EPISODE 1036

[INTRODUCTION]

[00:00:00] JM: Modern web development involves a complicated tool chain for managing dependencies. One part of this tool chain is the bundler, a tool that puts all of your code and dependencies together into static asset files. The most popular bundler is Webpack, which was originally released in 2012 before browsers widely supported ES modules.

Today, every major browser supports the ES module system, which improves the efficiency of JavaScript dependency management. Snowpack is a system for managing dependencies that takes advantage of the browser support for ES modules. Snowpack is made by Pika, a company that's developing a set of web technologies, including a CDN, a package catalog and a package code editor.

Fred Schott is the founder of Pika and the creator of Snowpack, and Fred joins the show to talk about his goals with Pika and the ways in which modern web development is changing.

[SPONSOR MESSAGE]

[00:01:06] JM: When I'm building a new product, G2i is the company that I call on to help me find a developer who can build the first version of my product. G2i is a hiring platform run by engineers that matches you with React, React Native, GraphQL and mobile engineers who you can trust. Whether you are a new company building your first product, like me, or an established company that wants additional engineering help, G2i has the talent that you need to accomplish your goals.

Go to softwareengineeringdaily.com/g2i to learn more about what G2i has to offer. We've also done several shows with the people who run G2i, Gabe Greenberg, and the rest of his team. These are engineers who know about the React ecosystem, about the mobile ecosystem, about GraphQL, React Native. They know their stuff and they run a great organization.

In my personal experience, G2i has linked me up with experienced engineers that can fit my budget, and the G2i staff are friendly and easy to work with. They know how product development works. They can help you find the perfect engineer for your stack, and you can go to softwareengineeringdaily.com/g2i to learn more about G2i.

Thank you to G2i for being a great supporter of Software Engineering Daily both as listeners and also as people who have contributed code that have helped me out in my projects. So if you want to get some additional help for your engineering projects, go to softwareengineeringdaily.com/g2i.

[INTERVIEW]

[00:02:56] JM: Fred Schott, welcome to Software Engineering Daily.

[00:02:58] FS: Thank you, pleasure to be here.

[00:03:00] JM: If we look at web applications from a very high level, how could they be faster?

[00:03:05] FS: That's a great question. That's really the core question of the project. There are a few different ways to tackle that. Our big focus right now is on the dev-side speed, where if you imagine your web app built with Webpack or any sort of bundler today, it's doing a ton of work behind-the-scenes to get you that application. It's taking your source code. It's taking your dependency code. It's bundling it all up. It's splitting it back out. There's a ton of complexity that we've built up over the last decade that's really gone us today where sometimes you'll hit save and you'll go back to your browser and it will take a few seconds to really refresh and see those changes.

I mean, one application I was working on last year, it was a 10-second refresh just to get – Sometimes when you hit save, go to the browser, wait 10 seconds and then see the change. There's a lot of just really complex things going on behind the scenes that contribute to that, and that's one of the first parts of this problem that we're taking a look at as a part of the Pika project.

[00:04:03] JM: Tell me what the main pain points of modern full stack JavaScript development are.

[00:04:09] FS: There are definitely a few. We see that complexity as a real blocker for anyone who's getting started or really even building an app at a certain amount of scale, where to take the beginner, for examples, you end up in this world where, "Okay, I'm ready to learn web development. I'm ready to start my first application," and you get hit by these like 10 things you need to do to get set up. Install this. Npm install that. Check out your Webpack config, or maybe you're not using a Webpack config and you're using like a Create React app. Then what you're looking at is running npm install and you get like 1,200 different dependencies.

I think the last time I check, it was 200 megabytes or dependencies into your modules. Most of it all tooling. It's a really high bar for a beginner just to jump into that world. Then especially if there's anything wrong that they end up having to debug, like there's a bug in their bundles or something is not right. In that world where complexity is a given, the best a tool can do is really just abstract on top of it, which means all that complexity is still there even for a beginner or a basic use case. That's a huge part that we are trying to tackle with Snowpack and our different projects.

[00:05:13] JM: Can you talk about the modern tools that a JavaScript developer is using? I think of Typescript and Webpack and just these big stack of tools, the npm, that a JavaScript developer has to use. I just like to get an overview for what the average developers is working with.

[00:05:35] FS: Yeah, sure. I mean, the best case scenario is you get to choose the tooling that you bring to the table. I like types. I like Typescript. I'm going to Typescript. That's a really nice story. It clearly would bring you a benefit for doing that. Babel, if you're trying to work on a web app and you want to control what comes out of that pipeline building, but you still want to write modern code, you can start to add little transforms that kind of touch your code and make it work on more devices. Letting you focus on the modern code set in a modern set of features.

Going down the stack, then you've got your bundler, you've got CSS preprocessors. You've got – Oh God! What else? Any sort of plugins that you want to add for Webpack or Babel. Certain custom features that you want to add to your bundles analytics, different insights into what you're building. You can really dig that stack out in terms of what you support, in terms of CSS, or SaaS, wasm. It's a whole world of options, which is really what makes the web so exciting, is it's a really kind of un-opinionated place to build things. Again, the problems that you end up taking on all that complexity at the start, at the get go, versus on-demand, kind of when you need it and when it makes sense for you.

[00:06:45] JM: Go a little bit deeper on the role of a bundler in web app development. What is a bundler?

[00:06:50] FS: Yeah, that's a great question. A bundler today, it's such a required piece of dev tooling that we don't really ever ask and stop and ask that question. That's a great place to dig in. A bundler is really about taking your code that you've read and getting it to run in the browser. That feels like a really basic thing, like the browser should be able to run the code that we're writing, but really it's not something that's been possible to write code for the browser directly really over the last 10 years, maybe 7 to 10 years I would say.

