EPISODE 984

[INTRODUCTION]

[0:00:00.3] JM: The JavaScript ecosystem stretches across front-end, back-end and middleware. There are newer tools, such as GraphQL, Gatsby and WebAssembly. There are frameworks like React, Vue and Angular. There's complex data handling with streams, caches and TensorFlow.js.

JavaScript is unlike any other ecosystem, because a single language can be used to construct every part of an application. Because JavaScript is used for such a broad spectrum of use cases, the amount of tooling available can be intimidating to someone new to the ecosystem.

Kevin Ball is a host of JS Party, a podcast on the changelog network. Kevin joins the show to give his perspective on the JavaScript ecosystem. In this episode, we discussed ES modules, the jam stack and the growing number of tools, libraries and workflows used by JavaScript developers.

We are hiring a software engineer who can work across both mobile and web. This role would work on softwaredaily.com, which is a Vue.js application, our iOS app and our Android application. We're looking for somebody who is very flexible and who learns very quickly and can produce high-quality code at a fast pace.

If you're interested in working with us, send me an e-mail, jeff@softwareengineeringdaily.com. We are looking for somebody who's hungry and somewhat entrepreneurial. I would love to work with you if you are well-versed and a fast learner. Just send me an e-mail, jeff@softwareengineeringdaily.com.

[SPONSOR MESSAGE]

[0:01:46.6] JM: As businesses become more integrated with their software than ever before, it has become possible to understand the business more clearly through monitoring, logging and advanced data visibility.

Sumo Logic is a continuous intelligence platform that builds tools for operations, security and cloud native infrastructure. The company has studied thousands of businesses to get an understanding of modern continuous intelligence, and then compiled that information into the continuous intelligence report, which is available at softwareengineeringdaily.com/sumologic.

The Sumo Logic continuous intelligence report contains statistics about the modern world of infrastructure. Here are some statistics I found particularly useful; 64% of the businesses in the survey were entirely on Amazon Web Services, which was vastly more than any other cloud provider, or multi-cloud, or on-prem deployment. That's a lot of infrastructure on AWS. Another factoid I found was that a typical enterprise uses 15 AWS services. One in three enterprises uses AWS lambda. Appears serverless is catching on. There are lots of other fascinating statistics in the continuous intelligence report, including information on database adoption, Kubernetes and web server popularity.

Go to softwareengineeringdaily.com/sumologic and download the continuous intelligence report today. Thank you to Sumo Logic for being a sponsor of Software Engineering Daily.

[INTERVIEW]

[0:03:36.7] JM: Kevin Ball, welcome to Software Engineering Daily.

[0:03:38.8] KB: Hey, Jeff. Good to be here.

[0:03:40.3] JM: I'd like us to take a tour through the modern world of JavaScript. The place I'd like to start is ES modules. Explain what an ES module is.

[0:03:50.7] KB: Well, let's start by going back a little bit to talk about how modules in JavaScript evolved over time. Unlike some languages that start with this concept of code isolation and modules and things like that, JavaScript when it originated, everything was in global scope. There was no concept of a module and separating things and pulling things out, because it started as a play language on the web.

Then as people started to do more serious software engineering with JavaScript, they wanted to use good practices, like code isolation and things like that. Initially, there were a bunch of what you might call user space solutions to that; folks who basically built up using the language as it existed, ways to create modules. That's where you get things like AMD, which is one of the first specifications, you could call it. It was essentially once again, a user space specification. If you write your code in this way, it will work with this tooling and we can load it dynamically and you get nice isolation and things like that.

Over time, the language became more mature and folks started saying, “Hey, we should actually have a first-class solution to this.” The first closed thing we got to that was when Node.js came around and they said, “Hey, we're writing server-side packages. We need a way to do this. We're not going to be shipping stuff up to the browser. We're not isolated in the same way. We don't have as limited scope in the same way. We're just going to make something happen,” and that was based more or less on AMD and CommonJS, which is another specification that came on that. That became the de facto standard is node and require, which is this CommonJS approach.

Then it went up one more level and it said, okay, node while very large and very popular and able to drive this de facto standard, that's not actually the language. We need a language level solution for how we encapsulate code and allow it to load in different ways. That was when ES modules evolved. This is something coming down from the TC39 specification community, the ECMAScript specification, where this is how modules function in the language of JavaScript as defined at the spec level. It's no longer user space. This is actually baked into the language and accessible.

Now because it's the web and it's messy and all these other things, there's been all these back and forth about okay, what about file extensions and how does this change different things? Because many of those de facto standards around node had built up lots of tooling and magic around this. It would just work in lots of ways that turned out to not actually work when you try to do them in cross-environment situations, like how does this work in a browser as compared on a server?

There's a lot of stuff there that has really been getting hammered out and still coming in, but that's the high-level is ES modules are the language level. Now we're bringing it down into the actual specification solution to a problem that's been solved in user space for years.

[0:06:41.1] JM: A classic example of a module that we needed is jQuery, right? jQuery is this big blob of things that we need out of our JavaScript infrastructure. Historically, we would just import it on a global basis and we would have it available to our entire JavaScript application. That was not perfect, but it did the trick. What is wrong with that architecturally? Why is it problematic to have a global variable?

[0:07:15.4] KB: Why is it problematic architecturally to have a global variable? I mean, this is actually an interesting question, because I think that's a lesson that gets over-applied and it gets fought about a lot in the web world when you start talking about CSS, which is still in many ways global in different ways. The fundamental challenge with having things that are global is it's really easy to break them. If you're trying to pull in code from lots of different places that have perceptions about what this thing is and is it going to be there or not going to be there and can I manipulate it, if there's just one, then those things can mess with each other and break with each other.

