EPISODE 1075

[INTRODUCTION]

[00:00:00] JM: Over the last 5 years, we development has matured considerably. React has become a standard for frontend component development. GraphQL has seen massive growth in adaption as a data-fetching middleware layer. The hosting platforms have expanded beyond AWS and Heroku to newer environments like Netlify and Vercel. These changes are collectively known as the JAMStack. The changes brought by the JAMStack, it races the question; How should an app be built today? Can a framework offer guidance for how the different layers of a JAMStack application should fit together?

RedwoodJSis one such framework for building JAMStack applications. We've done previous shows on Gatsby and NextJS. RedwoodJS is another framework that is approaching this problem.

Tom Preston-Werner is one of the creators of RedwoodJS. He's also the founder of GitHub and Chatterbug, which is a language learning app that he's currently working on. Tom is a prolific person in the world of software engineering. He joins the show to talk about the future of JAMStack development and his goals for RedwoodJS.

If you have an idea for an episode, whether it's about a company or a project you're working on, we would love to hear more about what you're building or what company you're working at. You can go to softwaredaily.com and submit a topic. We're always looking for good ideas and you can also support the show by becoming a subscriber and you can get ad-free episodes by doing that. Go to softwaredaily.com for all that information, and thanks for listening.

[SPONSOR MESSAGE]

[00:01:55] JM: JFrog Container Registry is a comprehensive registry that supports Docker containers and Helm chart repositories for your Kubernetes deployments. It supports not only local storage for your artifacts, but also proxying remote registries and repositories and virtual repositories to simplify configuration.

Use any number of Docker registries over the same backend providing quality gates and promotion between those different environments. Use JFrog Container Registry to search the custom metadata of the repositories. You can find out more about JFrog Container Registry by visiting softwareengineeringdaily.com/jfrog. That's softwareengineering.com/jfrog.

[INTERVIEW]

[00:02:51] JM: Tom, welcome to the show.

[00:02:52] TPW: Thanks. Great to be here.

[00:02:54] JM: React and GraphQL have been prominent for about 5 years, and I think of these two technologies as being transformative and having a – Kind of leaving a big mark in web development. How has web development changed since the release of those core Facebook technologies?

[00:03:11] TPW: I think React has finally brought a type of development to web developers that we've been seeking for a long time and never realized we wanted. This is something that I started seeing years ago in my GitHub days. I remember sitting down and seeing this templating system from Google and I think it was called C template or something like that, and what it offered was logicless templates that would allow you to separate your views from your backend and development them in isolation.

This is something, one tiny piece. For instance, one tiny piece that React makes available to you that's part of this bigger story of easing frontend development. Then combine that with a way to connect your frontend and your backend with GraphQL and make that just the way that clients talk to a server. It's just a beautiful arrangement.

I see this now with mobile apps. Everyone wants to use GraphQL because it's so natural. If you pair that with something with like React Native, then you're getting kind of the best of both worlds, and React Native is getting better and better every day. We're finally just – I don't know. It's developer experience. There's this new term that I see everywhere now and I don't know why it didn't exist before, but it's the way that I've always thought and it's the way that I think a lot of developers think about how they work, and that's developer experience.

[00:04:39] JM: What are the other important web technologies that have come out in addition to that Facebook set of frontend tools?

[00:04:46] TPW: Well, it's about CSS. CSS has improved radically over the last 10 years. We can do things now that we couldn't do before. We can rely on browsers. I mean, even just the browsers alone have improved so much and have a much better level of standardization across them that we can get more done faster. We spend less time looking up how do make our alignments work in every different browser. They just work now, much better, not all the time, but so much better. I'd say the browser is one of the biggest things, and it's nothing new. It's just standards.

[00:05:21] JM: Explain how these different web technologies fit together. What has this resulted in? What's the overall experience of a web developer today?

[00:05:32] TPW: Well. I think once you pick up the pieces, once you learn the pieces, which admittedly is not the simplest thing in the world, but you think about any discipline that requires complexity, you're going to have to learn your tools, and that's no different in the space of web development. But once you learn those tools, the power that you have and the rate that you can develop applications becomes I think better than it's ever been, and people still lament how complicated it is. But it's because we have a complicated world, and the things that people do now with the web and their phones and their computers and how distributed we are and that you need fast internet for a lot of things, this is just how we interact with the world.

The sites that we build today are inherently more complex than they used to be. The experience is I think daunting at first, but the tools that are available now are so much more powerful than they used to be, and anything that you can dream up in your mind, you can do, and you can do it at a reasonable cost with a small number of people, and that definitely wasn't the case even a decade ago.

[00:06:42] JM: The idea of a full stack framework around the modern web technologies has been percolating for a while. They've been a few attempts, things like Meteor. More recently, you have NextJS. Why hasn't there been a dominant Ruby on Rails style experience for the modern web ecosystem?

[00:07:08] TPW: I think it's been ruling for a while, but I think the real reason is that we've been in this sort of Cambrian explosion of possibility in the JavaScript and TypeScript world for many years, which is really healthy. You see this with any new technology. As it becomes popular, as people start realizing what they can do with it, then you get people building all kinds of building blocks and everything becomes a separated app. Everything is de-integrated, and that's great for experimentation. Everyone is trying everything. But there comes a point where people start to fatigue on everything that's new. You start to get different elements that have matured and can now be relied on.

