Pika

A Future Without Webpack

Snowpack installs npm packages that run natively in the browser. Do you still need a bundler?

> Update (03-29-2020): This article was originally written about a year ago about a new project called `@pika/web`. Early this year that project was superseded by Snowpack, a web application builder for less tooling and 10x faster iteration. This article has been updated all references from `@pika/web` to Snowpack.

The year is 1941. Your name is Richard Hubbell. You work at an experimental New York television studio owned by CBS. You are about to give one of the

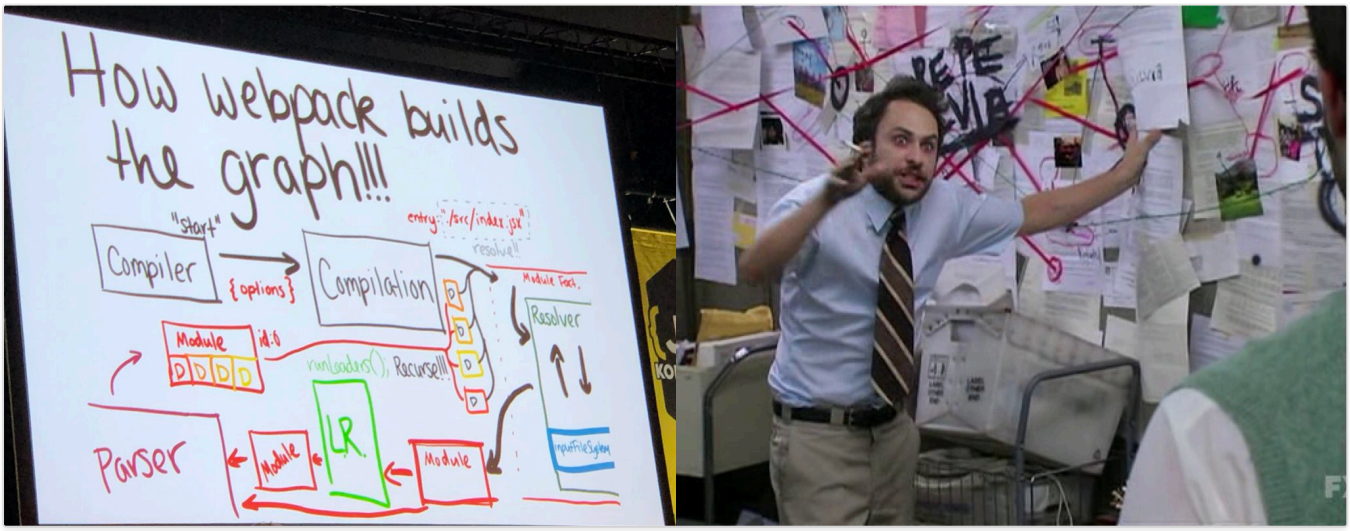
world's first major TV news broadcasts, and you have 15 minutes to fill. What do you do?

In a world that has only known radio, you stick to what you know. That is, you read the news. "Most of the [televised] newscasts featured Hubbell reading a script with only occasional cutaways to a map or still photograph." [1] It would be a while before anyone would show actual video clips on the TV news.

As a JavaScript developer in 2019, I can relate. We have this new JavaScript module system (ESM) that runs natively on the web. Yet we continue to use bundlers for every single thing that we build. Why?

Over the last several years, JavaScript bundling has morphed from a production-only optimization into a required build step for most web applications. Whether you love this or hate it, it's hard to deny that bundlers have added a ton of new complexity to web development -- a field of development that has always taken pride in its view-source, easy-to-get-started ethos.

Snowpack is an attempt to free web development from the bundler requirement. In 2019, you should use a bundler because you want to, not because you need to.



