Everything you need to know about CSS Variables

Ohans Emmanuel Follow
Feb 12, 2018 · 21 min read

This is the first chapter of my new ebook (available in PDF & Mobi format).

Most programming languages have support for variables. But sadly, CSS has lacked support for native variables from the very beginning.

You write CSS? Then no variables for you. Well, except if you were using a preprocessor like Sass.

Preprocessors like Sass sell the use of variables as a big add-on. A reason to try them. And you know what? It's a pretty darn good reason.

Well the web is moving fast. And I'm glad to report that **CSS now finally supports variables**. Top highlight

While preprocessors support a lot more features, the addition of CSS variables is a good one. This moves the web even closer to the future.

In this guide, I'll show you how variables work natively in CSS, and how you can use them to make your life a lot easier.

What you'll Learn

I'll first walk you through the basics of CSS Variables. I believe any decent attempt at understanding CSS Variables must begin here.

Learning the fundamentals is cool. What's even cooler is applying these fundamentals to build real-world apps.

So I'll wrap things up by showing you how to build 3 projects that show off

CSS variables and their ease of use. Here's a quick preview of these 3 projects.

Project 1: Creating Component Variations using CSS Variables

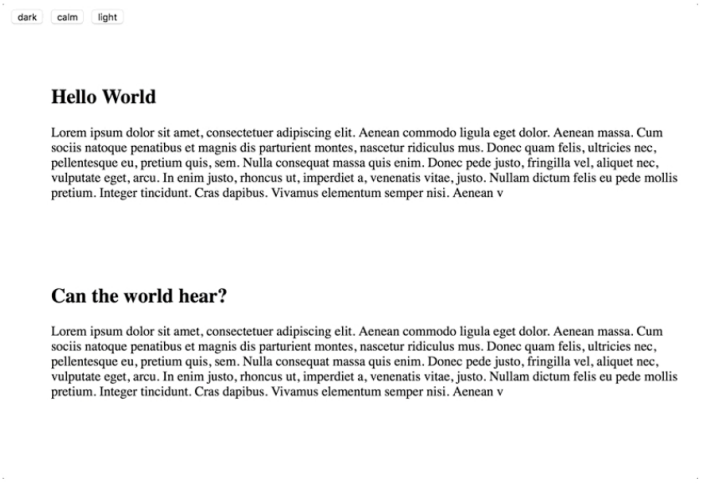
You may already be building component variations today. Whether you use React, Angular, or Vue, CSS Variables will make this process simpler.

Creating Component Variations using CSS variables

Check out the project on Codepen.

Project 2: Theme Styles with CSS Variables

You've likely see this somewhere. I'll show how easy CSS variables make creating site-wide theme styles.

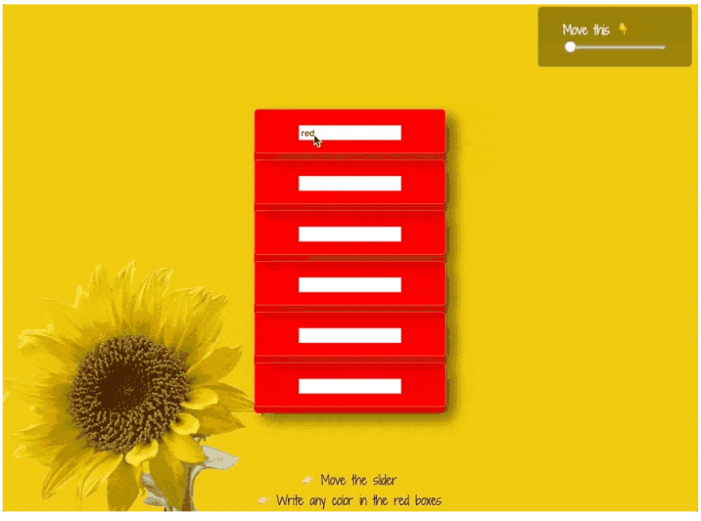


Site-wide theme styles using CSS variables

Check out the project on Codepen.

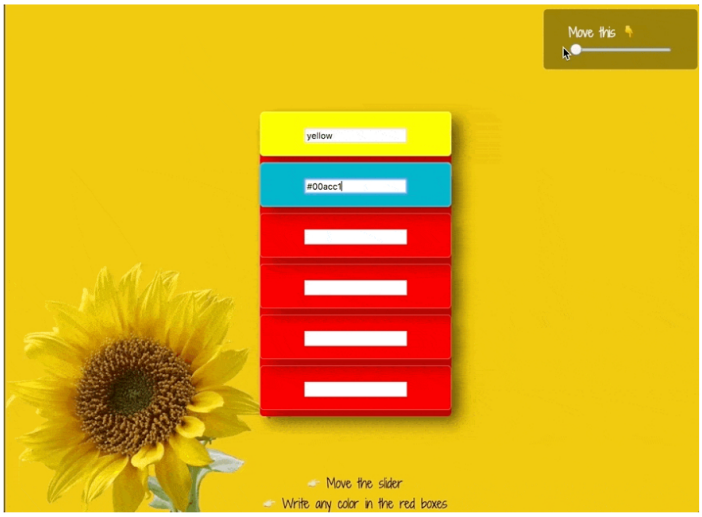
Project 3: Building the CSS Variable Booth

This is the final project. Don't mind the name. I couldn't come up with a better name.



The color of the boxes are dynamically updated

Notice how the colors of the boxes are dynamically updated, and how the box container is rotated in 3D space as the range input is changed.



This project shows off the ease of updating CSS variables with JavaScript, and the reactive goodies you get with it.

This is going to be fun!

Spend some time having fun with it on Codepen.

Note: The article assumes you have a good grasp of CSS. If you don't know CSS very well, or want to learn to create jaw-dropping UIs, I recommend taking my Advanced CSS Course (paid course that include 85 lessons). This article is an excerpt from the course. </Shameless plug> 😉

Why variables are so important

If you're new to variables in preprocessors or native CSS, here are a few reasons why variables are important.

Reason #1: More readable code

Without saying much, you can quickly tell how readable and more maintainable variables make any code base.

Reason #2: Ease of change across large documents

If you have all your constants saved in a separate file, you don't have to jump through thousands of lines of code when you want make a change to a variable.

It becomes super-easy. Just change it in one place, and voilà.

Reason #3: You can spot typos faster

It's a pain to search through lines of codes trying to spot an error. It's even more annoying if the error was due to a simple typo. They are difficult to spot. The good use of variables eliminates these hassles.

To this end, readability and maintainability are the big wins.

Thanks to CSS variables, now we can have these with native CSS too.

Defining CSS variables

Let me start with something you may already be familiar with: variables in JavaScript

JavaScript.