Especially if you look at the JavaScript ecosystem today, the trend is towards lots and lots of small packages. The tooling around packages and package installation and dependency management is so good that people said, “Well, why have large packages when it's just as easy to do 10 small packages and then each one of those has a tiny surface area I can test and do things?” That ecosystem doesn't work if everything is global, because if I'm installing a 1,000 packages and each one is depending on something global that they can actually mess around with and mess up, very quickly I'm going to end up in a territory where one of those is expecting one thing, but the other one has already manipulated it in some way, and so it's not quite matching expectations.

If you're going to be integrating different pieces of code where you don't control everything, which is fundamental to modern software development in general, right? Old days you look back browser applications, they're relatively simple, they're small. The entire code base can be owned by one person or one team. Whereas nowadays, if you look at a modern web application, look at Gmail, or Facebook, or one of these really advanced products, they've got maybe half a million lines of JavaScript.

If each team that's working on that, that's probably spread across five, six, 10, 20 teams. If each one is able to map with these global variables, you're going to shoot each other in the foot real quick.

[0:09:17.9] JM: Just to revisit, what problem do ES modules solve?

[0:09:22.8] KB: ES modules solve how do I isolate code into its own package essentially, whether it's within a single application, or package, or an external one, and reliably pull that into my own application or into another package to use? It's the same problem that essentially gems – It's a little more complicated, because it's also how you do code imports. You could think of a package, an ES module, it's the equivalent of even just importing code, like it built into the language of Python, or Go, or something like that where you can import code from one fi le into another fi le, that literally did not exist as part of the language before. It was packed together with user space tools that would put that together for you, right?

It's this fundamental problem that most languages had built in from the start. You wouldn't imagine writing a Python application where you couldn't import code from one file into another file. That was all user space solutions. That wasn't part of the language prior to ES module.

[0:10:27.2] JM: Why have ES modules been controversial?

[0:10:31.4] KB: Well, partly because they're in the web. The web, it's probably the – I don't think it's out there to say that the web is the widest and most diverse set of software stuff that exists, right? It's you have things that are running in a distributed environment across every device known to man, you have no control over how this thing is running, where this thing is running, other than it's in the browser. People did all sorts of crazy stuff.

On top of, you've got server-side solutions of JavaScript with node where you have that more traditional environment, there were already solutions that existed there that had been built up, that are subtly incompatible with ES modules. You have the JavaScript language being specified by folks who are mostly concerned about the web. They're mostly people coming from your browser companies, thinking about that use case. They're trying to build something that is also going to be utilized on the server, where there are already subtly different approaches being used.

I think there were also just missteps along the way. There were ways of people trying to make them different than just being JavaScript. There is an introduction of okay, we're going to tell what's a module and what's not by having a different file extension. That was problematic, because in the node world where people were mostly thinking about packages, everything is just file extension. It deals with it for you. You don't even have to include it file extension, because node has all this magic about looking up, is it here, is it there and what have you? It created these situations where you had discrepancies from the way people were used to thinking about the world.

One other aspect that I think is worth bringing up here that is pretty interesting and I think also relates to why ES modules have been very controversial is in the JavaScript world, we've gotten used to using features before they are fully specified, because there is incredible set of tooling that essentially allows you to transpile within the language. You can extend JavaScript to add new functionality and transpile it back to older functionality. The tooling for this is called Babel. It's a transpiler. The original use case was okay, browsers are slow to update. In fact, looking back five years, many of them did not automatically update. A user had to go and actually do something to update their browser, which meant that even though JavaScript was moving forward and adopting features that were valuable in bringing it from being a toy language into a your first-world, or a high-productivity, extremely powerful language, you couldn't write code in that way and run it on those old browsers, unless you had a way to translate that new syntax and this new code back just something those browsers could understand.

The JavaScript world has gotten used to using features and compiling them back to older browsers. Now this meant that when people started talking about ES modules and they said, “Hey, this is a really cool syntax. We're going to do this.” They could use it before it ever got specified, or built into any platform. It was not built in any browser. It was not built into node at all. People started using it by using Babel to transpile it and using Webpack and similar bundlers to package things together.

However, they were doing that based on the assumption that, “Hey, we can just call it .js file, the same way we've called every other JavaScript file we've ever worked with and it'll just work.” Then the specification morphed and initially said, “Oh, no. You're going to have to have a different file extension for these modules,” because they're really – they're different from the old stuff. I think that was fundamentally a misstep, but why it was so controversial was because we as a community had already started using these things as a way that we assumed we would be able to. Then it turned out that that wasn't going to be quite right.

[0:14:22.6] JM: Well, one way that applications in the JavaScript ecosystem get condensed and presented to the end-consumer is through a bundler. Can you explain what a JavaScript bundler is?

[0:14:38.4] KB: Yes. This comes from a couple of interesting things. One thing to remember whenever you're talking about JavaScript is the number one target for JavaScript is the web. That means that any code that someone is going to run has to get loaded by their browser over an Internet connection, probably majority of web access now is probably through a mobile phone connection. Many of those connections are pretty slow.

There's long-band work on saying how do we make that amount of code that we're shipping out to the browser as minimal as possible? Especially when there was no equivalent of assembly language out there. I couldn't ship a binary. I had to ship actual JavaScript. Bundlers are the next step of going back a number of ways. We had this approach years and years ago where we would concatenate all of our files and minify them. What that would get you is hey, I'm going to have a single file that has to be loaded, so you don't have to issue a bunch of HTTP requests, which is less relevant now that we have HTTP2 and things like that, so it's not as expensive to issue more requests, but it's still a thing.

