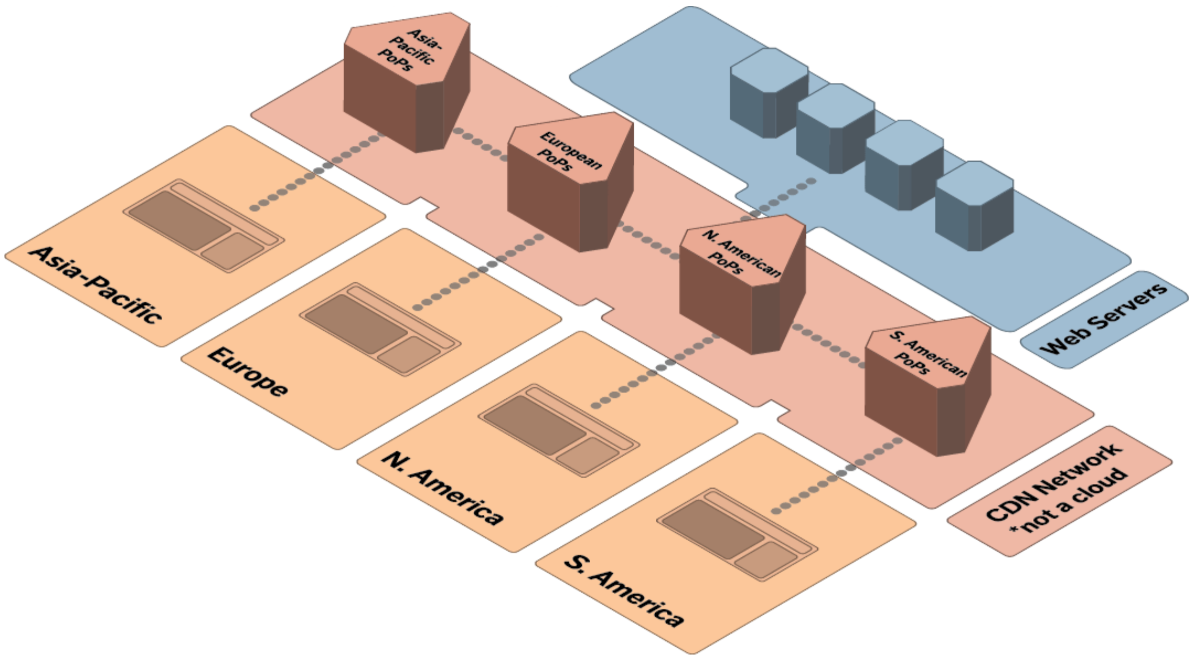
A Beginner's Guide to HTTP Cache Headers

by dofrance | May 12, 2014 | Information Technology | 0 comments



This article offers an exploration into HTTP caching headers and associated CDN behaviour. If you're looking to understand how caching headers fit into the modern web, or are just curious about what everyone is talking about around you, this resource is for you.

If you already understand the benefits of cache headers and are looking to extend your knowledge further, I'd recommend referring to the W3 documentation.

What Can Cache Headers Do for You?

Simply put, caching allows you to store your web assets on remote points along the way to your visitors' browsers. Of course the browser itself also maintains an aggressive cache, which keeps clients from having to continually ask your server for a resource each time it comes up.

Cache configuration for your web traffic is critical for any performant site. If you pay for bandwidth, make revenue from an e-commerce site, or even just like keeping your reputation as a web-literate developer intact, you need to know what caching gets you and how to set it up.

In the case of assets, things like your company logo, the favicon for your site, or your core CSS files aren't likely to change from request to request, so it is safe to tell the requester to hold onto their copy of the asset for a while. If your visitors were small children in the backseat asking, "Are we there yet?" repeatedly, this would be like telling them, "No, and we won't be for 20 more minutes, so remember my answer."

By cutting down on the requests your server has to deal with, you are able to handle more requests, and your users will enjoy a faster browsing experience. Generally, assets like images, JavaScript files, and style-sheets can all be cached fairly heavily, while assets that are dynamically generated, like dashboards, forums, or many types of web-applications, benefit less, if at all. If your concern is performance, your dynamic content will be shunted to a bare minimum of AJAX-type resources, while the rest of your assets will be heavily cached.

