Table of Contents

Get started with Blazor

Note

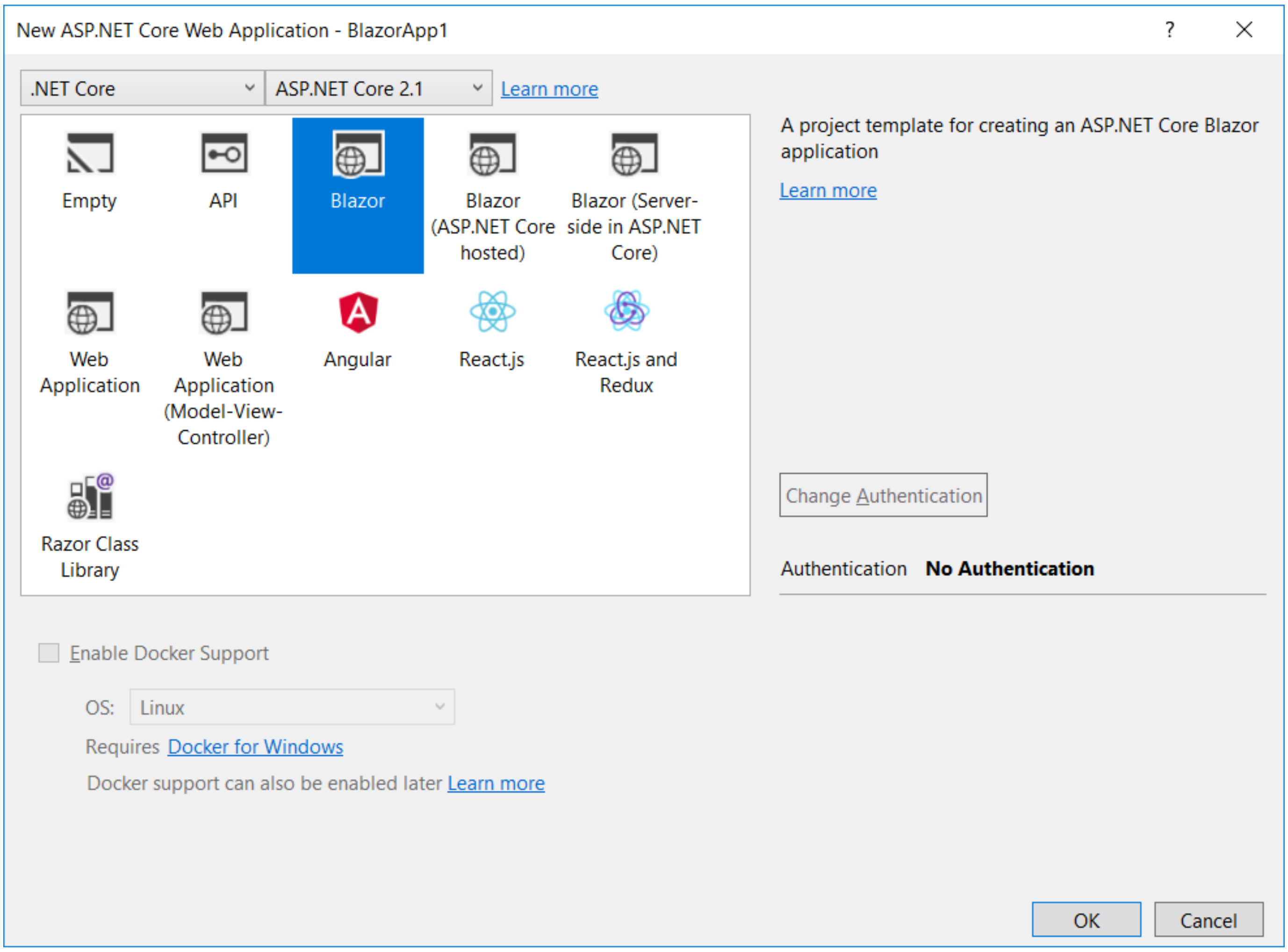
Blazor is an unsupported experimental web framework that shouldn't be used for production workloads at this time.

Setup

1. Install the .NET Core 2.1 SDK (2.1.300 or later).
2. Install Visual Studio 2017 (15.7 or later) with the *ASP.NET and web development* workload selected.
3. Install the latest Blazor Language Services extension from the Visual Studio Marketplace.

To create your first project from Visual Studio:

1. Select **File** > **New Project** > **Web** > **ASP.NET Core Web Application**.
2. Make sure **.NET Core** and **ASP.NET Core 2.1** (or later) are selected at the top.
3. Choose the Blazor template and select **OK**.


New ASP.NET Core Web Application - BlazorApp1

.NET Core | ASP.NET Core 2.1 | Learn more

Empty, API, Blazor (selected), Blazor (ASP.NET Core hosted), Blazor (Server-side in ASP.NET Core), Web Application, Web Application (Model-View-Controller), Angular, React.js, React.js and Redux, Razor Class Library

A project template for creating an ASP.NET Core Blazor application

Learn more

Authentication No Authentication

Enable Docker Support

OS: Linux

Requires Docker for Windows

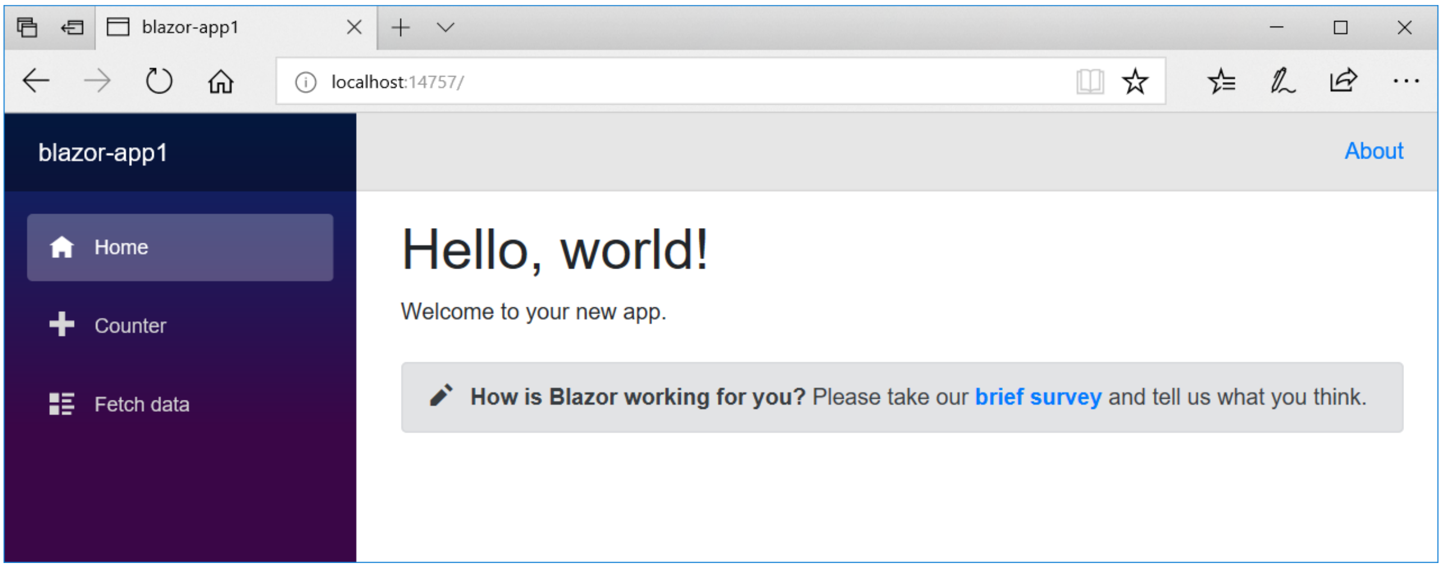
Docker support can also be enabled later Learn more

4. Press **Ctrl-F5** to run the app *without the debugger*. Running with the debugger (**F5**) isn't supported at this time.

You can also install and use the Blazor templates from the command-line:

```
dotnet new -i Microsoft.AspNetCore.Blazor.Templates
dotnet new blazor -o BlazorApp1
cd BlazorApp1
dotnet run
```

Congrats! You just ran your first Blazor app!



Hello, world!

Welcome to your new app.

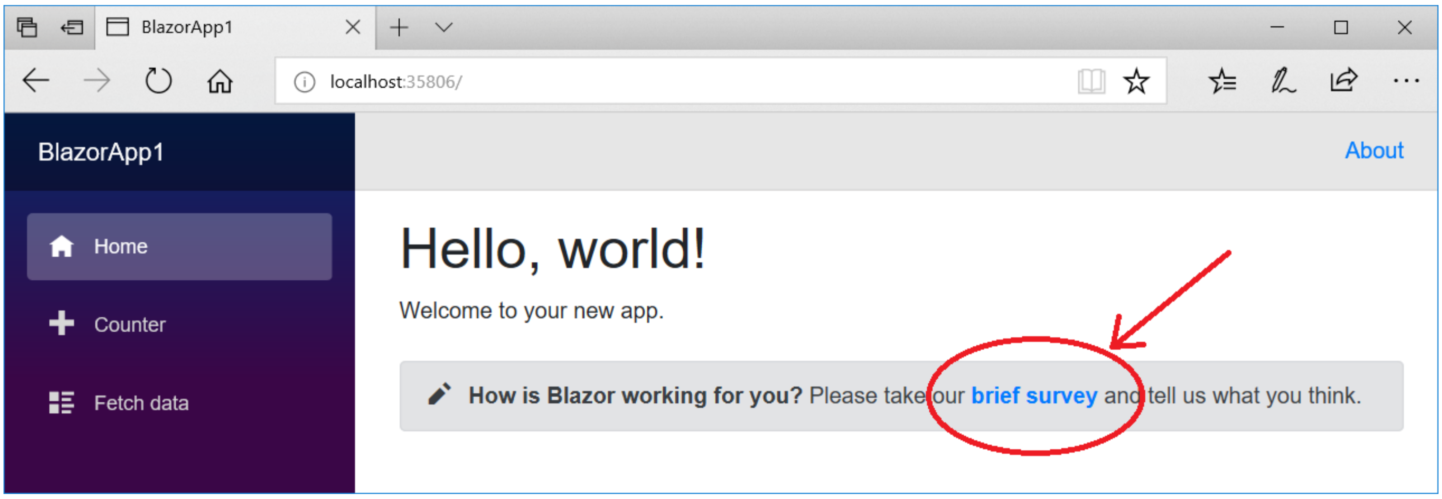
How is Blazor working for you? Please take our **brief survey** and tell us what you think.

Help & feedback

Your feedback is especially important to us during this experimental phase for Blazor. If you run into issues or have questions while trying out Blazor, please let us know!

- File issues on GitHub for any problems you run into or to make suggestions for improvements.
- Chat with us and the Blazor community on Gitter if you get stuck or to share how Blazor is working for you.

After you've tried out Blazor, please let us know what you think by taking our in-product survey. Just click the survey link shown on the app home page when running one of the Blazor project templates:



Hello, world!

Welcome to your new app.

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

What's next?

Build your first Blazor app

Build your first Blazor app

 Note

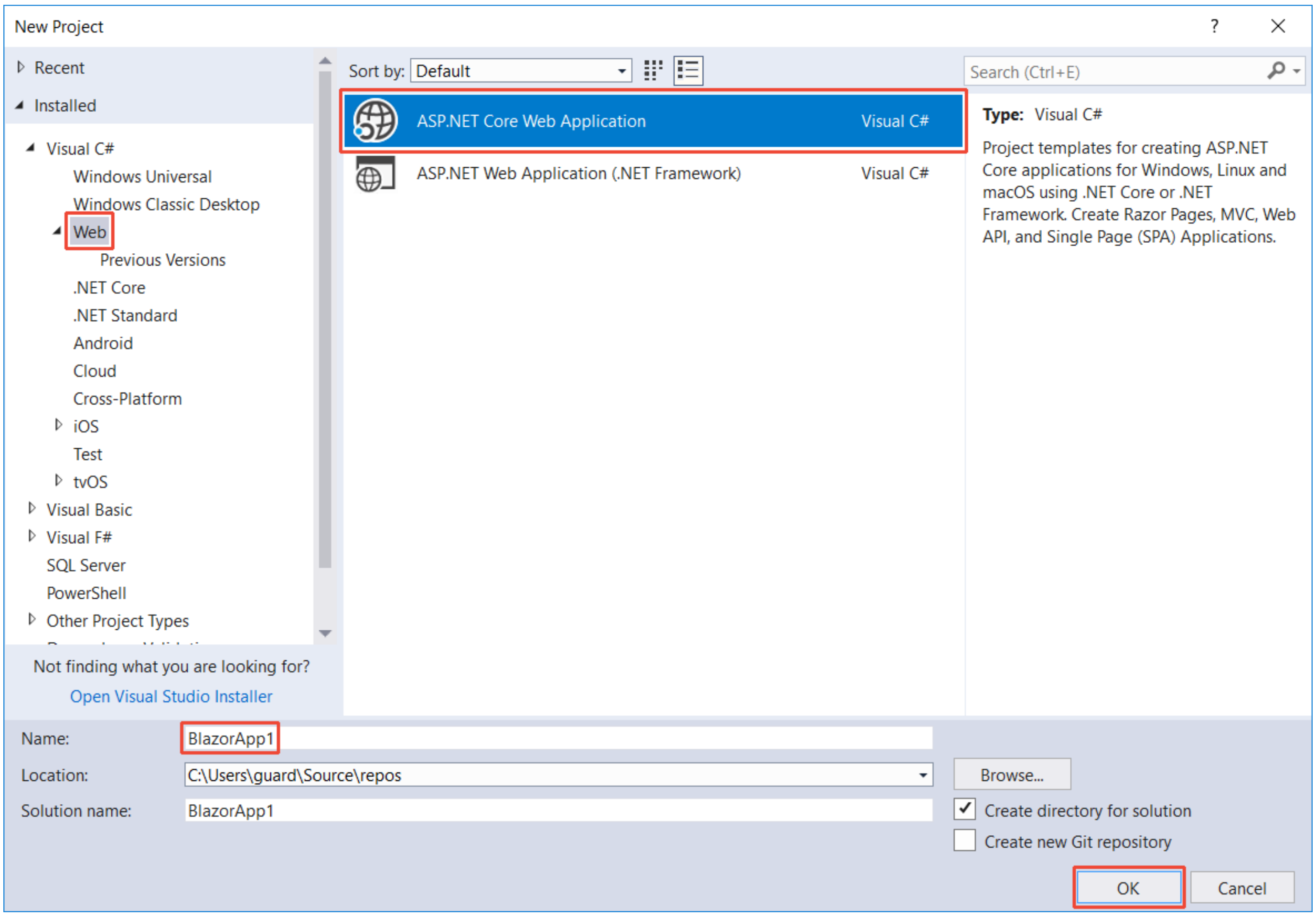
Blazor is an unsupported experimental web framework that shouldn't be used for production workloads at this time.

In this tutorial, you build a Blazor app step-by-step and quickly learn the basic features of the Blazor framework.

View or download sample code (how to download). See the Get started topic for prerequisites.

To create the project in Visual Studio:

1. Select **File** > **New** > **Project**. Select **Web** > **ASP.NET Core Web Application**. Name the project "BlazorApp1" in the **Name** field. Select **OK**.

