EPISODE 1020

[INTRODUCTION]

[00:00:00] JM: ReactJS developers have lots of options for building their applications, and those options are not easy to work through. State management, concurrency, networking and testing, these all have elements of complexity and a wide range of available tools. It's hard to know what to do. If you take a look at any specific area of JavaScript application development, you can find highly variable options.

Kent Dodds is a JavaScript teacher who focuses on React, JavaScript and testing. In today's episode, Kent provides best practices for building JavaScript applications specifically in React and he provides a great deal of advice on testing, which is unsurprising considering he owns testingjavascript.com. Kent is an excellent speaker and he's taught thousands of people about JavaScript. So it was a pleasure to have him on the show. Kent is also speaking at Reactathon, which is a San Francisco JavaScript conference taking place March 30th and 31st in San Francisco, and this is the last show in a week of interviews with different speakers from Reactathon. If you hear something you like, then you might want to hear more at Reactathon. You can also hear more podcast episodes about React by listening to the Reactathon Podcast, which is available at Reactathon.com/podcast.

[SPONSOR MESSAGE]

[00:01:29] 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]

[00:03:19] JM: Kent Dodds, welcome to Software Engineering Daily.

[00:03:22] KD: Thank you so much. I'm super happy to be here.

[00:03:25] JM: You've been part of the JavaScript ecosystem since before React won out as the most popular frontend framework. Did you take any lessons away from the framework churn on what causes an open source library to succeed over other rivaling technologies?

[00:03:44] KD: Ooh! That's a very interesting question. So just for a little bit of background, I joined the JavaScript ecosystem and actually the software developer ecosystem around the time that backbone was kind of the thing that most people were using, and then AngularJS started to become a thing. I think I had been doing JavaScript for about a year before AngularJS started to become the new hotness, and then it was another year and a half or so after that that React started to become pretty interesting to me.

In experiencing that first hand, I think that there are couple of things that kind of helped React win out as it has today. One of those would be that AngularJS has at least at the time and even still today, there is a huge amount of complexity within that library because it supports so many

things, whereas React is this so much smaller and simpler. It's a smaller slice of the frontend world. Conceptually, it's a fair amount simpler. On top of that, it also embraces JavaScript as the language for the platform upon which you're building, and I know the Angular people listening right now are like, "No. No. You're using JavaScript all the time, like Typescript and whatever."

But what I mean by that is the template DSL that you have to learn to use many of the other frameworks, you don't have to learn that to use React effectively. In fact, you are better off with React if you know JavaScript well. So that's one thing that really appealed to me personally when I was making the switch from Angular to React. When I was switching off of Angular, I was leaving behind a huge amount of knowledge that I had accumulated over the years because that was just not transferable at all.

That helped, I think in general, and just like as a general principle, React being a simpler solution for building UIs. I think the way that the component model came at just the right time, AngularJS was still all about directives and controllers and stuff like that where React embraced this component model making it really composable.

The really nice thing about that component model in general is you can have this really complex component that could be all inside of this black box and it really could be a tight sealed black box that the outside doesn't need to know anything about, and that just makes it easy for you to say, "You know what? This part of the code is super complicated. It could probably be made simpler. It could probably be made better." But the complexities within that black box do not leak outside of it. So I can still use this thing. It's still useful to my application, but it's not going to impact the simplicity of the rest of my application.

I think, yeah, a combination of that. Then also, we can't discount the issues at the time when AngularJS was moving over to Angular 2, and that was basically a big rewrite for the framework. Reacts itself has actually undergone a rewrite. Ember has undergone a rewrite, but the API that's been exposed was consistent for those, whereas with AngularJS to Angular 2, it was just so different. In every single way, there is no automated tool you could use to upgrade an entire codebase. So that I think contributed big time.

For me personally when I was working with AngularJS and starting looking into Angular 2, I was actually advising the Angular team on some things with how you do forums, because I was a big maintainer of the a forum library with AngularJS and I was giving talks at conferences, like this is what you have to look forward to. I just saw that as like this is basically – They could just renamed this thing, because it is so different from the original and it was going to be a huge burden. My company wasn't interested in rewriting to Angular 2 and I know many others are still on AngularJS.

So with the combination of all of those things, when it came down to it people are like, "Well, I can't stay on AngularJS. I need to move to a different framework to stay on the latest of things." The transition to Angular will be just as complicated or maybe even more complicated than the transition to React. I guess now I just evaluate them on a level playing field and I decide that React fits my mental model better than what Angular 2 is going to do, and also the fact the Angular 2 just took such a long time to come to production. A lot of people had already moved over from AngularJS to Angular – Or to React before Angular 2 is released. I did.

I think, yeah, lot of reasons that React kind of came out on top and some of those are subjective, but I think what is objective is React now is definitely the most popular frontend framework for the JavaScript ecosystem and not –

[00:08:36] JM: Do you feel like you're all-in on React at this point or do you try to keep up with Vue or Svelte or whatever else?

