It's fashionable to dislike CSS. There are lots of reasons why that's the case, but it boils down to this: CSS is *unpredictable*. If you've never had the experience of tweaking a style rule and accidentally breaking some layout that you thought was completely unrelated — usually when you're trying to ship — then you're either new at this or you're a much better programmer than the rest of us.

So the JavaScript community rolled up its sleeves and got to work. Over the last couple of years, there's been a Cambrian explosion of libraries aimed at making CSS behave, collectively referred to as *CSS-in-JS*.

What you might not realise is that **the biggest problems with CSS can be solved without CSS-in-JS**. Without those problems, writing CSS isn't just tolerable — it's enjoyable. And you don't have to find solutions to the additional problems that CSS-in-JS introduces.

This article isn't in any way intended as criticism of the hard work the CSS-in-JS community has done. It's one of the most active corners of the JS ecosystem, and new ideas are springing up every week. Instead, my purpose is to illustrate why an alternative approach — based on Single File Components with real CSS — is so damn delightful.

The biggest problem with CSS

Everything in CSS is global. Because of that, styles intended for one bit of markup often end up affecting another. Because of *that*, developers often resort to wild namespacing conventions (not 'rules', since they're very difficult to enforce) that mostly just increase your risk of RSI.

It gets worse when you're working on a team. No-one dares touch styles authored by someone else, because it's often unclear what they're doing, what markup they apply to, and what disasters will unfold if you remove them.

The consequence of all this is the **append-only stylesheet**. There's no way of knowing which code can safely be removed, so it's common to undo some existing style with another, more specific style — even on relatively small projects.

Single File Components change all that

The idea behind SFCs is simple: you write your components in an HTML file that (optionally) contains a `<style>` and `<script>` attribute describing the component's styles and behaviour. Svelte, Ractive, Vue and Polymer all follow this basic pattern.

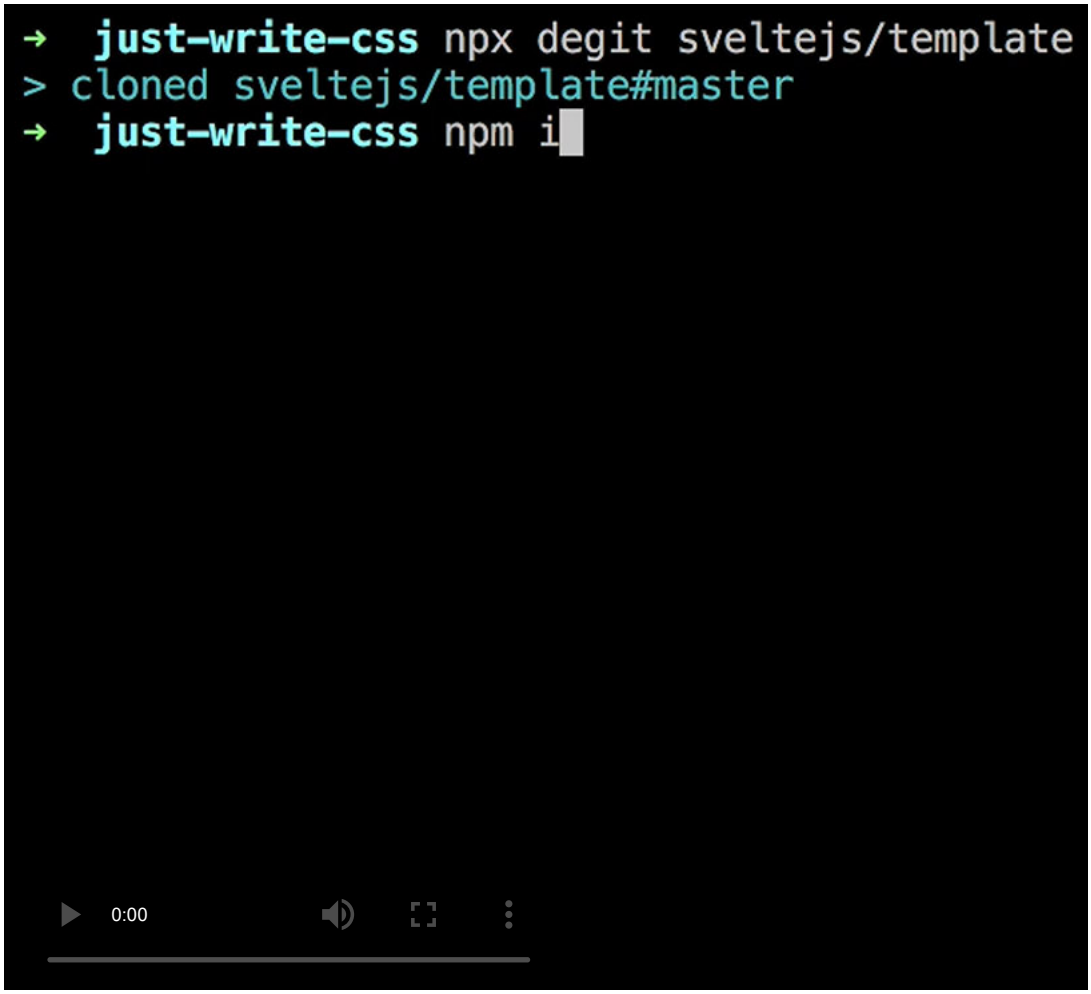
Read the introductory blog post if you're new to Svelte. Or read the testimonials.

