Hotrails

← Back to the list of chapters

Organizing CSS files in Ruby on Rails

Published on January 19, 2022

In this chapter, we will write some CSS using the BEM methodology to create a nice design system for our application.

Sponsor this project on Github!

This tutorial is open-source forever. If you want to support my work, you can sponsor it on Github! **I will invite you to a repository with the tutorial's source code**.

Become a sponsor

Creating a design system for our Rails app

Even if this is a Turbo Rails tutorial, this is not a reason to build an ugly quote editor.

I didn't know which library to use for this tutorial even though I'm a huge CSS fan:

- Should I use Bootstrap and not focus on design at all?
- Should I use Tailwind and lose all readers who don't like the *ugly* markup that is the consequence of the utility first approach?

I knew I would create a course website to showcase the quote editor, so I thought: **Why not build a design system for the course platform and the quote editor and show how I did it**?

That's what I ended up doing, and that's why, in this course, we will write our own CSS from scratch with the BEM methodology. If you like CSS, this is an excellent opportunity to learn a few tricks. If you don't like CSS, feel free to copy/paste the code snippets as they go and move on to the next chapter!

We will write most of the CSS we need in the tutorial in this chapter, and don't worry, we will start working with Turbo in the next chapter!

Our CSS architecture

CSS is often a misunderstood topic. Like any other type of code, it needs an architecture and some conventions to be a joy to work with. The best way to learn is to write CSS from scratch for this simple application.

The BEM methodology

For the naming conventions, we will use the BEM methodology. It is straightforward, and we can summarize it in **only three rules**.

1. Each component (or *block*) should have a unique name. Let's imagine a card component in our application. The `.card` CSS class should be defined in the `card.scss` file and its name should be unique. That's the letter *B* in "BEM", and it stands for *block*.
2. Each *block* can have many *elements*. If we take back our card example, each card may have a title and a body. In BEM we will name the corresponding CSS classes `.card__title` and `.card__body`. We ensure that naming conflicts are not possible by prefixing the element's name with the block's name. If we have another block `.box` with a title and a body, `.box__title` and `.box__body` won't conflict with `.card__title` and `.card__body` thus making sure there won't be any conflicts between those classes. That's the letter *E* in "BEM", and it stands for *element*.
3. Each block may have modifiers. If we take back our card example, maybe they can have different colors. The names of the modifiers could be `.card--primary` and `.card--secondary`. That's the letter *M* in "BEM", and it stands for *modifiers*.

With this simple methodology, naming becomes very easy, and it prevents us from having to deal with one of the most frustrating aspects of CSS: **naming conflicts**.

Organizing our CSS files

Now that we have a solid naming convention, it's time to talk about file organization. As this application is quite simple, we will have a simple

architecture.

Our `app/assets/stylesheets/` folder will contain four elements:

- The `application.sass.scss` manifest file to import all our styles.
- A `mixins/` folder where we'll add Sass mixins.
- A `config/` folder where we'll add our variables and global styles.
- A `components/` folder where we'll add our components.
- A `layouts/` folder where we'll add our layouts.

What's the difference between components and layouts?

Components are independent pieces of the web page. They should not be concerned about how they are laid out on the page but only their style. An excellent example of a component is a button. Buttons don't know where they will be placed on the page.

A layout, on the contrary, does not add style. It is only concerned about margins, centering, or anything that helps us position components relative to one another. A good example of a layout is a container that will center the content on the page. If you are curious, the Every layout book shows a list of layouts almost every web app uses and their associated CSS.

Once we have built our design system, we will be able to create new pages in no time without writing additional CSS by composing components and layouts!

Note about components and margins:

In theory, as components are concerned only about themselves, **they should not have outer margins**. When we design individual components, we don't know where they will end up being placed on a page. Let's take the example of a button. It wouldn't make sense to add margins to buttons as sometimes they will be *laid out* horizontally, sometimes they will be *laid out* vertically, with more or less space between them.