I think just now in the last year or so, you start seeing a lot of the building blocks having reached the level of sophistication that many people are choosing them and yet integration becomes the hard part. You can pick React, and you can pick GraphQL, and you can pick something like NextJS. You can pick styled components and storybook and all of these different things, and you can sit down and say, "Well, I can build a really awesome site out of this stuff. It's really great." But then you need to integrate them all yourself.

I think people started feeling this a year ago. I felt it a year ago and that's about when we started working on RedwoodJS, and it was specifically because you saw all these patterns out there. There was no standardization. There was no full stack opinionated framework that you could use to bypass all of these decisions.

Speaking again to new developers that want to come in and learn these things, coming in to frontend development and JavaScript and saying, "Okay. I'm ready to build this site. What do I do?" and then seeing the list of things that you have to do. That's hard. That's super hard. Something like Rails did a great job in the world of web development 15 years ago saying, "Hey, beginner. Come in here and just get started on your business logic. Just build out your database, your business logic, make your views," and you've got a website, and it's legit and it takes care of SQL injection for you, attacks, and it was beautiful. You could go farther faster.

The same thing I think is happening in the JavaScript world where people now see that we've explored the possibilities and now we need to see integration. Interestingly enough, right now you see not just one, but several competing frameworks being launched within the course of about a month. This has been in the air for a while. This follows the principle of the adjacent possible, which is sort of a theorization of how ideas happen where all of the things that we know you could consider as information that we keep in rooms, and these rooms have doors between each other.

All of the rooms that have information that we know or ideas that we've had have their lights on. They all have doors to each other. The adjacent possible is what lies in the rooms adjacent to the rooms that we currently have the lights on. When we open one of those doors to another room, we turn the light on and that idea has become possible, but only because we've had the ideas that are adjacent to it. This is why you see things like calculus invented simultaneously in two different places in the world by two different people. That's because of the adjacent possible. The same thing is happening in the JavaScript world. The ideas are out there. It's time for those ideas to happen, and now you'll see several different expressions of those ideas.

[00:10:48] JM: We've talked through what happens on the frontend and what happens for the application developer. What about the hosting providers? How have the hosting providers changed and how has that changed the application development?

[00:11:04] TPW: Well, when I started writing Ruby on Rails, getting your Ruby on Rails site on the internet in a production way was almost impossible. It was sad, because you had Rails, which was amazing revolutionary and what it offered, and yet getting it online was still really, really hard. What you see now is something completely different. It belongs to this greater notion that I've been chasing for a long time, which is the idea that you have something like a universal deployment machine, which is something where you have your code, you write your code, and then you push it somewhere. Say, you just commit and push to your git repository, and then some system can pick that up and put it in production, and that's it. The developer does nothing. Has no server maintenance. There are no other steps. It just happens.

You're starting to see providers today like Netlify, like Vercel. I think these are probably the two leading companies in the sector, and all of these of course built on AWS or Google Cloud, Microsoft Azure, etc., but these deployment machines, they're reaching a level where you just push to GitHub. It's picked up by one of these companies and it's online. That is a change that I think is going to unlock huge amounts of potential.

Again, think about beginners, people that are just entering the world of web application development. When you add things like, "Oh! Hey! So you want to build a production website and you want it to be able to scale automatically. Let me teach you about Kubernetes." At that point, I think you've lost. Don't get me wrong. Kubernetes is great, but Kubernetes is not great for people trying to build web applications. It's great for people like AWS or Google Cloud trying to build these massive infrastructural components that would then allow you to build something like a universal deployment machine on top of that.

I don't think people like me with my application development hat on should ever have to think about Kubernetes or even where your servers are. You should maybe be able to pick geographic locations and say, "My users are in Western Europe and the Americas," wherever they would happen to be, or I'm in Australia and my population of consumers is strictly there or maybe you're a developer in Africa and you say, "I don't care about North America, or South America, or Europe, or even Asia. I want to serve my customers here and I want the experience to be awesome." You could just say, "I want great coverage in these countries in Africa," and the deployment machine would be able to optimize your production environment for that specification.

[00:13:57] JM: In this newer hosting world, what role do serverless functions play?

[00:14:05] TPW: I think they could play an even bigger role than they do today. I think we're at the beginning of what serverless functions can do. They're already awesome, but they could be like triple mega awesome. If we remove some other restrictions that they have, then I think serverless functions can literally be your business logic server. That becomes your server. You no longer have a server, right? In a true serverless fashion, you don't worry about a server. Running a server, where that server is running, if you don't want to. All of your computation happens there. Now, that's not possible for all levels of computation right now because of the limitations that AWS and other providers have on their cloud functions, but I think we can there. I think we're on a road to getting there.

Then that's beautiful and that's really what RedwoodJS is optimized for. It's optimized for you to say I have a GraphQL client, I have a GraphQL server. On the server, I just want it to run wherever. I don't care. I want to put it in the cloud. If you do that on serverless functions, then you can get geographic distribution very easily. You get automatic scaling up to whatever level you want geographically or otherwise, and that could be the fundamental building block. You could build all the pieces of computation on that if you wanted.

