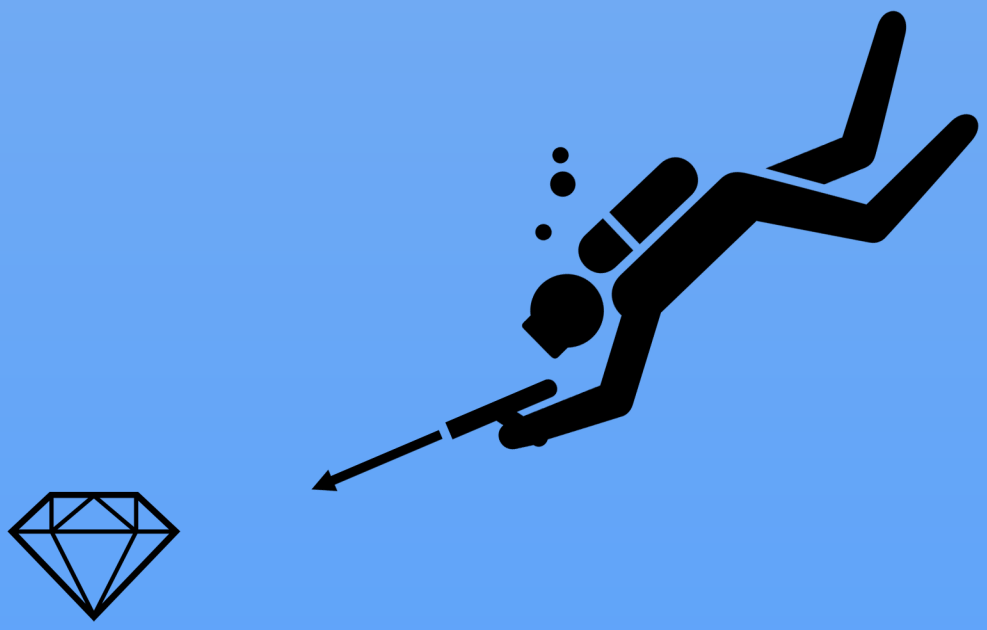
DEEP DIVE RAILS

A deep dive into the Rails initialization process and request cycle



RYAN BIGG

Deep Dive Rails

Table of Contents

Housekeeping 1

Preface 2

High-level overview 3

1. Starting Rails. 5

    1.1. [somewhere]/bin/rails. 7

    1.2. [railties gem]/bin/rails 9

    1.3. [railties gem]/lib/rails/cli.rb. 10

    1.4. [railties gem]/lib/rails/app_loader.rb. 12

    1.5. [our application]/bin/rails 14

    1.6. [railties gem]/lib/rails/commands.rb 15

    1.7. [railties gem]/lib/rails/commands_tasks.rb 16

    1.8. [railties gem]/lib/rails/commands/server.rb 18

Housekeeping

If you find a misteak while reading this book, please file an issue on the Errata Repo on GitHub with the following details:

- Which page (or section) you found the mistake
- What the mistake actually is
- And a helpful suggestion for fixing it

> **NOTE** Reviewers with access to Twist please leave your comments there as you read them. If you don't know what a Twist is, then please ignore this message.

That'd be great!

This far into the book and there's already one mistake. You're almost guaranteed that there will be others. Their ways are wily and devious. **BEWARE!**

Preface

This book will show you how Rails boots up and is particularly written for the 5.0.2 version of Rails.

This book has been a long time coming. I (Ryan) originally started writing something along these same lines when I was taking donations to work on Rails documentation back at the end of 2010. I got about 8,000 words into writing it as a Rails guide and figured that nobody would want to read a guide that long and so I gave up.

A lot of people have convinced me that they still would want to read a long guide/book/walkthrough of the boot procedure and so I thought "why not?" and started writing one.

This book is a deep-dive and if you wanted us to put a rating on the skill level required to get enjoyment out of this book, I would place it at "Intermediate". This is a book which assumes that you've got a moderately- good-but-not-expert-so-don't-stress-about-it kind of knowledge of Ruby and Rails. The book won't explain basic Ruby or Rails concepts to you, as you should already be familiar with those. If you know the difference between a model and a controller or a module and a class, this book will be easy for you to read.