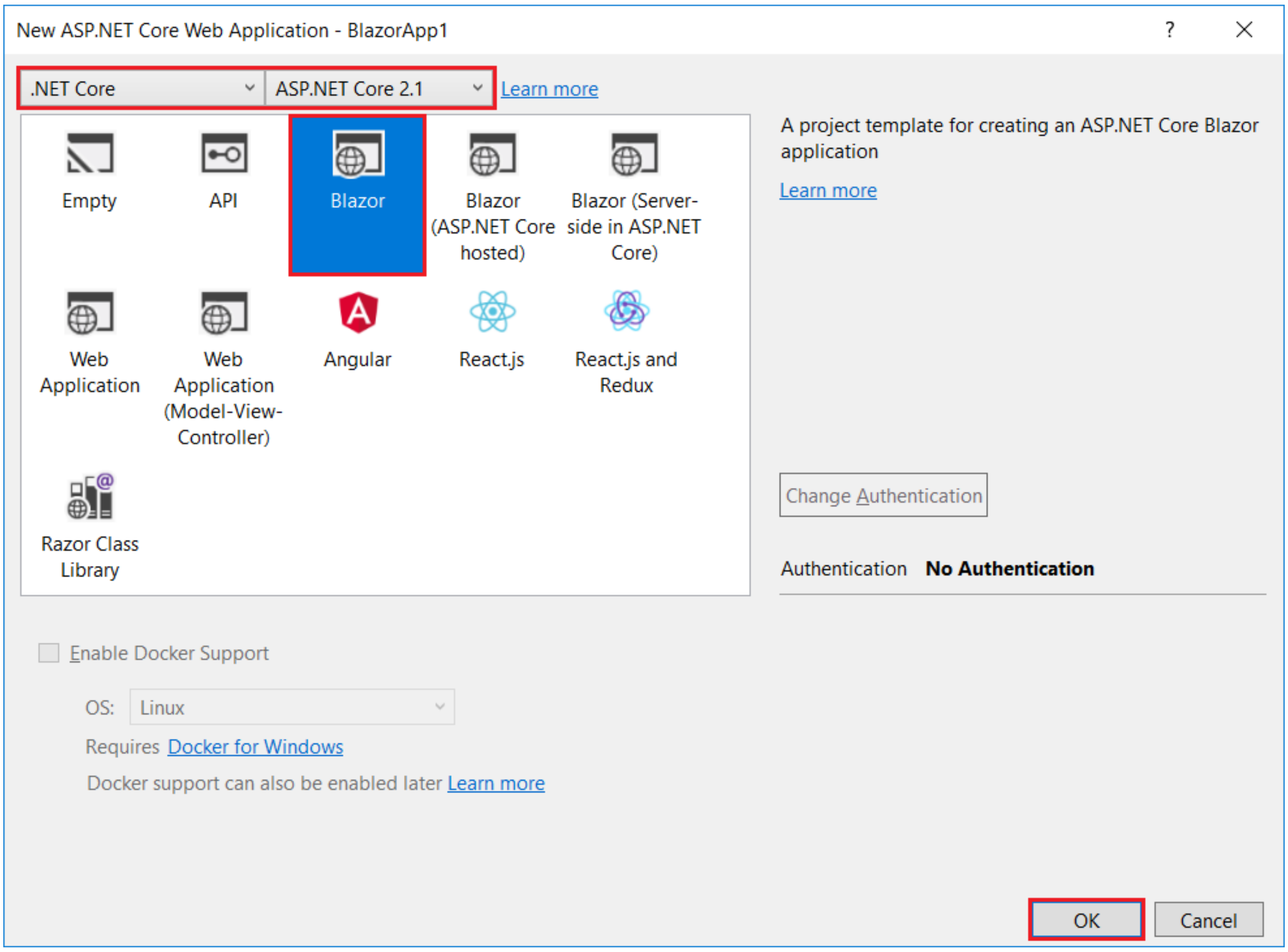


Type: Visual C#

Project templates for creating ASP.NET Core applications for Windows, Linux and macOS using .NET Core or .NET Framework. Create Razor Pages, MVC, Web API, and Single Page (SPA) Applications.

Name: BlazorApp1
Location: C:\Users\guard\Source\repos
Solution name: BlazorApp1
[✓] Create directory for solution
[ ] Create new Git repository

2. The **New ASP.NET Core Web Application** dialog appears. Make sure **.NET Core** is selected at the top. Select either **ASP.NET Core 2.0** or **ASP.NET Core 2.1**. Choose the **Blazor** template and select **OK**.


New ASP.NET Core Web Application - BlazorApp1

.NET Core | ASP.NET Core 2.1 | Learn more

A project template for creating an ASP.NET Core Blazor application

Learn more

Authentication **No Authentication**

Enable Docker Support

OS: Linux

Requires Docker for Windows

Docker support can also be enabled later Learn more

3. Once the project is created, press **Ctrl-F5** to run the app *without the debugger*. Running with the debugger (**F5**) isn't supported at this time.

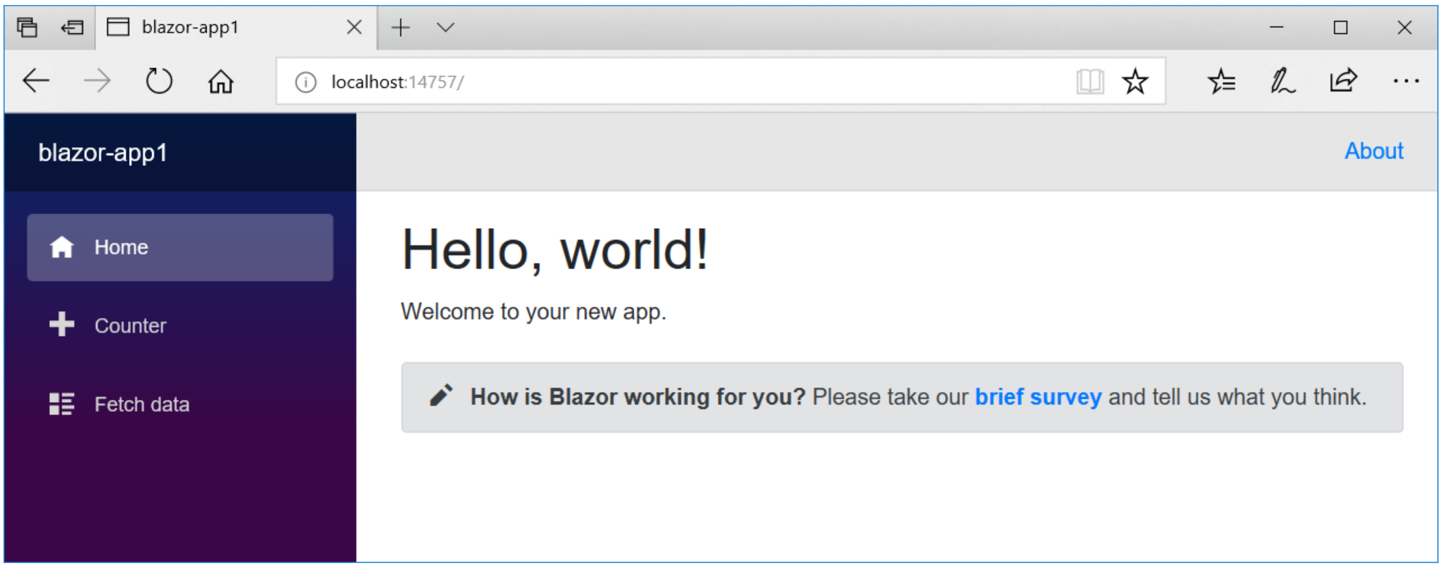
Note

If not using Visual Studio, create the Blazor app at a command prompt on Windows, macOS, or Linux:

```
dotnet new blazor -o BlazorApp1
cd BlazorApp1
dotnet run
```

Navigate to the app using the localhost address and port provided in the console window output after `dotnet run` is executed. Use **Ctrl-C** in the console window to shutdown the app.

The Blazor app runs in the browser:



Hello, world!

Welcome to your new app.

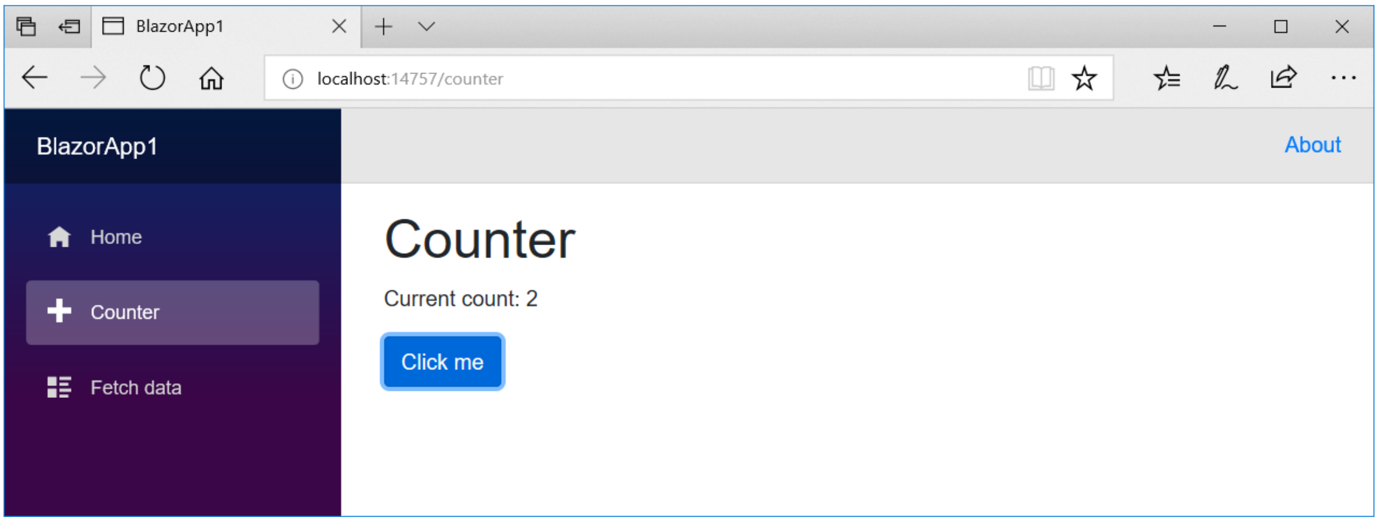
How is Blazor working for you? Please take our **brief survey** and tell us what you think.

Build components

1. Browse to each of the app's three pages: Home, Counter, and Fetch data.

   These three pages are implemented by the three Razor files in the *Pages* folder: *Index.cshtml*, *Counter.cshtml*, and *FetchData.cshtml*. Each of these files implements a Blazor component that's compiled and executed client-side in the browser.

2. Select the button on the Counter page.

   Counter

   Current count: 2

   Each time the button is selected, the counter is incremented without a page refresh. Normally, this kind of client-side behavior is handled in JavaScript; but here, it's implemented in C# and .NET by the `Counter` component.

3. Take a look at the implementation of the `Counter` component in the *Counter.cshtml* file:

```
@page "/counter"

<h1>Counter</h1>

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

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

@functions {
    int currentCount = 0;

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

The UI for the `Counter` component is defined using normal HTML. Dynamic rendering logic (for example, loops, conditionals, expressions) is added using an embedded C# syntax called Razor. The HTML markup and C# rendering logic are converted into a component class at build time. The name of the generated .NET class matches the name of the file.

Members of the component class are defined in a `@functions` block. In the `@functions` block, component state (properties, fields) and methods can be specified for event handling or for defining other component logic. These members can then be used as part of the component's rendering logic and for handling events.

When the button is selected, the `Counter` component's registered `onclick` handler is called (the `IncrementCount` method) and the `Counter` component regenerates its render tree. Blazor compares the new render tree against the previous one and applies any modifications to the browser Document Object Model (DOM). The displayed count is updated.

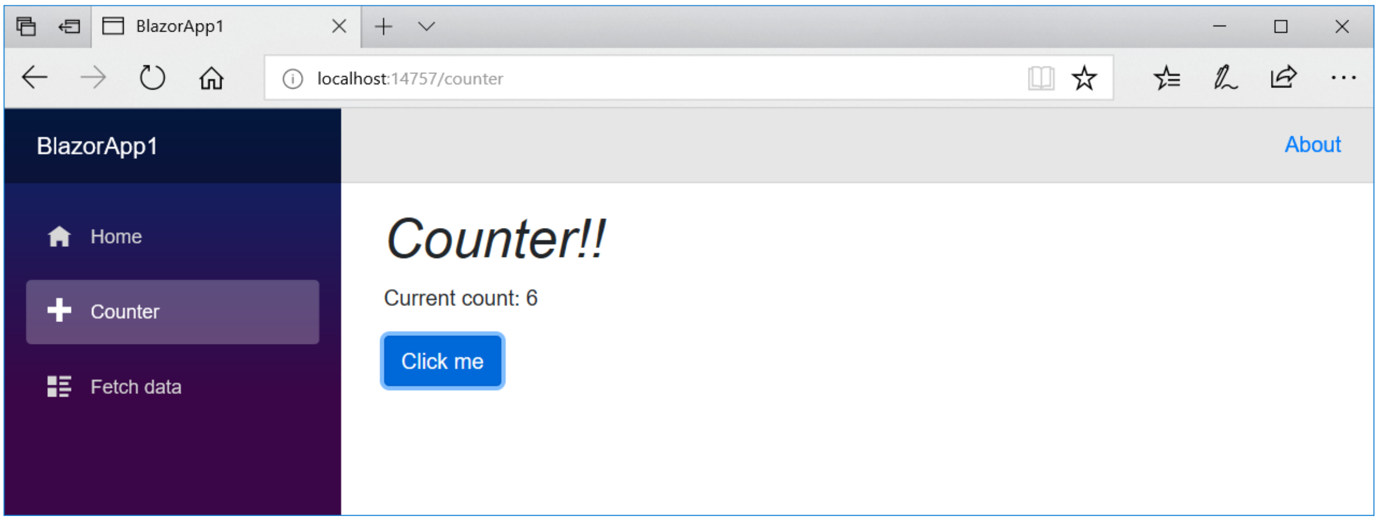
4. Update the markup for the `Counter` component to make the top-level header more *exciting*.

```
@page "/counter"
<h1><em>Counter!!</em></h1>
```

5. Also modify the C# logic of the `Counter` component to make the count increment by two instead of one.

```
@functions {
    int currentCount = 0;

    void IncrementCount()
    {
        currentCount += 2;
    }
}
```

6. Refresh the counter page in the browser to see the changes.



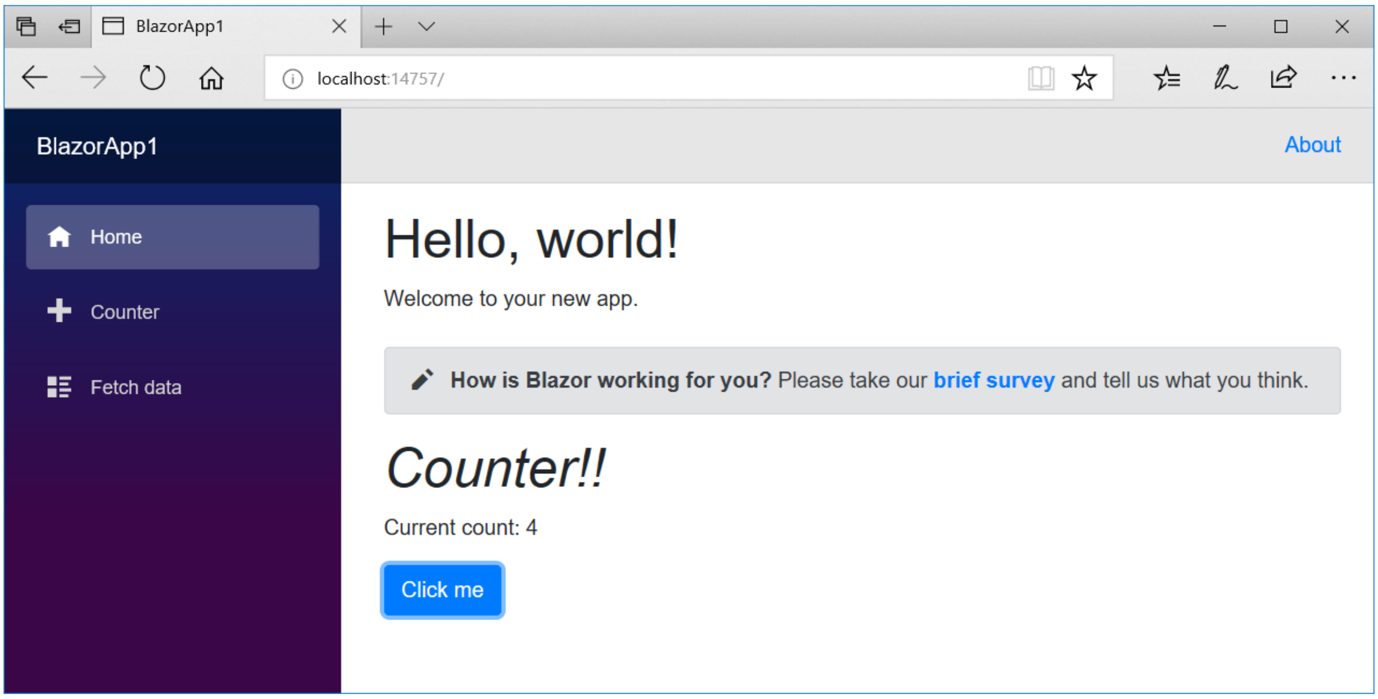
Use components

After a component is defined, the component can be used to implement other components. The markup for using a component looks like an HTML tag where the name of the tag is the component type.

1. Add a `Counter` component to the Home page of the app (*Index.cshtml*).

```
@page "/"

<h1>Hello, world!</h1>

Welcome to your new app.

