Web Caching

Improving perceived performance by having to do less and not repeating ourselves;)

Client Caching

This section provides a demonstration of how the server can assist clients in lessening the number and frequency of calls to the server for data. It uses caching headers but does not go into detail or breadth of coverage on all options available.

Caching Headers

References:

- `HTTP Caching in Ruby with Rails`
- `Increasing Application Performance with HTTP Cache Headers`
- `Browser Cache:  How ETags Works in Rails 3 and Rails 4`

ETag and Last-Modified ETags represent a thumb print of the state of a resource referenced by a URI. What is represented in that state and how the thumb print is not always the same.

1. Using the default implementation of the controller, inspect the default caching headers.

   ```
   > response=HTTParty.head("http://localhost:3000/movies/12345")
   > pp ["cache-control","etag","last-modified"].map {|h| {h=>response.header[h]}}
   [{"cache-control"=>"max-age=0, private, must-revalidate"},
    {"etag"=>"W/\"ea6bd9165ddcfb2be59b079aee9fcfca\""},
     {"last-modified"=>nil}]
   ```

   The default ETag is generated by Rails from a hash of the returned view. If you watch the source of the pages returned, you will see a new value for the `csrf-token` when the page is created.

   ```
   <meta name="csrf-token" content="uMUgxBuVYRNKa6p9...Mg==" />
   ```

   If you look at the JSON output, it is free of additional headers and has a stable, default ETag.

   ```
   > response=HTTParty.head("http://localhost:3000/movies/12345.json")
   > pp ["cache-control","etag","last-modified"].map {|h| {h=>response.header[h]}}
   [{"cache-control"=>"max-age=0, private, must-revalidate"},
    {"etag"=>"W/\"6601839f7087fc83ba7beb9478e5c9aa\""},
     {"last-modified"=>nil}]
   ```

   However, the default ETag requires the entire action to be performed and view rendered. The ETag is based on a hash of the resulting view.

2. We could calculate an `ETag` manually using the Active Model `cache_key`

   ```
   > Movie.find("12345").cache_key
    => "movies/12345-20160112171635666000000"

   > Digest::MD5.hexdigest(Movie.find("12345").cache_key)
    => "428600c9adc835d6feaefbab903ab72e"

   def show
     headers["ETag"]=Digest::MD5.hexdigest(@movie.cache_key)
   end
   ```

Notice how that specific value (the one we derived from the database `cache_key`) was the exact value returned in the response.

```
> HTTParty.head("http://localhost:3000/movies/12345.json").headers["ETag"]
  => "428600c9adc835d6feaefbab903ab72e"
```

3. We could also do the same for `Last-Modified` and have full control over that value.

```
> Movie.find("12345").updated_at.httpdate
 => "Tue, 12 Jan 2016 17:16:35 GMT"

def show
  headers["ETag"]=Digest::MD5.hexdigest(@movie.cache_key)
  headers["Last-Modified"]=>@movie.updated_at.httpdate
end

> HTTParty.head("http://localhost:3000/movies/12345.json").headers["Last-Modified"]
 => "Tue, 12 Jan 2016 17:16:35 GMT"
```

4. However, Rails provides some convenience methods that will perform the above roles for us and come with a default set of strategies of how they are calculated. We can simply assign the headers to the outgoing response with `fresh_when`. Simply adding that call will add an `ETag` and `Last-Modified` based on data content.

```
def show
  #    headers["ETag"]=Digest::MD5.hexdigest(@movie.cache_key)
  #    headers["Last-Modified"]=@movie.updated_at.httpdate
  fresh_when(@movie)
end
```

Notice the JSON and HTML pages now share a stable, consistent value and `fresh_when` has added an additional `last-modified` header that is also based on content.

```
> response=HTTParty.head("http://localhost:3000/movies/12345"); #...
[{"cache-control"=>"max-age=0, private, must-revalidate"},
 {"etag"=>"\"2ce808f07c5ff27a4698a87d73cf0d3b\""},
 {"last-modified"=>"Tue, 12 Jan 2016 17:16:35 GMT"}]
```

Actually the consistency between the data content types does not occur until the format is removed from the URL and moved to the header `Accept` field.

```
> response=HTTParty.head("http://localhost:3000/movies/12345.json"); #...
[{"cache-control"=>"max-age=0, private, must-revalidate"},
 {"etag"=>"\"dd7543eb8124a81a065c2d0629222e2c\""},
 {"last-modified"=>"Tue, 12 Jan 2016 17:16:35 GMT"}]
```

