Sass Basics

sass-lang.com/guide

Before you can use Sass, you need to set it up on your project. If you want to just browse here, go ahead, but we recommend you go install Sass first. Go here if you want to learn how to get everything setup.

Preprocessing

CSS on its own can be fun, but stylesheets are getting larger, more complex, and harder to maintain. This is where a preprocessor can help. Sass lets you use features that don't exist in CSS yet like variables, nesting, mixins, inheritance and other nifty goodies that make writing CSS fun again.

Once you start tinkering with Sass, it will take your preprocessed Sass file and save it as a normal CSS file that you can use in your website.

The most direct way to make this happen is in your terminal. Once Sass is installed, you can compile your Sass to CSS using the `sass` command. You'll need to tell Sass which file to build from, and where to output CSS to. For example, running `sass input.scss output.css` from your terminal would take a single Sass file, `input.scss` , and compile that file to `output.css` .

You can also watch individual files or directories with the `--watch` flag. The watch flag tells Sass to watch your source files for changes, and re-compile CSS each time you save your Sass. If you wanted to watch (instead of manually build) your `input.scss` file, you'd just add the watch flag to your command, like so:

```
sass --watch input.scss output.css
```

You can watch and output to directories by using folder paths as your input and output, and separating them with a colon. In this example:

```
sass --watch app/sass:public/stylesheets
```

Sass would watch all files in the `app/sass` folder for changes, and compile CSS to the `public/stylesheets` folder.

Variables

Think of variables as a way to store information that you want to reuse throughout your stylesheet. You can store things like colors, font stacks, or any CSS value you think you'll want to reuse. Sass uses the `$` symbol to make something a variable. Here's an example:

SCSS Syntax

```
$font-stack:    Helvetica, sans-serif;
$primary-color: #333;

body {
  font: 100% $font-stack;
  color: $primary-color;
}
```

CSS Output

```
body {
  font: 100% Helvetica, sans-serif;
  color: #333;
}
```

When the Sass is processed, it takes the variables we define for the `$font-stack` and `$primary-color` and outputs normal CSS with our variable values placed in the CSS. This can be extremely powerful when working with brand colors and keeping them consistent throughout the site.

Nesting

When writing HTML you've probably noticed that it has a clear nested and visual hierarchy. CSS, on the other hand, doesn't.

Sass will let you nest your CSS selectors in a way that follows the same visual hierarchy of your HTML. Be aware that overly nested rules will result in over-qualified CSS that could prove hard to maintain and is generally considered bad practice.

With that in mind, here's an example of some typical styles for a site's navigation:

SCSS Syntax

```
nav {
  ul {
    margin: 0;
    padding: 0;
    list-style: none;
  }

  li { display: inline-block; }

  a {
    display: block;
    padding: 6px 12px;
    text-decoration: none;
  }
}
```

CSS Output

```
nav ul {
  margin: 0;
  padding: 0;
  list-style: none;
}
nav li {
  display: inline-block;
}
nav a {
  display: block;
  padding: 6px 12px;
  text-decoration: none;
}
```

You'll notice that the `ul` , `li` , and `a` selectors are nested inside the `nav` selector. This is a great way to organize your CSS and make it more readable.

Partials

You can create partial Sass files that contain little snippets of CSS that you can include in other Sass files. This is a great way to modularize your CSS and help keep things easier to maintain. A partial is a Sass file named with a leading underscore. You might name it something like `_partial.scss` . The underscore lets Sass know that the file is only a partial file and that it should not be generated into a CSS file. Sass partials are used with the `@use` rule.

Modules

Compatibility:

Dart Sass
since 1.23.0

LibSass
✗

Ruby Sass
✗

▶

You don't have to write all your Sass in a single file. You can split it up however you want with the `@use` rule. This rule loads another Sass file as a *module*, which means you can refer to its variables, mixins, and functions in your Sass file with a namespace based on the filename. Using a file will also include the CSS it generates in your compiled output!

