ANNA UNIVERSITY, GUINDY, CHENNAI:: 600 025

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Course Code: CS6111

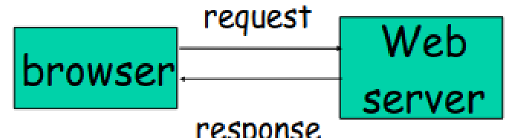
Course Name: Computer Networks

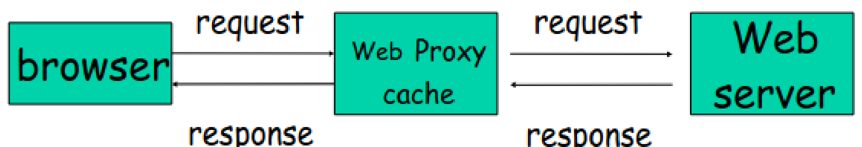
Date :06.09.2024

Web Caching:

Web pages can either be dynamic or static. Dynamic web pages are regenerated for every request and for every user, as opposed to static web pages which can just be pregenerated HTML files which only change when the user changes them. Therefore, dynamic web pages are typically not cached. For this reason, content being referred to in these notes is largely static content (static audio, video, web pages etc).

Simpler model: clients are read-only, only server updates data





Client-side caching

In the typical situation, the first time a browser retrieves cacheable resources, such as a web page, the resources will be stored in the local HTTP Cache. The next time the browser navigates to this page, resources will be loaded from the local storage, rather than re-downloaded from the origin server. The conservation of bandwidth and reduction in server-side processing is where the performance increase comes from.

A browser cache is an example of a forward cache, which sits outside of the web server's network. Other examples of a forward cache implementation might be at the corporate network level or with an intermediate proxy server.

Client-Side caches that are running at the browser level are private, which means that the cache is available only to a single client. A corporate-level cache may be shared, which means that different clients within the same corporate network can rely on using this cache, rather than resubmit an identical HTTP request to the server.

Server-side caching

The idea behind server-side caching is that an intermediate node or alternative server will be used to cache responses from the origin server. This is also known as a reverse cache. The goal is to reduce the load on the server, which in turn will reduce the overall latency of the request.

An example of a reverse cache is a content delivery network (*CDN*) that stores copies of resources for quicker delivery across different points of the network. A *CDN* might have nodes in many different countries and thus a lot of traffic in one locale will not have as much of an effect, if any, on other locations.

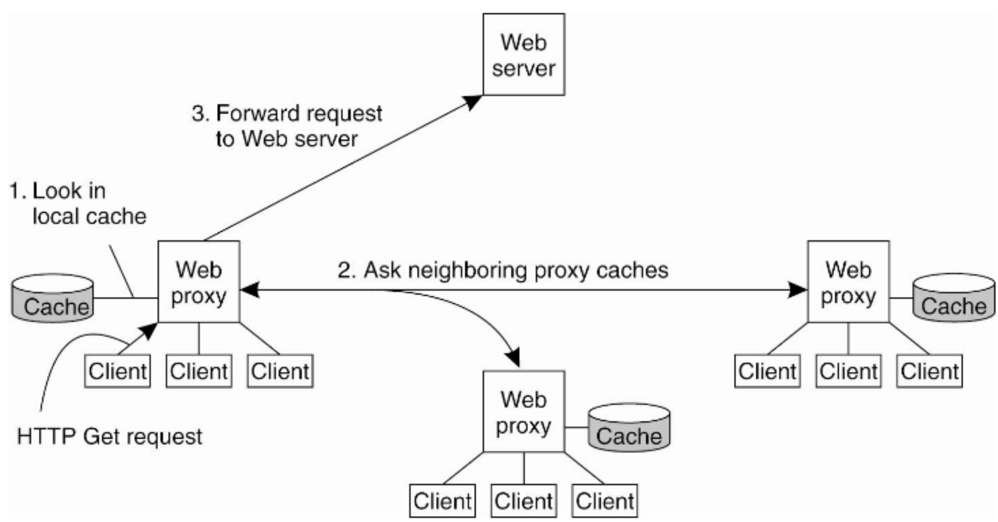
Similarly, this type of cache is normally shared. This means that different clients will be served copies of the same resource, making it unnecessary to contact the server with a HTTP request identical to one that another client has recently made.

Web Proxy Caching

Web caching uses a client-proxy-server architecture. Clients send requests to the proxy. Proxies can service the requests directly if they have the resources. If they do not have the resources, they go to the server to get the request processed.