Now we can say the JSON and HTML are representing the same state of the Movie. Rails is not using the exact algorithm that we manually used above (you can replace that), but it does show that ETag algorithms typically reflect the URI they were accessed by and sometimes who accessed them.

```
> response=HTTParty.head("http://localhost:3000/movies/12345",
       headers:{"Accept"=>"application/json"});#...
[{"cache-control"=>"max-age=0, private, must-revalidate"},
 {"etag"=>"\"2ce808f07c5ff27a4698a87d73cf0d3b\""},
 {"last-modified"=>"Tue, 12 Jan 2016 17:16:35 GMT"}]
```

Cache Revalidation Headers

With `ETag` and/or `Last-Modified` in place, we can validate whether what we have is current or stale using conditional cache validation headers:

- `If-Not-Match` : (Etag)
- `If-Modified-Since` : (Timestamp)

A variant of this was covered in the headers section and used for concurrency. In that case we were looking to execute an action only if the resource was in the same state we last saw it. In the case of caches, we are looking to get fresh copies of a resource if it is not in the same state we last saw it.

If the resource has not changed

- enable server-side to do less processing because client does not need a new copy
- report the resource has not changed to the client
- enable client-side to do less processing because nothing has changed

1. Get a current copy of the data and store off the ETag and Last-Modified.

   ```
   > response=HTTParty.get("http://localhost:3000/movies/12345",
                            headers:{"Accept"=>"application/json"})
   > response.response
    => #<Net::HTTPOK 200 OK readbody=true>
   > etag=response.header["etag"]
    => "\"dd7543eb8124a81a065c2d0629222e2c\""
   > last_modified=response.header["last-modified"]
    => "Tue, 12 Jan 2016 17:16:35 GMT"
   ```

2. If we make a simple request again, the action is called, the view is rendered, and the same payload is returned to us.

   ```
   > response=HTTParty.get("http://localhost:3000/movies/12345",
                            headers:{"Accept"=>"application/json"})
   > response.response
    => #<Net::HTTPOK 200 OK readbody=true>
   ```

3. If we add the `If-None-Match` header with the `ETag` or `If-Modified-Since` with the `Last-Modified` timestamp, we should get a `304/NOT_MODIFIED`. This allows the view processing to be bypassed.

   ```
   > response=HTTParty.get("http://localhost:3000/movies/12345",
                           headers:{"Accept"=>"application/json",
                                    "If-None-Match"=>etag})
   > response.response
    => #<Net::HTTPNotModified 304 Not Modified readbody=true>
   > response.body
    => nil


   > response=HTTParty.get("http://localhost:3000/movies/12345",
                           headers:{"Accept"=>"application/json",
                                    "If-Modified-Since"=>last_modified})
   > response.response
    => #<Net::HTTPNotModified 304 Not Modified readbody=true>
   > response.body
    => nil
   ```

4. We can also add conditional logic within the action that will only fire if the caller is not getting back a `304/NOT_MODIFIED`. In the example below, we always log when the action is accessed and then conditionally log when the caller has stale data and we are going to hand out a fresh copy. This gives us a chance to access one collection `movies` and then conditionally work with another (`movie_accesses`).

   ```
   def show
     #    fresh_when(@movie)
     @movie.movie_accesses.create(:action=>"show")
     if stale? @movie
       @movie.movie_accesses.create(:action=>"show-stale")
       #do some additional, expensive work here
     end
   end
   ```

   Notice that when the request is validated to be current, the conditional logic is not executed. That is shown by a non-query of our database.

   Note that `stale?` calls `fresh_when` under the covers so there is no need to call `fresh_when` when you discover the client has a stale copy of the resource.

   ```
   > response=HTTParty.get("http://localhost:3000/movies/12345",
                           headers:{"Accept"=>"application/json",
                                    "If-Modified-Since"=>last_modified})
   > response.response
    => #<Net::HTTPNotModified 304 Not Modified readbody=true>
   > pp Movie.find("12345").movie_accesses.pluck(:created_at, :action).to_a
   [[2016-01-12 18:37:08 UTC, "show"]]
   ```

   The techniques above work well for any resource URI, including resource collections commonly served by the index method. In this example, the `last_modified` value is being set to the most recent change in the collection. If we did not add this last-modified would not have been included and the ETag would have been based on the hash of the string of the resulting view. Place this line here saves Rails from having to render the view. This is specifically setting the last-modified to the most receive update to any of the members within the collection.

   ```
     def index
       @movies = Movie.all
       fresh_when last_modified: @movies.max(:updated_at)
     end
   ```

5. We can provide both `If-Modified-Since` and `If-None-Match` in the header and if either fires, our conditional logic will get triggered. Notice how this request passes in a bogus `ETag` and we get the additional entry in the `movies_accesses` collection. The first show is from the previous step.

