Do we need CSS after all?

CSS is not immune to the framework wars. This is a quick look at some of the available CSS frameworks.

- Bootstrap
- Foundation
- Semantic-UI
- Susy
- Material UI
- MaterializeCSS
- Gumby
- Pure
- Metro UI CSS 2.0
- Leaf Beta

They all solve the same problem of laying out and styling content but they all have different styles to solve the same problem of providing consistent, reusable and scalable CSS libraries.

Different CSS frameworks present different approaches to building a good CSS suite of tools. The problem is that some of these frameworks (Bootstrap and Foundation in particular) have grown insanely large over the years and they haven't always provided a customizer to build slimmed down builds until recently (Later builds of Bootstrap 3, Bootstrap 4 and Foundation 5 are the only ones that offer this functionality).

Of course there are people who say that we should just dump CSS altogether and Javascript all the things on the web. You can access the CSS Object Model (CSSOM) via Javascript and the new CSS Typed OM Level 1 from the Houdini project promises to make this even easier.

If we take writing CSS in Javascript to the extreme are to dump CSS frameworks altogether and create all the CSS using Javascript, just like we could with Javascript Stylesheets back in the Netscape 4 days... well, maybe not quite the same way... after all Netscape 4 (4.0 to 4.8) was the only browser that supported JSS.

But I disgress...

The idea is that, since all styles are accessible through Javascript using the

style method of the HTMLElement object , we should be able to manipulate the inline styles of elements in the document programmatically. We don't have to use yet another programming language to create our content. Furthermore, because we are using a full fledged programming language we can create very powerful "style sheets" and not having to worry about silly things like the cascade and specificity.

Let's start with a simple example. The HTML looks like this

```
<div>
  <p id="content">Color Me</p>

  <button id="b1">RED</button>
  <button id="b2">BLUE</button>
  <button id="reset">RESET</button>
</div>
```

The associated scripts grabs the elements and assigns them to variables. Then we add click event listeners so when the use clicks on either button the color of the text in the paragraph (id = content) will change to the corresponding color. I've also added a third button to reset the color to black using the same method as the other two buttons.

```
var myContent = document.getElementById("content");
var button1 = document.getElementById("b1");
var button2 = document.getElementById("b2");
var button3 = document.getElementById("reset");

button1.addEventListener("click", () => {
   myContent.style.color="red";
   myContent.style.backgroundColor="blue";
} , false);

button2.addEventListener("click", () => {
  myContent.style.color="blue";
} , false);

button2.addEventListener("click", () => {
```

```
  myContent.style.color="black";
} , false);
```

Remember where I said that it the style method would only work with inline properties? This means that if you have a stylesheet associated with the document the style method will not work on these properties on the styleSheet.

We can get around it by using the, extremely cumbersome method show below. It's broken into three parts. The first part is the HTML we'll use to demo the system

```
<div class="boxes box1">
  <p>content goes here</p>
</div>
```

The second part is the CSS we'll modify in Javascript. It's two basic rules one for all elements with class `boxes` and one for the specific element with class `box1`.

```
.boxes {
  padding: 2em;
}

.box1 {
  background-color: green;
}
```

The final part is the Javascript that will modify the CSS.

We first extract the stylesheet we want to work on into a variable to save ourselves some typing.

We then navigate the rules of the stylesheet using a 0-based scale and change the style attribute for that rule using Javascript's camel-case syntax. In this example we change the first rule's second style (0-based) background color.

```
var stylesheet = document.styleSheets[1];
```

```
stylesheet.cssRules[1].style.backgroundColor="blue";
```

Getting styles for an element

Once again, let's start with the simple solution.

According to MDN:

> The `style` property is not useful for learning about the element's style in general, since it represents only the CSS declarations set in the element's inline style attribute, not those that come from style rules elsewhere, such as style rules in the <head> section, or external style sheets. To get the values of all CSS properties for an element you should use `window.getComputedStyle()` instead.

The Window.getComputedStyle() method gives the values of all the CSS properties of an element after applying the active stylesheets after all processing is done. The returned style is live and will update itself whenever the styles for the element change.

In large projects it may become counter productive to manually search for each property in every stylesheet. If you need to work with larger stylesheets you can use something like the function below to analyze the element you want to work with.

The function will log all the properties inlined into the element's style. It will then iterate over the element's computer style and display those values as well.

```
function dumpComputedStyles(elem, prop) {

  var cs = window.getComputedStyle(elem,null);

  if (prop) {
    console.log(prop + " : " + cs.getPropertyValue(prop));
    return;
  }
```

```
  var len = cs.length;
  for (var i = 0; i < len; i++) {

    var style = cs[i];
    console.log(style + " : " + cs.getPropertyValue(style));
  }

}
```

Enter the CSS typed object model

> Thanks to Surma for writing the article this is based on

The CSS Typed Object Model, part of the Houdini family of specifications, provides a solution to having to create styles like the following

```
getElementById('#someDiv').style.height = getRandomInt() + 'px';
```

We are doing math, converting a number to a string to append a unit just to have the browser parse that string and convert it back to a number for the CSS engine. This gets uglier when the more complex your styles become. This also means we should always know what the final unit type for our calculation is and that we can't skip any unit assignment or the script will fail in unpredictable ways.