I'm going to concatenate it, so you only have to do a single request, and I'm going to minify it which is another one of these transpilation things, where any type of variable name or code name or whatever is going to get squashed down to very small letters, single-letter function names, etc., etc., and transformed into the – essentially as tightly as I possibly can compress this file. Then I'm going to ship that single bundle of JavaScript out to the web.

That's an old tradition. That's been around essentially, certainly as long as I've been doing web development, which is I don't want to say how long now. That's been with us more or less since the beginning of JavaScript, is we're going to put these things altogether, ship them out. Now in the old days when everything was global, you just had to make sure that you were putting those files in the right order, so that anything that depended on one thing happened after that thing was defined and put them all in a file and go.

In the new days, that dependency is more complicated. It's more complex in a good way. You have much finer-grained control of it, because you are importing modules, whether it's ES modules, or you're using AMD, or you’re using CommonJS and old-school node modules, you're pulling in code from all sorts of different ways in a complex dependency tree. We still want to take all of that together, smash it into maybe one file, maybe a set of files, but a small number of files and have it all work together.

A bundler is taking charge of that. At core, what a bundler is doing is it's crawling that dependency graph, figuring out what is the set of code that you're importing from all these different places that is needed to make this thing run, smashing it together into a file, which bundlers at a more advanced level may then split out into multiple files for different types of optimization; smashing it together, potentially minifying it, potentially doing other things on that, so that you have that blob of JavaScript that you can ship up to the browser.

[0:17:38.7] JM: How does a bundler fit into my workflow as a JavaScript developer? You've just given an overview of what purpose it is solving. How does it actually fit into what I am doing on a day-to-day basis?

[0:17:53.5] KB: Hopefully you can ignore it, because somebody else has set it up, because they are to this day a nightmare to configure, though there's been progress on that. Conceptually, it's the – if you're thinking about this from an old-school software development standpoint, it's your make file, or it's a piece of how you're making your project. You're writing your code, hopefully you don't have to worry about your bundler because it's already set up. Then when you're ready to run it, the bundler packages it up and ships it to the browser.

Typically, a set up will have a development mode, which is doing hot reload, so anytime a change is made, the bundler will rebundle and automatically refresh your page for you. Then when it comes time for deployment, it's just going to make it all up into your final packaged files and ship it out.

One concept that may be useful to talk about, there is the concept of an entry file. We're talking about this as a dependency tree. A tree starts at a single root node. What you'll do is you'll tell your bundler, "Hey, this is the entry point to my application, or this library, or whatever it is," and it will take that as its starting place, crawl down that tree and then bundle things up into a single file that is named predictably and you can customize that or configure that however you want, but that started at that single entry point.

In a common setup and a relatively simple setup, you probably have just one entry point. That is your top-level file. That's your app.js, or whatever it is. The bundler will crawl down that, package it up, you end up with a single app compiled JS that is then ready to run.

[0:19:21.3] JM: Considering we started with a conversation about modules, how do modules and the emergence and the rise in popularity of modules, how does that affect the ideal workflow that we would have with a bundler?

[0:19:36.9] KB: Chances are it doesn't affect them at all for you. Because if you're using a bundler, it's because you're already using some form of modules. In fact, chances are if you're writing JavaScript right now, you're probably even using something that looks like ES modules. If you're compiling up your code to be a single blob to send out, your bundler config is going to stay essentially the same.

The fact that we have ES modules shipping natively to the browser does enable us to do some things down the road as more and more of those modules exist and are supported, and you could see a world where we don't have to do that same level of bundling, because the browser handles all of those imports for us. Day-to-day right now, ES modules is a great way to structure your code. The bundler will keep working the way it's been working and I don't think there's any advantage to trying to move away from that.

[0:20:29.3] JM: When you say the browser could potentially handle the importing, what do you mean by that? What would that look like?

[0:20:37.3] KB: Yeah. Right now in a typical bundled setup, a JavaScript file is not loading other JavaScript files, unless you're explicitly writing JavaScript that says like, “Hey, go and fetch this file and add it to my HTML, or something like that.” It's not doing that dependency tree crawl. If you had an import statement, it wouldn't know what to do with it.

When you add ES modules, it now knows what to do with that. As long as that import statement is pointing to a fully qualified path, basically a URL, the browser can pull that, go and get it, come back and put it in place and run it as you would need it. That in bundler world, they're doing that for you; they're crawling it, they're packaging it up, so those import statements get compiled away to we're putting this code in this place and linking things up properly.

From a developer standpoint, chances are it's not going to make that much of a difference to you one way or another, which way you do it right now, except it's more of a pain to do it with the browser, because you have to do fully qualified paths and all of that. Whereas with a bundler, it can be smart and you can set up aliases and you can do all sorts of other smart things. I'm far from the biggest expert on ES modules in particular, but on JS Party, we just did an episode of that. We were picking the brain of one of the folks who is an expert in this area. The overwhelming message I came back from is don't worry about it yet.

If you're writing modules, if you're wanting to explore something new, go out and take your modules and make sure they’re ES module compatible and they're shipping that, if you want to explore writing tooling in that. If you're a line developer writing JavaScript, just keep using a bundler for now. It's not going to hurt you and it's going to work better right now.

[SPONSOR MESSAGE]

[0:22:21.3] JM: Being on-call is hard, but having the right tools for the job can make it easier. When you wake up in the middle of the night to troubleshoot the database, you should be able to have the database monitoring information right in front of you. When you're out to dinner and your phone buzzes because your entire application is down, you should be able to easily find out who pushed code most recently, so that you can contact them and find out how to troubleshoot the issue.