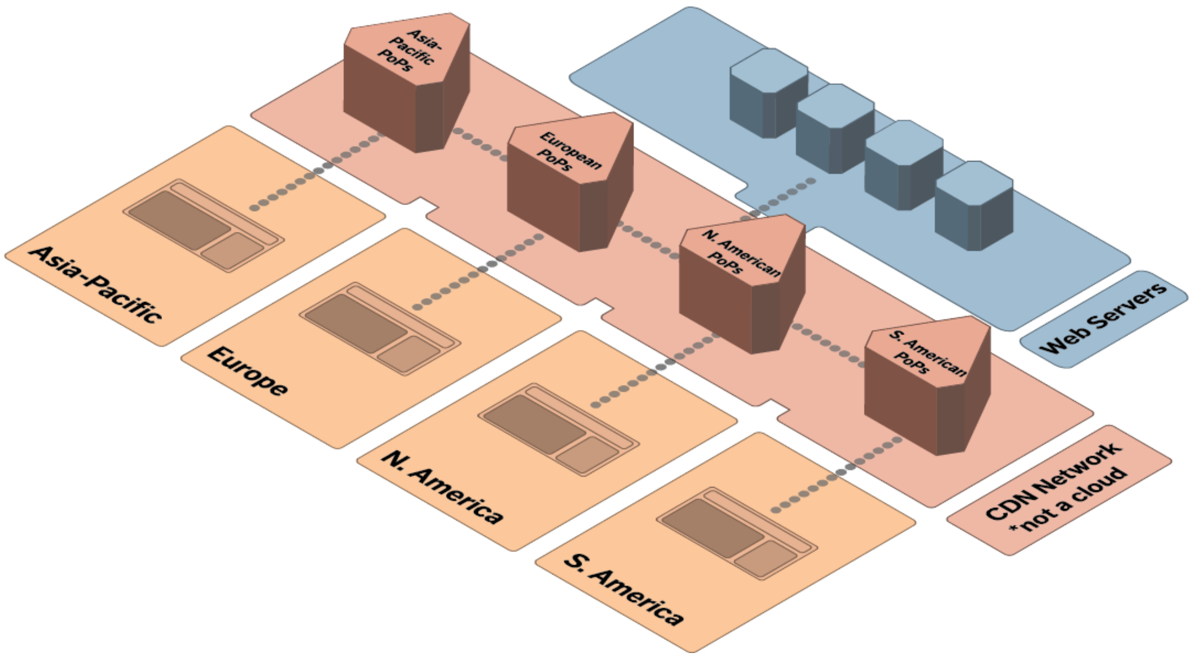
For Clients & CDNs

Historically, cache settings were all about the client's browser, so we shouldn't forget the benefits of properly considering the client and the way it keeps assets around during a session and on return visits. However, these days, with the advent of Content Delivery Networks (CDNs), a bigger concern is how caching is handled on the intermediary points of web traffic.

> *What are CDNs?*
>
> *In case you don't want to read the wiki article , I'll give you the crib notes here. Essentially, CDNs are servers, (note the plural), that sit between your end user and your server. Each of these servers are designed to cache your*

> *content according to the cache rules you set in the various HTTP headers outlined below.*

When configured properly, CDNs will deliver content to your end user from the fastest, (and typically closest) server available. Additionally, CDNs act as a buffer between you and your users. The number we are most concerned with is the cache hit ratio, which describes the percentage of requests the CDN was able to answer out of its own cache without having to bother our servers. Depending on your traffic and architecture, this number can get well into the high nineties, although even at lower figures you'll experience a gain. (Keep in mind that a low frequency of requests will mean a larger percentage of requests go back to your server, so the ratio is only meaningful when understood together with your cache times and the general load of your site.) A word of warning though; if you just setup your cache and fail to configure your caching headers correctly, it's quite possible to end up paying twice for every request.



A basic flow of data through a CDN. Your webservers provide content to the CDN intermediary servers, which have Points of Presence in various geographic regions. Lazy network diagram makers will usually just draw a cloud here.

Beyond their intended use for caching, CDNs also have a pleasant side-effect; provided you're dealing with a website, or a particularly well-crafted web application, in the event that your servers experience a momentary outage, your CDN may be able to buffer the experience for your end users, ensuring they never even notice.

The Basic Headers

So, you know what a CDN is, and you know you want to configure your response headers to make use of them. Most web servers make it trivial to set response headers, so I'll leave it to you, Google, and the man pages to figure that part out. For now, let's move onto what headers you should care about.

