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.**

Chapters

1 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`

1.1 `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>
<%= 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
--- !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`

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

Will render:

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

Title: Rails debugging guide
[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. 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")
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")
Rails.logger = Logger.new(STDOUT)
Rails.logger = Log4r::Logger.new("Application Log")
```

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 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
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.

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!!!"
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
    end
  end

  # ...

  private
    def article_params
      params.require(:article).permit(:title, :body, :published)
    end
end
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
    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 2017-08-20 20:53:10 +0900
Processing by ArticlesController#create as HTML
```

```
  Parameters: {"utf8"=>"✓",
"authenticity_token"=>"xhuIbSBFytHCE1agHgvrlKnSVIOGD6jltW2tO+P6a/ACjQ3igj
"article"=>{"title"=>"Debugging Rails", "body"=>"I'm learning how to prin
"commit"=>"Create Article"}
New article: {"id"=>nil, "title"=>"Debugging Rails", "body"=>"I'm learnin
"published"=>false, "created_at"=>nil, "updated_at"=>nil}
Article should be valid: true
   (0.1ms)  BEGIN
  SQL (0.4ms)  INSERT INTO "articles" ("title", "body", "published", "cre
$3, $4, $5) RETURNING "id"  [["title", "Debugging Rails"], ["body", "I'm
["published", "f"], ["created_at", "2017-08-20 11:53:10.010435"], ["updat
   (0.3ms)  COMMIT
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)
Started POST "/articles" for 127.0.0.1 at 2017-08-20 20:53:10 +0900
Processing by ArticlesController#create as HTML
  Parameters: {"utf8"=>"✓",
"authenticity_token"=>"xhuIbSBFytHCE1agHgvrlKnSVIOGD6jltW2tO+P6a/ACjQ3igj
"article"=>{"title"=>"Debugging Rails", "body"=>"I'm learning how to prin
"commit"=>"Create Article"}
New article: {"id"=>nil, "title"=>"Debugging Rails", "body"=>"I'm learnin
"published"=>false, "created_at"=>nil, "updated_at"=>nil}
Article should be valid: true
   (0.1ms)  BEGIN
  SQL (0.4ms)  INSERT INTO "articles" ("title", "body", "published", "cre
$3, $4, $5) RETURNING "id"  [["title", "Debugging Rails"], ["body", "I'm
["published", "f"], ["created_at", "2017-08-20 11:53:10.010435"], ["updat
   (0.3ms)  COMMIT
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.

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 you do exactly that by stamping log lines with subdomains, request ids, and anything else to aid debugging such applications.

```ruby
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"
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"
```

2.5 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}"
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}"}
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.

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 into the underlying 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
$ 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
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)
[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 Puma
=> Rails 5.1.0 application starting in development on
http://0.0.0.0:3000
=> Run `rails server -h` for more startup options
Puma starting in single mode...
* Version 3.4.0 (ruby 2.3.1-p112), codename: Owl Bowl Brawl
* Min threads: 5, max threads: 5
* Environment: development
* Listening on tcp://localhost:3000
Use Ctrl-C to stop
Started GET "/" for 127.0.0.1 at 2014-04-11 13:11:48 +0200
  ActiveRecord::SchemaMigration Load (0.2ms)  SELECT
"schema_migrations".* FROM "schema_migrations"
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)
=> Booting Puma
=> Rails 5.1.0 application starting in development on
http://0.0.0.0:3000
=> Run `rails server -h` for more startup options
Puma starting in single mode...
* Version 3.4.0 (ruby 2.3.1-p112), codename: Owl Bowl Brawl
* Min threads: 5, max threads: 5
* Environment: development
* Listening on tcp://localhost:3000
Use Ctrl-C to stop
Started GET "/" for 127.0.0.1 at 2014-04-11 13:11:48 +0200
  ActiveRecord::SchemaMigration Load (0.2ms)  SELECT
"schema_migrations".* FROM "schema_migrations"
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 your application. A good place to start is by asking the debugger for help. Type:
`help`