VictorOps is a collaborative incident response tool. VictorOps brings your monitoring data and your collaboration tools into one place, so that you can fix issues more quickly and reduce the pain of on-call. Go to victorops/sedaily and get a free t-shirt when you try out VictorOps. It's not just any t-shirt, it's an on-call shirt. When you're on-call, your tools should make the experience as good as possible. These tools include a comfortable t-shirt. If you visit victorops.com/sedaily and try out VictorOps, you can get that comfortable t-shirt.

VictorOps integrates with all of your services; Slack, Splunk, CloudWatch, Datadog, New Relic. Over time, VictorOps improves and delivers more value to you through machine learning. If you want to hear about how VictorOps works, you can listen to our episode with Chris Riley. VictorOps is a collaborative incident response tool. You can learn more about it, as well as get a free t-shirt when you check it out at victorops.com/sedaily.

Thanks for listening and thanks to VictorOps for being a sponsor.

[INTERVIEW CONTINUED]

[0:24:11.2] JM: JS Party is definitely a better podcast to dive into for people who are very serious about their JavaScript in there. If you want to hear a talk show experience for going into the minutiae of JavaScript and more introductory conversations as well, it's definitely a better podcast for that. Just channeling my own inner JS Party, for this episode I do want to continue down a just a list of things that I've been thinking about, or exploring in other episodes, or things that have come up in other episodes that we've done about JavaScript, or front-end development.

There is a term, JavaScript fatigue, or tooling fatigue. What does that mean? What are people fatigued about? Who is fatigued? Is this new developers, old developers, every developer? What are we talking about here?

[0:25:07.3] KB: Great question. This is coming from the fact that the JavaScript ecosystem is massive and moves faster than probably any other ecosystem I'm aware of. If you view our stats, you can look at for number of packages and different ecosystems and you can look at okay, how many PyPI packages can I install with Python? How many Ruby gems are there, whatever?

JavaScript is essentially an order of magnitude above everyone else and growing way faster. There are 500 new packages, do JavaScript packages added to the NPM registry every day. It is ridiculous. The language itself is evolving and evolving relatively rapidly. The language has, I think since 2015, there's a new spec published every year and it continues to make advances in progress.

What this means is that there's even more than in any software engineering job, there's a tremendous hamster wheel effect of trying to keep up, trying to keep up, what's new, what's different, what's new, what's different. The approaches that were modern and correct and the right way to do it two years ago are perceived now to be old-school and out-of-date and not there. Some of this is just perception. I just did an interview with somebody from Etsy and she said their mantra is, "We like boring tech." There's a lot to be said for that, right? You don't have to be using the latest and greatest, fanciest JavaScript framework and all of those different pieces to be writing good software, not in any way, shape or form.

There is this perception of constantly having to adopt new things, constantly having to adopt new changes and that's exhausting. There is some amount of truth to it in that the types of things that you can create and write now using a modern JavaScript framework are React, or Vue, or Angular, or even one of the more newcomers to the scene like Svelte, the type of application you can write the level of dynamic interactivity that you can create in a browser is simply worlds beyond what you could do five years ago with a mostly jQuery based application. It's incredible. You can write much more productively, because you have all this tooling in place.

There is some real need to learn and to grow and to adopt the new frameworks and technologies that are coming available, but so much of it is just perception, is this feeling of keeping up and you've got to keep up and there's a treadmill and you can't catch up. That creates fatigue.

[0:27:42.1] JM: There is definitely a sense that well, I mean, I think this is one of the reasons why there is this really, really rapid pace is there's a sense – there's almost a palpable future where front-end development is as easy as dragging and dropping, or as easy as a low-code tool, where you're building a user interface with the WYSIWYG and you're easily putting together these UI components. You're putting together basically a front-end application that does everything you need to do. If you wanted to dig deeper into it and optimize the performance of a particular React component, for example, you could do that. You would have no problem doing that.

We're not there yet. We are still in a time where the front-end developer has to do a lot of debugging and tweaking and typing of code. It's not a drag-and-drop UI experience yet. We still need separate roles for the designer and the front-end developer. Although, there does seem to be some palpable future where perhaps those two roles will intertwine and hard to know where that ends up. I think one thing we could discuss that seems to be hinting at that future is these component libraries, where it seems to be there's component systems that facilitate workflows between designers and front-end developers, these places where the designer and the front-end developer can mind-meld. Can you tell me the modern workflow between a designer and a front-end developer?

[0:29:31.0] KB: I can. First, I want to push back a little bit on the thesis here. Because I think there is this sense of oh, there's all this new stuff we can do that is making things easier. No front-end developer that I have talked to feels their job has gotten easier. In fact, there's a sense that it's getting more complex, because more and more things are moving to the front-end. It used to be that almost all of the front-end work that you were doing was presentational and you're doing most of your stuff with HTML and CSS and maybe a little bit of JavaScript, but all the heavy lifting and the logic of your application was living on a server. That world is long past.

More and more of the complex software engineering that's going on is moving to the front-end. You have logic that's moving there, you have even data management and things like that. You've got these complex state management systems, you've got Redux, you've got MobX, you've got GraphQL, all these different things going on to manage more and more of what used to live in the back-end in the front-end.

I think it is a little bit – there is a sense of, “Oh, my gosh. Drag and drop and no code is getting so powerful.” It is, but the result has been that we've continually wanted to do more. In fact, we've wanted to do more at a more rapid rate than the tooling has gotten better. The amount of complexity that has happened is happening on the front-end has simply skyrocketed relative to front-ends a while ago. I think it's good that we're having more tooling. The state of the art of what you can do without having to dive into code is going to continue to go forward. We've had no code web development systems for forever. We've got WordPress. We've got Squarespace. People have been able to build websites forever and even some amount of interactivity.

