Managing Front-End Assets

The Rails Asset Pipeline has given us the opportunity to manage files other than Ruby files as first-class citizens. Personally, I don't quite agree that they are first-class citizens, but perhaps well-treated second-class citizens. That being said, I've developed a set of practices over time that allows us to to enforce more Rails-isms in our use of the Asset Pipeline.

In each Rails app, we have an assets folder with the following subfolders:

- images
- javascripts
- stylesheets

In the views folder, we have subfolders that follow what controllers we have, and a layout folder, where we store layouts.

If we were to unify the way we manage views with the way we manage assets, then we would easily know where each line of CSS or JS code should reside. This guide helps you to accomplish that.

Separate assets for each layout

We have an application layout by default when we create new rails app, so let's start by creating 'application' in each subfolder in assets (images, javascripts, stylesheets). Some applications have multiple layouts, so this is to future proof our applications' assets by giving them a nice home.

In your respective manifest files, you can add the following:

```
// application.css.scss, assuming that SASS is used
@import "./application/**/*";


// application.js
// require_tree ./application
```

By doing so, we now have a good home for each of our assets, and they will all automatically be included. If there are shared assets, like logos, we can create a shared folder in images, javascripts and stylesheets subfolders, and update the manifest files accordingly.

If you manage your codebase tightly, you have one folder for each manifest file (except for the shared folder.) When your code base expands to the point where you will have multiple layouts, it's just a matter of creating a new layout file, JS and CSS manifest files, and the relevant folders to house the actual code.

As an example, if you have an e-commerce application, then you might want to create a separate layout for the checkout pages. The reason for this is that the checkout layout is usually designed to be simpler, so as not to distract users from making the purchase, so we typically remove links from the header and footer, and leave only the checkout UI. This means more money for the e-commerce company, and an almost entirely different layout.

Maintain a folder for layout elements

To facilitate assets that we use for the entirety of the layout, we will create a specific folder for that. We put things like styles and javascript for headers and footers, which are going to be used as part of all the pages that we render.

These elements are treated as different from reusable elements, in that while they are reused across all pages, they are always in the same exact location. They are more fixed elements, than they are reusable. Having a folder specifically to house these elements are useful.

I could spell it out step by step, but it's easier if I just show you, you should now have an assets folder that looks like this:

- images
  - application
    - layout
- javascripts
  - application
    - layout
      - base.js.coffee
  - application.js
- stylesheets
  - application
    - layout
      - base.css.scss
      - footer.css.scss
      - header.css.scss
  - application.css.scss

For the next sections, we're going to cover how we can best manage CSS code.

Manage your stylesheets as you would your controllers

This may go against the grain of what you've learnt in CSS, which is how to use the cascading nature of CSS to re-use code, but don't make your CSS reusable. The problem is that programmers are really bad at deciding what code is reusable or not. You think you do, but you don't.

What usually happens is that you think that this piece of code is going to be reused so, they put it in a 'module', targeted by reusable class (or worse still an id), and that's fine and all, until this module needs to be used in another page, but just a little bit different, and so a variant on the module is added, and that is the start of a downward spiral of the CSS code, as myriads of unmanageable variants, now called hacks, appear. Some classes work in conjunction with other classes, some utility classes are applicable everywhere. You start to wonder where it all went south.

There is a better way.

At the start of this guide, I mentioned that we will be doing this in a Rails like fashion. I actually meant that quite literally, so let's start by creating a controllers folder in stylesheets/application. In this folder, we will respect the way we manage controllers and namespaces in our app/controllers folder. For example:

- stylesheets
  - + layout
  - controllers
    - admin
      - users.css.scss
      - products.css.scss
    - products.css.scss
  - application.css.scss

The bottom line is that each controller gets its own CSS file.

For each page that you style, you will add the code to its controller file, **even if it is repeating styles from another page**. The idea is to avoid premature optimization, by creating the redundant code first.

While you are doing this, bear in mind that you will not need to create additional classes, because we are trying to cut down on the number of 'reusable' classes. Let's have an example lest things get too abstract, consider the following code:

```
// stylesheets/controllers/admin/products.css.scss


body.admin .products .index {
  h2 { font-size: 16px; }
  // ...
}
```

A couple of things to note here, firstly, each page that we serve in the Rails app will need to add classes like above to the body element. This helps us contain the code here so that we can make changes as necessary knowing that we will not affect anything else outside of this controller and action. We can do this with a helper method, that we put in the layout file, like so:

```
// place this in your application.html.slim
body class="#{controller.controller_path.parameterize.gsub('-
', '_')} #{controller.action_name}"


// this produces HTML like
<body class="admin_users show">
...
</body>


// allowing us fine grain targeting of actions, such as


body.admin_users {
  .index {
    h1 { font-size: 21px; }
  }
  .show {
    h1 { font-size: 18px; }
  }
}
```

Secondly, we are styling naked elements, so we don't add any classes to it. It's worth repeating that we want to cut down on the number of reusable classes, so this is by design.

My typical approach in each controller file is to style naked elements first, then Bootstrap element, and then finally elements that you assign custom classes to. These last set of rules might be candidates for reuse.

The key point here is to be very particular about who you let into the reusable club; you need to be a really fierce bouncer. Each piece of code really has to prove itself to be reusable before you pull it into a module (which we will cover in the next section.)

