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.

View Helpers for Debugging

One common task is to inspect the contents of a variable. Rails provides three different ways to do this:

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

`debug`

The `debug` helper will return a <pre> tag that renders the object using the YAML format. This will generate human-readable data from any object. For example, if you have this code in a view:

```
<%= 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
```

`to_yaml`

Alternatively, calling `to_yaml` on any object converts it to YAML. You can pass this converted object into the `simple_format` helper method to format the output. This is how `debug` does its magic.

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

The above code will render 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
```

`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 render:

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

Title: Rails debugging guide
```

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.

What is the Logger?

Rails makes use of the `ActiveSupport::Logger` class to write log information. Other loggers, such as `Log4r`, may also be substituted.

You can specify an alternative logger in `config/application.rb` or any other environment file, for example:

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

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

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

TIP: 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.

Log Levels

When something is logged, it's printed into the corresponding log if the log level of the message is equal to 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 without flooding your production log with unnecessary information.

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

Sending Messages

To write in the current log use the `logger.(debug|info|warn|error|fatal|unknown)` 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(article_params)
    logger.debug "New article: #{@article.attributes.inspect}"
    logger.debug "Article should be valid: #{@article.valid?}"

    if @article.save
      logger.debug "The article was saved and now the user is going to be redirected..."
      redirect_to @article, notice: 'Article was successfully created.'
    else
      render :new, status: :unprocessable_entity
    end
  end

  # ...

  private
    def article_params
      params.require(:article).permit(:title, :body, :published)
    end
end
```

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

```
Started POST "/articles" for 127.0.0.1 at 2018-10-18 20:09:23 -0400
Processing by ArticlesController#create as HTML
  Parameters: {"utf8"=>"✓",
"authenticity_token"=>"XLveDrKzF1SwaiNRPTaMtkrsTzedtebPPkmxEFIU0ordLjICSnXsSNfrdMa4ccyBjuGwnnEiQhEoMN6H1Gtz3A==",
 "article"=>{"title"=>"Debugging Rails", "body"=>"I'm learning how to print in logs.", "published"=>"0"},
"commit"=>"Create Article"}
New article: {"id"=>nil, "title"=>"Debugging Rails", "body"=>"I'm learning how to print in logs.",
"published"=>false, "created_at"=>nil, "updated_at"=>nil}
Article should be valid: true
   (0.0ms)  begin transaction
  ↳ app/controllers/articles_controller.rb:31
  Article Create (0.5ms)  INSERT INTO "articles" ("title", "body", "published", "created_at", "updated_at")
VALUES (?, ?, ?, ?, ?)  [["title", "Debugging Rails"], ["body", "I'm learning how to print in logs."],
["published", 0], ["created_at", "2018-10-19 00:09:23.216549"], ["updated_at", "2018-10-19 00:09:23.216549"]]
  ↳ app/controllers/articles_controller.rb:31
   (2.3ms)  commit transaction
  ↳ app/controllers/articles_controller.rb:31
The article was saved and now the user is going to be redirected...
Redirected to http://localhost:3000/articles/1
Completed 302 Found in 4ms (ActiveRecord: 0.8ms)
```

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.

Verbose Query Logs

When looking at database query output in logs, it may not be immediately clear why multiple database queries are triggered when a single method is called:

```
irb(main):001:0> Article.pamplemousse
  Article Load (0.4ms)  SELECT "articles".* FROM "articles"
  Comment Load (0.2ms)  SELECT "comments".* FROM "comments" WHERE "comments"."article_id" = ?  [["article_id",
1]]
  Comment Load (0.1ms)  SELECT "comments".* FROM "comments" WHERE "comments"."article_id" = ?  [["article_id",
2]]
  Comment Load (0.1ms)  SELECT "comments".* FROM "comments" WHERE "comments"."article_id" = ?  [["article_id",
3]]
=> #<Comment id: 2, author: "1", body: "Well, actually...", article_id: 1, created_at: "2018-10-19 00:56:10",
updated_at: "2018-10-19 00:56:10">
```

After running `ActiveRecord.verbose_query_logs = true` in the `bin/rails console` session to enable verbose query logs and running the method again, it becomes obvious what single line of code is generating all these discrete database calls:

```
irb(main):003:0> Article.pamplemousse
  Article Load (0.2ms)  SELECT "articles".* FROM "articles"
  ↳ app/models/article.rb:5
  Comment Load (0.1ms)  SELECT "comments".* FROM "comments" WHERE "comments"."article_id" = ?  [["article_id",
1]]
  ↳ app/models/article.rb:6
  Comment Load (0.1ms)  SELECT "comments".* FROM "comments" WHERE "comments"."article_id" = ?  [["article_id",
2]]
  ↳ app/models/article.rb:6
  Comment Load (0.1ms)  SELECT "comments".* FROM "comments" WHERE "comments"."article_id" = ?  [["article_id",
3]]
  ↳ app/models/article.rb:6
=> #<Comment id: 2, author: "1", body: "Well, actually...", article_id: 1, created_at: "2018-10-19 00:56:10",
updated_at: "2018-10-19 00:56:10">
```

Below each database statement you can see arrows pointing to the specific source filename (and line number) of the method that resulted in a database call. This can help you identify and address performance problems caused by N+1 queries: single database queries that generates multiple additional queries.

Verbose query logs are enabled by default in the development environment logs after Rails 5.2.

WARNING: We recommend against using this setting in production environments. It relies on Ruby's `Kernel#caller` method which tends to allocate a lot of memory in order to generate stacktraces of method calls.

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 you do 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 "[BCX] Stuff"
logger.tagged("BCX", "Jason") { logger.info "Stuff" }                   # Logs "[BCX] [Jason] Stuff"
logger.tagged("BCX") { logger.tagged("Jason") { logger.info "Stuff" } } # Logs "[BCX] [Jason] Stuff"
```

Impact of Logs on Performance

Logging will always have a small impact on the performance of your Rails app, particularly when logging to disk. Additionally, 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 too many calls to `Logger` 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.

Therefore, it's recommended to pass blocks to the logger methods, as these are only evaluated if the output level is the same as — 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, are only evaluated if debug is enabled. This performance savings are only really noticeable with large amounts of logging, but it's a good practice to employ.

INFO: This section was written by [Jon Cairns at a StackOverflow answer]() and it is licensed under [cc by-sa 4.0]().

Debugging with the `debug` 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 into the underlying Rails code.

Rails 7 includes the `debug` gem in the `Gemfile` of new applications generated by CRuby. By default, it is ready in the `development` and `test` environments. Please check its documentation for usage.

Entering a Debugging Session

By default, a debugging session will start after the `debug` library is required, which happens when your app boots. But don't worry, the session won't interfere your program.

To enter the debugging session, you can use `binding.break` and its aliases: `binding.b` and `debugger`. The following examples will use `debugger`:

```
class PostsController < ApplicationController
  before_action :set_post, only: %i[ show edit update destroy ]

  # GET /posts or /posts.json
  def index
    @posts = Post.all
    debugger
  end
  # ...
end
```

Once your app evaluates the debugging statement, it'll enter the debugging session:

```
Processing by PostsController#index as HTML
[2, 11] in ~/projects/rails-guide-example/app/controllers/posts_controller.rb
     2|   before_action :set_post, only: %i[ show edit update destroy ]
     3|
     4|   # GET /posts or /posts.json
     5|   def index
     6|     @posts = Post.all
=>   7|     debugger
     8|   end
     9|
    10|   # GET /posts/1 or /posts/1.json
    11|   def show
=>#0    PostsController#index at ~/projects/rails-guide-example/app/controllers/posts_controller.rb:7
  #1    ActionController::BasicImplicitRender#send_action(method="index", args=[]) at
~/.rbenv/versions/3.0.1/lib/ruby/gems/3.0.0/gems/actionpack-
7.1.0.alpha/lib/action_controller/metal/basic_implicit_render.rb:6
  # and 72 frames (use `bt' command for all frames)
(rdbg)
```

