Advanced CSS Custom Properties for Components

A demo deep dive on CSS variables, and how they can be leveraged to make your components more configurable and reusable. We'll explore inheritance, typing, composability, the JavaScript API, and other scary-sounding but actually not too bad concepts that reduce development effort and increase component utility. Supercharge your components with CSS custom properties!

– Session link
– Codepen demo collection

Welcome to Advanced CSS Custom Properties for Components. They say that brevity is the soul of wit, and that goes double for titles, so welcome to my witless talk. Thank you for being here, genuinely.

My name is Chris DeLuca, I'm a senior front end developer working at Lullabot. Call me weird, but I love component-based development, and I'm excited to share some of the things I've learned about implementing them using CSS custom properties.

What is this?

Let's get into what we'll be getting into this afternoon. This talk covers some patterns for using CSS custom properties, or variables, to customize components. It will be pretty demo heavy. You might see some examples that directly apply to your work, or you may not. My hope is that the demos at least get your creativity flowing to start playing with some of these patterns in your day to day.

Now, if you remember the 1998 American Godzilla movie, and why would you, there's a scene where a scientist, played by Matthew Broderick, has a plan to lure Godzilla into a trap with a massive amount of fish as bait.

The army delivers the pile of fish, and Broderick looks at a guy with a wry smile, and goes, "that's a lot of fish." There's a music cue, the whole thing. This massive, highly specific task has just been successfully carried out, at Broderick's explicit instruction, and exactly to his specifications, and yet…*he's* surprised.

I refuse to be similarly surprised.

Given the amount of live demos happening, something will inevitably go wrong. When that wrong thing does happen, and it will, to get past that awkward moment, I invite everyone here to yell, "That's a lot of fish!". Can we try it now? Beautiful.

The examples in this talk will use Twig and Single Directory Components, although the patterns we'll be talking about apply across component technologies.

Quick aside, if you aren't using Single Directory Components in your Drupal projects, I highly recommend you do. They're good.

What's a component?

Before we get to the patterns, I'd like to take a little time to set the stage. We're starting basic. Perhaps too basic, but I always like to define our terms. I think it gives a solid base to build off of.

So what is a component?

There are many valid definitions, but for our purposes, we're saying a component is a chunk of code that:

1. Is repeatable; it can be called multiple times from anywhere in the system.
2. Is encapsulated; it always contains any code it needs—HTML, CSS, JS.
3. Takes parameters; output can be manipulated via pre-defined inputs.

That third one is really what this talk is all about. Although, if we're talking strictly, taking parameters isn't really necessary for a component to be a component. However, most components do take parameters, and that's where the complexity lies, so let's round up on this one.

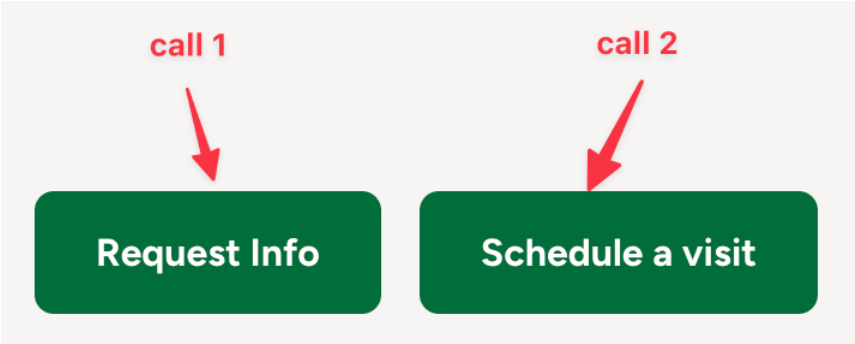
A component is…

– Repeatable
– Encapsulated
– Parameterized

Let's dive into an example of how you might normally configure a component.

We'll start with our good friend, the button. Boring, perhaps, but that's rude to say about any friend, much less a good one. Okay, what do we want to pay attention to here?

