# **CSS Color Architecture**

One of the hardest things in programming languages is organizing things in a way that will be easy to maintain and that ensures the code’s logic can be easily understood.

In this post, I want to explain how I work and organize the colors in our Appwrite design system project.

Like everything in life, we have a lot of good ways and even more wrong ways to do the same thing. Before I show you how I organize our CSS variables, let’s discuss the wrong ways to organize colors in CSS.



Note: *I have been aided by the Sass CSS preprocessor for some elements of the color architecture.*

# **Wrong Ways to Define CSS Color Variables**

In every method I encounter, I try to think of what is failing.

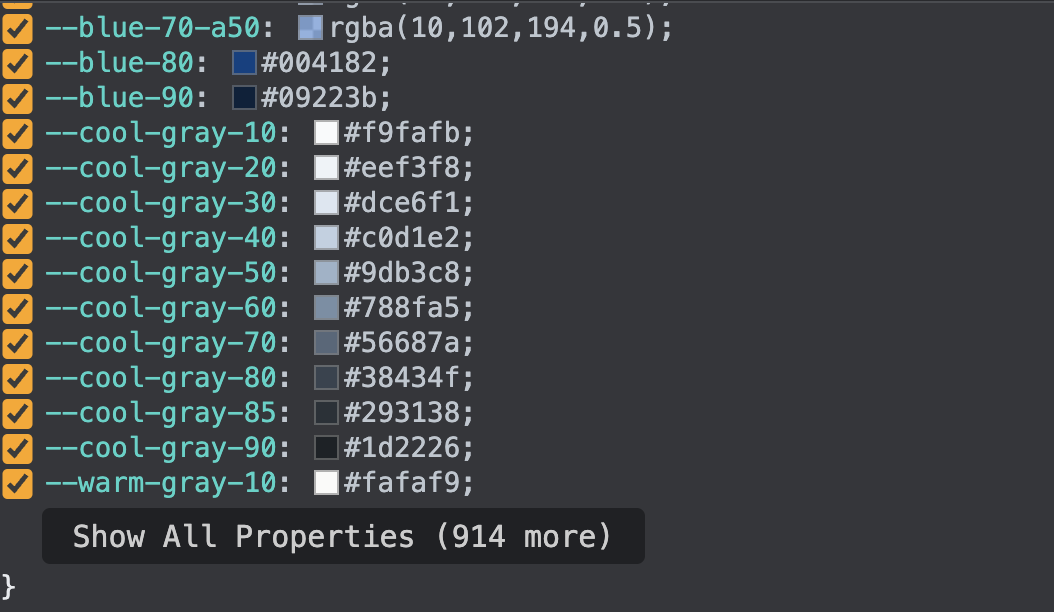
## 

## **Everything Global Method with Unsemantic Meaning**

This is a small example I took using the inspect feature from the root element on the LinkedIn website.

As you can see in the example below, there are more than 900 CSS variables on one :root selector!

In any situation, finding something in such an extensive list of variables is almost impossible.



## 

## **Logical Global Naming Convention with Meaningful Names**

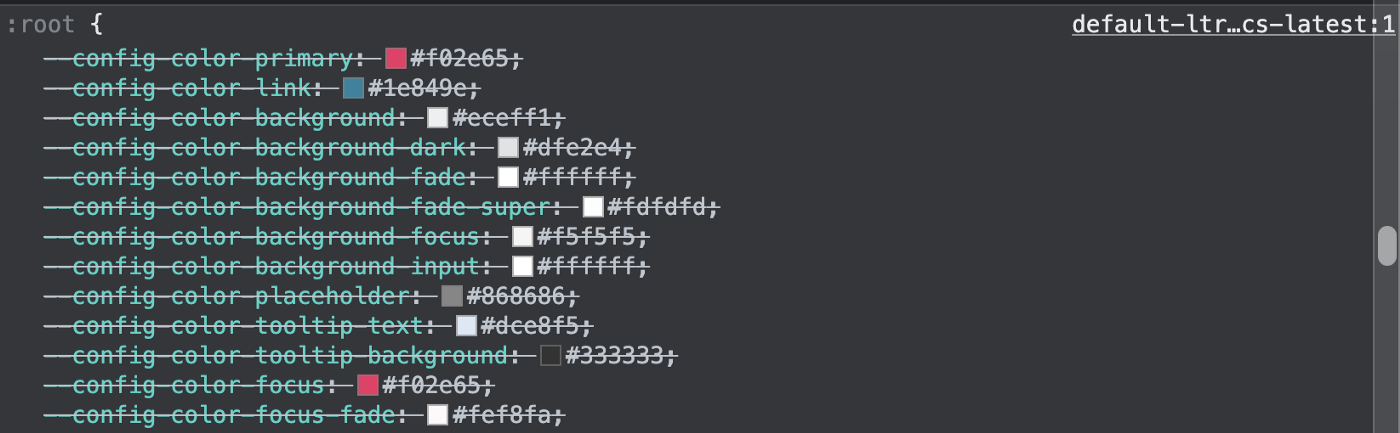
The problem with defining logical global naming, first and foremost, is that you have too many of them.

