EPISODE 1017

[INTRODUCTION]

[0:00:00.3] JM: React JS began to standardize front-end web development around 2015. The core ideas around one-way data binding, JSX and components caused many developers to embrace React with open arms. There's been a large number of educators that have emerged to help train developers wanting to learn React.

A new developer learning React has numerous questions around frameworks, state management, rendering and other best practices. In today's episode, those questions are answered by Ryan Florence, a Co-Founder of React Training. React Training is a company devoted to helping developers learn React and React training trains large companies, like Google and Netflix how to use React.

Ryan has a strong understanding of how to help developers be productive with React. In today's episode, he explains some of the fundamentals that commonly confuse new students of React. Ryan is also speaking at Reactathon, a San Francisco JavaScript conference taking place March 30th and 31st in San Francisco. This week, we'll be interviewing speakers from Reactathon. If you're interested in JavaScript and the React ecosystem, then stay tuned. If you hear something you like, you can check out the Reactathon conference in person.

[SPONSOR MESSAGE]

[0:01:30.1] 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]

[0:03:20.0] JM: Ryan Florence, welcome to Software Engineering Daily.

[0:03:22.4] RF: Hey, thanks for having me.

[0:03:24.0] JM: You've been part of the React community since the early days. How has React application development changed since the earliest days of the project?

[0:03:33.6] RF: Yeah, I've been – it was open source seven years ago and I've been using it about six years, which is unheard of in the JavaScript world. I've been blown away at how long it survived. Honestly, React itself – there's two Reacts, right? There's React the library and then there's all caps REACT, the whole community and ecosystem around it.

React itself over the years, I haven't really seen it change much. There were small little tweaks in the way you would define components, but it went a good four or five years without really changing until they released Hooks last year. That was the first real change that has happened in React application development.

To me, it got more in-line with the philosophy, or the idea behind React. The community itself, like everything around it, I think we've seen a few phases. There's the early adopter phase and then we – my business partner, Michael Jackson, we built a thing called React Router. That helped expand the use of React, because people now knew how to – React really is just little components, little parts of the page that you can build. You can control the whole page, or a small piece of the page, but people didn't really know how to put the whole thing together into a full application and that's where React Router helped adoption a bit there with React as well.

Then Redux. We got the Redux phase. Dan Abramov is a brilliant developer and gave an amazing talk showing off some stuff that he wanted to do. He want to do time travel debugging and he came up with this thing called Redux. Super cool library. We're all really impressed by it. People started using Redux a lot.

That phase is a little bit weird for me, because a lot of what makes React great got pushed to the sidelines and people don't even know that you could manage state in React. They thought that you had to bring in Redux to do it. I think we lost some innovation that we may have had sooner. Then, I think we're in a new phase now with Hooks and we have way more ability to compose behavior and our UI. There's my React over the years in a nutshell.

[0:05:51.7] JM: Let's talk about React pre-Hooks for a little bit. Before there were Hooks, what was the distinction between using a state manager, like Redux versus just managing your state with the primitives that are available in React without Redux?

[0:06:13.0] RF: Pre-Hooks. That's what –

[0:06:15.4] JM: Yeah, yeah.

[0:06:16.8] RF: Before we had Hooks.

[0:06:18.0] JM: Before we talk about Hooks. Right.

[0:06:20.0] RF: Yeah. State management is a huge term. You have to manage state for a little counter. Click a button and the count goes up. Then you also have to manage state from maybe you're caching your JSON responses from a server that you're sharing across your whole UI. State management is a huge topic, I guess.

Redux really shined, especially pre-Hooks, for the caching data from your server use case. That may not be why it was designed, but that is definitely how it got used is a screen shows up on the page, you make a request, so you dispatch in action. “Hey, we just mounted this screen.” Then you have maybe some middleware, or something that goes and fetches data from the server and then puts that data into the Redux store, which is you can just think of it as a cache. You can just think about it as a variable that you're putting things in. Then you could access that that data anywhere.

Maybe you fetched the user, or maybe you fetch a set of invoices and maybe you got two or three or 10 screens that need to list those invoices. You've got this client-side cache of that data. Now you could use Redux for more things than that. That's how I've seen people use it for the most part.

Before Hooks, it was I think probably the best way to manage that data. I call that explicit data. It's data that you need to see, you need to know about, versus implicit data, which is data that you don't care about state. You don't care about as an app, like is a drop-down menu open? If I've got a drop-down menu component, I don't care if it's open or not. That thing manages state on its own. Redux wasn't really useful for that thing, but very useful for the bigger application level state.