A simple JavaScript variable may be declared like so:

```
var amAwesome;
```

and then you can assign it some value like so:

```
amAwesome = "awesome string"
```

In CSS, a CSS variable is any "property" whose name begins with two dashes.

```
/*can you spot the variable here? */
.block {
 color: #8cacea;
--color: blue
}
```

CSS Variables are also called "Custom Properties"

Scoping CSS Variables

There's one more thing to point your attention to.

Remember that in JavaScript, variables have a scope. They may either have a `global` or `local` scope.

The same may be said of CSS variables.

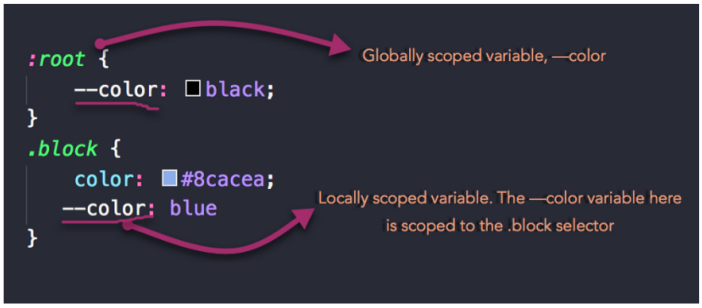
Consider the example below:

```
:root {
  --main-color: red
}
```

The `:root` selector allows you to target the highest-level element in the DOM, or document tree.

So, variables declared in this way are kind of scoped to the global scope.

Got that?



Example 1

Assuming you wanted to create a CSS variable that stored the primary color of a themed site.

How would you go about this?

1. You create the scoped selector. Use `:root` for a 'global' variable

```
:root {

}
```

2. Define the variable

```
:root {
 --primary-color: red
}
```

Remember, a CSS variable is any “property” whose name begins with two dashes e.g `--color`

That was simple.

Using CSS Variables

Once a variable has been defined and assigned a value, you can go ahead and use it within a property value.

There’s a bit of a gotcha though.

If you’re coming from the world of preprocessors, you must be used to using a variable by just referencing its name. For example:

```
$font-size: 20px

.test {
  font-size: $font-size
}
```

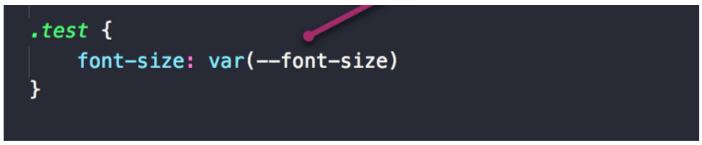
With native CSS variables, things are a little different. You reference a variable by using the `var()` function.

With the example above, using CSS Variables will yield this:

```
:root {
  --font-size: 20px
}

.test {
  font-size: var(--font-size)
}
```

Quite different.

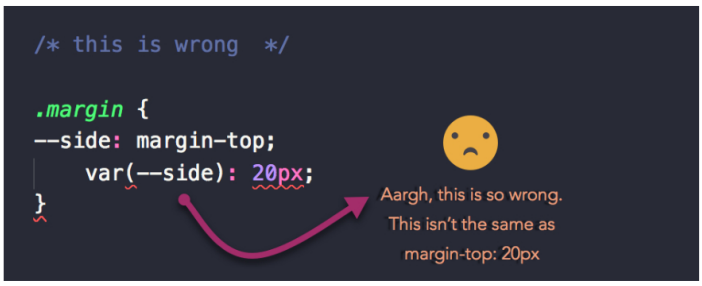


Remember to use the var function

Once you get that out of the way, you'll start to love CSS variables - a lot!

Another important note is that, unlike variables in Sass (or other preprocessors)—where you can use the variables in a lot of places, and do math like you want—you need to be careful with CSS variables. You'll mostly have them set as property values.

```
/*this is wrong*/
.margin {
--side: margin-top;
var(--side): 20px;
}
```



The declaration is thrown away as a syntax error for having an invalid property name

You also can't do math. You need the CSS `calc()` function for that. I'll discuss examples as we proceed.

```
/*this is wrong */
.margin {
--space: 20px * 2;
font-size:  var(--space);  //not 40px
}
```

If you must do math, then use the calc() function like so:

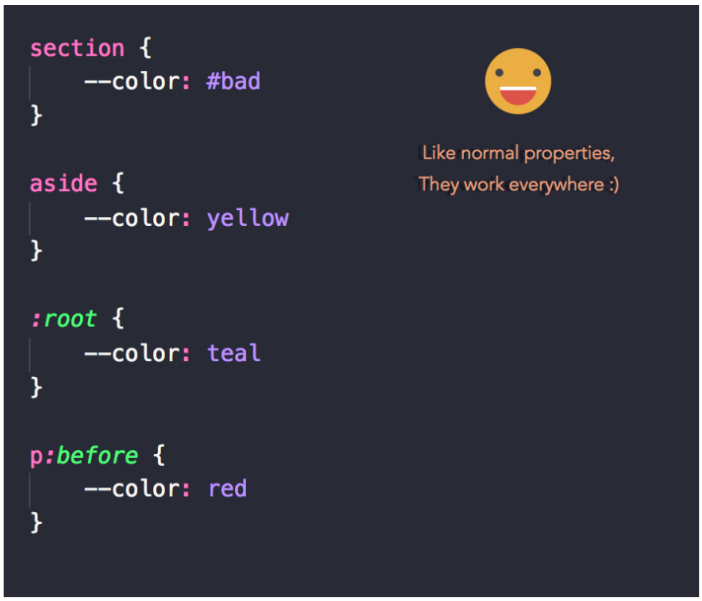
```
.margin {
--space: calc(20px * 2);
font-size:  var(--space);  /*equals 40px*/
}
```

Properties Worthy of Mention

Here are some behaviors that are worth mentioning.

1. Custom properties are ordinary properties, so they can be declared on any element.

Declare them on a paragraph element, section, aside, root, or even pseudo elements. They'll work as expected.



They behave like normal properties

2. CSS variables are resolved with the normal inheritance and cascade rules

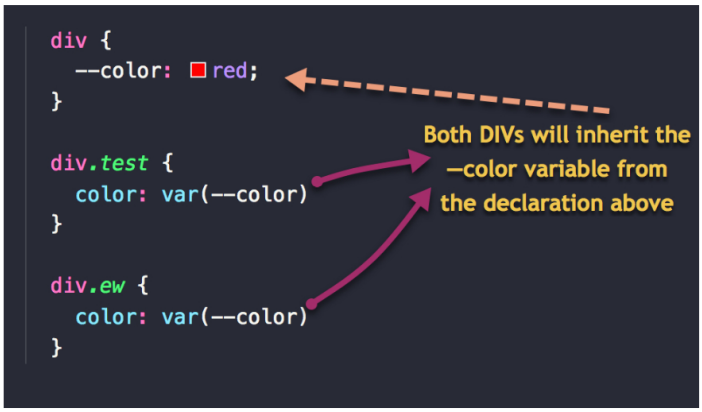
Consider the block of code below:

```
div {
  --color: red;
}

div.test {
  color: var(--color)
}

div.ew {
  color: var(--color)
}
```

As with normal variables, the `--color` value will be inherited by the divs.