That comes from a few different places. One is that, really, what we're doing is we're writing a module system of loading different files, but that we then have to send that to the browser that might not support something like an import statement or a required call. What we see is on one hand it's getting around this lack of a module system in the browser so that we can write the module system we want, but then we ship it to the browser and it's been built in a way that the browser understands.

The other, really, it's a lot of legacy technical decisions where essentially the browser of maybe 10 years ago, even 5 years ago, would bottleneck at anything more than a few, like let's say 6 requests per second. Essentially, the networking layer couldn't handle more than 6 requests, and because of HTTP 1, each of those requested assets, whether that was an image or a JavaScript file, those would be treated as separate connections, so different handshakes, different TCP connections. A lot of work going on behind-the-scenes just to make constant requests.

We're coming out of this world where that was really the default. You couldn't load more than a few things at a time, so you got to be really cautious about bundling as much as possible into those individual larger files. But now we're starting to enter this world where the browser can start to do a bit more. IE 11 is really the last browser that doesn't support ESM. So we're starting to see a world where this native module system, ESM, it's the import and export statements you're probably already writing. The browser now really supports that. We no longer need to get rid of those when we ship to production. We can start to just ship those statements directly to the browser.

The other is just HTTP 2, which is kind of I'd say the modern web networking layer today, and H3 which is upcoming. Both of those start to really simplify that networking store where there is no longer a bottleneck. You can now request multiple resources and they share one PCP connection. You get much faster loading. Then going into H3 upcoming, that will actually support parallel loading. Actually, streaming data down from a connection, a single connection across multiple requests.

We see, really, just a migration kind of trimming point moment that we're entering where a lot of the reasons for these bundling steps and these high-tooling workflows really don't make as much sense anymore.

[00:09:30] JM: Can you give me a brief history of JavaScript modules and how JavaScript modules relate to bundling and kind of their implications on this shift that you're talking about.

[00:09:44] FS: Yeah, definitely. There is actually a talk I give at Nebraska JS where I really kind of told this story, the history of modules and how it impacts this whole story. The three main eras, I call them. The first was we were all just doing whatever we wanted. If you were doing web development, this was like early 2010s or any time before then, you'd really just be like adding script tags everywhere and they had to load in a certain order, and maybe if you were smart on your own application code, you are using a model system like RequireJS, but it was really all runtime stuff. You'd call this defined function that would load your code. Really, imagine as if there was no foundational like import or require statement and you were just having to ship these function calls that essentially acted as your modules. It was this really Wild West where, again, you are able to do whatever you wanted, but the downside of that was there was really no standard way of doing these things. There was a few competing standards at the time that all solved different use cases.

What really changed that was node, node’s introduction into the CNN, and node’s promotion of CommonJS and npm as a whole. If you’ve seen a call to a require function to load a package, that's CommonJS and that's what node really pushed forward as their one standard way of doing things. Npm then really grew out of that where web developers looking and said, “Oh great! One standard way of doing things. That's awesome.”

What we can do is we can take that one standard way of doing things, we can get around these problems where that would never actually really run in the browser. You call require and then the browser has to basically at runtime stop everything it's doing and go load a package. That wouldn't really work on the web in the same way it works on node, which is more server-side.

When developers saw this one standard interface and one standard ecosystem they said, “That's great. We’ll take that.” Then they added tooling to essentially make that work for them. It’s this really critical moment in this story where everyone got on to one ecosystem and all the benefits of that. You can't understate how huge npm now is as an ecosystem and how much efficiency gains that gives us.

But the downside was that we as web developers really took on this agreement that said we will use this one standard way, but we will pay the cost of having to run tooling. Essentially taking an npm ecosystem that won't run in the browser directly, and then at build time, at development time, running these tools to make that work. A bit of a devil's bargain, because it brought us so much efficiency, but 10 years later now, we’re still seeing the cost of that with the time we spent waiting for bundlers or all the complexity that we see bundlers bringing in.

That really leads us to this last moment, which is ESM coming on the scene. That's the dev language. You guys are probably – I should say, anyone's probably writing this today and their application. Babel supports it. Webpack supports it, Rollup, all these tools have gone support for this language where you can use an import statement or an export statement and it all just loads within your ecosystem. It's a really clean, more statically analyzable language. Really works well for tree shaking and all these things that bundlers do well.

Again, we’re not really there yet were. Because npm is still written in this common JS way, that doesn't mean we can now ship that all directly to the browser. We’re still seeing this world where the tooling is required. We’ve built ourselves this ecosystem that won't run in the browser directly. Even with our own application code using modern syntax, we’re still not really there yet, where without some sort of tooling, the ability to ship that to the browser directly.

[00:13:05] JM: Browsers now have something called a shared module cache. Can you explain what a shared module cache is?

[00:13:12] FS: Yeah. Well, there're two sides of this, right? There’s kind of the legacy world and then the modern world, the modern world being more privacy-focused. Traditionally, your browser would load anything and then cache it, and this had this really neat performance story where if someone loaded jQuery, let’s say, and it’s older model from a CDN, the next site that loaded that jQuery file would actually be able to see that it already had it loaded in its shared cache. The browser was smart enough to actually reuse resources across different sites. You could start to have this world where you can visit a site that already have some of its code preloaded, cached, ready to run. No request needed.

That's still a model that makes sense for a lot of things, but the problem that we’re seeing now is that that really was started to be abused for fingerprinting where you could actually kind of, by checking that cache, see where a user had been before. There were some security implications that came out of this real shift towards a privacy-focused plan mindset in the browser.