In web caching, the proxy provides the service of caching i.e. the proxy caches content from the server and when the client browsers make a request, if the content is already in the cache, it returns the content. This helps in faster responses for the client, if the proxy is near the client, and reduced loads for the server.

Along with communicating with the server, these proxy caches can also communicate with each other. This mechanism is called “Cooperative Caching”.
Figure 2 shows one such scenario where a client sends a request to the web proxy. The web proxy will then look into its local cache for the requested web page. If it is a hit, then it will send the response back. In the case of miss, typically the web proxy will contact server. But in the case of cooperative caching, cache misses can be serviced by asking a nearby/local proxy instead of the server i.e. it will reach out to the near by proxies to get the data. In this case, all the caches act like one big logical cache, so the clients will see union of all the content stored in nearby caches rather than just the content cached in the local proxy. This can make fetching faster than getting data from the server



Consistency Issues

Recall a system has consistency if all the replicas see the same data at the same time. In the context of web caching, this means that when a browser fetches a page, we are guaranteed that the returned page is the most recent version. Web pages tend to change with time, so if we cached a webpage and the webpage changed on the server, the proxy may serve stale content. So, we need to ensure the consistency of cache web pages.
In tackling the issue of consistency, we need to in consideration, the the read frequency (how popular the web page is) and the update frequency (how often it changes). There are 2 approaches for maintaining consistency - pull-based and push-based.

Push Based Approaches:

Web pages tend to be updated over time
– Some objects are static, others are dynamic
– Different update frequencies (few minutes to few weeks)
• How can a proxy cache maintain consistency of cached data?
– Send invalidate or update
– Push versus pull

• Server tracks all proxies that have requested objects
• If a web page is modified, notify each proxy
• Notification types
– Indicate object has changed [invalidate]
– Send new version of object [update]
• How to decide between invalidate and updates?
– Pros and cons?
– One approach: send updates for more frequent objects,invalidate for rest

Advantages
– Provide tight consistency [minimal stale data]
– Proxies can be passive
• Disadvantages
– Need to maintain state at the server
• Recall that HTTP is stateless
• Need mechanisms beyond HTTP
– State may need to be maintained indefinitely
• Not resilient to server crashes

Pull Based Approaches:

Proxy is entirely responsible for maintaining consistency
• Proxy periodically polls the server to see if object has
changed
– Use if-modified-since HTTP messages
• Key question: when should a proxy poll?
– Server-assigned Time-to-Live (TTL) values
• No guarantee if the object will change in the interim

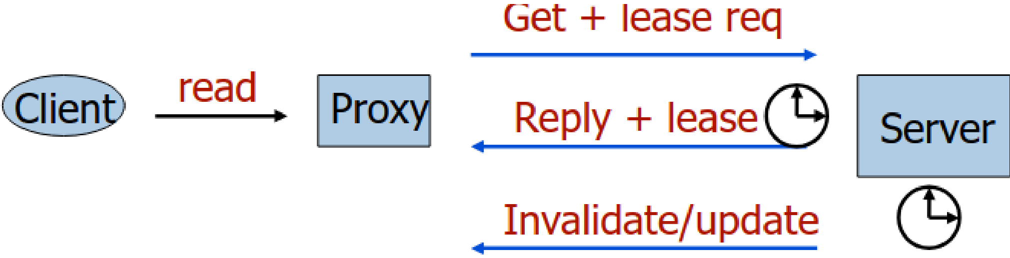
Proxy can dynamically determine the refresh interval
– Compute based on past observations
• Start with a conservative refresh interval
• Increase interval if object has not changed between two successive polls

• Decrease interval if object is updated between two polls
• Adaptive: No prior knowledge of object characteristics needed

Advantages
– Implementation using HTTP (If-modified-Since)
– Server remains stateless
– Resilient to both server and proxy failures
• Disadvantages
– Weaker consistency guarantees (objects can change between two polls and proxy will contain stale data until next poll)
• Strong consistency only if poll before every HTTP response
– More sophisticated proxies required
– High message overhead

Hybrid Approach:
Hybrid approach is based on both push and pull.
Lease: duration of time for which server agrees to notify proxy of modification
Lease is a contract between two entities (here the server and the proxy). It specifies the duration of time the server agrees to notify the proxy about any updates on the web page. Updates are no longer sent by the server after lease expiration, and the server will delete the state. Proxy can renew the lease. If the page is unpopular, proxy can decide not to renew the lease for that page