<SurveyPrompt Title="How is Blazor working for you?" />

<Counter />
```

2. Refresh the home page in the browser. Note the separate instance of the `Counter` component on the Home page.



Component parameters

Components can also have parameters, which are defined using private properties on the component class decorated with

`[Parameter]`. Use attributes to specify arguments for a component in markup.

1. Update the `Counter` component to have an `IncrementAmount` parameter that defaults to 1.

   ```
   @functions {
       int currentCount = 0;

       [Parameter]
       private int IncrementAmount { get; set; } = 1;

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

   Note

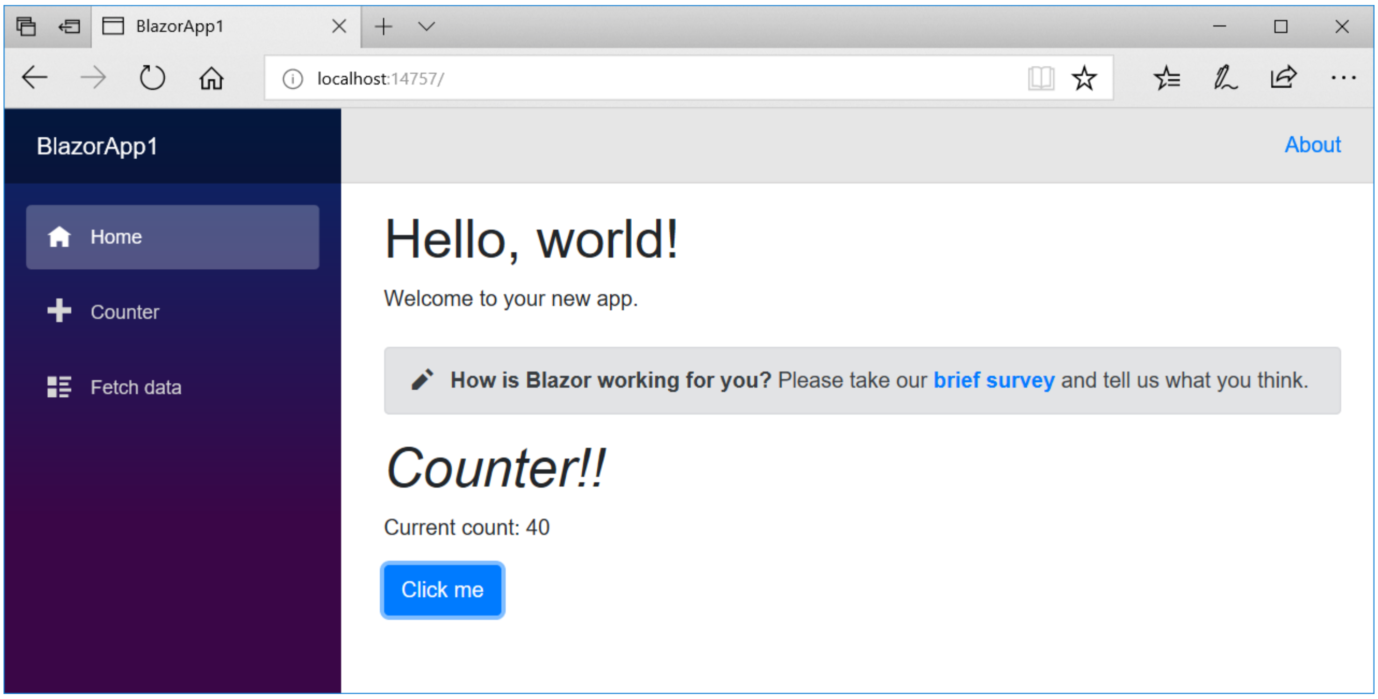
   From Visual Studio, you can quickly add a component parameter by using the `para` snippet. Type `para` and then press the `Tab` key twice.

2. On the Home page (*Index.cshtml*), change the increment amount for the `Counter` to 10 by setting an attribute that matches the name of the component's property for `IncrementCount`.

   ```
   <Counter IncrementAmount="10" />
   ```

3. Reload the page.

   The counter on the Home page now increments by 10, while the counter on the Counter page still increments by 1.



Hello, world!

Welcome to your new app.

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

Counter!!

Current count: 40

Route to components

The `@page` directive at the top of the *Counter.cshtml* file specifies that this component is a page to which requests can be routed. Specifically, the `Counter` component handles requests sent to `/Counter`. Without the `@page` directive, the component wouldn't handle routed requests, but the component could still be used by other components.

Dependency injection

Services registered with the app's service provider are available to components via dependency injection (DI). Services can be injected into a component using the `@inject` directive.

Take a look at the implementation of the `FetchData` component in *FetchData.cshtml*. The `@inject` directive is used to inject an HttpClient instance into the component.

```
@page "/fetchdata"
@inject HttpClient Http
```

The `FetchData` component uses the injected `HttpClient` to retrieve JSON data from the server when the component is initialized. Under the covers, the `HttpClient` provided by the Blazor runtime is implemented using JavaScript interop to call the underlying browser's Fetch API to send the request (from C#, it's possible to call any JavaScript library or browser API). The retrieved data is deserialized into the `forecasts` C# variable as an array of `WeatherForecast` objects.

```
@functions {
    WeatherForecast[] forecasts;

    protected override async Task OnInitAsync()
    {
        forecasts = await Http.GetJsonAsync<WeatherForecast[]>("/sample-data/weather.json");
    }

    class WeatherForecast
    {
        public DateTime Date { get; set; }
        public int TemperatureC { get; set; }
        public int TemperatureF { get; set; }
        public string Summary { get; set; }
    }
}
```

A `@foreach` loop is used to render each forecast instance as a row in the weather table.

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

Build a todo list

Add a new page to the app that implements a simple todo list.

1. Add an empty text file to the *Pages* folder named *Todo.cshtml*.

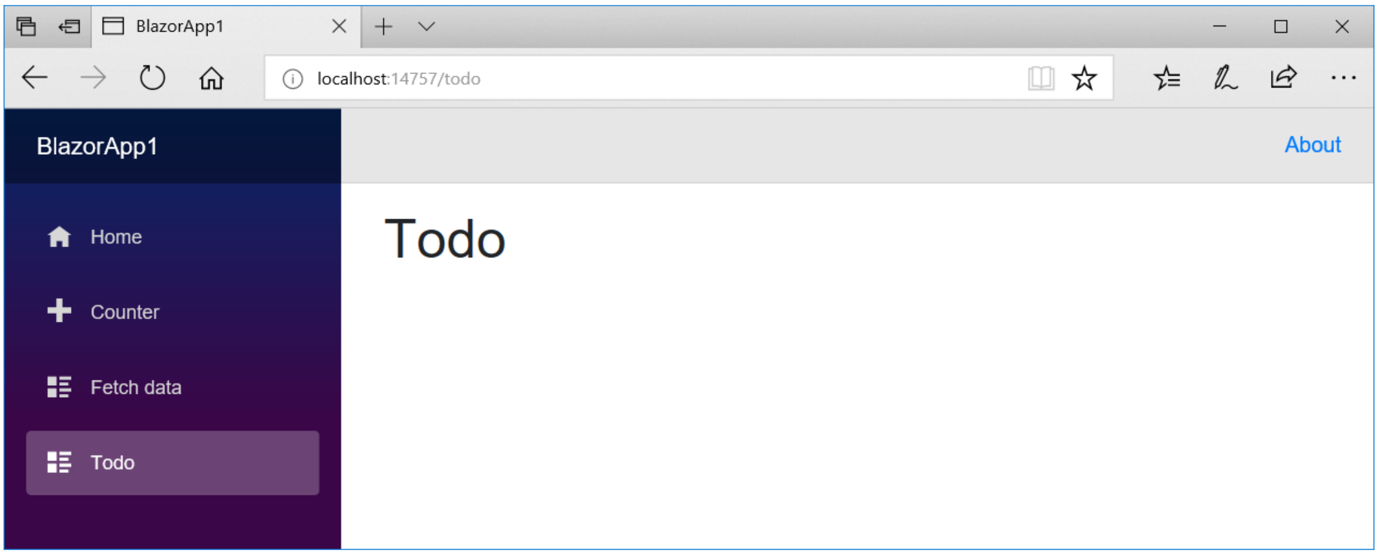
2. Provide the initial markup for the page.

```
@page "/todo"

<h1>Todo</h1>
```

3. Add the Todo page to the navigation bar by updating *Shared/NavMenu.cshtml*. Add a `NavLink` for the Todo page by adding the following list item markup below the existing list items.

```
<li class="nav-item px-3">
    <NavLink class="nav-link" href="todo">
        <span class="oi oi-list-rich" aria-hidden="true"></span> Todo
    </NavLink>
</li>
```

4. Refresh the app in the browser. See the new Todo page.



Todo

5. Add a *TodoItem.cs* file to the root of the project to hold a class to represent the todo items.

6. Use the following C# code for the `ToDoItem` class.

```
public class TodoItem
{
    public string Title { get; set; }
    public bool IsDone { get; set; }
}
```

7. Go back to the `Todo` component in *Todo.cshtml* and add a field for the todos in a `@functions` block. The `Todo` component uses this field to maintain the state of the todo list.

```
@functions {
    IList<TodoItem> todos = new List<TodoItem>();
}
```

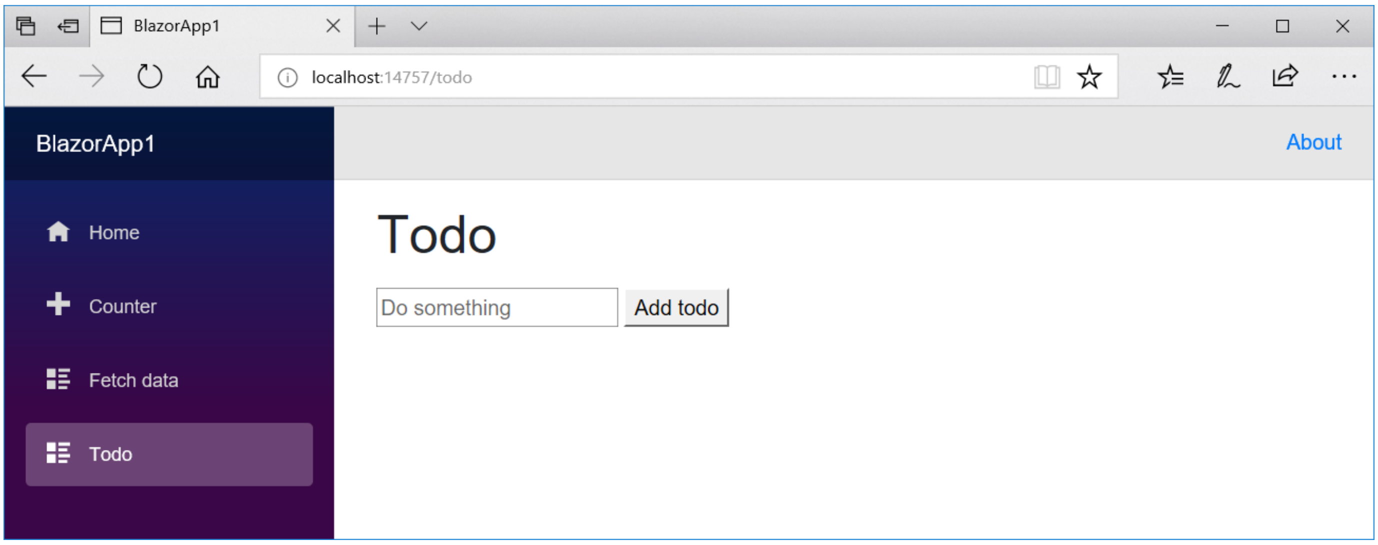
8. Add unordered list markup and a `foreach` loop to render each todo item as a list item.

```
@page "/todo"

<h1>Todo</h1>

<ul>
    @foreach (var todo in todos)
    {
        <li>@todo.Title</li>
    }
</ul>
```

9. The app requires UI elements for adding todos to the list. Add a text input and a button below the list.

```
@page "/todo"

<h1>Todo</h1>

<ul>
    @foreach (var todo in todos)
    {
        <li>@todo.Title</li>
    }
</ul>

<input placeholder="Something todo" />
<button>Add todo</button>

@functions {
    IList<TodoItem> todos = new List<TodoItem>();
}
```

10. Refresh the browser.



Nothing happens when the **Add todo** button is selected because no event handler is wired up to the button.

11. Add an `AddTodo` method to the `Todo` component and register it for button clicks using the `onclick` attribute.

```
<input placeholder="Something todo" />
<button onclick="@AddTodo">Add todo</button>

@functions {
    IList<TodoItem> todos = new List<TodoItem>();

    void AddTodo()
    {
        // Todo: Add the todo
    }
}
```

The `AddTodo` C# method is called every time the button is selected.

12. To get the title of the new todo item, add a `newTodo` string field and bind it to the value of the text input using the `bind` attribute.

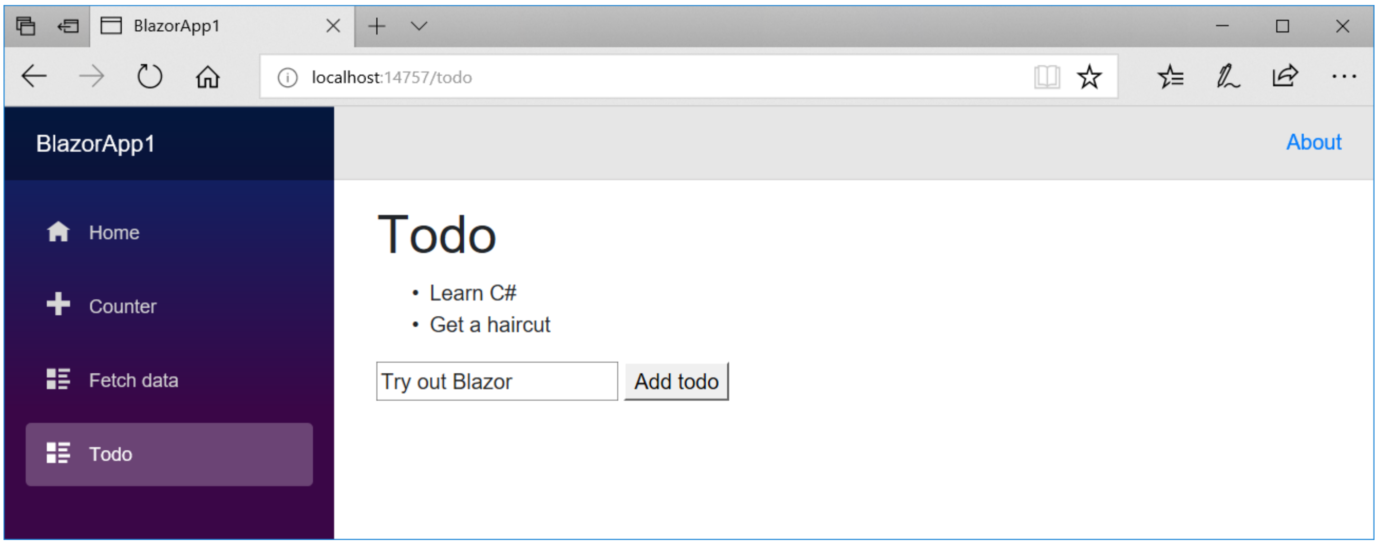
```
IList<TodoItem> todos = new List<TodoItem>();
string newTodo;
```

```
<input placeholder="Something todo" bind="@newTodo" />
```

13. Update the `AddTodo` method to add the `TodoItem` with the specified title to the list. Don't forget to clear the value of the text input by setting `newTodo` to an empty string.

```
@page "/todo"

<h1>Todo</h1>

<ul>
    @foreach (var todo in todos)
    {
        <li>@todo.Title</li>
    }
</ul>

<input placeholder="Something todo" bind="@newTodo" />
<button onclick="@AddTodo">Add todo</button>

@functions {
    IList<TodoItem> todos = new List<TodoItem>();
    string newTodo;

    void AddTodo()
    {
        if (!string.IsNullOrWhiteSpace(newTodo))
        {
            todos.Add(new TodoItem { Title = newTodo });
            newTodo = string.Empty;
        }
    }
}
```

14. Refresh the browser. Add some todos to the todo list.



15. Lastly, what's a todo list without check boxes? The title text for each todo item can be made editable as well. Add a check box input and text input for each todo item and bind their values to the `Title` and `IsDone` properties, respectively.

```
<ul>
    @foreach (var todo in todos)
    {
        <li>
            <input type="checkbox" bind="@todo.IsDone" />
            <input bind="@todo.Title" />
        </li>
    }
</ul>
```

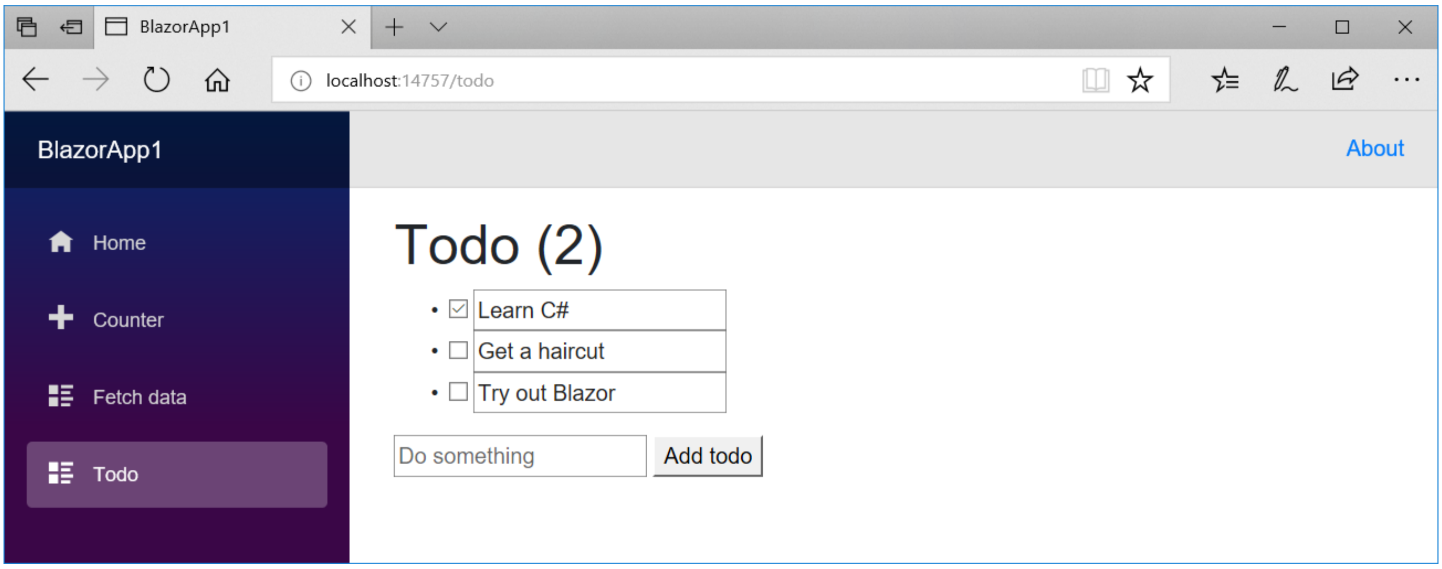
16. To verify that these values are bound, update the `h1` header to show a count of the number of todo items that are not yet done (`IsDone` is `false`).

```
<h1>Todo (@todos.Where(todo => !todo.IsDone).Count())</h1>
```

17. The completed `Todo` component should look like this:

```
@page "/todo"

<h1>Todo (@todos.Where(todo => !todo.IsDone).Count())</h1>

<ul>
    @foreach (var todo in todos)
    {
        <li>
            <input type="checkbox" bind="@todo.IsDone" />
            <input bind="@todo.Title" />
        </li>
    }
</ul>

<input placeholder="Something todo" bind="@newTodo" />
<button onclick="@AddTodo">Add todo</button>

@functions {
    IList<TodoItem> todos = new List<TodoItem>();
    string newTodo;

    void AddTodo()
    {
        if (!string.IsNullOrWhiteSpace(newTodo))
        {
            todos.Add(new TodoItem { Title = newTodo });
            newTodo = string.Empty;
        }
    }
}
```

Refresh the app in the browser. Try adding some todo items.

