HTTP caching

- Types of caches
  - Private caches
  - Shared cache
    - Proxy caches
    - Managed caches
- Heuristic caching
- Fresh and stale based on age
- Expires or max-age
- Vary
- Validation
  - If-Modified-Since
  - ETag/If-None-Match
  - Force Revalidation
- Don't cache
  - Do not share with others
  - Provide up-to-date content every time
  - Dealing with outdated implementations
  - What's lost by no-store
- Reload and force reload
  - Reload
  - Force reload
  - Avoiding revalidation
- Deleting stored responses
- Request collapse
- Common caching patterns
  - Default settings
  - Cache Busting
  - Validation

- Main resources
- More about managed caches
- See also

The HTTP cache stores a response associated with a request and reuses the stored response for subsequent requests.

There are several advantages to reusability. First, since there is no need to deliver the request to the origin server, then the closer the client and cache are, the faster the response will be. The most typical example is when the browser itself stores a cache for browser requests.

Also, when a response is reusable, the origin server does not need to process the request — so it does not need to parse and route the request, restore the session based on the cookie, query the DB for results, or render the template engine. That reduces the load on the server.

Proper operation of the cache is critical to the health of the system.

Types of caches

In the HTTP Caching spec, there are two main types of caches: **private caches** and **shared caches**.

Private caches

A private cache is a cache tied to a specific client — typically a browser cache. Since the stored response is not shared with other clients, a private cache can store a personalized response for that user.

On the other hand, if personalized contents are stored in a cache other than a private cache, then other users may be able to retrieve those contents — which may cause unintentional information leakage.

If a response contains personalized content and you want to store the response only in the private cache, you must specify a `private` directive.

```
Cache-Control: private
```

Personalized contents are usually controlled by cookies, but the presence of a cookie does not always indicate that it is private, and thus a cookie alone does not make the response private.

Shared cache

The shared cache is located between the client and the server and can store responses that can be shared among users. And shared caches can be further sub-classified into **proxy caches** and **managed caches**.

Proxy caches

In addition to the function of access control, some proxies implement caching to reduce traffic out of the network. This is usually not managed by the service developer, so it must be controlled by appropriate HTTP headers and so on. However, in the past, outdated proxy-cache implementations — such as implementations that do not properly understand the HTTP Caching standard — have often caused problems for developers.

Kitchen-sink headers like the following are used to try to work around "old and not updated proxy cache" implementations that do not understand current HTTP Caching spec directives like `no-store`.

```
Cache-Control: no-store, no-cache, max-age=0, must-revalidate, proxy-revalidate
```

However, in recent years, as HTTPS has become more common and client/server communication has become encrypted, proxy caches in the path can only tunnel a response and can't behave as a cache, in many cases. So in that scenario, there is no need to worry about outdated proxy cache implementations that cannot even see the response.

On the other hand, if a TLS bridge proxy decrypts all communications in a person-in-the-middle manner by installing a certificate from a CA (certificate authority) managed by the organization on the PC, and performs access control, etc. — it is possible to see the contents of the response and cache it. However, since CT (certificate transparency) has become widespread in recent years, and some browsers only allow certificates issued with an SCT (signed certificate timestamp), this method requires the application of an enterprise policy. In such a controlled environment, there is no need to worry about the proxy cache being "out of date and not updated".

Managed caches

Managed caches are explicitly deployed by service developers to offload the origin server and to deliver content efficiently. Examples include reverse proxies, CDNs, and service workers in combination with the Cache API.

The characteristics of managed caches vary depending on the product deployed. In most cases, you can control the cache's behavior through the `Cache-Control` header and your own configuration files or dashboards.

For example, the HTTP Caching specification essentially does not define a way to explicitly delete a cache — but with a managed cache, the stored response can be deleted at any time through dashboard operations, API calls, restarts, and so on. That allows for a more proactive caching strategy.

It is also possible to ignore the standard HTTP Caching spec protocols in favor of explicit manipulation. For example, the following can be specified to opt-out of a private cache or proxy cache, while using your own strategy to cache only in a managed cache.

```
Cache-Control: no-store
```

For example, Varnish Cache uses VCL (Varnish Configuration Language, a type of DSL) logic to handle cache storage, while service workers in combination with the Cache API allow you to create that logic in JavaScript.

