EPISODE 1016

[INTRODUCTION]

[00:00:00] JM: When ReactJS became popular, frontend web development became easier, but React is just a Vue layer. Developers often came to React expecting a full web development framework like Ruby on Rails or Jango and they were required to put together a set of tools to build their own framework and satisfy that purpose.

A full stack JavaScript framework has numerous requirements. How does it scale? How does it handle server-side rendering versus client side rendering? Should GraphQL be included by default? How should package management work?

Guillermo Rauch is the creator of NextJS, a popular framework for building React applications, and he's also the CEO of Zeit, a cloud hosting company. Guillermo joins the show to discuss NextJS and his vision for how the React ecosystem will evolve in the near future as features such as React Suspense and concurrent mode impact the developer experience.

Guillermo is also speaking at Reactathon, a San Francisco JavaScript conference taking place March 30th and 31st in San Francisco. This week we're going to be interviewing speakers from Reactathon, and 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]

[00:01:30] JM: When I'm building a new product, G2i is the company that I call on to help me find a developer who can build the first version of my product. G2i is a hiring platform run by engineers that matches you with React, React Native, GraphQL and mobile engineers who you can trust. Whether you are a new company building your first product, like me, or an established company that wants additional engineering help, G2i has the talent that you need to accomplish your goals.

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

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

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

[INTERVIEW]

[00:03:19] JM: Guillermo Rauch, welcome back to Software Engineering Daily.

[00:03:22] GR: Thanks for having me again.

[00:03:24] JM: We're about 6 years into the release of React. How has React change web development?

[00:03:29] GR: I've been saying for a while now that I think broadly the most exciting paradigm shift of React has been moving away from templates into components. If we had to summarize the great innovation I think has been to create a workfl ow for teams, to see the rise of the science systems, to give people greater usability, composition power, and ultimately empowering the frontend developer. I think before, and especially with templates, we're confi ned to servers, server rendering, things that would do like spin up a JVM box and write some templating language and just not care about the frontend as much, and I think React has made people gravitate in the opposite direction. Even teams that were not that fund of JS realized, "Hey, to build a world-class frontend, we'd probably have to use this React thing."

[00:04:25] JM: The development of a React application has gotten easier overtime. What was the boiler plate that was historically needed for starting a React application? How has that gotten simpler?

[00:04:36] GR: That's a great way to put it. I think there was a lot of boiler plate. In fact, when we started NextJS, which was solving the problem of making a React application top to bottom entire experience, we were seeing a lot of GitHub repos floating around that were basically copy-pastes of boiler plates. They weren't providing a framework on an altogether solution. They were like, "Hey, clone this boiler plate and then start making changes. Then you'll divert from the boiler plate at some point because you're not merging changes backend."

We created NextJS to solve that problem exactly. It was, "Okay, React started as a V in MVC." It was kind of like a component specific library and we wanted to create an entire application with React. NextJS kind of became that thing.

[00:05:26] JM: Many applications start out static. You have the term static website commonly used today. Overtime, they become increasingly dynamic. What are the issues that developer overtime as an app that might start out as "static" evolves to become more dynamic?

[00:05:45] GR: Yeah, that's a great way of looking at it. I think going back to the origins of React, you'd start with some HTML, whether it's created by a server or not, and then you'd just throw in your JS. Then we're like, "Okay, let's create the whole thing with JS." In that world, from the frontend world, you think about going static first, because you think about driving the entire app with JS.

I think what's happened over the years is that because of this need and want to use React to drive your entire business or whether it's your marketing page or whether it's ecommerce, whether it's your dashboard as an SPA, we want React to be there. We want that component model to be there. Things have evolved in such a way that they become hybrid. We called NextJS a hybrid framework, because we can say a certain page is fully static, for example. A certain page is fully dynamic, and certain pages are in between.

[00:06:41] JM: What are the other scalability issues of React applications that develop overtime as an application grows?

[00:06:47] GR: Well, going back to this idea of pages, one of the things that React developers – And in general I would say everyone that makes this transition to frontend heavy JS architectures faces. One of the first ones you face is you start with this idea of a monolithic app, a monolithic bundle. In fact, the word SBA, single page application, kind of already is lining into that. That really doesn't scale well because you start shipping a lot of JS just to drive what you think initially is, "Oh! It's going to be one page, or maybe a handful of pages."

