For more information on memory management, see Host and deploy ASP.NET Core server-side Blazor apps.

Inline child components into their parents

Consider the following portion of a parent component that renders child components in a loop:

razor

```razor
<div class="chat">
    @foreach (var message in messages)
    {
        <ChatMessageDisplay Message="message" />
    }
</div>
```

`ChatMessageDisplay.razor`:

razor

```razor
<div class="chat-message">
    <span class="author">@Message.Author</span>
    <span class="text">@Message.Text</span>
</div>

@code {
    [Parameter]
    public ChatMessage? Message { get; set; }
}
```

The preceding example performs well if thousands of messages aren't shown at once. To show thousands of messages at once, consider *not* factoring out the separate `ChatMessageDisplay` component. Instead, inline the child component into the parent. The following approach avoids the per-component overhead of rendering so many child components at the cost of losing the ability to rerender each child component's markup independently:

razor

```razor
<div class="chat">
    @foreach (var message in messages)
    {
        <div class="chat-message">
            <span class="author">@message.Author</span>
            <span class="text">@message.Text</span>
        </div>
    }
</div>
```

Define reusable `RenderFragments` in code

You might be factoring out child components purely as a way of reusing rendering logic. If that's the case, you can create reusable rendering logic without implementing additional components. In any component's `@code` block, define a RenderFragment. Render the fragment from any location as many times as needed:

razor

```razor
@RenderWelcomeInfo

<p>Render the welcome content a second time:</p>

@RenderWelcomeInfo

@code {
    private RenderFragment RenderWelcomeInfo = @<p>Welcome to your new app!
</p>;
}
```

To make RenderTreeBuilder code reusable across multiple components, declare the RenderFragment public and static:

razor

```razor
public static RenderFragment SayHello = @<h1>Hello!</h1>;
```

`SayHello` in the preceding example can be invoked from an unrelated component. This technique is useful for building libraries of reusable markup snippets that render without per-component overhead.

RenderFragment delegates can accept parameters. The following component passes the message (`message`) to the RenderFragment delegate:

razor

```razor
<div class="chat">
    @foreach (var message in messages)
    {
        @ChatMessageDisplay(message)
    }
</div>
```

```
@code {
    private RenderFragment<ChatMessage> ChatMessageDisplay = message =>
        @<div class="chat-message">
            <span class="author">@message.Author</span>
            <span class="text">@message.Text</span>
        </div>;
}
```

The preceding approach reuses rendering logic without per-component overhead. However, the approach doesn't permit refreshing the subtree of the UI independently, nor does it have the ability to skip rendering the subtree of the UI when its parent renders because there's no component boundary. Assignment to a RenderFragment delegate is only supported in Razor component files (`.razor`), and event callbacks aren't supported.

For a non-static field, method, or property that can't be referenced by a field initializer, such as `TitleTemplate` in the following example, use a property instead of a field for the RenderFragment:

C#

```
protected RenderFragment DisplayTitle =>
    @<div>
        @TitleTemplate
    </div>;
```

Don't receive too many parameters

If a component repeats extremely often, for example, hundreds or thousands of times, the overhead of passing and receiving each parameter builds up.

It's rare that too many parameters severely restricts performance, but it can be a factor. For a `TableCell` component that renders 4,000 times within a grid, each parameter passed to the component adds around 15 ms to the total rendering cost. Passing ten parameters requires around 150 ms and causes a UI rendering lag.

To reduce parameter load, bundle multiple parameters in a custom class. For example, a table cell component might accept a common object. In the following example, `Data` is different for every cell, but `Options` is common across all cell instances:

razor

```
@typeparam TItem

...
```

```
@code {
    [Parameter]
    public TItem? Data { get; set; }

    [Parameter]
    public GridOptions? Options { get; set; }
}
```

However, consider that it might be an improvement not to have a table cell component, as shown in the preceding example, and instead inline its logic into the parent component.

> ⓘ **Note**
>
> When multiple approaches are available for improving performance, benchmarking the approaches is usually required to determine which approach yields the best results.

For more information on generic type parameters (`@typeparam`), see the following resources:

- Razor syntax reference for ASP.NET Core
- ASP.NET Core Razor components
- ASP.NET Core Blazor templated components

Ensure cascading parameters are fixed

The CascadingValue component has an optional `IsFixed` parameter:

- If `IsFixed` is `false` (the default), every recipient of the cascaded value sets up a subscription to receive change notifications. Each `[CascadingParameter]` is **substantially more expensive** than a regular `[Parameter]` due to the subscription tracking.
- If `IsFixed` is `true` (for example, `<CascadingValue Value="someValue" IsFixed="true">`), recipients receive the initial value but don't set up a subscription to receive updates. Each `[CascadingParameter]` is lightweight and no more expensive than a regular `[Parameter]`.

Setting `IsFixed` to `true` improves performance if there are a large number of other components that receive the cascaded value. Wherever possible, set `IsFixed` to `true` on cascaded values. You can set `IsFixed` to `true` when the supplied value doesn't change over time.

Where a component passes `this` as a cascaded value, `IsFixed` can also be set to `true`:

razor

```razor
<CascadingValue Value="this" IsFixed="true">
    <SomeOtherComponents>
</CascadingValue>
```

For more information, see ASP.NET Core Blazor cascading values and parameters.

Avoid attribute splatting with `CaptureUnmatchedValues`

Components can elect to receive "unmatched" parameter values using the CaptureUnmatchedValues flag:

razor

```razor
<div @attributes="OtherAttributes">...</div>

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

This approach allows passing arbitrary additional attributes to the element. However, this approach is expensive because the renderer must:

- Match all of the supplied parameters against the set of known parameters to build a dictionary.
- Keep track of how multiple copies of the same attribute overwrite each other.

Use CaptureUnmatchedValues where component rendering performance isn't critical, such as components that aren't repeated frequently. For components that render at scale, such as each item in a large list or in the cells of a grid, try to avoid attribute splatting.

For more information, see ASP.NET Core Blazor attribute splatting and arbitrary parameters.

Implement `SetParametersAsync` manually

A significant source of per-component rendering overhead is writing incoming parameter values to `[Parameter]` properties. The renderer uses reflection to write the parameter values, which can lead to poor performance at scale.

In some extreme cases, you may wish to avoid the reflection and implement your own parameter-setting logic manually. This may be applicable when:

- A component renders extremely often, for example, when there are hundreds or thousands of copies of the component in the UI.
- A component accepts many parameters.
- You find that the overhead of receiving parameters has an observable impact on UI responsiveness.

In extreme cases, you can override the component's virtual SetParametersAsync method and implement your own component-specific logic. The following example deliberately avoids dictionary lookups:

razor

```razor
@code {
    [Parameter]
    public int MessageId { get; set; }

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

    [Parameter]
    public EventCallback<string> TextChanged { get; set; }

    [Parameter]
    public Theme CurrentTheme { get; set; }

    public override Task SetParametersAsync(ParameterView parameters)
    {
        foreach (var parameter in parameters)
        {
            switch (parameter.Name)
            {
                case nameof(MessageId):
                    MessageId = (int)parameter.Value;
                    break;
                case nameof(Text):
                    Text = (string)parameter.Value;
                    break;
                case nameof(TextChanged):
                    TextChanged = (EventCallback<string>)parameter.Value;
                    break;
                case nameof(CurrentTheme):
                    CurrentTheme = (Theme)parameter.Value;
                    break;
                default:
                    throw new ArgumentException($"Unknown parameter:
{parameter.Name}");
            }
        }

        return base.SetParametersAsync(ParameterView.Empty);
    }
}
```

In the preceding code, returning the base class SetParametersAsync runs the normal lifecycle method without assigning parameters again.

As you can see in the preceding code, overriding SetParametersAsync and supplying custom logic is complicated and laborious, so we don't generally recommend adopting this approach. In extreme cases, it can improve rendering performance by 20-25%, but you should only consider this approach in the extreme scenarios listed earlier in this section.

Don't trigger events too rapidly

Some browser events fire extremely frequently. For example, `onmousemove` and `onscroll` can fire tens or hundreds of times per second. In most cases, you don't need to perform UI updates this frequently. If events are triggered too rapidly, you may harm UI responsiveness or consume excessive CPU time.

Rather than use native events that rapidly fire, consider the use of JS interop to register a callback that fires less frequently. For example, the following component displays the position of the mouse but only updates at most once every 500 ms:

```razor
@implements IDisposable
@inject IJSRuntime JS

<h1>@message</h1>

<div @ref="mouseMoveElement" style="border:1px dashed red;height:200px;">
    Move mouse here
</div>

@code {
    private ElementReference mouseMoveElement;
    private DotNetObjectReference<MyComponent>? selfReference;
    private string message = "Move the mouse in the box";

    [JSInvokable]
    public void HandleMouseMove(int x, int y)
    {
        message = $"Mouse move at {x}, {y}";
        StateHasChanged();
    }

    protected override async Task OnAfterRenderAsync(bool firstRender)
    {
        if (firstRender)
        {
            selfReference = DotNetObjectReference.Create(this);
            var minInterval = 500;

            await JS.InvokeVoidAsync("onThrottledMouseMove",
                mouseMoveElement, selfReference, minInterval);
        }
    }

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

The corresponding JavaScript code registers the DOM event listener for mouse movement. In this example, the event listener uses Lodash's throttle function to limit the rate of invocations:

HTML

```html
<script
src="https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.17.20/lodash.min.js"
></script>
<script>
  function onThrottledMouseMove(elem, component, interval) {
    elem.addEventListener('mousemove', _.throttle(e => {
      component.invokeMethodAsync('HandleMouseMove', e.offsetX, e.offsetY);
    }, interval));
  }
</script>
```

Avoid rerendering after handling events without state changes

Components inherit from ComponentBase, which automatically invokes StateHasChanged after the component's event handlers are invoked. In some cases, it might be unnecessary or undesirable to trigger a rerender after an event handler is invoked. For example, an event handler might not modify component state. In these scenarios, the app can leverage the IHandleEvent interface to control the behavior of Blazor's event handling.

> **Note**
>
> The approach in this section doesn't flow exceptions to error boundaries. For more information and demonstration code that supports error boundaries by calling ComponentBase.DispatchExceptionAsync, see AsNonRenderingEventHandler + ErrorBoundary = unexpected behavior (dotnet/aspnetcore #54543).

To prevent rerenders for all of a component's event handlers, implement IHandleEvent and provide a IHandleEvent.HandleEventAsync task that invokes the event handler without calling StateHasChanged.

In the following example, no event handler added to the component triggers a rerender, so `HandleSelect` doesn't result in a rerender when invoked.

`HandleSelect1.razor`:

razor

```razor
@page "/handle-select-1"
@using Microsoft.Extensions.Logging
@implements IHandleEvent
@inject ILogger<HandleSelect1> Logger

<p>
    Last render DateTime: @dt
</p>

<button @onclick="HandleSelect">
    Select me (Avoids Rerender)
</button>

@code {
    private DateTime dt = DateTime.Now;

    private void HandleSelect()
    {
        dt = DateTime.Now;

        Logger.LogInformation("This event handler doesn't trigger a 
rerender.");
    }

    Task IHandleEvent.HandleEventAsync(
        EventCallbackWorkItem callback, object? arg) => 
callback.InvokeAsync(arg);
}
```

In addition to preventing rerenders after event handlers fire in a component in a global fashion, it's possible to prevent rerenders after a single event handler by employing the following utility method.

Add the following `EventUtil` class to a Blazor app. The static actions and functions at the top of the `EventUtil` class provide handlers that cover several combinations of arguments and return types that Blazor uses when handling events.

`EventUtil.cs`:

```
C#

using System;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Components;

public static class EventUtil
{
    public static Action AsNonRenderingEventHandler(Action callback)
        => new SyncReceiver(callback).Invoke;
    public static Action<TValue> AsNonRenderingEventHandler<TValue>(
            Action<TValue> callback)
        => new SyncReceiver<TValue>(callback).Invoke;
    public static Func<Task> AsNonRenderingEventHandler(Func<Task> callback)
        => new AsyncReceiver(callback).Invoke;
    public static Func<TValue, Task> AsNonRenderingEventHandler<TValue>(
            Func<TValue, Task> callback)
        => new AsyncReceiver<TValue>(callback).Invoke;

    private record SyncReceiver(Action callback)
        : ReceiverBase { public void Invoke() => callback(); }
    private record SyncReceiver<T>(Action<T> callback)
        : ReceiverBase { public void Invoke(T arg) => callback(arg); }
    private record AsyncReceiver(Func<Task> callback)
        : ReceiverBase { public Task Invoke() => callback(); }
    private record AsyncReceiver<T>(Func<T, Task> callback)
        : ReceiverBase { public Task Invoke(T arg) => callback(arg); }

    private record ReceiverBase : IHandleEvent
    {
        public Task HandleEventAsync(EventCallbackWorkItem item, object arg)
=>
            item.InvokeAsync(arg);
    }
}
```

Call `EventUtil.AsNonRenderingEventHandler` to call an event handler that doesn't trigger a render when invoked.

In the following example:

- Selecting the first button, which calls `HandleClick1`, triggers a rerender.
- Selecting the second button, which calls `HandleClick2`, doesn't trigger a rerender.
- Selecting the third button, which calls `HandleClick3`, doesn't trigger a rerender and uses event arguments (MouseEventArgs).

`HandleSelect2.razor`:

razor

```
@page "/handle-select-2"
@using Microsoft.Extensions.Logging
@inject ILogger<HandleSelect2> Logger

<p>
    Last render DateTime: @dt
</p>

<button @onclick="HandleClick1">
    Select me (Renders)
</button>

<button @onclick="EventUtil.AsNonRenderingEventHandler(HandleClick2)">
    Select me (Avoids Rerender)
</button>

<button @onclick="EventUtil.AsNonRenderingEventHandler<MouseEventArgs>
(HandleClick3)">
    Select me (Avoids Rerender and uses <code>MouseEventArgs</code>)
</button>

@code {
    private DateTime dt = DateTime.Now;

    private void HandleClick1()
    {
        dt = DateTime.Now;

        Logger.LogInformation("This event handler triggers a rerender.");
    }

    private void HandleClick2()
    {
        dt = DateTime.Now;

        Logger.LogInformation("This event handler doesn't trigger a
rerender.");
    }

    private void HandleClick3(MouseEventArgs args)
    {
        dt = DateTime.Now;
        Logger.LogInformation(
            "This event handler doesn't trigger a rerender. " +
            "Mouse coordinates: {ScreenX}:{ScreenY}",
            args.ScreenX, args.ScreenY);
    }
}
```

In addition to implementing the IHandleEvent interface, leveraging the other best practices described in this article can also help reduce unwanted renders after events are handled. For example, overriding ShouldRender in child components of the target component can be used to control rerendering.

Avoid recreating delegates for many repeated elements or components

Blazor's recreation of lambda expression delegates for elements or components in a loop can lead to poor performance.

The following component shown in the event handling article renders a set of buttons. Each button assigns a delegate to its `@onclick` event, which is fine if there aren't many buttons to render.

`EventHandlerExample5.razor`:

razor

```razor
@page "/event-handler-example-5"

<h1>@heading</h1>

@for (var i = 1; i < 4; i++)
{
    var buttonNumber = i;

    <p>
        <button @onclick="@(e => UpdateHeading(e, buttonNumber))">
            Button #@i
        </button>
    </p>
}

@code {
    private string heading = "Select a button to learn its position";

    private void UpdateHeading(MouseEventArgs e, int buttonNumber)
    {
        heading = $"Selected #{buttonNumber} at {e.ClientX}:{e.ClientY}";
    }
}
```

If a large number of buttons are rendered using the preceding approach, rendering speed is adversely impacted leading to a poor user experience. To render a large number of buttons with a callback for click events, the following example uses a collection of button objects that assign each button's `@onclick` delegate to an Action. The following approach doesn't require Blazor to rebuild all of the button delegates each time the buttons are rendered:

`LambdaEventPerformance.razor`:

razor

```razor
@page "/lambda-event-performance"

<h1>@heading</h1>

@foreach (var button in Buttons)
{
    <p>
        <button @key="button.Id" @onclick="button.Action">
            Button #@button.Id
        </button>
    </p>
}

@code {
    private string heading = "Select a button to learn its position";

    private List<Button> Buttons { get; set; } = new();

    protected override void OnInitialized()
    {
        for (var i = 0; i < 100; i++)
        {
            var button = new Button();

            button.Id = Guid.NewGuid().ToString();

            button.Action = (e) =>
            {
                UpdateHeading(button, e);
            };

            Buttons.Add(button);
        }
    }

    private void UpdateHeading(Button button, MouseEventArgs e)
    {
        heading = $"Selected #{button.Id} at {e.ClientX}:{e.ClientY}";
    }

    private class Button
    {
        public string? Id { get; set; }
        public Action<MouseEventArgs> Action { get; set; } = e => { };
    }
}
```

Optimize JavaScript interop speed

Calls between .NET and JavaScript require additional overhead because:

- Calls are asynchronous.
- Parameters and return values are JSON-serialized to provide an easy-to-understand conversion mechanism between .NET and JavaScript types.

Additionally for server-side Blazor apps, these calls are passed across the network.

Avoid excessively fine-grained calls

Since each call involves some overhead, it can be valuable to reduce the number of calls. Consider the following code, which stores a collection of items in the browser's localStorage:

C#

```
private async Task StoreAllInLocalStorage(IEnumerable<TodoItem> items)
{
    foreach (var item in items)
    {
        await JS.InvokeVoidAsync("localStorage.setItem", item.Id,
            JsonSerializer.Serialize(item));
    }
}
```

The preceding example makes a separate JS interop call for each item. Instead, the following approach reduces the JS interop to a single call:

C#

```
private async Task StoreAllInLocalStorage(IEnumerable<TodoItem> items)
{
    await JS.InvokeVoidAsync("storeAllInLocalStorage", items);
}
```

The corresponding JavaScript function stores the whole collection of items on the client:

```JavaScript
function storeAllInLocalStorage(items) {
  items.forEach(item => {
    localStorage.setItem(item.id, JSON.stringify(item));
  });
}
```

For Blazor WebAssembly apps, rolling individual JS interop calls into a single call usually only improves performance significantly if the component makes a large number of JS interop calls.

Consider the use of synchronous calls

Call JavaScript from .NET

This section only applies to client-side components.

JS interop calls are asynchronous, regardless of whether the called code is synchronous or asynchronous. Calls are asynchronous to ensure that components are compatible across server-side and client-side render modes. On the server, all JS interop calls must be asynchronous because they're sent over a network connection.

If you know for certain that your component only runs on WebAssembly, you can choose to make synchronous JS interop calls. This has slightly less overhead than making asynchronous calls and can result in fewer render cycles because there's no intermediate state while awaiting results.

To make a synchronous call from .NET to JavaScript in a client-side component, cast IJSRuntime to IJSInProcessRuntime to make the JS interop call:

```razor
@inject IJSRuntime JS

...

@code {
    protected override void HandleSomeEvent()
    {
        var jsInProcess = (IJSInProcessRuntime)JS;
        var value = jsInProcess.Invoke<string>
("javascriptFunctionIdentifier");
    }
}
```

When working with IJSObjectReference in ASP.NET Core 5.0 or later client-side components, you can use IJSInProcessObjectReference synchronously instead. IJSInProcessObjectReference implements IAsyncDisposable/IDisposable and should be disposed for garbage collection to prevent a memory leak, as the following example demonstrates:

razor

```razor
@inject IJSRuntime JS
@implements IAsyncDisposable

...

@code {
    ...
    private IJSInProcessObjectReference? module;

    protected override async Task OnAfterRenderAsync(bool firstRender)
    {
        if (firstRender)
        {
            module = await JS.InvokeAsync<IJSInProcessObjectReference>
("import",
            "./scripts.js");
        }
    }

    ...

    async ValueTask IAsyncDisposable.DisposeAsync()
    {
        if (module is not null)
        {
            await module.DisposeAsync();
        }
    }
}
```

Call .NET from JavaScript

This section only applies to client-side components.

JS interop calls are asynchronous, regardless of whether the called code is synchronous or asynchronous. Calls are asynchronous to ensure that components are compatible across server-side and client-side render modes. On the server, all JS interop calls must be asynchronous because they're sent over a network connection.

If you know for certain that your component only runs on WebAssembly, you can choose to make synchronous JS interop calls. This has slightly less overhead than making asynchronous calls and can result in fewer render cycles because there's no intermediate state while awaiting results.

To make a synchronous call from JavaScript to .NET in a client-side component, use `DotNet.invokeMethod` instead of `DotNet.invokeMethodAsync`.

Synchronous calls work if:

- The component is only rendered for execution on WebAssembly.
- The called function returns a value synchronously. The function isn't an `async` method and doesn't return a .NET Task or JavaScript Promise.

Use JavaScript `[JSImport]`/`[JSExport]` interop

JavaScript `[JSImport]`/`[JSExport]` interop for Blazor WebAssembly apps offers improved performance and stability over the JS interop API in framework releases prior to ASP.NET Core in .NET 7.

For more information, see JavaScript JSImport/JSExport interop with ASP.NET Core Blazor.

Ahead-of-time (AOT) compilation

Ahead-of-time (AOT) compilation compiles a Blazor app's .NET code directly into native WebAssembly for direct execution by the browser. AOT-compiled apps result in larger apps that take longer to download, but AOT-compiled apps usually provide better runtime performance, especially for apps that execute CPU-intensive tasks. For more information, see ASP.NET Core Blazor WebAssembly build tools and ahead-of-time (AOT) compilation.

Minimize app download size

Runtime relinking

For information on how runtime relinking minimizes an app's download size, see ASP.NET Core Blazor WebAssembly build tools and ahead-of-time (AOT) compilation.

Use `System.Text.Json`

Blazor's JS interop implementation relies on System.Text.Json, which is a high-performance JSON serialization library with low memory allocation. Using System.Text.Json shouldn't result in additional app payload size over adding one or more alternate JSON libraries.

For migration guidance, see How to migrate from Newtonsoft.Json to System.Text.Json.

Intermediate Language (IL) trimming

This section only applies to client-side Blazor scenarios.

Trimming unused assemblies from a Blazor WebAssembly app reduces the app's size by removing unused code in the app's binaries. For more information, see Configure the Trimmer for ASP.NET Core Blazor.

Lazy load assemblies

This section only applies to client-side Blazor scenarios.

Load assemblies at runtime when the assemblies are required by a route. For more information, see Lazy load assemblies in ASP.NET Core Blazor WebAssembly.

Compression

This section only applies to Blazor WebAssembly apps.

When a Blazor WebAssembly app is published, the output is statically compressed during publish to reduce the app's size and remove the overhead for runtime compression. Blazor relies on the server to perform content negotiation and serve statically-compressed files.

After an app is deployed, verify that the app serves compressed files. Inspect the **Network** tab in a browser's developer tools and verify that the files are served with `Content-Encoding: br` (Brotli compression) or `Content-Encoding: gz` (Gzip compression). If the host isn't serving compressed files, follow the instructions in Host and deploy ASP.NET Core Blazor WebAssembly.

Disable unused features

This section only applies to client-side Blazor scenarios.

Blazor WebAssembly's runtime includes the following .NET features that can be disabled for a smaller payload size:

- Blazor WebAssembly carries globalization resources required to display values, such as dates and currency, in the user's culture. If the app doesn't require localization, you may configure the app to support the invariant culture, which is based on the `en-US` culture.

- Adopting invariant globalization only results in using non-localized timezone names. To trim timezone code and data from the app, 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.

Test Razor components in ASP.NET Core Blazor

Article • 03/08/2024

ⓘ **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 Egil Hansen

Testing Razor components is an important aspect of releasing stable and maintainable Blazor apps.

To test a Razor component, the *component under test* (CUT) is:

- Rendered with relevant input for the test.
- Depending on the type of test performed, possibly subject to interaction or modification. For example, event handlers can be triggered, such as an `onclick` event for a button.
- Inspected for expected values. A test passes when one or more inspected values matches the expected values for the test.

Test approaches

Two common approaches for testing Razor components are end-to-end (E2E) testing and unit testing:

- **Unit testing**: Unit tests are written with a unit testing library that provides:
  - Component rendering.
  - Inspection of component output and state.
  - Triggering of event handlers and life cycle methods.
  - Assertions that component behavior is correct.

  bUnit is an example of a library that enables Razor component unit testing.

- **E2E testing**: A test runner runs a Blazor app containing the CUT and automates a browser instance. The testing tool inspects and interacts with the CUT through the browser. Playwright for .NET is an example of an E2E testing framework that can be used with Blazor apps.

In unit testing, only the Razor component (Razor/C#) is involved. External dependencies, such as services and JS interop, must be mocked. In E2E testing, the Razor component and all of its auxiliary infrastructure are part of the test, including CSS, JS, and the DOM and browser APIs.

Test scope describes how extensive the tests are. Test scope typically has an influence on the speed of the tests. Unit tests run on a subset of the app's subsystems and usually execute in milliseconds. E2E tests, which test a broad group of the app's subsystems, can take several seconds to complete.

Unit testing also provides access to the instance of the CUT, allowing for inspection and verification of the component's internal state. This normally isn't possible in E2E testing.

With regard to the component's environment, E2E tests must make sure that the expected environmental state has been reached before verification starts. Otherwise, the result is unpredictable. In unit testing, the rendering of the CUT and the life cycle of the test are more integrated, which improves test stability.

E2E testing involves launching multiple processes, network and disk I/O, and other subsystem activity that often lead to poor test reliability. Unit tests are typically insulated from these sorts of issues.

The following table summarizes the difference between the two testing approaches.

Expand table

Capability	Unit testing	E2E testing
Test scope	Razor component (Razor/C#) only	Razor component (Razor/C#) with CSS/JS
Test execution time	Milliseconds	Seconds
Access to the component instance	Yes	No
Sensitive to the environment	No	Yes
Reliability	More reliable	Less reliable

Choose the most appropriate test approach

Consider the scenario when choosing the type of testing to perform. Some considerations are described in the following table.

Expand table

Scenario	Suggested approach	Remarks
Component without JS interop logic	Unit testing	When there's no dependency on JS interop in a Razor component, the component can be tested without access to JS or the DOM API. In this scenario, there are no disadvantages to choosing unit testing.
Component with simple JS interop logic	Unit testing	It's common for components to query the DOM or trigger animations through JS interop. Unit testing is usually preferred in this scenario, since it's straightforward to mock the JS interaction through the IJSRuntime interface.
Component that depends on complex JS code	Unit testing and separate JS testing	If a component uses JS interop to call a large or complex JS library but the interaction between the Razor component and JS library is simple, then the best approach is likely to treat the component and JS library or code as two separate parts and test each individually. Test the Razor component with a unit testing library, and test the JS with a JS testing library.
Component with logic that depends on JS manipulation of the browser DOM	E2E testing	When a component's functionality is dependent on JS and its manipulation of the DOM, verify both the JS and Blazor code together in an E2E test. This is the approach that the Blazor framework developers have taken with Blazor's browser rendering logic, which has tightly-coupled C# and JS code. The C# and JS code must work together to correctly render Razor components in a browser.
Component that depends on 3rd party class library with hard-to-mock dependencies	E2E testing	When a component's functionality is dependent on a 3rd party class library that has hard-to-mock dependencies, such as JS interop, E2E testing might be the only option to test the component.

Test components with bUnit

There's no official Microsoft testing framework for Blazor, but the community-driven project bUnit provides a convenient way to unit test Razor components.

> **Note**
>
> bUnit is a third-party testing library and isn't supported or maintained by Microsoft.

bUnit works with general-purpose testing frameworks, such as MSTest, NUnit, and xUnit. These testing frameworks make bUnit tests look and feel like regular unit tests. bUnit tests integrated with a general-purpose testing framework are ordinarily executed with:

- Visual Studio's Test Explorer.
- dotnet test CLI command in a command shell.
- An automated DevOps testing pipeline.

> **Note**
>
> Test concepts and test implementations across different test frameworks are similar but not identical. Refer to the test framework's documentation for guidance.

The following demonstrates the structure of a bUnit test on the `Counter` component in an app based on a Blazor project template. The `Counter` component displays and increments a counter based on the user selecting a button in the page:

razor

```razor
@page "/counter"

<h1>Counter</h1>

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

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

@code {
    private int currentCount = 0;

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

The following bUnit test verifies that the CUT's counter is incremented correctly when the button is selected:

```razor
@code {
    [Fact]
    public void CounterShouldIncrementWhenClicked()
    {
        // Arrange
        using var ctx = new TestContext();
        var cut = ctx.Render(@<Counter />);
        var paraElm = cut.Find("p");

        // Act
        cut.Find("button").Click();

        // Assert
        var paraElmText = paraElm.TextContent;
        paraElm.MarkupMatches("Current count: 1");
    }
}
```

Tests can also be written in a C# class file:

```C#
public class CounterTests
{
    [Fact]
    public void CounterShouldIncrementWhenClicked()
    {
        // Arrange
        using var ctx = new TestContext();
        var cut = ctx.RenderComponent<Counter>();
        var paraElm = cut.Find("p");

        // Act
        cut.Find("button").Click();

        // Assert
        var paraElmText = paraElm.TextContent;
        paraElmText.MarkupMatches("Current count: 1");
    }
}
```

The following actions take place at each step of the test:

- *Arrange*: The `Counter` component is rendered using bUnit's `TestContext`. The CUT's paragraph element (`<p>`) is found and assigned to `paraElm`. In Razor syntax, a component can be passed as a RenderFragment to bUnit.

- *Act*: The button's element (`<button>`) is located and selected by calling `Click`, which should increment the counter and update the content of the paragraph tag (`<p>`). The paragraph element text content is obtained by calling `TextContent`.

- *Assert*: `MarkupMatches` is called on the text content to verify that it matches the expected string, which is `Current count: 1`.

Note

The `MarkupMatches` assert method differs from a regular string comparison assertion (for example, `Assert.Equal("Current count: 1", paraElmText);`). `MarkupMatches` performs a semantic comparison of the input and expected HTML markup. A semantic comparison is aware of HTML semantics, meaning things like insignificant whitespace is ignored. This results in more stable tests. For more information, see Customizing the Semantic HTML Comparison.

Additional resources

- Getting Started with bUnit: bUnit instructions include guidance on creating a test project, referencing testing framework packages, and building and running tests.
- How to create maintainable and testable Blazor components - Egil Hansen - NDC Oslo 2022

ASP.NET Core Blazor Progressive Web Application (PWA)

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.

A Blazor Progressive Web Application (PWA) is a single-page application (SPA) that uses modern browser APIs and capabilities to behave like a desktop app.

Blazor WebAssembly is a standards-based client-side web app platform, so it can use any browser API, including PWA APIs required for the following capabilities:

- Working offline and loading instantly, independent of network speed.
- Running in its own app window, not just a browser window.
- Being launched from the host's operating system start menu, dock, or home screen.
- Receiving push notifications from a backend server, even while the user isn't using the app.
- Automatically updating in the background.

The word *progressive* is used to describe these apps because:

- A user might first discover and use the app within their web browser like any other SPA.
- Later, the user progresses to installing it in their OS and enabling push notifications.

Create a project from the PWA template

Visual Studio

When creating a new **Blazor WebAssembly App**, select the **Progressive Web Application** checkbox.

Convert an existing Blazor WebAssembly app into a PWA

Convert an existing Blazor WebAssembly app into a PWA following the guidance in this section.

In the app's project file:

- Add the following `ServiceWorkerAssetsManifest` property to a `PropertyGroup`:

  XML

  ```xml
    ...
    <ServiceWorkerAssetsManifest>service-worker-
  assets.js</ServiceWorkerAssetsManifest>
  </PropertyGroup>
  ```

- Add the following `ServiceWorker` item to an `ItemGroup`:

  XML

  ```xml
  <ItemGroup>
    <ServiceWorker Include="wwwroot\service-worker.js"
      PublishedContent="wwwroot\service-worker.published.js" />
  </ItemGroup>
  ```

To obtain static assets, use **one** of the following approaches:

- Create a separate, new PWA project with the dotnet new command in a command shell:

  .NET CLI

  ```
  dotnet new blazorwasm -o MyBlazorPwa --pwa
  ```

  In the preceding command, the `-o|--output` option creates a new folder for the app named `MyBlazorPwa`.

  If you aren't converting an app for the latest release, pass the `-f|--framework` option. The following example creates the app for ASP.NET Core version 5.0:

  .NET CLI

  ```
  dotnet new blazorwasm -o MyBlazorPwa --pwa -f net5.0
  ```

- Navigate to the ASP.NET Core GitHub repository at the following URL, which links to `main` branch reference source and assets. Select the release that you're working with from the **Switch branches or tags** dropdown list that applies to your app.

  Blazor WebAssembly project template wwwroot folder (dotnet/aspnetcore GitHub repository main branch)

  > 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).

  From the source `wwwroot` folder either in the app that you created or from the reference assets in the `dotnet/aspnetcore` GitHub repository, copy the following files into the app's `wwwroot` folder:
  - `icon-192.png`
  - `icon-512.png`
  - `manifest.webmanifest`
  - `service-worker.js`
  - `service-worker.published.js`

In the app's `wwwroot/index.html` file:

- Add `<link>` elements for the manifest and app icon:

  HTML

  ```html
  <link href="manifest.webmanifest" rel="manifest" />
  <link rel="apple-touch-icon" sizes="512x512" href="icon-512.png" />
  <link rel="apple-touch-icon" sizes="192x192" href="icon-192.png" />
  ```

- Add the following `<script>` tag inside the closing `</body>` tag immediately after the `blazor.webassembly.js` script tag:

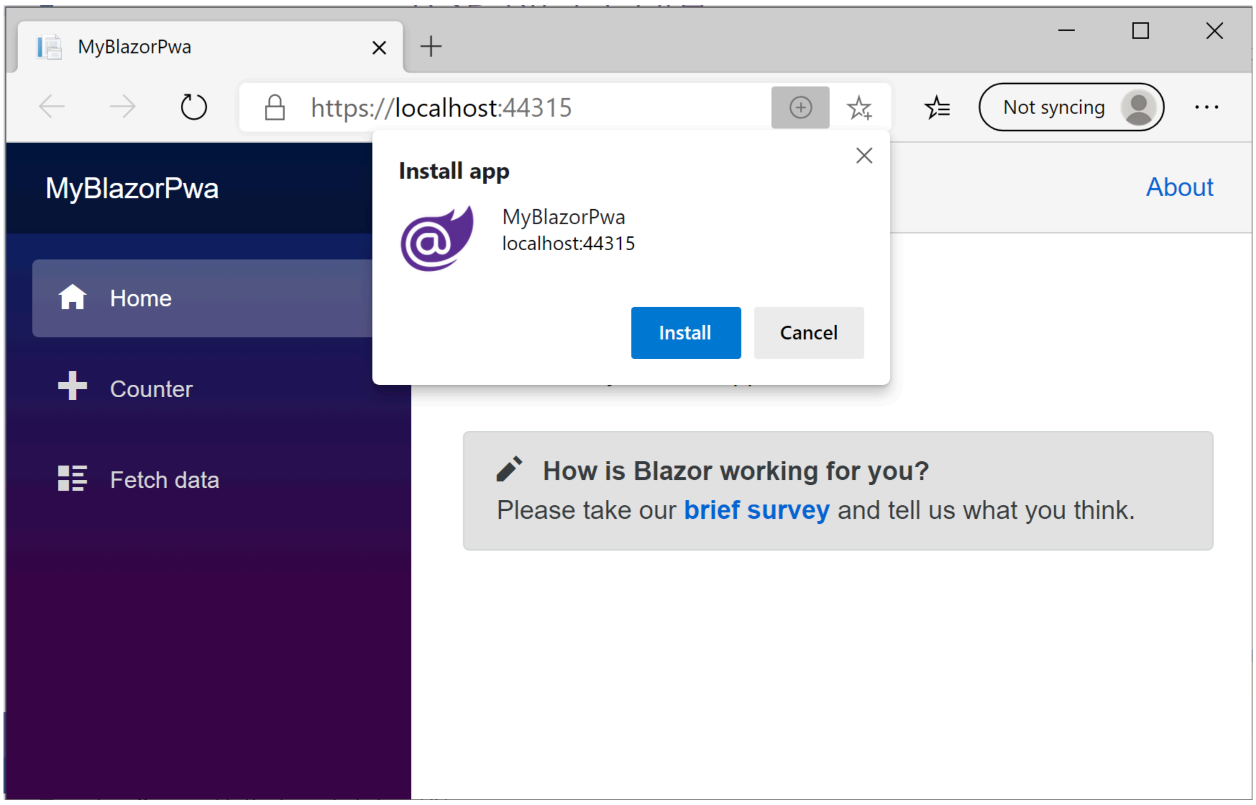
  HTML

  ```html
      ...
      <script>navigator.serviceWorker.register('service-worker.js');
  ```

