**SCSS best practices**

Sass creates Syntactically Awesome Style sheets, or at least that's what it is supposed to do  
when used effectively, Sass helps to build scalable and DRY CSS. When used incorrectly however, Sass can actually increase file size and add unnecessary or duplicate code.  
  
Below is a series of hints and tips to help you get the best out of Sass…

1**. Structure Your Sass**

Getting your site structure correct from the beginning is vital for any new Sass project. Using partials allows you to break the CSS up into smaller more manageable blocks of code that are easier to maintain and develop.  
  
Partial files are created using an underscore and are not output as separate CSS files. Each partial should be imported using a master Sass file (global.scss) in the root of the Sass folder.

**For example, here’s a sample folder structure to demonstrate this:**

sass/   
|   
|– base/   
|   |– \_reset.scss       # Reset/normalize   
|   |– \_font.scss  # Typography rules   
|   ...                  # Etc…   
|   
|– components/   
|   |– \_buttons.scss     # Buttons   
|   |– \_carousel.scss    # Carousel   
|   |– \_cover.scss       # Cover   
|   |– \_dropdown.scss    # Dropdown   
|   |– \_navigation.scss  # Navigation  
|   |– \_forms.scss       # Forms    
|   ...                  # Etc…   
|   
|– helpers/   
|   |– \_variables.scss   # Sass Variables   
|   |– \_functions.scss   # Sass Functions   
|   |– \_mixins.scss      # Sass Mixins   
|   |– \_helpers.scss     # Class & placeholders helpers   
|   ...                  # Etc…   
|   
|– layout/   
|   |– \_grid.scss        # Grid system   
|   |– \_header.scss      # Header   
|   |– \_footer.scss      # Footer   
|   |– \_sidebar.scss     # Sidebar   
|   ...                  # 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             # primary Sass file

As you can see, there is only one Sass file at the root level: main.scss. All the other files are divided into appropriate folders and prefixed with an underscore (\_) to tell Sass they are partial .scss files that shouldn’t be compiled to .css files. Indeed, it is the base file’s role to import and merge all of those.  
  
**Base**  
The base/ folder holds what we might call the boilerplate stuff for your project. In there, you might find the reset (or Normalize.css, or whatever), probably some stuff dealing with typography, and, depending on the project, maybe some other files.  
  
\_reset.scss or \_normalize.scss  
\_typography.scss  
  
**Helpers**  
The helpers/ folder (sometimes called utils/) gathers all Sass tools and helpers we’ll use across the project. Got a function? A mixin? Put it in there. This folder also contains a \_variables.scss file (sometimes \_config.scss) which holds all global variables for the project (for typography, color schemes, and so on).  
  
\_variables.scss  
\_mixins.scss  
\_functions.scss  
\_placeholders.scss (frequently named \_helpers.scss)  
  
**Layout**  
The layout/ directory (sometimes called partials/) usually contains a number of files, each of them setting some styles for the main sections of the layout (header, footer, and so on). It also contains the \_grid file which is the grid system used to build the layout.  
  
\_grid.scss  
\_header.scss  
\_footer.scss  
\_sidebar.scss  
\_forms.scss  
  
Having the navigation file (\_navigation.scss) in this folder could make sense although I use to put it in components/ (see next section). I think it would be better in the /layout folder but I’ll let you decide.  
  
**Components**  
For smaller components, there is the components/ folder (frequently called modules/). While layout/ is kind of macro (defining the global wireframe), components/ is more micro. It can contain all kinds of specific modules like a slider, a loader, a widget, or anything along those lines. There are usually a lot of files in components/ since your site is should be mostly composed of tiny modules.  
  
\_media.scss  
\_carousel.scss  
\_thumbnails.scss  
  
**Pages**  
If you have page-specific styles, I think it’s cool to put them in a pages/ folder and in a file named after the page. For example, it’s not uncommon to have very specific styles for the home page, so you’d have a \_home.scss file in pages/ dealing with this.  
  
\_home.scss  
\_contact.scss  
  
Depending on your deployment process, those files could be called on their own to avoid merging them with the others in the resulting stylesheet. It is really up to you. Where I work, we decided to make them not-partials in order to include them only on pages requiring them. For example, our home page has a very specific layout, compiling to about 200 lines of CSS. To prevent those rules from being loaded on every page, we include this file only on the home page.  
  
**Themes**  
If you are working on a large site with multiple themes like I do, having a themes/ folder can make sense. You can stuff all your theme/design related styles in there. This is definitely project-specific so be sure to include it only if you feel the need.  
  
\_theme.scss  
\_admin.scss  
  
**Vendors**  
And last but not least, you will probably have a vendors/ folder containing all the CSS files from external libraries and frameworks – Bootstrap, jQueryUI, Fancy Carousel Slider jQuery Powered, and so on. Putting those aside in the same folder is a good way to tell “Hey, this is not from me, not my code, not my responsibility”. Example  
  
bootstrap.scss  
jquery-ui.scss  
Select2.scss  
owl-carousel.scss  
  
On a side note, where I work we also have a vendors-extensions/ folder where we store files overriding some tiny bits from vendors. For example, we have a \_bootstrap.scss file in there that we can use to change some components in Bootstrap. This is to avoid editing the vendor files themselves, which is generally not a good idea.

**2. Use Sass Variables More Effectively**

Variables are one of the more straightforward features of Sass but are still on occasion used incorrectly. Creating a site-wide naming convention is essential when working with Variables. Without one, they become harder to understand and re-use.  
  
**Here are some tips for creating useful variables:**

* Don’t be too vague when naming your Variables.
* Have and stick to a naming convention (Modular, BEM, etc.)
* Ensure the variable use is justified.

**Here are some good examples:**$orange: #ffa600;   
$grey: #f3f3f3;   
$blue: #82d2e5;  
  
$link-primary: $orange;  
$link-secondary: $blue;  
$link-tertiary: $grey;  
  
$radius-button: 5px;  
$radius-tab: 5px;

**And some bad examples:**  
  
$link: #ffa600;  
$listStyle: none;  
$radius: 5px;

**3. Reduce Mixin Usage**

A mixin is a great way to include sections of code multiple times within a site. However, including a mixin is the same as copying and pasting the styles throughout the CSS file. It creates a mass of duplicate code and can bloat your CSS file.

A mixin therefore should only be used if an argument is present, to quickly create modified styles.

**Here’s an example:**

@mixin rounded-corner($arc) {  
   -moz-border-radius: $arc;  
   -webkit-border-radius: $arc;  
   border-radius: $arc;    
}

This rounded-corner mixin can be used in any situation simply by changing the value of $arc, making it a worthwhile mixin:

.tab-button {  
    @include rounded-corner(5px);   
}  
  
.cta-button {  
    @include rounded-corner(8px);   
}

**A bad example might look like this:**

@mixin cta-button {  
   padding: 10px;  
   color: #fff;  
   background-color: red;  
   font-size: 14px;  
   width: 150px;  
   margin: 5px 0;  
   text-align: center;  
   display: block;  
}

This mixin has no argument and would therefore be better written as a placeholder, which brings us to point 4.

**4. Embrace Placeholders**

Unlike mixins, placeholders can be used multiple times without adding any duplicate code. This makes them a much friendlier option for outputting DRY CSS:

%bg-image {  
   width: 100%;  
   background-position: center center;  
   background-size: cover;  
   background-repeat: no-repeat;  
}  
  
.image-one {  
   @extend %bg-image;  
   background-image:url(/img/image-one.jpg");  
}  
  
.image-two {  
   @extend %bg-image;  
   background-image:url(/img/image-two.jpg");  
}

**And the compiled CSS:**

.image-one, .image-two {  
   width: 100%;  
   background-position: center center;  
   background-size: cover;  
   background-repeat: no-repeat;  
}  
  
.image-one {  
   background-image:url(/img/image-one.jpg") ;  
}  
  
.image-two {  
   background-image:url(/img/image-two.jpg") ;  
}

The repeated code in the placeholder is output only once with only the unique styles being applied to the individual selectors. If unused, the placeholder styles are not output at all.

Tying in with point 3, placeholders can be used alongside mixins to reduce duplicate code and still keep the flexibility of a mixin…

/\* PLACEHOLDER   
============================================= \*/  
  
%btn {  
   padding: 10px;  
   color:#fff;  
   curser: pointer;  
   border: none;  
   shadow: none;  
   font-size: 14px;  
   width: 150px;  
   margin: 5px 0;  
   text-align: center;  
   display: block;  
}  
  
/\* BUTTON MIXIN   
============================================= \*/  
  
@mixin  btn-background($btn-background) {  
   @extend %btn;  
   background-color: $btn-background;  
   &:hover {  
       background-color: lighten($btn-background,10%);  
   }  
}  
  
/\* BUTTONS  
============================================= \*/  
  
.cta-btn {  
   @include btn-background(green);  
}  
  
.main-btn {  
   @include btn-background(orange);  
}  
  
.info-btn {  
   @include btn-background(blue);  
}

**5. Use Functions for Calculations**

Functions are used to perform calculations. A Sass function does not output any CSS. Instead, it returns a value that can be used in the CSS. This is useful for calculations that will be made throughout the site.

For example, functions are useful for calculating the percentage width of a given element:

@function calculate-width ($col-span) {  
   @return 100% / $col-span   
}  
  
.span-two {  
   width: calculate-width(2); // spans 2 columns, width = 50%  
}  
  
.span-three {  
   width: calculate-width(3); // spans 3 columns, width = 33.3%  
}

**6. Order Your Work**

Place all mixins, functions, placeholders and variables in their relevant partial file. Keeping blocks of code together will ensure they are easy to edit and reuse in the future.

Site-wide elements should be kept together in a base folder. The base folder should contain global variables such as fonts and colour schemes:

$font-primary: 'Roboto', sans-serif;   
$font-secondary: Arial, Helvetica, sans-serif;  
  
$color-primary: $orange;  
$color-secondary: $blue;  
$color-tertiary: $grey;

Module-specific mixins, functions, and variables should be kept within the correct module’s partial file:

$tab-radius: 5px;  
$tab-color: $grey;

**7. Limit Nesting**

Overusing nested rules in Sass can cause a lot of issues, from complex code to over-specificity and too much reliance on the HTML structure of a page. These things can cause issues further down the line and potentially increase the need for the inclusion of !important, which should generally be avoided.

**Here are some golden rules for nesting:**

* Never go more than 3 levels deep.
* Ensure the CSS output is clean and reusable.
* Use nesting when it makes sense to, not as a default option.

**8. Keep Things Simple**

The concluding point I’ll make in this post is to keep things as simple as possible. The purpose of Sass is to write cleaner more manageable CSS. Before creating any new mixins, variables, or functions, ensure that their presence will enhance development and not over complicate things. All Sass features are useful when used in the correct situations and in moderation.

Creating an endless list of variables without clear usage, or a complex function that is difficult to understand for anyone other than the author is not the intention and will not aid development or produce DRY CSS.

**SCSS/CSS beautifier (Pod1)**

1. **Indentation and Formatting**
   1. Keep each rule on a separate line. Exceptions include margin/padding, top/right/bottom/left (for cases where position: relative/absolute is being used).
   2. In most cases, try and keep x-axis values first when sharing a line with more than 1 rule (ex. left: 20px; top: 0px;)
2. Make every attempt to keep rule names in alphabetical order for speed and ease of visual scanning/browsing.
3. Make EVERY attempt to avoid defined heights. ONLY use a defined height if it’s essential to a (js) function, or as a last resort.
4. **Grouping Syntax**
5. Groupings are a section of styles, related by either a usability pattern or by a section of the layout or site.
6. Example of comment beginning a grouping:  
   /\* @group START Grouping Name \*/
7. Example of comment ending a grouping:  
   /\* @group END \*/
8. Groups can be nested (please use indentation/tab to visually nest)
9. The rules and selectors/styles within a group shall be indented 1 tab (4 spaces) from the
10. START/END group marker(s).