[0:08:08.5] JM: Were there any patterns around using Redux that caused applications to be built in a way that that wasn't architected maybe as soundly as it could have been architected in the absence of Redux? Were there certain anti-patterns around using Redux, where people leaned on a client-side cache in a way that was maybe not ideal?

[0:08:37.5] RF: Every app is different. For me, one of my frustrations, regrets, I don't know the word is, with Redux was we pushed away encapsulation; the idea that you could have a section of your application that could just work all on its own. It managed its own state, it knew how to

fetch its own data. You could just grab an invoice component, dump it on a page, it would go fetch that invoice and manage that all itself. Or even the more – the micro-interactions, like drop-down menus and things. People would start pushing that state into Redux as well, which now to use a drop-down menu component, not only do you have to bring in the component code, but now you've got to bring in these things called reducers and then you've got to combine that reducer into your application reducer.
Basically, if you pushed all of your state into Redux, all the state of your app, which is what a lot of people did, or at least felt they were supposed to do. If you push all that state into Redux, then there's no encapsulation and now you're just building a monolith. When you don't have encapsulation, you also have a harder time composing things together in ways that you didn't expect to. Yeah, I just felt that idea of shoving all your state into Redux hurt the idea of React, that you can just have one small piece of the page take care of itself. I gave a talk about this years ago at React Rally.

[0:10:05.7] JM: I mean, that's a great point, because I feel in – when I took computer science classes, there was a distinct anti-pattern around global variables. I just remember being told by people reviewing my code, or computer science professors that you need to be very careful with global variables. You can get very carried away and it's a dangerous pattern.

[0:10:30.9] RF: Yup. Yup. I mean, that's basically how Redux got used. There was education material and conversations on Twitter and conference talks that you need a state management library, like Redux in order to use React. Everyone would just shove all their state in it. In fact, I got up in front of one of our workshops. There were there about 60 people in the room. It was a React fundamentals workshop, so we start showing them React. I showed them, back then it was with a class component, just the function this .set state in a component, which is how you update an individual component, just itself.

You call it set state, give it some values and then React will update the screen for you based on that new state. I started getting all these puzzled looks from everybody. Oh, that. Sorry. Sorry, this was an advanced workshop, because they had been using React for a couple of years. Everyone was just looking at me all weird and in the first break, a few of them came up like, “What is this .set state? I thought we can't do that.” I was like, “What do you mean you can't do that?” They open up their laptop and showed me and they had a lint rule. In JavaScript, you got

linters to keep you from, I guess, using the bad parts of JavaScript or whatever. They had a lint rule that disallowed them from using the primary API of React, which was set state. They could not even use React state. They had to use Redux as state. Yeah, it was a wild time.

[0:12:01.3] JM: As we move forward in the timeline of React, you mentioned Hooks as transformational. Can you explain what Hooks are and how they have changed application development?

[0:12:14.1] RF: Yeah. Hooks and React, they give us a new layer of composition. We had it before you could twist components around. A component, generally think of something that you can see on the page, right? It actually renders some HTML. Before Hooks, if you wanted to abstract some behavior that didn't have any UI, say you wanted to abstract subscribing to the scroll position of the window, or you wanted to abstract reading and writing to local storage, or maybe you wanted to get the user's geolocation, the browser has an API to say, “Hey, what's the geo position, or geo location of the user right now?” None of those concerns have a UI, right? There's nothing you see. There are no elements you need to render.

In React, the only tool we had was a component. We started building these funny components that didn't render anything. They didn't return any elements. We just used them for their side effects. Then through some really terrible looking syntax, we called them ‘render props’. Instead of passing elements into the content, or the children of a component, you'd give it a JavaScript function. Then that function could then yield out to you whatever state you were finding, that local storage value, or that scroll position, or the geolocation of the user. Then you could use that data to render some UI.

It worked great. I loved doing it that way. We were twisting components into something that they weren’t for the sake of encapsulating that behavior and then sharing that behavior, or sharing that state that have figured out with the rest of your application. We didn't have a great way to compose non-visual behavior. Hooks gives us that ability to compose non-visual behavior. Instead of having to do those kinds of things in components, we have these Hooks. There are just a few basic ones. There are a couple for state, use state and use reducer.

If you use these hooks, then those will cause your component to re-render. Those then can be composed inside of – or sorry, let me back up. We got the couple for state. Then we've got one called ‘use effect’. Use effect is the thing that allows you to – it's the lifecycle of the component, right? The component just showed up in the page, now we want to go fetch some data or subscribe to the user’s geolocation.