[00:15:27] JM: Let's continue talking about these different pieces of web development. We will get to RedwoodJS. How does GraphQL fit into the modern architecture of a full stack JavaScript application?

[00:15:41] TPW: In my mind, you don't have to do it, right? You don't have to have GraphQL as an intermediary between a frontend and a backend. You could have a custom RPC protocol that you hide and do things transparently, and some of the Redwood competitors are doing, and that's super interesting. But to me, the beauty of having GraphQL as the intermediary between your frontend and your backend is twofold. Number one, it allows you out-of-the-gate if this is how you're thinking. If GraphQL is just an underpinning of how you think about building your full stack application, then you automatically get multi-client capabilities, and this is becoming more and more common in our world, right? You want to build a website today, but tomorrow you want to add a mobile client, and then maybe the day after that you want to run this software in the command line and you want a commandline client.

Instead of building the website, and this is how it happened in Rails today. You'd build the Rails website and you'd have your views and they'd be tightly coupled and that would be really easy, and then later on you'd say, "Okay, I need my second client, and Rails is kind of – I can kind of make it be a RESTful API endpoint too. So I'm going to do that." Then you have a third client and you start to feel the limitations of REST. You want things to be more efficient, especially when it comes to mobile, and you start saying, "I want to have GraphQL." So then you build a GraphQL API on top of your Rails app and there's some decent software to do that. That's not horrible, but it's duplicating the effort that you've already made in your business logic and it's also making it messier. It's probably violating principles of isolation of who is responsible for what in your backend, and it’s just more code to maintain.

The way that I think about it, is that GraphQL becomes the one and only mediator between your frontend in your backend, and that your backend is literally just a GraphQL API. If you think about it that way, then your website becomes a client for that and your mobile app becomes a client for that, and your commandline tool and the thing that you install on a kiosk at the mall and the app that you build for various automobiles, it's going to show up on the dashboard. All of these things are just clients for the same GraphQL API, which allows you to focus all of your efforts into building a beautiful and functional and usable GraphQL API not only for yourself, but should you want to open that up and make it a first-class citizen for other people to use and integrate your backend, then that's just natural. Then you can just do that.

Then point number two is that I think it really enforces better design of your application, because by decoupling your frontend and your backend, you're forced to think about those boundaries and who is responsible for what. When you build your frontend with React, you get this really natural way of doing data fetching and the views become responsible for their own data and they fetch that via GraphQL, which is very flexible, but also is easy to navigate.

For instance, when you have a new developer come in and they want to understand where the boundaries of the software are, all you have to do is say, “Here's a GraphQL API. You can see all the things that we do with the client and you can see exactly in the backend where all of those things begin and what code they call,” and it becomes very traceable and instrumentable. Think about the logging that you have when you have a GraphQL API. You know everything that's happening in a very fine-grained way without having to sort of sift through the haystack of data that is a set of log lines around what webpages are being called and what code those things might execute.

[SPONSOR MESSAGE]

[00:19:41] 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 CONTINUED]

[00:21:30] JM: What about the database? Describe the ideal database for a JAMStack application.

[00:21:37] TPW: I think an ideal database to me is going to be – It really depends. I’m going to fork this answer into two parts again. On one side, I really want a relational database. I want something that I can store my users in, and information about them, and settings, and all of these kinds of things, anything that they need to own and correlate across different users.

At the same time, I want probably a NoSQL database that I’m going to use for document storage. That can be a very convenient way to store things that I'll never join across other records and it's just really convenient. It can be very fast. It's easy to distribute. I probably want both eventually. It's hard to find any database that does both, but I think that's fine. I think each one you use for their best purpose. Then if you can get easy access to both of them, then your code on your server-side could be very clear about which data it's pulling from where and what is the key that correlates them one to the other.

I'll start there, and then I'll say the second point is that in my envisionment to have a full stack application development can work. I talk about a concept I call edge ready, or edge readiness, and that's the idea that eventually, and we're starting to get there, all of the pieces of your backend and frontend can be distributed around the world.

For instance, in Redwood – I'm going to give you snippets of Redwood, because this is just what's on my mind. This is how I think about these problems. In Redwood, for instance, you can put your frontend client on a CDN because it's just a React application, fully statically deliverable. You can put that on CDN and deliver it on the edge. With Lambda, you have the potential to distribute your business logic and put it also very near the edge. Now, that's not going to make a lot of sense unless your database is also distributed and put on the edge. At least you want read replicas placed near the edge. But if you can do that – And there are databases that can do that. Not as many as I wish there were. But if you do all three of those parts and you can get each of them near the edge, then you have a fully distributed application, at least from a read perspective. Writes are more difficult, obviously. But that's a really powerful thing, the level of scale that that allows you.

Right now it's very easy with a CDN to the frontend and with Lambda to do the business logic. The databases are the more difficult part. My ideal database is built from scratch to be distributed, that that's just the core concept of how it works. You'd see things like FaunaDB that are really embracing serverless and embracing this idea, and I'm super excited to see where they go. But it's non-relational, and so it has some limitations because of that. There are other companies like YugaByte that is creating a distributed relational database solution that's Postgres compatible. That is super interesting, and others have done this before, Cockroach. That's what I want. I want some relational. I want some NoSQL and I want it distributed around the world. I need at least three slaves. If I can get my writes distributed, then even better.