Now, what browsers will do is they will actually lock down your cache to each origin. Your site will cache within that origin. So facebook.com, that's a Facebook.com cache, but you can’t go to another site and then check, “Oh! If I loaded some JavaScript from some secret Facebook file to see if this is a Facebook user.” It will just pretend like that doesn't exist in your cache unless you're within Facebook's origin.

We have this more sharded caching strategy that we’ve move towards. We see that moving somewhere in between the two where you have the super naïve shared cache and then the super strict sharded cache. We see a settling somewhere between the two where there's no real technical limitation to having a shared caching strategy that is fingerprint safe and security-focused where you could add some level of entropy or some number of times that a number of origins that had to load a file before it would be cached. You could start to do a few things to really abstracting, kind of anonymize that user while still providing a shared cache.

We still see a really compelling story there, but that's a few years out, and really depends on where these browsers settle.

[00:15:15] JM: Okay. Well, we've been giving a lot of context for the world of web development as you see it. You're working on a collection of projects called Pika. What is Pika?

[00:15:28] FS: Pika really is all about seizing this moment, the moment that I described earlier, where we're leaving this more legacy tooling-heavy world and starting to explore what was really just the browser and the platform catching up and starting to support these use cases without tooling as a Band-Aid. We're really excited about the different ways that this changes web development when you actually are writing in a module language that the browser itself can understand.

We've tried a bunch of different things. It was a really big year just for exploring, trying different things. I'd say the biggest project that's come out of that year has been Snowpack. Snowpack is our take on web application building. What we really are focused on is revisiting this world where you are writing application code and a language that browser understands, but we still have these dependencies as MPM ecosystem that the browser can't understand.

The question we ask is why are we sending the whole application through this bundler when really all we need to worry about today are those dependencies. What we realize is that there's this whole world of tooling to explore where instead of actually bundling the entire application, you can really just focus a tooling on the dependencies themselves. Process the dependencies and then you write your application without any tooling if needed, or adding tooling on as you want it. That's a really different model where dependencies never change. Once they're installed, they're installed.

What we’re able to do is not just move the tooling, but also really reduce the amount of times you need to run your tools. Essentially, after install, running a one-time tool that changes those dependencies to make them web-ready. Doing everything a bundler would do to pull them into individual files, handle any sort of inconsistencies in the network and really making them web-ready for you. Doing all the things a bundler do before your dependency specifically.

Then that lets you build your application without the bundler. Writing import statements, writing export statements, shipping that to the browser, and the browser understands it. You actually don't need a lot of that tooling. You certainly can then start to still grab the tools you want. So you can grab Typescript, grab Babel, grab whatever you want, but it's that model of required versus opt-in, where now you're just pulling in tools that really only have to work on individual files at development time.

Run Babel, you hit save. Babel will process that one file, but it’s not having to re-bundle anything or reprocess entire bundles. It's on a file per file basis. What we’re able to see is this really, really sped up development workflow where you hit save and all that you needed really work on and process on the tooling level is that one file. You get really quick reactions and iteration speeds based on that model versus having to run a ton of tooling on a huge application bundle.

[SPONSOR MESSAGE]

[00:18:13] JM: DNS allows users to navigate to your web endpoints, and whether they are hitting your app from their mobile phone or accessing your website from their browser, DNS is critical infrastructure for any piece of software.

F5 Cloud Services build fast, reliable load-balancing and DNS services. For more than 20 years, F5 has been building load-balancing infrastructure, and today, F5 Cloud Services provides global DNS infrastructure for lightning fast access around the world. If you're looking for a scalable, high-quality DNS provider, visit f5.com/sedaily and get a free trial of F5 Cloud services.

Geolocation-based routing, health checking, DDoS protection and the stability and reliability of F5 networks. Go to f5.com/sedaily for a free trial of F5 Cloud Services, fast, scalable DNS and load-balancing infrastructure. That's f5.com/sedaily.

[INTERVIEW CONTINUED]

[00:19:22] JM: I'd like to go a little bit simpler on some of just the explanations you've given just to set a better stage for explaining Pika. Something you've said is just that there are certain dependencies that I think you said the browser understands versus dependencies that the browser does not understand. When you're talking about JavaScript dependencies that a browser does or does not understand, what do you mean by that?

[00:19:51] FS: Yeah. I mean, I live in this environment. Thank you for getting that clarification out of me. Really, there aren't many dependencies that the browser understands. I'd say almost nothing on npm today will just run on a kind of file per file basis in the browser, right? If you point to something on unpackge, for example, unpackage being the CDN for different npm files. Very few of them can run on their own without sending sort of additional tooling with just like – Versus the simplicity of a require React, and use that name React and you want that to just kind of run in your application. Not many packages support that today.

What we do is we just provide the tooling that gets them up to that place where they will run in the browser. Getting all those packages on npm that don't run in the browser, just running a one-time tool to essentially get them there, process them, remove any sort of code that was maybe written for node or for a bungler that wasn't written for the browser. Removing any inconsistencies or any required calls converting those to import calls. Just anything that the browser would need to run that file that otherwise wouldn't run in the browser without a bundler or some sort of tooling like Snowpack.

[00:20:57] JM: Okay. If I understand you correctly, there are certain dependencies, certain packages that somebody might be using in their application, for example, via npm, that are going to require a set of steps to make them natively available to the browser. Your goal with Pika is to shorten the accessibility to the browser of getting those in a consumable format.

[00:21:27] FS: Yeah. I'd say that's really the focus of Snowpack, which is our project within Pika that really tackles this problem. It's all about getting you dependencies that run in the browser so that you can build your application without a ton of tooling. No Webback. No Create React app. Nothing needed that you don't actually want to pull in, like a Typescript or a Babel. You're certainly welcome to bring in whatever tooling you want, but when you do, it just runs a lot faster because you're building off of this much lower tooling level versus a traditional application.

