Debugging Rails Applications

Ruby on Rails

Debugging Rails Applications

Debugging Rails Applications

This guide introduces techniques for debugging Ruby on Rails applications. After reading this guide, you will know:

- The purpose of debugging.
- How to track down problems and issues in your application that your tests aren't identifying.
- The different ways of debugging.
- How to analyze the stack trace.

1 View Helpers for Debugging

One common task is to inspect the contents of a variable. In Rails, you can do this with three methods:

- `debug`
- `to_yaml`
- `inspect`

1.1 `debug`

The `debug` helper will return a

```
tag that renders the object using the YAML format. This will generate human-reada
```

```
<%= debug @article %>
<p>
  <b>Title:</b>
  <%= @article.title %>
</p>
```

```
You'll see something like this:
```

```
--- !ruby/object Article
attributes:
  updated_at: 2008-09-05 22:55:47
  body: It's a very helpful guide for debugging your Rails app.
  title: Rails debugging guide
  published: t
  id: "1"
  created_at: 2008-09-05 22:55:47
attributes_cache: {}


Title: Rails debugging guide
```

1.2 `to_yaml`

Displaying an instance variable, or any other object or method, in YAML format can be achieved this way:

```
<%= simple_format @article.to_yaml %>
<p>
```

```
  <b>Title:</b>
  <%= @article.title %>
</p>
```

The `to_yaml` method converts the method to YAML format leaving it more readable, and then the `simple_format` helper is used to render each line as in the console. This is how `debug` method does its magic.

As a result of this, you will have something like this in your view:

```
--- !ruby/object Article
attributes:
updated_at: 2008-09-05 22:55:47
body: It's a very helpful guide for debugging your Rails app.
title: Rails debugging guide
published: t
id: "1"
created_at: 2008-09-05 22:55:47
attributes_cache: {}

Title: Rails debugging guide
```

1.3 `inspect`

Another useful method for displaying object values is `inspect`, especially when working with arrays or hashes. This will print the object value as a string. For example:

```
<%= [1, 2, 3, 4, 5].inspect %>
<p>
  <b>Title:</b>
  <%= @article.title %>
</p>
```

Will be rendered as follows:

```
[1, 2, 3, 4, 5]

Title: Rails debugging guide
```

2 The Logger

It can also be useful to save information to log files at runtime. Rails maintains a separate log file for each runtime environment.

2.1 What is the Logger?

Rails makes use of the `ActiveSupport::Logger` class to write log information. You can also substitute another logger such as `Log4r` if you wish.

You can specify an alternative logger in your `environment.rb` or any environment file:

```
Rails.logger = Logger.new(STDOUT)
Rails.logger = Log4r::Logger.new("Application Log")
```

Or in the `Initializer` section, add of the following

```
config.logger = Logger.new(STDOUT)
config.logger = Log4r::Logger.new("Application Log")
```

Info: By default, each log is created under `Rails.root/log/` and the log file is named after the environment in which the application is running.

2.2 Log Levels

When something is logged it's printed into the corresponding log if the log level of the message is equal or higher than the configured log level. If you want to know the current log level you can call the `Rails.logger.level` method.

The available log levels are: `:debug`, `:info`, `:warn`, `:error`, `:fatal`, and `:unknown`, corresponding to the log level numbers from 0 up to 5 respectively. To change the default log level, use

```
config.log_level = :warn # In any environment initializer, or
Rails.logger.level = 0 # at any time
```

This is useful when you want to log under development or staging, but you don't want to flood your production log with unnecessary information.

Info: The default Rails log level is `debug` in all environments.

2.3 Sending Messages

To write in the current log use the `logger.(debug|info|warn|error|fatal)` method from within a controller, model or mailer:

```
logger.debug "Person attributes hash: #{@person.attributes.inspect}"
logger.info "Processing the request..."
logger.fatal "Terminating application, raised unrecoverable error!!!"
```

Here's an example of a method instrumented with extra logging:

```
class ArticlesController < ApplicationController
  # ...

  def create
    @article = Article.new(params[:article])
    logger.debug "New article: #{@article.attributes.inspect}"
    logger.debug "Article should be valid: #{@article.valid?}"

    if @article.save
      flash[:notice] =  'Article was successfully created.'
      logger.debug "The article was saved and now the user is going to be redirec
      redirect_to(@article)
    else
      render action: "new"
    end
  end

  # ...
end
```

