```
HTML

<my-counter></my-counter>
```

For a complete example of how to create custom elements with Blazor, see the CustomElementsComponent component in the reference source.

> **Note**
>
> Documentation links to .NET reference source usually load the repository's default branch, which represents the current development for the next release of .NET. To select a tag for a specific release, use the **Switch branches or tags** dropdown list. For more information, see How to select a version tag of ASP.NET Core source code (dotnet/AspNetCore.Docs #26205).

Pass parameters

Pass parameters to your Razor component either as HTML attributes or as JavaScript properties on the DOM element.

The following `Counter` component uses an `IncrementAmount` parameter to set the increment amount of the **Click me** button.

`Counter.razor`:

razor

```razor
@page "/counter"

<h1>Counter</h1>

<p role="status">Current count: @currentCount</p>

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

@code {
    private int currentCount = 0;

    [Parameter]
    public int IncrementAmount { get; set; } = 1;

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

Render the `Counter` component with the custom element and pass a value to the `IncrementAmount` parameter as an HTML attribute. The attribute name adopts kebab-case syntax (`increment-amount`, not `IncrementAmount`):

HTML

```html
<my-counter increment-amount="10"></my-counter>
```

Alternatively, you can set the parameter's value as a JavaScript property on the element object. The property name adopts camel case syntax (`incrementAmount`, not `IncrementAmount`):

JavaScript

```
const elem = document.querySelector("my-counter");
elem.incrementAmount = 10;
```

You can update parameter values at any time using either attribute or property syntax.

Supported parameter types:

- Using JavaScript property syntax, you can pass objects of any JSON-serializable type.
- Using HTML attributes, you are limited to passing objects of string, boolean, or numerical types.

Generate Angular and React components

Generate framework-specific JavaScript (JS) components from Razor components for web frameworks, such as Angular or React. This capability isn't included with .NET, but is enabled by the support for rendering Razor components from JS. The JS component generation sample on GitHub demonstrates how to generate Angular and React components from Razor components. See the GitHub sample app's `README.md` file for additional information.

> ⚠ **Warning**
>
> The Angular and React component features are currently **experimental, unsupported, and subject to change or be removed at any time**. We welcome your feedback on how well this particular approach meets your requirements.

Render Razor components outside of ASP.NET Core

Article • 11/14/2024

> **ⓘ Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the .NET 9 version of this article.

Razor components can be rendered outside of the context of an HTTP request. You can render Razor components as HTML directly to a string or stream independently of the ASP.NET Core hosting environment. This is convenient for scenarios where you want to generate HTML fragments, such as for generating email content, generating static site content, or for building a content templating engine.

In the following example, a Razor component is rendered to an HTML string from a console app:

In a command shell, create a new console app project:

.NET CLI

```
dotnet new console -o ConsoleApp1
cd ConsoleApp1
```

In a command shell in the `ConsoleApp1` folder, add package references for Microsoft.AspNetCore.Components.Web and Microsoft.Extensions.Logging to the console app:

.NET CLI

```
dotnet add package Microsoft.AspNetCore.Components.Web
dotnet add package Microsoft.Extensions.Logging
```

In the console app's project file (`ConsoleApp1.csproj`), update the console app project to use the Razor SDK:

diff

```
- <Project Sdk="Microsoft.NET.Sdk">
+ <Project Sdk="Microsoft.NET.Sdk.Razor">
```

Add the following `RenderMessage` component to the project.

`RenderMessage.razor`:

razor

```razor
<h1>Render Message</h1>

<p>@Message</p>

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

Update the `Program` file:

- Set up dependency injection (IServiceCollection/BuildServiceProvider) and logging (AddLogging/ILoggerFactory).
- Create an HtmlRenderer and render the `RenderMessage` component by calling RenderComponentAsync.

Any calls to RenderComponentAsync must be made in the context of calling `InvokeAsync` on a component dispatcher. A component dispatcher is available from the HtmlRenderer.Dispatcher property.

C#

```
using Microsoft.AspNetCore.Components;
using Microsoft.AspNetCore.Components.Web;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;
using ConsoleApp1;

IServiceCollection services = new ServiceCollection();
services.AddLogging();

IServiceProvider serviceProvider = services.BuildServiceProvider();
ILoggerFactory loggerFactory =
serviceProvider.GetRequiredService<ILoggerFactory>();

await using var htmlRenderer = new HtmlRenderer(serviceProvider,
loggerFactory);

var html = await htmlRenderer.Dispatcher.InvokeAsync(async () =>
```

```
{
    var dictionary = new Dictionary<string, object?>
    {
        { "Message", "Hello from the Render Message component!" }
    };

    var parameters = ParameterView.FromDictionary(dictionary);
    var output = await htmlRenderer.RenderComponentAsync<RenderMessage>
(parameters);

    return output.ToHtmlString();
});

Console.WriteLine(html);
```

> **Note**
>
> Pass ParameterView.Empty to RenderComponentAsync when rendering the component without passing parameters.

Alternatively, you can write the HTML to a TextWriter by calling `output.WriteHtmlTo(textWriter)`.

The task returned by RenderComponentAsync completes when the component is fully rendered, including completing any asynchronous lifecycle methods. If you want to observe the rendered HTML earlier, call BeginRenderingComponent instead. Then, wait for the component rendering to complete by awaiting HtmlRootComponent.QuiescenceTask on the returned HtmlRootComponent instance.

ASP.NET Core built-in Razor components

Article • 11/11/2024

> **ⓘ Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the **.NET 9 version of this article**.

The following built-in Razor components are provided by the Blazor framework. For information on non-security-related project template components, see ASP.NET Core Blazor project structure. For information on security-related project template components, see the Security node articles.

- AntiforgeryToken
- AuthorizeView
- CascadingValue
- DataAnnotationsValidator
- DynamicComponent
- Editor<T>
- EditForm
- ErrorBoundary
- FocusOnNavigate
- HeadContent
- HeadOutlet
- ImportMap
- InputCheckbox
- InputDate
- InputFile
- InputNumber
- InputRadio
- InputRadioGroup
- InputSelect
- InputText
- InputTextArea
- LayoutComponentBase

- LayoutView
- NavigationLock
- NavLink
- PageTitle
- OwningComponentBase
- Paginator
- QuickGrid
- Router
- RouteView
- SectionContent
- SectionOutlet
- ValidationMessage
- ValidationSummary
- Virtualize

ASP.NET Core Blazor globalization and localization

Article • 10/14/2024

> **Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the **.NET 9 version of this article**.

This article explains how to render globalized and localized content to users in different cultures and languages.

Globalization and localization

For globalization, Blazor provides number and date formatting. For localization, Blazor renders content using the .NET Resources system.

A limited set of ASP.NET Core's localization features are supported:

✔ IStringLocalizer and IStringLocalizer<T> are supported in Blazor apps.

✘ IHtmlLocalizer, IViewLocalizer, and Data Annotations localization are ASP.NET Core MVC features and *not supported* in Blazor apps.

This article describes how to use Blazor's globalization and localization features based on:

- The Accept-Language header, which is set by the browser based on a user's language preferences in browser settings.
- A culture set by the app not based on the value of the Accept-Language header. The setting can be static for all users or dynamic based on app logic. When the setting is based on the user's preference, the setting is usually saved for reload on future visits.

For additional general information, see the following resources:

- Globalization and localization in ASP.NET Core
- .NET Fundamentals: Globalization

- .NET Fundamentals: Localization

Often, the terms *language* and *culture* are used interchangeably when dealing with globalization and localization concepts.

In this article, *language* refers to selections made by a user in their browser's settings. The user's language selections are submitted in browser requests in the Accept-Language header. Browser settings usually use the word "language" in the UI.

Culture pertains to members of .NET and Blazor API. For example, a user's request can include the Accept-Language header specifying a *language* from the user's perspective, but the app ultimately sets the CurrentCulture ("culture") property from the language that the user requested. API usually uses the word "culture" in its member names.

The guidance in this article doesn't cover setting the page's HTML language attribute (<html lang="...">), which accessiblity tools use. You can set the value statically by assigning a language to the `lang` attribute of the `<html>` tag or to `document.documentElement.lang` in JavaScript. You can dynamically set the value of `document.documentElement.lang` with JS interop.

> **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 article's examples.

Globalization

The @bind attribute directive applies formats and parses values for display based on the user's first preferred language that the app supports. @bind supports the @bind:culture parameter to provide a System.Globalization.CultureInfo for parsing and formatting a value.

The current culture can be accessed from the System.Globalization.CultureInfo.CurrentCulture property.

CultureInfo.InvariantCulture is used for the following field types (`<input type="{TYPE}" />`, where the `{TYPE}` placeholder is the type):

- `date`

- `number`

The preceding field types:

- Are displayed using their appropriate browser-based formatting rules.
- Can't contain free-form text.
- Provide user interaction characteristics based on the browser's implementation.

Blazor provides built-in support to render values in the current culture. Therefore, specifying a culture with @bind:culture isn't recommended when using the `date` and `number` field types.

The following field types have specific formatting requirements and aren't supported by all of the major browsers, so they aren't supported by Blazor:

- `datetime-local`
- `month`
- `week`

For current browser support of the preceding types, see Can I use.

.NET globalization and International Components for Unicode (ICU) support (Blazor WebAssembly)

Blazor WebAssembly uses a reduced globalization API and set of built-in International Components for Unicode (ICU) locales. For more information, see .NET globalization and ICU: ICU on WebAssembly.

To load a custom ICU data file to control the app's locales, see WASM Globalization Icu. Currently, manually building the custom ICU data file is required. .NET tooling to ease the process of creating the file is planned for .NET 10 in November, 2025.

Invariant globalization

This section only applies to client-side Blazor scenarios.

If the app doesn't require localization, configure the app to support the invariant culture, which is generally based on United States English (`en-US`). Using invariant globalization reduces the app's download size and results in faster app startup. Set the `InvariantGlobalization` property to `true` in the app's project file (`.csproj`):

XML

```xml
<PropertyGroup>
  <InvariantGlobalization>true</InvariantGlobalization>
</PropertyGroup>
```

Alternatively, configure invariant globalization with the following approaches:

- In `runtimeconfig.json`:

  JSON

  ```json
  {
    "runtimeOptions": {
      "configProperties": {
        "System.Globalization.Invariant": true
      }
    }
  }
  ```

- With an environment variable:
  - Key: `DOTNET_SYSTEM_GLOBALIZATION_INVARIANT`
  - Value: `true` or `1`

For more information, see Runtime configuration options for globalization (.NET documentation).

Timezone information

This section only applies to client-side Blazor scenarios.