[00:25:07] JM: Do you have any perspective on why those newer databases are – For the purposes of the average JAMStack application, why are those newer databases any different than Mongo?

[00:25:18] TPW: I am not an expert to say exactly why they would be better.

[00:25:24] JM: I actually think what has happened is somehow Fauna has inserted itself as the JAMStack database. For most people, I don't think it matters.

[00:25:35] TPW: Yeah. Well, I think there're some interesting concepts that they have there and that it was built for distribution from the very beginning and the characteristics that you get from those distributed writes is very powerful. Again, I haven't used a lot of those services directly. My life has been primarily in the relational world and we've made relational databases work extensively. At GitHub, it was primarily that. Other than the git storage itself that eventually went to a distributed hash table essentially. We never really used a lot of NoSQL stuff. Again, my expertise is limited, but the advantages of having that type of NoSQL storage I think are very powerful, but I’m not going to be an expert to be able to defend or rag upon any specific distributed NoSQL database at this time.

[00:26:26] JM: Yeah, and I have respect for all those distributed consistent, globally consistent database systems, and I just try to do the differential between them, and I guess I'm not an expert in it.

[00:26:43] TPW: Yeah. I've heard very good things about Atlas. So I have not used it myself, but I think there is – Again, there's a lot of power there, right? These are all tools that work for better or worse in different circumstances and it's really nice to have options.

[00:26:57] JM: Totally. What about the build tooling in the average JAMStack application? Tell me about the state of build tooling.

[00:27:07] TPW: The build is a really important phase for a lot of JAMstack applications. That's how you go from the code that you write to the generated code or the rendered HTML pages or whatever your strategy is, and it gives you a tremendous amount of power to turn your code into something else. In Redwood, this happens a little bit differently than in what you would traditionally consider a JAMStack thing right now. I'm sort of intentionally pushing the boundaries of what one might consider JAMStack.

JAMStack right now, for the most part, you're thinking of a content site that's going to have maybe a lot of articles and you might integrate with Stripe or Shopify or something to sell things or you will integrate with something like Algolia for search, and that's kind of the whole thing. Your database – You're not generally using a live database that you've created yourself for many of them. That was sort of the appeal of the JAMStack.

At the begging of it was, "Don't worry about a production database. Use these third-party services. They'll do this for you and you stay more protected. You don't have to deal with it. It's the happy path." But the deployment methodology and the developer experience of the JAMStack to me is aspirational. So bringing the full stack capabilities to the JAMStack experience is what I am really chasing with Redwood.

I'm sorry. Can you say the question again? I got deep into this answer and now I've forgotten where I started.

[00:28:34] JM: No. No problem. I was talking about the build tools, but we've talked around the contours of RedwoodJS enough, and now I want to just get directly into it. RedwoodJS is a framework for building full stack JAMStack applications. The JAMStack itself, this began as a concept for building what were originally called static sites. Tell me about how the static site model grew to encompass full stack applications and how RedwoodJS reflects that.

[00:29:08] TPW: Yeah, many years ago I wrote Jekyll, which was one of the very first static site generators, and that was awesome. Then we wrote GitHub pages to make it easy to put sites like that online by just pushing to a GitHub repository. I think the kernel of that is very powerful. That concept of push to publish is kind of what we called it at the time. Push to your repository to publish to the Internet. That idea took a long time to really mature, but then services like Netlify came out and said, "This is worth more than just doing for random personal blogs and open-source project landing pages," and said, "What if you could have a production-ready version of this?" That I think really kicked off a lot of excitement in that space.

Once you do that, once you start saying, “Use these methodologies of pre-rendering pages. Building them at build time and then pushing the result to the Internet so that they’re static.” Getting back to the roots of how we all began this amazing web journey by just authoring static HTML pages and getting them online, that's really appealing because of the performance and the security that you get from that approach.

Anytime you have building blocks, developers love the take those building blocks and then keep improving them and building things that you never even really imagine were possible with those bits, and then you get ideas from that and then you keep going. The JAMStack is no different than many instances of this journey and it leads you to people now starting to say, “Well, why can we do this for content sites where we’re pulling in articles and content from a CMS and then baking them into static pages and deploying those with a single push and getting them distributed on a CDN? Why can we do that, but not build full web apps that way and take advantage of a lot of those same characteristics while still increasing the advantages?” Again, it’s just pushing what's possible. Opening those doors to the adjacent possible, and that's how we got here, right? It's just that realization that you can push to a git repository, publish it online and you have a build step.

[00:31:38] JM I don’t know how much you've looked at the other prominent frameworks like NextJS, for example. Do you have any descriptions for why or how Redwood is different than NextJS or how it contrasts in opinion?

[00:31:52] TPW: Yeah. NextJS is amazing. It's been around for quite a few years. It has a high-degree of polish, and you can do really awesome things with it. It's really a great piece of software. For Redwood, there were a few things that I wanted to do differently. Routing, I wanted to be done in a different way. Just being able to have full control over that layer of the framework felt important to me that it wasn't going to be a battle between us and the NextJS authors and the things that they wanted to accomplish that might be different than what we wanted to accomplish in order to do that. There’re just things about how you do things. How code is structured? Where you put your files? It wasn't even necessarily huge things. It was the kind of a larger collection of smaller things.