   ```
   > response=HTTParty.get("http://localhost:3000/movies/12345",
                           headers:{"Accept"=>"application/json",
                                    "If-Modified-Since"=>last_modified,
                                    "If-None-Match"=>"123"})
   > response.response
    => #<Net::HTTPOK 200 OK readbody=true>
   > pp Movie.find("12345").movie_accesses.pluck(:created_at, :action).to_a
   [[2016-01-12 18:37:08 UTC, "show"],
    [2016-01-12 18:45:49 UTC, "show"],
    [2016-01-12 18:45:49 UTC, "show-stale"]]
   ```

6. Try this with your browser. In Chrome, open up a new tag, activate `Developer Tools`, and select `Network`. Select `Preserve log` at the top of the `Network` tab to preserve the traffic for each request.

- Navigate to `http://localhost:3000/movies/12345.json` and you will see the status come back 200/OK for the `/movies/12345.json` URI.

- Hit refresh and you will see the status come back 304/Not Modified and `If-Modified-Since` and `If-None-Match` headers were supplied.
- Click `disable-cache` at the top of the Network tab and hit refresh. The conditional headers are not sent to the Rails server and the full response is returned using a 200/OK.

So with the mentioned cache headers in place and use of conditional header tags, we can help prevent extra work on the server-side just by having the client tell us what they currently have. The demonstration application we have is trivial. Think of the case where we must federate queries to several back-end systems to answer a specific request. We can make use of the caching headers to save time for what has already been derived in the past.

Note too that 304/Not-Modified can also save our clients some work as well. Processing flows that are kicked off by the arrival of new data can be throttled by hearing that the data from the external services has not changed.

Cache-Control

That may be fine that clients can poll our movie application and determine that they have the most recent version available, but take into consideration client caches – whether they be in the browser or at the corporate firewall. What we want to do is give better hints as to how long this information is good for.

1. Start by requesting the state of a movie 10 times in rapid fire.

   - Clear out existing stats.

   ```
   > Movie.find("12345").movie_accesses.delete_all
    => ##
   ```

   - Notice how each call to HEAD on /movies/12345 results in a database access. Since we did not provide cache re-validation headers, we get a "stale client" hit and do the extra processing required each time.

   ```
   > 10.times.each {HTTParty.head("http://localhost:3000/movies/12345")}
    => 10
   > pp MovieAccess.where(:movie_id=>"12345",:action=>"show-stale").pluck(:created_at, :action)
   [[2016-01-12 19:26:20 UTC, "show-stale"],
    [2016-01-12 19:26:20 UTC, "show-stale"],
    [2016-01-12 19:26:20 UTC, "show-stale"],
    [2016-01-12 19:26:20 UTC, "show-stale"],
    [2016-01-12 19:26:20 UTC, "show-stale"],
    [2016-01-12 19:26:20 UTC, "show-stale"],
    [2016-01-12 19:26:21 UTC, "show-stale"],
    [2016-01-12 19:26:21 UTC, "show-stale"],
    [2016-01-12 19:26:21 UTC, "show-stale"],
    [2016-01-12 19:26:21 UTC, "show-stale"]]
   ```

2. Do the same with the browser. Open either the JSON or HTML page and hot refresh 10 times in a row.

   ```
   > Movie.find("12345").movie_accesses.delete_all
    => 50
   ```

   We should get a database access for each browser refresh. Notice that since the browser provided the cache revalidation header each time, we were able to save the extra work since they did had a fresh copy of the resource. However, we still had to poll our initial collection to determine the state.

   ```
   > pp MovieAccess.where(:movie_id=>"12345").pluck(:created_at, :action)
   [[2016-01-12 19:29:07 UTC, "show"],
    [2016-01-12 19:29:08 UTC, "show"],
    [2016-01-12 19:29:09 UTC, "show"],
    [2016-01-12 19:29:11 UTC, "show"],
    [2016-01-12 19:29:12 UTC, "show"],
   ```

```
 [2016-01-12 19:29:14 UTC, "show"],
 [2016-01-12 19:29:15 UTC, "show"],
 [2016-01-12 19:29:17 UTC, "show"],
 [2016-01-12 19:29:18 UTC, "show"],
 [2016-01-12 19:29:20 UTC, "show"]]
```

Now multiply that by all the users behind the corporate proxy trying to cache these calls. Why don't we give the client some additional information about the cachable aspects of our resource so that it can stop calling us entirely for periods of time. How many times does the postal service need to be called to lookup the city/state that goes with a zip code? Once an hour or day might be enough to get everyone the information they need without exchanging call/responses – even to reply with 304/Not-Modified.

Lets look into giving the client caches some help in taking on some delegated responsibility from the server and saving a few unnecessary calls.

