# **JavaScript**

## **Reliably send HTTP request as user leaves page**

<https://css-tricks.com/send-an-http-request-on-page-exit/>

Consider the scenario where we need to log user’s action like navigating to a different page, we would do something like this –



There’s nothing terribly complicated going on here. The link is permitted to behave as it normally would (not using e.preventDefault()), but before that behaviour occurs, a POST request is triggered on click. There’s no need to wait for any sort of response. **We just want it to be sent** to whatever service we are hitting.

On first glance, we might expect the dispatch of that request to be synchronous, after which we’d continue navigating away from the page while some other server successfully handles that request. But as it turns out, that’s not what always happens.

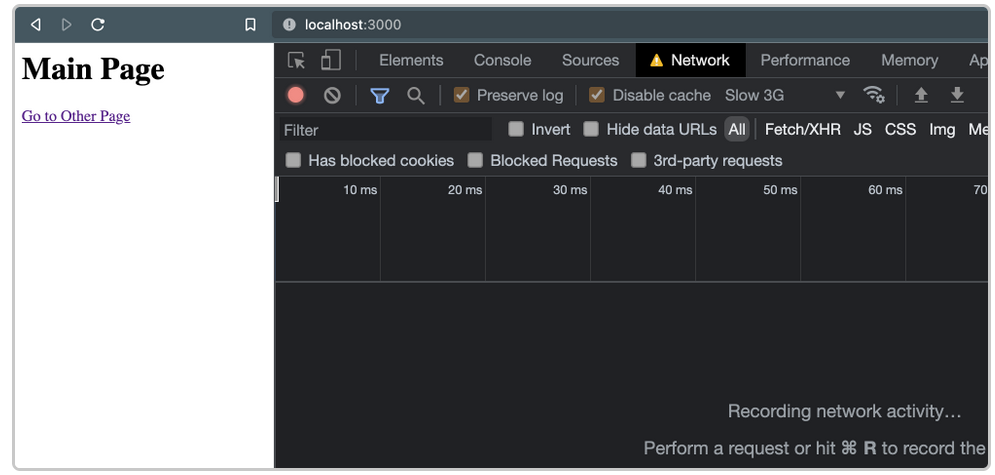
**Browsers don’t guarantee to preserve open HTTP requests**

When something occurs to terminate a page in the browser, there’s no guarantee that an in-process HTTP request will be successful ([see more](https://developers.google.com/web/updates/2018/07/page-lifecycle-api) about the “terminated” and other states of a page’s lifecycle). The reliability of those requests may depend on several things — network connection, application performance, and even the configuration of the external service itself.

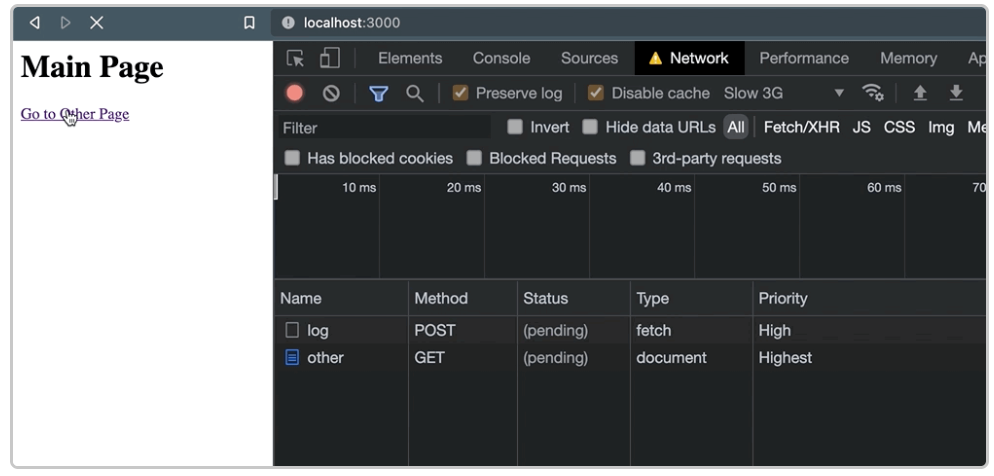
As a result, sending data at those moments can be anything but reliable, which presents a potentially significant problem if you’re relying on those logs to make data-sensitive business decisions.

