The Rails Initialization Process

January 13, 2015

This guide explains the internals of the initialization process in Rails as of Rails 4. It is an extremely in-depth guide and recommended for advanced Rails developers.

After reading this guide, you will know:

- How to use `rails server`.
- The timeline of Rails' initialization sequence.
- Where different files are required by the boot sequence.
- How the Rails::Server interface is defined and used.

This guide goes through every method call that is required to boot up the Ruby on Rails stack for a default Rails 4 application, explaining each part in detail along the way. For this guide, we will be focusing on what happens when you execute `rails server` to boot your app.

Paths in this guide are relative to Rails or a Rails application unless otherwise specified.

If you want to follow along while browsing the Rails source code, we recommend that you use the `t` key binding to open the file finder inside GitHub and find files quickly.

1 Launch!

Let's start to boot and initialize the app. A Rails application is usually started by running `rails console` or `rails server`.

1.1 railties/bin/rails

The `rails` in the command `rails server` is a ruby executable in your load path. This executable contains the following lines:

```
version = ">= 0"
load Gem.bin_path('railties', 'rails', version)
```

If you try out this command in a Rails console, you would see that this loads `railties/bin/rails`. A part of the file `railties/bin/rails.rb` has the following code:

```
require "rails/cli"
```

The file `railties/lib/rails/cli` in turn calls `Rails::AppRailsLoader.exec_app_rails`.

1.2 railties/lib/rails/app_rails_loader.rb

The primary goal of the function `exec_app_rails` is to execute your app's `bin/rails`. If the current directory does not have a `bin/rails`, it will navigate upwards until it finds a `bin/rails` executable. Thus one can invoke a `rails` command from anywhere inside a rails application.

For `rails server` the equivalent of the following command is executed:

```
$ exec ruby bin/rails server
```

1.3 bin/rails

This file is as follows:

```
#!/usr/bin/env ruby
APP_PATH = File.expand_path('../../config/application', __FILE__)
require_relative '../config/boot'
require 'rails/commands'
```

The `APP_PATH` constant will be used later in `rails/commands`. The `config/boot` file referenced here is the `config/boot.rb` file in our application which is responsible for loading Bundler and setting it up.

1.4 config/boot.rb

`config/boot.rb` contains:

```
# Set up gems listed in the Gemfile.
ENV['BUNDLE_GEMFILE'] ||= File.expand_path('../../Gemfile', __FILE__)

require 'bundler/setup' if File.exist?(ENV['BUNDLE_GEMFILE'])
```

In a standard Rails application, there's a `Gemfile` which declares all dependencies of the application. `config/boot.rb` sets `ENV['BUNDLE_GEMFILE']` to the location of this file. If the Gemfile exists, then `bundler/setup` is required. The require is used by Bundler to configure the load path for your Gemfile's dependencies.

A standard Rails application depends on several gems, specifically:

- actionmailer
- actionpack
- actionview
- activemodel
- activerecord
- activesupport
- arel
- builder
- bundler
- erubis
- i18n
- mail

- mime-types
- rack
- rack-cache
- rack-mount
- rack-test
- rails
- railties
- rake
- sqlite3
- thor
- tzinfo

1.5 rails/commands.rb

Once `config/boot.rb` has finished, the next file that is required is `rails/commands`, which helps in expanding aliases. In the current case, the `ARGV` array simply contains `server` which will be passed over:

```
ARGV << '--help' if ARGV.empty?

aliases = {
  "g"  => "generate",
  "d"  => "destroy",
  "c"  => "console",
  "s"  => "server",
  "db" => "dbconsole",
  "r"  => "runner"
}

command = ARGV.shift
command = aliases[command] || command

require 'rails/commands/commands_tasks'

Rails::CommandsTasks.new(ARGV).run_command!(command)
```

As you can see, an empty ARGV list will make Rails show the help snippet.

If we had used `s` rather than `server`, Rails would have used the `aliases` defined here to find the matching command.

