DO NOT READ THIS FILE ON GITHUB, GUIDES ARE PUBLISHED ON https://guides.rubyonrails.org.

Caching with Rails: An Overview

This guide is an introduction to speeding up your Rails application with caching.

Caching means to store content generated during the request-response cycle and to reuse it when responding to similar requests.

Caching is often the most effective way to boost an application's performance. Through caching, websites running on a single server with a single database can sustain a load of thousands of concurrent users.

Rails provides a set of caching features out of the box. This guide will teach you the scope and purpose of each one of them. Master these techniques and your Rails applications can serve millions of views without exorbitant response times or server bills.

After reading this guide, you will know:

- Fragment and Russian doll caching.
- How to manage the caching dependencies.
- Alternative cache stores.
- Conditional GET support.

Basic Caching

This is an introduction to three types of caching techniques: page, action and fragment caching. By default Rails provides fragment caching. In order to use page and action caching you will need to add `actionpack-page_caching` and `actionpack-action_caching` to your `Gemfile`.

By default, caching is only enabled in your production environment. You can play around with caching locally by running `rails dev:cache`, or by setting `config.action_controller.perform_caching` to `true` in `config/environments/development.rb`.

NOTE: Changing the value of `config.action_controller.perform_caching` will only have an effect on the caching provided by Action Controller. For instance, it will not impact low-level caching, that we address below.

Page Caching

Page caching is a Rails mechanism which allows the request for a generated page to be fulfilled by the web server (i.e. Apache or NGINX) without having to go through the entire Rails stack. While this is super fast it can't be applied to

every situation (such as pages that need authentication). Also, because the web server is serving a file directly from the filesystem you will need to implement cache expiration.

INFO: Page Caching has been removed from Rails 4. See the actionpack-page_caching gem.

Action Caching

Page Caching cannot be used for actions that have before filters - for example, pages that require authentication. This is where Action Caching comes in. Action Caching works like Page Caching except the incoming web request hits the Rails stack so that before filters can be run on it before the cache is served. This allows authentication and other restrictions to be run while still serving the result of the output from a cached copy.

INFO: Action Caching has been removed from Rails 4. See the actionpack-action_caching gem. See DHH's key-based cache expiration overview for the newly-preferred method.

Fragment Caching

Dynamic web applications usually build pages with a variety of components not all of which have the same caching characteristics. When different parts of the page need to be cached and expired separately you can use 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.

For example, if you wanted to cache each product on a page, you could use this code:

```
<% @products.each do |product| %>
  <% cache product do %>
    <%= render product %>
  <% end %>
<% end %>
```

When your application receives its first request to this page, Rails will write a new cache entry with a unique key. A key looks something like this:

```
views/products/index:bea67108094918eeba42cd4a6e786901/products/1
```

The string of characters in the middle is a template tree digest. It is a hash digest computed based on the contents of the view fragment you are caching. If you change the view fragment (e.g., the HTML changes), the digest will change, expiring the existing file.

A cache version, derived from the product record, is stored in the cache entry. When the product is touched, the cache version changes, and any cached fragments that contain the previous version are ignored.

TIP: Cache stores like Memcached will automatically delete old cache files.

If you want to cache a fragment under certain conditions, you can use `cache_if` or `cache_unless`:

```
<% cache_if admin?, product do %>
  <%= render product %>
<% end %>
```

Collection caching The `render` helper can also cache individual templates rendered for a collection. It can even one up the previous example with `each` by reading all cache templates at once instead of one by one. This is done by passing `cached: true` when rendering the collection:

```
<%= render partial: 'products/product', collection: @products, cached: true %>
```

All cached templates from previous renders will be fetched at once with much greater speed. Additionally, the templates that haven't yet been cached will be written to cache and multi fetched on the next render.

Russian Doll Caching

You may want to nest cached fragments inside other cached fragments. This is called Russian doll caching.

The advantage of Russian doll caching is that if a single product is updated, all the other inner fragments can be reused when regenerating the outer fragment.

As explained in the previous section, a cached file will expire if the value of `updated_at` changes for a record on which the cached file directly depends. However, this will not expire any cache the fragment is nested within.

For example, take the following view:

```
<% cache product do %>
  <%= render product.games %>
<% end %>
```

Which in turn renders this view:

```
<% cache game do %>
  <%= render game %>
<% end %>
```

If any attribute of game is changed, the `updated_at` value will be set to the current time, thereby expiring the cache. However, because `updated_at` will not be changed for the product object, that cache will not be expired and your app will serve stale data. To fix this, we tie the models together with the `touch` method:

```
class Product < ApplicationRecord
  has_many :games
```

```ruby
end

class Game < ApplicationRecord
  belongs_to :product, touch: true
end
```

With `touch` set to `true`, any action which changes `updated_at` for a game record will also change it for the associated product, thereby expiring the cache.

