PAWEŁ DĄBROWSKI

EFFECTIVE DEVELOPMENT WITH RAILS CONSOLE

Table of contents

Introduction

Console in Rails is one of those things you can't deal without if you want to build and maintain a web application effectively. Whether you access the development or production environment, it allows you to play with the application's code without accessing the server.

This book is a set of tips and tricks that you can perform in the Rails console to make your development process even more efficient and fun than before.

The code presented in this book was tested against Ruby 3.1.0 and Rails 7.0.1.

For more books visit https://longliveruby.com

Inspecting methods

You do not have to open text editor to inspect given class methods. You can do it via the console.

Finding where the given method was defined

The class can have two types of methods: instance and singleton. If you know the name of the method and the name of the class that can be a caller, you can easily find in which file it's defined and in which line.

To inspect the instance method, use the following code:

```
User.instance_method(:name).source_location
# => ["/app/models/user.rb", 2]
```

As a result, you will receive two elements array where the first element is the path to the file and the second element is the line number where the definition starts.

For inspecting the class method, you can use singleton_method instead of the instance_method, and the output will be in the same format:

```
User.singleton_method(:human?).source_location
# => ["/app/models/user.rb", 8]
```

Rendering the source of method

You don't have to open a text editor to view the code of the given method. It's a piece of good news, especially when it comes to methods defined somewhere in ruby gems.

In terms of the instance methods, grab the class name and invoke the instance_method method on it by passing the method name and then calling source on the result:

```
User.instance_method(:name).source
# => "  def name\n    \"\#{first_name} \#{last_name}\"\n  end\n"
```

The output is not well-formatted, but you can simply fix it by calling the display method on the result:

```
User.instance_method(:name).source.display
#   def name
#    "#{first_name} #{last_name}"
#   end
# => nil
```

In terms of the class method, replace instance_method with singleton_method:

```
User.singleton_method(:human?).source.display
#   def self.human?
#     true

#   end
# => nil
```

Searching for methods

If you don't know the exact name of the method, you can filter instance and class methods against the regular expression and receive an array of method names matching passed criteria.

For instance methods use the following code:

```
User.instance_methods.grep(/name$/)
# => [:name, :model_name, :store_full_class_name]
```

The expression we used above searches for all methods ends with name. The grep method returns an array with matching elements.

For class methods, use singleton_methods instead:

```ruby
User.singleton_methods.grep(/table_name$/)
# => [:schema_migrations_table_name, :internal_metadata_table_name]
```

Playing with Active Record

The Rails console is a perfect place to modify the records in the database. Few tricks can make this work even more effective and safe.

Sandbox mode

You should never run this mode in production. The sandbox mode allows you to run the console in a special mode that will rollback all changes you made to the database when you exit from the console.

While it sounds super helpful, the sandbox mode wraps all your queries into one big transaction, and it can cause locks on data for other users using the database. That's why I advise you not to use it on the production database.

You can enter into sandbox mode by passing --sandbox param:

```
rails c -sandbox
```

You can now perform any action you want, but all changes will automatically roll back when you type exit in the console.

Execute any SQL query

You can query the database using proper ActiveRecord models, but you can also quickly execute any SQL query against your database. To do this, call the execute method on the current database connection:

```
sql = "SELECT * FROM users"
ActiveRecord::Base.connection.execute(sql)
```

The result format might differ depending on the database engine you are using. For example, if you use PostgreSQL, PG::Result will be returned; if MySQL, then Mysql2::Result.

Reload current record

If you executed the Active Record query and assigned it to a variable:

```
result = User.where(first_name: "John")
```

and you want to execute it again; you don't have to write the code again. You can simply call the reload method on the ActiveRecord result, and the query will be executed again:

```
result = User.where(first_name: "John")
result.reload
```

You can do this when pulling a single record or pulling a collection. The reload method works as long as a result comes from Active Record.

Inspecting tables in the database

If you want to get the array with table names at your disposal, you can simply call the following line:

```
ActiveRecord::Base.connection.tables
```

On the other side, if you are interested in columns for a particular table, you can use another method that comes with the current connection object:

```
ActiveRecord::Base.connection.columns("users")
```

The above call returns an array of objects specific to the given database engine. Each object represents one column and provides information about its type, name, and other attributes.

Playing with Rails environment

With the Rails console, you don't have to access the browser to perform requests, render views to test helpers, or exit to the command line to perform rake tasks.

Inspect helper methods

By default, the helper object is available in the console. On this object, you can call any helper method you defined in files placed inside app/helpers directory:

```
helper.full_name("John", "Doe")
```

All methods you can use inside views are also available via the helper object. For example, sanitize:

```
helper.sanitize("<h1>header</h1><strong>text</strong>", tags: [
"h1"])
# => "<h1>header</h1>text"
```

Inspect paths

By default, the app object is available in the console. This object represents the whole application. If you can execute on it all the paths that you have defined in routes:

```
app.articles_path(id: 1)
# => "/articles/1"
```

If that's not enough, you can perform the request:

```
app.host = "localhost"
app.get(app.articles_path(id: 1))
```

When the request is completed, you can inspect app.response to see the response body and status:

```
app.response.status
# => 200

app.response.body
# => "..."
```

If you don't set the host to localhost, your request will be blocked due to the not authorized host, which is set to www.example.com by default.