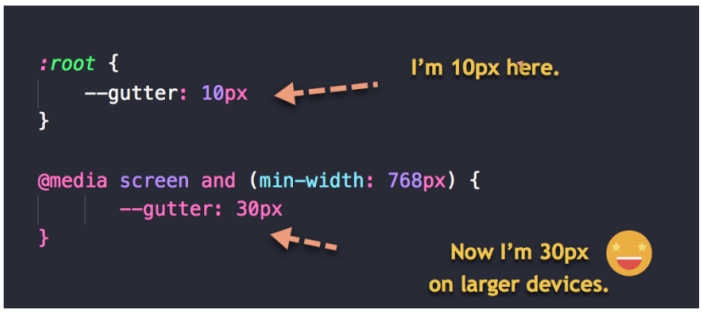
3. CSS variables can be made conditional with @media and other conditional rules

As with other properties, you can change the value of a CSS variable within a `@media` block or other conditional rules.

For example, the following code changes the value of the variable, gutter on larger devices.

```
:root {
 --gutter: 10px
}
```

```
@media screen and (min-width: 768px) {
    --gutter: 30px
}
```



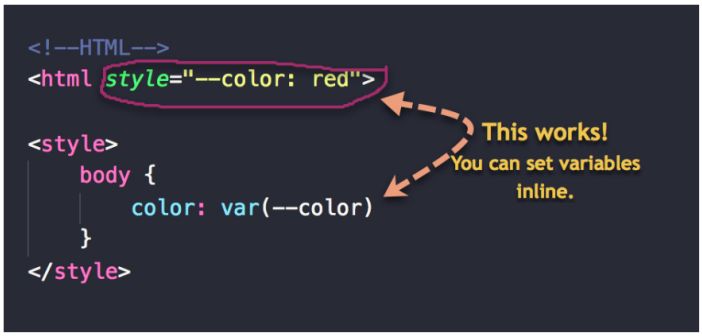
Useful bit for responsive design

4. CSS variables can be used in HTML's style attribute.

You can choose to set the value of your variables inline, and they'll still work as expected.

```
<!--HTML-->
<html style="--color: red">

<!--CSS-->
body {
  color: var(--color)
}
```



Set variables inline

CSS variables are case-sensitive. Be careful with this one. I save myself the stress and write variables in the lower case. Your mileage may differ.

```
/*these are two different variables*/
:root {
 --color: blue;
--COLOR: red;
}
```

Resolving Multiple Declarations

As with other properties, multiple declarations are resolved with the standard cascade.

Let's see an example:

```
/*define the variables*/
:root { --color: blue; }
div { --color: green; }
#alert { --color: red; }

/*use the variable */
```

```
* { color: var(--color); }
```

With the variable declarations above, what will be the color of the following elements?

```
<p>What's my color?</p>
<div>and me?</div>
<div id='alert'>
  What's my color too?
  <p>color?</p>
</div>
```

Can you figure that out?

The first paragraph will be `blue`. There is no direct `--color` definition set on a `p` selector, so it inherits the value from `:root`

```
:root { --color: blue; }
```

The first `div` will be `green` . That's pretty clear. There's a direct variable definition set on the `div`

```
div { --color: green; }
```

The `div` with the ID of `alert` will NOT be green. It will be `red`

```
#alert { --color: red; }
```

The ID has a direct variable scoping. As such, the value within the definition will override the others. The selector `#alert` is more specific.

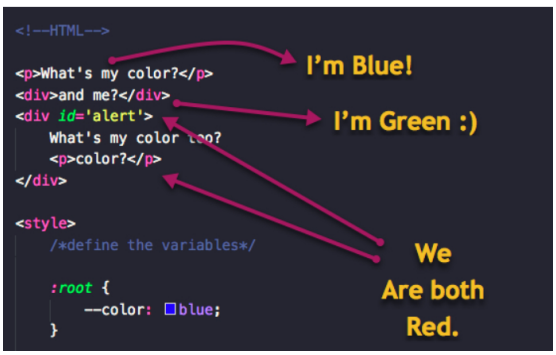
Finally, the `p` within the `#alert` will be… `red`

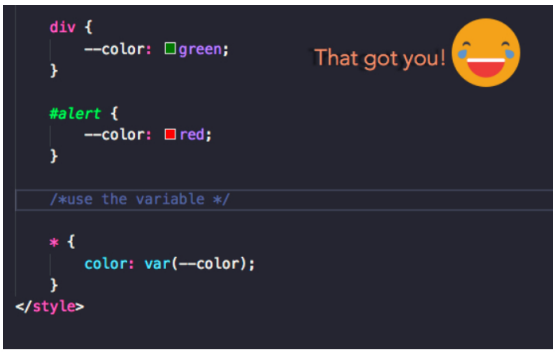
There's no variable declaration on the paragraph element. You may have expected the color to be `blue` owing to the declaration on the `:root` element.

```
:root { --color: blue; }
```

As with other properties, CSS variables are inherited. The value is inherited from the parent, `#alert`

```
#alert { --color: red; }
```





The solution to the Quiz

Resolving Cyclic Dependencies

A cyclic dependency occurs in the following ways:

1. When a variable depends on itself. That is, it uses a `var()` that refers to itself.

```
:root {
  --m: var(--m)
}

body {
  margin: var(--m)
}
```

2. When two or more variables refer to each other.

```
:root {
  --one: calc(var(--two) + 10px);
  --two: calc(var(--one) - 10px);
}
```

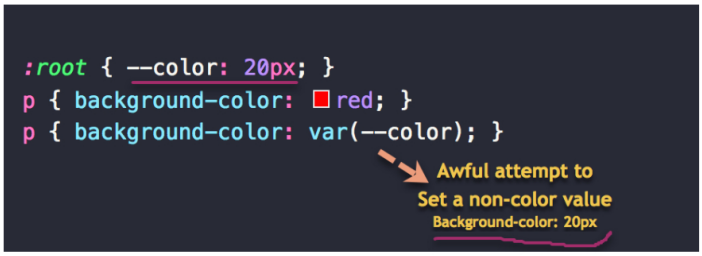
Be careful not to create cyclic dependencies within your code.

What Happens with Invalid Variables?

