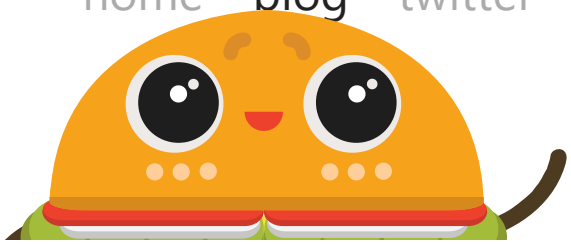
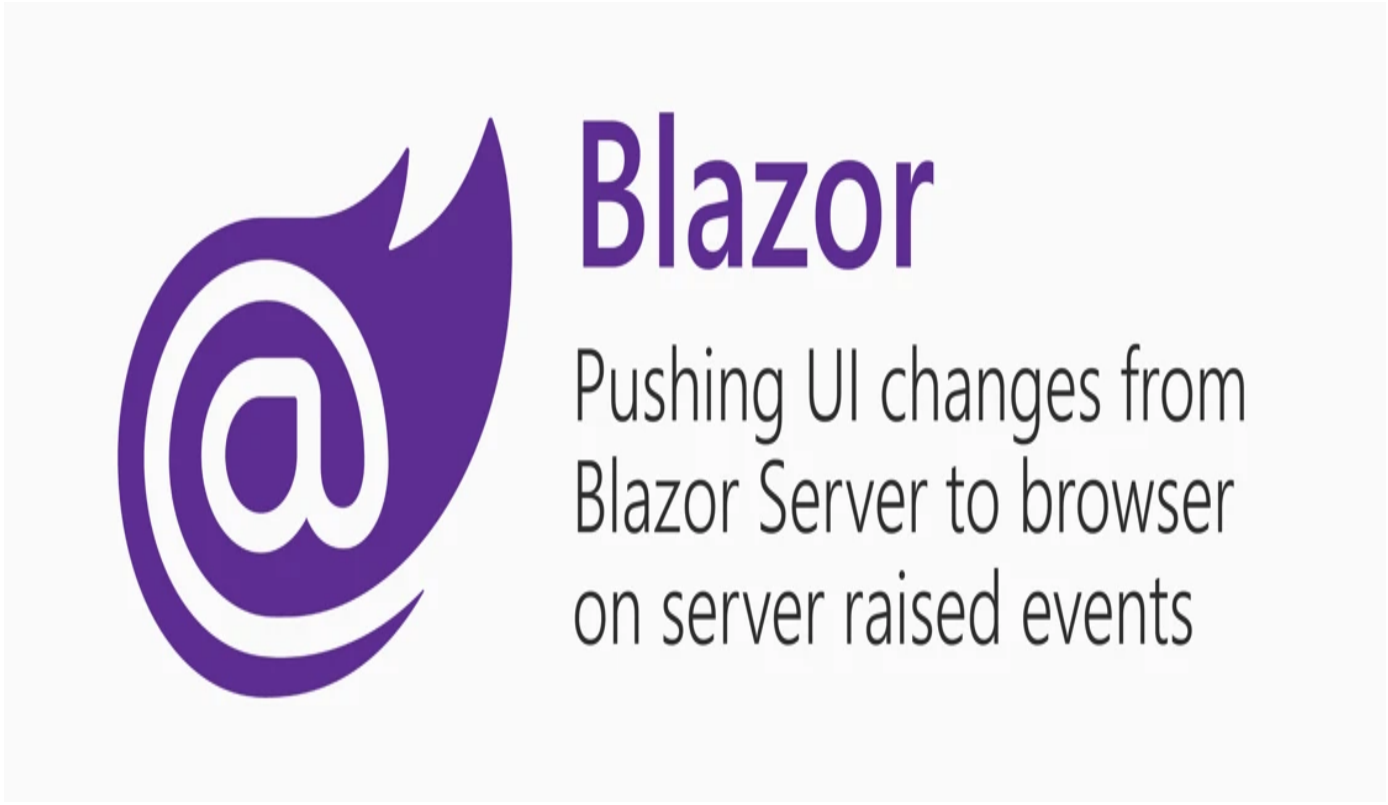


Pushing UI changes from Blazor Server to browser on server raised events

Niels Swimberghe - 10/12/2020 - .NET
Follow me on Twitter, buy me a coffee



A typical interaction between a web browser and a web server consists of a series of HTTP requests and responses. As opposed to HTTP, a websocket will open a persistent bi-directional connection where multiple messages can be sent back and forth. This also means the server can send messages to the client at any time. **Blazor Server is built on SignalR, and SignalR is built on websockets** among other techniques. The combination of these technologies allow **Blazor Server to push UI changes into the client without the client requesting those changes**.