1.6 rails/commands/command_tasks.rb

When one types an incorrect rails command, the `run_command` is responsible for throwing an error message. If the command is valid, a method of the same name is called.

```
COMMAND_WHITELIST = %(plugin generate destroy console server dbconsole application runner
new version help)
```

```
def run_command!(command)
  command = parse_command(command)
  if COMMAND_WHITELIST.include?(command)
    send(command)
  else
    write_error_message(command)
  end
end
```

With the `server` command, Rails will further run the following code:

```
def set_application_directory!
  Dir.chdir(File.expand_path('../../', APP_PATH)) unless
File.exist?(File.expand_path("config.ru"))
end

def server
  set_application_directory!
  require_command!("server")

  Rails::Server.new.tap do |server|
    # We need to require application after the server sets environment,
    # otherwise the --environment option given to the server won't propagate.
    require APP_PATH
    Dir.chdir(Rails.application.root)
    server.start
  end
end

def require_command!(command)
  require "rails/commands/#{command}"
end
```

This file will change into the Rails root directory (a path two directories up from `APP_PATH` which points at `config/application.rb`), but only if the `config.ru` file isn't found. This then requires `rails/commands/server` which sets up the `Rails::Server` class.

```
require 'fileutils'
require 'optparse'
require 'action_dispatch'
require 'rails'

module Rails
  class Server < ::Rack::Server
```

`fileutils` and `optparse` are standard Ruby libraries which provide helper functions for working with files and parsing options.

1.7 actionpack/lib/action_dispatch.rb

Action Dispatch is the routing component of the Rails framework. It adds functionality like routing, session, and common middlewares.

1.8 rails/commands/server.rb

The `Rails::Server` class is defined in this file by inheriting from `Rack::Server`. When `Rails::Server.new` is called, this calls the `initialize` method in `rails/commands/server.rb`:

```
def initialize(*)
  super
  set_environment
end
```

Firstly, `super` is called which calls the `initialize` method on `Rack::Server`.

1.9 Rack: lib/rack/server.rb

`Rack::Server` is responsible for providing a common server interface for all Rack-based applications, which Rails is now a part of.

The `initialize` method in `Rack::Server` simply sets a couple of variables:

```
def initialize(options = nil)
  @options = options
  @app = options[:app] if options && options[:app]
end
```

In this case, `options` will be `nil` so nothing happens in this method.

After `super` has finished in `Rack::Server`, we jump back to `rails/commands/server.rb`. At this point, `set_environment` is called within the context of the `Rails::Server` object and this method doesn't appear to do much at first glance:

```
def set_environment
  ENV["RAILS_ENV"] ||= options[:environment]
end
```

In fact, the `options` method here does quite a lot. This method is defined in `Rack::Server` like this:

```
def options
  @options ||= parse_options(ARGV)
end
```

Then `parse_options` is defined like this:

```
def parse_options(args)
  options = default_options
```

```
  # Don't evaluate CGI ISINDEX parameters.
  # http://www.meb.uni-bonn.de/docs/cgi/cl.html
  args.clear if ENV.include?("REQUEST_METHOD")

  options.merge! opt_parser.parse!(args)
  options[:config] = ::File.expand_path(options[:config])
  ENV["RACK_ENV"] = options[:environment]
  options
end
```

With the `default_options` set to this:

```
def default_options
  environment  = ENV['RACK_ENV'] || 'development'
  default_host = environment == 'development' ? 'localhost' : '0.0.0.0'

  {
    :environment => environment,
    :pid         => nil,
    :Port        => 9292,
    :Host        => default_host,
    :AccessLog   => [],
    :config      => "config.ru"
  }
end
```

There is no `REQUEST_METHOD` key in `ENV` so we can skip over that line. The next line merges in the options from `opt_parser` which is defined plainly in `Rack::Server`:

```
def opt_parser
  Options.new
end
```

The class **is** defined in `Rack::Server`, but is overwritten in `Rails::Server` to take different arguments. Its `parse!` method begins like this:

```
def parse!(args)
  args, options = args.dup, {}

  opt_parser = OptionParser.new do |opts|
    opts.banner = "Usage: rails server [mongrel, thin, etc] [options]"
    opts.on("-p", "--port=port", Integer,
            "Runs Rails on the specified port.", "Default: 3000") { |v| options[:Port] = v
}
  ...
```

This method will set up keys for the `options` which Rails will then be able to use to determine how its server should run. After `initialize` has finished, we jump back into `rails/server` where `APP_PATH` (which was set earlier) is required.

1.10 config/application

When `require APP_PATH` is executed, `config/application.rb` is loaded (recall that `APP_PATH` is defined in `bin/rails`). This file exists in your application and it's free for you to change based on your needs.