```
</script>
</body>
```

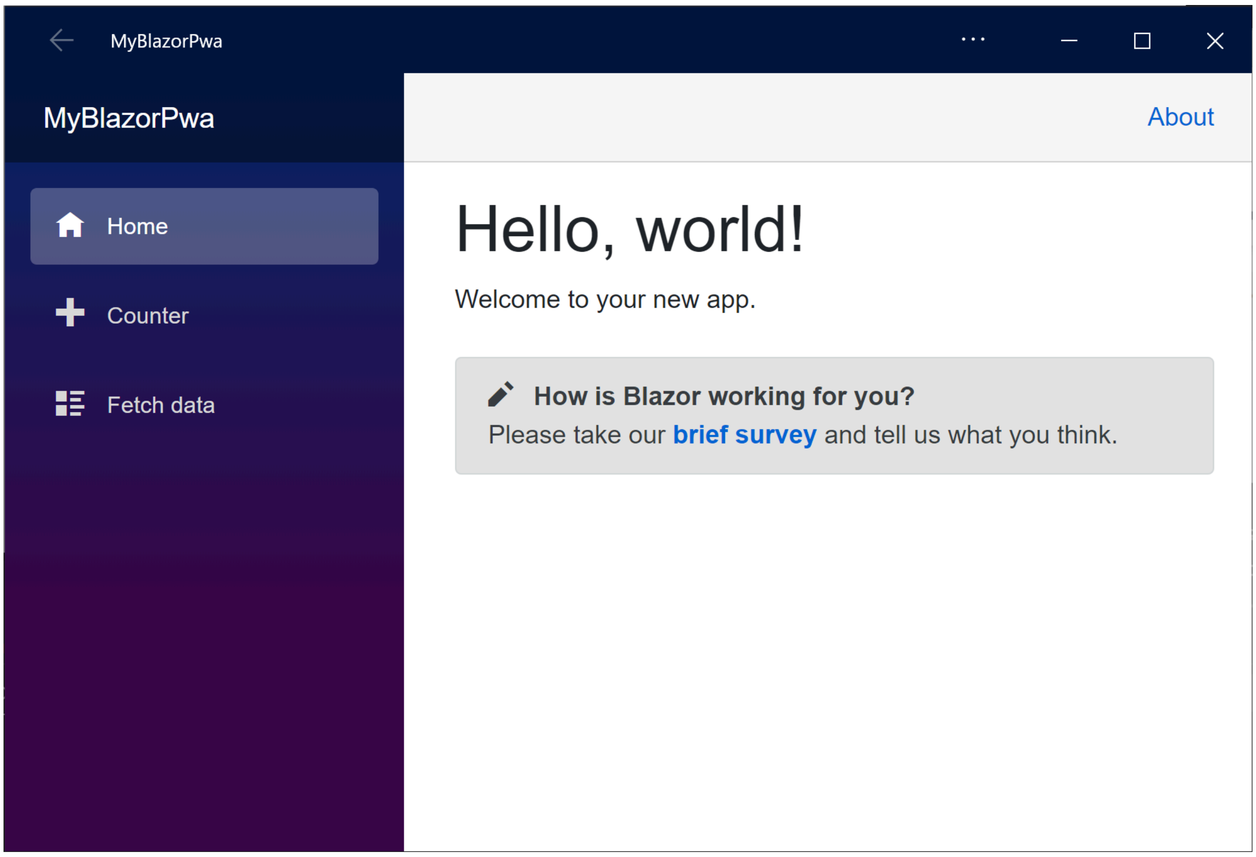
Installation and app manifest

When visiting an app created using the PWA template, users have the option of installing the app into their OS's start menu, dock, or home screen. The way this option is presented depends on the user's browser. When using desktop Chromium-based browsers, such as Edge or Chrome, an **Add** button appears within the URL bar. After the user selects the **Add** button, they receive a confirmation dialog:



On iOS, visitors can install the PWA using Safari's **Share** button and its **Add to Homescreen** option. On Chrome for Android, users should select the **Menu** button in the upper-right corner, followed by **Add to Home screen**.

Once installed, the app appears in its own window without an address bar:



Hello, world!

Welcome to your new app.

How is Blazor working for you?
Please take our brief survey and tell us what you think.

To customize the window's title, color scheme, icon, or other details, see the `manifest.json` file in the project's `wwwroot` directory. The schema of this file is defined by web standards. For more information, see MDN web docs: Web App Manifest.

Offline support

Apps created using the PWA template option have support for running offline. A user must first visit the app while they're online. The browser automatically downloads and caches all of the resources required to operate offline.

Important

Development support would interfere with the usual development cycle of making changes and testing them. Therefore, offline support is only enabled for *published* apps.

Warning

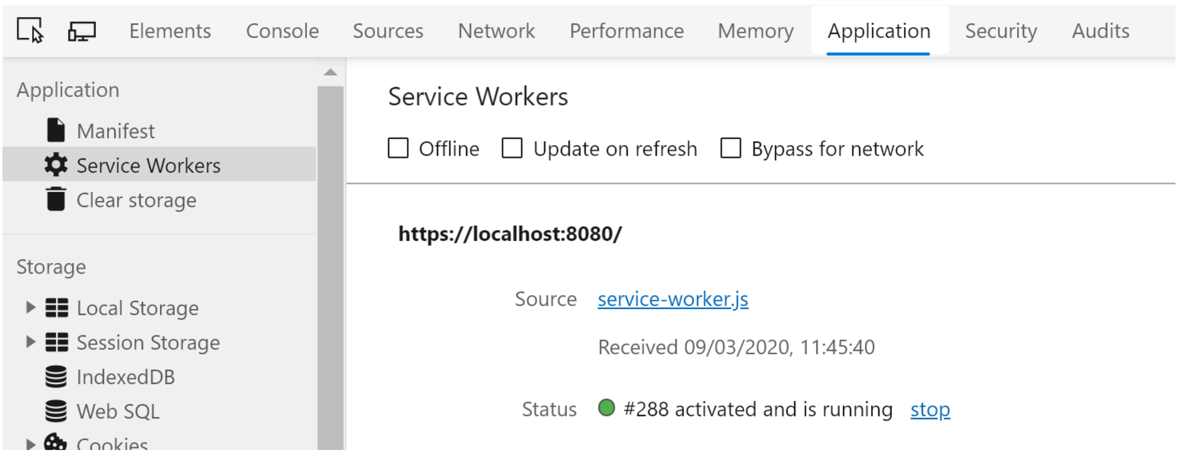
If you intend to distribute an offline-enabled PWA, there are several important warnings and caveats. These scenarios are inherent to offline PWAs and not specific to Blazor. Be sure to read and understand these caveats before making assumptions about how your offline-enabled app works.

To see how offline support works:

1. Publish the app. For more information, see Host and deploy ASP.NET Core Blazor.

2. Deploy the app to a server that supports HTTPS, and access the app in a browser at its secure HTTPS address.

3. Open the browser's dev tools and verify that a *Service Worker* is registered for the host on the **Application** tab:



Service Workers

☐ Offline ☐ Update on refresh ☐ Bypass for network

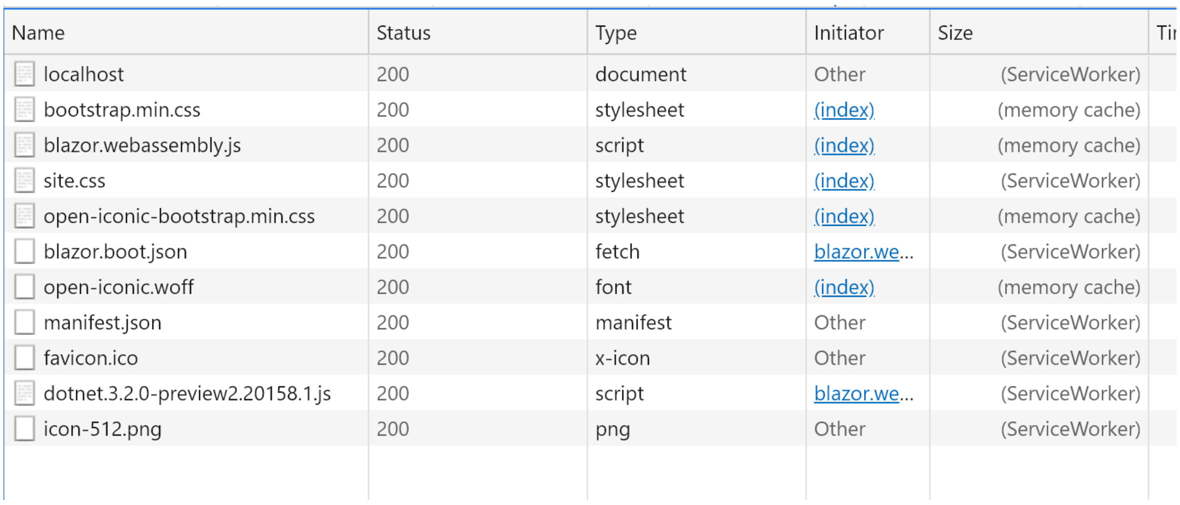
https://localhost:8080/

Source service-worker.js

Received 09/03/2020, 11:45:40

Status ● #288 activated and is running stop

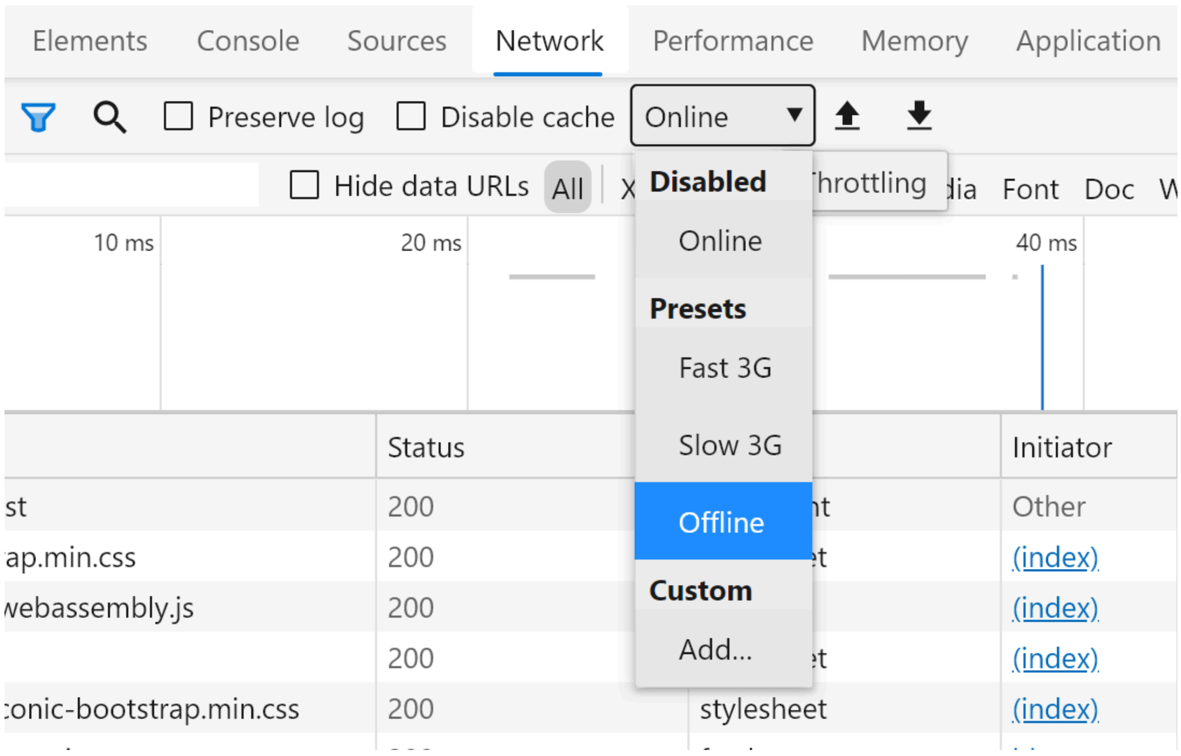
4. Reload the page and examine the **Network** tab. **Service Worker** or **memory cache** are listed as the sources for all of the page's assets:



Name	Status	Type	Initiator	Size	Ti
localhost	200	document	Other	(ServiceWorker)	
bootstrap.min.css	200	stylesheet	(index)	(memory cache)	
blazor.webassembly.js	200	script	(index)	(memory cache)	
site.css	200	stylesheet	(index)	(ServiceWorker)	
open-iconic-bootstrap.min.css	200	stylesheet	(index)	(memory cache)	
blazor.boot.json	200	fetch	blazor.we...	(ServiceWorker)	
open-iconic.woff	200	font	(index)	(memory cache)	
manifest.json	200	manifest	Other	(ServiceWorker)	
favicon.ico	200	x-icon	Other	(ServiceWorker)	
dotnet.3.2.0-preview2.20158.1.js	200	script	blazor.we...	(ServiceWorker)	
icon-512.png	200	png	Other	(ServiceWorker)	

5. To verify that the browser isn't dependent on network access to load the app, either:

   - Shut down the web server and see how the app continues to function normally, which includes page reloads. Likewise, the app continues to function normally when there's a slow network connection.
   - Instruct the browser to simulate offline mode in the **Network** tab:



Offline support using a service worker is a web standard, not specific to Blazor. For more information on service workers, see MDN web docs: Service Worker API. To learn more about common usage patterns for service workers, see Google Web: The Service Worker Lifecycle.

Blazor's PWA template produces two service worker files:

- `wwwroot/service-worker.js`, which is used during development.
- `wwwroot/service-worker.published.js`, which is used after the app is published.

To share logic between the two service worker files, consider the following approach:

- Add a third JavaScript file to hold the common logic.
- Use self.importScripts to load the common logic into both service worker files.

Cache-first fetch strategy

The built-in `service-worker.published.js` service worker resolves requests using a *cache-first* strategy. This means that the service worker prefers to return cached content, regardless of whether the user has network access or newer content is available on the server.

The cache-first strategy is valuable because:

- **It ensures reliability.** Network access isn't a boolean state. A user isn't simply online or offline:

- The user's device may assume it's online, but the network might be so slow as to be impractical to wait for.
    - The network might return invalid results for certain URLs, such as when there's a captive WIFI portal that's currently blocking or redirecting certain requests.

  This is why the browser's `navigator.onLine` API isn't reliable and shouldn't be depended upon.

- **It ensures correctness.** When building a cache of offline resources, the service worker uses content hashing to guarantee it has fetched a complete and self-consistent snapshot of resources at a single instant in time. This cache is then used as an atomic unit. There's no point asking the network for newer resources, since the only versions required are the ones already cached. Anything else risks inconsistency and incompatibility (for example, trying to use versions of .NET assemblies that weren't compiled together).

If you must prevent the browser from fetching `service-worker-assets.js` from its HTTP cache, for example to resolve temporary integrity check failures when deploying a new version of the service worker, update the service worker registration in `wwwroot/index.html` with updateViaCache set to 'none':

HTML

```html
<script>
  navigator.serviceWorker.register('/service-worker.js', {updateViaCache:
'none'});
</script>
```

Background updates

As a mental model, you can think of an offline-first PWA as behaving like a mobile app that can be installed. The app starts up immediately regardless of network connectivity, but the installed app logic comes from a point-in-time snapshot that might not be the latest version.

The Blazor PWA template produces apps that automatically try to update themselves in the background whenever the user visits and has a working network connection. The way this works is as follows:

- During compilation, the project generates a *service worker assets manifest*, which is named `service-worker-assets.js`. The manifest lists all the static resources that the app requires to function offline, such as .NET assemblies, JavaScript files, and

CSS, including their content hashes. The resource list is loaded by the service worker so that it knows which resources to cache.
- Each time the user visits the app, the browser re-requests `service-worker.js` and `service-worker-assets.js` in the background. The files are compared byte-for-byte with the existing installed service worker. If the server returns changed content for either of these files, the service worker attempts to install a new version of itself.
- When installing a new version of itself, the service worker creates a new, separate cache for offline resources and starts populating the cache with resources listed in `service-worker-assets.js`. This logic is implemented in the `onInstall` function inside `service-worker.published.js`.
- The process completes successfully when all of the resources are loaded without error and all content hashes match. If successful, the new service worker enters a *waiting for activation* state. As soon as the user closes the app (no remaining app tabs or windows), the new service worker becomes *active* and is used for subsequent app visits. The old service worker and its cache are deleted.
- If the process doesn't complete successfully, the new service worker instance is discarded. The update process is attempted again on the user's next visit, when hopefully the client has a better network connection that can complete the requests.

Customize this process by editing the service worker logic. None of the preceding behavior is specific to Blazor but is merely the default experience provided by the PWA template option. For more information, see MDN web docs: Service Worker API.

How requests are resolved

As described in the Cache-first fetch strategy section, the default service worker uses a *cache-first* strategy, meaning that it tries to serve cached content when available. If there is no content cached for a certain URL, for example when requesting data from a backend API, the service worker falls back on a regular network request. The network request succeeds if the server is reachable. This logic is implemented inside `onFetch` function within `service-worker.published.js`.

If the app's Razor components rely on requesting data from backend APIs and you want to provide a friendly user experience for failed requests due to network unavailability, implement logic within the app's components. For example, use try/catch around HttpClient requests.

Support server-rendered pages

Consider what happens when the user first navigates to a URL such as `/counter` or any other deep link in the app. In these cases, you don't want to return content cached as `/counter`, but instead need the browser to load the content cached as `/index.html` to start up your Blazor WebAssembly app. These initial requests are known as *navigation* requests, as opposed to:

- `subresource` requests for images, stylesheets, or other files.
- `fetch/XHR` requests for API data.

The default service worker contains special-case logic for navigation requests. The service worker resolves the requests by returning the cached content for `/index.html`, regardless of the requested URL. This logic is implemented in the `onFetch` function inside `service-worker.published.js`.

If your app has certain URLs that must return server-rendered HTML, and not serve `/index.html` from the cache, then you need to edit the logic in your service worker. If all URLs containing `/Identity/` need to be handled as regular online-only requests to the server, then modify `service-worker.published.js` `onFetch` logic. Locate the following code:

JavaScript

```javascript
const shouldServeIndexHtml = event.request.mode === 'navigate';
```

Change the code to the following:

JavaScript

```javascript
const shouldServeIndexHtml = event.request.mode === 'navigate'
  && !event.request.url.includes('/Identity/');
```

If you don't do this, then regardless of network connectivity, the service worker intercepts requests for such URLs and resolves them using `/index.html`.

Add additional endpoints for external authentication providers to the check. In the following example, `/signin-google` for Google authentication is added to the check:

JavaScript

```javascript
const shouldServeIndexHtml = event.request.mode === 'navigate'
  && !event.request.url.includes('/Identity/')
  && !event.request.url.includes('/signin-google');
```

No action is required for the `Development` environment, where content is always fetched from the network.

Control asset caching

If your project defines the `ServiceWorkerAssetsManifest` MSBuild property, Blazor's build tooling generates a service worker assets manifest with the specified name. The default PWA template produces a project file containing the following property:

```xml
<ServiceWorkerAssetsManifest>service-worker-
assets.js</ServiceWorkerAssetsManifest>
```

The file is placed in the `wwwroot` output directory, so the browser can retrieve this file by requesting `/service-worker-assets.js`. To see the contents of this file, open `/bin/Debug/{TARGET FRAMEWORK}/wwwroot/service-worker-assets.js` in a text editor. However, don't edit the file, as it's regenerated on each build.

The manifest lists:

- Any Blazor-managed resources, such as .NET assemblies and the .NET WebAssembly runtime files required to function offline.
- All resources for publishing to the app's `wwwroot` directory, such as images, stylesheets, and JavaScript files, including static web assets supplied by external projects and NuGet packages.

You can control which of these resources are fetched and cached by the service worker by editing the logic in `onInstall` in `service-worker.published.js`. The service worker fetches and caches files matching typical web file name extensions such as `.html`, `.css`, `.js`, and `.wasm`, plus file types specific to Blazor WebAssembly, such as `.pdb` files (all versions) and `.dll` files (ASP.NET Core in .NET 7 or earlier).

To include additional resources that aren't present in the app's `wwwroot` directory, define extra MSBuild `ItemGroup` entries, as shown in the following example:

```xml
<ItemGroup>
  <ServiceWorkerAssetsManifestItem Include="MyDirectory\AnotherFile.json"
    RelativePath="MyDirectory\AnotherFile.json"
AssetUrl="files/AnotherFile.json" />
</ItemGroup>
```

The `AssetUrl` metadata specifies the base-relative URL that the browser should use when fetching the resource to cache. This can be independent of its original source file name on disk.

> **Important**
>
> Adding a `ServiceWorkerAssetsManifestItem` doesn't cause the file to be published in the app's `wwwroot` directory. The publish output must be controlled separately. The `ServiceWorkerAssetsManifestItem` only causes an additional entry to appear in the service worker assets manifest.

Push notifications

Like any other PWA, a Blazor WebAssembly PWA can receive push notifications from a backend server. The server can send push notifications at any time, even when the user isn't actively using the app. For example, push notifications can be sent when a different user performs a relevant action.

The mechanism for sending a push notification is entirely independent of Blazor WebAssembly, since it's implemented by the backend server which can use any technology. If you want to send push notifications from an ASP.NET Core server, consider using a technique similar to the approach taken in the Blazing Pizza workshop.

The mechanism for receiving and displaying a push notification on the client is also independent of Blazor WebAssembly, since it's implemented in the service worker JavaScript file. For an example, see the approach used in the Blazing Pizza workshop.

Caveats for offline PWAs

Not all apps should attempt to support offline use. Offline support adds significant complexity, while not always being relevant for the use cases required.

Offline support is usually relevant only:

- If the primary data store is local to the browser. For example, the approach is relevant in an app with a UI for an IoT device that stores data in `localStorage` or IndexedDB.
- If the app performs a significant amount of work to fetch and cache the backend API data relevant to each user so that they can navigate through the data offline. If

the app must support editing, a system for tracking changes and synchronizing data with the backend must be built.
- If the goal is to guarantee that the app loads immediately regardless of network conditions. Implement a suitable user experience around backend API requests to show the progress of requests and behave gracefully when requests fail due to network unavailability.

Additionally, offline-capable PWAs must deal with a range of additional complications. Developers should carefully familiarize themselves with the caveats in the following sections.

Offline support only when published

During development you typically want to see each change reflected immediately in the browser without going through a background update process. Therefore, Blazor's PWA template enables offline support only when published.

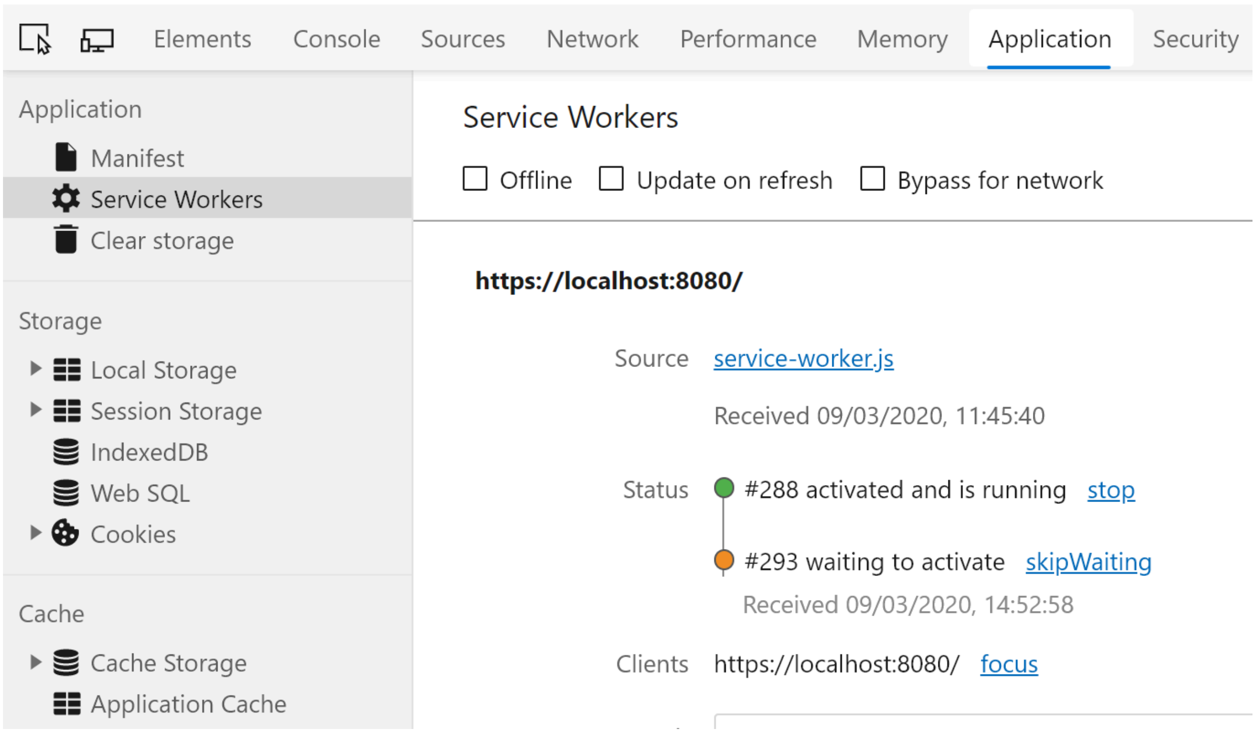
When building an offline-capable app, it's not enough to test the app in the `Development` environment. You must test the app in its published state to understand how it responds to different network conditions.

Update completion after user navigation away from app

Updates don't complete until the user has navigated away from the app in all tabs. As explained in the Background updates section, after you deploy an update to the app, the browser fetches the updated service worker files to begin the update process.

What surprises many developers is that, even when this update completes, it doesn't take effect until the user has navigated away in all tabs. It isn't sufficient to refresh the tab displaying the app, even if it's the only tab displaying the app. Until your app is completely closed, the new service worker remains in the *waiting to activate* status. This isn't specific to Blazor, but rather is a standard web platform behavior.

This commonly troubles developers who are trying to test updates to their service worker or offline cached resources. If you check in the browser's developer tools, you may see something like the following:



https://localhost:8080/

Source service-worker.js
Received 09/03/2020, 11:45:40

Status #288 activated and is running stop
#293 waiting to activate skipWaiting
Received 09/03/2020, 14:52:58

Clients https://localhost:8080/ focus

For as long as the list of "clients," which are tabs or windows displaying your app, is nonempty, the worker continues waiting. The reason service workers do this is to guarantee consistency. Consistency means that all resources are fetched from the same atomic cache.

When testing changes, you may find it convenient to select the "skipWaiting" link as shown in the preceding screenshot, then reload the page. You can automate this for all users by coding your service worker to skip the "waiting" phase and immediately activate on update. If you skip the waiting phase, you're giving up the guarantee that resources are always fetched consistently from the same cache instance.

Users may run any historical version of the app

Web developers habitually expect that users only run the latest deployed version of their web app, since that's normal within the traditional web distribution model. However, an offline-first PWA is more akin to a native mobile app, where users aren't necessarily running the latest version.

As explained in the Background updates section, after you deploy an update to your app, each existing user continues to use a previous version for at least one further visit because the update occurs in the background and isn't activated until the user thereafter navigates away. Plus, the previous version being used isn't necessarily the previous one you deployed. The previous version can be *any* historical version, depending on when the user last completed an update.

This can be an issue if the frontend and backend parts of your app require agreement about the schema for API requests. You must not deploy backward-incompatible API schema changes until you can be sure that all users have upgraded. Alternatively, block users from using incompatible older versions of the app. This scenario requirement is the same as for native mobile apps. If you deploy a breaking change in server APIs, the client app is broken for users who haven't yet updated.

If possible, don't deploy breaking changes to your backend APIs. If you must do so, consider using standard Service Worker APIs such as ServiceWorkerRegistration to determine whether the app is up-to-date, and if not, to prevent usage.

Interference with server-rendered pages

As described in the Support server-rendered pages section, if you want to bypass the service worker's behavior of returning `/index.html` contents for all navigation requests, edit the logic in your service worker.

All service worker asset manifest contents are cached

As described in the Control asset caching section, the file `service-worker-assets.js` is generated during build and lists all assets the service worker should fetch and cache.

Since this list includes everything emitted to `wwwroot`, including content supplied by external packages and projects, you must be careful not to put too much content there. If the `wwwroot` directory contains millions of images, the service worker tries to fetch and cache them all, consuming excessive bandwidth and most likely not completing successfully.

Implement arbitrary logic to control which subset of the manifest's contents should be fetched and cached by editing the `onInstall` function in `service-worker.published.js`.

Interaction with authentication

The PWA template can be used in conjunction with authentication. An offline-capable PWA can also support authentication when the user has initial network connectivity.

When a user doesn't have network connectivity, they can't authenticate or obtain access tokens. Attempting to visit the login page without network access results in a "network error" message. You must design a UI flow that allows the user perform useful tasks while offline without attempting to authenticate the user or obtain access tokens. Alternatively, you can design the app to gracefully fail when the network isn't available. If the app can't be designed to handle these scenarios, you might not want to enable offline support.

When an app that's designed for online and offline use is online again:

- The app might need to provision a new access token.
- The app must detect if a different user is signed into the service so that it can apply operations to the user's account that were made while they were offline.

To create an offline PWA app that interacts with authentication:

- Replace the AccountClaimsPrincipalFactory<TAccount> with a factory that stores the last signed-in user and uses the stored user when the app is offline.
- Enqueue operations while the app is offline and apply them when the app returns online.
- During sign out, clear the stored user.

The CarChecker sample app demonstrates the preceding approaches. See the following parts of the app:

- `OfflineAccountClaimsPrincipalFactory` (`Client/Data/OfflineAccountClaimsPrincipalFactory.cs`)
- `LocalVehiclesStore` (`Client/Data/LocalVehiclesStore.cs`)
- `LoginStatus` component (`Client/Shared/LoginStatus.razor`)

Additional resources

- Troubleshoot integrity PowerShell script
- Client-side SignalR cross-origin negotiation for authentication

Host and deploy ASP.NET Core Blazor

Article • 10/08/2024

ⓘ **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 host and deploy Blazor apps.

Publish the app

Apps are published for deployment in Release configuration.

Visual Studio

1. Select the **Publish {APPLICATION}** command from the **Build** menu, where the `{APPLICATION}` placeholder the app's name.
2. Select the *publish target*. To publish locally, select **Folder**.
3. Accept the default location in the **Choose a folder** field or specify a different location. Select the **`Publish`** button.

Publishing the app triggers a restore of the project's dependencies and builds the project before creating the assets for deployment. As part of the build process, unused methods and assemblies are removed to reduce app download size and load times.

Publish locations:

- Blazor Web App: The app is published into the `/bin/Release/{TARGET FRAMEWORK}/publish` folder. Deploy the contents of the `publish` folder to the host.
- Blazor WebAssembly: The app is published into the `bin\Release\net8.0\browser-wasm\publish\` folder. To deploy the app as a static site, copy the contents of the `wwwroot` folder to the static site host.

The `{TARGET FRAMEWORK}` in the preceding paths is the target framework (for example, `net8.0`).

IIS

To host a Blazor app in IIS, see the following resources:

- IIS hosting
  - Publish an ASP.NET Core app to IIS
  - Host ASP.NET Core on Windows with IIS
- Host and deploy ASP.NET Core server-side Blazor apps: Server apps running on IIS, including IIS with Azure Virtual Machines (VMs) running Windows OS and Azure App Service.
- Host and deploy ASP.NET Core Blazor WebAssembly: Includes additional guidance for Blazor WebAssembly apps hosted on IIS, including static site hosting, custom `web.config` files, URL rewriting, sub-apps, compression, and Azure Storage static file hosting.
- IIS sub-application hosting
  - Follow the guidance in the App base path section for the Blazor app prior to publishing the app. The examples use an app base path of `/CoolApp` and show how to obtain the base path from app settings or other configuration providers.
  - Follow the sub-application configuration guidance in Advanced configuration. The sub-app's folder path under the root site becomes the virtual path of the sub-app. For an app base path of `/CoolApp`, the Blazor app is placed in a folder named `CoolApp` under the root site and the sub-app takes on a virtual path of `/CoolApp`.

Sharing an app pool among ASP.NET Core apps isn't supported, including for Blazor apps. Use one app pool per app when hosting with IIS, and avoid the use of IIS's virtual directories for hosting multiple apps.

App base path

The *app base path* is the app's root URL path. Successful routing in Blazor apps requires framework configuration for any root URL path that isn't at the default app base path `/`.

Consider the following ASP.NET Core app and Blazor sub-app:

- The ASP.NET Core app is named `MyApp`:
  - The app physically resides at `d:/MyApp`.
  - Requests are received at `https://www.contoso.com/{MYAPP RESOURCE}`.
- A Blazor app named `CoolApp` is a sub-app of `MyApp`:
  - The sub-app physically resides at `d:/MyApp/CoolApp`.
  - Requests are received at `https://www.contoso.com/CoolApp/{COOLAPP RESOURCE}`.

Without specifying additional configuration for `CoolApp`, the sub-app in this scenario has no knowledge of where it resides on the server. For example, the app can't construct correct relative URLs to its resources without knowing that it resides at the relative URL path `/CoolApp/`. This scenario also applies in various hosting and reverse proxy scenarios when an app isn't hosted at a root URL path.

Background

An anchor tag's destination (href) can be composed with either of two endpoints:

- Absolute locations that include a scheme (defaults to the page's scheme if omitted), host, port, and path or just a forward slash (`/`) followed by the path.

  Examples: `https://example.com/a/b/c` or `/a/b/c`

- Relative locations that contain just a path and do not start with a forward slash (`/`). These are resolved relative to the current document URL or the `<base>` tag's value, if specified.

  Example: `a/b/c`

The presence of a trailing slash (`/`) in a configured app base path is significant to compute the base path for URLs of the app. For example, `https://example.com/a` has a base path of `https://example.com/`, while `https://example.com/a/` with a trailing slash has a base path of `https://example.com/a`.

There are three sources of links that pertain to Blazor in ASP.NET Core apps:

- URLs in Razor components (`.razor`) are typically relative.
- URLs in scripts, such as the Blazor scripts (`blazor.*.js`), are relative to the document.

If you're rendering a Blazor app from different documents (for example, `/Admin/B/C/` and `/Admin/D/E/`), you must take the app base path into account, or the base path is different when the app renders in each document and the resources are fetched from the wrong URLs.

There are two approaches to deal with the challenge of resolving relative links correctly:

- Map the resources dynamically using the document they were rendered on as the root.
- Set a consistent base path for the document and map the resources under that base path.

The first option is more complicated and isn't the most typical approach, as it makes navigation different for each document. Consider the following example for rendering a page `/Something/Else`:

- Rendered under `/Admin/B/C/`, the page is rendered with a path of `/Admin/B/C/Something/Else`.
- Rendered under `/Admin/D/E/`, the page is rendered ***at the same path*** of `/Admin/B/C/Something/Else`.

Under the first approach, routing offers IDynamicEndpointMetadata and MatcherPolicy, which in combination can be the basis for implementing a completely dynamic solution that determines at runtime about how requests are routed.

For the second option, which is the usual approach taken, the app sets the base path in the document and maps the server endpoints to paths under the base. The following guidance adopts this approach.

Server-side Blazor

Map the SignalR hub of a server-side Blazor app by passing the path to MapBlazorHub in the `Program` file:

```C#
app.MapBlazorHub("base/path");
```

The benefit of using MapBlazorHub is that you can map patterns, such as `"{tenant}"` and not just concrete paths.

You can also map the SignalR hub when the app is in a virtual folder with a branched middleware pipeline. In the following example, requests to `/base/path/` are handled by Blazor's SignalR hub:

```C#
app.Map("/base/path/", subapp => {
    subapp.UsePathBase("/base/path/");
    subapp.UseRouting();
    subapp.UseEndpoints(endpoints => endpoints.MapBlazorHub());
});
```

Configure the `<base>` tag, per the guidance in the Configure the app base path section.

Standalone Blazor WebAssembly

In a standalone Blazor WebAssembly app, only the `<base>` tag is configured, per the guidance in the Configure the app base path section.

Configure the app base path

To provide configuration for the Blazor app's base path of `https://www.contoso.com/CoolApp/`, set the app base path (<base>), which is also called the relative root path.

By configuring the app base path, a component that isn't in the root directory can construct URLs relative to the app's root path. Components at different levels of the directory structure can build links to other resources at locations throughout the app. The app base path is also used to intercept selected hyperlinks where the `href` target of the link is within the app base path URI space. The Router component handles the internal navigation.

Place the the `<base>` tag in `<head>` markup (location of <head> content) before any elements with attribute values that are URLs, such as the `href` attributes of `<link>` elements.

In many hosting scenarios, the relative URL path to the app is the root of the app. In these default cases, the app's relative URL base path is `/` configured as `<base href="/" />` in <head> content.

Note

In some hosting scenarios, such as GitHub Pages and IIS sub-apps, the app base path must be set to the server's relative URL path of the app.

- In a server-side Blazor app, use ***either*** of the following approaches:
  - Option 1: Use the `<base>` tag to set the app's base path (location of <head> content):

    HTML

    ```html
    <base href="/CoolApp/">
    ```

    The trailing slash is required.

- Option 2: Call UsePathBase ***first*** in the app's request processing pipeline (`Program.cs`) immediately after the WebApplicationBuilder is built (`builder.Build()`) to configure the base path for any following middleware that interacts with the request path:

  C#

  ```csharp
  app.UsePathBase("/CoolApp");
  ```

  Calling UsePathBase is recommended when you also wish to run the Blazor Server app locally. For example, supply the launch URL in `Properties/launchSettings.json`:

  XML

  ```xml
  "launchUrl": "https://localhost:{PORT}/CoolApp",
  ```

  The `{PORT}` placeholder in the preceding example is the port that matches the secure port in the `applicationUrl` configuration path. The following example shows the full launch profile for an app at port 7279:

  XML

  ```xml
  "BlazorSample": {
    "commandName": "Project",
    "dotnetRunMessages": true,
    "launchBrowser": true,
    "applicationUrl": "https://localhost:7279;http://localhost:5279",
    "launchUrl": "https://localhost:7279/CoolApp",
    "environmentVariables": {
      "ASPNETCORE_ENVIRONMENT": "Development"
  }
  ```

  For more information on the `launchSettings.json` file, see Use multiple environments in ASP.NET Core. For additional information on Blazor app base paths and hosting, see <base href="/" /> or base-tag alternative for Blazor MVC integration (dotnet/aspnetcore #43191).

- Standalone Blazor WebAssembly (`wwwroot/index.html`):

  HTML

  ```html
  <base href="/CoolApp/">
  ```

The trailing slash is required.

> ⓘ Note
>
> When using WebApplication (see Migrate from ASP.NET Core 5.0 to 6.0), app.UseRouting must be called after UsePathBase so that the Routing Middleware can observe the modified path before matching routes. Otherwise, routes are matched before the path is rewritten by UsePathBase as described in the Middleware Ordering and Routing articles.

Don't prefix links throughout the app with a forward slash. Either avoid the use of a path segment separator or use dot-slash (`./`) relative path notation:

- ❌ Incorrect: `<a href="/account">`
- ✓ Correct: `<a href="account">`
- ✓ Correct: `<a href="./account">`

In Blazor WebAssembly web API requests with the HttpClient service, confirm that JSON helpers (HttpClientJsonExtensions) don't prefix URLs with a forward slash (`/`):

- ❌ Incorrect: `var rsp = await client.GetFromJsonAsync("/api/Account");`
- ✓ Correct: `var rsp = await client.GetFromJsonAsync("api/Account");`

Don't prefix Navigation Manager relative links with a forward slash. Either avoid the use of a path segment separator or use dot-slash (`./`) relative path notation (`Navigation` is an injected NavigationManager):

- ❌ Incorrect: `Navigation.NavigateTo("/other");`
- ✓ Correct: `Navigation.NavigateTo("other");`
- ✓ Correct: `Navigation.NavigateTo("./other");`

In typical configurations for Azure/IIS hosting, additional configuration usually isn't required. In some non-IIS hosting and reverse proxy hosting scenarios, additional Static File Middleware configuration might be required:

- To serve static files correctly (for example, `app.UseStaticFiles("/CoolApp");`).
- To serve the Blazor script (`_framework/blazor.*.js`). For more information, see ASP.NET Core Blazor static files.

For a Blazor WebAssembly app with a non-root relative URL path (for example, `<base href="/CoolApp/">`), the app fails to find its resources *when run locally*. To overcome this problem during local development and testing, you can supply a *path base* argument that matches the `href` value of the `<base>` tag at runtime. **Don't include a trailing slash.** To pass the path base argument when running the app locally, execute the `dotnet watch` (or `dotnet run`) command from the app's directory with the `--pathbase` option:

.NET CLI

```
dotnet watch --pathbase=/{RELATIVE URL PATH (no trailing slash)}
```

For a Blazor WebAssembly app with a relative URL path of `/CoolApp/` (`<base href="/CoolApp/">`), the command is:

.NET CLI

```
dotnet watch --pathbase=/CoolApp
```

If you prefer to configure the app's launch profile to specify the `pathbase` automatically instead of manually with `dotnet watch` (or `dotnet run`), set the `commandLineArgs` property in `Properties/launchSettings.json`. The following also configures the launch URL (`launchUrl`):

JSON

```
"commandLineArgs": "--pathbase=/{RELATIVE URL PATH (no trailing slash)}",
"launchUrl": "{RELATIVE URL PATH (no trailing slash)}",
```

Using `CoolApp` as the example:

JSON

```
"commandLineArgs": "--pathbase=/CoolApp",
"launchUrl": "CoolApp",
```

Using either `dotnet watch` (or `dotnet run`) with the `--pathbase` option or a launch profile configuration that sets the base path, the Blazor WebAssembly app responds locally at `http://localhost:port/CoolApp`.

For more information on the `launchSettings.json` file, see Use multiple environments in ASP.NET Core. For additional information on Blazor app base paths and hosting, see <base href="/" /> or base-tag alternative for Blazor MVC integration (dotnet/aspnetcore #43191).

Obtain the app base path from configuration

The following guidance explains how to obtain the path for the `<base>` tag from an app settings file for different environments.

Add the app settings file to the app. The following example is for the `Staging` environment (`appsettings.Staging.json`):

JSON

```json
{
  "AppBasePath": "staging/"
}
```

In a server-side Blazor app, load the base path from configuration in `<head>` content:

razor

```razor
@inject IConfiguration Config

...

