Dynamic Rails Error Pages

Build custom 404 and 500 error pages utilizing ERB and your existing layouts and stylesheets.

Published February 12, 2015

Revised February 29, 2016: Updated routes (using :match) so that error pages work for all types of requests, not just GET.

Normally, 404 and 500 error pages are static HTML files that live in the `public` directory of a Rails application. These are boring, minimally-styled pages that don't get the same treatment as the rest of the app. This tutorial shows you how to move error pages into your Rails app as dynamic views that benefit from application styles, layouts, and view helpers.

tl;dr – **jump to the Rails code** and **the Capistrano bonus tip**

WHY ARE DYNAMIC ERROR PAGES PATICULARLY HANDY IN RAILS 4?

Starting with Rails 4, the production asset pipeline no longer generates filenames without cache-busters. This means that referencing `/assets/application.css` in your static `public/404.html` page won't work in a Rails 4 app! The file will not exist in the production environment. The only way to reliably reference your application stylesheet is to use the `stylesheet_link_tag` helper.

But error pages are static HTML pages; they can't use helpers, right? If you want nice-looking error pages in Rails 4, here are your options:

Option 1: No external styles. Don't reference your application stylesheet at all. Instead, use simple, static error pages with the necessary minimal CSS copied and pasted into each HTML file. This is the solution that ships with Rails.

- Works for simple apps that don't need custom-branded error pages.

Option 2: Monkey patch. Use static error pages and point to `/assets/application.css` for styling. Then, **monkey-patch Rails** to restore the pre-Rails 4 behavior so that the asset pipeline generates non-cache-busted filenames in production. Make sure not to send far-future expires headers for these files!

- Easiest option for migrating an existing app to Rails 4.

Option 3: Dynamic. Use dynamic view templates (ERB) for error pages, and take advantage of the `stylesheet_link_tag` helper to get the right cache-busted filename. Error pages can use your application styles. Be careful, though: if your Rails app is down, your error pages can't be accessed.

- Most flexible option. This is the solution I describe below.

OK, so you're ready to set up dynamic error pages in a Rails 4 app? Here's how to do it.

1 GENERATE AN ERRORS CONTROLLER AND VIEWS

```
rails generate controller errors not_found internal_server_error
```

This creates `app/controllers/errors_controller.rb` with corresponding view templates in `app/views/errors/` for the not found (404) and internal server error (500) pages. (Ruby methods declared with `def` can't start with numbers, so use the spelled-out error names instead.)

2 SEND THE RIGHT STATUS CODES

```
class ErrorsController < ApplicationController
  def not_found
    render(:status => 404)
  end

  def internal_server_error
    render(:status => 500)
  end
end
```

Normally Rails sends an HTTP status of `200 OK` when it renders a view template, but in this case we want 404 or 500 to be sent according to error page being rendered. This

requires a slight tweak to the `errors_controller.rb` that Rails generates. You don't need to specify the name of the template to render, because by convention it is the same as the action name.

3 CONFIGURE THE ROUTES

```
match "/404", :to => "errors#not_found", :via => :all
match "/500", :to => "errors#internal_server_error", :via => :all
```

Rails expects the error pages to be served from `/<error code>`. Adding these simple routes in `config/routes.rb` connects those requests to the appropriate actions of the errors controller. Using `match ... :via => :all` allows the error pages to be displayed for any type of request (GET, POST, PUT, etc.).

4 TELL RAILS TO USE OUR ROUTES FOR ERROR HANDLING

```
config.exceptions_app = self.routes
```

Add this line to `config/application.rb`. This tells Rails to serve error pages from the Rails app itself (i.e. the routes we just set up), rather than using static error pages in `public/`.

5 DELETE THE STATIC PAGES

```
rm public/{404,500}.html
```

Speaking of which, we don't need those static error pages anymore.

6 STYLE THE ERROR VIEWS

By default the `not_found.html.erb` and `internal_server_error.html.erb` views will use the application layout defined in `app/views/layouts/application.html.erb`. Most likely your application layout already has the `stylesheet_link_tag(:application)` helper, so your error pages have access to all those loaded styles. No more inline CSS, yay!

But it gets better: Since these error pages are just like any other Rails views, you can make use of a custom layout to DRY up the markup. Following Rails conventions, just create `app/views/layouts/errors.html.erb` and that template will automatically be applied to all error pages. Sweet.

7 TEST IT

Since the error pages are normal routes, you can test them in your browser by going directly to **http://localhost:3000/404** or **http://localhost:3000/500**. Use the resource inspector in the browser's developer console to double-check that the correct HTTP status codes are being sent.