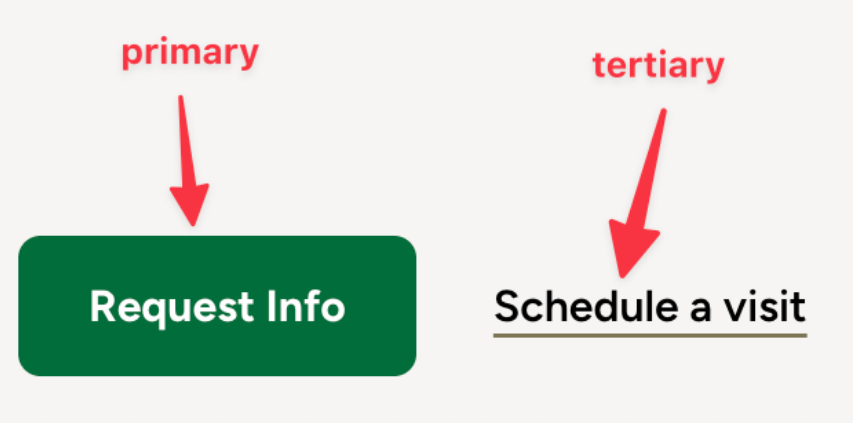
The button's display stays the same, while the data changes depending on what parameters it is passed. Each time it's called, the button will have a different URL and text. Straightforward.



To implement this, we have `text` and `url` parameters, which is how we're passing the data to our button.

```
{{ include('my-theme:button', {
    text: "Hello, world!",
    url: "https://example.com",
}) }}
```

Our button component can change its display as well, depending on the parameters we pass.



We can see that the button component accepts a `variation` parameter, and we can pass "primary", "secondary", or "tertiary" to it. Sometimes `variation` will be named something else, like `type`, or `style`, but the meaning is the same.

```
{{ include('my-theme:button', {
    variation: "primary",
    text: "Hello, world!",
    url: "https://example.com",
}) }}
```

Our variation will set an HTML property, usually a `class`, that so we can style the variations differently.

```
<a class="my-button my-button--{{ variation }}" href="{{ url }}">{{ text }}
</a>
```

```
.my-button {
  /* button code… */
  border: 2px solid rgb(231 0 119);
  background-color: rgb(231 0 119);
}

.my-button--secondary {
  background-color: transparent;
  color: black;
}
```

Demo

This is good!

I'm not advocating for the fall of component variations. They're great! Using CSS Custom Properties doesn't supplant traditional variation-based customization. It works alongside variations. We'll talk about how they work together in just a little bit. For now, what are these mysterious CSS Custom Properties I keep speaking of?

What are CSS Custom Properties?

They're variables in CSS. They follow the cascade, and they're computed at runtime, and they can be redefined. These are three important qualities that we'll leverage to enhance our components.

```
.my-selector {
  color: var(--primary-color);
}
```

A common pattern is to set a lot of custom properties on the `:root` element. Values like font sizes, and colors, and spacing sizes, and all types of little chunks of design information that can be used throughout our system.

```
:root {
  --primary-color: rebeccapurple;
  --font-sans-serif: Helvetica, Arial, sans-serif;
  --spacing-vertical-rhythm: 1.5rem;
}
```

This helps us build consistency throughout our website.

```
:root {
  --primary-color: rgb(231 0 119);
}

.my-button {
  /* button code… */
  border: 2px solid var(--primary-color);
  background-color: var(--primary-color);
}
```

Demo

Custom Properties set on the `:root` are available to use everywhere, but are centrally managed. The `:root` pseudo-selector is the highest parent of the page, so setting properties there guarantees that those custom properties can be accessed anywhere else on the page. This saves us some work if the design needs to be tweaked. Now, we only have to change the value in one place, rather than in every place that value is used.

```
.my-blockquote {
  border-inline-start: 4px solid var(--primary-color);
  padding-inline-start: 2ch;
  font-family: var(--font-sans-serif);
}

.my-blockquote > * {
  margin-block: var(--spacing-vertical-rhythm) 0;
}
```

Demo

This is also good!