The Context

After entering the debugging session, you can type in Ruby code as you're in a Rails console or IRB.

```
(rdbg) @posts    # ruby
[]
(rdbg) self
#<PostsController:0x0000000000aeb0>
(rdbg)
```

You can also use `p` or `pp` command to evaluate Ruby expressions (e.g. when a variable name conflicts with a debugger command).

```
(rdbg) p headers    # command
=> {"X-Frame-Options"=>"SAMEORIGIN", "X-XSS-Protection"=>"1; mode=block", "X-Content-Type-Options"=>"nosniff",
"X-Download-Options"=>"noopen", "X-Permitted-Cross-Domain-Policies"=>"none", "Referrer-Policy"=>"strict-
origin-when-cross-origin"}
(rdbg) pp headers    # command
{"X-Frame-Options"=>"SAMEORIGIN",
 "X-XSS-Protection"=>"1; mode=block",
 "X-Content-Type-Options"=>"nosniff",
 "X-Download-Options"=>"noopen",
 "X-Permitted-Cross-Domain-Policies"=>"none",
 "Referrer-Policy"=>"strict-origin-when-cross-origin"}
(rdbg)
```

Besides direct evaluation, debugger also helps you collect rich amount of information through different commands. Just to name a few here:

- `info` (or `i`) - Information about current frame.
- `backtrace` (or `bt`) - Backtrace (with additional information).
- `outline` (or `o`, `ls`) - Available methods, constants, local variables, and instance variables in the current scope.

The info command

It'll give you an overview of the values of local and instance variables that are visible from the current frame.

```
(rdbg) info    # command
%self = #<PostsController:0x0000000000af78>
@_action_has_layout = true
@_action_name = "index"
@_config = {}
@_lookup_context = #<ActionView::LookupContext:0x00007fd91a037e38 @details_key=nil, @digest_cache=...
@_request = #<ActionDispatch::Request GET "http://localhost:3000/posts" for 127.0.0.1>
@_response = #<ActionDispatch::Response:0x00007fd91a03ea08 @mon_data=#<Monitor:0x00007fd91a03e8c8>...
@_response_body = nil
@_routes = nil
@marked_for_same_origin_verification = true
@posts = []
@rendered_format = nil
```

The backtrace command

When used without any options, it lists all the frames on the stack:

```
=>#0    PostsController#index at ~/projects/rails-guide-example/app/controllers/posts_controller.rb:7
  #1    ActionController::BasicImplicitRender#send_action(method="index", args=[]) at
~/.rbenv/versions/3.0.1/lib/ruby/gems/3.0.0/gems/actionpack-
7.1.0.alpha/lib/action_controller/metal/basic_implicit_render.rb:6
  #2    AbstractController::Base#process_action(method_name="index", args=[]) at
~/.rbenv/versions/3.0.1/lib/ruby/gems/3.0.0/gems/actionpack-7.1.0.alpha/lib/abstract_controller/base.rb:214
  #3    ActionController::Rendering#process_action(#arg_rest=nil) at
~/.rbenv/versions/3.0.1/lib/ruby/gems/3.0.0/gems/actionpack-
7.1.0.alpha/lib/action_controller/metal/rendering.rb:53
  #4    block in process_action at ~/.rbenv/versions/3.0.1/lib/ruby/gems/3.0.0/gems/actionpack-
7.1.0.alpha/lib/abstract_controller/callbacks.rb:221
  #5    block in run_callbacks at ~/.rbenv/versions/3.0.1/lib/ruby/gems/3.0.0/gems/activesupport-
7.1.0.alpha/lib/active_support/callbacks.rb:118
  #6    ActionText::Rendering::ClassMethods#with_renderer(renderer=#<PostsController:0x0000000000af78>) at
~/.rbenv/versions/3.0.1/lib/ruby/gems/3.0.0/gems/actiontext-7.1.0.alpha/lib/action_text/rendering.rb:20
  #7    block {|controller=#<PostsController:0x0000000000af78>, action=#<Proc:0x00007fd91985f1c0
/Users/st0012/...|} in <class:Engine> (4 levels) at
~/.rbenv/versions/3.0.1/lib/ruby/gems/3.0.0/gems/actiontext-7.1.0.alpha/lib/action_text/engine.rb:69
  #8    [C] BasicObject#instance_exec at ~/.rbenv/versions/3.0.1/lib/ruby/gems/3.0.0/gems/activesupport-
7.1.0.alpha/lib/active_support/callbacks.rb:127
  ..... and more
```