When you mix effect and use effect and use state, these two hooks, you can now make your own hook. You can make your own hook called ‘use scroll position’. Then inside of that function, and that's what's beautiful about these hooks is they're just functions. There's no API to say to React, “Hey, this is a hook.” It's literally just a JavaScript function. If you use state and you use effect inside of your own custom, use scroll position, you can return out of that thing, the scroll position and you can subscribe to the window scroll event in the effect. Then in that effect, you can set the state of your use state hook that you made.

You can encapsulate all of that, the state of the position, the behavior to subscribe to that, even clean up, so that when the component on that you want to stop subscribing to that, and then you just return out whatever your current state is. Now anyone, anywhere can just bring in your one little function, use scroll position. Whenever the scroll position changes, it’s going to cause the component you used it in to re-render and it's going to return to you what the user’s location is.

Now instead of us having to twist components to share this behavior, we get plain functions. That means we get composition for free. These are just functions. You can compose these the exact same way you compose any functions, which I think is super cool.

[0:16:12.3] JM: A hook sounds similar to unobservable from there's the library RxJS, which allows you to build reactive systems in JavaScript out of observables that emit these events. How is a hook different than an observable?

[0:16:31.2] RF: I'm not incredibly familiar with RxJX. I have goofed with it a few times. Maybe people will be like, “Ryan doesn't know what he’s talking about here.” To me, observables have a push API, right? They push values to you. You'd subscribe to a thing and then give it a call back and then it's going to push those values to you.

A hook, you pull values. It has a return value. Instead of a callback where a value gets pushed into the callback, instead you get to ask for a value when you want it. For me, that's the main difference. I find pull API is a lot more composable and just easier to deal with. I don't think there is similars that might seem, because hooks are not a general abstraction for programming. They're a domain-specific thing for React.

They are twisted up all in the guts of React everywhere. It's not something that you'd use outside of React. It really just says, if you use state and you set, some new state it's going to cause the whole thing to re-render. Then the effect hook just lets you plug into the lifecycle of that rendering. RxJS is a general pattern for any programming. Hooks are simply for React and its render life cycle and how to start a new one.

[0:17:49.7] JM: Right. It seems proof of the desirability of React that in its infancy, people were contorting the React component system to fulfill – I mean, basically a front-end component system, a visual component system, they were contorting it to do behavior that was non-visual, I suppose because the basic dataflow aspects of React were so desirable that they basically wanted – people wanted non-visual components and React just had to – to evolve over time to accommodate the non-visual use cases in a better fashion.

[0:18:35.7] RF: It's more about composing that non-visual behavior. React has always been able to handle it really well. With before hooks, we had class components and they had life cycles, like did mount, did update, so you could do these non-visual behaviors after the component mounted, or after it updated. You could change the state of a component. Then in response to that, do some side effect that was non-visual.

I don't want to give the impression that React couldn't do non-visual behavior. Well, it did a great job of it. In fact, I had a better time in React than everything previous that I had used for non-visual behavior. The rub was how do you share that non-visual behavior? It was easy for an application-specific component to do these kinds of side effects, but it was hard to then take that and turn it into a reusable chunk of code that someone else could use. That's what hooks brings. It doesn't let us do anything new, except it lets us compose that behavior in a better way.

[SPONSOR MESSAGE]

[0:19:40.8] JM: If you can avoid it, you don't want to manage a database. That's why MongoDB made MongoDB Atlas, a global cloud database service that runs on AWS, GCP and Azure. You can deploy a fully managed MongoDB database in minutes with just a few clicks or API calls and MongoDB Atlas automates deployment and updates and scaling and more, so that you can focus on your application, instead of taking care of your database.

You can get started for free at mongodb.com/atlas. If you're already managing a MongoDB deployment, Atlas has a live migration service, so that you can migrate your database easily and with minimal downtime, then get back to what matters.

Stop managing your database and start using MongoDB Atlas. Go to mongodb.com/atlas.

[INTERVIEW CONTINUED]

[0:20:39.2] JM: There's a term that I've heard associated with the lifecycle of components called ‘prop drilling’, that apparently you've coined this term. Can you explain what the term ‘prop drilling’ means?

[0:20:54.7] RF: A long time ago when I very first started using React, I was working at an education company called Instructure, making a learning management system. I really loved React. I've been goofing around with it up on my own. We decided to start adopting it and using it for some of our new code. I was trying to come up with a little mini-workshop for my team, a little two, three-hour thing. I was just trying to think of the topics of what do you need to know to use React well? In a couple hours, what do you need to know?