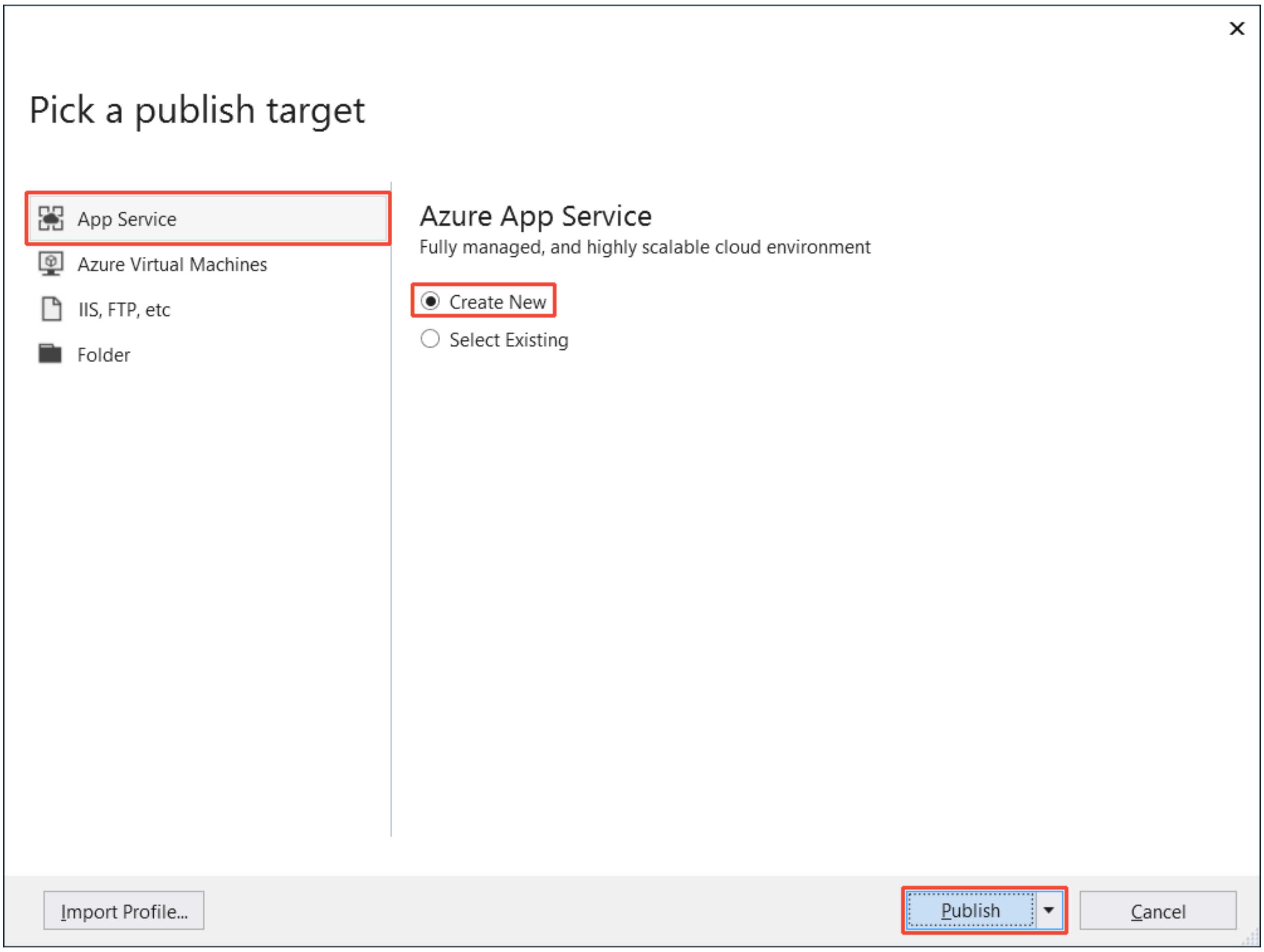


Publish and deploy

When using Visual Studio, perform the following steps to publish the Todo Blazor app to Azure:

1. Right-click on the project in **Solution Explorer** and select **Publish**.

2. In the **Pick a publish target** dialog, select **App Service** and **Create New**. Select **Publish**.

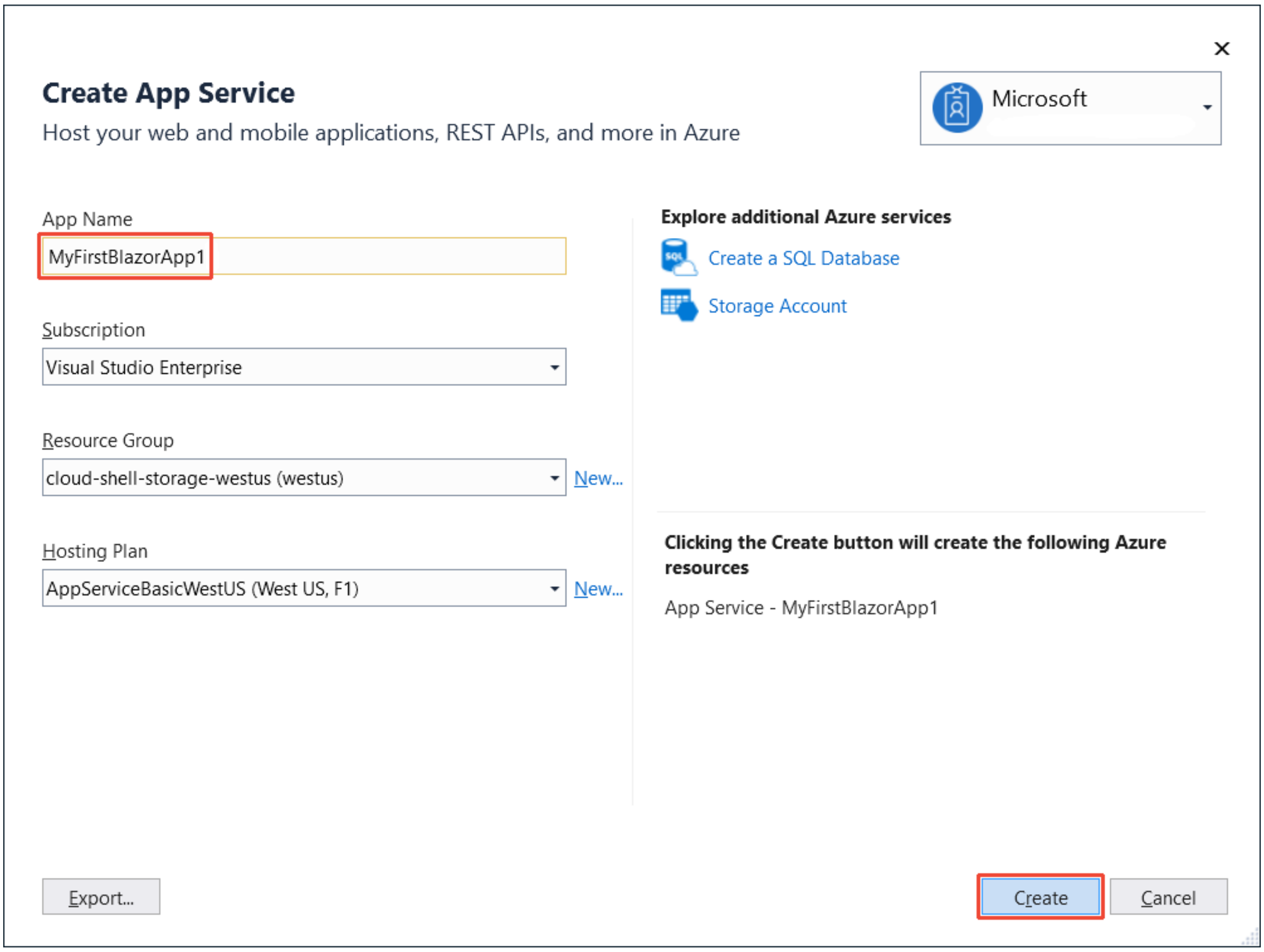

Pick a publish target

Azure App Service

Fully managed, and highly scalable cloud environment

[✓ Create New] Select Existing

3. In the **Create App Service** dialog, choose a name for the app and select the subscription, resource group, and hosting plan. Select **Create** to create the app service and publish the app.


Create App Service

Host your web and mobile applications, REST APIs, and more in Azure

Microsoft

App Name: MyFirstBlazorApp1

Subscription: Visual Studio Enterprise

Resource Group: cloud-shell-storage-westus (westus) New...

Hosting Plan: AppServiceBasicWestUS (West US, F1) New...

Explore additional Azure services

Create a SQL Database

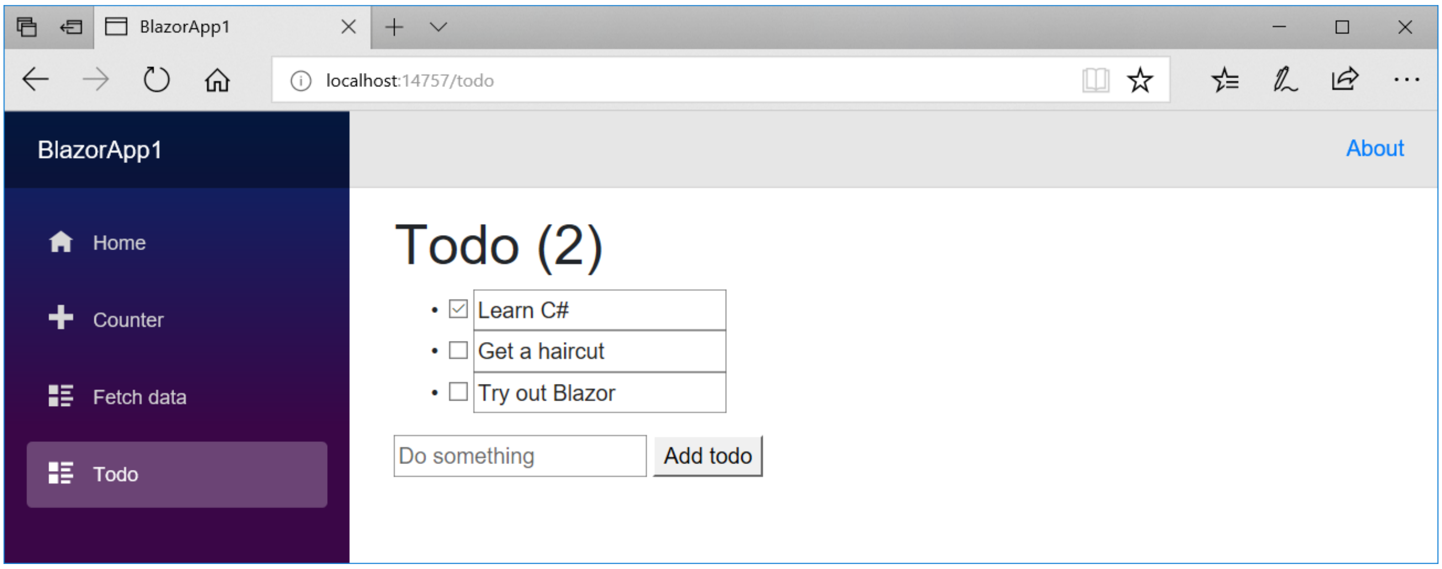
Storage Account

Clicking the Create button will create the following Azure resources

App Service - MyFirstBlazorApp1

Wait a minute or so for the app to be deployed.

The app should now be running in Azure. Mark the todo item to build your first Blazor app as *done*. Nice job!



Todo (2)

- ☑ Learn C#
- ☐ Get a haircut
- ☐ Try out Blazor

Note

If not using Visual Studio, publish the Blazor app at a command prompt on Windows, macOS, or Linux:

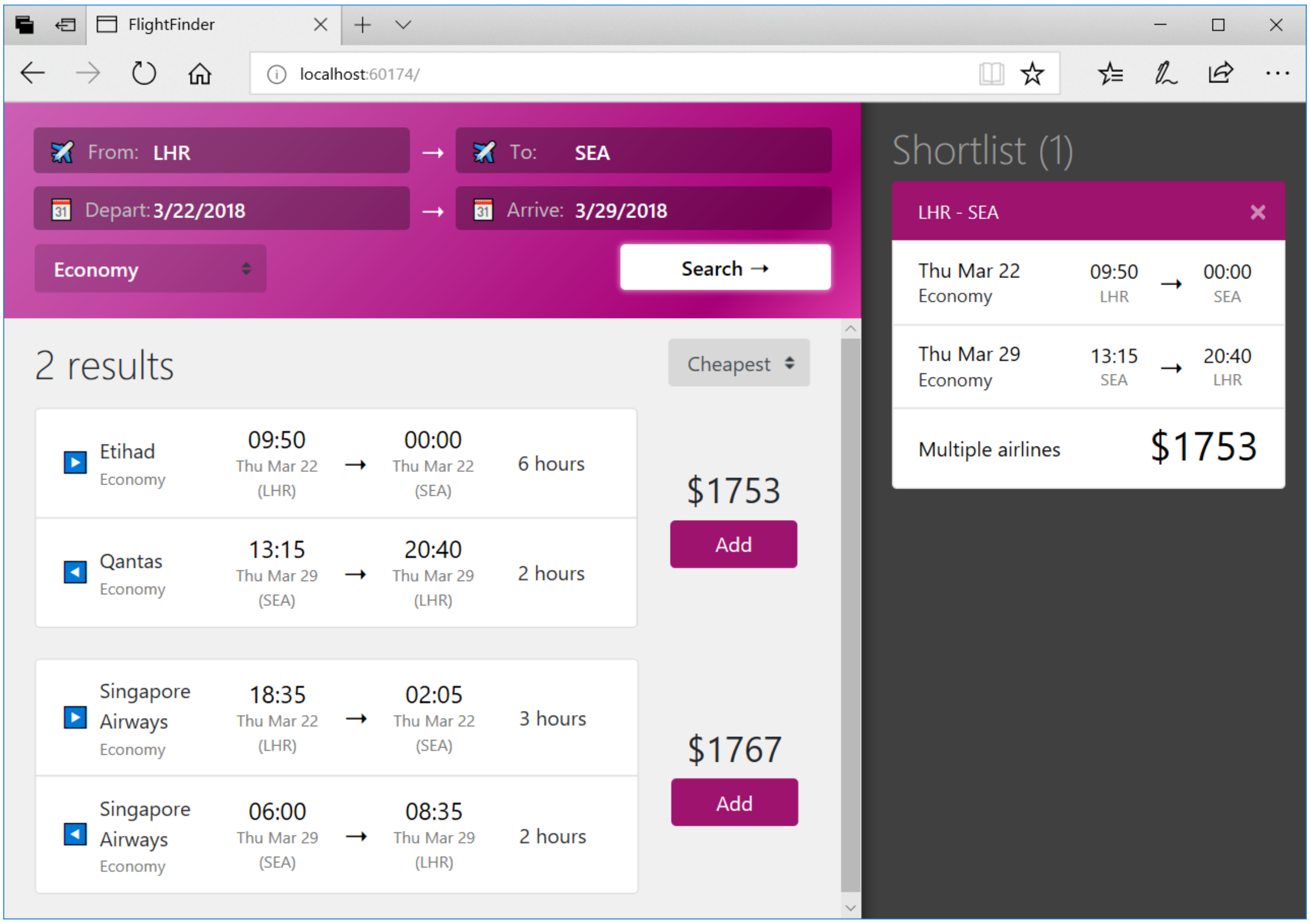
```
dotnet publish -c Release
```

The deployment is created in the */bin/Release/<target-framework>/publish* folder. Move the contents of the *publish* folder to the server or hosting service.

