



CONTENTS

Introduction

What is Sass?

So uh, how does it work?

Whats the deal with .sass vs .scss?

Why would I use Sass?

Set-Up

Variables

Math

Functions

Nesting

Imports

Extends & Placeholders

Mixins

Function Directives

Demo

Conclusion

Resources

// Tutorial //

Getting Started with Sass (with Interactive Examples)

Updated on September 15, 2020



By [Ken Wheeler](#)

Developer and author at DigitalOcean.



Introduction

POPULAR TOPICS ▾

Ubuntu

Linux Basics

JavaScript

React

Python

Security

Apache

MySQL

Databases

Docker

Kubernetes

Ebooks

Browse all topic tags

ALL TUTORIALS →

QUESTIONS ▾

Q&A

Ask a question

DigitalOcean Product Docs

Have you always wanted to learn Sass, but never quite made your move? Are you a Sass user, but feel like you could use a brush-up? Well then, read on, because today we are going to review the features of Sass and some of the cool things you can do with it.

What is Sass?

Sass (Syntactically Awesome Style Sheets) is a CSS preprocessor. It is to CSS what CoffeeScript is to Javascript. Sass adds a feature set to your stylesheet markup that makes writing styles fun again.

So uh, how does it work?

Funny you should ask. There are several ways you can compile Sass:

- The original Ruby Sass binary. Install it with `gem install sass`, and compile it by running `sassc myfile.scss myfile.css`.
- A GUI app such as [Hammer](#), [CodeKit](#), or [Compass](#)
- My personal favorite [libsass](#), which is a blazing fast Sass compiler written in C. You can also install libsass via NPM with [node-sass](#) (`npm install node-sass`).

Which one should you use? That depends on what you are doing.

I work with large-scale e-commerce codebases, so Ruby Sass is a little slow when compiling large source sets. I use `node-sass` in my build system, but I have to remain wary of the fact that `libsass` is not in 100% feature parity with Ruby Sass.

If you aren't a command-line person, the GUI apps are great. You can set them up to watch `.scss` files, so when you edit them they will compile automatically.

DigitalOcean Support

EVENTS ▼

Tech Talks

Hacktoberfest

Deploy

GET INVOLVED ▼

Community Newsletter

Hollie's Hub for Good

Write for DOnations

Community tools and integrations

Hatch Startup program

CREATE YOUR FREE
COMMUNITY
ACCOUNT! →

If you want to just screw around, or share examples, I highly recommend [Sassmeister](#). It is a web-based Sass playground that I will be using throughout this article.

Whats the deal with .sass vs .scss?

When Sass first came out, the main syntax was noticeably different from CSS. It used indentation instead of braces, didn't require semi-colons, and had shorthand operators. In short, it looked a lot like [Haml](#).

Some folks didn't take too kindly to the new syntax, and in version 3 Sass changed its main syntax to `.scss`. SCSS is a superset of CSS, and is basically written the exact same, but with all the fun new Sass features.

That said, you can still use the original syntax if you want to. I personally use `.scss`, and I will be using the `.scss` syntax in this article.

Why would I use Sass?

Good question. Sass makes writing maintainable CSS easier. You can get more done, in less code, more readably, in less time.

Do you need more of a reason than that?

Set-Up

Without any further ado, let's get this party started. If you want to try some of these concepts while following along, either:

- Install your compilation method of choice, and create a `style.scss` file.

Or

- Follow along on [Sassmeister](#)

Variables

That's right, variables. Sass brings variables to CSS.

Acceptable values for variables include numbers, strings, colors, null, lists, and maps.

Variables in Sass are scoped using the `$` symbol. Let's create our first variable:

```
$primaryColor: #eeffcc;
```

Copy

If you tried to compile this and didn't see anything in your CSS, you're doin' it right. Defining variables on their own doesn't actually output any CSS, it just sets it within the scope. You need to use it within a CSS declaration to see it:

```
$primaryColor: #eeffcc;

body {
  background: $primaryColor;
}
```

Copy

Speak of the devil (scope), did you know that Sass has variable scope? That's right, if you declare a variable within a selector, it is then scoped within that selector. Check it out:

```
$primaryColor: #eeccff;
```

Copy

```
body {
```

```
$primaryColor: #ccc;
background: $primaryColor;
}

p {
  color: $primaryColor;
}
```

When compiled, our paragraph selector's color is `#ecccff`.

