

ASP.NET Core Blazor

Article • 07/15/2022 • 7 minutes to read • [10 contributors](#)



In this article

[Components](#)

[Blazor Server](#)

[Blazor WebAssembly](#)

[Blazor Hybrid](#)

[JavaScript interop](#)

[Code sharing and .NET Standard](#)

[Additional resources](#)

Welcome to Blazor!

Blazor is a framework for building interactive client-side web UI with [.NET](#):

- Create rich interactive UIs using [C#](#) instead of [JavaScript](#) .
- Share server-side and client-side app logic written in .NET.
- Render the UI as HTML and CSS for wide browser support, including mobile browsers.
- Integrate with modern hosting platforms, such as [Docker](#).
- Build hybrid desktop and mobile apps with .NET and Blazor.

Using .NET for client-side web development offers the following advantages:

- Write code in C# instead of JavaScript.
- Leverage the existing .NET ecosystem of [.NET libraries](#).
- Share app logic across server and client.
- Benefit from .NET's performance, reliability, and security.
- Stay productive on Windows, Linux, or macOS with a development environment, such as [Visual Studio](#) or [Visual Studio Code](#) .
- Build on a common set of languages, frameworks, and tools that are stable, feature-rich, and easy to use.

Note

For a Blazor quick start tutorial, see [Build your first Blazor app](#) .

Components

Blazor apps are based on *components*. A component in Blazor is an element of UI, such as a page, dialog, or data entry form.

Components are .NET C# classes built into [.NET assemblies](#) that:

- Define flexible UI rendering logic.
- Handle user events.
- Can be nested and reused.
- Can be shared and distributed as [Razor class libraries](#) or [NuGet packages](#).

The component class is usually written in the form of a [Razor](#) markup page with a `.razor` file extension. Components in Blazor are formally referred to as *Razor components*, informally as *Blazor components*. Razor is a syntax for combining HTML markup with C# code designed for developer productivity. Razor allows you to switch between HTML markup and C# in the same file with [IntelliSense](#) programming support in Visual Studio. Razor Pages and MVC also use Razor. Unlike Razor Pages and MVC, which are built around a request/response model, components are used specifically for client-side UI logic and composition.


Blazor uses natural HTML tags for UI composition. The following Razor markup demonstrates a component (`Dialog.razor`) that displays a dialog and processes an event when the user selects a button:

razor	 Copy
<pre><div class="card" style="width:22rem"> <div class="card-body"> <h3 class="card-title">@Title</h3> <p class="card-text">@ChildContent</p> <button @onclick="OnYes">Yes!</button> </div> </div> @code { [Parameter] public RenderFragment? ChildContent { get; set; } [Parameter] public string? Title { get; set; } private void OnYes() { Console.WriteLine("Write to the console in C#! 'Yes' button selected."); } }</pre>	