```
(byebug) help

  break      -- Sets breakpoints in the source code
  catch      -- Handles exception catchpoints
  condition  -- Sets conditions on breakpoints
  continue   -- Runs until program ends, hits a breakpoint or reaches a
line
  debug      -- Spawns a subdebugger
  delete     -- Deletes breakpoints
  disable    -- Disables breakpoints or displays
  display    -- Evaluates expressions every time the debugger stops
  down       -- Moves to a lower frame in the stack trace
  edit       -- Edits source files
  enable     -- Enables breakpoints or displays
  finish     -- Runs the program until frame returns
  frame      -- Moves to a frame in the call stack
  help       -- Helps you using byebug
  history    -- Shows byebug's history of commands
  info       -- Shows several informations about the program being
debugged
  interrupt  -- Interrupts the program
  irb        -- Starts an IRB session
  kill       -- Sends a signal to the current process
  list       -- Lists lines of source code
  method     -- Shows methods of an object, class or module
  next       -- Runs one or more lines of code
  pry        -- Starts a Pry session
  quit       -- Exits byebug
  restart    -- Restarts the debugged program
  save       -- Saves current byebug session to a file
  set        -- Modifies byebug settings
  show       -- Shows byebug settings
  source     -- Restores a previously saved byebug session
  step       -- Steps into blocks or methods one or more times
  thread     -- Commands to manipulate threads
  tracevar   -- Enables tracing of a global variable
  undisplay  -- Stops displaying all or some expressions when program
stops
  untracevar -- Stops tracing a global variable
  up         -- Moves to a higher frame in the stack trace
  var        -- Shows variables and its values
  where      -- Displays the backtrace

(byebug)
(byebug) help

  break      -- Sets breakpoints in the source code
  catch      -- Handles exception catchpoints
  condition  -- Sets conditions on breakpoints
  continue   -- Runs until program ends, hits a breakpoint or reaches a
```

```
line
  debug      -- Spawns a subdebugger
  delete     -- Deletes breakpoints
  disable    -- Disables breakpoints or displays
  display    -- Evaluates expressions every time the debugger stops
  down       -- Moves to a lower frame in the stack trace
  edit       -- Edits source files
  enable     -- Enables breakpoints or displays
  finish     -- Runs the program until frame returns
  frame      -- Moves to a frame in the call stack
  help       -- Helps you using byebug
  history    -- Shows byebug's history of commands
  info       -- Shows several informations about the program being
debugged
  interrupt  -- Interrupts the program
  irb        -- Starts an IRB session
  kill       -- Sends a signal to the current process
  list       -- Lists lines of source code
  method     -- Shows methods of an object, class or module
  next       -- Runs one or more lines of code
  pry        -- Starts a Pry session
  quit       -- Exits byebug
  restart    -- Restarts the debugged program
  save       -- Saves current byebug session to a file
  set        -- Modifies byebug settings
  show       -- Shows byebug settings
  source     -- Restores a previously saved byebug session
  step       -- Steps into blocks or methods one or more times
  thread     -- Commands to manipulate threads
  tracevar   -- Enables tracing of a global variable
  undisplay  -- Stops displaying all or some expressions when program
stops
  untracevar -- Stops tracing a global variable
  up         -- Moves to a higher frame in the stack trace
  var        -- Shows variables and its values
  where      -- Displays the backtrace

(byebug)
```

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|
(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 and see the code above 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)
(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 know 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 /PathToProject/app/controllers/articles_controller.rb:8
    #1
ActionController::BasicImplicitRender.send_action(method#String,
*args#Array)
      at /PathToGems/actionpack-
5.1.0/lib/action_controller/metal/basic_implicit_render.rb:4
    #2  AbstractController::Base.process_action(action#NilClass,
*args#Array)
      at /PathToGems/actionpack-
5.1.0/lib/abstract_controller/base.rb:181
    #3  ActionController::Rendering.process_action(action, *args)
      at /PathToGems/actionpack-
5.1.0/lib/action_controller/metal/rendering.rb:30
...
(byebug) where
--> #0  ArticlesController.index
      at /PathToProject/app/controllers/articles_controller.rb:8
    #1
ActionController::BasicImplicitRender.send_action(method#String,
*args#Array)
      at /PathToGems/actionpack-
5.1.0/lib/action_controller/metal/basic_implicit_render.rb:4
    #2  AbstractController::Base.process_action(action#NilClass,
*args#Array)
      at /PathToGems/actionpack-
5.1.0/lib/abstract_controller/base.rb:181
    #3  ActionController::Rendering.process_action(action, *args)
      at /PathToGems/actionpack-
5.1.0/lib/action_controller/metal/rendering.rb:30
...
```

