Sustainable SCSS

Table of Contents

Sasstainability

Sass can be confusing if it's not organized correctly. In this unit, we will dive into the language's best practices.

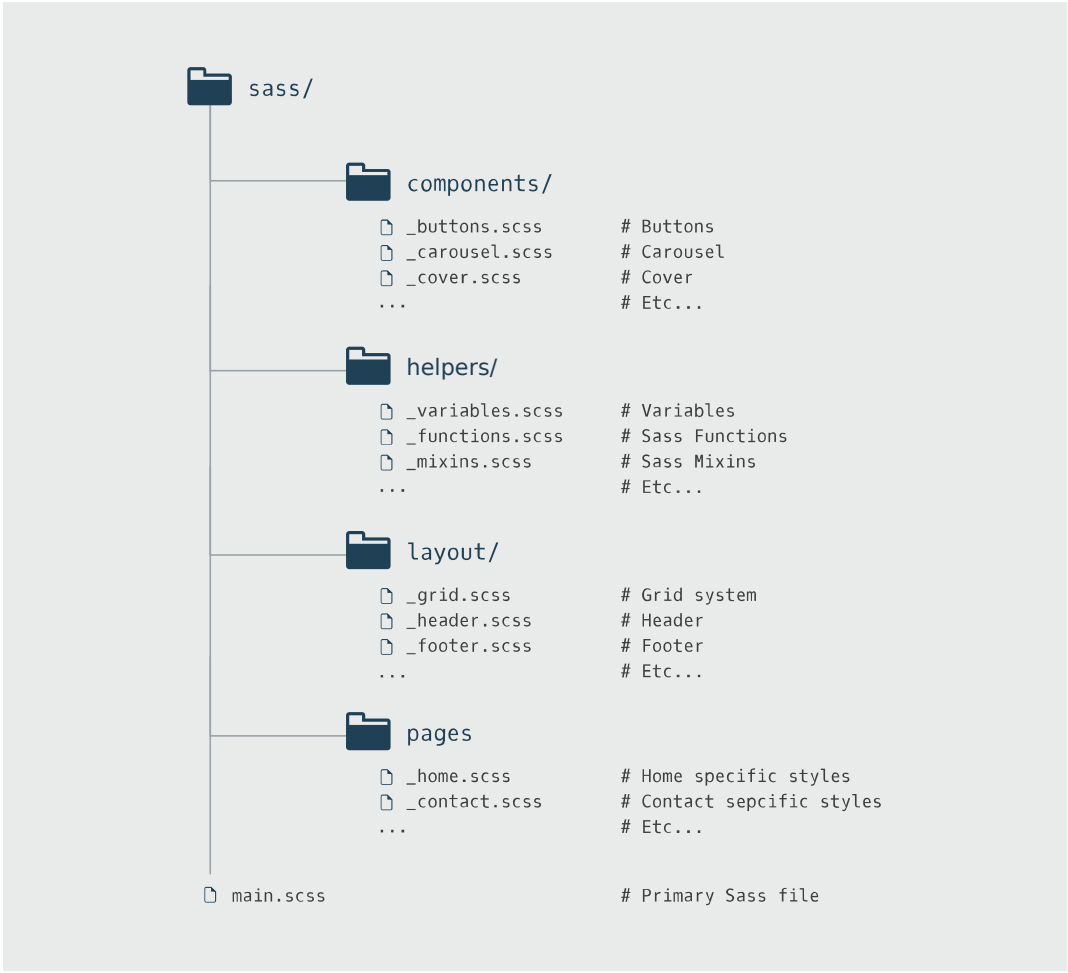
From file organization, to understanding when is best to include a mixin or extend a placeholder, we will learn about the methods that get the most out of Sass.

Sass Structure

We'll start with best practices for organizing files. As your web app or web page grows in complexity, so will the styles that go along with it. It's best to keep code organized.

Instructions

1. Here we have an example of a well-organized Sass file structure. Notice how the file structure makes it easy to think of the functionality of each component, facilitating the action of finding and updating files.



@Import in SCSS

In addition to having a solid file structure, a big part of staying organized is splitting up the logic into smaller manageable components.

Sass extends the existing CSS @import rule to allow including other SCSS and Sass files.

- Typically, all imported SCSS files are imported into a main SCSS file which is then combined to make a single CSS output file.
- The main/global SCSS file has access to any variables or mixins defined in its imported files. The `@import command takes a filename to import.

By default, @import looks for a Sass file in the same or otherwise specified directory but there are a few circumstances where it will behave just like a CSS @import rule:

- If the file's extension is **.css**
- If the filename begins with http://
- If the filename is a url()
- If the @import has any media queries

In addition to keeping code organized, importing files can also save you from repeating code. For example, if multiple SCSS files reference the same variables, importing a file with variables partial would save the trouble of redefining them each time.

Instructions

Analyze the file architecture currently set up for the given project.

The lemonade stand is a smaller project, so we have only set up a helper folder. As it grows, we could add partials for _pages, _components and more!

Organize with Partials

Partials in Sass are the files you split up to organize specific functionality in the codebase.

- They use an underscore _ prefix notation in the file name that tells Sass to hold off on compiling the file individually and instead import it.