That means if a managed cache intentionally ignores a `no-store` directive, there is no need to perceive it as being "non-compliant" with the standard. What you should do is, avoid using kitchen-sink headers, but carefully read the documentation of whatever managed-cache mechanism you're using, and ensure you're controlling the cache properly in the ways provided by the mechanism you've chosen to use.

Note that some CDNs provide their own headers that are effective only for that CDN (for example, `Surrogate-Control`). Currently, work is underway to define a `CDN-Cache-Control` header to standardize those.

Types of caches, including a private cache in the browser, a shared (proxy) cache, a reverse proxy c

Heuristic caching

HTTP is designed to cache as much as possible, so even if no `Cache-Control` is given, responses will get stored and reused if certain conditions are met. This is called **heuristic caching**.

For example, take the following response. This response was last updated 1 year ago.

```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 1024
Date: Tue, 22 Feb 2022 22:22:22 GMT
Last-Modified: Tue, 22 Feb 2021 22:22:22 GMT

<!doctype html>
…
```

It is heuristically known that content which has not been updated for a full year will not be updated for some time after that. Therefore, the client stores this response (despite the lack of `max-age`) and reuses it for a while. How long to reuse is up to the implementation, but the specification recommends about 10% (in this case 0.1 year) of the time after storing.

Heuristic caching is a workaround that came before `Cache-Control` support became widely adopted, and basically all responses should explicitly specify a `Cache-Control` header.

Fresh and stale based on age

Stored HTTP responses have two states: **fresh** and **stale**. The *fresh* state usually indicates that the response is still valid and can be reused, while the *stale* state means that the cached response has already expired.

The criterion for determining when a response is fresh and when it is stale is **age**. In HTTP, age is the time elapsed since the response was generated. This is similar to the TTL in other caching mechanisms.

Take the following example response (604800 seconds is one week):

```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 1024
Date: Tue, 22 Feb 2022 22:22:22 GMT
Cache-Control: max-age=604800

<!doctype html>
…
```

The cache that stored the example response calculates the time elapsed since the response was generated and uses the result as the response's *age*.

For the example response, the meaning of `max-age` is the following:

- If the age of the response is *less* than one week, the response is *fresh*.
- If the age of the response is *more* than one week, the response is *stale*.

As long as the stored response remains fresh, it will be used to fulfill client requests.

When a response is stored in a shared cache, it is possible to tell the client the age of the response. Continuing with the example, if the shared cache stored the response for one day, the shared cache would send the following response to subsequent client requests.

```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 1024
Date: Tue, 22 Feb 2022 22:22:22 GMT
Cache-Control: max-age=604800
Age: 86400

<!doctype html>
…
```

The client which receives that response will find it to be fresh for the remaining 518400 seconds, the difference between the response's `max-age` and `Age`.

Expires or max-age

In HTTP/1.0, freshness used to be specified by the `Expires` header.

The `Expires` header specifies the lifetime of the cache using an explicit time rather than by specifying an elapsed time.

```
Expires: Tue, 28 Feb 2022 22:22:22 GMT
```

However, the time format is difficult to parse, many implementation bugs were found, and it is possible to induce problems by intentionally shifting the system clock; therefore, `max-age` — for specifying an elapsed time — was adopted for `Cache-Control` in HTTP/1.1.

If both `Expires` and `Cache-Control: max-age` are available, `max-age` is defined to be preferred. So it is not necessary to provide `Expires` now that HTTP/1.1 is widely used.

Vary

The way that responses are distinguished from one another is essentially based on their URLs:

URL	Response body
`https://example.com/index.html`	`<!doctype html>...`
`https://example.com/style.css`	`body { ...`
`https://example.com/script.js`	`function main () { ...`

But the contents of responses are not always the same, even if they have the same URL. Especially when content negotiation is performed, the response from the server can depend on the values of the `Accept`, `Accept-Language`, and `Accept-Encoding` request headers.

