José D. Gómez R.

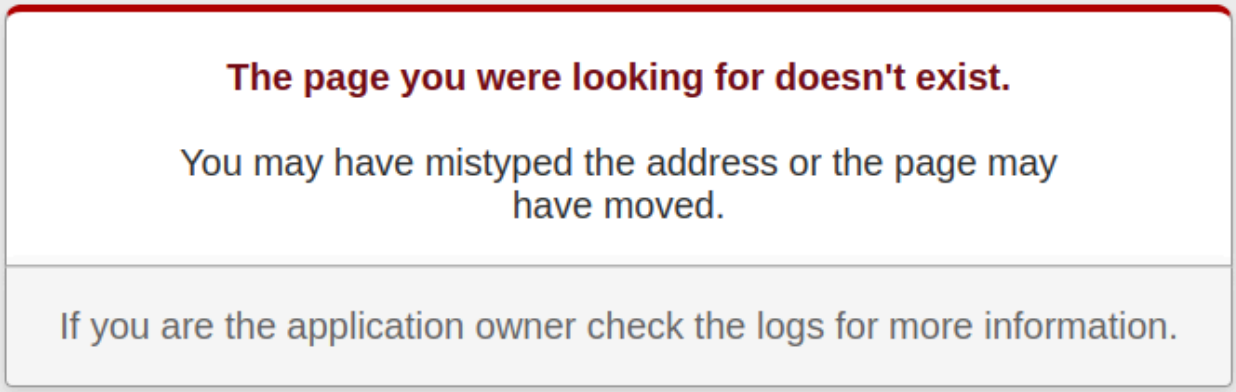
My personal bin to put stuff around, it will definitely change… or not? who knows

Adventures with rails/sprockets -> rails/propshaft

📅: 2023-09-15 – 🏷️: blog, ruby-on-rails, propshaft, assets

Join me in this review of my day debugging an asset pipeline rewritten from Sprockets to Propshaft for the 3rd time.

So my day starts the following screen:



And a colleague saying:

> I'm stuck with the asset pipeline, do you have time to have a look?
>
> — @colleague, sick of dealing with the asset pipeline

How does an asset pipeline works? [roughly*]

A lot of what's below here is my own understanding of how things work, to get a full overview you can always read the rails guide on assets pipeline

It all starts with sources. Asset sources are artifacts that may need to be processed in some way to result in a CSS File, JS File, SVG, whatever a browser can understand/render/execute.

The easiest "asset pipeline" we can think of is just: not having to process anything. Just have your application code render locations of your assets that the webserver can serve and boom, call it a day.

But as the world progresses, Front-end technologies get progressively complicated and involve a bunch of moving parts.

In the current day an asset pipeline usually has two sides:

- A Javascript Transpiler[1]: A piece of software that will transform isolated pieces of code into one, or many, big javascript entrypoint file to be served to the browser.

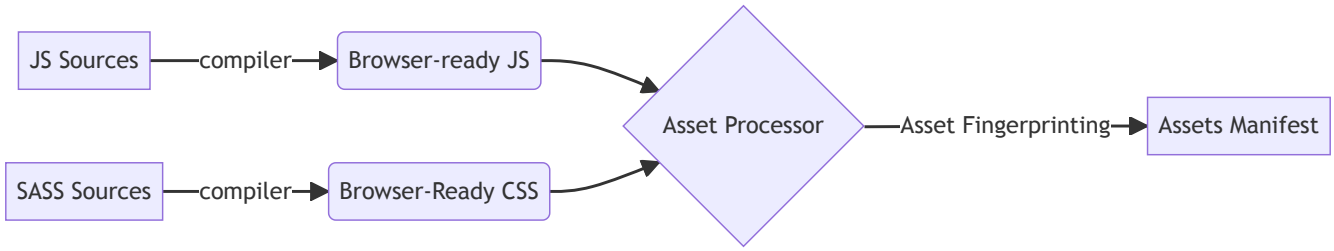
  Note the use of "pieces of code" because JS can also be written in… something else! Like in the old days when people used to write CoffeeScript. Although now Typescript the go-to choice (depending on whom you ask of course).

  The main goal is to have JS files that the browser can understand.

- A CSS Processor: Much like the JS Transpiler, this converts sources into CSS files. A popular choice is to use `sass`. But as usual, it's a matter of taste.

  The main goal is to have CSS files that the browser can understand.