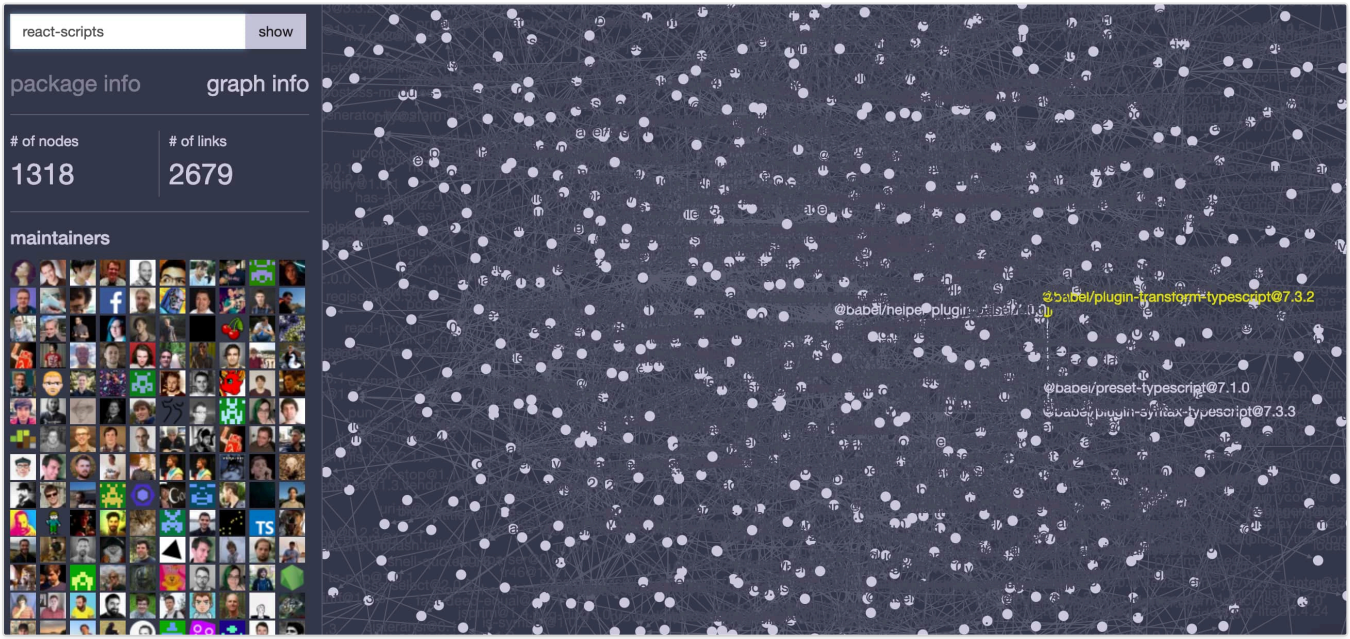
Credit: @stylishandy

Why We Bundle

JavaScript bundling is a modern take on an old concept. Back in the day (lol ~6 years ago) it was common to minify and concatenate JavaScript files together in production. This would speed up your site and get around HTTP/1.1's 2+ parallel request bottleneck.

How did this nice-to-have optimization become an absolute dev requirement? Well, that's the craziest part: Most web developers never specifically asked for bundling. Instead, we got bundling as a side-effect of something else, something that we wanted reallllllly badly: **npm.**

npm -- which at the time stood for "Node.js Package Manager" -- was on its way to becoming the largest code registry ever created. Frontend developers wanted in on the action. The only problem was that its Node.js-flavored module system (Common.js or CJS) wouldn't run on the web without bundling. So Browserify, Webpack, and the modern web bundler were all born.



Create React App visualized: 1,300 dependencies to run "Hello World"

Complexity Stockholm Syndrome

Today, it's nearly impossible to build for the web without using a bundler like Webpack. Hopefully, you use something like Create React App (CRA) to get

started quickly, but even this will install a complex, 200.9MB `node_modules/` directory of 1,300+ different dependencies just to run "Hello World!"

Like Richard Hubbell, we are all so steeped in this world of bundlers that it's easy to miss how things could be different. We have these great, modern ESM dependencies now (almost 50,000 on npm!). What's stopping us from running them directly on the web?

Well, a few things. 😕 It's easy enough to write web-native ESM code yourself, and it is true that some npm packages without dependencies can run directly on the web. Unfortunately, most will still fail to run. This can be due to either legacy dependencies of the package itself or the special way in which npm packages import dependencies by name.

This is why Snowpack was created.

Snowpack: Web Apps Without the Bundler

Snowpack installs modern npm dependencies in a way that lets them run natively in the browser, even if they have dependencies themselves. That's it. It's not a build tool and it's not a bundler (in the traditional sense, anyway). Snowpack is a dependency install-time tool that lets you dramatically reduce the need for other tooling and even skip Webpack or Parcel entirely.

```
npm install && npx Snowpack
✓ Snowpack installed web-native dependencies. [0.41s]
```

Snowpack checks your `package.json` manifest for any `"dependencies"` that export a valid ESM "module" entry point, and then installs them to a local `web_modules/` directory. Snowpack works on any ESM package, even ones with ESM & Common.js internal dependencies.