<head>
    ...
    <base href="/@(Config.GetValue<string>("AppBasePath"))" />
    ...
</head>
```

Alternatively, a server-side app can obtain the value from configuration for UsePathBase. Place the following code ***first*** in the app's request processing pipeline (`Program.cs`) immediately after the WebApplicationBuilder is built (`builder.Build()`). The following example uses the configuration key `AppBasePath`:

C#

```csharp
app.UsePathBase($"/{app.Configuration.GetValue<string>("AppBasePath")}");
```

In a client-side Blazor WebAssembly app:

- Remove the `<base>` tag from `wwwroot/index.html`:

  diff

  ```diff
  - <base href="..." />
  ```

- Supply the app base path via a HeadContent component in the `App` component (`App.razor`):

```razor
@inject IConfiguration Config

...

<HeadContent>
    <base href="/@(Config.GetValue<string>("AppBasePath"))" />
</HeadContent>
```

If there's no configuration value to load, for example in non-staging environments, the preceding `href` resolves to the root path `/`.

The examples in this section focus on supplying the app base path from app settings, but the approach of reading the path from IConfiguration is valid for any configuration provider. For more information, see the following resources:

- ASP.NET Core Blazor configuration
- Configuration in ASP.NET Core

Blazor Server `MapFallbackToPage` configuration

This section only applies to Blazor Server apps. MapFallbackToPage isn't supported in Blazor Web Apps and Blazor WebAssembly apps.

In scenarios where an app requires a separate area with custom resources and Razor components:

- Create a folder within the app's `Pages` folder to hold the resources. For example, an administrator section of an app is created in a new folder named `Admin` (`Pages/Admin`).

- Create a root page (`_Host.cshtml`) for the area. For example, create a `Pages/Admin/_Host.cshtml` file from the app's main root page (`Pages/_Host.cshtml`). Don't provide an `@page` directive in the Admin `_Host` page.

- Add a layout to the area's folder (for example, `Pages/Admin/_Layout.razor`). In the layout for the separate area, set the `<base>` tag `href` to match the area's folder (for example, `<base href="/Admin/" />`). For demonstration purposes, add `~/` to the static resources in the page. For example:

    - `~/css/bootstrap/bootstrap.min.css`
    - `~/css/site.css`
    - `~/BlazorSample.styles.css` (the example app's namespace is `BlazorSample`)
    - `~/_framework/blazor.server.js` (Blazor script)

- If the area should have its own static asset folder, add the folder and specify its location to Static File Middleware in `Program.cs` (for example, `app.UseStaticFiles("/Admin/wwwroot")`).

- Razor components are added to the area's folder. At a minimum, add an `Index` component to the area folder with the correct `@page` directive for the area. For example, add a `Pages/Admin/Index.razor` file based on the app's default `Pages/Index.razor` file. Indicate the Admin area as the route template at the top of the file (`@page "/admin"`). Add additional components as needed. For example, `Pages/Admin/Component1.razor` with an `@page` directive and route template of `@page "/admin/component1`.

- In `Program.cs`, call MapFallbackToPage for the area's request path immediately before the fallback root page path to the `_Host` page:

```C#
...
app.UseRouting();

app.MapBlazorHub();
app.MapFallbackToPage("~/Admin/{*clientroutes:nonfile}",
"/Admin/_Host");
app.MapFallbackToPage("/_Host");

app.Run();
```

Deployment

For deployment guidance, see the following topics:

- Host and deploy ASP.NET Core Blazor WebAssembly
- Host and deploy ASP.NET Core server-side Blazor apps

Host and deploy server-side Blazor apps

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 host and deploy server-side Blazor apps (Blazor Web Apps and Blazor Server apps) using ASP.NET Core.

Host configuration values

Server-side Blazor apps can accept Generic Host configuration values.

Deployment

Using a server-side hosting model, Blazor is executed on the server from within an ASP.NET Core app. UI updates, event handling, and JavaScript calls are handled over a SignalR connection.

A web server capable of hosting an ASP.NET Core app is required. Visual Studio includes a server-side app project template. For more information on Blazor project templates, see ASP.NET Core Blazor project structure.

Publish an app in Release configuration and deploy the contents of the `bin/Release/{TARGET FRAMEWORK}/publish` folder, where the `{TARGET FRAMEWORK}` placeholder is the target framework.

Scalability

When considering the scalability of a single server (scale up), the memory available to an app is likely the first resource that the app exhausts as user demands increase. The available memory on the server affects the:

- Number of active circuits that a server can support.

- UI latency on the client.

For guidance on building secure and scalable server-side Blazor apps, see the following resources:

- Threat mitigation guidance for ASP.NET Core Blazor static server-side rendering
- Threat mitigation guidance for ASP.NET Core Blazor interactive server-side rendering

Each circuit uses approximately 250 KB of memory for a minimal *Hello World*-style app. The size of a circuit depends on the app's code and the state maintenance requirements associated with each component. We recommend that you measure resource demands during development for your app and infrastructure, but the following baseline can be a starting point in planning your deployment target: If you expect your app to support 5,000 concurrent users, consider budgeting at least 1.3 GB of server memory to the app (or ~273 KB per user).

SignalR configuration

SignalR's hosting and scaling conditions apply to Blazor apps that use SignalR.

For more information on SignalR in Blazor apps, including configuration guidance, see ASP.NET Core Blazor SignalR guidance.

Transports

Blazor works best when using WebSockets as the SignalR transport due to lower latency, better reliability, and improved security. Long Polling is used by SignalR when WebSockets isn't available or when the app is explicitly configured to use Long Polling.

A console warning appears if Long Polling is utilized:

> Failed to connect via WebSockets, using the Long Polling fallback transport. This may be due to a VPN or proxy blocking the connection.

Global deployment and connection failures

Recommendations for global deployments to geographical data centers:

- Deploy the app to the regions where most of the users reside.
- Take into consideration the increased latency for traffic across continents. To control the appearance of the reconnection UI, see ASP.NET Core Blazor SignalR

guidance.
- Consider using the Azure SignalR Service.

Azure App Service

Hosting on Azure App Service requires configuration for WebSockets and session affinity, also called Application Request Routing (ARR) affinity.

> **Note**
>
> A Blazor app on Azure App Service doesn't require Azure SignalR Service.

Enable the following for the app's registration in Azure App Service:

- WebSockets to allow the WebSockets transport to function. The default setting is **Off**.
- Session affinity to route requests from a user back to the same App Service instance. The default setting is **On**.

1. In the Azure portal, navigate to the web app in **App Services**.
2. Open **Settings** > **Configuration**.
3. Set **Web sockets** to **On**.
4. Verify that **Session affinity** is set to **On**.

Azure SignalR Service

The optional Azure SignalR Service works in conjunction with the app's SignalR hub for scaling up a server-side app to a large number of concurrent connections. In addition, the service's global reach and high-performance data centers significantly aid in reducing latency due to geography.

The service isn't required for Blazor apps hosted in Azure App Service or Azure Container Apps but can be helpful in other hosting environments:

- To facilitate connection scale out.
- Handle global distribution.

> **Note**
>
> Stateful reconnect (WithStatefulReconnect) was released with .NET 8 but isn't currently supported for the Azure SignalR Service. For more information, see

Stateful Reconnect Support? (Azure/azure-signalr #1878).

In the event that the app uses Long Polling or falls back to Long Polling instead of WebSockets, you may need to configure the maximum poll interval (`MaxPollIntervalInSeconds`, default: 5 seconds, limit: 1-300 seconds), which defines the maximum poll interval allowed for Long Polling connections in the Azure SignalR Service. If the next poll request doesn't arrive within the maximum poll interval, the service closes the client connection.

For guidance on how to add the service as a dependency to a production deployment, see Publish an ASP.NET Core SignalR app to Azure App Service.

For more information, see:

- Azure SignalR Service
- What is Azure SignalR Service?
- ASP.NET Core SignalR production hosting and scaling
- Publish an ASP.NET Core SignalR app to Azure App Service

Azure Container Apps

For a deeper exploration of scaling server-side Blazor apps on the Azure Container Apps service, see Scaling ASP.NET Core Apps on Azure. The tutorial explains how to create and integrate the services required to host apps on Azure Container Apps. Basic steps are also provided in this section.

1. Configure Azure Container Apps service for session affinity by following the guidance in Session Affinity in Azure Container Apps (Azure documentation).

2. The ASP.NET Core Data Protection (DP) service must be configured to persist keys in a centralized location that all container instances can access. The keys can be stored in Azure Blob Storage and protected with Azure Key Vault. The DP service uses the keys to deserialize Razor components. To configure the DP service to use Azure Blob Storage and Azure Key Vault, reference the following NuGet packages:

   - Azure.Identity: Provides classes to work with the Azure identity and access management services.
   - Microsoft.Extensions.Azure: Provides helpful extension methods to perform core Azure configurations.
   - Azure.Extensions.AspNetCore.DataProtection.Blobs: Allows storing ASP.NET Core Data Protection keys in Azure Blob Storage so that keys can be shared across several instances of a web app.

- Azure.Extensions.AspNetCore.DataProtection.Keys: Enables protecting keys at rest using the Azure Key Vault Key Encryption/Wrapping feature.

> **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.

3. Update `Program.cs` with the following highlighted code:

C#

```csharp
using Azure.Identity;
using Microsoft.AspNetCore.DataProtection;
using Microsoft.Extensions.Azure;
var builder = WebApplication.CreateBuilder(args);
var BlobStorageUri = builder.Configuration["AzureURIs:BlobStorage"];
var KeyVaultURI = builder.Configuration["AzureURIs:KeyVault"];

builder.Services.AddRazorPages();
builder.Services.AddHttpClient();
builder.Services.AddServerSideBlazor();

builder.Services.AddAzureClientsCore();

builder.Services.AddDataProtection()
                .PersistKeysToAzureBlobStorage(new Uri(BlobStorageUri),
                                                new
DefaultAzureCredential())
                .ProtectKeysWithAzureKeyVault(new Uri(KeyVaultURI),
                                                new
DefaultAzureCredential());
var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.MapRazorPages();
app.Run();
```

The preceding changes allow the app to manage the DP service using a centralized, scalable architecture. DefaultAzureCredential discovers the container app managed identity after the code is deployed to Azure and uses it to connect to blob storage and the app's key vault.

4. To create the container app managed identity and grant it access to blob storage and a key vault, complete the following steps:
   a. In the Azure Portal, navigate to the overview page of the container app.
   b. Select **Service Connector** from the left navigation.
   c. Select **+ Create** from the top navigation.
   d. In the **Create connection** flyout menu, enter the following values:

      - **Container**: Select the container app you created to host your app.
      - **Service type**: Select **Blob Storage**.
      - **Subscription**: Select the subscription that owns the container app.
      - **Connection name**: Enter a name of `scalablerazorstorage`.
      - **Client type**: Select **.NET** and then select **Next**.

   e. Select **System assigned managed identity** and select **Next**.
   f. Use the default network settings and select **Next**.
   g. After Azure validates the settings, select **Create**.

   Repeat the preceding settings for the key vault. Select the appropriate key vault service and key in the **Basics** tab.

IIS

When using IIS, enable:

- WebSockets on IIS.
- Session affinity with Application Request Routing.

For more information, see the guidance and external IIS resource cross-links in Publish an ASP.NET Core app to IIS.

Kubernetes

Create an ingress definition with the following Kubernetes annotations for session affinity:

```YAML
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: <ingress-name>
  annotations:
    nginx.ingress.kubernetes.io/affinity: "cookie"
    nginx.ingress.kubernetes.io/session-cookie-name: "affinity"
    nginx.ingress.kubernetes.io/session-cookie-expires: "14400"
    nginx.ingress.kubernetes.io/session-cookie-max-age: "14400"
```

Linux with Nginx

Follow the guidance for an ASP.NET Core SignalR app with the following changes:

- Change the `location` path from `/hubroute` (`location /hubroute { ... }`) to the root path `/` (`location / { ... }`).
- Remove the configuration for proxy buffering (`proxy_buffering off;`) because the setting only applies to Server-Sent Events (SSE), which aren't relevant to Blazor app client-server interactions.

For more information and configuration guidance, consult the following resources:

- ASP.NET Core SignalR production hosting and scaling
- Host ASP.NET Core on Linux with Nginx
- Configure ASP.NET Core to work with proxy servers and load balancers
- NGINX as a WebSocket Proxy
- WebSocket proxying
- Consult developers on non-Microsoft support forums:
  - Stack Overflow (tag: blazor)
  - ASP.NET Core Slack Team
  - Blazor Gitter

Linux with Apache

To host a Blazor app behind Apache on Linux, configure `ProxyPass` for HTTP and WebSockets traffic.

In the following example:

- Kestrel server is running on the host machine.
- The app listens for traffic on port 5000.

```
ProxyPreserveHost   On
ProxyPassMatch      ^/_blazor/(.*) http://localhost:5000/_blazor/$1
ProxyPass           /_blazor ws://localhost:5000/_blazor
ProxyPass           / http://localhost:5000/
ProxyPassReverse    / http://localhost:5000/
```

Enable the following modules:

```
a2enmod   proxy
a2enmod   proxy_wstunnel
```

Check the browser console for WebSockets errors. Example errors:

- Firefox can't establish a connection to the server at ws://the-domain-name.tld/_blazor?id=XXX
- Error: Failed to start the transport 'WebSockets': Error: There was an error with the transport.
- Error: Failed to start the transport 'LongPolling': TypeError: this.transport is undefined
- Error: Unable to connect to the server with any of the available transports. WebSockets failed
- Error: Cannot send data if the connection is not in the 'Connected' State.

For more information and configuration guidance, consult the following resources:

- Configure ASP.NET Core to work with proxy servers and load balancers
- Apache documentation
- Consult developers on non-Microsoft support forums:
  - Stack Overflow (tag: blazor)
  - ASP.NET Core Slack Team
  - Blazor Gitter

Measure network latency

JS interop can be used to measure network latency, as the following example demonstrates.

`MeasureLatency.razor`:

razor

```
@inject IJSRuntime JS

<h2>Measure Latency</h2>

@if (latency is null)
{
    <span>Calculating...</span>
}
else
{
    <span>@(latency.Value.TotalMilliseconds)ms</span>
}

@code {
    private DateTime startTime;
    private TimeSpan? latency;

    protected override async Task OnAfterRenderAsync(bool firstRender)
    {
        if (firstRender)
        {
            startTime = DateTime.UtcNow;
            var _ = await JS.InvokeAsync<string>("toString");
            latency = DateTime.UtcNow - startTime;
            StateHasChanged();
        }
    }
}
```

For a reasonable UI experience, we recommend a sustained UI latency of 250 ms or less.

Memory management

On the server, a new circuit is created for each user session. Each user session corresponds to rendering a single document in the browser. For example, multiple tabs create multiple sessions.

Blazor maintains a constant connection to the browser, called a *circuit*, that initiated the session. Connections can be lost at any time for any of several reasons, such as when the user loses network connectivity or abruptly closes the browser. When a connection is lost, Blazor has a recovery mechanism that places a limited number of circuits in a "disconnected" pool, giving clients a limited amount of time to reconnect and re-establish the session (default: 3 minutes).

After that time, Blazor releases the circuit and discards the session. From that point on, the circuit is eligible for garbage collection (GC) and is claimed when a collection for the circuit's GC generation is triggered. One important aspect to understand is that circuits have a long lifetime, which means that most of the objects rooted by the circuit eventually reach Gen 2. As a result, you might not see those objects released until a Gen 2 collection happens.

Measure memory usage in general

Prerequisites:

- The app must be published in **Release** configuration. **Debug** configuration measurements aren't relevant, as the generated code isn't representative of the code used for a production deployment.
- The app must run without a debugger attached, as this might also affect the behavior of the app and spoil the results. In Visual Studio, start the app without debugging by selecting **Debug** > **Start Without Debugging** from the menu bar or `Ctrl`+`F5` using the keyboard.
- Consider the different types of memory to understand how much memory is actually used by .NET. Generally, developers inspect app memory usage in Task Manager on Windows OS, which typically offers an upper bound of the actual memory in use. For more information, consult the following articles:
    - .NET Memory Performance Analysis: In particular, see the section on Memory Fundamentals.
    - Work flow of diagnosing memory performance issues (three-part series): Links to the three articles of the series are at the top of each article in the series.

Memory usage applied to Blazor

We compute the memory used by blazor as follows:

(**Active Circuits** × **Per-circuit Memory**) + (**Disconnected Circuits** × **Per-circuit Memory**)

The amount of memory a circuit uses and the maximum potential active circuits that an app can maintain is largely dependent on how the app is written. The maximum number of possible active circuits is roughly described by:

Maximum Available Memory / **Per-circuit Memory** = **Maximum Potential Active Circuits**

For a memory leak to occur in Blazor, the following must be true:

- The memory must be allocated by the framework, not the app. If you allocate a 1 GB array in the app, the app must manage the disposal of the array.
- The memory must not be actively used, which means the circuit isn't active and has been evicted from the disconnected circuits cache. If you have the maximum active

circuits running, running out of memory is a scale issue, not a memory leak.
- A garbage collection (GC) for the circuit's GC generation has run, but the garbage collector hasn't been able to claim the circuit because another object in the framework is holding a strong reference to the circuit.

In other cases, there's no memory leak. If the circuit is active (connected or disconnected), the circuit is still in use.

If a collection for the circuit's GC generation doesn't run, the memory isn't released because the garbage collector doesn't need to free the memory at that time.

If a collection for a GC generation runs and frees the circuit, you must validate the memory against the GC stats, not the process, as .NET might decide to keep the virtual memory active.

If the memory isn't freed, you must find a circuit that isn't either active or disconnected and that's rooted by another object in the framework. In any other case, the inability to free memory is an app issue in developer code.

Reduce memory usage

Adopt any of the following strategies to reduce an app's memory usage:

- Limit the total amount of memory used by the .NET process. For more information, see Runtime configuration options for garbage collection.
- Reduce the number of disconnected circuits.
- Reduce the time a circuit is allowed to be in the disconnected state.
- Trigger a garbage collection manually to perform a collection during downtime periods.
- Configure the garbage collection in Workstation mode, which aggressively triggers garbage collection, instead of Server mode.

Heap size for some mobile device browsers

When building a Blazor app that runs on the client and targets mobile device browsers, especially Safari on iOS, decreasing the maximum memory for the app with the MSBuild property `EmccMaximumHeapSize` may be required. For more information, see Host and deploy ASP.NET Core Blazor WebAssembly.

Additional actions and considerations

- Capture a memory dump of the process when memory demands are high and identify the objects are taking the most memory and where are those objects are rooted (what holds a reference to them).
- You can examine the statistics on how memory in your app is behaving using `dotnet-counters`. For more information see Investigate performance counters (dotnet-counters).
- Even when a GC is triggered, .NET holds on to the memory instead of returning it to the OS immediately, as it's likely that it will reuse the memory the near future. This avoids committing and decommitting memory constantly, which is expensive. You'll see this reflected if you use `dotnet-counters` because you'll see the GCs happen and the amount of used memory go down to 0 (zero), but you won't see the working set counter decrease, which is the sign that .NET is holding on to the memory to reuse it. For more information on project file (`.csproj`) settings to control this behavior, see Runtime configuration options for garbage collection.
- Server GC doesn't trigger garbage collections until it determines it's absolutely necessary to do so to avoid freezing your app and considers that your app is the only thing running on the machine, so it can use all the memory in the system. If the system has 50 GB, the garbage collector seeks to use the full 50 GB of available memory before it triggers a Gen 2 collection.
- For information on disconnected circuit retention configuration, see ASP.NET Core Blazor SignalR guidance.

Measuring memory

- Publish the app in Release configuration.
- Run a published version of the app.
- Don't attach a debugger to the running app.
- Does triggering a Gen 2 forced, compacting collection (`GC.Collect(2, GCCollectionMode.Aggressive | GCCollectionMode.Forced, blocking: true, compacting: true)`) free the memory?
- Consider if your app is allocating objects on the large object heap.
- Are you testing the memory growth after the app is warmed up with requests and processing? Typically, there are caches that are populated when code executes for the first time that add a constant amount of memory to the footprint of the app.

Host and deploy ASP.NET Core Blazor WebAssembly

Article • 09/27/2024

ⓘ **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 host and deploy Blazor WebAssembly using ASP.NET Core, Content Delivery Networks (CDN), file servers, and GitHub Pages.

With the Blazor WebAssembly hosting model:

- The Blazor app, its dependencies, and the .NET runtime are downloaded to the browser in parallel.
- The app is executed directly on the browser UI thread.

This article pertains to the deployment scenario where the Blazor app is placed on a static hosting web server or service, .NET isn't used to serve the Blazor app. This strategy is covered in the Standalone deployment section, which includes information on hosting a Blazor WebAssembly app as an IIS sub-app.

Subdomain and IIS sub-application hosting

Subdomain hosting doesn't require special configuration of the app. You ***don't*** need to configure the app base path (the `<base>` tag in `wwwroot/index.html`) to host the app at a subdomain.

IIS sub-application hosting ***does*** require you to set the app base path. For more information and cross-links to further guidance on IIS sub-application hosting, see Host and deploy ASP.NET Core Blazor.

Decrease maximum heap size for some mobile device browsers

When building a Blazor app that runs on the client (`.Client` project of a Blazor Web App or standalone Blazor WebAssembly app) and targets mobile device browsers, especially Safari on iOS, decreasing the maximum memory for the app with the MSBuild property `EmccMaximumHeapSize` may be required. The default value is 2,147,483,648 bytes, which may be too large and result in the app crashing if the app attempts to allocate more memory with the browser failing to grant it. The following example sets the value to 268,435,456 bytes in the `Program` file:

```XML
<EmccMaximumHeapSize>268435456</EmccMaximumHeapSize>
```

For more information on Mono /WebAssembly MSBuild properties and targets, see WasmApp.Common.targets (dotnet/runtime GitHub repository) .

Webcil packaging format for .NET assemblies

Webcil is a web-friendly packaging format for .NET assemblies designed to enable using Blazor WebAssembly in restrictive network environments. Webcil files use a standard WebAssembly wrapper, where the assemblies are deployed as WebAssembly files that use the standard `.wasm` file extension.

Webcil is the default packaging format when you publish a Blazor WebAssembly app. To disable the use of Webcil, set the following MSBuild property in the app's project file:

```XML
<PropertyGroup>
  <WasmEnableWebcil>false</WasmEnableWebcil>
</PropertyGroup>
```

Customize how boot resources are loaded

Customize how boot resources are loaded using the `loadBootResource` API. For more information, see ASP.NET Core Blazor startup.

Compression

When a Blazor WebAssembly app is published, the output is statically compressed during publish to reduce the app's size and remove the overhead for runtime compression. The following compression algorithms are used:

- Brotli (highest level)
- Gzip

Blazor relies on the host to serve the appropriate compressed files. When hosting a Blazor WebAssembly standalone app, additional work might be required to ensure that statically-compressed files are served:

- For IIS `web.config` compression configuration, see the IIS: Brotli and Gzip compression section.
- When hosting on static hosting solutions that don't support statically-compressed file content negotiation, consider configuring the app to fetch and decode Brotli compressed files:

Obtain the JavaScript Brotli decoder from the google/brotli GitHub repository. The minified decoder file is named `decode.min.js` and found in the repository's js folder.

Note

If the minified version of the `decode.js` script (`decode.min.js`) fails, try using the unminified version (`decode.js`) instead.

Update the app to use the decoder.

In the `wwwroot/index.html` file, set `autostart` to `false` on Blazor's `<script>` tag:

HTML

```html
<script src="_framework/blazor.webassembly.js" autostart="false"></script>
```

After Blazor's `<script>` tag and before the closing `</body>` tag, add the following JavaScript code `<script>` block.

Blazor Web App:

HTML

```html
<script type="module">
  import { BrotliDecode } from './decode.min.js';
  Blazor.start({
    webAssembly: {
      loadBootResource: function (type, name, defaultUri, integrity) {
        if (type !== 'dotnetjs' && location.hostname !== 'localhost' && type
!== 'configuration' && type !== 'manifest') {
```

```
          return (async function () {
            const response = await fetch(defaultUri + '.br', { cache: 'no-
cache' });
            if (!response.ok) {
              throw new Error(response.statusText);
            }
            const originalResponseBuffer = await response.arrayBuffer();
            const originalResponseArray = new
Int8Array(originalResponseBuffer);
            const decompressedResponseArray =
BrotliDecode(originalResponseArray);
            const contentType = type ===
              'dotnetwasm' ? 'application/wasm' : 'application/octet-
stream';
            return new Response(decompressedResponseArray,
              { headers: { 'content-type': contentType } });
          })();
        }
      }
    }
  });
</script>
```

Standalone Blazor WebAssembly:

```
HTML

<script type="module">
  import { BrotliDecode } from './decode.min.js';
  Blazor.start({
    loadBootResource: function (type, name, defaultUri, integrity) {
      if (type !== 'dotnetjs' && location.hostname !== 'localhost' && type
!== 'configuration') {
        return (async function () {
          const response = await fetch(defaultUri + '.br', { cache: 'no-
cache' });
          if (!response.ok) {
            throw new Error(response.statusText);
          }
          const originalResponseBuffer = await response.arrayBuffer();
          const originalResponseArray = new
Int8Array(originalResponseBuffer);
          const decompressedResponseArray =
BrotliDecode(originalResponseArray);
          const contentType = type ===
            'dotnetwasm' ? 'application/wasm' : 'application/octet-stream';
          return new Response(decompressedResponseArray,
            { headers: { 'content-type': contentType } });
        })();
      }
    }
  });
</script>
```

For more information on loading boot resources, see ASP.NET Core Blazor startup.

To disable compression, add the `CompressionEnabled` MSBuild property to the app's project file and set the value to `false`:

XML

```xml
<PropertyGroup>
  <CompressionEnabled>false</CompressionEnabled>
</PropertyGroup>
```

The `CompressionEnabled` property can be passed to the dotnet publish command with the following syntax in a command shell:

.NET CLI

```
dotnet publish -p:CompressionEnabled=false
```

Rewrite URLs for correct routing

Routing requests for page components in a Blazor WebAssembly app isn't as straightforward as routing requests in a Blazor Server app. Consider a Blazor WebAssembly app with two components:

- `Main.razor`: Loads at the root of the app and contains a link to the `About` component (`href="About"`).
- `About.razor`: `About` component.

When the app's default document is requested using the browser's address bar (for example, `https://www.contoso.com/`):

1. The browser makes a request.
2. The default page is returned, which is usually `index.html`.
3. `index.html` bootstraps the app.
4. Router component loads, and the Razor `Main` component is rendered.

In the Main page, selecting the link to the `About` component works on the client because the Blazor router stops the browser from making a request on the Internet to `www.contoso.com` for `About` and serves the rendered `About` component itself. All of the requests for internal endpoints *within the Blazor WebAssembly app* work the same way: Requests don't trigger browser-based requests to server-hosted resources on the Internet. The router handles the requests internally.

If a request is made using the browser's address bar for `www.contoso.com/About`, the request fails. No such resource exists on the app's Internet host, so a *404 - Not Found* response is returned.

Because browsers make requests to Internet-based hosts for client-side pages, web servers and hosting services must rewrite all requests for resources not physically on the server to the `index.html` page. When `index.html` is returned, the app's Blazor router takes over and responds with the correct resource.

When deploying to an IIS server, you can use the URL Rewrite Module with the app's published `web.config` file. For more information, see the IIS section.

Standalone deployment

A *standalone deployment* serves the Blazor WebAssembly app as a set of static files that are requested directly by clients. Any static file server is able to serve the Blazor app.

Standalone deployment assets are published into either the `/bin/Release/{TARGET FRAMEWORK}/publish/wwwroot` or `bin\Release\{TARGET FRAMEWORK}\browser-wasm\publish\` folder (depending on the version of the .NET SDK in use), where the `{TARGET FRAMEWORK}` placeholder is the target framework.

Azure App Service

Blazor WebAssembly apps can be deployed to Azure App Services on Windows, which hosts the app on IIS.

Deploying a standalone Blazor WebAssembly app to Azure App Service for Linux isn't currently supported. We recommend hosting a standalone Blazor WebAssembly app using Azure Static Web Apps, which supports this scenario.

Azure Static Web Apps

Use one of the following approaches to deploy a Blazor WebAssembly app to Azure Static Web Apps:

- Deploy from Visual Studio
- Deploy from Visual Studio Code

- Deploy from GitHub

Deploy from Visual Studio

To deploy from Visual Studio, create a publish profile for Azure Static Web Apps:

1. Save any unsaved work in the project, as a Visual Studio restart might be required during the process.

2. In Visual Studio's **Publish** UI, select **Target** > **Azure** > **Specific Target** > **Azure Static Web Apps** to create a publish profile.

3. If the **Azure WebJobs Tools** component for Visual Studio isn't installed, a prompt appears to install the **ASP.NET and web development** component. Follow the prompts to install the tools using the Visual Studio Installer. Visual Studio closes and reopens automatically while installing the tools. After the tools are installed, start over at the first step to create the publish profile.

4. In the publish profile configuration, provide the **Subscription name**. Select an existing instance, or select **Create a new instance**. When creating a new instance in the Azure portal's **Create Static Web App** UI, set the **Deployment details** > **Source** to **Other**. Wait for the deployment to complete in the Azure portal before proceeding.

5. In the publish profile configuration, select the Azure Static Web Apps instance from the instance's resource group. Select **Finish** to create the publish profile. If Visual Studio prompts to install the Static Web Apps (SWA) CLI, install the CLI by following the prompts. The SWA CLI requires NPM/Node.js (Visual Studio documentation).

After the publish profile is created, deploy the app to the Azure Static Web Apps instance using the publish profile by selecting the **Publish** button.

Deploy from Visual Studio Code

To deploy from Visual Studio Code, see Quickstart: Build your first static site with Azure Static Web Apps.

Deploy from GitHub

To deploy from a GitHub repository, see Tutorial: Building a static web app with Blazor in Azure Static Web Apps.

IIS

IIS is a capable static file server for Blazor apps. To configure IIS to host Blazor, see Build a Static Website on IIS.

Published assets are created in the `/bin/Release/{TARGET FRAMEWORK}/publish` or `bin\Release\{TARGET FRAMEWORK}\browser-wasm\publish` folder, depending on which version of the SDK is used and where the `{TARGET FRAMEWORK}` placeholder is the target framework. Host the contents of the `publish` folder on the web server or hosting service.

web.config

When a Blazor project is published, a `web.config` file is created with the following IIS configuration:

- MIME types
- HTTP compression is enabled for the following MIME types:
  - `application/octet-stream`
  - `application/wasm`
- URL Rewrite Module rules are established:
  - Serve the sub-directory where the app's static assets reside (`wwwroot/{PATH REQUESTED}`).
  - Create SPA fallback routing so that requests for non-file assets are redirected to the app's default document in its static assets folder (`wwwroot/index.html`).

Use a custom `web.config`

To use a custom `web.config` file:

1. Place the custom `web.config` file in the project's root folder.
2. Publish the project. For more information, see Host and deploy ASP.NET Core Blazor.

If the SDK's `web.config` generation or transformation during publish either doesn't move the file to published assets in the `publish` folder or modifies the custom configuration in your custom `web.config` file, use any of the following approaches as needed to take full control of the process:

- If the SDK doesn't generate the file, for example, in a standalone Blazor WebAssembly app at `/bin/Release/{TARGET FRAMEWORK}/publish/wwwroot` or `bin\Release\{TARGET FRAMEWORK}\browser-wasm\publish`, depending on which version of the SDK is used and where the `{TARGET FRAMEWORK}` placeholder is the target framework, set the `<PublishIISAssets>` property to `true` in the project file (`.csproj`). Usually for standalone WebAssembly apps, this is the only required setting to move a custom `web.config` file and prevent transformation of the file by the SDK.

XML

```xml
<PropertyGroup>
  <PublishIISAssets>true</PublishIISAssets>
</PropertyGroup>
```

- Disable the SDK's `web.config` transformation in the project file (`.csproj`):

XML

```xml
<PropertyGroup>
  <IsTransformWebConfigDisabled>true</IsTransformWebConfigDisabled>
</PropertyGroup>
```

- Add a custom target to the project file (`.csproj`) to move a custom `web.config` file. In the following example, the custom `web.config` file is placed by the developer at the root of the project. If the `web.config` file resides elsewhere, specify the path to the file in `SourceFiles`. The following example specifies the `publish` folder with `$(PublishDir)`, but provide a path to `DestinationFolder` for a custom output location.

XML

```xml
<Target Name="CopyWebConfig" AfterTargets="Publish">
  <Copy SourceFiles="web.config" DestinationFolder="$(PublishDir)" />
</Target>
```

Install the URL Rewrite Module

The URL Rewrite Module is required to rewrite URLs. The module isn't installed by default, and it isn't available for install as a Web Server (IIS) role service feature. The module must be downloaded from the IIS website. Use the Web Platform Installer to install the module:

1. Locally, navigate to the URL Rewrite Module downloads page. For the English version, select **WebPI** to download the WebPI installer. For other languages, select the appropriate architecture for the server (x86/x64) to download the installer.
2. Copy the installer to the server. Run the installer. Select the **Install** button and accept the license terms. A server restart isn't required after the install completes.

Configure the website

Set the website's **Physical path** to the app's folder. The folder contains:

- The `web.config` file that IIS uses to configure the website, including the required redirect rules and file content types.
- The app's static asset folder.

Host as an IIS sub-app

If a standalone app is hosted as an IIS sub-app, perform either of the following:

- Disable the inherited ASP.NET Core Module handler.

  Remove the handler in the Blazor app's published `web.config` file by adding a `<handlers>` section to the `<system.webServer>` section of the file:

  XML

  ```xml
  <handlers>
    <remove name="aspNetCore" />
  </handlers>
  ```

- Disable inheritance of the root (parent) app's `<system.webServer>` section using a `<location>` element with `inheritInChildApplications` set to `false`:

  XML

  ```xml
  <?xml version="1.0" encoding="utf-8"?>
  <configuration>
    <location path="." inheritInChildApplications="false">
      <system.webServer>
        <handlers>
          <add name="aspNetCore" ... />
        </handlers>
        <aspNetCore ... />
      </system.webServer>
    </location>
  </configuration>
  ```

> **Note**
>
> Disabling inheritance of the root (parent) app's `<system.webServer>` section is the default configuration for published apps using the .NET SDK.

Removing the handler or disabling inheritance is performed in addition to configuring the app's base path. Set the app base path in the app's `index.html` file to the IIS alias used when configuring the sub-app in IIS.

Configure the app's base path by following the guidance in the Host and deploy ASP.NET Core Blazor article.

Brotli and Gzip compression

This section only applies to standalone Blazor WebAssembly apps.

IIS can be configured via `web.config` to serve Brotli or Gzip compressed Blazor assets for standalone Blazor WebAssembly apps. For an example configuration file, see web.config.

Additional configuration of the example `web.config` file might be required in the following scenarios:

- The app's specification calls for either of the following:
  - Serving compressed files that aren't configured by the example `web.config` file.
  - Serving compressed files configured by the example `web.config` file in an uncompressed format.
- The server's IIS configuration (for example, `applicationHost.config`) provides server-level IIS defaults. Depending on the server-level configuration, the app might require a different IIS configuration than what the example `web.config` file contains.

For more information on custom `web.config` files, see the Use a custom web.config section.

Troubleshooting

If a *500 - Internal Server Error* is received and IIS Manager throws errors when attempting to access the website's configuration, confirm that the URL Rewrite Module is installed. When the module isn't installed, the `web.config` file can't be parsed by IIS.

This prevents the IIS Manager from loading the website's configuration and the website from serving Blazor's static files.

For more information on troubleshooting deployments to IIS, see Troubleshoot ASP.NET Core on Azure App Service and IIS.

Azure Storage

Azure Storage static file hosting allows serverless Blazor app hosting. Custom domain names, the Azure Content Delivery Network (CDN), and HTTPS are supported.

When the blob service is enabled for static website hosting on a storage account:

- Set the **Index document name** to `index.html`.
- Set the **Error document path** to `index.html`. Razor components and other non-file endpoints don't reside at physical paths in the static content stored by the blob service. When a request for one of these resources is received that the Blazor router should handle, the *404 - Not Found* error generated by the blob service routes the request to the **Error document path**. The `index.html` blob is returned, and the Blazor router loads and processes the path.

If files aren't loaded at runtime due to inappropriate MIME types in the files' `Content-Type` headers, take either of the following actions:

- Configure your tooling to set the correct MIME types (`Content-Type` headers) when the files are deployed.

- Change the MIME types (`Content-Type` headers) for the files after the app is deployed.

  In Storage Explorer (Azure portal) for each file:

  1. Right-click the file and select **Properties**.
  2. Set the **ContentType** and select the **Save** button.

For more information, see Static website hosting in Azure Storage.

Nginx

The following `nginx.conf` file is simplified to show how to configure Nginx to send the `index.html` file whenever it can't find a corresponding file on disk.