A quick summary here:



After the sources have been converted into understandable artifacts by the browser, then comes another pretty critical piece of the process we need to understand: The Assets Manifest.

A tale of the before times

You see, in *ze good ol' days*, there was a folder where you dumped all your CSS/JS, make sure your webserver can serve it and referred to it in the HTML. Done deal, *izi-peezy*.

Nowadays there's many many many moving parts involved into this. Not only processing and serving them, but nowadays there's a bunch of caches involved (browser cache, CDN cache, etc.). We'll focus on a technique that's enabled by default: Asset Fingerprinting

To be short in this long-ass journey, Asset Fingerprinting is the process of: **Generating a unique filename per asset that changes if the assets contents also changes.**

That usually means: calculate a digest of the contents, append it to the filename and keep track of it somehow.

The problem now is that you have a bunch of random filenames like:

```
Poppins-Medium-85f225b62f54abf64418686830908d2e.ttf
Poppins-Medium-564476dfea32cf397f470e5dd607ed42.ttf
Poppins-Medium-3a41c57c68c387a88540df1125eb7f01.ttf
```

That's not quite intuitive to write in source code…

To solve this issue, a manifest is usually employed. This asset manifest is in essence a dictionary: You look for a source name, and you get the digest-appended name.

```
[Human-readable-name] -> [Name-with-Digest]
Poppins-Medium.ttf -> Poppins-Medium-3a41c57c68c387a88540df1125eb7f01.ttf
```

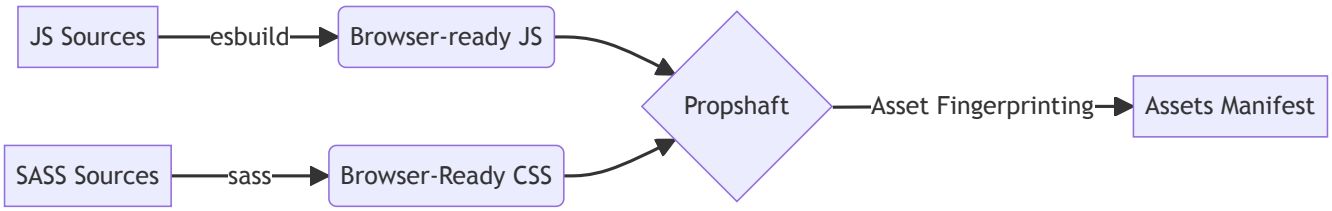
Even though there are 3 versions of `Poppins-Medium.ttf`, only one is kept in the manifest, that we will consider as the appropriate one. How this is computed is a bit besides the point of this journey.

Translating all these details to rails terms:

- JS: `esbuild` collects all JS Module-style entrypoints into a ready-to-execute single-file `js` entrypoints.
- CSS: `sass` collects all `sass` / `scss` files into ready-to-render `css` files.
- Asset Processor: rails/propshaft.

JS & CSS compilation run with `yarn` outside of rails. They output things to `app/assets/builds`.

Then rails/propshaft will process all foreign references and cross-match them with the assets it finds in the "load paths" (spoiler alert: `Rails.application.config.assets.paths`), and the result



Coming back to the rails problem at hand, let's dig deeper into the rails issue, now that we understand a bit the whole asset pipeline shenanigans.

Some debugging later, we get this screen:

Propshaft::MissingAssetError in Categories#index

Showing [redacted]/app/views/layouts/application.html.haml where line #10 raised:

The asset 'application.css' was not found in the load path.

Extracted source (around line #10):

```
8     = csrf_meta_tags
9     = csp_meta_tag
10    = stylesheet_link_tag "application", "data-turbo-track": "reload"
11    = javascript_include_tag "application", "data-turbo-track": "reload", defer: true
12  %body
13    = yield
```

Rails.root: [redacted]

Application Trace | Framework Trace | Full Trace
app/views/layouts/application.html.haml:10

Request

Parameters:

None

Toggle session dump

Toggle env dump

Response

Headers:

None

Now we get a "prettier" error, we can see a "cause".

> The asset 'application.css' was not found in the load path.

Aha, we're getting somewhere, so `application.css` is not found by the rails application… How does rails *"finds"* assets then?

From the app backtrace we can see:

```
stylesheet_link_tag "application", "data-turbo-track": "reload"
```

`stylesheet_link_tag` must be looking up `application.css` somewhere, and it doesn't find it.