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 *n* is the specified frame number. If you do that, `byebug` will display your new context.

```
(byebug) frame 2

[176, 185] in /PathToGems/actionpack-
5.1.0/lib/abstract_controller/base.rb
   176:       # is the intended way to override action dispatching.
   177:       #
   178:       # Notice that the first argument is the method to be
dispatched
   179:       # which is *not* necessarily the same as the action name.
   180:       def process_action(method_name, *args)
```

```
=> 181:         send_action(method_name, *args)
   182:       end
   183:
   184:       # Actually call the method associated with the action.
Override
   185:       # this method if you wish to change how action methods
are called,
(byebug)
(byebug) frame 2

[176, 185] in /PathToGems/actionpack-
5.1.0/lib/abstract_controller/base.rb
   176:       # is the intended way to override action dispatching.
   177:       #
   178:       # Notice that the first argument is the method to be
dispatched
   179:       # which is *not* necessarily the same as the action name.
   180:       def process_action(method_name, *args)
=> 181:         send_action(method_name, *args)
   182:       end
   183:
   184:       # Actually call the method associated with the action.
Override
   185:       # 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]` and `down [n]` commands in order to change the context *n* frames up or down the stack respectively. *n* 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 current thread is marked with a plus (+) sign.
- `thread stop n`: stops thread *n*.
- `thread resume n`: resumes thread *n*.
- `thread switch n`: switches the current thread context to *n*.

This command is very helpful 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, :@_request, :@_response,
:@_lookup_context,
 :@_action_name, :@_response_body,
:@marked_for_same_origin_verification,
 :@_config]
[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, :@_request, :@_response,
:@_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 }

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

```
   13      end
   14    end
   15
(byebug)
(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
[:@_action_has_layout, :@_routes, :@_request, :@_response,
:@_lookup_context,
 :@_action_name, :@_response_body,
:@marked_for_same_origin_verification,
 :@_config, :@articles]
(byebug) instance_variables
[:@_action_has_layout, :@_routes, :@_request, :@_response,
:@_lookup_context,
 :@_action_name, :@_response_body,
:@marked_for_same_origin_verification,
 :@_config, :@articles]
```

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

You can also step into **irb** mode with the command `irb` (of course!). This will start an irb session within the context you invoked it.

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

```
(byebug) help var

  [v]ar <subcommand>

  Shows variables and its values


  var all      -- Shows local, global and instance variables of self.
  var args     -- Information about arguments of the current scope
```

```
  var const    -- Shows constants of an object.
  var global   -- Shows global variables.
  var instance -- Shows instance variables of self or a specific
object.
  var local    -- Shows local variables in current scope.
(byebug) help var

  [v]ar <subcommand>

  Shows variables and its values


  var all      -- Shows local, global and instance variables of self.
  var args     -- Information about arguments of the current scope
  var const    -- Shows constants of an object.
  var global   -- Shows global variables.
  var instance -- Shows instance variables of self or a specific
object.
  var local    -- Shows local variables in current scope.
```

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)
(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 = #<ActiveRecord::AttributeSet:0x007fd0682a9b18
@attributes={"id"=>#
<ActiveRecord::Attribute::FromDatabase:0x007fd0682a9a00 @name="id",
@value_be...
@destroyed = false
@destroyed_by_association = nil
@marked_for_destruction = false
@new_record = true
@readonly = false
@transaction_state = nil
(byebug) var instance Article.new
@_start_transaction_state = {}
@aggregation_cache = {}
@association_cache = {}
@attributes = #<ActiveRecord::AttributeSet:0x007fd0682a9b18
@attributes={"id"=>#
<ActiveRecord::Attribute::FromDatabase:0x007fd0682a9a00 @name="id",
@value_be...
@destroyed = false
```

```
@destroyed_by_association = nil
@marked_for_destruction = false
@new_record = true
@readonly = false
@transaction_state = nil
```

You can also use `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
(byebug) display @articles
1: @articles = nil
```

The variables inside the displayed list will be printed with their values after you move in the stack. To stop displaying a variable use `undisplay n` where *n* 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 let's 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. `next` is similar to `step`, but while `step` stops at the next line of code executed, doing just a single step, `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, 6] in /PathToProject/app/models/article.rb
   1: class Article < ApplicationRecord
   2:   def self.find_recent(limit = 10)
   3:     byebug
=> 4:     where('created_at > ?', 1.week.ago).limit(limit)
   5:   end
   6: end

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

[1, 6] in /PathToProject/app/models/article.rb
   1: class Article < ApplicationRecord
   2:   def self.find_recent(limit = 10)
   3:     byebug
=> 4:     where('created_at > ?', 1.week.ago).limit(limit)
   5:   end
   6: end
```

```
(byebug)
```

If we use `next`, we won't go deep inside method calls. Instead, `byebug` will go to the next line within the same context. In this case, it is the last line of the current method, so `byebug` will return to the next line of the caller method.

```
(byebug) next
[4, 13] in /PathToProject/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)
(byebug) next
[4, 13] in /PathToProject/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, `byebug` will literally go to the next Ruby instruction to be executed -- in this case, Active Support's `week` method.