But what if we want to set a variable globally from within a declaration? Sass provides a `!global` flag that comes to our rescue:

```
$primaryColor: #ecccff;

body {
  $primaryColor: #ccc !global;
  background: $primaryColor;
}

p {
  color: $primaryColor;
}
```

Copy

When compiled, our paragraph selector's color is `#ccc`.

Another helpful flag, particularly when writing mixins, is the `!default` flag. This allows us to make sure there is a default value for a variable in the event that one is not provided. If a value is provided, it is overwritten:

```
$firstValue: 62.5%;

$firstValue: 24px !default;
```

Copy

```
body {  
  font-size: $firstValue;  
}
```

When compiled, our body font size is 62.5%.

Play with some variables below to see how the Sass you are writing is compiled to CSS:

[Play with this gist on SassMeister.](#)

Math

Unlike CSS, Sass allows us to use mathematical expressions! This is super helpful within mixins and allows us to do some really cool things with our markup.

Supported operators include:

Operator	Symbol
Addition	+
Subtraction	-
Division	/
Multiplication	*
Modulo	%
Equality	==
Inequality	!=

Before moving forward, I want to note two potential “gotchas” with Sass math.

First, because the `/` symbol is used in shorthand CSS font properties like `font: 14px/16px`, if you want to use the division operator on non-variable values, you need to wrap them in parentheses like:

```
$fontDiff: (14px/16px);
```

Copy

Second, you can't mix value units:

```
$container-width: 100% - 20px;
```

Copy

The above example won't work. Instead, for this particular example, you could use the CSS `calc` function, as it needs to be interpreted at render time.

Back to math, let's create a dynamic column declaration, based upon a base container width:

```
$container-width: 100%;

.container {
  width: $container-width;
}

.col-4 {
  width: $container-width / 4;
}
```

Copy

Output

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

Copy

```
.col-4 {  
  width: 25%;  
}
```

Awesome, right? Check out in the example below how we can further leverage Sass math to add margins. Play around with the values to see our example change:

[Play with this gist on SassMeister.](#)

Functions

The best part of Sass, in my opinion, is its built-in functions. You can see the full list [here](#). It is *EXTENSIVE*.

Have you ever wanted to make a cool-looking button, and then taken the time to mess around on a color wheel, trying to find the right shades for ‘shadowed’ parts?

Enter the `darken()` function. You can pass it a color and a percentage and it, wait for it, darkens your color. Check this demo out to see why this is cool:

[Play with this gist on SassMeister.](#)

Nesting

One of the most helpful, and also misused features of Sass, is the ability to nest declarations. With great power comes great responsibility, so let’s take a second to realize what this does, and in the wrong hands, what bad things it could do.

Basic nesting refers to the ability to have a declaration inside of a declaration. In normal CSS we might write:

```
.container {  
  width: 100%;  
}  
  
.container h1 {  
  color: red;  
}
```

Copy

But in Sass we can get the same result by writing:

```
.container {  
  width: 100%;  
  
  h1 {  
    color: red;  
  }  
}
```

Copy

That's bananas! So what if we want to reference the parent? This is achieved by using the ampersand symbol. Check out how we can leverage this to add pseudo selectors to anchor elements:

```
a.myAnchor {  
  color: blue;  
  
  &:hover {  
    text-decoration: underline;  
  }  
}
```

Copy

```
&.visited {  
  color: purple;  
}  
}
```

Now we know how to nest, but if we want to de-nest, we have to use the `@at-root` directive. Say we have a nest set up like so:

```
.first-component {  
  .text { font-size: 1.4rem; }  
  .button { font-size: 1.7rem; }  
  
  .second-component {  
    .text { font-size: 1.2rem; }  
    .button { font-size: 1.4rem; }  
  }  
}
```

Copy

After realizing that the second component might be used elsewhere, we have ourselves a pickle. Well, not really. `@at-root` to the rescue:

[Play with this gist on SassMeister.](#)

Cool huh? Nests are a really great way to save some time and make your styles readable, but over nesting can cause problems with over selection and file size. Always look at what your sass compiles to and try to follow the “inception rule”.

