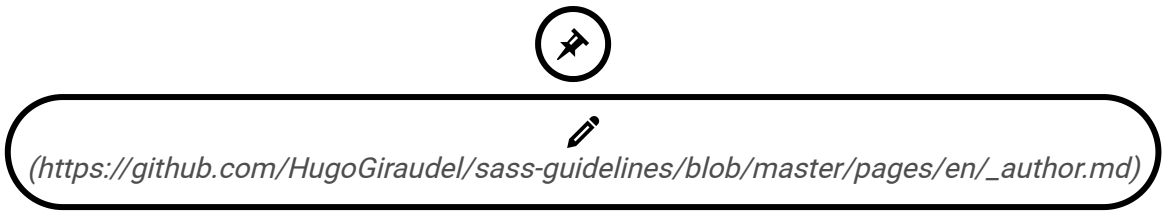
Sass Guidelines

An opinionated styleguide for writing sane, maintainable and scalable Sass.

About The Author



(https://github.com/HugoGiraudel/sass-guidelines/blob/master/pages/en/_author.md)

My name is Hugo Giraudel *(http://hugogiraudel.com)*, I am a 26 year-old French front-end developer, based in Berlin (Germany) since 2015, currently working at N26 *(https://n26.com)*.

I have been writing Sass for several years now and I am the author of many Sass-related projects such as SassDoc *(http://sassdoc.com)*, SitePoint Sass Reference *(http://sitepoint.com/sass-reference/)* and Sass-Compatibility *(http://sass-compatibility.github.io)*. If you are interested in more of my contributions to the Sass community, have a look at that list *(http://github.com/HugoGiraudel/awesome-sass)*.

I also happen to be the author of a book about CSS (in French) entitled CSS3 Pratique du Design Web *(http://css3-pratique.fr/)* (*Eyrolles* editions), as well as a book about Sass (in English) entitled Jump Start Sass *(https://learnable.com/books/jump-start-sass)* (*Learnable* editions).

Contributing

Sass Guidelines is a free project that I maintain in my spare time. Needless to say, it is quite a large amount of work to keep everything up-to-date, documented and relevant. Thankfully, I get helped by a lot of great contributors, especially when it comes to maintaining dozens of different translations. Be sure to thank them!

Now if you feel like contributing, please know that tweeting about it, spreading the word, or fixing a tiny typo by opening an issue or a pull-request on the GitHub repository *(https://github.com/HugoGiraudel/sass-guidelines)* would be great!

Last but not least before we start: if you enjoyed this document, or if it is useful for you or your team, please consider supporting it so I can keep working on it!

About Sass

This is how Sass *(http://sass-lang.com)* describes itself in its documentation *(http://sass-lang.com/documentation/file.SASS_REFERENCE.html)*:

> *Sass is an extension of CSS that adds power and elegance to the basic language.*

Sass' ultimate objective is to fix CSS' flaws. CSS, as we all know, is not the best language in the world [citation needed]. While very simple to learn, it can quickly get quite messy, especially on large projects.

This is where Sass comes in, as a meta-language, to improve CSS' syntax in order to provide extra features and handy tools. Meanwhile, Sass wants to be conservative regarding the CSS language.

The point is not to turn CSS into a fully featured programming language; Sass only wants to help where CSS fails. Because of this, getting started with Sass is no harder than learning CSS: it simply adds a couple of extra features *(http://sitepoint.com/sass-reference/)* on top of it.

That being said, there are many ways to use these features. Some good, some bad, some unusual. These guidelines are meant to give you a consistent and documented approach to writing Sass code.

Ruby Sass Or LibSass

Sass' first commit *(https://github.com/hcatlin/sass/commit/fa5048ba405619273e474a50400c7243fbff54fe)* goes back as far as late 2006, more than 10 years ago. Needless to say it has come a long way since then. Initially developed in Ruby, varied ports popped up here and there. The most successful one, LibSass *(http://webdesign.tutsplus.com/articles/getting-to-know-libsass--cms-23114)* (written in C/C++) is now close to being fully compatible with the original Ruby version.

In 2014, Ruby Sass and LibSass teams decided to wait for both versions to sync up before moving forward *(https://github.com/sass/libsass/wiki/The-LibSass-Compatibility-Plan)*. Since then, LibSass has been actively releasing versions to have feature-parity with its older sibling. The last remaining inconsistencies are gathered and listed by myself under the Sass-Compatibility *(http://sass-compatibility.github.io)* project. If you are aware of an incompatibility between the two versions that is not listed, please be kind enough to open an issue.

Coming back to choosing your compiler. Actually, it all depends on your project. If it is a Ruby on Rails project, you better use Ruby Sass, which is

perfectly suited for such a case. Also, be aware that Ruby Sass will always be the reference implementation and will always lead LibSass in features. And if you are looking to switch from Ruby Sass to LibSass *(http://www.sitepoint.com/switching-ruby-sass-libsass/)*, this article is for you.

On non-Ruby projects that need a workflow integration, LibSass is probably a better idea since it is mostly dedicated to being wrapped. So if you want to use, let's say Node.js, node-sass *(https://github.com/sass/node-sass)* is all chosen.

Sass Or SCSS

There is quite a lot of confusion regarding the semantics of the name *Sass*, and for good reason: Sass means both the preprocessor and its own syntax. Not very convenient, is it?

You see, Sass initially described a syntax of which the defining characteristic was its indentation-sensitivity. Soon enough, Sass maintainers decided to close the gap between Sass and CSS by providing a CSS-friendly syntax called *SCSS* for *Sassy CSS*. The motto is: if it's valid CSS, it's valid SCSS.

Since then, Sass (the preprocessor) has been providing two different syntaxes *(http://www.sitepoint.com/whats-difference-sass-scss/)*: Sass (not all-caps, please *(http://sassnotsass.com)*), also known as *the indented syntax*, and SCSS. Which one to use is pretty much up to you since both are strictly equivalent in features. It's only a matter of aesthetics at this point.

Sass' whitespace-sensitive syntax relies on indentation to get rid of braces, semi-colons and other punctuation symbols, leading to a leaner and shorter syntax. Meanwhile, SCSS is easier to learn since it's mostly some tiny extra bits on top of CSS.

I, myself, prefer SCSS over Sass because it is closer to CSS and friendlier to most developers. Because of that, SCSS is the default syntax throughout these guidelines. You can switch to Sass indented syntax in the side panel.

Other Preprocessors

Sass is a preprocessor among others. Its most serious competitor has to be Less *(http://lesscss.org/)*, a Node.js based preprocessor that has gotten quite popular thanks to the famous CSS framework Bootstrap *(http://getbootstrap.com/)* using it (until version 4). There is also Stylus *(http://learnboost.github.io/stylus/)*, a very permissive and flexible preprocessor however slightly harder to use and with a smaller community.

Why choose Sass over any other preprocessor? is still a valid question today. Not so long ago, we used to recommend Sass for Ruby-based projects because it was first made in Ruby and played well with Ruby on Rails. Now that LibSass has caught up (mostly) with original Sass, this is no longer relevant advice.

What I do like with Sass is its conservative approach to CSS. Sass' design is based on strong principles: much of the design approach comes naturally out of the core teams' beliefs that a) adding extra features has a complexity cost that needs to be justified by usefulness and, b) it should be easy to reason about what a given block of styles is doing by looking at that block alone. Also, Sass has a much sharper attention to detail than other preprocessors. As far as I can tell, the core designers care deeply about supporting every corner-case of CSS compatibility and making sure every general behavior is consistent. In other words, Sass is a software aimed at solving actual issues; helping to provide useful functionality to CSS where CSS falls short.

Preprocessors aside, we should also mention tools like PostCSS *(https://github.com/postcss/postcss)* and cssnext *(https://cssnext.github.io/)* which have received significant exposure these last few months.

PostCSS is commonly (and incorrectly) referred to as a "postprocessor". The assumption, combined with the unfortunate name, is that PostCSS parses over CSS that has already been processed by a preprocessor. While it can work this way, it is not a requirement so PostCSS is actually just a "processor".

It lets you access "tokens" of your stylesheets (like selectors, properties and values), process these with JavaScript to perform some operation of any kind and compile the results to CSS. For example, the popular prefixing library Autoprefixer *(https://github.com/postcss/autoprefixer)* is built with PostCSS. It parses every rule to see if vendor prefixes are needed by referencing the

browser support tool CanIUse *(http://caniuse.com)* and then removes and adds vendor prefixes that are needed.

This is incredibly powerful and great for building libraries that work with any preprocessor (as well as vanilla CSS), but PostCSS isn't particularly easy to use yet. You have to know a bit of JavaScript to build anything with it, and its API can be confusing at times. While Sass only provides a set of features that are useful to write CSS, PostCSS provides direct access to the CSS AST (*abstract syntax tree*) and JavaScript.

In short, Sass is somewhat easy and will solve most of your problems. On the other hand, PostCSS can be difficult to take in hand (if you aren't great with JavaScript) but turns out to be incredibly powerful. There's no reason why you can't and shouldn't use both. In fact, PostCSS offers an official SCSS parser for just this thing.

Note — Thanks to Cory Simmons *(https://github.com/corysimmons)* for his help and expertise on this section.

Introduction

(https://github.com/HugoGiraudel/sass-guidelines/blob/master/pages/en/_introduction.md)

Why A Styleguide

A styleguide is not just a pleasing document to read, picturing an ideal state for your code. It is a key document in a project's life, describing how and why code should be written. It may look like overkill for small projects, but it helps a lot in keeping the codebase clean, scalable and easily maintainable.

Needless to say, the more developers involved on a project, the more code guidelines are needed. Along the same lines, the bigger the project, the more a styleguide is a must.

Harry Roberts *(http://csswizardry.com)* states it very well in CSS Guidelines *(http://cssguidelin.es/#the-importance-of-a-styleguide)*:

> *A coding styleguide (note, not a visual styleguide) is a valuable tool for teams who:*
>
> - *build and maintain products for a reasonable length of time;*
> - *have developers of differing abilities and specialties;*
> - *have a number of different developers working on a product at any given time;*
> - *on-board new staff regularly;*
> - *have a number of codebases that developers dip in and out of.*

Disclaimer

First things first: **this is not a CSS styleguide**. This document will not discuss naming conventions for CSS classes, modular patterns and the question of IDs in the CSS world. These guidelines only aim at dealing with Sass-specific content.

Also, this styleguide is my own and therefore **very opinionated**. Think of it as a collection of methodologies and advice that I have polished and given over the years. It also gives me the opportunity to link to a handful of insightful resources, so be sure to check the *further readings*.

Obviously, this is certainly not the only way of doing things, and it may or may not suit your project. Feel free to pick from it and adapt it to your needs. As we say, *your mileage may vary*.

Key Principles

At the end of the day, if there is one thing I would like you to get from this whole styleguide, it is that **Sass should be kept as simple as it can be** *(http://www.sitepoint.com/keep-sass-simple/)*.

Thanks to my silly experiments like bitwise operators *(https://github.com/HugoGiraudel/SassyBitwise)*, iterators and generators *(https://github.com/HugoGiraudel/SassyIteratorsGenerators)* and a JSON parser *(https://github.com/HugoGiraudel/SassyJSON)* in Sass, we are all well aware of what one can do with this preprocessor.

Meanwhile, CSS is a simple language. Sass, being intended to write CSS, should not get much more complex than regular CSS. The KISS principle *(http://en.wikipedia.org/wiki/KISS_principle)* (Keep It Simple Stupid) is key here and may even take precedence over the DRY principle *(http://en.wikipedia.org/wiki/Don%27t_repeat_yourself)* (Don't Repeat Yourself) in some circumstances.

Sometimes, it's better to repeat a little to keep the code maintainable, rather than building a top-heavy, unwieldy, unnecessarily complicated system that is completely unmaintainable because it is overly complex.

Also, and let me quote Harry Roberts *(https://csswizardry.com)* once again, **pragmatism trumps perfection**. At some point, you will probably find yourself going against the rules described here. If it makes sense, if it feels right, do it. Code is just a means, not an end.

Extending The Guidelines

A large part of this styleguide is strongly opinionated. I have been reading and writing Sass for several years now, to the point where I now have a lot of principles when it comes to writing clean stylesheets. I understand that it might not please nor suit everybody, and this is perfectly normal.

Although, I believe that guidelines are made to be extended. Extending Sass Guidelines could be as simple as having a document stating that the code is following the guidelines from this styleguide except for a few things; in which case, specific rules would be explained below.

An example of styleguide extension can be found on the SassDoc repository *(https://github.com/SassDoc/sassdoc/blob/master/GUIDELINES.md)*:

> *This is an extension to Node Styleguide (https://github.com/felixge/node-style-guide) by Felix Geisendörfer. Anything from this document overrides what could be said in the Node Styleguide.*

Syntax & Formatting

(https://github.com/HugoGiraudel/sass-guidelines/blob/master/pages/en/_syntax.md)

If you ask me, the very first thing a styleguide should do is describe the way we want our code to look.

