Working with HTTP cache

briantorti.com/working-with-http-cache

September 27, 2014

The fastest network request is a request not performed. That's the job a HTTP cache: avoid unnecessary work. By understanding how it works, we can create web applications and APIs that are more responsive, by reducing the latency and the amount of used bandwidth.

There are two main types of cache: The *private* and the *shared*.

A private cache is what the web browser (or any other HTTP agent) stores locally, in each client's computer.
A shared cache is something that sits between the client and the origin server, and can serve multiple clients. It acts as a proxy, that intercepts requests and decides if the origin server needs to be called.

There are two aspects of a request that are analysed before asking a new version of a representation: **Freshness** and **validity**.

Freshness

When a representation that is stored in the cache is considered fresh, there is no need to even perform a request to the origin server, it can be served right away.
There are two HTTP headers used to indicate if a representation is fresh or not: `Expires` and `Cache-Control`.

Expires

The `Expires` header indicates when that representation should be considered stale (not fresh). It expects a specific HTTP date. Here's an example:

```
HTTP/1.1 200 OK
Content-Length: 31225
Content-type: text/html
Expires: Mon, 29 Sep 2014 10:00:00 GMT

[RESPONSE BODY]
```

Notice that if the date format is not correct, it will be considered stale. Also, you need to make sure that your web server clock and the cache are synchronized.

Cache-Control

In HTTP 1.1, the `Cache-Control` header is an alternative to `Expires`. If both the `Expires` and `Cache-Control` headers are found, `Expires` will be ignored.

`Cache-Control` works with a bunch of directives to specify how it should behave. We will talk about three of them: `max-age`, `private` and `no-cache`. You can see the entire list here.

max-age: This directive specifies for how many seconds (from the request time) the representation should be considered fresh. It works like the `Expires` header, but without the date issues.

private: Allows just a private cache to store it, but never a shared cache. This directive is used when the response is intended for a single user, so it makes no sense to store it in a shared cache.

no-cache: As the name says, it makes the request always be sent to the origin server.

Here's an example:

```
HTTP/1.1 200 OK
Content-Length: 31225
Content-Type: text/html
Cache-Control: max-age=3600; private

[RESPONSE BODY]
```

Validation

When a representation is considered stale (e.g. the `max-age` was exceeded), a request must be sent to the origin server. Although we need to pay the price of a network request, if we can identify that the representation is still the same, we can save some bandwidth by not sending this representation again. That's the job of the validation process, and this is done with what is called a **conditional request**.

There are two headers that can be used to support conditional requests, `Last-Modified` and `Etag`.

Last-Modified

The `Last-Modified` header contains a date that tells the client when this representation last changed.

```
HTTP/1.1 200 OK
Content-Length: 44181
Content-type: text/html
Last-Modified: Sun, 28 Set 2014 10:00:00 GMT

[RESPONSE BODY]
```

When a client receives a response that includes a `Last-Modified` header, it takes note of that, and, when it needs to perform the same request again, it includes a `If-Modified-Since` in the request headers, with the date that it received before:

```
GET / HTTP/1.1
If-Modified-Since: Sun, 28 Set 2014 10:00:00 GMT

[REQUEST BODY]
```

The origin server then checks if the representation was changed after the date received in the `If-Modified-Since` header, and, if it was not changed, it just sends a `304 Not Modified` response:

```
HTTP/1.1 304 Not Modified
Content-Length: 0
Last-Modified: Sun, 28 Set 2014 10:00:00 GMT
```

Even thought we still had to perform a network request, we avoid sending the same representation in the body, saving some bandwidth.

Etag

This is an “entity tag” that contains a string that changes whenever the representation changes. Usually a MD5 hash is used but it can be whatever you want.
It will work in the same way `Last-Modified` does. The benefit is that you don’t need to keep track of the modification date of a representation, as long as you use always the same algorithm to generate the `Etag` value (and you should be using), it can be regenerated when you need.

```
HTTP/1.1 200 OK
Content-Length: 44181
Content-type: text/html
Etag: "78q9y7-b37r-0o9a3bc"

[RESPONSE BODY]
```

The client will save this `Etag` value and send it back in a `If-None-Match` header for the next requests.

```
GET / HTTP/1.1
If-None-Match: "78q9y7-b37r-0o9a3bc"

[REQUEST BODY]
```

Then, if the origin server determines that the received value is still the same for the generated representation, it can just sent a `304 Not Modified`, saving some bandwidth.

HTTP cache at work, step by step

Putting the pieces together, we can have this scenario:

1) A request is performed to `/`;

2) The HTTP agent checks if there is a `fresh` copy of the requested representation. It does so my looking at the `Cache-Control` or `Expires` headers. If it finds a fresh copy, it just serves it to the client, and the origin server won't even know this request existed.

3) If a `fresh` copy is not found, the origin server will be asked to revalidate the representation, through a **conditional request**. This is done with the `If-Modified-Since` and/or `If-None-Match` headers.

4) If the origin server can validate the request, it will just return a `304 Not Modified` response, and the client will keep using the representation it already has stored.

HTTP cache at work, a practical example

To understand better this scenario, we will create a simple API that will incrementally add some cache capability.
I am going to use sinatra to create this API, and rack-cache as a reverse proxy cache. The same concepts could be applied with any other stack, I choose these two tools because they are pretty simple and won't get in our way to understand how the cache is working, as this is our goal here.