[00:21:57] JM: If I'm not loading them into my application in the historical pattern, what's going on with Snowpack? How is that – I guess, I'm sorry. I am sorry I am having a little bit of trouble with this. I'm totally not a frontend expert.

[00:22:14] FS: No. Not at all. It's a totally different way of building web apps. Not at all. I can give an example if that helps.

[00:22:20] JM: Please.

[00:22:20] FS: Of what's going in a traditional app.

[00:22:21] JM: Yeah. Yeah. Yeah.

[00:22:22] FS: Let's say you're Create React app and you write require or import React. So you're trying to load up React into your application so that you can create a component, create a page, whatever you want to do with React. What your bundler does at that moment when it sees that statement is that it takes all the files within React and any other dependencies of that file, that main application that you're building. It pulls them all together into one JavaScript file. Maybe 1 to 3 depending on how deep in the defaults you've kind of gone.

Your bundler is essentially bundling all those files that make up your web app together and it's processing them. It's running Babel or typescript. It's doing all of those work to get you really much more bundled application files that the browser will then run for you and you see your application, you see your Hello World.

What Snowpack does is that it actually at install time takes a look at React and says, “Okay, here's React. It's not really ready to run in the web. It's going to need some sort of bundling step. Let's run that right now. Let's run it once,” because dependencies, once they’re installed, don't change. Then give the user, the developer, this file that they can import directly from their application.

Inside of their application, they would still import React, but now that's a file that actually doesn't need any sort of additional tooling. It's really a single JavaScript file after we've taken a look at it and run on it. What you’re able to do then is actually just kind of skip the bundling step entirely where you were taking your entire application and bundling it together and running all these tools. Instead, you're able to just – Using an import statement, which is just this loading statement that any ESM import statement. You can write just import React or import from the path that React lives at, and now you can just run this whole application directly in the browser. No need for Webback. No need for a dev set up that's really complex.

[00:24:11] JM: In a world with Pika, what is your vision for how web development changes or maybe we could talk about Snowpack specifically first. How does my work as a developer change with Snowpack?

[00:24:29] FS: Yeah, we’re really focused on that change really for this more simple, for the better. A big thing right now is that we just see this really large foundation of tooling built up because it's been so required over this long period of time. What we see is more that that tooling should be optional, right? You can pull in a bundler if you want for a production. Maybe you really want a lot of control over how your application loads and you want it to be in this individual files for performance reasons or for whatever reason. You can bring those tooling in. Bring all that tooling in as you need it. But really what we see is this kind of democratization of what sort of tooling is required where you can start to build your application, really pulling in things that solve direct problems for you. If you want to build out a really simple application, you can skip the bundler for a long time. If you want to pull in Typescript, you can pull that in, and it's not a really complex change or addition.

In talking to different developers, we’ve just seen a ton of these really complex plug-in setups and all these different integrations within the bundler that really create this complex, almost just a ball of yarn of tooling that we really don't see as required anymore. If you've ever like dug through a Webpack config and felt like that was too much or too complicated, that's exactly the problem that we're trying to solve for.

[00:25:48] JM: If I want to use Typescript, for example, in my application and I'm going to be able to avoid the tooling associated with using Typescript or the build steps associated with Typescript. What would that look like with Snowpack? How would my life be easier if I was using Typescript but was using Snowpack?

[00:26:12] FS: Yeah. Going back to that example where every time you hit save and that simple kind of application, Webpack is doing all this work that it kind of inserts Typescript into. It will say, “Type check your files and then bundle this application.”

In our model, really, again, you'd get rid of that bundling part. Typescript really becomes a per file tool where I make a change to, let's call it my index file, the top file within my application. All Typescript will need to do is just be watching your tree of files and say, “Oh, this one file changed. Let me just go back through that one file and take a look.”

Babel is another one where instead of having to rewrite and rerun on every bundle, you can just rewrite and rerun on the individual file itself. Hitting save, you get that really small transform versus a much larger per bundle transform. It ends up running much faster than the traditional model.

[00:27:07] JM: I see. When am I feeling this speed increases? At every build time?

[00:27:14] FS: Yeah, it's both. It's both ad dev time time. As you're iterating on your site, you make a change, you hit save, go to the browser. Make a change, hit save, go to the browser. As you're iterating developing, this really speeds up the flow, because you're no longer doing any sort of bundling at dev time and you can only really – If you're doing any build tool setup, it's only on a per file basis. Speed up number one is all dev-focused, development time speed up.

Number two is just that your build time, if you are – Sorry. Building for deployment. Let's say you're getting ready to deploy this thing. Again, you no longer have to do any bundling. You can just kind of ship that application to the browser. It's really just a file copy to your host. A really quick static almost deployment, or if you want to do any sort of performance bundling for production itself, you're now able to take that on and all of your dependencies are already built for you. Again, we've done that transform ahead of time. Even at deployment, we see a huge speed up as well. It's really across-the-board between starting up your dev environment, actually hitting save and then seeing those changes reflected immediately. Then finally at the actual deploy to production stage, we see speedups across-the-board.

[00:28:22] JM: Can you walk through an example of what Snowpack is doing under the hood? I think you went through a little bit of this with React, but maybe you can just go through another example. I’m sorry that I'm having trouble understanding the internals too much, but I'd like to understand how Snowpack works in a little more detail.

[00:28:41] FS: Yeah, sure. I’ll even drill down further into like really what is that React package look like in a traditional model versus Snowpack.

[00:28:49] JM: Sure.

[00:28:49] FS: Really, what Snowpack does is it looks at three different problems within npm package today. The first is that a lot of those packages were only ever written for node. Even a lot of things like React, which seem more frontend-focused were written for this world of node that then bundlers kind of took on that ambiguity for them.