1. cache-control

If there were a default super-header for caching behaviour, this would be it. Typically you will see a string of settings for this header along the lines of:

```
cache-control: private, max-age=0, no-cache
```

These settings are called cache response directives, and are as follows:

```
private | public
```

Essentially they let intermediary caches know that a given response is specific to the end user and should not be cached. Do not make the mistake of assuming that this in any way provides you with some kind of security or privacy: Keep using SSL for that.

```
no-cache
```

When used alone, this guy lets you specify that caches should revalidate this resource every time, typically using the “etag” header outlined below. The fun bit comes when you specify a field name after the no-cache directive, which lets caches know that you can cache the response, provided that the named fields are stripped out; cookies would be a good example of why you might want to do this. I should note that older User Agents won't respect this trick, so you shouldn't depend on it too much.

```
no-store
```

This guy lets you specify that caches should not store this response. I know that may be surprising given the name, but there it is. Actually, if the cache in question is following the rules, it will also ensure that no part of the request is stored either. “no-store” was designed with sensitive information requirements in mind, and so is kind of like the G-Man of cache headers.

```
max-age
```

Traditionally, you would let caches know when an asset is expired by using the aptly-named “expires” header, discussed below. However, if you want to be more explicit, you may set a max-age, in seconds, which will override the expires header. Further reasons to use this directive are discussed below under the Caveats section.

```
s-maxage
```

Using our keen deductive skills, we can see some similarities between this header and the last one. “s-” is for shared, as in “shared cache”, as in CDN. These directives are explicitly for CDNs and other intermediary caches. When present, this directive overrides both the max-age and expires header, and most well behaved CDNs will obey it.

```
must-revalidate
```

This one is fun, although not applicable to most of us. For the sake of completeness, and in case your dev-team has some kind of trivia night where free beer is involved, we’ll discuss it. Essentially, if your responses include this directive, you are telling the cache that it needs to revalidate a cached asset on any subsequent request, and that it may not, under any circumstance, serve stale content (which is sometimes a desired behaviour). Of course, I say “under any circumstances”, but what I really mean is that there’s a big fat asterisk next to that claim. If your users are under “severe connectivity constraints”, (perhaps they are browsing from low-earth-orbit), then their user agents may serve stale content, provided they pinky-swear to tell their users that they’ve done so. Apparently this directive exists because some protocols require it, typically involving transactions.

```
no-transform
```

“Transform into what?”, you’re surely asking. Some proxies will convert image formats and other documents to improve performance. Presumably this was thought to be a feature that you should have to opt out of. If you don’t like the idea of your CDN making automated guesses about how your content should be encoded or formatted, I suggest including this header.

```
proxy-revalidate
```

Essentially the same as the “must-revalidate” directive, except it’s just for the shared caches. Why didn’t they call it “s-mustrevalidate”? I’m sure there exists a mailing list somewhere where you could find that debate, but for now, just know that like “s-maxage”, this directive is designed for intermediary proxies and not user agents. The idea here is that you validate each end-user only once between the proxy and their agent, but each new user should revalidate back to the server. I suspect if your service requires this feature, you probably already know about it.

As always, you should check out the spec for these settings if you need any clarification.

2. expires

Back in the day, this was the standard way to specify when an asset expired, and is just a basic date-time stamp. It is still fairly useful for older user agents, which cryptowebologists assure us still roam in the uncharted territories. On most modern systems, the “cache-control” headers “max-age” and “s-maxage” will take precedence, but it’s always good practice to set a matching value here for compatibility. Just make sure you format the date correctly, or it will be evaluated as an expired date:

```
Thu, 01 Dec 1983 20:00:00 GMT
```

Try to avoid setting the value to more than one year out as that breaks the specification, (see below for a discussion on cache time settings).

3. etag

Short for “entity-tag”, the etag is a unique identifier for the resource being requested, typically comprised of the hash of that resource, or a hash of the timestamp the resource was updated. Basically, this lets a client ask smarter questions of the CDNs, like “give me X if it’s different than the etag I already have.”

There’s a neat trick you can do with etags, which is to make them weak validators. This basically tells the user that although they are not the same, the two resources are functionally equivalent. Support for this feature is considered optional though, so you will want to do some testing with your providers, (discussed below).

4. vary

Oh wow. This one is fun. The “vary” header is extremely powerful, but can trip up what would otherwise be a simple caching scheme. IE has had issues handling the “vary” header in the past, and at one point, even Chrome was handling this in a funny way. Essentially, “vary” lets the caches know which of the headers to use to figure out if they have a valid cache for a request; if a cache were a giant key-value store, adding “vary” fields appends those values to the key, thus changing which requests are considered valid matches for what exists in the cache.

You would commonly set this to something like “Accept-Encoding” to make sure your gzip’ed assets get served where appropriate, saving you all that bandwidth you might otherwise waste. Additionally, setting:

```
vary: User-Agent
```

will put you in the SEO good-books if you happen to be serving different versions of your HTML/CSS depending on the User-Agent of the request. Google will note the header and have the Googlebot crawl your mobile content as well.