I have no criticism of Next, really. It's more that I wanted Redwood to be its own thing and we needed that level of control to free our minds in a way that we weren't just duplicating the principles of an existing framework, but that we could be unfettered and follow Redwood where it needed to go, especially from a developer experience perspective. That's also why we’re writing our own router, right? Which is a potentially stupid thing that one would do when you have such great routers also available. But Redwood has never totally been about using all of the pieces that are available. Although we do leverage a lot of the pieces that are available and that gives people a lot of comfort around how they build things. But if there are fundamental things that we want to chase around developer experience, then we will build our own version of it. That means that Redwood will look a little bit different than Next. It's optimized specifically for full stack applications, where Next really started as a proper JAMStack sort of a thing and is optimized for that and has kind of grown from that. We’re starting with the perspective that everything is integrated perfectly through this stack. So you assume that your frontend and your backend are going to talk via GraphQL. You assume that your router is going to allow you to have these page files that are going to go in a certain place and be named a certain way and that you don't have to import them into the route’s file, because it’s irritating and pointless. There are all these little decisions that we think add up to a level of polish and integration that is going to be hard to duplicate with some of the existing framework-ish pieces that are out there.

[00:34:36] JM: One piece of RedwoodJS is the cell concept. A cell is a declarative system for data fetching. Describe how a cell works.

[00:34:46] TPW: A cell is an abstraction that allows you to do your data fetching in a more declarative fashion. It's a common pattern in the React data fetching world. Say you're using Apollo, which is what we use for Redwood, and you have a frontend component, you have a React component and it wants to do that fetching. You’re going to call Apollo, use query or something similar, and you're going to get out of that some variables. You’ll get a loading bBoolean. You'll get some data. You'll get some indication of whether it was a success or failure, and then you need to test each one of those things and then render the appropriate output for that case, and it ends up with looking very imperative and it’s kind of messy and it's not super great to look at. We thought maybe we can do a little bit better. Maybe we can make this a little more declarative.

In Redwood, if you use a cell, which is under the hoods essentially just a higher order component that wraps a file that exports certain named constants. For instance, you export a success constant, that is a React component that's going to receive props that represent the data when a successful data fetch happened. Then in that React component, all you're thinking about is the success mode. What does this need to do when I have the data? So just take the data and render it just like you kind of want a React component to do.

Similarly, if you export a loading component, then we'll call that for you during the loading state, and in that loading component, all you have to do is think about what should my component look like when it's loading? It really cleans up the code a lot. It's not – I mean, it's a small amount of magic and it's really nothing particularly fancy, but I think it speaks more to our philosophy. How we think about things and how we really want to clean up the developer experience? It also allows us to get into the data fetching flow in a way that will allow us to continue optimizing how data fetching works.

Right now, one of the problems with GraphQL is what's called the waterfall problem in data fetching. So that happens when you have some nested components. Let's say you have the outer component that is going to do some data fetching to get some user information from a user profile, for instance. Then within that component, you have nested a component that is going to say, "Pull a list of their blog posts," or something. Some activity that they have, and it's going to list those on that same page.

Now it needs to do its own data fetching, right? We have a pattern where components are responsible for their own data fetching and each has their loading state. As you load the page, you get kind of the outer frame and then the inner components will fetch their data and then render. But when you have the components nested, then the inner components rely on the outer components finishing the renders before those inner components will be called. This is the waterfall problem, where each data fetch has to wait until the previous data fetch completed, which is not awesome if those two things didn't need to wait. Let's say each of them only required the user ID to run. Why are you waiting for that outer fetch to happen before that inner fetch runs?

We think that by getting into the data flow in a way that a cell does. From a framework perspective, we can do some optimizing for you without you ever having to think about it. Maybe we can batch those queries up and run them simultaneously by doing some static analysis on the GraphQL queries, and then injecting the right things into the right places during the build step, and that's totally transparent to you, and that happens because of some of these abstractions that we put into place.

[SPONSOR MESSAGE]

[00:39:05] JM: You probably do not enjoy searching for a job. Engineers don't like sacrificing their time to do phone screens, and we don't like doing whiteboard problems and working on tedious take home projects. Everyone knows the software hiring process is not perfect. But what's the alternative? Triplebyte is the alternative.

Triplebyte is a platform for finding a great software job faster. Triplebyte works with 400+ tech companies, including Dropbox, Adobe, Coursera and Cruise Automation. Triplebyte improves the hiring process by saving you time and fast-tracking you to final interviews. At triplebyte.com/sedaily, you can start your process by taking a quiz, and after the quiz you get interviewed by Triplebyte if you pass that quiz. If you pass that interview, you make it straight to multiple onsite interviews. If you take a job, you get an additional $1,000 signing bonus from Triplebyte because you use the link triplebyte.com/sedaily.

That $1,000 is nice, but you might be making much more since those multiple onsite interviews would put you in a great position to potentially get multiple offers, and then you could figure out what your salary actually should be. Triplebyte does not look at candidate's backgrounds, like resumes and where they've worked and where they went to school. Triplebyte only cares about whether someone can code. So I'm a huge fan of that aspect of their model. This means that they work with lots of people from nontraditional and unusual backgrounds.