My buddy had cleared *all* possible customizations from the ruby code, it's all defaults in a desperate effort to make this work.

What are the defaults? Time to source code!

After a quick visit to rails/propshaft's railtie:

```
Rails::Engine.initializer "propshaft.append_assets_path", group: :all do
  app.config.assets.paths.unshift(*paths["vendor/assets"].existent_directo
  app.config.assets.paths.unshift(*paths["lib/assets"].existent_directorie
  app.config.assets.paths.unshift(*paths["app/assets"].existent_directorie

  app.config.assets.paths = app.config.assets.paths.without(Array(app.con
end
```

The default paths are `vendor/assets`, `lib/assets`, `app/assets` and all its descendants. And surely enough, we found the issue.

My buddy just tried to make it work like the Sprocket times, and made changes to the `package.json` that were obscure to us at the time, but it looked like:

```
- "build:css": "sass ./app/assets/stylesheets/application.scss:./app/asset
+ "build:css": "sass ./app/assets/stylesheets/application.scss:./public/as
-     "build": "esbuild app/javascript/*.* --bundle --sourcemap --outdir=a
+ "build": "esbuild app/javascript/*.* --bundle --sourcemap --outdir=publ
```

By outputting to `./public/assets` instead of the default-known path: `/app/assets/`, now propshaft couldn't find the resulting CSS file from the `sass` build.

So we decided to allow propshaft read the `public/assets` directory, where the `yarn` tasks were outputting the results…

```
# in the initializer
Rails.application.config.assets.paths << Rails.root.join("public/assets")
```

Foreshadowing: that was a bad idea, *very bad idea*.

We didn't see the original error anymore, the application booted successfully and rendered… But the output of the `assets:precompile` task started showing some odd stuff.

```
Writing rails_admin/custom/mixins-911875cdfd5191767b7b7ef80e1d45e25a87d32e

[...]

Errno::ENAMETOOLONG: File name too long @ rb_sysopen - [...snip...]/public
```

That's *way* too long for a useful filename, something is off. Let's parse that long filename:

```
hash1=911875cdfd5191767b7b7ef80e1d45e25a87d32e
hash2=cb22d660bb3d9a6878dca29ed218770bf64b35e7

rails_admin/custom/mixins-[hash1]-[hash2]-[hash2]-[hash2]-[hash2]-[hash2].
```

Member when times were simpler? Asset fingerprinting explanation

Asset fingerprinting adds a digest to the end of the basename of the file. And because we added `public/assets` to the lookup paths of propshaft, **every asset there** is processed, digest is calculated and appended to the filename.

Our “solution” worked, it worked… *too well*. So that’s why the `app/assets/builds` directory exists! It’s a “staging area” of the compiled sources yet-to-become-assets for propshaft to collect and place in the output assets directory `public/assets` and build a manifest afterwards.

But `app/assets/builds` is such a weird choice for this, `app/` is meant to be source files, now it has compilation results? We weren’t that amazed at this fact…

We inclined to use `tmp/assets` to save the result of the `yarn` tasks, that way won’t:

- be in `app/`, compilation results are not source code.
- not collide with asset fingerprinting and generate long-long-long filenames.

And surely enough, after:

```
# in the initializer
Rails.application.config.assets.paths << Rails.root.join("tmp/assets")
```

Compilation worked, no digest of a digest of a digest file paths but… it wasn’t quite done.

After inspecting the resulting CSS we noticed that blocks of `scss` were working just fine, like:

```
@font-face {
  font-family: "Poppins";
  src: url("/Poppins-Medium.ttf") format("truetype");
  font-weight: 500;
  font-style: normal;
}
```

Was transpiled to:

```
@font-face {
  font-family: "Poppins";
  src: url("/assets/Poppins-Medium-95f2263e7bdb88c3d120ee2cd60394a2c694f7a
  font-weight: 500;
  font-style: normal;
}
```

But…

```
// [...] snipp snipp [...]
$bootstrap-icons-font-src: url("bootstrap-icons/font/fonts/bootstrap-icons
@import 'bootstrap-icons/font/bootstrap-icons';
// [...] snipp snipp [...]
```

Was transpiled to:

```
@font-face {
  font-display: block;
  font-family: "bootstrap-icons";
  src: url("bootstrap-icons/font/fonts/bootstrap-icons.woff2") format("wo
}
```

It’s clear that the `scss -> css` process works fenomenaly, variable was correcly interpolated. But the asset processor should’ve picked that location up and translated the `url`, so it has the `[digest]` just before the extension.