Whenever you want to go beyond to that, do something new and different, you need to get into the code. The amount that we've been wanting to do interesting things there on the front-end has far outpaced the ability of that tooling to catch up. The front-end teams at most companies I talk to are growing and expanding. More people are feeling the pain of we don't have enough skill on the front-end than I see on the back-end as well.

Coming back to the question about design and development and how those two things interact, it varies a lot by company, but what we seem to be moving towards is this concept of design systems and linked component libraries. Having within design a set of concepts, a set of specifications and standards, this is the typeface that we use, here are the font sizes that we use, here's the spacing, here are our components, here's how we're thinking about all these things, so that when a designer is working on something, they have a fixed library of tools to use.

This has been a thing in software engineering for a long time. We love our libraries. We love building out reusable bits that we can use over and over again and recombine in different ways. Well, that's coming to the design world. Then the translation of that into the front-end is often a component library, plus some additional styling stuff around typography and things like that.

When you've got that set up, or when you're moving towards that, a lot of that interaction comes back to this discussion of okay, what are the things that we have enabled in our design system right now? How do we fit the things that we want to do into that? If we can't, what's a way that we can extend that system to do what we want to do now and how does that play out in to our component library?

[0:32:54.5] JM: If you think about this future development workfl ows and I mean, who knows what it'll look like. If you think about just some high-level things that we have today, we have these driving frameworks. Actually, let's go deep on the frameworks right now and then we'll get to more futuristic things. I'd like to level set in the present and talk about the present-day frameworks.

To my mind, the main prominent frameworks to talk about are Vue.js and React. I think you still have Angular. You still have a lot of people in Angular. You have a thriving angular ecosystem. You do have Svelte. People are telling me to do a Svelte episode. I don't know anything about Svelte yet. Vue and React are really the two elephants in the room, as far as I can tell, and they're thriving, they're growing really quickly. Could you contrast the Vue and the React ecosystems for me?

[0:33:55.7] KB: Sure. First, let's talk a little bit about one thing that sometimes gets lost here is a lot of these frameworks have a lot in common. We on the front-end have more or less universally gone to a model where our development is organized by components and we think about the world in terms of components, which accept properties from your parents. You can think about that as arguments passed into a function. They maybe have some internal state, they maybe don't, and then they have child components. We're building these trees of components and the component is the fundamental organizing block.

Now if you look back to the JQuery days or things like that, that was not obvious as the development philosophy. That might have been true in the UI, but then the way you were organizing your code is different. That has been essentially universally adopted as the organizing framework for how we think about front-end frameworks and front-end JavaScript. React is doing that, Vue is doing that, Angular is doing that, Svelte is doing that, Amber is doing that, Dojo is doing that. Whatever framework you talk about, they’re pretty much – that's the approach they're doing.

The positive thing about that is no matter where you start on a framework, if you really dig in and start to understand how to do component-oriented development and how to think about your code base in that way, you're going to be able to take those skills of that understanding from you as you go from framework to framework.

Now to your question particularly about React and Vue, there's a number of differences both in the communities and in the way that the code was organized and thought about. I'm going to start from just how the projects are run. React is a project from Facebook. They have a heavily staffed internal team that are paid by Facebook to work on React. That heavily influences the direction of their development. They're very open about that. Dan Abramov is one of the most visible members of that core team. He says, “You know what? You all should know, we are making decisions based on what Facebook needs. The things that we're doing may not be optimal for you if you are not at Facebook scale or complexity.”

That I think shows up in lots of little ways, but it is something to be aware of. I think there's a lot of hype around, “Oh, this is the React way to do it. We're going to do it this way,” that actually does not scale down very well. It works very well at scale. Then when new folks are trying to do it, those practices actually don't work very well always for small applications.
There's a lot of, I almost want to say embrace of complexity in that ecosystem, where to a lesser degree than Angular, I think Angular actually goes even more in this direction of they have – The learning curve on Angular is really slow and long and it's a very complex framework with lots and lots of different interlocking pieces, because the mindset is enterprise and the people working on the framework and the primary users of the framework have these massive enterprise applications, where they are going to have all that complexity anyway.

Angular scales down very poorly to small applications and just individual components that you might embed in a regular vanilla website or things like that. React has a little bit of that problem as well. A lot of their time and energy is focused on the needs that Facebook has, which is fine and they're open about it. Though one of my wishlist items for the future is that they might put React out into a separate foundation, the way they did with GraphQL, so that it can be a little bit more responsive to the actual community using it.

Vue on the other hand from a development stance perspective, started as a BDFL style project. Benevolent Dictator For Life. Evan You created it. They are in the process of moving to being a completely community run organization. He's been essentially delegating more and more pieces and giving ownership. There's a well-established core team now with different areas of specialization and ownership. I think Evan You is still acting as a bit of a dictator there, but not nearly as much. They've adopted a very solid community RFC approach to adding new functionality and features and that has very much influenced the development of Vue3, which is supposed to launch any day now. That's the approach and development.

I think that actually ends up playing out a little bit in what you see in the communities. The Vue community feels much more bottom-up. It feels much more – how would you say it? Global and grassroots-driven. The React community is massive. There's lots of people. A lot of the very loud voices tend to be your more traditional tech voices. There's a lot of loud white men in that community, to not put it too bad. Whereas I feel when I've gone to Vue-related conferences, they tend to be much more diverse along gender directions, but also racial dimensions and also age dimensions. I found a lot more both young and old ends of the spectrum at Vue communities than I have the React, which just seems to be very traditional tech audience-centered.

In terms of approach using the frameworks, there are some ways that they are pretty different. React has very much embraced a functional programming approach. They try to do all sorts of things with having immutable state and everything is returning and you're trying to get to very functional ways of thinking about the world. Even sometimes when that's very confusing to users or to new folks, there's a philosophical approach their.

