Caching with Rails: An overview

This guide will teach you what you need to know about avoiding that expensive round-trip to your database and returning what you need to return to the web clients in the shortest time possible.

After reading this guide, you should be able to use and configure:

Page, action, and fragment caching

Sweepers

Alternative cache stores

Conditional GET support

Chapters

1 Basic Caching

This is an introduction to the three types of caching techniques that Rails provides by default without the use of any third party plugins.

To start playing with caching you'll want to ensure that `config.action_controller.perform_caching` is set to `true`, if you're running in development mode. This flag is normally set in the corresponding `config/environments/*.rb` and caching is disabled by default for development and test, and enabled for production.

```
config.action_controller.perform_caching =
true
```

1.1 Page Caching

Page caching is a Rails mechanism which allows the request for a generated page to be fulfilled by the webserver (i.e. Apache or nginx), without ever having to go through the Rails stack at all. Obviously, this is super-fast. Unfortunately, it can't be applied to every situation (such as pages that need authentication) and since the webserver is literally just serving a file from the filesystem, cache expiration is an issue that needs to be dealt with.

To enable page caching, you need to use the `caches_page` method.

```
class

ProductsController < ActionController


caches_page
:index


def

index

@products

= Products.all

end
end
```

Let's say you have a controller called `ProductsController` and an `index` action that lists all the products. The first time anyone requests `/products`, Rails will generate a file called `products.html` and the webserver will then look for that file before it passes the next request for `/products` to your Rails application.

By default, the page cache directory is set to `Rails.public_path` (which is usually set to the `public` folder) and this can be configured by changing the configuration setting `config.action_controller.page_cache_directory`. Changing the default from `public` helps avoid naming conflicts, since you may want to put other static html in `public`, but changing this will require web server reconfiguration to let the web server know where to serve the cached files from.

The Page Caching mechanism will automatically add a `.html` extension to requests for pages that do not have an extension to make it easy for the webserver to find those pages and this can be configured by changing the configuration setting `config.action_controller.page_cache_extension`.

In order to expire this page when a new product is added we could extend our example controller like this:

```
class

ProductsController < ActionController


caches_page
:index


def

index

@products

= Products.all

end


def

create

expire_page
:action

=>
:index

end

end
```

If you want a more complicated expiration scheme, you can use cache sweepers to expire cached objects when things change. This is covered in the section on Sweepers.

Page caching ignores all parameters. For example `/products?page=1` will be written out to the filesystem as `products.html` with no reference to the `page` parameter. Thus, if someone requests `/products?page=2` later, they will get the cached first page. Be careful when page caching GET parameters in the URL!

Page caching runs in an after filter. Thus, invalid requests won't generate spurious cache entries as long as you halt them. Typically, a redirection in some before filter that checks request preconditions does the job.

1.2 Action Caching

One of the issues with Page Caching is that you cannot use it for pages that require to restrict access somehow. This is where Action Caching comes in. Action Caching works like Page Caching except for the fact that the incoming web request does go from the webserver to the Rails stack and Action Pack so that before filters can be run on it before the cache is served. This allows authentication and other restriction to be run while still serving the result of the output from a cached copy.

Clearing the cache works in the exact same way as with Page Caching.

Let's say you only wanted authenticated users to call actions on `ProductsController`.

```
class

ProductsController < ActionController


before_filter
:authenticate

caches_action
:index


def

index

@products

= Product.all

end


def

create

expire_action
:action

=>
:index

end

end
```

You can also use `:if` (or `:unless`) to pass a Proc that specifies when the action should be cached. Also, you can use `:layout => false` to cache without layout so that dynamic information in the layout such as logged in user info or the number of items in the cart can be left uncached. This feature is available as of Rails 2.2.

You can modify the default action cache path by passing a `:cache_path` option. This will be passed directly to `ActionCachePath.path_for`. This is handy for actions with multiple possible routes that should be cached differently. If a block is given, it is called with the current controller instance.

Finally, if you are using memcached or Ehcache, you can also pass `:expires_in`. In fact, all parameters not used by `caches_action` are sent to the underlying cache store.

Action caching runs in an after filter. Thus, invalid requests won't generate spurious cache entries as long as you halt them. Typically, a redirection in some before filter that checks request preconditions does the job.

1.3 Fragment Caching

Life would be perfect if we could get away with caching the entire contents of a page or action and serving it out to the world. Unfortunately, dynamic web applications usually build pages with a variety of components not all of which have the same caching characteristics. In order to address such a dynamically created page where different parts of the page need to be cached and expired differently, Rails provides a mechanism called Fragment Caching.