One of the things that I really focused on in there in that material was this idea of prop drilling, which is in React, you're – it's like HTML. You can think of React components like your own custom elements a little bit. If you've got state in one component, maybe high up in the element tree and you want to get that state down low to something down at the bottom, maybe you've fetched all your users at the top of the app and you're making a whole list of those users and then maybe all the way down, there's this little avatar component for just one of the 100 users on the page, and you want to get that user’s avatar down to them.

You've got to pass that user object several layers down. You take it from the parent component, pass it to one of its children, that children takes that problem, passes it to the next child and that one takes that problem, passes it to the next child. You may find that you do that five, six, 10 levels down. What really happens is when you start refactoring, you start identifying pieces of your app. You identify the avatar component itself, where maybe that was just in-line of your list at first and you're like, “Hey, let's make an avatar component.” Now you got to pass that user to it.

That's what I called prop drilling. It's like, you have to drill this hole through all of these components that maybe don't really care about that prop, but they need it just to be able to keep on passing it down the tree. Then that the second part of that is to get data from down low in the element tree back up to the top. Maybe someone like clicks ‘delete user’ or something. That's all the way down six levels in some user component, you've got to get that action that they want to delete the user, all the way back up to the component that owns it. Maybe you’re six levels up from there.

Not only do you have to drill that prop down, this is why I came up with word drill is it's not so much about passing the prop down, it's about sending the information back up. If you send that data down six levels, you also have to send a function down all of those six levels. That's the hole that you drilled to then be able to throw that information back all the way up the tree.

[0:23:38.0] JM: You run React Training. These concepts that you're explaining, these are things that you teach in workshops and training scenarios. When you're teaching professional developers React, what are some of the common misunderstandings, or confusion points that you encounter?

[0:24:00.1] RF: Oh. I think one, well really, just JavaScript pretty much. We like to joke that our workshop is actually just a JavaScript workshop in disguise as a React workshop. Yeah, there's a lot of just JavaScript education that goes along with it, because React gets out of your way really quickly. Once you get the basic idea of elements and components, it's really just JavaScript. It's almost one of our instructors, David. He's like, “JavaScript’s almost mean about that puts JavaScript in your face.”

Outside of that, just how the render lifecycle that takes a little bit to help people understand how that works, because you're just looking at a function and somehow magically you call set state inside of there and then everything updates and all your variables are brand new inside of the function. What does that mean? We got to talk about function closures and scope.

Before hooks, we talked a lot about JavaScript context and this, this keyword doesn't work quite the same in JavaScript as most people expect it to. Yeah, and then I guess we've touched on this, that there's a misconception a lot of times that React can't manage state when that's half of what the library is is managing state. You don't have to bring in Redux or MobX or some of their state management thing. We did it for a couple of years before Redux even showed up. Yeah, Redux or React can manage its own state. I don't know if there are any other common misconceptions other than that.

[0:25:31.3] JM: I'd like to talk to you about the broader space of React application development. One thing I've talked about to a couple other guests recently is the type of frameworks that you can use when you're building a new React application. Today as I understand, the most prominent frameworks are you have create React app, which is typically for more boilerplate applications. You have Next.js, which is for I guess more sophisticated applications and when you want some complex choices of server-side rendering versus client-side rendering, for example. Then you have Gatsby, which I guess is its own thing. How would you contrast these frameworks?

[0:26:18.7] RF: They all have I think their sweet spot, but they're all very capable. Create React app is probably the simplest one. It doesn't do any server rendering. It doesn't do any – it's just you want to do a React app, but you don't want to learn Webpack is basically what create React app is for. You can just from the command line say npx create React app my app, whatever the name is and it'll just give you a big boilerplate for what people call a single-page application. There's no server. It's just a JavaScript bundle. You can deploy that anywhere.

Now you can use create React app and then server render that thing, but you're going to have to write your own code for that. Create React up will create the bundle for your app and you can

actually then take that bundle and server render it yourself. Yeah, like I said, you'll be right in your own code there. If you just want to get started with React, that is absolutely the way to go.

I think our website for a little while, now it was our own thing. But it wasn't really different. We didn't do server render or anything and we still got organic Google search result hits. It all worked out fine. Then you've got – let's talk about Gatsby. Gatsby, it's not only this, but it's the way that helps people understand it the most is think of a static site builder. I used to see, use one in Ruby called manic, I think. There's a really popular one in Ruby. I forgot what it's called. Jekyll, is that what it is?

[0:27:50.0] JM: I don't know if that's Ruby specific. I've never heard of Jekyll. Hugo.

[0:27:55.2] RF: That's a static site builder, right?

[0:27:56.5] JM: There's Hugo and Jekyll. I think those are both the static site builders.

