> *Wait, this new framework has a runtime? Ugh. Thanks, I'll pass.* ***– front end developers in 2018***

We're shipping too much code to our users. Like a lot of front end developers, I've been in denial about that fact, thinking that it was fine to serve 100kb of JavaScript on page load – just use one less .jpg! – and that what *really* mattered was performance once your app was already interactive.

But I was wrong. 100kb of .js isn't equivalent to 100kb of .jpg. It's not just the network time that'll kill your app's startup performance, but the time spent parsing and evaluating your script, during which time the browser becomes completely unresponsive. On mobile, those milliseconds rack up very quickly.

If you're not convinced that this is a problem, follow Alex Russell on Twitter. Alex hasn't been making many friends in the framework community lately, but he's not wrong. But the proposed alternative to using frameworks like Angular, React and Ember – Polymer – hasn't yet gained traction in the front end world, and it's certainly not for a lack of marketing.

Perhaps we need to rethink the whole thing.

What problem do frameworks *really* solve?

The common view is that frameworks make it easier to manage the complexity of your code: the framework abstracts away all the fussy implementation details with techniques like virtual DOM diffing. But that's not really true. At best, frameworks *move the complexity around*, away from code that you had to write and into code you didn't.

Instead, the reason that ideas like React are so wildly and deservedly successful is that they make it easier to manage the complexity of your *concepts*. Frameworks are primarily a tool for structuring your thoughts, not your code.

Given that, what if the framework *didn't actually run in the browser*? What if, instead, it converted your application into pure vanilla JavaScript, just like Babel converts ES2016+ to ES5? You'd pay no upfront cost of shipping a hefty runtime, and your app would get seriously fast, because there'd be no layers of abstraction between your app and the browser.

Introducing Svelte

Svelte is a new framework that does exactly that. You write your components using HTML, CSS and JavaScript (plus a few extra bits you can learn in under 5 minutes), and during your build process Svelte compiles them into tiny standalone JavaScript modules. By statically analysing the component template, we can make sure that the browser does as little work as possible.

The Svelte implementation of TodoMVC weighs 3.6kb zipped. For comparison, React plus ReactDOM *without any app code* weighs about 45kb zipped. It takes about 10x as long for the browser just to evaluate React as it does for Svelte to be up and running with an interactive TodoMVC.

And once your app *is* up and running, according to js-framework-benchmark **Svelte is fast as heck**. It's faster than React. It's faster than Vue. It's faster than Angular, or Ember, or Ractive, or Preact, or Riot, or Mithril. It's competitive with Inferno, which is probably the fastest UI framework in the world, for now, because Dominic Gannaway is a wizard. (Svelte is slower at removing elements. We're working on it.)

It's basically as fast as vanilla JS, which makes sense because it *is* vanilla JS – just vanilla JS that you didn't have to write.

But that's not the important thing

Well, it *is* important – performance matters a great deal. What's really exciting about this approach, though, is that we can finally solve some of the thorniest problems in web development.

Consider interoperability. Want to `npm install cool-calendar-widget` and use it in your app? Previously, you could only do that if you were already using (a correct version of) the framework that the widget was designed for – if `cool-calendar-widget` was built in React and you're using Angular then, well, hard cheese. But if the widget author used Svelte, apps that use it can be built using whatever technology you like. (On the TODO list: a way to convert Svelte components into web components.)

Or code splitting. It's a great idea (only load the code the user needs for the initial view, then get the rest later), but there's a problem – even if you only initially serve one React component instead of 100, *you still have to serve React itself*. With Svelte, code splitting can be much more effective, because the framework is embedded in the component, and the component is tiny.

Finally, something I've wrestled with a great deal as an open source maintainer: your users always want *their* features prioritised, and underestimate the cost of those features to people who don't need them. A framework author must always balance the long-term health of the project with the desire to meet their users' needs. That's incredibly difficult, because it's hard to anticipate – much less articulate – the consequences of incremental bloat, and it takes serious soft skills to tell people (who may have been enthusiastically evangelising your tool up to that point) that their feature isn't important enough. But with an approach like Svelte's, many features can be added with absolutely no cost to people who don't use them, because the code that implements those features just doesn't get generated by the compiler if it's unnecessary.

We're just getting started

Svelte is very new. There's a lot of work still left to do – creating build tool integrations, adding a server-side renderer, hot reloading, transitions, more documentation and examples, starter kits, and so on.

But you can already build rich components with it, which is why we've gone straight to a stable 1.0.0 release. Read the guide, try it out in the REPL, and head over to GitHub to help kickstart the next era of front end development.