This book will go through entirety of the Rails initialization process from start-to-finish. Yes, the whole bloody thing. You will learn about configuration options that you didn't even know existed. There are things in here that Rails does for you that I didn't even know Rails did for us; and I've been doing Rails for **ten years**.

Then — for a bit of extra fun — we'll go through the process of what happens when a very plain action is rendered from a Rails controller.

Enough waffling, let's get started!

High-level overview

Before diving deep into Rails, it's definitely best to get a high level overview of what exactly happens when Rails serves a request. The TL;DR version of it is that your browser makes a request to a webserver and it returns a response. That's Basic Internet Servers 101, which you probably already know if you're reading this book.

To start our highlevel overview, let's cover what happens when you're running a local Rails app and you enter `localhost:3000` into your browser window and hit enter. We'll skip over the DNS parts because that's out of scope of this book. What we'll be focussing on is what happens after that DNS resolution has completed.

When you type `localhost:3000` into your browser and hit enter, the browser makes a request to the server that's running on that port. By default, that server is the Puma webserver, which comes standard with every Rails 5+ application. Puma then accepts this request and passes the handling of this request off to Rack.

Rack is a gem that unifies the interface for all Ruby webservers. This means that you could choose to put a non-Puma server in front of your Rails app — such as Thin, WEBrick or Unicorn — and Rails won't care one bit because Rack provides the same interface to Rails no matter what server you're using. When Rack receives this request, Rack converts the request into a format that Rails's own internal API is compatible with and then asks Rails to serve the request. The request after this point is referred to as a "Rack environment", or "env" for short.

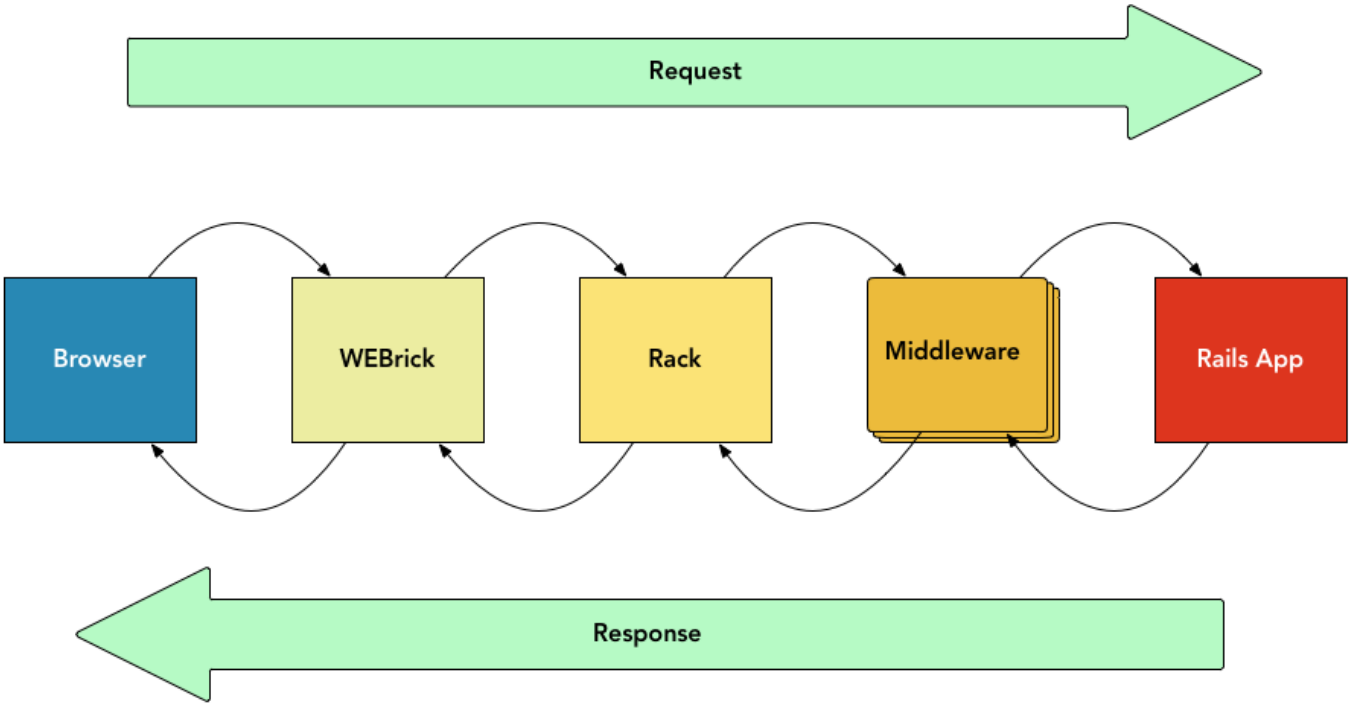
The Rails application passes the request (`env`) down its *middleware stack*. Each piece of middleware can modify the request on the way in, stop the request in its tracks and return a response, or modify the request on its way out. For the moment, let's assume that the request isn't stopped by any of these pieces, and goes through the entire middleware stack to the Rails application.