The Inception Rule: don't go more than four levels deep. *via*
<http://thesassway.com/>

If possible, don't nest more than four levels. If you, in a pinch, have to go five levels deep, Hampton Catlin isn't going to come to your house and fight you. Just try not to do it.

Imports

Easily my second favorite part of Sass, imports allow you to break your styles into separate files and import them into one another. This does wonders for organization and speed of editing.

We can import a .scss file using the `@import` directive:

```
@import "grid.scss";
```

[Copy](#)

In fact, you don't even really need the extension:

```
@import "grid";
```

[Copy](#)

Sass compilers also include a concept called “partials”. If you prefix a .sass or .scss file with an underscore, it will not get compiled to CSS. This is helpful if your file only exists to get imported into a master `style.scss` and not explicitly compiled.

Extends & Placeholders

In Sass, the `@extend` directive is an outstanding way to inherit already existing styles.

Lets use an `@extend` directive to extend an input's style if it has an `input-error` class:

```
.input {  
  border-radius: 3px;  
  border: 4px solid #ddd;  
  color: #555;  
  font-size: 17px;  
  padding: 10px 20px;  
  display: inline-block;  
  outline: 0;  
}  
  
.error-input {  
  @extend .input;  
  border: 4px solid #e74c3c;  
}
```

Copy

Please note, this does not copy the styles from `.input` into `.error-input`. Take a look at the compiled CSS in this example to see how it is intelligently handled:

[Play with this gist on SassMeister.](#)

But what about if we want to extend a declaration with a set of styles that doesn't already exist? Meet the placeholder selector.

```
%input-style {  
  font-size: 14px;  
}  
  
input {  
  @extend %input-style;
```

Copy

```
color: black;  
}
```

The placeholder selector works by prefixing a class name of your choice with a `%` symbol. It is never rendered outright, only the result of its extending elements are rendered in a single block.

Check out below how our previous example works with a placeholder:

[Play with this gist on SassMeister.](#)

Mixins

The mixin directive is an incredibly helpful feature of Sass, in that it allows you to include styles the same way `@extend` would, but with the ability to supply and interpret arguments.

Sass uses the `@mixin` directive to define mixins, and the `@include` directive to use them. Let's build a simple mixin that we can use for media queries!

Our first step is to define our mixin:

```
@mixin media($queryString){  
  
}
```

Copy

Notice we are calling our mixin `media` and adding a `$queryString` argument. When we include our mixin, we can supply a string argument that will be dynamically rendered. Let's put the guts in:

```
@mixin media($queryString){  
  @media #{ $queryString} {  
    @content;  
  }  
}
```

[Copy](#)

Because we want our string argument to render where it belongs, we use the Sass interpolation syntax, `#{ }`. When you put a variable in between the braces, it is printed rather than evaluated.

Another piece of our puzzle is the `@content` directive. When you wrap a mixin around content using braces, the wrapped content becomes available via the `@content` directive.

Finally, lets use our mixin with the `@include` directive:

```
.container {  
  width: 900px;  
  
  @include media("(max-width: 767px)") {  
    width: 100%;  
  }  
}
```

[Copy](#)

Check out the demo below to see how our new mixin renders media queries:

[Play with this gist on SassMeister.](#)

Function Directives

Function directives in Sass are similar to mixins, but instead of returning markup, they return values via the `@return` directive. They can be used to DRY (Don't repeat yourself) up your code and make everything more readable.