Fragment Caching allows a fragment of view logic to be wrapped in a cache block and served out of the cache store when the next request comes in.

As an example, if you wanted to show all the orders placed on your website in real time and didn't want to cache that part of the page, but did want to cache the part of the page which lists all products available, you could use this piece of code:

```
<% Order.find_recent.
each

do

|o| %>
```

```
<%= o.buyer.name %> bought <%= o.product.name %>
<%
end

%>

<% cache
do

%>

All available products:

<% Product.all.
each

do

|p| %>

<%= link_to p.name, product_url(p) %>

<%
end

%>
<%
end

%>
```

The cache block in our example will bind to the action that called it and is written out to the same place as the Action Cache, which means that if you want to cache multiple fragments per action, you should provide an `action_suffix` to the cache call:

```
<% cache(
:action

=>
'recent'
,
:action_suffix

=>
'all_products'
)
do

%>

All available products:
```

and you can expire it using the `expire_fragment` method, like so:

```
expire_fragment(
:controller

=>
'products'
,
:action

=>
'recent'
,
:action_suffix

=>
'all_products'
)
```

If you don't want the cache block to bind to the action that called it, You can also use globally keyed fragments by calling the `cache` method with a key, like so:

```
<% cache(
'all_available_products'
)
do

%>

All available products:
<%
end

%>
```

This fragment is then available to all actions in the `ProductsController` using the key and can be expired the same way:

```
expire_fragment(
'all_available_products'
)
```

1.4 Sweepers

Cache sweeping is a mechanism which allows you to get around having a ton of `expire_{page,action,fragment}` calls in your code. It does this by moving all the work required to expire cached content into an `ActionController::Caching::Sweeper` subclass. This class is an observer and looks for changes to an object via callbacks, and when a change occurs it expires the caches associated with that object in an around or after filter.

Continuing with our Product controller example, we could rewrite it with a sweeper like this:

```
class

ProductSweeper < ActionController::Caching::Sweeper

observe Product
# This sweeper is going to keep an eye on the Product model


# If our sweeper detects that a Product was created call this

def

after_create(product)

expire_cache_for(product)

end


# If our sweeper detects that a Product was updated call this

def

after_update(product)

expire_cache_for(product)

end


# If our sweeper detects that a Product was deleted call this

def

after_destroy(product)

expire_cache_for(product)

end


private

def

expire_cache_for(product)

# Expire the index page now that we added a new product

expire_page(
:controller

=>
'products'
,
:action

=>
'index'
)


# Expire a fragment

expire_fragment(
'all_available_products'
)

end
end
```

You may notice that the actual product gets passed to the sweeper, so if we were caching the edit action for each product, we could add an expire method which specifies the page we want to expire:

```
expire_action(
:controller

=>
'products'
,
:action

=>
'edit'
,
:id

=> product)
```

Then we add it to our controller to tell it to call the sweeper when certain actions are called. So, if we wanted to expire the cached content for the list and edit actions when the create action was called, we could do the following:

```
class

ProductsController < ActionController


before_filter
:authenticate

caches_action
:index

cache_sweeper
:product_sweeper


def

index

@products

= Product.all

end

end
```

1.5 SQL Caching

Query caching is a Rails feature that caches the result set returned by each query so that if Rails encounters the same query again for that request, it will use the cached result set as opposed to running the query against the database again.

For example:

```
class

ProductsController < ActionController


def

index

# Run a find query

@products

= Product.all


...


# Run the same query again

@products

= Product.all

end

end
```

The second time the same query is run against the database, it's not actually going to hit the database. The first time the result is returned from the query it is stored in the query cache (in memory) and the second time it's pulled from memory.

However, it's important to note that query caches are created at the start of an action and destroyed at the end of that action and thus persist only for the duration of the action. If you'd like to store query results in a more persistent fashion, you can in Rails by using low level caching.

2 Cache Stores

Rails provides different stores for the cached data created by action and fragment caches. Page caches are always stored on disk.

2.1 Configuration

You can set up your application's default cache store by calling `config.cache_store=` in the Application definition inside your `config/application.rb` file or in an Application.configure block in an environment specific configuration file (i.e. `config/environments/*.rb`). The first argument will be the cache store to use and the rest of the argument will be passed as arguments to the cache store constructor.

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

Alternatively, you can call `ActionController::Base.cache_store` outside of a configuration block.

You can access the cache by calling `Rails.cache`.

2.2 ActiveSupport::Cache::Store

This class provides the foundation for interacting with the cache in Rails. This is an abstract class and you cannot use it on its own. Rather you must use a concrete implementation of the class tied to a storage engine. Rails ships with several implementations documented below.