At the point where the Rails application receives the request, the application's router determines from the requested URL and method where to route that request to. If you have a route such as `root to: "home#index"` in your `config/routes.rb`, Rails will route a `GET /` request to this controller and action. This is exactly the request we're making right now when we make a call to `localhost:3000` — the request is `GET /`.

The router within Rails will then pass the request to the controller's action. The controller's action then might pull some stuff from a database and render an HTML template. Let's say that the controller action does indeed do that.

After that point, the action is done with the request and the response goes back the way the request came in, but in reverse. The response gets passed back up through the router, through the middleware, through Rack, back to the Puma server and then back to your browser.

Here's the whole process illustrated:



So that's a quick overview of how a request flows from your browser through to a Rails application and then how the corresponding response travels back to your browser.

Now that we've looked at the surface of what happens, it's time to commence our deep dive into Rails and how it handles incoming requests and their responses.

1. Starting Rails

To start a deep dive the best thing to do is to start at the beginning. What we want to learn is how Rails boots up and serves requests. In a local development environment, on a brand new Rails application, the way to boot up a Rails application is with this command:

```
rails s
```

If that command doesn't work for you, make sure you have installed Rails first:

```
gem install rails
```

If that command doesn't work, make sure you have installed Ruby first.

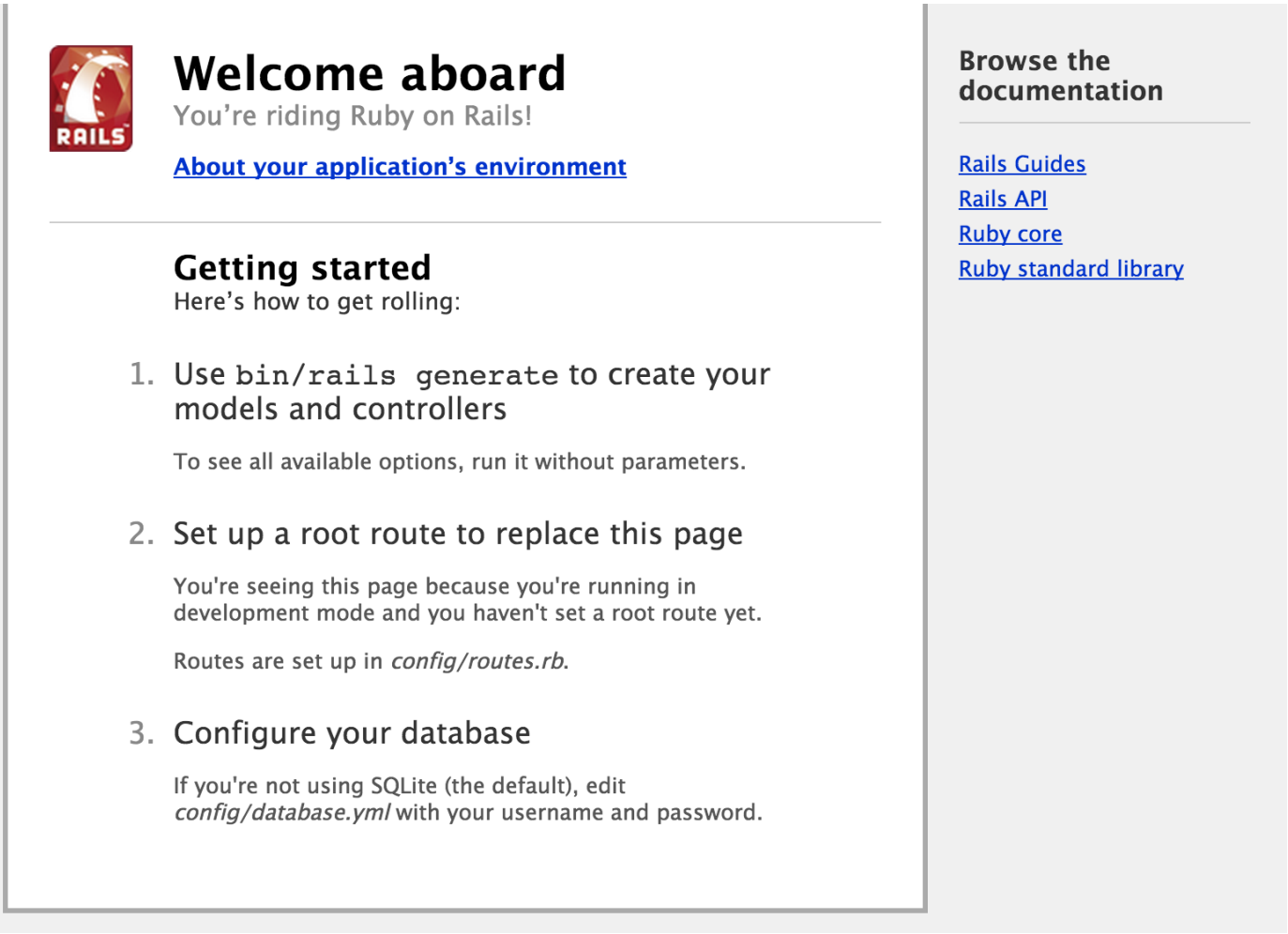