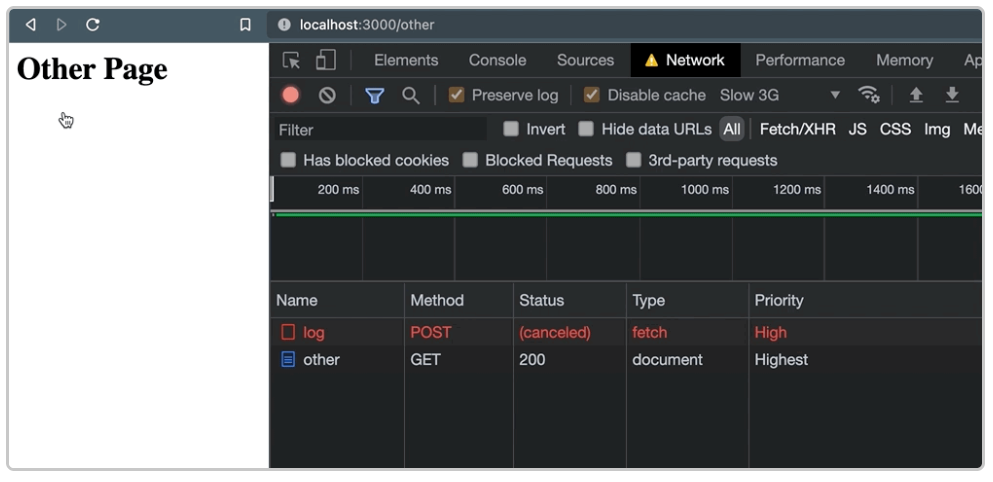
To help illustrate this unreliability, I set up a small Express application with a page using the code included above. When the link is clicked, the browser navigates to /other, but before that happens, a POST request is fired off.

While everything happens, I have the browser’s Network tab open, and I’m using a “Slow 3G” connection speed. Once the page loads and I’ve cleared the log out, things look pretty quiet-

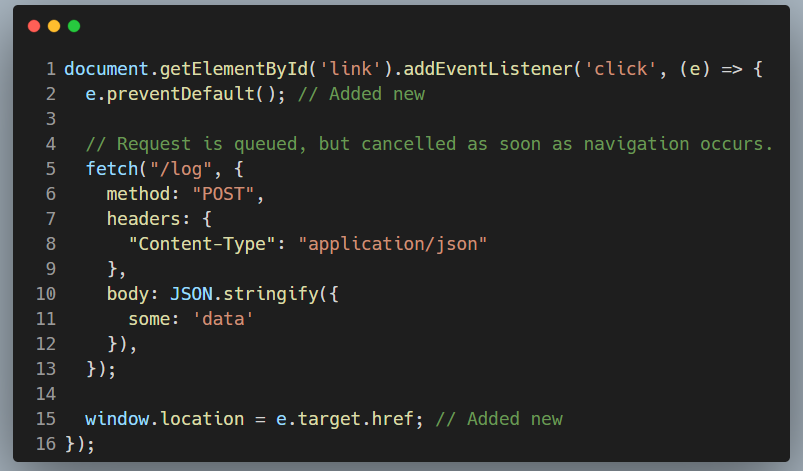


But as soon as the link is clicked, things go awry. When navigation occurs, the request is cancelled –





And that leaves us with little confidence that the external service was actually able process the request. Just to verify this behaviour, it also occurs when we navigate programmatically with window.location –

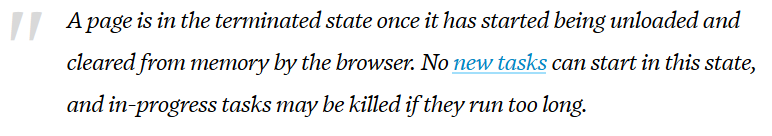


Regardless of how or when navigation occurs and the active page is terminated, those unfinished requests are at risk for being abandoned.

