Effective Cache Control

Posted on December 19, 2016 | 10 minutes | Kevin Sookocheff

The web supports a global network of billions of devices and users, and a key component of this support is effectively caching frequently accessed data along the request-response path. One of the key benefits in adopting a REST architectural style for your system is being able to leverage this existing infrastructure, taking advantage of the billions of dollars of investment already made in the web, and promoting loose coupling, performance and scalability.

Caching from 10,000 feet

Whenever a consumer requests a resource, the request goes through a series of caches, eventually terminating at the service hosting the resource. If any of these caches can satisfy the consumer request, the cache uses its local copy, saving the origin server and downstream caches from the burden of processing the request. For example, in the following figure, a consumer makes a request for a resource that is served by a server. In between the consumer and the server are a series of caches. Any one of these caches may satisfy the consumer's request without the request hitting the origin server.

(LayeredCaches.png)

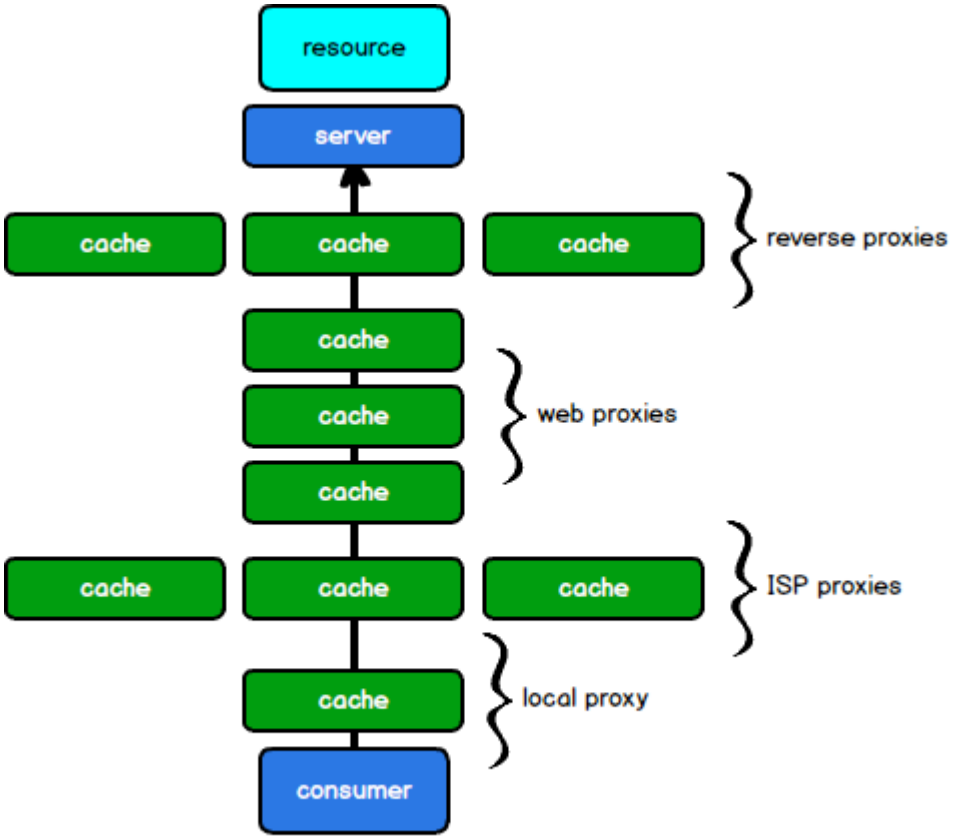

Figure 1. Web caches. REST In Practice, 2010.

Effective caching allows you to optimize this request-response path in a number of ways.

1. Reducing bandwidth by reducing the number of network hops required to satisfy a request.
2. Reduce latency by satisfying requests closer to the client.
3. Reducing server load by allowing caches to serve requests.

4. Increasing robustness by allowing caches to serve content, even if the origin server is unavailable.

To take advantage of these benefits, the origin server must choose effective caching behaviour using HTTP headers. By setting appropriate headers, the server indicates which responses can be cached, by whom, and for how long.

Making Responses Cacheable

By default, GET responses with a status code of 200, 203, 206, 300, 301 or 410 may be stored by a cache. The origin server can override this default setting appropriate `Expires` and `Cache-Control` headers.

Expires Header

The `Expires` header tells the cache whether they should serve a request using a cached copy of the response, or whether they should retrieve an updated response from the origin server. This header is specified as an absolute expiry time for a cached representation. Beyond that time, the cached representation is considered stale and must be revalidated by the origin server.

Generally speaking, you set late expiry times (up to one year in the future) for resources that do not change frequently, and set short expiry times for resources that change frequently. To indicate that a resource representation has already expired, set the `Expires` header equal to the `Date` header, or to 0.

As a rule, if you can determine an absolute expiry time for a response, set the `Expires` header appropriately.

