**HTTP Cache Headers**

There are two primary **cache Response headers**, Cache-Control and Expires

### [**Cache-Control**](https://devcenter.heroku.com/articles/increasing-application-performance-with-http-cache-headers#cache-control)

With this header in place, and set with a value **that enables caching**, the browser will cache the file for as long as specified. Without this header the browser will re-request the file on each subsequent request.

**public resources** can be cached not only by the **end-user’s browser** but also by any **intermediate proxies** that may be serving many other users as well.

**Cache-Control:public**

**private resources** are bypassed by intermediate proxies and can only be cached by the **end-client**.

**Cache-Control:private**

The value of the Cache-Control header is a composite one, indicating whether the resource is **public or private** while also indicating the **maximum amount of time** it can be cached before considered stale.

The **max-age** value sets a timespan for how long to cache the resource (in seconds).

Cache-Control:public, **max-age**=31536000

While the Cache-Control header turns on client-side caching and sets the max-age of a resource the **Expires** header is used to specify a specific point in time the resource is no longer valid.

1. [**Expires**](https://devcenter.heroku.com/articles/increasing-application-performance-with-http-cache-headers#expires)

When accompanying the Cache-Control header, Expires simply sets a date from which the cached resource should no longer be considered valid. From this date forward the browser will request a fresh copy of the resource. Until then, the browsers local cached copy will be used:

**If both Expires and max-age are set** **max-age** will take **precedence**.

Cache-Control:public

Expires: Mon, 25 Jun 2012 21:31:12 GMT

While Cache-Control and Expires tells the browser ***when*** to next retrieve the resource from the network a few additional headers specify ***how***to retrieve the resource from the network. These types of requests are known as **conditional requests**.

1. **Last-Modified**

Last-Modified: Mon, 03 Jan 2011 17:45:57 GMT

1. **Etag**

ETag: "15f0fff99ed5aae4edffdd6496d7131f"

**HTTP Conditional Request**

* **If-Modified-Since**
* **If-Non-Match**
* **HTTP status of 304 (not modified)**

Conditional requests are those where the browser can ask the server if it has an updated copy of the resource. The browser will send some information about the cached resource it holds and the server will determine whether updated content should be returned or the browser’s copy is the most recent. In the case of the latter an **HTTP status of 304 (not modified) is returned**.

Conditional requests are used:

* we can prevent a client from having to download bytes that it already has.
* we can prevent a client from making changes to a resource that has been changed by someone else since they retrieved it

In order to make a conditional HTTP request, a special type of HTTP header called a **validator** needs to be sent in the response from the server.

The **two validators** defined by the HTTP specification are

**Etag** and **Last-Modified**.   Validators are values that allow you to compare if two resource representations are the same. The definition of "same" depends on the flavor of validator and the opinion of the origin server.

Validator values are considered either [strong or weak](http://tools.ietf.org/html/rfc7232#section-2.1).  Strong validators tell you if the response body is identical. Weak validators are used to allow an origin server to group multiple slightly different representations together as equivalent.

**Last Modified Header**

The Last Modifies header field indicate the date and time at which origin server believe your file was last modified.  
**Last-Modified**: Tue, 15 Nov 1994 12:45:26 GMT

### [**Time-based**](https://devcenter.heroku.com/articles/increasing-application-performance-with-http-cache-headers#time-based)

A time-based conditional request ensures that only if the requested resource has changed since the browser’s copy was cached will the contents be transferred. If the cached copy is the most up-to-date then the server returns the 304 response code.

To enable conditional requests the application specifies the last modified time of a resource via the Last-Modified response header.

Cache-Control:public, max-age=31536000

Last-Modified: Mon, 03 Jan 2011 17:45:57 GMT

The next time the browser requests this resource it will only ask for the contents of the resource if they’re unchanged since this date using the If-Modified-Since request header

**If-Modified-Since**: Mon, 03 Jan 2011 17:45:57 GMT

**If the resource hasn’t changed since Mon, 03 Jan 2011 17:45:57 GMT the server will return with an empty body with the 304 response code**.

## Etag

### [**Content-based**](https://devcenter.heroku.com/articles/increasing-application-performance-with-http-cache-headers#content-based)

The ETag (or Entity Tag) works in a similar way to the Last-Modified header except its value is a digest of the resources contents (for instance, an MD5 hash). This allows the server to identify if the cached contents of the resource are different to the most recent version.

This tag is useful when for when **the last modified date is difficult to determine**.

Cache-Control:public, max-age=31536000

ETag: "15f0fff99ed5aae4edffdd6496d7131f"

On subsequent browser requests the **If-None-Match** request header is sent with the **ETag** value of the last requested version of the resource.