Setting Custom Properties on the `:root` to make them globally available is incredibly useful, for all the reasons I just mentioned. This pattern is great!

However, that's where we often stop. As good as this pattern is, it kind of bothers me if this is all we end up using Custom Properties for.

Custom Properties are Constants?

We're essentially treating custom properties as constants. A constant is similar to a variable, but its value never changes during the runtime of the program. But Custom Properties have so much more to offer. They're more dynamic, more flexible, more, dare I say, compelling. Custom Properties are…variables.

```
const myConst = "Hello";
myConst = "Goodbye"; // illegal!
```

One of the important properties of Custom Properties I mentioned earlier was that their value can be re-defined. Sure, but how is that useful? In our example here, `--primary-color` will be `red`, but…who cares?

```
:root {
   --primary-color: green;
   --primary-color: red; /* legal! */
}
```

Let's go back to our button example.

Demo

We're using Custom Properties as actual variables, overriding them inside their selector. We're still leveraging traditional variations and global Custom Properties set on the `:root`. In fact, not only are all these approaches living in harmony with each other, we're integrating them together.

It also offers a clear API for extensions in the future, showing which parts are meant to be changed, and which parts should be more stable.

Okay, that works, but it's not too much different than writing CSS without re-defined custom properties.

Let's say you have a landing page type, where editors can configure which components appear in a list, and in which order, maybe using Layout Builder, or Paragraphs, or Drupal Canvas.

Editors place components into sections, which can have either a light or dark background. The components inside need to be styled to accommodate both color schemes.

Custom Properties offer a nice way to solve this.

Remember that Custom Properties follow the cascade. So we can re-define the property on a component parent, and have the value flow down.

https://codepen.io/bronzehedwick/pen/VYjMbwV?editors=1100

The code is simple and clear. If we need to support a new background color, all we need to do is re-define the custom properties in a new selector.

Custom Properties vs Variations

I think of these as two separate, quasi-related concepts. Both are used to customize components. Custom properties target a single property, whereas a variation is a collection of properties, which can themselves be a custom property. Both are also nice API surfaces for other developers to leverage.

I reach for a variation when there's a larger function the component needs to achieve. I reach for Custom Properties when I need granular control. Both of these, as we've seen, can and do work well together.

Complexity is King

I think this pattern really shines when the system gets more complicated. And don't they all?

Imagine you have a component that needs `3rem` padding on either side.

No problem. You're done before you wake up.

```
.my-component {
    padding-inline: 3rem;
}
```

Now, imagine that we need `3rem` padding on the left and right for every component, and that the value should be consistent across all components. And there's fifty of them.

A bit of legwork, but no problem. Add a custom property to the `:root`, and apply it to every component.

```
:root {
    --component-spacing: 3rem;
}

.my-component {
    padding-inline: var(--component-spacing);
}

.my-other-component {
    padding-inline: var(--component-spacing);
}

/* etc */
```

Now, imagine that this padding treatment is only for landing pages. On text heavy pages, the components have less room, so they need to have only `2rem` of padding.

No problem. You want to stay consistent, so you add a variation, and override the component. You'll need to do that for all fifty components, and that feels heavy for such a small change, but it works.

```
:root {
  --component-spacing: 3rem;
  --component-spacing-small: 2rem;
}

.my-component--small-padding {
  padding-inline: var(--component-spacing-small);
}

.my-component--small-padding {
  padding-inline: var(--component-spacing-small);
}

/* etc */
```

Now, imagine that all these components can also appear in a sidebar. That's even more space constrained, so components there can only have `1rem` of padding.