[0:27:59.8] RF: Yeah. Yeah. The idea is you can write some code. It's like you're on a server, right? You get to build these abstractions and share code and use code to generate your user interface. At the end of the day, you're just going to ship some HTML, so there's a little build process to take your code, that then spits out some HTML and then those are just flat files that you could upload to a CDN or something.

Like I said, Gatsby isn't just a static site builder, but that its foundation. You can still build really cool dynamic things with it as well. What it does is it'll take all of static markup, but then when the page loads, it then layers the React app on top of it. I guess, it's server rendering. It's CDN rendering, flat file rendering. That's the fastest way to do it.

Then the JavaScript downloads layers over the top a client-side router, the events, dynamic behavior, all that stuff. Now as the user navigates around Gatsby who has already eagerly loaded the pages that they're probably going to go to next. Well, it's not that smart about it, but it'll preload the other pages in the app. Now when you click on a link, you're going to get there instantly without a full server hip.

Then Next.js, they've made some changes recently that I haven't kept up on. I've been pretty busy over here working on our company. Next.js also has this same ability that Gatsby does to compile it down to a static site, that then layers JavaScript on top of it. It goes a step farther and actually does some server-side rendering. Your code actually runs on the server when the user hits the page. This allows you to do more dynamic things; for example, fetch some data. They've got some hooks in there. I think it's called get initial props or something like that, or they may have changed it.

When the user hits the page, you can go fetch data just like a normal server rendered app and then render that data in your UI. With a static site like Gatsby, you can't do that. You'd have to know your data at build time, so now your data is limited to when you built and then you'll have to go fetch new stuff. Gatsby doesn't really work for data that's going to be changing a lot, like tweets or something, right? You want to see somebody's – some new posts that are happening a lot. Not great for a Gatsby. But Next, you can do that thing and get that more dynamic data in the server render. They probably have other stuff in there now too that I don't know about, but that's the difference is your code actually runs on the server before you send the page.

[0:30:31.8] JM: Got it. The next thing I want to talk to you about is GraphQL. I know that if you're a sophisticated React developer, you're often building your new applications using GraphQL, but it's obviously not a necessity. When you're talking to people that you're teaching React, do you try to prescribe that they should use GraphQL from day one? Or do you just ignore GraphQL, because it would introduce more complexity?

[0:31:02.9] RF: Yeah, with our workshops, we don't cover a whole lot with GraphQL. It's a really interesting technology. We're in the middle of building something new that we're not ready to announce or anything yet. We're going to start with GraphQL from the beginning. Just talking to some people from my previous job and they've started using GraphQL and having a lot of success. It is a great way to work with data over a network. You have two choices; you can either – I guess you got three choices. Either build one-off endpoints for each screen that you build, right? This is our gradebook. If I'm going to render this gradebook, what data do I need? You can make an endpoint on your server that says, "Here's all the data you need for the gradebook."

Or you can do a more restful approach. If you take rest to its logical conclusion than most your UI, you're going to have to be making 70 requests to the server in order to get all of the data that you would need for a more complex UI, or at least a dozen, right? You're going to have to hit a lot of endpoints. Then the client, you're going to have to construct that stuff back together into objects and stuff, or build those relationships inside of the client.

What would be best is if we get to send SQL over the network, right? Just send a SQL query over there, or SQL. I never know how I'm supposed to say that. I think of GraphQL as being able to just send a database query over the network. It's a safe query language for the network. You can just say, “This this is what I need.” Instead of writing a SQL query, I write a GraphQL query. How that's handled on the back-end, there's a lot of ways to do it. You can you can have a GraphQL first API, I guess, database, something Hasura, if you've ever heard of that really cool project. It's backed by PostgresSQL.

Or you have your existing REST API, you have your existing databases, you can build a layer in front of it, so that your client applications can just send GraphQL and then your server, instead of your client, your server can then figure out, “Okay, what are the 12 REST end-points I need to hit to get all of this data?” Because if the client has to do it, then every page that needs that data has to do this and every app that your Android app, your iOS app, your handful of web apps, all of them are going to have to be doing these shenanigans of getting the whole data for this complex page together.

GraphQL, especially with a node server in between your client apps and your real database is a really great way to just simplify all the code and all of your clients and give yourself a query language over the network.

[0:33:50.4] JM: You and your co-founder of React Training, Michael Jackson, you're both known for helping create React Router. I just like to get your perspective on React Router and how its usage has evolved over time and how the how the project has evolved over time.