```
(byebug) step

[49, 58] in /PathToGems/activesupport-
5.1.0/lib/active_support/core_ext/numeric/time.rb
   49:
   50:   # Returns a Duration instance matching the number of weeks
provided.
   51:   #
   52:   #   2.weeks # => 14 days
   53:   def weeks
=> 54:     ActiveSupport::Duration.weeks(self)
   55:   end
   56:   alias :week :weeks
   57:
   58:   # Returns a Duration instance matching the number of
```

```
fortnights provided.
(byebug)
(byebug) step

[49, 58] in /PathToGems/activesupport-
5.1.0/lib/active_support/core_ext/numeric/time.rb
   49:
   50:   # Returns a Duration instance matching the number of weeks
provided.
   51:   #
   52:   #   2.weeks # => 14 days
   53:   def weeks
=> 54:     ActiveSupport::Duration.weeks(self)
   55:   end
   56:   alias :week :weeks
   57:
   58:   # Returns a Duration instance matching the number of
fortnights provided.
(byebug)
```

This is one of the best ways to find bugs in your code.

You can also use `step n` or `next n` to move forward `n` steps at once.

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

For example, in the previous situation

```
[4, 13] in /PathToProject/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
```

```
Successfully created breakpoint with id 1
[4, 13] in /PathToProject/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
Successfully created breakpoint with id 1
```

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

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

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

You can also enable or disable breakpoints:

- `enable breakpoints [n [m [...]]]`: allows a specific breakpoint list or all breakpoints to stop your program. This is the default state when you create a breakpoint.
- `disable breakpoints [n [m [...]]]`: make certain (or all) breakpoints 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 *exception-name* 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 [n]`: resumes program execution at the address where your script last stopped; any breakpoints set at that address are bypassed. The optional argument `n` allows you to specify a line number to set a one-time breakpoint which is deleted when that breakpoint is reached.
- `finish [n]`: 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:n]`: edit file named *file* using the editor specified by the EDITOR environment variable. A specific line *n* can also be given.

3.11 Quitting

To exit the debugger, use the `quit` command (abbreviated to `q`). Or, type `q!` to bypass the `Really quit? (y/n)` prompt and exit unconditionally.

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 behavior:

```
(byebug) help set

  set <setting> <value>

  Modifies byebug settings

  Boolean values take "on", "off", "true", "false", "1" or "0". If you
  don't specify a value, the boolean setting will be enabled.
Conversely,
  you can use "set no<setting>" to disable them.

  You can see these environment settings with the "show" command.

  List of supported settings:

  autosave       -- Automatically save command history record on exit
  autolist       -- Invoke list command on every stop
  width          -- Number of characters per line in byebug's output
  autoirb        -- Invoke IRB on every stop
  basename       -- <file>:<line> information after every stop uses
short paths
  linetrace      -- Enable line execution tracing
  autopry        -- Invoke Pry on every stop
```

```
  stack_on_error -- Display stack trace when `eval` raises an exception
  fullpath       -- Display full file names in backtraces
  histfile       -- File where cmd history is saved to. Default:
./.byebug_history
  listsize       -- Set number of source lines to list by default
  post_mortem    -- Enable/disable post-mortem mode
  callstyle      -- Set how you want method call parameters to be
displayed
  histsize       -- Maximum number of commands that can be stored in
byebug history
  savefile       -- File where settings are saved to. Default:
~/.byebug_save
(byebug) help set

  set <setting> <value>

  Modifies byebug settings

  Boolean values take "on", "off", "true", "false", "1" or "0". If you
  don't specify a value, the boolean setting will be enabled.
Conversely,
  you can use "set no<setting>" to disable them.

  You can see these environment settings with the "show" command.

  List of supported settings:

  autosave       -- Automatically save command history record on exit
  autolist       -- Invoke list command on every stop
  width          -- Number of characters per line in byebug's output
  autoirb        -- Invoke IRB on every stop
  basename       -- <file>:<line> information after every stop uses
short paths
  linetrace      -- Enable line execution tracing
  autopry        -- Invoke Pry on every stop
  stack_on_error -- Display stack trace when `eval` raises an exception
  fullpath       -- Display full file names in backtraces
  histfile       -- File where cmd history is saved to. Default:
./.byebug_history
  listsize       -- Set number of source lines to list by default
  post_mortem    -- Enable/disable post-mortem mode
  callstyle      -- Set how you want method call parameters to be
displayed
  histsize       -- Maximum number of commands that can be stored in
byebug history
  savefile       -- File where settings are saved to. Default:
~/.byebug_save
```

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 callstyle short
set listsize 25
set callstyle short
set listsize 25
```

4 Debugging with the `web-console` gem

Web Console is a bit like `byebug`, 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.

4.1 Console

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

For example, in a controller:

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

Or in a view:

```
<% console %>

<h2>New Post</h2>
<% 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.

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

4.2 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`.

4.3 Settings

- `config.web_console.whitelisted_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.

5 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.

5.1 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.

6 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.
- **Pry** An IRB alternative and runtime developer console.

7 References

- **byebug Homepage**
- **web-console Homepage**

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 on the **rubyonrails-docs mailing list**.