SCSS Syntax

```
// _base.scss
$font-stack:    Helvetica, sans-serif;
$primary-color: #333;

body {
  font: 100% $font-stack;
  color: $primary-color;
}

// styles.scss
@use 'base';

.inverse {
  background-color: base.$primary-color;
  color: white;
}
```

CSS Output

```
body {
  font: 100% Helvetica, sans-serif;
  color: #333;
}

.inverse {
  background-color: #333;
  color: white;
}
```

Notice we're using `@use 'base';` in the `styles.scss` file. When you use a file you don't need to include the file extension. Sass is smart and will figure it out for you.

Mixins

Some things in CSS are a bit tedious to write, especially with CSS3 and the many vendor prefixes that exist. A mixin lets you make groups of CSS declarations that you want to reuse throughout your site. You can even pass in values to make your mixin more flexible. A good use of a mixin is for vendor prefixes. Here's an example for `transform`.

SCSS Syntax

```
@mixin transform($property) {
  -webkit-transform: $property;
  -ms-transform: $property;
  transform: $property;
}
.box { @include transform(rotate(30deg)); }
```

CSS Output

```
.box {
  -webkit-transform: rotate(30deg);
  -ms-transform: rotate(30deg);
  transform: rotate(30deg);
}
```

To create a mixin you use the `@mixin` directive and give it a name. We've named our mixin `transform`. We're also using the variable `$property` inside the parentheses so we can pass in a transform of whatever we want. After you create your mixin, you can then use it as a CSS declaration starting with `@include` followed by the name of the mixin.

Extend/Inheritance

This is one of the most useful features of Sass. Using `@extend` lets you share a set of CSS properties from one selector to another. It helps keep your Sass very DRY. In our example we're going to create a simple series of messaging for errors, warnings and successes using another feature which goes hand in hand with extend, placeholder classes. A placeholder class is a special type of class that only prints when it is extended, and can help keep your compiled CSS neat and clean.

SCSS Syntax

```
/* This CSS will print because %message-shared is extended. */
%message-shared {
  border: 1px solid #ccc;
  padding: 10px;
  color: #333;
}

// This CSS won't print because %equal-heights is never extended.
%equal-heights {
  display: flex;
  flex-wrap: wrap;
}

.message {
  @extend %message-shared;
}

.success {
  @extend %message-shared;
  border-color: green;
}

.error {
  @extend %message-shared;
  border-color: red;
}

.warning {
  @extend %message-shared;
  border-color: yellow;
}
```

What the above code does is tells `.message` , `.success` , `.error` , and `.warning` to behave just like `%message-shared` . That means anywhere that `%message-shared` shows up, `.message` , `.success` , `.error` , & `.warning` will too. The magic happens in the generated CSS, where each of these classes will get the same CSS properties as `%message-shared` . This helps you avoid having to write multiple class names on HTML elements.

You can extend most simple CSS selectors in addition to placeholder classes in Sass, but using placeholders is the easiest way to make sure you aren't extending a class that's nested elsewhere in your styles, which can result in unintended selectors in your CSS.

Note that the CSS in `%equal-heights` isn't generated, because `%equal-heights` is never extended.

Operators

Doing math in your CSS is very helpful. Sass has a handful of standard math operators like `+` , `-` , `*` , `/` , and `%` . In our example we're going to do some simple math to calculate widths for an `aside` & `article` .

SCSS Syntax

```
.container {
  width: 100%;
}

article[role="main"] {
  float: left;
  width: 600px / 960px * 100%;
}

aside[role="complementary"] {
  float: right;
  width: 300px / 960px * 100%;
}
```

CSS Output

```css
.container {
  width: 100%;
}

article[role="main"] {
  float: left;
  width: 62.5%;
}

aside[role="complementary"] {
  float: right;
  width: 31.25%;
}
```

We've created a very simple fluid grid, based on 960px. Operations in Sass let us do something like take pixel values and convert them to percentages without much hassle.