Furthermore, the logs also show:

```
Unable to resolve 'bootstrap-icons/font/fonts/bootstrap-icons.woff' for m
```

That’s an indication there, when CSS `url` function refers to something not existing it will be kept as-is and log an error in the task output.

Now the question is: how to make `propshaft` aware of that path?

You guessed it: More paths in the initializer.

The `$bootstrap-icons-font-src` says the `bootstrap-icons` fonts are in `bootstrap-icons/font/fonts`, that's relative to `node_modules`, so in theory… font's *should exist* in `node_modules/bootstrap-icons/font/fonts`.

```
ls -1 node_modules/bootstrap-icons/font/fonts
bootstrap-icons.woff
bootstrap-icons.woff2
```

Cool, now I guess we can just add that path, and propshaft knows where to find things.

```
Rails.application.config.assets.paths << Rails.root.join('node_modules/boo
```

And surely enough, it shows up in the logs:

```
Writing bootstrap-icons-f8b71410af9b67092d6fc3f72d4756b9e9a2cf7a.woff
```

But the "unable to resolve" message below persists:

```
Unable to resolve 'bootstrap-icons/font/fonts/bootstrap-icons.woff' for m
```

Time to dig deeper, what's that `Rails.application.config.assets.paths` for? And as usual with rails… source code is the way to go.

We want to know how's `bootstrap-icons` is being referred in the manifest. After spending a significant amount of time going around the source code, here are the important facts to get to the bottom of this:

- `Rails.application.config.assets.paths` ends up being used and distributed through the project in rails/propshaft's `Propshaft::Assembly`.
- The assets manifest is not longer configurable (as it was in sprockets).
- `Rails.application.config.assets.output_path` when not defined defaults to: `public/assets`, which is important because `Propshaft::Processor::MANIFEST_FILENAME` is relative to that path.

With the manifest at hand, let's see what we can find there:

```
jq < public/assets/.manifest.json | grep 'bootstrap-icons'
  "bootstrap-icons.woff": "bootstrap-icons-f8b71410af9b67092d6fc3f72d4756b
  "bootstrap-icons.woff2": "bootstrap-icons-a76c0fb01876e9f9abd10fe5c48d6c
```

Aha! We're referring to them with a long path `bootstrap-icons/font/fonts/bootstrap-icons.woff2`, but they're actually reported in the manifest as bare files: `bootstrap-icons.woff2`.

This means all assets referred are relative to any directory specified in: `Rails.application.config.assets.paths`. Probably collisions are settled by order in the array (first path matched is returned).

Let's change that path line then to:

```
Rails.application.config.assets.paths << Rails.root.join('node_modules')
```

It should not harm… right…. RIGHT??? Spoiler alert: it does.

I'm not even going to put the full output, but suffice to say, it found shit:

```
jq < public/assets/.manifest.json | grep 'bootstrap-icons' | wc -l
1962
```

```
jq < public/assets/.manifest.json | grep 'bootstrap-icons.css'
  "bootstrap-icons/font/bootstrap-icons.css": "bootstrap-icons/font/bootst
```

The asset pipeline captured pretty much anything inside `node_modules` directory, and its descendants, including but not limited to these funny examples:

```
jq < public/assets/.manifest.json | egrep -v '(css|js|svg)": '
# symlinks
  ".bin/esbuild": ".bin/esbuild",
  ".bin/sass": ".bin/sass",
# json files
  "@babel/runtime/helpers/esm/package.json": "@babel/runtime/helpers/esm/p
# licenses
  "@babel/runtime/LICENSE": "@babel/runtime/LICENSE",
# typescript files
  "rails_admin/node_modules/@popperjs/core/lib/utils/mergeByName.d.ts": "r
# flow files
  "rails_admin/node_modules/@popperjs/core/lib/utils/mergeByName.js.flow":
```

There’s no pretty solution to this, as usual. The main issue is that we’re working on a path-based logic. Anything in that path will be included or excluded. You don’t get file-specific include/exclude filters.