Here's an example of the log generated when this controller action is executed:

```
Processing ArticlesController#create (for 127.0.0.1 at 2008-09-08 11:52:54) [POS
  Session ID: BAh7BzoMY3NyZl9pZCIlMDY5MWU1M2I1ZDRjODBlMzkyMWI1OTg2NWQyNzViZjYiCm
vbkNvbnRyb2xsZXI6OkZsYXNoOjpGbGFzaEhhc2h7AAY6CkB1c2VkewA=--b18cd92fba90eacf8137e
  Parameters: {"commit"=>"Create", "article"=>{"title"=>"Debugging Rails",
 "body"=>"I'm learning how to print in logs!!!", "published"=>"0"},
 "authenticity_token"=>"2059c1286e93402e389127b1153204e0d1e275dd", "action"=>"cre
New article: {"updated_at"=>nil, "title"=>"Debugging Rails", "body"=>"I'm learnin
 "published"=>false, "created_at"=>nil}
Article should be valid: true
  Article Create (0.000443)   INSERT INTO "articles" ("updated_at", "title", "boo
 "created_at") VALUES('2008-09-08 14:52:54', 'Debugging Rails',
 'I''m learning how to print in logs!!!', 'f', '2008-09-08 14:52:54')
The article was saved and now the user is going to be redirected...
Redirected to # Article:0x20af760>
Completed in 0.01224 (81 reqs/sec) | DB: 0.00044 (3%) | 302 Found [http://localho
```

Adding extra logging like this makes it easy to search for unexpected or unusual behavior in your logs. If you add extra logging, be sure to make sensible use of log levels to avoid filling your production logs with useless trivia.

2.4 Tagged Logging

When running multi-user, multi-account applications, it's often useful to be able to filter the logs using some custom rules. `TaggedLogging` in Active Support helps in doing exactly that by stamping log lines with subdomains, request ids, and anything else to aid debugging such applications.

```
logger = ActiveSupport::TaggedLogging.new(Logger.new(STDOUT))
logger.tagged("BCX") { logger.info "Stuff" }                            # Logs "
logger.tagged("BCX", "Jason") { logger.info "Stuff" }                   # Logs "
logger.tagged("BCX") { logger.tagged("Jason") { logger.info "Stuff" } } # Logs "
```

2.5 Impact of Logs on Performance

Logging will always have a small impact on performance of your rails app, particularly when logging to disk. However, there are a few subtleties:

Using the `:debug` level will have a greater performance penalty than `:fatal`, as a far greater number of strings are being evaluated and written to the log output (e.g. disk).

Another potential pitfall is that if you have many calls to `Logger` like this in your code:

```
logger.debug "Person attributes hash: #{@person.attributes.inspect}"
```

In the above example, There will be a performance impact even if the allowed output level doesn't include debug. The reason is that Ruby has to evaluate these strings, which includes instantiating the somewhat heavy `String` object and interpolating the variables, and which takes time. Therefore, it's recommended to pass blocks to the logger methods, as these are only evaluated if the output level is the same or included in the allowed level (i.e. lazy loading). The same code rewritten would be:

```
logger.debug {"Person attributes hash: #{@person.attributes.inspect}"}
```

The contents of the block, and therefore the string interpolation, is only evaluated if debug is enabled. This performance savings is only really noticeable with large amounts of logging, but it's a good practice to employ.

3 Debugging with the `byebug` gem

When your code is behaving in unexpected ways, you can try printing to logs or the console to diagnose the problem. Unfortunately, there are times when this sort of error tracking is not effective in finding the root cause of a problem. When you actually need to journey into your running source code, the debugger is your best companion.

The debugger can also help you if you want to learn about the Rails source code but don't know where to start. Just debug any request to your application and use this guide to learn how to move from the code you have written deeper into Rails code.

3.1 Setup

You can use the `byebug` gem to set breakpoints and step through live code in Rails. To install it, just run:

```
$ gem install byebug
```

Inside any Rails application you can then invoke the debugger by calling the `byebug` method.