There are a ton of like – I don’t know if you’ve ever seen a require call to a node-specific package. That would never actually live in the browser. First problem is that you have this really node-specific code kind of all over the npm ecosystem, even in frontend packages. What Snowpack will do is it will actually try to polyfill behaviors where it can and essentially kind of paper over those node-specific behaviors in the same way that a traditional bundler does today. The first transform, make your package that might've been written for node, really get rid of anything node-specific. So that’s just kind of run anywhere.

The second transform that we do is that what node supports – Again, this all comes back to this idea that we’ve inherited this ecosystem from nodes. All the nodeisms become web developer problems as well. The second nodeism is that requiring by a package name or may be requiring a path that's incomplete. If you've ever seen one of those imports or require calls that doesn't have a .js at the end, all of that is really node-specific. The browser has no idea how to go look up React by name or dependency of React by name and it is no way to, "Okay, there was no file extension included in this path." Is that a file that ends with JS? Is it a directory? I don't know.

We take care of all these things as well, where sometimes the dependencies of your own dependencies. They're not really referenced in the way that the browser would ever be able to understand. They're importing packages by name. They're importing by an incomplete file path. We as Snowpack and, again, what traditional bundlers do, is we take those and we paper over them. We rewrite them so that they work and that they're actually pointed to individual files at that level.

Then the third thing that we do is that we really just take a look at the actual package itself where Lodash is a great example. Lodash, as a package, is I think like a thousand files. It's a ton of different files within Lodash itself. If you imagine that loading on the browser, again, you're no longer being penalized for multiple requests, but still that's a lot of different files to really go out and make those requests for.

The third thing that we do is we take individual packages and we pull in all the individual files within that to just give you a single JavaScript file. Again, it's really similar what a bundler would do to a whole application, but taken at the dependency level, what we're doing is we're pulling this entire package into one file. You end up with a ReactJS file. That's what then you're able to use and load directly in the browser that it contains all the code of React ready to go and ready to work for you.We really bundle in the same way a traditional app bundler does, but we take those same problems and we really just focus on the dependencies themselves, giving you dependencies as single files that would then run in the browser directly.

[00:31:43] JM: Okay. To summarize, the difference between Snowpack and another bundler would be that Snowpack gets everything into a single file?

[00:31:56] FS: Yeah. It gets you dependencies in the single files so that you don't need to rebuild or re-bundle every time you spin up something like Webpack. They're just there ready to go. They're static. They don't change after install and they're ready to be imported into your application directly without any other additional tooling.

[00:32:13] JM: Got it. Pika is a collection of projects, and you've got a package catalog. If I'm up package publisher, like if I am the creator of React, for example, how would my usage of the package catalog differ from the npm package catalog?

[00:32:36] FS: Yeah, that's a really interesting question, because that's really how we got started on the first project we ever released, was just a catalog for finding more modern packages, packages that were written as themselves, packages written in this newer ESM import-export syntax. Originally, what we are taking a look at was the fact that we had this new syntax ESM, but so few packages actually supported it within their own code. Again, they were coming from this npm model where everything was written for node, common JS. Not really leveraging this new language that was a really powerful kind of improvement to any package. It was something that could be better tree-shaken so you could kind of remove unused code at build time.

ESM was this really cool technological breakthrough for module systems, but we weren't seeing a lot of usage in the ecosystem. What we wanted to do was build a catalog where you can go and find the packages that were using this. Really starting when there were only about 40,000 packages at the time, and now, since then, that I think we're were up 200,000. It's really taken off since then and just having a place to go and find these more modern packages. Essentially, we get to act like this catalog for npm, but really focused on web developers.

We've built out of that not just the ability to find these more modern packages, but start to really showcase better information about the more tailored to web developers and even give you a sort of like a REPL to play around with each package within that catalog. We're really focused on building an npm search that isn't just for node or the general developer, but really focused on web developers.

[00:34:09] JM: I thought that like everyone who's building a full stack JavaScript application is using node. What is the alternative? What is the situation where a developer is not using node?

[00:34:23] FS: Every web developer is using node, right? If you're using any sort of tooling or maybe a node-powered server. Node is pretty, yeah, ephemeral in that web developer’s world. The difference is that sometimes packages are written assuming that their code is going to run in node. Sometimes a package will be really heavy in terms of what it’s being built with, because it doesn't expect to be run on a user's device. It's expected to be run on the server side where file lookups and different node packages all exist.

With that ends up looking like is that it's really easy if you’re just using npm’s general search catalog to find a pack and then say, “Oh! This is really great. It has a ton of usage. I’m going to go pull it into my application and use it on the frontend.”

Sometimes you then run into this problem where that package was never really written for the frontend, and Webpack and its attempts to paper over those issues will start to pull in a lot of extra code that you really weren't expecting to come in with, different polyfills or really node-specific dependencies.

We see this problem where the node ecosystem and the npm catalog are really focused on node and this node ecosystem, but what we were seeing was that that wasn't really taking into account the performance story of a web app where code size and code complexity is really important to building a performant app.

[00:35:37] JM: Really what we’re talking about here is packages that were originally designed to run on the server that people now want to run on the client?

[00:35:48] FS: Yeah. Yeah, there are a few packages like that that were so popular in node that people started to just reach for them because they are popular in their frontend app as well. Certain packages like Request is a really good example. That was a really early node HTTP request library. Making an API call, requesting some sort of response from a server. That really never was built for the frontend, and I did a lot of work on that package myself. It was really built using a lot of node internals, node streams. It was a really node-specific package that then caused a lot of issues for anyone trying to pull it in for the web. That was a big sort of solution that we saw missing from the ecosystem. It was a catalog that was really built for frontend developers.