Shared Partial Caching

It is possible to share partials and associated caching between files with different mime types. For example shared partial caching allows template writers to share a partial between HTML and JavaScript files. When templates are collected in the template resolver file paths they only include the template language extension and not the mime type. Because of this templates can be used for multiple mime types. Both HTML and JavaScript requests will respond to the following code:

```ruby
render(partial: 'hotels/hotel', collection: @hotels, cached: true)
```

Will load a file named `hotels/hotel.erb`.

Another option is to include the full filename of the partial to render.

```ruby
render(partial: 'hotels/hotel.html.erb', collection: @hotels, cached: true)
```

Will load a file named `hotels/hotel.html.erb` in any file mime type, for example you could include this partial in a JavaScript file.

Managing dependencies

In order to correctly invalidate the cache, you need to properly define the caching dependencies. Rails is clever enough to handle common cases so you don't have to specify anything. However, sometimes, when you're dealing with custom helpers for instance, you need to explicitly define them.

Implicit dependencies Most template dependencies can be derived from calls to `render` in the template itself. Here are some examples of render calls that `ActionView::Digestor` knows how to decode:

```ruby
render partial: "comments/comment", collection: commentable.comments
render "comments/comments"
render 'comments/comments'
render('comments/comments')

render "header" translates to render("comments/header")

render(@topic)         translates to render("topics/topic")
render(topics)         translates to render("topics/topic")
render(message.topics) translates to render("topics/topic")
```

On the other hand, some calls need to be changed to make caching work properly. For instance, if you're passing a custom collection, you'll need to change:

```
render @project.documents.where(published: true)
```

to:

```
render partial: "documents/document", collection: @project.documents.where(published: true)
```

Explicit dependencies Sometimes you'll have template dependencies that can't be derived at all. This is typically the case when rendering happens in helpers. Here's an example:

```
<%= render_sortable_todolists @project.todolists %>
```

You'll need to use a special comment format to call those out:

```
<%# Template Dependency: todolists/todolist %>
<%= render_sortable_todolists @project.todolists %>
```

In some cases, like a single table inheritance setup, you might have a bunch of explicit dependencies. Instead of writing every template out, you can use a wildcard to match any template in a directory:

```
<%# Template Dependency: events/* %>
<%= render_categorizable_events @person.events %>
```

As for collection caching, if the partial template doesn't start with a clean cache call, you can still benefit from collection caching by adding a special comment format anywhere in the template, like:

```
<%# Template Collection: notification %>
<% my_helper_that_calls_cache(some_arg, notification) do %>
  <%= notification.name %>
<% end %>
```

External dependencies If you use a helper method, for example, inside a cached block and you then update that helper, you'll have to bump the cache as well. It doesn't really matter how you do it, but the MD5 of the template file must change. One recommendation is to simply be explicit in a comment, like:

```
<%# Helper Dependency Updated: Jul 28, 2015 at 7pm %>
<%= some_helper_method(person) %>
```

Low-Level Caching

Sometimes you need to cache a particular value or query result instead of caching view fragments. Rails' caching mechanism works great for storing **any** kind of information.

The most efficient way to implement low-level caching is using the `Rails.cache.fetch` method. This method does both reading and writing to the cache. When passed only a single argument, the key is fetched and value from the cache is returned. If a block is passed, that block will be executed in the event of a cache miss. The return value of the block will be written to the cache under the given cache key, and that return value will be returned. In case of cache hit, the cached value will be returned without executing the block.

Consider the following example. An application has a `Product` model with an instance method that looks up the product's price on a competing website. The data returned by this method would be perfect for low-level caching:

```
class Product < ApplicationRecord
  def competing_price
    Rails.cache.fetch("#{cache_key_with_version}/competing_price", expires_in: 12.hours) do
      Competitor::API.find_price(id)
    end
  end
end
```

NOTE: Notice that in this example we used the `cache_key_with_version` method, so the resulting cache key will be something like `products/233-20140225082222765838000/competing`. `cache_key_with_version` generates a string based on the model's class name, `id`, and `updated_at` attributes. This is a common convention and has the benefit of invalidating the cache whenever the product is updated. In general, when you use low-level caching, you need to generate a cache key.

Avoid caching instances of Active Record objects Consider this example, which stores a list of Active Record objects representing superusers in the cache:

```
# super_admins is an expensive SQL query, so don't run it too often
Rails.cache.fetch("super_admin_users", expires_in: 12.hours) do
  User.super_admins.to_a
end
```

You should **avoid** this pattern. Why? Because the instance could change. In production, attributes on it could differ, or the record could be deleted. And in development, it works unreliably with cache stores that reload code when you make changes.