With that said, we settled on:

```
Rails.application.config.assets.paths << Rails.root.join("node_modules/boo
```

With it, the resulting manifest looks like:

```
jq < public/assets/.manifest.json | grep 'bootstrap-icons'
  "fonts/bootstrap-icons.woff": "fonts/bootstrap-icons-f8b71410af9b67092d6
  "fonts/bootstrap-icons.woff2": "fonts/bootstrap-icons-a76c0fb01876e9f9ab
  "bootstrap-icons.json": "bootstrap-icons-be63c3b0cc95f7496368e195843792a
  "bootstrap-icons.css": "bootstrap-icons-bcd28b803d6d96d863de260eef795aa7
  "bootstrap-icons.scss": "bootstrap-icons-abeb873a2c816f4ff54d00148e86519
```

It's… good enough tbh, not bare files, not quite as namespaced as I'd love to but we gotta work under with the constraints.

The final step would be checking that `propshaft` actually replaced the wrong `url`.

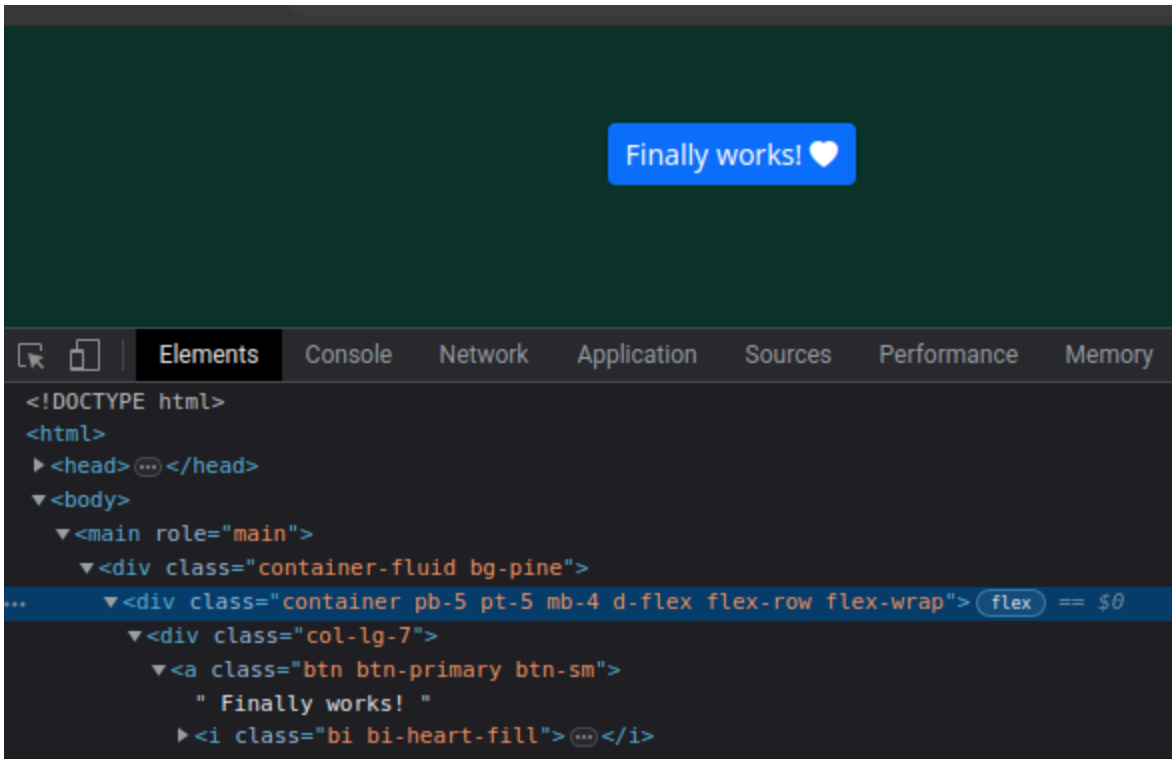
So we adjust our snippet to:

```
$bootstrap-icons-font-src: url("fonts/bootstrap-icons.woff2") format("woff
@import 'bootstrap-icons/font/bootstrap-icons';
```

And we get… 🥁🥁🥁

```
@font-face {
  font-display: block;
  font-family: "bootstrap-icons";
  src: url("/assets/fonts/bootstrap-icons-a76c0fb01876e9f9abd10fe5c48d6da2
}
```

FINALLY works!



And that was only *ONE* day of debugging an asset pipeline, I hope I don't have to jump into this again tbh.

1. Nobody likes the term *Transpiler* anymore, to me: if the output of the compiler still can't be used by the end-result we're *transpiling* and not *compiling*. ↩