• Issue lease on first request, send notification until expiry
– Need to renew lease upon expiry
• Smooth tradeoff between state and messages exchanged
– Zero duration => polling, Infinite leases => server-push
• Efficiency depends on the lease duration



Policies for Leases Duration

• Age-based lease
– Based on bi-modal nature of object lifetimes
– Larger the expected lifetime longer the lease

• Renewal-frequency based
– Based on skewed popularity
– Proxy at which objects is popular gets longer lease

• Server load based
– Based on adaptively controlling the state space
– Shorter leases during heavy load

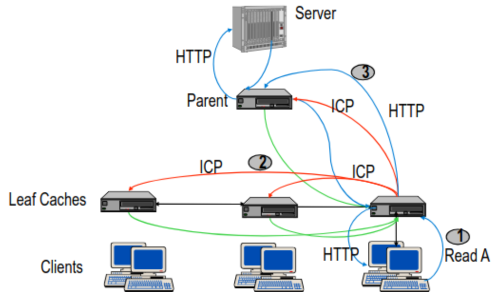
Cooperative Caching

In coperative caching a collection of proxies cooperate with each other to service client requests
• Caching infrastructure can have multiple web proxies

The client sends a request to one of the proxies. If it is a cache miss, the proxy will send requests to its peers (red arrows) and parent using ICP (Internet Cache Protocol) messages. If none of the peers have it, they will send back the non-availability response to the proxy (green arrows). The proxy will then forward the HTTP request to its parent and the whole process will recurse until a cache hit occurs or until the server is reached. The data will then be sent back as response - and will flow down the hierarchy, back to the client
– Proxies can be arranged in a hierarchy or other structures
• Overlay network of proxies: content distribution network
– Proxies can cooperate with one another
• Answer client requests
• Propagate server notifications

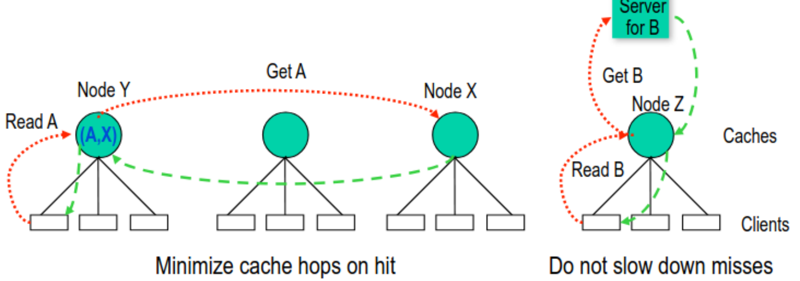
Hierarchical Proxy Caching

This works well when a nearby proxy actually has the content. If there is a global miss - no one in the hierarchy has the content, latency will increase significantly. Clearly, there is a lot of messaging overhead.
Also, browser has to wait for longer times in the case of cache miss on the proxy. This will affect performance.
Latency could increase - it may have been faster to just directly request from the server in the event of a cache miss on the proxy.



Examples: Squid, Harvest

Locating and Accessing Data in the Flattened Network



Minimize cache hops on hit

Do not slow down misses

Properties

- Lookup is local
- Hit at most 2 hops
- Miss at most 2 hops (1 extra on wrong hint)

Web Cache using HTTP :

In HTTP communication process directives can also be given to define how the exchanged information will be stored and this is then what we call HTTP caching. This information about caching occurs in the request and response headers and they define the behavior desired by the client or server. We can also say that the main purpose of caching is to improve communication performance by reusing a previous response message to satisfy a current request.

In its most basic form, the caching process works as follows:

1. The website page requests a resource from the origin server.
2. The system checks the cache to see if there is already a stored copy of the resource.
3. If the resource is cached, the result will be a cache hit response and the resource will be delivered from the cache.
4. If the resource is not cached, cache loss will result and the file will be accessed in its original source.
5. After the resource is cached, it will continue to be accessed there until it expires or the cache is cleared.

How caching works

When a browser requests a file from a server, the server responds with the file and some cache headers. The browser then caches the file based on these headers. The next time the browser requests the same file, it checks its cache to see if it already has a copy. If it does, and the file hasn't expired, the browser serves the cached version of the file. If the file has expired or if the browser has been told not to cache it, the browser requests a fresh copy of the file from the server.

Caching works differently depending on the type of cache being used. There are two main types of caches: browser caches and CDN caches.

Browser caches

Browser caches are local caches that are used by web browsers to store copies of files. When a browser requests a file, it first checks its local cache to see if it already has a copy. If it does, and the file hasn't expired, the browser serves the cached version of the file. If the file has expired, or if the browser has been told not to cache it, the browser requests a fresh copy of the file from the server.