[00:08:44] KD: Yeah. I'm definitely all-in on React personally and professionally, but I do try to keep tabs on what's going on in Vue and in Svelte and whatever else is coming up next day. I think any software engineer who doesn't want to get left behind with flash or something would find that it's wise to keep abreast of what's going on in the overall community, for sure.

[00:09:11] JM: React has been around for I guess 6 or 7 years at this point. Really popular probably for 5 years. What are the most notable changes that have come to React since it came out?

[00:09:24] KD: Yeah. As you said, React was released 6 years ago and there have been some changes to the framework. I think maybe one of the most notable was the change from react.create class to classes, and that changed like just how you create these components. That was pretty significant, but the cool thing about that was the API itself was so similar that they were able to write an automated tool that would just automatically update all of your code, not necessarily from a create class API to classes, but to a separate package that you could add and still maintain support for your old components and then write all of your new components as classes, which is what Facebook does.

Actually, that's one thing that I really like about React. As much as I'm not a huge fan of Facebook at all, either as a company or even of a platform for my own personal use, the fact that React is being built to support the needs of a platform like Facebook is really awesome because it means that if the React team says, “Hey, we need to make this big change. We want to move from react.create class to official classes because that's where the web is going,” then they have to consider the 50,000 or 100,000 other components that are built within Facebook before they can make a change like that. So they either have to write an automated tool to make that migration really painless. Actually, I believe at Facebook, if you're going to make a breaking change, you have to be the one responsible for updating the code that you're going to break. So it behooves them to make that as easy as possible.

Anyway, that was one change that happened. It was really easy to upgrade that. Then another change that is pretty significant is the Hooks API that was released about a year ago, and that allowed us to do pretty much everything that you can do with React component using just regular functions and it really kind of changed the way that you think about components and lifecycles in a way that makes it harder for bugs to pop up, at least like maybe it takes a little bit more work when you're developing to make sure you get it right but you end up shipping for your bugs to production, which I think is a good thing.

Then we’ve got another change coming up pretty soon with how asynchrony works in Reacts, and that'll be a pretty significant change. But all of these changes totally have a really nice migration path or they’re totally backward-compatible. Facebook still has components that are years and years old written in the very early days of React that are their still being run on

production on the latest version of React today. I don't know of any other framework that can say that and I just really appreciate that about React.

[00:12:07] JM: React has this recent changes you mentioned called Suspense, which improves the experience for asynchronous loading. Explain how Suspense works.

[00:12:17] KD: How much time do you have? No. I'm just kidding. Yeah. Suspense, it's an interesting idea around how allowing the users of React to queue React in on when it's okay to do rendering or when you'd rather wait to render certain things. It's kind of complicated and a little hard to do over audio, but there is a really great talk that Dan Abramov gave about two years ago at JS Iceland that people should take a look at, for sure, because it explains the use cases in a way that I haven't seen anybody else explain it any better since then.

The basic idea here or the problem that we're trying to solve is that the web is inherently asynchronous. Not only like waiting for buttons to be clicked or the user to do something, but also the network. When the app gets loaded, we need to go make a request. The user clicks on this button and so now we got to go request more data and we can't show the user – We can show the user a loading screen, but we can't actually show them anything useful until that stuff comes back. But then you have all these problems where you have a flash of loading state because maybe we make the request, we show the loading spinner and then, "Oh!" 100 milliseconds later or 200 milliseconds later, that request comes back, and so we render the final version. That's what we call a flash of loading state and that makes your app feel slower than it actually is.

In that scenario, it'd be better to not respond to the user at all. They click on it. You just wait a hundred milliseconds and, "Oh! The stuff is back. We'll just render it out." So then the app feels slower. It may have taken a hundred milliseconds to actually respond, but it actually feels faster because by the time it responded, it was actually the real data.

There are a lot of complexities like that. It's not just for that, but it's in general managing asynchrony from a user experience perspective as well as the code and the developer's perspective is tricky business. So what Suspense allows you to do is allows you to create these boundaries around areas of your React component tree to say, "From this Suspense component

on down, if any of these components says that it's not ready," we call that suspending, "so if any of these components suspends, then I don't want you to do anything with the rest of this tree. Just wait for that component to say that it's ready to go." There's an API for that. It's pretty reasonable. Then when it's ready to go, then we continue to render that part of the component tree. If it takes too long for that to happen, let's say it's not 100 milliseconds, but it's like two seconds to get that data, then you can provide a fallback to say, "Hey, if this does take a little bit too long, then render this instead. Maybe it's like a skeleton UI or something like that. Try to make it look at least it's partially loaded or something or show it some indication to the user that work is happening."

Then once that suspending component has received its data, then it informs React, "Hey, I'm ready to render." React will re-render that part of the component tree and then the user sees the final results. Today's like current way that we do this is by managing all of that state inside of our component. We have some loading state. We have error state and all of these different states that we manage. Thanks to Hooks, like this is actually pretty simple to abstract. But at the same time, it's harder to coordinate these types of loading experiences so you don't just wind up with 12 loading spinners when the user lands on the page, which also will make the user feel like the app is slower than it really is.
Suspense, the whole idea around it, is just making asynchrony a better experience for the user and a better experience for the developer who is trying to marshal all of these to coordinate well together.