Vue has not approached that much as much. They've centered around this concept of reactivity, where you are changing objects, rather than having immutable objects that then get processed and return new things. You're changing objects and observing those changes and having things react to those changes.

Interestingly enough, React, though it has that in the word, does not use this concept of reactivity. They use state changes as you're doing some functional transformation then replacing the state, whereas Vue, you're changing an existing blob of state and things react to that.

[0:39:51.0] JM: Totally changing the subject rapidly, because I want to move through several different other topics and then get some perspective on where you think the future is headed. How would you defi ne the term JAMstack? How has the rise of the JAMstack affected front-end development trends?

[0:40:09.9] KB: Great question. First, just simple, what does the acronym mean? JAMstack stands for – JAM stands for JavaScript, APIs and Markup. The concept of JAMstack is essentially going further and further towards this concept of separated front-end and back-end. Many applications have started to be architected, where you will have a JavaScript application as your front-end and then some server application that is your back-end. All of the communication between those things is via APIs.

Contrast this to I don't know, 10 years ago when most web applications, your server was rendering your HTML and then you are also putting out JavaScript that might manipulate that on the client. The trend is towards this concept of a SPA, single-page application, where you have a JavaScript app that is doing things and maybe it's loaded all at once, or maybe a pre-compiles something so you can load a page that's HTML. You have this separated front-end from your back-end.

JAMstack is going way in that direction. It's saying, "Okay, let's forget that idea of having a server application at all." Yes, it will still be there, but we'll call it an API, or just APIs. In fact, a lot of times we'll try to use third party API, so we don't have to build those pieces ourselves. We're going to focus entirely on building this front-end of the application. What does that look like? Can we generate it all using JavaScript and markup? Then anything dynamic, we talk to the APIs.

Big picture, it's part of this trend towards pre-compilation. We've over time observe that one, it's often faster, especially it's more responsive as you go to ship a bunch of JavaScript out and then only once that you can cache at a CDN and then only ship data out from your core servers. That's one step. Oh, there's some slowness there, because that JavaScript still has to render a page that's not responsive right away. Well, what if you could ship, just you could pre-compile your HTML and strip your JavaScript? All of that's going out and then all the data is only loaded from APIs. We're pre-compiling more and more things. Maybe we could pre-compile all of our pages, or almost all of our pages and have those out, put them out on a CDN, so it's very close to whoever is loading it, have it load very fast. Then only go back as much as possible, or as when you need dynamic data. Only go back to an API, then everything else stay out on what they call the edge.

That's the concept that JavaScript, or JAMstack is pushing towards. It's saying, "Let's push things further out and let's pre-compiled more." One of the big frameworks in the JAMstack world is this framework called Gatsby, which is a React-based framework for generating websites and applications. The concept there is they're trying to pre-compile as much as possible. They want to pre-compile every page. Even though you might have your data loaded from a database traditionally, maybe you use a WordPress background, or maybe you have a traditional database exposed to an API, got a bunch of content there that's going to generate your pages, maybe product pages, or blog pages, or what have you, Gatsby is going to go and fetch that data not when somebody requests the page, but at compile time.

When you're shipping a new version of your application, it's going to go out, crawl through everything it needs to crawl through to get the data to render your pages, pre-render all of those pages and then you stick them up on a CDN, so that when somebody tries to load it, it's all there right away. It doesn't have to touch any database or processing to get there.

[0:43:41.5] JM: Got it.

[SPONSOR MESSAGE]

[0:43:51.1] 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.

If you want to get some additional help for your engineering projects, go to softwareengineeringdaily.com/g2i.

[INTERVIEW CONTINUED]

[0:45:40.1] JM: Continuing our trend of disjoint topics and then we'll bring some of these together, is WebAssembly impacting front-end development today?

[0:45:48.1] KB: Good question. When you say impacting, there's a couple ways I could interpret that. One, as your typical JavaScript or front-end developer, no, you're not thinking about it at all. However, it's already being included in ways that you're using and you're not aware of. For example, every web developer is using different types of dev tools and looking at when we're doing all that minification and compiling of things, you need tooling to be able to go back to what your original source code is, so there's these things called source maps that basically map from your compiled code to your pre-compiled code.

The tooling that is working on those source maps is written and shipped in WebAssembly, and so are a bunch of other libraries. I think your average JavaScript developer, that's going to continue to be there, or your average front-end developer, that's going to continue to be their interaction with WebAssembly is under the covers. They're going to use a library that happens to be implemented in WebAssembly.

What WebAssembly does for you though is it allows folks who are not traditional front-end developers to bring, or not traditional web developers to bring their applications to the front-end. For example, I talked with a guy last year who he has a gaming engine that he had built with Unity. It’s written in C++ and it was functioning – You would download this game engine, whatever, and just use it as an installed application. By compiling it to WebAssembly, he built a React-based front-end and suddenly, his entire gaming engine and your ability to write games was available in the browser.

That's where I think WebAssembly is going to impact the front-end. It's not for how it's going to change current front-end developers. For them, it'll most likely be hidden in libraries, just utilize, it's another tool in the tool chest, you import it the same way you would JavaScript. What it does do is it allows you to bring things that are traditionally not web applications, or people with skill sets that are not traditional web development skill sets and move them into the front-end.

[0:47:44.9] JM: Indeed. Impacting us in small ways today, almost inevitably impacting us in big ways in the indeterminate time horizon future.

[0:47:54.8] KB: One way to think about the web and browsers is the web is the most widely distributed and most popularly used delivery system in the world. It used to be that that was only available to documents. It was a document delivery network. Then we added applications, but only if you're willing to use the particular programming paradigms and languages, particularly JavaScript of the web.