Client side events such as button clicks are send to the Blazor Server, the server changes its state, and re-renders. Though, in some applications there are also events that are are raised elsewhere. In this article, those types of events will be referred to as 'server raised events'. Server raised events could be triggered by

- invocation of Web API's or webhooks
- a different user interacting with your server
- Queues and Event Buses
- Query Notifications in SQL Server to notify the application of data changes
- listening to data changes in real-time database such as Firestore (see walkthrough)

When these events cause the state to change, **Blazor Server will not automatically re-render the components**. Two problems need to be resolve for the components to update:

1. Blazor Server needs to be notified of the state changes. You can resolve this by calling `StateHasChanged()`.
2. The code responding to the event may **not be on the same thread as the renderer**.
   Calling `StateHasChanged()` may be ignored or throw an exception when done off the renderer's thread.
   Instead, you can pass an action to `InvokeAsync` which will invoke the action on the right thread and within the renderer's synchronization context.

When you call `StateHasChanged()` in the action passed to `InvokeAsync`, Blazor Server will successfully re-render the components and push the changes to the browser using SignalR.
To simulate server raised events, you can use a `Timer` that will invoke code on an interval.

Server raised events sample

> You can find the sample code on this GitHub Repository.

Prerequisites:

- .NET Core 3.1 or higher (download here)

Follow these steps to simulate server raised events in Blazor Server:

1. Create a new Blazor Server application by running these commands:

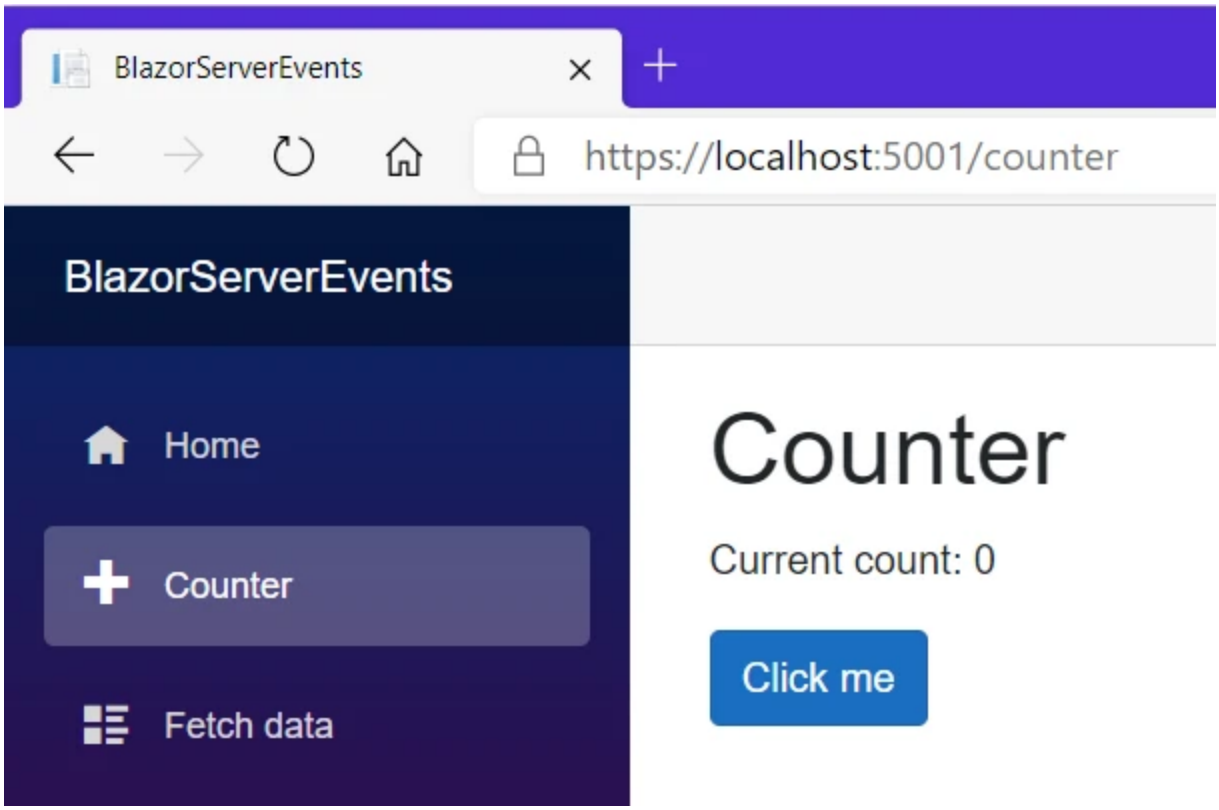
```
mkdir BlazorServerSample
cd BlazorServerSample
dotnet new blazorserver
```

2. Update the component `Pages\Counter.razor` to match the code below:

```
@page "/counter"
@implements IDisposable
@using System.Timers

<h1>Counter</h1>

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

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

@code {
    private int currentCount = 0;

    private void IncrementCount()
    {
        currentCount++;
        Console.WriteLine($"Count incremented: {currentCount}");
    }

    private Timer timer;

    protected override void OnAfterRender(bool firstRender)
    {
        if (firstRender)
        {
            timer = new Timer();
            timer.Interval = 1000;
            timer.Elapsed += OnTimerInterval;
            timer.AutoReset = true;
            // Start the timer
            timer.Enabled = true;
        }
        base.OnAfterRender(firstRender);
    }

    private void OnTimerInterval(object sender, ElapsedEventArgs e)
    {
        IncrementCount();
    }

    public void Dispose()
    {
        // During prerender, this component is rendered without calling OnAfterRender
        // this mean timer will be null so we have to check for null or use the Null-
        timer?.Dispose();
    }
}
```

Counter.razor will now configure a timer with an interval of 1000ms aka 1s. This will call `IncrementCount()` every second for as long as you are viewing the counter-page.
When navigating to another page, `Dispose` will be called and the timer resource will be cleaned up stopping the interval.