Adopting invariant globalization only results in using non-localized timezone names. To trim timezone code and data, which reduces the app's download size and results in faster app startup, apply the `<InvariantTimezone>` MSBuild property with a value of `true` in the app's project file:

XML

```xml
<PropertyGroup>
  <InvariantTimezone>true</InvariantTimezone>
</PropertyGroup>
```

> ⓘ Note

<BlazorEnableTimeZoneSupport> overrides an earlier `<InvariantTimezone>` setting. We recommend removing the `<BlazorEnableTimeZoneSupport>` setting.

Demonstration component

The following `CultureExample1` component can be used to demonstrate Blazor globalization and localization concepts covered by this article.

`CultureExample1.razor`:

razor

```
@page "/culture-example-1"
@using System.Globalization

<h1>Culture Example 1</h1>

<ul>
    <li><b>CurrentCulture</b>: @CultureInfo.CurrentCulture</li>
    <li><b>CurrentUICulture</b>: @CultureInfo.CurrentUICulture</li>
</ul>

<h2>Rendered values</h2>

<ul>
    <li><b>Date</b>: @dt</li>
    <li><b>Number</b>: @number.ToString("N2")</li>
</ul>

<h2><code>&lt;input&gt;</code> elements that don't set a <code>type</code>
</h2>

<p>
    The following <code>&lt;input&gt;</code> elements use
    <code>CultureInfo.CurrentCulture</code>.
</p>

<ul>
    <li><label><b>Date:</b> <input @bind="dt" /></label></li>
    <li><label><b>Number:</b> <input @bind="number" /></label></li>
</ul>

<h2><code>&lt;input&gt;</code> elements that set a <code>type</code></h2>

<p>
    The following <code>&lt;input&gt;</code> elements use
    <code>CultureInfo.InvariantCulture</code>.
</p>

<ul>
```

```
    <li><label><b>Date:</b> <input type="date" @bind="dt" /></label></li>
    <li><label><b>Number:</b> <input type="number" @bind="number" /></label>
</li>
</ul>

@code {
    private DateTime dt = DateTime.Now;
    private double number = 1999.69;
}
```

The number string format (`N2`) in the preceding example (`.ToString("N2")`) is a standard .NET numeric format specifier. The `N2` format is supported for all numeric types, includes a group separator, and renders up to two decimal places.

Optionally, add a menu item to the navigation in the `NavMenu` component (`NavMenu.razor`) for the `CultureExample1` component.

Dynamically set the culture from the `Accept-Language` header

Add the Microsoft.Extensions.Localization package to the app.

The Accept-Language header is set by the browser and controlled by the user's language preferences in browser settings. In browser settings, a user sets one or more preferred languages in order of preference. The order of preference is used by the browser to set quality values (`q`, 0-1) for each language in the header. The following example specifies United States English, English, and Costa Rican Spanish with a preference for United States English or English:

Accept-Language: en-US,en;q=0.9,es-CR;q=0.8

The app's culture is set by matching the first requested language that matches a supported culture of the app.

In ***client-side development***, set the `BlazorWebAssemblyLoadAllGlobalizationData` property to `true` in the client-side app's project file (`.csproj`):

XML

```
<PropertyGroup>

<BlazorWebAssemblyLoadAllGlobalizationData>true</BlazorWebAssemblyLoadAllGlo
balizationData>
</PropertyGroup>
```

> **Note**
>
> If the app's specification requires limiting the supported cultures to an explicit list, see the Dynamically set the client-side culture by user preference section of this article.

Apps are localized using Localization Middleware. Add localization services to the app with AddLocalization.

Add the following line to the `Program` file where services are registered:

```
C#

builder.Services.AddLocalization();
```

In ***server-side development***, specify the app's supported cultures before any middleware that might check the request culture. Generally, place Request Localization Middleware immediately before calling MapRazorComponents. The following example configures supported cultures for United States English and Costa Rican Spanish:

```
C#

app.UseRequestLocalization(new RequestLocalizationOptions()
    .AddSupportedCultures(new[] { "en-US", "es-CR" })
    .AddSupportedUICultures(new[] { "en-US", "es-CR" }));
```

For information on ordering the Localization Middleware in the middleware pipeline of the `Program` file, see ASP.NET Core Middleware.

Use the `CultureExample1` component shown in the Demonstration component section to study how globalization works. Issue a request with United States English (`en-US`). Switch to Costa Rican Spanish (`es-CR`) in the browser's language settings. Request the webpage again.

> **Note**
>
> Some browsers force you to use the default language setting for both requests and the browser's own UI settings. This can make changing the language back to one that you understand difficult because all of the setting UI screens might end up in a language that you can't read. A browser such as Opera is a good choice for

testing because it permits you to set a default language for webpage requests but leave the browser's settings UI in your language.

When the culture is United States English (`en-US`), the rendered component uses month/day date formatting (`6/7`), 12-hour time (`AM`/`PM`), and comma separators in numbers with a dot for the decimal value (`1,999.69`):

- **Date**: 6/7/2021 6:45:22 AM
- **Number**: 1,999.69

When the culture is Costa Rican Spanish (`es-CR`), the rendered component uses day/month date formatting (`7/6`), 24-hour time, and period separators in numbers with a comma for the decimal value (`1.999,69`):

- **Date**: 7/6/2021 6:49:38
- **Number**: 1.999,69

Statically set the client-side culture

Set the `BlazorWebAssemblyLoadAllGlobalizationData` property to `true` in the app's project file (`.csproj`):

XML

```
<PropertyGroup>

<BlazorWebAssemblyLoadAllGlobalizationData>true</BlazorWebAssemblyLoadAllGlo
balizationData>
</PropertyGroup>
```

The app's culture can be set in JavaScript when Blazor starts with the `applicationCulture` Blazor start option. The following example configures the app to launch using the United States English (`en-US`) culture.

Prevent Blazor autostart by adding `autostart="false"` to Blazor's <script> tag:

HTML

```
<script src="{BLAZOR SCRIPT}" autostart="false"></script>
```

In the preceding example, the `{BLAZOR SCRIPT}` placeholder is the Blazor script path and file name. For the location of the script, see ASP.NET Core Blazor project structure.

Add the following `<script>` block after Blazor's <script> tag and before the closing `</body>` tag:

Blazor Web App:

HTML

```
<script>
  Blazor.start({
    webAssembly: {
      applicationCulture: 'en-US'
    }
  });
</script>
```

Standalone Blazor WebAssembly:

HTML

```
<script>
  Blazor.start({
    applicationCulture: 'en-US'
  });
</script>
```

The value for `applicationCulture` must conform to the BCP-47 language tag format. For more information on Blazor startup, see ASP.NET Core Blazor startup.

An alternative to setting the culture Blazor's start option is to set the culture in C# code. Set CultureInfo.DefaultThreadCurrentCulture and CultureInfo.DefaultThreadCurrentUICulture in the `Program` file to the same culture.

Add the System.Globalization namespace to the `Program` file:

C#

```
using System.Globalization;
```

Add the culture settings before the line that builds and runs the WebAssemblyHostBuilder (`await builder.Build().RunAsync();`):

C#

```
CultureInfo.DefaultThreadCurrentCulture = new CultureInfo("en-US");
CultureInfo.DefaultThreadCurrentUICulture = new CultureInfo("en-US");
```

> **Note**
>
> Currently, Blazor WebAssembly apps only load resources based on DefaultThreadCurrentCulture. For more information, see Blazor WASM only relies on the current culture (current UI culture isn't respected) (dotnet/aspnetcore #56824).

Use the `CultureExample1` component shown in the Demonstration component section to study how globalization works. Issue a request with United States English (`en-US`). Switch to Costa Rican Spanish (`es-CR`) in the browser's language settings. Request the webpage again. When the requested language is Costa Rican Spanish, the app's culture remains United States English (`en-US`).

Statically set the server-side culture

Server-side apps are localized using Localization Middleware. Add localization services to the app with AddLocalization.

In the `Program` file:

```
C#

builder.Services.AddLocalization();
```

Specify the static culture in the `Program` file before any middleware that might check the request culture. Generally, place Request Localization Middleware immediately before MapRazorComponents. The following example configures United States English:

```
C#

app.UseRequestLocalization("en-US");
```

The culture value for UseRequestLocalization must conform to the BCP-47 language tag format.

For information on ordering the Localization Middleware in the middleware pipeline of the `Program` file, see ASP.NET Core Middleware.

Use the `CultureExample1` component shown in the Demonstration component section to study how globalization works. Issue a request with United States English (`en-US`). Switch to Costa Rican Spanish (`es-CR`) in the browser's language settings. Request the

webpage again. When the requested language is Costa Rican Spanish, the app's culture remains United States English (`en-US`).

Dynamically set the client-side culture by user preference

Examples of locations where an app might store a user's preference include in browser local storage (common for client-side scenarios), in a localization cookie or database (common for server-side scenarios), or in an external service attached to an external database and accessed by a web API. The following example demonstrates how to use browser local storage.

Add the Microsoft.Extensions.Localization package to the app.

> **Note**
>
> For guidance on adding packages to .NET apps, see the articles under *Install and manage packages* at Package consumption workflow (NuGet documentation). Confirm correct package versions at NuGet.org.

Set the `BlazorWebAssemblyLoadAllGlobalizationData` property to `true` in the project file:

XML

```
<PropertyGroup>

<BlazorWebAssemblyLoadAllGlobalizationData>true</BlazorWebAssemblyLoadAllGlobalizationData>
</PropertyGroup>
```

The app's culture for client-side rendering is set using the Blazor framework's API. A user's culture selection can be persisted in browser local storage.

Provide JS functions after Blazor's <script> tag to get and set the user's culture selection with browser local storage:

HTML

```
<script>
  window.blazorCulture = {
    get: () => window.localStorage['BlazorCulture'],
    set: (value) => window.localStorage['BlazorCulture'] = value
```

```
  };
</script>
```

> **Note**
>
> The preceding example pollutes the client with global functions. For a better approach in production apps, see JavaScript isolation in JavaScript modules.

Add the namespaces for System.Globalization and Microsoft.JSInterop to the top of the `Program` file:

C#

```
using System.Globalization;
using Microsoft.JSInterop;
```

Remove the following line:

diff

```
- await builder.Build().RunAsync();
```

Replace the preceding line with the following code. The code adds Blazor's localization service to the app's service collection with AddLocalization and uses JS interop to call into JS and retrieve the user's culture selection from local storage. If local storage doesn't contain a culture for the user, the code sets a default value of United States English (`en-US`).

