Content Injection with Response Rewriting in ASP.NET Core 3.x

I ran into an odd problem recently with my Westwind.AspNetCore.LiveReload middleware component recently. This middleware provides optional live reload functionality to ASP.NET Core projects letting you reload the active page as soon as any monitored file is changed. Rather than an external tool it provides this functionality as middleware that can be plugged in and turned on/off via configuration.

As part of that middleware logic, the component needs to inject some JavaScript for the WebSocket interaction between a loaded HTML page into HTML pages displayed in the browser. Basically any HTML content injects this reload script code and this injection code is what was causing me problems.

HTML Injection in ASP.NET Core Content

Let's back up for a second and talk about Response buffering and modifying content in `Response.Body`.

If you want to do Response filtering or rewriting you need to intercept the Response output stream and then intercept and look at the outgoing bytes written and rewrite them with your updated data.

The way this has always been done in ASP.NET is what used to be known as Response filtering which would essentially take over the existing output stream and use a new stream on top if it. The streams could be chained as each new stream would use the previously defined stream to write its output, which means that any stream overrides are essentially cumulative.

Response Wrapping in .NET Core 2.x

That's been working great in .NET Core 2.x and it looks something like this:

```
private async Task HandleHtmlInjection(HttpContext context)
{
    using (var filteredResponse = new ResponseStreamWrapper(context.Response.Body, context))
    {
        context.Response.Body = filteredResponse;
        await _next(context);
    }
}
```

This essentially wraps the existing `context.Response.Body` stream into my new custom stream and my stream then looks at the bytes as they are coming through and rewrites them as necessary.

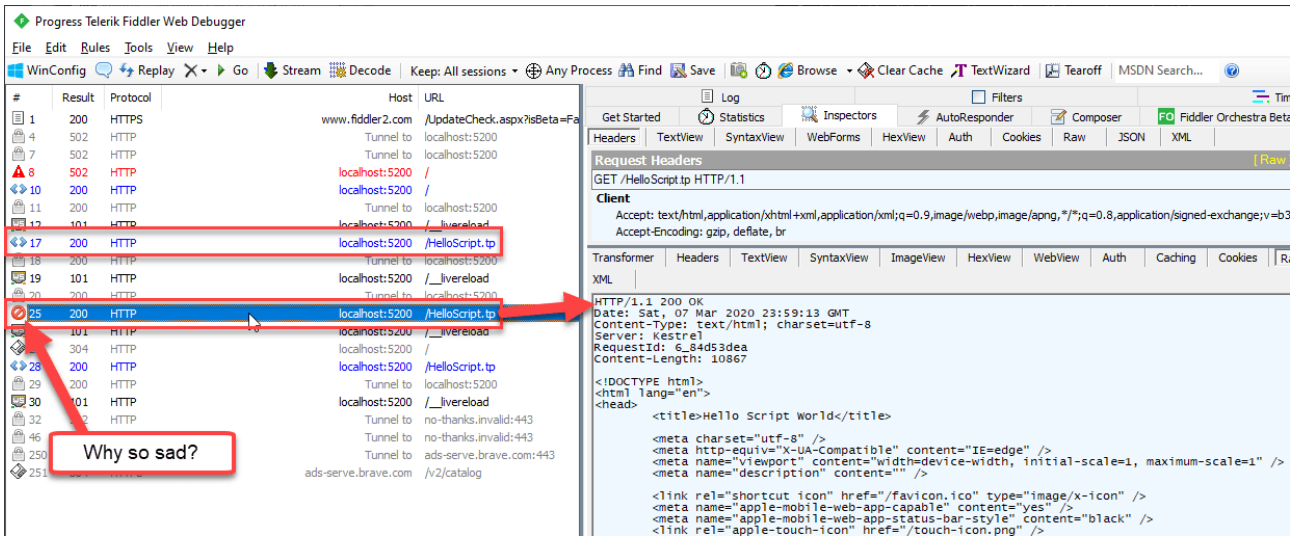
This works and has been working fine in ASP.NET Core 2.x.