Every frame comes with:

- Frame identifier
- Call location

- Additional information (e.g. block or method arguments)

This will give you a great sense about what's happening in your app. However, you probably will notice that:

- There are too many frames (usually 50+ in a Rails app).
- Most of the frames are from Rails or other libraries you use.

Don't worry, the `backtrace` command provides 2 options to help you filter frames:

- `backtrace [num]` - only show `num` numbers of frames, e.g. `backtrace 10` .
- `backtrace /pattern/` - only show frames with identifier or location that matches the pattern, e.g. `backtrace /MyModel/` .

It's also possible to use these options together: `backtrace [num] /pattern/` .

The outline command

This command is similar to `pry` and `irb` 's `ls` command. It will show you what's accessible from you current scope, including:

- Local variables
- Instance variables
- Class variables
- Methods & their sources
- ...etc.

```
ActiveSupport::Configurable#methods: config
AbstractController::Base#methods:
  action_methods  action_name  action_name=  available_action?  controller_path  inspect
  response_body
ActionController::Metal#methods:
  content_type       content_type=  controller_name  dispatch          headers
  location           location=      media_type       middleware_stack  middleware_stack=
  middleware_stack?  performed?     request          request=          reset_session
  response           response=      response_body=   response_code     session
  set_request!       set_response!  status           status=           to_a
ActionView::ViewPaths#methods:
  _prefixes  any_templates?  append_view_path   details_for_lookup  formats     formats=  locale
  locale=    lookup_context  prepend_view_path  template_exists?    view_paths
AbstractController::Rendering#methods: view_assigns

# .....

PostsController#methods: create  destroy  edit  index  new  show  update
instance variables:
  @_action_has_layout  @_action_name    @_config  @_lookup_context                      @_request
  @_response           @_response_body  @_routes  @marked_for_same_origin_verification  @posts
  @rendered_format
class variables: @@raise_on_missing_translations  @@raise_on_open_redirects
```

Breakpoints

There are many ways to insert and trigger a breakpoint in the debugger. In additional to adding debugging statements (e.g. `debugger`) directly in your code, you can also insert breakpoints with commands:

- `break` (or `b`)
  - `break` - list all breakpoints
  - `break <num>` - set a breakpoint on the `num` line of the current file
  - `break <file:num>` - set a breakpoint on the `num` line of `file`
  - `break <Class#method>` or `break <Class.method>` - set a breakpoint on `Class#method` or `Class.method`
  - `break <expr>.<method>` - sets a breakpoint on `<expr>` result's `<method>` method.
- `catch <Exception>` - set a breakpoint that'll stop when `Exception` is raised
- `watch <@ivar>` - set a breakpoint that'll stop when the result of current object's `@ivar` is changed (this is slow)

And to remove them, you can use:

- `delete` (or `del`)
  - `delete` - delete all breakpoints
  - `delete <num>` - delete the breakpoint with id `num`