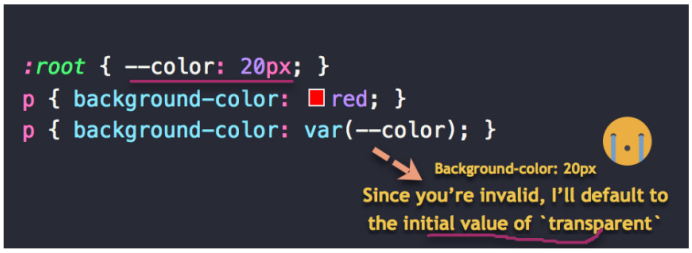
Syntax errors are discarded, but invalid `var()` substitutions default to either the initial or inherited value of the property in question.

Consider the following:

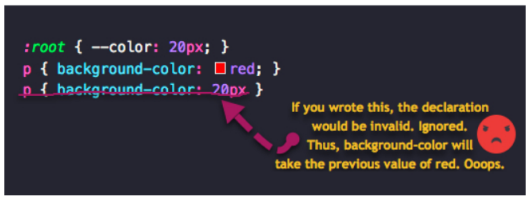
```
:root { --color: 20px; }
p { background-color: red; }
p { background-color: var(--color); }
```



As expected, `--color` is substituted into `var()` but the property value, `background-color: 20px` is invalid after the substitution. Since `background-color` isn't an inheritable property, the value will default to its `initial` value of `transparent`.



Note that if you had written `background-color: 20px` without any variable substitutes, the particular background declaration would have been invalid. The previous declaration will then be used.



The case is different when you write the declaration yourself

Be Careful While Building Single Tokens

When you set the value of a property as indicated below, the `20px` is interpreted as a single token.

```
font-size: 20px
```

A simple way to put that is, the value `20px` is seen as a single 'entity.'

You need to be careful when building single tokens with CSS variables.

For example, consider the following block of code:

```
:root {
 --size: 20
}

div {
  font-size: var(--size)px /*WRONG*/
}
```

You may have expected the value of `font-size` to yield `20px`, but that is wrong.

The browser interprets this as `20 px`

Note the space after the `20`

Thus, if you must create single tokens, have a variable represent the entire token. For example, `--size: 20px`, or use the `calc` function e.g `calc(var(--size) * 1px)` where `--size` is equal to `20`

Don't worry if you don't get this yet. I'll explain it in more detail in a coming example.

Let's build stuff!

Now this is the part of the article we've been waiting for.

I'll walk you through practical applications of the concepts discussed by building a few useful projects.

Let's get started.

Project 1: Creating Component Variations using CSS Variables

Consider the case where you need to build two different buttons. Same base styles, just a bit of difference.

In this case, the properties that differ are the `background-color` and `border-color` of the variant.

So, how would you do this?

Here's the typical solution.

Create a base class, say `.btn` and add the variant classes. Here's an example markup:

```
<button class="btn">Hello</button>
<button class="btn red">Hello</button>
```

`.btn` would contain the base styles on the button. For example:

```
.btn {
  padding: 2rem 4rem;
  border: 2px solid black;
  background: transparent;
  font-size: 0.6em;
  border-radius: 2px;
}

/*on hover */
.btn:hover {
  cursor: pointer;
  background: black;
  color: white;
}
```

So, where does the variant come in?

Here:

```
/* variations */
```

```
.btn.red {
  border-color: red
}
.btn.red:hover {
  background: red
}
```

You see how we are duplicating code here and there? This is good, but we could make it better with CSS variables.

What's the first step?

Substitute the varying colors with CSS variables, and don't forget to add default values for the variables!

```
.btn {
   padding: 2rem 4rem;
   border: 2px solid var(--color, black);
   background: transparent;
   font-size: 0.6em;
   border-radius: 2px;
 }

 /*on hover*/
 .btn:hover {
  cursor: pointer;
   background: var(--color, black);
   color: white;
 }
```

When you do this: `background: var(--color, black)` you're saying, set the background to the value of the variable `--color` . However, if the variable doesn't exist, use the default value of `black`

This is how you set default variable values. Just like you do in JavaScript or any other programming language.

Here's the good part.

With the variants, you just supply the new value of the CSS variable as under:

```
.btn.red {
   --color: red
 }
```

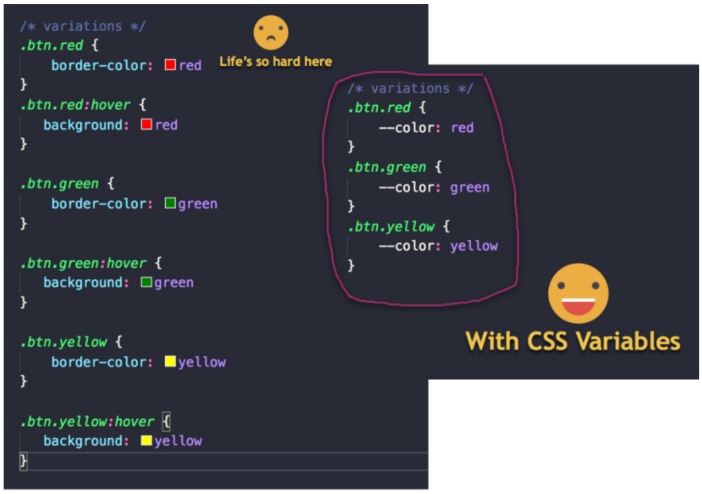
That's all. Now when the `.red` class is used, the browser notes the different `--color` variable value, and immediately updates the appearance of the button.

This is really good if you spend a lot of time building reusable components.

Here's a side by side comparison:

Without CSS Variables VS with CSS Variables

Oh, and if you had more variants, you just saved yourself a lot of extra typing.

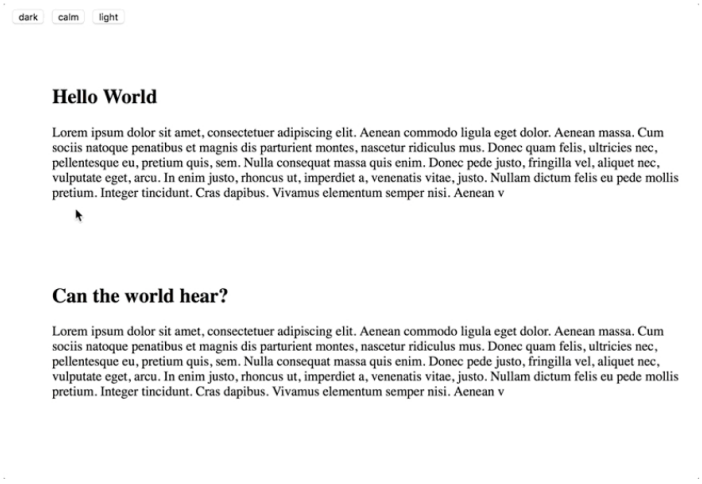


See the difference??

Project 2: Themed Sites with CSS Variables

I'm sure you've come across them before. Themed sites give the user the feel of customization. Like they are in control.