The second problem with creating dark-mode themes is that web designers do not work in the logic that specific colors need to change to other colors in dark mode. This can cause the creation of too many types of variables that are hard to understand or maintain.

**Example from old Appwrite console 1.0:**

****

Furthermore, the overrides of global colors make the code’s debugging unclear, with all the crossline of the overrides in the inspect element of Chrome.



After understanding these issues, I thought about how to better structure our CSS variables’ colors.

# 

# **Group Colors by Common Hue**

My main idea was to create a CSS variable for every group color family on the hue spectrum that would connect every group of colors, such as blue, green, orange, and red. In this way, I can change the primary hue of each one of the group colors and easily replace all types of this color family with another color by changing the primary hue of the group of colors.

The only issue I had with this approach was that the color had been defined with hex code colors, which are a type of RGB colors that do not follow any logic in a regular way of how the human mind interprets colors.



In this case, I chose to convert the HEX/RGB colors into HSL colors and try to find the typical hue of every group of colors.

## 

## **Converting from HEX/RGB to HSL**

If we take the information (blue) colors and convert them into HSL colors, we can see that all the hues (first value) are not the same, and they range between hue values 188 and 192.

The main idea was to keep the first value as another CSS variable.

:root {

--color-info-hue: /\* ? \*/;

--color-info-10: 189 87% 97%; /\* #F1FCFE \*/

--color-info-50: 192 90% 89%; /\* #C8F2FC \*/

--color-info-100: 189 100% 38%; /\* #00A7C3 \*/

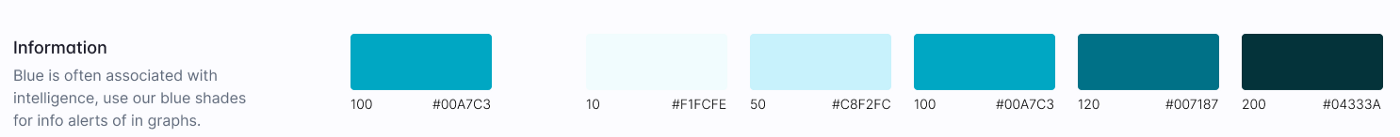
--color-info-120: 190 100% 26%; /\* #007187 \*/

--color-info-200: 188 87% 12%; /\* #04333A \*/

}

To solve this issue, I decided to use the CSS calc() function, which will subtract or add the difference to the base hue value.

I decided to take the base hue from the primary color from every group of colors; in our case, it was the “100” color, and for the info (blue) group of colors it had a hue value of 189.



The result looked like this:

:root {

--color-info-hue: 189;

--color-info-10: var(--color-information-hue) 87% 97%; /\* #F1FCFE \*/

--color-info-50: calc(var(--color-information-hue) + 3) 90% 89%; /\* #C8F2FC \*/

--color-info-100: var(--color-information-hue) 100% 38%; /\* #00A7C3 \*/

--color-info-120: calc(var(--color-information-hue) + 1) 100% 26%; /\* #007187 \*/

--color-info-200: calc(var(--color-information-hue) - 1) 87% 12%; /\* #04333A \*/

}

In this way, I can play with the hue and control and change all colors created from it.

## 

## **How to Use These Variables?**

To use the variables, every call of them needs to be wrapped with the hsl() function, for example:

background-color: hsl( var(--color-info-100) );

The reason I am not adding the hsl() function into the variable value itself is that I want an easy way to control the opacity of the colors if needed.

Example:

background-color: hsl( var(--color-info-100) / 0.5 ); /\* with 50% opacity \*/

In this way, I defined all the global colors used for all our modes, such as **light mode** (default)/**dark mode**.

# 

# **Private Local Variable Logic**

Because we have different color modes (light/dark modes), in most cases, every partial color will change to another color in the second color mode.

In my method, all the colors are global, both light and dark mode colors, meaning that I need every partial on variables.

To not have a “mess” of too many global variables, I use the concept of private variables for each partial.

To give the perspective that the variable is private, I start it with p-; for example:

.partial {

--p-variable-name: value;

}

In our button partial, for example, I have the main private variable for the text color, background-color, and border-color.

.button {

/\* Light-mode Theme \*/

--p-text-color: value;

--p-button-color: value;

--p-border-color: value;

}

The usage of those variables looks like this:

.button {

color: hsl( var(--p-text-color) );

background-color: hsl( var(--p-button-color) );

border-color: hsl( var(--p-border-color) );

}

Variables in a complex partial, such as a button, have a lot of states such as default/hover/focus/active/disabled. Those essential inner variables call other inner variables to present all those states.