Here's an example:

```
class PeopleController < ApplicationController
  def new
    byebug
    @person = Person.new
  end
end
```

3.2 The Shell

As soon as your application calls the `byebug` method, the debugger will be started in a debugger shell inside the terminal window where you launched your application server, and you will be placed at the debugger's prompt `(byebug)`. Before the prompt, the code around the line that is about to be run will be displayed and the current line will be marked by '=>'. Like this:

```
[1, 10] in /PathTo/project/app/controllers/articles_controller.rb
    3:
    4:   # GET /articles
    5:   # GET /articles.json
    6:   def index
    7:     byebug
=>  8:     @articles = Article.find_recent
    9:
   10:     respond_to do |format|
   11:       format.html # index.html.erb
   12:        format.json { render json: @articles }

(byebug)
```

If you got there by a browser request, the browser tab containing the request will be hung until the debugger has finished and the trace has finished processing the entire request.

For example:

```
=> Booting WEBrick
=> Rails 4.2.0 application starting in development on http://0.0.0.0:3000
=> Run `rails server -h` for more startup options
=> Notice: server is listening on all interfaces (0.0.0.0). Consider using 127.0
=> Ctrl-C to shutdown server
[2014-04-11 13:11:47] INFO  WEBrick 1.3.1
[2014-04-11 13:11:47] INFO  ruby 2.1.1 (2014-02-24) [i686-linux]
[2014-04-11 13:11:47] INFO  WEBrick::HTTPServer#start: pid=6370 port=3000


Started GET "/" for 127.0.0.1 at 2014-04-11 13:11:48 +0200
  ActiveRecord::SchemaMigration Load (0.2ms)  SELECT "schema_migrations".* FROM '
Processing by ArticlesController#index as HTML

[3, 12] in /PathTo/project/app/controllers/articles_controller.rb
    3:
    4:   # GET /articles
    5:   # GET /articles.json
    6:   def index
    7:     byebug
=>  8:     @articles = Article.find_recent
    9:
   10:     respond_to do |format|
   11:       format.html # index.html.erb
   12:       format.json { render json: @articles }

(byebug)
```

Now it's time to explore and dig into your application. A good place to start is by asking the debugger for help. Type: `help`

```
(byebug) help

byebug 2.7.0

Type 'help <command-name>' for help on a specific command

Available commands:
backtrace  delete   enable  help       list    pry next  restart  source     up
break      disable  eval    info       method  ps        save     step       var
catch      display  exit    interrupt  next    putl      set      thread
condition  down     finish  irb        p       quit      show     trace
continue   edit     frame   kill       pp      reload    skip     undisplay
```

Info: To view the help menu for any command use `help <command-name>` at the debugger prompt.

For example: . You can abbreviate any debugging command by supplying just enough letters to distinguish them from other commands, so you can also use `l` for the `list` command, for example.

To see the previous ten lines you should type `list-` (or `l-`)

```
(byebug) l-

[1, 10] in /PathTo/project/app/controllers/articles_controller.rb
   1  class ArticlesController < ApplicationController
   2    before_action :set_article, only: [:show, :edit, :update, :destroy]
   3
   4    # GET /articles
   5    # GET /articles.json
   6    def index
   7      byebug
   8      @articles = Article.find_recent
   9
   10      respond_to do |format|
```

This way you can move inside the file, being able to see the code above and over the line where you added the `byebug` call. Finally, to see where you are in the code again you can type `list=`

```
(byebug) list=

[3, 12] in /PathTo/project/app/controllers/articles_controller.rb
    3:
    4:    # GET /articles
    5:    # GET /articles.json
    6:   def index
    7:     byebug
=>  8:     @articles = Article.find_recent
    9:
   10:     respond_to do |format|
   11:       format.html # index.html.erb
   12:       format.json { render json: @articles }

(byebug)
```

3.3 The Context

When you start debugging your application, you will be placed in different contexts as you go through the different parts of the stack.

The debugger creates a context when a stopping point or an event is reached. The context has information about the suspended program which enables the debugger to inspect the frame stack, evaluate variables from the perspective of the debugged program, and contains information about the place where the debugged program is stopped.