No…problem? That's another variation for all fifty components, for the same little sized change. But, screw it, throw it on the pile. Next!

```
:root {
  --component-spacing: 3rem;
  --component-spacing-small: 2rem;
  --component-spacing-extra-small: 1rem;
}

.my-component--extra-small-padding {
  padding-inline: var(--component-spacing-extra-small);
}

.my-component--extra-small-padding {
  padding-inline: var(--component-spacing-extra-small);
}

/* etc */
```

Now, imagine that for some of these spots, the template isn't always aware of its context, meaning we can't pass in the right variation for every place a component will render. Unrealistic, I know.

Okay, not great, but you know what?

No problem. You target a unique selector that wraps the components that can't get the right variation, and add it to the variation styles. It's not pretty, but it works.

```
.my-place-that-cant-get-a-variation .my-component,
.my-component--my-variation {

/* etc */
```

You do have to do that for every one of the fifty components, however. Your code is starting to get bloated and messy. The chance for styles to end up where they shouldn't and other display bugs just shot up.

Now, imagine that this variation dead-zone affects every component placement type we've coded: landing pages, text pages, and sidebars. That means you'll need to manage this extra code for every variation. That bit of extra boilerplate code is ballooning across all fifty components.

Now, imagine that this site will continue to change. More components, more places to put them, more variations. More more more.

Now…imagine all that is real.

No. Problem.

That's a bad time.

Or, you could set a few custom properties on every component once, define a default on the `:root`, then re-define them on a parent for each placement type, bypassing the whole issue.

```
.my-component {
    padding-inline: var(--component-spacing);
}

:root {
    --component-spacing: 3rem;
}

.my-text-page {
    --component-spacing: 2rem;
}

.my-sidebar {
    --component-spacing: 1rem;
}
```

Demo

Because Custom Properties are computed at runtime, they can be included in a `calc()` function like any other value. Or `tan()` or `sin()` functions, but let's stay away from trigonometry and stick to the friendly `calc()`. Computing custom properties can be really useful when there's a set of related values that share a common base.

```
.my-component {
  padding-block: var(--spacing-base);
  padding-inline: calc(var(--spacing-base) / 2));
}
```

For example, think of a table component that has multiple settings for how much whitespace is contained in each cell.

Demo

One more thing

Custom Properties provide, I think, a compelling API surface for others to customize your components in flexible ways that you still define. All flexibility comes with a cost. We know this. Misuse of the system can and will occur given enough time and attention. It won't solve your life, but one way Custom Properties can help is by type checking.

Yes, it's true. We can have more control over a custom property by defining it with `@property` method.

This is more verbose than the straightforward assignments we've been doing, but it has advantages.

First, it is not set on a scope, like `:root` or another selector. You're just registering the property, and it can be redefined anywhere.

Second, you can tell the property which types it will accept via `syntax`. We're showing a single type here, but multiple can be defined. Any value assigned to the property that are of a different type will be discarded, and an error will be shown in the browser dev tools. This is not possible when you define a Custom Property otherwise. Those properties will accept any value, even if it is invalid for the use case. This is especially useful for creating a pit of success for other developers, and gets more useful the larger the system is and the larger the user base is. Think: a design system.

There's two other `@property` aspects here: `inherits` and `initial-value`. Inherits is `true` by default, and controls whether children will get this Custom Property when it's set on a parent. I've never personally found a real use case for turning off inheritance, but if you know of one, please yell at me after the session!

Finally, there's initial value. It does what it says on the tin. Whatever value you set here, will be the value of the Custom Property, unless otherwise redefined.

```
@property --rotation {
  syntax: '<angle>';
  inherits: true;
  initial-value: 0deg;
}
```

Demo

What happened?

Custom Properties…

– Are CSS variables
– Are not a replacement for component variations
– Work in harmony with variations
– Can be used globally to create consistency
– Can be re-defined to save effort
– Shine at scale
– Can be used in math
– Can be type checked

So long, and thanks for all the fish!

Chris DeLuca

https://www.chrisdeluca.me

AppState:

- https://families.appstate.edu
- https://families.appstate.edu/supporting-your-student