```
events { }
http {
    server {
        listen 80;

        location / {
            root      /usr/share/nginx/html;
            try_files $uri $uri/ /index.html =404;
        }
    }
}
```

When setting the NGINX burst rate limit with limit_req, Blazor WebAssembly apps may require a large `burst` parameter value to accommodate the relatively large number of requests made by an app. Initially, set the value to at least 60:

```
http {
    server {
        ...

        location / {
            ...

            limit_req zone=one burst=60 nodelay;
        }
    }
}
```

Increase the value if browser developer tools or a network traffic tool indicates that requests are receiving a *503 - Service Unavailable* status code.

For more information on production Nginx web server configuration, see Creating NGINX Plus and NGINX Configuration Files.

Apache

To deploy a Blazor WebAssembly app to Apache:

1. Create the Apache configuration file. The following example is a simplified configuration file (`blazorapp.config`):

```
<VirtualHost *:80>
    ServerName www.example.com
    ServerAlias *.example.com

    DocumentRoot "/var/www/blazorapp"
    ErrorDocument 404 /index.html

    AddType application/wasm .wasm

    <Directory "/var/www/blazorapp">
        Options -Indexes
        AllowOverride None
    </Directory>

    <IfModule mod_deflate.c>
        AddOutputFilterByType DEFLATE text/css
        AddOutputFilterByType DEFLATE application/javascript
        AddOutputFilterByType DEFLATE text/html
        AddOutputFilterByType DEFLATE application/octet-stream
        AddOutputFilterByType DEFLATE application/wasm
        <IfModule mod_setenvif.c>
            BrowserMatch ^Mozilla/4 gzip-only-text/html
            BrowserMatch ^Mozilla/4.0[678] no-gzip
            BrowserMatch bMSIE !no-gzip !gzip-only-text/html
        </IfModule>
    </IfModule>

    ErrorLog /var/log/httpd/blazorapp-error.log
    CustomLog /var/log/httpd/blazorapp-access.log common
</VirtualHost>
```

1. Place the Apache configuration file into the `/etc/httpd/conf.d/` directory.

2. Place the app's published assets (`/bin/Release/{TARGET FRAMEWORK}/publish/wwwroot`, where the `{TARGET FRAMEWORK}` placeholder is the target framework) into the `/var/www/blazorapp` directory (the location specified to `DocumentRoot` in the configuration file).

3. Restart the Apache service.

For more information, see mod_mime and mod_deflate.

GitHub Pages

The default GitHub Action, which deploys pages, skips deployment of folders starting with underscore, for example, the `_framework` folder. To deploy folders starting with underscore, add an empty `.nojekyll` file to the Git branch.

Git treats JavaScript (JS) files, such as `blazor.webassembly.js`, as text and converts line endings from CRLF (carriage return-line feed) to LF (line feed) in the deployment pipeline. These changes to JS files produce different file hashes than Blazor sends to the client in the `blazor.boot.json` file. The mismatches result in integrity check failures on the client. One approach to solving this problem is to add a `.gitattributes` file with `*.js binary` line before adding the app's assets to the Git branch. The `*.js binary` line configures Git to treat JS files as binary files, which avoids processing the files in the deployment pipeline. The file hashes of the unprocessed files match the entries in the `blazor.boot.json` file, and client-side integrity checks pass. For more information, see ASP.NET Core Blazor WebAssembly .NET runtime and app bundle caching.

To handle URL rewrites, add a `wwwroot/404.html` file with a script that handles redirecting the request to the `index.html` page. For an example, see the SteveSandersonMS/BlazorOnGitHubPages GitHub repository :

- wwwroot/404.html
- Live site

When using a project site instead of an organization site, update the `<base>` tag in `wwwroot/index.html`. Set the `href` attribute value to the GitHub repository name with a trailing slash (for example, `/my-repository/`). In the SteveSandersonMS/BlazorOnGitHubPages GitHub repository , the base `href` is updated at publish by the .github/workflows/main.yml configuration file .

> ⓘ **Note**
>
> The SteveSandersonMS/BlazorOnGitHubPages GitHub repository isn't owned, maintained, or supported by the .NET Foundation or Microsoft.

Standalone with Docker

A standalone Blazor WebAssembly app is published as a set of static files for hosting by a static file server.

To host the app in Docker:

- Choose a Docker container with web server support, such as Ngnix or Apache.
- Copy the `publish` folder assets to a location folder defined in the web server for serving static files.
- Apply additional configuration as needed to serve the Blazor WebAssembly app.

For configuration guidance, see the following resources:

- Nginx section or Apache section of this article

- Docker Documentation

Host configuration values

Blazor WebAssembly apps can accept the following host configuration values as command-line arguments at runtime in the development environment.

Content root

The `--contentroot` argument sets the absolute path to the directory that contains the app's content files (content root). In the following examples, `/content-root-path` is the app's content root path.

- Pass the argument when running the app locally at a command prompt. From the app's directory, execute:

  .NET CLI

  ```
  dotnet watch --contentroot=/content-root-path
  ```

- Add an entry to the app's `launchSettings.json` file in the **IIS Express** profile. This setting is used when the app is run with the Visual Studio Debugger and from a command prompt with `dotnet watch` (or `dotnet run`).

  JSON

  ```
  "commandLineArgs": "--contentroot=/content-root-path"
  ```

- In Visual Studio, specify the argument in **Properties** > **Debug** > **Application arguments**. Setting the argument in the Visual Studio property page adds the argument to the `launchSettings.json` file.

  Console

  ```
  --contentroot=/content-root-path
  ```

Path base

The `--pathbase` argument sets the app base path for an app run locally with a non-root relative URL path (the `<base>` tag `href` is set to a path other than `/` for staging and production). In the following examples, `/relative-URL-path` is the app's path base. For more information, see App base path.

> **Important**
>
> Unlike the path provided to `href` of the `<base>` tag, don't include a trailing slash (`/`) when passing the `--pathbase` argument value. If the app base path is provided in the `<base>` tag as `<base href="/CoolApp/">` (includes a trailing slash), pass the command-line argument value as `--pathbase=/CoolApp` (no trailing slash).

- Pass the argument when running the app locally at a command prompt. From the app's directory, execute:

  .NET CLI

  ```
  dotnet watch --pathbase=/relative-URL-path
  ```

- Add an entry to the app's `launchSettings.json` file in the **IIS Express** profile. This setting is used when running the app with the Visual Studio Debugger and from a command prompt with `dotnet watch` (or `dotnet run`).

  JSON

  ```
  "commandLineArgs": "--pathbase=/relative-URL-path"
  ```

- In Visual Studio, specify the argument in **Properties** > **Debug** > **Application arguments**. Setting the argument in the Visual Studio property page adds the argument to the `launchSettings.json` file.

  Console

  ```
  --pathbase=/relative-URL-path
  ```

URLs

The `--urls` argument sets the IP addresses or host addresses with ports and protocols to listen on for requests.

- Pass the argument when running the app locally at a command prompt. From the app's directory, execute:

.NET CLI

```
dotnet watch --urls=http://127.0.0.1:0
```

- Add an entry to the app's `launchSettings.json` file in the **IIS Express** profile. This setting is used when running the app with the Visual Studio Debugger and from a command prompt with `dotnet watch` (or `dotnet run`).

JSON

```
"commandLineArgs": "--urls=http://127.0.0.1:0"
```

- In Visual Studio, specify the argument in **Properties** > **Debug** > **Application arguments**. Setting the argument in the Visual Studio property page adds the argument to the `launchSettings.json` file.

Console

```
--urls=http://127.0.0.1:0
```

Configure the Trimmer

Blazor performs Intermediate Language (IL) trimming on each Release build to remove unnecessary IL from the output assemblies. For more information, see Configure the Trimmer for ASP.NET Core Blazor.

Change the file name extension of DLL files

This section applies to ASP.NET Core 6.x and 7.x. In ASP.NET Core in .NET 8 or later, .NET assemblies are deployed as WebAssembly files (`.wasm`) using the Webcil file format. In ASP.NET Core in .NET 8 or later, this section only applies if the Webcil file format has been disabled in the app's project file.

If a firewall, anti-virus program, or network security appliance is blocking the transmission of the app's dynamic-link library (DLL) files (`.dll`), you can follow the guidance in this section to change the file name extensions of the app's published DLL files.

Note

Changing the file name extensions of the app's DLL files might not resolve the problem because many security systems scan the content of the app's files, not merely check file extensions.

For a more robust approach in environments that block the download and execution of DLL files, use ASP.NET Core in .NET 8 or later, which packages .NET assemblies as WebAssembly files (`.wasm`) using the Webcil file format. For more information, see the *Webcil packaging format for .NET assemblies* section in an 8.0 or later version of this article.

Third-party approaches exist for dealing with this problem. For more information, see the resources at Awesome Blazor.

After publishing the app, use a shell script or DevOps build pipeline to rename `.dll` files to use a different file extension in the directory of the app's published output.

In the following examples:

- PowerShell (PS) is used to update the file extensions.
- `.dll` files are renamed to use the `.bin` file extension from the command line.
- Files listed in the published `blazor.boot.json` file with a `.dll` file extension are updated to the `.bin` file extension.
- If service worker assets are also in use, a PowerShell command updates the `.dll` files listed in the `service-worker-assets.js` file to the `.bin` file extension.

To use a different file extension than `.bin`, replace `.bin` in the following commands with the desired file extension.

On Windows:

PowerShell

```
dir {PATH} | rename-item -NewName { $_.name -replace ".dll\b",".bin" }
((Get-Content {PATH}\blazor.boot.json -Raw) -replace '.dll"','.bin"') | Set-
Content {PATH}\blazor.boot.json
```

In the preceding command, the `{PATH}` placeholder is the path to the published `_framework` folder (for example, `.\bin\Release\net6.0\browser-wasm\publish\wwwroot\_framework` from the project's root folder).

If service worker assets are also in use:

PowerShell

```
((Get-Content {PATH}\service-worker-assets.js -Raw) -replace
'.dll"','.bin"') | Set-Content {PATH}\service-worker-assets.js
```

In the preceding command, the `{PATH}` placeholder is the path to the published `service-worker-assets.js` file.

On Linux or macOS:

Console

```
for f in {PATH}/*; do mv "$f" "`echo $f | sed -e 's/\.dll/.bin/g'`"; done
sed -i 's/\.dll"/.bin"/g' {PATH}/blazor.boot.json
```

In the preceding command, the `{PATH}` placeholder is the path to the published `_framework` folder (for example, `.\bin\Release\net6.0\browser-wasm\publish\wwwroot\_framework` from the project's root folder).

If service worker assets are also in use:

Console

```
sed -i 's/\.dll"/.bin"/g' {PATH}/service-worker-assets.js
```

In the preceding command, the `{PATH}` placeholder is the path to the published `service-worker-assets.js` file.

To address the compressed `blazor.boot.json.gz` and `blazor.boot.json.br` files, adopt either of the following approaches:

- Remove the compressed `blazor.boot.json.gz` and `blazor.boot.json.br` files. **Compression is disabled with this approach.**
- Recompress the updated `blazor.boot.json` file.

The preceding guidance for the compressed `blazor.boot.json` file also applies when service worker assets are in use. Remove or recompress `service-worker-assets.js.br` and `service-worker-assets.js.gz`. Otherwise, file integrity checks fail in the browser.

The following Windows example for .NET 6 uses a PowerShell script placed at the root of the project. The following script, which disables compression, is the basis for further modification if you wish to recompress the `blazor.boot.json` file.

`ChangeDLLExtensions.ps1:`:

PowerShell

```powershell
param([string]$filepath,[string]$tfm)
dir $filepath\bin\Release\$tfm\browser-wasm\publish\wwwroot\_framework |
rename-item -NewName { $_.name -replace ".dll\b",".bin" }
((Get-Content $filepath\bin\Release\$tfm\browser-
wasm\publish\wwwroot\_framework\blazor.boot.json -Raw) -replace
'.dll"','.bin"') | Set-Content $filepath\bin\Release\$tfm\browser-
wasm\publish\wwwroot\_framework\blazor.boot.json
Remove-Item $filepath\bin\Release\$tfm\browser-
wasm\publish\wwwroot\_framework\blazor.boot.json.gz
Remove-Item $filepath\bin\Release\$tfm\browser-
wasm\publish\wwwroot\_framework\blazor.boot.json.br
```

If service worker assets are also in use, add the following commands:

PowerShell

```powershell
((Get-Content $filepath\bin\Release\$tfm\browser-
wasm\publish\wwwroot\service-worker-assets.js -Raw) -replace
'.dll"','.bin"') | Set-Content $filepath\bin\Release\$tfm\browser-
wasm\publish\wwwroot\_framework\wwwroot\service-worker-assets.js
Remove-Item $filepath\bin\Release\$tfm\browser-
wasm\publish\wwwroot\_framework\wwwroot\service-worker-assets.js.gz
Remove-Item $filepath\bin\Release\$tfm\browser-
wasm\publish\wwwroot\_framework\wwwroot\service-worker-assets.js.br
```

In the project file, the script is executed after publishing the app for the `Release` configuration:

XML

```xml
<Target Name="ChangeDLLFileExtensions" AfterTargets="AfterPublish"
Condition="'$(Configuration)'=='Release'">
  <Exec Command="powershell.exe -command &quot;&amp; {
.\ChangeDLLExtensions.ps1 '$(SolutionDir)' '$(TargetFramework)'}&quot;" />
</Target>
```

ⓘ **Note**

When renaming and lazy loading the same assemblies, see the guidance in Lazy load assemblies in ASP.NET Core Blazor WebAssembly.

Usually, the app's server requires static asset configuration to serve the files with the updated extension. For an app hosted by IIS, add a MIME map entry (`<mimeMap>`) for the new file extension in the static content section (`<staticContent>`) in a custom `web.config` file. The following example assumes that the file extension is changed from `.dll` to `.bin`:

XML

```xml
<staticContent>
  ...
  <mimeMap fileExtension=".bin" mimeType="application/octet-stream" />
  ...
</staticContent>
```

Include an update for compressed files if compression is in use:

```
<mimeMap fileExtension=".bin.br" mimeType="application/octet-stream" />
<mimeMap fileExtension=".bin.gz" mimeType="application/octet-stream" />
```

Remove the entry for the `.dll` file extension:

diff

```diff
- <mimeMap fileExtension=".dll" mimeType="application/octet-stream" />
```

Remove entries for compressed `.dll` files if compression is in use:

diff

```diff
- <mimeMap fileExtension=".dll.br" mimeType="application/octet-stream" />
- <mimeMap fileExtension=".dll.gz" mimeType="application/octet-stream" />
```

For more information on custom `web.config` files, see the Use a custom web.config section.

Prior deployment corruption

Typically on deployment:

- Only the files that have changed are replaced, which usually results in a faster deployment.
- Existing files that aren't part of the new deployment are left in place for use by the new deployment.

In rare cases, lingering files from a prior deployment can corrupt a new deployment. Completely deleting the existing deployment (or locally-published app prior to deployment) may resolve the issue with a corrupted deployment. Often, deleting the existing deployment ***once*** is sufficient to resolve the problem, including for a DevOps build and deploy pipeline.

If you determine that clearing a prior deployment is always required when a DevOps build and deploy pipeline is in use, you can temporarily add a step to the build pipeline to delete the prior deployment for each new deployment until you troubleshoot the exact cause of the corruption.

Resolve integrity check failures

When Blazor WebAssembly downloads an app's startup files, it instructs the browser to perform integrity checks on the responses. Blazor sends SHA-256 hash values for DLL (`.dll`), WebAssembly (`.wasm`), and other files in the `blazor.boot.json` file, which isn't cached on clients. The file hashes of cached files are compared to the hashes in the `blazor.boot.json` file. For cached files with a matching hash, Blazor uses the cached files. Otherwise, files are requested from the server. After a file is downloaded, its hash is checked again for integrity validation. An error is generated by the browser if any downloaded file's integrity check fails.

Blazor's algorithm for managing file integrity:

- Ensures that the app doesn't risk loading an inconsistent set of files, for example if a new deployment is applied to your web server while the user is in the process of downloading the application files. Inconsistent files can result in a malfunctioning app.
- Ensures the user's browser never caches inconsistent or invalid responses, which can prevent the app from starting even if the user manually refreshes the page.
- Makes it safe to cache the responses and not check for server-side changes until the expected SHA-256 hashes themselves change, so subsequent page loads involve fewer requests and complete faster.

If the web server returns responses that don't match the expected SHA-256 hashes, an error similar to the following example appears in the browser's developer console:

> Failed to find a valid digest in the 'integrity' attribute for resource 'https://myapp.example.com/_framework/MyBlazorApp.dll' with computed SHA-256 integrity 'IIa70iwvmEg5WiDV17OpQ5eCztNYqL186J56852RpJY='. The resource has been blocked.

In most cases, the warning doesn't indicate a problem with integrity checking. Instead, the warning usually means that some other problem exists.

For Blazor WebAssembly's boot reference source, see the Boot.WebAssembly.ts file in the dotnet/aspnetcore GitHub repository.

> **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).

Diagnosing integrity problems

When an app is built, the generated `blazor.boot.json` manifest describes the SHA-256 hashes of boot resources at the time that the build output is produced. The integrity check passes as long as the SHA-256 hashes in `blazor.boot.json` match the files delivered to the browser.

Common reasons why this fails include:

- The web server's response is an error (for example, a *404 - Not Found* or a *500 - Internal Server Error*) instead of the file the browser requested. This is reported by the browser as an integrity check failure and not as a response failure.
- Something has changed the contents of the files between the build and delivery of the files to the browser. This might happen:
  - If you or build tools manually modify the build output.
  - If some aspect of the deployment process modified the files. For example if you use a Git-based deployment mechanism, bear in mind that Git transparently converts Windows-style line endings to Unix-style line endings if you commit files on Windows and check them out on Linux. Changing file line endings change the SHA-256 hashes. To avoid this problem, consider using .gitattributes to treat build artifacts as binary files.
  - The web server modifies the file contents as part of serving them. For example, some content distribution networks (CDNs) automatically attempt to minify HTML, thereby modifying it. You may need to disable such features.
- The `blazor.boot.json` file fails to load properly or is improperly cached on the client. Common causes include either of the following:
  - Misconfigured or malfunctioning custom developer code.

    - One or more misconfigured intermediate caching layers.

To diagnose which of these applies in your case:

1. Note which file is triggering the error by reading the error message.
2. Open your browser's developer tools and look in the *Network* tab. If necessary, reload the page to see the list of requests and responses. Find the file that is triggering the error in that list.
3. Check the HTTP status code in the response. If the server returns anything other than *200 - OK* (or another 2xx status code), then you have a server-side problem to diagnose. For example, status code 403 means there's an authorization problem, whereas status code 500 means the server is failing in an unspecified manner. Consult server-side logs to diagnose and fix the app.
4. If the status code is *200 - OK* for the resource, look at the response content in browser's developer tools and check that the content matches up with the data expected. For example, a common problem is to misconfigure routing so that requests return your `index.html` data even for other files. Make sure that responses to `.wasm` requests are WebAssembly binaries and that responses to `.dll` requests are .NET assembly binaries. If not, you have a server-side routing problem to diagnose.
5. Seek to validate the app's published and deployed output with the Troubleshoot integrity PowerShell script.

If you confirm that the server is returning plausibly correct data, there must be something else modifying the contents in between build and delivery of the file. To investigate this:

- Examine the build toolchain and deployment mechanism in case they're modifying files after the files are built. An example of this is when Git transforms file line endings, as described earlier.
- Examine the web server or CDN configuration in case they're set up to modify responses dynamically (for example, trying to minify HTML). It's fine for the web server to implement HTTP compression (for example, returning `content-encoding: br` or `content-encoding: gzip`), since this doesn't affect the result after decompression. However, it's *not* fine for the web server to modify the uncompressed data.

Troubleshoot integrity PowerShell script

Use the integrity.ps1 PowerShell script to validate a published and deployed Blazor app. The script is provided for PowerShell Core 7 or later as a starting point when the app has integrity issues that the Blazor framework can't identify. Customization of the script might be required for your apps, including if running on version of PowerShell later than version 7.2.0.

The script checks the files in the `publish` folder and downloaded from the deployed app to detect issues in the different manifests that contain integrity hashes. These checks should detect the most common problems:

- You modified a file in the published output without realizing it.
- The app wasn't correctly deployed to the deployment target, or something changed within the deployment target's environment.
- There are differences between the deployed app and the output from publishing the app.

Invoke the script with the following command in a PowerShell command shell:

PowerShell

```
.\integrity.ps1 {BASE URL} {PUBLISH OUTPUT FOLDER}
```

In the following example, the script is executed on a locally-running app at `https://localhost:5001/`:

PowerShell

```
.\integrity.ps1 https://localhost:5001/
C:\TestApps\BlazorSample\bin\Release\net6.0\publish\
```

Placeholders:

- `{BASE URL}`: The URL of the deployed app. A trailing slash (`/`) is required.
- `{PUBLISH OUTPUT FOLDER}`: The path to the app's `publish` folder or location where the app is published for deployment.

Note

When cloning the `dotnet/AspNetCore.Docs` GitHub repository, the `integrity.ps1` script might be quarantined by Bitdefender or another virus scanner present on the system. Usually, the file is trapped by a virus scanner's *heuristic scanning* technology, which merely looks for patterns in files that might indicate the presence of malware. To prevent the virus scanner from quarantining the file, add an exception to the virus scanner prior to cloning the repo. The following example is a typical path to the script on a Windows system. Adjust the path as needed for other systems. The `{USER}` placeholder is the user's path segment.

```
C:\Users\
{USER}\Documents\GitHub\AspNetCore.Docs\aspnetcore\blazor\host-and-
deploy\webassembly\_samples\integrity.ps1
```

Warning: *Creating virus scanner exceptions is dangerous and should only be performed when you're certain that the file is safe.*

Comparing the checksum of a file to a valid checksum value doesn't guarantee file safety, but modifying a file in a way that maintains a checksum value isn't trivial for malicious users. Therefore, checksums are useful as a general security approach. Compare the checksum of the local `integrity.ps1` file to one of the following values:

- SHA256: `32c24cb667d79a701135cb72f6bae490d81703323f61b8af2c7e5e5dc0f0c2bb`
- MD5: `9cee7d7ec86ee809a329b5406fbf21a8`

Obtain the file's checksum on Windows OS with the following command. Provide the path and file name for the `{PATH AND FILE NAME}` placeholder and indicate the type of checksum to produce for the `{SHA512|MD5}` placeholder, either `SHA256` or `MD5`:

Console

```
CertUtil -hashfile {PATH AND FILE NAME} {SHA256|MD5}
```

If you have any cause for concern that checksum validation isn't secure enough in your environment, consult your organization's security leadership for guidance.

For more information, see Overview of threat protection by Microsoft Defender Antivirus.

Disable integrity checking for non-PWA apps

In most cases, don't disable integrity checking. Disabling integrity checking doesn't solve the underlying problem that has caused the unexpected responses and results in losing the benefits listed earlier.

There may be cases where the web server can't be relied upon to return consistent responses, and you have no choice but to temporarily disable integrity checks until the underlying problem is resolved.

To disable integrity checks, add the following to a property group in the Blazor WebAssembly app's project file (`.csproj`):

```xml
<BlazorCacheBootResources>false</BlazorCacheBootResources>
```

`BlazorCacheBootResources` also disables Blazor's default behavior of caching the `.dll`, `.wasm`, and other files based on their SHA-256 hashes because the property indicates that the SHA-256 hashes can't be relied upon for correctness. Even with this setting, the browser's normal HTTP cache may still cache those files, but whether or not this happens depends on your web server configuration and the `cache-control` headers that it serves.

> **Note**
>
> The `BlazorCacheBootResources` property doesn't disable integrity checks for Progressive Web Applications (PWAs). For guidance pertaining to PWAs, see the Disable integrity checking for PWAs section.

We can't provide an exhaustive list of scenarios where disabling integrity checking is required. Servers can answer a request in arbitrary ways outside of the scope of the Blazor framework. The framework provides the `BlazorCacheBootResources` setting to make the app runnable at the cost of *losing a guarantee of integrity that the app can provide*. Again, we don't recommend disabling integrity checking, especially for production deployments. Developers should seek to solve the underlying integrity problem that's causing integrity checking to fail.

A few general cases that can cause integrity issues are:

- Running on HTTP where integrity can't be checked.
- If your deployment process modifies the files after publish in any way.
- If your host modifies the files in any way.

Disable integrity checking for PWAs

Blazor's Progressive Web Application (PWA) template contains a suggested `service-worker.published.js` file that's responsible for fetching and storing application files for offline use. This is a separate process from the normal app startup mechanism and has its own separate integrity checking logic.

Inside the `service-worker.published.js` file, following line is present:

```JavaScript
.map(asset => new Request(asset.url, { integrity: asset.hash }));
```

To disable integrity checking, remove the `integrity` parameter by changing the line to the following:

```JavaScript
.map(asset => new Request(asset.url));
```

Again, disabling integrity checking means that you lose the safety guarantees offered by integrity checking. For example, there is a risk that if the user's browser is caching the app at the exact moment that you deploy a new version, it could cache some files from the old deployment and some from the new deployment. If that happens, the app becomes stuck in a broken state until you deploy a further update.

ASP.NET Core Blazor WebAssembly .NET runtime and app bundle caching

Article • 07/09/2024

> **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.

When a Blazor WebAssembly app loads in the browser, the app downloads boot resources from the server:

- JavaScript code to bootstrap the app
- .NET runtime and assemblies
- Locale specific data

Except for Blazor's boot resources file (`blazor.boot.json`), WebAssembly .NET runtime and app bundle files are cached on clients. The `blazor.boot.json` file contains a manifest of the files that make up the app that must be downloaded along with a hash of the file's content that's used to detect whether any of the boot resources have changed. Blazor caches downloaded files using the browser Cache API.

When Blazor WebAssembly downloads an app's startup files, it instructs the browser to perform integrity checks on the responses. Blazor sends SHA-256 hash values for DLL (`.dll`), WebAssembly (`.wasm`), and other files in the `blazor.boot.json` file. The file hashes of cached files are compared to the hashes in the `blazor.boot.json` file. For cached files with a matching hash, Blazor uses the cached files. Otherwise, files are requested from the server. After a file is downloaded, its hash is checked again for integrity validation. An error is generated by the browser if any downloaded file's integrity check fails.

Blazor's algorithm for managing file integrity:

- Ensures that the app doesn't risk loading an inconsistent set of files, for example if a new deployment is applied to your web server while the user is in the process of downloading the application files. Inconsistent files can result in a malfunctioning app.

- Ensures the user's browser never caches inconsistent or invalid responses, which can prevent the app from starting even if the user manually refreshes the page.
- Makes it safe to cache the responses and not check for server-side changes until the expected SHA-256 hashes themselves change, so subsequent page loads involve fewer requests and complete faster.

If the web server returns responses that don't match the expected SHA-256 hashes, an error similar to the following example appears in the browser's developer console:

> Failed to find a valid digest in the 'integrity' attribute for resource 'https://myapp.example.com/_framework/MyBlazorApp.dll' with computed SHA-256 integrity 'IIa70iwvmEg5WiDV17OpQ5eCztNYqL186J56852RpJY='. The resource has been blocked.

In most cases, the warning doesn't indicate a problem with integrity checking. Instead, the warning usually means that some other problem exists.

> **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).

Diagnosing integrity problems

When an app is built, the generated `blazor.boot.json` manifest describes the SHA-256 hashes of boot resources at the time that the build output is produced. The integrity check passes as long as the SHA-256 hashes in `blazor.boot.json` match the files delivered to the browser.

Common reasons why this fails include:

- The web server's response is an error (for example, a *404 - Not Found* or a *500 - Internal Server Error*) instead of the file the browser requested. This is reported by the browser as an integrity check failure and not as a response failure.
- Something has changed the contents of the files between the build and delivery of the files to the browser. This might happen:
  - If you or build tools manually modify the build output.

  - If some aspect of the deployment process modified the files. For example if you use a Git-based deployment mechanism, bear in mind that Git transparently converts Windows-style line endings to Unix-style line endings if you commit files on Windows and check them out on Linux. Changing file line endings change the SHA-256 hashes. To avoid this problem, consider using .gitattributes to treat build artifacts as binary files.
  - The web server modifies the file contents as part of serving them. For example, some content distribution networks (CDNs) automatically attempt to minify HTML, thereby modifying it. You may need to disable such features.
- The `blazor.boot.json` file fails to load properly or is improperly cached on the client. Common causes include either of the following:
  - Misconfigured or malfunctioning custom developer code.
  - One or more misconfigured intermediate caching layers.

To diagnose which of these applies in your case:

1. Note which file is triggering the error by reading the error message.
2. Open your browser's developer tools and look in the *Network* tab. If necessary, reload the page to see the list of requests and responses. Find the file that is triggering the error in that list.
3. Check the HTTP status code in the response. If the server returns anything other than *200 - OK* (or another 2xx status code), then you have a server-side problem to diagnose. For example, status code 403 means there's an authorization problem, whereas status code 500 means the server is failing in an unspecified manner. Consult server-side logs to diagnose and fix the app.
4. If the status code is *200 - OK* for the resource, look at the response content in browser's developer tools and check that the content matches up with the data expected. For example, a common problem is to misconfigure routing so that requests return your `index.html` data even for other files. Make sure that responses to `.wasm` requests are WebAssembly binaries and that responses to `.dll` requests are .NET assembly binaries. If not, you have a server-side routing problem to diagnose.
5. Seek to validate the app's published and deployed output with the Troubleshoot integrity PowerShell script.

If you confirm that the server is returning plausibly correct data, there must be something else modifying the contents in between build and delivery of the file. To investigate this:

- Examine the build toolchain and deployment mechanism in case they're modifying files after the files are built. An example of this is when Git transforms file line endings, as described earlier.

- Examine the web server or CDN configuration in case they're set up to modify responses dynamically (for example, trying to minify HTML). It's fine for the web server to implement HTTP compression (for example, returning `content-encoding: br` or `content-encoding: gzip`), since this doesn't affect the result after decompression. However, it's *not* fine for the web server to modify the uncompressed data.

Troubleshoot integrity PowerShell script

Use the integrity.ps1 PowerShell script to validate a published and deployed Blazor app. The script is provided for PowerShell Core 7 or later as a starting point when the app has integrity issues that the Blazor framework can't identify. Customization of the script might be required for your apps, including if running on version of PowerShell later than version 7.2.0.

The script checks the files in the `publish` folder and downloaded from the deployed app to detect issues in the different manifests that contain integrity hashes. These checks should detect the most common problems:

- You modified a file in the published output without realizing it.
- The app wasn't correctly deployed to the deployment target, or something changed within the deployment target's environment.
- There are differences between the deployed app and the output from publishing the app.

Invoke the script with the following command in a PowerShell command shell:

PowerShell

```
.\integrity.ps1 {BASE URL} {PUBLISH OUTPUT FOLDER}
```

In the following example, the script is executed on a locally-running app at `https://localhost:5001/`:

PowerShell

```
.\integrity.ps1 https://localhost:5001/
C:\TestApps\BlazorSample\bin\Release\net6.0\publish\
```

Placeholders:

- `{BASE URL}`: The URL of the deployed app. A trailing slash (`/`) is required.

- `{PUBLISH OUTPUT FOLDER}`: The path to the app's `publish` folder or location where the app is published for deployment.

ⓘ **Note**

When cloning the `dotnet/AspNetCore.Docs` GitHub repository, the `integrity.ps1` script might be quarantined by Bitdefender or another virus scanner present on the system. Usually, the file is trapped by a virus scanner's *heuristic scanning* technology, which merely looks for patterns in files that might indicate the presence of malware. To prevent the virus scanner from quarantining the file, add an exception to the virus scanner prior to cloning the repo. The following example is a typical path to the script on a Windows system. Adjust the path as needed for other systems. The `{USER}` placeholder is the user's path segment.

```
C:\Users\
{USER}\Documents\GitHub\AspNetCore.Docs\aspnetcore\blazor\host-and-
deploy\webassembly\_samples\integrity.ps1
```

Warning: *Creating virus scanner exceptions is dangerous and should only be performed when you're certain that the file is safe.*

Comparing the checksum of a file to a valid checksum value doesn't guarantee file safety, but modifying a file in a way that maintains a checksum value isn't trivial for malicious users. Therefore, checksums are useful as a general security approach. Compare the checksum of the local `integrity.ps1` file to one of the following values:

- SHA256: `32c24cb667d79a701135cb72f6bae490d81703323f61b8af2c7e5e5dc0f0c2bb`
- MD5: `9cee7d7ec86ee809a329b5406fbf21a8`

Obtain the file's checksum on Windows OS with the following command. Provide the path and file name for the `{PATH AND FILE NAME}` placeholder and indicate the type of checksum to produce for the `{SHA512|MD5}` placeholder, either `SHA256` or `MD5`:

Console

```
CertUtil -hashfile {PATH AND FILE NAME} {SHA256|MD5}
```

If you have any cause for concern that checksum validation isn't secure enough in your environment, consult your organization's security leadership for guidance.

For more information, see Overview of threat protection by Microsoft Defender Antivirus.

Disable resource caching and integrity checks for non-PWA apps

In most cases, don't disable integrity checking. Disabling integrity checking doesn't solve the underlying problem that has caused the unexpected responses and results in losing the benefits listed earlier.

There may be cases where the web server can't be relied upon to return consistent responses, and you have no choice but to temporarily disable integrity checks until the underlying problem is resolved.

To disable integrity checks, add the following to a property group in the Blazor WebAssembly app's project file (`.csproj`):

XML

```xml
<BlazorCacheBootResources>false</BlazorCacheBootResources>
```

`BlazorCacheBootResources` also disables Blazor's default behavior of caching the `.dll`, `.wasm`, and other files based on their SHA-256 hashes because the property indicates that the SHA-256 hashes can't be relied upon for correctness. Even with this setting, the browser's normal HTTP cache may still cache those files, but whether or not this happens depends on your web server configuration and the `cache-control` headers that it serves.

ⓘ Note

The `BlazorCacheBootResources` property doesn't disable integrity checks for Progressive Web Applications (PWAs). For guidance pertaining to PWAs, see the Disable integrity checking for PWAs section.

We can't provide an exhaustive list of scenarios where disabling integrity checking is required. Servers can answer a request in arbitrary ways outside of the scope of the Blazor framework. The framework provides the `BlazorCacheBootResources` setting to make the app runnable at the cost of *losing a guarantee of integrity that the app can provide*. Again, we don't recommend disabling integrity checking, especially for production deployments. Developers should seek to solve the underlying integrity problem that's causing integrity checking to fail.

A few general cases that can cause integrity issues are:

- Running on HTTP where integrity can't be checked.
- If your deployment process modifies the files after publish in any way.
- If your host modifies the files in any way.

Disable resource caching and integrity checks for PWAs

Blazor's Progressive Web Application (PWA) template contains a suggested `service-worker.published.js` file that's responsible for fetching and storing application files for offline use. This is a separate process from the normal app startup mechanism and has its own separate integrity checking logic.

Inside the `service-worker.published.js` file, following line is present:

JavaScript

```javascript
.map(asset => new Request(asset.url, { integrity: asset.hash }));
```

To disable integrity checking, remove the `integrity` parameter by changing the line to the following:

JavaScript

```javascript
.map(asset => new Request(asset.url));
```

Again, disabling integrity checking means that you lose the safety guarantees offered by integrity checking. For example, there is a risk that if the user's browser is caching the app at the exact moment that you deploy a new version, it could cache some files from the old deployment and some from the new deployment. If that happens, the app becomes stuck in a broken state until you deploy a further update.

Additional resources

Boot resource loading

Avoid HTTP caching issues when upgrading ASP.NET Core Blazor apps

Article • 04/05/2024

ⓘ **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.

When Blazor apps are incorrectly upgraded or configured, it can result in non-seamless upgrades for existing users. This article discusses some of the common HTTP caching issues that can occur when upgrading Blazor apps across major versions. It also provides some recommended actions to ensure a smooth transition for your users.

While future Blazor releases might provide better solutions for dealing with HTTP caching issues, it's ultimately up to the app to correctly configure caching. Proper caching configuration ensures that the app's users always have the most up-to-date version of the app, improving their experience and reducing the likelihood of encountering errors.

Common problems that negatively impact the user upgrade experience include:

- **Incorrect handling of project and package updates**: This happens if you don't update all of the app's deployed projects to use the same major framework version or if you use packages from a previous version when a newer version is available as part of the major upgrade.
- **Incorrect configuration of caching headers**: HTTP caching headers control how, where, and for how long the app's responses are cached. If headers aren't configured correctly, users might receive stale content.
- **Incorrect configuration of other layers**: Content Delivery Networks (CDNs) and other layers of the deployed app can cause issues if incorrectly configured. For example, CDNs are designed to cache and deliver content to improve performance and reduce latency. If a CDN is incorrectly serving cached versions of assets, it can lead to stale content delivery to the user.

Detect and diagnose upgrade issues

Upgrade issues typically appear as a failure to start the app in the browser. Normally, a warning indicates the presence of a stale asset or an asset that's missing or inconsistent with the app.

- First, check if the app loads successfully within a clean browser instance. Use a private browser mode to load the app, such as Microsoft Edge InPrivate mode or Google Chrome Incognito mode. If the app fails to load, it likely means that one or more packages or the framework wasn't correctly updated.
- If the app loads correctly in a clean browser instance, then it's likely that the app is being served from a stale cache. In most cases, a hard browser refresh with `Ctrl`+`F5` flushes the cache, which permits the app to load and run with the latest assets.
- If the app continues to fail, then it's likely that a stale CDN cache is serving the app. Try to flush the DNS cache via whatever mechanism your CDN provider offers.

Recommended actions before an upgrade

The prior process for serving the app might make the update process more challenging. For example, avoiding or incorrectly using caching headers in the past can lead to current caching problems for users. You can take the actions in the following sections to mitigate the issue and improve the upgrade process for users.

Align framework packages with the framework version

Ensure that framework packages line up with the framework version. Using packages from a previous version when a newer version is available can lead to compatibility issues. It's also important to ensure that all of the app's deployed projects use the same major framework version. This consistency helps to avoid unexpected behavior and errors.

Verify the presence of correct caching headers

The correct caching headers should be present on responses to resource requests. This includes `ETag`, `Cache-Control`, and other caching headers. The configuration of these headers is dependent on the hosting service or hosting server platform. They are particularly important for assets such as the Blazor script (`blazor.webassembly.js`) and anything the script downloads.

Incorrect HTTP caching headers may also impact service workers. Service workers rely on caching headers to manage cached resources effectively. Therefore, incorrect or missing headers can disrupt the service worker's functionality.

Use `Clear-Site-Data` to delete state in the browser

Consider using the Clear-Site-Data header to delete state in the browser.

Usually the source of cache state problems is limited to the HTTP browser cache, so use of the `cache` directive should be sufficient. This action can help to ensure that the browser fetches the latest resources from the server, rather than serving stale content from the cache.

You can optionally include the `storage` directive to clear local storage caches at the same time that you're clearing the HTTP browser cache. However, apps that use client storage might experience a loss of important information if the `storage` directive is used.

Append a query string to the Blazor script tag

If none of the previous recommended actions are effective, possible to use for your deployment, or apply to your app, consider temporarily appending a query string to the Blazor script's `<script>` tag source. This action should be enough in most situations to force the browser to bypass the local HTTP cache and download a new version of the app. There's no need to read or use the query string in the app.

In the following example, the query string `temporaryQueryString=1` is temporarily applied to the `<script>` tag's relative external source URI:

HTML