**If-None-Match: "15f0fff99ed5aae4edffdd6496d7131f"**

As with the If-Modified-Since header, if the current version has the same ETag value, indicating its value is the same as the browser’s cached copy, then an HTTP status of 304 is returned.

1. **What to Cache**

### [**Static assets**](https://devcenter.heroku.com/articles/increasing-application-performance-with-http-cache-headers#static-assets)

Normally this will include **static files** that are served by the application such as **images, CSS file and Javascript files**. As these files are typically re-requested on each page, a large performance improvement can be had with little effort.

In these instances, you should set the Cache-Control header, with a **max-age** value of **a year** in the future from the time of the request. It is recommended that **Expires** should be set to a similar value.

1 year is 31536000 seconds

Cache-Control:public; **max-age**=31536000

**Expires**: Mon, 25 Jun 2013 21:31:12 GMT

It is not generally a good idea to go any further than this as greater time periods are not supported by the [RFC](http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.9) and may be ignored.

* [**Dynamic content**](https://devcenter.heroku.com/articles/increasing-application-performance-with-http-cache-headers#dynamic-content)

Dynamic content is much more nuanced. For each and every resource, the developer must assess how heavily it can be cached and what the implications might be of serving stale content to the user. Two examples would be the contents of a **blog RSS feed** (which will not change more than once every few hours), to the **JSON packets** which drive a user’s Twitter timeline (updating once every few seconds). In these cases it would be reasonable to cache the resources for as long as you believe possible without causing issues for the end user.

### [**Private content**](https://devcenter.heroku.com/articles/increasing-application-performance-with-http-cache-headers#private-content)

Private content (ie. that which can be considered sensitive and subject to security measures) requires even more assessment. Not only do you as the developer need to determine the cacheability of a particular resource, but you also need to consider the impact of having intermediary caches (such as web proxies) caching the files which may be outside of the users control. If in doubt, it is a safe option to not cache these items at all.

Should end-client caching still be desirable you can ask for resources to only be cached privately (i.e only within the end-user’s browser cache):

Cache-Control:private, max-age=31536000

## [Cache prevention](https://devcenter.heroku.com/articles/increasing-application-performance-with-http-cache-headers#cache-prevention)

Highly secure or variable resources often require no caching. For instance, anything involving a shopping cart checkout process. Unfortunately, merely omitting cache headers will not work as many modern web browsers cache items based on their own internal algorithms. In such cases **it is necessary to tell the browser to explicitly to not cache items.**

In addition to public and private the Cache-Control header can specify no-cache and no-store which informs the browser to not cache the resources under any circumstances.

Both values are required as IE uses no-cache, and Firefox uses no-store.

Cache-Control:no-cache, no-store

## Cache FlowChart

## 

#### **Cacheable Methods**

* **GET and HEAD** responses may be cacheable.
* **POST** responses may be cacheable, but will only be served to a subsequent GET request. A POST request will never receive a cached response.
* The response to **PUT**, **DELETE**, CONNECT, TRACE and OPTIONS are not cacheable.

#### **Is No-store present?**

In this test we much check both the response cache-control header that comes back from the server and the equivalent request header. If either contain the no-store directive then response should not be held onto for any longer than it takes to return it to the client.

#### **Is shared cache?**

HTTP Caches are classified into two distinct types, shared and private.  A shared cache stores responses that are to be reused by more than one user.  Shared caches are the ones you find sitting in front of web servers, or at the edge of corporate networks.  Private caches are usually pieces of software that are either built into the client OS or the client application.