At any time you can call the `backtrace` command (or its alias `where`) to print the backtrace of the application. This can be very helpful to know how you got where you are. If you ever wondered about how you got somewhere in your code, then `backtrace` will supply the answer.

```
(byebug) where
--> #0  ArticlesController.index
      at /PathTo/project/test_app/app/controllers/articles_controller.rb:8
    #1  ActionController::ImplicitRender.send_action(method#String, *args#Array)
      at /PathToGems/actionpack-4.2.0/lib/action_controller/metal/implicit_rende
    #2  AbstractController::Base.process_action(action#NilClass, *args#Array)
      at /PathToGems/actionpack-4.2.0/lib/abstract_controller/base.rb:189
    #3  ActionController::Rendering.process_action(action#NilClass, *args#NilCla
      at /PathToGems/actionpack-4.2.0/lib/action_controller/metal/rendering.rb:1
...
```

The current frame is marked with `-->`. You can move anywhere you want in this trace (thus changing the context) by using the `frame _n_` command, where is the specified frame number. If you do that, `byebug` will display your new context.

```
(byebug) frame 2

[184, 193] in /PathToGems/actionpack-4.2.0/lib/abstract_controller/base.rb
   184:       # is the intended way to override action dispatching.
   185:       #
   186:       # Notice that the first argument is the method to be dispatched
   187:       # which is *not* necessarily the same as the action name.
   188:       def process_action(method_name, *args)
=> 189:         send_action(method_name, *args)
   190:       end
   191:
   192:       # Actually call the method associated with the action. Override
   193:       # this method if you wish to change how action methods are called,

(byebug)
```

The available variables are the same as if you were running the code line by line. After all, that's what debugging is.

You can also use `up [n]` (`u` for abbreviated) and `down [n]` commands in order to change the context frames up or down the stack respectively. defaults to one. Up in this case is towards higher-numbered stack frames, and down is towards lower-numbered stack frames.

3.4 Threads

The debugger can list, stop, resume and switch between running threads by using the `thread` command (or the abbreviated `th`). This command has a handful of options:

- `thread` shows the current thread.
- `thread list` is used to list all threads and their statuses. The plus + character and the number indicates the current thread of execution.
- `thread stop _n_` stop thread .
- `thread resume _n_` resumes thread .
- `thread switch _n_` switches the current thread context to .

This command is very helpful, among other occasions, when you are debugging concurrent threads and need to verify that there are no race conditions in your code.

3.5 Inspecting Variables

Any expression can be evaluated in the current context. To evaluate an expression, just type it!

This example shows how you can print the instance variables defined within the current context:

```
[3, 12] in /PathTo/project/app/controllers/articles_controller.rb
    3:
    4:   # GET /articles
    5:   # GET /articles.json
    6:   def index
    7:     byebug
=>  8:     @articles = Article.find_recent
    9:
   10:     respond_to do |format|
   11:       format.html # index.html.erb
   12:       format.json { render json: @articles }

(byebug) instance_variables
[:@_action_has_layout, :@_routes, :@_headers, :@_status, :@_request,
 :@_response, :@_env, :@_prefixes, :@_lookup_context, :@_action_name,
 :@_response_body, :@marked_for_same_origin_verification, :@_config]
```

As you may have figured out, all of the variables that you can access from a controller are displayed. This list is dynamically updated as you execute code. For example, run the next line using `next` (you'll learn more about this command later in this guide).

```
(byebug) next
[5, 14] in /PathTo/project/app/controllers/articles_controller.rb
   5     # GET /articles.json
   6     def index
   7       byebug
   8       @articles = Article.find_recent
   9
=> 10       respond_to do |format|
   11         format.html # index.html.erb
   12        format.json { render json: @articles }
   13      end
   14    end
   15
(byebug)
```

And then ask again for the instance_variables:

```
(byebug) instance_variables.include? "@articles"
true
```

Now `@articles` is included in the instance variables, because the line defining it was executed.

Info: You can also step into mode with the command `irb` (of course!). This way an irb session will be started within the context you invoked it. But be warned: this is an experimental feature.

The `var` method is the most convenient way to show variables and their values. Let's let `byebug` to help us with it.

```
(byebug) help var
v[ar] cl[ass]                   show class variables of self
v[ar] const <object>            show constants of object
v[ar] g[lobal]                  show global variables
v[ar] i[nstance] <object>       show instance variables of object
v[ar] l[ocal]                   show local variables
```

This is a great way to inspect the values of the current context variables. For example, to check that we have no local variables currently defined.

```
(byebug) var local
(byebug)
```

You can also inspect for an object method this way:

```
(byebug) var instance Article.new
@_start_transaction_state = {}
@aggregation_cache = {}
@association_cache = {}
@attributes = {"id"=>nil, "created_at"=>nil, "updated_at"=>nil}
@attributes_cache = {}
@changed_attributes = nil
...
```

Info: The commands `p` (print) and `pp` (pretty print) can be used to evaluate Ruby expressions and display the value of variables to the console.