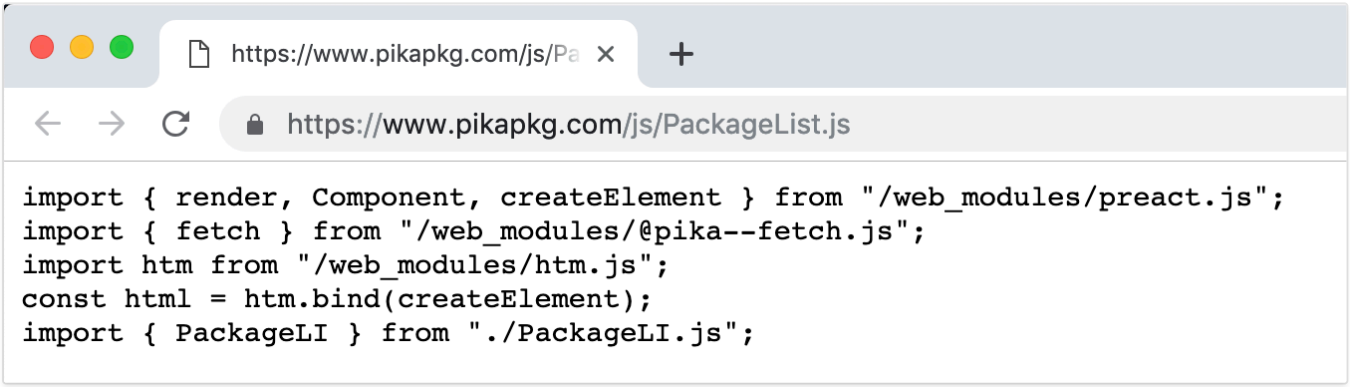
Installed packages run in the browser because Snowpack bundles each package into a single, web-ready ESM `.js` file. For example: The entire "preact" package is installed to `web_modules/preact.js`. This takes care of anything bad that the package may be doing internally, while preserving the original package interface.

"Ah ha!" you might say. *"That just hides bundling in a different place!"*

Exactly! Snowpack leverages bundling internally to output web-native npm dependencies, which was the main reason that many of us started using bundlers in the first place!

With Snowpack all the complexity of the bundler is internalized in a single install-time tool. You never need to touch another line of bundler configuration if you don't want to. But of course, you can continue to use whatever other tools you like: Beef up your dev experience (Babel, TypeScript) or optimize how you ship in production (Webpack, Rollup).

This is the entire point of Snowpack: Bundle because you want to, not because you need to.



PS: Oh yea, and view source is back!

Performance

Installing each dependency this way (as a single JS file) gets you one big performance boost over most bundler setups: dependency caching. When you bundle all of your dependencies together into a single large `vendor.js` file, updating one dependency can force your users to re-download the entire bundle. Instead, with Snowpack, updating a single package won't bust the rest of the user's cache.

Snowpack saves you from this entire class of performance footguns introduced by bundlers. Duplicated code across bundles, slow first page load due to

unused/unrelated code, gotchas and bugs across upgrades to Webpack's ecosystem... Entire articles and tools are devoted to solving these issues.

To be clear, leaving your application source unbundled isn't all sunshine and roses, either. Large JavaScript files do compress better over the wire than smaller, more granular files. And while multiple smaller files load just as well over HTTP/2, the browser loses time parsing before then making follow-up requests for imports.

It all comes down to a tradeoff between performance, caching efficiency, and how much complexity you feel comfortable with. And again, this is the entire point of Snowpack: Add a bundler because it makes sense to your situation, not because you have no other choice.

The Snowpack App Strategy

Snowpack has completely changed our approach to web development. Here is the process we used to build pika.dev, and how we recommend you build your next web application in 2019:

1. For new projects, skip the bundler. Write you application using modern ESM syntax and use Snowpack to install npm dependencies that runs natively on the web. No tooling required.
2. Add tooling as you go. Add TypeScript if you want a type system, add Babel if you want to use experimental JavaScript features, and add Terser if you want JS minification. After 6+ months, pika.dev is still happily at this phase.

3. When you feel the need & have the time, experiment by adding a simple bundler for your application source code. Performance test it. Is it faster on first page load? Second page load? If so, ship it!
4. Keep optimizing your bundler config as your application grows.
5. When you have enough money, hire a Webpack expert. Congratulations! If you have the resources to hire a Webpack expert you have officially made it.