Lets go ahead and create a function directive to clean up our grid calculations from our grid demo:

```
@function getColumnWidth($width, $columns,$margin){  
  @return ($width / $columns) - ($margin * 2);  
}
```

Copy

Now we can use this function in our code below:

```
$container-width: 100%;  
$column-count: 4;  
$margin: 1%;  
  
.container {  
  width: $container-width;  
}  
  
.column {  
  background: #1abc9c;  
  height: 200px;  
  display: block;  
  float: left;  
  width: getColumnWidth($container-width,$column-count,$margin);  
  margin: 0 $margin;  
}
```

Copy

Pretty cool, eh?

Demo

Now that we have all these tools at our disposal, how about we build our own configurable grid framework? Lets roll:

Lets begin by creating a map of settings:

```
$settings: (  
  maxWidth: 800px,  
  columns: 12,  
  margin: 15px,  
  breakpoints: (  
    xs: "(max-width : 480px)",  
    sm: "(max-width : 768px) and (min-width: 481px)",  
    md: "(max-width : 1024px) and (min-width: 769px)",  
    lg: "(min-width : 1025px)"  
  )  
);
```

Copy

Next lets write a mixin that renders our framework:

```
@mixin renderGridStyles($settings){  
  
}
```

Copy

We are going to need to render markup for each breakpoint, so lets iterate through our breakpoints and call our media mixin. Lets use the `map-get` method to get our breakpoint values, and our `@each` directive to iterate through our breakpoints:


```
@mixin renderGridStyles($settings){
  $breakpoints: map-get($settings, "breakpoints");
  @each $key, $breakpoint in $breakpoints {
    @include media($breakpoint) {

    }
  }
}
```

Copy

We need to render the actual grid markup within our iteration, so let's create a `renderGrid` mixin. Let's use the `map-get` method to get our map values, and our `@while` directive to iterate through columns with `$i` as our index. We render our class name using interpolation.

```
@mixin renderGrid($key, $settings) {
  $i: 1;
  @while $i ≤ map-get($settings, "columns") {
    .col-#{ $key }-#{ $i } {
      float: left;
      width: 100% * $i / map-get($settings, "columns");
    }
    $i: $i+1;
  }
}
```

Copy

Next, let's add container and row styles:

```
.container {
  padding-right: map-get($settings, "margin");
  padding-left: map-get($settings, "margin");
  margin-right: auto;
  margin-left: auto;
}
```

Copy

```
}  
  
.row {  
  margin-right: map-get($settings, "margin") * -1;  
  margin-left: map-get($settings, "margin") * -1;  
}
```

It's alive! Check out the demo of our framework below:

[Play with this gist on SassMeister.](#)

Conclusion

You may reach this point and think that we have covered quite a bit of Sass, but really it is just the tip of the iceberg. Sass is an extremely powerful tool that you can do some really incredible things with. I look forward to following up with an article on advanced concepts, but until then Happy Sassing and check out some of the resources below:

Resources

- [The Sass Way](#) - A phenomenal source of Sass tutorials.
 - [Hugo Giraudel](#) - An amazing tech writer & Sass wizard with a keen focus on Sass.
 - [SassNews](#) - A Twitter account managed by Stuart Robson that will keep you in the know.
-

Want to learn more? Join the DigitalOcean Community!

Join our DigitalOcean community of over a million developers for free! Get help and share knowledge in our Questions & Answers section, find tutorials and tools that will help you grow as a developer and scale your project or business, and subscribe to topics of interest.

Sign up →

About the authors



[Ken Wheeler](#) Author

Developer and author at DigitalOcean.


Still looking for an answer?

Ask a question


Search for more help

Comments

Leave a comment



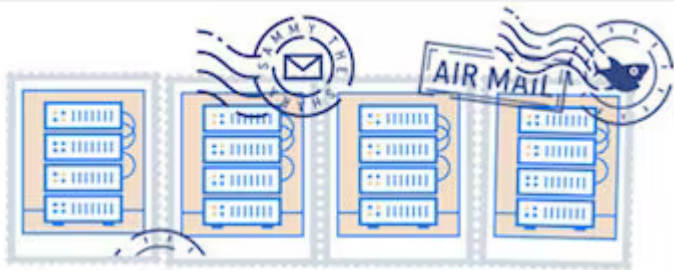
Leave a comment...