In practice, your users won't be going to these pages directly. Instead, you'll want to make sure these pages render when an error occurs. To test this behavior locally, change this setting in `config/environments/development.rb`:

```
config.consider_all_requests_local = false
```

Setting this option to `false` tells Rails to show error pages, rather than the stack traces it normally shows in development mode. Restart the Rails server after making this change.

Now trigger an error, either by going to a non-existent path, or drop a `raise "boom!"` statement in your app somewhere to cause an exception. The dynamic error pages should be displayed.

DRAWBACKS

Dynamic error pages let us use the power of the Rails view layer, but this has its own drawbacks.

If the error page has errors. Syntax errors, database outages, or other catastrophes can lead to dynamic error pages that themselves fail to render. If this happens, not only can't users interact with your app, they won't be able to see your fancy error page!

Luckily Rails is smart enough to recognize this situation an avoid an infinite loop. As a last resort, Rails will display a simple plaintext error message:

> *500 Internal Server Error*

> *If you are the administrator of this website, then please read this web application's log file and/or the web server's log file to find out what went wrong.*

If Rails has completely crashed. When a Rails application is proxied by a web server like Nginx, the web server can be configured to serve static files from `public/`. Theoretically, if your Rails application completely crashed, Nginx could still serve a static error page, like `public/500.html`.

But with dynamic error pages this is not possible. By definition, Rails has to be up and running in order for those error pages to be displayed. You'll need a static error page for this scenario.

So let's generate one!

BONUS: AUTO-GENERATING A STATIC ERROR PAGE WITH CAPISTRANO

Assuming you deploy using Capistrano 3, you can use Capistrano to also generate a static `public/500.html` page whenever your application is deployed. With proper Nginix configuration, this error page can be served even in the unfortunate scenario when your Rails app is completely offline.

1 DEFINE A CAPISTRANO TASK

```
task :generate_500_html do
  on roles(:web) do |host|
    public_500_html = File.join(release_path, "public/500.html")
    execute :curl, "-k", "https://#{host.hostname}/500", "> #{public_500_html}"
  end
end
after "deploy:published", :generate_500_html
```

This instructs Capistrano to request the `/500` route of your application and save the resulting HTML to `public/500.html`. This happens on every successful deploy. Now your app has a static 500 error page that looks just like your dymamic one, automatically!

2 CONFIGURE NGINX

```
error_page 500 502 503 504 /500.html;
location = /500.html {
```

```
  root /path/to/your/app/public;
}
```

When added to the `server {…}` block of your Nginx site configuration, this tells Nginx to serve the `public/500.html` file whenever it gets a 5xx error from Rails. More importantly, this will also be triggered if Rails is completely offline and the upstream connection from Nginx to Rails fails.

3 TEST IT

After deploying these changes, test it out by stopping your Rails app (e.g. stopping Unicorn). Now try accessing the app in a browser: you should still see the custom 500 error page, thanks to Nginx. Nice!

IS IT WORTH IT?

Considering the effort it takes to set up dynamic error pages, including covering all the edge cases, is it worth it? I think so. Here's why I plan on using dynamic error pages for my Rails apps:

- Moving error pages into my `app/views` alongside the rest of my application views means it's easier to keep their design in sync with everything else.
- Helpers and especially layouts are a godsend for cranking out error pages that are styled consistently and match the rest of my app.
- I can use my application stylesheet in error pages without monkey-patching!

I'd love to hear your questions or comments on this article: just mention me on Twitter or drop me an email. Thanks! —*m*

MORE OF MY ARTICLES YOU MAY ENJOY

Setting up Sublime Text 3 for Rails Development

I've been a satisfied Sublime user for the past three years, using it primarily for Rails development. Here are the packages, preferences, and tips I recommend for getting the most out of this excellent editor.

Rails OS X Setup Guide

Installing an rbenv-based Rails stack on El Capitan, Yosemite, or Mavericks

Build and Deploy a Rails VPS, Part 1

Start by provisioning an Ubuntu 14.04 VPS, then install Ruby with rbenv.

ActiveRecord Strict Validations, Minitest, and Shoulda

Are you using thoughtbot's Shoulda gems with Minitest? How about strict validations in ActiveRecord? Here's why I think these are great things to add to your Rails backpack of tools, and how to set them up.

Use Minitest for Your Next Rails Project

Minitest is a fast, easy to read alternative to RSpec for writing Rails tests, but it can be confusing at first. Here's how I set up Minitest with Rails, and the gotchas I encountered along the way.

All articles