1.11 Rails::Server#start

After `config/application` is loaded, `server.start` is called. This method is defined like this:

```
def start
  print_boot_information
  trap(:INT) { exit }
  create_tmp_directories
  log_to_stdout if options[:log_stdout]

  super
  ...
end

private

  def print_boot_information
    ...
    puts "=> Run `rails server -h` for more startup options"
    ...
    puts "=> Ctrl-C to shutdown server" unless options[:daemonize]
  end

  def create_tmp_directories
    %w(cache pids sessions sockets).each do |dir_to_make|
      FileUtils.mkdir_p(File.join(Rails.root, 'tmp', dir_to_make))
    end
  end

  def log_to_stdout
    wrapped_app # touch the app so the logger is set up

    console = ActiveSupport::Logger.new($stdout)
    console.formatter = Rails.logger.formatter
    console.level = Rails.logger.level

    Rails.logger.extend(ActiveSupport::Logger.broadcast(console))
  end
```

This is where the first output of the Rails initialization happens. This method creates a trap for `INT` signals, so if you `CTRL-C` the server, it will exit the process. As we can see from the code here, it will create the `tmp/`

cache, tmp/pids, tmp/sessions and tmp/sockets directories. It then calls wrapped_app which is responsible for creating the Rack app, before creating and assigning an instance of ActiveSupport::Logger.

The super method will call Rack::Server.start which begins its definition like this:

```
def start &blk
  if options[:warn]
    $-w = true
  end

  if includes = options[:include]
    $LOAD_PATH.unshift(*includes)
  end

  if library = options[:require]
    require library
  end

  if options[:debug]
    $DEBUG = true
    require 'pp'
    p options[:server]
    pp wrapped_app
    pp app
  end

  check_pid! if options[:pid]

  # Touch the wrapped app, so that the config.ru is loaded before
  # daemonization (i.e. before chdir, etc).
  wrapped_app

  daemonize_app if options[:daemonize]

  write_pid if options[:pid]

  trap(:INT) do
    if server.respond_to?(:shutdown)
      server.shutdown
    else
      exit
    end
  end

  server.run wrapped_app, options, &blk
end
```

The interesting part for a Rails app is the last line, `server.run`. Here we encounter the `wrapped_app` method again, which this time we're going to explore more (even though it was executed before, and thus memoized by now).

```
@wrapped_app ||= build_app app
```

The `app` method here is defined like so:

```
def app
  @app ||= options[:builder] ? build_app_from_string : build_app_and_options_from_config
end
...
private
  def build_app_and_options_from_config
    if !::File.exist? options[:config]
      abort "configuration #{options[:config]} not found"
    end

    app, options = Rack::Builder.parse_file(self.options[:config], opt_parser)
    self.options.merge! options
    app
  end

  def build_app_from_string
    Rack::Builder.new_from_string(self.options[:builder])
  end
```

The `options[:config]` value defaults to `config.ru` which contains this:

```
# This file is used by Rack-based servers to start the application.

require ::File.expand_path('../config/environment', __FILE__)
run <%= app_const %>
```

The `Rack::Builder.parse_file` method here takes the content from this `config.ru` file and parses it using this code:

```
app = new_from_string cfgfile, config

...

def self.new_from_string(builder_script, file="(rackup)")
  eval "Rack::Builder.new {\n" + builder_script + "\n}.to_app",
    TOPLEVEL_BINDING, file, 0
end
```

The `initialize` method of `Rack::Builder` will take the block here and execute it within an instance of `Rack::Builder`. This is where the majority of the initialization process of Rails happens. The `require` line for `config/environment.rb` in `config.ru` is the first to run:

```
require ::File.expand_path('../config/environment', __FILE__)
```

1.12 config/environment.rb

This file is the common file required by `config.ru` (`rails server`) and Passenger. This is where these two ways to run the server meet; everything before this point has been Rack and Rails setup.

This file begins with requiring `config/application.rb`:

```
require File.expand_path('../application', __FILE__)
```

1.13 config/application.rb

This file requires `config/boot.rb`:

```
require File.expand_path('../boot', __FILE__)
```

But only if it hasn't been required before, which would be the case in `rails server` but **wouldn't** be the case with Passenger.

Then the fun begins!

2 Loading Rails

The next line in `config/application.rb` is:

```
require 'rails/all'
```

2.1 railties/lib/rails/all.rb

This file is responsible for requiring all the individual frameworks of Rails:

```
require "rails"

%w(
  active_record
  action_controller
  action_view
  action_mailer
  rails/test_unit
  sprockets
).each do |framework|
  begin
    require "#{framework}/railtie"
  rescue LoadError
  end
end
```

This is where all the Rails frameworks are loaded and thus made available to the application. We won't go into detail of what happens inside each of those frameworks, but you're encouraged to try and explore them on your own.

For now, just keep in mind that common functionality like Rails engines, I18n and Rails configuration are all being defined here.