To get started, just go to triplebyte.com/sedaily and take a quiz to get started. There's very little risk and you might find yourself in a great position getting multiple onsite interviews from just one quiz and a Triplebyte interview. Go to triplebyte.com/sedaily to try it out.

Thank you to Triplebyte.

[INTERVIEW CONTINUED]

[00:41:22] JM: How does a Redwood application use GraphQL? I know Prisma is integrated into the stack as well. Describe what role GraphQL plays.

[00:41:36] TPW: GraphQL is how the frontend talks to the backend. At RedwoodJS application right now, out-of-the-box, ships with two sides we call them. Ships with the API side and the web side. The web side is a React client, a React-based client, and the API side is an implementation of a GraphQL UL server, and that ends up being deployed. If you deployed it at something like Netlify, it will deploy it for you on to AWS Lambdas. GraphQL is how the frontend talk to the backend. You do this data fetching. You make a GarphQL call. You get all the characteristics and ease of use of GraphQL, and on the backend, you're just building out a GraphQL API. These things are all well-known. How we do these things is still evolving. Best practices especially are still evolving, but you have the advantage of all of the tooling that has been built and will continue to be built around GraphQL to continue improving the experience over time. That's where GraphQL fits in, and then I think there was a second part of the question you had as well?

[00:42:38] JM: Well, I was wondering about Prisma.

[00:42:42] TPW: Right. Prisma is the query builder. They call it a query builder more than an OR, because it's not really producing objects. It's more producing just plain old Ruby object data structure of the content that you're pulling out of the database. We'll call it a query builder. GraphQL and Prisma don't in our implementation don't really interact. This is Prisma 2, which is a different approach than they took from Prisma 1 stuff. We should be clear about that.

Prisma 2 is specifically query builder, and right now they support a couple of relational databases, but the idea is that long term they will support many different databases including things like Mongo and Fauna and other NoSQL databases. You'll have the same interface potentially to both your relational and non-relational database all from the same Prisma client.

GraphQL and Prisma don't really interact, but I will speak to how you build out the backend of a RedwoodJS application, because I think it has some interesting bits to it. If you've ever built out a GraphQL API and JavaScript using something like Apollo, it's a pretty common pattern that you define your – You’ll have your GraphQL SDL, a schema definition language that you'll write that represents the definition of the API. Then you'll have resolvers that you write that will map the functions that need to be called to the SDL definitions. Then you need to build out what's in those resolvers.

The problem with this is that it ends up kind of messy. I mean, even doing simple things like merging together your SDL schemas if you want to split it into multiple files. If you want to do that yourself, you’re going to spend like a day trying to figure out how to do that. In Redwood, that's just out-of-the-box. It’s all are configured for you. And all you do is write these SDL.JS files that contain your SDL definition. That's exported as a const, and then we’re going to automatically map that to another file that we call a service file that is going to represent the implementation of that chunk of your GraphQL API, but the interesting bit is that it's done in a way that makes that code reusable from other parts of your backend. I’ll explain how that works.

If you have the query called posts, for instance, then in your services file, we will look for a function that you export called posts, and that function is going to take only the parameters, the arguments that are needed to make that run, of which there might not be any. But this is not how the resolvers generally work. They’re going to get the root element of your GraphQL call. They're going to get the arguments. They’re also going to get the parent, in case you have a nested field, and then they’re going to get an info object that represents other contexts around the call.

The problem with this is that if you write your code this way directly in a resolver file, for instance, this code becomes essentially impossible to reuse from elsewhere. When you're building out your business logic, it would be really nice to think about your business logic as another layer of the API. So you have kind of an internal API. We really liked that idea, that your services would be an internal API that you can use from one service to call into another service. In doing this, you start thinking about your backend as a set of services that are well-defined, and this all goes towards the question of maintainability in software.

From my experience, building large, complex systems always comes down to maintainability. Once you get 3, or 4, or 4, or 10 years in, all of your fanciness is forgotten and the only thing you ever wished that you did was focus more on maintainable code. So out-of-the-box, Redwood is trying to encourage you into best practices around isolating different parts of your backend code to sort of keep you safe when it comes to continued development. So these service files are built in a way that they'll get automatically picked up by the GraphQL server because those functions get automatically mapped. So you just have to export functions that are named in the right way and we will then inject as a global variable the context that you need to use them. So you get the database that you can use in any of them as a top level item. You actually import that explicitly, but then you also get access to context as essentially a global variable in those services files so that you're not having to have it passed to you.

That means that you can call those functions. Let's say you want to get a list of posts from somewhere else in your application. Well, now all you have to do is call into that services file and call that posts function and you pass it maybe some filter statement, and now it's just going to work. It's just going to work the normal way that you would call a JavaScript code. But the beauty is that that same function is being picked up by the GraphQL server and also being used as the implementation of a GraphQL API.

[00:48:08] JM: Right. You've talked through a lot of the different nomenclature that I wanted to get to, services and routing. When I create a new Redwood application from scratch, what exactly is happening and where is it getting associated with the GitHub repo? Is it getting associated with any backend hosting? Because like that's one of the things that I think that is different from the Ruby on Rails generation and this generation, is this generation is a little more aware of certain third-party proprietary services that are nonetheless really important, like things like GitHub and Netlify. When I create a new Redwood application, is it integrating with any of these services right off the bat?