The benefit of following the Rails controller folder for managing CSS makes it easy for any Rails developer to know exactly where a piece of CSS code lies or should lie. They do not have to guess, or wade through all the CSS files, or less often a style guide document, in order to find the right place to find or write code.

Too often we see front end code bases start with some sort of wide-eyed methodology, and then devolve into a cesspool of random styles. With this, it's an iron-clad structure that any beginner Rails developer can stick to.

Promote repeating code to reusable status

So here's where we get to the interesting part so pay attention. As mentioned in the previous section, each line of code really has to prove its reusability before you start converting them into more reusable pieces of code. When you detect code that is copied a few times, it becomes a prime candidate for reuse, but the question is where do we put them.

This may rub you the wrong way if you come from the Don't Repeat Yourself (DRY) camp. However, CSS code are notorious for getting out of hand, as you create newer classes for each edge case you want to support. This approach relies on the controller/action structure to hold off on assigning new classes until absolutely sure they are needed, or to keep the edge case rules there, and only extract the common rules to the module.

Typically, we use a framework like Bootstrap, which already provides some definition and facility for reusable components. We will take advantage of that and start with customizing common Bootstrap elements, and then work on more custom elements later.

The first thing about using Bootstrap is, use as much Bootstrap as possible. Bootstrap elements are mobile first, so using them confers a lot of advantages out of the box; writing your own mobile responsive code is a Herculean task. Soon though, we will need to customize the look and feel of Bootstrap elements. We start off by firstly creating a folder called bootstrap, and then creating files based on the elements they are styling. Your file structure should look like this:

- stylesheets
  - + layout
  - + controllers
  - bootstrap
    - btn.css.scss
    - variables.css.scss
  - application.css.scss

As an example:

```
// stylesheets/bootstrap/btn.css.scss
.btn.btn-primary {
  font-size: 32px;
}
```

Word of caution: don't go rushing into styling Bootstrap elements straight off even though its meant for reuse. You still have to abide by the rules that we set forth above: only make code reusable when they are repeated several times throughout the codebase.

Now, if you have some code is that is reused a number of times, but isn't exactly a Bootstrap element, this is a candidate for the modules folder, which now brings our folder structure to this state:

- stylesheets
  - + layout
  - + controllers
  - + boostrap
  - modules
    - article-listing.css.scss
  - application.css.scss

As an example:

```
//stylesheets/modules/article.css.scss
.article {
  .article-body p { line-height: 1.2; }
}
```

When defining my own modules, I follow SMACSS principles loosely, which is also quite close to how we are approaching the rest of the CSS.

Treat your reusable folders as sacred places. Do not defile them with code that is not reusable enough or you will have a huge mess to deal with later.

Manage fonts and icons with Google Fonts and Fontello

When building the front-end of the project, it is vital that we start with the right fonts. How else will we be able to get the font weight and size right? If possible, let's do one pass and do it right, no coming back to fix things up.

Google Fonts is really easy to integrate:

```
// views/layouts/application.html.slim
= stylesheet_link_tag 'application', 'http://fonts.googleapi
s.com/css?family=Ubuntu', media: 'all'


// use it in your CSS like so:
font-family: 'Ubuntu', sans-serif;
```

Using fonts is easy, and that's why we want to manage our icons the same way we do fonts. Icons can be really tedious to work with, especially having to update them.

Enter Fontello.

The Fontello site already hosts a number of popular icon fonts, and we ought to use them for projects as much as possible. If we have custom SVG icons or fonts, we can drag and drop them to the site and produce the font file for use in our own apps. In future, we ought to explore integrating with the site through their API.

Since we are storing the font files locally (as opposed to loading them from an external party like in Google Fonts), let's create a fonts folder in assets, so your structure becomes like:

- assets
  - fonts
    - fontello.eot
    - fontello.svg
    - fontello.ttf
    - fontello.woff
  - + images
  - + javascripts
  - - stylesheets
    - shared
      - fontello
        - fontello.scss
        - fontello-codes.scss

Do note also that there are a couple of CSS files that Fontello adds, which essentially contain the classes we need to add icons to our front-end code.

To make use of Fontello, we typically use the i element as such:

```
// any HTML Slim file
i.icon-user
```

Write less JavaScript in Rails apps

There are times when rails developers use JS as a crutch, as a way of side-stepping some Rails-ism. This is often not the most optimal solution, when properly designing your controllers, actions and routes will achieve the same goals. Since JS is still a second class citizen, we should strive to **write less JS code in a Rails app**.

In the context of this guide, we will assume that we are not using JavaScript to write a static app, where the rails app can act as an API, and the JS does most of the heavy lifting interacting with the user. That is an entirely different ball game for a different guide.

As such, we will be dealing with a typical monolithic Rails app, which I would argue is the best way to start out for most projects, and then the team can explore something a little bit more fancy later. Dealing with complex set ups from the get go, is not very lean, typically requires larger teams (otherwise productivity is lost), and does not add business value straight away.

With that out of the way, we can split up the JS code that we write into two groups:

- static JS code that we can load and cache one time
- dynamic JS code that we have to render from scratch on a per action basis

Typically, we see static JS code being used for making jQuery at load time.