3. Update the `show` method to include two caching headers: `Expires` and `Cache-Control`. They overlap in meaning and if they ever conflict, `Cache-Control` is suppose to take precidence. What we are saying to the caller is

   - The document you are receiving will expire on the specified data (which happens to be in 10secs nad agree with Cache-Control). Do not serve this content after the provided date
   - The document is not specific to an individual caller. You may cache this document for other callers as well. If this information was specific to the caller (e.g., a personal bank statement), then Cache-Control would either be set to `nocache` or `private` to keep the resource from being served to other clients.
   - The maximum time to cache the document should not exceed 10secs (that is where the overap is with Expires)

   Notice that Rails has a special method that can be used to set the Cache-Control response header. For Expires, we need to manually set the response header.

```
  def show
    @movie.movie_accesses.create(:action=>"show")
    if stale? @movie
      @movie.movie_accesses.create(:action=>"show-stale")
      #do some additional, expensive work here
    secs=10
    response.headers["Expires"] = secs.seconds.from_now.httpdate
#    response.headers["Cache-Control"] = "public, max-age=#{secs}"
    expires_in secs.seconds, :public=>true
    end
```

   This is what the header values turn out to be on a sample call.

```
  etag:
  - '"dd7543eb8124a81a065c2d0629222e2c"'
  last-modified:
  - Tue, 12 Jan 2016 17:16:35 GMT
  expires:
  - Tue, 12 Jan 2016 19:52:25 GMT
  cache-control:
  - max-age=10, public
```

   It is tough to find a browser lazy enough and willing to take on caching responsibilities beyond cache revalidation so lets turn to some automated processing in the next step. This is something we have better control over.

4. Lets try to make this a bit more obvious with HTTParty and a simplistic caching gem called `dry_ice`. We are only using it in the client, but to make easy use of it within the rails console, we add it to the Gemfile. CacheBar provides an HTTP caching layer for HTTParty calls.

```
gem 'httparty'
gem 'dry_ice'
```

```
$ bundle
```

Create a services directory and add it to the loaded paths

```
$ mkdir app/services
```

```
# config/application.rb
config.eager_load_paths += %W( #{config.root}/app/services )
```

Add the following class to `app/services`. Notice the use of `Rails.cache`. Rails provides an internal cache API/implementation that this will be leveraging. We can use it ourselves for "low-level caching", but we won't dive that deep. What we intend to do here is have the cache return us a representation of the resource that is within cache constraints or call the service to get a fresh copy.

```
# app/services/cached_ws.rb
class CachedWS
  include HTTParty
  include HTTParty::DryIce
#  debug_output $stdout
  base_uri "http://localhost:3000"
  cache Rails.cache
end
```

When we run the following script, notice how the database is only polled every 9 to 12 secs (3sec sleep and a 10 sec cache timeout).

```
> 10.times.each do |x|
   p "look=#{x}, accesses=#{Movie.find("12345").movie_accesses.where(:action=>"show").count}"
   CachedWS.get("/movies/12345.json").parsed_response
   sleep(3.seconds)
   end
"look=0, accesses=0"
"look=1, accesses=1"
"look=2, accesses=1"
"look=3, accesses=1"
"look=4, accesses=1"
"look=5, accesses=2"
"look=6, accesses=2"
"look=7, accesses=2"
"look=8, accesses=2"
"look=9, accesses=3"
```

Our document is trivially small, but the protocol exchange demonstrates what can be done to offload some work that may not have to get done during steady-state.

Server Caching

References:

- Caching with Rails: An verview

In the previous sections we looked into seeing what we can do to get the client to help our server-side implementation do less. In this section we will look at ways to make the stable products of the server more efficient to re-use.

Rails has several types of caching that can be globally turned on/off in environment-specific profiles. The following was enabled in the development profile for this purposes of this demo.

```
# config/environments/development.rb
  config.action_controller.perform_caching = true
```

Rails several levels of caching

- page caching - serving up pre-rendered views without calling actions
- action caching -
- fragment caching -
- low-level caching -

Page and Action caching have been officially moved out of Rails and into separate, optional gems in favor of smaller-grain caching implemented using fragment caching. That does not mean there is not a place for page and action caching. Lets look at them here because they have a very simply entry point to get started in this area.

Page Caching

Reference: `Action Pack Caching`

What we are essentially doing here is dynamically building static content to serve out through web server proxies standanding in front of the Rails server.

- page cache
  - writes static files to directory
  - lazily updates files only when accessed
  - invalidates/removes files on events like updates
  - directory cleared of stale content using sweeper
- web server
  - serves a public single URI to the public
  - looks for content first in static content directory
  - makes request to Rails server if static content is missing

This is considered a niave approach to caching because caller identity tends to matter. However, if it can be used it is blazingly fast.

