Extracted from:

Crafting Rails 4 Applications

Expert Practices for Everyday Rails Development

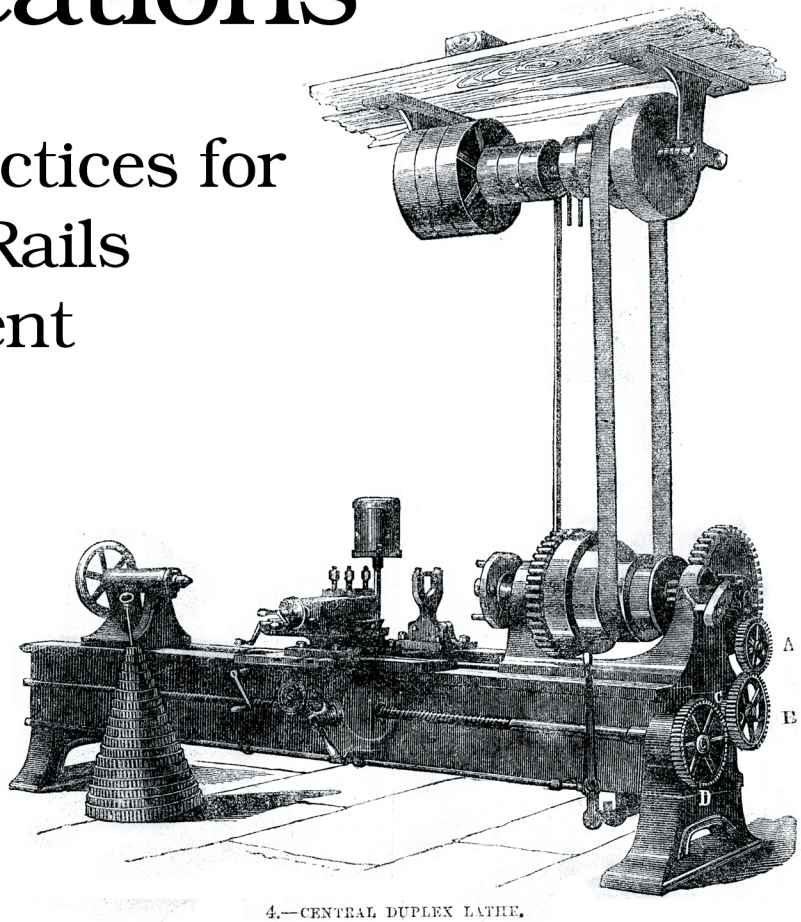
The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Crafting Rails 4 Applications

Expert Practices for Everyday Rails Development



4.—CENTRAL DUPLEX LATHE.

José Valim

edited by Brian P. Hogan

The Facets of Ruby Series

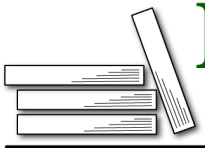
Crafting Rails 4 Applications

Expert Practices for Everyday Rails Development

Jose Valim

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina



Pragmatic Bookshelf

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *http://pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-937785-55-0
Encoded using the finest acid-free high-entropy binary digits.
Book version: B2.0—June 21, 2013

In the previous chapters, we analyzed the Rails rendering stack inside and out. You learned that when a request reaches a controller, the controller gathers the required information to render a template. The template is retrieved from one of the resolvers, compiled, rendered fully and embedded in a layout. At the end of this process, you have a Ruby string representing this template. The string is set as the HTTP response and sent back to the client.

This approach works fine for the vast majority of applications. However, in some particular cases, we need to send our response in smaller chunks. Sometimes, those smaller chunks may be infinite; we keep on sending chunks to the client until the connection between the server and the client is closed.

Whenever we send a response in chunks, we say the server is *streaming* data to the client. Since Rails was built with the more traditional request-response scenario in mind, streaming support was added and improved in Rails over time and we are going to explore it in this chapter.

To explore how streaming works, let's write a Rails plugin that sends data to the browser whenever one of our stylesheets changes. The browser will use this information to reload the current page stylesheets, allowing developers to see changes in the HTML page as they modify their assets file, without a need to manually refresh the page in the web browser.