CDN caches

CDN caches are distributed caches that are used by Content Delivery Networks (CDNs) to store copies of files. When a browser requests a file from a website that is using a CDN, the request is sent to the CDN instead of the origin server. If the CDN has a cached copy of the file, it serves it directly to the browser. This can greatly reduce the amount of time and resources needed to load the file, as the request doesn't need to travel all the way to the origin server.

CDN caches can be configured in a number of different ways, depending on the needs of the website. Some CDNs use a "pull" model, where the CDN only caches files when they are requested

by a browser. Other CDNs use a "push" model, where the origin server sends files to the CDN proactively before they are requested by a browser.

Types of Caching

The *type of cache* is defined according to where the content is stored.

- **Browser cache** - this storage is done in the browser. All browsers have a local storage, which is usually used to retrieve previously accessed resources. This type of cache is private since stored resources are not shared.
- **Proxy cache** - this storage, also called intermediate caching, is done on the proxy server, between the client and the origin server. This is a type of shared cache as it's used by multiple clients and is usually maintained by providers.
- **Gateway cache** - also called reverse proxy, it's a separate, independent layer, and this storage is between the client and the application. It caches the requests made by the client and sends them to the application and does the same with the responses, sending from the application to the client. If a resource is requested again, the cache returns the response before reaching the application. It's also a shared cache, but by servers not users.
- **Application cache** - this storage is done in the application. It allows the developer to specify which files the browser should cache and make them available to users even when they are offline.

HTTP Cache Headers

One of the best ways to have a website which is fast, efficient, and accessible to as many users as possible is by using HTTP caching headers. These headers tell web browsers and other HTTP clients how to cache and serve content from your website.

What are HTTP cache headers?

HTTP cache headers are instructions that web servers send to web browsers, telling them how to cache and serve content. These headers are sent with every HTTP request and response. They can be used to control how frequently a browser caches a file, how long the cache should keep the file, and what should be done when the file is expired.

HTTP cache headers are important because they help reduce the amount of time and resources needed to load a web page. By caching content, a browser can serve it more quickly without having to request it from the server every time a user visits the page. This can improve website performance, reduce server load, and improve the overall user experience.

Types of HTTP cache headers

Caches work with content mainly through freshness and validation. A fresh representation is available instantly from a cache while a validated representation rarely sends the entire representation again if it hasn't changed. In cases where there is no validator present (e.g. `ETag` or `Last-Modified` header), and a lack of explicit freshness info, it will usually (but not always) be considered uncacheable. Let's shift our focus to the kind of headers you should be concerned about.

1. `Cache-Control`

Every resource can define its own caching policy via the `Cache-Control` HTTP header. `Cache-Control` directives control who caches the response, under what conditions and for how long.

Requests that don't need server communication are considered the best requests: local copies of the responses allow the elimination of network latency as well as data charges resulting from data transfers. The HTTP specification enables the server to send several different `Cache-Control` directives which control how and for how long individual responses are cached by browsers among other intermediate caches such as a CDN.

```
Cache-Control: private, max-age=0, no-cache
```

These settings are referred to as response directives. They are as follows:

`public` vs `private`

A response that is marked `public` can be cached even in cases where it is associated with an HTTP authentication or the HTTP response status code is not cacheable normally. In most cases, a response marked `public` isn't necessary, since explicit caching information (e.g. `max-age`) shows that a response is cacheable anyway.

On the contrary, a response marked `private` can be cached (by the browser) but such responses are typically intended for single users hence they aren't cacheable by intermediate caches (e.g. HTML pages with private user info can be cached by a user's browser but not by a CDN).

`no-cache` and `no-store`

`no-cache` shows that returned responses can't be used for subsequent requests to the same URL before checking if server responses have changed. If a proper `ETag` (validation token) is present as a result, `no-cache` incurs a roundtrip in an effort to validate cached responses. Caches can however eliminate downloads if the resources haven't changed. In other words, web browsers might cache the assets but they have to check on every request if the assets have changed (304 response if nothing has changed).

On the contrary, `no-store` is simpler. This is the case because it disallows browsers and all intermediate caches from storing any versions of returned responses, such as responses containing private/personal information or banking data. Every time users request this asset, requests are sent to the server. The assets are downloaded every time.