- fast
  - pre-rendered views being served
- good for
  - dynamic content that stays stable for periods of time
  - content served without regard to caller
- not appropriate for
  - content that varies per user (e.g., login, preferences)
  - content that is very dynamic
- separate gem
  - Gemfile: `gem 'actionpack-page_caching'`

Setup

1. After we add the gem we turn on caching and assign a directory for it to place the rendered content files.

```
# config/environments/development.rb
  config.action_controller.perform_caching = true
  config.action_controller.page_cache_directory = "#{Rails.root.to_s}/public/page_cache"
```

2. We add a `caches_page` to the controller and identify the actions that get page caching. Note that we have cloned out initial Movies Controller for use with page caching example.

```
class MoviePagesController < ApplicationController
  before_action :set_movie, only: [:show, :edit, :update, :destroy]
  caches_page :index, :show
```

3. We add page expiration on update methods to notify the caching layer to delete the cached copy.

```
  def update
    respond_to do |format|
      if @movie.update(movie_params)
        expire_page action: "show", id:@movie, format: request.format.symbol
        expire_page action: "index", format: request.format.symbol
...
  def destroy
    @movie.movie_accesses.create(:action=>"destroy")
    @movie.destroy
    expire_page action: "show", id:@movie, format: request.format.symbol
    expire_page action: "index", format: request.format.symbol
...
```

4. The rendered content is written to files in the public directory based on the URI. The example below shows the result of calling the `index` and `show` methods.

```
public/page_cache/
|-- movie_pages
|   `-- 12345.json
`-- movie_pages.json
```

5. A look at a sample cache file shows that just file file contents as returned to the caller were written into the file.

```
{"id":"12345","title":"rocky25","updated_at":"2016-01-12T17:16:35.666Z"}
```

Scenario

1. Clear `public/page_cache` of files.

```
$ rm -rf public/page_cache/
```

2. Request a specific movie

```
> response=HTTParty.get("http://localhost:3000/movie_pages/12345.json")
> response.response
 => #<Net::HTTPOK 200 OK readbody=true>
```

3. Look into the `public/page_cache/` directory for cached content.

```
public/
|-- 404.html
|-- 422.html
|-- 500.html
|-- favicon.ico
|-- page_cache
|   `-- movie_pages
|       `-- 12345.json
`-- robots.txt
```

4. Post an update to the document, which invokes the

   ```
   > response=HTTParty.put("http://localhost:3000/movie_pages/12345.json",:body=>{:movie=>{:title=>"rocky2
   > response.response
    => #<Net::HTTPOK 200 OK readbody=true>
   ```

5. Verify the cached file was deleted.

   ```
   public/
   |-- 404.html
   |-- 422.html
   |-- 500.html
   |-- favicon.ico
   |-- page_cache
   |   `-- movie_pages
   `-- robots.txt
   ```

Scenario Overtime At this point the official demo is over and we are just playing so ignore some of the quick hacks. The following simulates what the web server would do if it were looking for static content prior to delegating to our controller. Our controller gets called, but the database is not accessed. We perform a check of the cache and perform a redirect if the file exists.

1. Add the following wrapper method around `set_movie`. It checks for an existing cached file. If found, performs a redirect to the static copy of the content. If it does not exist it continues the path to `set_movie`

   ```
   def set_cached_movie
     @@page_cache_directory ||= Rails.application.config.action_controller.page_cache_directory
     uri="/movie_pages/#{params[:id]}.#{request.format.symbol}"
     file_path="#{@@page_cache_directory}/#{uri}"
     if File.exists?(file_path)
       redirect_to "/page_cache#{uri}"
     else
       Rails.logger.debug("cached file #{file_path} not found")
       set_movie
     end
   end
   ```

2. Update the before actions for the controller to have the show method use this hack of a method.

   ```
   class MoviePagesController < ApplicationController
     before_action :set_movie, only: [:edit, :update, :destroy]
     before_action :set_cached_movie, only: [:show]
     caches_page :index, :show
   ```

3. Clear the directory cache and perform a GET of the movie 10 times and check results.

   ```
   $ rm -rf public/page_cache/
   ```

```
> Movie.find("12345").movie_accesses.delete_all;
> 10.times.each do |x|
    p x;
    response=HTTParty.get("http://localhost:3000/movie_pages/12345.json")
end
```

Notice the action method was only called for the first request.

```
> pp MovieAccess.where(:movie_id=>"12345").pluck(:created_at, :action);nil
[[2016-01-13 00:01:38 UTC, "cached-page#show"],
 [2016-01-13 00:01:38 UTC, "cached-page#show-stale"]]
```

4. Clear the cache and modify the loop to add an update every third call.

```
$ rm -rf public/page_cache/
```

```
> Movie.find("12345").movie_accesses.delete_all
> 10.times.each do |x|
     response=HTTParty.get("http://localhost:3000/movie_pages/12345.json")
     if (x%3==0)
       HTTParty.put("http://localhost:3000/movie_pages/12345.json",:body=>{:movie=>{:title=>"rocky#{x}"
     end
   end
```