[00:48:52] TPW: It can. It's available. You don't have to. We want to try to be as agnostic as possible while still offering really a high-degree of integration. If you spin up a new Redwood project, from Yarn, you just type Yarn create Redwood – app, and then give it a name. That'll pull down the latest skeleton of what a Redwood app looks like and then do some post-processing on it. Then you have a Redwood app that is essentially ready to run. You spin it up and you'll get a placeholder that says you need to add a route to add a homepage, whatever. But we don't force you to use git. We don't create and commit to a git repository for you, which we could, and we discussed doing. But it felt like a little bit of overreach. Not everyone is a git user, and it's easy enough for you to [inaudible 00:49:43] a git repository and do your first commit, right? It's not a huge burden.

But once you're there, then you are automatically set up for a Netlify deploy. Meaning you'll have to plug in a certain thing into your Redwood. [inaudible 00:49:56] file to deploy it. But out-of-the-box, it's ready to deploy. You'll have to set up your own database right now, because the database side of this universal deployment machine, an idea that I'm working towards, is not yet fully complete. Eventually, I would love for Netlify to be able to spin up a database for you and you're done, right? You create your Netlify project. You link it go your GitHub repository. You push to it. You specified what database you want and some of its characteristics, and then Netlify just does it for you and manage it for you, the same way that they do your lambdas.

I think that's the type of integration that we want, and not just with Netlify. We would love to support every vendor. So we're building out that right now, actually, and we're doing it in a plug-in architecture so that we'll build our deployment strategies as plugins and then other people can do the same, or you can write your own. If you have a custom thing that you do with your own servers and you want to do it in a special way, then you'll just write your own sort of plug-in and put it in your repository or reference it from a package on NPM, and then you're ready to go. We aim for flexibility when you want it, but easy out-of-the-box, just push it and forget if that's what you want.

[00:51:17] JM: All right. Well, we could talk a lot more about Redwood, but I've got a lot of broader ecosystem questions, and one is something that I've been asking other people about also, the intersection of low-code and JAMStack. I think of JAMstack is a very high-level, but nonetheless programming system, and I think of the low-code as fairly low-level, but accessible to a general-interest person. Do you see some kind of intersection are merging between the low-code ecosystem and the JAMStack?

[00:51:56] TPW: I think there's opportunity there. You already see a desire for that I think from the use of CMSs where many of us run into this, where we have this great website and it's beautiful and perfect and it's all in GitHub, and then your marketing team says, “Hey, could we change this title on this page?” and you’re like, “Oh yeah! No problem. All right. I’m going to get you a GitHb account and you just go to this page, and look, you can even change it online, and then it will issue a pull request and I can review it.” Again, it’s just not the type of experience that most people want. You can make it work, but most people don't want it to work that way, right? You want a CMS experience. You want to be able to go in there and say, “Oh! Here is the text. I'm going to edit it. Oh, and maybe I can drag and drop around some images or whatever I need to do in a more graphical way. Then I hit save and maybe it even has a review process, and then it's going to go live on the site and I don't have to worry about screwing up the HTML of the page, because that's terrifying.”

I think the same desire is true of other parts of how this stuff comes together. That's great for content for specific content on specific pages, but what if you want all new types of pages, but you want to integrate with some other third-party service? I think a lot of these things can be abstracted up a level to where the user doesn't have to be a developer and its producing code underneath just like the CMS that's pulling in external data for an article or a landing page, is at the end of the day, just injecting that text and other stuff into an HTML page. You could think of a lot of these low or even no-code things as helping people that aren't developers do things that they were never able to do before. At the same time, helping developers do the things that they would normally hoping a text editor to do, accomplish things more quickly without having to write code. I mean, developers like to write codes. This is not necessarily – I think a lot of these things are more to democratize the building of websites than it is for preventing coders from opening text editors.

[00:54:19] JM: Do you think there will be a GitHub equivalent for low-code applications?

[00:54:27] TPW: Let me think about that. How do you mean exactly?

[00:54:33] JM: You know, like I make my low-code marketplace thing and I host it on low-code hub and people could fork it or make their own marketplace things.

[00:54:46] TPW: I think it depends on how many of these different things there are and how do you use them? Because a lot of these things want to be web applications, and so what does it mean to host the web application on something like GitHub, right? How are you envisioning that these things would be run exactly? I guess is the developer spinning them up and running these things, but that sort of defeats the purpose? Like you kind of want them already running and then it's just like a direct – Then your idea is just a directory with links to services?

[00:55:17] JM: Right. Right. Right.

[00:55:19] TPW: I'm not sure.

[00:55:20] JM: Yeah, fair enough. How will the role of GitHub in software development change in the next five years?

[00:55:26] TPW: I think GitHub continues to dig into extending its capabilities around the lifecycle of development. You start to see things like the ability to do builds on GitHubs now, GitHub actions. That's one instance of taking this concept that's out in the world already and abstracting it up a level and then making it useful for more things than you thought, right? So you could replicate a CI system with GitHub actions. This is always how we've thought at GitHub. It's always been about taking these ideas and pulling them up a level of abstraction to make them more generically useful across different ecosystems.