For example, for English content returned with an `Accept-Language: en` header and cached, it is undesirable to then reuse that cached response for requests that have an `Accept-Language: ja` request header. In this case, you can cause the responses to be cached separately — based on language — by adding `Accept-Language` to the value of the `Vary` header.

```
Vary: Accept-Language
```

That causes the cache to be keyed based on a composite of the response URL and the `Accept-Language` request header — rather than being based just on the response URL.

URL	`Accept-Language`	Response body
`https://example.com/index.html`	`ja-JP`	`<!doctype html>...`
`https://example.com/index.html`	`en-US`	`<!doctype html>...`
`https://example.com/style.css`	`ja-JP`	`body { ...`
`https://example.com/script.js`	`ja-JP`	`function main () { ...`

Also, if you are providing content optimization (for example, for responsive design) based on the user agent, you may be tempted to include `User-Agent` in the value of the `Vary` header. However, the `User-Agent` request header generally has a very large number of variations, which drastically reduces the chance that the cache will be reused. So if possible, instead consider a way to vary behavior based on feature detection rather than based on the `User-Agent` request header.

For applications that employ cookies to prevent others from reusing cached personalized content, you should specify `Cache-Control: private` instead of specifying a cookie for `Vary`.

Validation

Stale responses are not immediately discarded. HTTP has a mechanism to transform a stale response into a fresh one by asking the origin server. This is called **validation**, or sometimes, **revalidation**.

Validation is done by using a **conditional request** that includes an `If-Modified-Since` or `If-None-Match` request header.

If-Modified-Since

The following response was generated at 22:22:22 and has a `max-age` of 1 hour, so you know that it is fresh until 23:22:22.

```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 1024
Date: Tue, 22 Feb 2022 22:22:22 GMT
Last-Modified: Tue, 22 Feb 2022 22:00:00 GMT
Cache-Control: max-age=3600

<!doctype html>
…
```

At 23:22:22, the response becomes stale and the cache cannot be reused. So the request below shows a client sending a request with an `If-Modified-Since` request header, to ask the server if there have been any changes made since the specified time.

```
GET /index.html HTTP/1.1
Host: example.com
Accept: text/html
If-Modified-Since: Tue, 22 Feb 2022 22:00:00 GMT
```

The server will respond with `304 Not Modified` if the content has not changed since the specified time.

Since this response only indicates "no change", there is no response body — there's just a status code — so the transfer size is extremely small.

```
HTTP/1.1 304 Not Modified
Content-Type: text/html
Date: Tue, 22 Feb 2022 23:22:22 GMT
Last-Modified: Tue, 22 Feb 2022 22:00:00 GMT
Cache-Control: max-age=3600
```

Upon receiving that response, the client reverts the stored stale response back to being fresh and can reuse it during the remaining 1 hour.

The server can obtain the modification time from the operating-system file system, which is relatively easy to do for the case of serving static files. However, there are some problems; for example, the time format is complex and difficult to parse, and distributed servers have difficulty synchronizing file-update times.

To solve such problems, the `ETag` response header was standardized as an alternative.

ETag/If-None-Match

The value of the `ETag` response header is an arbitrary value generated by the server. There are no restrictions on how the server must generate the value, so servers are free to set the value based on whatever means they choose — such as a hash of the body contents or a version number.

As an example, if a hash value is used for the `ETag` header and the hash value of the `index.html` resource is `33a64df5`, the response will be as follows:

```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 1024
Date: Tue, 22 Feb 2022 22:22:22 GMT
ETag: "33a64df5"
Cache-Control: max-age=3600

<!doctype html>
…
```

If that response is stale, the client takes the value of the `ETag` response header for the cached response, and puts it into the `If-None-Match` request header, to ask the server if the resource has been modified:

```
GET /index.html HTTP/1.1
Host: example.com
Accept: text/html
If-None-Match: "33a64df5"
```

The server will return `304 Not Modified` if the value of the `ETag` header it determines for the requested resource is the same as the `If-None-Match` value in the request.