The main methods to call are `read`, `write`, `delete`, `exist?`, and `fetch`. The fetch method takes a block and will either return an existing value from the cache, or evaluate the block and write the result to the cache if no value exists.

There are some common options used by all cache implementations. These can be passed to the constructor or the various methods to interact with entries.

> `:namespace` – This option can be used to create a namespace within the cache store. It is especially useful if your application shares a cache with other applications. The default value will include the application name and Rails environment.

`:compress` – This option can be used to indicate that compression should be used in the cache. This can be useful for transferring large cache entries over a slow network.

`:compress_threshold` – This options is used in conjunction with the `:compress` option to indicate a threshold under which cache entries should not be compressed. This defaults to 16 kilobytes.

`:expires_in` – This option sets an expiration time in seconds for the cache entry when it will be automatically removed from the cache.

`:race_condition_ttl` – This option is used in conjunction with the `:expires_in` option. It will prevent race conditions when cache entries expire by preventing multiple processes from simultaneously regenerating the same entry (also known as the dog pile effect). This option sets the number of seconds that an expired entry can be reused while a new value is being regenerated. It's a good practice to set this value if you use the `:expires_in` option.

2.3 ActiveSupport::Cache::MemoryStore

This cache store keeps entries in memory in the same Ruby process. The cache store has a bounded size specified by the `:size` options to the initializer (default is 32Mb). When the cache exceeds the allotted size, a cleanup will occur and the least recently used entries will be removed.

```
ActionController::Base.cache_store =
:memory_store
,
:size

=>
64
.megabytes
```

If you're running multiple Ruby on Rails server processes (which is the case if you're using mongrel_cluster or Phusion Passenger), then your Rails server process instances won't be able to share cache data with each other. This cache store is not appropriate for large application deployments, but can work well for small, low traffic sites with only a couple of server processes or for development and test environments.

This is the default cache store implementation.

2.4 ActiveSupport::Cache::FileStore

This cache store uses the file system to store entries. The path to the directory where the store files will be stored must be specified when initializing the cache.

```
ActionController::Base.cache_store =
:file_store
,
"/path/to/cache/directory"
```

With this cache store, multiple server processes on the same host can share a cache. Servers processes running on different hosts could share a cache by using a shared file system, but that set up would not be ideal and is not recommended. The cache store is appropriate for low to medium traffic sites that are served off one or two hosts.

Note that the cache will grow until the disk is full unless you periodically clear out old entries.

2.5 ActiveSupport::Cache::MemCacheStore

This cache store uses Danga's `memcached` server to provide a centralized cache for your application. Rails uses the bundled `memcached-client` gem by default. This is currently the most popular cache store for production websites. It can be used to provide a single, shared cache cluster with very a high performance and redundancy.

When initializing the cache, you need to specify the addresses for all memcached servers in your cluster. If none is specified, it will assume memcached is running on the local host on the default port, but this is not an ideal set up for larger sites.

The `write` and `fetch` methods on this cache accept two additional options that take advantage of features specific to memcached. You can specify `:raw` to send a value directly to the server with no serialization. The value must be a string or number. You can use memcached direct operation like `increment` and `decrement` only on raw values. You can also specify `:unless_exist` if you don't want memcached to overwrite an existing entry.

```
ActionController::Base.cache_store =
:mem_cache_store
,
"cache-1.example.com"
,
"cache-2.example.com"
```

2.6 ActiveSupport::Cache::EhcacheStore

If you are using JRuby you can use Terracotta's Ehcache as the cache store for your application. Ehcache is an open source Java cache that also offers an enterprise version with increased scalability, management, and commercial support. You must first install the jruby-ehcache-rails3 gem (version 1.1.0 or later) to use this cache store.

```
ActionController::Base.cache_store =
:ehcache_store
```

When initializing the cache, you may use the `:ehcache_config` option to specify the Ehcache config file to use (where the default is "ehcache.xml" in your Rails config directory), and the :cache_name option to provide a custom name for your cache (the default is rails_cache).

In addition to the standard `:expires_in` option, the `write` method on this cache can also accept the additional `:unless_exist` option, which will cause the cache store to use Ehcache's `putIfAbsent` method instead of `put`, and therefore will not overwrite an existing entry. Additionally, the `write` method supports all of the properties exposed by the **Ehcache Element class** , including:

Property	Argument Type	Description
elementEvictionData	ElementEvictionData	Sets this element's eviction data instance.
eternal	boolean	Sets whether the element is eternal.
timeToIdle, tti	int	Sets time to idle
timeToLive, ttl, expires_in	int	Sets time to Live
version	long	Sets the version attribute of the ElementAttributes object.