Below is the basic example we'll build.



So, how easy do the CSS variables make this?

We'll have a look.

Just before that, I wanted to mention that this example is quite important. With this example, I'll introduce the concept of updating CSS variables with JavaScript.

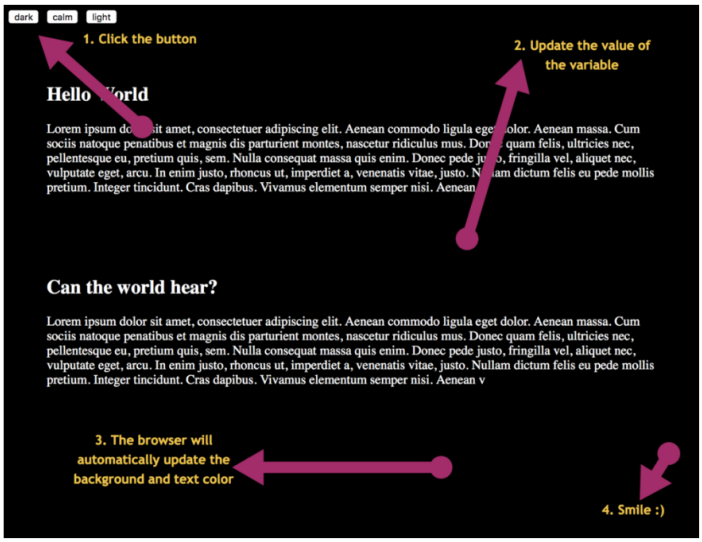
It is fun!

You'll love it.

What we really want to do.

The beauty of CSS variables is their reactive nature . As soon as they are updated, whatever property has the value of the CSS variable gets updated as well.

Conceptually, here's an image that explains the process with regards to the example at hand.



The process

So, we need some JavaScript for the click listener.

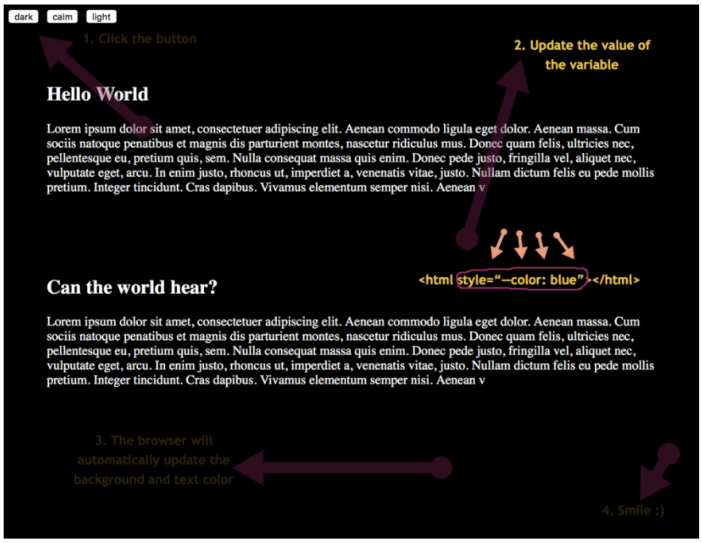
For this simple example, the background and color of the text of the entire page is based off of CSS variables.

When you click any of the buttons above, they set the CSS variable to some other color. As a result of that, the background of the page is updated.

Hey, that's all there is to it.

Uh, one more thing.

When I say the CSS variable is set to some other value, how's that done?



Set the variable inline

CSS variables will take effect even if they are set inline. With JavaScript, we get a hold of the root document, and we set the new value for the CSS variable inline.

Got that?

That's a lot of talking... let's do the real thing.

That's a lot of talking—let's do the real thing.

The initial markup

The initial markup needed is this:

```
<div class="theme">
  <button value="dark">dark</button>
  <button value="calm">calm</button>
  <button value="light">light</button>
</div>

<article>
...
</article>
```

The markup consists of three buttons within a `.theme` parent element. To keep things short I have truncated the content within the `article` element. Within this `article` element is the content of the page.

Styling the Page

The success of this project begins with the styling of the page. The trick is simple.

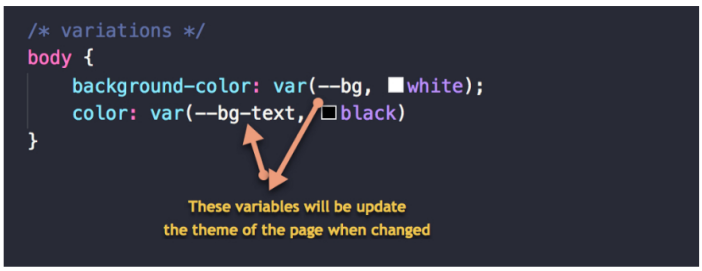
Instead of just setting the `background-color` and `color` of the page in stone, we will set them based on variables.

Here's what I mean.

```
body {
  background-color: var(--bg, white);
  color: var(--bg-text, black)
}
```

The reason for this is kind of obvious. Whenever a button is clicked, we will change the value of both variables within the document.

Upon this change, the overall style of the page will be updated. Easy-peasy.



So, let's go ahead and handle the update from JavaScript.

Getting into the JavaScript

I'll go ahead and spit out all the JavaScript needed for this project.

```
const root = document.documentElement
const themeBtns = document.querySelectorAll('.theme > button')

themeBtns.forEach((btn) => {
  btn.addEventListener('click', handleThemeUpdate)
})

function handleThemeUpdate(e) {
  switch(e.target.value) {
    case 'dark':
      root.style.setProperty('--bg', 'black')
```

```
      root.style.setProperty('--bg-text', 'white')
      break
    case 'calm':
       root.style.setProperty('--bg', '#B3E5FC')
       root.style.setProperty('--bg-text', '#37474F')
      break
    case 'light':
      root.style.setProperty('--bg', 'white')
      root.style.setProperty('--bg-text', 'black')
      break
  }
}
```

Don't let that scare you. It's a lot easier than you probably think.

First off, keep a reference to the root element, `const root = document.documentElement`

The root element here is the `HTML` element. You'll see why this is important in a bit. If you're curious, it is needed to set the new values of the CSS variables.

Also, keep a reference to the buttons too, `const themeBtns = document.querySelectorAll('.theme > button')`

`querySelectorAll` yields an array-like data structure we can loop over. Iterate over each of the buttons and add an event listener to them, upon click.

Here's how:

```
themeBtns.forEach((btn) => {
  btn.addEventListener('click', handleThemeUpdate)
})
```

Where's the `handleThemeUpdate` function? I'll discuss that next.