Typed CSS will reduce some of these issues.

Instead of strings you will be working on an element's `StylePropertyMap` or `styleMap`, where each CSS attribute has it's own key and corresponding value type.

Attributes like width have `LengthValue` as their value type. A LengthValue is a dictionary of all CSS units like em, rem, px, percent, etc.

Some properties like box-sizing just accept certain keywords and therefore have a `KeywordValue` value type. The validity of those attributes could then be checked at runtime.

```
myElement.styleMap.set("opacity", new CSSNumberValue(3));
myElement.styleMap.set("z-index", new CSSNumberValue(15"z-index"nsole.log(

"z-index"// 15.4

var computedStyle = getComputedStyleMap(myElement);
var opacity = computedStyle.get("opacity");
var zIndex = computedStyle.get("z-index");
```

After executing the script, the value of opacity is 1 (opacity is range-restricted), and the value of zIndex is 15 (z-index is rounded to an integer value).

All the examples above use a single type of unit. But there are instances where we may want to work with multiple unit types, like in the example below:

```
<div style="margin-left: calc(5em + 50%);" id="div1"></div>
```

If we want to know how a Houdini enabled browser would handle this style we can query the styleMap for the document using something like the JS code below.

```
var myDiv = document.getElementById('#div1');
myDiv.styleMap.get('margin-left') 'margin-left'// => {em: 5, percent: 50}
```

For more information see the CSS Typed OM specification.

Mix and matching Javascript and CSS example: Font Face Observer

The best way I've seen of combining CSS and Javascript is the Font Face Observer library by Bram Stein. It leverages CSS and Javascript to make sure fonts are loaded successfully before they are used and that a suitable fallback is available if fonts fail to load. Instead of manipulating individual CSS selectors with Javascript we add and remove pre-defined classes.

The css is basic. We create three definitions of our body class:

- The first one loads when the classes for the following use cases are not loaded
- The second one (`.fonts-loaded`)matches when the fonts have loaded
- The last one (`.fonts-failed`) matches when the fonts fail to load

```
/* Basic font stack*/
body {
  font-family: Verdana, sans-serif;
}

/* Font stack when fonts are loaded */
.fonts-loaded body {
  font-family: "notosans_regular", Verdana, sans-serif;
}

"notosans_regular"/* Same font stack as basic but for when font loading fa
.fonts-failed body {
  font-family: Verdana, sans-serif;
}
```

The Javascript belows assumes that the Font Face Observer library has already been loaded in the page you're using.

Once we do that we create variables for each of the fonts we want to load using `new FontFaceObserver(font-name)` syntax.

We also add variables for `document.documentElement`.

```
var mono = new FontFaceObserver('notomono_regular');
var sans = new FontFaceObserver('notosans_regular');
var italic = new FontFaceObserver('notosans_italic');
var bold = new FontFaceObserver('notosans_bold');
var bolditalic = new FontFaceObserver('notosans_bolditalic');

var html = document.documentElement;
```

We first add the `.class-loading` attribute as a placeholder while we continue the loading process.

We use `Promise.all` to simultaneously load all the fonts we defined earlier. When all the fonts have loaded then we remove the `.fonts-loading` class and add the `.fonts-loaded` class instead.

If the fonts fail to load because it took longer than 3000 milliseconds or any other reason the class that will replace `.fonts-loading` is `.fonts-failed`

```
html.classList.add('fonts-loading');

Promise.all([
  mono.load(), sans.load(), bold.load(), italic.load(), bolditalic.load()
]).then(() => {
  html.classList.remove('fonts-loading');
  html.classList.add('fonts-loaded');
  console.log('All fonts have loaded.');
}).catch(() =>{
  html.classList.remove('fonts-loading');
  html.classList.add('fonts-failed');
  console.log('One or more fonts failed to load')
});
```

Since we configured different font stacks for each of the classes we get the a basic system font loaded quickly while the web font loads. Once it has loaded the web font replaces the original. Make sure you use similar fonts throughout the project. A really good tool to make sure your fonts match and what changes, if any, you need to make to your replacement fonts is Monica Dinculescu's Font style matcher

Also take into account that, in an ideal world, we would load 4 different fonts for each typeface we use: One for the following types:

- Normal font
- Bold
- Italics
- Bold / Italics

If the browser doesn't find a suitable bold or italic version of the typeface you are using it will create algorithmic replacements. As Alan Stearns puts it:

> Browsers can do terrible things to type. If text is styled as bold

> or italic and the typeface family does not include a bold or italic font, browsers will compensate by trying to create bold and italic styles themselves. The results are an awkward mimicry of real type design.
>
> Say No to Faux Bold

The downside is that 4 font faces can get really big in terms of file size. Be mindful on the size and number of resources you use on the page.

Links and Resources

- JavaScript and CSS
- Setting CSS Styles Using JavaScript
- Dynamic Style: Maniuplating CSS with Javascript
- JavaScript: Change CSS
- HTMLElement.style
- Javascript Stylesheets