Then as you start adding features, that bundle of JS becomes kind of very hard to manage and it takes very long to download. It takes very long to parse, compile, evaluate. It takes long to hydrate. Kind of one of the downfalls that we saw early on with the kind of frontend fat client or JS heavy frontend movement was we're spending a lot of time looking at spinners. We're spending a lot of time in what Chrome engineers, for example, call time to interactive.

When you go to a website, well, you want to A, see the content right away, and B, you want to interact with that content right away. With React, that kind of became a problem early on and this is one of the things that we spent a lot of time in engineering with NextJS fixing, which is can we have React as the engine but also have very fast page loads? I'd say this is probably one of the most important challenges that people face when they make this transition, is how do I retain that really blazing fast page load? How do I make it interactive right away and it still retain all these great developer experience primitives?

[00:08:36] JM: As you mentioned, the rendering for a given webpage, a developer has a lot of options today and NextJS accounts for some of these different options, and I want to get into that. But first, the motivation for different options of rendering. If server-side rendering, if client-side rendering, you have some options in between. You can use things like service workers to get some flexibility there. Describe the different range of options for how a developer can render a page.

[00:09:09] GR: Yeah. This is actually another one of – Going back to your question of like what are the difficulties that developers face. I would say this is a pretty big one. How do I render? Do

I use fully client-side? Do I use a static-side generation? What do I do at build time? What do I do at runtime? I would broadly say that there’s a very large category of pages where you as a customer and a user of the web, you want to go and immediately get the content. The best way to do that is to give the customer HTML with a critical CSS inline.

We accounted for this with NextJS with our support for static-side generation. Even though originally NextJS was all about, “Okay, you define your get initial props data fetcher and we’ll do server rendering.” We looked at the problem and we realized, “Well, there’s a lot of different ways of tackling this problem and one fundamentally strong and very scalable way was what if we just produce HTML at built time?”

The categories of pages where that’s a really great fit ranged from marketing landing pages, things like logs where, again, you go to a blog post and if you think about it from the perspective of an SPA or a single page application you would think, “Oh, I’m going to see a spinner, then I’m going to see a skeleton, then the blog post content would come in,” and that’s a very rare sight. I don’t think anyone in the web is used to that kind of experience.

Instead, not only because of making crawlers happier, because crawlers are still today like Google will execute JS, but of course we’ll make them happier if they can crawl the entire site faster and not execute JS. It’s also for the end user’s sake, where you go to a page and you just want the content fast. That’s why we’re calling it increasingly this hybrid solution, because we realized like with that ability to generate content in pages at build time, you get this incredible benefit of great performance, CDN distribution of your content where you’re not confined to routing the user always to a single server.

But then, okay, how do I go beyond? This is kind of the next problem that comes up where like users are used to very, very long build times sometimes. How do I go beyond that? This is where I think Jamstack has offered a very compelling solution where you can complement what you’ve generated at build time with client-side JS.

A really interesting pattern that we’ve seen is, “Okay, you can generate sort of the –” Going back to the Jamstack idea of you can generate the M, the markup, of your pages. NextJS offers you this multipage system where the markup of different pages is going to be different. Then you boot into client-side JS to complete the experience.

[00:12:01] JM: To give a little more detail on that, the term hydration is often used in the same context as this rendering process. Hydration refers to fi lling an object that has been instantiated or exists in memory, but it doesn't have the data yet. So you want to hydrate it with the data. How much control does a developer actually want to have over that process of component hydration? What defaults do you want to give them and when do you want to give them options to hydrate with perhaps application-specifi c preferences?

[00:12:37] GR: I believe strongly that developers want full control over the amount of data that comes pre-hydrated, or pre-rendered with a page and I think that's where we'll see a lot of interesting application design preferences stem from. An example is, as I mentioned, when you go to certain categories of pages that have data that is very critical to render immediately, I call this type of page a commercial transaction page where, for example, you get to a product and you want to buy it right away, or you get to the page of your startup and you want to sign up or learn more or contact sales or whatever it is. We want to minimize the time to that transaction to occur.