[00:16:07] JM: Could you describe in more context how a hook interacts with the Suspense mode?

[00:16:12] KD: Yeah. Sure. One thing that I'll mention before I jump into that is that suspense is actually a supported feature today and it's been around for over a year, and that supports lazy loading of codes. You have this react.lazy API where you can use a dynamic import. So if the user goes to a login screen, they just get the code for the login screen and then when they go to the settings page,then we dynamically load that could later. So it helps with initial page load performance.

Suspense has been around to support that use case for over a year. The Suspense mode that we're talking about is suspense for data fetching in combination with what's called concurrent mode in React and using those two together is what gets you all these extra superpowers. How Suspense interacts with hooks is actually they're pretty unrelated to each other.

You can use Suspense with class components just fine. There's really nothing about the two of them. The only relation that they have is that you can have a custom hook that is responsible for suspending a component. The way that that works is this is actually – I get the similar response too when people see GSX for the first time. They're like, "Wait, that feels very wrong." But this is how it works today and it's probably going to work that way when it's actually stable.

But inside of your component, before you return your GSX while you're in that render phase, you create a promise or you take a promise that's been created already for making this fetch request. That's preferable. You don't want to create promises in your render, but you reference promises that have been created and you say, "Oh! That promise hasn't actually resolved yet. So I'm going to throw that promise." So it's like throwing an error. You say throw that promise. That throw will be caught by React and I will say, "Oh! You threw me a promise. So I'm going to wait until this promise resolves before I try to re-render this component again."

How that interacts with hooks is that you can put that promise throwing inside of a hook and that abstraction or that hook abstraction can hideaway the implementation detail of actually throwing a promise, because nobody actually wants to do that in their components and nobody will. That will all happen within abstractions especially like data management libraries or fetching libraries and stuff.

As far as the developer is concerned, they're looking at this code and they don't see anything in there that indicates that this is happening asynchronously, which actually makes that code way easier to work with. Then the user experience is improved because they're leveraging all of the research that the React team has done to create this concurrent mode plus suspense API. They don't really necessarily interact, but you can use them together to do some cool things like that.

[SPONSOR MESSAGE]

[00:19:08] JM: Apache Cassandra is an open source distributed database that was first created to meet the scalability and availability needs of Facebook, Amazon and Google. In previous episodes of Software Engineering Daily we have covered Cassandra's architecture and its benefits, and we're happy to have Datastax, the largest contributed to the Cassandra project since day one as a sponsor of Software Engineering Daily.

Datastax provides Datastax enterprise, a powerful distribution of Cassandra created by the team that has contributed the most to Cassandra. Datastax enterprise enables teams to develop faster, scale further, achieve operational simplicity, ensure enterprise security and run mixed workloads that work with the latest graph, search and analytics technology all running across hybrid and multi-cloud infrastructure.

More than 400 companies including Cisco, Capital One, and eBay run Datastax to modernize their database infrastructure, improve scalability and security, and deliver on projects such as customer analytics, IoT and e-commerce.

To learn more about Apache Cassandra and Datastax's enterprise, go to datastax.com/sedaily. That's Datastax with an X, D-A-T-A-S-T-A-X, @datastax.com/sedaily.

Thank you to Datastax for being a sponsor of Software Engineering Daily. It's a great honor to have Datastax as a sponsor, and you can go to datastax.com/sedaily to learn more.

[INTERVIEW CONTINUED]

[00:20:47] JM: Could you distinguish between what Suspense is versus what concurrent mode is?

[00:20:53] KD: Sure. Concurrent mode is something that the React team has been researching for an extremely long time and only in the last – I think it was like two years ago when React 16 was released. That's when concurrent mode or that the code necessary for concurrent mode was added to React. That was a complete rewrite of the React framework to support concurrent mode and it was called React Fiber, that rewrite, but now it's just React.

The idea behind it is performance – The big secret to improving performance is less code. You either load less code or you run less code, but that's how you make any software faster is less code. It may not be less characters. That's not what I'm talking about. It's just home how much time is the computer taking to execute your code?

So because JavaScript is single-threaded, the time that the browser is taking to execute your JavaScript code is time that it can't have to update the browser or what the user is looking at as the user interacts with the app. So let's say that you have an interaction that takes 500 milliseconds because it's just running so much JavaScript to make that interaction happen. During that 500 milliseconds, the user is unable to interact with the application at all. That is just a really unfortunate scenario, especially for users on mobile. They experience this a lot more.

So what the React team discovered is, "Hey, sometimes you can absolutely reduce how much code you're running and like there are optimizations you can make as the developer who's creating the application," but sometimes you just really have to run all of that code. There's a reason that code exists. But if you can run some of it now and then some of it later and then the rest of it at another time, you just split up that work. Then you can let the browser take a break from the JavaScript code and then let the user interact with the app. When the user is done, then we jump back in with that work we are previously working on. So that's what Fiber was all about. That's what concurrent mode is all about, is it says, "Hey, I'm going to –" React says, "I'm going to run all of this code, and every now and then I'm going to just check, like, "Am I ready or can I continue to run code or do I need to yield to the browser and just pause what I'm doing?" Like sometimes it's like, "Oh, I finished all my work. I don't need to yield anything," but sometimes it needs to yield back to the browser because something it's doing is really heavy. It will yield to the browser. Let the user interact with the app, whatever, and then go back into the JavaScript code.

