

## Lograge - Taming Rails' Default Request Logging

Lograge is an attempt to bring sanity to Rails' noisy and unusable, unparsable and, in the context of running multiple processes and servers, unreadable default logging output. Rails' default approach to log everything is great during development, it's terrible when running it in production. It pretty much renders Rails logs useless to me.

Lograge is a work in progress. I appreciate constructive feedback and criticism. My main goal is to improve Rails' logging and to show people that they don't need to stick with its defaults anymore if they don't want to.

Instead of trying solving the problem of having multiple lines per request by switching Rails' logger for something that outputs syslog lines or adds a request token, Lograge replaces Rails' request logging entirely, reducing the output per request to a single line with all the important information, removing all that clutter Rails likes to include and that gets mingled up so nicely when multiple processes dump their output into a single file.

Instead of having an unparsable amount of logging output like this:

```
1 Started GET "/" for 127.0.0.1 at 2012-03-10 14:28:14 +0100
2 Processing by HomeController#index as HTML
3   Rendered text template within layouts/application (0.0ms)
4   Rendered layouts/_assets.html.erb (2.0ms)
5   Rendered layouts/_top.html.erb (2.6ms)
6   Rendered layouts/_about.html.erb (0.3ms)
7   Rendered layouts/_google_analytics.html.erb (0.4ms)
8 Completed 200 OK in 79ms (Views: 78.8ms | ActiveRecord: 0.0ms)
```

you get a single line with all the important information, like this:

```
1 method=GET path=/ format=json controller=HomeController action=index
  status=200 duration=79.0 view=78.8 db=0.0
```

The second line is easy to grasp with a single glance and still includes all the relevant information as simple key-value pairs. The syntax is heavily inspired by the log output of the Heroku router. It doesn't include any timestamp by default, instead it assumes you use a proper log formatter instead.

### Supported Ruby and Rails Releases

Lograge is actively tested against current and officially supported Ruby and Rails releases. That said, Lograge *should* work with older releases.

- 
- Rails: Edge, 6.1, 6.0, 5.2
  - Rubies:
    - MRI: HEAD, 3.1.0-preview1, 3.0, 2.7, 2.6
    - JRuby: HEAD, 9.2, 9.1
    - TruffleRuby: HEAD, 21.3

## Installation

In your Gemfile

```
1 gem "lograge"
```

Enable it in an initializer or the relevant environment config:

```
1 # config/initializers/lograge.rb
2 # OR
3 # config/environments/production.rb
4 Rails.application.configure do
5   config.lograge.enabled = true
6 end
```

If you're using Rails 5's API-only mode and inherit from `ActionController::API`, you must define it as the controller base class which lograge will patch:

```
1 # config/initializers/lograge.rb
2 Rails.application.configure do
3   config.lograge.base_controller_class = 'ActionController::API'
4 end
```

If you use multiple base controller classes in your application, specify an array:

```
1 # config/initializers/lograge.rb
2 Rails.application.configure do
3   config.lograge.base_controller_class = ['ActionController::API', '
    ActionController::Base']
4 end
```

You can also add a hook for own custom data

```
1 # config/environments/staging.rb
2 Rails.application.configure do
3   config.lograge.enabled = true
4
5   # custom_options can be a lambda or hash
6   # if it's a lambda then it must return a hash
7   config.lograge.custom_options = lambda do |event|
8     # capture some specific timing values you are interested in
```

---

```
9      { :name => "value", :timing => some_float.round(2), :host => event.  
10        payload[:host] }  
11    end  
end
```

Or you can add a timestamp:

```
1  Rails.application.configure do  
2    config.lograge.enabled = true  
3  
4    # add time to lograge  
5    config.lograge.custom_options = lambda do |event|  
6      { time: Time.now }  
7    end  
8  end
```