[0:34:10.1] RF: Yeah. We released that as we were learning React. You can just look at React Router and the APIs and you can just see the two of us learning React is really probably the best way to describe it. Yeah, so in the beginning – actually, the very, very, very first one we had

was it was one of our better ones. It was so early. Everyone wanted data – or they wanted a server rendering and all this other stuff. We didn't know what we were doing, and so we came up with these APIs that were okay. That's what V3. Version 1, 2, 3, all almost identical. We had to make a couple of tweaks to a couple APIs and break them. Version 1, 2 and 3 are basically the same thing.

We still support V3. You can run React Router V3 on a React 16.9, I think is what we're at right now. Thanks to Tim Door. He has been diligently maintaining that for years. One day, Michael and I had given a workshop. I can't remember the city we were in, like Chicago or something. We were standing at the elevator, because we just finished the workshop talking about React Router. We were talking about how awkward it was for us in our workshops, because we would do these two days, show people all these cool patterns, different ways to compose with React, different ways to share behavior and state.

Then we'd get to the lesson on React Router and it was like, "Forget everything we just taught you. It doesn't work here." We were like, "What have we learned now about React that we can take to React Router?" That's where React Router version 4 came from. It was big API change, upset a lot of people, but I mean, we were learning React with everybody else too. We're just a couple of guys trying to run a company. I think it was a good choice and it just made everything a little bit more dynamic.

I keep using that word composable. It'd be hard to really explain why they React Router API was more composable without throwing some code in front of us. It took all the principles of React that we taught in our workshops and that as a community, we had all learned and we applied it to React Router. We just hung out like that for a few years, honestly. I think three, four years now we haven't touched that API. Maybe we're a little bit scared of getting everyone angry again by changing an API, because React, React is pretty much always backwards compatible.

For a library like React Router, it's a little bit harder to maintain that, because we're a level away from React, right? If we want to use the new React features, we'll probably end up breaking something of ours.

Anyway, so we went a few years no API changes. Now we just released on Friday last week React Router version 6 alpha. Four and five are pretty much synced to that. We made a little change in there. We added some hooks in version 5, but we didn't break any API there. V6 now, we've updated all the code to use hooks. We've made it a whole lot smarter. We have this matching algorithm now that gets rid of a bunch of problems that we had before with that people had to do themselves constructing up their routes. It's half the size of what it used to be.

It's got a whole bunch of more features, it's more composable, it's smarter. We have these things called relative routes and links now. You used to have to construct the entire URL for your link. Now it's smart enough to just inherit the link above it, or the route above it. Yeah, and it's half the size. We're really excited about it. We've got even bigger plans for it as well, but we want to get this to a stable 1.0 release first. Or sorry, 6.0 release.

[SPONSOR MESSAGE]

[0:37:53.6] JM: DigitalOcean makes infrastructure simple. I continue to use DigitalOcean, because of the low friction and attention to user experience. DigitalOcean has kept the experience simple and I can spin up a server in less than a minute and get high-quality performance for a low price.

For an application that needs to scale, DigitalOcean has CPU-optimized droplets, memory-optimized droplets, managed databases, managed Kubernetes and many more products. DigitalOcean has the flexibility to choose the right instance for the right workload and you can mix and match different configurations of CPU and RAM.

If you get stuck, DigitalOcean has thousands of high-quality tutorials, responsive Q&A forums and a customer team who treats customers respectfully. DigitalOcean lets developers focus on what they are building.

Visit do.co/sedaily and receive $100 in credit over 60 days. That $100 can be put towards hosting, or infrastructure and that includes managed databases, a managed Kubernetes service and more. If you want to get started with Kubernetes, DigitalOcean is a great place to go. You

can use your $100 to start building your distributed system and you can get that $100 in credit for free at do.co/sedaily.

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

[INTERVIEW CONTINUED]

[0:39:29.1] JM: On the subject of routing, whenever I'm using a native app, routing always feels so much smoother than when I'm using a browser-based application. Why does routing and state transition for native apps feel so much more smooth than if I'm on the browser?

[0:39:49.4] RF: Oh, probably because React Router is slow. I'm just kidding. What's funny about a web app is instead of a website, I guess, is that you're actually an application inside of an application. A native app that you get the whole screen, you get everything. You can even control the color of the status bar. In a website, you don't even get to control what happens when they swipe from the left side of the screen, or when they start scrolling. What happens to the UI on iPhones anyway? The whole UI shifts around when you start scrolling. The address bar shrinks, the buttons of the bottom slide away and then you scroll up, then everything shows up again. You still have the control and swiping left to navigate back on native, on an iPhone you can't do that on the web, because that is already built in for what the browser does.

There are interactions that we just – they don't let us do, because there's no way to say, “Hey, web browser, get out of here and let me do my own gestures.” Then yeah, I don't know. Well, browsers just aren't as good at animating these devices for some reason. Like get an iPad pro, I got a big – I got one of those big iPad pros. I don't know what they're like, 11 inches or something.