```html
<script src="_framework/blazor.webassembly.js?temporaryQueryString=1">
</script>
```

After all of the app's users have reloaded the app, the query string can be removed.

Alternatively, you can apply a persistent query string with relevant versioning. The following example assumes that the version of the app matches the .NET release version (`8` for .NET 8):

HTML

```html
<script src="_framework/blazor.webassembly.js?version=8"></script>
```

For the location of the Blazor script `<script>` tag, see ASP.NET Core Blazor project structure.

Configure the Trimmer for ASP.NET Core Blazor

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.

This article explains how to control the Intermediate Language (IL) Trimmer when building a Blazor app.

Blazor WebAssembly performs Intermediate Language (IL) trimming to reduce the size of the published output. Trimming occurs when publishing an app.

Trimming may have detrimental effects for the published app. In apps that use reflection, the IL Trimmer often can't determine the required types for runtime reflection and trim them away. For example, complex framework types for JS interop, such as KeyValuePair, might be trimmed and not available at runtime for JS interop calls. In these cases, we recommend creating your own custom types instead. The IL Trimmer is also unable to react to an app's dynamic behavior at runtime. To ensure the trimmed app works correctly once deployed, test published output frequently while developing.

Configuration

To configure the IL Trimmer, see the Trimming options article in the .NET Fundamentals documentation, which includes guidance on the following subjects:

- Disable trimming for the entire app with the `<PublishTrimmed>` property in the project file.
- Control how aggressively unused IL is discarded by the IL Trimmer.
- Stop the IL Trimmer from trimming specific assemblies.
- "Root" assemblies for trimming.
- Surface warnings for reflected types by setting the `<SuppressTrimAnalysisWarnings>` property to `false` in the project file.
- Control symbol trimming and debugger support.

- Set IL Trimmer features for trimming framework library features.

Default trimmer granularity

The default trimmer granularity for Blazor apps is `partial`. To trim all assemblies, change the granularity to `full` in the app's project file:

XML

```xml
<ItemGroup>
  <TrimMode>full</TrimMode>
</ItemGroup>
```

For more information, see Trimming options (.NET documentation).

Additional resources

- Trim self-contained deployments and executables
- ASP.NET Core Blazor performance best practices

Deployment layout for ASP.NET Core hosted Blazor WebAssembly apps

Article • 09/12/2024

This article explains how to enable hosted Blazor WebAssembly deployments in environments that block the download and execution of dynamic-link library (DLL) files.

ⓘ **Note**

This guidance addresses environments that block clients from downloading and executing DLLs. In .NET 8 or later, Blazor uses the Webcil file format to address this problem. For more information, see Host and deploy ASP.NET Core Blazor WebAssembly. Multipart bundling using the experimental NuGet package described by this article isn't supported for Blazor apps in .NET 8 or later. You can use the guidance in this article to create your own multipart bundling NuGet package for .NET 8 or later.

Blazor WebAssembly apps require dynamic-link libraries (DLLs) to function, but some environments block clients from downloading and executing DLLs. In a subset of these environments, changing the file name extension of DLL files (.dll) is sufficient to bypass security restrictions, but security products are often able to scan the content of files traversing the network and block or quarantine DLL files. This article describes one approach for enabling Blazor WebAssembly apps in these environments, where a multipart bundle file is created from the app's DLLs so that the DLLs can be downloaded together bypassing security restrictions.

A hosted Blazor WebAssembly app can customize its published files and packaging of app DLLs using the following features:

- JavaScript initializers that allow customizing the Blazor boot process.
- MSBuild extensibility to transform the list of published files and define *Blazor Publish Extensions*. Blazor Publish Extensions are files defined during the publish process that provide an alternative representation for the set of files required to run a published Blazor WebAssembly app. In this article, a Blazor Publish Extension is created that produces a multipart bundle with all of the app's DLLs packed into a single file so that the DLLs can be downloaded together.

The approach demonstrated in this article serves as a starting point for developers to devise their own strategies and custom loading processes.

⚠ Warning

Any approach taken to circumvent a security restriction must be carefully considered for its security implications. We recommend exploring the subject further with your organization's network security professionals before adopting the approach in this article. Alternatives to consider include:

- Enable security appliances and security software to permit network clients to download and use the exact files required by a Blazor WebAssembly app.
- Switch from the Blazor WebAssembly hosting model to the **Blazor Server hosting model**, which maintains all of the app's C# code on the server and doesn't require downloading DLLs to clients. Blazor Server also offers the advantage of keeping C# code private without requiring the use of web API apps for C# code privacy with Blazor WebAssembly apps.

Experimental NuGet package and sample app

The approach described in this article is used by the *experimental* Microsoft.AspNetCore.Components.WebAssembly.MultipartBundle package (NuGet.org) for apps targeting .NET 6 or later. The package contains MSBuild targets to customize the Blazor publish output and a JavaScript initializer to use a custom boot resource loader, each of which are described in detail later in this article.

Experimental code (includes the NuGet package reference source and CustomPackagedApp sample app)

⚠ Warning

Experimental and preview features are provided for the purpose of collecting feedback and aren't supported for production use.

Later in this article, the Customize the Blazor WebAssembly loading process via a NuGet package section with its three subsections provide detailed explanations on the configuration and code in the `Microsoft.AspNetCore.Components.WebAssembly.MultipartBundle` package. The detailed explanations are important to understand when you create your own strategy and custom loading process for Blazor WebAssembly apps. To use the published, experimental, unsupported NuGet package without customization as a **local demonstration**, perform the following steps:

1. Use an existing hosted Blazor WebAssembly solution or create a new solution from the Blazor WebAssembly project template using Visual Studio or by passing the -ho|--hosted option to the dotnet new command (`dotnet new blazorwasm -ho`). For more information, see Tooling for ASP.NET Core Blazor.

2. In the **Client** project, add the experimental `Microsoft.AspNetCore.Components.WebAssembly.MultipartBundle` 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.

3. In the **Server** project, add an endpoint for serving the bundle file (`app.bundle`). Example code can be found in the Serve the bundle from the host server app section of this article.

4. Publish the app in Release configuration.

Customize the Blazor WebAssembly loading process via a NuGet package

> Warning
>
> The guidance in this section with its three subsections pertains to building a NuGet package from scratch to implement your own strategy and custom loading process. The *experimental* Microsoft.AspNetCore.Components.WebAssembly.MultipartBundle package (NuGet.org) for .NET 6 and 7 is based on the guidance in this section. When using the provided package in a **local demonstration** of the multipart bundle download approach, you don't need to follow the guidance in this section. For guidance on how to use the provided package, see the Experimental NuGet package and sample app section.

Blazor app resources are packed into a multipart bundle file and loaded by the browser via a custom JavaScript (JS) initializer. For an app consuming the package with the JS initializer, the app only requires that the bundle file is served when requested. All of the other aspects of this approach are handled transparently.

Four customizations are required to how a default published Blazor app loads:

- An MSBuild task to transform the publish files.
- A NuGet package with MSBuild targets that hooks into the Blazor publishing process, transforms the output, and defines one or more Blazor Publish Extension files (in this case, a single bundle).
- A JS initializer to update the Blazor WebAssembly resource loader callback so that it loads the bundle and provides the app with the individual files.
- A helper on the host **Server** app to ensure that the bundle is served to clients on request.

Create an MSBuild task to customize the list of published files and define new extensions

Create an MSBuild task as a public C# class that can be imported as part of an MSBuild compilation and that can interact with the build.

The following are required for the C# class:

- A new class library project.
- A project target framework of `netstandard2.0`.
- References to MSBuild packages:
  - Microsoft.Build.Framework
  - Microsoft.Build.Utilities.Core

> **Note**
>
> The NuGet package for the examples in this article are named after the package provided by Microsoft, `Microsoft.AspNetCore.Components.WebAssembly.MultipartBundle`. For guidance on naming and producing your own NuGet package, see the following NuGet articles:
>
> - Package authoring best practices
> - Package ID prefix reservation

`Microsoft.AspNetCore.Components.WebAssembly.MultipartBundle.Tasks/Microsoft.AspNetCore.Components.WebAssembly.MultipartBundle.Tasks.csproj`:

XML

```xml
<Project Sdk="Microsoft.NET.Sdk">
```

```
  <PropertyGroup>
    <TargetFramework>netstandard2.0</TargetFramework>
    <LangVersion>8.0</LangVersion>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.Build.Framework" Version="
{VERSION}" />
    <PackageReference Include="Microsoft.Build.Utilities.Core" Version="
{VERSION}" />
  </ItemGroup>

</Project>
```

Determine the latest package versions for the `{VERSION}` placeholders at NuGet.org:

- Microsoft.Build.Framework
- Microsoft.Build.Utilities.Core

To create the MSBuild task, create a public C# class extending Microsoft.Build.Utilities.Task (not System.Threading.Tasks.Task) and declare three properties:

- `PublishBlazorBootStaticWebAsset`: The list of files to publish for the Blazor app.
- `BundlePath`: The path where the bundle is written.
- `Extension`: The new Publish Extensions to include in the build.

The following example `BundleBlazorAssets` class is a starting point for further customization:

- In the `Execute` method, the bundle is created from the following three file types:
  - JavaScript files (`dotnet.js`)
  - WASM files (`dotnet.wasm`)
  - App DLLs (`.dll`)
- A `multipart/form-data` bundle is created. Each file is added to the bundle with its respective descriptions via the Content-Disposition header and the Content-Type header.
- After the bundle is created, the bundle is written to a file.
- The build is configured for the extension. The following code creates an extension item and adds it to the `Extension` property. Each extension item contains three pieces of data:
  - The path to the extension file.
  - The URL path relative to the root of the Blazor WebAssembly app.
  - The name of the extension, which groups the files produced by a given extension.

After accomplishing the preceding goals, the MSBuild task is created for customizing the Blazor publish output. Blazor takes care of gathering the extensions and making sure that the extensions are copied to the correct location in the publish output folder (for example, `bin\Release\net6.0\publish`). The same optimizations (for example, compression) are applied to the JavaScript, WASM, and DLL files as Blazor applies to other files.

`Microsoft.AspNetCore.Components.WebAssembly.MultipartBundle.Tasks/BundleBlazorAssets.cs`:

C#

```csharp
using System.IO;
using System.Net.Http;
using System.Net.Http.Headers;
using Microsoft.Build.Framework;
using Microsoft.Build.Utilities;

namespace Microsoft.AspNetCore.Components.WebAssembly.MultipartBundle.Tasks
{
    public class BundleBlazorAssets : Task
    {
        [Required]
        public ITaskItem[]? PublishBlazorBootStaticWebAsset { get; set; }

        [Required]
        public string? BundlePath { get; set; }

        [Output]
        public ITaskItem[]? Extension { get; set; }

        public override bool Execute()
        {
            var bundle = new MultipartFormDataContent(
                "--0a7e8441d64b4bf89086b85e59523b7d");

            foreach (var asset in PublishBlazorBootStaticWebAsset)
            {
                var name = 
Path.GetFileName(asset.GetMetadata("RelativePath"));
                var fileContents = File.OpenRead(asset.ItemSpec);
                var content = new StreamContent(fileContents);
                var disposition = new ContentDispositionHeaderValue("form-
data");
                disposition.Name = name;
                disposition.FileName = name;
                content.Headers.ContentDisposition = disposition;
                var contentType = Path.GetExtension(name) switch
                {
                    ".js" => "text/javascript",
                    ".wasm" => "application/wasm",
                    _ => "application/octet-stream"
                };
                content.Headers.ContentType =
                    MediaTypeHeaderValue.Parse(contentType);
                bundle.Add(content);
            }

            using (var output = File.Open(BundlePath,
FileMode.OpenOrCreate))
            {
                output.SetLength(0);

bundle.CopyToAsync(output).ConfigureAwait(false).GetAwaiter()
                    .GetResult();
                output.Flush(true);
            }

            var bundleItem = new TaskItem(BundlePath);
            bundleItem.SetMetadata("RelativePath", "app.bundle");
            bundleItem.SetMetadata("ExtensionName", "multipart");

            Extension = new ITaskItem[] { bundleItem };

            return true;
        }
    }
}
```

Author a NuGet package to automatically transform the publish output

Generate a NuGet package with MSBuild targets that are automatically included when the package is referenced:

- Create a new Razor class library (RCL) project.
- Create a targets file following NuGet conventions to automatically import the package in consuming projects. For example, create `build\net6.0\{PACKAGE ID}.targets`, where `{PACKAGE ID}` is the package identifier of the package.
- Collect the output from the class library containing the MSBuild task and confirm the output is packed in the right location.
- Add the necessary MSBuild code to attach to the Blazor pipeline and invoke the MSBuild task to generate the bundle.

The approach described in this section only uses the package to deliver targets and content, which is different from most packages where the package includes a library DLL.

⚠ Warning

The sample package described in this section demonstrates how to customize the Blazor publish process. **The sample NuGet package is for use as a local demonstration only. Using this package in production is not supported.**

ⓘ Note

The NuGet package for the examples in this article are named after the package provided by Microsoft, `Microsoft.AspNetCore.Components.WebAssembly.MultipartBundle`. For guidance on naming and producing your own NuGet package, see the following NuGet articles:

- Package authoring best practices
- Package ID prefix reservation

`Microsoft.AspNetCore.Components.WebAssembly.MultipartBundle/Microsoft.AspNetCore.Components.WebAssembly.MultipartBundle.csproj`:

XML

```xml
<Project Sdk="Microsoft.NET.Sdk.Razor">

  <PropertyGroup>
    <NoWarn>NU5100</NoWarn>
    <TargetFramework>net6.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
    <Description>
      Sample demonstration package showing how to customize the Blazor
publish
      process. Using this package in production is not supported!
    </Description>
    <IsPackable>true</IsPackable>
    <IsShipping>true</IsShipping>
    <IncludeBuildOutput>false</IncludeBuildOutput>
  </PropertyGroup>

  <ItemGroup>
    <None Update="build\**"
          Pack="true"
          PackagePath="%(Identity)" />
    <Content Include="_._"
             Pack="true"
             PackagePath="lib\net6.0\_._" />
  </ItemGroup>
  <Target Name="GetTasksOutputDlls"
          BeforeTargets="CoreCompile">
    <MSBuild
Projects="..\Microsoft.AspNetCore.Components.WebAssembly.MultipartBundle.Tasks\Microsoft.AspNetCore.Components.WebAssembly.MultipartBundle.Tasks.csproj"
             Targets="Publish;PublishItemsOutputGroup"
             Properties="Configuration=Release">
      <Output TaskParameter="TargetOutputs"
              ItemName="_TasksProjectOutputs" />
    </MSBuild>
    <ItemGroup>
      <Content Include="@(_TasksProjectOutputs)"
               Condition="'%(_TasksProjectOutputs.Extension)' == '.dll'"
               Pack="true"
               PackagePath="tasks\%(_TasksProjectOutputs.TargetPath)"
               KeepMetadata="Pack;PackagePath" />
    </ItemGroup>
  </Target>

</Project>
```

ⓘ **Note**

The `<NoWarn>NU5100</NoWarn>` property in the preceding example suppresses the warning about the assemblies placed in the `tasks` folder. For more information, see NuGet Warning NU5100.

Add a `.targets` file to wire up the MSBuild task to the build pipeline. In this file, the following goals are accomplished:

- Import the task into the build process. Note that the path to the DLL is relative to the ultimate location of the file in the package.
- The `ComputeBlazorExtensionsDependsOn` property attaches the custom target to the Blazor WebAssembly pipeline.
- Capture the `Extension` property on the task output and add it to `BlazorPublishExtension` to tell Blazor about the extension. Invoking the task in the target produces the bundle. The list of published files is provided by the Blazor WebAssembly pipeline in the `PublishBlazorBootStaticWebAsset` item group. The bundle path is defined using the `IntermediateOutputPath` (typically inside the `obj` folder). Ultimately, the bundle is copied automatically to the correct location in the publish output folder (for example, `bin\Release\net6.0\publish`).

When the package is referenced, it generates a bundle of the Blazor files during publish.

`Microsoft.AspNetCore.Components.WebAssembly.MultipartBundle/build/net6.0/Microsoft.AspNetCore.Components.WebAssembly.MultipartBundle.targets`:

XML

```xml
<Project>
  <UsingTask

TaskName="Microsoft.AspNetCore.Components.WebAssembly.MultipartBundle.Tasks.
BundleBlazorAssets"

AssemblyFile="$(MSBuildThisProjectFileDirectory)..\..\tasks\Microsoft.AspNet
Core.Components.WebAssembly.MultipartBundle.Tasks.dll" />

  <PropertyGroup>
    <ComputeBlazorExtensionsDependsOn>
      $(ComputeBlazorExtensionsDependsOn);_BundleBlazorDlls
    </ComputeBlazorExtensionsDependsOn>
  </PropertyGroup>

  <Target Name="_BundleBlazorDlls">
    <BundleBlazorAssets
      PublishBlazorBootStaticWebAsset="@(PublishBlazorBootStaticWebAsset)"
      BundlePath="$(IntermediateOutputPath)bundle.multipart">
      <Output TaskParameter="Extension"
              ItemName="BlazorPublishExtension"/>
    </BundleBlazorAssets>
  </Target>

</Project>
```

Automatically bootstrap Blazor from the bundle

The NuGet package leverages JavaScript (JS) initializers to automatically bootstrap a Blazor WebAssembly app from the bundle instead of using individual DLL files. JS initializers are used to change the Blazor boot resource loader and use the bundle.

To create a JS initializer, add a JS file with the name `{NAME}.lib.module.js` to the `wwwroot` folder of the package project, where the `{NAME}` placeholder is the package identifier. For example, the file for the Microsoft package is named `Microsoft.AspNetCore.Components.WebAssembly.MultipartBundle.lib.module.js`. The exported functions `beforeWebAssemblyStart` and `afterWebAssemblyStarted` handle loading.

The JS initializers:

- Detect if the Publish Extension is available by checking for `extensions.multipart`, which is the extension name (`ExtensionName`) provided in the Create an MSBuild task to customize the list of published files and define new extensions section.
- Download the bundle and parse the contents into a resources map using generated object URLs.
- Update the boot resource loader (options.loadBootResource) with a custom function that resolves the resources using the object URLs.
- After the app has started, revoke the object URLs to release memory in the `afterWebAssemblyStarted` function.

`Microsoft.AspNetCore.Components.WebAssembly.MultipartBundle/wwwroot/Microsoft.AspNetCore.Components.WebAssembly.MultipartBundle.lib.module.js`:

JavaScript

```javascript
const resources = new Map();

export async function beforeWebAssemblyStart(options, extensions) {
  if (!extensions || !extensions.multipart) {
    return;
  }

  try {
    const integrity = extensions.multipart['app.bundle'];
    const bundleResponse =
      await fetch('app.bundle', { integrity: integrity, cache: 'no-cache'
});
    const bundleFromData = await bundleResponse.formData();
    for (let value of bundleFromData.values()) {
      resources.set(value, URL.createObjectURL(value));
    }
    options.loadBootResource = function (type, name, defaultUri, integrity)
{
      return resources.get(name) ?? null;
    }
  } catch (error) {
    console.log(error);
  }
}

export async function afterWebAssemblyStarted(blazor) {
  for (const [_, url] of resources) {
    URL.revokeObjectURL(url);
  }
}
```

Serve the bundle from the host server app

Due to security restrictions, ASP.NET Core doesn't serve the `app.bundle` file. A request processing helper is required to serve the file when it's requested by clients.

> ⓘ **Note**
>
> Since the same optimizations are transparently applied to the Publish Extensions that are applied to the app's files, the `app.bundle.gz` and `app.bundle.br` compressed asset files are produced automatically on publish.

Place C# code in `Program.cs` of the **Server** project immediately before the line that sets the fallback file to `index.html` (`app.MapFallbackToFile("index.html");`) to respond to a request for the bundle file (for example, `app.bundle`):

C#

```csharp
app.MapGet("app.bundle", (HttpContext context) =>
{
    string? contentEncoding = null;
    var contentType =
        "multipart/form-data; boundary=\"-
-0a7e8441d64b4bf89086b85e59523b7d\"";
    var fileName = "app.bundle";

    var acceptEncodings = context.Request.Headers.AcceptEncoding;

    if (Microsoft.Net.Http.Headers.StringWithQualityHeaderValue
        .StringWithQualityHeaderValue
        .TryParseList(acceptEncodings, out var encodings))
    {
        if (encodings.Any(e => e.Value == "br"))
        {
            contentEncoding = "br";
            fileName += ".br";
        }
        else if (encodings.Any(e => e.Value == "gzip"))
        {
            contentEncoding = "gzip";
            fileName += ".gz";
        }
    }

    if (contentEncoding != null)
    {
        context.Response.Headers.ContentEncoding = contentEncoding;
    }

    return Results.File(
        app.Environment.WebRootFileProvider.GetFileInfo(fileName)
```

```
            .CreateReadStream(), contentType);
});
```

The content type matches the type defined earlier in the build task. The endpoint checks for the content encodings accepted by the browser and serves the optimal file, Brotli (`.br`) or Gzip (`.gz`).

ASP.NET Core Blazor with Entity Framework Core (EF Core)

Article • 10/28/2024

ⓘ **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 Entity Framework Core (EF Core) in server-side Blazor apps.

Server-side Blazor is a stateful app framework. The app maintains an ongoing connection to the server, and the user's state is held in the server's memory in a *circuit*. One example of user state is data held in dependency injection (DI) service instances that are scoped to the circuit. The unique application model that Blazor provides requires a special approach to use Entity Framework Core.

ⓘ **Note**

This article addresses EF Core in server-side Blazor apps. Blazor WebAssembly apps run in a WebAssembly sandbox that prevents most direct database connections. Running EF Core in Blazor WebAssembly is beyond the scope of this article.

This guidance applies to components that adopt interactive server-side rendering (interactive SSR) in a Blazor Web App.

Secure authentication flow required for production apps

This article uses a local database that doesn't require user authentication. Production apps should use the most secure authentication flow available. For more information on authentication for deployed test and production Blazor apps, see the articles in the Blazor *Security and Identity* node.

For Microsoft Azure services, we recommend using *managed identities*. Managed identities securely authenticate to Azure services without storing credentials in app code. For more information, see the following resources:

- What are managed identities for Azure resources? (Microsoft Entra documentation)
- Azure services documentation
  - Managed identities in Microsoft Entra for Azure SQL
  - How to use managed identities for App Service and Azure Functions

Sample app

The sample app was built as a reference for server-side Blazor apps that use EF Core. The sample app includes a grid with sorting and filtering, delete, add, and update operations.

The sample demonstrates use of EF Core to handle optimistic concurrency. However, native database-generated concurrency tokens aren't supported for SQLite databases, which is the database provider for the sample app. To demonstrate concurrency with the sample app, adopt a different database provider that supports database-generated concurrency tokens (for example, the SQL Server provider).

View or download sample code (how to download): Select the folder that matches the version of .NET that you're adopting. Within the version folder, access the sample named `BlazorWebAppEFCore`.

The sample uses a local SQLite database so that it can be used on any platform. The sample also configures database logging to show the SQL queries that are generated. This is configured in `appsettings.Development.json`:

JSON

```json
{
  "DetailedErrors": true,
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning",
      "Microsoft.Hosting.Lifetime": "Information",
      "Microsoft.EntityFrameworkCore.Database.Command": "Information"
    }
  }
}
```

The grid, add, and view components use the "context-per-operation" pattern, where a context is created for each operation. The edit component uses the "context-per-component" pattern, where a context is created for each component.

> ⓘ Note
>
> Some of the code examples in this topic require namespaces and services that aren't shown. To inspect the fully working code, including the required @using and @inject directives for Razor examples, see the sample app.

Build a Blazor movie database app tutorial

For a tutorial experience building an app that uses EF Core to work with a database, see Build a Blazor movie database app (Overview). The tutorial shows you how to create a Blazor Web App that can display and manage movies in a movie database.

Database access

EF Core relies on a DbContext as the means to configure database access and act as a *unit of work*. EF Core provides the AddDbContext extension for ASP.NET Core apps that registers the context as a *scoped* service. In server-side Blazor apps, scoped service registrations can be problematic because the instance is shared across components within the user's circuit. DbContext isn't thread safe and isn't designed for concurrent use. The existing lifetimes are inappropriate for these reasons:

- **Singleton** shares state across all users of the app and leads to inappropriate concurrent use.
- **Scoped** (the default) poses a similar issue between components for the same user.
- **Transient** results in a new instance per request; but as components can be long-lived, this results in a longer-lived context than may be intended.

The following recommendations are designed to provide a consistent approach to using EF Core in server-side Blazor apps.

- Consider using one context per operation. The context is designed for fast, low overhead instantiation:

  C#

  ```csharp
  using var context = new MyContext();

  return await context.MyEntities.ToListAsync();
  ```

- Use a flag to prevent multiple concurrent operations:

  C#

  ```csharp
  if (Loading)
  {
      return;
  }

  try
  {
      Loading = true;

      ...
  }
  finally
  {
      Loading = false;
  }
  ```

  Place operations after the `Loading = true;` line in the `try` block.

  Thread safety isn't a concern, so loading logic doesn't require locking database records. The loading logic is used to disable UI controls so that users don't inadvertently select buttons or update fields while data is fetched.

- If there's any chance that multiple threads may access the same code block, inject a factory and make a new instance per operation. Otherwise, injecting and using the context is usually sufficient.

- For longer-lived operations that take advantage of EF Core's change tracking or concurrency control, scope the context to the lifetime of the component.

New `DbContext` instances

The fastest way to create a new DbContext instance is by using `new` to create a new instance. However, there are scenarios that require resolving additional dependencies:

- Using DbContextOptions to configure the context.
- Using a connection string per DbContext, such as when you use ASP.NET Core's Identity model. For more information, see Multi-tenancy (EF Core documentation).

⚠ **Warning**

Don't store app secrets, connection strings, credentials, passwords, personal identification numbers (PINs), private C#/.NET code, or private keys/tokens in client-side code, which is ***always insecure***. In test/staging and production environments, server-side Blazor code and web APIs should use secure authentication flows that avoid maintaining credentials within project code or configuration files. Outside of local development testing, we recommend avoiding the use of environment variables to store sensitive data, as environment variables aren't the most secure approach. For local development testing, the Secret Manager tool is recommended for securing sensitive data. For more information, see Securely maintain sensitive data and credentials.

The recommended approach to create a new DbContext with dependencies is to use a factory. EF Core 5.0 or later provides a built-in factory for creating new contexts.

The following example configures SQLite and enables data logging. The code uses an extension method (`AddDbContextFactory`) to configure the database factory for DI and provide default options:

C#

```
builder.Services.AddDbContextFactory<ContactContext>(opt =>
    opt.UseSqlite($"Data Source={nameof(ContactContext.ContactsDb)}.db"));
```

The factory is injected into components and used to create new `DbContext` instances.

In the home page of the sample app, `IDbContextFactory<ContactContext>` is injected into the component:

razor

```
@inject IDbContextFactory<ContactContext> DbFactory
```

A `DbContext` is created using the factory (`DbFactory`) to delete a contact in the `DeleteContactAsync` method:

razor

```
private async Task DeleteContactAsync()
{
    using var context = DbFactory.CreateDbContext();
    Filters.Loading = true;

    if (Wrapper is not null && context.Contacts is not null)
    {
        var contact = await context.Contacts
            .FirstAsync(c => c.Id == Wrapper.DeleteRequestId);
        if (contact is not null)
        {
            context.Contacts?.Remove(contact);
            await context.SaveChangesAsync();
        }
    }

    Filters.Loading = false;

    await ReloadAsync();
}
```

ⓘ Note

`Filters` is an injected `IContactFilters`, and `Wrapper` is a component reference to the `GridWrapper` component. See the `Home` component (`Components/Pages/Home.razor`) in the sample app.

Scope to the component lifetime

You may wish to create a DbContext that exists for the lifetime of a component. This allows you to use it as a unit of work and take advantage of built-in features, such as change tracking and concurrency resolution.

You can use the factory to create a context and track it for the lifetime of the component. First, implement IDisposable and inject the factory as shown in the `EditContact` component (`Components/Pages/EditContact.razor`):

razor

```razor
@implements IDisposable
@inject IDbContextFactory<ContactContext> DbFactory
```

The sample app ensures the context is disposed when the component is disposed:

C#

```csharp
public void Dispose() => Context?.Dispose();
```

Finally, OnInitializedAsync is overridden to create a new context. In the sample app, OnInitializedAsync loads the contact in the same method:

C#

```
protected override async Task OnInitializedAsync()
{
    Busy = true;

    try
    {
        Context = DbFactory.CreateDbContext();

        if (Context is not null && Context.Contacts is not null)
        {
            var contact = await Context.Contacts.SingleOrDefaultAsync(c =>
c.Id == ContactId);

            if (contact is not null)
            {
                Contact = contact;
            }
        }
    }
    finally
    {
        Busy = false;
    }
}
```

Enable sensitive data logging

EnableSensitiveDataLogging includes application data in exception messages and framework logging. The logged data can include the values assigned to properties of entity instances and parameter values for commands sent to the database. Logging data with EnableSensitiveDataLogging is a **security risk**, as it may expose passwords and other Personally Identifiable Information (PII) when it logs SQL statements executed against the database.

We recommend only enabling EnableSensitiveDataLogging for development and testing:

C#

```
#if DEBUG
    services.AddDbContextFactory<ContactContext>(opt =>
        opt.UseSqlite($"Data Source={nameof(ContactContext.ContactsDb)}.db")
        .EnableSensitiveDataLogging());
#else
    services.AddDbContextFactory<ContactContext>(opt =>
        opt.UseSqlite($"Data Source=
{nameof(ContactContext.ContactsDb)}.db"));
#endif
```

Additional resources

- EF Core documentation
- Blazor samples GitHub repository (dotnet/blazor-samples)
- Blazor *Security and Identity* node articles

ASP.NET Core Blazor advanced scenarios (render tree construction)

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 the advanced scenario for building Blazor render trees manually with RenderTreeBuilder.

⚠ **Warning**

Use of RenderTreeBuilder to create components is an *advanced scenario*. A malformed component (for example, an unclosed markup tag) can result in undefined behavior. Undefined behavior includes broken content rendering, loss of app features, and ***compromised security***.

Manually build a render tree (`RenderTreeBuilder`)

RenderTreeBuilder provides methods for manipulating components and elements, including building components manually in C# code.

Consider the following `PetDetails` component, which can be manually rendered in another component.

`PetDetails.razor`:

```razor
<h2>Pet Details</h2>

<p>@PetDetailsQuote</p>

@code
```

```
{
    [Parameter]
    public string? PetDetailsQuote { get; set; }
}
```

In the following `BuiltContent` component, the loop in the `CreateComponent` method generates three `PetDetails` components.

In RenderTreeBuilder methods with a sequence number, sequence numbers are source code line numbers. The Blazor difference algorithm relies on the sequence numbers corresponding to distinct lines of code, not distinct call invocations. When creating a component with RenderTreeBuilder methods, hardcode the arguments for sequence numbers. **Using a calculation or counter to generate the sequence number can lead to poor performance.** For more information, see the Sequence numbers relate to code line numbers and not execution order section.

`BuiltContent.razor`:

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

<PageTitle>Built Content</PageTitle>

<h1>Built Content Example</h1>

<div>
    @CustomRender
</div>

<button @onclick="RenderComponent">
    Create three Pet Details components
</button>

@code {
    private RenderFragment? CustomRender { get; set; }

    private RenderFragment CreateComponent() => builder =>
    {
        for (var i = 0; i < 3; i++)
        {
            builder.OpenComponent(0, typeof(PetDetails));
            builder.AddAttribute(1, "PetDetailsQuote", "Someone's best
friend!");
            builder.CloseComponent();
        }
    };
    private void RenderComponent() => CustomRender = CreateComponent();
}
```

⚠ Warning

The types in Microsoft.AspNetCore.Components.RenderTree allow processing of the *results* of rendering operations. These are internal details of the Blazor framework implementation. These types should be considered *unstable* and subject to change in future releases.

Sequence numbers relate to code line numbers and not execution order

Razor component files (`.razor`) are always compiled. Executing compiled code has a potential advantage over interpreting code because the compile step that yields the compiled code can be used to inject information that improves app performance at runtime.

A key example of these improvements involves *sequence numbers*. Sequence numbers indicate to the runtime which outputs came from which distinct and ordered lines of code. The runtime uses this information to generate efficient tree diffs in linear time, which is far faster than is normally possible for a general tree diff algorithm.

Consider the following Razor component file (`.razor`):

razor

```razor
@if (someFlag)
{
    <text>First</text>
}

Second
```

The preceding Razor markup and text content compiles into C# code similar to the following:

C#

```csharp
if (someFlag)
{
    builder.AddContent(0, "First");
}
```

```
builder.AddContent(1, "Second");
```

When the code executes for the first time and `someFlag` is `true`, the builder receives the sequence in the following table.

Expand table

Sequence	Type	Data
0	Text node	First
1	Text node	Second

Imagine that `someFlag` becomes `false` and the markup is rendered again. This time, the builder receives the sequence in the following table.

Expand table

Sequence	Type	Data
1	Text node	Second

When the runtime performs a diff, it sees that the item at sequence `0` was removed, so it generates the following trivial *edit script* with a single step:

- Remove the first text node.

The problem with generating sequence numbers programmatically

Imagine instead that you wrote the following render tree builder logic:

```C#
var seq = 0;

if (someFlag)
{
    builder.AddContent(seq++, "First");
}

builder.AddContent(seq++, "Second");
```

The first output is reflected in the following table.

Expand table

Sequence	Type	Data
0	Text node	First
1	Text node	Second

This outcome is identical to the prior case, so no negative issues exist. `someFlag` is `false` on the second rendering, and the output is seen in the following table.

Expand table

Sequence	Type	Data
0	Text node	Second

This time, the diff algorithm sees that *two* changes have occurred. The algorithm generates the following edit script:

- Change the value of the first text node to `Second`.
- Remove the second text node.

Generating the sequence numbers has lost all the useful information about where the `if/else` branches and loops were present in the original code. This results in a diff **twice as long** as before.

This is a trivial example. In more realistic cases with complex and deeply nested structures, and especially with loops, the performance cost is usually higher. Instead of immediately identifying which loop blocks or branches have been inserted or removed, the diff algorithm must recurse deeply into the render trees. This usually results in building longer edit scripts because the diff algorithm is misinformed about how the old and new structures relate to each other.

Guidance and conclusions

- App performance suffers if sequence numbers are generated dynamically.
- The necessary information doesn't exist to permit the framework to generate sequence numbers automatically at runtime unless the information is captured at compile time.
- Don't write long blocks of manually-implemented RenderTreeBuilder logic. Prefer `.razor` files and allow the compiler to deal with the sequence numbers. If you're unable to avoid manual RenderTreeBuilder logic, split long blocks of code into

smaller pieces wrapped in OpenRegion/CloseRegion calls. Each region has its own separate space of sequence numbers, so you can restart from zero (or any other arbitrary number) inside each region.

- If sequence numbers are hardcoded, the diff algorithm only requires that sequence numbers increase in value. The initial value and gaps are irrelevant. One legitimate option is to use the code line number as the sequence number, or start from zero and increase by ones or hundreds (or any preferred interval).
- For loops, the sequence numbers should increase in your source code, not in terms of runtime behavior. The fact that, at runtime, the numbers repeat is how the diffing system realises you're in a loop.
- Blazor uses sequence numbers, while other tree-diffing UI frameworks don't use them. Diffing is far faster when sequence numbers are used, and Blazor has the advantage of a compile step that deals with sequence numbers automatically for developers authoring `.razor` files.

Tutorial: Create an ASP.NET Core app with Angular in Visual Studio

Article • 11/08/2024

In this article, you learn how to build an ASP.NET Core project to act as an API backend and an Angular project to act as the UI.

Visual Studio includes ASP.NET Core Single Page Application (SPA) templates that support Angular and React. The templates provide a built-in Client App folder in your ASP.NET Core projects that contains the base files and folders of each framework.

You can use the method described in this article to create ASP.NET Core Single Page Applications that:

- Put the client app in a separate project, outside from the ASP.NET Core project
- Create the client project based on the framework CLI installed on your computer

Note

This article describes the project creation process using the updated template in Visual Studio 2022 version 17.8.