Inspecting rake tasks

Let's say that we have the following rake task defined:

```
namespace :users do
 desc 'Truncate all users'
 task truncate: :environment do
   puts 'invoked!'
 end
end
```

In the console, we can invoke this rake task. The first step is to load all rake tasks:

```
Rails.application.load_tasks
```

and then we can invoke it in the following way:

```
Rake::Task['users:truncate'].invoke
```

If your task requires any arguments, simply pass them in the correct order to the invoke method.

Reflecting changes in the code

If you have the console open and have made some changes to your code, you don't have to relaunch the console. All you need to do is to call reload!:

```
helper.some_method
# => undefined method
reload!
# => Reloading …
# => true
helper.some_method
# => true
```

Making work with console more comfortable

There are a few simple ways you can make the experience of working with a console even more pleasant.

Accessing the last returned value

Many times you have to use the latest printed value in the console. You can either execute the same code again or use simple helper:

```
3.1.0 :001 > 2 + 2
 => 4
3.1.0 :002 > _
 => 4
```

The floor character would give you the latest value printed in the console. After that, you can do whatever you want - assign to a variable, for example, or pass directly to another method.

Silence SQL logs when executing queries

If you need to update many records in the console, it can produce tons of Rails logs that will flood the whole console:

```
User.find_each { |u| u.touch }
  TRANSACTION (0.1ms)  COMMIT
  TRANSACTION (0.1ms)  BEGIN
  User Update (0.2ms)  UPDATE "users" SET "updated_at" = $1 WHERE
"users"."id" = $2  [["updated_at", "2022-01-28 13:13:07.883801"],
["id", 1002]]
  TRANSACTION (0.1ms)  COMMIT
...
```

Rails provides simple logs wrapper that can silence all logs:

```
ActiveRecord::Base.logger.silence do
  User.find_each { |u| u.touch }
end
```

With the above version, no longs will be printed inside the console.

Switch the current context

If you are working with some class or object and you have often to repeat its name, for example the User object:

```
user.first_name
# => "John"
user.do_something
# => false
user.do_something_else
# => true
```

you can change the console context to this object (or class) and then call the methods or attributes directly:

```
irb user
first_name
# => "John"
do_something
# => false
do_something_else
# => true
```

You can quickly exit the current context by writing exit.

Inspect any class or object

You can get an excellent output containing variables, methods, and other information about a given object or class by using the ls function and passing this object (or class):

```
user = User.new
ls user
User#methods: name
instance variables: @association_cache @attributes @destroyed
@destroyed_by_association
class variables: @@configurations
# many more…
```

You can already be familiar with the ls command from Linux or Mac OS. This command lists the files and directories in the current directory when executed in the command line.

Output object in the YAML format

You can debug a given object in a YAML format by passing it to the y method provided by the Kernel module:

```
names = ["Tim", "John", "Tom"]
y names
---
- Tim
- John
- Tom
=> nil
```

The method is handy when debugging hashes with many pairs as well.

Temporary alter any method of the object

Sometimes, you have to execute some updated methods in the production environment, but you don't want to commit changes and make the deployment.

Ruby is so flexible that you can update the method only for the given instance and run it as a typical method of the given object in its context.

For example, let's assume that we have the following method in the User model:

```
class User < ApplicationRecord
  def trigger_update_process
    return if some_complex_policy_call

    # perform some other actions
  end
end
```

We want to skip the some_complex_policy_call for just one user instance in the console and perform other actions. You can either manually trigger all actions or temporary edit the trigger_update_process method:

```
user = User.find(...)

def user.trigger_update_process
  # perform some other actions
end

user.trigger_update_process
```

Now, the function trigger_update_process will run without the return if some_complex_policy_call line just for this one instance. The change will be temporary, and it will go when you exit the console or load the object again.

Make your own methods available in the console by default

In one of the projects, we used to use one user in the console to trigger some processes and pass him as the person to whom the result is reported. It looked like that:

```
user = User.find(17623)
SomeBackgroundJob.perform_async(user.id)
```

You have to remember the exact id, and you can run into trouble if you use any other user. To minimize the risk and make the usage of the console more pleasant, you can provide this object under a meaningful method. To do this, you have to update config/application.rb and make the following change:

```
module MyProject
  class Application < Rails::Application
    console do
     module MyProjectConsole
       def report_user
         User.find(17623)
       end
     end

     Rails::ConsoleMethods.include(MyProjectConsole)
   end
  end
end
```

Next time you will load the console, you can just simply use report_user:

```
SomeBackgroundJob.perform_async(report_user.id)
```