When several developers are involved in writing CSS on the same project(s), it is only a matter of time before one of them starts doing things their own way. Code guidelines that promote consistency not only prevent this, but also help when it comes to reading and updating the code.

Roughly, we want (shamelessly inspired by CSS Guidelines *(http://cssguidelin.es/#syntax-and-formatting)*):

- two (2) spaces indents, no tabs;
- ideally, 80-characters wide lines;
- properly written multi-line CSS rules;
- meaningful use of whitespace.

```
// Yep
.foo {
  display: block;
  overflow: hidden;
  padding: 0 1em;
}

// Nope
.foo {
    display: block; overflow: hidden;

    padding: 0 1em;
}
```

Strings

Believe it or not, strings play quite a large role in both CSS and Sass ecosystems. Most CSS values are either lengths or identifiers, so it actually is quite crucial to stick to some guidelines when dealing with strings in Sass.

ENCODING

To avoid any potential issue with character encoding, it is highly recommended to force UTF-8 *(http://en.wikipedia.org/wiki/UTF-8)* encoding in the main stylesheet using the `@charset` directive. Make sure it is the very first element of the stylesheet and there is no character of any kind before it.

```
@charset 'utf-8';
```

QUOTES

CSS does not require strings to be quoted, not even those containing spaces. Take font-family names for instance: it doesn't matter whether you wrap them in quotes for the CSS parser.

Because of this, Sass *also* does not require strings to be quoted. Even better (and *luckily*, you'll concede), a quoted string is strictly equivalent to its unquoted twin (e.g. `'abc'` is strictly equal to `abc`).

That being said, languages that do not require strings to be quoted are definitely a minority and so, **strings should always be wrapped with single quotes** (`'`) in Sass (single being easier to type than double on *qwerty* keyboards). Besides consistency with other languages, including CSS' cousin JavaScript, there are several reasons for this choice:

- color names are treated as colors when unquoted, which can lead to serious issues;
- most syntax highlighters will choke on unquoted strings;
- it helps general readability;
- there is no valid reason not to quote strings.

```
// Yep
$direction: 'left';

// Nope
$direction: left;
```

Note — As per the CSS specifications, the `@charset` directive should be declared in double quotes to be considered valid *(http://www.w3.org/TR/css3-syntax/#charset-rule)*. However, Sass takes care of this when compiling to CSS so the authoring has no impact on the final result. You can safely stick to single quotes, even for `@charset`.

STRINGS AS CSS VALUES

Specific CSS values (identifiers) such as `initial` or `sans-serif` require not to be quoted. Indeed, the declaration `font-family: 'sans-serif'` will silently fail because CSS is expecting an identifier, not a quoted string. Because of this, we do not quote those values.

```
// Yep
$font-type: sans-serif;

// Nope
$font-type: 'sans-serif';

// Okay I guess
$font-type: unquote('sans-serif');
```

Hence, we can make a distinction between strings intended to be used as CSS values (CSS identifiers) like in the previous example, and strings when sticking to the Sass data type, for instance map keys.

We don't quote the former, but we do wrap the latter in single quotes.

STRINGS CONTAINING QUOTES

If a string contains one or several single quotes, one might consider wrapping the string with double quotes (") instead, in order to avoid escaping characters within the string.

```
// Okay
@warn 'You can\'t do that.';

// Okay
@warn "You can't do that.";
```

URLS

URLs should be quoted as well, for the same reasons as above:

```
// Yep
.foo {
  background-image: url('/images/kittens.jpg');
}

// Nope
.foo {
  background-image: url(/images/kittens.jpg);
}
```

Numbers

In Sass, number is a data type including everything from unitless numbers to lengths, durations, frequencies, angles and so on. This allows calculations to be run on such measures.

ZEROS

Numbers should display leading zeros before a decimal value less than one. Never display trailing zeros.

```
// Yep
.foo {
  padding: 2em;
  opacity: 0.5;
}

// Nope
.foo {
  padding: 2.0em;
  opacity: .5;
}
```

Note — In Sublime Text and other editors providing a regular-expression powered search and replace, it is very easy to add a leading zero to (most if not all) float numbers. Simply replace `\s+\.(\d+)` with `\ 0.$1`. Do not forget the space before the `0` though.

UNITS

When dealing with lengths, a `0` value should never ever have a unit.

```
// Yep
$length: 0;

// Nope
$length: 0em;
```

Note — Beware, this practice should be limited to lengths only. Having a unitless zero for a time property such as `transition-delay` is not allowed. Theoretically, if a unitless zero is specified for a duration, the declaration is deemed invalid and should be discarded. Not all browsers are that strict, but some are. Long story short: only omit the unit for lengths.

The most common mistake I can think of regarding numbers in Sass, is thinking that units are just some strings that can be safely appended to a number. While that sounds true, it is certainly not how units work. Think of units as algebraic symbols. For instance, in the real world, multiplying 5 inches by 5 inches gives you 25 square inches. The same logic applies to Sass.

To add a unit to a number, you have to multiply this number by *1 unit*.

```
$value: 42;

// Yep
$length: $value * 1px;

// Nope
$length: $value + px;
```

Note that adding *0 member of that unit* also works, but I would rather recommend the aforementioned method since adding *0 unit* can be a bit confusing. Indeed, when trying to convert a number to another compatible unit, adding 0 will not do the trick. More on that in this article *(http://css-tricks.com/snippets/sass/correctly-adding-unit-number/)*.

```
$value: 42 + 0px;
// -> 42px

$value: 1in + 0px;
// -> 1in

$value: 0px + 1in;
// -> 96px
```

In the end, it really depends on what you are trying to achieve. Just keep in mind that adding the unit as a string is not a good way to proceed.

To remove the unit of a value, you have to divide it by *one unit of its kind*.

```
$length: 42px;

// Yep
$value: $length / 1px;

// Nope
$value: str-slice($length + unquote(''), 1, 2);
```

Appending a unit as a string to a number results in a string, preventing any additional operation on the value. Slicing the numeric part of a number with a unit also results in a string. This is not what you want. Use lengths, not strings *(http://hugogiraudel.com/2013/09/03/use-lengths-not-strings/)*.

CALCULATIONS

Top-level numeric calculations should always be wrapped in parentheses. Not only does this requirement dramatically improve readability, it also prevents some edge cases by forcing Sass to evaluate the contents of the parentheses.

```
// Yep
.foo {
  width: (100% / 3);
}

// Nope
.foo {
  width: 100% / 3;
}
```

MAGIC NUMBERS

“Magic number” is an old school programming *(http://en.wikipedia.org/wiki/Magic_number_(programming)#Unnamed_numerical_constants)* term for *unnamed numerical constant*. Basically, it’s just a

random number that happens to *just work*™ yet is not tied to any logical explanation.

Needless to say **magic numbers are a plague and should be avoided at all costs**. When you cannot manage to find a reasonable explanation for why a number works, add an extensive comment explaining how you got there and why you think it works. Admitting you don't know why something works is still more helpful to the next developer than them having to figure out what's going on from scratch.

```
/**
 * 1. Magic number. This value is the lowest I could f
 * `.foo` with its parent. Ideally, we should fix it p
 */
.foo {
  top: 0.327em; /* 1 */
}
```

On topic, CSS-Tricks has a terrific article *(http://css-tricks.com/magic-numbers-in-css/)* about magic numbers in CSS that I encourage you to read.

Colors

Colors occupy an important place in the CSS language. Naturally, Sass ends up being a valuable ally when it comes to manipulating colors, mostly by providing a handful of powerful functions *(http://sass-lang.com/documentation/Sass/Script/Functions.html)*.

Sass is so useful when it comes to manipulating colors that articles have flourished all over the internet about this very topic. May I recommend a few reads:

- How to Programmatically Go From One Color to Another *(http://thesassway.com/advanced/how-to-programtically-go-from-one-color-to-another-in-sass)*
- Using Sass to Build Color Palettes *(http://www.sitepoint.com/using-sass-build-color-palettes/)*

- Dealing with Color Schemes in Sass *(http://www.sitepoint.com/dealing-color-schemes-sass/)*

COLOR FORMATS

In order to make colors as simple as they can be, my advice would be to respect the following order of preference for color formats:

1. HSL notation *(http://en.wikipedia.org/wiki/HSL_and_HSV)*;
2. RGB notation *(http://en.wikipedia.org/wiki/RGB_color_model)*;
3. Hexadecimal notation (lowercase and shortened).

CSS color keywords should not be used, unless for rapid prototyping. Indeed, they are English words and some of them do a pretty bad job at describing the color they represent, especially for non-native speakers. On top of that, keywords are not perfectly semantic; for instance `grey` is actually darker than `darkgrey`, and the confusion between `grey` and `gray` can lead to inconsistent usages of this color.

The HSL representation is not only the easiest one for the human brain to comprehend[citation needed], it also makes it easy for stylesheet authors to tweak the color by adjusting the hue, saturation and lightness individually.

RGB still has the benefit of showing right away if the color is more of a blue, a green or a red. Therefore it might be better than HSL in some situations, especially when describing a pure red, green or blue. Although it does not make it easy to build a color from the three parts.

Lastly, hexadecimal is close to indecipherable for the human mind. Use it only as a last resort if you have to.

```
// Yep
.foo {
  color: hsl(0, 100%, 50%);
}

// Also yep
.foo {
  color: rgb(255, 0, 0);
}

// Meh
.foo {
  color: #f00;
}

// Nope
.foo {
  color: #FF0000;
}

// Nope
.foo {
  color: red;
}
```

When using HSL or RGB notation, always add a single space after a comma (`,`) and no space between parentheses (`(`, `)`) and content.

```
// Yep
.foo {
  color: rgba(0, 0, 0, 0.1);
  background: hsl(300, 100%, 100%);
}

// Nope
.foo {
  color: rgba(0,0,0,0.1);
  background: hsl( 300, 100%, 100% );
}
```

COLORS AND VARIABLES

When using a color more than once, store it in a variable with a meaningful name representing the color.

```
$sass-pink: hsl(330, 50%, 60%);
```

Now you are free to use this variable wherever you want. However, if your usage is strongly tied to a theme, I would advise against using the variable as is. Instead, store it in another variable with a name explaining how it should be used.

```
$main-theme-color: $sass-pink;
```

Doing this would prevent a theme change leading to something like `$sass-pink: blue`. This article *(http://davidwalsh.name/sass-color-variables-dont-suck)* does a good job at explaining why thinking your color variables through is important.

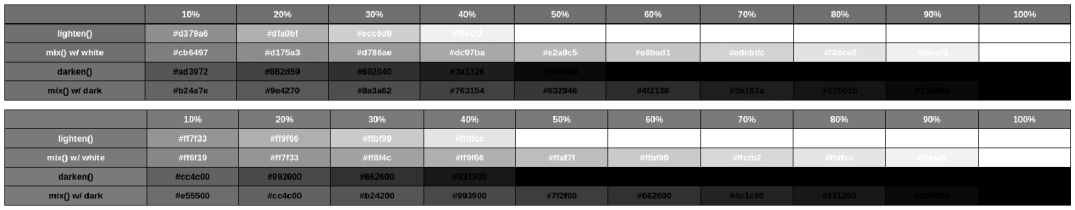
LIGHTENING AND DARKENING COLORS

Both `lighten` *(http://sass-lang.com/documentation/Sass/Script/Functions.html#lighten-instance_method)* and `darken` *(http://sass-lang.com/documentation/Sass/Script/Functions.html#darken-instance_method)* functions manipulate the lightness of a color in the HSL space by adding to or subtracting from the lightness in the HSL space. Basically, they are nothing but aliases for the `$lightness` parameter of the `adjust-color` *(http://sass-lang.com/documentation/Sass/Script/Functions.html#adjust_color-instance_method)* function.

The thing is, those functions often do not provide the expected result. On the other hand, the `mix` *(http://sass-lang.com/documentation/Sass/Script/Functions.html#mix-instance_method)* function is a nice way to lighten or darken a color by mixing it with either `white` or `black`.

The benefit of using `mix` rather than one of the two aforementioned functions is that it will progressively go to black (or white) as you decrease the

proportion of the color, whereas `darken` and `lighten` will quickly blow out a color all the way to black or white.



