web.dev SIGN IN

Prevent unnecessary network requests with the HTTP Cache

Nov 5, 2018 • Updated Apr 17, 2020
Appears in: Network reliability

Jeff Posnick
Twitter · GitHub · Glitch · Blog

Ilya Grigorik
Twitter · GitHub

Fetching resources over the network is both slow and expensive:

- Large responses require many roundtrips between the browser and the server.
- Your page won't load until all of its critical resources have downloaded completely.
- If a person is accessing your site with a limited mobile data plan, every unnecessary network request is a waste of their money.

How can you avoid unnecessary network requests? The browser's HTTP Cache is your first line of defense. It's not necessarily the most powerful or flexible approach, and you have limited control over the lifetime of cached responses, but it's effective, it's supported in all browsers, and it doesn't require much work.

This guide shows you the basics of an effective HTTP caching implementation.

There isn't actually a single API called the HTTP Cache. It's the general name for a collection of web platform APIs. Those APIs are supported in all browsers:

- `Cache-Control`
- `ETag`
- `Last-Modified`

How the HTTP Cache works

All HTTP requests that the browser makes are first routed to the browser cache to check whether there is a valid cached response that can be used to fulfill the request. If there's a match, the response is read from the cache, which eliminates both the network latency and the data costs that the transfer incurs.

The HTTP Cache's behavior is controlled by a combination of request headers and response headers. In an ideal scenario, you'll have control over both the code for your web application (which will determine the request headers) and your web server's configuration (which will determine the response headers).

Check out MDN's HTTP Caching article for a more in-depth conceptual overview.

Request headers: stick with the defaults (usually)

While there are a number of important headers that should be included in your v
app's outgoing requests, the browser almost always takes care of setting them c

browser's understanding of the current values in the HTTP Cache.

This is good news—it means that you can continue including tags like `<img src="my-image.png">` in your HTML, and the browser automatically takes care of HTTP caching for you, without extra effort.

★ Developers who do need more control over the HTTP Cache in their web application have an alternative—you can "drop down" a level, and manually use the Fetch API, passing it `Request` objects with specific `cache` overrides set. That's beyond the scope of this guide, though!

Response headers: configure your web server

The part of the HTTP caching setup that matters the most is the headers that your web server adds to each outgoing response. The following headers all factor into effective caching behavior:

- `Cache-Control`. The server can return a `Cache-Control` directive to specify how, and for how long, the browser and other intermediate caches should cache the individual response.
- `ETag`. When the browser finds an expired cached response, it can send a small token (usually a hash of the file's contents) to the server to check if the file has changed. If the server returns the same token, then the file is the s and there's no need to re-download it.
- `Last-Modified`. This header serves the same purpose as `ETag`, but use time-based strategy to determine if a resource has changed, as opposed t

Some web servers have built-in support for setting those headers by default, while others leave the headers out entirely unless you explicitly configure them. The specific details of *how* to configure headers varies greatly depending on which web server you use, and you should consult your server's documentation to get the most accurate details.

To save you some searching, here are instructions on configuring a few popular web servers:

- Express
- Apache
- nginx
- Firebase Hosting
- Netlify

Leaving out the `Cache-Control` response header does not disable HTTP caching! Instead, browsers effectively guess what type of caching behavior makes the most sense for a given type of content. Chances are you want more control than that offers, so take the time to configure your response headers.

Which response header values should you use?

There are two important scenarios that you should cover when configuring your server's response headers.

Long-lived caching for versioned URLs

Versioned URLs are a good practice because they make it easier to invalidate cached responses.

READ MORE

When responding to requests for URLs that contain "fingerprint" or versioning information, and whose contents are never meant to change, add `Cache-Control: max-age=31536000` to your responses.

Setting this value tells the browser that when it needs to load the same URL anytime over the next one year (31,536,000 seconds; the maximum supported value), it can immediately use the value in the HTTP Cache, without having to make a network request to your web server at all. That's great—you've immediately gained the reliability and speed that comes from avoiding the network!

Build tools like webpack can automate the process of assigning hash fingerprints to your asset URLs.

★ You can also add the `immutable` property to your `Cache-Control` header as a further optimization, though it will be ignored in some browsers.

Server revalidation for unversioned URLs

Unfortunately, not all of the URLs you load are versioned. Maybe you're not able t
include a build step prior to deploying your web app, so you can't add hashes t

use your web app if they need to remember that the URL to visit is `https://example.com/index.34def12.html`. So what can you do for those URLs?

This is one scenario in which you need to admit defeat. HTTP caching alone isn't powerful enough to avoid the network completely. (Don't worry—you'll soon learn about service workers, which will provide the support we need to swing the battle back in your favor.) But there are a few steps you can take to make sure that network requests are as quick and efficient as possible.