We know the general gist of what this command does: it starts a Rails server. But what is it *really* doing? To find out, we're going to need to run this command in a real Rails application, so let's create one of those now and name it inventively "ddr":

```
rails new ddr
```

Now that we have our application up, let's see what happens when we run that `rails s` command. We'll see this output:

```
=> Booting Puma
=> Rails 5.0.2 application starting in development on http://localhost:3000
=> Run `rails server -h` for more startup options
Puma starting in single mode...
* Version 3.8.2 (ruby 2.4.0-p0), codename: Snowy Sagebrush
* Min threads: 5, max threads: 5
* Environment: development
* Listening on tcp://0.0.0.0:3000
Use Ctrl-C to stop
```

You're probably already familiar with that output — or something very similar to it — already. Of course when we go to http://localhost:3000 in our browser we'll see the very familiar "Welcome Aboard" page.



How does this command work? We can find out on any UNIX system by running the `command -v` command:

```
command -v rails
```

The `command` utility will tell us where a program lives in the user's `$PATH`; just as long as we give it the `-v` switch. If you're on Windows, you probably have something like this but we're not Windows users, so you'll have to figure that out yourself.

On my machine, this prints out this:

```
/Users/ryanbigg/.gem/ruby/2.4.0/bin/rails
```

For you it might be something very similar, or very different. What that path is isn't necessarily important in this case. It's good to know that the `bin` directory was added to your `$PATH` by whatever manages your Ruby installation. On my machine, that is a combination of chruby and ruby-install.

What is ultimately important is what's inside that file.