The break command

Set a breakpoint with specified line number - e.g. `b 28`

```
[20, 29] in ~/projects/rails-guide-example/app/controllers/posts_controller.rb
    20|   end
    21|
    22|   # POST /posts or /posts.json
    23|   def create
    24|     @post = Post.new(post_params)
=>  25|     debugger
    26|
    27|     respond_to do |format|
    28|       if @post.save
    29|         format.html { redirect_to @post, notice: "Post was successfully created." }
=>#0    PostsController#create at ~/projects/rails-guide-example/app/controllers/posts_controller.rb:25
  #1    ActionController::BasicImplicitRender#send_action(method="create", args=[]) at
~/.rbenv/versions/3.0.1/lib/ruby/gems/3.0.0/gems/actionpack-
7.0.0.alpha2/lib/action_controller/metal/basic_implicit_render.rb:6
  # and 72 frames (use `bt' command for all frames)
(rdbg) b 28    # break command
#0  BP - Line  /Users/st0012/projects/rails-guide-example/app/controllers/posts_controller.rb:28 (line)
```

```
(rdbg) c    # continue command
[23, 32] in ~/projects/rails-guide-example/app/controllers/posts_controller.rb
    23|   def create
    24|     @post = Post.new(post_params)
    25|     debugger
    26|
    27|     respond_to do |format|
=>  28|       if @post.save
    29|         format.html { redirect_to @post, notice: "Post was successfully created." }
    30|         format.json { render :show, status: :created, location: @post }
    31|       else
    32|         format.html { render :new, status: :unprocessable_entity }
=>#0    block {|format=#<ActionController::MimeResponds::Collec...|} in create at ~/projects/rails-guide-
example/app/controllers/posts_controller.rb:28
  #1    ActionController::MimeResponds#respond_to(mimes=[]) at
~/.rbenv/versions/3.0.1/lib/ruby/gems/3.0.0/gems/actionpack-
7.0.0.alpha2/lib/action_controller/metal/mime_responds.rb:205
  # and 74 frames (use `bt' command for all frames)

Stop by #0  BP - Line  /Users/st0012/projects/rails-guide-example/app/controllers/posts_controller.rb:28
(line)
```

Set a breakpoint on a given method call - e.g. `b @post.save`

```
[20, 29] in ~/projects/rails-guide-example/app/controllers/posts_controller.rb
    20|   end
    21|
    22|   # POST /posts or /posts.json
    23|   def create
    24|     @post = Post.new(post_params)
=>  25|     debugger
    26|
    27|     respond_to do |format|
    28|       if @post.save
    29|         format.html { redirect_to @post, notice: "Post was successfully created." }
=>#0    PostsController#create at ~/projects/rails-guide-example/app/controllers/posts_controller.rb:25
  #1    ActionController::BasicImplicitRender#send_action(method="create", args=[]) at
~/.rbenv/versions/3.0.1/lib/ruby/gems/3.0.0/gems/actionpack-
7.0.0.alpha2/lib/action_controller/metal/basic_implicit_render.rb:6
  # and 72 frames (use `bt' command for all frames)
(rdbg) b @post.save    # break command
```

```
#0  BP - Method  @post.save at /Users/st0012/.rbenv/versions/3.0.1/lib/ruby/gems/3.0.0/gems/activerecord-
7.0.0.alpha2/lib/active_record/suppressor.rb:43
```

```
(rdbg) c    # continue command
[39, 48] in ~/.rbenv/versions/3.0.1/lib/ruby/gems/3.0.0/gems/activerecord-
7.0.0.alpha2/lib/active_record/suppressor.rb
    39|         SuppressorRegistry.suppressed[name] = previous_state
    40|       end
    41|     end
    42|
    43|     def save(**) # :nodoc:
=>  44|       SuppressorRegistry.suppressed[self.class.name] ? true : super
    45|     end
    46|
    47|     def save!(**) # :nodoc:
    48|       SuppressorRegistry.suppressed[self.class.name] ? true : super
=>#0    ActiveRecord::Suppressor#save(#arg_rest=nil) at
~/.rbenv/versions/3.0.1/lib/ruby/gems/3.0.0/gems/activerecord-7.0.0.alpha2/lib/active_record/suppressor.rb:44
  #1    block {|format=#<ActionController::MimeResponds::Collec...|} in create at ~/projects/rails-guide-
example/app/controllers/posts_controller.rb:28
  # and 75 frames (use `bt' command for all frames)

Stop by #0  BP - Method  @post.save at
/Users/st0012/.rbenv/versions/3.0.1/lib/ruby/gems/3.0.0/gems/activerecord-
7.0.0.alpha2/lib/active_record/suppressor.rb:43
```