The following `Cache-Control` values can help you fine-tune where and how unversioned URLs are cached:

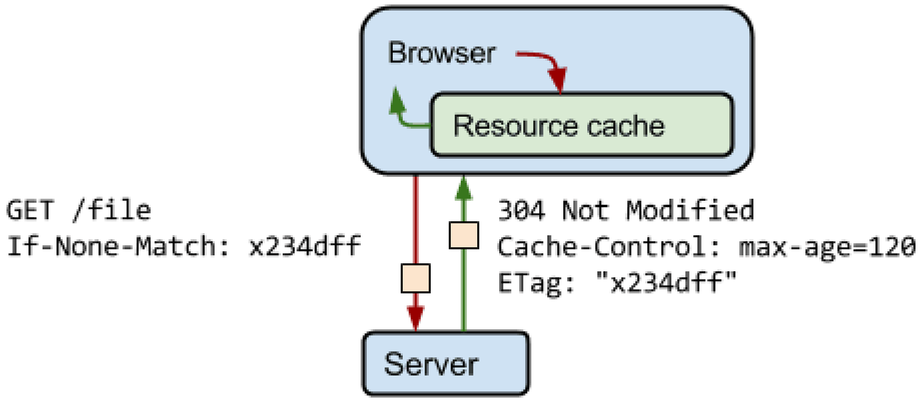
- `no-cache`. This instructs the browser that it must revalidate with the server every time before using a cached version of the URL.
- `no-store`. This instructs the browser and other intermediate caches (like CDNs) to never store any version of the file.
- `private`. Browsers can cache the file but intermediate caches cannot.
- `public`. The response can be stored by any cache.

Check out Appendix: `Cache-Control` flowchart to visualize the process of deciding which `Cache-Control` value(s) to use. Note also that `Cache-Control` can accept a comma-separated list of directives. See Appendix: `Cache-Control` examples.

Along with that, setting one of two additional response headers can also help: ei `ETag` or `Last-Modified`. As mentioned in Response headers, `ETag` and `Las Modified` both serve the same purpose: determining if the browser needs to re-download a cached file that has expired. `ETag` is the recommended approach because it's more accurate.

By setting `ETag` or `Last-Modified`, you'll end up making the revalidation request much more efficient. They end up triggering the `If-Modified-Since` or `If-None-Match` request headers that were mentioned in Request headers.

When a properly configured web server sees those incoming request headers, it can confirm whether the version of the resource that the browser already has in its HTTP Cache matches the latest version on the web server. If there's a match, then the server can respond with a `304 Not Modified` HTTP response, which is the equivalent of "Hey, keep using what you've already got!" There's very little data to transfer when sending this type of response, so it's usually much faster than having to actually send back a copy of the actual resource being requested.



The browser requests `/file` from the server and includes the `If-None-Match` header to instruct the server to only return the full file if the `ETag` of the file on the server doesn't match the browser's `If-None-Ma` value. In this case, the 2 values did match, so the browser returns a `304 Not Modified` response with instructions on how much longer the file should be cached (`Cache-Control: max-age=120`).

Summary

much work to set up.

The following `Cache-Control` configurations are a good start:

- `Cache-Control: no-cache` for resources that should be revalidated with the server before every use.
- `Cache-Control: no-store` for resources that should never be cached.
- `Cache-Control: max-age=31536000` for versioned resources.

And the `ETag` or `Last-Modified` header can help you revalidate expired cache resources more efficiently.

Try it! Try the HTTP Cache codelab to get hands-on experience with `Cache-Control` and `ETag` in Express.