Cache-Control Header

The `Cache-Control` header determines whether a response is cacheable, by whom, and for how long. The origin server specifies this behaviour using *directives*. A directive can make a normally uncacheable response cacheable, or cacheable responses uncacheable.

- `public` : indicates that the response may be cached, even if it would normally be non-cacheable.
- `private` : indicates that the response may be cached by local (typically browser) caches only.

- `no-cache` : indicates that caches must revalidate responses with the origin server on every request.
- `no-store` : indicates that content is uncacheable by *all* caches.

In addition to controlling whether or not a response is cacheable, the `Cache-Control` header allows you to set the expiration time of a resource.

- `max-age` : gives a time to live of the resource in seconds, after which any local and shared caches must revalidate the response.
- `s-maxage` : gives a time to live of the resource in seconds, after which any shared caches must revalidate the response.

Lastly, the `Cache-Control` header can specify when a response must be revalidated by the origin server. This can be enormously useful in balancing strong consistency with reducing bandwidth and latency.

- `must-revalidate` : indicates a normally uncacheable response is cacheable, but requires a cache to revalidate stale responses before using a cached response. This forces revalidation requests to travel all the way to the origin server, but an efficient validation mechanism on the server will prevent complex service logic from being invoked on each request.
- `proxy-revalidate` : similar to `must-revalidate` , but applied only to shared caches.
- `stale-while-revalidate` : allows a cache to serve a stale response while a revalidation happens in the background. This directive favors reduced latency over consistency of data by allowing a cache to serve stale data while a non-blocking request to revalidate happens in the background.
- `stale-if-error` : allows a cache to serve a stale response if there is an error during revalidation. This directive favors availability over consistency by allowing a cache to return stale data during origin server failure.

Validators

Cacheable responses should also include a validator, either in the `ETag` or `Last-Modified` header.

- `ETag` : An Etag, or entity tag, is an opaque token that the server associates with a particular state of a resource. Whenever the resource state changes, the entity tag should be changed accordingly.
- `Last-Modified` : The Last-Modified header indicates the last point in time when the resource was changed.

To revalidate a response with the origin server, a cache uses the value in the Etag or Last-Modified header to do a conditional GET. In a conditional GET, the server may respond with an HTTP 304 response, with no response body, indicating that a resource has not been modified. In this case, the cache is free to serve a cached response and save the server from having to recompute a value and then send the entire response back over the network.

Cache Control In Practice

We've covered a lot of terminology. Let's take a step back and look at how to set response headers in practice.

Basic Cache Control

The first example shows a request for a spreadsheet with id `9879`, and a response that contains an `Expires` header.

Request:

```
GET /spreadsheets/9879
HOST: wdesk.com
```

Response:

```
HTTP/1.1 200 OK
Content-Length: ...
Content-Type: application/json
Date: Fri, 25 Nov 2016 13:59:12 GMT
Expires: Sat, 25 Nov 2016 13:59:12 GMT

{
    ...
}
```

In the response headers, the server signals that this response can be cached until the date and time specified in the `Expires` header. The response will travel back to the client through proxies and will be cached by those proxies according to the `Expires` header. Any subsequent requests for this same resource that happen before

this time will be served the cached response, any requests after this time will travel all the way to the origin server, where a fresh copy of the data is returned.

The second example shows a response from a GET request that contain a `Cache-Control` header.

Request:

```
GET /spreadsheets/9879
HOST: wdesk.com
```

Response:

```
HTTP/1.1 200 OK
Content-Length: ...
Content-Type: application/json
Date: Fri, 25 Nov 2016 13:59:12 GMT
Cache-Control: max-age=3600

{
    ...
}
```

The `Cache-Control` header signals that this response can be cached for up to 3600 seconds (60 minutes). After this time period has elapsed, the cache should drop this data from its local store and revalidate that data on subsequent requests for the same resource.

Conditional Requests

In both of these cases, if a user requests a resource that is in the cache, and the cached response has expired (older than its `max-age` or past the `Expires` date), then the cache makes a *conditional request* to the origin server to determine if the cached response is still valid and can be reused. A conditional request attempts to save bandwidth by exchanging only HTTP headers rather than headers and entity bodies.

Conditional requests are handled in one of two ways, depending on whether the response was cached with a `Last-Modified` header or an `ETag` header. In either case, the cache effectively asks the origin server, "has the response been modified since the last time you gave it to me"? If the resource has not changed, the origin server responds with a `304 Not Modified` response, if necessary specifying new `Expires` and `Cache-Control` headers to replace those set by the cache. If the resource has changed, the origin server responds with a `200 OK` status and the full response representation in the payload body.

The following exchange shows a GET request returning a `200 OK` status and payload body, followed by a conditional GET resulting in a `304 Not Modified` response. The conditional GET request sets the `If-Modified-Since` header to the value the client received in the `Last-Modified` header of the response. Since the resource has not been modified since that point in time, the origin server responds with `304 Not Modified`.