**But why are they cancelled?**

The root of the issue is that, by default, XHR requests (via fetch or XMLHttpRequest) are asynchronous and non-blocking. As soon as the request is queued, the actual *work* of the request is handed off to a browser-level API behind the scenes.

As it relates to performance, this is good — you don’t want requests hogging the main thread. But it also means there’s a risk of them being deserted when a page enters into that “terminated” state, leaving no guarantee that any of that behind-the-scenes work reaches completion. [Here’s how Google summarizes](https://developers.google.com/web/updates/2018/07/page-lifecycle-api#states) that specific lifecycle state:



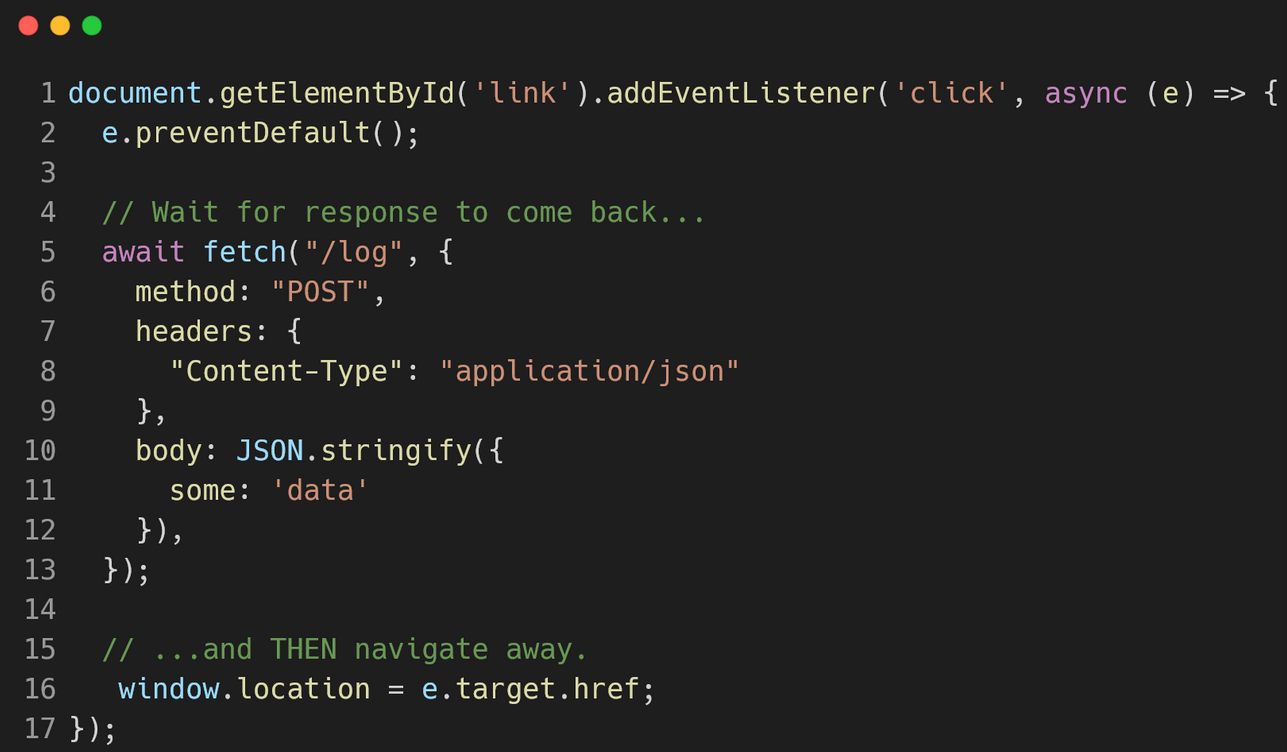
New tasks: <https://html.spec.whatwg.org/multipage/webappapis.html#queue-a-task>

In short, the browser is designed with the assumption that when a page is dismissed, there’s no need to continue to process any background processes queued by it.