C#

```
builder.Services.AddLocalization();

var host = builder.Build();

const string defaultCulture = "en-US";

var js = host.Services.GetRequiredService<IJSRuntime>();
var result = await js.InvokeAsync<string>("blazorCulture.get");
var culture = CultureInfo.GetCultureInfo(result ?? defaultCulture);

if (result == null)
{
    await js.InvokeVoidAsync("blazorCulture.set", defaultCulture);
}

CultureInfo.DefaultThreadCurrentCulture = culture;
```

```
CultureInfo.DefaultThreadCurrentUICulture = culture;

await host.RunAsync();
```

> ⓘ Note
>
> Currently, Blazor WebAssembly apps only load resources based on DefaultThreadCurrentCulture. For more information, see Blazor WASM only relies on the current culture (current UI culture isn't respected) (dotnet/aspnetcore #56824).

The following `CultureSelector` component shows how to perform the following actions:

- Set the user's culture selection into browser local storage via JS interop.
- Reload the component that they requested (`forceLoad: true`), which uses the updated culture.

`CultureSelector.razor`:

razor

```
@using System.Globalization
@inject IJSRuntime JS
@inject NavigationManager Navigation

<p>
    <label>
        Select your locale:
        <select @bind="selectedCulture"
@bind:after="ApplySelectedCultureAsync">
            @foreach (var culture in supportedCultures)
            {
                <option value="@culture">@culture.DisplayName</option>
            }
        </select>
    </label>
</p>

@code
{
    private CultureInfo[] supportedCultures = new[]
    {
        new CultureInfo("en-US"),
        new CultureInfo("es-CR"),
    };

    private CultureInfo? selectedCulture;

    protected override void OnInitialized()
```

```
    {
        selectedCulture = CultureInfo.CurrentCulture;
    }

    private async Task ApplySelectedCultureAsync()
    {
        if (CultureInfo.CurrentCulture != selectedCulture)
        {
            await JS.InvokeVoidAsync("blazorCulture.set",
selectedCulture!.Name);

            Navigation.NavigateTo(Navigation.Uri, forceLoad: true);
        }
    }
}
```

> ⓘ Note
>
> For more information on IJSInProcessRuntime, see Call JavaScript functions from .NET methods in ASP.NET Core Blazor.

Inside the closing tag of the `</main>` element in the `MainLayout` component (`MainLayout.razor`), add the `CultureSelector` component:

razor

```
<article class="bottom-row px-4">
    <CultureSelector />
</article>
```

Use the `CultureExample1` component shown in the Demonstration component section to study how the preceding example works.

Dynamically set the server-side culture by user preference

Examples of locations where an app might store a user's preference include in browser local storage (common for client-side scenarios), in a localization cookie or database (common for server-side scenarios), or in an external service attached to an external database and accessed by a web API. The following example demonstrates how to use a localization cookie.

> ⓘ Note

The following example assumes that the app adopts ***global*** interactivity by specifying the interactive server-side rendering (interactive SSR) on the `Routes` component in the `App` component (`Components/App.razor`):

razor

```
<Routes @rendermode="InteractiveServer" />
```

If the app adopts ***per-page/component*** interactivity, see the remarks at the end of this section to modify the render modes of the example's components.

Add the Microsoft.Extensions.Localization package to the app.

> **Note**
>
> For guidance on adding packages to .NET apps, see the articles under *Install and manage packages* at Package consumption workflow (NuGet documentation). Confirm correct package versions at NuGet.org.

Server-side apps are localized using Localization Middleware. Add localization services to the app with AddLocalization.

In the `Program` file:

C#

```
builder.Services.AddLocalization();
```

Set the app's default and supported cultures with RequestLocalizationOptions.

Before the call to MapRazorComponents in the request processing pipeline, place the following code:

C#

```
var supportedCultures = new[] { "en-US", "es-CR" };
var localizationOptions = new RequestLocalizationOptions()
    .SetDefaultCulture(supportedCultures[0])
    .AddSupportedCultures(supportedCultures)
    .AddSupportedUICultures(supportedCultures);

app.UseRequestLocalization(localizationOptions);
```

For information on ordering the Localization Middleware in the middleware pipeline, see ASP.NET Core Middleware.

The following example shows how to set the current culture in a cookie that can be read by the Localization Middleware.

The following namespaces are required for the `App` component:

- System.Globalization
- Microsoft.AspNetCore.Localization

Add the following to the top of the `App` component file (`Components/App.razor`):

razor

```
@using System.Globalization
@using Microsoft.AspNetCore.Localization
```

Add the following `@code` block to the bottom of the `App` component file:

razor

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

    protected override void OnInitialized()
    {
        HttpContext?.Response.Cookies.Append(
            CookieRequestCultureProvider.DefaultCookieName,
            CookieRequestCultureProvider.MakeCookieValue(
                new RequestCulture(
                    CultureInfo.CurrentCulture,
                    CultureInfo.CurrentUICulture)));
    }
}
```

For information on ordering the Localization Middleware in the middleware pipeline, see ASP.NET Core Middleware.

If the app isn't configured to process controller actions:

- Add MVC services by calling AddControllers on the service collection in the `Program` file:

  C#

```
builder.Services.AddControllers();
```

- Add controller endpoint routing in the `Program` file by calling MapControllers on the IEndpointRouteBuilder (`app`):

```
C#

app.MapControllers();
```

To provide UI to allow a user to select a culture, use a *redirect-based approach* with a localization cookie. The app persists the user's selected culture via a redirect to a controller. The controller sets the user's selected culture into a cookie and redirects the user back to the original URI. The process is similar to what happens in a web app when a user attempts to access a secure resource, where the user is redirected to a sign-in page and then redirected back to the original resource.

`Controllers/CultureController.cs`:

```
C#

using Microsoft.AspNetCore.Localization;
using Microsoft.AspNetCore.Mvc;

[Route("[controller]/[action]")]
public class CultureController : Controller
{
    public IActionResult Set(string culture, string redirectUri)
    {
        if (culture != null)
        {
            HttpContext.Response.Cookies.Append(
                CookieRequestCultureProvider.DefaultCookieName,
                CookieRequestCultureProvider.MakeCookieValue(
                    new RequestCulture(culture, culture)));
        }

        return LocalRedirect(redirectUri);
    }
}
```

> ⚠ **Warning**

Use the LocalRedirect action result, as shown in the preceding example, to prevent open redirect attacks. For more information, see Prevent open redirect attacks in ASP.NET Core.

The following `CultureSelector` component shows how to call the `Set` method of the `CultureController` with the new culture. The component is placed in the `Shared` folder for use throughout the app.

`CultureSelector.razor`:

razor

```
@using System.Globalization
@inject IJSRuntime JS
@inject NavigationManager Navigation

<p>
    <label>
        Select your locale:
        <select @bind="selectedCulture"
@bind:after="ApplySelectedCultureAsync">
            @foreach (var culture in supportedCultures)
            {
                <option value="@culture">@culture.DisplayName</option>
            }
        </select>
    </label>
</p>

@code
{
    private CultureInfo[] supportedCultures = new[]
    {
        new CultureInfo("en-US"),
        new CultureInfo("es-CR"),
    };

    private CultureInfo? selectedCulture;

    protected override void OnInitialized()
    {
        selectedCulture = CultureInfo.CurrentCulture;
    }

    private async Task ApplySelectedCultureAsync()
    {
        if (CultureInfo.CurrentCulture != selectedCulture)
        {
            var uri = new Uri(Navigation.Uri)
                .GetComponents(UriComponents.PathAndQuery,
UriFormat.Unescaped);
```

```
            var cultureEscaped = Uri.EscapeDataString(selectedCulture.Name);
            var uriEscaped = Uri.EscapeDataString(uri);

            Navigation.NavigateTo(
                $"Culture/Set?culture={cultureEscaped}&redirectUri=
{uriEscaped}",
                forceLoad: true);
        }
    }
}
```

Add the `CultureSelector` component to the `MainLayout` component. Place the following markup inside the closing `</main>` tag in the `Components/Layout/MainLayout.razor` file:

razor

```
<article class="bottom-row px-4">
    <CultureSelector />
</article>
```

Use the `CultureExample1` component shown in the Demonstration component section to study how the preceding example works.

The preceding example assumes that the app adopts ***global*** interactivity by specifying the Interactive Server render mode on the `Routes` component in the `App` component (`Components/App.razor`):

razor

```
<Routes @rendermode="InteractiveServer" />
```

If the app adopts ***per-page/component*** interactivity, make the following changes:

- Add the Interactive Server render mode to the top of the `CultureExample1` component file (`Components/Pages/CultureExample1.razor`):

  razor

  ```
  @rendermode InteractiveServer
  ```

- In the app's main layout (`Components/Layout/MainLayout.razor`), apply the Interactive Server render mode to the `CultureSelector` component:

  razor

```
<CultureSelector @rendermode="InteractiveServer" />
```

Dynamically set the culture in a Blazor Web App by user preference

This section applies to Blazor Web Apps that adopt Auto (Server and WebAssembly) interactivity.

Examples of locations where an app might store a user's preference include in browser local storage (common for client-side scenarios), in a localization cookie or database (common for server-side scenarios), both local storage and a localization cookie (Blazor Web Apps with server and WebAssembly components), or in an external service attached to an external database and accessed by a web API. The following example demonstrates how to use browser local storage for client-side rendered (CSR) components and a localization cookie for server-side rendered (SSR) components.

Updates to the `.Client` project

Add the Microsoft.Extensions.Localization package to the `.Client` project.

> **Note**
>
> For guidance on adding packages to .NET apps, see the articles under *Install and manage packages* at Package consumption workflow (NuGet documentation). Confirm correct package versions at NuGet.org.

Set the `BlazorWebAssemblyLoadAllGlobalizationData` property to `true` in the `.Client` project file:

XML

```
<PropertyGroup>

<BlazorWebAssemblyLoadAllGlobalizationData>true</BlazorWebAssemblyLoadAllGlobalizationData>
</PropertyGroup>
```

Add the namespaces for System.Globalization and Microsoft.JSInterop to the top of the `.Client` project's `Program` file:

C#

```
using System.Globalization;
using Microsoft.JSInterop;
```

Remove the following line:

diff

```
- await builder.Build().RunAsync();
```

Replace the preceding line with the following code. The code adds Blazor's localization service to the app's service collection with AddLocalization and uses JS interop to call into JS and retrieve the user's culture selection from local storage. If local storage doesn't contain a culture for the user, the code sets a default value of United States English (`en-US`).

C#

```
builder.Services.AddLocalization();

var host = builder.Build();

const string defaultCulture = "en-US";

var js = host.Services.GetRequiredService<IJSRuntime>();
var result = await js.InvokeAsync<string>("blazorCulture.get");
var culture = CultureInfo.GetCultureInfo(result ?? defaultCulture);

if (result == null)
{
    await js.InvokeVoidAsync("blazorCulture.set", defaultCulture);
}