Login to Comment

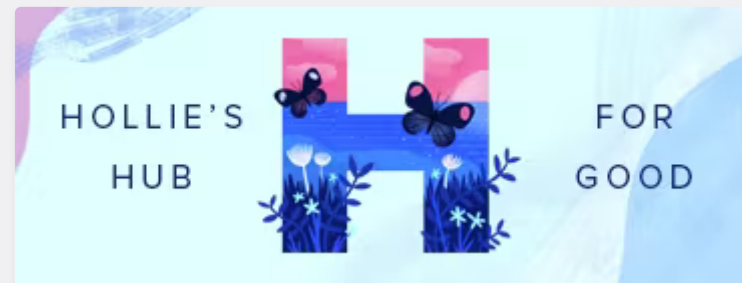


This work is licensed under a Creative Commons Attribution-NonCommercial- ShareAlike 4.0 International License.



GET OUR BIWEEKLY NEWSLETTER

Sign up for Infrastructure as a
Newsletter.



HOLLIE'S
HUB

FOR
GOOD

HOLLIE'S HUB FOR GOOD

Working on improving health and
education, reducing inequality,
and spurring economic growth?
We'd like to help.





BECOME A CONTRIBUTOR

You get paid; we donate to tech nonprofits.

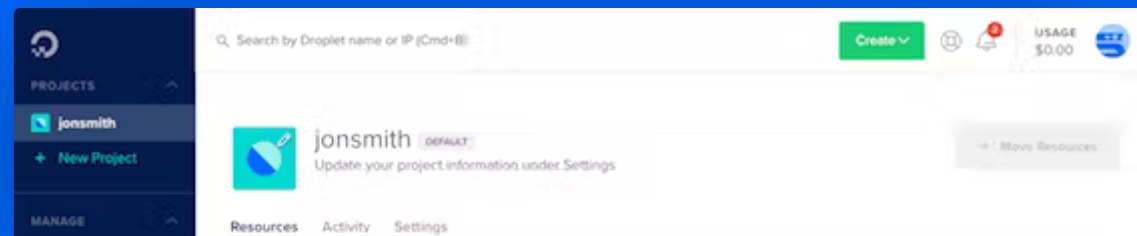
[Featured on Community](#) [Kubernetes Course](#) [Learn Python 3](#) [Machine Learning in Python](#) [Getting started with Go](#) [Intro to Kubernetes](#)

[DigitalOcean Products](#) [Virtual Machines](#) [Managed Databases](#) [Managed Kubernetes](#) [Block Storage](#) [Object Storage](#) [Marketplace](#) [VPC Load Balancers](#)

Welcome to the developer cloud

DigitalOcean makes it simple to launch in the cloud and scale up as you grow – whether you're running one virtual machine or ten thousand.

[Learn More](#)





© 2022 DigitalOcean, LLC. All rights reserved.

Company

[About](#)
[Leadership](#)
[Blog](#)
[Careers](#)
[Partners](#)
[Referral Program](#)
[Press](#)
[Legal](#)
[Security & Trust Center](#)

Contact

[Get Support](#)
[Trouble Signing In?](#)
[Sales](#)
[Report Abuse](#)
[System Status](#)
[Share your ideas](#)

Products

[Pricing](#)
[Products Overview](#)
[Droplets](#)
[Kubernetes](#)
[Managed Databases](#)
[Spaces](#)
[Marketplace](#)
[Load Balancers](#)
[Block Storage](#)
[API Documentation](#)
[Documentation](#)
[Release Notes](#)

Community

[Tutorials](#)
[Q&A](#)
[Tools and Integrations](#)
[Tags](#)
[Write for DigitalOcean](#)
[Presentation Grants](#)
[Hatch Startup Program](#)
[Shop Swag](#)
[Research Program](#)
[Open Source](#)
[Code of Conduct](#)