**So, what are our options?**

Perhaps the most obvious approach to avoid this problem is, as much as possible, to delay the user action until the request returns a response. In the past, this has been done the wrong way by use of the [synchronous flag](https://xhr.spec.whatwg.org/#synchronous-flag) supported within XMLHttpRequest. But using it completely blocks the main thread, causing a host of performance issues so the idea shouldn’t even be entertained. In fact, it’s on its way out of the platform (Chrome v80+has already removed it <https://developers.google.com/web/updates/2019/12/chrome-80-deps-rems>).

Instead, if you’re going to take this type of approach, it’s better to wait for a Promise to resolve as a response is returned. After it’s back, you can safely perform the behaviour. Using our snippet from earlier, that might look something like this –



That gets the job done, but there are some non-trivial drawbacks.

**First, it compromises the user’s experience by delaying the desired behaviour from occurring.** Collecting analytics data certainly benefits the business (and hopefully future users), but it’s less than ideal to make your present users to pay the cost to realize those benefits. Not to mention, as an external dependency, any latency or other performance issues within the service itself will be surfaced to the user. If timeouts from your analytics service cause a customer from completing a high-value action, everyone loses.

**Second, this approach isn’t as reliable as it initially sounds, since some termination behaviours can’t be programmatically delayed.** For example, e.preventDefault() is useless in delaying someone from closing a browser tab. So, at best, it’ll cover collecting data for some user actions, but not enough to be able to trust it comprehensively.

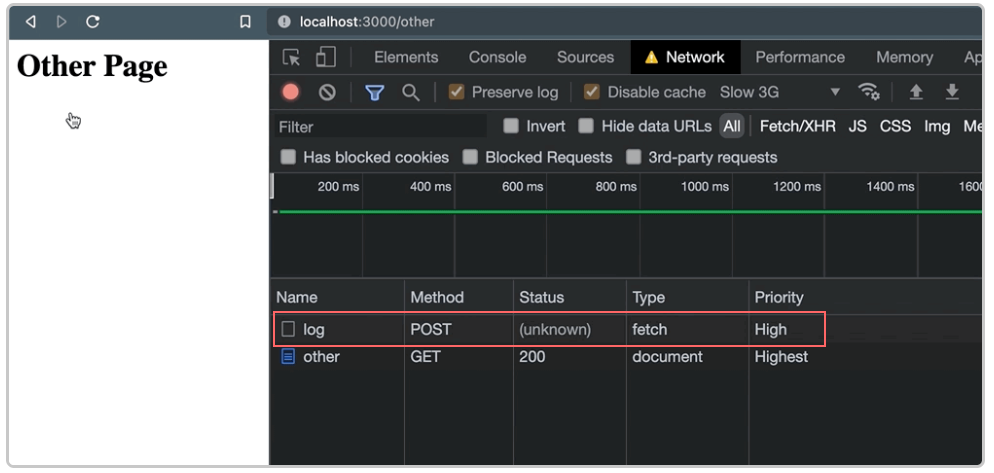
**Instructing the browsers to preserve outstanding requests**

Thankfully, there are options to *preserve* outstanding HTTP requests that are built into the vast majority of browsers, and that don’t require user experience to be compromised.

*Using Fetch’s* keepalive *flag*: <https://fetch.spec.whatwg.org/#request-keepalive-flag> If the keepalive flag is set as true when using fetch, the corresponding request will remain open, even if the page that initiated that request is terminated. Using our initial example, that’d make for an implementation that looks like this –



When that link is clicked and page navigation occurs, no request cancellation occurs –