1.1. [somewhere]/bin/rails

Let's open this file now and see:

[somewhere]/bin/rails

```
#!/usr/bin/env ruby
#
# This file was generated by RubyGems.
#
# The application 'railties' is installed as part of a gem, and
# this file is here to facilitate running it.
#

require 'rubygems'

version = ">= 0.a"

if ARGV.first
  str = ARGV.first
  str = str.dup.force_encoding("BINARY") if str.respond_to? :force_encoding
  if str =~ /\A_(.*)_\z/ and Gem::Version.correct?($1) then
    version = $1
    ARGV.shift
  end
end

load Gem.activate_bin_path('railties', 'rails', version)
```

This file is an executable and uses the shebang [1: Bash Shebang: http://en.wikipedia.org/wiki/Shebang_%28Unix%29] to parse the remainder of the program through a Ruby interpreter. Without this line, it would be interpreted as whatever shell scripting language you're using. That's probably going to be Bash unless you're one of those hipster types that use zsh.

The next non-commented thing in this file is a require to `rubygems`. This provides us with the RubyGem classes + methods (such as `Gem.activate_bin_path`) which are used later on this file.

After that, there's a bit of a dance in order to determine if we've specified a version for this `rails` command. What this `if ARGV.first` code does is enable this feature:

```
rails _4.2.0_ new app
```

It allows you to load a *different* version of Rails and run that instead of whatever RubyGems would pick up by default.

Interestingly enough, this feature isn't unique to Rails. You can actually use it with any executable provided by a gem. For instance, if we wanted to run a particular version of the `rack` gem when we ran `rackup`, we could use the same syntax:

```
rackup _2.0.1_ config.ru
```

This file generated by RubyGems will first check to see if the first argument is a version, and if it is then it will load that particular version of the gem. Let's take a look at how that happens.

The code checks to see if the first argument (`ARGV.first`) matches this regular expression:

```
/\A_(.*)_\z/
```

If it does, *and* if that looks like a "correct" version according to RubyGems [2: For what RubyGems considers a "correct" version number, read these code comments from `Gem::Version`: http://git.io/vfs0Y], that version is used for what comes next.

What comes next is these two lines which really kickstart the whole process:

```
gem 'railties', version
load Gem.bin_path('railties', 'rails', version)
```

If `version` has been specified, then that version of the `railties` gem will be loaded. If we specified `_4.2.0_` when running our `rails` command, then that version will be loaded. If we don't specify any version, then the latest-and-greatest version that's installed will be used.

Is it "rail tees" or "rail ties"?

The `railties` gem provides features that *tie* together Rails components. So this gem's name is pronounced "rail ties", rather than "rail tees"... in case you were wondering.

The final line in this file looks in the `bin` path of the specific version of the `railties` gem and loads the `rails` file in that path. We can find what script that's finding by running the same code with `ruby -e`:

```
ruby -rubygems -e "puts Gem.bin_path('railties', 'rails', '>=0')"
```

You can change the `>=0` to a particular version of Rails and see the path change; providing of course that you have that specific version of the `railties` gem installed. On my machine, this command prints out this:

```
/Users/ryanbigg/.gem/ruby/2.4.0/gems/railties-5.0.1/exe/rails
```

Let's look at what this file does.

1.2. [railties gem]/bin/rails

The contents of this file are this:

```
#!/usr/bin/env ruby[]

git_path = File.expand_path('../../../.git', __FILE__)

if File.exist?(git_path)
  railties_path = File.expand_path('../../lib', __FILE__)
  $:.unshift(railties_path)
end
require "rails/cli"
```

This file first checks to see if there's a `.git` directory three directories up from the path where this file is. There would only be a `.git` directory if we were attempting to use this file in a cloned version of Rails. In this case, the `railties` gem might not be in the `$LOAD_PATH` already or if it is, it might be a different version to the one in the repo. The code inside the `if` here adds the `lib` directory to the beginning of the `$LOAD_PATH` using its shorthand variant of `$:` and `unshift`. [3: The code inside the `if` is not ran in a typical Rails application, but is still worth covering for those who are curious to know what it does.]

The final line in this file requires the file `rails/cli`. The code above it can give us some indication of where we might find it: in the `railties` gem itself. Indeed, that is where it is.

The railties gem

The railties gem provides ties between Rails and its related components. The `rails` gem doesn't contain much itself, but rather acts as a way to pull in all of Rails related components together. It's the `railties` gem which provides the "brains" of every Rails application.