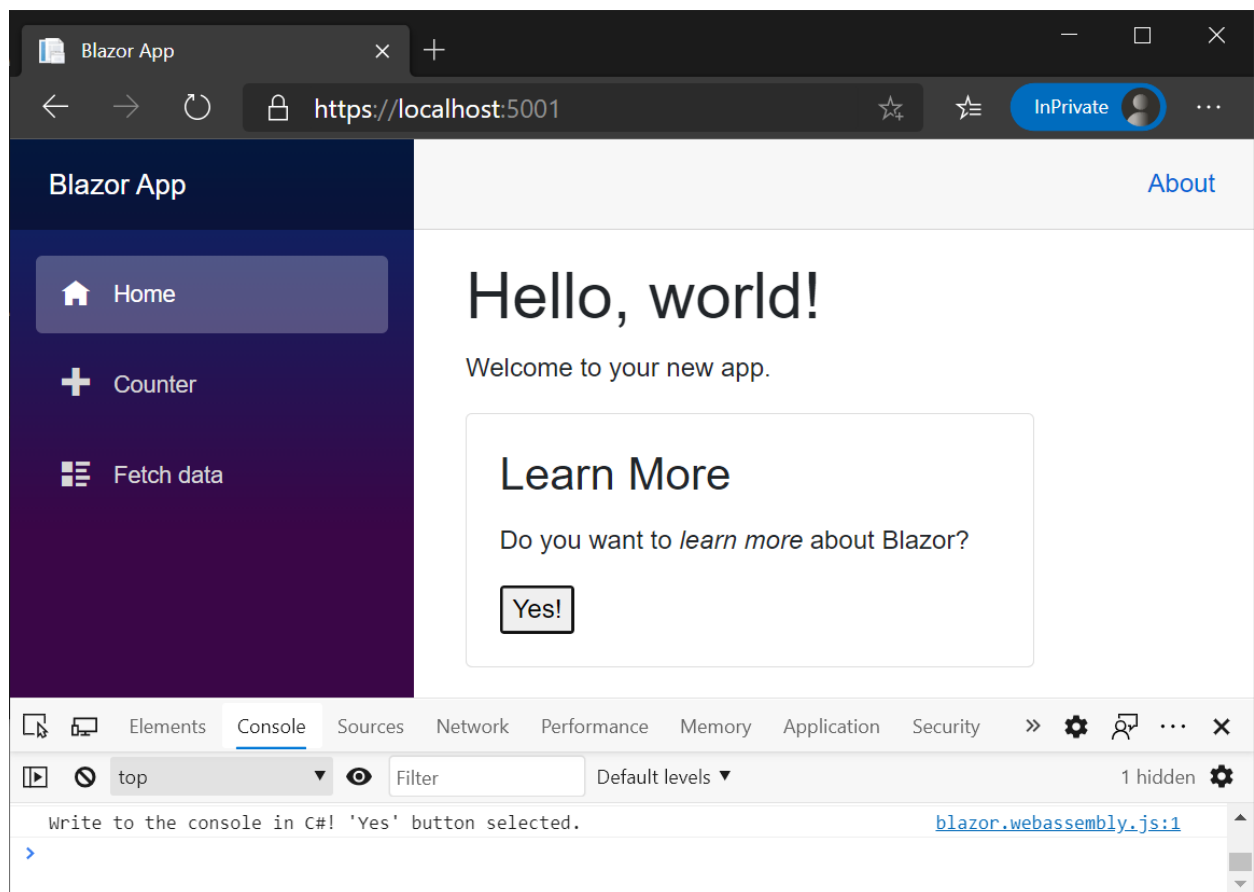
```
}  
}
```

In the preceding example, `OnYes` is a C# method triggered by the button's `onclick` event. The dialog's text (`ChildContent`) and title (`Title`) are provided by the following component that uses this component in its UI.

The `Dialog` component is nested within another component using an HTML tag. In the following example, the `Index` component (`Pages/Index.razor`) uses the preceding `Dialog` component. The tag's `Title` attribute passes a value for the title to the `Dialog` component's `Title` property. The `Dialog` component's text (`ChildContent`) are set by the content of the `<Dialog>` element. When the `Dialog` component is added to the `Index` component, [IntelliSense in Visual Studio](#) speeds development with syntax and parameter completion.

razor	 Copy
<pre>@page "/" <h1>Hello, world!</h1> <p> Welcome to your new app. </p> <Dialog Title="Learn More"> Do you want to <i>learn more</i> about Blazor? </Dialog></pre>	

The dialog is rendered when the `Index` component is accessed in a browser. When the button is selected by the user, the browser's developer tools console shows the message written by the `OnYes` method:



Components render into an in-memory representation of the browser's [Document Object Model \(DOM\)](#) called a *render tree*, which is used to update the UI in a flexible and efficient way.