But if the server determines the requested resource should now have a different `ETag` value, the server will instead respond with a `200 OK` and the latest version of the resource.

> **NOTE:** RFC9110 prefers that servers send both `ETag` and `Last-Modified` for a `200` response if possible. During cache revalidation, if both `If-Modified-Since` and `If-None-Match` are present, then `If-None-Match` takes precedence for the validator. If you are only considering caching, you may think that `Last-Modified`

> is unnecessary. However, `Last-Modified` is not just useful for caching; it is a standard HTTP header that is also used by content-management (CMS) systems to display the last-modified time, by crawlers to adjust crawl frequency, and for other various purposes. So considering the overall HTTP ecosystem, it is better to provide both `ETag` and `Last-Modified`.

Force Revalidation

If you do not want a response to be reused, but instead want to always fetch the latest content from the server, you can use the `no-cache` directive to force validation.

By adding `Cache-Control: no-cache` to the response along with `Last-Modified` and `ETag` — as shown below — the client will receive a `200 OK` response if the requested resource has been updated, or will otherwise receive a `304 Not Modified` response if the requested resource has not been updated.

```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 1024
Date: Tue, 22 Feb 2022 22:22:22 GMT
Last-Modified: Tue, 22 Feb 2022 22:00:00 GMT
ETag: "deadbeef"
Cache-Control: no-cache

<!doctype html>
…
```

> **NOTE:** The `no-cache` directive (or equivalent, such as `max-age=0, must-revalidate`) does not guarantee revalidation for history navigations — such as those made using the Back button. If the back/forward cache (bfcache) is used, the browser restores a snapshot of the page without revalidating. Even when bfcache is not used, the browser may still serve the cached response without revalidating. This is allowed by the specification because history navigations are usually treated as restoring a snapshot of a historical session and not a new request for a previously visited page.

It is often stated that the combination of `max-age=0` and `must-revalidate` has the same meaning as `no-cache`.

```
Cache-Control: max-age=0, must-revalidate
```

`max-age=0` means that the response is immediately stale, and `must-revalidate` means that it must not be reused without revalidation once it is stale — so, in combination, the semantics seem to be the same as `no-cache`.

However, that usage of `max-age=0` is a remnant of the fact that many implementations prior to HTTP/1.1 were unable to handle the `no-cache` directive — and so to deal with that limitation, `max-age=0` was used as a workaround.

But now that HTTP/1.1-conformant servers are widely deployed, there's no reason to ever use that `max-age=0` and `must-revalidate` combination — you should instead just use `no-cache`.

Don't cache

The `no-cache` directive does not prevent the storing of responses but instead prevents the reuse of responses without revalidation.

If you don't want a response stored in any cache, use `no-store`.

```
Cache-Control: no-store
```

However, in general, a "do not cache" requirement in practice amounts to the following set of circumstances:

- Don't want the response stored by anyone other than the specific client, for privacy reasons.
- Want to provide up-to-date information always.
- Don't know what could happen in outdated implementations.

Under that set of circumstances, `no-store` is not always the most-appropriate directive.

The following sections look at the circumstances in more detail.

Do not share with others

It would be problematic if a response with personalized content is unexpectedly visible to other users of a cache.

In such a case, using the `private` directive will cause the personalized response to only be stored with the specific client and not be leaked to any other user of the cache.

```
Cache-Control: private
```

Provide up-to-date content every time

The `no-store` directive prevents a response from being stored, but does not delete any already-stored response for the same URL.

In other words, if there is an old response already stored for a particular URL, returning `no-store` will not prevent the old response from being reused.

However, a `no-cache` directive will force the client to send a validation request before reusing any stored response.

```
Cache-Control: no-cache
```

If the server does not support conditional requests, you can force the client to access the server every time and always get the latest response with `200 OK`.

Dealing with outdated implementations

As a workaround for outdated implementations that ignore `no-store`, you may see kitchen-sink headers such as the following being used.

```
Cache-Control: no-store, no-cache, max-age=0, must-revalidate, proxy-revalidate
```

It is recommended to use `no-cache` as an alternative for dealing with such outdated implementations, and it is not a problem if `no-cache` is given from the beginning, since the server will always receive the request.

