Welcome to Cybage

Document Name	SCSS/SASS Style Guide
Version No.	V.1
Release Date	30th Apr 2015

Document Template History

Version No.	Authored / Modified by	Reviewed by, Date	Approved by, Date	Date Remark / Change History
V1.0	Yogesh Gaikwad			Compiled as on 7th Apr 2015
V1.0	Yogesh Gaikwad			Code updates

This document contains guidelines for web applications. This document's primary motive is to implement code consistency and best practices to all FET projects. By maintaining consistency in coding styles and conventions, we can ease the burden of code maintenance, and mitigate risk of breakage in the future.

By adhering to best practices, we ensure optimized page loading, performance and maintainable code.

Contents

1. OBJECTIVE

Sass is the most mature, stable, and powerful professional grade CSS extension language. SASS promotes DRY coding style and object oriented CSS structure. This is a set of coding conventions and rules for use while working on SASS (Syntactically Awesome StyleSheets).

These Standards are inspired by the SASS developer communities and contributed by Cybage developers.

SCSS and SASS are bifurcated based on the syntax so don't get confused. They are not two different libraries. In this document it will be referred with the name SASS.

SASS is one of the popular CSS pre-processor. It allows developer or designer to use variables, nested rules, mixins, inline imports, and more, all with fully CSS-compatible syntax. SCSS helps keep large stylesheets well-organized, and get small stylesheets up and running quickly, particularly with the help of the Compass style library.

More than, what is SASS? This document talks about style guide for SASS.

2. WHY SASS?

- Fully CSS3-compatible
- Language extensions such as variables, nesting, and mixins
- Many useful functions for manipulating colors and other values
- Advanced features like control directives for libraries
- Well-formatted, customizable output

3. WHAT IS SASS?

There are training videos uploaded by SASS for developers on YouTube which are quite helpful.
YouTube Playlist

Before any developer starts working on SASS he shall refer SASS documentation as a reference point.

4. GENERAL THUMB RULE

- Class name should be meaningful and generic in nature. Do not add color or style type to class name like blue-button, search-box, etc.
- Vendor classes should not be altered instead write new classes as overwrites.
- Extends and Mixins should be placed before standard properties
- Use soft tabs with a two space indention
- Add a space after :
- Add a space after // comments
- Stick with classes instead of IDs for styling
- Add a space after commas in values e.g. rgba(#000, 0.5)

- Write numbers at the end of mathematic operations e.g. $b-space * 0.5
- Limit nesting as much as possible

Base values are typically stored in Config/Base SCSS and derived from that point on.

5. NUMBER GAME

- Line heights are unitless, e.g. 1.5
- Use ems at wherever possible if spacing is not fix and liquid in nature
- Font sizes to be mentioned in ems
- Value zero is unitless, e.g. width: 0;

6. FILE NAMING

Partials are named _partial.scss. Files which don't have prefix as "_" will be compiled as independent files and needs to be included separately in to HTML. In our case we will be keeping all files as partials except styles.scss. styles.scss is our start point.

7. COMPILATION

You can use your code editors as compilers or Scout, Koala, Compass, Grails SASS/SCSS plugin to compile your SCSS code. It is always better to use sourcemap supporting plugins. Sourcemap can show you the line number of SCSS code in browser developer tools which makes SASS debugging easier.
Ref: http://thesassway.com/intermediate/using-source-maps-with-sass

8. DEPLOYMENT

Always compress, uglify and gzip CSS files. It also includes use of image compression tools. As of now there is no compression tool is being used due to lesser images.

9. CHECK-IN / CHECK-OUT

- You can check-in / check-out SCSS files to version controlling tool but do not push them to live as they are of no use on deployment server.
- Only compiled .css files which are referred in HTML pages needs to be committed.

10. COMMENTS

Always comment your code properly. These comments can be referred by you or either another developer for future use. Comments should be self-explanatory and has enough information about the code.

```
// ************************************
//   First Level
//   -> Description
// ************************************

// ------------------------------------
//   Second Level
// ------------------------------------

// ----- Third Level ----- //

// Fourth Level
```

11. NAMING CONVENTIONS

- Base variable names or class name should use y-acronym (hyphen between words)
- If the name comprises two words, utilize camelCase—for instance, taskList.