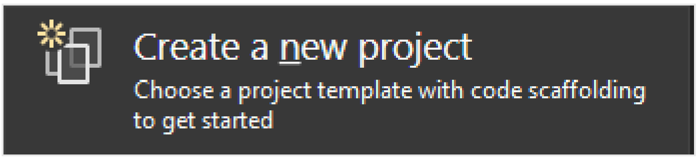
Prerequisites

Make sure to install the following:

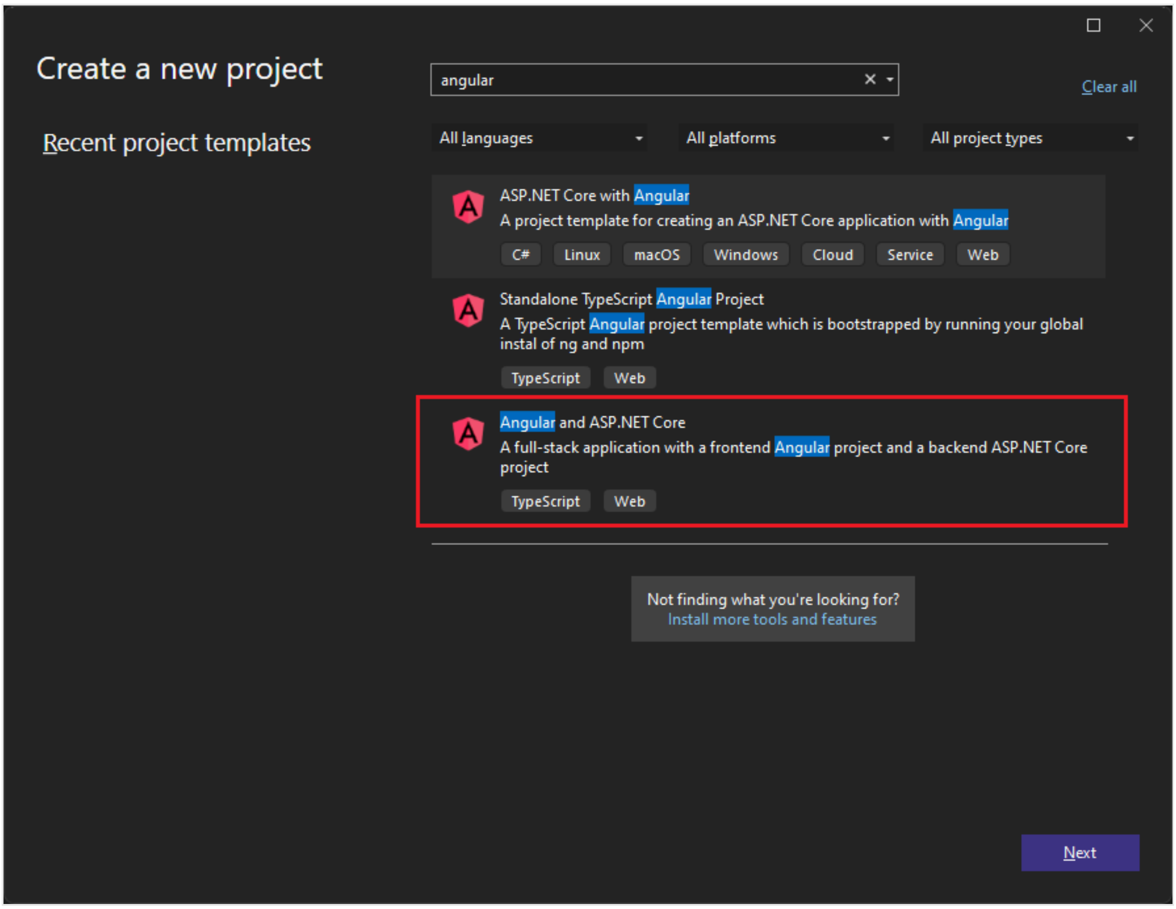
- Visual Studio 2022 version 17.8 or later with the **ASP.NET and web development** workload installed. Go to the Visual Studio downloads page to install it for free. If you need to install the workload and already have Visual Studio, go to **Tools** > **Get Tools and Features...**, which opens the Visual Studio Installer. Choose the **ASP.NET and web development** workload, then choose **Modify**.
- npm (https://www.npmjs.com/), which is included with Node.js
- Angular CLI (https://angular.dev/tools/cli), which can be the version of your choice.

Create the frontend app

1. In the Start window (choose **File** > **Start Window** to open), select **Create a new project**.


Create a new project
Choose a project template with code scaffolding to get started

2. Search for Angular in the search bar at the top and then select **Angular and ASP.NET Core**.

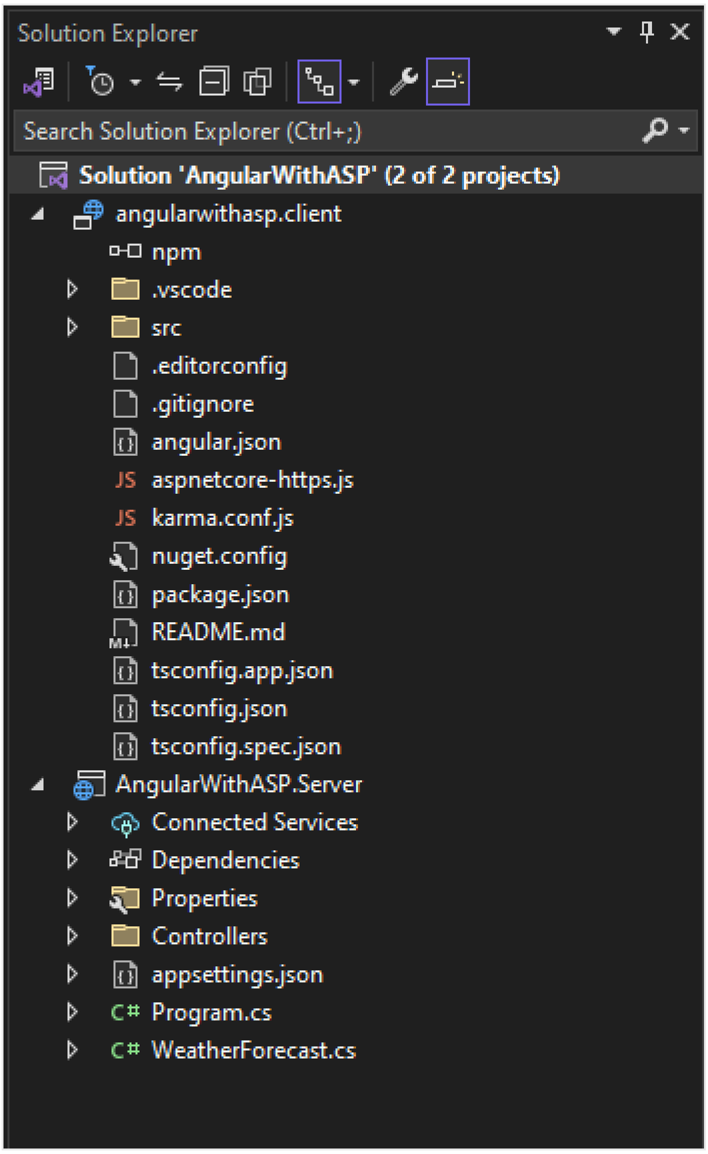

Create a new project
Recent project templates
angular
Clear all
All languages
All platforms
All project types
ASP.NET Core with Angular
A project template for creating an ASP.NET Core application with Angular
C#
Linux
macOS
Windows
Cloud
Service
Web
Standalone TypeScript Angular Project
A TypeScript Angular project template which is bootstrapped by running your global instal of ng and npm
TypeScript
Web
Angular and ASP.NET Core
A full-stack application with a frontend Angular project and a backend ASP.NET Core project
TypeScript
Web
Not finding what you're looking for?
Install more tools and features
Next

3. Name the project **AngularWithASP** and then select **Next**.

   In the Additional Information dialog, make sure that **Configure for HTTPS** is enabled. In most scenarios, leave the other settings at the default values.

4. Select **Create**.

   Solution Explorer shows the following::



Solution 'AngularWithASP' (2 of 2 projects)
- angularwithasp.client
  - npm
  - .vscode
  - src
  - .editorconfig
  - .gitignore
  - angular.json
  - aspnetcore-https.js
  - karma.conf.js
  - nuget.config
  - package.json
  - README.md
  - tsconfig.app.json
  - tsconfig.json
  - tsconfig.spec.json
- AngularWithASP.Server
  - Connected Services
  - Dependencies
  - Properties
  - Controllers
  - appsettings.json
  - Program.cs
  - WeatherForecast.cs

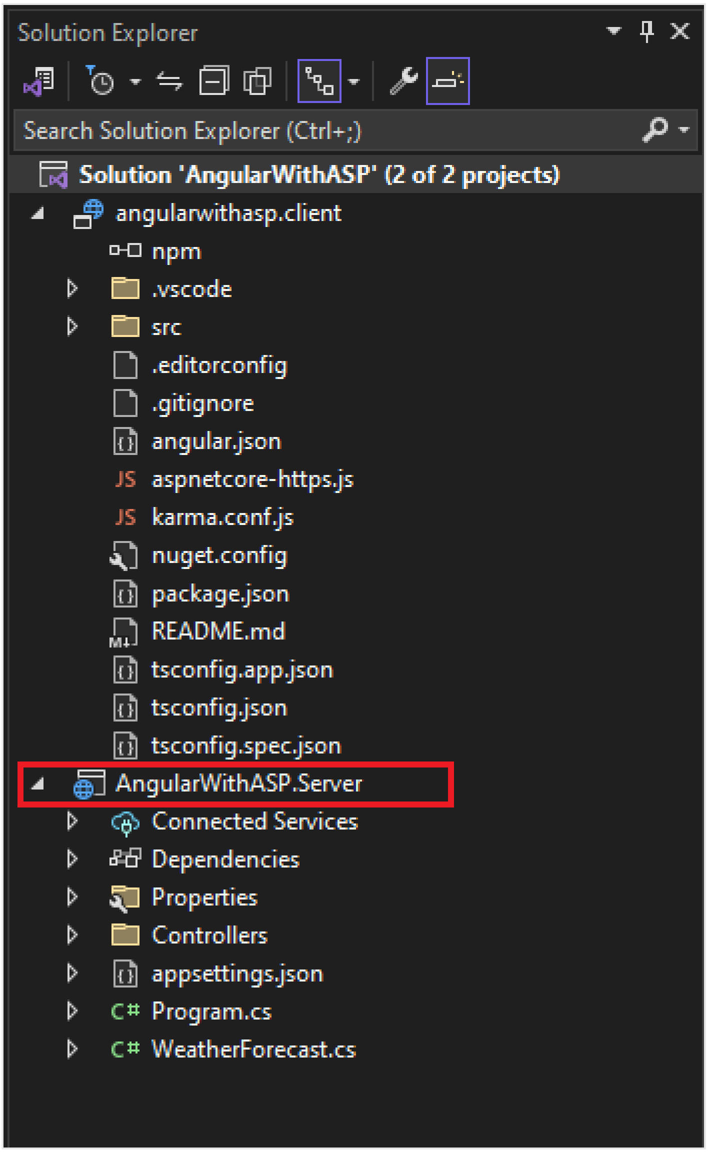
Compared to the standalone Angular template, you see some new and modified files for integration with ASP.NET Core:

- aspnetcore-https.js
- proxy.conf.js
- package.json(modified)
- angular.json(modified)
- app.components.ts
- app.module.ts

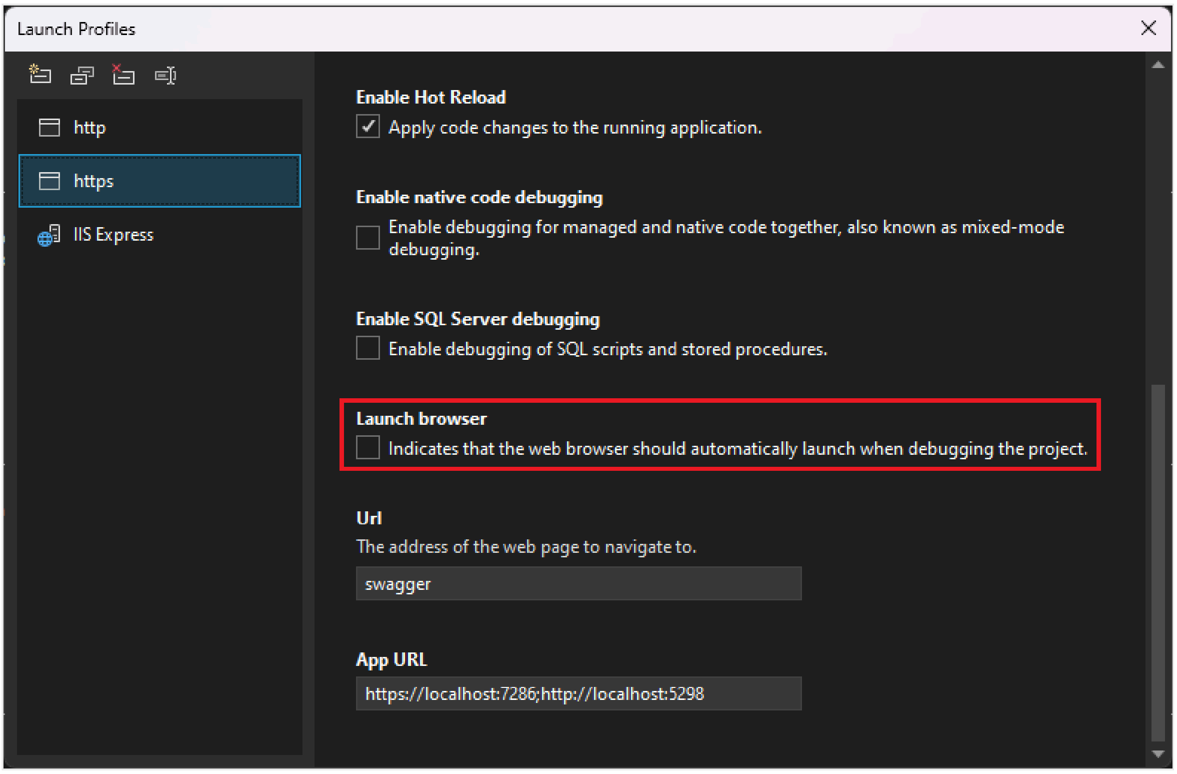
For more information on some of these project files, see Next steps.

Set the project properties

1. In Solution Explorer, right-click the **AngularWithASP.Server** project and choose **Properties**.



2. In the Properties page, open the **Debug** tab and select **Open debug launch profiles UI** option. Uncheck the **Launch Browser** option for the **https** profile or the profile named after the ASP.NET Core project, if present.


Launch Profiles

http

https

IIS Express

Enable Hot Reload
[✓] Apply code changes to the running application.

Enable native code debugging
[ ] Enable debugging for managed and native code together, also known as mixed-mode debugging.

Enable SQL Server debugging
[ ] Enable debugging of SQL scripts and stored procedures.

Launch browser
[ ] Indicates that the web browser should automatically launch when debugging the project.

Url
The address of the web page to navigate to.
swagger

App URL
https://localhost:7286;http://localhost:5298

This value prevents opening the web page with the source weather data.

ⓘ Note

In Visual Studio, `launch.json` stores the startup settings associated with the **Start** button in the Debug toolbar. `launch.json` must be located under the `.vscode` folder.

3. Right-click the solution in Solution Explorer and select **Properties**. Verify that the Startup project settings are set to **Multiple projects**, and that the Action for both projects is set to **Start**.

Start the project

Press **F5** or select the **Start** button at the top of the window to start the app. Two command prompts appear:

- The ASP.NET Core API project running
- The Angular CLI running the ng start command

ⓘ Note

Check console output for messages. For example there might be a message to update Node.js.

The Angular app appears and is populated via the API. If you don't see the app, see Troubleshooting.

Publish the project

Starting in Visual Studio 2022 version 17.3, you can publish the integrated solution using the Visual Studio Publish tool.

Note

To use publish, create your JavaScript project using Visual Studio 2022 version 17.3 or later.

1. In Solution Explorer, right-click the **AngularWithASP.Server** project and select **Add** > **Project Reference**.

   Make sure the **angularwithasp.client** project is selected.

2. Choose **OK**.

3. Right-click the ASP.NET Core project again and select **Edit Project File**.

   This opens the `.csproj` file for the project.

4. In the `.csproj` file, make sure the project reference includes a `<ReferenceOutputAssembly>` element with the value set to `false`.

   This reference should look like the following.

   XML

   ```xml
    <ProjectReference
   Include="..\angularwithasp.client\angularwithasp.client.esproj">
      <ReferenceOutputAssembly>false</ReferenceOutputAssembly>
    </ProjectReference>
   ```

5. Right-click the ASP.NET Core project and choose **Reload Project** if that option is available.

6. In *Program.cs*, make sure the following code is present.

   C#

   ```csharp
   app.UseDefaultFiles();
   app.UseStaticFiles();
   ```

```
// Configure the HTTP request pipeline.
if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();
}
```

7. To publish, right click the ASP.NET Core project, choose **Publish**, and select options to match your desired publish scenario, such as Azure, publish to a folder, etc.

   The publish process takes more time than it does for just an ASP.NET Core project, since the `npm run build` command gets invoked when publishing. The BuildCommand runs `npm run build` by default.

   If you publish to a folder, see ASP.NET Core directory structure for more information on the files added to the *publish* folder.

Troubleshooting

Proxy error

You may see the following error:

Windows Command Prompt

```
[HPM] Error occurred while trying to proxy request /weatherforecast from
localhost:4200 to https://localhost:5001 (ECONNREFUSED)
(https://nodejs.org/api/errors.html#errors_common_system_errors)
```

If you see this issue, most likely the frontend started before the backend.

- Once you see the backend command prompt up and running, just refresh the Angular app in the browser.
- Also, verify that the backend is configured to start before the front end. To verify, select the solution in Solution Explorer, choose **Properties** from the **Project menu**. Next, select **Configure Startup Projects** and make sure that the backend ASP.NET Core project is first in the list. If it's not first, select the project and use the Up arrow button to make it the first project in the launch list.

Verify port

If the weather data doesn't load correctly, you may also need to verify that your ports are correct.

1. Go to the `launchSettings.json` file in your ASP.NET Core project (in the *Properties* folder). Get the port number from the `applicationUrl` property.

   If there are multiple `applicationUrl` properties, look for one using an `https` endpoint. It should look similar to `https://localhost:7049`.

2. Then, go to the `proxy.conf.js` file for your Angular project (look in the *src* folder). Update the target property to match the `applicationUrl` property in *launchSettings.json*. When you update it, that value should look similar to this:

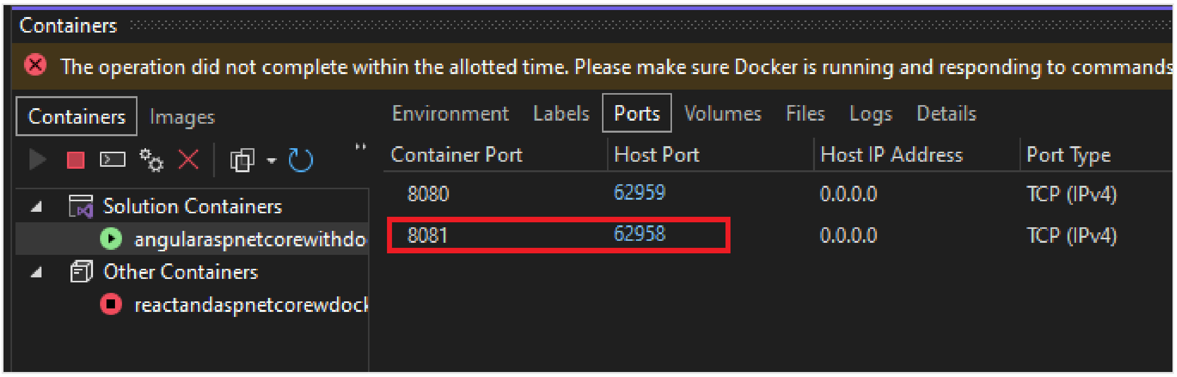
   JavaScript

   ```
   target: 'https://localhost:7049',
   ```

Docker

If you create the project with Docker support enabled, take the following steps:

1. After the app loads, get the Docker HTTPS port using the Containers window in Visual Studio. Check the **Environment** or **Ports** tab.

   Containers

   The operation did not complete within the allotted time. Please make sure Docker is running and responding to commands

Container Port	Host Port	Host IP Address	Port Type
8080	62959	0.0.0.0	TCP (IPv4)
8081	62958	0.0.0.0	TCP (IPv4)

2. Open the `proxy.conf.js` file for the Angular project. Update the `target` variable to match the HTTPS port in the Containers window. For example, in the following code:

   JavaScript

   ```
   const target = env.ASPNETCORE_HTTPS_PORT ?
   `https://localhost:${env.ASPNETCORE_HTTPS_PORT}` :
      env.ASPNETCORE_URLS ? env.ASPNETCORE_URLS.split(';')[0] :
   'https://localhost:7209';
   ```

change `https://localhost:7209` to the matching HTTPS port (in this example, `https://localhost:62958`).

3. Restart the app.

Next steps

For more information about SPA applications in ASP.NET Core, see the Angular section under Developing Single Page Apps. The linked article provides additional context for project files such as *aspnetcore-https.js* and *proxy.conf.js*, although details of the implementation are different due to project template differences. For example, instead of a ClientApp folder, the Angular files are contained in a separate project.

For MSBuild information specific to the client project, see MSBuild properties for JSPS.

Tutorial: Create an ASP.NET Core app with React in Visual Studio

Article • 11/06/2024

In this article, you learn how to build an ASP.NET Core project to act as an API backend and a React project to act as the UI.

Currently, Visual Studio includes ASP.NET Core Single Page Application (SPA) templates that support Angular and React. The templates provide a built-in Client App folder in your ASP.NET Core projects that contains the base files and folders of each framework.

You can use the method described in this article to create ASP.NET Core Single Page Applications that:

- Put the client app in a separate project, outside from the ASP.NET Core project
- Create the client project based on the framework CLI installed on your computer

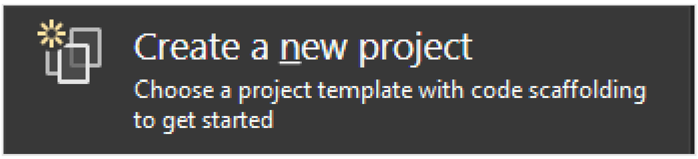
> **Note**
>
> This article describes the project creation process using the updated template in Visual Studio 2022 version 17.11, which uses the Vite CLI.

Prerequisites

- Visual Studio 2022 version 17.11 or later with the **ASP.NET and web development** workload installed. Go to the Visual Studio downloads page to install it for free. If you need to install the workload and already have Visual Studio, go to **Tools** > **Get Tools and Features...**, which opens the Visual Studio Installer. Choose the **ASP.NET and web development** workload, then choose **Modify**.
- npm (https://www.npmjs.com/), which is included with Node.js

Create the frontend app

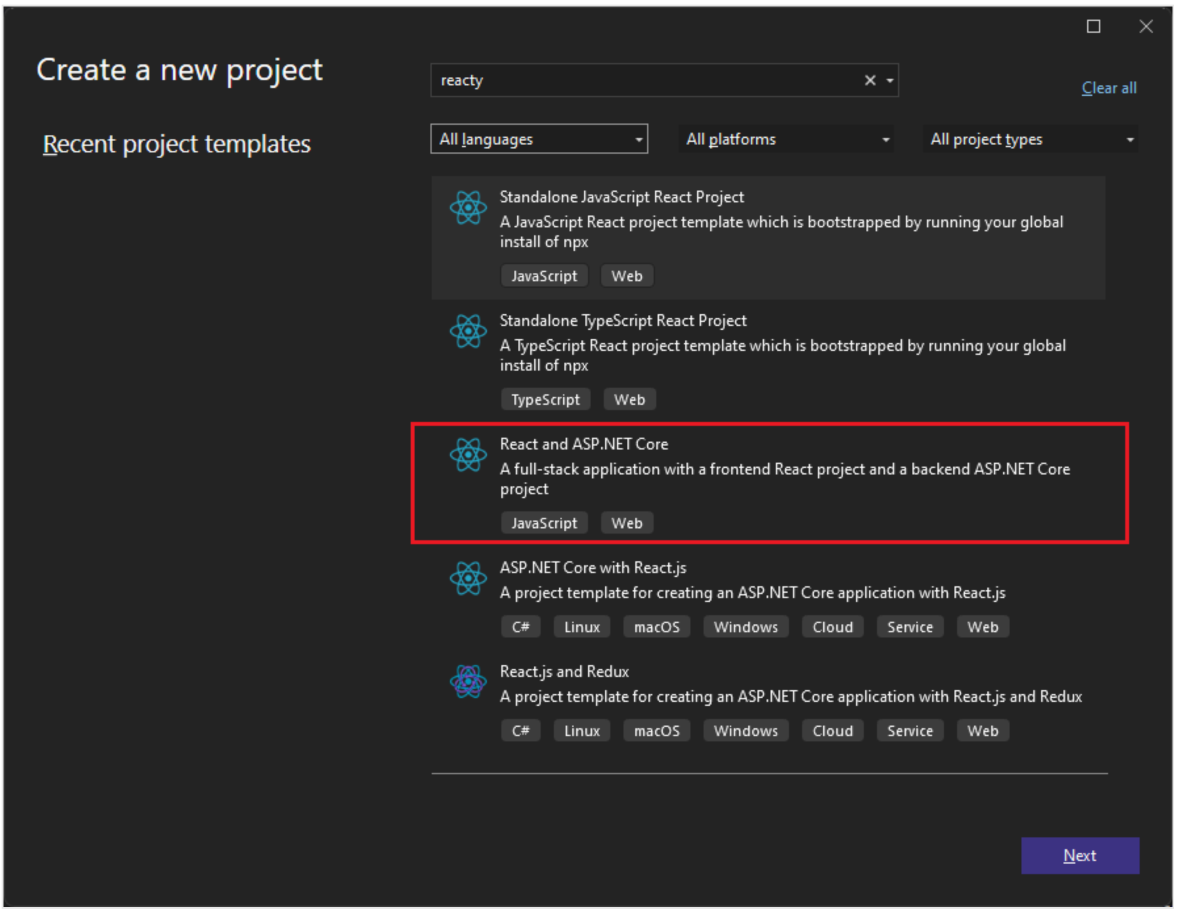
1. In the Start window, select **Create a new project**.



Create a new project

Choose a project template with code scaffolding to get started

2. Search for React in the search bar at the top and then select **React and ASP.NET Core**. This template is a JavaScript template.



Standalone JavaScript React Project
A JavaScript React project template which is bootstrapped by running your global install of npx
JavaScript Web

Standalone TypeScript React Project
A TypeScript React project template which is bootstrapped by running your global install of npx
TypeScript Web

React and ASP.NET Core
A full-stack application with a frontend React project and a backend ASP.NET Core project
JavaScript Web

ASP.NET Core with React.js
A project template for creating an ASP.NET Core application with React.js
C# Linux macOS Windows Cloud Service Web

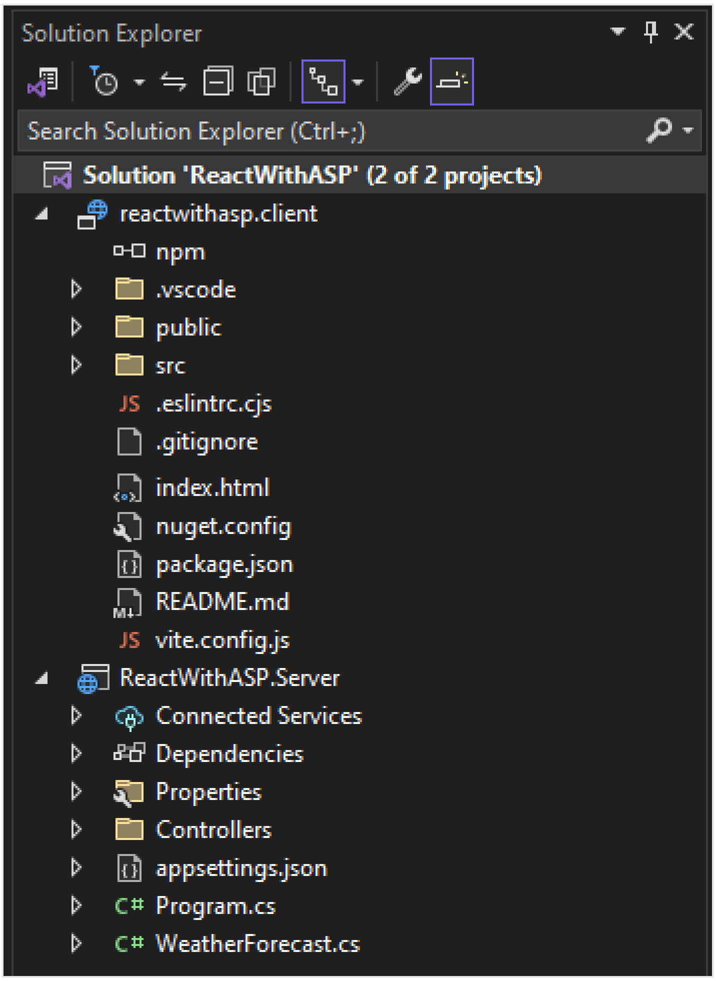
React.js and Redux
A project template for creating an ASP.NET Core application with React.js and Redux
C# Linux macOS Windows Cloud Service Web

3. Name the project **ReactWithASP** and then select **Next**.

   In the Additional Information dialog, make sure that **Configure for HTTPS** is enabled. In most scenarios, leave the other settings at the default values.

4. Select **Create**.

   Solution Explorer shows the following project information:


Solution Explorer

Search Solution Explorer (Ctrl+;)

```
Solution 'ReactWithASP' (2 of 2 projects)
  reactwithasp.client
    npm
    .vscode
    public
    src
    .eslintrc.cjs
    .gitignore
    index.html
    nuget.config
    package.json
    README.md
    vite.config.js
  ReactWithASP.Server
    Connected Services
    Dependencies
    Properties
    Controllers
    appsettings.json
    Program.cs
    WeatherForecast.cs
```

Compared to the standalone React template, you see some new and modified files for integration with ASP.NET Core:

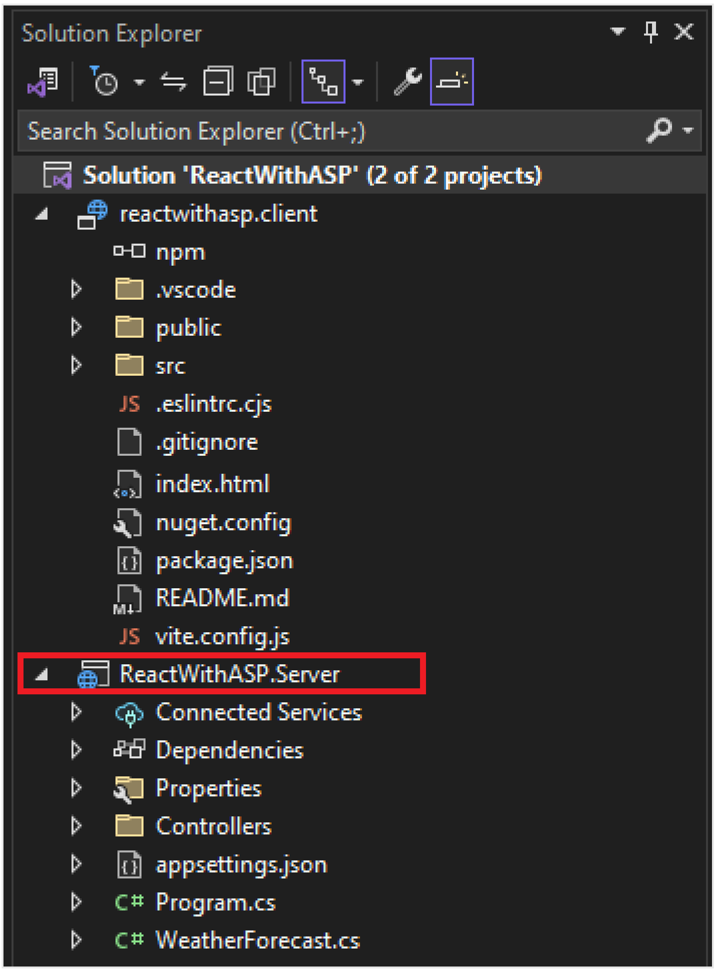
- vite.config.js
- App.js (modified)
- App.test.js (modified)

5. Select an installed browser from the Debug toolbar, such as Chrome or Microsoft Edge.

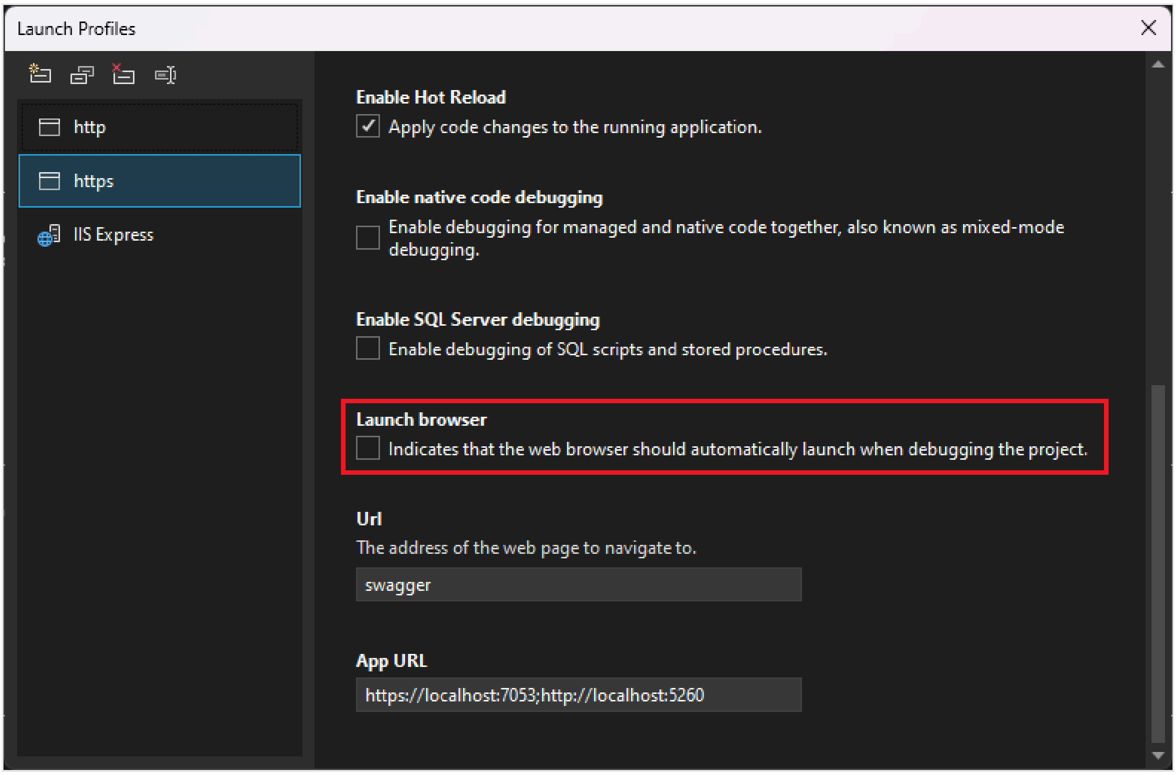
   If the browser you want is not yet installed, install the browser first, and then select it.

Set the project properties

1. In Solution Explorer, right-click the **ReactWithASP.Server** project and choose **Properties**.



2. In the Properties page, open the **Debug** tab and select **Open debug launch profiles UI** option. Uncheck the **Launch Browser** option for the **https** profile or the profile named after the ASP.NET Core project, if present.



Enable Hot Reload
[✓] Apply code changes to the running application.

Enable native code debugging
[ ] Enable debugging for managed and native code together, also known as mixed-mode debugging.

Enable SQL Server debugging
[ ] Enable debugging of SQL scripts and stored procedures.

Launch browser
[ ] Indicates that the web browser should automatically launch when debugging the project.

Url
The address of the web page to navigate to.
swagger

App URL
https://localhost:7053;http://localhost:5260

This value prevents opening the web page with the source weather data.

> **Note**
>
> In Visual Studio, `launch.json` stores the startup settings associated with the **Start** button in the Debug toolbar. Currently, `launch.json` must be located under the `.vscode` folder.

3. Right-click the solution in Solution Explorer and select **Properties**. Verify that the Startup project settings are set to **Multiple projects**, and that the Action for both projects is set to **Start**.

Start the project

Press **F5** or select the **Start** button at the top of the window to start the app. Two command prompts appear:

- The ASP.NET Core API project running
- The Vite CLI showing a message such as `VITE v4.4.9 ready in 780 ms`

> **Note**
>
> Check console output for messages. For example there might be a message to update Node.js.

The React app appears and is populated via the API. If you don't see the app, see Troubleshooting.

Publish the project

1. In Solution Explorer, right-click the **ReactWithASP.Server** project and select **Add** > **Project Reference**.

   Make sure the **reactwithasp.client** project is selected.

2. Choose **OK**.

3. Right-click the ASP.NET Core project again and select **Edit Project File**.

   This opens the `.csproj` file for the project.

4. In the `.csproj` file, make sure the project reference includes a `<ReferenceOutputAssembly>` element with the value set to `false`.

   This reference should look like the following.

   XML
   ```xml
    <ProjectReference
   Include="..\reactwithasp.client\reactwithasp.client.esproj">
      <ReferenceOutputAssembly>false</ReferenceOutputAssembly>
    </ProjectReference>
   ```

5. Right-click the ASP.NET Core project and choose **Reload Project** if that option is available.

6. In *Program.cs*, make sure the following code is present.

   C#
   ```csharp
   app.UseDefaultFiles();
   app.UseStaticFiles();

   // Configure the HTTP request pipeline.
   if (app.Environment.IsDevelopment())
   {
      app.UseSwagger();
      app.UseSwaggerUI();
   }
   ```

7. To publish, right click the ASP.NET Core project, choose **Publish**, and select options to match your desired publish scenario, such as Azure, publish to a folder, etc.

   The publish process takes more time than it does for just an ASP.NET Core project, since the `npm run build` command gets invoked when publishing. The BuildCommand runs `npm run build` by default.

   If you publish to a folder, see ASP.NET Core directory structure for more information on the files added to the *publish* folder.

Troubleshooting

Proxy error

You may see the following error:

Windows Command Prompt

```
[HPM] Error occurred while trying to proxy request /weatherforecast from
localhost:4200 to https://localhost:7183 (ECONNREFUSED)
(https://nodejs.org/api/errors.html#errors_common_system_errors)
```

If you see this issue, most likely the frontend started before the backend.