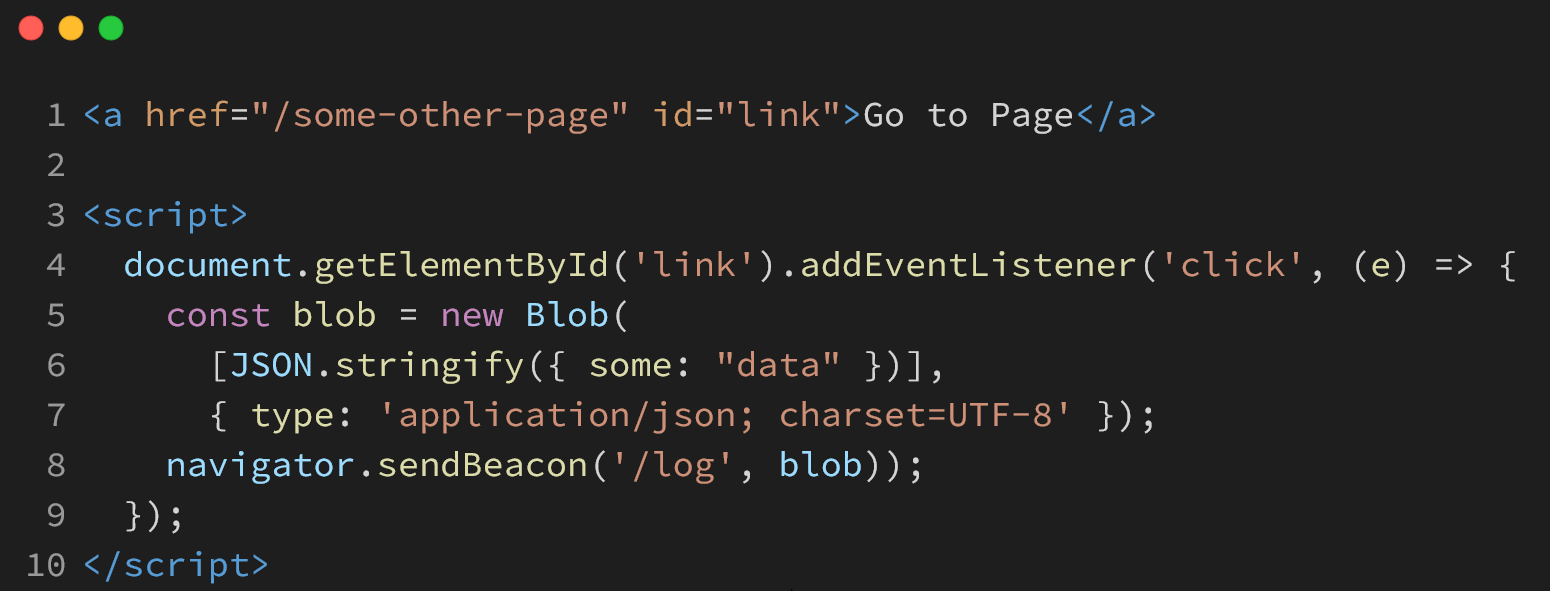
A one-liner like this an easy fix, especially when it’s part of a commonly used browser API. But if we’re looking for a more focused option with a simpler interface, there’s another way with virtually the same browser support –

*Using* Navigator.sendBeacon()

The Navigator.sendBeacon() function is specifically intended for sending one-way requests (beacons – <https://w3c.github.io/beacon/#sec-processing-model>). A basic implementation looks like this, sending a POST with stringified JSON and a “text/plain” Content-Type –