Now this happens very, very quickly. But what the end result is for the user is an app that feels so much more snappy, and there are certain like interactions that the user can do that can jump ahead of the things that React is doing as well. They built out this scheduler and that's actually an npm package and they want to build this into the browser so that other frameworks can take advantage of this and they're actually working with standards committees to make that happen. But it allows you to sketch a work to say, "Hey this is really high-priority. Even if the user is

interacting with this, we really need to make this JavaScript happen," or "At this network request that came back, it's not super imperative that we jump ahead of the user typing into this field. So we'll go ahead and let them type. When they're done, then I'm going to continue to do my work."

It's just this really fascinating scheduling capability that React has built into itself, and that's what concurrent mode is all about. It's enabling developers to utilize that scheduling capability that's now built into React. Today, with the way that React is shipped, you can use concurrent mode with the experimental build of React or you can just use synchronous mode. I don't know if that's what they're referring to it as, but normal mode, and both of these are supported today. Eventually, I think concurrent mode is going to be the only thing.

[00:24:47] JM: You touched on an element of web development that I want to get your perspective on. JavaScript is fundamentally single-threaded, and the browser is often switching between the user's application code and painting the elements that are being loaded on the browser. I just like to get your perspective for how React and modern JavaScript simulate the experience of working in a multithreaded environment versus the realities of JavaScript being single-threaded.

[00:25:28] KD: Yeah. With concurrent mode, because React is a framework, it's calling into your code as the user of React. So you'll create these components and you'll hand those over to React, and then React will call your functions. Then when your functions are done being called, then React will continue. Every single one of your function components, React is going to be calling into, and then that function component will return the React elements that it needs to render.

Because React is in control of all of this, then it's able to do these checks to say, "Okay, based on how long I've been running right now, I have X-amount of milliseconds left before I need to yield to the browser, and so I'm going to go ahead and call this next piece of user code." Then when that user code comes back, then it does another check, "Okay. Now, I need to yield to the browser, because I don't have time to call another piece of user code."

I'm not on the React team, and I've tried to read the source code and I understand some of it, but there may be some nuances here that I'm missing. But the basic idea is because React is in

control of calling into your functions, it's able to do that kind of management of calling into or of yielding to the browser when it's time for the browser to do some work.

[00:26:48] JM: You are a full-time instructor, and I want to ask you some about that a little more in the future. But you spent a lot of time interacting with people that are learning JavaScript and learning React, and I think one of the things that confuses people is React state management. This is often – Were almost always done around Redux. Can you explain some of the common misunderstandings that people have around React state management?

[00:27:17] KD: Yeah, that's a great question. I think that maybe two years ago, we could say that application state management was done with Redux – Or like mostly. But in the last few years, Redux has fallen out of favor quite a bit and I don't think that it's the de facto standard anymore. There are a few reasons for this, but first I'd like to talk about why Redux took off so much. I'll preface all of this by saying I used Redux only one-time in a production application just enough for me to realize that I didn't like Redux at all. Not Redux the technology, but Redux the – I guess what Redux encourages developers to do, which I think is not good, and we'll get into that.

The Redux came at a time that we call the Flus wars, where Facebook came out with the specification. It was basically a talk in a document that said, "This is how we manage state in our applications to avoid certain inconsistency bugs." So a ton of different libraries came out. I even built one actually for managing state in this kind of architecture.

Eventually, Dan Abramov was scheduled to give a talk at a conference and he wanted to do this time travel debugging thing, and so he built a little flex implementation that would allow him to do this kind of based on The Elm Architecture, and the talk went super well and everybody asked him how they can use this project that he used to make that happen.

He'd open sourced it and people started using it, and that's when Redux became super popular. But it didn't just become popular because of a talk. IT became popular because it solved the problem. The problem is if we go back to AngularJS, we have dependency injection, and that's how we get different services and data around to all of our components. It's just this single name space that's available everywhere in the application.

With React, you don't really have that. Instead with React, you have this explicit passing of props from one component to another, and that actually works out really nicely, because with dependency injection, you have all of these global name space that's implicit and you can register things later and it just can be a bit of a headache. It's kind of like global namespace on window.

But with React, you're doing this explicit prof passing, which makes it a lot easier to track, “Okay, where is this thing coming from? Well, it's coming from the parent component.” The problem is that when you have a tree that's of sufficient size of like a regular production application, you're going to have some trouble with passing these props down like six different layers of components where you’re passing it from component A through component B, C, D and E where component is the only thing that actually needs that promt. So B and C and D, they don't care anything about it, they’re just forwarding it along. So then you start moving components around and you have to make sure you're passing the right prop, “Oh, I don't know if I need that prop. Maybe I do. So I’ll just keep doing it,” but you don't. So now you have this extra prop you’re doing all this extra work for for no reason.