You can also keep the original (and verbose) Rails logger by following this configuration:

```
1  Rails.application.configure do  
2    config.lograge.keep_original_rails_log = true  
3  
4    config.lograge.logger = ActiveSupport::Logger.new "#{Rails.root}/log/  
5      lograge_#{Rails.env}.log"  
6  end
```

You can then add custom variables to the event to be used in `custom_options` (available via the `event.payload` hash, which has to be processed in `custom_options` method to be included in log output, see above):

```
1  # app/controllers/application_controller.rb  
2  class ApplicationController < ActionController::Base  
3    def append_info_to_payload(payload)  
4      super  
5      payload[:host] = request.host  
6    end  
7  end
```

Alternatively, you can add a hook for accessing controller methods directly (e.g. `request` and `current_user`). This hash is merged into the log data automatically.

```
1  Rails.application.configure do  
2    config.lograge.enabled = true  
3  
4    config.lograge.custom_payload do |controller|  
5      {  
6        host: controller.request.host,  
7        user_id: controller.current_user.try(:id)  
8      }  
9    end  
10  end
```

---

To further clean up your logging, you can also tell Lograge to skip log messages meeting given criteria. You can skip log messages generated from certain controller actions, or you can write a custom handler to skip messages based on data in the log event:

```
1 # config/environments/production.rb
2 Rails.application.configure do
3   config.lograge.enabled = true
4
5   config.lograge.ignore_actions = ['HomeController#index', 'AController
#an_action']
6   config.lograge.ignore_custom = lambda do |event|
7     # return true here if you want to ignore based on the event
8   end
9 end
```

Lograge supports multiple output formats. The most common is the default lograge key-value format described above. Alternatively, you can also generate JSON logs in the `json_event` format used by Logstash.

```
1 # config/environments/production.rb
2 Rails.application.configure do
3   config.lograge.formatter = Lograge::Formatters::Logstash.new
4 end
```

*Note:* When using the logstash output, you need to add the additional gem `logstash-event`. You can simply add it to your Gemfile like this

```
1 gem "logstash-event"
```

Done.

The available formatters are:

```
1 Lograge::Formatters::Lines.new
2 Lograge::Formatters::Cee.new
3 Lograge::Formatters::Graylog2.new
4 Lograge::Formatters::KeyValue.new # default lograge format
5 Lograge::Formatters::KeyValueDeep.new
6 Lograge::Formatters::Json.new
7 Lograge::Formatters::Logstash.new
8 Lograge::Formatters::LTSV.new
9 Lograge::Formatters::Raw.new # Returns a ruby hash object
```

In addition to the formatters, you can manipulate the data yourself by passing an object which responds to `#call`:

```
1 # config/environments/production.rb
```

---

```
2 Rails.application.configure do
3   config.lograge.formatter = ->(data) { "Called #{data[:controller]}" }
4   # data is a ruby hash
5 end
```

## Internals

Thanks to the notification system that was introduced in Rails 3, replacing the logging is easy. Lograge unhooks all subscriptions from `ActionController::LogSubscriber` and `ActionView::LogSubscriber`, and hooks in its own log subscription, but only listening for two events: `process_action` and `redirect_to` (in case of standard controller logs). It makes sure that only subscriptions from those two classes are removed. If you happened to hook in your own, they'll be safe.

Unfortunately, when a redirect is triggered by your application's code, ActionController fires two events. One for the redirect itself, and another one when the request is finished. Unfortunately, the final event doesn't include the redirect, so Lograge stores the redirect URL as a thread-local attribute and refers to it in `process_action`.

The event itself contains most of the relevant information to build up the log line, including view processing and database access times.

While the LogSubscribers encapsulate most logging pretty nicely, there are still two lines that show up no matter what. The first line that's output for every Rails request, you know, this one:

```
1 Started GET "/" for 127.0.0.1 at 2012-03-12 17:10:10 +0100
```

And the verbose output coming from rack-cache:

```
1 cache: [GET /] miss
```

Both are independent of the LogSubscribers, and both need to be shut up using different means.