3. Run the application by running the command: `dotnet run`
4. Open a browser and navigate to the Blazor Server Application, usually at `https://localhost:5001`
5. Navigate to the counter-page and observe the changes



Unfortunately, the "Current count" does not change every second even though you programmed it to do so. You can confirm the interval is working as expected by looking at the console output, but the `currentCount` state change is not reflected in the browser. This is what the output looks like:

```
dotnet run
# output:
#   info: Microsoft.Hosting.Lifetime[0]
#         Now listening on: https://localhost:5001
#   info: Microsoft.Hosting.Lifetime[0]
#         Now listening on: http://localhost:5000
#   info: Microsoft.Hosting.Lifetime[0]
#         Application started. Press Ctrl+C to shut down.
#   info: Microsoft.Hosting.Lifetime[0]
#         Hosting environment: Development
#   info: Microsoft.Hosting.Lifetime[0]
#         Content root path: C:\Users\niels\source\repos\BlazorServerEvents
#   Count incremented: 1
#   Count incremented: 2
#   Count incremented: 3
#   Count incremented: 4
#   Count incremented: 5
#   Count incremented: 6
#   Count incremented: 7
#   Count incremented: 8
```

Click the "Click me" button and suddenly the number shown in the browser reflects the `currentCount` on the server.

When client events are send to Blazor Server, you don't have to worry about which thread you are on and you don't need to explicitly notify Blazor of the state change. All that happens automatically.
When you click the "Click me" button, `IncrementCount` is invoked and will increment the `currentCount` field which is also incremented by the timer. This results in Blazor re-rendering the component with the latest `currentCount` state.

To make sure that the state is in sync on both client and server, add `InvokeAsync(() => StateHasChanged());` to the timer interval callback. Update the function `OnTimerInterval` as below:

```
private void OnTimerInterval(object sender, ElapsedEventArgs e)
{
    IncrementCount();
    InvokeAsync(() => StateHasChanged());
}
```

When you re-run the application again, the number in the browser will be incremented every second!

Events raised outside of the Blazor Component

The timer illustrates how to handle server raised events that are being raised from within the Blazor Component class, but other events may be raised outside of the component. In that case, how would you send the event data to the active components? There are multiple ways to solve this problem, here are two suggestions:

- If the event is raised within the same .NET Core server instance (not load balanced), you could use a library like 'Blazor.EventAggregator' which allows you to publish/subscribe to events. Using this library, you can publish an event from a webhook and subscribe to an event inside your Blazor component.
- In a load balanced environment or when your webhook is hosted separately, you could use one of many event/pub/sub systems (like Redis) which allows you to submit an event from one server and have it delivered to multiple servers.

Summary

Websockets are bi-directional persistent connections over which multiple messages can be send. Websockets enable servers to push messages to the client at any time. Blazor Server is built on SignalR, which is built on websockets. Due to the bi-directional persistent connection, **Blazor Server can push UI changes to the browser without the browser requesting those changes**. Instead the changes to the UI can be triggered by server raised events. To ensure that the state and UI change are pushed to the client, you have to invoke `InvokeAsync(() => StateHasChanged());`.

This counter example may be anti-climatic, but imagine using a real-time database instead of a timer. Your Blazor Server UI can re-render in real-time!

> Check out this walkthrough "Building Real-Time Applications with Blazor Server and Firestore".

Topics

.NET | .NET Core | ASP.NET | ASP.NET Core | dotnet | dotnet core | aspnet core | Blazor | Blazor Server

Author

Niels Swimberghe

Niels Swimberghe is a Belgian American software engineer, a technical content creator at Twilio, and a Microsoft MVP. Niels is the .NET editor for Twilio Blog. Get in touch with Niels on Twitter @RealSwimburger and follow Niels' personal blog on .NET, Azure, and web development at swimburger.net.

Related Posts

See All Posts

Real-time applications with Blazor Server and Firestore

10/16/2020 - DotNet

Blazor Server is built on SignalR, which is built on websockets. Among things, websockets enable Blazor Server to push changes from the server to the browser at any time. You can build real-time UI's when you

combine this with a real-time database.

How to run code after Blazor component has rendered

8/17/2020 - .NET

Blazor components render their template whenever state has changed and sometimes you need to invoke some code after rendering has completed. This blog post will show you how to run code after your Blazor component has rendered, on every render or as needed.

How to deploy Blazor WASM & Azure Functions to Azure Static Web Apps using GitHub

8/24/2020 - .NET

With ASP.NET Blazor WebAssembly you can create .NET applications that run inside of the browser . The output of a Blazor WASM project are all static files. You can deploy these applications to static site hosts, such as Azure Static Web Apps and GitHub Pages.