Well, what WebAssembly does is say, “Hey, you know what? Screw that. Whatever programming language you want to use, you now have access to the largest app platform in the world, way bigger than any mobile app platform, way bigger than any desktop app platform. Everybody can use this and you can now access that from whatever language you want.”

[0:48:40.7] JM: Right. Define the term ‘micro front-end’ for me.

[0:48:46.6] KB: Micro front-end is –

[0:48:49.6] JM: You can put it in the context of Twitter.

[0:48:51.5] KB: I'll put it in the context of is something that I think your audience may be more familiar with, which is this idea of microservices. Microservices on the back-end have been a popular topic for a while, with lots of opinions on whether they're good or not. Essentially what they let you do is organize your code base as a set of independent services that can evolve independently, that don't have links to each other than via an API and whatever guarantees you have on that API.

That has the downside of dramatically increased operational complexity, but the upside of each one of these particular things is isolated, probably easier to test and more importantly, can be owned and evolved by its own team using their own technology choices without impacting the other services. Very valuable particularly in large companies, where you have lots of teams and synchronization costs between those teams is very high.

Micro front-ends is saying, “Hey, we can do that same thing on the front-end. Why not? We can have this team own this part of the front-end.” Maybe if I'm Amazon or something like that, I might have a whole team focused on the navigation. I have another whole team that's focused on the product page and I have another whole team that's focused on the shopping cart and all these different things. Some folks said, “Okay, well let's let each of those teams make their decisions and ship each one of those out as a micro front-end and then they can only interact.”

The challenge there is there's substantially more operational complexity there than there is in microservices, because one, you have to be really rigorous about what are the APIs, how do they interact each other? Two, because we've got all of these different frameworks and since everything's running in the browser, the frameworks themselves have to get shipped out and they have to go over the wire to the browser to run it. If you're writing micro front-ends in React and Vue and all these other things, anything that's running those micro front-ends needs those libraries out there.

In a microservices, I don't have to worry about your servers. It doesn't matter what's running on my server, because all that's coming back to you is the data from the API. Micro front-end, you can quickly end up – or environment, you can quickly end up in a situation where your friend then is loading hundreds of kilobytes, maybe even megabytes of JavaScript to run each of these front-ends.

Now there are other solutions. There are folks who say, "Well, we're not going to restrict you entirely, but we are going to restrict you to one framework, so they don't end up with that." Or they do some integration layer, where essentially they have a proxy in between that weaves together the front-ends, rather than loading them all from directly to the browser. It weaves them together rendered output in a proxy server of some sort and then ships that final version out.

There's lots of different implementation things, but the high-level there is it's trying to apply the concept of microservices to the front-end, but it turns out there's even more operational complexity in getting it right.

[0:51:46.9] JM: Is that to say that this is something that people don't actually use? This is just a Twitter talking point?

[0:51:52.6] KB: No. I think there are folks actually using it.

[0:51:54.4] JM: By the way, it basically being the idea like, let's say I'm a sizably large company. Let's say I'm Airbnb. I don't think everybody does this, but let's say I'm Airbnb, we want to give the developers lots of freedom and we want to give them the ability for Airbnb experiences, they can develop their system in Vue and Airbnb, the home sharing platform is developed in Svelte. These different teams can work with their different front-end systems and that's all hunky-dory, and that's the idea of the micro front-end; it's really the bring your own front-end.

[0:52:34.1] KB: Kind of. Yes, though let me clarify a little bit more. It is not that uncommon to have different sections of your application owned by different teams potentially using different frameworks. For example, an admin dashboard that's written in Vue, but a customer-facing experience that's written in React. Or even an old customer, I was chatting with somebody from Etsy and there – I believe it was their internals have been rewritten with React for a long time, but the customer facing stuff is all still being rendered by PHP and then there's some jQuery that does stuff, right?

That's not that uncommon divided by product area and it's often not that bad, because you don't have – often, the same people aren't using the same – all those things at once. Maybe it's divided by a user type. In Etsy’s case, might be customers versus shop owners, or even just when I go to a page, it's going to be a brand new page refresh, but that's okay, because it has to load this new library.

Where micro front-end I think really starts to talk is when you're talking about that subdivision at the component level, so I have a single page with a dozen different components on it and this component is owned by this team and that component is owned by that team and those things can evolve independently, that's where it gets really complicated in terms of allowing fully different frameworks and you have to be really thoughtful, or you'll quickly end up in a situation where you have catastrophic levels of JavaScript going to the browser.

I think there are people trying to solve this, because there are some benefits to being able to isolate things in this way. I think most of the people actually using this still put a restriction. They say, okay, you can own a lot of things independently and we're going to stitch them together and we're not going to have them as part of the same codebase maybe, but you still have to use React. They at least eliminate the multi front-end loading lots and lots of different front-end frameworks on the same page load.

[0:54:26.1] JM: Okay. There's a whole bucket of other things that I didn't get to explore with you that I wanted to, things like machine learning and Tailwind CSS and GraphQL and so on, and maybe those will have to wait for some future conversation or something. One thing I really wanted to get your perspective on, just because it's come up in a lot of recent episodes is how we do get to this drag-and-drop world, assuming you believe this is a reality. It is happening to some extent in the low-code environment. To me, this seems a very important trend.

I mean, I think some people who listen to the show think I've latched on to a brain virus of the low-code, no code stuff. To me, it seems this vision of WYSIWYG software development is finally coming to fruition in some sense. It's hard to know how immature it is. It's hard to know where I stand exactly how widespread the use is. The biggest thing I'm curious about that I feel we can actually discuss today is what bridges the gap between these two ecosystems, the low-code, drag-and-drop, perhaps in many cases proprietary-based interface builder ecosystem, versus the build your own from the ground-up JavaScript-based classic framework world? How do we bridge the divide between those two ecosystems and what is the nature of the divide as it stands today?