If it is the shared cache that you are concerned about, you can make sure to prevent unintended caching by also adding `private`:

```
Cache-Control: no-cache, private
```

What's lost by `no-store`

You may think adding `no-store` would be the right way to opt-out of caching.

However, it's not recommended to grant `no-store` liberally, because you lose many advantages that HTTP and browsers have, including the browser's back/forward cache.

Therefore, to get the advantages of the full feature set of the web platform, prefer the use of `no-cache` in combination with `private`.

Reload and force reload

Validation can be performed for requests as well as responses.

The **reload** and **force reload** actions are common examples of validation performed from the browser side.

Reload

For recovering from window corruption or updating to the latest version of the resource, browsers provide a reload function for users.

A simplified view of the HTTP request sent during a browser reload looks as follows:

```
GET / HTTP/1.1
Host: example.com
Cache-Control: max-age=0
If-None-Match: "deadbeef"
If-Modified-Since: Tue, 22 Feb 2022 20:20:20 GMT
```

(The requests from Chrome, Edge, and Firefox look very much like the above; the requests from Safari will look a bit different.)

The `max-age=0` directive in the request specifies "reuse of responses with an age of 0 or less" — so, in effect, intermediately stored responses are not reused.

As a result, a request is validated by `If-None-Match` and `If-Modified-Since`.

That behavior is also defined in the Fetch standard and can be reproduced in JavaScript by calling `fetch()` with the cache mode set to `no-cache` (note that `reload` is not the right mode for this case):

```
// Note: "reload" is not the right mode for a normal reload; "no-cache" is
fetch("/", { cache: "no-cache" });
```

Force reload

Browsers use `max-age=0` during reloads for backward-compatibility reasons — because many outdated implementations prior to HTTP/1.1 did not understand `no-cache`. But `no-cache` is fine now in this use case, and **force reload** is an additional way to bypass cached responses.

The HTTP Request during a browser **force reload** looks as follows:

```
GET / HTTP/1.1
Host: example.com
Pragma: no-cache
Cache-Control: no-cache
```

(The requests from Chrome, Edge, and Firefox look very much like the above; the requests from Safari will look a bit different.)

Since that's not a conditional request with `no-cache`, you can be sure you'll get a `200 OK` from the origin server.

That behavior is also defined in the Fetch standard and can be reproduced in JavaScript by calling `fetch()` with the cache mode set to `reload` (note that it's not `force-reload`):

```
// Note: "reload" — rather than "no-cache" — is the right mode for a "force reload"
fetch("/", { cache: "reload" });
```

Avoiding revalidation

Content that never changes should be given a long `max-age` by using cache busting — that is, by including a version number, hash value, etc., in the request URL.

However, when the user reloads, a revalidation request is sent even though the server knows that the content is immutable.

To prevent that, the `immutable` directive can be used to explicitly indicate that revalidation is not required because the content never changes.

```
Cache-Control: max-age=31536000, immutable
```

That prevents unnecessary revalidation during reloads.

Note that, instead of implementing that directive, Chrome has changed its implementation so that revalidation is not performed during reloads for subresources.

Deleting stored responses

There is no way to delete responses on an intermediate server that have been stored with a long `max-age`.

Imagine that the following response from `https://example.com/` was stored.

```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 1024
Cache-Control: max-age=31536000

<!doctype html>
…
```

You may want to overwrite that response once it expired on the server, but there is nothing the server can do once the response is stored — since no more requests reach the server due to caching.

One of the methods mentioned in the specification is to send a request for the same URL with an unsafe method such as `POST`, but for many clients that is difficult to do.

The `Clear-Site-Data: cache` header and directive value can be used to clear browser caches — but has no effect on intermediate caches. Otherwise responses will remain in the browser cache until `max-age` expires, unless the user manually performs a reload, force-reload, or clear-history action.

Caching reduces access to the server, which means that the server loses control of that URL. If the server does not want to lose control of a URL — for example, in the case that a resource is frequently updated — you should add `no-cache` so that the server will always receive requests and send the intended responses.