We call this prop drilling, and it is a bit of a problem. Even though it's nice that you can track exactly where things are coming from, it’s problematic because it just takes a lot of work to maintain. Then eventually you do things like spreading all of the props across these components and that can lead to some problems as well. Prop drilling can be a bit of the pain.

What Redux did was the integration with React is it used this API called context that has been supported in React for a very long time. But during that time, it had this big warning on it that said, “This API will change. Do not use this.” So that freaked everybody out. They’re like, “Well, I'm not going to use that thing.” But Redux did and so did React Router and lots of these popular like theming libraries and CSS and JS libraries all use this API that was eventually going to change, and they just said, “Well, I'll deal with it when it changes. But right now it solves this problem of getting data from component A all the way down to component E without having to pass it between these other props in the hierarchy.”

This context API is what Redux used to be able to get your data from the top of the tree down to the bottom of the tree in a way that was really easy and it used these higher order components to make that really easy to do. That the problem that Redux solved wasn't necessarily unidirectional data flow and time travel debugging and stuff. It was actually just making it easier for developers to get their data to the components they needed without having to go through all these layers of prop drilling. That's why Redux became popular, because it was the first real like reasonably simple way to do that without using APIs that the React team told you not to use.

In the recent years, the context API got some love from the React team and they made it official and they made it actually quite nice to use. Now you can actually start solving that same original problem without Redux and in the process you sidestep all of the issues that you get with redux. Let me talk about some of those really quick.

Redux in itself is not necessarily a bad thing. The point is that you want to keep your state as close as possible to where it's being used, because that makes it easier to move components around. It makes it easier to delete state that is no longer in use. It makes it easier because you don't have to open up like 30 files to find the flow of data in your application. In general, Redux made it simpler for people because they found that if I needed to share data between two sibling components, I'd have to keep on moving it up and up and up the tree until eventually I'm at the very top of the tree. So just putting into Redux, it makes it a little bit easier since I don't have to prop drill all over the place.

The problem is that when people are so used to using Redux to manage state in that way, they eventually start putting everything into Redux. So now you have your form error state in Redux. You have every keystroke of the user in these form inputs going into state. You have the, “Is modal open state like happening in Redux as well?” You have to start name spacing this stuff because it's in this giant Redux store.

So this becomes a problem, because now when all you have to do is update a component that has a checkbox that says whether or not it should show a certain message or something, now you have to open up like three or four different files that are who knows where to manage that simple interaction. Not only that. When you open those files and start making changes, the changes that you make could have a completely unknown impacts on the rest of the codebase.

You have to – All of a sudden, instead of the nice component model that we love from React where it's just this black box and it doesn't impact anywhere else in my app, now, literally, every single component has wires connecting it to every other component in the application.

That's what I don't like about Redux. It's not the technology in itself, like if you're being responsible about what you put into the Redux store and the state that you put in there actually should be global and does care about the other state that's in there, then yes, that's reasonable. But, literally, every Redux implementation that I've seen in the wild just puts everything into Redux and it has these problems that I've talked about.

Anyway, as far as like state in general and like how you manage state today, I actually have a lot of thoughts on this, and all I'll stop talking here in a second. But I think one of the big problems that we have just when we're talking about state in React or any frontend system is that we kind of combine all state, all data that can change overtime, which we just call state, into one category, when actually there are many different categories of state in our applications. The stuff that comes from the server, that's actually not state. That's a server cache and it's something that you can't reliably expect to always be consistent. So we cache it on to the client so we can display that, but you actually have no way of really knowing if that's the state of the server. So it's not state. It's actually a cache, but we combine that with the application state like our modal is open, or the menu is open, or we've got the user checked this checkbox. That's UI state and that's different from our server cache.

When we combine it all together, it makes them both more complicated. But if we can logically separate those two, then all of a sudden our application state management becomes a lot simpler. This is something that I've been thinking about a lot recently and I'm still kind of working through my thoughts on it, but those are a bunch of words for you to think on.

[SPONSOR MESSAGE]

[00:36:08] JM: When you need to focus on building software, you don't want to get bogged down by your database. MongoDB is an intuitive, flexible document database that lets you get to building. MongoDB's document model is a natural way to represent data so that you can focus on what matters. MongoDB Atlas is the best way to use MongoDB from the company that

creates MongoDB. It's a global cloud database service that gives you all the developer productivity of MongoDB plus the added simplicity of a fully managed database service. You can get started for free with MongoDB Atlas by going to mongodb.com/atlas. Thank you to MongoDB, and you can get started for free by going to mongodb.com/atlas.

[INTERVIEW CONTINUED]

[00:37:01] JM: To take it to a related to an episode that we did recently. I talked to somebody from the Slack frontend team, and Slack uses a pattern that I hadn't seen before. Maybe this is common in frontend. I don't do as much coverage of the frontend world is a should, but they use this pattern where if you want to snapshot your Slack state so that you can – If I go sit down at my desktop. I work in Slack and then I go and go sit down in another desktop and work in Slack. In order to maintain the same state, they snapshot the Redux store and put it in a CDN and that way I can access the state by just – They download it from the CDN and they push it out to my Slack application. I just use it to – I guess, explore with you, is an infrequent pattern where the Redux store or whatever state management store you're using me is saved to the cloud and then when you reboot the application somewhere else,you load that state?