We don't know in advance the *context* in which individual components will be used. That's the responsibility of layouts. As a design system grows, components will be much easier to reuse if it's easy to make them collaborate

with one another. It will be much easier if those components don't have external margins.

With that said, in this tutorial, we will break this rule. We will add margins directly on the components themselves. For example, the `.quote` CSS class will have a bottom margin. In real life, we might want to delegate this responsibility to a margin utility or a `.stack` layout to make it easier to reuse the `.quote` component in other *contexts*.

As this application will not evolve and I didn't want to make things too complex, I chose to break the *no external margin on components rule*. However, it is a great rule to keep in mind next time you build a real design system!

Enough theory; we are now ready to write some fantastic SASS code. Let's practice!

Using our CSS architecture on our quote editor

The mixins folder

The mixins folder is the smallest of all. It will only contain one file called `_media.scss` in which we will define the breakpoints of our media queries. In our application, we will have only one breakpoint that we will call `tabletAndUp`:

```
// app/assets/stylesheets/mixins/_media.scss

@mixin media($query) {
  @if $query == tabletAndUp {
    @media (min-width: 50rem) { @content; }
  }
}
```

Let's break down this piece of code together. When writing CSS, **we will first write the CSS for mobile**:

```
.my-component {
  // The CSS for mobile
```

```
}
```

Then we might want to have some **overrides for tablets and larger screens sizes**. In that case, using our media query makes things very easy:

```
.my-component {
  // The CSS for mobile

  @include media(tabletAndUp) {
    // The CSS for screens bigger than tablets
  }
}
```

This is what we call ***mobile first approach***. First, we define our CSS for the smallest screen sizes (mobiles), then add overrides for larger screen sizes. It's a good practice to use this kind of mixin in our Sass code. If we later want to add more breakpoints, such as `laptopAndUp` or `desktopAndUp`, it becomes very easy to do so.

It also clarifies what the breakpoint corresponds to as `tabletAndUp` is easier to read than `50rem`.

It also helps us keep our code DRY by avoiding repeating the magic `50rem` number everywhere:

```
.component-1 {
  // The CSS for mobile

  @media (min-width: 50rem) {
    // The CSS for screens bigger than 50rem
  }
}

.component-2 {
  // The CSS for mobile

  @media (min-width: 50rem) {
    // The CSS for screens bigger than 50rem
  }
}
```

Imagine that we want to change `50rem` to `55rem` at some point. This would be a maintenance nightmare, as we would have to change the value in all components.

Last but not least, having a curated list of breakpoints in a single place helps us choose the most relevant one a limit our options!

There was quite a lot to say for this first CSS file, but it is a really useful one. Our quote editor must be responsive, and this is a simple yet powerful implementation for breakpoints.

The configuration folder

One of the most important files is the variables file to make a robust design system. This is where we will choose beautiful colors, readable fonts and make a spaces scale to ensure consistent spacing.

Creating a variables file might look a bit magical throwing variables around, but *rules* like the *multiple of 8 rule* for spacings and font sizes to help us build the variables file. For font sizes and spaces, I almost always use the same scales.

When creating a website from scratch, I often choose a "personality" that will guide the choice of fonts and colors.

Enough design talk; let's start building this variable file!

First, let's start with everything we need for our text design, such as fonts families, colors, sizes, and line heights.

```
// app/assets/stylesheets/config/_variables.scss

:root {
  // Simple fonts
  --font-family-sans: 'Lato', -apple-system, BlinkMacSystemFont, 'Segoe

  // Classical line heights
  --line-height-headers: 1.1;
  --line-height-body:    1.5;

  // Classical and robust font sizes system
```

```
  --font-size-xs: 0.75rem;   // 12px
  --font-size-s: 0.875rem;   // 14px
  --font-size-m: 1rem;       // 16px
  --font-size-l: 1.125rem;   // 18px
  --font-size-xl: 1.25rem;   // 20px
  --font-size-xxl: 1.5rem;   // 24px
  --font-size-xxxl: 2rem;    // 32px
  --font-size-xxxxl: 2.5rem; // 40px

  // Three different text colors
  --color-text-header: hsl(0, 1%, 16%);
  --color-text-body:   hsl(0, 5%, 25%);
  --color-text-muted:  hsl(0, 1%, 44%);
}
```

This first set of variables will help us ensure our text design is consistent throughout our application.