`max-age`

The `max-age` directive states the maximum amount of time in seconds that fetched responses are allowed to be used again (from the time when a request is made). For instance, `max-age=90` indicates that an asset can be reused (remains in the browser cache) for the next 90 seconds.

`s-maxage`

The "s-" stands for shared as in shared cache. This directive is explicitly for CDNs among other intermediary caches. This directive overrides the `max-age` directive and expires header field when present. KeyCDN also obeys this directive.

`must-revalidate`

The `must-revalidate` directive is used to tell a cache that it must first revalidate an asset with the origin after it becomes stale. The asset must not be delivered to the client without doing an end-to-end revalidation. In short, stale assets must first be verified and expired assets should not be used.

`proxy-revalidate`

The `proxy-revalidate` directive is the same as the `must-revalidate` directive, however, it only applies to shared caches such as proxies. It is useful in the event that a proxy services many users that need to be authenticated one by one. A response to an authenticated request can be stored in the user's cache without needing to revalidate it each time as they are known and have already been authenticated. However, `proxy-revalidate` allows proxies to still revalidate for new users that have not been authenticated yet.

`no-transform`

The `no-transform` directive tells any intermediary such as a proxy or cache server to not make any modifications whatsoever to the original asset. The `Content-Encoding`, `Content-Range`, and `Content-Type` headers must remain unchanged. This can occur if a non-transparent proxy decides to make modifications to assets in order to save space. However, this can cause serious problems in the event that the asset must remain identical to the original entity-body although it must also pass through the proxy.

According to Google, the `Cache-Control` header is all that's needed in terms of specifying caching policies. Other methods are available, which we'll go over in this article, however, are **not required** for optimal performance.

The `Cache-Control` header is defined as part of HTTP/1.1 specifications and supersedes previous headers (e.g. `Expires`) used to specify response caching policies. `Cache-Control` is supported by all modern browsers so that's all we need.

2. `Pragma`

The old `Pragma` header accomplishes many things most of them characterized by newer implementations. We are however most concerned with the `Pragma: no-cache` directive which is interpreted by newer implementations as `Cache-Control: no-cache`. You don't need to be concerned about this directive because it's a request header that will be ignored by KeyCDN's edge servers. It is however important to be aware of the directive for the overall understanding. Going forward, there won't be new HTTP directives defined for `Pragma`.

3. `Expires`

A couple of years back, this was the main way of specifying when assets expire. `Expires` is simply a basic date-time stamp. It's fairly useful for old user agents which still roam unchartered territories. It is, however, important to note that `Cache-Control` headers, `max-age` and `s-maxage` still take precedence on most modern systems. It's however good practice to set matching values here for the sake of compatibility. It's also important to ensure you format the date properly or it might be considered as expired.

```
Expires: Sun, 03 May 2015 23:02:37 GMT
```

To avoid breaking the specification, avoid setting the date value to more than a year.

4. Validators

`ETag`

This type of validation token (the standard in HTTP/1.1):

- Is communicated via the `ETag` HTTP header (by the server).
- Enables efficient resource updates where no data is transfered if the resource doesn't change.

The following example will illustrate this. 90 seconds after the initial fetch of an asset, initiates the browser a new request (the exact same asset). The browser looks up the local cache and finds the previously cached response but cannot use it because it's expired. This is the point where the browser requests the full content from the server. The problem with it this is that if the resource hasn't changed, there is absolutely no reason for downloading the same asset that is already in the CDN cache.

Validation tokens are solving this problem. The edge server creates and returns arbitrary tokens, that are stored in the `ETag` header field, which are typically a hash or other fingerprints of content of existing files. Clients don't need to know how the tokens are generated but need to send them to the server on subsequent requests. If the tokens are the same then resources haven't changed thus downloads can be skipped.

The web browser provides the `ETag` token automatically within the `If-None-Match` HTTP request header. The server then checks tokens against current assets in the cache. A `304 Not Modified` response will tell the browser if an asset in the cache hasn't been changed and therefore allowing a renewal for another 90 seconds. It's important to note that these assets don't need to be downloaded again which **saves bandwidth and time**.

How do web developers benefit from efficient revalidation?

Browsers do most (if not) all the work for web developers. For instance, they automatically detect if validation tokens have been previously specified and appending them to outgoing requests and updating cache timestamps as required based on responses from servers. Web developers are therefore left with one job only which is ensuring servers provide the required `ETag` tokens. KeyCDN's edge servers fully support the `ETag` header.