[00:38:10] KD: That's an excellent question. I don't think that – That's actually a really cool technique. I don't think that I've seen that in wide use, but it makes total sense and it's not something you'd have to use Redux to be able to accomplish. But I can see how Redux could make that a little easier. But I have definitely seen that sort of technique used for error monitoring and reporting. If the user experience is an error, let's snapshot our state, send it to our monitoring service so that we can see what state the user was in.

Yeah, I think that – Actually, because Redux makes that a little bit easier, then that's a pretty good use case for Redux. I would just caution anybody who wants to use Redux to be really thoughtful about what state they put into their Redux store and maybe explore a couple other alternatives as well.

[00:38:57] JM: What are the most common performance issues that you see in React applications?

[00:39:01] KD: That is a good question. I get questions about performance so often as an instructor. Pretty much every workshop that I give, regardless of the level of experience of the attendees has some question about performance, and most of the time it's kind of this armchair performance where you're sitting back and you look at something and you say, "I think that's probably slow." When you actually measure it and you're like, "Yeah, you're right. It can only do 4 million operations per second." If you need to do more than that, then maybe look at something else.

Yeah. Most of the performance problems that I see – Again, I mentioned this earlier, but the number one solution to performance problems is less code. So you load less code, you run less code. I think that the lowest hanging fruit for most people is loading less code. I don't think people are code splitting as much as they should, and it's so simple to do, especially since we have react.lazy and Suspense. So you can code split on any component in a way that's pretty simple. You don't want to go overboard, but you definitely can be methodical and surgical about where you're doing code splitting and save yourself quite a few bites that you sent to the user.

You certainly don't need your entire application loaded on the login screen. Nobody should be doing that. In fact, I would argue that you may not even need React on the login screen. But that's a subject for another time. That's one half of the thing. The other half is running less code. I think the number one thing that people try to optimize for when they're talking about running less code is re-renders of components.

Whenever there is a state change in a React application, React takes the component where that change occurred, and it re-renders that component which will trigger a cascading effect of all other children of that component will be rendered. Now, I actually have a blog post about this. I think it's titled One Simple Trick to Speed up Your React Apps or something like that. But if your component receives its children as a prop rather than just rendering its new React elements itself, then React can actually like pre-optimize to say, "Oh, these children couldn't possibly have changed. So I'm not going to bother re-rendering those."

Restructuring the way that you write your components and actually writing things in this way is I would say more idiomatic even though not as many people do it this way. It just makes more sense to write things this way rather than having the component render the JSX or the React

elements that it's rendering. Accepting those as a prop to the component and saying, “This is where those should go. Whatever they are.” But that that's one optimization that you can make that like it's just a little bit of a restructure.

Another thing that like as part of this whole re-render problem, a lot of people just see re-renders as bad. If it didn't need to re-render, then that was a bad thing. So what they'll do is – Let's take a scenario that is actually I see quite frequently. Let's say that we have a component that takes 50 milliseconds to render. That would be a very long render, but we’ll just use that as the number. If it takes 50 milliseconds to re-render, that's very slow, and so we want to – Then we noticed that it's actually re-rendering unnecessarily two times. It renders necessarily once and then – Well, let’s just say unnecessary one extra time. So now it's taking a hundred milliseconds to round, when it really should only take 50.

What people will do is they'll use optimizations like react.memo to say if none of my profs changed, then I should not re-render. Okay. It’s either the profs changed or the context it was consuming or a couple of other reasons that it would re-render. React.memo just says, “If none of the profs change, then I should not re-render.” Okay. So we fix that problem and we can move on with our day. Now this thing is not re-rendering unnecessarily. It's rendering only one time and it’s taking 50 milliseconds.

Let's look at developer B who looks at the problem and says, “Wow! It's re-rendering twice. That's taking a hundred milliseconds. That's a really long time. I wonder if I could speed that up.” So they optimize the component instead and they make that component now render in 5 milliseconds. Okay, so now it's only taking 10 milliseconds to render twice and we’re like, “I've got bigger fish to fry. I'm just going to move on.”

The difference at the end of the day is eventually your component absolutely has to render, like something changed, it does need to render. With developer A, their component is going to take 50 milliseconds when it renders because it was necessary. Whereas with developer B, it's going to take 5 milliseconds because it was necessary. So you're better off there.

Then also with developer A, to avoid that unnecessary re-render, they had to wrap it in this react.memo thing which adds complexity to the code because not only do you have to add that

complexity, which isn't very much, but if you need to add a custom prof checker. I forget what it's called. A custom function to determine whether it should update. If you do that, then you do add quite a bit of a complexity, but you also have to check everywhere that's using that component and ensure that the profs of its passing are going to be consistent between renders.