According to the README for railties, the gem is responsible for:

- handling the bootstrapping process for a Rails application;
- managing the rails command line interface;
- and provides the Rails generators core.

The second dot-point there is exactly what we're about to look at. We'll be spending a bit of time in the `railties` gem.

1.3. [railties gem]/lib/rails/cli.rb

The contents of this file are:

[railties gem]/lib/rails/cli.rb

```
require 'rails/app_loader'

# If we are inside a Rails application this method performs an exec and thus
# the rest of this script is not run.
Rails::AppLoader.exec_app

require 'rails/ruby_version_check'
Signal.trap("INT") { puts; exit(1) }

if ARGV.first == 'plugin'
  ARGV.shift
  require 'rails/commands/plugin'
else
  require 'rails/commands/application'
end
```

The first line of this file requires `rails/app_loader`, and the second line runs a method called `Rails::AppLoader.exec_app` which would seem like it has been defined in that file based on the similarity of the class name and the filename. As the comment says, if we're inside a Rails application this is where this file stops and

`exec_app` takes over. We'll get to that in a minute. Let's see what the rest of the file has for us first.

The next line is a require to `rails/ruby_version_check` which requires this file:

[railties gem]/lib/rails/ruby_version_check.rb

```
if RUBY_VERSION < '2.2.2' && RUBY_ENGINE == 'ruby'
  desc = defined?(RUBY_DESCRIPTION) ? RUBY_DESCRIPTION : "ruby #{RUBY_VERSION}
(#{RUBY_RELEASE_DATE})"
  abort <<-end_message

    Rails 5 requires Ruby 2.2.2 or newer.

    You're running
      #{desc}

    Please upgrade to Ruby 2.2.2 or newer to continue.

  end_message
end
```

The code here checks the `RUBY_VERSION` constant defined by the Ruby interpreter that we're using. If that is less than Ruby 2.2.2, this file will abort the process using the `abort` method. The message it will abort with is everything between `< end_message` and its counterpart, `end_message`; a Heredoc.

This is done because there is some code in Rails which just isn't compatible with earlier versions of Ruby.

Let's jump back to the `[railties gem]/lib/rails/cli.rb` file and the remainder of its code:

[railties gem]/lib/rails/cli.rb

```
Signal.trap("INT") { puts; exit(1) }

if ARGV.first == 'plugin'
  ARGV.shift
  require 'rails/commands/plugin'
else
  require 'rails/commands/application'
end
```

The `Signal.trap` method here will trap an `INT` signal (aka `SIGINT` or "interrupt signal"); typically issued by `Command+C` or `Control+C` shortcuts. If that code encounters one of these signals, it will `exit` with a code of `1`, indicating that the process exited unusually. If that `Signal.trap` call wasn't here then we would see a stacktrace if we hit `Command+C` or `Control+C` at any point during the running of the Rails server.

The final lines for this file check if the first argument provided to the `rails` command is "plugin". If that's the case, it will require `rails/commands/plugin`. This would happen if you were to run the command `rails plugin new`, for example. If the argument isn't "plugin" then it's assumed you want to do something with the application, and `rails/commands/application` is loaded instead.

To find out what those do, go on and explore yourself. Consider it the first piece of homework in this book! The book will continue to focus on what happens when we start an application, and the next step begins with the file that we first saw required at the top of `rails/cli.rb`: `rails/app_rails_loader.rb`.

1.4. [railties gem]/lib/rails/app_loader.rb

This file is the longest file we've seen so far. Let's look at it in parts. The first part contains this:

```
require 'pathname'
require 'rails/version'

module Rails
  module AppLoader # :nodoc:
    extend self

    RUBY = Gem.ruby
    EXECUTABLES = ['bin/rails', 'script/rails']
    BUNDLER_WARNING = <<EOS
Looks like your app's ./bin/rails is a stub that was generated by Bundler.

In Rails #{Rails::VERSION::MAJOR}, your app's bin/ directory contains executables that
are versioned
like any other source code, rather than stubs that are generated on demand.

Here's how to upgrade:

  bundle config --delete bin    # Turn off Bundler's stub generator
  rails app:update:bin          # Use the new Rails 5 executables
  git add bin                   # Add bin/ to source control

You may need to remove bin/ from your .gitignore as well.

When you install a gem whose executable you want to use in your app,
generate it and add it to source control:

  bundle binstubs some-gem-name
  git add bin/new-executable

EOS
```