5. pragma

Another beast from the days of yore, the “pragma” header does many things, and most of them are honoured by newer implementations. The directive we’re most concerned with is:

```
pragma: no-cache
```

which gets interpreted by newer implementations as:

```
cache-control: no-cache
```

I would not generally recommend worrying about it, but for the sake of completeness, there it is. No new HTTP directives will be defined for “pragma” going forward.

Some Caveats

Now that we’ve gotten some of the standard and expected behaviours out of the way, we should probably mention that not every CDN or User Agent will behave according to the specification, which I’m sure is not news to anyone familiar with browser compatibility issues. For this reason, it is a good idea to test your services before launching a live resource to make sure the behaviour you get is what you expected; it will save you money.

Also, you may have noted that a lot of the headers seem to be either duplicated or overlapping. Some of this is because there are subtle differences between the different methods, and also because the web is shifting over from HTTP/1.0 to HTTP/1.1, which uses the “cache-control” much more heavily. Generally it is safe to set both and let the CDNs and User Agents figure out which one they care to listen to.

1. Compression

Remember that “Accept-Encoding” example for the “vary” header we talked about? That’s your new best friend if you intend to serve compressed content, which I hope you do to increase performance and save on bandwidth. CDN providers that get a request with “gzip” as an accepted encoding are expected to request the compressed asset from the origin server, or to serve a cached version of that compressed asset. Historically, this has been a sticking point for some CDNs, and for people wishing to use something like S3 to serve their files, although modern CDNs are able to perform the compression operation themselves if need be.

Things to watch out for when the CDN serves compressed assets is that they'll often ensure that both uncompressed (AKA identity) and gzip'ed version are in their cache, regardless of which was requested. There is a time delay as they perform that operation, so any testing you do will have to take that delay into account.

2. SSL

A CDN is basically a Man-In-The-Middle, meaning you need to think about your HTTPS traffic and how it gets back to your server. Many CDNs will pipe a request forhttps://somesite.com/asset to your servers as http://somesite.com/asset, so if your server logic depended on that being SSL, either reconsider your logic or ask your CDN to redirect to an HTTPS origin URL. Most CDNs are capable of doing so, with varying degrees of flexibility.

But What about Dynamic Content?

Generally the rule of thumb for dynamic content, like the HTML files of a WordPress blog, is to set "cache-control: no-cache" and prevent the CDNs or User Agents from storing the asset. For most applications, serving your own dynamic content is probably fine, however if you need to boost performance, read on.

Typical Dynamic Content

```
HTTP/1.1 200 OK
Server: Apache
X-Rack-Cache: miss
ETag: "e6811cdbcedf972c5e8105a89f637d39-gzip"
Status: 200
Content-Type: text/html; charset=utf-8
Expires: Mon, 29 Apr 2013 21:44:55 GMT
Cache-Control: max-age=0, no-cache, no-store
Pragma: no-cache
Date: Mon, 29 Apr 2013 21:44:55 GMT
```

An example of a section of a header for dynamic content. Note the missed cache status, and the cache-control directives all set to 0 seconds.

Most dynamic content has a shelf-life and is not nearly as volatile as we assume. Things like “Active Users” are probably valid for 10 or 20 seconds, depending on your site. Dashboards that report daily figures could certainly stand to be cached for a few minutes. News feeds could certainly stand to be cached for a while, especially if you remember to set an “etag”. If your site is seeing lots of load, it may be worth trying out a short cache on your dynamic assets.

An Analysis of Cache Time

So what types of cache times should you consider setting? Again, this will require you to consider things like the rate of traffic to your site, how sizeable your assets are, and how large your cache is, (yes, there are space allocations, so don’t go over them).

Additionally, you should consider the main tradeoff: speed and performance vs. control. If you want to update an asset and need the new version to be live immediately, you may run into trouble if you thought a one-year cache time was a good idea, especially if you set that for Users (max-age) and not just CDNs (s-maxage).

The longest you can set your headers and still be following the specification is one year, or 31536000 seconds, but this is *not a very good idea*. That’s like getting a face tattoo; it stays around forever, barring expensive or painful removal surgery, which accurately describes the annoyances of having to trick caches into refetching an asset through name changes or hoping you remembered to correctly set your etag and that your users and CDNs implemented them correctly. If your servers can’t stand to have your CDNs ask once a day if your profile picture has changed you should upgrade your servers.