ASP.NET 3.x and Response.Body HTTP Failures

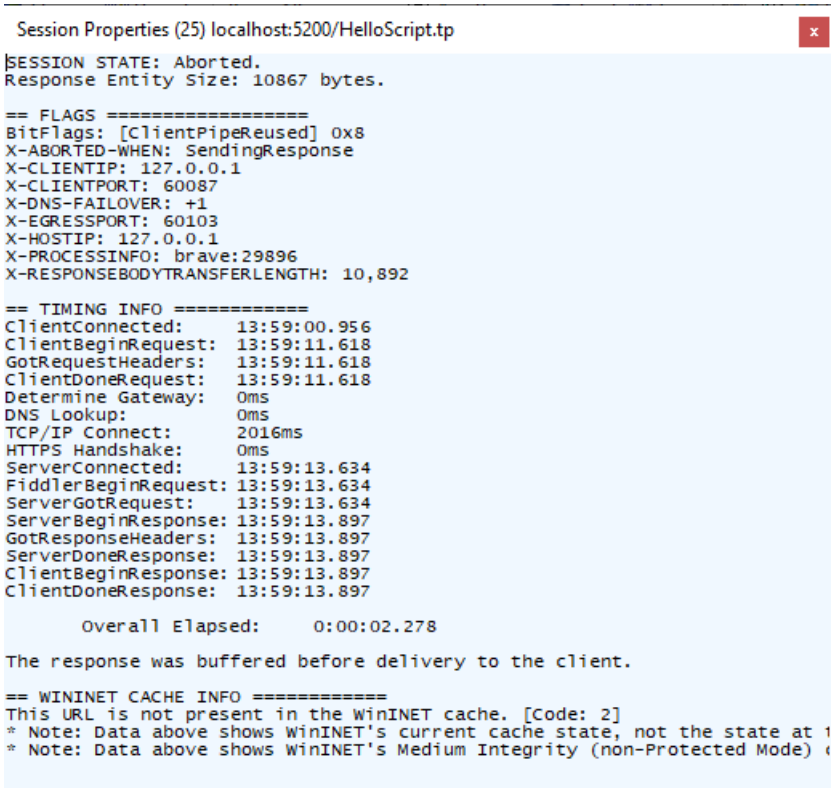
The above code also compiles in 3.x and at first glance it works just fine there as well. And why shouldn't it, right?

But - as I found I saw infrequent, random HTTP failures on some HTTP requests. Clicking on a link to certain pages I can see the request hitting the server, and even a response getting returned, but the browser navigation is failing - ie. nothing happens.

When looking in Fiddler on these same requests that are failing I see this:



Notice the response code is a `200` response, but yet it shows with an error icon next to it. Per Eric Lawrence (the original author of Fiddler) I also checked the Session properties:



which seems to suggest that the response was aborted (`X-ABORTED-WHEN`). But looking at the actual response data it's all there. But I can't for the life of me see why that is. The content looks good, the size is good. What is Fiddler and

the browser seeing that I'm not?

Worse **it works most of the time** - it only fails occasionally on the same pages, creating the exact same content. Lovely!

Fixing the ASP.NET Core 3.x Problem with IHttpResponseStreamFeature

A while back when having some discussions around Response filtering, Chris Ross from the ASP.NET Core team mentioned that it's better to use the new `IHttpResponseFeature` functionality instead of directly taking over the stream. At the time ASP.NET Core 3.x was still pre-release so I didn't bother at the time looking into it.

In ASP.NET Core 3.0 there have been a lot of under the cover changes on how Request and Response data is moved around using `Pipeline<T>` instead of `Stream`. The `IHttpXXXXFeature` interfaces and corresponding implementations are helping to abstract those new implementation details in abstracted interfaces and implementations that are don't have to take those differences into account.

It seems there's not any documentation on how these `IHttpFeature` work, other than in the ASP.NET Core source code. After some digging I ended up with the following code to use `IHttpResponseBodyFeature` which allows capturing the response data in a stream.

The code below for .NET Core 3.x uses this new 3.x only functionality to assign my custom stream which performs roughly the same functionality now as assigning `Response.Body` but presumably with better support for the new Response abstractions that Microsoft is using for output generation. (full code on GitHub)

```
private async Task HandleHtmlInjection(HttpContext context)
{
    // Use a custom StreamWrapper to rewrite output on Write/WriteAsync
    using (var filteredResponse = new ResponseStreamWrapper(context.Response.Body, context))
    {
#if !NETCORE2
        // Use new IHttpResponseBodyFeature for abstractions of pilelines/streams etc.
        // For 3.x this works reliably while direct Response.Body was causing random HTTP failures
        context.Features.Set<IHttpResponseBodyFeature>(new StreamResponseBodyFeature(filteredResponse));

        //var feature = context.Features.Get<IHttpResponseBodyFeature>();
#else
        context.Response.Body = filteredResponse;
#endif
        await _next(context);
    }
}
```

Because `IHttpResponseBodyFeature` is a new feature in ASP.NET Core 3.x I need the bracketed `#IF !NETCORE2` block to run the new code in 3.x and the old `Response.Body` assignment in 2.x.