It is also essential to keep consistent spacings for paddings and margins in the application for a good design. Let's build a simple spaces scale that we'll be able to use in our application everywhere:

```
// app/assets/stylesheets/config/_variables.scss

:root {
  // All the previous variables

  // Classical and robust spacing system
  --space-xxxs: 0.25rem; // 4px
  --space-xxs: 0.375rem; // 6px
  --space-xs: 0.5rem;    // 8px
  --space-s: 0.75rem;    // 12px
  --space-m: 1rem;       // 16px
  --space-l: 1.5rem;     // 24px
  --space-xl: 2rem;      // 32px
  --space-xxl: 2.5rem;   // 40px
  --space-xxxl: 3rem;    // 48px
  --space-xxxxl: 4rem;   // 64px
}
```

To make our design, we will also need colors, of course.

```
// app/assets/stylesheets/config/_variables.scss

:root {
  // All the previous variables

  // Application colors
  --color-primary:          hsl(350, 67%, 50%);
  --color-primary-rotate:   hsl(10, 73%, 54%);
  --color-primary-bg:       hsl(0, 85%, 96%);
  --color-secondary:        hsl(101, 45%, 56%);
  --color-secondary-rotate: hsl(120, 45%, 56%);
  --color-tertiary:         hsl(49, 89%, 64%);
  --color-glint:            hsl(210, 100%, 82%);

  // Neutral colors
  --color-white:      hsl(0, 0%, 100%);
  --color-background: hsl(30, 50%, 98%);
  --color-light:      hsl(0, 6%, 93%);
  --color-dark:       var(--color-text-header);
}
```

The last part of our variables file will contain various user interface styles such as border radiuses and box shadows. Once again, the goal is to ensure consistency in our application.

```
// app/assets/stylesheets/config/_variables.scss

:root {
  // All the previous variables

  // Border radius
  --border-radius: 0.375rem;

  // Border
  --border: solid 2px var(--color-light);

  // Shadows
  --shadow-large:  2px 4px 10px hsl(0 0% 0% / 0.1);
  --shadow-small:  1px 3px 6px hsl(0 0% 0% / 0.1);
}
```

That's it! That's all the variables needed to design our quote editor! (I use the same variables on the website you are currently reading).

The next step in designing our application is to apply those variables to global styles i.e., styles that are all the same in the whole application. These styles should represent reasonable styling defaults for HTML tags. We should not use any CSS classes here. For example, in the file below, we:

- Style default text
- Style default links
- Reset all margins and paddings to zero as it should be layout classes responsibility

Those changes are global to all the pages of our application. They can be easily overridden because they are only applied to HTML tags and thus, have a low specificity.

```
// app/assets/stylesheets/config/_reset.scss

*,
*::before,
*::after {
  box-sizing: border-box;
}

* {
  margin: 0;
  padding: 0;
}

html {
  overflow-y: scroll;
  height: 100%;
}

body {
  display: flex;
  flex-direction: column;
  min-height: 100%;

  background-color: var(--color-background);
  color: var(--color-text-body);
  line-height: var(--line-height-body);
  font-family: var(--font-family-sans);
}
```

```
img,
picture,
svg {
  display: block;
  max-width: 100%;
}

input,
button,
textarea,
select {
  font: inherit;
}

h1,
h2,
h3,
h4,
h5,
h6 {
  color: var(--color-text-header);
  line-height: var(--line-height-headers);
}

h1 {
  font-size: var(--font-size-xxxl);
}

h2 {
  font-size: var(--font-size-xxl);
}

h3 {
  font-size: var(--font-size-xl);
}

h4 {
  font-size: var(--font-size-l);
}

a {
  color: var(--color-primary);
  text-decoration: none;
  transition: color 200ms;
```

```
  &:hover,
  &:focus,
  &:active {
    color: var(--color-primary-rotate);
  }
}
```

Now that we have our global style in place, we can design our individual components.

