thoughtbot

Introduction to Conditional HTTP Caching with Rails

Damian Galarza – November 25, 2014 UPDATED ON March 23, 2019

WEB, RAILS, HTTP, PERFORMANCE

HTTP provides developers with a powerful set of tools to cache responses. Often times we don't want a client to blindly cache content that it has been given. We may not be able to rely on setting specific expiration headers either. Instead we need a way for the client to ask the server whether or not a resource has been updated.

HTTP provides us with the ability to do this with conditional caching. A client can make a request to the server and find out of the server has a new version of the resource available.

Rails provides us with tools to take advantage of this. Before we jump into Rails, let's discuss some of the common ways to check if a client's cache is fresh or stale.

ETags

ETags, short for entity tags, are a common way to conditionally verify an HTTP cache. An ETag is a digest which represents the contents of a given resource.

When a response is returned by the server it will include an ETag to represent the resource's state as part of the HTTP response headers. Subsequent HTTP requests which want to know whether or not the resource has changed since the last request can send along the stored ETag via the `If-None-Match` header.

respond with a “304 Not Modified” status.

If the resource has changed since the last time the client has requested the resource the server will respond with a new ETag and the updated response.

Last-Modified Timestamp

Another header that a server can supply is the `Last-Modified` header.

As the name implies, this will return a timestamp of when the resource was last modified. The client can then include an `If-Modified-Since` header with the `Last-Modified` timestamp value to ask the server if the resource has been modified since the previous request.

Again, if the resource has not been modified since the last request, the server can respond with a “304 Not Modified” status. Otherwise, it will return an updated representation of the resource and a new `Last-Modified` timestamp for next time.

Caching with fresh_when

Now that we’ve discussed the ETags and Last-Modified HTTP headers, let’s take a look at how we might use them within our Rails application.

ActionController provides us with a powerful method called `fresh_when`. It will use either an ETag or a Last-Modified timestamp to determine whether or not a resource is fresh or stale.

Imagine a simple Rails blog application with the following posts route / controller:

```
# app/controllers/posts_controller.rb
class PostsController < ApplicationController
  def show
```

```
# config/routes.rb
Rails.application.routes.draw do
  resources :posts
end
```

If we made a curl request to an individual post we might see something like:

```
curl -i http://localhost:3000/posts/1
```

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8
Etag: "4af7e17fc369c6d1af99f8994d8fd387"
Server: WEBrick/1.3.1 (Ruby/2.1.2/2014-05-08)

<!DOCTYPE html>
<html>
  <body>
    <h1>Blog post 1</h1>
    <p>lorem ipsum</p>
  </body>
</html>
```

Notice that we have not done anything particularly noteworthy, yet if we take a look at the headers returned as a part of the response we'll see that Rails has sent along an ETag to represent the resource being returned.

If we keep making the same request via curl we'll see a different ETag value even though the resource has not been modified since our previous request. This goes against the notion of what an ETag would be used for.

We'll need to modify our controller to instruct it that a request is fresh when a post has not been modified since the last request.

```
  def show
    @post = Post.find(params[:id])

    fresh_when @post
  end
end
```

Now if we make our previous request over again we'll see that our ETag no longer changes on repeated requests:

```
curl -i http://localhost:3000/posts/1
```

```
Content-Length: 667
Content-Type: text/html; charset=utf-8
Etag: "534279bfc931d4236713095ffd3efb28"
Last-Modified: Wed, 12 Nov 2014 15:44:46 GMT
Server: WEBrick/1.3.1 (Ruby/2.1.3/2014-09-19)
```

Upon closer inspection of the updated HTTP headers returned in the response, we'll notice that the server has also started returning a `Last-Modified` timestamp for the post resource.

Our work is not complete.

In order to use our cache we need to verify it against the server with the ETag or last modified timestamp with either the `If-None-Match` or `If-Modified-Since` headers respectively.