Request collapse

The shared cache is primarily located before the origin server and is intended to reduce traffic to the origin server.

Thus, if multiple identical requests arrive at a shared cache at the same time, the intermediate cache will forward a single request on behalf of itself to the origin, which can then reuse the result for all clients. This is called ***request collapse***.

Request collapse occurs when requests are arriving at the same time, so even if `max-age=0` or `no-cache` is given in the response, it will be reused.

If the response is personalized to a particular user and you do not want it to be shared in collapse, you should add the `private` directive:

Request collapse shown as multiple clients sending GET requests and a cache consolidating them in

Common caching patterns

There are many directives in the `Cache-Control` spec, and it may be difficult to understand all of them. But most websites can be covered by a combination of a handful of patterns.

This section describes the common patterns in designing caches.

Default settings

As mentioned above, the default behavior for caching (that is, for a response without `Cache-Control`) is not simply "don't cache" but implicit caching according to so-called "heuristic caching".

To avoid that heuristic caching, it's preferable to explicitly give all responses a default `Cache-Control` header.

To ensure that by default the latest versions of resources will always be transferred, it's common practice to make the default `Cache-Control` value include `no-cache`:

```
Cache-Control: no-cache
```

In addition, if the service implements cookies or other login methods, and the content is personalized for each user, `private` must be given too, to prevent sharing with other users:

```
Cache-Control: no-cache, private
```

Cache Busting

The resources that work best with caching are static immutable files whose contents never change. And for resources that *do* change, it is a common best practice to change the URL each time the content changes, so that the URL unit can be cached for a longer period.

As an example, consider the following HTML:

```
<script src="bundle.js"></script>
<link rel="stylesheet" href="build.css" />
<body>
  hello
</body>
```

In modern web development, JavaScript and CSS resources are frequently updated as development progresses. Also, if the versions of JavaScript and CSS resources that a client uses are out of sync, the display will break.

So the HTML above makes it difficult to cache `bundle.js` and `build.css` with `max-age`.

Therefore, you can serve the JavaScript and CSS with URLs that include a changing part based on a version number or hash value. Some of the ways to do that are shown below.

```
# version in filename
bundle.v123.js

# version in query
bundle.js?v=123

# hash in filename
bundle.YsAIAAAA-QG4G6kCMAMBAAAAAAAoK.js

# hash in query
bundle.js?v=YsAIAAAA-QG4G6kCMAMBAAAAAAAoK
```

Since the cache distinguishes resources from one another based on their URLs, the cache will not be reused again if the URL changes when a resource is updated.

```
<script src="bundle.v123.js"></script>
<link rel="stylesheet" href="build.v123.css" />
<body>
  hello
</body>
```

With that design, both JavaScript and CSS resources can be cached for a long time. So how long should `max-age` be set to? The QPACK specification provides an answer to that question.

QPACK is a standard for compressing HTTP header fields, with tables of commonly-used field values defined.

Some commonly-used cache-header values are shown below.

```
36 cache-control max-age=0
37 cache-control max-age=604800
38 cache-control max-age=2592000
39 cache-control no-cache
40 cache-control no-store
41 cache-control public, max-age=31536000
```

If you select one of those numbered options, you can compress values in 1 byte when transferred via HTTP3.

Numbers `37`, `38`, and `41` are for periods of one week, one month, and one year.

Because the cache removes old entries when new entries are saved, the probability that a stored response still exists after one week is not that high — even if `max-age` is set to 1 week. Therefore, in practice, it does not make much difference which one you choose.

Note that number `41` has the longest `max-age` (1 year), but with `public`.

The `public` value has the effect of making the response storable even if the `Authorization` header is present.

> **NOTE:** The `public` directive should only be used if there is a need to store the response when the `Authorization` header is set. It is not required otherwise, because a response will be stored in the shared cache as long as `max-age` is given.