[0:41:05.5] JM: I don't have one. I have an old bad Kindle Fire that I don't use anymore, but I have not really – I've not been subsumed into the world of tablet computing.

[0:41:12.9] RF: Yeah. I've got this big iPad pro, which is actually a great digital audio workstation with GarageBand. It's really cool experience. Anyway, its screen is basically as big as my laptop screen. You go into the iBooks Store, which that's as big as a screen that our

browsers normally have on our computer and it's got some great animations and great interactions. I start goofing around trying to build something like that. I don't know why, but web browsers just don't animate that stuff as smoothly and I'm using straight-up CSS transforms and should all be hardware accelerated and it just doesn't have the same feel.

I don't know. I'm with you. I have that same question. Why can't we make those animations feel that nice? However on native, it is very difficult to deep link. That's a big topic on native. How do you deep link? It happens all the time, right? You get out of an app and then you come back in and you're not where you wanted to be. On the web, we don't have that problem. Deep linking is – that's just what we are. That's how it was designed. Maybe it doesn't animate or feel as smooth, but we do have some stuff that's built in that I think is pretty cool.

[0:42:20.4] JM: Continuing the subject of various things related to React, why is TypeScript used in so many React applications? Why is TypeScript useful?

[0:42:31.1] RF: I don't know. You're talking to a guy who's never used a type language before.

[0:42:35.0] JM: You could tell me why it's not useful then.

[0:42:39.5] RF: We actually this week have just begun the rewrite of React Router into TypeScript. Last week, we have a project called reach UI and we just switched that over to TypeScript as well. I'm not a TypeScript hater. TypeScript is pretty cool, because so the reason that we're using it in our libraries is not so much for us, but for everybody else. If you install React Router and you're using TypeScript, it's like half of people building React apps now. At least half of our clients seem to be using TypeScript.

If you're in the React game and you're building shared components, you can't ignore TypeScript. We were doing our own types just two separate files, but we weren't actually authoring in TypeScript, but then they get out of date, or they might be wrong, which is a risk there, so we were like, "Let's just do it. Let's just write it in TypeScript." Reason number one is because everyone's using – or not everyone. A lot of people are using TypeScript and we want our types for them to be reliable and for them to know that they're going to be correct. They're not just some afterthought. That's first reason we did it.

The second reason is the intellisense, the hints, the popups when you're writing your code are really cool. We document our APIs, we have all these different types of properties that our components can take and people don't read the docs. If you use in TypeScript, you start typing a route or a menu button or a combo box and you get this nice little pop-up of exactly what props it takes and what those types are. That is huge for people. If you haven't done it, you don't quite get it. You're like, “Oh, no. I know the React API. I'm fine.” You get into a code base with a lot of shared pieces, it is really nice to be able to know what types it accepts and if you're doing the right thing.

[0:44:23.9] JM: The application architecture for something built in React, it's evolved as new primitives have gotten introduced in to the programming framework. We've talked about hooks, we've talked about Redux. There's also Suspense. Suspense is a newer primitive that can help with asynchronous data loading. How does Suspense improve the application experience for the end-user and what does the programmer do to use Suspense?

[0:44:54.5] RF: They've been goofing around on Suspense for a long time. It's really promising I'm really excited about it, but it is still yet to be seen what that's going to shake out to look like, or be. The idea is in React when you change state, you click a button, you click a link to a new route and the screen changes. A lot of time, you need to go and fetch some data. In React, it's not going to wait for that data. The new screen shows up. We've all used these apps where you click something and then you get a face full of spinners. Then you get this really choppy loading experience, as all these different spinners finish loading their data and now you've got a page. With a slow network, it's not terrible, right? You click, you get some immediate response from the UI that your click worked and then you get a bunch of spinners and then if you wait a second or two, or five seconds, it's fine. The data shows up and you're not too upset. The app felt responsive the whole time.

The problem shows up when the app is – or when the network is fast, because when the network is fast, you click a link and then it's like, some pages with their spinners and stuff, it's like, I don't know. It's just it's so bouncy and so flashy, so quick. Within 500 milliseconds, you see several things show up and then disappear and then new things show up. It just feels

terrible when you have a fast network and synchronous rendering is I guess, what we could call it.

To fix that problem, you have to move all of that data loading logic higher and higher and higher in the app, until maybe you're at the very top. You don't have a whole lot of control over when to finish that transition to the next page. You can move all the data up to the top and always wait for all of the data. You click a link. Let's just keep the old screen on the page and just sit here until all the data shows up and then go to the next page.