12. SEQUENCING

1. @extend(s)
2. @include(s)
3. Regular styles
4. Nested pseudo styling
5. Nested selectors last

```
.widget {
  @extend %module;
  @include transition(all 0.3s ease);
  background: $gray;
  &:hover {
    background: $white;
  }
  &::before {
    content: "";
    display: block;
  }
  > h3 {
    @include transform(rotate(90deg));
    border-bottom: 1px solid $white;
  }
}
```

6. NESTING LEVEL

Max three level deeper nesting you can use. In case of very deeper level of nesting, your component becomes difficult to use without the structure you have created for it. So keep it simple and unplug-gable without much effort.

7. STATES

Generally added via JavaScript, states are similar to modifiers but carry conditional context. is- denotes a state, such as is-active, and they're utilized as such:

```
// ************************************
//   Button
//   -> Action points
// ************************************
.btn {
     // Styles
}
// ------------------------------------
//   States
// ------------------------------------
.btn.is-active{
      background: $c-highlight;
}
```

8. CONTEXT

The most common case tends to be positioning context. If you have a dropdown structure that's being positioned absolutely, the parent element should be (at least) positioned relatively:

```
// ************************************
//   Dropdown
//   -> Revealed information
// ************************************
.dropdown {
     // Styles
}

// ------------------------------------
//   Context
// ------------------------------------
.has-dropdown {
     position: relative
}
```

Note: Same case with display for calendar control in grid. td should have class like .has-calendar

9. IMAGES

- bg-* for background images
- logo-* for logos
- img-* for content images
- Sub-folders for larger groups

10. MANIFEST

Now that we've touched on naming and basic ideas, let's put it all together. Following is the ideal setup wherever your styles are found. You can alter them as per the application need.

```
_SCSS
   styles.scss
   _variables.scss
   _base/
      _reset.scss
      _helpers.scss
      _config.scss
      _tools.scss
   _modules/
      _dropdown.scss
      _inputbox.scss
      _navigation.scss
   _layouts/
   _themes/
   vendor/
```

styles.scss file is a start file for compiler and final compressed version of all SCSS files. _variable.scss files will store all the values of commonly used properties.

Utility

This file from _base folder shall contain utility functions or mixins which can be used application wide. e.g. class named .free which applies 100% width to any of the HTML element. This can't be specific to any control and can't be kept in to mixin or variable files due to its nature of non-tag based style.

Few more example are like, clearfix, margin flush, spacing flush, etc.

If you have a paragraph and margin-bottom for it needs to be flushed then this can simply achieved using .mbf class,

```
<p class="mbf">Lorem ipsum dolor sit amet </p>
```

.mbf stands for margin bottom flush and can be used irrespective of markup hierarchy.

Note: if there is a need of applying multiple utility classes to any element then it is better to create a new compound class having all those utilities.

Modules

These are reusable components and of portable nature. That means components can be easily plugged or removed from any of the project. Your component/module can be used in another project as well so always think of its generic nature.

We have modules like, button, grid, input boxes, radio buttons, etc.

How to determine whether component is to be created as a module?

- Answer below questions,

- Is this component independent enough
- Can this component used in another project without a much alterations?

If answer is “YES” then it is a module otherwise it is a structure for sure.

E.g. in IS compliance we have a grid and containers associated to it. If you go to deeper level of controls then container, grid, buttons, pagination modules can be used independently but if you put them together, it becomes a structure which needs to be style in structure files.

Note: do not mix module and structure styles. We call structure as layout as well.

States

Currently, we are styling states of modules within the same file as module. To avoid any confusion at developer level and to keep it uncluttered, sub types are merged.

Note: As of now application level states are not in the picture. If that comes in to the picture then it will be maintained at layouts. So that states of module and layout will be bifurcated and maintained easily.

Structures / Layouts

When component is comprises of multiple components and responsive to browser layouts then it is certainly a structure. In earlier point of module we have seen this. Using layouts you can control themes as well.

Understand it correctly; responsive nature needs to be written in both module and layouts.

- If my button needs to be shown 50% of mobile width and shown 200px on desktop then this style shall be handled under module.
- If we have 4 button aligned horizontally in desktop and needs to be aligned vertically in mobile then it becomes your layout styling.

Note: Do not mix-up with module/layout responsive styling

Always create your module of nature portable.

If you can change few lines of CSS inside a module and use it between different projects then consider it as a portable one, and you can call it as an independent module. On the other hand, if try to port a module and find yourself rewriting a significant amount of code, then the module is likely better classified as a structure/layout.

Vendor

Vendor is a folder we secured for third party stylesheets. In our case it is bootstrap and few other libraries. In future we shall not use SCSS of bootstrap as we want it to be loaded from CDN to avoid load on web server. So in case bootstrap version is changed then we need not to worry. Just change of version in <link> source shall work for us.

11. NORMALIZE

Normalize.css is a popular HTML5-ready alternative to CSS resets. It makes browsers render all elements more consistently and in line with modern standards, precisely targeting only the styles that need normalizing. This is the Sass/Compass port of that file, which utilizes legacy IE support variables, a CSS3 box sizing mixin and vertical rhythm mixins. https://github.com/JohnAlbin/normalize-scss

Currently we are utilizing normalize of bootstrap as default. If you want to alter then be very sure about it otherwise it will certainly mess your structuring.

12. IMPORTS

Files found in SCSS are imported into the manifest in a particular order, while Components and Structures are typically included alphabetically. The dependencies like compass, colors, and mixins generate no compiled CSS at all, they are purely code dependencies. Import sequence decides your override pattern, so be very sure about your import sequence.

```
// ************************************
//   Project Name
//   -> Manifest
// ************************************

// ------------------------------------
//   Vendor
// ------------------------------------

@import "compass";
@import “vendor/boostrap”


// ------------------------------------
//   Base
// ------------------------------------

@import "_base/reset";
```

```
@import "_base/utilities";
@import "_base/config";

// ------------------------------------
//   Components
// ------------------------------------

@import "_modules/dropdown";
@import "_modules/grid";

// ------------------------------------
//   Structures
// ------------------------------------

// Structure imports


// ------------------------------------
//   Temporary Envelope
// ------------------------------------
@import "shame";
```

13. TEMPORARY ENVELOPE (SHAME FILE)

Collaboration with developers can sometimes become difficult, but here is the way to help mitigate the risk. At the bottom of the styles.scss file, there's a comment block that looks like this:

```
// ------------------------------------
//   Temporary Envelope
// ------------------------------------
@import "shame";
```

It imports a file named shame. It should be your last line of code. Any temporary styles, quick fixes can be added in _shame.scss file at the root level of SCSS folder, which will allow the maintainer of the SCSS to rewrite or sort the styles appropriately. Those we are not authenticated to make changes to core structure or components, they can define their own styles and request to accommodate it in the base.

It is not a myth! ☺ See - http://csswizardry.com/2013/04/shame-css/

14. FUNCTIONS

Since there are many functions made available by SASS itself, we barely declare any. So required functions which are not available with SASS library can be added here.

Ref: http://sass-lang.com/documentation/Sass/Script/Functions.html

15. MIXINS

Mixins are created as a part of DRY code. Entries in the Mixin section should always take arguments and have the ability to differ when utilized.

16. EXTENDS

Extends are collections of rules to use either directly in the markup, or to extend within modules. Better way to extend a base class and add features on top of it.

Avoid %className sign syntax to avoid selector bloat.

Ref: http://csswizardry.com/2014/01/extending-silent-classes-in-sass/

17. @FONT-FACE

Font-face mixin from _utility.scss is inherited pattern of Bourbon's font-face mixin used to simplify the daunting syntax, and also list fallback files needed for browser compatibility. Make sure you add all required font extensions (.eot, .woff, .ttf, .svg) to font folder.

Ref: https://github.com/thoughtbot/bourbon/blob/master/app/assets/stylesheets/css3/_font-face.scss

```
// -------------------------------------
//   @font-face
// -------------------------------------

// ----- Gotham ----- //
@include font-face('gotham', '/fonts/gotham');
```

OR

```
@include font-face('gotham', '/fonts/gotham', ‘bold’, ‘italic’);
```

18. VARIABLES

All variables in the application are defined in _variables.scss file and are prefixed by their role or respective Component/Structure.

- $b-* for base variables
- $c-* for colors
- $g-* for breakpoints
- $componentName-* for Components
- $structureName-* for Structures

Alternately, you can define these variables at component level when component construction is in the initial phase. Once you are sure it is at a good level, variables can be moved to common file.

For colors we will be using color palettes which can be easily altered for components based on the themes. So change in the palettes can change your whole color theme without a much do.

```
// ------------------------------------
//   Colors
// ------------------------------------

// ----- Palette ----- //
$cerulean: #017ba7;
$forest: #7ba05b;
$gainsboro: #ecf0f1;
$gold: #ffd700;
$jet: #343434;
$scarlet: #ff3f00;
$white: #fff;

// ----- Base ----- //

$c-background-invert: $white;
$c-background: $gainsboro;
$c-border: lighten($jet, 30%);
$c-error: $scarlet;
$c-highlight: $cerulean;
$c-text-invert: $white;
$c-text: $jet;
$c-subdue: lighten($cerulean, 40%);
$c-success: $forest;
$c-warning: $gold;

// ----- Components ----- //
// Example: $row--a-background: $c-highlight;

// ----- Structures ----- //
// Example: $dropdown-link-color: $c-subdue;

// ------------------------------------
//   Base
// ------------------------------------

// ----- Borders & Box Shadow ----- //
$b-borderRadius: 3px;
$b-borderStyle: solid;
$b-borderWidth: 2px;
$b-border: $b-borderWidth $b-borderStyle $c-border;
$b-boxShadow: 0 2px 0 rgba($jet, 0.25);

// ----- Typography ----- //
$b-fontFamily-heading: 'OpenSans', sans-serif;
$b-fontFamily: 'OpenSans', sans-serif;
$b-fontSize: 16px;
```

```
$b-fontSize-s: 75%;
$b-fontSize-m: 90%;
$b-fontSize-l: 115%;
$b-lineHeight: 1.5;

// ----- Sizing ----- //
$b-space: em(20px);
$b-space-s: 0.5 * $b-space;
$b-space-l: 2 * $b-space;
$b-space-xl: 4 * $b-space;

// -------------------------------------
//   Components
// -------------------------------------

// ----- Grid ----- //
$g-s: em(480px);
$g-m: em(800px);
$g-l: em(1024px);

// -------------------------------------
//   Structures
// -------------------------------------

// ----- Pagination ----- //
// Example: $pagination-width: em(200px);
```

In case of pagination, it is a collection of multiple modules. E.g. we have used label, dropdown, links, buttons and go to textfield. So whole pagination will be called as structure and elements inside it will be identified as controls.

19. CREATING A BASE SCSS FOR HTML

Base contains all of the tag-level settings for default HTML elements. Tag based styling for anchors, headings, paragraphs, lists, and everything else that doesn't have a class name attached. There are so many possible HTML elements to cover therefore only relevant tags will be styled.

We have distributed tag based style in to 3 different parts:

HTML-BODY

html and body styles at the top, and then divide the remaining content into Block and Inline sections.

```
// ************************************
//   Base
//   -> Tag-level settings
// ************************************

html {
     background: $c-background;
     color: $c-base;
```

```
    font-family: $b-fontFamily;
    font-size: $b-fontSize;
    line-height: $b-lineHeight;
}

body {
    font-size: 100%;
}
```

Block elements

This section covers block level elements like, headings, paragraphs, lists, as well as figures, blockquotes, and more. Do not collapse listing styles here to avoid any further complexity.

```
// -------------------------------------
//   Block Content
// -------------------------------------

ul, p {
  margin-bottom: $b-space;
  margin-top: 0;
}

li {
  margin-bottom: $b-space-s;
  margin-top: 0;
}

// ----- Headers ----- //

h1, .h1,
h2, .h2,
h3, .h3,
h4, .h4 {
  font-family: $b-fontFamily-heading;
  font-weight: bold;
  line-height: 1.2;
  margin-bottom: $b-space-xs;
  margin-top: 0;
}
```

Inline elements

Yes, you guessed right. Here you can add the style related to inline elements e.g. a, strong, code, em, etc. These are the elements which do not force any line break before or after them.

```
// -------------------------------------
//   Inline Content
// -------------------------------------

a {
  border-bottom: $b-border;
```

```
  color: $c-highlight;
  text-decoration: none;
}

  &:hover,
  &:focus {
    border-bottom-color: $c-highlight;
    color: $c-subdue;
}
```