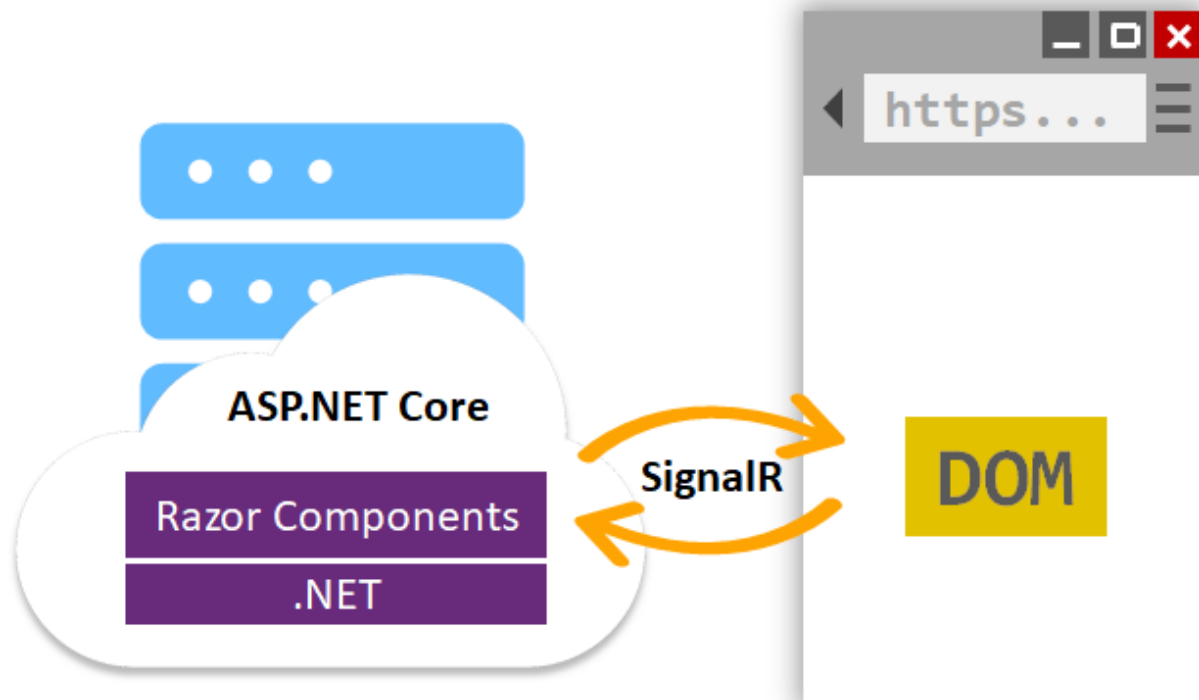
Blazor Server

Blazor Server provides support for hosting Razor components on the server in an ASP.NET Core app. UI updates are handled over a [SignalR](#) connection.

The runtime stays on the server and handles:

- Executing the app's C# code.
- Sending UI events from the browser to the server.
- Applying UI updates to a rendered component that are sent back by the server.

The connection used by Blazor Server to communicate with the browser is also used to handle JavaScript interop calls.



Blazor Server apps render content differently than traditional models for rendering UI in ASP.NET Core apps using Razor views or Razor Pages. Both models use the [Razor language](#) to describe HTML content for rendering, but they significantly differ in *how* markup is rendered.

When a Razor Page or view is rendered, every line of Razor code emits HTML in text form. After rendering, the server disposes of the page or view instance, including any state that was produced. When another request for the page occurs, the entire page is rerendered to HTML again and sent to the client.

Blazor Server produces a graph of components to display similar to an HTML or XML Document Object Model (DOM). The component graph includes state held in properties and fields. Blazor evaluates the component graph to produce a binary representation of the markup, which is sent to the client for rendering. After the connection is made between the client and the server, the component's static prerendered elements are replaced with interactive elements. Prerendering the content on the server makes the app feel more responsive on the client.

After the components are interactive on the client, UI updates are triggered by user interaction and app events. When an update occurs, the component graph is rerendered, and a UI *diff* (difference) is calculated. This diff is the smallest set of DOM edits required to update the UI on the client. The diff is sent to the client in a binary format and applied by the browser.

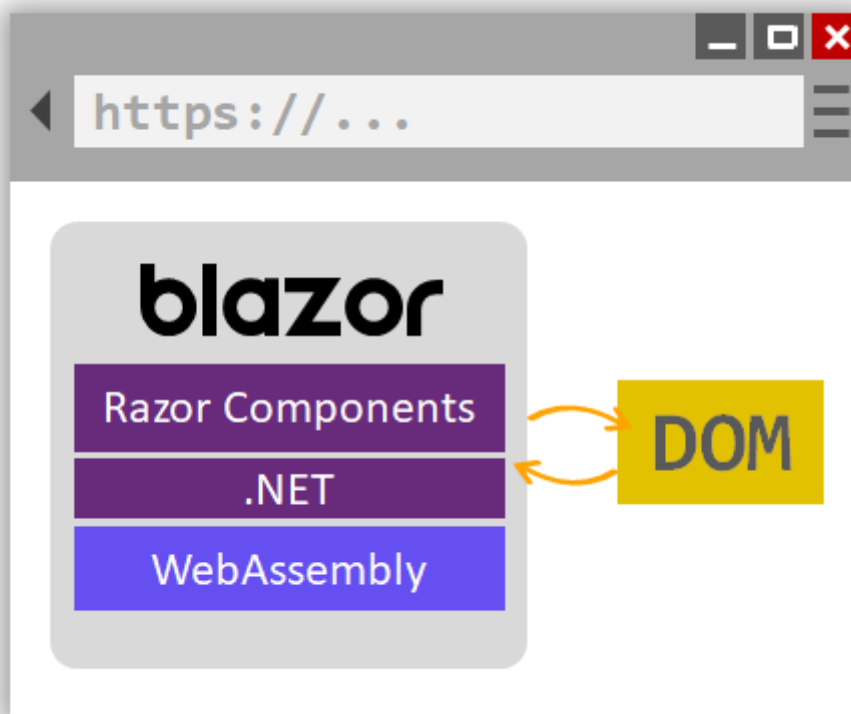
A component is disposed after the user navigates away from the component.

Blazor WebAssembly

Blazor WebAssembly is a [single-page app \(SPA\) framework](#) for building interactive client-side web apps with .NET. Blazor WebAssembly uses open web standards without plugins or recompiling code into other languages. Blazor WebAssembly works in all modern web browsers, including mobile browsers.