That'd be great for a fast network, because you're going to click a link and you'll feel like you get there pretty quickly and you weren't sitting on that page. For a slow network, that's terrible, because now you're just looking at the old page for a really long time. Maybe you got six pieces of data and four of them are there and they're the four that matter. The other two are just some who knows what. It would be nice to be able to transition to the next page if you had those four pieces of data, even though you don't have the other two.

Suspense lets us bend all of those trade-offs. Instead of having to move all of our fetching code all the way up to top, you can keep some of it closer to the UI that needs it. You'll still want to probably move some of it up, but that's a different conversation. Then Suspense will allow you to adjust to the network. If the network is fast, suspense is going to let you wait for all of that data, so that we're not flashing spinners at people. If the network is slow, maybe we'll hang out on the old page for a little bit after you click a link and then we've got four of the six pieces, it's been two seconds, let's – or one second. Let's go to the next page and show them the content we have, but then have spinners up for the content that we don't.

It's really a set of tools that are going to help us declaratively orchestrate this stuff. It probably sounds like you have to do a lot more than what you really need to do. You don't have to specify okay, these pieces. I'll wait for these ones. I won't. Just by placing some components in the right spots, we'll be able to bend all those trade-offs and be able to give a great experience for people with a fast or a slow network when you've got asynchronous stuff.

Another thing that's cool about it is you can force the order of asynchronous things. Imagine a bunch of images on the page and a bunch of tiles, let's say. When all those tiles come in the

wrong order, it feels weird to the user. There's this thing called suspense list and that will allow you to say, “Hey, you can show loading indicators on these things, but don't do them out of order.” If you've got 12 tiles on the page, if the 12th one loads it's not going to show up until the 11th one loads. It gives this really great experience where they all load in in order, instead of the choppy whack-a-mole feeling.

Yeah, just lots of little tools. Not lots. A couple little tools to help us orchestrate the asynchronous behavior that we have in a user interface to try to get rid of the choppiness, get rid of too many spinners and get rid of the whack-a-mole feel when a page loads up.

[0:49:28.2] JM: There's a large market of people who are teaching JavaScript and people who are teaching React, more specifically. Has it gotten too crowded? Are there too many JavaScript educators at this point? Or has there never been enough JavaScript education and the market is just starving for more information about JavaScript?

[0:49:53.2] RF: You know what cracks me up is when somebody makes a blog post or something, or tweets, or whatever and then somebody else is like, “Yeah, this is just computer science from 40 years ago thing. We already knew this. You think that you're the one coming up with this new idea or this new programming language, or this new framework? It’s just reinventing ideas that we already knew.”

That always cracks me up, because what that means is that the programming community has failed to teach what it has learned to the next generation. We're not going to reinvent those concepts if they were continually taught from more than they were discovered. Yeah, I think there are lots of people that are amazing programmers, there are lots of people that are amazing communicators in the Venn diagram, the intersection.

I think there's a lot of people that have both of those skills. We just haven't really rewarded the communication side very much. We have this stereotypical idea of the hermit programmer, who just wants to be left alone and has no social skills. It's not how it is at all. There's so many interesting programmers who can communicate well too. I think we're just now seeing that with the Internet, we're able to incentivize people to actually teach what they've learned.

I don't feel like it's crowded. We're growing. We're used to just be Michael and I and now we've got some employees, we got some contractors. I think, we have a team of eight people now running our company, involved in one way or another and we're still growing. No, it doesn't feel crowded to me. One of the things that I do to check, to try to estimate how many people are using React or Angular or Vue, or anything is to look at the Chrome's – the developer tools for React on the Chrome – what's it called? The Chrome store. There's 2 million people who have the React developer tools installed. 2 million. That's a ton of people and that's just React. I think there's plenty of room.

[0:51:52.8] JM: It's true. It's really hard to benchmark how many developers there are in the world.

[0:51:57.1] RF: Yeah, there's tons. There's tons. We mostly do training for companies in the US. Although, we got people go into Ireland and Australia and shoot, where else? We got four international ones coming up here soon. We're mostly in the US. Our online stuff, our online stuff is mostly outside of the United States.

It's easy for me as someone from the United States to just think about that's the whole world when it's totally not. Yeah, online, man, you can reach everybody and there are tons of people out there wanting to learn how to program, or get better at it.

[0:52:31.6] JM: All right. Well, Ryan Florence, it's been really great talking to you. Thanks for coming on Software Engineering Daily.

[0:52:35.9] RF: Yeah, thanks for having me.

[END OF INTERVIEW]

[0:52:46.6] 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.

[END]