You can use also `display` to start watching variables. This is a good way of tracking the values of a variable while the execution goes on.

```
(byebug) display @articles
1: @articles = nil
```

The variables inside the displaying list will be printed with their values after you move in the stack. To stop displaying a variable use `undisplay _n_` where is the variable number (1 in the last example).

3.6 Step by Step

Now you should know where you are in the running trace and be able to print the available variables. But lets continue and move on with the application execution.

Use `step` (abbreviated `s`) to continue running your program until the next logical stopping point and return control to the debugger.

You may also use `next` which is similar to step, but function or method calls that appear within the line of code are executed without stopping.

Info: You can also use `step n` or `next n` to move forwards `n` steps at once.

The difference between `next` and `step` is that `step` stops at the next line of code executed, doing just a single step, while `next` moves to the next line without descending inside methods.

For example, consider the following situation:

```
Started GET "/" for 127.0.0.1 at 2014-04-11 13:39:23 +0200
Processing by ArticlesController#index as HTML

[1, 8] in /home/davidr/Proyectos/test_app/app/models/article.rb
   1: class Article < ActiveRecord::Base
   2:
   3:   def self.find_recent(limit = 10)
   4:     byebug
=> 5:     where('created_at > ?', 1.week.ago).limit(limit)
   6:   end
   7:
   8: end

(byebug)
```

If we use `next`, we want go deep inside method calls. Instead, byebug will go to the next line within the same context. In this case, this is the last line of the method, so `byebug` will jump to next next line of the previous frame.

```
(byebug) next
Next went up a frame because previous frame finished

[4, 13] in /PathTo/project/test_app/app/controllers/articles_controller.rb
    4:   # GET /articles
    5:   # GET /articles.json
    6:   def index
    7:     @articles = Article.find_recent
    8:
=>  9:     respond_to do |format|
   10:       format.html # index.html.erb
   11:       format.json { render json: @articles }
   12:     end
   13:   end

(byebug)
```

If we use `step` in the same situation, we will literally go the next ruby instruction to be executed. In this case, the activesupport's `week` method.

```
(byebug) step

[50, 59] in /PathToGems/activesupport-4.2.0/lib/active_support/core_ext/numeric/
   50:     ActiveSupport::Duration.new(self * 24.hours, [[:days, self]])
   51:   end
   52:   alias :day :days
   53:
   54:   def weeks
=> 55:     ActiveSupport::Duration.new(self * 7.days, [[:days, self * 7]])
   56:   end
   57:   alias :week :weeks
   58:
   59:   def fortnights

(byebug)
```

This is one of the best ways to find bugs in your code, or perhaps in Ruby on Rails.

3.7 Breakpoints

A breakpoint makes your application stop whenever a certain point in the program is reached. The debugger shell is invoked in that line.

You can add breakpoints dynamically with the command `break` (or just `b`). There are 3 possible ways of adding breakpoints manually:

- `break line`: set breakpoint in the in the current source file.
- `break file:line [if expression]`: set breakpoint in the number inside the . If an is given it must evaluated to to fire up the debugger.
- `break class(.|\#)method [if expression]`: set breakpoint in (. and # for class and instance method respectively) defined in . The works the same way as with file:line.

For example, in the previous situation

```
[4, 13] in /PathTo/project/app/controllers/articles_controller.rb
     4:   # GET /articles
     5:   # GET /articles.json
     6:   def index
     7:     @articles = Article.find_recent
     8:
=>   9:     respond_to do |format|
    10:       format.html # index.html.erb
    11:       format.json { render json: @articles }
    12:     end
    13:   end

(byebug) break 11
Created breakpoint 1 at /PathTo/project/app/controllers/articles_controller.rb:1
```

Use `info breakpoints _n_` or `info break _n_` to list breakpoints. If you supply a number, it lists that breakpoint. Otherwise it lists all breakpoints.

```
(byebug) info breakpoints
Num Enb What
1   y   at /PathTo/project/app/controllers/articles_controller.rb:11
```

To delete breakpoints: use the command `delete _n_` to remove the breakpoint number . If no number is specified, it deletes all breakpoints that are currently active.

```
(byebug) delete 1
(byebug) info breakpoints
No breakpoints.
```

You can also enable or disable breakpoints:

- `enable breakpoints`: allow a list or all of them if no list is specified, to stop your program. This is the default state when you create a breakpoint.
- `disable breakpoints`: the will have no effect on your program.