For more information, see the Host and deploy topic.

Additional resources

For a more involved Blazor sample app, check out the Flight Finder sample on GitHub.


FlightFinder
localhost:60174/
From: LHR
To: SEA
Depart: 3/22/2018
Arrive: 3/29/2018
Economy
Search →
2 results
Cheapest
Etihad Economy 09:50 Thu Mar 22 (LHR) → 00:00 Thu Mar 22 (SEA) 6 hours
Qantas Economy 13:15 Thu Mar 29 (SEA) → 20:40 Thu Mar 29 (LHR) 2 hours
$1753
Add
Singapore Airways Economy 18:35 Thu Mar 22 (LHR) → 02:05 Thu Mar 22 (SEA) 3 hours
Singapore Airways Economy 06:00 Thu Mar 29 (SEA) → 08:35 Thu Mar 29 (LHR) 2 hours
$1767
Add
Shortlist (1)
LHR - SEA
Thu Mar 22 Economy 09:50 LHR → 00:00 SEA
Thu Mar 29 Economy 13:15 SEA → 20:40 LHR
Multiple airlines $1753

Introduction to Blazor

By Steve Sanderson, Daniel Roth, and Luke Latham

Note

Blazor is an unsupported experimental web framework that shouldn't be used for production workloads at this time.

Blazor is an experimental .NET web framework using C#/Razor and HTML that runs in the browser with WebAssembly. Blazor provides all of the benefits of a client-side web UI framework using .NET on the client and optionally on the server.

Why use .NET in the browser?

Web development has improved in many ways over the years, but building modern web apps still poses challenges. Using .NET in the browser offers many advantages that can help make web development easier and more productive:

- **Stability and consistency**: .NET provides standardized programming frameworks across platforms that are stable, feature-rich, and easy to use.
- **Modern innovative languages**: .NET languages are constantly improving with innovative new language features.
- **Industry-leading tools**: The Visual Studio product family provides a fantastic .NET development experience across platforms on Windows, Linux, and macOS.
- **Speed and scalability**: .NET has a strong history of performance, reliability, and security for app development. Using .NET as a full-stack solution makes it easier to build fast, reliable, and secure apps.
- **Full-stack development that leverages existing skills**: C#/Razor developers use their existing C#/Razor skills to write client-side code and share server and client-side logic among apps.
- **Wide browser support**: Blazor runs on .NET using open web standards in the browser with no plugins and no code transpilation. It works in all modern web browsers, including mobile browsers.

How Blazor runs .NET in the browser

Running .NET code inside web browsers is made possible by a relatively new technology, WebAssembly (abbreviated *wasm*). WebAssembly is an open web standard and is supported in web browsers without plugins. WebAssembly is a compact bytecode format optimized for fast download and maximum execution speed.

WebAssembly code can access the full functionality of the browser via JavaScript interop. At the same time, WebAssembly code runs in the same trusted sandbox as JavaScript to prevent malicious actions on the client machine.

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

1. C# code files and Razor files are compiled into .NET assemblies.
2. The assemblies and the .NET runtime are downloaded to the browser.
3. Blazor uses JavaScript to bootstrap the .NET runtime and configures the runtime to load required assembly references. Document object model (DOM) manipulation and browser API calls are handled by the Blazor runtime via JavaScript interoperability.

To support older browsers that don't support WebAssembly, Blazor falls back to using an asm.js-based .NET runtime.

Blazor components

Blazor apps are built with *components*. A component is a piece of UI, such as a page, dialog, or data entry form. Components can be nested, reused, and shared between projects.

In Blazor, a component is a .NET class. The class can either be written directly, as a C# class (**.cs*), or more commonly in the form of a Razor markup page (**.cshtml*).

Razor is a syntax for combining HTML markup with C# code. Razor is designed for developer productivity, allowing the developer to switch between markup and C# in the same file with IntelliSense support. The following markup is an example of a basic custom dialog component in a Razor file (*DialogComponent.cshtml*):

```
<div>
    <h2>@Title</h2>
    @BodyContent
    <button onclick=@OnOK>OK</button>
</div>

@functions {
    public string Title { get; set; }
    public RenderFragment BodyContent { get; set; }
    public Action OnOK { get; set; }
}
```

When this component is used elsewhere in the app, IntelliSense speeds development with syntax and parameter completion.

Components can be:

- Nested.
- Created with Razor (*.*cshtml*) or C# (*.*cs*) code.
- Shared via class libraries.
- Unit tested without requiring a browser DOM.

Infrastructure

Blazor offers the core facilities that most apps require, including:

- Layouts
- Routing
- Dependency injection

All of these features are optional. When one of these features isn't used in an app, the implementation is stripped out of the app when published by the Intermediate Language (IL) Linker.

Code sharing and .NET Standard

Blazor apps can reference and use existing .NET Standard libraries. .NET Standard is a formal specification of .NET APIs that are common across .NET implementations. Blazor supports .NET Standard 2.0 or higher. APIs that aren't applicable inside a web browser (for example, accessing the file system, opening a socket, threading, and other features) throw PlatformNotSupportedException. .NET Standard class libraries can be shared across server code and in browser-based apps.

JavaScript interop

For apps that require third-party JavaScript libraries and browser APIs, WebAssembly is designed to interoperate with JavaScript. Blazor is 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. For more information, see JavaScript interop.

Optimization

For client-side apps, payload size is critical. Blazor optimizes payload size to reduce download times. For example, unused parts of .NET assemblies are removed during the build process, HTTP responses are compressed, and the .NET runtime and assemblies are cached in the browser.

Deployment

Use Blazor to build a pure standalone client-side app or a full-stack ASP.NET Core app that contains both server and client apps:

- In a **standalone client-side app**, the Blazor app is compiled into a *dist* folder that only contains static files. The files can be hosted on Azure App Service, GitHub Pages, IIS (configured as a static file server), Node.js servers, and many other servers and services. .NET isn't required on the server in production.
- In a **full-stack ASP.NET Core app**, code can be shared between server and client apps. The resulting ASP.NET Core server app, which serves the Blazor client-side UI and other server-side API endpoints, can be built and deployed to any cloud or on-premise host supported by ASP.NET Core.

Blazor components

By Luke Latham and Daniel Roth

Note

Blazor is an unsupported experimental web framework that shouldn't be used for production workloads at this time.

View or download sample code (how to download). See the Get started topic for prerequisites.

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

Component classes

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

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

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

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

```
<h1 style="font-style:@_headingFontStyle">@_headingText</h1>

@functions {
    private string _headingFontStyle = "italic";
    private string _headingText = "Put on your new Blazor!";
}
```

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

Using components

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

The following markup renders a `HeadingComponent` (*HeadingComponent.cshtml*) instance:

```
<HeadingComponent />
```

Component parameters

Components can have *component parameters*, which are defined using *non-public* properties on the component class decorated with `[Parameter]`. Use attributes to specify arguments for a component in markup.

In the following example, the `ParentComponent` sets the value of the `Title` property of the `ChildComponent`:

ParentComponent.cshtml:

```
@page "/ParentComponent"

<h1>Parent-child example</h1>

<ChildComponent Title="Panel Title from Parent">
    Child content of the child component is supplied by the parent component.
</ChildComponent>
```

ChildComponent.cshtml:

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

@functions {
    [Parameter]
    private string Title { get; set; }

    [Parameter]
    private RenderFragment ChildContent { get; set; }
}
```

Child content

Components can set the content in another component. The assigning component provides the content between the tags that specify the receiving component. For example, a `ParentComponent` can provide content that is to be rendered by a `ChildComponent` by placing the content inside **<ChildComponent>** tags.

ParentComponent.cshtml:

```
@page "/ParentComponent"

<h1>Parent-child example</h1>

<ChildComponent Title="Panel Title from Parent">
    Child content of the child component is supplied by the parent component.
</ChildComponent>
```

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

ChildComponent.cshtml:

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

@functions {
    [Parameter]
    private string Title { get; set; }

    [Parameter]
    private RenderFragment ChildContent { get; set; }
}
```

Note

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

Data binding

Data binding to both components and DOM elements is accomplished with the `bind` attribute. The following example binds the `ItalicsCheck` property to the check box's checked state:

```
<input type="checkbox" class="form-check-input" id="italicsCheck" bind="@_italicsCheck" />
```

When the check box is selected and cleared, the property's value is updated to `true` and `false`, respectively.

The check box is updated in the UI only when the component is rendered, not in response to changing the property's value. Since components render themselves after event handler code executes, property updates are usually reflected in the UI immediately.

Using `bind` with a `CurrentValue` property (`<input bind="@CurrentValue" />`) is essentially equivalent to the following:

```
<input value="@CurrentValue"
    onchange="@((UIValueEventArgs __e) => CurrentValue = __e.Value)" />
```

When the component is rendered, the `value` of the input element comes from the `CurrentValue` property. When the user types in the text box, the `onchange` is fired and the `CurrentValue` property is set to the changed value. In reality, the code generation is a little more complex because `bind` deals with a few cases where type conversions are performed. In principle, `bind` associates the current value of an expression with a `value` attribute and handles changes using the registered handler.

Format strings

Data binding works with DateTime format strings (but not other format expressions at this time, such as currency or number formats):

```
<input bind="@StartDate" format-value="yyyy-MM-dd" />

@functions {
    [Parameter]
    private DateTime StartDate { get; set; } = new DateTime(2020, 1, 1);
}
```

The `format-value` attribute specifies the date format to apply to the `value` of the `input` element. The format is also used to parse the value when an `onchange` event occurs.

Component parameters

Binding also recognizes component parameters, where `bind-{property}` can bind a property value across components.

The following component uses `ChildComponent` and binds the `ParentYear` parameter from the parent to the `Year` parameter on the child component:

Parent component:

```
@page "/ParentComponent"

<h1>Parent Component</h1>

<p>ParentYear: @ParentYear</p>

<ChildComponent bind-Year="@ParentYear" />

<button class="btn btn-primary" onclick="@ChangeTheYear">Change Year to 1986</button>

@functions {
    [Parameter]
    private int ParentYear { get; set; } = 1978;

    void ChangeTheYear()
    {
        ParentYear = 1986;
    }
}
```

Child component:

```
<h2>Child Component</h2>

<p>Year: @Year</p>

@functions {
    [Parameter]
    private int Year { get; set; }

    [Parameter]
    private Action<int> YearChanged { get; set; }
}
```

The `Year` parameter is bindable because it has a companion `YearChanged` event that matches the type of the `Year` parameter.

Loading the `ParentComponent` produces the following markup:

```
<h1>Parent Component</h1>

<p>ParentYear: 1978</p>

<h2>Child Component</h2>

<p>Year: 1978</p>
```

If the value of the `ParentYear` property is changed by selecting the button in the `ParentComponent`, the `Year` property of the `ChildComponent` is updated. The new value of `Year` is rendered in the UI when the `ParentComponent` is rerendered:

```
<h1>Parent Component</h1>

<p>ParentYear: 1986</p>

<h2>Child Component</h2>

<p>Year: 1986</p>
```

Event handling

Blazor provides event handling features. For an HTML element attribute named `on<event>` (for example, `onclick`, `onsubmit`) with a delegate-typed value, Blazor treats the attribute's value as an event handler. The attribute's name always starts with `on`.

The following code calls the `UpdateHeading` method when the button is selected in the UI:

```
<button class="btn btn-primary" onclick="@UpdateHeading">
    Update heading
</button>

@functions {
    void UpdateHeading(UIMouseEventArgs e)
    {
        ...
    }
}
```

The following code calls the `CheckboxChanged` method when the check box is changed in the UI:

```
<input type="checkbox" class="form-check-input" onchange="@CheckboxChanged" />

@functions {
    void CheckboxChanged()
    {
        ...
    }
}
```

For some events, event-specific event argument types are permitted. If access to one of these event types isn't necessary, it isn't required in the method call.

The list of supported event arguments is:

- UIEventArgs
- UIChangeEventArgs
- UIKeyboardEventArgs
- UIMouseEventArgs

Lambda expressions can also be used:

```
<button onclick="@(e => Console.WriteLine("Hello, world!"))">Say hello</button>
```

Capture references to components

Component references provide a way get a reference to a component instance so that you can issue commands to that instance, such as `Show` or `Reset`. To capture a component reference, add a `ref` attribute to the child component and then define a field with the same name and the same type as the child component.

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

@functions {
    MyLoginDialog loginDialog;

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

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

Important

The `loginDialog` variable is only populated after the component is rendered and its output includes the `MyLoginDialog` element because until then there is nothing to reference. To manipulate components references after the component has finished rendering, use the `OnAfterRenderAsync` or `OnAfterRender` lifecycle methods.

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

Note

Do **not** use component references to mutate the state of child components. Instead, always use normal declarative parameters to pass data to child components. This causes child components to rerender at the correct times automatically.

Lifecycle methods

`OnInitAsync` and `OnInit` execute code after the component has been initialized. To perform an asynchronous operation, use `OnInitAsync` and use the `await` keyword:

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

For a synchronous operation, use `OnInit`:

```
protected override void OnInit()
{
    ...
}
```

`OnParametersSetAsync` and `OnParametersSet` are called when a component has received parameters from its parent and the values are assigned to properties. These methods are executed after `OnInit` during component initialization.

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

```
protected override void OnParametersSet()
{
    ...
}
```

`OnAfterRenderAsync` and `OnAfterRender` are called each time after a component has finished rendering. Element and component references are populated at this point. Use this stage to perform additional initialization steps using the rendered content, such as activating third-party JavaScript libraries that operate on the rendered DOM elements.

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

```
protected override void OnAfterRender()
{
    ...
}
```

`SetParameters` can be overridden to execute code before parameters are set:

```
public override void SetParameters(ParameterCollection parameters)
{
    ...

    base.SetParameters(parameters);
}
```

If `base.SetParameters` isn't invoked, the custom code can interpret the incoming parameters value in any way required. For example, the incoming parameters aren't required to be assigned to the properties on the class.

`ShouldRender` can be overridden to suppress refreshing of the UI. If the implementation returns `true`, the UI is refreshed. Even if `ShouldRender` is overridden, the component is always initially rendered.

```
protected override bool ShouldRender()
{
    var renderUI = true;

    return renderUI;
}
```

Component disposal with IDisposable

If a component implements IDisposable, the Dispose method is called when the component is removed from the UI. The following component uses `@implements IDisposable` and the `Dispose` method:

```
@using System
@implements IDisposable

...

@functions {
    public void Dispose()
    {
        ...
    }
}
```

Routing

Routing in Blazor is achieved by providing a route template to each accessible component in the app.

When a **.cshtml* file with an `@page` directive is compiled, the generated class is given a RouteAttribute specifying the route template. At runtime, the router looks for component classes with a `RouteAttribute` and renders whichever component has a route template that matches the requested URL.

Multiple route templates can be applied to a component. The following component responds to requests for `/BlazorRoute` and `/DifferentBlazorRoute`:

```
@page "/BlazorRoute"
@page "/DifferentBlazorRoute"

<h1>Blazor routing</h1>
```

Route parameters

Blazor components can receive route parameters from the route template provided in the `@page` directive. The Blazor client-side router uses route parameters to populate the corresponding component parameters.

RouteParameter.cshtml:

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

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

@functions {
    [Parameter]
    private string Text { get; set; } = "fantastic";
}
```

Optional parameters aren't supported, so two `@page` directives are applied in the example above. The first permits navigation to the component without a parameter. The second `@page` directive takes the `{text}` route parameter and assigns the value to the `Text` property.

Base class inheritance for a "code-behind" experience

Blazor component files (*.cshtml*) mix HTML markup and C# processing code in the same file. The `@inherits` directive can be used to provide Blazor with a "code-behind" experience that separates component markup from processing code.

The sample app shows how a component can inherit a base class, `BlazorRocksBase`, to provide the component's properties and methods.

BlazorRocks.cshtml:

```
@page "/BlazorRocks"
@*
    The inherit directive provides the properties and methods
    of the BlazorRocksBase class to this component.
*@
@inherits BlazorRocksBase

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

BlazorRocksBase.cs:

```
using Microsoft.AspNetCore.Blazor.Components;

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

The base class should derive from BlazorComponent.

Razor support

Razor directives

Razor directives active with Blazor apps are shown in the following table.

DIRECTIVE	DESCRIPTION
@functions	Adds a C# code block to a component.
`@implements`	Implements an interface for the generated component class.

DIRECTIVE	DESCRIPTION
@inherits	Provides full control of the class that the component inherits.
@inject	Enables service injection from the service container. For more information, see Dependency injection into views.
`@layout`	Specifies a layout component. Layout components are used to avoid code duplication and inconsistency.
@page	Specifies that the component should handle requests directly. The `@page` directive can be specified with a route and optional parameters. Unlike Razor Pages, the `@page` directive doesn't need to be the first directive at the top of the file. For more information, see Routing.
@using	Adds the C# `using` directive to the generated component class.
@addTagHelper	Use `@addTagHelper` to use a component in a different assembly than the app's assembly.

Conditional attributes

Blazor conditionally renders attributes based on the .NET value. If the value is `false` or `null`, Blazor won't render the attribute. If the value is `true`, the attribute is rendered minimized.

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

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

@functions {
    [Parameter]
    private bool IsCompleted { get; set; }
}
```

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

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

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

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

Additional information on Razor

For more information on Razor, see the Razor syntax reference. Note that not all of the features of Razor are available in Blazor at this time.