Running .NET code inside web browsers is made possible by [WebAssembly](#) (abbreviated `wasm`). WebAssembly is a compact bytecode format optimized for fast download and maximum execution speed. WebAssembly is an open web standard and supported in web browsers without plugins.

WebAssembly code can access the full functionality of the browser via JavaScript, called *JavaScript interoperability*, often shortened to *JavaScript interop* or *JS interop*. .NET code executed via WebAssembly in the browser runs in the browser's JavaScript sandbox with the protections that the sandbox provides against malicious actions on the client machine.



When a Blazor WebAssembly app is built and run in a browser:

- C# code files and Razor files are compiled into .NET assemblies.
- The assemblies and the [.NET runtime](#) are downloaded to the browser.
- Blazor WebAssembly bootstraps the .NET runtime and configures the runtime to load the assemblies for the app. The Blazor WebAssembly runtime uses JavaScript interop to handle DOM manipulation and browser API calls.

The size of the published app, its *payload size*, is a critical performance factor for an app's usability. A large app takes a relatively long time to download to a browser, which diminishes the user experience. Blazor WebAssembly optimizes payload size to reduce download times:

- Unused code is stripped out of the app when it's published by the [Intermediate Language \(IL\) Trimmer](#).
- HTTP responses are compressed.
- The .NET runtime and assemblies are cached in the browser.

Blazor Hybrid

Hybrid apps use a blend of native and web technologies. A *Blazor Hybrid* app uses Blazor in a native client app. Razor components run natively in the .NET process and render web UI to an embedded Web View control using a local interop channel. WebAssembly isn't used in Hybrid apps. Hybrid apps encompass the following technologies:

- [.NET Multi-platform App UI \(.NET MAUI\)](#): A cross-platform framework for creating native mobile and desktop apps with C# and XAML.
- [Windows Presentation Foundation \(WPF\)](#): A UI framework that is resolution-independent and uses a vector-based rendering engine, built to take advantage of modern graphics hardware.
- [Windows Forms](#): A UI framework that creates rich desktop client apps for Windows. The Windows Forms development platform supports a broad set of app development features, including controls, graphics, data binding, and user input.

For more information on creating Blazor Hybrid apps with the preceding frameworks, see the following articles:

- [ASP.NET Core Blazor hosting models](#)
- [ASP.NET Core Blazor Hybrid](#)

JavaScript interop

For apps that require third-party JavaScript libraries and access to browser APIs, components interoperate with JavaScript. Components are capable of using any library or API that JavaScript is able to use. C# code can [call into JavaScript code](#), and JavaScript code can [call into C# code](#).

Code sharing and .NET Standard

Blazor implements the [.NET Standard](#), which enables Blazor projects to reference libraries that conform to .NET Standard specifications. .NET Standard is a formal specification of .NET APIs that are common across .NET implementations. .NET Standard class libraries can be shared across different .NET platforms, such as Blazor, .NET Framework, .NET Core, Xamarin, Mono, and Unity.

APIs that aren't applicable inside of a web browser (for example, accessing the file system, opening a socket, and threading) throw a [PlatformNotSupportedException](#).

Additional resources

- [WebAssembly](#)
- [ASP.NET Core Blazor hosting models](#)
- [Use ASP.NET Core SignalR with Blazor](#)
- [Call JavaScript functions from .NET methods in ASP.NET Core Blazor](#)
- [Call .NET methods from JavaScript functions in ASP.NET Core Blazor](#)
- [mono/mono GitHub repository](#)
- [C# Guide](#)
- [Razor syntax reference for ASP.NET Core](#)
- [HTML](#)
- [Awesome Blazor](#) community links
- [Blazor samples GitHub repository \(dotnet/blazor-samples\)](#)

Recommended content

[Build a Blazor todo list app](#)

Build a Blazor app step-by-step.

[ASP.NET Core Blazor supported platforms](#)

Learn about the supported platforms for ASP.NET Core Blazor.

[Blazor app hosting models](#)

Learn the different ways to host a Blazor app, including in the browser on WebAssembly or on the server.

[Architecture comparison of ASP.NET Web Forms and Blazor](#)

Learn how the architectures of ASP.NET Web Forms and Blazor compare.

[ASP.NET Core Blazor hosting models](#)

Learn about Blazor hosting models and how to pick which one to use.

[Tooling for ASP.NET Core Blazor](#)

Learn about the tools available to build Blazor apps on various platforms.

[ASP.NET Core Blazor tutorials](#)

Learn how to build Blazor apps with the tutorials listed in this article.

[Project structure for Blazor apps](#)

Learn how the project structures of ASP.NET Web Forms and Blazor projects compare.

Show less ^