That has this spidering effect where maybe that component got those profs from somewhere else and so you have to keep on going and memoize all these values just because you wanted to make sure that this one component didn't re-render unnecessarily. where if you just went with developer B's approach and said, "Let's not care about how often it re-renders and just speed it up so it doesn't take so long," then it you don't have that spidering effects and your app is faster to boot. Eventually, it absolutely does need to re-render. So you just want that to be faster anyway.

Anyway, that's a long way to answer your question, but this is something I see a lot, is people just instantly reach for how do I reduce the number of times this thing is rendering when they – And before you should ask yourself that question, you should ask how can I make it so that this thing renders faster?

Now, there are sometimes when it is rendering all the times it needs to, or it's going as fast as it possibly can, but it's still re-rendering unnecessarily. Because I'm rendering 700 of these things, I need to make sure it's only rendering when necessary. There are those times, and that's when you reach for these optimizations. That's why those optimizations exist. But reaching them as the first solution often has the spidering effect that goes throughout your whole application, making it more complex. That's a problem that I see quite often.

[00:45:28] JM: You've spent a lot of time studying how to properly test JavaScript applications. Give me your condensed thesis on how to test JavaScript applications.

[00:45:39] KD: Yeah. The philosophy – I have done a lot of work on testing with JavaScript, and the philosophy that I've come up with over the years is the more your test resembled the way your software is used, the more confidence they can give you. What that means is if you are writing tests that – Well, let's take a step back. Your software, if we're talking about React components or backend or anything, your software probably has two users. It has the end-user

that’s actually interacting with your application and it has the developer user that's using your API or rendering your React components or calling your function, whatever it is. So you have those two users. Those are two use cases your software needs to continue to support or like it’s useless, right?

When you start writing tests that don't resemble the way that your software is used by those two users, what you're doing is you're inviting a new user that your software needs to support. I call this the test user or the third user. The test user, the reason that we write tests is to have confidence that our software is going to continue to work. I know some people like to write tests for their workflows, and that's awesome. Feel free to do that, but the reason that we commit test to source control and run them on CI is to make sure that our software continues to work as it was designed, that it continues to support those two users.

Once you start writing tests that use your software differently than those two users, then you've created this third user where you have another point of failure. So if you change something that breaks the use case of the test user, literally, nobody in the world cares except for that test user. What that means is your test user exists for themselves and they're not paying you any money and you're not making the world any better by supporting the use cases of that test user. There's no reason to bother supporting the use cases of that test user.

Now, sometimes we have to make tradeoffs. Testing is just up and down tradeoffs where you have to mock the credit card charging company because you don't have enough of a credit limit or something, like I don't want to charge my credit card every time I check out. So you do have to mock some things. You have to poke holes in reality sometimes. You make that tradeoff, but you acknowledge the fact that that is not using your software in the way that it really should be used and you counteract that by maybe having a smoke test that tests against young some other service, or like Stripe I know has their test service. So you can maybe interact with something like that and just have one test that make sure that at least this thing works, but then the rest of your test can operate on a mock. But that's my whole like condensed philosophy around testing, is the closer you can align your test to the way your software is used by the two users you actually care about, the more confidence it can give you, and without giving you confidence, a test is useless.

That's why I created testing javascript.com so I could teach people how to do this effectively with the tools that I think do this the most effectively. That's why created the testing library for React and Angular and Vue and Ember. Well, Ember is technically not – There is not an implementation there, but you can make one. Like Puppeteer, Test Café, Cyprus. I have this testing library so that people can do a better job of just naturally writing tests the way that their software is used.

[00:49:03] JM: Writing tests has always bored me to tears, and I know that sometimes you have to write tests. Perhaps most times you have to write tests, but can you tell me when can I avoid it? When can I comfortably avoid, defensively avoid writing any tests at all?

[00:49:26] KD: Yeah, that's a great question, and you're not alone. So many people do not like writing tests, and it's very reasonable to not like it, because really it can be often difficult to identify how this is helping the bottom line or helping you make the world a better place. Whatever it is you're trying to do. Yeah, places where you can avoid writing tests are things where you don't care if it breaks. You don't care if you'd get a call at 2 AM in the morning about this thing breaking, or even better, you wouldn't get a call at 2 AM in the morning because nobody cares. A good example of this, I was just working on an open source library I have today that very few people use, and I have no test on it. Lots of people see me as the JavaScript testing guy.

[00:50:11] JM: How dare you?

[00:50:12] KD: Yeah. Exactly. I was live streaming and I had a bunch of people watching and I knew that some of them were going to be like, "What the – You don't have test? What's wrong with you?" I just preemptively mentioned like nobody uses this thing, or very few people do, and the people who do can just use the previous version until I get this fixed if I break anything, and it's not going to bring down their CI or anything like that. I don't really care too much.

If you have some software that you're writing and you don't really care too much about the impact of breakage, and this could happen even within an application. It's actually a really important distinction to make that if there's an area of software that it doesn't matter that much if it breaks, then that's not your place to be focusing your time and effort.