CultureInfo.DefaultThreadCurrentCulture = culture;
CultureInfo.DefaultThreadCurrentUICulture = culture;

await host.RunAsync();
```

> ⓘ **Note**
>
> Currently, Blazor WebAssembly apps only load resources based on DefaultThreadCurrentCulture. For more information, see Blazor WASM only relies on the current culture (current UI culture isn't respected) (dotnet/aspnetcore #56824).

Add the following `CultureSelector` component to the `.Client` project.

The component adopts the following approaches to work for either SSR or CSR components:

- The display name of each available culture in the dropdown list is provided by a dictionary because client-side globalization data include localized text of culture display names that server-side globalization data provides. For example, server-side localization displays `English (United States)` when `en-US` is the culture and `Ingles ()` when a different culture is used. Because localization of the culture display names isn't available with Blazor WebAssembly globalization, the display name for United States English on the client for any loaded culture is just `en-US`. Using a custom dictionary permits the component to at least display full English culture names.
- When user changes the culture, JS interop sets the culture in local browser storage and a controller action updates the localization cookie with the culture. The controller is added to the app later in the Server project updates section.

`Pages/CultureSelector.razor`:

razor

```
@using System.Globalization
@inject IJSRuntime JS
@inject NavigationManager Navigation

<p>
    <label>
        Select your locale:
        <select @bind="@selectedCulture"
@bind:after="ApplySelectedCultureAsync">
            @foreach (var culture in supportedCultures)
            {
                <option value="@culture">@cultureDict[culture.Name]</option>
            }
        </select>
    </label>
</p>

@code
{
    private Dictionary<string, string> cultureDict =
        new()
        {
            { "en-US", "English (United States)" },
            { "es-CR", "Spanish (Costa Rica)" }
        };

    private CultureInfo[] supportedCultures = new[]
```

```
    {
        new CultureInfo("en-US"),
        new CultureInfo("es-CR"),
    };

    private CultureInfo? selectedCulture;

    protected override void OnInitialized()
    {
        selectedCulture = CultureInfo.CurrentCulture;
    }

    private async Task ApplySelectedCultureAsync()
    {
        if (CultureInfo.CurrentCulture != selectedCulture)
        {
            await JS.InvokeVoidAsync("blazorCulture.set",
selectedCulture!.Name);

            var uri = new Uri(Navigation.Uri)
                .GetComponents(UriComponents.PathAndQuery,
UriFormat.Unescaped);
            var cultureEscaped = Uri.EscapeDataString(selectedCulture.Name);
            var uriEscaped = Uri.EscapeDataString(uri);

            Navigation.NavigateTo(
                $"Culture/Set?culture={cultureEscaped}&redirectUri=
{uriEscaped}",
                forceLoad: true);
        }
    }
}
```

> **Note**
>
> For more information on IJSInProcessRuntime, see Call JavaScript functions from .NET methods in ASP.NET Core Blazor.

In the `.Client` project, place the following `CultureClient` component to study how globalization works for CSR components.

`Pages/CultureClient.razor`:

razor

```
@page "/culture-client"
@rendermode InteractiveWebAssembly
@using System.Globalization

<PageTitle>Culture Client</PageTitle>
```

```
<h1>Culture Client</h1>

<ul>
    <li><b>CurrentCulture</b>: @CultureInfo.CurrentCulture</li>
    <li><b>CurrentUICulture</b>: @CultureInfo.CurrentUICulture</li>
</ul>

<h2>Rendered values</h2>

<ul>
    <li><b>Date</b>: @dt</li>
    <li><b>Number</b>: @number.ToString("N2")</li>
</ul>

<h2><code>&lt;input&gt;</code> elements that don't set a <code>type</code>
</h2>

<p>
    The following <code>&lt;input&gt;</code> elements use
    <code>CultureInfo.CurrentCulture</code>.
</p>

<ul>
    <li><label><b>Date:</b> <input @bind="dt" /></label></li>
    <li><label><b>Number:</b> <input @bind="number" /></label></li>
</ul>

<h2><code>&lt;input&gt;</code> elements that set a <code>type</code></h2>

<p>
    The following <code>&lt;input&gt;</code> elements use
    <code>CultureInfo.InvariantCulture</code>.
</p>

<ul>
    <li><label><b>Date:</b> <input type="date" @bind="dt" /></label></li>
    <li><label><b>Number:</b> <input type="number" @bind="number" /></label>
</li>
</ul>

@code {
    private DateTime dt = DateTime.Now;
    private double number = 1999.69;
}
```

Server project updates

Add the Microsoft.Extensions.Localization package to the server project.

> **Note**

For guidance on adding packages to .NET apps, see the articles under *Install and manage packages* at Package consumption workflow (NuGet documentation). Confirm correct package versions at NuGet.org.

Server-side apps are localized using Localization Middleware. Add localization services to the app with AddLocalization.

In the server project's `Program` file where services are registered:

C#

```
builder.Services.AddLocalization();
```

Set the app's default and supported cultures with RequestLocalizationOptions.

Before the call to MapRazorComponents in the request processing pipeline, place the following code:

C#

```
var supportedCultures = new[] { "en-US", "es-CR" };
var localizationOptions = new RequestLocalizationOptions()
    .SetDefaultCulture(supportedCultures[0])
    .AddSupportedCultures(supportedCultures)
    .AddSupportedUICultures(supportedCultures);

app.UseRequestLocalization(localizationOptions);
```

The following example shows how to set the current culture in a cookie that can be read by the Localization Middleware.

The following namespaces are required for the `App` component:

- System.Globalization
- Microsoft.AspNetCore.Localization

Add the following to the top of the `App` component file (`Components/App.razor`):

razor

```
@using System.Globalization
@using Microsoft.AspNetCore.Localization
```

The app's culture for client-side rendering is set using the Blazor framework's API. A user's culture selection can be persisted in browser local storage for CSR components.

After the Blazor's <script> tag, provide JS functions to get and set the user's culture selection with browser local storage:

HTML

```
<script>
  window.blazorCulture = {
    get: () => window.localStorage['BlazorCulture'],
    set: (value) => window.localStorage['BlazorCulture'] = value
  };
</script>
```

> **Note**
>
> The preceding example pollutes the client with global functions. For a better approach in production apps, see JavaScript isolation in JavaScript modules.

Add the following `@code` block to the bottom of the `App` component file:

razor

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

    protected override void OnInitialized()
    {
        HttpContext?.Response.Cookies.Append(
            CookieRequestCultureProvider.DefaultCookieName,
            CookieRequestCultureProvider.MakeCookieValue(
                new RequestCulture(
                    CultureInfo.CurrentCulture,
                    CultureInfo.CurrentUICulture)));
    }
}
```

If the server project isn't configured to process controller actions:

- Add MVC services by calling AddControllers on the service collection in the `Program` file:

  C#

  ```
  builder.Services.AddControllers();
  ```

- Add controller endpoint routing in the `Program` file by calling MapControllers on the IEndpointRouteBuilder (`app`):

  C#

  ```
  app.MapControllers();
  ```

To allow a user to select a culture for SSR components, use a *redirect-based approach* with a localization cookie. The app persists the user's selected culture via a redirect to a controller. The controller sets the user's selected culture into a cookie and redirects the user back to the original URI. The process is similar to what happens in a web app when a user attempts to access a secure resource, where the user is redirected to a sign-in page and then redirected back to the original resource.

`Controllers/CultureController.cs`:

C#

```
using Microsoft.AspNetCore.Localization;
using Microsoft.AspNetCore.Mvc;

[Route("[controller]/[action]")]
public class CultureController : Controller
{
    public IActionResult Set(string culture, string redirectUri)
    {
        if (culture != null)
        {
            HttpContext.Response.Cookies.Append(
                CookieRequestCultureProvider.DefaultCookieName,
                CookieRequestCultureProvider.MakeCookieValue(
                    new RequestCulture(culture, culture)));
        }

        return LocalRedirect(redirectUri);
    }
}
```

> ⚠ **Warning**
>
> Use the LocalRedirect action result, as shown in the preceding example, to prevent open redirect attacks. For more information, see Prevent open redirect attacks in ASP.NET Core.

Add the `CultureSelector` component to the `MainLayout` component. Place the following markup inside the closing `</main>` tag in the `Components/Layout/MainLayout.razor` file:

razor

```razor
<article class="bottom-row px-4">
    <CultureSelector @rendermode="InteractiveAuto" />
</article>
```

Use the `CultureExample1` component shown in the Demonstration component section to study how the preceding example works.

In the server project, place the following `CultureServer` component to study how globalization works for SSR components.

`Components/Pages/CultureServer.razor`:

razor

```razor
@page "/culture-server"
@rendermode InteractiveServer
@using System.Globalization

<PageTitle>Culture Server</PageTitle>

<h1>Culture Server</h1>

<ul>
    <li><b>CurrentCulture</b>: @CultureInfo.CurrentCulture</li>
    <li><b>CurrentUICulture</b>: @CultureInfo.CurrentUICulture</li>
</ul>

<h2>Rendered values</h2>

<ul>
    <li><b>Date</b>: @dt</li>
    <li><b>Number</b>: @number.ToString("N2")</li>
</ul>

<h2><code>&lt;input&gt;</code> elements that don't set a <code>type</code>
</h2>

<p>
    The following <code>&lt;input&gt;</code> elements use
    <code>CultureInfo.CurrentCulture</code>.
</p>

<ul>
    <li><label><b>Date:</b> <input @bind="dt" /></label></li>
    <li><label><b>Number:</b> <input @bind="number" /></label></li>
</ul>
```

```
<h2><code>&lt;input&gt;</code> elements that set a <code>type</code></h2>

<p>
    The following <code>&lt;input&gt;</code> elements use
    <code>CultureInfo.InvariantCulture</code>.
</p>

<ul>
    <li><label><b>Date:</b> <input type="date" @bind="dt" /></label></li>
    <li><label><b>Number:</b> <input type="number" @bind="number" /></label>
</li>
</ul>

@code {
    private DateTime dt = DateTime.Now;
    private double number = 1999.69;
}
```

Add both the `CultureClient` and `CultureServer` components to the sidebar navigation in `Components/Layout/NavMenu.razor`:

razor

```
<div class="nav-item px-3">
    <NavLink class="nav-link" href="culture-server">
        <span class="bi bi-list-nested-nav-menu" aria-hidden="true"></span>
Culture (Server)
    </NavLink>
</div>
<div class="nav-item px-3">
    <NavLink class="nav-link" href="culture-client">
        <span class="bi bi-list-nested-nav-menu" aria-hidden="true"></span>
Culture (Client)
    </NavLink>
</div>
```

Interactive Auto components

The guidance in this section also works for components that adopt the Interactive Auto render mode:

razor

```
@rendermode InteractiveAuto
```

Localization

If the app doesn't already support dynamic culture selection, add the Microsoft.Extensions.Localization package to the app.