[00:36:31] JM: One part of Pika we haven't talked about much is the package code editor. If I'm somebody that wants to build a JavaScript package or a component, you have this package code editor. Can you explain how that improves the experience of the developer who is creating a package?

[00:36:49] FS: Yeah. That's one of our more new, still experimental things that we're working on, and we're actually looking to open source that pretty soon. The idea being that if you've ever built a package, npm really doesn't give you a lot to get started with. It will give you essentially a package JSON, and that's it.

What you then have to do as a developer building this package is you have to pull in your different tooling, whether that'd be Typescript, your different test libraries, your different builds. You have to not just pull these tools in, but then set them all up yourself. Most developers that I talked to, they essentially have this like kind of copy-paste package structure that they use that they kind of just copy from package to package. It's not a really great set up for really easing anyone into building a package, especially as we see this complexity of the package building process where you want a Typescript or you want some sort of build process within your packages itself.

Npm was only ever written assuming that you'd basically just have a JavaScript file and that's it. That's not really the world we live in anymore where there's a lot more tooling involved to build a package well. The code editor really existed to solve that problem, where instead of being a ton of setup and configuration, the editor itself was really built for package creation than package editing. You would get this environment where, really, it would just give you a place to write source code and the editor itself would start to run typescript, publish that package for you, run your tests for you, really pull in everything that it takes to build a package into the editor itself.

It's a really similar approach to Rome. If you've heard of Rome, it's the creator of Babel and Yarn. His new project which same exact concept, let's get rid of all these different compete in and kind of disconnected tools and build one ecosystem that solves this problem really holistically. We're the exact same approach, but really at the editor level.

[00:38:36] JM: Another project is a CDN for npm packages. How does the package CDN fit into your overall vision for Pika?

[00:38:47] FS: That's a really important part of the way we see Pika growing. It was another part of the code editor story, was building out this registry for packages. A place to publish packages where not only were they JavaScript code that you could npm install, but that it almost live on the web at this almost more like a CDN than a registry where you could pull these packages. A great example of how this all fit together was that the code editor would actually be able to run your tests by pulling them and essentially importing them off of the registry live in the browser.

We saw there'd just been a ton of room for the registry itself to kind of upgrade to this new world where packages were living and breathing online versus being this sort of really heavy artifact the you had to npm install and then run a bunch of tooling just to get it back into the browser.

We've really since taken a look at that registry and the CDN and trying to bring them together into a cohesive story where any package could be loaded from our CDN, weather it built with our code editor or it lives on npm. From that, really giving you this really nice performance story where the assets that you load from the CDN can start to be optimized at a really interesting way.

One example is that we actually differentially serve every package coming out of our CDN based on the browser. If you make a request to our CDN with, let's say, older version of Firefox from 2018, we'll give you code that has been polyfilled and transpiled. All those things that most web apps are doing today to get that code to work for that browser.

But the interesting part is that then if a modern Chrome browser, let's say, comes in and makes a request, we'll see that it actually has support for all the latest features and we won't end up transpiling that as much. We'll actually give it much smaller, more efficient code. There's a really interesting thing where you can do when you start to own the CDN in that way. You can start to make these decisions about performance based on the person requiring that code. We see that is a really interesting approach to web app development and just web hosting in general.

[00:40:47] JM: Tell me about that your interaction with the people who use Pika. I'd love to hear about both the package development and release process as well as the package consumption process.

[00:41:00] FS: Yeah. We really are just testing that code developer internally. We have a group of Patreon supporters that are really part of this early beta for that. We've seen a few different packages come out of that and really simplified node tooling to set up, nothing like that. Most of our community that was really based on that Snowpack world where you're using our different tools to actually help you build apps today. You don't need to stray as far from the existing world that you're living in. You can kind of just run this as a package install step and then get those files for you in development. That's really where we see that story about speeding up your development, removing tooling, removing complexity. There's just a really a whole movement around removing that tooling and going un-built or unbundled that's really exciting to see that grow out of this movement.

[SPONSOR MESSAGE]

[00:41:55] JM: Looking for a job is painful, and if you are in software and you have the skillset needed to get a job in technology, it can sometimes seem very strange that it takes so long to find a job that's a good fit for you.

Vettery is an online hiring marketplace to connect highly-qualified workers with top companies. Vettery keeps the quality of workers and companies on the platform high, because Vettery vets both workers and companies access is exclusive and you can apply to find a job through Vetter by going to vetter.com/sedaily. That's V-E-T-T-E-R-Y.com/sedaily.

Once you're accepted to Vettery, you have access to a modern hiring process. You can set preferences for location, experience level, salary requirements and other parameters so that you only get job opportunities that appeal to you.

No more of those recruiters sending you blind messages that say they are looking for a Java rockstar with 35 years of experience who's willing to relocate to Antarctica. We all know that there is a better way to find a job. So check out vettery.com/sedaily and get a $300 sign-up bonus if you accept a job through Vettery.

Vettery is changing the way people get hired and the way that people hire. So check outvettery.com/sedaily and get a $300 at bonus if you accept a job through Vettery. That's V-E-T-T-E-R-Y.com/sedaily.

Thank you to Vettery for being a sponsor of Software Engineering Daily.

[INTERVIEW CONTINUED]

[00:43:45] JM: I'd like to just revisit and kind of review the thrust of Pika and the changes to web development because – I mean, this mostly for my own education. It's like I did some shows recently about React. I've done a lot of shows about React in in the past, but that every time I revisit the frontend, it just makes me realize how confused I am about the different steps involved in developing a frontend application. But I'm really having just some trouble understanding the thrust of Pika. I guess it relates to my confusion around ES modules and in the impact that ES modules have had on web development. Maybe you could just start from the topic of ES modules and explain like what is an ES module and how is that changing the web in a way that facilitates this need for Pika.