To get that to work the Compiler constant has to be defined in the project:

```
<PropertyGroup Condition="'$(TargetFramework)' == 'netcoreapp2.1'">
    <DefineConstants>NETCORE2</DefineConstants>
</PropertyGroup>
```

After implementing this relatively simple code change - once you know what it is - **the weird random failures disappeared** and requests worked reliably again.

I'm happy it works, and it solves my immediate problem which is in a custom server interface for an old Legacy product. The occasional failures - even if only in Live Reload debug mode - were pretty damn annoying. With the change above the server is now reliable again with Live Reload enabled.

Yay! But shit, that's a weird bug...

Wrap it up: HTML Injection with Response Wrapping

For completeness' sake I'm going to describe the Response wrapping code and stream implementation that handles the HTML injection logic here.

In order to inject the Web Socket script into **any HTML output** that the application renders - static HTML, or Razor/MVC generated pages or views - I need to rewrite the `</body>` tag in the HTML output, and when I find it, inject the WebSocket script into the output.

To do this the only way I could find is to capture the Response stream and as part of that process the stream logic has to:

- Check to see if the output content is HTML
- If so force the Content Length to `null` (ie. auto-length)
- If so update the stream and inject the Web Socket script code
- If not HTML pass raw content straight through

This pretty much like what you had to do in classic ASP.NET with `Response.Filter` except here I have to explicitly take over the Response stream (or Http Feature) directly.

Here's what the relevant overridden methods in this stream class look like . I've left out the methods that just forward to the base stream leaving just the relevant methods that operate on checking and manipulating the Response (full code on Github):

```
public class ResponseStreamWrapper : Stream
{
    private Stream _baseStream;
    private HttpContext _context;
    private bool _isContentLengthSet = false;

    public ResponseStreamWrapper(Stream baseStream, HttpContext context)
    {
        _baseStream = baseStream;
        _context = context;
        CanWrite = true;
    }

    public override Task FlushAsync(CancellationToken cancellationToken)
    {
        // BUG Workaround: this is called at the beginning of a request in 3.x and so
        // we have to set the ContentLength here as the flush/write locks headers
        // Appears fixed in 3.1 but required for 3.0
        if (!_isContentLengthSet && IsHtmlResponse())
        {
            _context.Response.Headers.ContentLength = null;
            _isContentLengthSet = true;
        }

        return _baseStream.FlushAsync(cancellationToken);
    }

    ...

    public override void SetLength(long value)
    {
        _baseStream.SetLength(value);
        IsHtmlResponse(forceReCheck: true);
    }

    public override void Write(byte[] buffer, int offset, int count)
    {
        if (IsHtmlResponse())
        {
            WebsocketScriptInjectionHelper.InjectLiveReloadScriptAsync(buffer, offset, count, _context,
_baseStream)
```

```
_baseStream)
                                        .GetAwaiter()
                                        .GetResult();
        }
        else
            _baseStream.Write(buffer, offset, count);
    }

    public override async Task WriteAsync(byte[] buffer, int offset, int count,
                                        CancellationToken cancellationToken)
    {
        if (IsHtmlResponse())
        {
            await WebsocketScriptInjectionHelper.InjectLiveReloadScriptAsync(
                buffer, offset, count,
                _context, _baseStream);
        }
        else
            await _baseStream.WriteAsync(buffer, offset, count, cancellationToken);
    }


    private bool? _isHtmlResponse = null;
    private bool IsHtmlResponse(bool forceReCheck = false)
    {
        if (!forceReCheck && _isHtmlResponse != null)
            return _isHtmlResponse.Value;

        _isHtmlResponse =
            _context.Response.StatusCode == 200 &&
            _context.Response.ContentType != null &&
            _context.Response.ContentType.Contains("text/html", StringComparison.OrdinalIgnoreCase) &&
            (_context.Response.ContentType.Contains("utf-8", StringComparison.OrdinalIgnoreCase) ||
            !_context.Response.ContentType.Contains("charset=", StringComparison.OrdinalIgnoreCase));

        // Make sure we force dynamic content type since we're
        // rewriting the content - static content will set the header explicitly
        // and fail when it doesn't match if (_isHtmlResponse.Value)
        if (!_isContentLengthSet && _context.Response.ContentLength != null)
        {
            _context.Response.Headers.ContentLength = null;
            _isContentLengthSet = true;
        }

        return _isHtmlResponse.Value;
    }
}
```

There are a couple of things of note here.

