The Rails Initialization Process

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

After reading this guide, you will know:

- How to use `bin/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 application, explaining each part in detail along the way. For this guide, we will be focusing on what happens when you execute `bin/rails server` to boot your app.

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

TIP: 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.

Launch!

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

`bin/rails`

This file is as follows:

```
#!/usr/bin/env ruby
APP_PATH = File.expand_path('../config/application', __dir__)
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.

`config/boot.rb`

`config/boot.rb` contains:

```
ENV['BUNDLE_GEMFILE'] ||= File.expand_path('../Gemfile', __dir__)

require "bundler/setup" # Set up gems listed in the 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.

`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:

```
require "rails/command"

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

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

Rails::Command.invoke command, ARGV
```

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

`rails/command.rb`

When one types a Rails command, `invoke` tries to lookup a command for the given namespace and executes the command if found.

If Rails doesn't recognize the command, it hands the reins over to Rake to run a task of the same name.

As shown, `Rails::Command` displays the help output automatically if the `namespace` is empty.

```
module Rails
  module Command
    class << self
      def invoke(full_namespace, args = [], **config)
        namespace = full_namespace = full_namespace.to_s

        if char = namespace =~ /:(\w+)$/
          command_name, namespace = $1, namespace.slice(0, char)
        else
          command_name = namespace
        end

        command_name, namespace = "help", "help" if command_name.blank? ||
HELP_MAPPINGS.include?(command_name)
        command_name, namespace = "version", "version" if %w( -v --version
).include?(command_name)

        command = find_by_namespace(namespace, command_name)
        if command && command.all_commands[command_name]
          command.perform(command_name, args, config)
        else
          find_by_namespace("rake").perform(full_namespace, args, config)
        end
      end
    end
  end
end
```

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

```
module Rails
  module Command
    class ServerCommand < Base # :nodoc:
      def perform
        extract_environment_option_from_argument
        set_application_directory!
        prepare_restart

        Rails::Server.new(server_options).tap do |server|
          # Require application after server sets environment to propagate
          # the --environment option.
          require APP_PATH
          Dir.chdir(Rails.application.root)

          if server.serveable?
            print_boot_information(server.server, server.served_url)
            after_stop_callback = -> { say "Exiting" unless options[:daemon] }
            server.start(after_stop_callback)
          else
            say rack_server_suggestion(using)
          end
        end
      end
    end
  end
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 starts up the `Rails::Server` class.

`actionpack/lib/action_dispatch.rb`

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

`rails/commands/server/server_command.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/server_command.rb`:

```
module Rails
  class Server < ::Rack::Server
    def initialize(options = nil)
      @default_options = options || {}
      super(@default_options)
      set_environment
    end
  end
end
```

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

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 several variables:

```
module Rack
  class Server
    def initialize(options = nil)
      @ignore_options = []

      if options
        @use_default_options = false
        @options = options
        @app = options[:app] if options[:app]
      else
        argv = defined?(SPEC_ARGV) ? SPEC_ARGV : ARGV
        @use_default_options = true
        @options = parse_options(argv)
      end
    end
  end
end
```

In this case, return value of `Rails::Command::ServerCommand#server_options` will be assigned to `options`. When lines inside if statement is evaluated, a couple of instance variables will be set.

`server_options` method in `Rails::Command::ServerCommand` is defined as follows:

```
module Rails
  module Command
```

```
      class ServerCommand
        no_commands do
          def server_options
            {
              user_supplied_options: user_supplied_options,
              server:                using,
              log_stdout:            log_to_stdout?,
              Port:                  port,
              Host:                  host,
              DoNotReverseLookup:    true,
              config:                options[:config],
              environment:           environment,
              daemonize:             options[:daemon],
              pid:                   pid,
              caching:               options[:dev_caching],
              restart_cmd:           restart_command,
              early_hints:           early_hints
            }
          end
        end
      end
    end
  end
```

The value will be assigned to instance variable `@options`.

After `super` has finished in `Rack::Server`, we jump back to `rails/commands/server/server_command.rb`. At this point, `set_environment` is called within the context of the `Rails::Server` object.

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

After `initialize` has finished, we jump back into the server command where `APP_PATH` (which was set earlier) is required.

`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.

`Rails::Server#start`

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