`Last-Modified`

The `Last-Modified` header indicates the time a document last changed which is the most common validator. It can be seen as a legacy validator from the time of HTTP/1.0. When a cache stores an asset including a `Last-Modified` header, it can utilize it to query the server if that representation has changed over time (since it was last seen). This can be done using an `If-Modified-Since` request header field.

An HTTP/1.1 origin server should send both, the `ETag` and the `Last-Modified` value. More details can be found in section 13.3.4 in the RFC2616.

KeyCDN example response header:

```
HTTP/1.1 200 OK
Server: keycdn-engine
Date: Mon, 27 Apr 2015 18:54:37 GMT
Content-Type: text/css
Content-Length: 44660
Connection: keep-alive
Vary: Accept-Encoding
Last-Modified: Mon, 08 Dec 2014 19:23:51 GMT
ETag: "5485fac7-ae74"
Cache-Control: max-age=533280
Expires: Sun, 03 May 2015 23:02:37 GMT
X-Cache: HIT
X-Edge-Location: defr
Access-Control-Allow-Origin: *
Accept-Ranges: bytes
```

You can check your HTTP Cache Headers using KeyCDN's HTTP Header Checker tool.

5. Extension `Cache-Control` directives

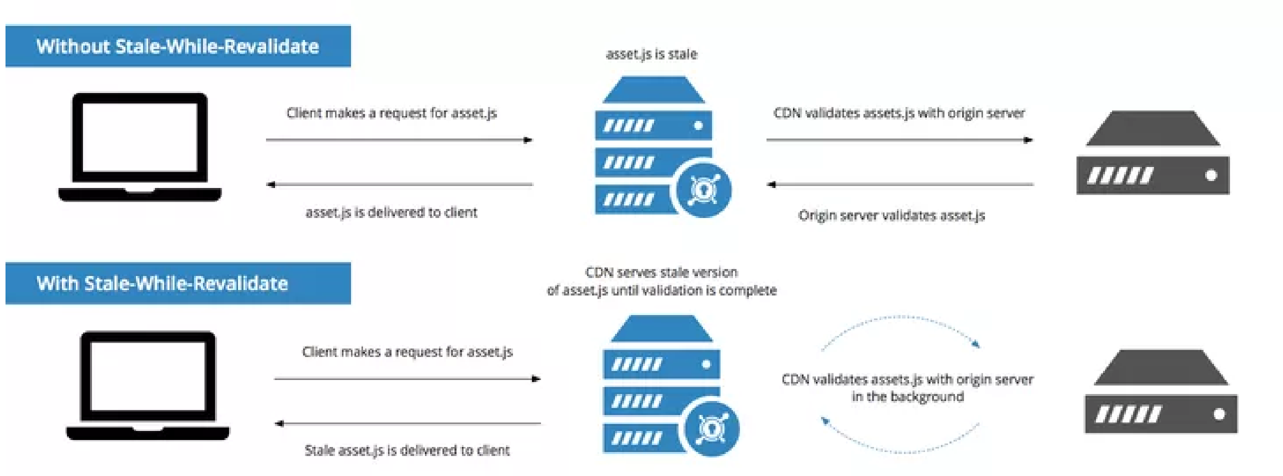
Apart from the well-known `Cache-Control` directives outlined in the first section of this article, there also exists other directives which can be used as extensions to `Cache-Control` resulting in a better user experience for your visitors.

`immutable`

No conditional revalidation will be triggered even if the user explicitly refreshes a page. The immutable directive tells the client that the response body will not change over time, therefore, there is no need to check for updates as long as it is unexpired.

`stale-while-revalidate`

The `stale-while-revalidate` directive allows for a stale asset to be served while it is revalidated in the background.



