CSS Architecture

A high level view of the common css approaches. This work was inspired by the book Enduring CSS by ben Frian.

Specificty Problem for large scale applications

As developers we are often faced with the challenge of trying to design our system in a way which is easy to read and reason about. The cascading nature of css can create a number of headaches because of how specificty is assigned. The different problems outlined below attempt to solve the specificty issue through a conscious design of the css naming convention.

This table summarizes problem of name clashing

selector	inline	ID	class	type
.widget	0	0	1	0
aside#sidebar .widget	0	1	1	1
.class-on-sidebar .widget	0	0	2	0

1. different weights of selectors are confusing to learn and understand

2. can lead to the use of !important when specificty is not understood, effectively creating a specificty war
3. leads to css styling that is hard to reason about, and bloated selectors which obfuscate the relationship between the DOM element and the actual style applied to it

> Low specificity: If specificity battles start between selectors, the code quality starts to nosedive. Having a low specificity will help maintain the integrity of a large project's CSS for a lot longer.

Limitations of IDs as selectors

> In summary, they are far more specific than a class selector – therefore making overrides more difficult. Plus they can only be used once in the page anyway so their efficacy is limited.

However, to keep specificty lower you can do the following:

```
[id="Thing"] {
/* Property/Values Here */
}
```

Object Oriented CSS

The best exmple of this approach is Atomic CSS where the class name is composed of different features which make up or compose the overall styling. The philosophy behind atomic css is to break the css into a

variety of building blocks which can be declared inside the element. In this way, the component retains control of the styling, and the user can create variables and customize the stylistic building blocks for the application.

Here is on example:

```
    <div class="Bgc(#0280ae.5) C(#fff) P(20px)">A big callout section</div>
```

In this case, Bgc would be background color and the element inside the () is the value for that background color. Taking this one step further, you could even write Bgc(grey-color) and abstract the grey-color into a variable which can be used throughout your application.

Using the shorthand nomenclature for Atomic css, the grid system can be vastly shortened. Here is an example of an atomic css grid system for inline blocks:

```
<div>
   <div class="IbBox W(1/3) P(20px) Bgc(#0280ae.5)">Box 1</div>
    <div class="IbBox W(1/3) P(20px) Bgc(#0280ae.8)">Box 2</div>
    <div class="IbBox W(1/3) P(20px) Bgc(#0280ae">Box 3</div>
</div>
```

Above, the IbBox determines the type to be an inline-block, the width is 1/3, P(20px) = padding 20px, and Bgc is background color and the .5 is opacity 50%.

Pros

- Variables can be passed along in a more intuitive manner, closer to the DOM structure.
- Grid system is more obvious than a standard bootstrap row approach which abstracts the underlying CSS.
- Is more composable and reduces the effort needed to override using stylesheets

Cons

- Responsive Web design is still a pain because you need to add a media breakpoint name to an atomic class in order to apply specific styling. So you end up with something like this:

```
<div class="D(ib)--sm W(50%)--sm W(25%)--lg P(20px) Bgc(#0280ae.5)">1</div>
<div class="D(ib)--sm W(50%)--sm W(25%)--lg P(20px) Bgc(#0280ae.6)">2</div>
<div class="D(ib)--sm W(50%)--sm W(25%)--lg P(20px) Bgc(#0280ae.8)">3</div>
<div class="D(ib)--sm W(50%)--sm W(25%)--lg P(20px) Bgc(#0280ae)">4</div>
```

(In which case it is almost better to use media querieis in css style sheets.)

- sometimes you really don't want this level of detail when composing an element, but a more general

component style will suffice such as label

Scalable Modular CSS

The idea behind this approach is really to organize your code or group the css into flexible elements which also promote long term maintainability. SMCSS uses the following categorizations for css:

- Base
- Layout
- Module
- State
- Theme

Why Categorize?

The idea here is to codify patterns, which can lead to resuable code. The SMCSS approach is fairly agnostic to any particular class naming convetion, however they do recommend that the different categorizies be condensed and used as prefixes for your styles. For example, one might use layout-row-2 or l-row-2 for a layout style. However, when it comes to modules and components it makese more sense to use the name of the component itself.