2.2 Back to config/environment.rb

The rest of config/application.rb defines the configuration for the Rails::Application which will be used once the application is fully initialized. When config/application.rb has finished loading Rails and defined the application namespace, we go back to config/environment.rb, where the application is initialized. For example, if the application was called Blog, here we would find Rails.application.initialize!, which is defined in rails/application.rb.

2.3 railties/lib/rails/application.rb

The initialize! method looks like this:

```
def initialize!(group=:default) #:nodoc:
  raise "Application has been already initialized." if @initialized
  run_initializers(group, self)
  @initialized = true
  self
end
```

As you can see, you can only initialize an app once. The initializers are run through the run_initializers method which is defined in railties/lib/rails/initializable.rb:

```
def run_initializers(group=:default, *args)
  return if instance_variable_defined?(:@ran)
  initializers.tsort_each do |initializer|
    initializer.run(*args) if initializer.belongs_to?(group)
  end
  @ran = true
end
```

The run_initializers code itself is tricky. What Rails is doing here is traversing all the class ancestors looking for those that respond to an initializers method. It then sorts the ancestors by name, and runs them. For example, the Engine class will make all the engines available by providing an initializers method on them.

The Rails::Application class, as defined in railties/lib/rails/application.rb defines bootstrap, railtie, and finisher initializers. The bootstrap initializers prepare the application (like initializing the logger) while the finisher initializers (like building the middleware stack) are run last. The railtie initializers are the initializers which have been defined on the Rails::Application itself and are run between the bootstrap and finishers.

After this is done we go back to Rack::Server.

2.4 Rack: lib/rack/server.rb

Last time we left when the app method was being defined:

```
def app
  @app ||= options[:builder] ? build_app_from_string : build_app_and_options_from_config
```

```
end
...
private
  def build_app_and_options_from_config
    if !::File.exist? options[:config]
      abort "configuration #{options[:config]} not found"
    end

    app, options = Rack::Builder.parse_file(self.options[:config], opt_parser)
    self.options.merge! options
    app
  end

  def build_app_from_string
    Rack::Builder.new_from_string(self.options[:builder])
  end
```

At this point `app` is the Rails app itself (a middleware), and what happens next is Rack will call all the provided middlewares:

```
def build_app(app)
  middleware[options[:environment]].reverse_each do |middleware|
    middleware = middleware.call(self) if middleware.respond_to?(:call)
    next unless middleware
    klass = middleware.shift
    app = klass.new(app, *middleware)
  end
  app
end
```

Remember, `build_app` was called (by `wrapped_app`) in the last line of `Server#start`. Here's how it looked like when we left:

```
server.run wrapped_app, options, &blk
```

At this point, the implementation of `server.run` will depend on the server you're using. For example, if you were using Puma, here's what the `run` method would look like:

```
...
DEFAULT_OPTIONS = {
  :Host => '0.0.0.0',
  :Port => 8080,
  :Threads => '0:16',
  :Verbose => false
}

def self.run(app, options = {})
```

```
  options  = DEFAULT_OPTIONS.merge(options)

  if options[:Verbose]
    app = Rack::CommonLogger.new(app, STDOUT)
  end

  if options[:environment]
    ENV['RACK_ENV'] = options[:environment].to_s
  end

  server   = ::Puma::Server.new(app)
  min, max = options[:Threads].split(':', 2)

  puts "Puma #{::Puma::Const::PUMA_VERSION} starting..."
  puts "* Min threads: #{min}, max threads: #{max}"
  puts "* Environment: #{ENV['RACK_ENV']}"
  puts "* Listening on tcp://#{options[:Host]}:#{options[:Port]}"

  server.add_tcp_listener options[:Host], options[:Port]
  server.min_threads = min
  server.max_threads = max
  yield server if block_given?

  begin
    server.run.join
  rescue Interrupt
    puts "* Gracefully stopping, waiting for requests to finish"
    server.stop(true)
    puts "* Goodbye!"
  end

end
```

We won't dig into the server configuration itself, but this is the last piece of our journey in the Rails initialization process.

This high level overview will help you understand when your code is executed and how, and overall become a better Rails developer. If you still want to know more, the Rails source code itself is probably the best place to go next.

3 Feedback

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

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

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

the master branch. Check the Ruby on Rails Guides Guidelines for style and conventions.

If for whatever reason you spot something to fix but cannot patch it yourself, please open an issue.

And last but not least, any kind of discussion regarding Ruby on Rails documentation is very welcome in the rubyonrails-docs mailing list.