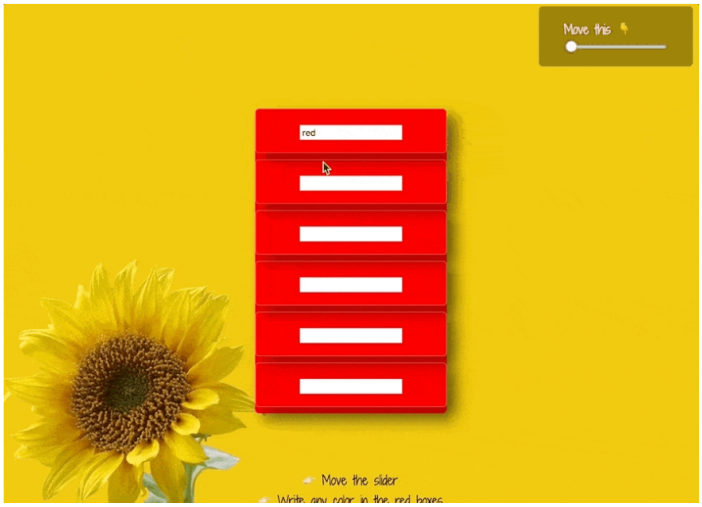
Every button being clicked will have the `handleThemeUpdate` as its callback function. It becomes important to note what button was clicked and then perform the right operation.

In the light of that, a switch `operator` is used, and some operations are carried out based on the value of the button being clicked.

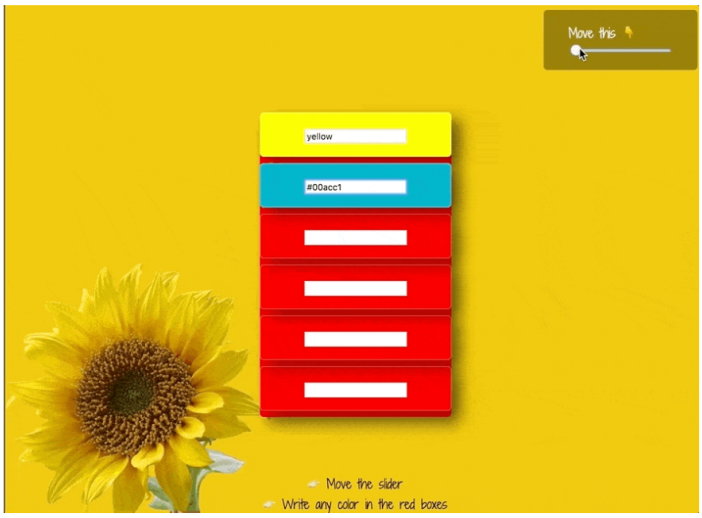
Go ahead and take a second look at the block of JavaScript code. You'll understand it a lot better now.

Project 3: Building the CSS Variable Booth 💥

In case you missed it, here's what we'll build:



Remember that the color of the boxes are dynamically updated, and that the box container is rotated in 3d space as the range input is changed.



You can go ahead and play with it on Codepen.

This is a superb example of updating CSS variables with JavaScript and the reactivity that comes with it.

Let's see how to build this.

The Markup

Here are the needed components.

1. A range input
2. A container to hold the instructions
3. A section to hold a list of other boxes, each containing input fields

The markup turns out simple.

Here it is:

```
<main class="booth">
  <aside class="slider">
    <label>Move this 👇 </label>
    <input class="booth-slider" type="range" min="-50" max="50"
value="-50" step="5"/>
  </aside>

  <section class="color-boxes">
    <div class="color-box" id="1"><input value="red"/></div>
    <div class="color-box" id="2"><input/></div>
    <div class="color-box" id="3"><input/></div>
    <div class="color-box" id="4"><input/></div>
    <div class="color-box" id="5"><input/></div>
```

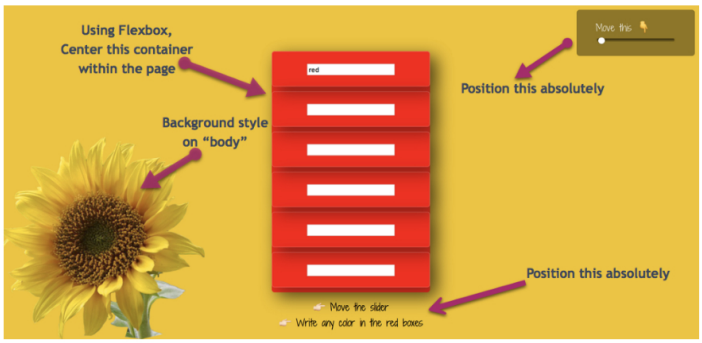
```
    <div class="color-box" id="6"><input/></div>
  </section>

  <footer class="instructions">
    👉 Move the slider<br/>
    👉 Write any color in the red boxes
  </footer>
</main>
```

Here are a few things to point your attention to.

1. The range input represents values from `-50` to `50` with a step value of `5` Also, the value of the range input is the minimum value, `-50`
2. If you aren't sure how the range input works, check it out on w3schools
3. Note how the section with class `.color-boxes` contains other `.color-box` containers. Within these containers exist input fields.
4. It is perhaps worth mentioning that the first input has a default value of red.

Having understood the structure of the document, go ahead and style it like so:



1. Take the `.slider` and `.instructions` containers out of the document flow. Position them absolutely.
2. Give the `body` element a sunrise background color and garnish the background with a flower in the bottom left corner
3. Position the `color-boxes` container in the center
4. Style the `color-boxes` container

Let's knock these off.

The following will fix the first task.

```
/* Slider */
.slider,
.instructions {
  position: absolute;
  background: rgba(0,0,0,0.4);
  padding: 1rem 2rem;
  border-radius: 5px
}
.slider {
  right: 10px;
  top: 10px;
}
.slider > * {
  display: block;
}


/* Instructions */
.instructions {
  text-align: center;
  bottom: 0;
  background: initial;
  color: black;
```

```
}
```

The code snippet isn't as complex as you think. I hope you can read through and understand it. If not, drop a comment or tweet.

Styling the `body` is a little more involved. Hopefully, you understand CSS well.

Since we aspire to style the element with a background color and a background image, it's perhaps the best bet to use the `background` shorthand property to set multiple backgrounds.

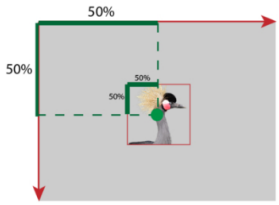
Here it is:

```
body {
  margin: 0;
  color: rgba(255,255,255,0.9);
  background: url('http://bit.ly/2FiPrRA') 0 100%/340px no-repeat,
var(--primary-color);
  font-family: 'Shadows Into Light Two', cursive;
}
```

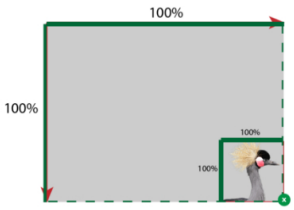
The `url` bit is the link to the sunrise flower.