Since this plugin is going to have its own controller, assets, routes and more, we will rely on the power provided by Rails engines so we can add those functionalities as if they were part of a Rails application, but then bundle it in a gem to share across different projects.

5.1 Extending Rails with Engines

Rails engines allow our plugin to have its own controllers, models, helpers, views, assets, and routes, just like in a regular Rails application. Let's generate a plugin called live_assets using the Rails plugin generator. But this time we'll pass the --full flag, which will generate directories for models, controllers, routes and more:

```
$ rails plugin new live_assets --full
```

In addition to the files the generator normally creates for us, the --full flag also generates these files:

- an app directory with controllers, models and others
- a config/routes.rb file for routes
- a lib/live_assets/engine.rb file declaring our engine

- an empty test/integration/navigation_test.rb file that we can add our integration tests to

The most important file here is lib/live_assets/engine.rb, so let's take a closer look at it:

live_assets/1_live/lib/live_assets/engine.rb
```
module LiveAssets
  class Engine < ::Rails::Engine
  end
end
```

In order to create an engine, we need to inherit from Rails::Engine and ensure our new engine is loaded as soon as possible. The generator we ran already did this for us by placing this line in lib/live_assets.rb:

live_assets/1_live/lib/live_assets.rb
```
require "live_assets/engine"

module LiveAssets
end
```

Creating a Rails::Engine is quite similar to creating a Rails::Railtie. This is because a Rails::Engine is nothing more than a Rails::Railtie with some default initializers and the Paths API, which we'll see next.

Paths

A Rails::Engine does not have hard-coded paths. This means we are not required to place our models or controllers in app/; we can put them anywhere we choose. For instance, we can configure our engine to load our controllers from lib/controllers instead of app/controllers as follows:

```
module LiveAssets
  class Engine < Rails::Engine
    paths["app/controllers"] = ["lib/controllers"]
  end
end
```

We can also have Rails load our controllers from both app/controllers and lib/controllers:

```
module LiveAssets
  class Engine < Rails::Engine
    paths["app/controllers"] << "lib/controllers"
  end
end
```

Those paths have the same semantics as in a Rails application: if you have a controller named LiveAssetsController inside app/controllers/live_assets_controller.rb or

lib/controllers/live_assets_controller.rb, the controller will be loaded automatically when you need it, it doesn't need to be explicitly required.

For now, we are going to follow the conventional path and stick our controllers at app/controllers, so don't apply the previous changes. We can check all customizable paths for an engine by inspecting the Rails source code:

rails/railties/lib/rails/engine/configuration.rb

```
def paths
  @paths ||= begin
    paths = Rails::Paths::Root.new(@root)

    paths.add "app",                 eager_load: true, glob: "*"
    paths.add "app/assets",          glob: "*"
    paths.add "app/controllers",     eager_load: true
    paths.add "app/helpers",         eager_load: true
    paths.add "app/models",          eager_load: true
    paths.add "app/mailers",         eager_load: true
    paths.add "app/views"

    paths.add "app/controllers/concerns", eager_load: true
    paths.add "app/models/concerns",      eager_load: true

    paths.add "lib",                 load_path: true
    paths.add "lib/assets",          glob: "*"
    paths.add "lib/tasks",           glob: "**/*.rake"

    paths.add "config"
    paths.add "config/environments", glob: "#{Rails.env}.rb"
    paths.add "config/initializers", glob: "**/*.rb"
    paths.add "config/locales",      glob: "*.{rb,yml}"
    paths.add "config/routes.rb"

    paths.add "db"
    paths.add "db/migrate"
    paths.add "db/seeds.rb"

    paths.add "vendor",              load_path: true
    paths.add "vendor/assets",       glob: "*"

    paths
  end
end
```

The previous snippet shows the engine also specifies which paths should be eager loaded and which ones shuold not, besides also listing paths to locales, migrations and more. However, declaring a path is not enough, something has to be done with such paths at some point after all. That's where initializers come in.