BEM Naming Convention

Block Element Modifier or (BEM) is an approach to naming css classes in order to bypass the specificity

problem and overriding hell in CSS.

```
.site-search {} /* Block */
.site-search__field {} /* Element */
.site-search--full {} /* Modifier */
```

Pitfalls in BEM

1. Grandchildren selectors. Don't use parent_grandparent_child like card_header_div_label. Instead think of the block level element that this element belongs to. if it is in a react component, this would be the Component name. So Header.jsx, and you would follow up by naming the element inside the Header. So you would have Header__text.

More examples of pitfalls

Pros

- Easy naming system
- More intuitive about the relationship between styling and the component
- easy to write rules to evaluate

Cons

- Modifiers can be confusing -- does it change state? Does it reflect a component attribute?
- Block level element makes sense in traditional HTML DOM structure, but what about in react context with components? Or what about when the block level element changes.

Modifier Problem

With BEM, it can be confusing to reason about which to use:

```
<!-- standalone state hook -->
<div class="c-card is-active">
    […]
</div>

<!-- or BEM modifier -->
<div class="c-card c-card--is-active">
    […]
</div>
```

In this case it almost seems better to use is-active class to address simple state change since this indicate DOM element change, not a component modifier.

BEM naming enhancements aka BEMIT

Hungarian Notation, you can use c-, for Components, o-, for Objects, u-, for Utilities, and is-/has- for States (there are plenty more detailed in the linked post).

Responsive suffixes using @ breakpoint.

```
<div class="o-media@md  c-user  c-user--premium">
  <img src="" alt="" class="o-media__img@md  c-user__photo  c-avatar" />
  <p class="o-media__body@md  c-user__bio">...</p>
</div>
```

```
@media print {
  /* you must escape the @ character */
  .u-hidden\@print {
    display: none;
  }
}
```

Enduring CSS

Takes a lot from BEM styling but uses the idea of a context space as a prefix. However, the modifiers are removed and instead replaced with the idea of a variant. And instead of Block Element, we use instead

Component_Node-variant. So the class would follow this convention ns-Component_Node-variant.

- ns : The micro-namespace (always lower-case)
- -Component: The Component name (always upper camel-case)
- _Node: The child node of a component (always upper camel-case preceded by an underscore)
- -variant: The optional variant of a node (always lower-case and preceded by a hyphen)

If you have a cardlist label node inside a card, you would look to the containing component (works nicely with react class syntax). So you might have sw-Card_Label.

What would a variant be?

It is optinoal and only used when there are slight variations between two nodes. For example you might two labels but they differ in terms of background color and text color. You might do sw-Card_Label-primary and sw-Card_Label-secondary.

The ten commandments of Enduring CSS

1. Thou shalt have a single source of truth for all key selectors
2. Thou shalt not nest, unless thou art nesting media queries or overrides
3. Thou shalt not use ID selectors, even if thou thinkest thou hast to
4. Thou shalt not write vendor prefixes in the authoring style sheets
5. Thou shalt use variables for sizing, colours and z-index
6. Thou shalt always write rules mobile first (avoid max-width)

7. Use mixins sparingly, and avoid @extend
8. Thou shalt comment all magic numbers and browser hacks
9. Thou shalt not inline images
10. Thou shalt not write complicated CSS when simple CSS will work just as well

Global Styles?

When the need arises, the author of Enduring CSS advocates us a globalCSS file.

> In that folder would be any variables, mixins, global image assets, any font or iconfont files, a basic CSS reset file and any global CSS needed.

Pros

- Reasoning about the Component architecture is more consistent with the actual js arhiecture
- The use of micro-namespacing helps declutter competing component styling in the app
- single class keeps specificty in check

Cons

- like with BEM's modifiers, state changes don't fit easily into the concept of a variant
- animations? what if you wanted to signal the transitional states of a component (fly-in onload)
- It can be easy to overmodularize and make everything a component

Summary of the above approaches

- You can do a compositional level approach using classes "p(10px) m(10px)" like that advocated by Atomic Css.
- You can use Single Class Rules, like BEM which enforces class pattern
- There are BEM variants which try to keep styling close to the source
- There are BEM variants one called Enduring CSS which uses sc-Component_Node-variant
- There's a growing request to use inline styles

Extra resources

Compare Selector Speed