The catch command

Stop when an exception is raised - e.g. `catch ActiveRecord::RecordInvalid`

```
[20, 29] in ~/projects/rails-guide-example/app/controllers/posts_controller.rb
    20|   end
    21|
    22|   # POST /posts or /posts.json
    23|   def create
    24|     @post = Post.new(post_params)
=>  25|     debugger
    26|
    27|     respond_to do |format|
    28|       if @post.save!
    29|         format.html { redirect_to @post, notice: "Post was successfully created." }
=>#0    PostsController#create at ~/projects/rails-guide-example/app/controllers/posts_controller.rb:25
  #1    ActionController::BasicImplicitRender#send_action(method="create", args=[]) at
~/.rbenv/versions/3.0.1/lib/ruby/gems/3.0.0/gems/actionpack-
7.0.0.alpha2/lib/action_controller/metal/basic_implicit_render.rb:6
  # and 72 frames (use `bt' command for all frames)
(rdbg) catch ActiveRecord::RecordInvalid    # command
#1  BP - Catch  "ActiveRecord::RecordInvalid"
```

```
(rdbg) c    # continue command
[75, 84] in ~/.rbenv/versions/3.0.1/lib/ruby/gems/3.0.0/gems/activerecord-
7.0.0.alpha2/lib/active_record/validations.rb
    75|     def default_validation_context
    76|       new_record? ? :create : :update
    77|     end
    78|
    79|     def raise_validation_error
=>  80|       raise(RecordInvalid.new(self))
    81|     end
    82|
    83|     def perform_validations(options = {})
    84|       options[:validate] == false || valid?(options[:context])
=>#0    ActiveRecord::Validations#raise_validation_error at
~/.rbenv/versions/3.0.1/lib/ruby/gems/3.0.0/gems/activerecord-7.0.0.alpha2/lib/active_record/validations.rb:80
  #1    ActiveRecord::Validations#save!(options={}) at
```

```
~/.rbenv/versions/3.0.1/lib/ruby/gems/3.0.0/gems/activerecord-7.0.0.alpha2/lib/active_record/validations.rb:53
  # and 88 frames (use `bt' command for all frames)

Stop by #1  BP - Catch  "ActiveRecord::RecordInvalid"
```

The watch command

Stop when the instance variable is changed - e.g. `watch @_response_body`

```
[20, 29] in ~/projects/rails-guide-example/app/controllers/posts_controller.rb
    20|   end
    21|
    22|   # POST /posts or /posts.json
    23|   def create
    24|     @post = Post.new(post_params)
=>  25|     debugger
    26|
    27|     respond_to do |format|
    28|       if @post.save!
    29|         format.html { redirect_to @post, notice: "Post was successfully created." }
=>#0    PostsController#create at ~/projects/rails-guide-example/app/controllers/posts_controller.rb:25
  #1    ActionController::BasicImplicitRender#send_action(method="create", args=[]) at
~/.rbenv/versions/3.0.1/lib/ruby/gems/3.0.0/gems/actionpack-
7.0.0.alpha2/lib/action_controller/metal/basic_implicit_render.rb:6
  # and 72 frames (use `bt' command for all frames)
(rdbg) watch @_response_body    # command
#0  BP - Watch  #<PostsController:0x00007fce69ca5320> @_response_body =
```

```
(rdbg) c    # continue command
[173, 182] in ~/.rbenv/versions/3.0.1/lib/ruby/gems/3.0.0/gems/actionpack-
7.0.0.alpha2/lib/action_controller/metal.rb
   173|       body = [body] unless body.nil? || body.respond_to?(:each)
   174|       response.reset_body!
   175|       return unless body
   176|       response.body = body
   177|       super
=> 178|     end
   179|
   180|     # Tests if render or redirect has already happened.
   181|     def performed?
   182|       response_body || response.committed?
=>#0    ActionController::Metal#response_body=(body=["<html><body>You are being <a href=\"ht...) at
~/.rbenv/versions/3.0.1/lib/ruby/gems/3.0.0/gems/actionpack-7.0.0.alpha2/lib/action_controller/metal.rb:178
#=> ["<html><body>You are being <a href=\"ht...
  #1    ActionController::Redirecting#redirect_to(options=#<Post id: 13, title: "qweqwe", content:...,
response_options={:allow_other_host=>false}) at ~/.rbenv/versions/3.0.1/lib/ruby/gems/3.0.0/gems/actionpack-
7.0.0.alpha2/lib/action_controller/metal/redirecting.rb:74
  # and 82 frames (use `bt' command for all frames)

Stop by #0  BP - Watch  #<PostsController:0x00007fce69ca5320> @_response_body =  -> ["<html><body>You are
being <a href=\"http://localhost:3000/posts/13\">redirected</a>.</body></html>"]
(rdbg)
```