> **Note**
>
> For guidance on adding packages to .NET apps, see the articles under *Install and manage packages* at Package consumption workflow (NuGet documentation). Confirm correct package versions at NuGet.org.

Client-side localization

Set the `BlazorWebAssemblyLoadAllGlobalizationData` property to `true` in the app's project file (`.csproj`):

```xml
<PropertyGroup>

<BlazorWebAssemblyLoadAllGlobalizationData>true</BlazorWebAssemblyLoadAllGlobalizationData>
</PropertyGroup>
```

In the `Program` file, add namespace the namespace for System.Globalization to the top of the file:

```csharp
using System.Globalization;
```

Add Blazor's localization service to the app's service collection with AddLocalization:

```csharp
builder.Services.AddLocalization();
```

Server-side localization

Use Localization Middleware to set the app's culture.

If the app doesn't already support dynamic culture selection:

- Add localization services to the app with AddLocalization.

- Specify the app's default and supported cultures in the `Program` file. The following example configures supported cultures for United States English and Costa Rican Spanish.

```
C#

builder.Services.AddLocalization();
```

Place Request Localization Middleware before any middleware that might check the request culture. Generally, place the middleware immediately before calling MapRazorComponents:

```
C#

var supportedCultures = new[] { "en-US", "es-CR" };
var localizationOptions = new RequestLocalizationOptions()
    .SetDefaultCulture(supportedCultures[0])
    .AddSupportedCultures(supportedCultures)
    .AddSupportedUICultures(supportedCultures);

app.UseRequestLocalization(localizationOptions);
```

For information on ordering the Localization Middleware in the middleware pipeline, see ASP.NET Core Middleware.

If the app should localize resources based on storing a user's culture setting, use a localization culture cookie. Use of a cookie ensures that the WebSocket connection can correctly propagate the culture. If localization schemes are based on the URL path or query string, the scheme might not be able to work with WebSockets, thus fail to persist the culture. Therefore, the recommended approach is to use a localization culture cookie. See the Dynamically set the server-side culture by user preference section of this article to see an example Razor expression that persists the user's culture selection.

Example of localized resources

The example of localized resources in this section works with the prior examples in this article where the app's supported cultures are English (`en`) as a default locale and Spanish (`es`) as a user-selectable or browser-specified alternate locale.

Create a resource file for each locale. In the following example, resources are created for a `Greeting` string in English and Spanish:

- English (`en`): `Hello, World!`

- Spanish (`es`): `¡Hola, Mundo!`

> **Note**
>
> The following resource file can be added in Visual Studio by right-clicking the `Pages` folder and selecting **Add** > **New Item** > **Resources File**. Name the file `CultureExample2.resx`. When the editor appears, provide data for a new entry. Set the **Name** to `Greeting` and **Value** to `Hello, World!`. Save the file.
>
> If using Visual Studio Code, we recommend installing Tim Heuer's ResX Viewer and Editor. Add an empty `CultureExample2.resx` file to the `Pages` folder. The extension automatically takes over managing the file in the UI. Select the **Add New Resource** button. Follow the instructions to add an entry for `Greeting` (key), `Hello, World!` (value), and `None` (comment). Save the file. If you close and re-open the file, you can see the `Greeting` resource.
>
> Tim Heuer's ResX Viewer and Editor isn't owned or maintained by Microsoft and isn't covered by any Microsoft Support Agreement or license.

The following demonstrates a typical resource file. You can manually place resource files into the app's `Pages` folder if you prefer not to use built-in tooling with an integrated development environment (IDE), such as Visual Studio's built-in resource file editor or Visual Studio Code with an extension for creating and editing resource files.

`Pages/CultureExample2.resx`:

XML

```
<?xml version="1.0" encoding="utf-8"?>
<root>
  <xsd:schema id="root" xmlns=""
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:msdata="urn:schemas-
microsoft-com:xml-msdata">
    <xsd:import namespace="http://www.w3.org/XML/1998/namespace" />
    <xsd:element name="root" msdata:IsDataSet="true">
      <xsd:complexType>
        <xsd:choice maxOccurs="unbounded">
          <xsd:element name="metadata">
            <xsd:complexType>
              <xsd:sequence>
                <xsd:element name="value" type="xsd:string" minOccurs="0" />
              </xsd:sequence>
              <xsd:attribute name="name" use="required" type="xsd:string" />
              <xsd:attribute name="type" type="xsd:string" />
              <xsd:attribute name="mimetype" type="xsd:string" />
              <xsd:attribute ref="xml:space" />
```

```
            </xsd:complexType>
          </xsd:element>
          <xsd:element name="assembly">
            <xsd:complexType>
              <xsd:attribute name="alias" type="xsd:string" />
              <xsd:attribute name="name" type="xsd:string" />
            </xsd:complexType>
          </xsd:element>
          <xsd:element name="data">
            <xsd:complexType>
              <xsd:sequence>
                <xsd:element name="value" type="xsd:string" minOccurs="0"
msdata:Ordinal="1" />
                <xsd:element name="comment" type="xsd:string" minOccurs="0"
msdata:Ordinal="2" />
              </xsd:sequence>
              <xsd:attribute name="name" type="xsd:string" use="required"
msdata:Ordinal="1" />
              <xsd:attribute name="type" type="xsd:string"
msdata:Ordinal="3" />
              <xsd:attribute name="mimetype" type="xsd:string"
msdata:Ordinal="4" />
              <xsd:attribute ref="xml:space" />
            </xsd:complexType>
          </xsd:element>
          <xsd:element name="resheader">
            <xsd:complexType>
              <xsd:sequence>
                <xsd:element name="value" type="xsd:string" minOccurs="0"
msdata:Ordinal="1" />
              </xsd:sequence>
              <xsd:attribute name="name" type="xsd:string" use="required" />
            </xsd:complexType>
          </xsd:element>
        </xsd:choice>
      </xsd:complexType>
    </xsd:element>
  </xsd:schema>
  <resheader name="resmimetype">
    <value>text/microsoft-resx</value>
  </resheader>
  <resheader name="version">
    <value>2.0</value>
  </resheader>
  <resheader name="reader">
    <value>System.Resources.ResXResourceReader, System.Windows.Forms,
Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089</value>
  </resheader>
  <resheader name="writer">
    <value>System.Resources.ResXResourceWriter, System.Windows.Forms,
Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089</value>
  </resheader>
  <data name="Greeting" xml:space="preserve">
    <value>Hello, World!</value>
```

```
  </data>
</root>
```

> **Note**
>
> The following resource file can be added in Visual Studio by right-clicking the `Pages` folder and selecting **Add** > **New Item** > **Resources File**. Name the file `CultureExample2.es.resx`. When the editor appears, provide data for a new entry. Set the **Name** to `Greeting` and **Value** to `¡Hola, Mundo!`. Save the file.
>
> If using Visual Studio Code, we recommend installing Tim Heuer's ResX Viewer and Editor. Add an empty `CultureExample2.resx` file to the `Pages` folder. The extension automatically takes over managing the file in the UI. Select the **Add New Resource** button. Follow the instructions to add an entry for `Greeting` (key), `¡Hola, Mundo!` (value), and `None` (comment). Save the file. If you close and re-open the file, you can see the `Greeting` resource.

The following demonstrates a typical resource file. You can manually place resource files into the app's `Pages` folder if you prefer not to use built-in tooling with an integrated development environment (IDE), such as Visual Studio's built-in resource file editor or Visual Studio Code with an extension for creating and editing resource files.

`Pages/CultureExample2.es.resx`:

XML

```
<?xml version="1.0" encoding="utf-8"?>
<root>
  <xsd:schema id="root" xmlns=""
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:msdata="urn:schemas-
microsoft-com:xml-msdata">
    <xsd:import namespace="http://www.w3.org/XML/1998/namespace" />
    <xsd:element name="root" msdata:IsDataSet="true">
      <xsd:complexType>
        <xsd:choice maxOccurs="unbounded">
          <xsd:element name="metadata">
            <xsd:complexType>
              <xsd:sequence>
                <xsd:element name="value" type="xsd:string" minOccurs="0" />
              </xsd:sequence>
              <xsd:attribute name="name" use="required" type="xsd:string" />
              <xsd:attribute name="type" type="xsd:string" />
              <xsd:attribute name="mimetype" type="xsd:string" />
              <xsd:attribute ref="xml:space" />
            </xsd:complexType>
          </xsd:element>
```

```
          <xsd:element name="assembly">
            <xsd:complexType>
              <xsd:attribute name="alias" type="xsd:string" />
              <xsd:attribute name="name" type="xsd:string" />
            </xsd:complexType>
          </xsd:element>
          <xsd:element name="data">
            <xsd:complexType>
              <xsd:sequence>
                <xsd:element name="value" type="xsd:string" minOccurs="0"
msdata:Ordinal="1" />
                <xsd:element name="comment" type="xsd:string" minOccurs="0"
msdata:Ordinal="2" />
              </xsd:sequence>
              <xsd:attribute name="name" type="xsd:string" use="required"
msdata:Ordinal="1" />
              <xsd:attribute name="type" type="xsd:string"
msdata:Ordinal="3" />
              <xsd:attribute name="mimetype" type="xsd:string"
msdata:Ordinal="4" />
              <xsd:attribute ref="xml:space" />
            </xsd:complexType>
          </xsd:element>
          <xsd:element name="resheader">
            <xsd:complexType>
              <xsd:sequence>
                <xsd:element name="value" type="xsd:string" minOccurs="0"
msdata:Ordinal="1" />
              </xsd:sequence>
              <xsd:attribute name="name" type="xsd:string" use="required" />
            </xsd:complexType>
          </xsd:element>
        </xsd:choice>
      </xsd:complexType>
    </xsd:element>
  </xsd:schema>
  <resheader name="resmimetype">
    <value>text/microsoft-resx</value>
  </resheader>
  <resheader name="version">
    <value>2.0</value>
  </resheader>
  <resheader name="reader">
    <value>System.Resources.ResXResourceReader, System.Windows.Forms,
Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089</value>
  </resheader>
  <resheader name="writer">
    <value>System.Resources.ResXResourceWriter, System.Windows.Forms,
Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089</value>
  </resheader>
  <data name="Greeting" xml:space="preserve">
    <value>¡Hola, Mundo!</value>
  </data>
</root>
```

The following component demonstrates the use of the localized `Greeting` string with IStringLocalizer<T>. The Razor markup `@Loc["Greeting"]` in the following example localizes the string keyed to the `Greeting` value, which is set in the preceding resource files.