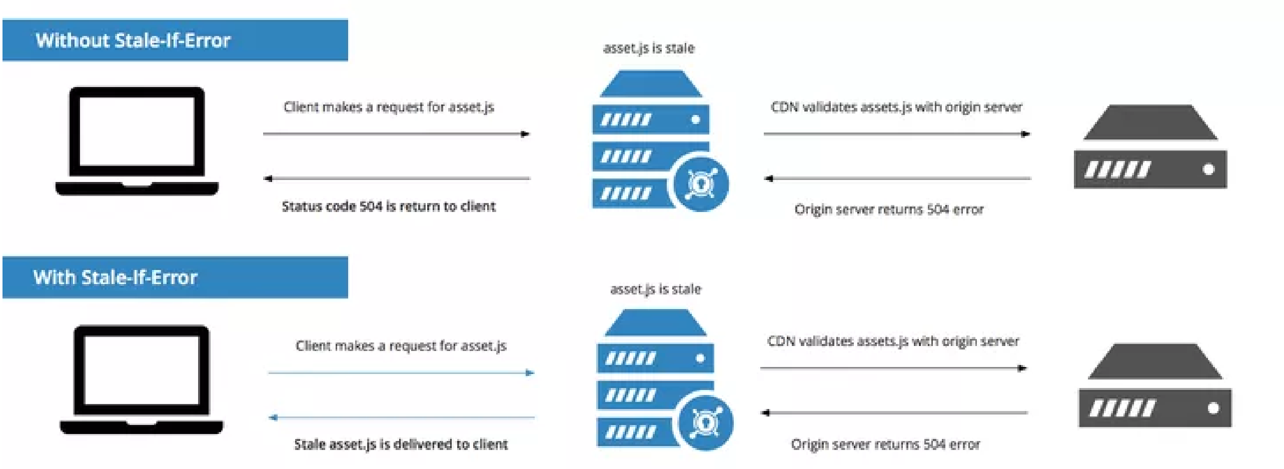
A `stale-while-revalidate` value is defined to tell the cache that it has a certain amount of time to validate the asset in the background while continuing to deliver the stale one. An example of this would look like the following:

```
Cache-Control: max-age=2592000, stale-while-revalidate=86400
```

Learn more about the `stale-while-revalidate` directive in our `stale-while-revalidate` and `stale-if-error` guide.

`stale-if-error`

The `stale-if-error` directive is very similar to the `stale-while-revalidate` directive in that it serves stale content when the `max-age` expires. However, the `stale-if-error` only returns stale content if the origin server returns an error code (e.g. `500`, `502`, `503`, or `504`) when the cache attempts to revalidate the asset.



Therefore, instead of showing visitors an error page, stale content is delivered to them for a predefined period of time. During this time it is the goal that the error has been resolved and that the asset can be revalidated.

KeyCDN and HTTP cache headers

At KeyCDN, we understand the importance of HTTP cache headers and their role in optimizing website performance. KeyCDN allows you define your HTTP cache headers as you see fit. The ability to set the Expire and Max Expire values directly within the dashboard makes it very easy to configure things on the CDN side.

Furthermore, if you rather have even more control over your HTTP cache headers you can disable the Ignore Cache Control feature in your Zone settings and have KeyCDN honor all of your cache headers from the origin. This is very useful in the event that you need to exclude a certain asset or group of assets from the CDN.

TL;DR

The `Cache-Control` (in particular), along with the `ETag` header field are modern mechanisms to control freshness and validity of your assets. The other values are only used for backward compatibility.

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.

http

```
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`.

http

```
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.

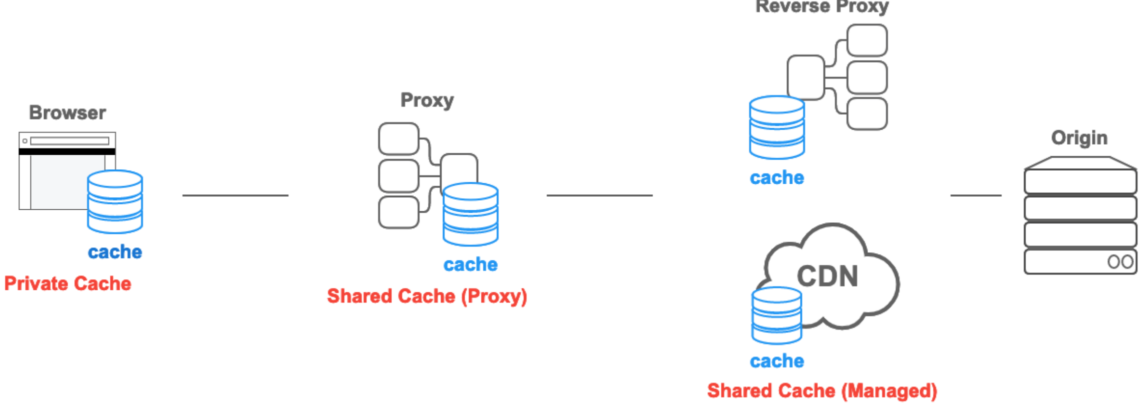
It is also possible to ignore the standard HTTP Caching spec protocols in favor of explicit manipulation. For example, the following can be specified to opt-out of a private cache or proxy cache, while using your own strategy to cache only in a managed cache.

http

```
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.



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

```
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

```
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

```
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.

http

```
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.

http

```
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

```
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.

http

```
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

```
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

```
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:

http

```
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.

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

```
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>
…
```

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

http

```
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`.

http

```
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.

http

```
Cache-Control: private
```

In such a case, even if `no-store` is given, `private` must also be given.

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.

http

```
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.

http

```
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`:

http

```
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:

http

```
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):

js

```
// 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:

http

```
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`):

js

```
// 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.

http

```
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 basically no way to delete responses that have already been stored with a long `max-age`.

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

http

```
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 that is usually difficult to intentionally do for many clients.