For the first one, the starting line of every Rails request log, Lograge replaces code in `Rails::Rack::Logger` to remove that particular log line. It's not great, but it's just another unnecessary output and would still clutter the log files. Maybe a future version of Rails will make this log line an event as well.

To remove rack-cache's output (which is only enabled if caching in Rails is enabled), Lograge disables verbosity for rack-cache, which is unfortunately enabled by default.

There, a single line per request. Beautiful.

---

## Action Cable

Starting with version 0.11.0, Lograge introduced support for Action Cable logs. This proved to be a particular challenge since the framework code is littered with multiple (and seemingly random) logger calls in a number of internal classes. In order to deal with it, the default Action Cable logger was silenced. As a consequence, calling logger e.g. in user-defined `Connection` or `Channel` classes has no effect - `Rails.logger` (or any other logger instance) has to be used instead.

Additionally, while standard controller logs rely on `process_action` and `redirect_to` instrumentations only, Action Cable messages are generated from multiple events: `perform_action`, `subscribe`, `unsubscribe`, `connect`, and `disconnect`. `perform_action` is the only one included in the actual Action Cable code and others have been added by monkey patching `ActionCable::Channel::Base` and `ActionCable::Connection::Base` classes.

## What it doesn't do

Lograge is opinionated, very opinionated. If the stuff below doesn't suit your needs, it may not be for you.

Lograge removes `ActionView` logging, which also includes rendering times for partials. If you're into those, Lograge is probably not for you. In my honest opinion, those rendering times don't belong in the log file, they should be collected in a system like New Relic, Librato Metrics or some other metrics service that allows graphing rendering percentiles. I assume this for everything that represents a moving target. That kind of data is better off being visualized in graphs than dumped (and ignored) in a log file.

Lograge doesn't yet log the request parameters. This is something I'm actively contemplating, mainly because I want to find a good way to include them, a way that fits in with the general spirit of the log output generated by Lograge. If you decide to include them be sure that sensitive data like passwords and credit cards are not stored via `filtered_parameters` or another means. The payload does already contain the params hash, so you can easily add it in manually using `custom_options`:

```
1 # production.rb
2 YourApp::Application.configure do
3   config.lograge.enabled = true
4   config.lograge.custom_options = lambda do |event|
5     exceptions = %w(controller action format id)
6     {
7       params: event.payload[:params].except(*exceptions)
8     }
9   end
10 end
```

---

## FAQ

### Logging errors / exceptions

Our first recommendation is that you use exception tracking services built for purpose ;)

If you absolutely *must* log exceptions in the single-line format, you can do something similar to this example:

```
1 # config/environments/production.rb
2
3 YourApp::Application.configure do
4   config.lograge.enabled = true
5   config.lograge.custom_options = lambda do |event|
6     {
7       exception: event.payload[:exception], # ["ExceptionClass", "the
8         message"]
9       exception_object: event.payload[:exception_object] # the
10         exception instance
11     }
12   end
13 end
```

The `:exception` is just the basic class and message whereas the `:exception_object` is the actual exception instance. You can use both / either. Be mindful when including this, you will probably want to cherry-pick particular attributes and almost definitely want to `join` the `backtrace` into something without newline characters.

### Handle ActionController::RoutingError

Add a `get '*unmatched_route'`, to: `'application#route_not_found'` rule to the end of your `routes.rb`. Then add a new controller action in your `application_controller.rb`.

```
1 def route_not_found
2   render 'error_pages/404', status: :not_found
3 end
```

#146

### Alternative & Related Projects

- `rails_semantic_logger` is a similar project with different functionality.
- `simple_structured_logger` adds structured logging to the rest of your application

---

## **Contributing**

See the CONTRIBUTING.md file for further information.

## **License**

MIT. Code extracted from Travis CI.

(c) Mathias Meyer

See [LICENSE.txt](#) for details.