> *Quickstart for the impatient: the Sapper docs, and the starter template*

If you had to list the characteristics of the perfect Node.js web application framework, you'd probably come up with something like this:

1. It should do server-side rendering, for fast initial loads and no caveats around SEO
2. As a corollary, your app's codebase should be universal — write once for server *and* client
3. The client-side app should *hydrate* the server-rendered HTML, attaching event listeners (and so on) to existing elements rather than re-rendering them
4. Navigating to subsequent pages should be instantaneous
5. Offline, and other Progressive Web App characteristics, must be supported out of the box
6. Only the JavaScript and CSS required for the first page should load initially. That means the framework should do automatic code-splitting at the route level, and support dynamic `import(...)` for more granular manual control
7. No compromise on performance
8. First-rate developer experience, with hot module reloading and all the trimmings
9. The resulting codebase should be easy to grok and maintain
10. It should be possible to understand and customise every aspect of the system — no webpack configs locked up in the framework, and as little hidden 'plumbing' as possible
11. Learning the entire framework in under an hour should be easy, and not just for experienced developers

Next.js is close to this ideal. If you haven't encountered it yet, I strongly recommend going through the tutorials at learnnextjs.com. Next introduced a brilliant idea: all the pages of your app are files in a `your-project/pages` directory, and each of those files is just a React component.

Everything else flows from that breakthrough design decision. Finding the code responsible for a given page is easy, because you can just look at the filesystem rather than playing 'guess the component name'. Project structure bikeshedding is a thing of the past. And the combination of SSR (server-side rendering) and code-splitting — something the React Router team gave up on, declaring 'Godspeed those who attempt the server-rendered, code-split apps' — is trivial.

But it's not perfect. As churlish as it might be to list the flaws in something *so, so good*, there are some:

- Next uses something called 'route masking' to create nice URLs (e.g. `/blog/hello-world` instead of `/post?slug=hello-world`). This undermines the guarantee about directory structure corresponding to app structure, and forces you to maintain configuration that translates between the two forms
- All your routes are assumed to be universal 'pages'. But it's very common to need routes that only render on the server, such as a 301 redirect or an API endpoint that serves the data for your pages, and Next doesn't have a great solution for this. You can add logic to your `server.js` file to handle these cases, but it feels at odds with the declarative approach taken for pages
- To use the client-side router, links can't be standard `<a>` tags. Instead, you have to use framework-specific `<Link>` components, which is impossible in the markdown content for a blog post such as this one, for example

The real problem, though, is that all that goodness comes for a price. The simplest possible Next app — a single 'hello world' page that renders some static text — involves 66kb of gzipped JavaScript. Unzipped, it's 204kb, which is a non-trivial amount of code for a mobile device to parse at a time when performance is a critical factor determining whether or not your users will stick around. And that's the *baseline*.

We can do better!

The compiler-as-framework paradigm shift

Svelte introduced a radical idea: what if your UI framework wasn't a framework at all, but a compiler that turned your components into standalone JavaScript modules? Instead of using a library like React or Vue, which knows nothing about your app and must therefore be a one-size-fits-all solution, we can ship highly-optimised vanilla JavaScript. Just the code your app needs, and without the memory and performance overhead of solutions based on a virtual DOM.

The JavaScript world is moving towards this model. Stencil, a Svelte-inspired framework from the Ionic team, compiles to web components. Glimmer *doesn't* compile to standalone JavaScript (the pros and cons of which deserve a separate blog post), but the team is doing some fascinating research around compiling templates to bytecode. (React is getting in on the action, though their current research focuses on optimising your JSX app code, which is arguably more similar to the ahead-of-time optimisations that Angular, Ractive and Vue have been doing for a few years.)

What happens if we use the new model as a starting point?