[0:56:07.5] KB: Yeah, that's a really interesting question. I think one way to think about this is rather than thinking about it as there's a no code world and there's a code world, think about it as increasingly powerful abstractions and increasingly powerful tooling. The amount of stuff that we can get out of the box today, whether it's front-end or back-end for doing development is astounding. Both as a developer looking at tools, like React or Vue on the front-end, or looking at managing servers on the back-end with Kubernetes, or not having to manage servers on the back-end because you use a platform as a service like Heroku or things like that, the amount of stuff that I used to have to worry about that I don't have to worry about at all is shocking.

That is playing out on the UI builder side as well. UI builders have gotten more and more powerful, more and more integrated and able to do more and more things. I think those trends continue to rise. The question is where is the line where the majority of situations that we are wanting to accomplish are below that line of it's already been solved, it's automated away, I just have to wire things together.

I think actually, JAMstack is an interesting driver of that. It's pushing things in that direction, because it's pushing, it's creating an ecosystem where for example, Zapier is a viable company, where I can build a business that is only about creating drag-and-drop relationships between third-party services. I can do that, because there's enough demand from people building UIs to do that, or there's enough demand from people who are building things on WordPress, or who are essentially doing no code right now, that want to do things that traditionally you'd have to write code to do, you'd have to run a little server somewhere. Well instead, I'm just going to wire them together with Zapier and it'll just go.

I don't think it's a line in the sand, before this point we're going to be coding and after this point, we're going to be no coding. I think it's just saying, what is the level of work that we need to be doing? What is it that we're trying to accomplish and how much of that can be done without writing custom code?

I don't know where the line is today, because I'm actually – I don't honestly care that much about no code, because I like coding. I think that code has come very far. I think we're going along those lines when we talk about building a bespoke application from the ground-up, there were zero people doing that today. Functionally, equivalent to zero, because everyone's building on top of a framework like React. Or they're building on top of, even if they're building on top of jQuery, right? There's this massively developed piece of tooling that has had thousands of hours of time put into it that makes your life easy now.

The typical JavaScript application that you ship may touch or use either in the actual code, or as part of the build process over a thousand independent open source packages. You install the application template essentially from create React app, which is here's a standard template for building your React application and it installs a thousand packages. Most of those for things that get built are part of the build system compiled away, but it's touching all those things. That's all software. You didn't have to build. Conceptually, it's no code, right? If no code just means code, I don't have to worry about it.

Yeah, I think we're already getting there in many ways and people who are doing application development right now are sitting on top of the same foundation of the people who are using drag-and-drop platforms to create applications. It's the same stuff. It's open source packages that are and available API frameworks that interact with each other that anyone can access.

[1:00:04.1] JM: All right. Well, just abbreviated discussion at the end here about software podcasting. Podcasting as a way to explore and disseminate information about software engineering. You and I both do this thing. How does this fit in to the educational path of a software developer? I mean, we all know that you can only succeed as a software developer if you're continually learning. I mean, I guess it does depend on your definition of success, because I mean, you can just learn one paradigm of software and just maintain code bases in that paradigm for the rest of your life and have a very good living.

I guess, you don't necessarily have to continually do the reinvention thing, but certainly most people make some habit out of it. What's your perspective on the software podcasting medium and how does it look going forward? Is it a durable medium?

[1:01:10.7] KB: Oh, absolutely. Absolutely. I'm not just tooting my own horn there, because I obviously do have a little bit invested in podcasting doing well. For me, podcasting gives you as a listener, scalable access to the type of content, or information that you would otherwise have to go to a conference to get. We actually on JS Party, we did an episode, amusingly label the wonderful thing about tigers. We did this episode on learning and how we learn about different things.

One of the things that came up there that I thought was fascinating was different mediums help you learn at different levels. Going to a conference is great for learning what you should learn about. It's great for getting you excited. It's great for inspiring you. It's great for getting a big picture. It's really bad for digging into the nitty-gritty details and building out a technical skill. I would put podcasting in that same bucket. Podcasting is a wonderful way to discover what you should be learning about. It's a great way to get a sense of how people are thinking about things. It's a really bad way to learn a particular tactical skill.

I think that piece, especially if we look in a world that is so incredibly filled with abundance, I mean, look at what I mentioned in terms of JavaScript, 500 new open source packages every day, figuring out what you should be learning about is a huge part of the challenge of learning. What should I learn? Podcasting is a scalable way to disseminate that same type of curation, that same type of inspiration and that same type of access to what are the brightest people in the field learning about, thinking about, speaking about in a way that you can listen to in your pocket as you go for a run. You don't have to travel halfway across the world to go to a conference on. You can listen to from anywhere.

[1:03:01.2] JM: Kevin Ball, thank you for coming on Software Engineering Daily. It's been great talking.

[1:03:04.1] KB: Absolutely.

[END OF INTERVIEW]

[1:03:14.5] JM: As a programmer, you think in objects. With MongoDB, so does your database. MongoDB is the most popular document-based database built for modern application developers in the cloud era.

Millions of developers use MongoDB to power the world's most innovative products and services, from cryptocurrency to online gaming, IoT and more. Try out MongoDB today with Atlas, the global cloud database service that runs on AWS, Azure and Google Cloud. Configure, deploy and connect to your database in just a few minutes.

Check it out at MongoDB.com/Atlas. That's MongoDB.com/Atlas. Thank you to MongoDB for being a sponsor of Software Engineering Daily.

[END]