There is also a specification for a `Clear-Site-Data: cache` header and value, but not all browsers support it — and even when it's used, it only affects browser caches and has no effect on intermediate caches.

Therefore, it should be assumed that any stored response will remain for its `max-age` period 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.

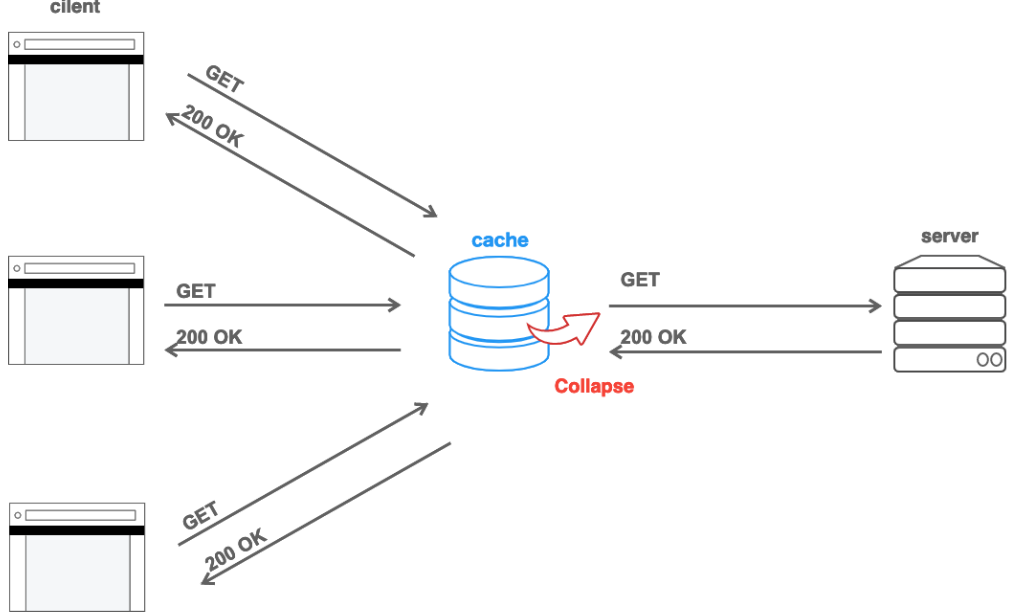
Request collapse

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:



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`:

http

```
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:

http

```
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:

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.

html

```
<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.

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.

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).

http

```
# 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.

http

```
# 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.

http

```
# bundle.v123.js
HTTP/1.1 200 OK
Content-Type: application/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.

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.

html

```html
<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

```http
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`:

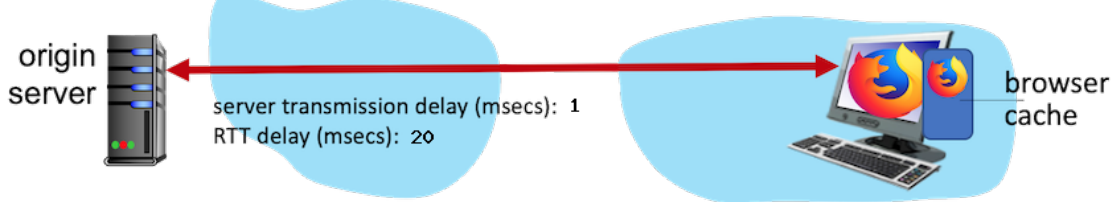
http

```http
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.

Browser Caching

Consider an HTTP server and client as shown in the figure below. Suppose that the RTT delay between the client and server is 20 msecs; the time a server needs to transmit an object into its outgoing link is 1 msecs; and any other HTTP message not containing an object has a negligible (zero) transmission time. Suppose the client again makes 60 requests, one after the other, waiting for a reply to a request before sending the next request.



Assume the client is using HTTP 1.1 and the IF-MODIFIED-SINCE header line. Assume 30% of the objects requested have NOT changed since the client downloaded them (before these 60 downloads are performed)