Breakpoint options

In addition to different types of breakpoints, you can also specify options to achieve more advanced debugging workflow. Currently, the debugger supports 4 options:

- `do: <cmd or expr>` - when the breakpoint is triggered, execute the given command/expression and continue the program:
  - `break Foo#bar do: bt` - when `Foo#bar` is called, print the stack frames
- `pre: <cmd or expr>` - when the breakpoint is triggered, execute the given command/expression before stopping:
  - `break Foo#bar pre: info` - when `Foo#bar` is called, print its surrounding variables before stopping.
- `if: <expr>` - the breakpoint only stops if the result of `<expr` > is true:
  - `break Post#save if: params[:debug]` - stops at `Post#save` if `params[:debug]` is also true

- `path: <path_regexp>` - the breakpoint only stops if the event that triggers it (e.g. a method call) happens from the given path:
    - `break Post#save if: app/services/a_service` - stops at `Post#save` if the method call happens at a method matches Ruby regexp `/app\/services\/a_service/`.

Please also note that the first 3 options: `do:`, `pre:` and `if:` are also available for the debug statements we mentioned earlier. For example:

```
[2, 11] in ~/projects/rails-guide-example/app/controllers/posts_controller.rb
     2|   before_action :set_post, only: %i[ show edit update destroy ]
     3|
     4|   # GET /posts or /posts.json
     5|   def index
     6|     @posts = Post.all
=>   7|     debugger(do: "info")
     8|   end
     9|
    10|   # GET /posts/1 or /posts/1.json
    11|   def show
=>#0    PostsController#index at ~/projects/rails-guide-example/app/controllers/posts_controller.rb:7
  #1    ActionController::BasicImplicitRender#send_action(method="index", args=[]) at
~/.rbenv/versions/3.0.1/lib/ruby/gems/3.0.0/gems/actionpack-
7.0.0.alpha2/lib/action_controller/metal/basic_implicit_render.rb:6
  # and 72 frames (use `bt' command for all frames)
(rdbg:binding.break) info
%self = #<PostsController:0x00000000017480>
@_action_has_layout = true
@_action_name = "index"
@_config = {}
@_lookup_context = #<ActionView::LookupContext:0x00007fce3ad336b8 @details_key=nil, @digest_cache=...
@_request = #<ActionDispatch::Request GET "http://localhost:3000/posts" for 127.0.0.1>
@_response = #<ActionDispatch::Response:0x00007fce3ad397e8 @mon_data=#<Monitor:0x00007fce3ad396a8>...
@_response_body = nil
@_routes = nil
@marked_for_same_origin_verification = true
@posts = #<ActiveRecord::Relation [#<Post id: 2, title: "qweqwe", content: "qweqwe", created_at: "...
@rendered_format = nil
```

Program your debugging workflow

With those options, you can script your debugging workflow in one line like:

```
def create
  debugger(do: "catch ActiveRecord::RecordInvalid do: bt 10")
  # ...
end
```

