There is still no JavaScript on this website.

I'll keep it that way as long as I can. Why do I care so much?

Web developers rarely get the chance to make a simple website.

Creating even the most basic website in ${currentYear} comes with pretty high expectations. It's supposed to look professional, to use a design system for constent spacing and typography, to have a bunch of metadata for SEO and accessibility, to be fast, and to be constantly available. Web development has professionalized, and those are good professional standards to have! Yet it does set the barrier to entry kind of high, and moreover, it makes "hobbyist" web development all but impossible. Which is a shame; the open Web was supposed to be the place where everyone could have a page.

What we do instead

I mentioned before that a frustrated software developer who shares my Web app fatigue will sometimes blame the platform, the cyclical hype for new frameworks, the whippersnappers reinventing old patterns or the giant companies trying to make their bloated, special-purpose frameworks frameworks into unofficial standards. All of those things do contribute, but they're all consequences of one thing: all us developers of Web frameworks and tools are trying to flatten out the contours of the Web platform.

We see some of these contours as "bugs" or missing features: HTML has no logic structures, CSS has no modularity and strange, opaque specificity, JavaScript has no standard library, you name it. There are plenty of flaws and oversights in the design of HTML, CSS and JavaScript, but some limitations are purposeful. They are "affordances", like the shaped grip of a tool handle that is only comfortable to hold in the correct, safe position. They are hints that if you need to do something that HTML, CSS or JavaScript make difficult, then you might be trying to make a Web page into something that isn't a document.

- HTML doesn't have includes or logic tags because HTML is a document format, not a language. HTML is meant to be either written manually or generated by code. It shouldn't generate itself.
- CSS selectors are hierarchical because documents are hierarchical; if you eschew everything but class selectors, your CSS is probably targeting a more complex and decentralized view than a simple document tree.
- JavaScript lacks a "standard library" because such a library isn't necessary for most Web pages. A module full of sorting algorithm strategies (for instance) would see little use on Web pages, so why should my phone have to download, parse and execute it?

What to not do

Don't bundle so much!

Some of my most satisfying work has been on bundlers ad build systems. I love making tools for other tool-users, and I enjoy the sense that we've made the Web easier to learn by automating a lot of the worst parts of putting a website together.

Bundlers have their place, but most websites don't need them. To wit:

- ES6 Modules are supported now, in all major browsers. The browser can recognize both static and dynamic imports, and fetch imported scripts on demand. This might feel like a potential performance hit--that's hundreds of modules and hundreds of requests!--but there's another case where if the native platform feature isn't suitable, then it may be a sign that your code isn't suited. Should a webpage need hundreds of node_modules and module imports, it might be trying to do the wrong thing.
- Modern JavaScript syntax features, like optional?.chaining", can replace a lot of boilerplate.
- CSS3 and its big bunch of fancy selectors work everywhere now. CSS3 functions, variables, and arithmetic are now always available, not to mention shadow DOM. Preprocessors like SASS and PostCSS are still useful as organizing tools, but their original purpose was to provide primitives like variables and mixins. They are largely obsolete.
- Much of the defensive user-agent-sniffing code and feature detection code (like Modernizr and browserslist)running on websites is just no longer necessary.

Don't reject the native features!

There's a lot of understandable tension about who really directs the roadmap of the "supposedly open" Web. Browser makers, Google first and foremost, are de facto controllers of the specification. That's bad and it should change, but it won't be software engineers who change it; that's a political and economic problem. In the meantime, big evil corporations may pressure the design of new Web standards, but at least the standards development does happen out in the open. That's part of the Web platform, and it's a reason that it is very safe to try newer features as they land in browsers. Much of the "missing functionality" that we've often filled with big JavaScript piles has landed, in the native Web platform itself. But there is resistance to migrating to those new features.

- **Aesthetic:** Now that websites are "digital experiences", they have to compete with each other on vibe. Usually this means the designers and product managers want pixel-level control over the visual display of everything. The Web isn't designed for that, in the first place, but it's also perverse because it discourages improvement of the underlying Web itself that we wrote so much JavaScript to escape.HTML has number, date, and color selectors now. They're styleable, even. They fire events. But nobody bothers with them, most of the time; it's hard to get them to look exactly like the proposed design. Well, maybe it should be! Maybe there should be fewer different visual ways out there to choose a date or a color. I think people might appreciate that.
- **Trust and control:** Google Chrome landed CSS Nesting in version 112. This means that one of the basic value propositions of preprocessors like SASS is now part of the native platform. Why not use it? Well, it might not quite work the way we expect it to. Chrome may withdraw the proposal anyway, and then you'll have to refactor it back. When you're developing a general purpose application, these risks seem pointless to take; you can just do it all custom, and then you're not beholden to the WCAG's whims. But if you are making a Web document, it suddenly makes a lot more sense to adopt new stuff. In the worst case, the document "gracefully degrades".

Don't iterate too much on developer experience.

It's a game of diminishing returns. I know this the hard way. Developer experience has been my chief passion for the last few years; I believed that learnable, capable tools could further democratize the web and open it up as a creation platform to people beyond the typically understood "programmer" skillset. I still believe that, but I don't think we're close.

One of the key goals of developer experience work is to minimize feedback loops. If the software you're making can change its behavior right in front of you, as you change the code, then you can more easily enter a flow state. You can use your whole brain. You can learn how the system works intuitively, and then later, where necessary, you'll find it easier to understand how it works formally.

Software development has historically had a problem with this. Some of the old wait times are now mitigated by faster computers, and some have been ameliorated by complex incremental caching. The latter is pretty hard to understand, let alone maintain.

But we want to have it! Saving your work, uploading your changes, refreshing the webpage, scrolling to where you were, entering the same data into the form, over and over after every change is not just mind-numbing and tedious. It interrupts flow state. It makes it harder for you to learn the details of, say, a CSS declaration and all its possible values. So it passed that we invented live reloading and hot module replacement. And incremental builds. And component-based refresh. These are all wonderful things, when they work.

But they too often don't. They are brittle structures! In order for hot module replacement on a React component to work, the system must first parse and analyze the component file to identify what exported value is the component itself. Then it has to register that component in its own tracking cache and keep track of which emitted bundle contains that definition. Then it has to produce a specially-formatted delta object representing what changes need to happen to that component's source code alone, and then, how to safely reinitialize the module, propagate changes out to its dependents, and "reseal" the bundle for the next change. And then it has to tell the frontend, which it already instrumented with a websocket and bundle-integrated code for listening to it.

All of that allows you to make some changes to a React component and watch it update all by itself, with nothing else changing in the open app. Without all of that, changing a React component live would...refresh the page. Even compounded for every much time has this saved? How much complexity and computing power (and framework maintainer time) go into each of these saved moments, amortized over the years?

Probably just enough to make it worth it...until it breaks.

Coming up soon, part three: ***"What DO we do? Instead?"***

Return to part one*.*