3.8 Catching Exceptions

The command `catch exception-name` (or just `cat exception-name`) can be used to intercept an exception of type when there would otherwise be no handler for it.

To list all active catchpoints use `catch`.

3.9 Resuming Execution

There are two ways to resume execution of an application that is stopped in the debugger:

- `continue` [line-specification] (or `c`): resume program execution, at the address where your script last stopped; any breakpoints set at that address are bypassed. The optional argument line-specification allows you to specify a line number to set a one-time breakpoint which is deleted when that breakpoint is reached.
- `finish` [frame-number] (or `fin`): execute until the selected stack frame returns. If no frame number is given, the application will run until the currently selected frame returns. The currently selected frame starts out the most-recent frame or 0 if no frame positioning (e.g up, down or frame) has been performed. If a frame number is given it will run until the specified frame returns.

3.10 Editing

Two commands allow you to open code from the debugger into an editor:

- `edit [file:line]`: edit using the editor specified by the EDITOR environment variable. A specific can also be given.

3.11 Quitting

To exit the debugger, use the `quit` command (abbreviated `q`), or its alias `exit`.

A simple quit tries to terminate all threads in effect. Therefore your server will be stopped and you will have to start it again.

3.12 Settings

`byebug` has a few available options to tweak its behaviour:

- `set autoreload`: Reload source code when changed (default: true).
- `set autolist`: Execute `list` command on every breakpoint (default: true).
- `set listsize _n_`: Set number of source lines to list by default to (default: 10)
- `set forcestep`: Make sure the `next` and `step` commands always move to a new line.

You can see the full list by using `help set`. Use `help set _subcommand_` to learn about a particular `set` command.

Info: You can save these settings in an `.byebugrc` file in your home directory. The debugger reads these global settings when it starts. For example:

```
set forcestep
set listsize 25
```

4 Debugging Memory Leaks

A Ruby application (on Rails or not), can leak memory - either in the Ruby code or at the C code level.

In this section, you will learn how to find and fix such leaks by using tool such as Valgrind.

4.1 Valgrind

Valgrind is a Linux-only application for detecting C-based memory leaks and race conditions.

There are Valgrind tools that can automatically detect many memory management and threading bugs, and profile your programs in detail. For example, if a C extension in the interpreter calls `malloc()` but doesn't properly call `free()`, this memory won't be available until the app terminates.

For further information on how to install Valgrind and use with Ruby, refer to Valgrind and Ruby by Evan Weaver.

5 Plugins for Debugging

There are some Rails plugins to help you to find errors and debug your application. Here is a list of useful plugins for debugging:

- Footnotes Every Rails page has footnotes that give request information and link back to your source via TextMate.
- Query Trace Adds query origin tracing to your logs.
- Query Reviewer This rails plugin not only runs "EXPLAIN" before each of your select queries in development, but provides a small DIV in the rendered output of each page with the summary of warnings for each query that it analyzed.
- Exception Notifier Provides a mailer object and a default set of templates for sending email notifications when errors occur in a Rails application.
- Better Errors Replaces the standard Rails error page with a new one containing more contextual information, like source code and variable inspection.
- RailsPanel Chrome extension for Rails development that will end your tailing of development.log. Have all information about your Rails app requests in the browser - in the Developer Tools panel. Provides insight to db/rendering/total times, parameter list, rendered views and more.

6 References

- ruby-debug Homepage
- debugger Homepage
- byebug Homepage
- Article: Debugging a Rails application with ruby-debug
- Ryan Bates' debugging ruby (revised) screencast
- Ryan Bates' stack trace screencast
- Ryan Bates' logger screencast
- Debugging with ruby-debug

Feedback

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

Please contribute if you see any typos or factual errors. To get started, you can read our documentation contributions section.

You may also find incomplete content, or stuff that is not up to date. Please do add any missing documentation for master. Make sure to check Edge Guides first to verify if the issues are already fixed or not on the master branch. 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.