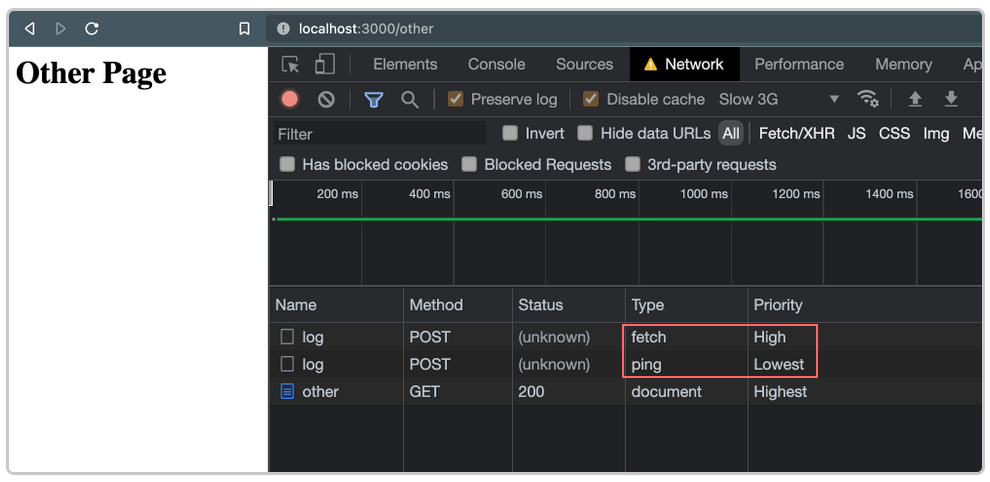


But this API doesn’t permit to send custom headers. So, in order for us to send our data as “application/json”, we’ll need to make a small tweak and use a Blob –

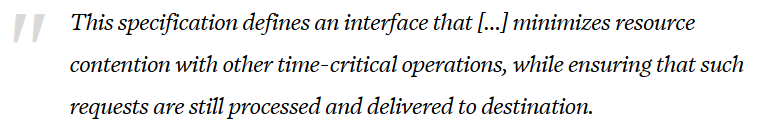


In the end, we get the same result — a request that’s allowed to complete even after page navigation. But there’s something more going on that may give it an edge over fetch(): beacons are sent with a low priority.

To demonstrate, here’s what’s shown in the Network tab when both fetch() with keepalive *and* sendBeacon() are used at the same time –



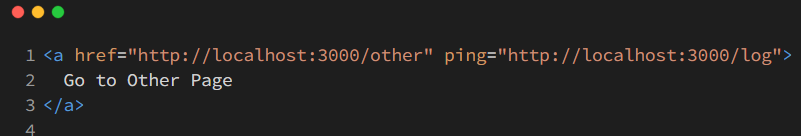
By default, fetch() gets a “High” priority, while the beacon (noted as the “ping” type above) have the “Lowest” priority. For requests that aren’t critical to the functionality of the page, this is a good thing. Taken straight from the Beacon specification (<https://www.w3.org/TR/beacon/>) –



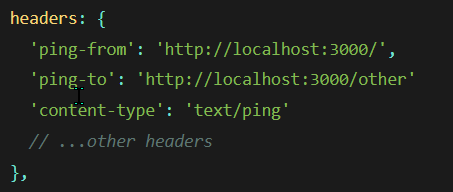
Put another way, sendBeacon() ensures its requests stay out of the way of those that really matter for our application and our user’s experience.

*An honourable mention for the* ping *attribute*:

It’s worth mentioning that a growing number of browsers support the ping attribute <https://css-tricks.com/the-ping-attribute-on-anchor-links/> When attached to links, it’ll fire off a small POST request –



And those requests headers will contain the page on which the link was clicked (ping-from), as well as the href value of that link (ping-to) –



It’s technically similar to sending a beacon, but has a few notable limitations –

1. **It’s strictly limited for use on links,** which makes it a non-starter if we need to track data associated with other interactions, like button clicks or form submissions.
2. **Browser support is good,** [**but not great**](https://caniuse.com/ping)**.** At this time, Firefox specifically doesn’t have it enabled by default.
3. **We’re unable to send any custom data along with the request.** As mentioned above, the most we’ll get is a couple of ping-\* headers, along with whatever other headers are along for the ride.

All things considered, ping is a good tool if you’re fine with sending simple requests and don’t want to write any custom JavaScript. But if you’re needing to send anything of more substance, it might not be the best thing to reach for.

**So, which one to choose?**

There are definitely tradeoffs to using either fetch with keepalive or sendBeacon() to send your last-second requests. To help discern which is the most appropriate for different circumstances, here are some things to consider –

We can go with fetch + keepalive if –

* We need to easily pass custom headers with the request.
* We want to make a GET request to a service, rather than a POST.
* We’re supporting older browsers (like IE) and already have a fetch polyfill being loaded.

But sendBeacon is a better choice if –

* We’re making simple service requests that don’t need much customization.
* We prefer the cleaner, more elegant API.
* We want to guarantee that our requests don’t compete with other high-priority requests being sent in the application.

Note: Service workers can be an alternative??