This sets up a couple of constants: `RUBY`, `EXECUTABLES` and `BUNDLER_WARNING`. The `RUBY` constant defined here will be the path to the Ruby binary, which on my machine is `/Users/ryanbigg/.rubies/ruby-2.4.0/bin/ruby`. All of these constants are used in the latter half of the file, which contains the `exec_app` method that was called from `rails/cli.rb`:

```
def exec_app
  original_cwd = Dir.pwd

  loop do
    if exe = find_executable
      contents = File.read(exe)

      if contents =~ /(APP|ENGINE)_PATH/
        exec RUBY, exe, *ARGV
        break # non reachable, hack to be able to stub exec in the test suite
      elsif exe.end_with?('bin/rails') && contents.include?('This file was generated by
Bundler')
        $stderr.puts(BUNDLER_WARNING)
        Object.const_set(:APP_PATH, File.expand_path('config/application', Dir.pwd))
        require File.expand_path('../boot', APP_PATH)
        require 'rails/commands'
        break
      end
    end

    # If we exhaust the search there is no executable, this could be a
    # call to generate a new application, so restore the original cwd.
    Dir.chdir(original_cwd) and return if Pathname.new(Dir.pwd).root?

    # Otherwise keep moving upwards in search of an executable.
    Dir.chdir('..')
  end
end

def find_executable
  EXECUTABLES.find { |exe| File.file?(exe) }
end
```

The `exec_app_rails` method is quite long! What this method will do is attempt to find one of `bin/rails` or `script/rails` at the current path, and then if it can't find it there then it will go up a directory and look there instead, going as far up the directory tree as it can. When it does find one of `bin/rails` or `script/rails`, it executes it using `exec` and this code:

```
if contents =~ /(APP|ENGINE)_PATH/
  exec RUBY, exe, *ARGV
```

This code will start a new process with the `RUBY` constant defined earlier, and will invoke the executable using a command like this:

```
/Users/ryanbigg/.rubies/ruby-2.4.0/bin/ruby bin/rails s
```

If this executable exists and it *doesn't* contain one of `APP_PATH` or `ENGINE_PATH`, then the code inside of `app_loader.rb` falls to the `elsif`:

```
elsif exe.end_with?('bin/rails') && contents.include?('This file was generated by
Bundler')
  $stderr.puts(BUNDLER_WARNING)
  Object.const_set(:APP_PATH, File.expand_path('config/application', Dir.pwd))
  require File.expand_path('../boot', APP_PATH)
  require 'rails/commands'
  break
end
```

This checks if the file's path is `bin/rails`, and if the contents include a message that indicates if it was generate by Bundler. If you run `bundle install --binstubs`, it will override Rails's default `bin/rails` and lead down this path. The Rails-generated version of this file contains `APP_PATH` and code that will be used to boot our application. The `BUNDLER_WARNING` message will tell you how to work around this problem, while the remainder of this code will do what the proper `bin/rails` should've done in the first place.

Let's assume the happy path here — where you haven't run `bundle install --binstubs`, or if you have you've fixed it up — and move on from here. The `exec` method has been given a path to our `bin/rails` script within our application and will now execute that.

1.5. [our application]/bin/rails

This file is much shorter than the previous file, consisting of only 4 lines:

[our application]/bin/rails

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

Because this file has been called from `exec`, the first line in the file needs to re-establish that we're running Ruby code, and it does this with the same Shebang we saw back in our original `[somewhere]/bin/rails` — that's the one provided by the railties gem when it's installed.

After that, it defines a constant called `APP_PATH` which will be used to indicate to Rails that we're booting an application and not an engine. The path points to `config/application.rb` in our application, using `_\_dir\__` to refer to the current directory.