These options are passed to the `write` method as Hash options using either camelCase or underscore notation, as in the following examples:

```
Rails.cache.write(
'key'
,
'value'
,
:time_to_idle

=>
60
.seconds,
:timeToLive

=>
600
.seconds)
caches_action
:index
,
:expires_in

=>
60
.seconds,
:unless_exist

=>
true
```

For more information about Ehcache, see **http://ehcache.org/** . For more information about Ehcache for JRuby and Rails, see **http://ehcache.org/documentation/jruby.html**

2.7 Custom Cache Stores

You can create your own custom cache store by simply extending `ActiveSupport::Cache::Store` and implementing the appropriate methods. In this way, you can swap in any number of caching technologies into your Rails application.

To use a custom cache store, simple set the cache store to a new instance of the class.

```
ActionController::Base.cache_store = MyCacheStore.
new
```

2.8 Cache Keys

The keys used in a cache can be any object that responds to either `:cache_key` or to `:to_param`. You can implement the `:cache_key` method on your classes if you need to generate custom keys. Active Record will generate keys based on the class name and record id.

You can use Hashes and Arrays of values as cache keys.

```
# This is a legal cache key
Rails.cache.read(
:site

=>
"mysite"
,
:owners

=> [owner_1, owner_2])
```

The keys you use on `Rails.cache` will not be the same as those actually used with the storage engine. They may be modified with a namespace or altered to fit technology backend constraints. This means, for instance, that you can't save values with `Rails.cache` and then try to pull them out with the `memcache-client` gem. However, you also don't need to worry about exceeding the memcached size limit or violating syntax rules.

3 Conditional GET support

Conditional GETs are a feature of the HTTP specification that provide a way for web servers to tell browsers that the response to a GET request hasn't changed since the last request and can be safely pulled from the browser cache.

They work by using the `HTTP_IF_NONE_MATCH` and `HTTP_IF_MODIFIED_SINCE` headers to pass back and forth both a unique content identifier and the timestamp of when the content was last changed. If the browser makes a request where the content identifier (etag) or last modified since timestamp matches the server's version then the server only needs to send back an empty response with a not modified status.

It is the server's (i.e. our) responsibility to look for a last modified timestamp and the if-none-match header and determine whether or not to send back the full response. With conditional-get support in Rails this is a pretty easy task:

```
class

ProductsController < ApplicationController


def

show

@product

= Product.find(params[
:id
])


# If the request is stale according to the given timestamp and etag value

# (i.e. it needs to be processed again) then execute this block

if
```

```
++

stale?(
:last_modified

=>
@product
.updated_at.utc,
:etag

=>
@product
)

respond_to
do

|wants|

# ... normal response processing

end

end


# If the request is fresh (i.e. it's not modified) then you don't need to do

# anything. The default render checks for this using the parameters

# used in the previous call to stale? and will automatically send a

# :not_modified. So that's it, you're done.

end
end
```

If you don't have any special response processing and are using the default rendering mechanism (i.e. you're not using respond_to or calling render yourself) then you've got an easy helper in fresh_when:

```
class

ProductsController < ApplicationController


# This will automatically send back a :not_modified if the request is fresh,

# and will render the default template (product.*) if it's stale.


def

show

@product

= Product.find(params[
:id
])

fresh_when
:last_modified

=>
@product
.published_at.utc,
:etag

=>
@product

end
end
```

4 Further reading

- Scaling Rails Screencasts

5 Changelog

Feb 17, 2011: Document 3.0.0 changes to ActiveSupport::Cache
May 02, 2009: Formatting cleanups
April 26, 2009: Clean up typos in submitted patch
April 1, 2009: Made a bunch of small fixes
February 22, 2009: Beefed up the section on cache_stores
December 27, 2008: Typo fixes
November 23, 2008: Incremental updates with various suggested changes and formatting cleanup
September 15, 2008: Initial version by Aditya Chadha

Feedback

You're encouraged to help improve the quality of this guide.

If you see any typos or factual errors you are confident to patch, please clone **docrails** and push the change yourself. That branch of Rails has public write access. Commits are still reviewed, but that happens after you've submitted your contribution. **docrails** is cross-merged with master periodically.

You may also find incomplete content, or stuff that is not up to date. Please do add any missing documentation for master. Check the **Ruby on Rails Guides Guidelines** for style and conventions.

If for whatever reason you spot something to fix but cannot patch it yourself, please **open an issue**.

And last but not least, any kind of discussion regarding Ruby on Rails documentation is very welcome in the **rubyonrails-docs mailing list**.