[00:44:43] FS: Yeah, definitely. ESM is really the technology that we are so focused in and kind of just pushing adaption for, promoting it across whether you're building an application or you're building a package. What we see as its main thing, or really a few of them. One is just that no matter what you're building, it's essentially just a better language for every platform or tool to understand. Essentially, going from a require call, which is this more traditional common JS. Basically the thing that everything has been built with up till ESM was created in that node kind of npm era. The required call was really powerful because it let you load a dependency, but it was a runtime call. It was really hard for bundlers or any sort of tooling to go and see, "Okay. Is this the string? Is this a variable? What am I even loading here?" It was a really dynamic module system, which meant that it was really hard to optimize for at build time.

One of the really exciting things about ESM is that it's actually built in the language. It's a really statically analyzable import statement and a really statically analyzable export statement, which means that tooling can then go and kind of, without really need as much complexity as you'd traditionally think, go through your file system and get a really clear understanding of what is imported from where. It can statically analyze everything about your application and how it's connected. How your files are connected to each other. That gives it a ton of control over how it can then bundle. How it can remove dead code. How it can remove things that aren't even being imported by anyone. So why even keep them around?

It's a really analyzable file language, import language. Then browsers, module systems, bundlers, anyone can really optimize for. We see that as one of the big benefits, especially at that packaging level to let your consumers have that ability to remove dead code and optimize.

From there, we see it really just – It is a big language change and that now there's, for the first real time, a module system that runs natively in the browser. Through the entire history of web development, there's never really been this stability of the browser to have one JavaScript file load another. That seems really foundational, but it's actually really modern in the way that the web has been built. This understanding that JavaScript really is the language of the web today where you want to build an application in JavaScript that then loads more in JavaScript. That's something that just has never been supported.

This idea that we are coming from a world where that has never been supported, we need to use tooling to essentially ship our own module system to the browser, to today where now it can actually write code that the browser will just natively run and understand. That's a really fundamental shift that just wasn't supported back even five years ago. Writing code that the browser understands is really fundamental to how you can start to speed up your development where you don't need to write code that the browser doesn't understand. It's a different model where actually you end up writing code that kind of takes you two steps backwards, versus instead writing code that leverages this new system that the browser then natively understands. So you don't need to do a ton of extra work on top of that.

[00:47:45] JM: It is the root of it that this ES module system, since it's natively – It's statically typed or it has some kind of type inference system where you can understand, you can cut down on the dependencies that you might be speculatively adding to a package?

[00:48:07] FS: Yeah. I wouldn't call it a type system, but you can definitely statically analyze what files are loading from where and what are they actually pulling from that file individually. Not really the types of those files or any sort of type system data, but definitely you can really easily analyze, “Okay this – My index file, my top-level application is only grabbing, let's say, the Lodash object merge function.” But I don't need to then pull in the entire Lodash library and all those 100 different methods. I can actually, as a bundler or as Snowpack, the install step, see the you only need that one thing and then go from there. Only pull that one thing into your final application.

[00:48:46] JM: I see. Okay. A package like Lodash might have hundred different methods. As a developer, maybe I only am using one of those methods in my application, and therefore I can cut down on application bloat by just fetching that one method from Lodash and loading that into my application?

[00:49:09] FS: Yup. That's essentially what ESM is really good at, because it's so statically analyzable. Import that one method from Lodash. Versus the more traditional model and common JS specifically being this dynamic module system. You'd basically import Lodash from of a runtime call. You had to basically do your best to understand what that runtime call that require call was doing. Still, you’re then just getting a variable back. Who knows what you're doing to that variable later? It's really hard to pull apart the individual things you're using in the more traditional common JS language.

[00:49:44] JM: Is there like metric for how much this cuts down on the size of applications or the latency of applications? It doesn't have to be like a hard number, but maybe you some case studies that you can refer to.

[00:50:00] FS: Yeah, it really depends on how you're using is different things. Really, it's on a case-by-case basis, but something Lodash, I wish I had the numbers in front of me, but it's a pretty large package, and if you’re only loading that one method off of it, that's a huge saving right off the bat. I don't have a real-life case studies, but it's been the best way to see if this can help you, is just to – As you run Webpack, turn tree shaking on-and-off and see if this is doing anything for you.

[00:50:29] JM: So you're building company around Pika, right?

[00:50:31] FS: Yeah. Yeah, we're really, really focused on this future being a big part of web development.

[00:50:37] JM: Tell me about like what your vision for the company looks like.

[00:50:41] FS: Well, part of it is around supporting these tooling, so these different tools, like Snowpack and really getting them kind of the investment that they deserve. The other is really looking at the CDM as actually a way to change how we think about development where we're able to host these dependencies in a really interesting way for performance where we can actually optimize those dependencies for performance in a way that, again, in that traditional model where everything soup together. It's kind of harder to do much optimization when you don't have the context of a source which might change rapidly over the day as you redeploy and dependency that might not change very often.

What we're able to do is actually host some of those dependencies in a way that actually is much more performant than the traditional default application and starting to look at how we can build performance and build integrations that really start to pull from our CDN in production even.

[00:51:37] JM: Cool. I have a few broad industry questions just relating to how fronted web development is changing. Web assembly, is that actually changing frontend web development today or is it kind of still too early to really speculate on how web assembly will significantly change frontend, unless we're talking about very specific things like Dropbox using a compression library or something like that?