The next set of properties `0 100%` represent the background position of the image.

Here's an illustration of how CSS background positioning works:



From: the advanced guide to CSS



From: the advanced guide to CSS

The other bit after the forward slash represents the `background-size` This has been set to `340px` If you made this smaller, the image would be smaller too.

`no-repeat`, you might figure out what that does. It prevents the background from repeating itself.

Finally, anything that comes after the comma is a second background declaration. This time we've only set the `background-color` to `var(primary-color)`

Oops, that's a variable.

The implication of this is that you have to define the variable. Here's how:

```
:root {
  --primary-color: rgba(241,196,15 ,1)
}
```

The primary color there is the sunrise yellow color. No big deal. We'll set some more variables in there soon.

Now, let's center the `color-boxes`

```
main.booth {
  min-height: 100vh;

  display: flex;
  justify-content: center;
  align-items: center;
}
```

The main container acts as a flex container and rightly positions the direct child in the center of the page. This happens to be our beloved `color-box` container

Let's make the color-boxes container and its children elements pretty.

First, the child elements:

```
.color-box {
  padding: 1rem 3.5rem;
  margin-bottom: 0.5rem;
  border: 1px solid rgba(255,255,255,0.2);
  border-radius: 0.3rem;
  box-shadow: 10px 10px 30px rgba(0,0,0,0.4);
}
```

That will do it. There's a beautiful shadow added too. That'll get us some cool effects.

That is not all. Let's style the overall `container-boxes` container:

```
/* Color Boxes */
.color-boxes {
  background: var(--secondary-color);
  box-shadow: 10px 10px 30px rgba(0,0,0,0.4);
  border-radius: 0.3rem;

  transform: perspective(500px) rotateY( calc(var(--slider) *
1deg));
  transition: transform 0.3s
}
```

Oh my!

There's a lot in there.

Let me break it down.

Here's the simple bit:

```
.color-boxes {
   background: var(--secondary-color);
   box-shadow: 10px 10px 30px rgba(0,0,0,0.4);
   border-radius: 0.3rem;
```

```
  border-radius: 0.31em;
}
```

You know what that does, huh?

There's a new variable in there. That should be taken of by adding it to the root selector.

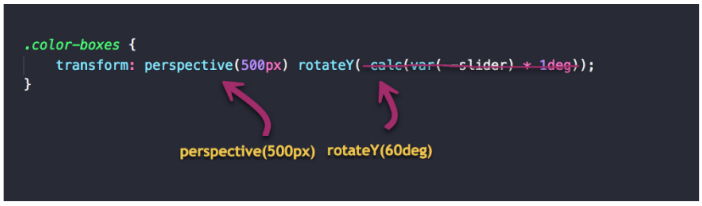
```
:root {
  --primary-color: rgba(241,196,15 ,1);
  --secondary-color: red;
}
```

The secondary color is red. This will give the container a red background.

Now to the part that probably confused you:

```
/* Color Boxes */
.color-boxes {
  transform: perspective(500px) rotateY( calc(var(--slider) *
1deg));
  transition: transform 0.3s
}
```

For a moment, we could simplify the value of the transform property above.



For example:

```
transform: perspective(500px) rotateY( 30deg);
```

The transform shorthand applies two different functions. One, the perspective, and the other, the rotation along the Y axis.

Hmmm, so what's the deal with the `perspective` and `rotateY` functions?

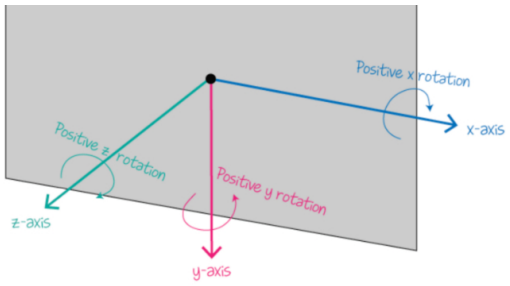
The perspective() function is applied to an element that is being transformed in 3D space. It activates the three dimensional space and gives the element depth along the z-axis.

You can read more about the perspective function on codrops.

The `rotateY` function, what's the deal with that?

Upon activation the 3d space, the element has the planes x, y, z. The `rotateY` function rotates the element along the `Y` plane.

The following diagram from codrops is really helpful for visualizing this.



The positive direction of rotation along the three axes. Notice how if you stand at the end of each vector and look towards the origin, the clockwise rotation matches the one shown in the image.

Codrops

I hope that blew off some of the steam.

Back to where we started.

```
.color-boxes {
    transform: perspective(500px) rotateY(calc(var(--slider) * 1deg));
}
```

perspective(500px) rotateY(60deg)

When you move the slider, do you know what function affects the rotation of the `.container-box` ?

It's the rotateY function being invoked. The box is rotated along the Y axis.

Since the value passed into the rotateY function will be updated via JavaScript, the value is represented with a variable.

```
.color-boxes {
    transform: perspective(500px) rotateY( calc(var(--slider) * 1deg));
}
```

CSS variable

So, why multiply by the variable by 1deg?

As a general rule of thumb, and for explicit freedom, it is advised that when building single tokens, you store values in your variables without units.

You can convert them to any unit you want by doing a multiplication via the `calc` function.

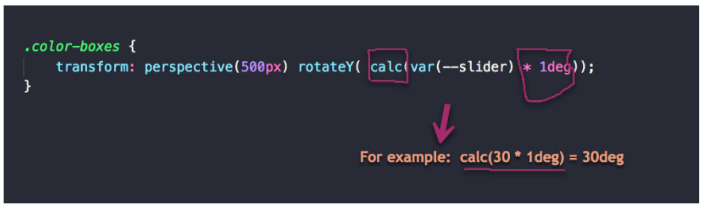
```
.color-boxes {
    transform: perspective(500px) rotateY( calc(var(--slider) * 1deg));
}
```

Plain number

This allows you to do 'whatever' you want with these values when you have them. Want to convert to `deg` or as a ratio of the user's viewport `vw` , you can whatever you want.

In this case, we are converting the number to have a degree by multiplying the

In this case, we are converting the number to have a degree by multiplying the "number" value by 1deg



Since CSS doesn't understand math, you have to pass this arithmetic into the calc function to be properly evaluated by CSS.

Once that is done, we're good to go. The value of this variable can be updated in JavaScript as much as we like.