And then the debugger will run the scripted command and insert the catch breakpoint

```
(rdbg:binding.break) catch ActiveRecord::RecordInvalid do: bt 10
#0  BP - Catch  "ActiveRecord::RecordInvalid"
[75, 84] in ~/.rbenv/versions/3.0.1/lib/ruby/gems/3.0.0/gems/activerecord-
7.0.0.alpha2/lib/active_record/validations.rb
    75|     def default_validation_context
    76|       new_record? ? :create : :update
    77|     end
    78|
    79|     def raise_validation_error
=>  80|       raise(RecordInvalid.new(self))
    81|     end
    82|
    83|     def perform_validations(options = {})
    84|       options[:validate] == false || valid?(options[:context])
=>#0    ActiveRecord::Validations#raise_validation_error at
~/.rbenv/versions/3.0.1/lib/ruby/gems/3.0.0/gems/activerecord-7.0.0.alpha2/lib/active_record/validations.rb:80
  #1    ActiveRecord::Validations#save!(options={}) at
~/.rbenv/versions/3.0.1/lib/ruby/gems/3.0.0/gems/activerecord-7.0.0.alpha2/lib/active_record/validations.rb:53
  # and 88 frames (use `bt' command for all frames)
```

Once the catch breakpoint is triggered, it'll print the stack frames

```
Stop by #0  BP - Catch  "ActiveRecord::RecordInvalid"

(rdbg:catch) bt 10
=>#0    ActiveRecord::Validations#raise_validation_error at
~/.rbenv/versions/3.0.1/lib/ruby/gems/3.0.0/gems/activerecord-7.0.0.alpha2/lib/active_record/validations.rb:80
  #1    ActiveRecord::Validations#save!(options={}) at
~/.rbenv/versions/3.0.1/lib/ruby/gems/3.0.0/gems/activerecord-7.0.0.alpha2/lib/active_record/validations.rb:53
  #2    block in save! at ~/.rbenv/versions/3.0.1/lib/ruby/gems/3.0.0/gems/activerecord-
7.0.0.alpha2/lib/active_record/transactions.rb:302
```

This technique can save you from repeated manual input and make the debugging experience smoother.

You can find more commands and configuration options from its documentation.

Autoloading Caveat

Debugging with `debug` works fine most of the time, but there's an edge case: If you evaluate an expression in the console that autoloads a namespace defined in a file, constants in that namespace won't be found.

For example, if the application has these two files:

```ruby
# hotel.rb
class Hotel
end

# hotel/pricing.rb
module Hotel::Pricing
end
```

and `Hotel` is not yet loaded, then

```
(rdbg) p Hotel::Pricing
```

will raise a `NameError`. In some cases, Ruby will be able to resolve an unintended constant in a different scope.

If you hit this, please restart your debugging session with eager loading enabled (`config.eager_load = true`).

Stepping commands line `next` , `continue` , etc., do not present this issue. Namespaces defined implicitly only by subdirectories are not subject to this issue either.

See ruby/debug#408 for details.

Debugging with the `web-console` gem

Web Console is a bit like `debug` , but it runs in the browser. In any page you are developing, you can request a console in the context of a view or a controller. The console would be rendered next to your HTML content.

Console

Inside any controller action or view, you can invoke the console by calling the `console` method.

For example, in a controller:

```ruby
class PostsController < ApplicationController
  def new
    console
    @post = Post.new
  end
end
```

Or in a view:

```erb
<% console %>
```

```
<h2>New Post</h2>
```

This will render a console inside your view. You don't need to care about the location of the `console` call; it won't be rendered on the spot of its invocation but next to your HTML content.

The console executes pure Ruby code: You can define and instantiate custom classes, create new models, and inspect variables.

NOTE: Only one console can be rendered per request. Otherwise `web-console` will raise an error on the second `console` invocation.

Inspecting Variables

You can invoke `instance_variables` to list all the instance variables available in your context. If you want to list all the local variables, you can do that with `local_variables`.

Settings

- `config.web_console.allowed_ips`: Authorized list of IPv4 or IPv6 addresses and networks (defaults: `127.0.0.1/8, ::1`).
- `config.web_console.whiny_requests`: Log a message when a console rendering is prevented (defaults: `true`).

Since `web-console` evaluates plain Ruby code remotely on the server, don't try to use it in production.

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 tools such as Valgrind.

Valgrind

Valgrind is an 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.

Find a Memory Leak

There is an excellent article about detecting and fixing memory leaks at Derailed, which you can read here.

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:

- Query Trace Adds query origin tracing to your logs.
- 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.
- Pry An IRB alternative and runtime developer console.

References

- web-console Homepage
- debug homepage