First, install `sinatra` and `rack-cache`, in case you don't have them installed already:

```
gem install sinatra rack-cache
```

Then, we will create a simple `sinatra` app, without any caching capability:

```
# server.rb

require 'sinatra'

set :port, 1234

get '/' do
  # some interesting code would be executed here
  # for now, we are just sleeping for 5 seconds
  sleep 5

  "the resource representation"
end
```

To run this server, just run `ruby server.rb`. It should be accessible at `http://localhost:1234`. Notice that you'll need to kill and start the server again after each change.

When we send a request to this endpoint, you will notice that it will take 5 seconds until we get a response back. To create this request, I'm going to use `curl(1)` (with the `-i` parameter, so we can see the headers).

```
$ curl -i http://localhost:1234

HTTP/1.1 200 OK
Content-Type: text/html;charset=utf-8
Content-Length: 27
X-Xss-Protection: 1; mode=block
X-Content-Type-Options: nosniff
X-Frame-Options: SAMEORIGIN
Server: WEBrick/1.3.1 (Ruby/2.1.2/2014-05-08)
Date: Sun, 28 Sep 2014 19:54:16 GMT
Connection: Keep-Alive

the resource representation
```

We can see that there's no cache-related header in this response. Every time we send this request, it'll hit the origin server, and we'll have to wait at least 5 seconds to get the response. Also, we are always receiving the response body, even if it didn't change, causing unnecessary use of bandwidth.

So let's start to fix this.

First, we are going to add `rack-cache` as our reverse proxy cache. It should be pretty simple, as it's just a rack middleware:

```
# server.rb

require 'sinatra'

# we require rack-cache
require 'rack-cache'

set :port, 1234

# and start using it
use Rack::Cache

get '/' do
  sleep 5

  "the resource representation"
end
```

Now that we have `rack-cache` in place, we can start to take advantage of it. First we'll add a `Cache-Control` header, that is going to tell the client that this representation should be considered fresh for 10 seconds:

```
# server.rb

require 'sinatra'
require 'rack-cache'

set :port, 1234

use Rack::Cache

get '/' do
  sleep 5

  # add a Cache-Controller header, setting the max-age to 10 seconds
  cache_control :public, max_age: 10
  "the resource representation"
end
```

And that's it. If you try to hit this endpoint again, here's what you get:

```
$ curl -i http://localhost:1234

HTTP/1.1 200 OK
Content-Type: text/html;charset=utf-8
Cache-Control: public, max-age=10
Content-Length: 27
Date: Sun, 28 Sep 2014 20:17:05 GMT
X-Content-Digest: 904c355ca45f6806b252aa62329fa8ac149011ac
Age: 0
X-Rack-Cache: stale, invalid, store
X-Xss-Protection: 1; mode=block
X-Content-Type-Options: nosniff
X-Frame-Options: SAMEORIGIN
Server: WEBrick/1.3.1 (Ruby/2.1.2/2014-05-08)
Connection: Keep-Alive

the resource representation
```

Now that we have the header `Cache-Control` in place, the next request should return instantaneously, as it's not hitting the origin server. That's all it takes to have the `freshness` process working. The next step is the `validation` process, and it is almost as easy.

So we are already saving some network traffic by avoiding unnecessary requests while the representation is still fresh, but once it gets stale, we are still retrieving the entire representation in the response body, even if it didn't change at all. Let's fix that.

```
# server.rb

require 'sinatra'
require 'rack-cache'

set :port, 1234

use Rack::Cache

get '/' do
  sleep 5

  representation = "the resource representation"

  cache_control :public, max_age: 10
  # we add the Etag header with a MD5 hash of
  # the representation
  etag Digest::MD5.hexdigest(representation)

  representation
end
```

Now, performing the same request to this endpoint, when the representation is stale (10 seconds after the first request), this is what we get:

```
$ curl -i http://localhost:1234

HTTP/1.1 200 OK
Content-Type: text/html;charset=utf-8
Cache-Control: public, max-age=10
Etag: "f8d36c97fa01826fe14c1989e373d6e4"
Content-Length: 27
Date: Sun, 28 Sep 2014 20:29:48 GMT
X-Content-Digest: 904c355ca45f6806b252aa62329fa8ac149011ac
Age: 0
X-Rack-Cache: miss, store
X-Xss-Protection: 1; mode=block
X-Content-Type-Options: nosniff
X-Frame-Options: SAMEORIGIN
Server: WEBrick/1.3.1 (Ruby/2.1.2/2014-05-08)
Connection: Keep-Alive

the resource representation
```

We can see the `Etag` header there. All we need to do is to save that value, and send it in the `If-None-Match` header for the next request:

```
curl -i http://localhost:1234 --header 'If-None-Match:
"f8d36c97fa01826fe14c1989e373d6e4"'

HTTP/1.1 304 Not Modified
Cache-Control: public, max-age=10
Etag: "f8d36c97fa01826fe14c1989e373d6e4"
X-Content-Digest: 904c355ca45f6806b252aa62329fa8ac149011ac
Date: Sun, 28 Sep 2014 20:31:02 GMT
Age: 0
X-Rack-Cache: stale, valid, store
X-Content-Type-Options: nosniff
Server: WEBrick/1.3.1 (Ruby/2.1.2/2014-05-08)
Connection: Keep-Alive
```

Now, instead of getting a `200 OK` response, with the entire representation in the body, we get a `304 Not Modified`, that does not include a body message. That saves us some bandwidth, as we don't need to send that entire representation, that can be pretty big, in the response.

Conclusion

In a time where performance is a feature, doing good use of HTTP caching is one of the simplest ways to create applications and APIs that are more responsive. With the tools that we have available today, it's becoming easier and easier to use these well-established HTTP capabilities, but understanding how they work is the first step, as none of these tools will be able to understand your specific requirements.