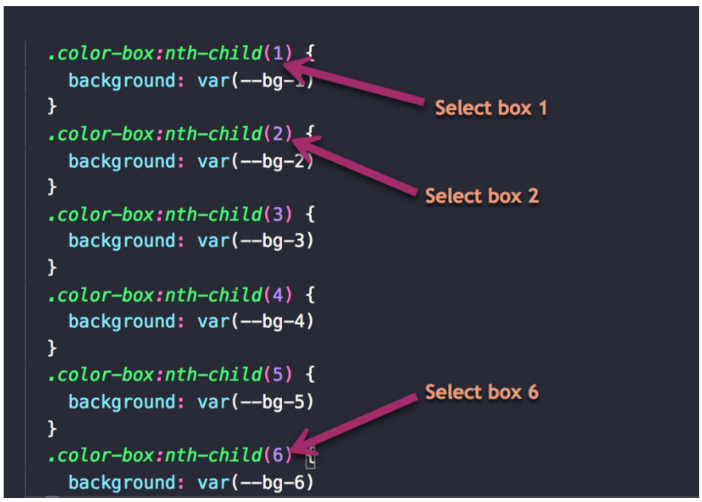
Now, there's just one bit of CSS remaining.

Here it is:

```
/* Handle colors for each color box */
.color-box:nth-child(1) {
  background: var(--bg-1)
}
.color-box:nth-child(2) {
  background: var(--bg-2)
}
.color-box:nth-child(3) {
  background: var(--bg-3)
}
.color-box:nth-child(4) {
  background: var(--bg-4)
}
.color-box:nth-child(5) {
  background: var(--bg-5)
}
.color-box:nth-child(6) {
  background: var(--bg-6)
}
```

So, what's this voodoo?

First off, the nth-child selector selects each of the child boxes.



There's a bit of foresight needed here. We know we will be updating the background color of each box. We also know that this background color has to be represented by a variable so it is accessible via JavaScript. Right?

We could go ahead and do this:

```
.color-box:nth-child(1) {
  background: var(--bg-1)
}
```

Easy.

There's one problem though. If this variable isn't present, what happens?

We need a fallback.

This works:

```
.color-box:nth-child(1) {
  background: var(--bg-1, red)
}
```

In this particular case, I have chosen NOT to provide any fallbacks.

If a variable used within a property value is invalid, the property will take on its initial value.

Consequently, when `--bg-1` is invalid or NOT available, the background will default to its initial value of transparent.

Initial values refer to the values of a property when they aren't explicitly set. For example, if you don't set the `background-color` of an element, it will default to `transparent`

Initial values are kind of default property values.

Let's write some JavaScript

There's very little we need to do on the JavaScript side of things.

First let's handle the slider.

We just need 5 lines for that!

```
const root = document.documentElement
const range = document.querySelector('.booth-slider')

//as slider range's value changes, do something
range.addEventListener('input', handleSlider)

function handleSlider (e) {
  let value = e.target.value
  root.style.setProperty('--slider', value)
}
```

That was easy, huh?

Let me explain just in case I lost you.

First off, keep a reference to the slider element, `const range = document.querySelector('.booth-slider')`

Set up an event listener for when the value of the range input changes, `range.addEventListener('input', handleSlider)`

Write the callback, `handleSlider`

```
function handleSlider (e) {
  let value = e.target.value
  root.style.setProperty('--slider', value)
}
```

`root.style.setProperty('--slider', value)` says, get the `root` element (HTML) , grab its style, and set a property on it.

Handling the color changes

This is just as easy as handling the slider value change.

Here's how:

```
const inputs = document.querySelectorAll('.color-box > input')

//as the value in the input changes, do something.
inputs.forEach(input => {
  input.addEventListener('input', handleInputChange)
})

function handleInputChange (e) {
  let value = e.target.value
  let inputId = e.target.parentNode.id
  let inputBg = `--bg-${inputId}`
  root.style.setProperty(inputBg, value)
}
```

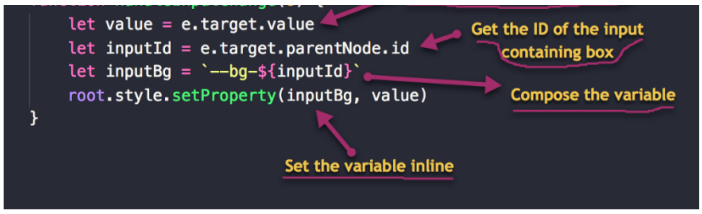
Keep a reference to all the text inputs, `const inputs = document.querySelectorAll('.color-box > input')`

Set up an event listener on all the inputs:

```
inputs.forEach(input => {
   input.addEventListener('input', handleInputChange)
})
```

Write the `handleInputChange` function:

```
function handleInputChange (e) {
  let value = e.target.value
  let inputId = e.target.parentNode.id
  let inputBg = `--bg-${inputId}`
  root.style.setProperty(inputBg, value)
}
```



Phew…

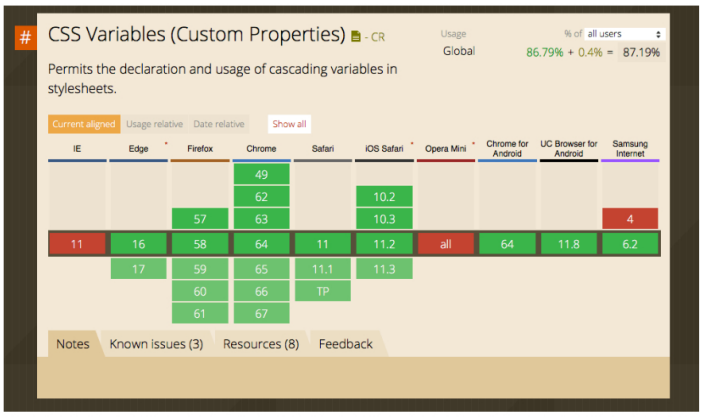
That's it!

Project's done.

How did I miss this?

I had completed and edited the initial draft of this article when I noticed I didn't mention browser support anywhere. So, let me fix my mess.

Browser support for CSS variables (aka custom properties) isn't bad at all. It's pretty good, with decent support across all modern browsers (over 87% at the time of this writing).



caniuse

So, can you use CSS variables in production today? I'll say yes! Be sure to check what the adoption rate is for yourself, though.

On the bright side, you can use a preprocessor like Myth. It'll preprocess your 'future' CSS into something you use today. How cool, huh?

If you have some experience using postCSS, that's equally a great way to use future CSS today. Here's a postCSS module for CSS variables.

That's it. I'm done here.

Oops, but I've got Questions!

Get the Ebook for offline reading, and also get a **private** slack invite where you can ask me anything.

That's a fair deal, right?

Catch you later! 💖

CSS Tech Web Development Technology Design

9.8K claps 33

Ohans Emmanuel

Author, Understanding Redux. I Love God. I Love GF a little too much 💖 🤣
http://thereduxjsbooks.com

Follow

freeCodeCamp.org

Stories worth reading about programming and technology from our open source community.

Follow

Never miss a story from **freeCodeCamp.org**, when you sign up for Medium. Learn more

GET UPDATES