Notice how

- show is called after each update
- the cached copy is used until invalidated by the next update

```
ruby > pp MovieAccess.where(:movie_id=>"12345").pluck(:created_at, :action);nil D | {"find"=>"movie_acc
"filter"=>{"movie_id"=>"12345"}, "projection"=>{"created_at"=>1, "action"=>1}} [[2016-01-13
00:25:06 UTC, "cached-page#show"], [2016-01-13 00:25:06 UTC, "cached-page#show-stale"],
[2016-01-13 00:25:06 UTC, "update"], [2016-01-13 00:25:06 UTC, "cached-page#show"], [2016-01-13
00:25:06 UTC, "cached-page#show-stale"], [2016-01-13 00:25:06 UTC, "update"], [2016-01-13
00:25:06 UTC, "cached-page#show"], [2016-01-13 00:25:06 UTC, "cached-page#show-stale"],
[2016-01-13 00:25:06 UTC, "update"], [2016-01-13 00:25:06 UTC, "cached-page#show"], [2016-01-13
00:25:06 UTC, "cached-page#show-stale"], [2016-01-13 00:25:06 UTC, "update"]]
```

Action Caching Reference: `Action Pack Caching`

Similar to page caching except that the rails filters (e.g., authentication) are run before serving out the cached copies. This allows more dynamic aspects to be considered.

Since all incoming calls have to be pre-processed in this approach, the cached data is not written out in a form to be directly consumed by static web content servers. It leverages the internal caching mechanism within Rails and supports things like custom cache keys and retention times.

- action cache
  - entire output is placed into cache
  - runs all pre-filters
  - enables authentication to be factored into the result
  - uses fragment cache internally
  - content automatically aged off with condiguration property
  - control down to the controller, action, and call properties
- separate gem
  - Gemfile: `gem ’actionpack-action_caching`

Setup

1. After we add the gem we turn on caching and assign a directory for it to place the rendered content files.

   ```
   # config/environments/development.rb
     config.action_controller.perform_caching = true
   ```

2. We add a `caches_action` to the controller and identify the actions that get action caching. Note that we have cloned out initial Movies Controller for use with page caching example.

   ```
   class MovieActionsController < ApplicationController
     before_action :set_movie, only: [:show, :edit, :update, :destroy]
     caches_action :index, :show, expires_in: 20.seconds
   ```

3. We insert some cache expirations when for when a resource is changed or deleted. This invalidates/evicts the cache entry. Notice these calls are called `expire_action` and not `expire_page`

   ```
     def update
   respond_to do |format|
     if @movie.update(movie_params)
       expire_action action: "show", id:@movie, format: request.format.symbol
       expire_action action: "index", format: request.format.symbol
       fresh_when(@movie)
   ...
     def destroy
   @movie.movie_accesses.create(:action=>"cached-action#destroy")
   @movie.destroy
   expire_action action: "show", id:@movie, format: request.format.symbol
   expire_action action: "index", format: request.format.symbol
   ...
   ```

That is pretty much it to get started. The cache will use `Rails.cache` under the hood. You can configure what that cache uses for storage through the configuration. The following optional configuration writes this cache to a file under `/tmp/cache/action_cache`

```
config.cache_store = :file_store, "./tmp/cache/action_cache"
```

Scenario

1. Shutdown the server, remove the contents of the `tmp` directory, and restart.

   ```
   (Control+C)
   $ rm -rf tmp/*
   $ rails s
   ```

2. Clear the database of movie accesses

   ```
   > Movie.find("12345").movie_accesses.delete_all
   ```

3. Perform a single GET on an existing movie. Notice the following file showing up in the `action_cache` directory. If you take a peek inside this file you will see that it is more than just the content.

   ```
   mp/cache/
   `-- action_cache
       `-- 182
           `-- D21
               `-- views%2Flocalhost%3A3000%2Fmovie_actions%2F12345.json
   ```

When we look at our database accesses, we see that this first request resulted in a full database lookup.

```
>  pp MovieAccess.where(:movie_id=>"12345").pluck(:created_at, :action);nil
[[2016-01-13 01:55:20 UTC, "cached-action#show"],
 [2016-01-13 01:55:20 UTC, "cached-action#show-stale"]]
```

4. Clear the movie access log and request the movie ten times in a loop under 20 secs.

```
> Movie.find("12345").movie_accesses.delete_all
> 10.times.each {HTTParty.get("http://localhost:3000/movie_actions/12345.json")}
```