The components folder

The components folder will contain styles for our individual components. Now that we have a solid set of variables, designing components will be straightforward.

Let's start by, *believe it or not*, our most complex components: buttons. We will start with the base `.btn` class and then add four *modifiers* for the different styles:

```
// app/assets/stylesheets/components/_btn.scss

.btn {
  display: inline-block;
  padding: var(--space-xxs) var(--space-m);
  border-radius: var(--border-radius);
  background-origin: border-box; // Invisible borders with linear gradie
  background-color: transparent;
  border: solid 2px transparent;
  font-weight: bold;
  text-decoration: none;
  cursor: pointer;
  outline: none;
  transition: filter 400ms, color 200ms;

  &:hover,
  &:focus,
  &:focus-within,
  &:active {
    transition: filter 250ms, color 200ms;
  }
```

```
  // Modifiers will go there
}
```

The `.btn` class is an inline-block element to which we added default styles such as padding, border-radius, and transition. Note that with Sass, the `&` sign corresponds to the selector in which the `&` is directly nested. In our case `&:hover`, will be translated by Sass to `.btn:hover` in CSS.

We then need to add four modifiers to create the four variants of the button component we will need in our application:

```
// app/assets/stylesheets/components/_btn.scss

.btn {
  // All the previous code

  &--primary {
    color: var(--color-white);
    background-image: linear-gradient(to right, var(--color-primary), va

    &:hover,
    &:focus,
    &:focus-within,
    &:active {
      color: var(--color-white);
      filter: saturate(1.4) brightness(115%);
    }
  }

  &--secondary {
    color: var(--color-white);
    background-image: linear-gradient(to right, var(--color-secondary),

    &:hover,
    &:focus,
    &:focus-within,
    &:active {
      color: var(--color-white);
      filter: saturate(1.2) brightness(110%);
    }
  }

  &--light {
```

```
    color: var(--color-dark);
    background-color: var(--color-light);

    &:hover,
    &:focus,
    &:focus-within,
    &:active {
      color: var(--color-dark);
      filter: brightness(92%);
    }
  }

  &--dark {
    color: var(--color-white);
    border-color: var(--color-dark);
    background-color: var(--color-dark);

    &:hover,
    &:focus,
    &:focus-within,
    &:active {
      color: var(--color-white);
    }
  }
}
```

The same rule applies here: `&--primary` will be translated to `.btn--primary` by the Sass preprocessor.

That was quite a lot of CSS for a simple button! The following components will be much simpler. Let's continue with the `.quote` component:

```
// app/assets/stylesheets/components/_quote.scss

.quote {
  display: flex;
  justify-content: space-between;
  align-items: center;
  gap: var(--space-s);

  background-color: var(--color-white);
  border-radius: var(--border-radius);
  box-shadow: var(--shadow-small);
```

```
  margin-bottom: var(--space-m);
  padding: var(--space-xs);

  @include media(tabletAndUp) {
    padding: var(--space-xs) var(--space-m);
  }

  &__actions {
    display: flex;
    flex: 0 0 auto;
    align-self: flex-start;
    gap: var(--space-xs);
  }
}
```

The `.quote` component is just a flex container with a `.quote__actions` element containing the buttons to edit and delete the quote. Thanks to the BEM methodology, our CSS files are very clean!

Let's now design our inline forms. In the previous chapter, we defined the Simple Form wrappers that will be used for the forms in our application. We will need two components to design our forms: `.form` and `.visually-hidden`.

```
config.wrappers :default, class: "form__group" do |b|
  b.use :html5
  b.use :placeholder
  b.use :label, class: "visually-hidden"
  b.use :input, class: "form__input", error_class: "form__input--invalid
end
```