Instead, cache the ID or some other primitive data type. For example:

```
# super_admins is an expensive SQL query, so don't run it too often
ids = Rails.cache.fetch("super_admin_user_ids", expires_in: 12.hours) do
  User.super_admins.pluck(:id)
end
User.where(id: ids).to_a
```

SQL Caching

Query caching is a Rails feature that caches the result set returned by each query. 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 < ApplicationController

  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 with low-level caching.

Cache Stores

Rails provides different stores for the cached data (apart from SQL and page caching).

Configuration

You can set up your application's default cache store by setting the `config.cache_store` configuration option. Other parameters can be passed as arguments to the cache store's constructor:

```
config.cache_store = :memory_store, { size: 64.megabytes }
```

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

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

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 that can be 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.
- `:compress` - Enabled by default. Compresses cache entries so more data can be stored in the same memory footprint, leading to fewer cache evictions and higher hit rates.
- `:compress_threshold` - Defaults to 1kB. Cache entries larger than this threshold, specified in bytes, are compressed.
- `:expires_in` - This option sets an expiration time in seconds for the cache entry, if the cache store supports it, 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.
- `:coder` - This option allows to replace the default cache entry serialization mechanism by a custom one. The `coder` must respond to `dump` and `load`, and passing a custom coder disable automatic compression.

Connection Pool Options By default the `MemCacheStore` and `RedisCacheStore` use a single connection per process. This means that if you're using Puma, or another threaded server, you can have multiple threads waiting for the connection to become available. To increase the number of available connections you can enable connection pooling.

First, add the `connection_pool` gem to your Gemfile:

```
gem 'connection_pool'
```

Next, pass the `:pool_size` and/or `:pool_timeout` options when configuring the cache store:

```
config.cache_store = :mem_cache_store, "cache.example.com", { pool_size: 5, pool_timeout: 5
```

- `:pool_size` - This option sets the number of connections per process (defaults to 5).
- `:pool_timeout` - This option sets the number of seconds to wait for a connection (defaults to 5). If no connection is available within the timeout, a `Timeout::Error` will be raised.

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

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

```
config.cache_store = MyCacheStore.new
```

ActiveSupport::Cache::MemoryStore

This cache store keeps entries in memory in the same Ruby process. The cache store has a bounded size specified by sending the `:size` option 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.

```
config.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 Phusion Passenger or puma clustered mode), 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. However, it can work well for small, low traffic sites with only a couple of server processes, as well as development and test environments.

New Rails projects are configured to use this implementation in development environment by default.

NOTE: Since processes will not share cache data when using `:memory_store`, it will not be possible to manually read, write, or expire the cache via the Rails console.

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.

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

With this cache store, multiple server processes on the same host can share a cache. This cache store is appropriate for low to medium traffic sites that are served off one or two hosts. Server processes running on different hosts could share a cache by using a shared file system, but that setup is not recommended.

As the cache will grow until the disk is full, it is recommended to periodically clear out old entries.

This is the default cache store implementation (at `"#{root}/tmp/cache/"`) if no explicit `config.cache_store` is supplied.

ActiveSupport::Cache::MemCacheStore

This cache store uses Danga's `memcached` server to provide a centralized cache for your application. Rails uses the bundled `dalli` 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 high performance and redundancy.

When initializing the cache, you should specify the addresses for all memcached servers in your cluster, or ensure the `MEMCACHE_SERVERS` environment variable has been set appropriately.

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

If neither are specified, it will assume memcached is running on localhost on the default port (`127.0.0.1:11211`), but this is not an ideal setup for larger sites.

```
config.cache_store = :mem_cache_store # Will fallback to $MEMCACHE_SERVERS, then 127.0.0.1:
```

See the `Dalli::Client` documentation for supported address types.

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 operations 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.

ActiveSupport::Cache::RedisCacheStore

The Redis cache store takes advantage of Redis support for automatic eviction when it reaches max memory, allowing it to behave much like a Memcached cache server.

Deployment note: Redis doesn't expire keys by default, so take care to use a dedicated Redis cache server. Don't fill up your persistent-Redis server with volatile cache data! Read the Redis cache server setup guide in detail.

For a cache-only Redis server, set `maxmemory-policy` to one of the variants of allkeys. Redis 4+ supports least-frequently-used eviction (`allkeys-lfu`),

an excellent default choice. Redis 3 and earlier should use least-recently-used eviction (`allkeys-lru`).

Set cache read and write timeouts relatively low. Regenerating a cached value is often faster than waiting more than a second to retrieve it. Both read and write timeouts default to 1 second, but may be set lower if your network is consistently low-latency.

By default, the cache store will not attempt to reconnect to Redis if the connection fails during a request. If you experience frequent disconnects you may wish to enable reconnect attempts.

Cache reads and writes never raise exceptions; they just return `nil` instead, behaving as if there was nothing in the cache. To gauge whether your cache is hitting exceptions, you may provide an `error_handler` to report to an exception gathering service. It must accept three keyword arguments: `method`, the cache store method that was originally called; `returning`, the value that was returned to the user, typically `nil`; and `exception`, the exception that was rescued.

To get started, add the redis gem to your Gemfile:

```
gem 'redis'
```

You can enable support for the faster hiredis connection library by additionally adding its ruby wrapper to your Gemfile:

```
gem 'hiredis'
```

Redis cache store will automatically require and use hiredis if available. No further configuration is needed.

Finally, add the configuration in the relevant `config/environments/*.rb` file:

```
config.cache_store = :redis_cache_store, { url: ENV['REDIS_URL'] }
```

A more complex, production Redis cache store may look something like this:

```
cache_servers = %w(redis://cache-01:6379/0 redis://cache-02:6379/0)
config.cache_store = :redis_cache_store, { url: cache_servers,

  connect_timeout:    30,  # Defaults to 20 seconds
  read_timeout:       0.2, # Defaults to 1 second
  write_timeout:      0.2, # Defaults to 1 second
  reconnect_attempts: 1,   # Defaults to 0

  error_handler: -> (method:, returning:, exception:) {
    # Report errors to Sentry as warnings
    Raven.capture_exception exception, level: 'warning',
      tags: { method: method, returning: returning }
  }
}
```

ActiveSupport::Cache::NullStore

This cache store is scoped to each web request, and clears stored values at the end of a request. It is meant for use in development and test environments. It can be very useful when you have code that interacts directly with `Rails.cache` but caching interferes with seeing the results of code changes.

```
config.cache_store = :null_store
```

Cache Keys

The keys used in a cache can be any object that responds to either `cache_key` or `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 `dalli` gem. However, you also don't need to worry about exceeding the memcached size limit or violating syntax rules.

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.cache_key_with_version
      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
```

Instead of an options hash, you can also simply pass in a model. Rails will use the `updated_at` and `cache_key_with_version` methods for setting `last_modified` and `etag`:

```
class ProductsController < ApplicationController
  def show
    @product = Product.find(params[:id])

    if stale?(@product)
      respond_to do |wants|
        # ... normal response processing
      end
    end
  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
```

Sometimes we want to cache response, for example a static page, that never gets expired. To achieve this, we can use `http_cache_forever` helper and by doing

so browser and proxies will cache it indefinitely.

By default cached responses will be private, cached only on the user's web browser. To allow proxies to cache the response, set `public: true` to indicate that they can serve the cached response to all users.

Using this helper, `last_modified` header is set to `Time.new(2011, 1, 1).utc` and `expires` header is set to a 100 years.

WARNING: Use this method carefully as browser/proxy won't be able to invalidate the cached response unless browser cache is forcefully cleared.

```
class HomeController < ApplicationController
  def index
    http_cache_forever(public: true) do
      render
    end
  end
end
```

Strong v/s Weak ETags

Rails generates weak ETags by default. Weak ETags allow semantically equivalent responses to have the same ETags, even if their bodies do not match exactly. This is useful when we don't want the page to be regenerated for minor changes in response body.

Weak ETags have a leading `W/` to differentiate them from strong ETags.

```
W/"618bbc92e2d35ea1945008b42799b0e7" → Weak ETag
"618bbc92e2d35ea1945008b42799b0e7" → Strong ETag
```

Unlike weak ETag, strong ETag implies that response should be exactly the same and byte by byte identical. Useful when doing Range requests within a large video or PDF file. Some CDNs support only strong ETags, like Akamai. If you absolutely need to generate a strong ETag, it can be done as follows.

```
class ProductsController < ApplicationController
  def show
    @product = Product.find(params[:id])
    fresh_when last_modified: @product.published_at.utc, strong_etag: @product
  end
end
```

You can also set the strong ETag directly on the response.

```
response.strong_etag = response.body # => "618bbc92e2d35ea1945008b42799b0e7"
```

Caching in Development

It's common to want to test the caching strategy of your application in development mode. Rails provides the rails command `dev:cache` to easily toggle caching on/off.

```
$ bin/rails dev:cache
Development mode is now being cached.
$ bin/rails dev:cache
Development mode is no longer being cached.
```

NOTE: By default, when development mode caching is *off*, Rails uses `ActiveSupport::Cache::NullStore`.

References

- DHH's article on key-based expiration
- Ryan Bates' Railscast on cache digests