(For the rest of this article we'll be using Svelte, obviously. But if the idea of using a template language makes you shudder — your fears are misplaced, but that's a topic for another day — then just use Vue which lets you use JSX in your SFCs.)

Several wonderful things happen as a result:

- Your styles are *scoped to the component*. No more leakage, no more unpredictable cascade. And no more sesquipedalian classnames designed to prevent conflicts.
- You don't need to go spelunking through your folder structure to find the rules that are breaking your stuff.
- The compiler (in Svelte's case) can **identify and remove unused styles**. No more append-only stylesheets!

Let's see what that looks like in practice.




```
<figcaption>
    Is this what they mean by 'use the platform'?
</figcaption>
```

Every code editor already knows about CSS, so there's a good chance that you'll get autocomplete, linting, syntax highlighting and so on — all without additional JS-fatigue-inducing tools.

And because it's real CSS, rather than some camelCased quotes-everywhere impostor, we can take advantage of the 'tweak in devtools, paste back into our source code' workflow, which I personally couldn't live without. Notice that we get CSS sourcemaps out of the box, so you can instantly pinpoint the lines in question. It's hard to overstate the importance of this: when you're in WYSIWYG mode, you're not thinking in terms of your component tree, so having a robust way to figure out *where these damn styles came from* is essential. Doubly so if someone else originally wrote the component. (I promise you, this is the single biggest productivity boost to your CSS workflow. If you're writing styles without sourcemaps, you are almost certainly wasting a lot of time. I know I was.)

Svelte transforms your selectors (using an attribute that's also applied to affected elements, though the exact mechanism is unimportant and subject to change) to achieve the scoping. It warns on and removes any unused rules, then it minifies the result and lets you write it out to a `.css` file. There's also an experimental new option to compile to web components, using shadow DOM to encapsulate the styles, if that's your jam.

This is all possible because your CSS is parsed (with css-tree) and statically analysed in the context of your markup. Static analysis opens the doors to all kinds of exciting future possibilities — smarter optimisations, a11y hints — that are much harder if your styles are computed dynamically at runtime. We're just getting started.

But we can add tools to do [x]!

If your reaction to the video was 'fine, but if we use TypeScript and write plugins for each editor then we can get all the autocomplete and syntax highlighting stuff' — in other words, if you believe that in order to achieve parity with CSS it makes sense to build, document, promote and maintain a fleet of ancillary projects — then, well, you and I may never see eye to eye!

We don't have all the answers — yet

Having said all that, CSS-in-JS does point to answers to some lingering questions:

- How can we install styles from npm?
- How can we reuse constants that are defined in a single place?
- How can we compose declarations?

Personally, I haven't found these issues to outweigh the benefits of the approach outlined above. You may well have a different set of priorities, and they may be reason enough for you to abandon CSS.

But at the end of the day, you have to know CSS anyway. Love it or loathe it, you must at least *learn* it. As custodians of the web, we have a choice: create abstractions that steepen the web dev learning curve yet further, or work together to fix the bad parts of CSS. I know which I choose.