Raw HTML

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

Warning

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

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

```
@((MarkupString)myMarkup)

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

Blazor layouts

By Rainer Stropek

▯ Note

Blazor is an unsupported experimental web framework that shouldn't be used for production workloads at this time.

Blazor apps typically contain more than one page. Layout elements, such as menus, copyright messages, and logos, must be present on all pages. Copying the code of these layout elements into all of the pages of an app isn't an efficient solution. Such duplication is hard to maintain and probably leads to inconsistent content over time. *Layouts* solve this problem.

What are layouts?

Technically, a layout is just another Blazor component. A layout is defined in a Razor template or in C# code and can contain data binding, dependency injection, and other ordinary features of components. Two additional aspects turn a *component* into a *layout*:

- The layout component must inherit from BlazorLayoutComponent. `BlazorLayoutComponent` defines a `Body` property that contains the content to be rendered inside the layout.
- The layout component uses the `Body` property to specify where the body content should be rendered using the Razor syntax `@Body`. During rendering, `@Body` is replaced by the content of the layout.

The following code sample shows the Razor template of a layout component. Note the use of `BlazorLayoutComponent` and `@Body`:

```
@inherits BlazorLayoutComponent

<header>
    <h1>ERP Master 3000</h1>
</header>

<nav>
    <a href="master-data">Master Data Management</a>
    <a href="invoicing">Invoicing</a>
    <a href="accounting">Accounting</a>
</nav>

@Body

<footer>
    &copy; by @CopyrightMessage
</footer>

@functions {
    public string CopyrightMessage { get; set; }
    ...
}
```

Use a layout in a component

Use the Razor directive `@layout` to apply a layout to a component. The compiler converts this directive into a LayoutAttribute, which is applied to the component class.

The following code sample demonstrates the concept. The content of this component is inserted into the *MasterLayout* at the position of `@Body`:

```
@layout MasterLayout

@page "/master-data"

<h2>Master Data Management</h2>
...
```

Centralized layout selection

Every folder of a Blazor app can optionally contain a template file named *_ViewImports.cshtml*. The compiler includes the directives specified in the view imports file in all of the Razor templates in the same folder and recursively in all of its subfolders. Therefore, a *_ViewImports.cshtml* file containing `@layout MainLayout` ensures that all of the components in a folder use the *MainLayout* layout. There's no need to repeatedly add `@layout` to all of the **.cshtml* files.

Note that the default template for Blazor apps uses the *_ViewImports.cshtml* mechanism for layout selection. A newly created app contains the *_ViewImports.cshtml* file in the *Pages* folder.

Nested layouts

Blazor apps can consist of nested layouts. A component can reference a layout which in turn references another layout. For example, nesting layouts can be used to reflect a multi-level menu structure.

The following code samples show how to use nested layouts. The *CustomersComponent.cshtml* file is the component to display. Note that the component references the layout `MasterDataLayout`.

CustomersComponent.cshtml:

```
@layout MasterDataLayout

@page "/master-data/customers"

<h1>Customer Maintenance</h1>
...
```

The *MasterDataLayout.cshtml* file provides the `MasterDataLayout`. The layout references another layout, `MainLayout`, where it's going to be embedded.

MasterDataLayout.cshtml:

```
@layout MainLayout
@inherits BlazorLayoutComponent

<nav>
    <!-- Menu structure of master data module -->
    ...
</nav>

@Body
```

Finally, `MainLayout` contains the top-level layout elements, such as the header, footer, and main menu.

MainLayout.cshtml:

```
@inherits BlazorLayoutComponent

<header>...</header>
<nav>...</nav>

@Body
```

Dependency injection in Blazor

By Rainer Stropek

Note

Blazor is an unsupported experimental web framework that shouldn't be used for production workloads at this time.

Blazor has dependency injection (DI) built-in. Apps can use built-in services by having them injected into components. Apps can also define custom services and make them available via DI.

What is dependency injection?

DI is a technique for accessing services configured in a central location. This can be useful to:

- Share a single instance of a service class across many components (known as a *singleton* service).
- Decouple components from particular concrete service classes and only reference abstractions. For example, an interface `IDataAccess` is implemented by a concrete class `DataAccess`. When a component uses DI to receive an `IDataAccess` implementation, the component isn't coupled to the concrete type. The implementation can be swapped, perhaps to a mock implementation in unit tests.

Blazor's DI system is responsible for supplying instances of services to components. DI also resolves dependencies recursively so that services themselves can depend on further services. DI is configured during startup of the app. An example is shown later in this topic.

Add services to DI

After creating a new app, examine the `Startup.ConfigureServices` method:

```
public void ConfigureServices(IServiceCollection services)
{
    // Add custom services here
}
```

The `ConfigureServices` method is passed an IServiceCollection, which is a list of service descriptor objects (ServiceDescriptor). Services are added by providing service descriptors to the service collection. The following code sample demonstrates the concept:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddSingleton<IDataAccess, DataAccess>();
}
```

Services can be configured with the following lifetimes:

METHOD	DESCRIPTION
Singleton	DI creates a *single instance* of the service. All components requiring this service receive a reference to this instance.
Transient	Whenever a component requires this service, it receives a *new instance* of the service.
Scoped	Blazor doesn't currently have the concept of DI scopes. `Scoped` behaves like `Singleton`. Therefore, prefer `Singleton` and avoid `Scoped`.

Blazor's DI system is based on the DI system in ASP.NET Core. For more information, see Dependency injection in ASP.NET Core.

Default services

Blazor provides default services that are automatically added to the service collection of an app. The following table shows a list of the default services currently provided by Blazor's BrowserServiceProvider.

METHOD	DESCRIPTION
IUriHelper	Helpers for working with URIs and navigation state (singleton).
HttpClient	Provides methods for sending HTTP requests and receiving HTTP responses from a resource identified by a URI (singleton). Note that this instance of `HttpClient` uses the browser for handling the HTTP traffic in the background. HttpClient.BaseAddress is automatically set to the base URI prefix of the app.

Note that it is possible to use a custom services provider instead of the default `BrowserServiceProvider` that's added by the default template. A custom service provider doesn't automatically provide the default services listed in the table. Those services must be added to the new service provider explicitly.

Request a service in a component

Once services are added to the service collection, they can be injected into the components' Razor templates using the `@inject` Razor directive. `@inject` has two parameters:

- Type name: The type of the service to inject.
- Property name: The name of the property receiving the injected app service. Note that the property doesn't require manual creation. The compiler creates the property.

Multiple `@inject` statements can be used to inject different services.

The following example shows how to use `@inject`. The service implementing `Services.IDataAccess` is injected into the component's property `DataRepository`. Note how the code is only using the `IDataAccess` abstraction:

```
@page "/customer-list"
@using Services
@inject IDataAccess DataRepository

<ul>
    @if (Customers != null)
    {
        @foreach (var customer in Customers)
        {
            <li>@customer.FirstName @customer.LastName</li>
        }
    }
</ul>

@functions {
    private IReadOnlyList<Customer> Customers;

    protected override async Task OnInitAsync()
    {
        // The property DataRepository received an implementation
        // of IDataAccess through dependency injection. Use
        // DataRepository to obtain data from the server.
        Customers = await DataRepository.GetAllCustomersAsync();
    }
}
```

Internally, the generated property (`DataRepository`) is decorated with the InjectAttribute attribute. Typically, this attribute isn't used directly. If a base class is required for components and injected properties are also required for the base class, `InjectAttribute` can be manually added:

```
public class ComponentBase : BlazorComponent
{
    // Blazor's dependency injection works even if using the
    // InjectAttribute in a component's base class.
    [Inject]
    protected IDataAccess DataRepository { get; set; }
    ...
}
```

In components derived from the base class, the `@inject` directive isn't required. The `InjectAttribute` of the base class is sufficient:

```
@page "/demo"
@inherits ComponentBase

<h1>...</h1>
...
```

Dependency injection in services

Complex services might require additional services. In the prior example, `DataAccess` might require Blazor's default service `HttpClient`. `@inject` or the `InjectAttribute` can't be used in services. *Constructor injection* must be used instead. Required services are added by adding parameters to the service's constructor. When dependency injection creates the service, it recognizes the services it requires in the constructor and provides them accordingly.

The following code sample demonstrates the concept:

```
public class DataAccess : IDataAccess
{
    // The constructor receives an HttpClient via dependency
    // injection. HttpClient is a default service offered by Blazor.
    public DataAccess(HttpClient client)
    {
        ...
    }
    ...
}
```

Note the following prerequisites for constructor injection:

- There must be one constructor whose arguments can all be fulfilled by dependency injection. Note that additional parameters not covered by DI are allowed if default values are specified for them.
- The applicable constructor must be *public*.
- There must only be one applicable constructor. In case of an ambiguity, DI throws an exception.

Additional resources

- Dependency injection in ASP.NET Core

Blazor routing

By Luke Latham

▯ Note

Blazor is an unsupported experimental web framework that shouldn't be used for production workloads at this time.

Learn how to route requests in a client-side Blazor app and about the NavLink component.

View or download sample code (how to download). See the Get started topic for prerequisites.

Route templates

The **<Router>** component enables routing, and a route template is provided to each accessible component. The **<Router>** component appears in the *App.cshtml* file:

```
<Router AppAssembly=typeof(Program).Assembly />
```

When a **.cshtml* file with an `@page` directive is compiled, the generated class is given a RouteAttribute specifying the route template. At runtime, the router looks for component classes with a `RouteAttribute` and renders whichever component has a route template that matches the requested URL.

Multiple route templates can be applied to a component. In the sample app, the following component responds to requests for `/BlazorRoute` and `/DifferentBlazorRoute`.

Pages/BlazorRoute.cshtml:

```
@page "/BlazorRoute"
@page "/DifferentBlazorRoute"

<h1>Blazor routing</h1>
```

Route parameters

The Blazor client-side router uses route parameters to populate the corresponding component parameters with the same name (case insensitive).

Pages/RouteParameter.cshtml:

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

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

@functions {
    [Parameter]
    private string Text { get; set; } = "fantastic";
```

Optional parameters aren't supported yet, so two `@page` directives are applied in the example above. The first permits navigation to the component without a parameter. The second `@page` directive takes the `{text}` route parameter and assigns the value to the `Text` property.

NavLink component

Use a NavLink component in place of HTML **<a>** elements when creating navigation links. A NavLink component behaves like an **<a>** element, except it toggles an `active` CSS class based on whether its `href` matches the current URL. The `active` class helps a user understand which page is the active page among the navigation links displayed.

The NavMenu component in the sample app creates a Bootstrap nav bar that demonstrates how to use NavLink components. The following markup shows the first two NavLinks in the *Shared/NavMenu.cshtml* file.

```
<div class="navbar-collapse collapse">
    <ul class="nav navbar-nav">
        <li>
            <NavLink href="" Match=NavLinkMatch.All>
                <span class="glyphicon glyphicon-home"></span> Home
            </NavLink>
        </li>
        <li>
            <NavLink href="BlazorRoute">
                <span class="glyphicon glyphicon-th-list"></span> Blazor Route
            </NavLink>
        </li>
```

There are two NavLinkMatch options:

- `NavLinkMatch.All` – Specifies that the NavLink should be active when it matches the entire current URL.
- `NavLinkMatch.Prefix` – Specifies that the NavLink should be active when it matches any prefix of the current URL.

In the preceding example, the Home NavLink (`href=""`) matches all URLs and always receives the `active` CSS class. The second NavLink only receives the `active` class when the user visits the BlazorRoute component (`href="BlazorRoute"`).

Blazor JavaScript interop

By Javier Calvarro Nelson, Daniel Roth, and Luke Latham

Note

Blazor is an unsupported experimental web framework that shouldn't be used for production workloads at this time.

A Blazor app can invoke JavaScript functions from .NET and .NET methods from JavaScript code.

Invoke JavaScript functions from .NET methods

There are times when Blazor .NET code is required to call a JavaScript function. For example, a JavaScript call can expose browser capabilities or functionality from a JavaScript library to the Blazor app.

To call into JavaScript from .NET, use the `IJSRuntime` abstraction, which is accessible from `JSRuntime.Current`. The `InvokeAsync<T>` method on `IJSRuntime` takes an identifier for the JavaScript function you wish to invoke along with any number of JSON serializable arguments. The function identifier is relative to the global scope (`window`). For example if you wish to call `window.someScope.someFunction`, the identifier is `someScope.someFunction`. There's no longer any need to register the function before it's called. The return type `T` must also be JSON serializable.

exampleJsInterop.js:

ExampleJsInterop.cs:

```
public class ExampleJsInterop
{
    public static Task<string> Prompt(string message)
    {
        // Implemented in exampleJsInterop.js
        return JSRuntime.Current.InvokeAsync<string>(
            "exampleJsFunctions.showPrompt",
            message);
    }
}
```

The `IJSRuntime` abstraction is asynchronous to allow for out-of-process scenarios. If the app runs in-process and you want to invoke a JavaScript function synchronously, downcast to `IJSInProcessRuntime` and call `Invoke<T>` instead. We recommend that most JavaScript interop libraries use the async APIs to ensure the libraries are available in all Blazor scenarios, client-side or server-side.

Capture references to elements

Some JavaScript interop scenarios require references to HTML elements. For example, a UI library may require an element reference for initialization, or you might need to call command-like APIs on an element, such as `focus` or `play`.

You can capture references to HTML elements in a component by adding a `ref` attribute to the HTML element and then defining a field of type `ElementRef` whose name matches the value of the `ref` attribute.

The following example shows capturing a reference to the username input element:

```
<input ref="username" ... />

@functions {
    ElementRef username;
}
```

Note

Do **not** use captured element references as a way of populating the DOM. Doing so may interfere with Blazor's declarative rendering model.

As far as .NET code is concerned, an `ElementRef` is an opaque handle. The *only* thing you can do with it is pass it through to JavaScript code via JavaScript interop. When you do so, the JavaScript-side code receives an `HTMLElement` instance, which it can use with normal DOM APIs.

For example, the following code defines a .NET extension method that enables setting the focus on an element:

mylib.js:

ElementRefExtensions.cs:

```
using Microsoft.AspNetCore.Blazor;
using Microsoft.JSInterop;
using System.Threading.Tasks;

namespace MyLib
{
    public static class MyLibElementRefExtensions
    {
        public static Task Focus(this ElementRef elementRef)
        {
            return JSRuntime.Current.InvokeAsync<object>("myLib.focusElement", elementRef);
        }
    }
}
```

Now you can focus inputs in any of your components:

```
@using MyLib

<input ref="username" />
<button onclick="@SetFocus">Set focus</button>

@functions {
    ElementRef username;

    void SetFocus()
    {
        username.Focus();
    }
}
```

Important

The `username` variable is only populated after the component renders and its output includes the `<input>` element. If you try to pass an unpopulated `ElementRef` to JavaScript code, the JavaScript code receives `null`. To manipulate element references after the component has finished rendering (to set the initial focus on an element) use the `OnAfterRenderAsync` or `OnAfterRender` component lifecycle methods.

Invoke .NET methods from JavaScript functions

To invoke a static .NET method from JavaScript, use the `DotNet.invokeMethod` or `DotNet.invokeMethodAsync` functions. Pass in the identifier of the static method you wish to call, the name of the assembly containing the function, and any arguments. Again, the async version is preferred to support out-of-process scenarios. To be invokable from JavaScript, the .NET method must be public, static, and decorated with `[JSInvokable]`. By default, the method identifier is the method name, but you can specify a different identifier using the `JSInvokableAttribute` constructor. Calling open generic methods isn't currently supported.

JavaScriptInteroperable.cs:

```
public class JavaScriptInvokable
{
    [JSInvokable]
    public static Task<int[]> ReturnArrayAsync()
    {
        return Task.FromResult(new int[] { 1, 2, 3 });
    }
}
```

dotnetInterop.js:

New in Blazor 0.5.0, you can also call .NET instance methods from JavaScript. To invoke a .NET instance method from JavaScript, first pass the .NET instance to JavaScript by wrapping it in a `DotNetObjectRef` instance. The .NET instance is passed by reference to JavaScript, and you can invoke .NET instance methods on the instance using the `invokeMethod` or `invokeMethodAsync` functions. The .NET instance can also be passed as an argument when invoking other .NET methods from JavaScript.

ExampleJsInterop.cs:

```
public class ExampleJsInterop
{
    public static Task SayHello(string name)
    {
        return JSRuntime.Current.InvokeAsync<object>(
            "exampleJsFunctions.sayHello",
            new DotNetObjectRef(new HelloHelper(name)));
    }
}
```

exampleJsInterop.js:

HelloHelper.cs:

```
public class HelloHelper
{
    public HelloHelper(string name)
    {
        Name = name;
    }

    public string Name { get; set; }

    [JSInvokable]
    public string SayHello() => $"Hello, {Name}!";
}
```

Output:

```
Hello, Blazor!
```

Share interop code in a Blazor class library

JavaScript interop code can be included in a Blazor class library (`dotnet new blazorlib`), which allows you to share the code in a NuGet package.

The Blazor class library handles embedding JavaScript resources in the built assembly. The JavaScript files are placed in the *wwwroot* folder, and the tooling takes care of embedding the resources when the library is built.

The built NuGet package is referenced in the project file of a Blazor app just as any normal NuGet package is referenced. After the app has been restored, app code can call into JavaScript as if it were C#.

Blazor debugging

Daniel Roth

Note

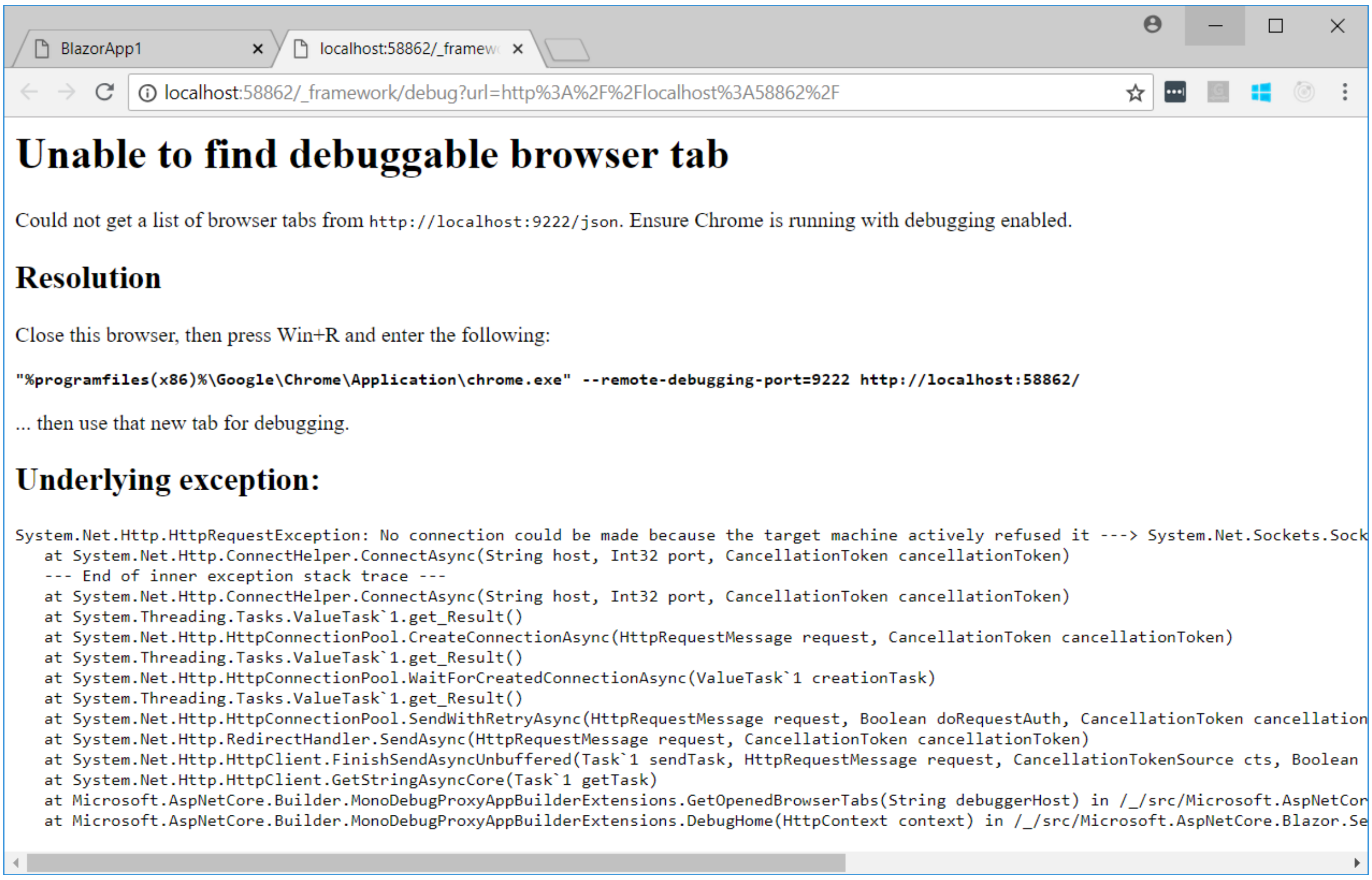
Blazor is an unsupported experimental web framework that shouldn't be used for production workloads at this time.

Blazor has some *very early* support for debugging client-side Blazor apps running on WebAssembly in Chrome. While this initial debugging support is very limited and unpolished, it shows the basic debugging infrastructure coming together.

To debug a client-side Blazor app in Chrome:

- Build a Blazor app in `Debug` configuration (the default for non-published apps).
- Run the Blazor app in Chrome.
- With the keyboard focus on the app (not in the dev tools panel, which you should probably close for a less confusing debugging experience), select the following Blazor-specific keyboard shortcut:
  - `Shift+Alt+D` on Windows/Linux
  - `Shift+Cmd+D` on macOS

Run Chrome with remote debugging enabled to debug the Blazor app. If remote debugging is disabled, an error page is generated by Chrome. The error page contains instructions for running Chrome with the debugging port open so that the Blazor debugging proxy can connect to the app. *Close all Chrome instances* and restart Chrome as instructed.


BlazorApp1
localhost:58862/_framew
localhost:58862/_framework/debug?url=http%3A%2F%2Flocalhost%3A58862%2F

Unable to find debuggable browser tab

Could not get a list of browser tabs from `http://localhost:9222/json`. Ensure Chrome is running with debugging enabled.

Resolution

Close this browser, then press Win+R and enter the following:

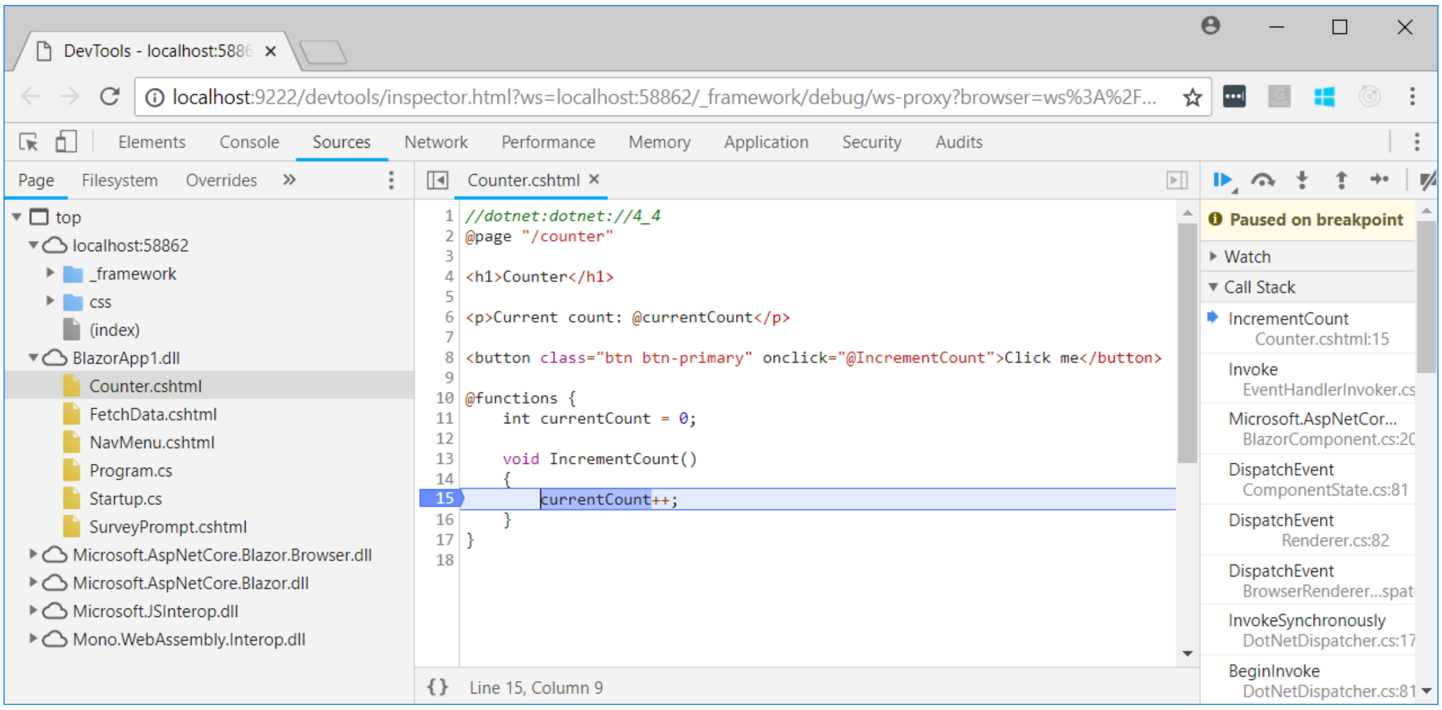
```
"%programfiles(x86)%\Google\Chrome\Application\chrome.exe" --remote-debugging-port=9222 http://localhost:58862/
```

... then use that new tab for debugging.

Underlying exception:

```
System.Net.Http.HttpRequestException: No connection could be made because the target machine actively refused it ---> System.Net.Sockets.Sock
   at System.Net.Http.ConnectHelper.ConnectAsync(String host, Int32 port, CancellationToken cancellationToken)
   --- End of inner exception stack trace ---
   at System.Net.Http.ConnectHelper.ConnectAsync(String host, Int32 port, CancellationToken cancellationToken)
   at System.Threading.Tasks.ValueTask`1.get_Result()
   at System.Net.Http.HttpConnectionPool.CreateConnectionAsync(HttpRequestMessage request, CancellationToken cancellationToken)
   at System.Threading.Tasks.ValueTask`1.get_Result()
   at System.Net.Http.HttpConnectionPool.WaitForCreatedConnectionAsync(ValueTask`1 creationTask)
   at System.Threading.Tasks.ValueTask`1.get_Result()
   at System.Net.Http.HttpConnectionPool.SendWithRetryAsync(HttpRequestMessage request, Boolean doRequestAuth, CancellationToken cancellation
   at System.Net.Http.RedirectHandler.SendAsync(HttpRequestMessage request, CancellationToken cancellationToken)
   at System.Net.Http.HttpClient.FinishSendAsyncUnbuffered(Task`1 sendTask, HttpRequestMessage request, CancellationTokenSource cts, Boolean
   at System.Net.Http.HttpClient.GetStringAsyncCore(Task`1 getTask)
   at Microsoft.AspNetCore.Builder.MonoDebugProxyAppBuilderExtensions.GetOpenedBrowserTabs(String debuggerHost) in /_/src/Microsoft.AspNetCor
   at Microsoft.AspNetCore.Builder.MonoDebugProxyAppBuilderExtensions.DebugHome(HttpContext context) in /_/src/Microsoft.AspNetCore.Blazor.Se
```

Once Chrome is running with remote debugging enabled, the debugging keyboard shortcut opens a new debugger tab. After a moment, the *Sources* tab shows a list of the .NET assemblies in the app. Expand each assembly and find the *.cs/.cshtml* source files available for debugging. Set breakpoints, switch back to the app's tab, and the breakpoints are hit. After a breakpoint is hit, single-step (`F10`) or resume (`F8`) normally.



```
//dotnet:dotnet://4_4
@page "/counter"

<h1>Counter</h1>

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

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

@functions {
    int currentCount = 0;

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

Blazor provides a debugging proxy that implements the Chrome DevTools Protocol and augments the protocol with .NET-specific information. When debugging keyboard shortcut is pressed, Blazor points the Chrome DevTools at the proxy. The proxy connects to the browser window you're seeking to debug (hence the need to enable remote debugging).

You might be wondering why we don't just use browser source maps. Source maps allow the browser to map compiled files back to their original source files. However, Blazor does not map C# directly to JS/WASM (at least not yet). Instead, Blazor does IL interpretation within the browser, so source maps aren't relevant.

Note that the debugger capabilities are **very limited.** You can currently only:

- Single-step through the current method (`F10`) or resume (`F8`).
- In the *Locals* display, observe the values of any local variables of type `int`, `string`, and `bool`.
- See the call stack, including call chains that go from JavaScript into .NET and from .NET to JavaScript.

You *can't*:

- Step into child methods (`F11`).
- Observe the values of any locals that aren't an `int`, `string`, or `bool`.
- Observe the values of any class properties or fields.
- Hover over variables to see their values
- Evaluate expressions in the console.
- Step across async calls.
- Perform most other ordinary debugging scenarios.

Development of further debugging scenarios is an on-going focus of the engineering team.

Host and deploy Blazor

By Luke Latham, Rainer Stropek, and Daniel Roth

Note

Blazor is an unsupported experimental web framework that shouldn't be used for production workloads at this time.

Publish the app

Blazor apps are published for deployment in Release configuration with the dotnet publish command. An IDE may handle executing the `dotnet publish` command automatically using its built-in publishing features, so it might not be necessary to manually execute the command from a command prompt depending on the development tools in use.

```
dotnet publish -c Release
```

`dotnet publish` 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. The deployment is created in the */bin/Release/<target-framework>/publish* folder.

The assets in the *publish* folder are deployed to the web server. Deployment might be a manual or automated process depending on the development tools in use.

Configure the Linker

Blazor performs Intermediate Language (IL) linking on each build to remove unnecessary IL from the output assemblies. You can control assembly linking on build. For more information, see Configure the Linker.

Rewrite URLs for correct routing

Routing requests for page components in a client-side app isn't as simple as routing requests to a server-side, hosted app. Consider a client-side app with two pages:

- ***Main.cshtml*** – Loads at the root of the app and contains a link to the About page (`href="About"`).
- ***About.cshtml*** – About page.

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. Blazor's router loads and the Razor Main page (*Main.cshtml*) is displayed.

On the Main page, selecting the link to the About page loads the About page. Selecting the link to the About page 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 About page itself. All of the requests for internal pages *within the client-side 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 client-side router takes over and responds with the correct resource.

App base path

The app base path is the virtual app root path on the server. For example, an app that resides on the Contoso server in a virtual folder at `CoolBlazorApp/` is reached at `https://www.contoso.com/CoolBlazorApp` and has a virtual base path of `CoolBlazorApp/`. By setting the app base path to `CoolBlazorApp/`, the app is made aware of where it virtually resides on the server. The app can use the app base path to construct URLs relative to the app root from a component that isn't in the root directory. This allows components that exist at different levels of the directory structure to build links to other resources at locations throughout the app. The app base path is also used to intercept hyperlink clicks where the `href` target of the link is within the app base path URI space—the Blazor router handles the internal navigation.

In many hosting scenarios, the server's virtual path to the app is the root of the app. In these cases, the app base path is an empty string, which is the default configuration for a Blazor app. In other hosting scenarios, such as GitHub Pages and IIS virtual directories, the app base path must be set to the server's virtual path to the app. To set the Blazor app's base path, add or update the **<base>** tag in *index.html* on the **<head>** tag. Set the `href` attribute value to `<virtual-path>/` (the trailing slash is required), where `<virtual-path>/` is the full virtual app root path on the server for the app.

Deployment models

There are two deployment models for Blazor apps:

- Hosted deployment with ASP.NET Core – Hosted deployment uses an ASP.NET Core app on the server to host the Blazor app.
- Standalone deployment – Standalone deployment places the Blazor app on a static hosting web server or service, where .NET isn't used to serve the Blazor app.

Hosted deployment with ASP.NET Core

In a hosted deployment, an ASP.NET Core app handles single-page application routing and Blazor app hosting. The published ASP.NET Core app, along with one or more Blazor apps that it hosts, is deployed to the web server or hosting service.

To host a Blazor app, the ASP.NET Core app must:

- Reference the Blazor app project.
- Reference the Microsoft.AspNetCore.Blazor.Server package in its project file.
- Configure Blazor app hosting with the `UseBlazor` extension method on IApplicationBuilder in `Startup.Configure`.