   The first access resulted in a database access and everything else used the cached copy.

```
>  pp MovieAccess.where(:movie_id=>"12345").pluck(:created_at, :action);nil
D | {"find"=>"movie_accesses", "filter"=>{"movie_id"=>"12345"}, "projection"=>{"created_at"=>1, "actio
[[2016-01-13 02:02:47 UTC, "cached-action#show"],
 [2016-01-13 02:02:47 UTC, "cached-action#show-stale"]]
```

5. Do the same thing except add a PUT on every 4th call. Notice that show is invoked to do work on the GET that immediately follows the PUT. Our `expire_action` call in the PUT caused the data to be evicted from the cache.

```
> Movie.find("12345").movie_accesses.delete_all
> 10.times.each do |x|
     HTTParty.get("http://localhost:3000/movie_actions/12345.json")
    if (x%4==0)
     HTTParty.put("http://localhost:3000/movie_actions/12345.json",:body=>{:movie=>{:title=>"rocky#{x}"
     end
    end
>  pp MovieAccess.where(:movie_id=>"12345").pluck(:created_at, :action)
[[2016-01-13 03:03:34 UTC, "cached-action#show"],
 [2016-01-13 03:03:34 UTC, "cached-action#show-stale"],
 [2016-01-13 03:03:34 UTC, "cached-action#update"],
 [2016-01-13 03:03:34 UTC, "cached-action#show"],
 [2016-01-13 03:03:34 UTC, "cached-action#show-stale"],
 [2016-01-13 03:03:34 UTC, "cached-action#update"],
 [2016-01-13 03:03:34 UTC, "cached-action#show"],
 [2016-01-13 03:03:34 UTC, "cached-action#show-stale"],
 [2016-01-13 03:03:34 UTC, "cached-action#update"],
 [2016-01-13 03:03:34 UTC, "cached-action#show"],
 [2016-01-13 03:03:34 UTC, "cached-action#show-stale"]]
```

6. Do the same thing except add a 10 second sleep in between loop iterations. You can alternatively reduce the cache expiration time or number of loops to finish sooner. To entertain yourself while this is going on – observe the differences in database interactions for each call. The before action is being called to locate the movie and that is looking it up in the database. Every 20 seconds our show method gets called and does a full database interaction.

   Notice we get five (5) calls to show for the ten (10) calls because we configured the movie to expire in 20 seconds and we waited 10 seconds in between calls.

```
> Movie.find("12345").movie_accesses.delete_all
> 10.times.each {|x| p x;HTTParty.get("http://localhost:3000/movie_actions/12345.json"); sleep 10.secon
>  pp MovieAccess.where(:movie_id=>"12345").pluck(:created_at, :action);nil
[[2016-01-13 02:10:39 UTC, "cached-action#show"],
 [2016-01-13 02:10:39 UTC, "cached-action#show-stale"],
 [2016-01-13 02:10:59 UTC, "cached-action#show"],
 [2016-01-13 02:10:59 UTC, "cached-action#show-stale"],
 [2016-01-13 02:11:19 UTC, "cached-action#show"],
 [2016-01-13 02:11:19 UTC, "cached-action#show-stale"],
 [2016-01-13 02:11:39 UTC, "cached-action#show"],
 [2016-01-13 02:11:39 UTC, "cached-action#show-stale"],
 [2016-01-13 02:11:59 UTC, "cached-action#show"],
 [2016-01-13 02:11:59 UTC, "cached-action#show-stale"]]
```

7. Before quiting, remember that caching can be turned on/off with a master switch in the configuration. If we turn off caching in the environment file, restart the server, and re-run a quick loop test you will see that caching is now turned off and the show action is called with every GET.

```
config.action_controller.perform_caching = false
```

```
Control+C
$ rails s
```

```
> Movie.find("12345").movie_accesses.delete_all
> 10.times.each {HTTParty.get("http://localhost:3000/movie_actions/12345.json")}
>  MovieAccess.where(:movie_id=>"12345").pluck(:created_at, :action).count
 => 20
```

We did not vary our output at all with information like user or category of caller. However, to partition the cache look into using the `cache_path` option within the `cashes_action` callback. That allows us to do things on a per action and call basis.

We also have not tied outselves to the file system. As we will see later we can configure the cache to use other mechanisms like in-memory and shared memory caches.

Other Caching Mechanisms

The Rails guide covers three additional caching strategies:

- Fragment Caching
  - fine grain
  - used to independently cache sections of web page content
  - used as the implementation for Action Caching (i.e., you are already using it)
  - backed by Rails.cache
- Low Level Caching
  - Rails.cache
  - manual caching through keys and retention properties
- SQL Caching
  - SQL results sets are cached based on queries issued to the database

Cache Stores

You already saw an example of a file-based implementation of caching for action caching. Rails caching can be configured with other strategies.