I'll give you an instance of this as it played out in the earlier days of GitHub. A lot of the early users on GitHub were Ruby and Ruby on Rails developers. We had the opportunity to be package manager. We're like, "If people put their Ruby gems on GitHub, then we could – Technically, it's possible that we could just grab those and become a package repository in addition to rubygems.org, and people could install those, and they could install like anything. You wouldn't have to ever put it on any other package manager if you didn't want to. We could just be a package manager, right? You just install directly from GitHub. That would be cool, right? This was a case of a thing where like, "We can do this. So we should do this," and so we did it and it was kind of cool. Then people started wondering like, "Okay. Why did you just do this for Ruby? Are you guys just hosting Ruby projects? I thought that you were more than that? But I guess if you're just hosting Ruby projects, then – I don't know. I'm a Python person. Do I belong here?"

There were some other problems with it, like it had some – There were some things that were a little bit weird and how you had to use it and how you had specify what the package was. So we sat down and we thought about it and we decided to shut it down, and it was specifically because the idea, that idea of a package manager wasn't abstracted up high enough. It wasn't that we were now a package manager for everything. It was that we were just a package manager for Ruby stuff. So we didn't want to be a niche player. We didn't want to be just Ruby hosting. We wanted be hosting for everything, and that meant not endorsing any specific language in that way.

So we pulled back from that, and then from then that became the strategy. I say with great irony, now that they just bought NPM, I guess. But I think that's more of a service to the developer community than anything else at this point. But I think, in general, that's how it goes. It’s like come up with these great ideas for what people are already trying to do, but do it in a way that allows them to build even more on top of that than what you thought you could do.

[00:58:46] JM: All right. If you are not working on RedwoodJS – By the way, I know you work on a lot of different projects at the same time, but if you are not focused on Redwood, what would you be working on? Do you have any other ideas brewing just in your head or things that you've started to tinker on?

[00:59:02] TPW: Sure! I have plenty of ideas. One thing that we do currently – I mean, I can answer this as there are other things that I'm doing. I mean, I'm running another startup called Chatterbug. That’s for foreign language learning. Check that out. Chatterbug.com. I'm doing that. I'm also running and helping to run with my wife what we call Preston-Warner Ventures, which is our sort of investment firm, if you will. We do angel investments. We do a lot of nonprofit grants around family-planning and women's health around the world internationally. We also have a program that's part of Preston-Warner Ventures called Hypothesis, which is a place where we fund projects that we wish existed, but don't necessarily have enough time ourselves to work on.

Redwood is actually one of these hypothesis projects, even though I actually do work on it, but I have hired several other people to work on it with me. So we have a core team of four people, three of which essentially are working full-time and then I act mainly as sort of project manager around it, but then do coding when I have time, of which during COVID-19 times –I have three kids, and so I have no time. But I mean, there're lots of ideas. I have ideas for a VR space racing game. I want to fund the creation of a book or an online tool/visualization thing/written material around learning quantum physics if you only have an algebra background. What does it actually take to understand the mathematics of quantum physics? Take me there. All I know is algebra. Teach me only the method I need to know in order to fully grok the mathematics behind quantum physics. I've never seen that. That's something that I want to fund.

I would love to also work on – This is the way that I work on things these days, right? When you have limited time, the ability to invest in things, then the way that you work on things is by helping other people work on them. Another thing I would love is more of an official work on homelessness in San Francisco, like why is that such a persistent problem? What are the drivers? Where is the money going in San Francisco? We put a lot of money into this problem and we don't see a lot of solutions out the other end. So just help me erase my ignorance. I would love to fund that.

But I’m stretched pretty thin right now. I mean, I'm getting into climate change action as well. We’ll be funding some efforts around climate change. So I've been doing a lot of reading around that. I love the technology of the energy industry. How can we do better? How do we build out a grid can support wind and solar as a bigger part of our energy production? Will I get into the energy industry someday? Not impossible. It’s in my mind.

[01:02:02] JM: Cool. All right. Tom, well, it's been really great talking, and thanks for making GitHub. It's been very influential all my life. Thank you for all your work.

[01:02:10] TPW: Well, you're welcome. I love doing it. I love building products that make people's lives easier. I love building developer tools and making it easier for people to learn stuff.

[01:02:19] JM: Okay. Awesome. Well, thank you, Tom.

[01:02:22] TPW: All right. Thank you so much.

[END OF INTERVIEW]

[01:02:32] JM: Scaling a SQL cluster has historically been a difficult task. CockroachDB makes scaling your relational database much easier. CockroachDB is a distributed SQL database that makes it simple to build resilient, scalable applications quickly. CockroachDB is Postgres compatible, giving the same familiar SQL interface that database developers have used for years.

But unlike older databases, scaling with CockroachDB is handled within the database itself so you don’t need to manage shards from your client application. Because the data is distributed, you won’t lose data if a machine or data center goes down. CockroachDB is resilient and adaptable to any environment. You can host it on-prem, you can run it in a hybrid cloud and you can even deploy it across multiple clouds.

Some of the world’s largest banks and massive online retailers and popular gaming platforms and developers from companies of all sizes trust CockroachDB with their most critical data. Sign up for a free 30-day trial and get a free T-shirt at cockroachlabs.com/sedaily.

Thanks to Cockroach Labs for being a sponsor, and nice work with CockroachDB.

[END]