The third line requires `config/boot.rb` using `require_relative` which is one of those new-fangled Ruby 1.9 features, and the first of many reasons why `ruby_version_check.rb` exists! That file comprises of these lines:

[our application]/config/boot.rb

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

require 'bundler/setup' # Set up gems listed in the Gemfile.
```

This file configures the environment variable `BUNDLE_GEMFILE`; but only if it's not already set. The final line requires `bundler/setup`, which is responsible for adding all the gems in the `Gemfile` to the `$LOAD_PATH`. This is what will enable some of the `require` calls to work in a little while.

The final line of `[our application]/bin/rails` will lead into where we go next: back to the Railties gem.

1.6. [railties gem]/lib/rails/commands.rb

This file contains:

[railties gem]/lib/rails/commands.rb

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

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

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

require 'rails/commands/commands_tasks'

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

This is the file where those nice little shortcuts like `rails g` and `rails s` are defined. What this code will do is pick the first argument from `ARGV` with a quick `shift`, and then check to see if that argument has a key in the `aliases` method. If it doesn't, then it's assumed to be the command itself. So it doesn't matter if you run "rails server" or "rails s" after this point, because command will always be "server".

The next line requires `rails/commands/commands_tasks`. This is used to run the command that we've given it by initializing a new instance of `Rails::CommandsTasks`, and then calling `run_command!` on that, passing in the `command` variable as the only argument to this method. Let's look at what this does.

1.7. [railties gem]/lib/rails/commands_tasks.rb

This is another large file. We're only interested in the `server` command, so let's look first at what the `run_command!` method does, and then the `server` command.

[railties gem]/lib/rails/commands_tasks.rb

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

...

def run_command!(command)
  command = parse_command(command)

  if COMMAND_WHITELIST.include?(command)
    send(command)
  else
    run_rake_task(command)
  end
end

...

def parse_command(command)
  case command
  when '--version', '-v'
    'version'
  when '--help', '-h'
    'help'
  else
    command
  end
end
```

The `run_command` method relies on a few other parts of this file, which are included in the above example. First, the `parse_command` method is called. This standardises the command that was passed in if that command was one of `'--version'`, `'-v'`, `'--help'` or `'-h'`. This is similar to the aliasing that was done previously in `rails/commands.rb`.

Next, the `COMMAND_WHITELIST` constant is checked. If the command isn't listed there, then the `run_rake_task` method is called, which will attempt to run a rake task with the same name as the command. This is the behaviour that was introduced in Rails 5 which will allow you to run either `rake db:migrate` or `rails db:migrate`.

We're on the path of the `server` command, and what `run_command!` does after checking if the command is on the whitelist is it calls `send` to call a method that matches that command's name. In the case of our sleuthing, this is the `server` method:

[railties gem]/lib/rails/commands/commands_tasks.rb

```
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
```

The `server` command first calls `set_application_directory!`, and then comment above that method does a great job at explaining what that method is for:

[railties gem]/lib/rails/commands/commands_tasks.rb

```
private

# Change to the application's path if there is no config.ru file in current directory.
# This allows us to run `rails server` from other directories, but still get
# the main config.ru and properly set the tmp directory.
def set_application_directory!
  Dir.chdir(File.expand_path('../../', APP_PATH)) unless
File.exist?(File.expand_path("config.ru"))
end
```

Next, the `require_command!` method is called, which is simply this:

[railties gem]/lib/rails/commands/commands_tasks.rb

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

ALl that's doing is requiring `rails/commands/server.rb`, which defines the `Rails::Server` constant that we're about to use. Let's look at that file now.

1.8. [railties gem]/lib/rails/commands/server.rb

The `server` method that we just looked at starts the Rails server by running this code:

[railties gem]/lib/rails/commands/server.rb

```
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
```

Since this is initializing an instance of the `Rails::Server` class, let's look at the `initialize` method from `Rails::Server` first:

[railties gem]/lib/rails/commands/server.rb

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

To understand what this method is about to do, we need to take a *huge* step back and look at how the `Rails::Server` class is defined:

[railties gem]/lib/rails/commands/server.rb

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

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

There is a lot of complexity hidden in these lines. In order to understand what's about to happen we'll need to look at what the `action_dispatch` and `rails` requires are doing. It turns out that they do quite a lot.