- `:file_store, (path)`
- `:memory_store, {options}`
  - size: (e.g., 64.megabytes)
- `:mem_cache_store, (host), (host)`
- ...
- custom

Memory Store

1. Change the `cache_store` to `:memory_store`

   ```
   config.cache_store = :memory_store
   ```

2. Stop the server, delete the contents of `tmp/`, restart the server

   ```
   Control+C
   rm -rf tmp/*
   $ rails s
   ```

3. Execute the GET 10 times. Notice we have the expected number of calls to `show`.

   ```
   > Movie.find("12345").movie_accesses.delete_all
   > 10.times.each {HTTParty.get("http://localhost:3000/movie_actions/12345.json")}
   >  pp MovieAccess.where(:movie_id=>"12345").pluck(:created_at, :action)
   [[2016-01-13 03:17:23 UTC, "cached-action#show"],
    [2016-01-13 03:17:23 UTC, "cached-action#show-stale"]]
   ```

   Notice there are not longer files written to the cache directory.

   ```
   tmp/
   |-- cache
   |-- pids
   |   `-- server.pid
   |-- sessions
   `-- sockets
   ```

File Store (covered in earlier examples)

Assembly

This section covers the basics of how the application was assembled. The unique difference this application has is thar it logs activity with the model in a separate collection from the model under test.

Core Application

1. Create a new application

   ```
   $ rails new caching-movies
   $ cd caching-movies
   ```

2. Update Gemfile, run bundle, install Mongoid, and start server

   ```
   # Gemfile
   gem 'mongoid', '~>5.0.0'
   gem 'httparty'

   # config/application.rb
   Mongoid.load!("./config/mongoid.yml")

   $ bundle
   $ rails s
   ```

Create the Models

1. Create the model scaffolding

   ```
   $ rails g scaffold Movie id title updated_at
   $ rails g model MovieAccess created_at accessed_by action
   ```

2. Update the model classes.

   ```
   class Movie
     include Mongoid::Document
     include Mongoid::Timestamps::Updated
     field :id, type: String
     field :title, type: String

     has_many :movie_accesses
   end

   class MovieAccess
     include Mongoid::Document
     include Mongoid::Timestamps::Created
     field :accessed_by, type: String
     field :action, type: String

     belongs_to :movie
   end
   ```

3. Verify data model using the rails console. We should be able to create a movie and store access to that movie. We will use this to keep track of impact of changes made.

```
> movie=Movie.create(:id=>"12345", :title=>"rocky25")
> movie.movie_accesses.create(:accessed_by=>"123",:action=>"created")

> pp Movie.find(movie.id).attributes
{"_id"=>"12345", "title"=>"rocky25", "updated_at"=>2016-01-11 00:02:03 UTC}
 => {"_id"=>"12345", "title"=>"rocky25", "updated_at"=>2016-01-11 00:02:03 UTC}

> pp Movie.find(movie.id).movie_accesses.pluck(:created_at, :accessed_by, :action)
[[2016-01-11 00:02:09 UTC, "123", "created"]]
```

4. Fix the JSON view of the movie

   ```
   json.extract! @movie, :id, :title, :created_at
   ```

5. Update the Movies controller to add a MovieAccess for each time the movie is grabbed from the database and why.

   ```
   # app/controllers/movies_controller.rb
     def show
       @movie.movie_accesses.create(:action=>"show")
     end
     def create
       @movie = Movie.new(movie_params)

       respond_to do |format|
         if @movie.save
           @movie.movie_accesses.create(:action=>"create")
   ```

```
  def update
    respond_to do |format|
      if @movie.update(movie_params)
        @movie.movie_accesses.create(:action=>"create")

  def destroy
    @movie.movie_accesses.create(:action=>"destroy")
    @movie.destroy
```

6. Update the ApplicationController to permit WS interaction.

```
# app/controllers/application_controller.rb
class ApplicationController < ActionController::Base
  # Prevent CSRF attacks by raising an exception.
  # For APIs, you may want to use :null_session instead.
  #protect_from_forgery with: :exception
  protect_from_forgery with: :null_session
end
```

7. Test out WS access and controller functionality

```
> response=HTTParty.post("http://localhost:3000/movies",:body=>{:movie=>{:id=>"123456", :title=>"rocky2

> pp Movie.find("123456").attributes
D | {"find"=>"movies", "filter"=>{"_id"=>"123456"}}
{"_id"=>"123456", "title"=>"rocky25", "updated_at"=>2016-01-11 00:13:01 UTC}

> pp Movie.find("123456").movie_accesses.pluck(:created_at, :accessed_by, :action)
[[2016-01-11 00:13:01 UTC, nil, "create"],
 [2016-01-11 00:13:01 UTC, nil, "show"]]
```

8. Create clones of the Movies controller (MoviePages and MovieActions). Adjust the generated views s that they reference the Movie model class and not MovieAction and MoviePage.

Last Updated: 2016-01-12