- Once you see the backend command prompt up and running, just refresh the React app in the browser.
- Also, verify that the backend is configured to start before the front end. To verify, select the solution in Solution Explorer, choose **Properties** from the **Project menu**. Next, select **Configure Startup Projects** and make sure that the backend ASP.NET Core project is first in the list. If it's not first, select the project and use the Up arrow button to make it the first project in the launch list.

Verify ports

If the weather data doesn't load correctly, you may also need to verify that your ports are correct.

1. Make sure that the port numbers match. Go to the `launchSettings.json` file in the ASP.NET Core **ReactWithASP.Server** project (in the *Properties* folder). Get the port number from the `applicationUrl` property.

   If there are multiple `applicationUrl` properties, look for one using an `https` endpoint. It looks similar to `https://localhost:7183`.

2. Open the `vite.config.js` file for the React project. Update the `target` property to match the `applicationUrl` property in *launchSettings.json*. The updated value looks similar to the following:

   JavaScript

   ```javascript
   target: 'https://localhost:7183/',
   ```

Privacy error

You may see the following certificate error:

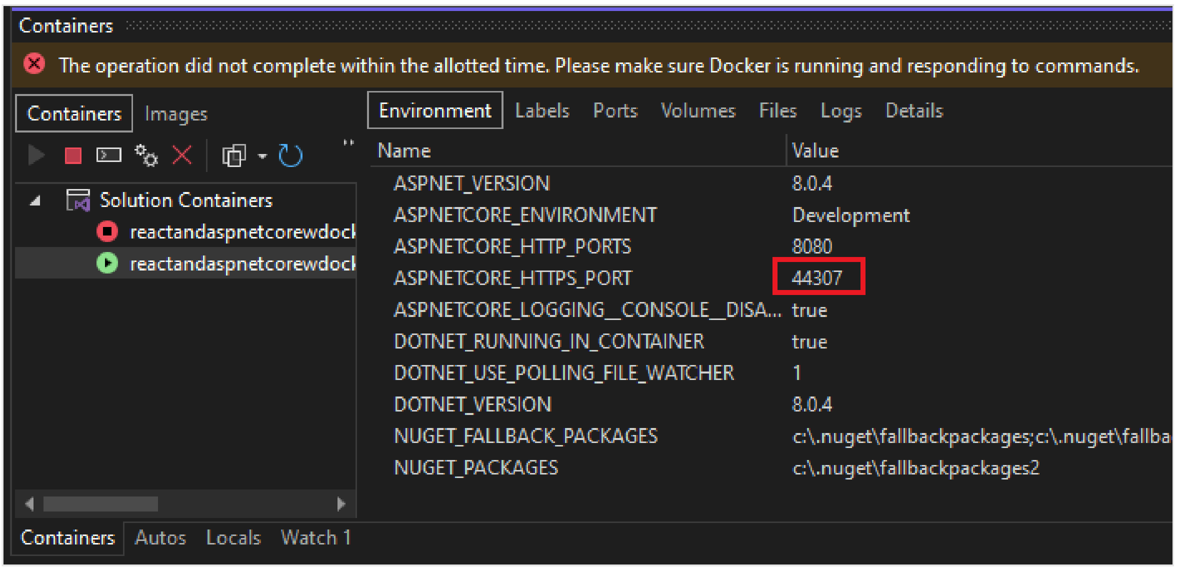
```
Your connection isn't private
```

Try deleting the React certificates from *%appdata%\local\asp.net\https* or *%appdata%\roaming\asp.net\https*, and then retry.

Docker

If you create the project with Docker support enabled, take the following steps:

1. After the app loads, get the Docker HTTPS port using the Containers window in Visual Studio. Check the **Environment** or **Ports** tab.



Name	Value
ASPNET_VERSION	8.0.4
ASPNETCORE_ENVIRONMENT	Development
ASPNETCORE_HTTP_PORTS	8080
ASPNETCORE_HTTPS_PORT	44307
ASPNETCORE_LOGGING__CONSOLE__DISA...	true
DOTNET_RUNNING_IN_CONTAINER	true
DOTNET_USE_POLLING_FILE_WATCHER	1
DOTNET_VERSION	8.0.4
NUGET_FALLBACK_PACKAGES	c:\.nuget\fallbackpackages;c:\.nuget\fallba
NUGET_PACKAGES	c:\.nuget\fallbackpackages2

2. Open the `vite.config.js` file for the React project. Update the `target` variable to match the HTTPS port in the Containers window. For example, in the following code:

JavaScript

```javascript
const target = env.ASPNETCORE_HTTPS_PORT ?
`https://localhost:${env.ASPNETCORE_HTTPS_PORT}` :
   env.ASPNETCORE_URLS ? env.ASPNETCORE_URLS.split(';')[0] :
'https://localhost:7143';
```

change `https://localhost:7143` to the matching HTTPS port (in this example, `https://localhost:44307`).

3. Restart the app.

Next steps

For more information about SPA applications in ASP.NET Core, see the React section under Developing Single Page Apps. The linked article provides additional context for project files such as *aspnetcore-https.js*, although details of the implementation are different based on the template differences. For example, instead of a ClientApp folder, the React files are contained in a separate project.

For MSBuild information specific to the client project, see MSBuild properties for JSPS.

Tutorial: Create an ASP.NET Core app with Vue in Visual Studio

Article • 11/06/2024

In this article, you learn how to build an ASP.NET Core project to act as an API backend and a Vue project to act as the UI.

Visual Studio includes ASP.NET Core Single Page Application (SPA) templates that support Angular, React, and Vue. The templates provide a built-in Client App folder in your ASP.NET Core projects that contains the base files and folders of each framework.

You can use the method described in this article to create ASP.NET Core Single Page Applications that:

- Put the client app in a separate project, outside from the ASP.NET Core project
- Create the client project based on the framework CLI installed on your computer

ⓘ **Note**

This article describes the project creation process using the updated template in Visual Studio 2022 version 17.11, which uses the Vite CLI.

Prerequisites

Make sure to install the following:

- Visual Studio 2022 version 17.11 or later with the **ASP.NET and web development** workload installed. Go to the Visual Studio downloads page to install it for free. If you need to install the workload and already have Visual Studio, go to **Tools** > **Get Tools and Features...**, which opens the Visual Studio Installer. Choose the **ASP.NET and web development** workload, then choose **Modify**.
- npm (https://www.npmjs.com/), which is included with Node.js.

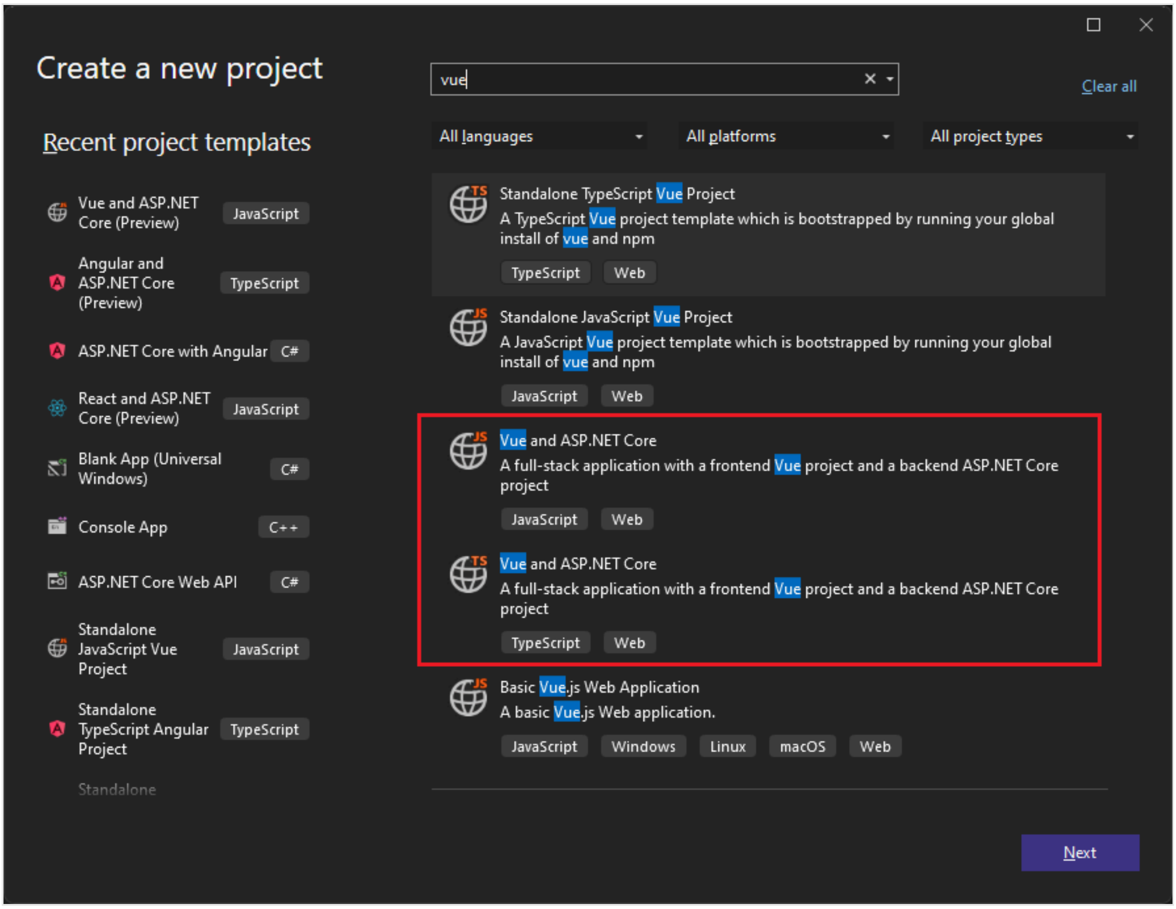
Create the frontend app

1. In the Start window (choose **File** > **Start Window** to open), select **Create a new project**.

Create a new project

Choose a project template with code scaffolding to get started

2. Search for Vue in the search bar at the top and then select **Vue and ASP.NET Core** with either JavaScript or TypeScript as the selected language.



Create a new project

vue

Clear all

Recent project templates

All languages | All platforms | All project types

Vue and ASP.NET Core (Preview)	JavaScript
Angular and ASP.NET Core (Preview)	TypeScript
ASP.NET Core with Angular	C#
React and ASP.NET Core (Preview)	JavaScript
Blank App (Universal Windows)	C#
Console App	C++
ASP.NET Core Web API	C#
Standalone JavaScript Vue Project	JavaScript
Standalone TypeScript Angular Project	TypeScript
Standalone	

Standalone TypeScript Vue Project
A TypeScript Vue project template which is bootstrapped by running your global install of vue and npm
TypeScript · Web

Standalone JavaScript Vue Project
A JavaScript Vue project template which is bootstrapped by running your global install of vue and npm
JavaScript · Web

Vue and ASP.NET Core
A full-stack application with a frontend Vue project and a backend ASP.NET Core project
JavaScript · Web

Vue and ASP.NET Core
A full-stack application with a frontend Vue project and a backend ASP.NET Core project
TypeScript · Web

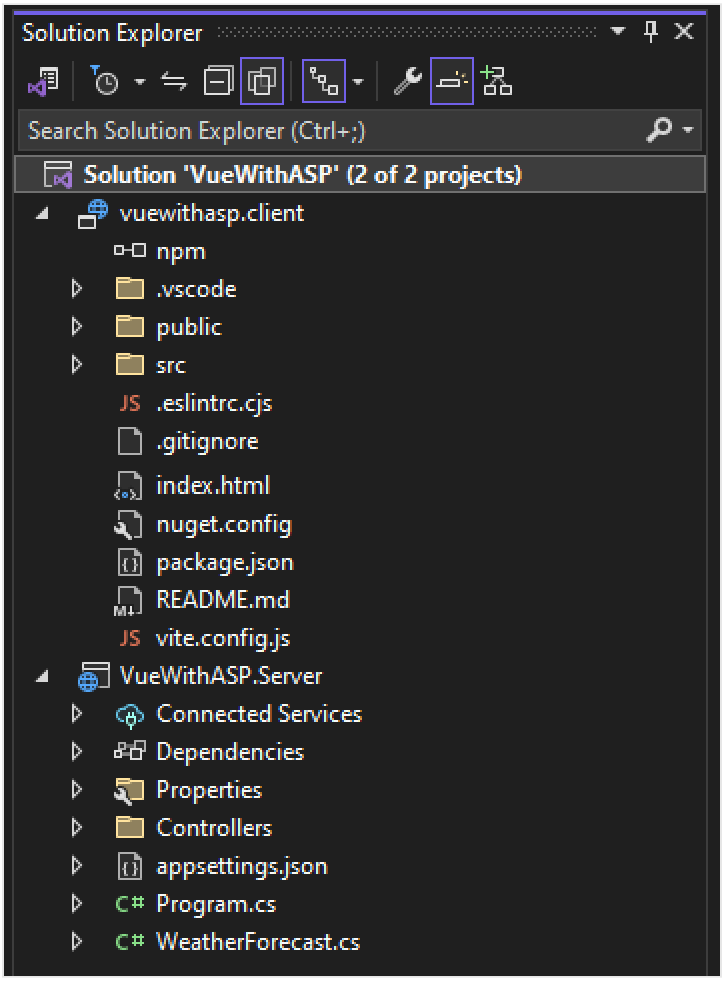
Basic Vue.js Web Application
A basic Vue.js Web application.
JavaScript · Windows · Linux · macOS · Web

3. Name the project **VueWithASP** and then select **Next**.

   In the Additional Information dialog, make sure that **Configure for HTTPS** is enabled. In most scenarios, leave the other settings at the default values.

4. Select **Create**.

   Solution Explorer shows the following project information:


Solution Explorer

Search Solution Explorer (Ctrl+;)

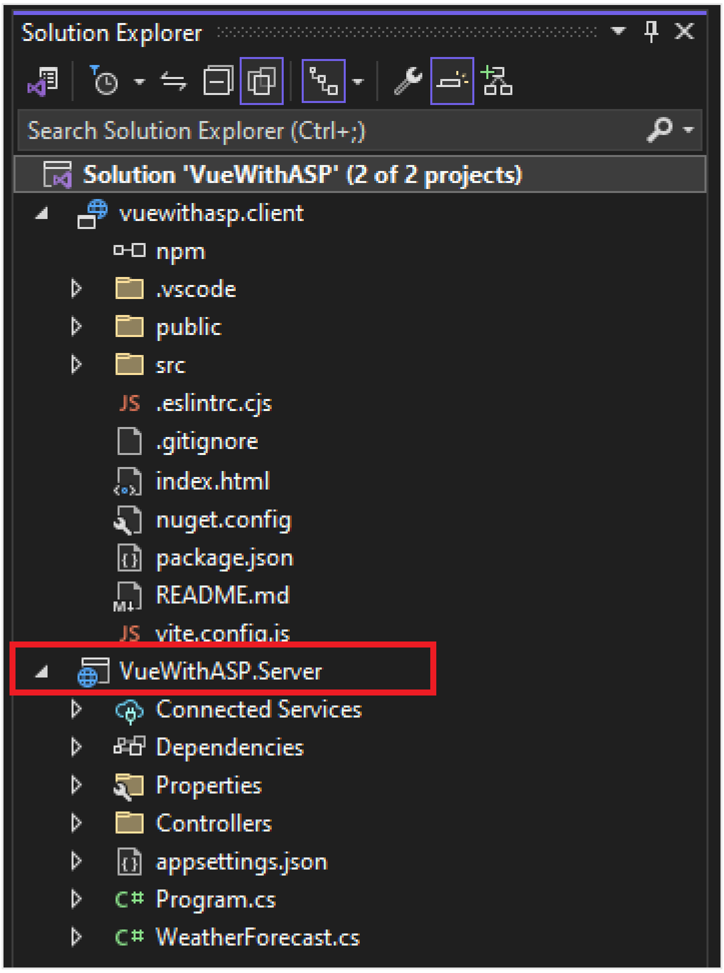
Solution 'VueWithASP' (2 of 2 projects)
- vuewithasp.client
  - npm
  - .vscode
  - public
  - src
  - .eslintrc.cjs
  - .gitignore
  - index.html
  - nuget.config
  - package.json
  - README.md
  - vite.config.js
- VueWithASP.Server
  - Connected Services
  - Dependencies
  - Properties
  - Controllers
  - appsettings.json
  - Program.cs
  - WeatherForecast.cs

Compared to the standalone Vue template, you see some new and modified files for integration with ASP.NET Core:

- vite.config.json (modified)
- HelloWorld.vue (modified)
- package.json (modified)

Set the project properties

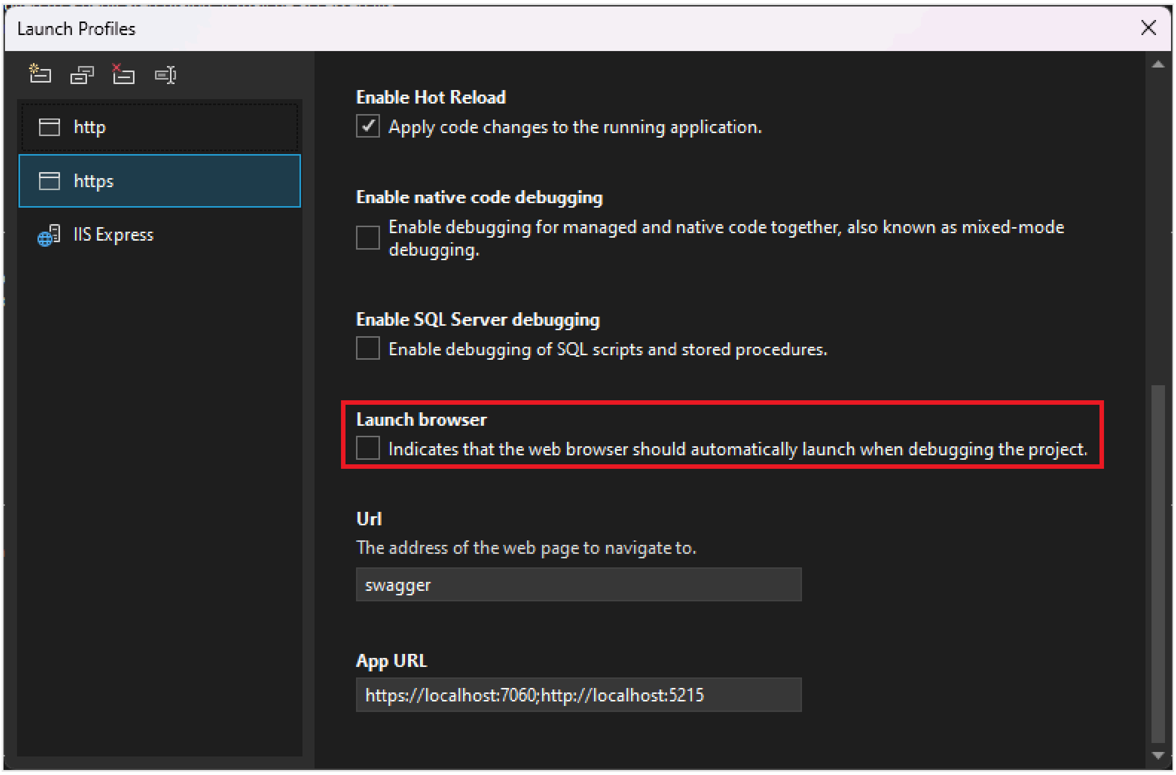
1. In Solution Explorer, right-click the **VueWithASP.Server** and choose **Properties**.


Solution Explorer

Search Solution Explorer (Ctrl+;)

Solution 'VueWithASP' (2 of 2 projects)
- vuewithasp.client
  - npm
  - .vscode
  - public
  - src
  - .eslintrc.cjs
  - .gitignore
  - index.html
  - nuget.config
  - package.json
  - README.md
  - vite.config.js
- VueWithASP.Server
  - Connected Services
  - Dependencies
  - Properties
  - Controllers
  - appsettings.json
  - Program.cs
  - WeatherForecast.cs

2. In the Properties page, open the **Debug** tab and select **Open debug launch profiles UI** option. Uncheck the **Launch Browser** option for the **https** profile or the profile named after the ASP.NET Core project, if present.


Launch Profiles

- http
- https
- IIS Express

Enable Hot Reload
☑ Apply code changes to the running application.

Enable native code debugging
☐ Enable debugging for managed and native code together, also known as mixed-mode debugging.

Enable SQL Server debugging
☐ Enable debugging of SQL scripts and stored procedures.

Launch browser
☐ Indicates that the web browser should automatically launch when debugging the project.

Url
The address of the web page to navigate to.
swagger

App URL
https://localhost:7060;http://localhost:5215

This value prevents opening the web page with the source weather data.

> ⓘ **Note**
>
> In Visual Studio, `launch.json` stores the startup settings associated with the **Start** button in the Debug toolbar. Currently, `launch.json` must be located under the `.vscode` folder.

3. Right-click the solution in Solution Explorer and select **Properties**. Verify that the Startup project settings are set to **Multiple projects**, and that the Action for both projects is set to **Start**.

Start the project

Press **F5** or select the **Start** button at the top of the window to start the app. Two command prompts appear:

- The ASP.NET Core API project running
- The Vite CLI showing a message such as `VITE v4.4.9 ready in 780 ms`

> ⓘ **Note**
>
> Check console output for messages. For example there might be a message to update Node.js.

The Vue app appears and is populated via the API. If you don't see the app, see Troubleshooting.

Publish the project

Starting in Visual Studio 2022 version 17.3, you can publish the integrated solution using the Visual Studio Publish tool.

> ⓘ **Note**
>
> To use publish, create your JavaScript project using Visual Studio 2022 version 17.3 or later.

1. In Solution Explorer, right-click the **VueWithASP.Server** project and select **Add** > **Project Reference**.

Make sure the **vuewithasp.client** project is selected.

2. Choose **OK**.

3. Right-click the ASP.NET Core project again and select **Edit Project File**.

   This opens the `.csproj` file for the project.

4. In the `.csproj` file, make sure the project reference includes a `<ReferenceOutputAssembly>` element with the value set to `false`.

   This reference should look like the following.

   XML

   ```xml
    <ProjectReference
   Include="..\vuewithasp.client\vuewithasp.client.esproj">
      <ReferenceOutputAssembly>false</ReferenceOutputAssembly>
    </ProjectReference>
   ```

5. Right-click the ASP.NET Core project and choose **Reload Project** if that option is available.

6. In *Program.cs*, make sure the following code is present.

   C#

   ```csharp
   app.UseDefaultFiles();
   app.UseStaticFiles();

   // Configure the HTTP request pipeline.
   if (app.Environment.IsDevelopment())
   {
      app.UseSwagger();
      app.UseSwaggerUI();
   }
   ```

7. To publish, right click the ASP.NET Core project, choose **Publish**, and select options to match your desired publish scenario, such as Azure, publish to a folder, etc.

   The publish process takes more time than it does for just an ASP.NET Core project, since the `npm run build` command gets invoked when publishing. The BuildCommand runs `npm run build` by default.

   If you publish to a folder, see ASP.NET Core directory structure for more information on the files added to the *publish* folder.

Troubleshooting

Proxy error

You may see the following error:

```
[HPM] Error occurred while trying to proxy request /weatherforecast from
localhost:4200 to https://localhost:5173 (ECONNREFUSED)
(https://nodejs.org/api/errors.html#errors_common_system_errors)
```

If you see this issue, most likely the frontend started before the backend.

- Once you see the backend command prompt up and running, just refresh the Vue app in the browser.
- Also, verify that the backend is configured to start before the front end. To verify, select the solution in Solution Explorer, choose **Properties** from the **Project menu**. Next, select **Configure Startup Projects** and make sure that the backend ASP.NET Core project is first in the list. If it's not first, select the project and use the Up arrow button to make it the first project in the launch list.

Otherwise, if the port is in use, try incrementing the port number by **1** in `launchSettings.json` and *vite.config.js*.

Privacy error

You may see the following certificate error:

```
Your connection isn't private
```

Try deleting the Vue certificates from *%appdata%\local\asp.net\https* or *%appdata%\roaming\asp.net\https*, and then retry.

Verify ports

If the weather data doesn't load correctly, you may also need to verify that your ports are correct.

1. Make sure that the port numbers match. Go to the `launchSettings.json` file in your ASP.NET Core project (in the *Properties* folder). Get the port number from the `applicationUrl` property.

   If there are multiple `applicationUrl` properties, look for one using an `https` endpoint. It should look similar to `https://localhost:7142`.

2. Then, go to the `vite.config.js` file for your Vue project. Update the `target` property to match the `applicationUrl` property in *launchSettings.json*. When you update it, that value should look similar to this:

   JavaScript

   ```javascript
   target: 'https://localhost:7142/',
   ```

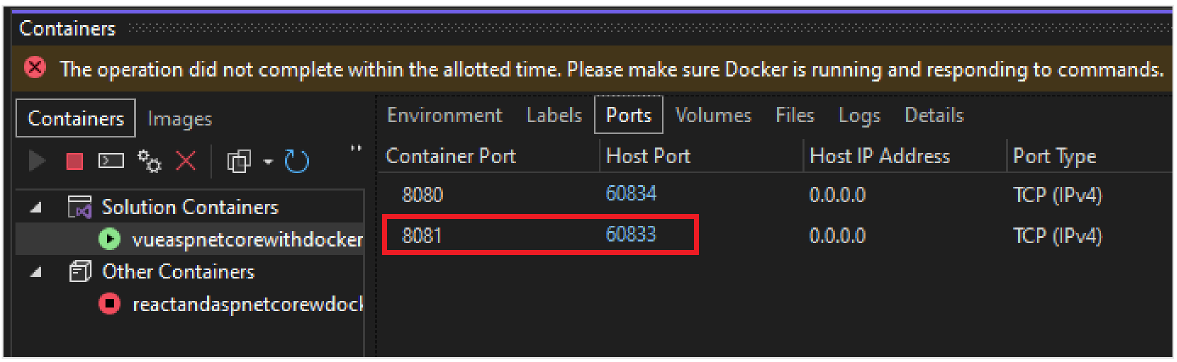
Outdated version of Vue

If you see the console message **Could not find the file 'C:\Users\Me\source\repos\vueprojectname\package.json'** when you create the project, you may need to update your version of the Vite CLI. After you update the Vite CLI, you may also need to delete the `.vuerc` file in *C:\Users\[yourprofilename]*.

Docker

If you create the project with Docker support enabled, take the following steps:

1. After the app loads, get the Docker HTTPS port using the Containers window in Visual Studio. Check the **Environment** or **Ports** tab.



Containers

The operation did not complete within the allotted time. Please make sure Docker is running and responding to commands.

Container Port	Host Port	Host IP Address	Port Type
8080	60834	0.0.0.0	TCP (IPv4)
8081	60833	0.0.0.0	TCP (IPv4)

2. Open the `vite.config.js` file for the Vue project. Update the `target` variable to match the HTTPS port in the Containers window. For example, in the following code:

JavaScript

```javascript
const target = env.ASPNETCORE_HTTPS_PORT ?
`https://localhost:${env.ASPNETCORE_HTTPS_PORT}` :
   env.ASPNETCORE_URLS ? env.ASPNETCORE_URLS.split(';')[0] :
'https://localhost:7163';
```

change `https://localhost:7163` to the matching HTTPS port (in this example, `https://localhost:60833`).

3. Restart the app.

If you are using a Docker configuration created in older versions of Visual Studio, the backend may start up using the Docker profile and not listen on the configured port 5173. To resolve:

Edit the Docker profile in the `launchSettings.json` by adding the following properties:

JSON

```json
"httpPort": 5175,
"sslPort": 5173
```

Next steps

For more information about SPA applications in ASP.NET Core, see Developing Single Page Apps. The linked article provides additional context for project files such as *aspnetcore-https.js*, although details of the implementation are different due to differences between the project templates and the Vue.js framework vs. other frameworks. For example, instead of a ClientApp folder, the Vue files are contained in a separate project.

For MSBuild information specific to the client project, see MSBuild properties for JSPS.

Feedback

Was this page helpful? Yes No

Provide product feedback | Ask the community

JavaScript and TypeScript in Visual Studio

Article • 06/27/2024

Visual Studio 2022 provides rich support for JavaScript development, both using JavaScript directly, and also using the TypeScript programming language, which was developed to provide a more productive and enjoyable JavaScript development experience, especially when developing projects at scale. You can write JavaScript or TypeScript code in Visual Studio for many application types and services.

JavaScript language service

The JavaScript experience in Visual Studio 2022 is powered by the same engine that provides TypeScript support. This engine gives you better feature support, richness, and integration immediately out-of-the-box.

The option to restore to the legacy JavaScript language service is no longer available. Users have the new JavaScript language service out-of-the-box. The new language service is solely based on the TypeScript language service, which is powered by static analysis. This service enables us to provide you with better tooling, so your JavaScript code can benefit from richer IntelliSense based on type definitions. The new service is lightweight and consumes less memory than the legacy service, providing you with better performance as your code scales. We also improved performance of the language service to handle larger projects.

TypeScript support

By default, Visual Studio 2022 provides language support for JavaScript and TypeScript files to power IntelliSense without any specific project configuration.

For compiling TypeScript, Visual Studio gives you the flexibility to choose which version of TypeScript to use on a per-project basis.

In MSBuild compilation scenarios such as ASP.NET Core, the TypeScript NuGet package is the recommended method of adding TypeScript compilation support to your project. Visual Studio will give you the option to add this package the first time you add a TypeScript file to your project. This package is also available at any time through the NuGet package manager. When the NuGet package is used, the corresponding language service version will be used for language support in your project. Note: The minimum supported version of this package is 3.6.

Projects configured for npm, such as Node.js projects, can specify their own version of the TypeScript language service by adding the TypeScript npm package. You can specify the version using the npm manager in supported projects. Note: The minimum supported version of this package is 2.1.

The TypeScript SDK has been deprecated in Visual Studio 2022. Existing projects that rely on the SDK should be upgraded to use the NuGet package. For projects that cannot be upgraded immediately, the SDK is still available on the Visual Studio Marketplace and as an optional component in the Visual Studio installer.

Tip

For projects developed in Visual Studio 2022, we encourage you to use the TypeScript NuGet or the TypeScript npm package for greater portability across different platforms and environments. For more information, see **Compile TypeScript code using NuGet** and **Compile TypeScript code using tsc**.

Project templates

Starting in Visual Studio 2022, there is a new JavaScript/TypeScript project type (.esproj), called the JavaScript Project System (JSPS), which allows you to create standalone Angular, React, and Vue projects in Visual Studio. These front-end projects are created using the framework CLI tools you have installed on your local machine, so the version of the template is up to you. To migrate from existing Node.js projects to the new project system, see Migrate Node.js projects. For MSBuild information for the new project type, see MSBuild properties for JSPS

Within these new projects, you can run JavaScript and TypeScript unit tests, easily add and connect ASP.NET Core API projects and download your npm modules using the npm manager. Check out some of the quickstarts and tutorials to get started. For more information, see Visual Studio tutorials | JavaScript and TypeScript.

Note

A simplified, updated template is available starting in Visual Studio 2022 version 17.5. Compared to the ASP.NET SPA templates available in Visual Studio, the *.esproj*

SPA templates for ASP.NET Core provide better npm dependency management, and better build and publish support.

Feedback

Was this page helpful? Yes No

Overview of Single Page Apps (SPAs) in ASP.NET Core

Article • 06/18/2024

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.

Visual Studio provides project templates for creating single-page apps (SPAs) based on JavaScript frameworks such as Angular, React, and Vue that have an ASP.NET Core backend. These templates:

- Create a Visual Studio solution with a frontend project and a backend project.
- Use the Visual Studio project type for JavaScript and TypeScript (*.esproj*) for the frontend.
- Use an ASP.NET Core project for the backend.

Projects created by using the Visual Studio templates can be run from the command line on Windows, Linux, and macOS. To run the app, use `dotnet run --launch-profile https` to run the server project. Running the server project automatically starts the frontend JavaScript development server. The `https` launch profile is currently required.

Visual Studio tutorials

To get started, follow one of the tutorials in the Visual Studio documentation:

- Create an ASP.NET Core app with Angular
- Create an ASP.NET Core app with React
- Create an ASP.NET Core app with Vue

For more information, see JavaScript and TypeScript in Visual Studio

ASP.NET Core SPA templates

Visual Studio includes templates for building ASP.NET Core apps with a JavaScript or TypeScript frontend. These templates are available in Visual Studio 2022 version 17.8 or later with the **ASP.NET and web development** workload installed.

The Visual Studio templates for building ASP.NET Core apps with a JavaScript or TypeScript frontend offer the following benefits:

- Clean project separation for the frontend and backend.
- Stay up-to-date with the latest frontend framework versions.
- Integrate with the latest frontend framework command-line tooling, such as Vite.
- Templates for both JavaScript & TypeScript (only TypeScript for Angular).
- Rich JavaScript and TypeScript code editing experience.
- Integrate JavaScript build tools with the .NET build.
- npm dependency management UI.
- Compatible with Visual Studio Code debugging and launch configuration.
- Run frontend unit tests in Test Explorer using JavaScript test frameworks.

Legacy ASP.NET Core SPA templates

Earlier versions of the .NET SDK included what are now legacy templates for building SPA apps with ASP.NET Core. For documentation on these older templates, see the ASP.NET Core 7.0 version of the SPA overview and the Angular and React articles.

Use the Angular project template with ASP.NET Core

Article • 09/29/2023

ⓘ **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.

Visual Studio provides project templates for creating single-page apps (SPAs) based on JavaScript frameworks such as Angular, React, and Vue that have an ASP.NET Core backend. These templates:

- Create a Visual Studio solution with a frontend project and a backend project.
- Use the Visual Studio project type for JavaScript and TypeScript (*.esproj*) for the frontend.
- Use an ASP.NET Core project for the backend.

Projects created by using the Visual Studio templates can be run from the command line on Windows, Linux, and macOS. To run the app, use `dotnet run --launch-profile https` to run the server project. Running the server project automatically starts the frontend JavaScript development server. The `https` launch profile is currently required.

Visual Studio tutorial

To get started with an Angular project, follow the Create an ASP.NET Core app with Angular tutorial in the Visual Studio documentation.

For more information, see JavaScript and TypeScript in Visual Studio

ASP.NET Core SPA templates

Visual Studio includes templates for building ASP.NET Core apps with a JavaScript or TypeScript frontend. These templates are available in Visual Studio 2022 version 17.8 or later with the **ASP.NET and web development** workload installed.

The Visual Studio templates for building ASP.NET Core apps with a JavaScript or TypeScript frontend offer the following benefits:

- Clean project separation for the frontend and backend.
- Stay up-to-date with the latest frontend framework versions.
- Integrate with the latest frontend framework command-line tooling, such as Vite .
- Templates for both JavaScript & TypeScript (only TypeScript for Angular).
- Rich JavaScript and TypeScript code editing experience.
- Integrate JavaScript build tools with the .NET build.
- npm dependency management UI.
- Compatible with Visual Studio Code debugging and launch configuration.
- Run frontend unit tests in Test Explorer using JavaScript test frameworks.

Legacy ASP.NET Core SPA templates

Earlier versions of the .NET SDK included what are now legacy templates for building SPA apps with ASP.NET Core. For documentation on these older templates, see the ASP.NET Core 7.0 version of the SPA overview and the Angular and React articles.

Use React with ASP.NET Core

Article • 06/18/2024

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.

Visual Studio provides project templates for creating single-page apps (SPAs) based on JavaScript frameworks such as Angular, React, and Vue that have an ASP.NET Core backend. These templates:

- Create a Visual Studio solution with a frontend project and a backend project.
- Use the Visual Studio project type for JavaScript and TypeScript (*.esproj*) for the frontend.
- Use an ASP.NET Core project for the backend.

Projects created by using the Visual Studio templates can be run from the command line on Windows, Linux, and macOS. To run the app, use `dotnet run --launch-profile https` to run the server project. Running the server project automatically starts the frontend JavaScript development server. The `https` launch profile is currently required.

Visual Studio tutorial

To get started, follow the Create an ASP.NET Core app with React tutorial in the Visual Studio documentation.

For more information, see JavaScript and TypeScript in Visual Studio

ASP.NET Core SPA templates

Visual Studio includes templates for building ASP.NET Core apps with a JavaScript or TypeScript frontend. These templates are available in Visual Studio 2022 version 17.8 or later with the **ASP.NET and web development** workload installed.

The Visual Studio templates for building ASP.NET Core apps with a JavaScript or TypeScript frontend offer the following benefits:

- Clean project separation for the frontend and backend.
- Stay up-to-date with the latest frontend framework versions.
- Integrate with the latest frontend framework command-line tooling, such as Vite.
- Templates for both JavaScript & TypeScript (only TypeScript for Angular).
- Rich JavaScript and TypeScript code editing experience.
- Integrate JavaScript build tools with the .NET build.
- npm dependency management UI.
- Compatible with Visual Studio Code debugging and launch configuration.
- Run frontend unit tests in Test Explorer using JavaScript test frameworks.

Legacy ASP.NET Core SPA templates

Earlier versions of the .NET SDK included what are now legacy templates for building SPA apps with ASP.NET Core. For documentation on these older templates, see the ASP.NET Core 7.0 version of the SPA overview and the Angular and React articles.

Client-side library acquisition in ASP.NET Core with LibMan

Article • 06/18/2024

By Scott Addie

Library Manager (LibMan) is a lightweight, client-side library acquisition tool. LibMan downloads popular libraries and frameworks from the file system or from a content delivery network (CDN). The supported CDNs include CDNJS, jsDelivr, and unpkg. The selected library files are fetched and placed in the appropriate location within the ASP.NET Core project.

LibMan use cases

LibMan offers the following benefits:

- Only the library files you need are downloaded.
- Additional tooling, such as Node.js, npm, and WebPack, isn't necessary to acquire a subset of files in a library.
- Files can be placed in a specific location without resorting to build tasks or manual file copying.

LibMan isn't a package management system. If you're already using a package manager, such as npm or yarn, continue doing so. LibMan wasn't developed to replace those tools.

Additional resources

- Use LibMan with ASP.NET Core in Visual Studio
- Use the LibMan CLI with ASP.NET Core
- LibMan GitHub repository

Use the LibMan CLI with ASP.NET Core

Article • 06/18/2024

Library Manager (LibMan) is a lightweight, client-side library acquisition tool. LibMan downloads popular libraries and frameworks from the file system or from a content delivery network (CDN). The supported CDNs include CDNJS, jsDelivr, and unpkg. The selected library files are fetched and placed in the appropriate location within the ASP.NET Core project.