```
module Rails
  class Server < ::Rack::Server
    def start(after_stop_callback = nil)
      trap(:INT) { exit }
      create_tmp_directories
      setup_dev_caching
      log_to_stdout if options[:log_stdout]

      super()
      # ...
    end

    private
      def setup_dev_caching
        if options[:environment] == "development"
          Rails::DevCaching.enable_by_argument(options[:caching])
        end
      end

      def create_tmp_directories
        %w(cache pids 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

        unless ActiveSupport::Logger.logger_outputs_to?(Rails.logger, STDOUT)
          Rails.logger.extend(ActiveSupport::Logger.broadcast(console))
        end
      end
  end
end
```

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`, and `tmp/sockets` directories. It then enables caching in development if `bin/rails server` is called with `--dev-caching`. Finally, it 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 as follows:

```
module Rack
  class Server
    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).
        handle_profiling(options[:heapfile], options[:profile_mode],
options[:profile_file]) do
          wrapped_app
        end

        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
    end
  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).

```
module Rack
  class Server
    def wrapped_app
      @wrapped_app ||= build_app app
    end
  end
end
```

The `app` method here is defined like so:

```
module Rack
  class Server
    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)
        @options.merge!(options) { |key, old, new| old }
        app
      end

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

  end
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_relative "config/environment"

run Rails.application
```

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

```
module Rack
  class Builder
    def self.load_file(path, opts = Server::Options.new)
      # ...
      app = new_from_string cfgfile, config
      # ...
    end

    # ...

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

`config/environment.rb`

This file is the common file required by `config.ru` (`bin/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_relative "application"
```

`config/application.rb`

This file requires `config/boot.rb`:

```
require_relative "boot"
```

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

Then the fun begins!

Loading Rails

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

```
require "rails/all"
```

`railties/lib/rails/all.rb`

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

```
require "rails"

%w(
  active_record/railtie
  active_storage/engine
  action_controller/railtie
  action_view/railtie
  action_mailer/railtie
  active_job/railtie
  action_cable/engine
  action_mailbox/engine
  action_text/engine
  rails/test_unit/railtie
).each do |railtie|
  begin
    require 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.

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`. Here, the application is initialized with `Rails.application.initialize!`, which is defined in `rails/application.rb`.

`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
```

You can only initialize an app once. The Railtie 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` .

Note: Do not confuse Railtie initializers overall with the load_config_initializers initializer instance or its associated config initializers in `config/initializers` .

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

Rack: lib/rack/server.rb

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

```
module Rack
  class Server
    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)
        @options.merge!(options) { |key, old, new| old }
        app
      end

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

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

```ruby
module Rack
  class Server
    private
      def build_app(app)
        middleware[options[:environment]].reverse_each do |middleware|
          middleware = middleware.call(self) if middleware.respond_to?(:call)
          next unless middleware
          klass, *args = middleware
          app = klass.new(app, *args)
        end
        app
      end
  end
end
```

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

```ruby
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:

```ruby
module Rack
  module Handler
    module Puma
      # ...
      def self.run(app, options = {})
        conf   = self.config(app, options)

        events = options.delete(:Silent) ? ::Puma::Events.strings :
::Puma::Events.stdio

        launcher = ::Puma::Launcher.new(conf, :events => events)

        yield launcher if block_given?
        begin
          launcher.run
        rescue Interrupt
          puts "* Gracefully stopping, waiting for requests to finish"
          launcher.stop
          puts "* Goodbye!"
        end
      end
```

```
        # ...
      end
    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.