Initializers

An engine has several initializers that are responsible for booting the engine. These initializers are more low-level and should not be confused with the ones available inside your application's config/initializers. Let's explore an example:

rails/railties/lib/rails/engine.rb

```
initializer :add_view_paths do
  views = paths["app/views"].existent
  unless views.empty?
    ActiveSupport.on_load(:action_controller){ prepend_view_path(views) }
    ActiveSupport.on_load(:action_mailer){ prepend_view_path(views) }
  end
end
```

This initializer is responsible for adding our engine views, usually defined in app/views, to ActionController::Base and ActionMailer::Base as soon as they are loaded, allowing a Rails application to use the templates defined in an engine. To see all initializers defined in a Rails::Engine, we can start a new Rails console under test/dummy with rails console and type the following:

```
Rails::Engine.initializers.map(&:name) # =>
  [:set_load_path, :set_autoload_paths, :add_routing_paths,
   :add_locales, :add_view_paths, :load_environment_config,
   :append_assets_path, :prepend_helpers_path,
   :load_config_initializers, :engines_blank_point]
```

Working with an engine is pretty much the same as working with a Rails application. Since we know how to build applications, implementing our streaming plugin should feel familiar.

5.2 Live Streaming

In order to show how streaming works, let's create a controller called LiveAssetsController at app/controllers/live_assets_controller.rb that includes the ActionController::Live functionality and streams "hello world" continuously:

live_assets/1_live/app/controllers/live_assets_controller.rb

```
class LiveAssetsController < ActionController::Base
  include ActionController::Live

  def hello
    while true
      response.stream.write "Hello World\n"
      sleep 1
    end
  rescue IOError
    response.stream.close
  end
end
```

Our controller provides an action named hello() that streams Hello World every second. If, for any reason, the connection between the server and the client drops, response.stream.write will fail with IOError which we need to rescue and then properly close our stream.

We also need a route for this endpoint:

live_assets/1_live/config/routes.rb

```
Rails.application.routes.draw do
  get "/live_assets/:action", to: "live_assets"
end
```

We are almost ready to try out our streaming endpoint. However, since a Rails engine cannot run on its own, we need to start it via the application in test/dummy. Furthermore, the streaming functionality doesn't work in WEBrick, the server that ships with Ruby which is used by default by Rails, since WEBrick would first buffer our response before sending it to the client and, given our response is infinite, we would never see anything at all. For this reason, let's add Puma[1] as a development dependency to our gemspec:

live_assets/1_live/live_assets.gemspec

```
s.add_development_dependency "puma"
```

And finally go into the test/dummy directory and run rails s. Rails now starts Puma instead of WEBrick:

```
=> Booting Puma
=> Rails 4.0.0 application starting in development on http://0.0.0.0:3000
=> Call with -d to detach
=> Ctrl-C to shutdown server
```

Most browsers will also try to buffer the streaming response and it may take a while before they decide to show us anything. So, to test that our streaming endpoint really works, we'll use cURL[2] which works via the command line. Let's give curl a try:

```
$ curl -v localhost:3000/live_assets/hello
> GET /live_assets/hello HTTP/1.1
> User-Agent: curl/7.24.0 (x86_64-apple-darwin12.0)
> Host: localhost:3000
> Accept: */*
>
< HTTP/1.1 200 OK
< X-Frame-Options: SAMEORIGIN
< X-XSS-Protection: 1; mode=block
< X-Content-Type-Options: nosniff
```

1. http://puma.io/
2. http://curl.haxx.se/

```
< X-UA-Compatible: chrome=1
< Cache-Control: no-cache
< Content-Type: text/html; charset=utf-8
< X-Request-Id: f21f8c0d-d496-4bfa-944c-cd01b44b87ee
< X-Runtime: 0.003120
< Transfer-Encoding: chunked
<
Hello World
Hello World
```

Each second, you will see a new "Hello World" line appear on the screen over and over. This means our streaming end point is working. Hit CTRL+C in your keyboard to stop it as we are ready to move to more complex examples!