Prerequisites

- latest .NET SDK

Installation

The following command installs LibMan:

.NET CLI

```
dotnet tool install -g Microsoft.Web.LibraryManager.Cli
```

Note

By default the architecture of the .NET binaries to install represents the currently running OS architecture. To specify a different OS architecture, see dotnet tool install, --arch option. For more information, see GitHub issue dotnet/AspNetCore.Docs #29262.

A .NET Core Global Tool is installed from the Microsoft.Web.LibraryManager.Cli NuGet package.

Usage

Console

```
libman
```

To view the installed LibMan version:

Console

```
libman --version
```

To view the available CLI commands:

Console

```
libman --help
```

The preceding command displays output similar to the following:

Console

```
 1.0.163+g45474d37ed

Usage: libman [options] [command]

Options:
  --help|-h  Show help information
  --version  Show version information

Commands:
  cache      List or clean libman cache contents
  clean      Deletes all library files defined in libman.json from the
project
  init       Create a new libman.json
  install    Add a library definition to the libman.json file, and download
the
             library to the specified location
  restore    Downloads all files from provider and saves them to specified
             destination
  uninstall  Deletes all files for the specified library from their
specified
             destination, then removes the specified library definition from
             libman.json
  update     Updates the specified library

Use "libman [command] --help" for more information about a command.
```

The following sections outline the available CLI commands.

Initialize LibMan in the project

The `libman init` command creates a `libman.json` file if one doesn't exist. The file is created with the default item template content.

Synopsis

```console
libman init [-d|--default-destination] [-p|--default-provider] [--verbosity]
libman init [-h|--help]
```

Options

The following options are available for the `libman init` command:

- `-d|--default-destination <PATH>`

  A path relative to the current folder. Library files are installed in this location if no `destination` property is defined for a library in `libman.json`. The `<PATH>` value is written to the `defaultDestination` property of `libman.json`.

- `-p|--default-provider <PROVIDER>`

  The provider to use if no provider is defined for a given library. The `<PROVIDER>` value is written to the `defaultProvider` property of `libman.json`. Replace `<PROVIDER>` with one of the following values:
  - `cdnjs`
  - `filesystem`
  - `jsdelivr`
  - `unpkg`

- `-h|--help`

  Show help information.

- `--verbosity <LEVEL>`

  Set the verbosity of the output. Replace `<LEVEL>` with one of the following values:
  - `quiet`
  - `normal`
  - `detailed`

Examples

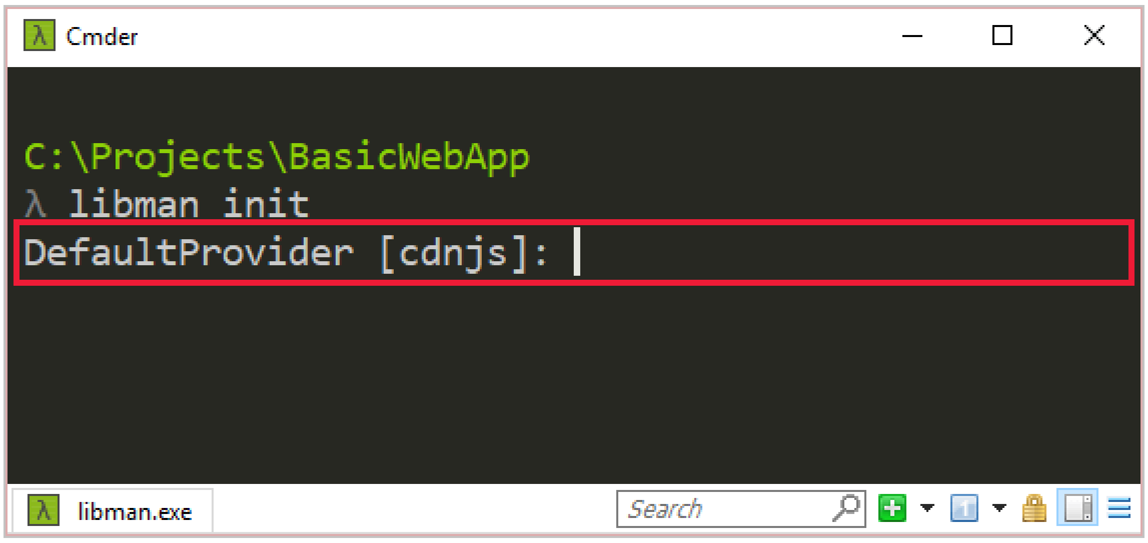
To create a `libman.json` file in an ASP.NET Core project:

- Navigate to the project root.
- Run the following command:

Console

```console
libman init
```

- Type the name of the default provider, or press `Enter` to use the default CDNJS provider. Valid values include:
  - `cdnjs`
  - `filesystem`
  - `jsdelivr`
  - `unpkg`



```
C:\Projects\BasicWebApp
λ libman init
DefaultProvider [cdnjs]:
```

A `libman.json` file is added to the project root with the following content:

JSON

```json
{
  "version": "1.0",
  "defaultProvider": "cdnjs",
  "libraries": []
}
```

Add library files

The `libman install` command downloads and installs library files into the project. A `libman.json` file is added if one doesn't exist. The `libman.json` file is modified to store configuration details for the library files.

Synopsis

Console

```
libman install <LIBRARY> [-d|--destination] [--files] [-p|--provider] [--
verbosity]
libman install [-h|--help]
```

Arguments

`LIBRARY`

The name of the library to install. This name may include version number notation (for example, `@1.2.0`).

Options

The following options are available for the `libman install` command:

- `-d|--destination <PATH>`

  The location to install the library. If not specified, the default location is used. If no `defaultDestination` property is specified in `libman.json`, this option is required.

  Note: There are limitations to the destination path. For example, when the package source has a full project structure and not just the distribution folder, you can't specify moving a folder. For more information, see Issue #407 and Issue #702

- `--files <FILE>`

  Specify the name of the file to install from the library. If not specified, all files from the library are installed. Provide one `--files` option per file to be installed. Relative paths are supported too. For example: `--files dist/browser/signalr.js`.

- `-p|--provider <PROVIDER>`

  The name of the provider to use for the library acquisition. Replace `<PROVIDER>` with one of the following values:
  - `cdnjs`
  - `filesystem`
  - `jsdelivr`
  - `unpkg`

If not specified, the `defaultProvider` property in `libman.json` is used. If no `defaultProvider` property is specified in `libman.json`, this option is required.

- `-h|--help`

  Show help information.

- `--verbosity <LEVEL>`

  Set the verbosity of the output. Replace `<LEVEL>` with one of the following values:
  - `quiet`
  - `normal`
  - `detailed`

Examples

Consider the following `libman.json` file:

JSON

```json
{
  "version": "1.0",
  "defaultProvider": "cdnjs",
  "libraries": []
}
```

To install the jQuery version 3.2.1 `jquery.min.js` file to the *wwwroot/scripts/jquery* folder using the CDNJS provider:

Console

```console
libman install jquery@3.2.1 --provider cdnjs --destination
wwwroot/scripts/jquery --files jquery.min.js
```

The `libman.json` file resembles the following:

JSON

```json
{
  "version": "1.0",
  "defaultProvider": "cdnjs",
  "libraries": [
    {
      "library": "jquery@3.2.1",
      "destination": "wwwroot/scripts/jquery",
      "files": [
        "jquery.min.js"
      ]
    }
  ]
}
```

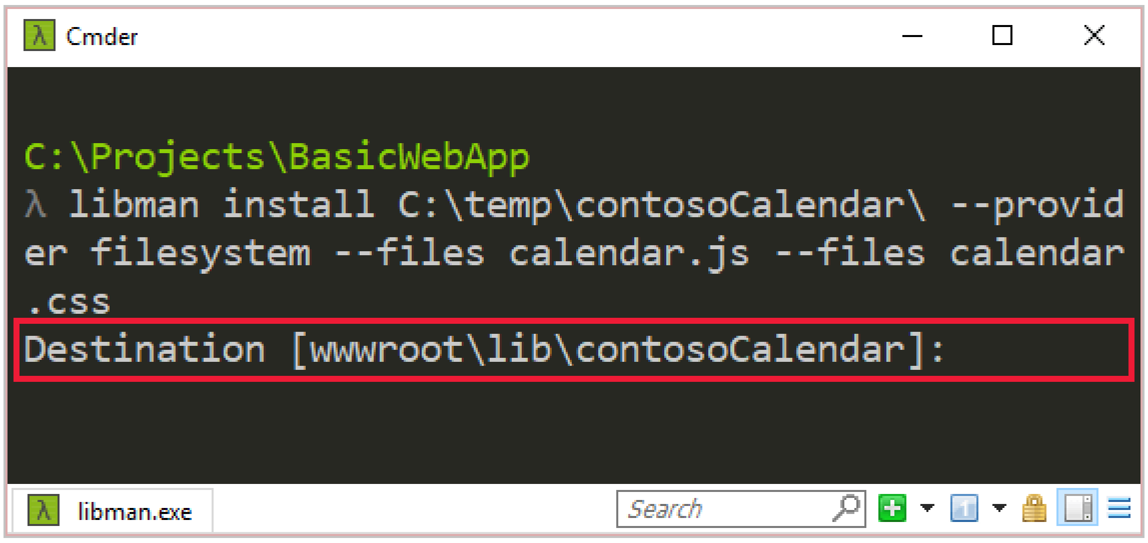
To install the `calendar.js` and `calendar.css` files from *C:\temp\contosoCalendar\* using the file system provider:

Console

```
libman install C:\temp\contosoCalendar\ --provider filesystem --files
calendar.js --files calendar.css
```

The following prompt appears for two reasons:

- The `libman.json` file doesn't contain a `defaultDestination` property.
- The `libman install` command doesn't contain the `-d|--destination` option.



```
C:\Projects\BasicWebApp
λ libman install C:\temp\contosoCalendar\ --provid
er filesystem --files calendar.js --files calendar
.css
Destination [wwwroot\lib\contosoCalendar]:
```

After accepting the default destination, the `libman.json` file resembles the following:

JSON

```
{
  "version": "1.0",
  "defaultProvider": "cdnjs",
  "libraries": [
    {
      "library": "jquery@3.2.1",
      "destination": "wwwroot/scripts/jquery",
      "files": [
        "jquery.min.js"
      ]
    },
    {
      "library": "C:\\temp\\contosoCalendar\\",
      "provider": "filesystem",
      "destination": "wwwroot/lib/contosoCalendar",
      "files": [
        "calendar.js",
        "calendar.css"
      ]
    }
  ]
}
```

Restore library files

The `libman restore` command installs library files defined in `libman.json`. The following rules apply:

- If no `libman.json` file exists in the project root, an error is returned.
- If a library specifies a provider, the `defaultProvider` property in `libman.json` is ignored.
- If a library specifies a destination, the `defaultDestination` property in `libman.json` is ignored.

Synopsis

Console

```
libman restore [--verbosity]
libman restore [-h|--help]
```

Options

The following options are available for the `libman restore` command:

- `-h|--help`

  Show help information.

- `--verbosity <LEVEL>`

  Set the verbosity of the output. Replace `<LEVEL>` with one of the following values:
  - `quiet`
  - `normal`
  - `detailed`

Examples

To restore the library files defined in `libman.json`:

```Console
libman restore
```

Delete library files

The `libman clean` command deletes library files previously restored via LibMan. Folders that become empty after this operation are deleted. The library files' associated configurations in the `libraries` property of `libman.json` aren't removed.

Synopsis

```Console
libman clean [--verbosity]
libman clean [-h|--help]
```

Options

The following options are available for the `libman clean` command:

- `-h|--help`

  Show help information.

- `--verbosity <LEVEL>`

  Set the verbosity of the output. Replace `<LEVEL>` with one of the following values:
  - `quiet`
  - `normal`
  - `detailed`

Examples

To delete library files installed via LibMan:

Console

```
libman clean
```

Uninstall library files

The `libman uninstall` command:

- Deletes all files associated with the specified library from the destination in `libman.json`.
- Removes the associated library configuration from `libman.json`.

An error occurs when:

- No `libman.json` file exists in the project root.
- The specified library doesn't exist.

If more than one library with the same name is installed, you're prompted to choose one.

Synopsis

Console

```
libman uninstall <LIBRARY> [--verbosity]
libman uninstall [-h|--help]
```

Arguments

`LIBRARY`

The name of the library to uninstall. This name may include version number notation (for example, `@1.2.0`).

Options

The following options are available for the `libman uninstall` command:

- `-h|--help`

  Show help information.

- `--verbosity <LEVEL>`

Set the verbosity of the output. Replace `<LEVEL>` with one of the following values:

- `quiet`
- `normal`
- `detailed`

Examples

Consider the following `libman.json` file:

JSON

```json
{
  "version": "1.0",
  "defaultProvider": "cdnjs",
  "libraries": [
    {
      "library": "jquery@3.3.1",
      "files": [
        "jquery.min.js",
        "jquery.js",
        "jquery.min.map"
      ],
      "destination": "wwwroot/lib/jquery/"
    },
    {
      "provider": "unpkg",
      "library": "bootstrap@4.1.3",
      "destination": "wwwroot/lib/bootstrap/"
    },
    {
      "provider": "filesystem",
      "library": "C:\\temp\\lodash\\",
      "files": [
        "lodash.js",
        "lodash.min.js"
      ],
      "destination": "wwwroot/lib/lodash/"
    }
  ]
}
```

- To uninstall jQuery, either of the following commands succeed:

  Console

  ```console
  libman uninstall jquery
  ```

  Console

```
libman uninstall jquery@3.3.1
```

- To uninstall the Lodash files installed via the `filesystem` provider:

  Console

  ```
  libman uninstall C:\temp\lodash\
  ```

Update library version

The `libman update` command updates a library installed via LibMan to the specified version.

An error occurs when:

- No `libman.json` file exists in the project root.
- The specified library doesn't exist.

If more than one library with the same name is installed, you're prompted to choose one.

Synopsis

Console

```
libman update <LIBRARY> [-pre] [--to] [--verbosity]
libman update [-h|--help]
```

Arguments

`LIBRARY`

The name of the library to update.

Options

The following options are available for the `libman update` command:

- `-pre`

  Obtain the latest prerelease version of the library.

- `--to <VERSION>`

  Obtain a specific version of the library.

- `-h|--help`

  Show help information.

- `--verbosity <LEVEL>`

  Set the verbosity of the output. Replace `<LEVEL>` with one of the following values:
  - `quiet`
  - `normal`
  - `detailed`

Examples

- To update jQuery to the latest version:

  ```Console
  libman update jquery
  ```

- To update jQuery to version 3.3.1:

  ```Console
  libman update jquery --to 3.3.1
  ```

- To update jQuery to the latest prerelease version:

  ```Console
  libman update jquery -pre
  ```

Manage library cache

The `libman cache` command manages the LibMan library cache. The `filesystem` provider doesn't use the library cache.

Synopsis

Console

```
libman cache clean [<PROVIDER>] [--verbosity]
libman cache list [--files] [--libraries] [--verbosity]
libman cache [-h|--help]
```

Arguments

`PROVIDER`

Only used with the `clean` command. Specifies the provider cache to clean. Valid values include:

- `cdnjs`
- `filesystem`
- `jsdelivr`
- `unpkg`

Options

The following options are available for the `libman cache` command:

- `--files`

  List the names of files that are cached.

- `--libraries`

  List the names of libraries that are cached.

- `-h|--help`

  Show help information.

- `--verbosity <LEVEL>`

  Set the verbosity of the output. Replace `<LEVEL>` with one of the following values:
  - `quiet`
  - `normal`
  - `detailed`

Examples

- To view the names of cached libraries per provider, use one of the following commands:

  Console

  ```
  libman cache list
  ```

  Console

  ```
  libman cache list --libraries
  ```

  Output similar to the following is displayed:

  Console

  ```
  Cache contents:
  ---------------
  unpkg:
      knockout
      react
      vue
  cdnjs:
      font-awesome
      jquery
      knockout
      lodash.js
      react
  ```

- To view the names of cached library files per provider:

  Console

  ```
  libman cache list --files
  ```

  Output similar to the following is displayed:

  Console

  ```
  Cache contents:
  ---------------
  unpkg:
      knockout:
          <list omitted for brevity>
      react:
          <list omitted for brevity>
      vue:
          <list omitted for brevity>
  cdnjs:
  ```

```
    font-awesome
        metadata.json
    jquery
        metadata.json
        3.2.1\core.js
        3.2.1\jquery.js
        3.2.1\jquery.min.js
        3.2.1\jquery.min.map
        3.2.1\jquery.slim.js
        3.2.1\jquery.slim.min.js
        3.2.1\jquery.slim.min.map
        3.3.1\core.js
        3.3.1\jquery.js
        3.3.1\jquery.min.js
        3.3.1\jquery.min.map
        3.3.1\jquery.slim.js
        3.3.1\jquery.slim.min.js
        3.3.1\jquery.slim.min.map
    knockout
        metadata.json
        3.4.2\knockout-debug.js
        3.4.2\knockout-min.js
    lodash.js
        metadata.json
        4.17.10\lodash.js
        4.17.10\lodash.min.js
    react
        metadata.json
```

Notice the preceding output shows that jQuery versions 3.2.1 and 3.3.1 are cached under the CDNJS provider.

- To empty the library cache for the CDNJS provider:

  Console

  ```
  libman cache clean cdnjs
  ```

  After emptying the CDNJS provider cache, the `libman cache list` command displays the following:

  Console

  ```
  Cache contents:
  ---------------
  unpkg:
      knockout
      react
      vue
  ```

```
cdnjs:
    (empty)
```

- To empty the cache for all supported providers:

  Console

  ```
  libman cache clean
  ```

  After emptying all provider caches, the `libman cache list` command displays the following:

  Console

  ```
  Cache contents:
  ---------------
  unpkg:
      (empty)
  cdnjs:
      (empty)
  ```

Additional resources

- Install a Global Tool
- Use LibMan with ASP.NET Core in Visual Studio
- LibMan GitHub repository

Use LibMan with ASP.NET Core in Visual Studio

Article • 06/18/2024

By Scott Addie

Visual Studio has built-in support for LibMan in ASP.NET Core projects, including:

- Support for configuring and running LibMan restore operations on build.
- Menu items for triggering LibMan restore and clean operations.
- Search dialog for finding libraries and adding the files to a project.
- Editing support for `libman.json`—the LibMan manifest file.

View or download sample code (how to download)

Prerequisites

- Visual Studio 2019 with the **ASP.NET and web development** workload

Add library files

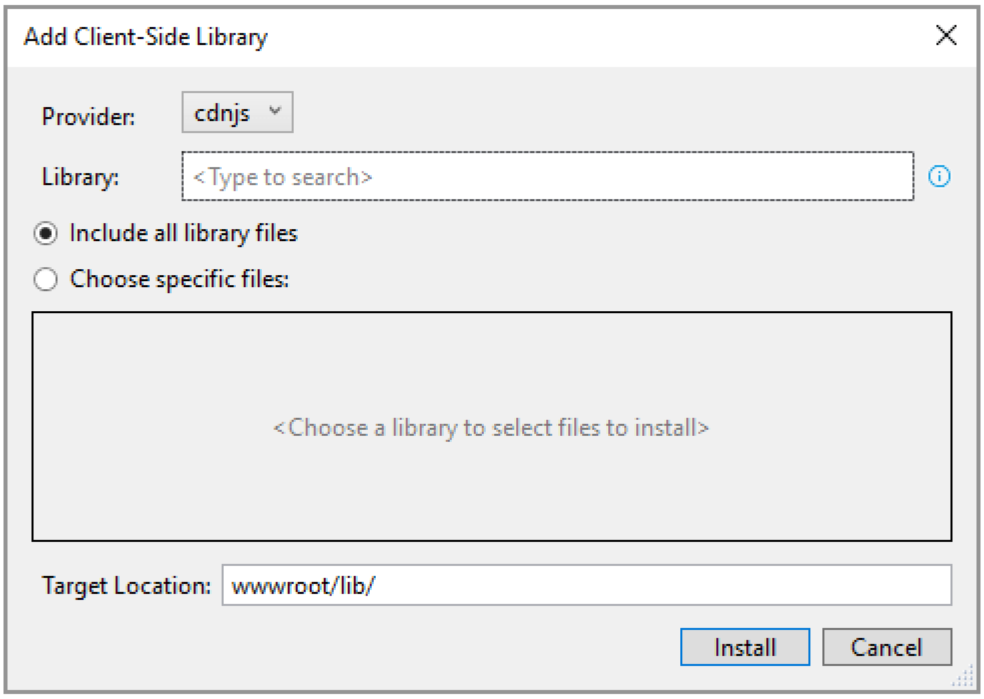
Library files can be added to an ASP.NET Core project in two different ways:

1. Use the Add Client-Side Library dialog
2. Manually configure LibMan manifest file entries

Use the Add Client-Side Library dialog

Follow these steps to install a client-side library:

- In **Solution Explorer**, right-click the project folder in which the files should be added. Choose **Add** > **Client-Side Library**. The **Add Client-Side Library** dialog appears:


Add Client-Side Library
Provider: cdnjs
Library: <Type to search>
Include all library files
Choose specific files:
<Choose a library to select files to install>
Target Location: wwwroot/lib/
Install
Cancel

- Select the library provider from the **Provider** drop down. CDNJS is the default provider.
- Type the library name to fetch in the **Library** text box. IntelliSense provides a list of libraries beginning with the provided text.
- Select the library from the IntelliSense list. Notice the library name is suffixed with the `@` symbol and the latest stable version known to the selected provider.
- Decide which files to include:
  - Select the **Include all library files** radio button to include all of the library's files.
  - Select the **Choose specific files** radio button to include a subset of the library's files. When the radio button is selected, the file selector tree is enabled. Check the boxes to the left of the file names to download.
- Specify the project folder for storing the files in the **Target Location** text box. As a recommendation, store each library in a separate folder.

  The suggested **Target Location** folder is based on the location from which the dialog launched:
  - If launched from the project root:
    - *wwwroot/lib* is used if *wwwroot* exists.
    - *lib* is used if *wwwroot* doesn't exist.
  - If launched from a project folder, the corresponding folder name is used.

  The folder suggestion is suffixed with the library name. The following table illustrates folder suggestions when installing jQuery in a Razor Pages project.

Expand table

Launch location	Suggested folder
project root (if *wwwroot* exists)	*wwwroot/lib/jquery/*
project root (if *wwwroot* doesn't exist)	*lib/jquery/*
Pages folder in project	*Pages/jquery/*

- Click the **Install** button to download the files, per the configuration in `libman.json`.
- Review the **Library Manager** feed of the **Output** window for installation details. For example:

Console

```
Restore operation started...
Restoring libraries for project LibManSample
Restoring library jquery@3.3.1... (LibManSample)
wwwroot/lib/jquery/jquery.min.js written to destination (LibManSample)
wwwroot/lib/jquery/jquery.js written to destination (LibManSample)
wwwroot/lib/jquery/jquery.min.map written to destination (LibManSample)
Restore operation completed
1 libraries restored in 2.32 seconds
```

Manually configure LibMan manifest file entries

All LibMan operations in Visual Studio are based on the content of the project root's LibMan manifest (`libman.json`). You can manually edit `libman.json` to configure library files for the project. Visual Studio restores all library files once `libman.json` is saved.

To open `libman.json` for editing, the following options exist:

- Double-click the `libman.json` file in **Solution Explorer**.
- Right-click the project in **Solution Explorer** and select **Manage Client-Side Libraries**. †
- Select **Manage Client-Side Libraries** from the Visual Studio **Project** menu. †

† If the `libman.json` file doesn't already exist in the project root, it will be created with the default item template content.

Visual Studio offers rich JSON editing support such as colorization, formatting, IntelliSense, and schema validation. The LibMan manifest's JSON schema is found at https://json.schemastore.org/libman .

With the following manifest file, LibMan retrieves files per the configuration defined in the `libraries` property. An explanation of the object literals defined within `libraries` follows:

- A subset of jQuery version 3.3.1 is retrieved from the CDNJS provider. The subset is defined in the `files` property—`jquery.min.js`, `jquery.js`, and *jquery.min.map*. The files are placed in the project's *wwwroot/lib/jquery* folder.
- The entirety of Bootstrap version 4.1.3 is retrieved and placed in a *wwwroot/lib/bootstrap* folder. The object literal's `provider` property overrides the `defaultProvider` property value. LibMan retrieves the Bootstrap files from the unpkg provider.
- A subset of Lodash was approved by a governing body within the organization. The `lodash.js` and `lodash.min.js` files are retrieved from the local file system at *C:\temp\lodash\*. The files are copied to the project's *wwwroot/lib/lodash* folder.

JSON

```json
{
  "version": "1.0",
  "defaultProvider": "cdnjs",
  "libraries": [
    {
      "library": "jquery@3.3.1",
      "files": [
        "jquery.min.js",
        "jquery.js",
        "jquery.min.map"
      ],
      "destination": "wwwroot/lib/jquery/"
    },
    {
      "provider": "unpkg",
      "library": "bootstrap@4.1.3",
      "destination": "wwwroot/lib/bootstrap/"
    },
    {
      "provider": "filesystem",
      "library": "C:\\temp\\lodash\\",
      "files": [
        "lodash.js",
        "lodash.min.js"
      ],
      "destination": "wwwroot/lib/lodash/"
    }
  ]
}
```

ⓘ **Note**

LibMan only supports one version of each library from each provider. The `libman.json` file fails schema validation if it contains two libraries with the same library name for a given provider.

Restore library files

To restore library files from within Visual Studio, there must be a valid `libman.json` file in the project root. Restored files are placed in the project at the location specified for each library.

Library files can be restored in an ASP.NET Core project in two ways:

1. Restore files during build
2. Restore files manually

Restore files during build

LibMan can restore the defined library files as part of the build process. By default, the *restore-on-build* behavior is disabled.

To enable and test the restore-on-build behavior:

- Right-click `libman.json` in **Solution Explorer** and select **Enable Restore Client-Side Libraries on Build** from the context menu.

- Click the **Yes** button when prompted to install a NuGet package. The Microsoft.Web.LibraryManager.Build NuGet package is added to the project:

  XML

  ```xml
  <PackageReference Include="Microsoft.Web.LibraryManager.Build"
  Version="1.0.113" />
  ```

- Build the project to confirm LibMan file restoration occurs. The `Microsoft.Web.LibraryManager.Build` package injects an MSBuild target that runs LibMan during the project's build operation.

- Review the **Build** feed of the **Output** window for a LibMan activity log:

  Console

```
1>------ Build started: Project: LibManSample, Configuration: Debug Any
CPU ------
1>
1>Restore operation started...
1>Restoring library jquery@3.3.1...
1>Restoring library bootstrap@4.1.3...
1>
1>2 libraries restored in 10.66 seconds
1>LibManSample ->
C:\LibManSample\bin\Debug\netcoreapp2.1\LibManSample.dll
========== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped
==========
```

When the restore-on-build behavior is enabled, the `libman.json` context menu displays a **Disable Restore Client-Side Libraries on Build** option. Selecting this option removes the `Microsoft.Web.LibraryManager.Build` package reference from the project file. Consequently, the client-side libraries are no longer restored on each build.

Regardless of the restore-on-build setting, you can manually restore at any time from the `libman.json` context menu. For more information, see Restore files manually.

Restore files manually

To manually restore library files:

- For all projects in the solution:
  - Right-click the solution name in **Solution Explorer**.
  - Select the **Restore Client-Side Libraries** option.
- For a specific project:
  - Right-click the `libman.json` file in **Solution Explorer**.
  - Select the **Restore Client-Side Libraries** option.

While the restore operation is running:

- The Task Status Center (TSC) icon on the Visual Studio status bar will be animated and will read *Restore operation started*. Clicking the icon opens a tooltip listing the known background tasks.

- Messages will be sent to the status bar and the **Library Manager** feed of the **Output** window. For example:

  Console

  ```
  Restore operation started...
  Restoring libraries for project LibManSample
  ```

```
Restoring library jquery@3.3.1... (LibManSample)
wwwroot/lib/jquery/jquery.min.js written to destination (LibManSample)
wwwroot/lib/jquery/jquery.js written to destination (LibManSample)
wwwroot/lib/jquery/jquery.min.map written to destination (LibManSample)
Restore operation completed
1 libraries restored in 2.32 seconds
```

Delete library files

To perform the *clean* operation, which deletes library files previously restored in Visual Studio:

- Right-click the `libman.json` file in **Solution Explorer**.
- Select the **Clean Client-Side Libraries** option.

To prevent unintentional removal of non-library files, the clean operation doesn't delete whole directories. It only removes files that were included in the previous restore.

While the clean operation is running:

- The TSC icon on the Visual Studio status bar will be animated and will read *Client libraries operation started*. Clicking the icon opens a tooltip listing the known background tasks.
- Messages are sent to the status bar and the **Library Manager** feed of the **Output** window. For example:

Console

```
Clean libraries operation started...
Clean libraries operation completed
2 libraries were successfully deleted in 1.91 secs
```

The clean operation only deletes files from the project. Library files stay in the cache for faster retrieval on future restore operations. To manage library files stored in the local machine's cache, use the LibMan CLI.

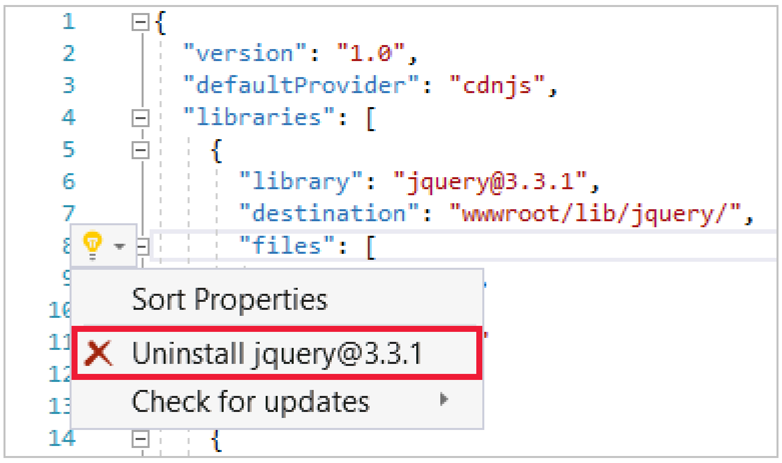
Uninstall library files

To uninstall library files:

- Open `libman.json`.

- Position the caret inside the corresponding `libraries` object literal.

- Click the light bulb icon that appears in the left margin, and select **Uninstall <library_name>@<library_version>**:



```
{
  "version": "1.0",
  "defaultProvider": "cdnjs",
  "libraries": [
    {
      "library": "jquery@3.3.1",
      "destination": "wwwroot/lib/jquery/",
      "files": [
```

Alternatively, you can manually edit and save the LibMan manifest (`libman.json`). The restore operation runs when the file is saved. Library files that are no longer defined in `libman.json` are removed from the project.

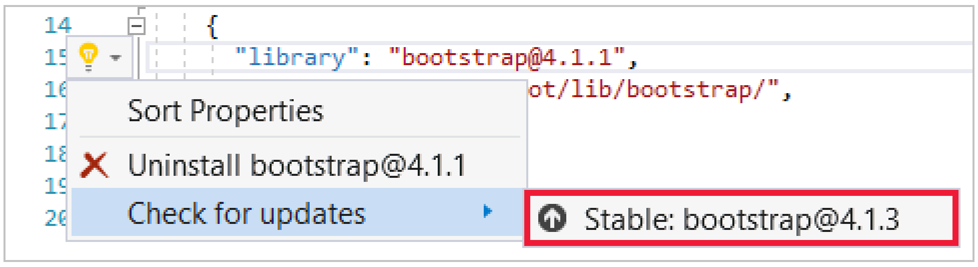
Update library version

To check for an updated library version:

- Open `libman.json`.
- Position the caret inside the corresponding `libraries` object literal.
- Click the light bulb icon that appears in the left margin. Hover over **Check for updates**.

LibMan checks for a library version newer than the version installed. The following outcomes can occur:

- A **No updates found** message is displayed if the latest version is already installed.
- The latest stable version is displayed if not already installed.



```
    {
      "library": "bootstrap@4.1.1",
                                    ot/lib/bootstrap/",
```

- If a pre-release newer than the installed version is available, the pre-release is displayed.

To downgrade to an older library version, manually edit the `libman.json` file. When the file is saved, the LibMan restore operation:

- Removes redundant files from the previous version.
- Adds new and updated files from the new version.

Additional resources

- Use the LibMan CLI with ASP.NET Core
- LibMan GitHub repository

JavaScript `[JSImport]`/`[JSExport]` interop in .NET WebAssembly

Article • 08/22/2024

ⓘ **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 Aaron Shumaker

This article explains how to interact with JavaScript (JS) in client-side WebAssembly using JS `[JSImport]`/`[JSExport]` interop (System.Runtime.InteropServices.JavaScript API).

`[JSImport]`/`[JSExport]` interop is applicable when running a .NET WebAssembly module in a JS host in the following scenarios:

- JavaScript `[JSImport]`/`[JSExport]` interop with a WebAssembly Browser App project.
- JavaScript JSImport/JSExport interop with ASP.NET Core Blazor.
- Other .NET WebAssembly platforms that support `[JSImport]`/`[JSExport]` interop.

Prerequisites

.NET SDK (latest version)

Any of the following project types:

- A WebAssembly Browser App project created according to JavaScript `[JSImport]`/`[JSExport]` interop with a WebAssembly Browser App project.
- A Blazor client-side project created according to JavaScript JSImport/JSExport interop with ASP.NET Core Blazor.
- A project created for a commercial or open-source platform that supports `[JSImport]`/`[JSExport]` interop (System.Runtime.InteropServices.JavaScript API).

Sample app

View or download sample code (how to download): Select an 8.0 or later version folder that matches the version of .NET that you're adopting. Within the version folder, access the sample named `WASMBrowserAppImportExportInterop`.

JS interop using `[JSImport]`/`[JSExport]` attributes

The `[JSImport]` attribute is applied to a .NET method to indicate that a corresponding JS method should be called when the .NET method is called. This allows .NET developers to define "imports" that enable .NET code to call into JS. Additionally, an Action can be passed as a parameter, and JS can invoke the action to support a callback or event subscription pattern.

The `[JSExport]` attribute is applied to a .NET method to expose it to JS code. This allows JS code to initiate calls to the .NET method.

Importing JS methods

The following example imports a standard built-in JS method (`console.log`) into C#. `[JSImport]` is limited to importing methods of globally-accessible objects. For example, `log` is a method defined on the `console` object, which is defined on the globally-accessible object `globalThis`. The `console.log` method is mapped to a C# proxy method, `ConsoleLog`, which accepts a string for the log message:

```C#
public partial class GlobalInterop
{
    [JSImport("globalThis.console.log")]
    public static partial void ConsoleLog(string text);
}
```

In `Program.Main`, `ConsoleLog` is called with the message to log:

```C#
GlobalInterop.ConsoleLog("Hello World!");
```

The output appears in the browser's console.

The following demonstrates importing a method declared in JS.

The following custom JS method (`globalThis.callAlert`) spawns an alert dialog (window.alert) with the message passed in `text`:

JavaScript

```javascript
globalThis.callAlert = function (text) {
  globalThis.window.alert(text);
}
```

The `globalThis.callAlert` method is mapped to a C# proxy method (`CallAlert`), which accepts a string for the message:

C#

```csharp
using System.Runtime.InteropServices.JavaScript;

public partial class GlobalInterop
{
    [JSImport("globalThis.callAlert")]
    public static partial void CallAlert(string text);
}
```

In `Program.Main`, `CallAlert` is called, passing the text for the alert dialog message:

C#

```csharp
GlobalInterop.CallAlert("Hello World");
```

The C# class declaring the `[JSImport]` method doesn't have an implementation. At compile time, a source-generated partial class contains the .NET code that implements the marshalling of the call and types to invoke the corresponding JS method. In Visual Studio, using the **Go To Definition** or **Go To Implementation** options respectively navigates to either the source-generated partial class or the developer-defined partial class.

In the preceding example, the intermediate `globalThis.callAlert` JS declaration is used to wrap existing JS code. This article informally refers to the intermediate JS declaration as a *JS shim*. JS shims fill the gap between the .NET implementation and existing JS capabilities/libraries. In many cases, such as the preceding trivial example, the JS shim isn't necessary, and methods could be imported directly, as demonstrated in the earlier `ConsoleLog` example. As this article demonstrates in the upcoming sections, a JS shim can:

- Encapsulate additional logic.
- Manually map types.
- Reduce the number of objects or calls crossing the interop boundary.
- Manually map static calls to instance methods.

Loading JavaScript declarations

JS declarations which are intended to be imported with `[JSImport]` are typically loaded in the context of the same page or JS host that loaded .NET WebAssembly. This can be accomplished with:

- A `<script>...</script>` block declaring inline JS.
- A script source (`src`) declaration (`<script src="./some.js"></script>`) that loads an external JS file (`.js`).
- A JS ES6 module (`<script type='module' src="./moduleName.js"></script>`).
- A JS ES6 module loaded using JSHost.ImportAsync from .NET WebAssembly.

Examples in this article use JSHost.ImportAsync. When calling ImportAsync, client-side .NET WebAssembly requests the file using the `moduleUrl` parameter, and thus it expects the file to be accessible as a static web asset, much the same way as a `<script>` tag retrieves a file with a `src` URL. For example, the following C# code within a WebAssembly Browser App project maintains the JS file (`.js`) at the path `/wwwroot/scripts/ExampleShim.js`:

```C#
await JSHost.ImportAsync("ExampleShim", "/scripts/ExampleShim.js");
```

Depending on the platform that's loading WebAssembly, a dot-prefixed URL, such as `./scripts/`, might refer to an incorrect subdirectory, such as `/_framework/scripts/`, because the WebAssembly package is initialized by framework scripts under `/_framework/`. In that case, prefixing the URL with `../scripts/` refers to the correct path. Prefixing with `/scripts/` works if the site is hosted at the root of the domain. A typical approach involves configuring the correct base path for the given environment with an HTML `<base>` tag and using the `/scripts/` prefix to refer to the path relative to the base path. Tilde notation `~/` prefixes aren't supported by JSHost.ImportAsync.

ⓘ **Important**