Request:

```
GET /spreadsheets/1234 HTTP/1.1
Host: wdesk.com
```

Response:

```
HTTP/1.1 200 OK
Content-Length: ...
Content-Type: application/json
Date: Fri, 25 Nov 2016 06:24:22 GMT
Last-Modified: Fri, 25 Nov 2016 06:24:22 GMT
ETag: 74f4be4b

{
    ...
}
```

Request:

```
GET /order/1234 HTTP/1.1
Host: wdesk.com
If-Modified-Since: Fri, 25 Nov 2016 06:24:22 GMT
```

Response:

```
HTTP/1.1 304 Not Modified
```

In this exchange, the server has signalled that the data has not changed, and that the proxy can serve the cached content. By using a conditional request, the cache and server save the network cost of transmitting the full payload body back to the cache. It may also save the server from complex processing required to generate a full response body.

The following exchange follows the same pattern, using the `If-None-Match` header and an `ETag` value.

Request:

```
GET /spreadsheets/1234 HTTP/1.1
Host: wdesk.com
```

Response:

```
HTTP/1.1 200 OK
Content-Length: ...
Content-Type: application/json
Date: Fri, 25 Nov 2016 06:24:22 GMT
Last-Modified: Fri, 25 Nov 2016 06:24:22 GMT
Expires: Sat, 25 Nov 2016 13:59:12 GMT
ETag: 74f4be4b

{
    ...
}
```

Request:

```
GET /order/1234 HTTP/1.1
Host: wdesk.com
If-None-Match: "74f4be4b"
```

Response:

```
HTTP/1.1 304 Not Modified
```

In this exchange, the server has signalled that the data has not changed, because the ETag header associated with the resource has not changed. Again, the cache and server save the network cost of transmitting the full payload body back to the cache, and the processing cost of generating a full response body.

Using Cache-Control Directives

The `Cache-Control` header allows you to mix and match different cache control directives to achieve a desired affect. For example, each of the following are valid Cache-Control values.

Response can be cached by browser and any intermediary caches for up to 1 day:

```
Cache-Control: max-age=86400
```

Response can be cached by browsers for up to 10 seconds:

```
Cache-Control: private, max-age=10
```

Response can be cached by browsers for up to 10 seconds and by intermediary caches for up to 300 seconds:

```
Cache-Control: max-age=10, s-maxage=300
```

Response can be cached by browsers for up to 10 seconds, by any intermediary caches for up to 300 seconds, and shared caches must revalidate every request with the origin server,:

```
Cache-Control: max-age=10, s-maxage=300, proxy-revalidate
```

Handling Authentication

By default, any request-response pair with an `Authorization` header is not cacheable, meaning for most API endpoints, responses are not cached. However, all is not lost. We can still allow caches to store authorized responses, while requiring the origin server to revalidate authorization credentials on each request. How? The

`public` directive makes a response cacheable by both local and shared caches, and `max-age=0` requires a cache to revalidate a cached representation (using a conditional GET) before releasing it.

Response can be cached by browsers and intermediary caches, but must be revalidated by the origin server before being released to the client:

```
Cache-Control: public, max-age=0
```

When revalidating the request, the cache will include the `Authorization` header supplied by the client. If the origin server responds with `401 Unauthorized`, the cache will not release the cached resource representation. This combination is useful if you want to authorize every request, but still want to take advantage of the web's caching infrastructure for performance, scalability, and resilience.

Effective Caching

By applying effective caching to your resources, you can take advantage of all of the web's existing infrastructure to improve the performance and scalability of your application. This article provides an overview of each of the cache control mechanisms available to a server, and how to use them. Consider how caching can affect your application, and design your application to harness the power of the web.

Tags: #http (https://sookocheff.com/tags/http/) #caching (https://sookocheff.com/tags/caching/) #cache control (https://sookocheff.com/tags/cache-control/)

See also

- Optimistic Locking in a REST API (/post/api/optimistic-locking-in-a-rest-api/)

← PREVIOUS POST (HTTPS://SOOKOCHEFF.COM/POST/API/REST-ASSURED-TESTS-AGAINST-API-GATEWAY/)

NEXT POST → (HTTPS://SOOKOCHEFF.COM/POST/API/CHECKING-FOR-NULL-IN-API-GATEWAY/)

 (mailto:kevin@sookocheff.com)  (https://www.facebook.com/soofaloofa)  (https://github.com/soofaloofa)

 (https://twitter.com/soofaloofa)  (https://linkedin.com/in/kevinsookocheff)

 (https://paypal.me/paypal.me/soofaloofa)  ()

Hugo v0.58.1 (https://gohugo.io) powered • Theme Beautiful Hugo (https://github.com/halogenica/beautifulhugo) adapted from Beautiful Jekyll (https://deanattali.com/beautiful-jekyll/)