Really, testing is no different from any software. The reason that we test or the reason we write software is so that we can avoid doing things manually. That's what software is all about, is like how do I make this thing that I do manually happen faster? How do I crunch these numbers faster? How do I analyze this data faster? Whatever it is. That's what software is for.

Tests are exactly the same. How do I make sure that my changes do not break when I ship this to production? That's what tests are for. If it's not something that you care too much about, then there's no reason to automate it. We've got this one page that like six of our users use. They only use it like twice a month. I'm not going to write test for that, and we don't change that code very much anyway. When it does break, they don't mind reaching out to us, whatever it is. There are so many other ways that I can make the world a better place in my day that I'm not going to waste my time writing tests for that. Instead I'm going to build this feature that will improve the experience for my users.

You have to weigh this and just like think about, "Okay. I've got X-number of things to do. Some of those are tests, and we have this one page that we have so many people using it because it's our checkout flow. When they click checkout, if it doesn't work, that is so bad for my business. I'll lose millions of dollars for that checkout button. So yes, I am absolutely writing tests for that. I'm not getting around it, because that's how I can make the world a better place, the best, is by making sure I never break that page."

Then once you got those tests in place, then you think, "Okay, what's the next thing I could do to make the world a better place?" Well, there's this feature that people have been asking me about and I haven't had time to build it. I think that will make the world a better place better than writing tests for the settings page. I don't see writing tests as any different from writing software features. It's really, "Okay, I've got this priority of things to do. What's the next thing on my list that will help me make the world a better place better than anything else?" Sometimes that's test. Sometimes that's features. Sometimes that's bugs and sometimes that's going home and playing with your kids, whatever it is.

[00:53:07] JM: All right, last question. It's 2020. What does JavaScript fatigue mean today?

[00:53:13] KD: Yeah, JavaScript fatigue. I lived through it. I was back in the backbone days, before it really ramped up and then AngularJS came out and then React and Ember were going on and Angular 2 was coming. Then we had the Flux wars and there are like 30 different libraries. Then of the tools like webpack versus browser 5 and then – Boy! Parcel? What is this? We've got all these different tools coming out and testing tools as well. I think that people are still innovating a lot. You could still build an application pretty well using the tools of five years ago, but you can build an application faster and better using the tools of today. You just have to acknowledge the fact that the tools of today are still being iterated on and improved.

Yeah, what is JavaScript fatigue mean today? I think that we experience it less and the tools like have really solidified pretty well. People have a pretty good understanding of what different tools are good for, what they're not good for, and even better than that, we have a better roadmap for new people to follow where we say, "Hey, if you want to get into React, then you're going to use Create React App, or you can use Gatbsy, or NextJS." Those tools themselves have a really good ecosystem within themselves, lots of resources and they're just really user-friendly. You don't have to make as many decisions. I think that's actually where JS fatigue comes in and play most is where people have to make a lot of decisions about things, and Vue is actually very good at this and Ember as well where they don't let you make decisions. They just made them all for you and you just follow it and 90% of the time you're going to be just fine following those decisions that they've made for you.

React especially I think is prone to decision fatigue or JS fatigue because the React team is all about making the core library really good and then they leave it to the community to fight over what's best. We do still have a little bit of that in the React community. But I think we're slowly kind of coming to a consensus on what the best tool for 90% of the jobs is or what a good tool for 90% of the jobs is. I think we're going to continue to go in that direction.

We do still have cool new things that are coming out that people should continue to analyze. Like just two year ago, I released React testing library. Svelte has been out for a while, but it was just a year ago that Svelte 3 came out and it's considerably different and much better. You still want to keep up with what's going on, but I think that we're slowly – Like we're maturing as a community. We haven't been around for as long as many of the other communities. There's just so much innovation that's happening in our ecosystem because we're maturing in a day and

age with GitHub and npm and with these tools that make this kind of open collaboration and maybe even premature collaboration possible. I think that's kind of why and understanding – Maybe explains a little bit why that fatigue is slowly tapering off and I think eventually will mostly go away.

[00:56:22] JM: Kent, thanks for coming on the show. It's been great talking.

[00:56:23] KD: Yeah, thank you so much.

[END OF INTERVIEW]

[00:56:34] JM: As a company grows, the software infrastructure becomes a large complex distributed system. Without standardized applications or security policies, it can become difficult to oversee all the vulnerabilities that might exist across all of your physical machines, virtual machines, containers and cloud services. ExtraHop is a cloud-native security company that detects threats across your hybrid infrastructure. ExtraHop has vulnerability detection running up and down your networking stock from L2 to L7 and it helps you spot, investigate and respond to anomalous behavior using more than 100 machine learning models.

At extrahop.com/cloud, you can learn about how ExtraHop delivers cloud-native network detection and response. ExtraHop will help you find misconfigurations and blind spots in your infrastructure and stay in compliance. Understand your identity and access management payloads to look for credential harvesting and brute force attacks and automate the security settings of your cloud provider integrations. Visit extrahop.com/cloud to find out how ExtraHop can help you secure your enterprise.

Thank you to ExtraHop for being a sponsor of Software Engineering Daily, if you want to check out ExtraHop and support the show, go to extrahop.com/cloud.

[END]