The code of my button variables looks like this:

.button {

/\* Light Theme \*/

--p-text-color: var(--p-text-color-default);

--p-button-color: var(--p-button-color-default);

--p-border-color: var(--p-border-color-default);

--p-text-color-default: var(--color-neutral-5);

--p-button-color-default: var(--color-primary-200);

--p-border-color-default: var(--color-primary-300);

--p-text-color-hover: var(--p-text-color-default);

--p-button-color-hover: var(--color-primary-100);

--p-border-color-hover: var(--p-border-color-default);

--p-text-color-focus: var(--p-text-color-default);

--p-button-color-focus: var(--color-primary-200);

--p-border-color-focus: var(--color-primary-200);

--p-text-color-active: var(--p-text-color-default);

--p-button-color-active: var(--color-primary-300);

--p-border-color-active: var(--color-primary-300);

--p-text-color-disabled: var(--color-neutral-50);

--p-button-color-disabled: var(--color-neutral-10);

--p-border-color-disabled: var(--color-neutral-10);

}

## 

## **Defining States of Button**

What is nice, now, is that for every change I want to make to the buttons, I only need to update the variable’s value.

Thus, these variables will affect what they are being called by the main color variables for every state such as :hover, :focus, :active, :disabled, and so on. See the example below:

**Basic State Definitions (written in Sass):**

/\* global Sass Variable \*/

$disabled: ":disabled, .is-disabled";

.button {

&:is(:hover) {

&:where(:not(#{$disabled})) {

--p-text-color: var(--p-text-color-hover);

--p-button-color: var(--p-button-color-hover);

--p-border-color: var(--p-border-color-hover);

}

}

&:is(:focus-visible) {

&:where(:not(#{$disabled})) {

--p-text-color: var(--p-text-color-focus);

--p-button-color: var(--p-button-color-focus);

--p-border-color: var(--p-border-color-focus);

}

}

&:is(:active) {

&:where(:not(#{$disabled})) {

--p-text-color: var(--p-text-color-active);

--p-button-color: var(--p-button-color-active);

--p-border-color: var(--p-border-color-active);

}

}

&:where(#{$disabled}) {

--p-text-color: var(--p-text-color-disabled);

--p-button-color: var(--p-button-color-disabled);

--p-border-color: var(--p-border-color-disabled);

}

}

I use the Sass variable disabled so that I can use the style of the disabled button on other elements, such as link elements. Example:

**Sass Code:**

/\* global Sass Variable \*/

$disabled: ":disabled, .is-disabled";

.button {

&:where(#{$disabled}) {

--p-text-color: var(--p-text-color-disabled);

--p-button-color: var(--p-button-color-disabled);

--p-border-color: var(--p-border-color-disabled);

}

}

**Compiled CSS:**

.button:where(:disabled, .is-disabled) {

--p-text-color: var(--p-text-color-disabled);

--p-button-color: var(--p-button-color-disabled);

--p-border-color: var(--p-border-color-disabled);

}

**Will target:**

<button class="button" disabled> </button>

<a class="button is-disabled"> </a>

# 

# **Dark Mode Treatment**

After taking care of all the button light-mode states, we now want to take care of our dark-mode state.

Before doing so, I define another global Sass variable representing the dark-mode CSS class state. This state class name will be used in most of our partials to create unique colors for dark mode.

$theme-dark: ".theme-dark";

It is better to define this class on the <HTML> element, of course, only in the dark-mode state.

If defining it on the <HTML> element is an issue, it can be defined on the <body> element instead.

<body class="theme-dark"> </body>

This is done to achieve easy global control of all the HTML elements.

## 

## **Dark Mode Treatment Inside the Partial**

To create the definition of dark mode in the button partial, I add this code segment at the bottom of the partial:

.button {

/\* regular styles and light-mode definitions \*/

#{$theme-dark} & {

/\* definitions for dark-mode \*/

}

}

This Sass code will compile to this selector:

.button { /\* regular styles and light-mode definitions \*/ }

.theme-dark .button { /\* definitions for dark-mode \*/ }

Because all the states of the buttons are already declared, the only thing left to define is the dark mode theme; again, all the states’ private colors will need to be reviewed.

If some colors remain the same, they do not need to be updated in dark mode.

.button {

#{$theme-dark} & {

/\* changed colors \*/

--p-border-color-default: var(--color-primary-200);

--p-button-color-hover: var(--color-primary-100);

--p-border-color-hover: var(--color-primary-100);

--p-border-color-focus: var(--color-primary-300);

--p-border-color-active: var(--color-primary-300);

--p-text-color-disabled: var(--color-neutral-100);

--p-button-color-disabled: var(--color-neutral-150);

--p-border-color-disabled: var(--color-neutral-150);

}

}

What is nice about this method is that **we do not need to repeat any CSS selectors or any properties definitions**.