Sane Static Content Headers

```
HTTP/1.1 200 OK
Cache-Control: no-transform,public,max-age=300,s-maxage=900
Content-Type: text/html; charset=UTF-8
Date: Mon, 29 Apr 2013 16:38:15 GMT
ETag: "bbea5db7e1785119a7f94fdd504c546e"
Last-Modified: Sat, 27 Apr 2013 00:44:54 GMT
Server: AmazonS3
Vary: Accept-Encoding
X-Cache: HIT
```

A sample of cache settings for a static asset served from S3. Here we see that the cache has been asked to store the asset for 900 seconds rather than the 300 seconds set for User Agents. Also note the “x-cache: HIT” header, indicating the CDN served the request.

There is one exception to the “thou-shalt-not-set-one-year-headers” commandment, or more accurately, there’s a bit of a hack to get around their pitfalls: if you’ve configured your site to generate resource names, you can rename your assets each time you publish a new version. Typically, this will involve an incrementing version number, a date-time stamp, or a hash of the contents, much like an “etag”, being appended to the file name, so that you end up with things like “core.E89A123FFDB…432D687B.css” and the like. It’s not pretty, but really, who cares? Also, this lets you set one-year-headers without worrying about updating your resources.

A handy table best explains the cache time trade-off. Assuming a web asset get 500 requests per minute, then the following Hit Ratios are possible for each cache time:

Cache time (minutes)	Hit Ratio	Request to Origin / Hr
1	99.8%	60
5	99.96%	12
20	99.99%	3
60	99.997%	1
86400	99.9998%	<1

There. It’s spelt out. What kind of Hit Ratio do you need? Typically 60 seconds to an hour is an ideal trade-off. For pseudo-dynamic content, you can still use CDNs, just start working in the under-60-seconds range, as appropriate for the request.

Testing Your CDN

So, onto the testing. Always check that the headers are coming through CDNs the way you expect when setting up new services. Typically a CDN will insert some kind of “x-” header to indicate that it hit, missed, or served expired content. What we’re looking for is a convenient way to look at this and other headers, and to set them on our requests for testing. There are more than a few tools out there to help, though these are the ones I’ve found most handy.

1. Web Inspector

Probably the most accessible method, simply right-click on a webpage in Chrome, click Inspect Element, navigate to the Network tab, hit refresh, and click on the HTML asset at the top. If it’s not selected by default, checkout the Headers tab to see all your request and response headers. Chrome also lets you set the user agent you request with, and gives you the option to not use your local cache, which is handy when you’re testing.

2. Charles Proxy et al.

Tools like Charles Proxy will let you route traffic through them and manipulate DNS lookups, record headers, and see performance statistics, all in a clean, easy-to-use GUI. These tools are generally more task-specific than the Web Inspector and other browser tools, and generally worth the small licensing fee if you work with web requests in any significant way.

3. cURL

Quick, easy, and highly flexible, cURL lets you make web requests directly from the command line and check out the responding headers. Helpful flags include *-A* for user agents, *-b* for cookies, *-F* for form data, *-H* for setting headers, and *-I* to request the header only. Very handy for sanity checks, and extremely powerful.

4. hurl.it

hurl.it is essentially cURL with a prettier interface, and in a webbrowser. You’re able to set the headers you pass in to the request, as well as view the full header and body response. Although you don’t quite get the robustness of the command line, it’s still very useful for a quick header examination, and is probably the fastest way to get access to headers if you’re in a hurry.

5. Python and Requests

Requests is a great Python package for making web requests, and is useful for quick checks from the REPL, or from slightly more complex scripted checks. Using Requests comes with the added benefit of letting you write a test-suite for your web assets to assert the status of response headers.

Notes from the Field and Parting Thoughts

So now you’ve made it all the way through our quick guide to cache headers, and you’re eager to get going on setting your own. Excellent. However, now it’s time to put some of this in perspective.

Most web servers, like Apache and Nginx, will do a lot of this work for you. Really, the “cache-control” header is the only one you’re going to have to work with. Web browsers are typically set up to cache aggressively to improve user experience, so often you’ll be fighting to prevent caching rather than ensuring it. Generally this means that you set a “/static” path and configure its caching headers to allow caching up to some reasonable interval like 300 seconds. Next, you would ensure that your root path “/” has “cache-control: no-cache” enabled, or better yet route dynamic content straight to your servers and only have “/static” use your CDN. This is a healthy starting point for most purposes. If your hosting bill is astronomical, or you get to use operations cost savings as beer money, then consider tweaking your settings.

CDNs will often give you the option of breaking protocol and will cache for whatever period you like, regardless of the headers. They also take liberties with the protocols depending on their interpretation of expected behaviour, which is why it’s important to test the headers you get out of your CDN and compare them to your servers. Consider the expected behaviour outlined here to be a baseline and keep an eye out for variance.

Good luck, and happy caching!

http://www.mobify.com/blog/beginners-guide-to-http-cache-headers/