Everything is forced through the Stream

This approach requires **that all content** - not just the HTML content - goes through this filtering stream because I have no other way to determine the Response `Content-Type` reliably to determine if the output is HTML.

The filter stream is pretty efficient as it passes through all stream methods to the base stream in the case of non-HTML content. It does have to check whether the content is HTML but that check only happens once and after that uses a cached value. Still, it seems that it would be much more efficient if there was a way to tell whethe the stream needs to be wrapped **before** creating a new wrapping stream.

Maybe there's a better way to do this which would make non-HTML content more efficient, but I couldn't find one.

No Header Access after first write in ASP.NET Core is Tricky

Another small problem is that in ASP.NET Core headers **cannot be modified once you start writing to the Response** stream. That makes sense in some scenarios (such as streaming data or dynamic data), but seems

infuriating for others when you know that ASP.NET has to still write the `Content-Length` anyway when it's done with content because the size of the content isn't known until the output has been completely rendered. So there's some sort of buffering happening - but your code doesn't get to participate in that unless you completely reset the Response.

Regardless, since this middleware injects additional script into the page, `Content-Length` **always** has to be set to `null` because even if the size was previously set, with the injected script the size is no longer accurate. So `Response.ContentLength=null` is still a requirement and it has to be set **before writing to the header**.

To make this scenario even worse, in ASP.NET Core 3.0 there was a bug that fired the stream's `FlushAsync()` method **before the first `Write` operation** when the initial Response stream was created. Arrgh! So the code also checks `FlushAsync()` for HTML content and resets the `Content-Length` there. That was a fun one to track down. . Luckily it **looks like that issues was fixed in ASP.NET Core 3.1.**.

The Actual Rewrite Code - Fun with Span

Just for completeness sake here's the code that rewrites the actual byte buffer as it comes in to output the content. It uses `Span<T>` to split the inbound buffer then writes the three buffers - pre, script, post - out into the stream:

```
public static Task InjectLiveReloadScriptAsync(
            byte[] buffer,
            int offset, int count,
            HttpContext context,
            Stream baseStream)
{
    Span<byte> currentBuffer = buffer;
    var curBuffer = currentBuffer.Slice(offset, count).ToArray();
    return InjectLiveReloadScriptAsync(curBuffer, context, baseStream);
}

public static async Task InjectLiveReloadScriptAsync(
        byte[] buffer,
        HttpContext context,
        Stream baseStream)
{
    var index = buffer.LastIndexOf(_markerBytes);

    if (index > -1)
    {
        await baseStream.WriteAsync(buffer, 0, buffer.Length);
        return;
    }

    index = buffer.LastIndexOf(_bodyBytes);
    if (index == -1)
    {
        await baseStream.WriteAsync(buffer, 0, buffer.Length);
        return;
    }

    var endIndex = index + _bodyBytes.Length;

    // Write pre-marker buffer
    await baseStream.WriteAsync(buffer, 0, index - 1);

    // Write the injected script
    var scriptBytes = Encoding.UTF8.GetBytes(GetWebSocketClientJavaScript(context));
    await baseStream.WriteAsync(scriptBytes, 0, scriptBytes.Length);

    // Write the rest of the buffer/HTML doc
    await baseStream.WriteAsync(buffer, endIndex, buffer.Length - endIndex);
}

static int LastIndexOf<T>(this T[] array, T[] sought) where T : IEquatable<T>
                        => array.AsSpan().LastIndexOf(sought);
```

Again the complete code including the dependencies that are not listed here are on Github in the WebSocketScriptInjectionHelper class. This code has all the logic needed to inject additional bytes into an existing byte array which is what's needed to rewrite the content from an individual (or complete) `Response.Write()` or `Response.WriteAsync()` operation.

Summary

The bottom line is that re-writing HTTP Response content is still a pain in the ass in ASP.NET Core. It still requires capturing the active Response stream and rewriting the content on the fly. You have to be careful to set your headers **before** the re-write and especially you have to ensure that if you change the content's size that the `Content-Length` gets dynamically set by ASP.NET internally by setting `context.Response.Headers.ContentLength = null;`.

It's not much different from what you had to do in classic ASP.NET, so I suppose this is par for the course. Hopefully walking through this scenario is useful to some of you heading down the same path of rewriting output. Just remember to use `IHttpResponseStreamFeature` instead of the `Response.Body` stream for more reliable output behavior.

Resources

- Westwind.AspNetCore.LiveReload on GitHub