In this method, I only need to update the private state color variables that need to change.

# 

# **More Types of Buttons**

In our project, we needed to have more types of buttons.

Because we have already created a solid structure, we only need to define those variables according to the new state of the button.

## 

## **Define New State**

To define a new state, we add our new state class (is-secondary):

<button class=”button is-secondary”></button>

Now, to update the colors for the new type of button, we just use the private colors, one for light mode and the second for dark mode:

.button {

&.is-secondary {

/\* Light Mode \*/

--p-text-color-default: var(--color-neutral-100);

--p-button-color-default: var(--color-neutral-5);

--p-border-color-default: var(--color-neutral-30);

--p-text-color-hover: var(--p-text-color-default);

--p-button-color-hover: var(--color-neutral-10);

--p-border-color-hover: var(--p-border-color-default);

--p-text-color-focus: var(--p-text-color-default);

--p-button-color-focus: var(--p-button-color-default);

--p-border-color-focus: var(--transparent);

--p-text-color-active: var(--color-neutral-300);

--p-button-color-active: var(--color-neutral-30);

--p-border-color-active: var(--color-neutral-30);

--p-text-color-disabled: var(--color-neutral-50);

--p-button-color-disabled: var(--p-button-color-default);

--p-border-color-disabled: var(--color-neutral-30);

/\*\* Dark Mode \*\*/

#{$theme-dark} & {

--p-text-color-default: var(--color-neutral-5);

--p-button-color-default: var(--color-neutral-300);

--p-border-color-default: var(--color-neutral-150);

--p-text-color-hover: var(--p-text-color-default);

--p-button-color-hover: var(--transparent);

--p-border-color-hover: var(--color-neutral-120);

--p-text-color-focus: var(--p-text-color-default);

--p-button-color-focus: var(--p-button-color-default);

--p-border-color-focus: var(--transparent);

--p-text-color-active: var(--p-text-color-default);

--p-button-color-active: var(--p-button-color-default);

--p-border-color-active: var(--color-neutral-100);

--p-text-color-disabled: var(--color-neutral-100);

--p-button-color-disabled: var(--p-button-color-default);

--p-border-color-disabled: var(--color-neutral-150);

}

}

}

As you can see, we only define variables here, without any properties or any state selector pseudo-class such as hover, :focus, and so on.

CodePen Demo:

# 

# **Global Colors State**

In most cases, we do not want to define global color variables that are updated to other colors in dark mode.

However, while this is correct for “most cases,” in some cases we want to define a state color that looks like one specific color in light mode and another in dark mode.

## 

## **Global Logic Colors**

For that, I created another solution, which I am calling “global logic colors.”

For these, I created global CSS variables that are defined in a separate :root selector; of course, they call other global color variables.

For dark mode, these variables are changed to another global color variable. Example:

:root {

/\* Global Logic Colors – Light Mode \*/

--color-text-info: var(--color-info-100);

--color-text-danger: var(--color-danger-100);

--color-text-warning: var(--color-warning-100);

--color-text-success: var(--color-success-100);

--color-border: var(--color-neutral-10);

--scroll-color: var(--color-neutral-50);

#{$theme-dark} {

/\* Global Logic Colors – Dark Mode \*/

--color-text-info: var(--color-information-120);

--color-text-danger: var(--color-danger-120);

--color-text-warning: var(--color-warning-120);

--color-text-success: var(--color-success-120);

--color-border: var(--color-neutral-200);

--scroll-color: var(--color-neutral-150);

}

}

These CSS variables are used in two ways:

1. **Direct usage inside a partial**

.icon-checked { color: hsl( var(--color-text-success) ); }

1. **As a global utility class**

/\* Global Utilities colors classes \*/

.u-disabled { color: hsl( var(--color-text-disabled) ); }

.u-offline { color: hsl( var(--color-text-offline) ); }

.u-info { color: hsl( var(--color-text-info). ); }

.u-error { color: hsl( var(--color-text-danger). ); }

.u-warning { color: hsl( var(--color-text-warning). ); }

.u-success { color: hsl( var(--color-text-success) ); }

The global utility classes can be used directly on an element and will provide different colors according to the light mode or dark mode theme.

In both ways, the colors are updated according to the state of the color mode scheme.

# 

# **To Summarize**

In this post, I have shown you some of the ways that I use to solve complex structures of colors that we need in common complex cases.

# 

# **Final Words**

That’s all.

I hope you have enjoyed this article and learned from my experience.

If you like this post, I would appreciate likes and sharing :-)

You can follow me via [**Twitter**](https://twitter.com/eladsc).

**Who Am I?**

I am Elad Shechter, a CSS architect at Appwrite.