Learn more

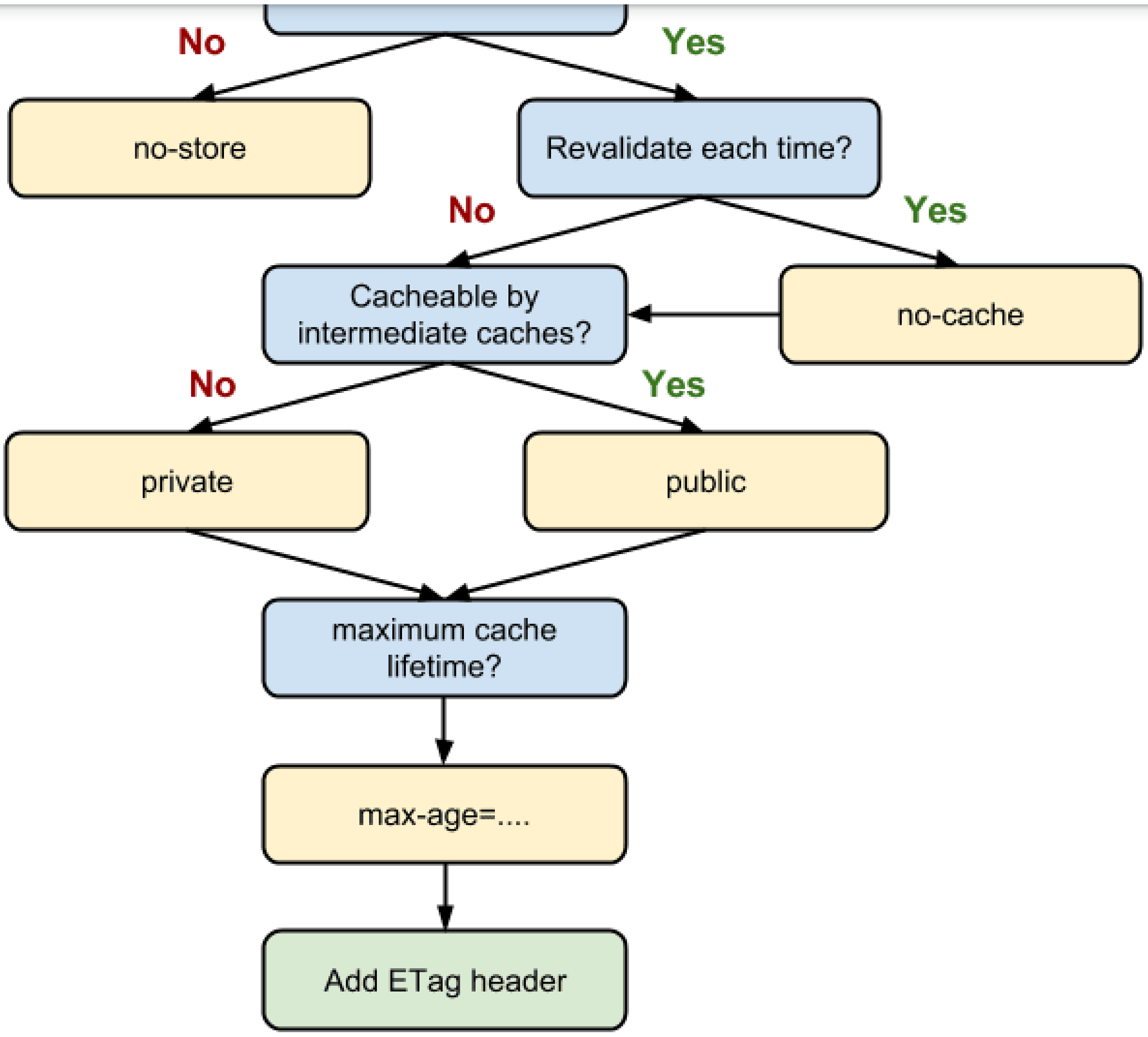
If you're looking to go beyond the basics of using the `Cache-Control` header, check out Jake Archibald's Caching best practices & max-age gotchas guide.

Appendix: More tips

If you have more time, here are further ways that you can optimize your usage of the HTTP Cache:

- Minimize churn. If part of a resource (such as a CSS file) updates frequently, whereas the rest of the file does not (such as library code), consider splitting the frequently updating code into a separate file and using a short duration caching strategy for the frequently updating code and a long caching duration strategy for the code that doesn't change often.
- Check out the new `stale-while-revalidate` directive if some degree of staleness is acceptable in your `Cache-Control` policy.

Appendix: `Cache-Control` flowchart



Appendix: `Cache-Control` examples

`Cache-Control` **value**	**Explanation**
`max-age=86400`	The response can be cached by browsers and intermediary caches for up to 1 day (60 seconds x 60 minutes x 24 hours).
`private, max-age=600`	The response can be cached by the browser (but not intermediary caches) for up to 10 minutes (60 seconds x 10 minutes).

`public, max-age=31536000`	The response can be stored by any cache for 1 year.
`no-store`	The response is not allowed to be cached and must be fetched in full on every request.

Last updated: Apr 17, 2020 Improve article

Codelabs

See it in action

Learn more and put this guide into action. Configuring HTTP caching behavior

Give feedback

All fields optional

	Yes	No
Was this page helpful?	○	○
Did this page help you complete your goal(s)?	○	○
Did this page have the information you needed?	○	○
Was this page's information accurate?	○	○
Was this page easy to read?	○	○

web.dev SIGN IN

← Return to all articles

Contribute

File a bug

View source

Related content

Updates

Web Fundamentals

Case studies

DevWeb Content Firehose

Podcasts

Connect

Twitter

web.dev SIGN IN

Medium

Google Developers

Chrome

Firebase

Google Cloud Platform

All products

ENGLISH (en)

Terms & Privacy

Community Guidelines