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }

    app.UseBlazor<Client.Program>();
}
```

The `UseBlazor` extension method performs the following tasks:

- Configure Static File Middleware to serve Blazor's static assets from the *dist* folder. In the Development environment, the files in *wwwroot* folder are served.
- Configure single-page application routing for resource requests that aren't for actual files that exist on disk. The app serves the default document (*wwwroot/index.html*) for any request that hasn't been served by a prior Static File Middleware instance. For example, a request to receive a page from the app that should be handled by the Blazor router on the client is rewritten into a request for the *wwwroot/index.html* page.

When the ASP.NET Core app is published, the Blazor app is included in the published output so that the ASP.NET Core app and the Blazor app can be deployed together. For more information on ASP.NET Core app hosting and deployment, see Host and deploy

ASP.NET Core.

For information on deploying to Azure App Service, see the following topics:

Publish to Azure with Visual Studio
Learn how to publish an ASP.NET Core-hosted Blazor app to Azure App Service using Visual Studio.

Publish to Azure with CLI tools
Learn how to publish an ASP.NET Core app to Azure App Service using the Git command-line client.

Standalone deployment

In a standalone deployment, only the Blazor client-side app is deployed to the server or hosting service. An ASP.NET Core server-side app isn't used to host the Blazor app. The Blazor app's static files are requested by the browser directly from the static file web server or service.

When deploying a standalone Blazor app from the published *dist* folder, any web server or hosting service that serves static files can host a Blazor app.

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* folder. 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 are set for the following file extensions:
  - **.dll*: `application/octet-stream`
  - **.json*: `application/json`
  - **.wasm*: `application/wasm`
  - **.woff*: `application/font-woff`
  - **.woff2*: `application/font-woff`
- 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 (*<assembly_name>\dist\<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 (*<assembly_name>\dist\index.html*).

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 an 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 Blazor 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.

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 IIS.

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;
        }
    }
}
```

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

Nginx in Docker

To host Blazor in Docker using Nginx, setup the Dockerfile to use the Alpine-based Nginx image. Update the Dockerfile to copy the *nginx.config* file into the container.

Add one line to the Dockerfile, as shown in the following example:

```
FROM nginx:alpine
COPY ./bin/Release/netstandard2.0/publish /usr/share/nginx/html/
COPY nginx.conf /etc/nginx/nginx.conf
```

GitHub Pages

To handle URL rewrites, add a *404.html* file with a script that handles redirecting the request to the *index.html* page. For an example implementation provided by the community, see Single Page Apps for GitHub Pages (rafrex/spa-github-pages on GitHub). An example using the community approach can be seen at blazor-demo/blazor-demo.github.io on GitHub (live site).

When using a project site instead of an organization site, add or update the **<base>** tag in *index.html*. Set the `href` attribute value to `<repository-name>/`, where `<repository-name>/` is the GitHub repository name.

Blazor hosting models

By Daniel Roth

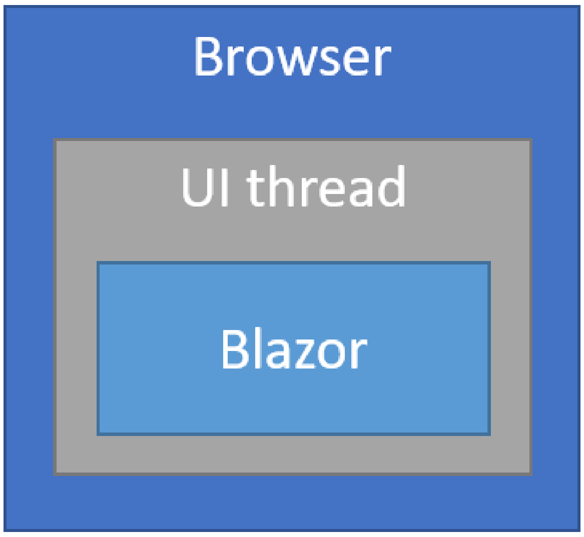
Note

Blazor is an unsupported experimental web framework that shouldn't be used for production workloads at this time.

Blazor is a client-side web framework designed primarily to run in the browser on a WebAssembly-based .NET runtime. Blazor supports multiple hosting models, including out-of-process hosting models, where the Blazor component logic runs separately from the UI thread. Regardless of the hosting model, the Blazor app and component models remain *the same*. This article discusses the available hosting models for Blazor.

Client-side (in-process) hosting model

The principal hosting model for Blazor is running client-side in the browser. In this model, the Blazor app, its dependencies, and the .NET runtime are downloaded to the browser, and the app is executed directly on the browser UI thread. All UI updates and event handling happens within the same process. The app assets can be deployed as static files using whatever web server is preferred (see Host and deploy).



To create a Blazor app using the client-side hosting model, use the "Blazor" or "Blazor (ASP.NET Core Hosted)" project templates (`blazor` or `blazorhosted` template when using dotnet new at a command prompt). The included *blazor.webassembly.js* script handles:

- Downloading the .NET runtime, the app, and its dependencies.
- Initialization of the runtime to run the app.

The benefits of the client-side hosting model are:

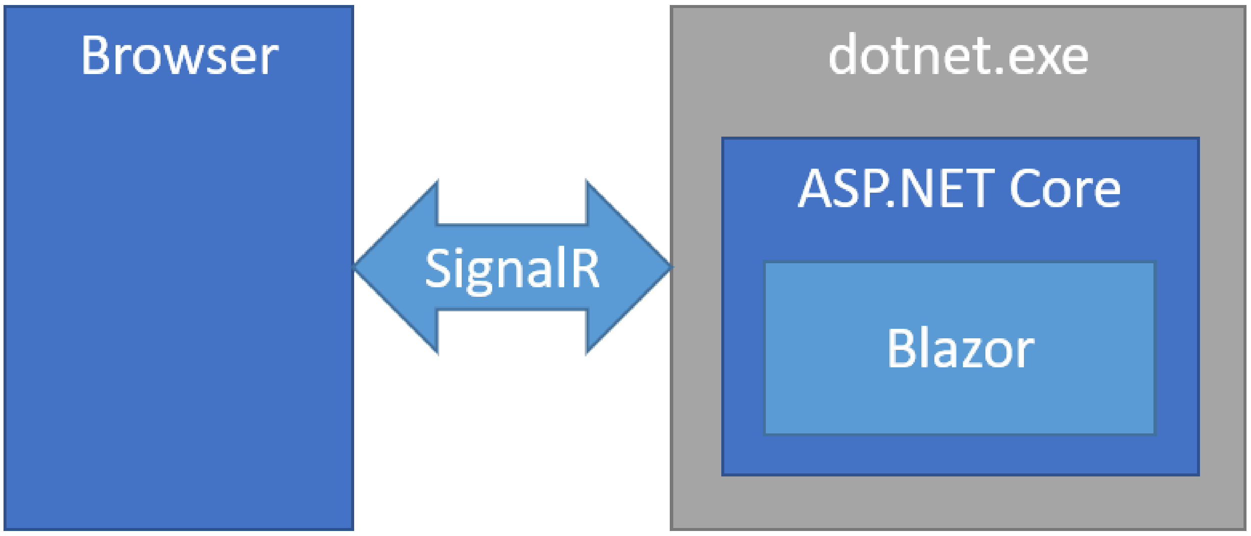
- No .NET server-side dependency
- Rich interactive UI
- Fully leverage the client resources and capabilities
- Offload work from the server to the client
- Support offline scenarios

The downsides to the client-side hosting model are:

- Restricted to the capabilities of the browser
- Requires more capable client hardware and software (for example, WebAssembly support)
- Larger download size and app load time
- Less mature .NET runtime and tooling support (for example, limitations in .NET Standard support and debugging)

Server-side hosting model

In the 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.



To create a Blazor app using the server-side hosting model, use the "Blazor (Server-side on ASP.NET Core)" template (`blazorserver` when using dotnet new at a command prompt). An ASP.NET Core app hosts the Blazor server-side app and sets up the SignalR endpoint where clients connect. The ASP.NET Core app references the Blazor `Startup` class to both add the server-side Blazor services and to add the Blazor app to the request handling pipeline:

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        ...
        services.AddServerSideBlazor<App.Startup>();
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env)
    {
        ...
        app.UseServerSideBlazor<App.Startup>();
    }
}
```

The *blazor.server.js* script establishes the client connection. It's the app's responsibility to persist and restore app state as needed (for example, in the event of a lost network connection).

The benefits of the server-side hosting model are:

- You can still write your entire app with .NET and C# using the Blazor component model.
- Your app still has a rich interactive feel and avoids unnecessary page refreshes.
- Your app download size is significantly smaller, and the initial app load time is much faster.
- Your Blazor component logic can take full advantage of server capabilities, including using any .NET Core compatible APIs.
- Because you're running on .NET Core on the server, existing .NET tooling, such as debugging, works as expected.
- Server-side hosting works with thin clients (for example, browsers that don't support WebAssembly and resource constrained devices).

The downsides of the server-side hosting model are:

- Latency: every user interaction now involves a network hop.
- No offline support: if the client connection goes down, the app stops working.

- Scalability: the server must manage multiple client connections and handle client state.

Configure the Linker

By Luke Latham

Note

Blazor is an unsupported experimental web framework that shouldn't be used for production workloads at this time.

Blazor performs Intermediate Language (IL) linking during each Release mode build to remove unnecessary IL from the output assemblies.

You can control assembly linking with either of the following approaches:

- Disable linking globally with an MSBuild property.
- Control linking on a per-assembly basis with a configuration file.

Disable linking with an MSBuild property

Linking is enabled by default in Release mode when an app is built, which includes publishing. To disable linking for all assemblies, set the `<BlazorLinkOnBuild>` MSBuild property to `false` in the project file:

```
<PropertyGroup>
  <BlazorLinkOnBuild>false</BlazorLinkOnBuild>
</PropertyGroup>
```

Control linking with a configuration file

Linking can be controlled on a per-assembly basis by providing an XML configuration file and specifying the file as an MSBuild item in the project file.

The following is an example configuration file (*Linker.xml*):

```
<?xml version="1.0" encoding="UTF-8" ?>
<!--
  This file specifies which parts of the BCL or Blazor packages must not be
  stripped by the IL Linker even if they aren't referenced by user code.
-->
<linker>
  <assembly fullname="mscorlib">
    <!--
      Preserve the methods in WasmRuntime because its methods are called by
      JavaScript client-side code to implement timers.
      Fixes: https://github.com/aspnet/Blazor/issues/239
    -->
    <type fullname="System.Threading.WasmRuntime" />
  </assembly>
  <assembly fullname="System.Core">
    <!--
      System.Linq.Expressions* is required by Json.NET and any
      expression.Compile caller. The assembly isn't stripped.
    -->
    <type fullname="System.Linq.Expressions*" />
  </assembly>
  <!--
    In this example, the app's entry point assembly is listed. The assembly
    isn't stripped by the IL Linker.
  -->
  <assembly fullname="MyCoolBlazorApp" />
</linker>
```

To learn more about the file format for the configuration file, see IL Linker: Syntax of xml descriptor.

Specify the configuration file in the project file with the `BlazorLinkerDescriptor` item:

```
<ItemGroup>
  <BlazorLinkerDescriptor Include="Linker.xml" />
</ItemGroup>
```

Frequently asked questions (FAQ) about Blazor

Note

Blazor is an unsupported experimental web framework that shouldn't be used for production workloads at this time.

What is Blazor?

Blazor is a single-page web app framework built on .NET that runs in the browser with WebAssembly.

I'm new to .NET. What's .NET?

.NET is a free, cross-platform, open source developer platform for building many different types of apps (desktop, mobile, games, web). .NET includes a managed runtime, a standard set of libraries, and support for multiple modern programming languages: C#, F#, and VB. You can get started with .NET in 10 min.

Why would I use .NET for web development?

Using .NET in the browser offers many advantages that can help make web development easier and more productive:

- **Stable and consistent**: .NET offers standard APIs, tools, and build infrastructure across all .NET platforms that are stable, feature rich, and easy to use.
- **Modern innovative languages**: .NET languages like C# and F# make programming a joy and keep getting better with innovative new language features.
- **Industry leading tools**: The Visual Studio product family provides a great .NET development experience on Windows, Linux, and macOS.
- **Fast and scalable**: .NET has a long history of performance, reliability, and security on the server. Using .NET as a full-stack solution makes it easier to scale your apps.

How can you run .NET in a web browser?

Running .NET in the browser is made possible by a relatively new standardized web technology called WebAssembly. WebAssembly is a "portable, size- and load-time-efficient format suitable for compilation to the web." Code compiled to WebAssembly can run in any browser at native speeds. To run .NET binaries in a web browser, we use a .NET runtime (specifically Mono) that has been compiled to WebAssembly.

Does Blazor compile my entire .NET-based app to WebAssembly?

No, a Blazor app consists of normal compiled .NET assemblies that are downloaded and run in a web browser using a WebAssembly-based .NET runtime. Only the .NET runtime itself is compiled to WebAssembly. That said, support for full static ahead of time (AoT) compilation of the app to WebAssembly may be something we add further down the road.

Wouldn't the app download size be huge if it also includes a .NET runtime?

Not necessarily. .NET runtimes come in all shapes in sizes. Early Blazor prototypes used a compact .NET runtime (including assembly execution, garbage collection, threading) that compiled to a mere 60KB of WebAssembly. Blazor now runs on Mono, which is currently significantly larger. However, opportunities for size optimization abound, including merging and trimming the runtime and application binaries. Other potential download size mitigations include caching and using a CDN.

What features will Blazor support?

Blazor will support all of the features of a modern single-page app framework:

- A component model for building composable UI
- Routing
- Layouts
- Forms and validation
- Dependency injection
- JavaScript interop
- Live reloading in the browser during development
- Server-side rendering
- Full .NET debugging both in browsers and in the IDE
- Rich IntelliSense and tooling
- Ability to run on older (non-WebAssembly) browsers via asm.js
- Publishing and app size trimming

Can I use Blazor without running .NET on the server?

Yes, a Blazor app can be deployed as a set of static files without the need for any .NET support on the server.

Can I use Blazor with ASP.NET Core on the server?

Yes! Blazor optionally integrates with ASP.NET Core to provide a seamless and consistent full-stack web development solution.

Is Blazor a .NET port of an existing JavaScript framework?

Blazor is a *new framework* inspired by existing modern single-page app frameworks, such as React, Angular, and Vue.

How can I try out Blazor?

To build your first Blazor web app check out our getting started guide.

Why is Blazor an "experimental" project?

Blazor is an experimental project because there are still many questions to answer about its viability and appeal. The purposes of this initial experimental phase are to:

- Work through technical issues.
- Gauge interest and to listen to feedback.

We're optimistic about Blazor's future; but at this time, Blazor isn't a committed product and should be considered pre-alpha.

Is this Silverlight all over again?

No, Blazor is a .NET web framework based on HTML and CSS that runs in the browser using open web standards. It doesn't require a plugin, and it works on mobile devices and older browsers.

Does Blazor use XAML?

No, Blazor is a web framework based on HTML, CSS, and other standard web technologies.

Is WebAssembly supported in all browsers?

Yes, WebAssembly has achieved cross-browser consensus and all modern browsers now support WebAssembly.

Does Blazor work on mobile browsers?

Yes, modern mobile browsers also support WebAssembly.

What about older browsers that don't support WebAssembly? For example, does Blazor work in IE?

For older browsers that don't support WebAssembly, Blazor falls back to using an asm.js-based .NET runtime. Using asm.js is slower and has a larger download size but is still quite functional.

Can I use .NET Standard libraries with Blazor?

Yes, the .NET runtime used for Blazor supports .NET Standard 2.0. APIs that aren't supported in the browser throw *Not Supported* exceptions.

Don't you need features like garbage collection and threading added to WebAssembly to make this work?

No, WebAssembly in its current state is sufficient. The .NET runtime handles its own garbage collection and threading concerns.

Can I use existing JavaScript libraries with Blazor?

Yes, Blazor apps can call into JavaScript through JavaScript interop APIs.

Can I access the DOM from a Blazor app?

You can access the DOM through JavaScript interop from .NET code. However, Blazor is a component-based framework that minimizes the need to access the DOM directly.

Why Mono? Why not .NET Core or CoreRT?

Mono is a Microsoft-sponsored open source implementation of the .NET Framework. Mono is used by Xamarin to build native client apps for Android, iOS, and macOS. Xamarin is also used by Unity for game development. Microsoft's Xamarin team announced their plans to add support for WebAssembly to Mono, and they've been making steady progress (Mono and WebAssembly - Updates on Static Compilation 1/16/2018). Because Blazor is a client-side web UI framework targeted at WebAssembly, Mono is a natural fit.

By comparison, .NET Core is primarily used for server apps and for cross-platform console apps. .NET Core can be used for creating an ASP.NET Core backend for a Blazor app but not for building the client app itself. CoreRT is a .NET Core runtime optimized for AoT compilation; and while it has WebAssembly aspirations, the project is still a work-in-progress and not a shipping product.

Where did the name "Blazor" come from?

Blazor makes heavy use of Razor, a markup syntax for HTML and C#. **Browser + Razor = Blazor!** When pronounced, it's also the name of a swanky jacket worn by hipsters that have excellent taste in fashion, style, and programming languages.