[00:52:09] FS: Yeah, that's a really interesting question. I don't know if anyone knows the answer of how that'll all shake out, but it's clearly showing really big wins for those really process-heavy. There's a great application that Google released where it actually – I think it's called Squoosh or Squish, Squoosh.app. It uses WASM to actually run image compression for you. So you can drop an image in and then immediately get back a compressed, a smaller file without much loss there. Really, really interesting things we can do when you can start to run this more just bare metal code in your browser.

I'm not sure if it will change too much today or really about how we build applications just given how much interaction there is with a DOM. It's much harder to kind of breakthrough that barrier of DOM manipulation versus WASM kind of machine level code. We today see it really as this thing that you can lean on for specific tasks. But certainly, who knows how that will transform as it grows. It's a really exciting kind of part of the web right now.

[00:53:10] JM: You worked at Google for a few years on the Polymer team. What are your reflections on web development, from your time on the Polymer team?

[00:53:20] FS: That's a great question. That was a great experience to see the web and really this ESM problem at this early kind of transformative part as it was getting specked out. ESM was just kind of being launched in the first browsers, really early kind of transformative years for this technology. That was a really interesting exercise in seeing that, and then seeing the web components community really built out of that idea that you don't need all this complexity. The team was really focused on trying to remove complexity where it could. Certainly, web components are growing out of that period now that they're supported in all browsers. It's an interesting time for web components as a whole as well. But really, what I took out of that was how interesting this ESM technology was and the opportunities for simplifying JavaScript through that technology alone.

[00:54:06] JM: Definitely. How has the React ecosystem changed due to ES modules?

[00:54:13] FS: That's a good question. React itself is actually not published as an ES module today. A ton of tooling goes into just this fact that the core part of React isn't actually using this more modern module system. I know they're working on it, but it's been a long time that that issue has been open and it's something that's just I think a harder, more kind of basic part of their own tooling within Facebook and React as a project.

In terms of the ecosystem as a whole, the nice part is that, again, the bundler and Snowpack, whatever you're using, that can really paper over these issues. Even though React itself is an ESM, plenty of the ecosystem is using this module system. Again, it's really what most of us as web developers are writing our applications in today. Even with them kind of lagging behind a bit, we're still seen a ton of adaption even within that ecosystem just based on the optimization benefits alone.

[00:55:03] JM: Do you have any perspective on how Google and Facebook differ when it comes to their perspectives and their plans on frontend web development?

[00:55:13] FS: I could make some guesses. I don't have any real true insight. It's been a while since I've been at Google. I mean, seeing them in their different approaches to the web, I'd say Google is really looking at this from a performance first kind of perspective. They really want the web to be fast. I believe they see their role as keeping the web fast and helping developers get there.

A lot of the problems brought in by this more complex toolset is it's really actually pretty difficult to build an application that's fast. The defaults really don't get you there. So you really need to go out of your way to learn how cold bundling works, code splitting work. Which different packages you can use? How to architecture applications? It all factors into performance in a way that's really hard to grasp and get control of and you kind of regularly need to stay on top of. I'd say that's Google really. That's kind of if I had to guess where they see themselves.

Facebook definitely takes that they're more bottom-up approach, where they're investing in the sort of tools with React, and Rome is again I think a Facebook project. They're really taking the more I think tooling-heavy or tooling-centric approach.

[00:56:16] JM: Last question. Fast forward three years. What role will CDNs and edge computing play in frontend engineering?

[00:56:27] FS: Well, that's something that I think is really changing right now. The three years, it's exactly – I mean, even one would be an interesting question.

[00:56:34] JM: Sure. Whatever.

[00:56:36] FS: Yeah. Zeit and Netlify and all those Jamstack offerings I think really kind of hit home this idea that when you go fully static, you can do a few really interesting things. Meaning, when you're not actually generating much code at the server level, when really the server's job is just to give you this frontend application that is running quickly and maybe server-side generated, but again, it's frontend code. When you start to do that and think in those terms, you can start to really plug in the CDNs much more easily than before, where the CDN almost becomes as de facto pluggable thing.

Zeit and Netlify, really essentially being CDNs themselves and giving you that performance benefit of a globally distributed site for free. That's a huge win that I don't know if everyone really captures how well that speeds up your performance to just be globally distributed at that level. It really hammers home this move from backend to full stack, and now we seem to finally be in this final phase where everything is frontend. You build a frontend app and then the server's job is just to kind of get that to the user as quickly as possible. That's a very different model than even five years ago where the server was doing a lot of generation and you kind of have to stitch together these server-side template renderings and these frontend responses. It was kind of this much more murky situation whereas now we really see the move being towards fully frontend applications.

[00:57:56] JM: Fred, thanks for coming on the show. It’s been great talking to you.

[00:57:58] FS: Yeah, you as well. Thanks for having me.

[END OF INTERVIEW]

[00:58:09] JM: Over the last few months, I've started hearing about Retool. Every business needs internal tools, but if we’re being honest, I don't know of many engineers who really enjoy building internal tools. It can be hard to get engineering resources to build back-office applications and it’s definitely hard to get engineers excited about maintaining those back-office applications. Companies like a Doordash, and Brex, and Amazon use Retool to build custom internal tools faster.

The idea is that internal tools mostly look the same. They're made out of tables, and dropdowns, and buttons, and text inputs. Retool gives you a drag-and-drop interface so engineers can build these internal UIs in hours, not days, and they can spend more time building features that customers will see. Retool connects to any database and API. For example, if you are pulling data from Postgres, you just write a SQL query. You drag a table on to the canvas.

If you want to try out Retool, you can go to retool.com/sedaily. That's R-E-T-O-O-L.com/sedaily, and you can even host Retool on-premise if you want to keep it ultra-secure. I've heard a lot of good things about Retool from engineers who I respect. So check it out at retool.com/sedaily.

[END]