_filename.scss

- To import this partial into the main file - or the file that encapsulates the important rules and the bulk of the project styles - omit the underscore.

For example, to import a file named **_variables.scss**, add the following line of code:

```
@import "variables";
```

The global file imports all the components and centralizes the logic.

1. At the top of **main.scss**, import the variables partial:

```
@import "helper/variables";
```

A reminder that helper refers to resources like variables, mixins, functions, etc. These are the elements that "help" make up the foundation of your codebase as it grows.

Compile to see your changes in the browser and inspect them in the output of **main.css**.

@Extend

Many times, when styling elements, we want the styles of one class to be applied to another in addition to its own individual styles, so the traditional approach is to give the element both classes.

```
<span class="lemonade"></span>

<span class="lemonade strawberry"></span>
```

This is a potential bug in maintainability because then both classes always have to be included in the HTML in order for the styles to be applied.

Enter Sass's @extend. All we have to do is make our strawberry class extend .lemonade and we will no longer have this dilemma.

```
.lemonade {
  border: 1px yellow;
  background-color: #fdd;
}
.strawberry {
  @extend .lemonade;
  border-color: pink;
```

```
}
```

If you observe CSS output, you can see how @extend is working to apply the .lemonade rules to .strawberry:

```
.lemonade, .strawberry {
  border: 1px yellow;
  background-color: #fdd;
}

.strawberry {
  border-color: pink;
}
```

If we think of .lemonade as the extendee, and of .strawberry as the extender, we can then think of Sass appending the extender selector to the rule declarations in the extendee definition.

This makes it easy to maintain HTML code by removing the need to have multiple classes on an element.

1. In **main.scss**, define an .absolute class selector as follows:

```
.absolute {
   position: absolute
}
```

Extend this class inside of both .slogan and inside of span:

```
@extend .absolute;
```

Compile to see your changes in the browser and inspect them in the output of **main.css**.

Note: Ideally, classes we extend should have more than one property, but this example should serve to illustrate the concepts of extending.

%Placeholders

Sometimes, you may create classes solely for the purpose of extending them and never actually use them inside your HTML.

Sass anticipated this and allows for a special type of selector called a *placeholder*, which behaves just like a class or id selector, but use the % notation instead of # or .

Placeholders prevent rules from being rendered to CSS on their own and only become active once they are extended anywhere an id or class could be extended.

```
a%drink {
   font-size: 2em;
   background-color: $lemon-yellow;
}

.lemonade {
 @extend %drink;
 //more rules
}
```

would translate to

```
  a.lemonade {
    font-size: 2em;
    background-color: $lemon-yellow;
 }

.lemonade {
  //more rules
}
```

Placeholders are a nice way to consolidate rules that never actually get used on their own in the HTML.

1. Notice how we never actually use `.absolute` anywhere in the HTML? Remove it from **main.scss** and place it inside **helper/_placeholders.scss**:

```
%absolute{
   position: absolute;
}
```

Now replace your old extend line with the placeholder extend notation:

```
@extend %absolute;
```

Compile to see your changes in the browser and inspect them in the output of **main.css**.

@Extend vs @Mixin

Recall that mixins, unlike extended selectors, insert the code inside the selector's rules wherever they are included, only including "original" code if they are assigning a new value to the rule's properties via an argument.

Let's look at the `@mixin` and `@extend` constructs closely and compare output:

```
@mixin no-variable {
  font-size: 12px;
  color: #FFF;
  opacity: .9;
}

%placeholder {
  font-size: 12px;
  color: #FFF;
  opacity: .9;
}

span {
  @extend %placeholder;
}

div {
  @extend %placeholder;
}

p {
  @include no-variable;
}
```

```
h1 {
  @include no-variable;
}
```

it would compile to:

```
span, div{
  font-size: 12px;
  color: #FFF;
  opacity: .9;
}

p {
  font-size: 12px;
  color: #FFF;
  opacity: .9;
  //rules specific to p
}

h1 {
  font-size: 12px;
  color: #FFF;
  opacity: .9;
  //rules specific to h1
}
```

We can clearly see extending results in way cleaner and more efficient output with as little repetition as possible.

As a general rule of thumb, you should

- Try to only create mixins that take in an argument, otherwise you should extend.
- Always look at your CSS output to make sure your extend is behaving as you intended.

1. Remove the `center` mixin in the **helper/_mixins.scss** partial that does not take in a variable. Convert the placeholder named `%center` inside the **helper/_placeholders.scss** partial.

Be sure to change the include statements to extend inside both `span` and `h1`:

```
@extend %center;
```

Compile to see your changes in the browser and inspect them in the output of **main.css**.

Generalizations

- **Sustainability** is key in Sass, planning out the structure of your files and sticking to naming conventions for both variables, mixins, and selectors can reduce complexity.
- Understanding **CSS output** is also essential to writing sustainable SCSS. In order to make the best choices about what functions and directives to use, it is important to first understand how this will translate in CSS.
- Mixins should only be used if they take in an argument, otherwise, you should **extend the selector's rules,** whether it be a class, id, or placeholder.

In addition to the directives you have learned in this course, be sure to check out the many

additional available Sass functions and directives.