So if the response is personalized with basic authentication, the presence of `public` may cause problems. If you are concerned about that, you can choose the second-longest value, `38` (1 month).

```
# response for bundle.v123.js

# If you never personalize responses via Authorization
Cache-Control: public, max-age=31536000

# If you can't be certain
Cache-Control: max-age=2592000
```

Validation

Don't forget to set the `Last-Modified` and `ETag` headers, so that you don't have to re-transmit a resource when reloading. It's easy to generate those headers for pre-built static files.

The `ETag` value here may be a hash of the file.

```
# response for bundle.v123.js
Last-Modified: Tue, 22 Feb 2022 20:20:20 GMT
ETag: "YsAIAAAA-QG4G6kCMAMBAAAAAAAoK"
```

In addition, `immutable` can be added to prevent validation on reload.

The combined result is shown below.

```
# bundle.v123.js
HTTP/1.1 200 OK
Content-Type: text/javascript
Content-Length: 1024
Cache-Control: public, max-age=31536000, immutable
Last-Modified: Tue, 22 Feb 2022 20:20:20 GMT
ETag: "YsAIAAAA-QG4G6kCMAMBAAAAAAAoK"
```

Cache busting is a technique to make a response cacheable over a long period by changing the URL when the content changes. The technique can be applied to all subresources, such as images.

> **NOTE:** When evaluating the use of `immutable` and QPACK: If you're concerned that `immutable` changes the predefined value provided by QPACK, consider that in this case, the `immutable` part can be encoded separately by splitting the `Cache-Control` value into two lines — though this is dependent on the encoding algorithm a particular QPACK implementation uses.

```
Cache-Control: public, max-age=31536000
Cache-Control: immutable
```

Main resources

Unlike subresources, main resources cannot be cache busted because their URLs can't be decorated in the same way that subresource URLs can be.

If the following HTML itself is stored, the latest version cannot be displayed even if the content is updated on the server side.

```
<script src="bundle.v123.js"></script>
<link rel="stylesheet" href="build.v123.css" />
<body>
  hello
</body>
```

For that case, `no-cache` would be appropriate — rather than `no-store` — since we don't want to store HTML, but instead just want it to always be up-to-date.

Furthermore, adding `Last-Modified` and `ETag` will allow clients to send conditional requests, and a `304 Not Modified` can be returned if there have been no updates to the HTML:

```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 1024
Cache-Control: no-cache
Last-Modified: Tue, 22 Feb 2022 20:20:20 GMT
ETag: "AAPuIbAOdvAGEETbgAAAAAAABAAE"
```

That setting is appropriate for non-personalized HTML, but for a response that gets personalized using cookies — for example, after a login — don't forget to also specify `private`:

```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 1024
Cache-Control: no-cache, private
Last-Modified: Tue, 22 Feb 2022 20:20:20 GMT
ETag: "AAPuIbAOdvAGEETbgAAAAAAABAAE"
Set-Cookie: __Host-SID=AHNtAyt3fvJrUL5g5tnGwER; Secure; Path=/; HttpOnly
```

The same can be used for `favicon.ico`, `manifest.json`, `.well-known`, and API endpoints whose URLs cannot be changed using cache busting.

Most web content can be covered by a combination of the two patterns described above.

More about managed caches

With the method described in previous sections, subresources can be cached for a long time by using cache busting, but main resources (which are usually HTML documents) can't be.

Caching main resources is difficult because, using just standard directives from the HTTP Caching specification, there's no way to actively delete cache contents when content is updated on the server.

However, it is possible by deploying a managed cache such as a CDN or service worker.

For example, a CDN that allows cache purging via an API or dashboard operation would allow for a more aggressive caching strategy by storing the main resource and explicitly purging the relevant cache only when an update occurs on the server.

A service worker could do the same if it could delete the contents in the Cache API when an update occurs on the server.

For more information, see the documentation for your CDN, and consult the service worker documentation.

See also

- `Cache-Control` HTTP header
- RFC 9111: Hypertext Transfer Protocol (HTTP/1.1): Caching
- Caching Tutorial - Mark Nottingham

□□□ #3
□ □□□ □□□ 2026-10-31 03:27:53 UTC
□ cocobook-sync □□□ 2026-11-04 17:35:26 UTC