Let's start with the `.form` component. Note that it has the same border width and vertical padding as the `.btn` component to be the same height:

```
// app/assets/stylesheets/components/_form.scss

.form {
  display: flex;
  flex-wrap: wrap;
  gap: var(--space-xs);

  &__group {
```

```
    flex: 1;
  }

  &__input {
    display: block;
    width: 100%;
    max-width: 100%;
    padding: var(--space-xxs) var(--space-xs);
    border: var(--border);
    border-radius: var(--border-radius);
    outline: none;
    transition: box-shadow 250ms;

    &:focus {
      box-shadow: 0 0 0 2px var(--color-glint);
    }

    &--invalid {
      border-color: var(--color-primary);
    }
  }
}
```

Last but not least, let's add the `.visually-hidden` component:

```
// app/assets/stylesheets/components/_visually_hidden.scss

// Shamelessly stolen from Bootstrap

.visually-hidden {
  position: absolute !important;
  width: 1px !important;
  height: 1px !important;
  padding: 0 !important;
  margin: -1px !important;
  overflow: hidden !important;
  clip: rect(0, 0, 0, 0) !important;
  white-space: nowrap !important;
  border: 0 !important;
}
```

You might wonder why we need a `.visually-hidden` component to hide the input label and not simply remove it from the DOM or use `display: none;`. For accessibility purposes, all form inputs should have labels that can be

interpreted by screen readers even if they are not visible on the web page. This is why most applications use a `.visually-hidden` component. We don't have to learn the CSS by heart as we can steal the component from the Bootstrap source code.

No form would be complete without a way to display error messages. Let's add the simple component to display errors in red:

```
// app/assets/stylesheets/components/_error_message.scss

.error-message {
  width: 100%;
  color: var(--color-primary);
  background-color: var(--color-primary-bg);
  padding: var(--space-xs);
  border-radius: var(--border-radius);
}
```

That's it; we have all the components we need to complete the CRUD on our `Quote` model. We just need two layouts now, and we will be done with the CSS!

The layouts folder

The container is a layout that almost every web application has. It is used to center content on the page and limit its maximum width. We will use it on every page of our quote editor. Here is the implementation we will use:

```
// app/assets/stylesheets/layouts/_container.scss

.container {
  width: 100%;
  padding-right: var(--space-xs);
  padding-left: var(--space-xs);
  margin-left: auto;
  margin-right: auto;

  @include media(tabletAndUp) {
    padding-right: var(--space-m);
    padding-left: var(--space-m);
    max-width: 60rem;
  }
}
```

The header is a layout we will use twice in this tutorial on the `Quotes#index` and the `Quotes#show` page. It will contain the title of the page and the main action button:

```
// app/assets/stylesheets/layouts/_header.scss

.header {
  display: flex;
  flex-wrap: wrap;
  gap: var(--space-s);
  justify-content: space-between;
  margin-top: var(--space-m);
  margin-bottom: var(--space-l);

  @include media(tabletAndUp) {
    margin-bottom: var(--space-xl);
  }
}
```

Now that we have all our layouts, we wrote all the necessary CSS for our application. We just have to import all those files in the manifest file for Sass to bundle them.

The manifest file

Finally, we have to import all of those CSS files in our `application.sass.scss` for all our design files to be added to a single compiled CSS file.

```
// app/assets/stylesheets/application.sass.scss

// Mixins
@import "mixins/media";

// Configuration
@import "config/variables";
@import "config/reset";

// Components
@import "components/btn";
@import "components/error_message";
@import "components/form";
```

```
@import "components/visually_hidden";
@import "components/quote";

// Layouts
@import "layouts/container";
@import "layouts/header";
```

Here we are; with only a few CSS files, we created a nice design for our CRUD on the `Quote` model. We will add new components in the following chapters as we need them, but we already wrote most of our CSS.

It's now time to start working on what we are the most interested in: **Turbo**! Take a small break and see you in the next chapter, where we will talk about **Turbo Drive**!

← previous next →

Get notified when I write new articles

If you liked this article and want to keep up with Ruby on Rails and Hotwire, you can subscribe to my newsletter (no spam, no tracking, unsubscribe any time)!

Subscribe to the newsletter

Github Twitter Newsletter

Made with remotely