Introducing Sapper

The name comes from the term for combat engineers, and is also short for Svelte app maker

Sapper is the answer to that question. **Sapper is a Next.js-style framework that aims to meet the eleven criteria at the top of this article while dramatically reducing the amount of code that gets sent to the browser.** It's implemented as Express-compatible middleware, meaning it's easy to understand and customise.

The same 'hello world' app that took 204kb with React and Next weighs just 7kb with Sapper. That number is likely to fall further in the future as we explore the space of optimisation possibilities, such as not shipping any JavaScript *at all* for pages that aren't interactive, beyond the tiny Sapper runtime that handles client-side routing.

What about a more 'real world' example? Conveniently, the RealWorld project, which challenges frameworks to develop an implementation of a Medium clone, gives us a way to find out. The Sapper implementation takes 39.6kb (11.8kb zipped) to render an interactive homepage.

Code-splitting isn't free — if the reference implementation used code-splitting, it would be larger still

The entire app costs 132.7kb (39.9kb zipped), which is significantly smaller than the reference React/Redux implementation at 327kb (85.7kb), but even if it was as large it would *feel* faster because of code-splitting. And that's a crucial point. We're told we need to code-split our apps, but if your app uses a traditional framework like React or Vue then there's a hard lower bound on the size of your initial code-split chunk — the framework itself, which is likely to be a significant portion of your total app size. With the Svelte approach, that's no longer the case.

But size is only part of the story. Svelte apps are also extremely performant and memory-efficient, and the framework includes powerful features that you would sacrifice if you chose a 'minimal' or 'simple' UI library.

Trade-offs

The biggest drawback for many developers evaluating Sapper would be 'but I like React, and I already know how to use it', which is fair.

If you're in that camp, I'd invite you to at least try alternative frameworks. You might be pleasantly surprised! The Sapper RealWorld implementation totals 1,201 lines of source code, compared to 2,377 for the reference implementation, because you're able to express concepts very concisely using Svelte's template syntax (which takes all of five minutes to master). You get scoped CSS, with unused style removal and minification built-in, and you can use preprocessors like LESS if you want. You no longer need to use Babel. SSR is ridiculously fast, because it's just string concatenation. And we recently introduced svelte/store, a tiny global store that synchronises state across your component hierarchy with zero boilerplate. The worst that can happen is that you'll end up feeling vindicated!

But there are trade-offs nonetheless. Some people have a pathological aversion to any form of 'template language', and maybe that applies to you. JSX proponents will clobber you with the 'it's just JavaScript' mantra, and therein lies React's greatest strength, which is that it is infinitely flexible. That flexibility comes with its own set of trade-offs, but we're not here to discuss those.

And then there's *ecosystem*. The universe around React in particular — the devtools, editor integrations, ancillary libraries, tutorials, StackOverflow answers, hell, even job opportunities — is unrivalled. While it's true that citing 'ecosystem' as the main reason to choose a tool is a sign that you're stuck on a local maximum, apt to be marooned by the rising waters of progress, it's still a major point in favour of incumbents.

Roadmap

We're not at version 1.0.0 yet, and a few things may change before we get there. Once we do (soon!), there are a lot of exciting possibilities.

I believe the next frontier of web performance is 'whole-app optimisation'. Currently, Svelte's compiler operates at the component level, but a compiler that understood the boundaries *between* those components could generate even more efficient code. The React team's Prepack research is predicated on a similar idea, and the Glimmer team is doing some interesting work in this space. Svelte and Sapper are well positioned to take advantage of these ideas.

Speaking of Glimmer, the idea of compiling components to bytecode is one that we'll probably steal in 2018. A framework like Sapper could conceivably determine which compilation mode to use based on the characteristics of your app. It could even serve JavaScript for the initial route for the fastest possible startup time, then lazily serve a bytecode interpreter for subsequent routes, resulting in the optimal combination of startup size and total app size.

Mostly, though, we want the direction of Sapper to be determined by its users. If you're the kind of developer who enjoys life on the bleeding edge and would like to help shape the future of how we build web apps, please join us on GitHub and Discord.