Add the namespace for Microsoft.Extensions.Localization to the app's `_Imports.razor` file:

razor

```
@using Microsoft.Extensions.Localization
```

`CultureExample2.razor`:

razor

```
@page "/culture-example-2"
@using System.Globalization
@inject IStringLocalizer<CultureExample2> Loc

<h1>Culture Example 2</h1>

<ul>
    <li><b>CurrentCulture</b>: @CultureInfo.CurrentCulture</li>
    <li><b>CurrentUICulture</b>: @CultureInfo.CurrentUICulture</li>
</ul>

<h2>Greeting</h2>

<p>
    @Loc["Greeting"]
</p>

<p>
    @greeting
</p>

@code {
    private string? greeting;

    protected override void OnInitialized()
    {
        greeting = Loc["Greeting"];
    }
}
```

Optionally, add a menu item for the `CultureExample2` component to the navigation in the `NavMenu` component (`NavMenu.razor`).

WebAssembly culture provider reference source

To further understand how the Blazor framework processes localization, see the WebAssemblyCultureProvider class in the ASP.NET Core reference source.

> **Note**
>
> Documentation links to .NET reference source usually load the repository's default branch, which represents the current development for the next release of .NET. To select a tag for a specific release, use the **Switch branches or tags** dropdown list. For more information, see How to select a version tag of ASP.NET Core source code (dotnet/AspNetCore.Docs #26205).

Shared resources

To create localization shared resources, adopt the following approach.

- Create a dummy class with an arbitrary class name. In the following example:
  - The app uses the `BlazorSample` namespace, and localization assets use the `BlazorSample.Localization` namespace.
  - The dummy class is named `SharedResource`.
  - The class file is placed in a `Localization` folder at the root of the app.

  `Localization/SharedResource.cs`:

  C#

  ```
  namespace BlazorSample.Localization;

  public class SharedResource
  {
  }
  ```

- Create the shared resource files with a **Build Action** of `Embedded resource`. In the following example:

  - The files are placed in the `Localization` folder with the dummy `SharedResource` class (`Localization/SharedResource.cs`).

    - Name the resource files to match the name of the dummy class. The following example files include a default localization file and a file for Spanish (`es`) localization.
    - `Localization/SharedResource.resx`
    - `Localization/SharedResource.es.resx`

    > ⚠ **Warning**
    >
    > When following the approach in this section, you can't simultaneously set LocalizationOptions.ResourcesPath and use IStringLocalizerFactory.Create to load resources.

- To reference the dummy class for an injected IStringLocalizer<T> in a Razor component, either place an @using directive for the localization namespace or include the localization namespace in the dummy class reference. In the following examples:
    - The first example states the `Localization` namespace for the `SharedResource` dummy class with an @using directive.
    - The second example states the `SharedResource` dummy class's namespace explicitly.

    In a Razor component, use ***either*** of the following approaches:

    razor

    ```
    @using Localization
    @inject IStringLocalizer<SharedResource> Loc
    ```

    razor

    ```
    @inject IStringLocalizer<Localization.SharedResource> Loc
    ```

For additional guidance, see Globalization and localization in ASP.NET Core.

Location override using "Sensors" pane in developer tools

When using the location override using the **Sensors** pane in Google Chrome or Microsoft Edge developer tools, the fallback language is reset after prerendering. Avoid

setting the language using the **Sensors** pane when testing. Set the language using the browser's language settings.

For more information, see Blazor Localization does not work with InteractiveServer (dotnet/aspnetcore #53707).

Additional resources

- Set the app base path
- Globalization and localization in ASP.NET Core
- Globalizing and localizing .NET applications
- Resources in .resx Files
- Microsoft Multilingual App Toolkit
- Localization & Generics
- Calling InvokeAsync(StateHasChanged) causes page to fallback to default culture (dotnet/aspnetcore #28521)
- Blazor Localization does not work with InteractiveServer (dotnet/aspnetcore #53707) (Location override using "Sensors" pane)

ASP.NET Core Blazor forms overview

Article • 10/21/2024

> **ⓘ Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the **.NET 9 version of this article**.

This article explains how to use forms in Blazor.

Input components and forms

The Blazor framework supports forms and provides built-in input components:

- Bound to an object or model that can use data annotations.
  - HTML forms with the `<form>` element.
  - EditForm components.
- Built-in input components.

> **ⓘ Note**
>
> Unsupported ASP.NET Core validation features are covered in the **Unsupported validation features** section.

The Microsoft.AspNetCore.Components.Forms namespace provides:

- Classes for managing form elements, state, and validation.
- Access to built-in Input* components.

A project created from the Blazor project template includes the namespace in the app's `_Imports.razor` file, which makes the namespace available to the app's Razor components.

Standard HTML forms are supported. Create a form using the normal HTML `<form>` tag and specify an `@onsubmit` handler for handling the submitted form request.

`StarshipPlainForm.razor`:

---

razor

```razor
@page "/starship-plain-form"
@inject ILogger<StarshipPlainForm> Logger

<form method="post" @onsubmit="Submit" @formname="starship-plain-form">
    <AntiforgeryToken />
    <div>
        <label>
            Identifier:
            <InputText @bind-Value="Model!.Id" />
        </label>
    </div>
    <div>
        <button type="submit">Submit</button>
    </div>
</form>

@code {
    [SupplyParameterFromForm]
    private Starship? Model { get; set; }

    protected override void OnInitialized() => Model ??= new();

    private void Submit() => Logger.LogInformation("Id = {Id}", Model?.Id);

    public class Starship
    {
        public string? Id { get; set; }
    }
}
```

In the preceding `StarshipPlainForm` component:

- The form is rendered where the `<form>` element appears. The form is named with the @formname directive attribute, which uniquely identifies the form to the Blazor framework.
- The model is created in the component's `@code` block and held in a public property (`Model`). The `[SupplyParameterFromForm]` attribute indicates that the value of the associated property should be supplied from the form data. Data in the request that matches the property's name is bound to the property.
- The InputText component is an input component for editing string values. The `@bind-Value` directive attribute binds the `Model.Id` model property to the InputText component's Value property.
- The `Submit` method is registered as a handler for the `@onsubmit` callback. The handler is called when the form is submitted by the user.

> **ⓘ Important**

Always use the **@formname** directive attribute with a unique form name.

Blazor enhances page navigation and form handling by intercepting the request in order to apply the response to the existing DOM, preserving as much of the rendered form as possible. The enhancement avoids the need to fully load the page and provides a much smoother user experience, similar to a single-page app (SPA), although the component is rendered on the server. For more information, see ASP.NET Core Blazor routing and navigation.

Streaming rendering is supported for plain HTML forms.

> **Note**
>
> Documentation links to .NET reference source usually load the repository's default branch, which represents the current development for the next release of .NET. To select a tag for a specific release, use the **Switch branches or tags** dropdown list. For more information, see How to select a version tag of ASP.NET Core source code (dotnet/AspNetCore.Docs #26205).

The preceding example includes antiforgery support by including an AntiforgeryToken component in the form. Antiforgery support is explained further in the Antiforgery support section of this article.

To submit a form based on another element's DOM events, for example `oninput` or `onblur`, use JavaScript to submit the form (submit (MDN documentation)).

Instead of using plain forms in Blazor apps, a form is typically defined with Blazor's built-in form support using the framework's EditForm component. The following Razor component demonstrates typical elements, components, and Razor code to render a webform using an EditForm component.

`Starship1.razor`:

razor

```razor
@page "/starship-1"
@inject ILogger<Starship1> Logger

<EditForm Model="Model" OnSubmit="Submit" FormName="Starship1">
    <div>
        <label>
            Identifier:
            <InputText @bind-Value="Model!.Id" />
        </label>
```

```
    </div>
    <div>
        <button type="submit">Submit</button>
    </div>
</EditForm>

@code {
    [SupplyParameterFromForm]
    private Starship? Model { get; set; }

    protected override void OnInitialized() => Model ??= new();

    private void Submit() => Logger.LogInformation("Id = {Id}", Model?.Id);

    public class Starship
    {
        public string? Id { get; set; }
    }
}
```

In the preceding `Starship1` component:

- The EditForm component is rendered where the `<EditForm>` element appears. The form is named with the @formname directive attribute, which uniquely identifies the form to the Blazor framework.
- The model is created in the component's `@code` block and held in a public property (`Model`). The property is assigned to the EditForm.Model parameter. The `[SupplyParameterFromForm]` attribute indicates that the value of the associated property should be supplied from the form data. Data in the request that matches the property's name is bound to the property.
- The InputText component is an input component for editing string values. The `@bind-Value` directive attribute binds the `Model.Id` model property to the InputText component's Value property.
- The `Submit` method is registered as a handler for the OnSubmit callback. The handler is called when the form is submitted by the user.

> **ⓘ Important**
>
> Always use the **@formname** directive attribute with a unique form name.

Blazor enhances page navigation and form handling for EditForm components. For more information, see ASP.NET Core Blazor routing and navigation.

Streaming rendering is supported for EditForm.

> **Note**
>
> Documentation links to .NET reference source usually load the repository's default branch, which represents the current development for the next release of .NET. To select a tag for a specific release, use the **Switch branches or tags** dropdown list. For more information, see How to select a version tag of ASP.NET Core source code (dotnet/AspNetCore.Docs #26205).

†For more information on property binding, see ASP.NET Core Blazor data binding.

In the next example, the preceding component is modified to create the form in the `Starship2` component:

- OnSubmit is replaced with OnValidSubmit, which processes assigned event handler if the form is valid when submitted by the user.
- A ValidationSummary component is added to display validation messages when the form is invalid on form submission.
- The data annotations validator (DataAnnotationsValidator component†) attaches validation support using data annotations:
  - If the `<input>` form field is left blank when the **`Submit`** button is selected, an error appears in the validation summary (ValidationSummary component‡) ("`The Id field is required.`") and `Submit` is **not** called.
  - If the `<input>` form field contains more than ten characters when the **`Submit`** button is selected, an error appears in the validation summary ("`Id is too long.`"). `Submit` is **not** called.
  - If the `<input>` form field contains a valid value when the **`Submit`** button is selected, `Submit` is called.

†The DataAnnotationsValidator component is covered in the Validator component section. ‡The ValidationSummary component is covered in the Validation Summary and Validation Message components section.

