←

It's common for modern hosts to cache static assets in a flexible, but not most optimal way. Let's explore why that is and what we can do to push cache performance (for some assets) even further.

Sep 10, 2023 / 4,997

We've got it real good when it comes to standing up websites these days (especially static ones). Modern hosts like Vercel and Netlify take care of a *lot* out of the box, shielding us from the meticulous, complicated stuff.

Caching is one of them. To accommodate the widest range of users, many providers will cache static assets with a `Cache-Control` header of `public`, `max-age=0`, `must-revalidate`. Translation:

> ***Cache this thing, but immediately let it go stale and ask the origin server if there's a fresh copy to download next time around.***

It's a smart default. On each subsequent visit to a page, the browser will *always* check in with the origin server (or global CDN) for the latest version of the asset, but it'll only actually perform a full download *if those assets have changed*. If everything's the same, you'll get a `304 Not Modified` response back, and the browser's "stale" version will be used. Like this:

non-initial page views with default caching

Even though it's performing a distinct GET request to perform that check, if it returns with 304, it'll end up being far smaller & faster than full fetch with a 200. For example, here are a couple of actual requests for a font file from my site. The one that came back with a 304 had a significantly smaller footprint.

screenshot comparing the duration of a 304 and a 200 response

So, while it may sound counterintuitive to immediately let something cached go stale, it's a good balance between performance and flexibility. It'll save some headaches too. You won't end up with someone being served outdated CSS or a previous deployment's JavaScript bundle. Ship as much as you want, and your users will get the latest. If you'd like to dig more into this specific caching strategy, both and have some information on the philosophy behind it.

Even so, if you consider yourself a performance scrutineer, it should bother you that you're leaving some speed on the table by sticking with these defaults. Your caching *could be a little more aggressive*, and in my opinion, it's a no-brainer for particular types of assets.

Some Things Never Change

For a typical content site, most of the assets served via URL aren't dynamic. The same pile of CSS will be used for every visitor. Same story for fonts, individual images, and JavaScript bundles. Certain things just aren't designed to change based on who made the request or when it was performed, and it's technically wasteful to perform an extra HTTP request (albeit a small one) when you're likely to get back 304 anyway.

You can virtually eliminate those unnecessary requests by using a cache header like this instead.

```
public, max-age=31560000, immutable
```

Here's what it means:

- `public` - the asset can be stored in any cache between (and including) the browser and origin server
- `max-age=31560000` - the cache doesn't have to consider it "stale" until a full year has passed
- `immutable` - the browser is explicitly instructed to NOT reach out to origin/CDN just to check if something newer is available (no more revalidation requests)

With that policy in place, after a page has been visited for the first time, each asset is loaded straight from the cache, and the flow ends up looking more like this:

non-initial page views with smarter caching

As long as the browser's still got a cached copy of the asset (identified by unique URL), it'll be a full year before it ever checks origin again. That makes page performance marginally better, and you can feel a little better about yourself as an performance-minded engineer.

Nothing Lasts Forever

Ok. It's foolishly optimistic to say you'll never need to refresh assets before a year crawls by. You'll update a logo, refresh your site design, or swap out your fonts. It'll inevitably happen.

But serving fresh assets in those scenarios is a problem easy to solve with an age-old cache-busting tactic: **fingerprinting**. Every time an asset's URL changes (it gets a new "fingerprint"), it'll force the browser and any intermediary cache to treat it as *a completely different asset*. The URL serves as the cache key, and when it changes, it gets a new identity.

Most frameworks and site builders already do this for you out-of-the-box, by the way, so it's likely that you'll need to do nothing to benefit from it (at least for some asset types). For example, my site's on Astro. On every build, each static asset is given a very unique name:

And for the resources that aren't auto-fingerprinted, you'll get the same benefit using a different file name. For example:

```
- <img src="./logo.svg" alt="site logo" />
+ <img src="./logo-v2.svg" alt="site logo" />
```

All of this, by the way, is a good reason *not* to use overly aggressive caching on your page's HTML itself. The URL of your home page will likely never change, and so it's just not practical to cache it for a year with no revalidation. That's the advantage static assets have over an HTML document. The cache keys of static resources only matter to the HTML that references them. As long as the code's pointed to the correct versions, it doesn't matter what they're named or how frequently they're changed.

How do I do it?

Implementing this highly depends on how you're hosting your site, but before you do, get clear on which types of assets you'd like to cache more aggressively. For my own site, that's every `.js`, `.css`, and `.woff2` file (my images are already routed through , so I'm good there). That list is probably similar for you too.

Customizing Headers on Vercel

If you're on Vercel, you can update your `vercel.json` file to on the assets you target. I use this:

```
{
    "headers": [
        {
            "source": "/(.+\\.js|.+\\.css|.+\\.woff2)",
            "headers": [
                {
                    "key": "Cache-Control",
                    "value": "public, max-age=31560000, immutable"
                }
            ]
        }
    ]
}
```

Be careful writing that pattern, . Vercel follows the syntax – not `RegExp`. That's caused a moment or two of extreme frustration for me.

Customizing Headers on Netlify

Netlify has its own ways to using a `_headers` or `netlify.toml` file. Here's the same setup using a `netlify.toml`:

```
[[headers]]
  for = "/*.(css|js|woff2)"
  [headers.values]
  Cache-Control = "public, max-age=31536000, immutable"
```

Using Cloudflare

If you're on a provider that doesn't permit customizing response headers so easily, you're not out of luck. You can set up a Cloudflare account to act as a reverse proxy and set the response headers using a :

Or, if you'd like to do it in a more interesting way, intercept the request and modify the response with a Cloudflare Worker. Use something like and it'll amount to less than 30 lines of code:

```
import { Router, IRequest } from "itty-router";

const router = Router();

router.get("/*.(css|js|woff2)", async (request: IRequest) => {
  const response = await fetch(request);

    return new Response(response.body, {
        status: response.status,
        statusText: response.statusText,
        headers: {
            ...response.headers,
            "cache-control": "public, max-age=31560000, immutable",
        },
    });
});

export default {
    async fetch(
        request: Request,
        env: {},
        context: ExecutionContext
    ): Promise<Response> {
    context.passThroughOnException();
```

```
    return router.handle(request, env, context).then((response) => response);
    },
};
```

This, by the way, is the same approach used to cache every image optimized by . I love it.

Spend more time in the Network Tab.

The only reason I started thinking about this was because I was curious about the network activity behind any given page load on my own site. It struck me as too much, considering my site's fairly simple and doesn't have any ads or other network-chatty things. It was fun, I learned a lot, and my site came out a little quicker in the process.

Dive into those tools yourself, and maybe you'll emerge with a similar tip of your own. If you do, I'd love to hear it.

Alex MacArthur is a software developer working for Dave Ramsey in Nashville, TN. Soli Deo gloria.

Leave a Free Comment

Powered by JamComments

0 comments