	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
lighten()	#d379a6	#dfa0bf	#ecc6d9	#f5e2f2						
mix() w/ white	#cb6497	#d175a3	#d786ae	#dc97ba	#e2a9c5	#e8bad1	#edcbdc	#f3dce8	#f8edf3	
darken()	#ad3972	#862d59	#602040	#3a1326	#160609					
mix() w/ dark	#b24a7e	#9e4270	#8a3a62	#763154	#632946	#4f2138	#3b182a	#27101b	#130809	

	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
lighten()	#ff7f33	#ff9f66	#ffbf99	#ffdfcc						
mix() w/ white	#ff6f19	#ff7f33	#ff8f4c	#ff9f66	#ffaf7f	#ffbf99	#ffcfb2	#ffdfcc	#ffefe5	
darken()	#cc4c00	#993900	#662600	#331300						
mix() w/ dark	#e55500	#cc4c00	#b24200	#993900	#7f2f00	#662500	#4c1c00	#321200	#190900	

Illustration of the difference between `lighten/darken` and `mix` by KatieK (http://codepen.io/KatieK2/pen/tejhz/)

If you don't want to write the `mix` function every time, you can create two easy-to-use functions `tint` and `shade` (which are also a part of Compass (http://compass-style.org/reference/compass/helpers/colors/#shade)) to do the same thing:

```
/// Slightly lighten a color
/// @access public
/// @param {Color} $color - color to tint
/// @param {Number} $percentage - percentage of `$colc
/// @return {Color}
@function tint($color, $percentage) {
  @return mix(white, $color, $percentage);
}

/// Slightly darken a color
/// @access public
/// @param {Color} $color - color to shade
/// @param {Number} $percentage - percentage of `$colc
/// @return {Color}
@function shade($color, $percentage) {
  @return mix(black, $color, $percentage);
}
```

Note — The `scale-color` (http://sass-lang.com/documentation/Sass/Script/Functions.html#scale_color-instance_method) function is designed to scale properties more fluidly by taking into account how high or low they already are. It should

provide results that are as nice as `mix`'s but with a clearer calling convention. The scaling factor isn't exactly the same though.

Lists

Lists are the Sass equivalent of arrays. A list is a flat data structure (unlike maps) intended to store values of any type (including lists, leading to nested lists).

Lists should respect the following guidelines:

- either inlined or multilines;
- necessarily on multilines if too long to fit on an 80-character line;
- unless used as is for CSS purposes, always comma separated;
- always wrapped in parenthesis;
- trailing comma if multilines, not if inlined.

```
// Yep
$font-stack: ('Helvetica', 'Arial', sans-serif);

// Yep
$font-stack: (
  'Helvetica',
  'Arial',
  sans-serif,
);

// Nope
$font-stack: 'Helvetica' 'Arial' sans-serif;

// Nope
$font-stack: 'Helvetica', 'Arial', sans-serif;

// Nope
$font-stack: ('Helvetica', 'Arial', sans-serif,);
```

When adding new items to a list, always use the provided API. Do not attempt to add new items manually.

```
$shadows: (0 42px 13.37px hotpink);

// Yep
$shadows: append($shadows, $shadow, comma);

// Nope
$shadows: $shadows, $shadow;
```

In this article *(http://hugogiraudel.com/2013/07/15/understanding-sass-lists/)*, I go through a lot of tricks and tips to handle and manipulate lists correctly in Sass.

Maps

With Sass, stylesheet authors can define maps — the Sass term for associative arrays, hashes or even JavaScript objects. A map is a data structure associating keys to values. Both keys and values can be of any data type, including maps although I would not recommend using complex data types as map keys, if only for the sake of sanity.

Maps should be written as follows:

- space after the colon (`:`);
- opening brace (`(`) on the same line as the colon (`:`);
- **quoted keys** if they are strings (which represents 99% of the cases);
- each key/value pair on its own new line;
- comma (`,`) at the end of each key/value;
- **trailing comma** (`,`) on last item to make it easier to add, remove or reorder items;
- closing brace (`)`) on its own new line;
- no space or new line between closing brace (`)`) and semi-colon (`;`).

Illustration:

```
// Yep
$breakpoints: (
  'small': 767px,
  'medium': 992px,
  'large': 1200px,
);

// Nope
$breakpoints: ( small: 767px, medium: 992px, large: 12
```

Write-ups about Sass maps are many given how longed-for this feature was. Here are 3 that I recommend: Using Sass Maps *(http://www.sitepoint.com/using-sass-maps/)*, Extra Map functions in Sass *(http://www.sitepoint.com/extra-map-functions-sass/)*, Real Sass, Real Maps *(http://blog.grayghostvisuals.com/sass/real-sass-real-maps/)*.

CSS Ruleset

At this point, this is mostly revising what everybody knows, but here is how a CSS ruleset should be written (at least, according to most guidelines, including CSS Guidelines *(http://cssguidelin.es/#anatomy-of-a-ruleset)*):

- related selectors on the same line; unrelated selectors on new lines;
- the opening brace (`{`) spaced from the last selector by a single space;
- each declaration on its own new line;
- a space after the colon (`:`);
- a trailing semi-colon (`;`) at the end of all declarations;
- the closing brace (`}`) on its own new line;
- a new line after the closing brace `}`.

Illustration:

```
// Yep
.foo, .foo-bar,
.baz {
  display: block;
  overflow: hidden;
  margin: 0 auto;
}

// Nope
.foo,
.foo-bar, .baz {
    display: block;
    overflow: hidden;
    margin: 0 auto }
```

Adding to those CSS-related guidelines, we want to pay attention to:

- local variables being declared before any declarations, then spaced from declarations by a new line;
- mixin calls with no `@content` coming before any declaration;
- nested selectors always coming after a new line;
- mixin calls with `@content` coming after any nested selector;
- no new line before a closing brace (`}`).

Illustration:

```
.foo, .foo-bar,
.baz {
  $length: 42em;

  @include ellipsis;
  @include size($length);
  display: block;
  overflow: hidden;
  margin: 0 auto;

  &:hover {
    color: red;
  }

  @include respond-to('small') {
    overflow: visible;
  }
}
```

Declaration Sorting

I cannot think of many topics where opinions are as divided as they are regarding declaration sorting in CSS. Concretely, there are two factions here:

- sticking to the alphabetical order;
- ordering declarations by type (position, display, colors, font, miscellaneous…).

There are pros and cons for both ways. On one hand, alphabetical order is universal (at least for languages using the latin alphabet) so there is no argument about sorting one property before another. However, it seems extremely weird to me to see properties such as `bottom` and `top` not right next to each other. Why should animations appear before the display type? There are a lot of oddities with alphabetical ordering.

```
.foo {
  background: black;
  bottom: 0;
  color: white;
  font-weight: bold;
  font-size: 1.5em;
  height: 100px;
  overflow: hidden;
  position: absolute;
  right: 0;
  width: 100px;
}
```

On the other hand, ordering properties by type makes perfect sense. Every font-related declarations are gathered, `top` and `bottom` are reunited and reading a ruleset kind of feels like reading a short story. But unless you stick to some conventions like Idiomatic CSS *(https://github.com/necolas/idiomatic-css)*, there is a lot of room for interpretation in this way of doing things. Where would `white-space` go: font or display? Where does `overflow` belong exactly? What is the property order within a group (it could be alphabetical, oh the irony)?

```
.foo {
  height: 100px;
  width: 100px;
  overflow: hidden;
  position: absolute;
  bottom: 0;
  right: 0;
  background: black;
  color: white;
  font-weight: bold;
  font-size: 1.5em;
}
```

There is also another interesting subtree of type ordering called Concentric CSS *(https://github.com/brandon-rhodes/Concentric-CSS)*, that seems to be quite popular as well. Basically, Concentric CSS relies on the box-model to define an order: starts outside, moves inward.

```
.foo {
  width: 100px;
  height: 100px;
  position: absolute;
  right: 0;
  bottom: 0;
  background: black;
  overflow: hidden;
  color: white;
  font-weight: bold;
  font-size: 1.5em;
}
```

I must say I cannot decide myself. A recent poll on CSS-Tricks *(http://css-tricks.com/poll-results-how-do-you-order-your-css-properties/)* determined that over 45% developers order their declarations by type against 14% alphabetically. Also, there are 39% that go full random, including myself.

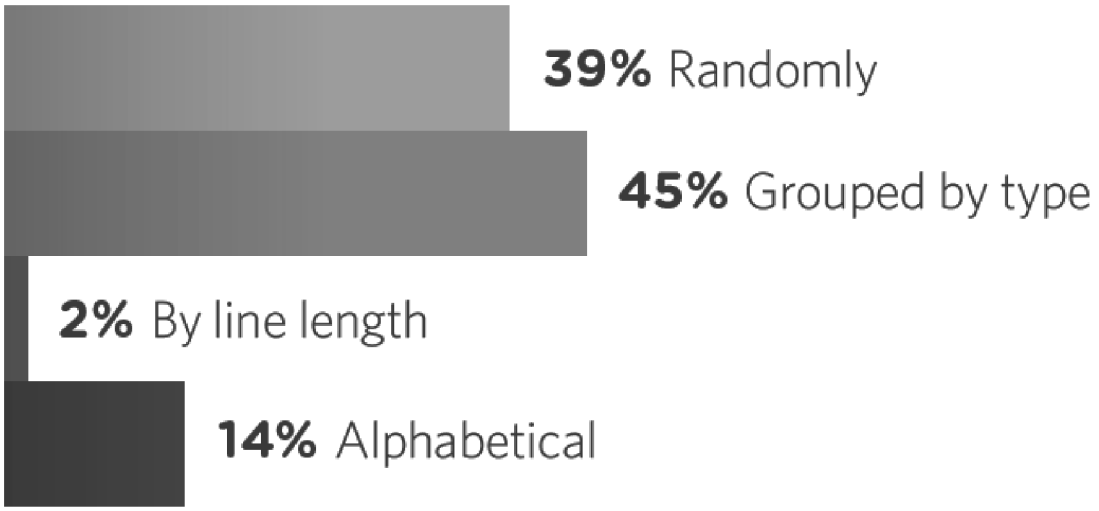


Chart showing how developers order their CSS declarations

Because of this, I will not impose a choice in this styleguide. Pick the one you prefer, as long as you are consistent throughout your stylesheets (i.e. not the *random* option).

Note — A recent study *(http://peteschuster.com/2014/12/reduce-file-size-css-sorting/)* shows that using CSS Comb *(https://github.com/csscomb/csscomb.js)* (which uses type ordering *(https://github.com/csscomb/csscomb.js/blob/master/config/cssco*

mb.json)) for sorting CSS declarations ends up shortening the average file size under Gzip compression by 2.7%, compared to 1.3% when sorting alphabetically.

Selector Nesting

One particular feature Sass provides that is being overly misused by many developers is *selector nesting*. Selector nesting offers a way for stylesheet authors to compute long selectors by nesting shorter selectors within each others.

GENERAL RULE

For instance, the following Sass nesting:

```
.foo {
  .bar {
    &:hover {
      color: red;
    }
  }
}
```

… will generate this CSS:

```
.foo .bar:hover {
  color: red;
}
```

Along the same lines, since Sass 3.3 it is possible to use the current selector reference (`&`) to generate advanced selectors. For instance:

```
.foo {
  &-bar {
    color: red;
  }
}
```

… will generate this CSS:

```
.foo-bar {
  color: red;
}
```

This method is often used along with BEM naming conventions *(http://csswizardry.com/2013/01/mindbemding-getting-your-head-round-bem-syntax/)* to generate `.block__element` and `.block--modifier` selectors based on the original selector (i.e. `.block` in this case).

> **Note —** While it might be anecdotal, generating new selectors from the current selector reference (`&`) makes those selectors unsearchable in the codebase since they do not exist per se.

The problem with selector nesting is that it ultimately makes code more difficult to read. One has to mentally compute the resulting selector out of the indentation levels; it is not always quite obvious what the CSS will end up being.

This statement becomes truer as selectors get longer and references to the current selector (`&`) more frequent. At some point, the risk of losing track and not being able to understand what's going on anymore is so high that it is not worth it.

To prevent such situations, we talked a lot about the Inception rule *(http://thesassway.com/beginner/the-inception-rule)* a few years back. It advised against nesting more than 3 levels deep, as a reference to the movie Inception from Christopher Nolan. I would be more drastic and recommend to **avoid selector nesting as much as possible**.

While there are obviously a few exceptions to this rule as we'll see in the next section, this opinion seems to be quite popular. You can read about it more in details in Beware of Selector Nesting *(http://www.sitepoint.com/beware-selector-nesting-sass/)* and Avoid nested selectors for more modular CSS *(http://thesassway.com/intermediate/avoid-nested-selectors-for-more-modular-css)*.

EXCEPTIONS

For starters, it is allowed and even recommended to nest pseudo-classes and pseudo-elements within the initial selector.

```
.foo {
  color: red;

  &:hover {
    color: green;
  }

  &::before {
    content: 'pseudo-element';
  }
}
```

Using selector nesting for pseudo-classes and pseudo-elements not only makes sense (because it deals with closely related selectors), it also helps keep everything about a component at the same place.

Also, when using component-agnostic state classes such as `.is-active`, it is perfectly fine to nest it under the component's selector to keep things tidy.

```
.foo {
  // …

  &.is-active {
    font-weight: bold;
  }
}
```

Last but not least, when styling an element because it happens to be contained within another specific element, it is also fine to use nesting to keep everything about the component at the same place.

```
.foo {
  // …

  .no-opacity & {
    display: none;
  }
}
```

As with everything, the specifics are somewhat irrelevant, consistency is key. If you feel fully confident with selector nesting, then use selector nesting. Just make sure your whole team is okay with that.

Naming Conventions

(https://github.com/HugoGiraudel/sass-guidelines/blob/master/pages/en/_naming.md)

In this section, we will not deal with the best CSS naming conventions for maintainability and scale; not only is that up to you, it's also out of the scope of a Sass styleguide. I suggest those recommended by CSS Guidelines *(http://cssguidelin.es/#naming-conventions)*.

There are a few things you can name in Sass, and it is important to name them well so the whole code base looks both consistent and easy to read:

- variables;
- functions;
- mixins.

Sass placeholders are deliberately omitted from this list since they can be considered as regular CSS selectors, thus following the same naming pattern as classes.

Regarding variables, functions and mixins, we stick to something very *CSS-y*: **lowercase hyphen-delimited**, and above all meaningful.

```
$vertical-rhythm-baseline: 1.5rem;

@mixin size($width, $height: $width) {
  // …
}

@function opposite-direction($direction) {
  // …
}
```

Constants

If you happen to be a framework developer or library writer, you might find yourself dealing with variables that are not meant to be updated in any circumstances: constants. Unfortunately (or fortunately?), Sass does not provide any way to define such entities, so we have to stick to strict naming conventions to make our point.

As for many languages, I suggest all-caps snakerized variables when they are constants. Not only is this a very old convention, but it also contrasts well with usual lowercased hyphenated variables.

```
// Yep
$CSS_POSITIONS: (top, right, bottom, left, center);

// Nope
$css-positions: (top, right, bottom, left, center);
```

If you really want to play with the ideas of constants in Sass, you should read this dedicated article *(http://www.sitepoint.com/dealing-constants-sass/)*.

Namespace

If you intend to distribute your Sass code, in the case of a library, a framework, a grid system or whatever, you might want to consider namespacing all your variables, functions, mixins and placeholders so it does not conflict with anyone else's code.

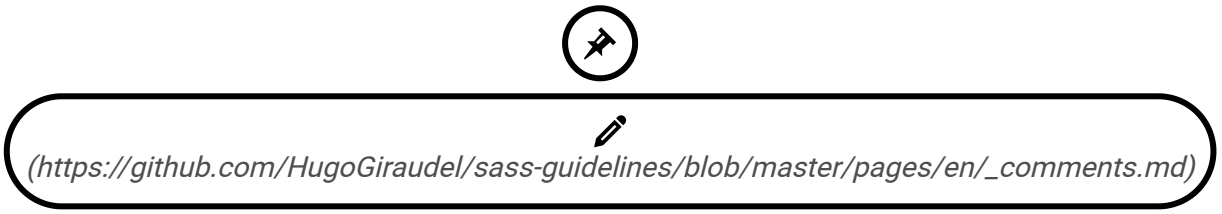
For instance, if you work on a *Sassy Unicorn* project that is meant to be distributed, you could consider using `su-` as a namespace. It is specific enough to prevent any naming collisions and short enough not to be a pain to write.

```
$su-configuration: ( … );

@function su-rainbow($unicorn) {
  // …
}
```

Kaelig *(http://kaelig.fr)* has a very insightful article about the global CSS namespace *(http://blog.kaelig.fr/post/44554267597/please-respect-the-global-css-namespace)*, in case this topic is of any interest to you.

Note — Note that automatic namespacing is definitely a design goal for the upcoming `@import` revamp from Sass 4.0. As that comes closer to fruition, it will become less and less useful to do manual namespacing; eventually, manually namespaced libraries may actually be harder to use.

Commenting



(https://github.com/HugoGiraudel/sass-guidelines/blob/master/pages/en/_comments.md)

CSS is a tricky language, full of hacks and oddities. Because of this, it should be heavily commented, especially if you or someone else intend to read and update the code 6 months or 1 year from now. Don't let you or anybody else be in the position of *I-didn't-write-this-oh-my-god-why*.

As simple as CSS can get, there is still a lot of room for comments. These could be explaining:

- the structure and/or role of a file;
- the goal of a ruleset;
- the idea behind a magic number;
- the reason for a CSS declaration;
- the order of CSS declarations;
- the thought process behind a way of doing things.

And I probably forgot a lot of other various reasons as well. Commenting takes very little time when done seamlessly along with the code so do it at the right time. Coming back at a piece of code to comment it is not only completely unrealistic but also extremely annoying.

Writing Comments

Ideally, *any* CSS ruleset should be preceded by a C-style comment explaining the point of the CSS block. This comment also hosts numbered explanations regarding specific parts of the ruleset. For instance:

```
/**
 * Helper class to truncate and add ellipsis to a stri
 * on a single line.
 * 1. Prevent content from wrapping, forcing it on a s
 * 2. Add ellipsis at the end of the line.
 */
.ellipsis {
  white-space: nowrap; /* 1 */
  text-overflow: ellipsis; /* 2 */
  overflow: hidden;
}
```

Basically everything that is not obvious at first glance should be commented. There is no such thing as too much documentation. Remember that you cannot *comment too much*, so get on fire and write comments for everything that is worth it.

When commenting a Sass-specific section, use Sass inline comments instead of a C-style block. This makes the comment invisible in the output,

even in expanded mode during development.

```
// Add current module to the list of imported modules.
// `!global` flag is required so it actually updates t
$imported-modules: append($imported-modules, $module)
```

Note that this way of doing things is also supported by CSS Guidelines in its Commenting *(http://cssguidelin.es/#commenting)* section.

Documentation

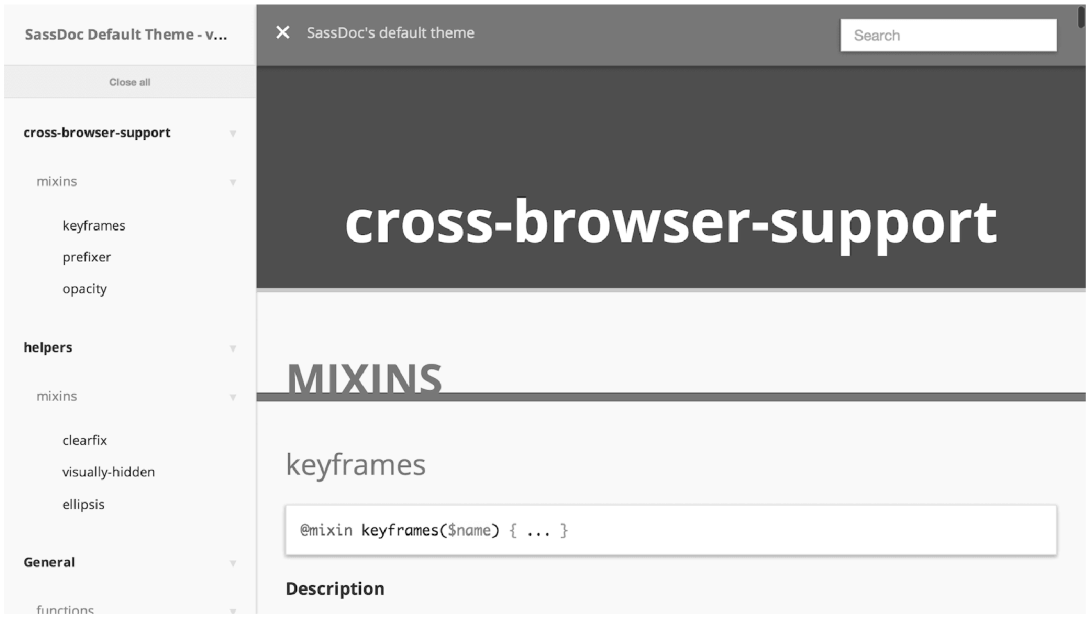
Every variable, function, mixin and placeholder that is intended to be reused all over the codebase should be documented as part of the global API using SassDoc *(http://sassdoc.com)*.

```
/// Vertical rhythm baseline used all over the code ba
/// @type Length
$vertical-rhythm-baseline: 1.5rem;
```

Note — Three slashes (`/`) required.

SassDoc has two major roles:

- forcing standardized comments using an annotation-based system for everything that is part of a public or private API;
- being able to generate an HTML version of the API documentation by using any of the SassDoc endpoints (CLI tool, Grunt, Gulp, Broccoli, Node…).



Documentation generated by SassDoc

Here is an example of a mixin extensively documented with SassDoc:

```
/// Mixin helping defining both `width` and `height` s
///
/// @author Hugo Giraudel
///
/// @access public
///
/// @param {Length} $width - Element's `width`
/// @param {Length} $height [$width] - Element's `heig
///
/// @example scss - Usage
///   .foo {
///     @include size(10em);
///   }
///
///   .bar {
///     @include size(100%, 10em);
///   }
///
/// @example css - CSS output
///   .foo {
///     width: 10em;
///     height: 10em;
///   }
///
///   .bar {
///     width: 100%;
///     height: 10em;
///   }
@mixin size($width, $height: $width) {
  width: $width;
  height: $height;
}
```

Architecture

(https://github.com/HugoGiraudel/sass-guidelines/blob/master/pages/en/_architecture.md)

Architecting a CSS project is probably one of the most difficult things you will have to do in a project's life. Keeping the architecture consistent and meaningful is even harder.

Fortunately, one of the main benefits of using a CSS preprocessor is having the ability to split the codebase over several files without impacting performance (like the `@import` CSS directive would do). Thanks to Sass's overload of the `@import` directive, it is perfectly safe (and actually recommended) to use as many files as necessary in development, all compiled into a single stylesheet when going to production.

On top of that, I cannot stress enough the need for folders, even on small scale projects. At home, you don't drop every sheet of paper into the same box. You use folders; one for the house/flat, one for the bank, one for bills, and so on. There is no reason to do otherwise when structuring a CSS project. Split the codebase into meaningful separated folders so it is easy to find stuff later when you have to come back to the code.

There are a lot of popular architectures *(http://www.sitepoint.com/look-different-sass-architectures/)* for CSS projects: OOCSS *(http://www.smashingmagazine.com/2011/12/12/an-introduction-to-object-oriented-css-oocss/)*, Atomic Design *(http://bradfrost.com/blog/post/atomic-web-design/)*, Bootstrap *(http://getbootstrap.com/)*-like, Foundation *(http://foundation.zurb.com/)*-like… They all have their merits, pros and cons.

I, myself, use an approach that happens to be quite similar to SMACSS *(https://smacss.com/)* from Jonathan Snook *(http://snook.ca/)*, which focuses on keeping things simple and obvious.

> **Note —** I have learnt that architecture is most of the time very specific to the project. Feel free to discard completely or adapt the proposed solution so that you deal with a system that suits your needs.

Components

There is a major difference between making it *work*, and making it *good*. Again, CSS is quite a messy language [citation needed]. The less CSS we have, the merrier. We don't want to deal with megabytes of CSS code. To keep stylesheets short and efficient—and this will not be any surprise to you—it is usually a good idea to think of an interface as a collection of components.

Components can be anything, as long as they:

- do one thing and one thing only;
- are re-usable and re-used across the project;
- are independent.

For instance, a search form should be treated as a component. It should be reusable, at different positions, on different pages, in various situations. It should not depend on its position in the DOM (footer, sidebar, main content…).

Most of any interface can be thought of as little components and I highly recommend you stick to this paradigm. This will not only shorten the amount of CSS needed for the whole project, but also happens to be much easier to maintain than a chaotic mess where everything is flustered.

Component Structure

Ideally, components should exist in their own Sass partial (within the `components/` folder, as is described in the 7-1 pattern), such as `components/_button.scss`. The styles described in each component file should only be concerned with:

- the style of the component itself;
- the style of the component's variants, modifiers, and/or states;
- the styles of the component's descendents (i.e. children), if necessary.

If you want your components to be able to be themed externally (e.g. from a theme inside the `themes/` folder), limit the declarations to only structural

styles, such as dimensions (width/height), padding, margins, alignment, etc. Exclude styles such as colors, shadows, font rules, background rules, etc.

A component partial can include component-specific variables, placeholders, and even mixins and functions. Keep in mind, though, that you should avoid referencing (i.e. `@import`-ing) component files from other component files, as this can make your project's dependency graph an unmaintainable tangled mess.

Here is an example of a button component partial:

```
// Button-specific variables
$button-color: $secondary-color;

// … include any button-specific:
// - mixins
// - placeholders
// - functions

/**
 * Buttons
 */
.button {
  @include vertical-rhythm;
  display: block;
  padding: 1rem;
  color: $button-color;
  // … etc.

  /**
   * Inlined buttons on large screens
   */
  @include respond-to('medium') {
    display: inline-block;
  }
}

/**
 * Icons within buttons
 */
.button > svg {
  fill: currentcolor;
  // … etc.
}

/**
 * Inline button
 */
.button--inline {
  display: inline-block;
}
```

Note — Thanks to David Khourshid *(https://twitter.com/davidkpiano)* for his help and expertise on this section.

The 7-1 Pattern

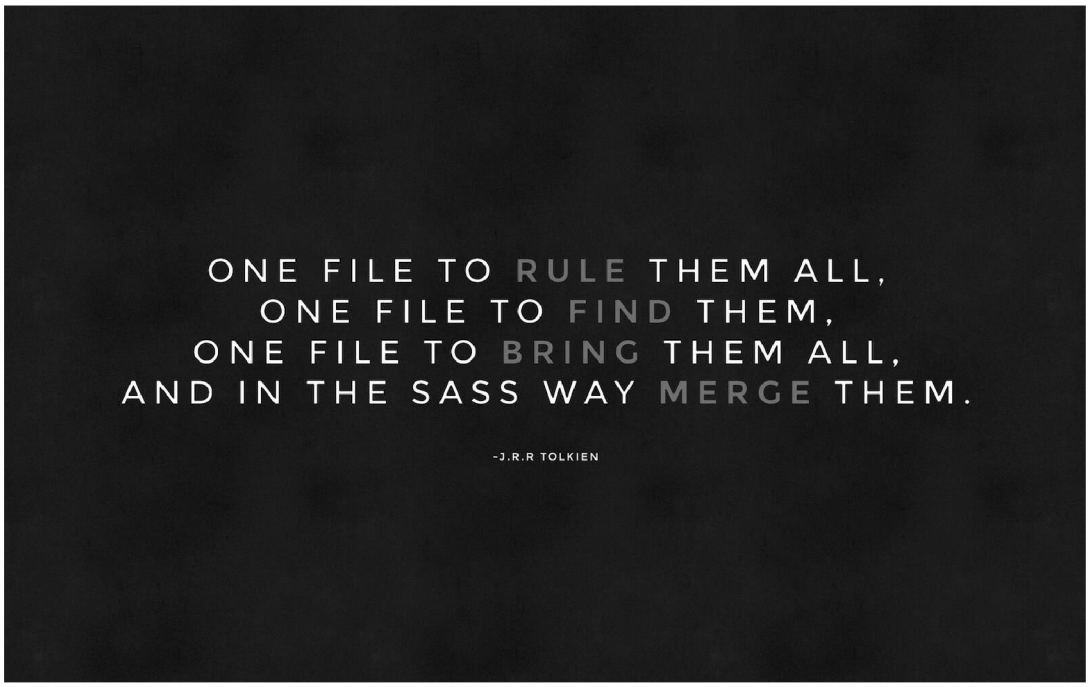
Back to architecture, shall we? I usually go with what I call the *7-1 pattern*: 7 folders, 1 file. Basically, you have all your partials stuffed into 7 different folders, and a single file at the root level (usually named `main.scss`) which imports them all to be compiled into a CSS stylesheet.

- `base/`
- `components/`
- `layout/`
- `pages/`
- `themes/`
- `abstracts/`
- `vendors/`

And of course:

- `main.scss`

Note — If you are looking to use the 7-1 pattern, there is a boilerplate *(https://github.com/HugoGiraudel/sass-boilerplate)* ready on GitHub. It should contain everything you need to get started with this architecture.



Wallpaper by Julien He (https://twitter.com/julien_he)

Ideally, we can come up with something like this:

```
sass/
|
|- abstracts/
|   |- _variables.scss    # Sass Variables
|   |- _functions.scss    # Sass Functions
|   |- _mixins.scss       # Sass Mixins
|   |- _placeholders.scss # Sass Placeholders
|
|- base/
|   |- _reset.scss        # Reset/normalize
|   |- _typography.scss   # Typography rules
|   …                     # Etc.
|
|- components/
|   |- _buttons.scss      # Buttons
|   |- _carousel.scss     # Carousel
|   |- _cover.scss        # Cover
|   |- _dropdown.scss     # Dropdown
|   …                     # Etc.
|
|- layout/
|   |- _navigation.scss   # Navigation
|   |- _grid.scss         # Grid system
|   |- _header.scss       # Header
|   |- _footer.scss       # Footer
|   |- _sidebar.scss      # Sidebar
|   |- _forms.scss        # Forms
|   …                     # Etc.
|
|- pages/
|   |- _home.scss         # Home specific styles
|   |- _contact.scss      # Contact specific styles
|   …                     # Etc.
|
|- themes/
|   |- _theme.scss        # Default theme
|   |- _admin.scss        # Admin theme
|   …                     # Etc.
|
|- vendors/
```

```
|   |- _bootstrap.scss    # Bootstrap
|   |- _jquery-ui.scss    # jQuery UI
|   …                     # Etc.
|
`- main.scss              # Main Sass file
```

Note — Files follow the same naming conventions described above: they are hyphen-delimited.

BASE FOLDER

The `base/` folder holds what we might call the boilerplate code for the project. In there, you might find the reset file, some typographic rules, and probably a stylesheet defining some standard styles for commonly used HTML elements (that I like to call `_base.scss`).

- `_base.scss`
- `_reset.scss`
- `_typography.scss`

Note — If your project uses *a lot* of CSS animations, you might consider adding an `\_animations.scss` file in there containing the `@keyframes` definitions of all your animations. If you only use a them sporadically, let them live along the selectors that use them.

LAYOUT FOLDER

The `layout/` folder contains everything that takes part in laying out the site or application. This folder could have stylesheets for the main parts of the site (header, footer, navigation, sidebar…), the grid system or even CSS styles for all the forms.

- `_grid.scss`
- `_header.scss`
- `_footer.scss`
- `_sidebar.scss`

- `_forms.scss`
- `_navigation.scss`

> **Note —** The `layout/` folder might also be called `partials/`, depending on what you prefer.

COMPONENTS FOLDER

For smaller components, there is the `components/` folder. While `layout/` is *macro* (defining the global wireframe), `components/` is more focused on widgets. It contains all kind of specific modules like a slider, a loader, a widget, and basically anything along those lines. There are usually a lot of files in `components/` since the whole site/application should be mostly composed of tiny modules.

- `_media.scss`
- `_carousel.scss`
- `_thumbnails.scss`

> **Note —** The `components/` folder might also be called `modules/`, depending on what you prefer.

PAGES FOLDER

If you have page-specific styles, it is better to put them in a `pages/` folder, in a file named after the page. For instance, it's not uncommon to have very specific styles for the home page hence the need for a `_home.scss` file in `pages/`.

- `_home.scss`
- `_contact.scss`

> **Note —** Depending on your deployment process, these files could be called on their own to avoid merging them with the others in the

resulting stylesheet. It is really up to you.

THEMES FOLDER

On large sites and applications, it is not unusual to have different themes. There are certainly different ways of dealing with themes but I personally like having them all in a `themes/` folder.

- `_theme.scss`
- `_admin.scss`

> **Note —** This is very project-specific and is likely to be non-existent on many projects.

ABSTRACTS FOLDER

The `abstracts/` folder gathers all Sass tools and helpers used across the project. Every global variable, function, mixin and placeholder should be put in here.

The rule of thumb for this folder is that it should not output a single line of CSS when compiled on its own. These are nothing but Sass helpers.

- `_variables.scss`
- `_mixins.scss`
- `_functions.scss`
- `_placeholders.scss`

When working on a very large project with a lot of abstract utilities, it might be interesting to group them by topic rather than type, for instance typography (`_typography.scss`), theming (`_theming.scss`), etc. Each file contains all the related helpers: variables, functions, mixins and placeholders. Doing so can make the code easier to browse and maintain, especially when files are getting very long.

Note — The `abstracts/` folder might also be called `utilities/` or `helpers/`, depending on what you prefer.

VENDORS FOLDER

And last but not least, most projects will have a `vendors/` folder containing all the CSS files from external libraries and frameworks – Normalize, Bootstrap, jQueryUI, FancyCarouselSliderjQueryPowered, and so on. Putting those aside in the same folder is a good way to say “Hey, this is not from me, not my code, not my responsibility”.

- `_normalize.scss`
- `_bootstrap.scss`
- `_jquery-ui.scss`
- `_select2.scss`

If you have to override a section of any vendor, I recommend you have an 8th folder called `vendors-extensions/` in which you may have files named exactly after the vendors they overwrite.

For instance, `vendors-extensions/_bootstrap.scss` is a file containing all CSS rules intended to re-declare some of Bootstrap’s default CSS. This is to avoid editing the vendor files themselves, which is generally not a good idea.

MAIN FILE

The main file (usually labelled `main.scss`) should be the only Sass file from the whole code base not to begin with an underscore. This file should not contain anything but `@import` and comments.

Files should be imported according to the folder they live in, one after the other in the following order:

1. `abstracts/`
2. `vendors/`
3. `base/`
4. `layout/`

5. `components/`
6. `pages/`
7. `themes/`

In order to preserve readability, the main file should respect these guidelines:

- one file per `@import`;
- one `@import` per line;
- no new line between two imports from the same folder;
- a new line after the last import from a folder;
- file extensions and leading underscores omitted.

```
@import 'abstracts/variables';
@import 'abstracts/functions';
@import 'abstracts/mixins';
@import 'abstracts/placeholders';

@import 'vendors/bootstrap';
@import 'vendors/jquery-ui';

@import 'base/reset';
@import 'base/typography';

@import 'layout/navigation';
@import 'layout/grid';
@import 'layout/header';
@import 'layout/footer';
@import 'layout/sidebar';
@import 'layout/forms';

@import 'components/buttons';
@import 'components/carousel';
@import 'components/cover';
@import 'components/dropdown';

@import 'pages/home';
@import 'pages/contact';

@import 'themes/theme';
@import 'themes/admin';
```

There is another way of importing partials that I deem valid as well. On the bright side, it makes the file more readable. On the other hand, it makes updating it slightly more painful. Anyway, I'll let you decide which is best, it does not matter much. For this way of doing, the main file should respect these guidelines:

- one `@import` per folder;
- a linebreak after `@import`;
- each file on its own line;
- a new line after the last import from a folder;

- file extensions and leading underscores omitted.

```
@import
  'abstracts/variables',
  'abstracts/functions',
  'abstracts/mixins',
  'abstracts/placeholders';

@import
  'vendors/bootstrap',
  'vendors/jquery-ui';

@import
  'base/reset',
  'base/typography';

@import
  'layout/navigation',
  'layout/grid',
  'layout/header',
  'layout/footer',
  'layout/sidebar',
  'layout/forms';

@import
  'components/buttons',
  'components/carousel',
  'components/cover',
  'components/dropdown';

@import
  'pages/home',
  'pages/contact';

@import
  'themes/theme',
  'themes/admin';
```

About Globbing

In computer programming, glob patterns specify sets of filenames with wildcard characters, such as `*.scss`. To a general extent, globbing means matching a set of files based on an expression instead of a list of filenames. When applied to Sass, it means importing partials into the main file with a glob pattern rather than by listing them individually. This would lead to a main file looking like this:

```
@import 'abstracts/*';
@import 'vendors/*';
@import 'base/*';
@import 'layout/*';
@import 'components/*';
@import 'pages/*';
@import 'themes/*';
```

Sass does not support file globbing out of the box because it can be a dangerous feature as CSS is known to be order-dependant. When dynamically importing files (which usually goes in alphabetical order), one does not control the source order anymore, which can lead to hard to debug side-effects.

That being said, in a strict component-based architecture with extra care not to leak any style from one partial to the other, the order should not really matter anymore, which would allow for glob imports. This would make it easier to add or remove partials as carefully updating the main file would no longer be required.

When using Ruby Sass, there is a Ruby gem called sass-globbing *(https://github.com/chriseppstein/sass-globbing)* that enables exactly that behavior. If running on node-sass, one can rely either on Node.js, or whatever build tool they use to handle the compilation (Gulp, Grunt, etc.).

Shame File

There is an interesting concept that has been made popular by Harry Roberts *(http://csswizardry.com)*, Dave Rupert *(http://daverupert.com)* and Chris Coyier *(http://css-tricks.com)* that consists of putting all the CSS

declarations, hacks and things we are not proud of in a shame file *(http://csswizardry.com/2013/04/shame-css-full-net-interview/)*. This file, dramatically titled `_shame.scss`, would be imported after any other file, at the very end of the stylesheet.

```
/**
 * Nav specificity fix.
 *
 * Someone used an ID in the header code (`#header a {
 * nav selectors (`.site-nav a {}`). Use !important to
 * have time to refactor the header stuff.
 */
.site-nav a {
    color: #BADA55 !important;
}
```

Responsive Web Design And Breakpoints

(https://github.com/HugoGiraudel/sass-guidelines/blob/master/pages/en/_rwd.md)

I do not think we still have to introduce Responsive Web Design now that it is everywhere. However you might ask yourself *why is there a section about RWD in a Sass styleguide?* Actually there are quite a few things that can be done to make working with breakpoints easier, so I thought it would not be such a bad idea to list them here.

Naming Breakpoints

I think it is safe to say that media queries should not be tied to specific devices. For instance, this is definitely a bad idea to try targeting iPads or Blackberry phones specifically. Media queries should take care of a range of screen sizes, until the design breaks and the next media query takes over.

For the same reasons, breakpoints should not be named after devices but something more general. Especially since some phones are now bigger than tablets, some tablets bigger than some tiny screen computers, and so on…

```
// Yep
$breakpoints: (
  'medium': (min-width: 800px),
  'large': (min-width: 1000px),
  'huge': (min-width: 1200px),
);

// Nope
$breakpoints: (
  'tablet': (min-width: 800px),
  'computer': (min-width: 1000px),
  'tv': (min-width: 1200px),
);
```

At this point, any naming convention *(http://css-tricks.com/naming-media-queries/)* that makes crystal clear that a design is not intimately tied to a specific device type will do the trick, as long as it gives a sense of magnitude.

```
$breakpoints: (
  'seed': (min-width: 800px),
  'sprout': (min-width: 1000px),
  'plant': (min-width: 1200px),
);
```

Note — The previous examples uses nested maps to define breakpoints, however this really depends on what kind of breakpoint manager you use. You could opt for strings rather than inner maps for more flexibility (e.g. `'(min-width: 800px)'`).

Breakpoint Manager

Once you have named your breakpoints the way you want, you need a way to use them in actual media queries. There are plenty of ways to do so but I

must say I am a big fan of the breakpoint map read by a getter function. This system is both simple and efficient.

```
/// Responsive breakpoint manager
/// @access public
/// @param {String} $breakpoint - Breakpoint
/// @requires $breakpoints
@mixin respond-to($breakpoint) {
  $raw-query: map-get($breakpoints, $breakpoint);

  @if $raw-query {
    $query: if(
      type-of($raw-query) == 'string',
      unquote($raw-query),
      inspect($raw-query)
    );

    @media #{$query} {
      @content;
    }
  } @else {
    @error 'No value found for `#{$breakpoint}`. '
         + 'Please make sure it is defined in `$breakp
  }
}
```

Note — Obviously, this is a fairly simplistic breakpoint manager. If you need a slightly more permissive one, may I recommend you do not reinvent the wheel and use something that has been proven effective such as Sass-MQ *(https://github.com/sass-mq/sass-mq)*, Breakpoint *(http://breakpoint-sass.com/)* or include-media *(https://github.com/eduardoboucas/include-media)*.

If you are looking to read more on how to approach Media Queries in Sass, both SitePoint *(http://www.sitepoint.com/managing-responsive-breakpoints-sass/)* (from yours, truly) and CSS-Tricks *(http://css-tricks.com/approaches-media-queries-sass/)* have nice articles on this.

Media Queries Usage

Not so long ago, there was quite a hot debate about where media queries should be written: do they belong within selectors (as Sass allows it) or strictly dissociated from them? I have to say I am a fervent defender of the *media-queries-within-selectors* system, as I think it plays well with the ideas of *components*.

```
.foo {
  color: red;

  @include respond-to('medium') {
    color: blue;
  }
}
```

Leading to the following CSS output:

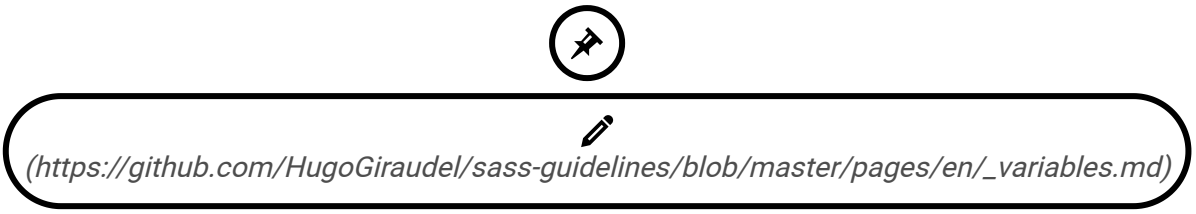
```
.foo {
  color: red;
}

@media (min-width: 800px) {
  .foo {
    color: blue;
  }
}
```

You might hear that this convention results in duplicated media queries in the CSS output. That is definitely true. Although, tests have been made *(http://sasscast.tumblr.com/post/38673939456/sass-and-media-queries)* and the final word is that it doesn't matter once Gzip (or any equivalent) has done its thing:

> *… we hashed out whether there were performance implications of combining vs scattering Media Queries and came to the conclusion that the difference, while ugly, is minimal at worst, essentially non-existent at best.*
> *— Sam Richards, regarding Breakpoint (http://sasscast.tumblr.com/post/38673939456/sass-and-media-queries)*

Now, if you really are concerned about duplicated media queries, you can still use a tool to merge them such as this gem *(https://github.com/aaronjensen/sass-media_query_combiner)* however I feel like I have to warn you against possible side-effects of moving CSS code around. You are not without knowing that source order is important.

Variables



(https://github.com/HugoGiraudel/sass-guidelines/blob/master/pages/en/_variables.md)

Variables are the essence of any programming language. They allow us to reuse values without having to copy them over and over again. Most importantly, they make updating a value very easy. No more find and replace or manual crawling.

However CSS is nothing but a huge basket containing all our eggs. Unlike many languages, there are no real scopes in CSS. Because of this, we have to pay real attention when adding variables at the risk of witnessing conflicts.

My advice would be to only create variables when it makes sense to do so. Do not initiate new variables for the heck of it, it won't help. A new variable should be created only when all of the following criteria are met:

- the value is repeated at least twice;
- the value is likely to be updated at least once;

- all occurrences of the value are tied to the variable (i.e. not by coincidence).

Basically, there is no point declaring a variable that will never be updated or that is only being used at a single place.

Scoping

Variable scoping in Sass has changed over the years. Until fairly recently, variable declarations within rulesets and other scopes were local by default. However when there was already a global variable with the same name, the local assignment would change the global variable. Since version 3.4, Sass now properly tackles the concept of scopes and create a new local variable instead.

The docs talk about *global variable shadowing*. When declaring a variable that already exists on the global scope in an inner scope (selector, function, mixin…), the local variable is said to be *shadowing* the global one. Basically, it overrides it just for the local scope.

The following code snippet explains the *variable shadowing* concept.

```
// Initialize a global variable at root level.
$variable: 'initial value';

// Create a mixin that overrides that global variable.
@mixin global-variable-overriding {
  $variable: 'mixin value' !global;
}

.local-scope::before {
  // Create a local variable that shadows the global c
  $variable: 'local value';

  // Include the mixin: it overrides the global variab
  @include global-variable-overriding;

  // Print the variable's value.
  // It is the **local** one, since it shadows the glc
  content: $variable;
}

// Print the variable in another selector that does nc
// It is the **global** one, as expected.
.other-local-scope::before {
  content: $variable;
}
```

`!default` Flag

When building a library, a framework, a grid system or any piece of Sass that is intended to be distributed and used by external developers, all configuration variables should be defined with the `!default` flag so they can be overwritten.

```
$baseline: 1em !default;
```

Thanks to this, a developer can define their own `$baseline` variable *before* importing your library without seeing their value redefined.

```
// Developer's own variable
$baseline: 2em;

// Your library declaring `$baseline`
@import 'your-library';

// $baseline == 2em;
```

`!global` Flag

The `!global` flag should only be used when overriding a global variable from a local scope. When defining a variable at root level, the `!global` flag should be omitted.

```
// Yep
$baseline: 2em;

// Nope
$baseline: 2em !global;
```

Multiple Variables Or Maps

There are advantages of using maps rather than multiple distinct variables. The main one is the ability to loop over a map, which is not possible with distinct variables.

Another pro of using a map is the ability to create a little getter function to provide a friendlier API. For instance, consider the following Sass code:

```
/// Z-indexes map, gathering all Z layers of the appli
/// @access private
/// @type Map
/// @prop {String} key - Layer's name
/// @prop {Number} value - Z value mapped to the key
$z-indexes: (
  'modal': 5000,
  'dropdown': 4000,
  'default': 1,
  'below': -1,
);

/// Get a z-index value from a layer name
/// @access public
/// @param {String} $layer - Layer's name
/// @return {Number}
/// @require $z-indexes
@function z($layer) {
  @return map-get($z-indexes, $layer);
}
```

Extend

(https://github.com/HugoGiraudel/sass-guidelines/blob/master/pages/en/_extend.md)

The `@extend` directive is a powerful feature that is frequently misunderstood. In general, it makes it possible to tell Sass to style a selector A as though it also matched selector B. Needless to say, this can be a valuable ally when writing modular CSS.

However, the true purpose of `@extend` is to maintain the relationships (constraints) within extended selectors between rulesets. What exactly does this mean?

- Selectors have *constraints* (e.g. `.bar` in `.foo > .bar` must have a parent `.foo`);
- These constraints are *carried over* to the extending selector (e.g. `.baz { @extend .bar; }` will produce `.foo > .bar, .foo > .baz`);
- The declarations of the extended selector will be shared with the extending selector.

Given that, it's straightforward to see how extending selectors with lenient constraints can lead to selector explosion. If `.baz .qux` extends `.foo .bar`, the resulting selector can be `.foo .baz .qux` or `.baz .foo .qux`, as both `.foo` and `.baz` are general ancestors. They can be parents, grandparents, etc.

Always try to define relationships via selector placeholders *(http://www.sitepoint.com/sass-reference/placeholders/)*, not actual selectors. This will give you the freedom to use (and change) any naming convention you have for your selectors, and since relationships are only defined once inside the placeholders, you are far less likely to produce unintended selectors.

For inheriting styles, only use `@extend` if the extending `.class` or `%placeholder` selector *is a kind of* the extended selector. For instance, an `.error` is a kind of `.warning`, so `.error` can `@extend .warning`.

```
%button {
  display: inline-block;
  // … button styles

  // Relationship: a %button that is a child of a %moc
  %modal > & {
    display: block;
  }
}

.button {
  @extend %button;
}

// Yep
.modal {
  @extend %modal;
}

// Nope
.modal {
  @extend %modal;

  > .button {
    @extend %button;
  }
}
```

There are many scenarios where extending selectors are helpful and worthwhile. Always keep in mind these rules so you can `@extend` with care:

- Use extend on `%placeholders` primarily, not on actual selectors.
- When extending classes, only extend a class with another class, *never* a complex selector *(http://www.w3.org/TR/selectors4/#syntax)*.
- Directly extend a `%placeholder` as few times as possible.
- Avoid extending general ancestor selectors (e.g. `.foo .bar`) or general sibling selectors (e.g. `.foo ~ .bar`). This is what causes selector explosion.

Note — It is often said that `@extend` helps with the file size since it combines selectors rather than duplicating properties. That is true, however the difference is negligible once Gzip *(http://en.wikipedia.org/wiki/Gzip)* has done its compression.

That being said, if you cannot use Gzip (or any equivalent) then switching to a `@extend` approach might be valuable, especially if stylesheet weight is your performance bottleneck.

Extend And Media Queries

You should only extend selectors within the same media scope (`@media` directive). Think of a media query as another constraint.

```
%foo {
  content: 'foo';
}

// Nope
@media print {
  .bar {
    // This doesn't work. Worse: it crashes.
    @extend %foo;
  }
}

// Yep
@media print {
  .bar {
    @at-root (without: media) {
      @extend %foo;
    }
  }
}

// Yep
%foo {
  content: 'foo';

  &-print {
    @media print {
      content: 'foo print';
    }
  }
}

@media print {
  .bar {
    @extend %foo-print;
  }
}
```

Opinions seem to be extremely divided regarding the benefits and problems from `@extend` to the point where many developers including myself have been

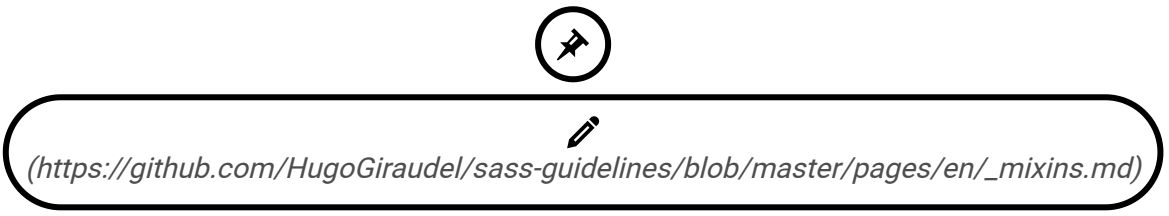
advocating against it, as you can read in the following articles:

- What Nobody Told you About Sass Extend *(http://www.sitepoint.com/sass-extend-nobody-told-you/)*
- Why You Should Avoid Extend *(http://www.sitepoint.com/avoid-sass-extend/)*
- Don't Over Extend Yourself *(http://pressupinc.com/blog/2014/11/dont-overextend-yourself-in-sass/)*

That being said and to sum up, I would advise to use `@extend` only for maintaining relationships within selectors. If two selectors are characteristically similar, that is the perfect use-case for `@extend`. If they are unrelated but share some rules, a `@mixin` might suit you better. More on how to choose between the two in this write-up *(http://csswizardry.com/2014/11/when-to-use-extend-when-to-use-a-mixin/)*.

> **Note —** Thanks to David Khourshid *(https://twitter.com/davidkpiano)* for his help and expertise on this section.

Mixins



(https://github.com/HugoGiraudel/sass-guidelines/blob/master/pages/en/_mixins.md)

Mixins are one of the most used features from the whole Sass language. They are the key to reusability and DRY components. And for good reason: mixins allow authors to define styles that can be reused throughout the stylesheet without needing to resort to non-semantic classes such as `.float-left`.

They can contain full CSS rules and pretty much everything that is allowed anywhere in a Sass document. They can even take arguments, just like

functions. Needless to say, the possibilities are endless.

But I feel I must warn you against abusing the power of mixins. Again, the keyword here is *simplicity*. It might be tempting to build extremely powerful mixins with massive amounts of logic. It's called over-engineering and most developers suffer from it. Don't over think your code, and above all keep it simple. If a mixin ends up being longer than 20 lines or so, then it should be either split into smaller chunks or completely revised.

Basics

That being said, mixins are extremely useful and you should be using some. The rule of thumb is that if you happen to spot a group of CSS properties that always appear together for a reason (i.e. not a coincidence), you can put them in a mixin instead. The micro-clearfix hack from Nicolas Gallagher *(http://nicolasgallagher.com/micro-clearfix-hack/)* deserves to be put in a (argumentless) mixin for instance.

```
/// Helper to clear inner floats
/// @author Nicolas Gallagher
/// @link http://nicolasgallagher.com/micro-clearfix-h
@mixin clearfix {
  &::after {
    content: '';
    display: table;
    clear: both;
  }
}
```

Another valid example would be a mixin to size an element, defining both `width` and `height` at the same time. Not only would it make the code lighter to type, but also easier to read.

```
/// Helper to size an element
/// @author Hugo Giraudel
/// @param {Length} $width
/// @param {Length} $height
@mixin size($width, $height: $width) {
  width: $width;
  height: $height;
}
```

For more complex examples of mixins, have a look at this mixin to generate CSS triangles *(http://www.sitepoint.com/sass-mixin-css-triangles/)*, this mixin to create long shadows *(http://www.sitepoint.com/ultimate-long-shadow-sass-mixin/)* or this mixin to polyfill CSS gradients for old browsers *(http://www.sitepoint.com/building-linear-gradient-mixin-sass/)*.

Argument-Less Mixins

Sometimes mixins are used only to avoid repeating the same group of declarations over and over again, yet do not need any parameter or have sensible enough defaults so that we don't necessarily have to pass arguments.

In such cases, we can safely omit the parentheses when calling them. The `@include` keyword (or `+` sign in indented-syntax) already acts as a indicator that the line is a mixin call; there is no need for extra parentheses here.

```
// Yep
.foo {
  @include center;
}

// Nope
.foo {
  @include center();
}
```

Arguments List

When dealing with an unknown number of arguments in a mixin, always use an `arglist` rather than a list. Think of `arglist` as the 8th hidden undocumented data type from Sass that is implicitly used when passing an arbitrary number of arguments to a mixin or a function whose signature contains `...`.

```
@mixin shadows($shadows...) {
  // type-of($shadows) == 'arglist'
  // …
}
```

Now, when building a mixin that accepts a handful of arguments (understand 3 or more), think twice before merging them out as a list or a map thinking it will be easier than passing them all one by one.

Sass is actually pretty clever with mixins and function declarations, so much so that you can actually pass a list or a map as an arglist to a function/mixin so that it gets parsed as a series of arguments.

```
@mixin dummy($a, $b, $c) {
  // …
}

// Yep
@include dummy(true, 42, 'kittens');

// Yep but nope
$params: (true, 42, 'kittens');
$value: dummy(nth($params, 1), nth($params, 2), nth($p

// Yep
$params: (true, 42, 'kittens');
@include dummy($params...);

// Yep
$params: (
  'c': 'kittens',
  'a': true,
  'b': 42,
);
@include dummy($params...);
```

For more information on whether it is best to use multiple arguments, a list or an argument list, SitePoint has a nice piece on the topic *(http://www.sitepoint.com/sass-multiple-arguments-lists-or-arglist/)*.

Mixins And Vendor Prefixes

It might be tempting to define custom mixins to handle vendor prefixes for unsupported or partially supported CSS properties. But we do not want to do this. First, if you can use Autoprefixer *(https://github.com/postcss/autoprefixer)*, use Autoprefixer. It will remove Sass code from your project, will always be up-to-date and will necessarily do a much better job than you at prefixing stuff.

Unfortunately, Autoprefixer is not always an option. If you use either Bourbon *(http://bourbon.io/)* or Compass *(http://compass-style.org/)*, you may already

know that they both provide a collection of mixins that handle vendor prefixes for you. Use those.

If you cannot use Autoprefixer and use neither Bourbon nor Compass, then and only then, you can have your own mixin for prefixing CSS properties. But. Please do not build a mixin per property, manually printing each vendor.

```
// Nope
@mixin transform($value) {
  -webkit-transform: $value;
  -moz-transform: $value;
  transform: $value;
}
```

Do it the clever way.

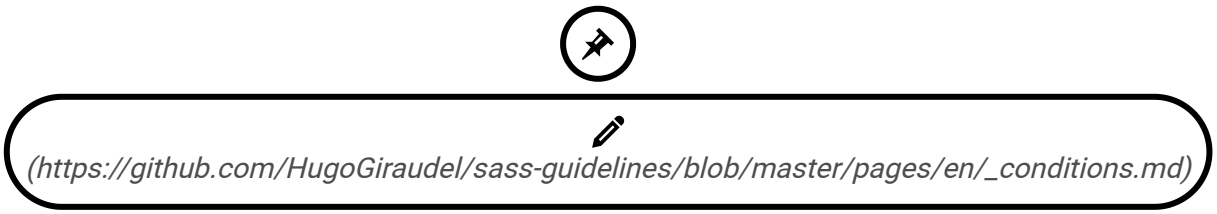
```
/// Mixin helper to output vendor prefixes
/// @access public
/// @author HugoGiraudel
/// @param {String} $property - Unprefixed CSS propert
/// @param {*} $value - Raw CSS value
/// @param {List} $prefixes - List of prefixes to outp
@mixin prefix($property, $value, $prefixes: ()) {
  @each $prefix in $prefixes {
    -#{$prefix}-#{$property}: $value;
  }

  #{$property}: $value;
}
```

Then using this mixin should be very straightforward:

```
.foo {
  @include prefix(transform, rotate(90deg), ('webkit',
}
```

Please keep in mind this is a poor solution. For instance, it cannot deal with complex polyfills such as those required for Flexbox. In that sense, using Autoprefixer would be a far better option.

Conditional Statements



(https://github.com/HugoGiraudel/sass-guidelines/blob/master/pages/en/_conditions.md)

You probably already know that Sass provides conditional statements via the `@if` and `@else` directives. Unless you have some medium to complex logic in your code, there is no need for conditional statements in your everyday stylesheets. Actually, they mainly exist for libraries and frameworks.

Anyway, if you ever find yourself in need of them, please respect the following guidelines:

- No parentheses unless they are necessary;
- Always an empty new line before `@if`;
- Always a line break after the opening brace (`{`);
- `@else` statements on the same line as previous closing brace (`}`).
- Always an empty new line after the last closing brace (`}`) unless the next line is a closing brace (`}`).

```
// Yep
@if $support-legacy {
  // …
} @else {
  // …
}

// Nope
@if ($support-legacy == true) {
  // …
}
@else {
  // …
}
```

When testing for a falsy value, always use the `not` keyword rather than testing against `false` or `null`.

```
// Yep
@if not index($list, $item) {
  // …
}

// Nope
@if index($list, $item) == null {
  // …
}
```

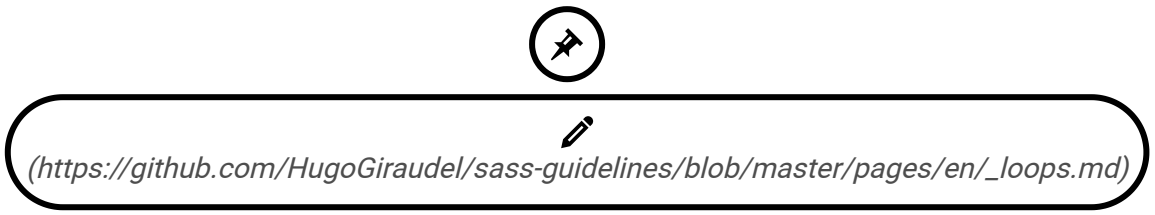
Always put the variable part on the left side of the statement, and the (un)expected result on the right. Reversed conditional statements often are more difficult to read, especially to unexperienced developers.

```
// Yep
@if $value == 42 {
  // …
}

// Nope
@if 42 == $value {
  // …
}
```

When using conditional statements within a function to return a different result based on some condition, always make sure the function still has a `@return` statement outside of any conditional block.

```
// Yep
@function dummy($condition) {
  @if $condition {
    @return true;
  }

  @return false;
}

// Nope
@function dummy($condition) {
  @if $condition {
    @return true;
  } @else {
    @return false;
  }
}
```

Loops



Because Sass provides complex data structures such as lists and maps, it is no surprise that it also gives a way for authors to iterate over those entities.

However, the presence of loops usually implies moderately complex logic that probably does not belong to Sass. Before using a loop, make sure it makes sense and that it actually solves an issue.

Each

The `@each` loop is definitely the most-used out of the three loops provided by Sass. It provides a clean API to iterate over a list or a map.

```
@each $theme in $themes {
  .section-#{$theme} {
    background-color: map-get($colors, $theme);
  }
}
```

When iterating on a map, always use `$key` and `$value` as variable names to enforce consistency.

```
@each $key, $value in $map {
  .section-#{$key} {
    background-color: $value;
  }
}
```

Also be sure to respect those guidelines to preserve readability:

- Always an empty new line before `@each`;
- Always an empty new line after the closing brace (`}`) unless the next line is a closing brace (`}`).

For

The `@for` loop might be useful when combined with CSS' `:nth-*` pseudo-classes. Except for these scenarios, prefer an `@each` loop if you *have to* iterate over something.

```
@for $i from 1 through 10 {
  .foo:nth-of-type(#{$i}) {
    border-color: hsl($i * 36, 50%, 50%);
  }
}
```

Always use `$i` as a variable name to stick to the usual convention and unless you have a really good reason to, never use the `to` keyword: always use `through`. Many developers do not even know Sass offers this variation; using it might lead to confusion.

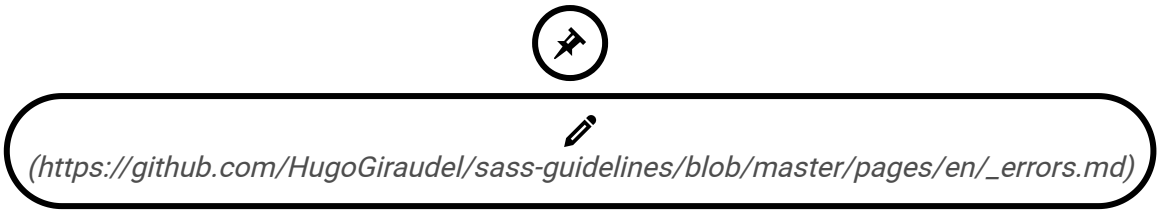
Also be sure to respect those guidelines to preserve readability:

- Always an empty new line before `@for`;
- Always an empty new line after the closing brace (`}`) unless the next line is a closing brace (`}`).

While

The `@while` loop has absolutely no use case in a real Sass project, especially since there is no way to break a loop from the inside. **Do not use it**.

Warnings And Errors



If there is a feature that is often overlooked by Sass developers, it is the ability to dynamically output warnings and errors. Indeed, Sass comes with three custom directives to print content in the standard output system (CLI, compiling app…):

- `@debug`;
- `@warn`;
- `@error`.

Let's put `@debug` aside since it is clearly intended to debug SassScript, which is not our point here. We are then left with `@warn` and `@error` which are noticeably identical except that one stops the compiler while the other does not. I'll let you guess which does what.

Now, there is a lot of room in a Sass project for warnings and errors. Basically any mixin or function expecting a specific type or argument could throw an error if something went wrong, or display a warning when doing an assumption.

Warnings

Take this function from Sass-MQ *(https://github.com/sass-mq/sass-mq)* attempting to convert a `px` value to `em`, for instance:

```
@function mq-px2em($px, $base-font-size: $mq-base-font
  @if unitless($px) {
    @warn 'Assuming #{$px} to be in pixels, attempting
    @return mq-px2em($px + 0px);
  } @else if unit($px) == em {
    @return $px;
  }
   @return ($px / $base-font-size) * 1em;
}
```

If the value happens to be unitless, the function assumes the value is meant to be expressed in pixels. At this point, an assumption may be risky so the user should be warned that the software did something that could be considered unexpected.

Errors

Errors, unlike warnings, prevent the compiler from going any further. Basically, they stop the compilation and display a message in the output stream as well as the stack trace, which is handy for debugging. Because of this, errors should be thrown when there is no way for the program to keep running. When possible, try to work around the issue and display a warning instead.

As an example, let's say you build a getter function to access values from a specific map. You could throw an error if the requested key does not exist in the map.

```
/// Z-indexes map, gathering all Z layers of the appli
/// @access private
/// @type Map
/// @prop {String} key - Layer's name
/// @prop {Number} value - Z value mapped to the key
$z-indexes: (
  'modal': 5000,
  'dropdown': 4000,
  'default': 1,
  'below': -1,
);

/// Get a z-index value from a layer name
/// @access public
/// @param {String} $layer - Layer's name
/// @return {Number}
/// @require $z-indexes
@function z($layer) {
  @if not map-has-key($z-indexes, $layer) {
    @error 'There is no layer named `#{$layer}` in $z-
         + 'Layer should be one of #{map-keys($z-index
  }

  @return map-get($z-indexes, $layer);
}
```

For more information on how to use `@error` efficiently, this introduction about error handling *(http://webdesign.tutsplus.com/tutorials/an-introduction-to-error-handling-in-sass--cms-19996)* should help you.

Tools

(https://github.com/HugoGiraudel/sass-guidelines/blob/master/pages/en/_tools.md)

What's nice about a CSS preprocessor as popular as Sass is that it comes with a whole ecosystem of frameworks, plugins, libraries and tools. After 8 years of existence, we are getting closer and closer to the point where everything that can be written in Sass has been written in Sass *(http://hugogiraudel.com/2014/10/27/rethinking-atwoods-law/)*.

However my advice would to be to lower the number of dependencies to the strict minimum. Managing dependencies is some sort of hell you don't want to be part of. Plus, there is little to no need for external dependencies when it comes to Sass.

Compass

Compass *(http://compass-style.org/)* is the main Sass framework *(http://www.sitepoint.com/compass-or-bourbon-sass-frameworks/)* out there. Developed by Chris Eppstein *(https://twitter.com/chriseppstein)*, one of the two core designers of Sass, I don't see it dramatically losing in popularity for a while, if you want my opinion.

Still, I do not use Compass anymore *(http://www.sitepoint.com/dont-use-compass-anymore/)*, the main reason is that it slows Sass down a lot. Ruby Sass is quite slow in itself, so adding more Ruby and more Sass on top of it doesn't really help.

The thing is, we use very little from the whole framework. Compass is huge. Cross-browser compatibility mixins is just the tip of the iceberg. Math functions, image helpers, spriting… There is so much that can be done with this great piece of software.

Unfortunately, this is all sugar and there is no killer feature in there. An exception could be made of the sprite builder which is *really great*, but Grunticon *(https://github.com/filamentgroup/grunticon)* and Grumpicon *(http://grumpicon.com/)* do the job as well, and have the benefit of being pluggable in the build process.

Anyway, I do not forbid the use of Compass although I would not recommend it either, especially since it is not LibSass-compatible (even if efforts have been made in that direction). If you feel better using it, fair enough, but I don't think you'll get much from it at the end of the day.

Note — Ruby Sass is currently going under some outstanding optimizations that are specifically targeted at logic-heavy styles with many functions and mixins. They should dramatically improve performance to the point where Compass and other frameworks might not be slowing Sass anymore.

Grid Systems

Not using a grid system is not an option now that Responsive Web Design is all over the place. To make designs look consistent and solid across all sizes, we use some sort of grid to lay out the elements. To avoid having to code this grid work over and over again, some brilliant minds made theirs reusable.

Let me put this straight: I am not a big fan of grid systems. Of course I do see the potential, but I think most of them are completely overkill and are mostly used to draw red columns on a white background in nerdy designers' speaker decks. When is the last time you thought *thank-God-I-have-this-tool-to-build-this-2-5-3.1-π-grid*? That's right, never. Because in most cases, you just want the usual regular 12-columns grid, nothing fancy.

If you are using a CSS framework for your project like Bootstrap *(http://getbootstrap.com/)* or Foundation *(http://foundation.zurb.com/)*, chances are high it includes a grid system already in which case I would recommend to use it to avoid having to deal with yet another dependency.

If you are not tied to a specific grid system, you will be pleased to know there are two top-notch Sass powered grid engines out there: Susy *(http://susy.oddbird.net/)* and Singularity *(https://github.com/at-import/Singularity)*. Both do much more than you will ever need so you can pick the one you prefer between these two and be sure all your edge cases—even the most nifty ones—will be covered. If you ask me, Susy has a slightly better community, but that's my opinion.

Or you can head over to something a bit more casual, like csswizardry-grids *(https://github.com/csswizardry/csswizardry-grids)*. All in all, the choice will not have much of an impact on your coding style, so this is pretty much up to you at this point.

SCSS-Lint

Linting code is very important. Usually, following guidelines from a styleguide helps reducing the amount of code quality mistakes but nobody's perfect and there are always things to improve. So you could say that linting code is as important as commenting it.

SCSS-lint *(https://github.com/causes/scss-lint)* is a tool to help you keep your SCSS files clean and readable. It is fully customisable and easy to integrate with your own tools.

Fortunately, SCSS-lint recommendations are very similar to those described in this document. In order to configure SCSS-lint according to Sass Guidelines, may I recommend the following setup:

```
linters:

  BangFormat:
    enabled: true
    space_before_bang: true
    space_after_bang: false

  BemDepth:
    enabled: true
    max_elements: 1

  BorderZero:
    enabled: true
    convention: zero

  ChainedClasses:
    enabled: false

  ColorKeyword:
    enabled: true

  ColorVariable:
    enabled: false

  Comment:
    enabled: false

  DebugStatement:
    enabled: true

  DeclarationOrder:
    enabled: true

  DisableLinterReason:
    enabled: true

  DuplicateProperty:
    enabled: false

  ElsePlacement:
```

```
  enabled: true
  style: same_line

EmptyLineBetweenBlocks:
  enabled: true
  ignore_single_line_blocks: true

EmptyRule:
  enabled: true

ExtendDirective:
  enabled: false

FinalNewline:
  enabled: true
  present: true

HexLength:
  enabled: true
  style: short

HexNotation:
  enabled: true
  style: lowercase

HexValidation:
  enabled: true

IdSelector:
  enabled: true

ImportantRule:
  enabled: false

ImportPath:
  enabled: true
  leading_underscore: false
  filename_extension: false

Indentation:
```

```
  enabled: true
  allow_non_nested_indentation: true
  character: space
  width: 2

LeadingZero:
  enabled: true
  style: include_zero

MergeableSelector:
  enabled: false
  force_nesting: false

NameFormat:
  enabled: true
  convention: hyphenated_lowercase
  allow_leading_underscore: true

NestingDepth:
  enabled: true
  max_depth: 1

PlaceholderInExtend:
  enabled: true

PrivateNamingConvention:
  enabled: true
  prefix: _

PropertyCount:
  enabled: false

PropertySortOrder:
  enabled: false

PropertySpelling:
  enabled: true
  extra_properties: []

PropertyUnits:
```

```
  enabled: false

PseudoElement:
  enabled: true

QualifyingElement:
  enabled: true
  allow_element_with_attribute: false
  allow_element_with_class: false
  allow_element_with_id: false

SelectorDepth:
  enabled: true
  max_depth: 3

SelectorFormat:
  enabled: true
  convention: hyphenated_lowercase
  class_convention: '^(?:u|is|has)\-[a-z][a-zA-Z0-9]

Shorthand:
  enabled: true

SingleLinePerProperty:
  enabled: true
  allow_single_line_rule_sets: false

SingleLinePerSelector:
  enabled: true

SpaceAfterComma:
  enabled: true

SpaceAfterPropertyColon:
  enabled: true
  style: one_space

SpaceAfterPropertyName:
  enabled: true
```

```
SpaceAfterVariableColon:
  enabled: true
  style: at_least_one_space

SpaceAfterVariableName:
  enabled: true

SpaceAroundOperator:
  enabled: true
  style: one_space

SpaceBeforeBrace:
  enabled: true
  style: space
  allow_single_line_padding: true

SpaceBetweenParens:
  enabled: true
  spaces: 0

StringQuotes:
  enabled: true
  style: single_quotes

TrailingSemicolon:
  enabled: true

TrailingZero:
  enabled: true

TransitionAll:
  enabled: false

UnnecessaryMantissa:
  enabled: true

UnnecessaryParentReference:
  enabled: true

UrlFormat:
```

```
    enabled: false

  UrlQuotes:
    enabled: true

  VariableForProperty:
    enabled: false

  VendorPrefixes:
    enabled: true
    identifier_list: base
    include: []
    exclude: []

  ZeroUnit:
    enabled: true
```

If you are not convinced of the necessity to use SCSS-lint, I recommend reading these great articles: Clean Up your Sass with SCSS-lint *(http://blog.martinhujer.cz/clean-up-your-sass-with-scss-lint/)*, Improving Sass code quality on theguardian.com *(http://www.theguardian.com/info/developer-blog/2014/may/13/improving-sass-code-quality-on-theguardiancom)* and An Auto-Enforceable SCSS Styleguide *(http://davidtheclark.com/scss-lint-styleguide/)*.

Note — If you want to plug SCSS lint into your Grunt build process, you will be pleased to know there is a Grunt plugin for that called grunt-scss-lint *(https://github.com/ahmednuaman/grunt-scss-lint)*.

Also, if you are on the hunt for a neat application that works with SCSS-lint and the like, the guys at Thoughtbot *(http://thoughtbot.com/)* (Bourbon, Neat…) are working on Hound *(https://houndci.com/)*.

Too Long; Didn't Read

(https://github.com/HugoGiraudel/sass-guidelines/blob/master/pages/en/_tldr.md)

These guidelines are quite long and sometimes it is good to have them summed up in a shorter version. Below is this summary.

Key Principles

- Having a styleguide is all about **consistency**. If you disagree with some rules from Sass Guidelines, fair enough as long as you are consistent. ↩
- Sass should be kept as simple as it can be. Avoid building complex systems unless absolutely necessary. ↩
- Keep in mind that sometimes *KISS* (Keep It Simple, Stupid) is better than *DRY* (Don't Repeat Yourself). ↩

Syntax & Formatting

- An indentation is made of two (2) spaces, no tabs. ↩
- Lines should be, as much as possible, shorter than 80 characters. Feel free to split them into several lines when necessary. ↩
- CSS should be properly written, possibly following the CSS Guidelines *(http://cssguidelin.es)* from Harry Roberts. ↩
- Whitespace is free, use it to separate items, rules and declarations. Do not hesitate to leave empty lines, it never hurts. ↩

STRINGS

- Declaring the `@charset` directive on top of the stylesheet is highly recommended. ↩
- Unless applied as CSS identifiers, strings should be quoted using single quotes. URLs should also be quoted. ↩

NUMBERS

- Sass makes no distinction between numbers, integers, floats so trailing zeros (0) should be omitted. However, leading zeros (0) help readability and should be added. ↩
- A zero (0) length should not have a unit. ↩
- Units manipulation should be thought as arithmetic operations, not string operations. ↩
- In order to improve readability, top-level calculations should be wrapped in parentheses. Also, complex math operations might be splitted into smaller chunks. ↩
- Magic numbers dramatically hurt code maintainability and should be avoided at all time. When in doubt, extensively explain the questionable value. ↩

COLORS

- Colors should be expressed in HSL when possible, then RGB, then hexadecimal (in a lowercase and shortened form). Color keywords should be avoided. ↩
- Prefer `mix(..)` instead of `darken(..)` and `lighten(..)` when lightening or darkening a color. ↩

LISTS

- Unless used as a direct mapping to space-separated CSS values, lists should be separated with commas. ↩
- Wrapping parentheses should also be considered to improve readability. ↩
- Inlined lists should not have a trailing comma, multi-line lists should have it. ↩

MAPS

- Maps containing more than a single pair are written on several lines. ↩
- To help maintainability, the last pair of a map should have a trailing comma. ↩
- Map keys that happen to be strings should be quoted as any other string. ↩

DECLARATION SORTING

- The system used for declaration sorting (alphabetical, by type, etc.) doesn't matter as long as it is consistent. ↩

SELECTOR NESTING

- Avoid selector nesting when it is not needed (which represents most of the cases). ↩
- Use selector nesting for pseudo-classes and pseudo-elements. ↩
- Media queries can also be nested inside their relevant selector. ↩

Naming Conventions

- It is best to stick to CSS naming conventions which are (except a few errors) lowercase and hyphen-delimited. ↩

Commenting

- CSS is a tricky language; do not hesitate to write very extensive comments about things that look (or are) fishy. ↩
- For variables, functions, mixins and placeholders establishing a public API, use SassDoc comments. ↩

Variables

- Do use the `!default` flag for any variable part of a public API that can be safely changed. ↩
- Do not use the `!global` flag at root level as it might constitue a violation of Sass syntax in the future. ↩

Extend

- Stick to extending placeholders, not existing CSS selectors. ↩

- Extend a placeholder as few times as possible in order to avoid side effects. ↩

Made with love by Hugo Giraudel *(http://hugogiraudel.com)*