`Starship2.razor`:

razor

```razor
@page "/starship-2"
@using System.ComponentModel.DataAnnotations
@inject ILogger<Starship2> Logger

<EditForm Model="Model" OnValidSubmit="Submit" FormName="Starship2">
    <DataAnnotationsValidator />
    <ValidationSummary />
```

```
    <label>
        Identifier:
        <InputText @bind-Value="Model!.Id" />
    </label>
    <button type="submit">Submit</button>
</EditForm>

@code {
    [SupplyParameterFromForm]
    private Starship? Model { get; set; }

    protected override void OnInitialized() => Model ??= new();

    private void Submit() => Logger.LogInformation("Id = {Id}", Model?.Id);

    public class Starship
    {
        [Required]
        [StringLength(10, ErrorMessage = "Id is too long.")]
        public string? Id { get; set; }
    }
}
```

Handle form submission

The EditForm provides the following callbacks for handling form submission:

- Use OnValidSubmit to assign an event handler to run when a form with valid fields is submitted.
- Use OnInvalidSubmit to assign an event handler to run when a form with invalid fields is submitted.
- Use OnSubmit to assign an event handler to run regardless of the form fields' validation status. The form is validated by calling EditContext.Validate in the event handler method. If Validate returns `true`, the form is valid.

Clear a form or field

Reset a form by clearing its model back its default state, which can be performed inside or outside of an EditForm's markup:

razor

```
<button @onclick="ClearForm">Clear form</button>

...
```

```
private void ClearForm() => Model = new();
```

Alternatively, use an explicit Razor expression:

razor

```
<button @onclick="@(() => Model = new())">Clear form</button>
```

Reset a field by clearing its model value back to its default state:

razor

```
<button @onclick="ResetId">Reset Identifier</button>

...

private void ResetId() => Model!.Id = string.Empty;
```

Alternatively, use an explicit Razor expression:

razor

```
<button @onclick="@(() => Model!.Id = string.Empty)">Reset
Identifier</button>
```

There's no need to call StateHasChanged in the preceding examples because StateHasChanged is automatically called by the Blazor framework to rerender the component after an event handler is invoked. If an event handler isn't used to invoke the methods that clear a form or field, then developer code should call StateHasChanged to rerender the component.

Antiforgery support

Antiforgery services are automatically added to Blazor apps when AddRazorComponents is called in the `Program` file.

The app uses Antiforgery Middleware by calling UseAntiforgery in its request processing pipeline in the `Program` file. UseAntiforgery is called after the call to UseRouting. If there are calls to UseRouting and UseEndpoints, the call to UseAntiforgery must go between them. A call to UseAntiforgery must be placed after calls to UseAuthentication and UseAuthorization.

The AntiforgeryToken component renders an antiforgery token as a hidden field, and the `[RequireAntiforgeryToken]` attribute enables antiforgery protection. If an antiforgery check fails, a 400 - Bad Request response is thrown and the form isn't processed.

For forms based on EditForm, the AntiforgeryToken component and `[RequireAntiforgeryToken]` attribute are automatically added to provide antiforgery protection.

For forms based on the HTML `<form>` element, manually add the AntiforgeryToken component to the form:

razor

```razor
<form method="post" @onsubmit="Submit" @formname="starshipForm">
    <AntiforgeryToken />
    <input id="send" type="submit" value="Send" />
</form>

@if (submitted)
{
    <p>Form submitted!</p>
}

@code{
    private bool submitted = false;

    private void Submit() => submitted = true;
}
```

> ⚠ **Warning**
>
> For forms based on either **EditForm** or the HTML `<form>` element, antiforgery protection can be disabled by passing `required: false` to the `[RequireAntiforgeryToken]` attribute. The following example disables antiforgery and is ***not recommended*** for public apps:
>
> razor
>
> ```razor
> @using Microsoft.AspNetCore.Antiforgery
> @attribute [RequireAntiforgeryToken(required: false)]
> ```

For more information, see ASP.NET Core Blazor authentication and authorization.

Mitigate overposting attacks

Statically-rendered server-side forms, such as those typically used in components that create and edit records in a database with a form model, can be vulnerable to an *overposting* attack, also known as a *mass assignment* attack. An overposting attack occurs when a malicious user issues an HTML form POST to the server that processes data for properties that aren't part of the rendered form and that the developer doesn't wish to allow users to modify. The term "overposting" literally means that the malicious user has *over*-POSTed with the form.

Overposting isn't a concern when the model doesn't include restricted properties for create and update operations. However, it's important to keep overposting in mind when working with static SSR-based Blazor forms that you maintain.

To mitigate overposting, we recommend using a separate view model/data transfer object (DTO) for the form and database with create (insert) and update operations. When the form is submitted, only properties of the view model/DTO are used by the component and C# code to modify the database. Any extra data included by a malicious user is discarded, so the malicious user is prevented from conducting an overposting attack.

Enhanced form handling

Enhance navigation for form POST requests with the Enhance parameter for EditForm forms or the `data-enhance` attribute for HTML forms (`<form>`):

razor

```
<EditForm ... Enhance ...>
    ...
</EditForm>
```

HTML

```
<form ... data-enhance ...>
    ...
</form>
```

❌ You can't set enhanced navigation on a form's ancestor element to enable enhanced form handling.

HTML

```
<div ... data-enhance ...>
    <form ...>
        <!-- NOT enhanced -->
```

```
    </form>
</div>
```

Enhanced form posts only work with Blazor endpoints. Posting an enhanced form to non-Blazor endpoint results in an error.

To disable enhanced form handling:

- For an EditForm, remove the Enhance parameter from the form element (or set it to `false`: `Enhance="false"`).
- For an HTML `<form>`, remove the `data-enhance` attribute from form element (or set it to `false`: `data-enhance="false"`).

Blazor's enhanced navigation and form handing may undo dynamic changes to the DOM if the updated content isn't part of the server rendering. To preserve the content of an element, use the `data-permanent` attribute.

In the following example, the content of the `<div>` element is updated dynamically by a script when the page loads:

HTML

```
<div data-permanent>
    ...
</div>
```

To disable enhanced navigation and form handling globally, see ASP.NET Core Blazor startup.

For guidance on using the `enhancedload` event to listen for enhanced page updates, see ASP.NET Core Blazor routing and navigation.

Examples

Examples don't adopt enhanced form handling for form POST requests, but all of the examples can be updated to adopt the enhanced features by following the guidance in the Enhanced form handling section.

To demonstrate how forms work with data annotations validation, example components rely on System.ComponentModel.DataAnnotations API. If you wish to avoid an extra line of code in components that use data annotations, make the namespace available throughout the app's components with the imports file (`_Imports.razor`):

```
```

razor

```
@using System.ComponentModel.DataAnnotations
```

Form examples reference aspects of the Star Trek universe. Star Trek is a copyright ©1966-2023 of CBS Studios and Paramount.

Client-side validation requires a circuit

In Blazor Web Apps, client-side validation requires an active Blazor SignalR circuit. Client-side validation isn't available to forms in components that have adopted static server-side rendering (static SSR). Forms that adopt static SSR are validated on the server after the form is submitted.

Unsupported validation features

All of the data annotation built-in validators are supported in Blazor except for the [Remote] validation attribute.

jQuery validation isn't supported in Razor components. We recommend any of the following approaches:

- Follow the guidance in ASP.NET Core Blazor forms validation for either:
  - Server-side validation in a Blazor Web App that adopts an interactive render mode.
  - Client-side validation in a standalone Blazor Web Assembly app.
- Use native HTML validation attributes (see Client-side form validation (MDN documentation)).
- Adopt a third-party validation JavaScript library.

For statically-rendered forms on the server, a new mechanism for client-side validation is under consideration for .NET 10 in late 2025. For more information, see Create server rendered forms with client validation using Blazor without a circuit (dotnet/aspnetcore #51040).

Additional resources

- ASP.NET Core Blazor file uploads
- Blazor samples GitHub repository (dotnet/blazor-samples) (how to download)
- ASP.NET Core GitHub repository (dotnet/aspnetcore) forms test assets

ASP.NET Core Blazor input components

Article • 10/18/2024

> **Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the .NET 9 version of this article.

This article describes Blazor's built-in input components.

Input components

The Blazor framework provides built-in input components to receive and validate user input. The built-in input components in the following table are supported in an EditForm with an EditContext.

The components in the table are also supported outside of a form in Razor component markup. Inputs are validated when they're changed and when a form is submitted.

Expand table

Input component	Rendered as...
InputCheckbox	`<input type="checkbox">`
InputDate<TValue>	`<input type="date">`
InputFile	`<input type="file">`
InputNumber<TValue>	`<input type="number">`
InputRadio<TValue>	`<input type="radio">`
InputRadioGroup<TValue>	Group of child InputRadio<TValue>
InputSelect<TValue>	`<select>`
InputText	`<input>`
InputTextArea	`<textarea>`

For more information on the InputFile component, see ASP.NET Core Blazor file uploads.

All of the input components, including EditForm, support arbitrary attributes. Any attribute that doesn't match a component parameter is added to the rendered HTML element.

Input components provide default behavior for validating when a field is changed:

- For input components in a form with an EditContext, the default validation behavior includes updating the field CSS class to reflect the field's state as valid or invalid with validation styling of the underlying HTML element.
- For controls that don't have an EditContext, the default validation reflects the valid or invalid state but doesn't provide validation styling to the underlying HTML element.

Some components include useful parsing logic. For example, InputDate<TValue> and InputNumber<TValue> handle unparseable values gracefully by registering unparseable values as validation errors. Types that can accept null values also support nullability of the target field (for example, `int?` for a nullable integer).

The InputNumber<TValue> component supports the type="range" attribute, which creates a range input that supports model binding and form validation, typically rendered as a slider or dial control rather than a text box:

```razor
<InputNumber @bind-Value="..." max="..." min="..." step="..." type="range"
/>
```

For more information on the InputFile component, see ASP.NET Core Blazor file uploads.

Example form

The following `Starship` type, which is used in several of this article's examples and examples in other *Forms* node articles, defines a diverse set of properties with data annotations:

- `Id` is required because it's annotated with the RequiredAttribute. `Id` requires a value of at least one character but no more than 16 characters using the StringLengthAttribute.
- `Description` is optional because it isn't annotated with the RequiredAttribute.
- `Classification` is required.

- The `MaximumAccommodation` property defaults to zero but requires a value from one to 100,000 per its RangeAttribute.
- `IsValidatedDesign` requires that the property have a `true` value, which matches a selected state when the property is bound to a checkbox in the UI (`<input type="checkbox">`).
- `ProductionDate` is a DateTime and required.