```
curl -i -H 'If-None-Match: "534279bfc931d4236713095ffd3efb28"' http://
```

```
Last-Modified: Wed, 12 Nov 2014 15:44:46 GMT
Server: WEBrick/1.3.1 (Ruby/2.1.3/2014-09-19)
```

Notice we now receive a 304 Not Modified response, instructing our client that it can use its cached version of the content. Browsers will automatically send a long a stored ETag and Last-Modified timestamp with conditional caching headers for us so we do not actually have to do anything to use this in the browser.

Let's take a look at the similar request with the `If-Modified-Since` header:

```
curl -i -H 'If-Modified-Since: Wed, 12 Nov 2014 15:44:46 GMT' http://l
```

```
HTTP/1.1 304 Not Modified
Etag: "534279bfc931d4236713095ffd3efb28"
Last-Modified: Wed, 12 Nov 2014 15:44:46 GMT
Server: WEBrick/1.3.1 (Ruby/2.1.3/2014-09-19)
```

You'll notice that we get the same results, the post has not been modified since the last time and we receive a 304 Not Modified response.

Conclusion

What have we gained through all of this? We are still making requests to our server to see if a resource has been modified before rendering.

To really understand what we've achieved we can take a look at our Rails log before we had caching:

```
Started GET "/posts/1" for 127.0.0.1 at 2014-11-12 11:33:57 -0500
Processing by PostsController#show as */*
```

```
  Rendered posts/show.html.erb within layouts/application (0.1ms)
  Rendered application/_flashes.html.erb (0.0ms)
  Rendered application/_analytics.html.erb (0.1ms)
  Rendered application/_javascript.html.erb (16.1ms)
Completed 200 OK in 35ms (Views: 33.4ms | ActiveRecord: 0.3ms)
```

We can see that our request takes about 33ms and has to render our posts/show template within the layout along with some other partials. We can also see that most of our time is spent within that rendering process. ActiveRecord took less than a millisecond to run.

Next, let's take a look at our log with conditional caching in place, and put it to use by sending over a `If-None-Match` header:

```
Started GET "/posts/1" for 127.0.0.1 at 2014-11-12 11:39:52 -0500
Processing by PostsController#show as */*
  Parameters: {"id"=>"1"}
  Post Load (0.3ms)  SELECT  "posts".* FROM "posts"  WHERE "posts"."id
LIMIT 1  [["id", 1]]
Completed 304 Not Modified in 2ms (ActiveRecord: 0.3ms)
```

Notice we've cut off our entire rendering process from the request. Since the client's cached response is fresh there is no reason for Rails to go through the rendering process. It can load in the post via ActiveRecord and verify if it has changed since the previous request.

Our requests went from taking about 33ms to complete to 2ms. That's about a 94% improvement in performance. Not only is Rails skipping the rendering process, but our responses are now much lighter as they have no body to return.

This may not be much in our trivial example but for applications which have a lot of complicated logic behind rendering this can save lots of time.

utilized by the user's browser.

Getting more

Our simple `fresh_when` solution is not without its limitations.

It cannot handle user specific content. If we wanted to display anything different on a per-user basis (even if that's the name in the header), we won't be able to cache the content.

Another issue is that the cache is not being shared among users. Our performance gain is only realized when a single user looks at the same post over and over again.

Lastly, we're limited to a simple `render` call in our controller. What if we wanted something more customized like `render json:` or `render xml:`? We'll build on top of what we've learned next time to overcome these limitations and improve our caching even further.

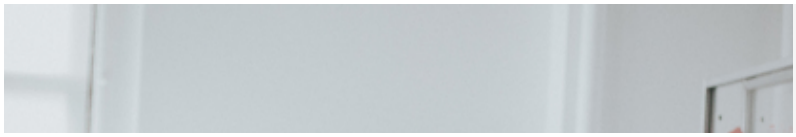
Further reading

- Learn How to Evaluate Your Rails JSON API for Performance Improvements
- Learn about "Origin Pull" CDNs and how to use them in Rails in DNS to CDN to Origin

---

If you enjoyed this post, you might also like:

Copycopter's client: so fast
Custom Formats for DateTime
Class-Based Generic Views in Django



Upgrade your codebase

thoughtbot

understand the current state of your code quality, speed up delivery times, improve developer happiness, and level up your user experience

Learn more about a Code Audit