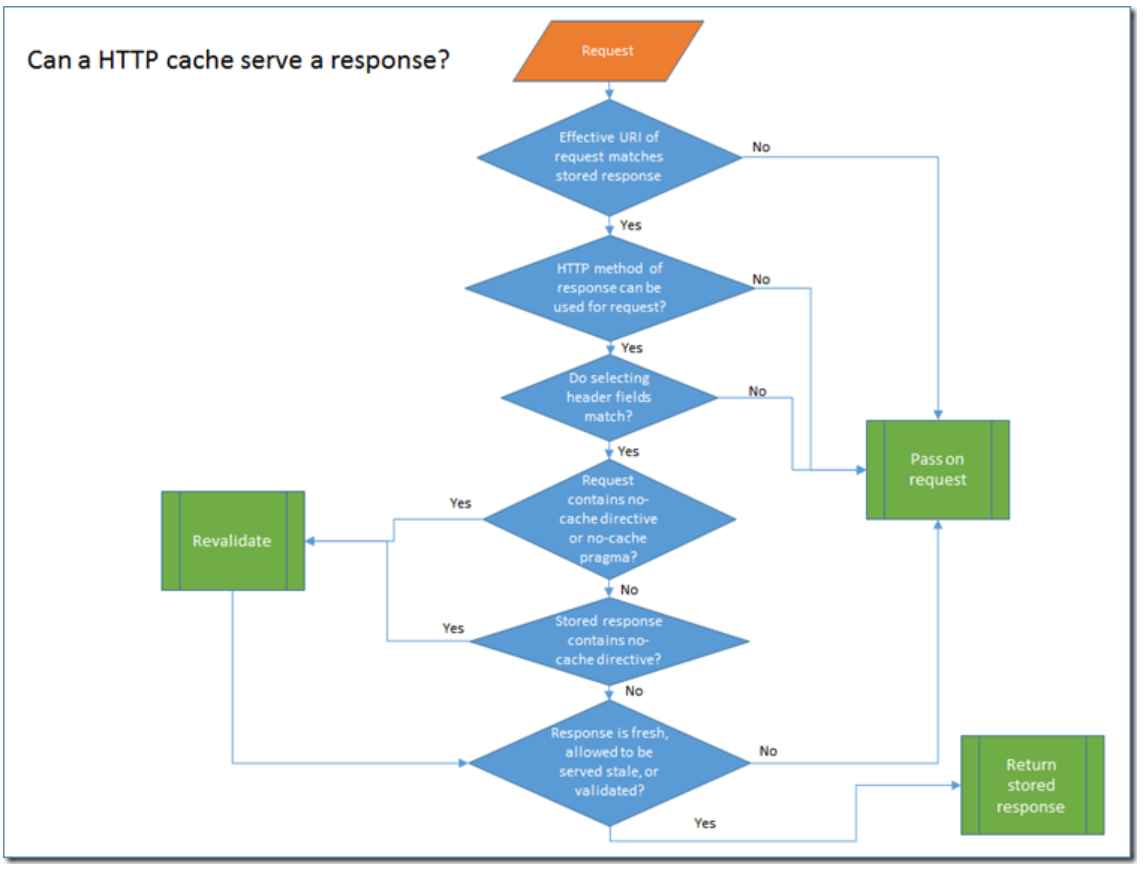
Probably the most important difference in the behaviour of private caches is that they are allowed to store responses that contain authentication headers.  This behavior which keeps your authentication credentials out of shared caches is probably one of the best arguments for using authorization header instead of some custom header or URI query parameter.

The presence of headers like must-revalidate, public and s-max-age override this limitation on shared caches not being able to store responses with an authorization header.  I understand why using public and s-max-age might do that, but I’m puzzled as to why must-revalidate does.

#### **Contains Freshness Information?**

If a response contains freshness directives like max-age, Expires, or s-max-age, then we know that the server considers the response cacheable.

Once a response has been stored, then future requests may reuse that stored response.



* Effective URI matches stored response?

The **request URI** is used as part of the **primary cache lookup key**.  The other part is the request method which we will talk about next.  The term “Effective URI” is used because on the server side [some processing](http://tools.ietf.org/html/draft-ietf-httpbis-p1-messaging-26#section-5.5) is needed to reconstruct the URI that the client used to make the request.

* Can HTTP request method return cached responses?

The second part of the **primary cache lookup key** is the **HTTP method**.  As we mentioned earlier, only GET and HEAD requests can return cached responses.  However, because the GET and HEAD request for the same resource return different representations they are treated as distinct cache entries.  I wondered if it might be acceptable to generate the HEAD cached response directly from a stored GET response by simply stripping off the body.  However, I’m not sure if that is allowed because you can use HEAD requests to freshen stored GET responses.

* **Does request or response contain no-cache directive?**

A **request** that contains a **no-cache** directive in either the cache-control header or the pragma header will not allow a stored response to be used directly even if it is fresh.  **Before it can be used, the cache must make a conditional request back to the server to confirm that the stored response is still valid.**  Once that is confirmed, then the stored response can be returned.  So, to re-iterate, just because you sent a request with no-cache, doesn’t mean that you won’t get a response served from a cache.  However, you will be guaranteed that it is up to date.

The same revalidation process occurs if the stored response contains a no-cache header.  Most people are surprised when they find out that **no-cache** doesn’t mean “don’t cache”.  It simply **means “must-revalidate even if still fresh**”.

You might notice that there is no check for the no-store request header in the diagram.  The **no-store**request header is only tested when determining if a response can be cached.  If some other user of a shared cache issues a request to a resource that is cacheable and then you issue a request to the same resource with no-store, you could still return a cached response.

* So can we finally serve this response?

**If the stored response is still fresh**.  I.e. the expired date has not passed, or the date retrieved plus max-age has not passed, then the response can be served.  If the response is stale and the client sends a max-stale directive then it may also be possible to serve the stale response.  And finally, if we have just finished re-validating the response, then we can return it.