`Starship.cs`:

C#

```
using System.ComponentModel.DataAnnotations;

namespace BlazorSample;

public class Starship
{
    [Required]
    [StringLength(16, ErrorMessage = "Identifier too long (16 character
limit).")]
    public string? Id { get; set; }

    public string? Description { get; set; }

    [Required]
    public string? Classification { get; set; }

    [Range(1, 100000, ErrorMessage = "Accommodation invalid (1-100000).")]
    public int MaximumAccommodation { get; set; }

    [Required]
    [Range(typeof(bool), "true", "true", ErrorMessage = "Approval
required.")]
    public bool IsValidatedDesign { get; set; }

    [Required]
    public DateTime ProductionDate { get; set; }
}
```

The following form accepts and validates user input using:

- The properties and validation defined in the preceding `Starship` model.
- Several of Blazor's built-in input components.

When the model property for the ship's classification (`Classification`) is set, the option matching the model is checked. For example, `checked="@(Model!.Classification == "Exploration")"` for the classification of an exploration ship. The reason for explicitly setting the checked option is that the value of a `<select>` element is only present in the

browser. If the form is rendered on the server after it's submitted, any state from the client is overridden with state from the server, which doesn't ordinarily mark an option as checked. By setting the checked option from the model property, the classification always reflects the model's state. This preserves the classification selection across form submissions that result in the form rerendering on the server. In situations where the form isn't rerendered on the server, such as when the Interactive Server render mode is applied directly to the component, explicit assignment of the checked option from the model isn't necessary because Blazor preserves the state for the `<select>` element on the client.

`Starship3.razor`:

razor

```razor
@page "/starship-3"
@inject ILogger<Starship3> Logger

<h1>Starfleet Starship Database</h1>

<h2>New Ship Entry Form</h2>

<EditForm Model="Model" OnValidSubmit="Submit" FormName="Starship3">
    <DataAnnotationsValidator />
    <ValidationSummary />
    <div>
        <label>
            Identifier:
            <InputText @bind-Value="Model!.Id" />
        </label>
    </div>
    <div>
        <label>
            Description (optional):
            <InputTextArea @bind-Value="Model!.Description" />
        </label>
    </div>
    <div>
        <label>
            Primary Classification:
            <InputSelect @bind-Value="Model!.Classification">
                <option value="">
                    Select classification ...
                </option>
                <option checked="@(Model!.Classification == "Exploration")"
                    value="Exploration">
                    Exploration
                </option>
                <option checked="@(Model!.Classification == "Diplomacy")"
                    value="Diplomacy">
                    Diplomacy
                </option>
```

```
                <option checked="@(Model!.Classification == "Defense")"
                    value="Defense">
                    Defense
                </option>
            </InputSelect>
        </label>
    </div>
    <div>
        <label>
            Maximum Accommodation:
            <InputNumber @bind-Value="Model!.MaximumAccommodation" />
        </label>
    </div>
    <div>
        <label>
            Engineering Approval:
            <InputCheckbox @bind-Value="Model!.IsValidatedDesign" />
        </label>
    </div>
    <div>
        <label>
            Production Date:
            <InputDate @bind-Value="Model!.ProductionDate" />
        </label>
    </div>
    <div>
        <button type="submit">Submit</button>
    </div>
</EditForm>

@code {
    [SupplyParameterFromForm]
    private Starship? Model { get; set; }

    protected override void OnInitialized() =>
        Model ??= new() { ProductionDate = DateTime.UtcNow };

    private void Submit()
    {
        Logger.LogInformation("Id = {Id} Description = {Description} " +
            "Classification = {Classification} MaximumAccommodation = " +
            "{MaximumAccommodation} IsValidatedDesign = " +
            "{IsValidatedDesign} ProductionDate = {ProductionDate}",
            Model?.Id, Model?.Description, Model?.Classification,
            Model?.MaximumAccommodation, Model?.IsValidatedDesign,
            Model?.ProductionDate);
    }
}
```

The EditForm in the preceding example creates an EditContext based on the assigned `Starship` instance (`Model="..."`) and handles a valid form. The next example demonstrates how to assign an EditContext to a form and validate when the form is submitted.

In the following example:

- A shortened version of the earlier `Starfleet Starship Database` form (`Starship3` component) is used that only accepts a value for the starship's Id. The other `Starship` properties receive valid default values when an instance of the `Starship` type is created.
- The `Submit` method executes when the **`Submit`** button is selected.
- The form is validated by calling EditContext.Validate in the `Submit` method.
- Logging is executed depending on the validation result.

`Starship4.razor`:

razor

```razor
@page "/starship-4"
@inject ILogger<Starship4> Logger

<EditForm EditContext="editContext" OnSubmit="Submit" FormName="Starship4">
    <DataAnnotationsValidator />
    <div>
        <label>
            Identifier:
            <InputText @bind-Value="Model!.Id" />
        </label>
    </div>
    <div>
        <button type="submit">Submit</button>
    </div>
</EditForm>

@code {
    private EditContext? editContext;

    [SupplyParameterFromForm]
    private Starship? Model { get; set; }

    protected override void OnInitialized()
    {
        Model ??=
            new()
                {
                    Id = "NCC-1701",
                    Classification = "Exploration",
                    MaximumAccommodation = 150,
                    IsValidatedDesign = true,
                    ProductionDate = new DateTime(2245, 4, 11)
                };
        editContext = new(Model);
    }

    private void Submit()
```

```
    {
        if (editContext != null && editContext.Validate())
        {
            Logger.LogInformation("Submit: Form is valid");
        }
        else
        {
            Logger.LogInformation("Submit: Form is INVALID");
        }
    }
}
```

> ⓘ Note
>
> Changing the EditContext after it's assigned is **not** supported.

Multiple option selection with the `InputSelect` component

Binding supports multiple option selection with the InputSelect<TValue> component. The @onchange event provides an array of the selected options via event arguments (ChangeEventArgs). The value must be bound to an array type, and binding to an array type makes the multiple attribute optional on the InputSelect<TValue> tag.

In the following example, the user must select at least two starship classifications but no more than three classifications.

`Starship5.razor`:

razor

```
@page "/starship-5"
@using System.ComponentModel.DataAnnotations
@inject ILogger<Starship5> Logger

<h1>Bind Multiple <code>InputSelect</code> Example</h1>

<EditForm EditContext="editContext" OnValidSubmit="Submit"
FormName="Starship5">
    <DataAnnotationsValidator />
    <ValidationSummary />
    <div>
        <label>
            Select classifications (Minimum: 2, Maximum: 3):
            <InputSelect @bind-Value="Model!.SelectedClassification">
                <option
value="@Classification.Exploration">Exploration</option>
```

```
                <option value="@Classification.Diplomacy">Diplomacy</option>
                <option value="@Classification.Defense">Defense</option>
                <option value="@Classification.Research">Research</option>
            </InputSelect>
        </label>
    </div>
    <div>
        <button type="submit">Submit</button>
    </div>
</EditForm>

@if (Model?.SelectedClassification?.Length > 0)
{
    <div>@string.Join(", ", Model.SelectedClassification)</div>
}

@code {
    private EditContext? editContext;

    [SupplyParameterFromForm]
    private Starship? Model { get; set; }

    protected override void OnInitialized()
    {
        Model = new();
        editContext = new(Model);
    }

    private void Submit() => Logger.LogInformation("Submit: Processing
form");

    private class Starship
    {
        [Required]
        [MinLength(2, ErrorMessage = "Select at least two
classifications.")]
        [MaxLength(3, ErrorMessage = "Select no more than three
classifications.")]
        public Classification[]? SelectedClassification { get; set; } =
            new[] { Classification.None };
    }

    private enum Classification { None, Exploration, Diplomacy, Defense,
Research }
}
```

For information on how empty strings and `null` values are handled in data binding, see the Binding InputSelect options to C# object null values section.

Binding `InputSelect` options to C# object `null` values

For information on how empty strings and `null` values are handled in data binding, see ASP.NET Core Blazor data binding.

Display name support

Several built-in components support display names with the InputBase<TValue>.DisplayName parameter.

In the `Starfleet Starship Database` form (`Starship3` component) of the Example form section, the production date of a new starship doesn't specify a display name:

razor

```
<label>
    Production Date:
    <InputDate @bind-Value="Model!.ProductionDate" />
</label>
```

If the field contains an invalid date when the form is submitted, the error message doesn't display a friendly name. The field name, "`ProductionDate`" doesn't have a space between "`Production`" and "`Date`" when it appears in the validation summary:

> The ProductionDate field must be a date.

Set the DisplayName property to a friendly name with a space between the words "`Production`" and "`Date`":

razor

```
<label>
    Production Date:
    <InputDate @bind-Value="Model!.ProductionDate"
        DisplayName="Production Date" />
</label>
```

The validation summary displays the friendly name when the field's value is invalid:

> The Production Date field must be a date.

Error message template support

InputDate<TValue> and InputNumber<TValue> support error message templates:

- InputDate<TValue>.ParsingErrorMessage
- InputNumber<TValue>.ParsingErrorMessage

In the `Starfleet Starship Database` form (`Starship3` component) of the Example form section with a friendly display name assigned, the `Production Date` field produces an error message using the following default error message template:

```
css

The {0} field must be a date.
```

The position of the `{0}` placeholder is where the value of the DisplayName property appears when the error is displayed to the user.

```
razor

<label>
    Production Date:
    <InputDate @bind-Value="Model!.ProductionDate"
        DisplayName="Production Date" />
</label>
```

> The Production Date field must be a date.

Assign a custom template to ParsingErrorMessage to provide a custom message:

```
razor

<label>
    Production Date:
    <InputDate @bind-Value="Model!.ProductionDate"
        DisplayName="Production Date"
        ParsingErrorMessage="The {0} field has an incorrect date value." />
</label>
```

> The Production Date field has an incorrect date value.

ASP.NET Core Blazor forms binding

Article • 10/18/2024

> **Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the .NET 9 version of this article.

This article explains how to use binding in Blazor forms.

`EditForm` / `EditContext` model

An EditForm creates an EditContext based on the assigned object as a cascading value for other components in the form. The EditContext tracks metadata about the edit process, including which form fields have been modified and the current validation messages. Assigning to either an EditForm.Model or an EditForm.EditContext can bind a form to data.

Model binding

Assignment to EditForm.Model:

razor

```
<EditForm ... Model="Model" ...>
    ...
</EditForm>

@code {
    [SupplyParameterFromForm]
    private Starship? Model { get; set; }

    protected override void OnInitialized() => Model ??= new();
}
```

Context binding

Assignment to EditForm.EditContext: