CSS Guidelines

Guideline and advice for writing sane, scalable and maintainable CSS Code

Introduction

CSS is not a pretty language. While it is simple to learn and get started with, it soon becomes problematic at any reasonable scale. There isn't much we can do to change how CSS works, but we can make changes to the way we author and structure it.

In working on large, long-running projects, with dozens of developers of differing specialities and abilities, it is important that we all work in a unified way in order to

- keep stylesheets maintainable.
- keep code transparent, sane, and readable.
- keep stylesheets scalable.

There are a variety of techniques we must employ in order to satisfy these goals, and this document contains recommendations and approaches that will help us to do so.

The Importance of a Coding Styleguide

A coding styleguide will:

- set a standard for code quality across our projects.
- help with building and maintaining projects for a longer period of time.
- help developers with different technical abilities on-board our team easier.
- help with having a number of different developers working on a project at any given time.
- help with project bouncing; dipping in and out of a project will be easier.

This coding styleguide should be learned, understood fully, and implemented at all times on a project that is bound to one. Any deviation from the styleguide should be fully justified and accepted by other team memebrs.

Syntax and Formatting

One of the simplest forms of a styleguide is a set of rules regarding syntax and formatting. Having a standard way of writing (literally writing) CSS means that code will always look and feel familiar to all members of the team.

Multiple Files

Using the LESS preprocessor gives us the option to split our CSS codebase to multiple files with ease. We must split our CSS to files according to the module/component we are styling.

Table of Contents

Maintaining a table of contents for a project creates a substantial overhead, but the benefits outweigh the costs. A table of contents will:

- Make it easier to make-sense of the project folder structure
- Make it easier to develop new features for old projects

A table of contents should:

- Sit in our main CSS file (with all the imports/includes, usually _app.less)
- Describe each import
- Contain important info any developer should know (any nasty fixes in the project? any complicated animations?)

Guidelines:

- Each section is separated with 2 newlines
- Each component is separated with 1 newline
- Each section is labeled with “=SECTION=”
- Each component is labeled with “ - Component”
- Every imported LESS file must be documented in the Table of Contents

Table of contents example for Golfco

```
/**
 * CONTENTS
 *
 *
 * =GLOBAL=
 * _404.less            --------------- 404 Page styling.
 * _vars.less           --------------- Gloabl variables and configurations.
 *
 *
 * =UI=
 * /fonts               --------------- Fonts declerations.
 *      _content.less   --------------- Content font decleration.
 *      _heading.less   --------------- Heading font decleration.
 *      _icons.less     --------------- Icons font decleration.
 * _back-to-top.less    --------------- Back to top button styling.
 * _body.less           --------------- General declerations, styles and
resets.
```

```
 * _button.less         --------------- Button styling, general component.
 * _checkbox.less       --------------- Checkbox styling.
 * _input.less          --------------- Input styling.
 * _modal.less          --------------- Modal styling (quickview is
seperate).
 *
 *
 * =Components=
 *  - Banner
 *    /classes             --------------- Custom slide classes.
 *            _description --------------- slide description classes.
 *            _image       --------------- slide image classes.
 *            _laebl       --------------- slide label classes.
 *    /layouts             --------------- Custom banner layouts.
 *            _arrows_bot_right.less --------------- This layout will force
the slider arrows
 *                                                   to be at the bottom
right corner of the slider,
 *                                                   with white background.
 *            _buy_the_look.less     --------------- This layout will add
large paddings
 *                                                   to the sides of the
slider.
 *
 *  - Checkout...
 *
 *
 * =Important=
 * This project has a nice quick-view animation that
 * combines JS event-triggered classes with CSS animations
 * the animation was created by Asher and can be found in
 * components/product/layouts/_quickview.less file,
 * There is a transition variable for the animation
 * @quickview-transition-time; This variable controls
 * the timing for the whole animation.
 **/
```

80 Characters Lines

Where possible, limit CSS files width to 80 characters. Reasons for this include

- the ability to have multiple files open side by side;
- viewing CSS on sites like GitHub, or in terminal windows;
- providing a comfortable line length for comments.

```
/**
 * I am a long comment, but i will still follow the
 * 80 character line rule to make it easier on other
```

```
 * developers to view my code. You're welcome.
 **/
```

Titling

Begin every new section with a title. The title should

- Be styled as shown;
- Be prefixed with a hashtag;
- Be all uppercase letters;
- Have all word separated with a hyphen;
- Have an empty newline after it;
- Have 5 newlines before it (if not at the beginning of a file).

```
/*-------------------------------------*\
  #PRODUCT-GALLERY
\*-------------------------------------*/

.product-page-gallery {

    color: blue;
}




/*-------------------------------------*\
  #PRODUCT-TABS
\*-------------------------------------*/

.product-tabs {

    color: red;
}
```

Anatomy of a Ruleset

What we call different parts of a ruleset

```
[selector] {
  [property]: [value];
  [<--declaration--->]
}
```

Proper Ruleset Syntax

This is the correct syntax for a ruleset

- A space is inserted between the selector and the opening bracket;
- An empty newline is inserted between the opening bracket and the first declaration;
- All declarations are grouped into 3 categories: Positioning, Box-model, Other;
- Don't use units for empty values (0 instead of 0rem);
- Omit the leading zero for decimal values (.5rem instead of 0.5rem);
- Use 0 instead of “none” for supported values (background, border...);
- Insert a newline between each group;
- Always use double quotes (content, urls...);
- Space rulesets by a newline.

```
// Correct
.foo {

    position: absolute;
    top: 0;
    right: 0;

    display: block;
    padding: 1rem;
    margin: .5rem;

    color: red;
    box-shadow: 0 2px 2px 2px rgba(0, 0, 0, .5);
    background: 0;
}

// Incorrect
.foo{
    position: absolute;
    display: block;
    top: 0;
    right: 0;
    margin: 0.5rem;
    padding: 1rem;
    color: red;
    background: none;
    box-shadow: 0px 2px 2px 2px rgba(0, 0, 0, .5);
}
```

Alignment

When appropriate, align values for easier column-editing

```
.foo {

    position: absolute;
    top:    0;
    right:  0;
    bottom: 0;
    left:   0;

    margin-left:    1rem;
    margin-right:   1rem;
    padding-top:    1rem;
    padding-bottom: 1rem;
}
```

Pseudo Elements

Generally we should avoid displaying text with a pseudo element, but when obligated, always convert to unicode.

Use this tool: https://r12a.github.io/app-conversion/

- Always use one colon when styling a pseudo element;
- The content declaration must be the first declaration;

```
.foo:before {

    content: "";

    display: block;
}
```

Naming Conventions

Naming conventions in CSS are hugely useful in making your code more strict, more transparent, and more informative.

- LESS variables should be hyphen (-) delimited
- LESS variables for different viewports should be prefixed with the appropriate device
    - ls - large screen
    - dt - desktop
    - tb - tablet
    - sp - mobile
- Classes should be hyphen (-) delimited

Selectors

The right choice of selectors makes the whole difference.

- You should strive to use only classes
- If you must use an ID, use the attribute selector

```
// Correct
[id="product"] {

    color: blue;
}

// Incorrect
#product {

    color: red;
}
```

- If you need to increase a specificity, chain the class with itself

```
.foo.foo {

    color: red;
}
```

- Don't couple your classes to an element type, unless you require your HTML to use that class in a specific element.
- Don't nest over 2 levels, if you do, rethink your process.
- Mind selector performance.

Units

Using proper units is important for responsiveness.

- Use rem's for anything that needs to scale with the viewport but does not require viewport units.
- Set pixel font size on the html and body elements, and scale your project according to that.
- Use pixel units for fixed sizes such as borders, touch boxes, etc…

!important

Using !important is strictly forbidden!.. or is it?
Using !important to increase a selector's specificity is not a good idea, if you need to do this you should re-think your code.

There are 2 acceptable uses for !important

- Overriding javascript
- Making sure a style is applied

```
/* Acceptable */
.hide {
    display: none !important;
}

/* Not Acceptable */
.product {

    color: red !important;
}

#product {

    color: red;
}
```