That's why with NextJS's hopes for static-side generation, you can say, "The markup comes pre-hydrated or prefilled with the data," but then there is a hydration process that happens on the client. This is where the React engine will try to re-conciliate everything has come pre-rendered with event listeners, and trigger animations, and mounting hooks and so on.

What we'll want there is to minimize that time to, for example, if you click a certain button, we went to give you a results right away. That's why it's so interesting to discuss a time for first contentful paint, but also a time to interact. I think those are very important. Going back to, "Okay, where does the developer gain the flexibility?" I think in some cases the developer will be able to pre-generate markup that optimizes the lot for that time to cotnentful paint. Because, again, the famous Amazon rule is every hundred milliseconds you add, you start selling like less. In general, it just doesn't feel right to wait for a content that could have been cached in a public CDN network. That's sort of one of my takes there.

But then on the other hand, there is more specialized types of applications where, for example, the continent cannot be easily cached or there is interesting concerns with regards to privacy or data ownership, right? A great example that I present here is, for example, if you were to build the WhatsApp application, which I believe is already built with React. That page, like let's say we call it /chat or /groups or /threads, that page will probably only come pre-hydrated, let's say, with a data of maybe you're logged in or you’re – Or this is thread list and it’s going to render shell only. Then client-side code needs to boot, in the case of WhatsApp, to actually establish end-to-end encryption, to be able to get the data and render it on the screen or talk to even a mobile device.

In our view of the world, the developer will make this decision on a per page basis or what we call the entry point. Maybe whatsapp.com/, and it should just be a static page that comes pre-hydrated and prefilled with all these data. But then a /chat will be this shell that then requires client-side JS to boot up really quickly still, because, again, we still want to make the user's as happy as possible as fast as possible. But the sort of rendering decision has shifted a little bit and the data is now owned by the user. That means that we probably don't want a server to necessarily touch it or see it or deal with it, because we increase the service for security problems to also exist.

This is the beauty of React. It's proven that it works remarkably well to pre-build HTML and deliver it at scale, but also to build very basic HTML than then boots very rich JS on the client side to create a very immersive application-like experiences.

[00:16:26] JM: I like that example, because you’re citing a reason why you would want to do client-side rendering that's not explicitly for performance.

[00:16:37] GR: Corrects. I think this ones that has become very clear over the past two years to us as we work more with NextGS users that are operating a very, very large-scale, and then security becomes a concern, right? If you're subscribing to a worldview of I only do server rendering. Then you start increasing the surface of complexity of your application, because what happens is let’s say something as simple as error tracking. You now have to deal with the error tracking lifecycle for the server part and the error tracking lifecycle for the client part. Whereas if, again, we go back to this WhatsApp example, they have the ability to now ship the shell of the application as static HTML. It gets downloaded from as close as the user as possibly in an edge CDN network, and then the only security error tracking and debugging context to worry about is a client-side.

Now, if you ever talk to a security engineer, it's a dreamlike situation because JS is running in a sandbox environment, which itself the browser processes sandboxed environment. There is only one piece of code audit and it gets even more exciting to start thinking about multiplatform JS. I truly think that React will get to the point where you write WhatsApp once and you deliver it to four or five major platforms.

Today it's like kind of blurry, like even Facebook, they built web.whatsup.com and they still have native clients. But I think over long periods of time, this sort of amazing economical advantage of you analyze the security of your application once in one context. You write one unified component system. You write one set of data fetching hooks that – This is something I excited about a lot, is that React has made the data layer reusable now. Not just the UI layer. Imagine that you're creating the next WhatsApp. You now have this uniform view of the world. I use one technology, it secures scalable and multiplatform.

[00:18:39] JM: The concepts we've been discussing, how have they inspired NextJS?

[00:18:45] GR: Very congruently, I think NextJS has made a lot of progress in this hybrid view of the world. We know that each page is an entry point into the application. For those who haven't used NextJS, we like to give you the two second story of how to start, which is you create a pages directory and then obviously you add – You're going to add next, React Raect DOM. Then inside the pages directory, every file that you create [inaudible 00:19:10] .JS about .JS, app.JS, settings.JS becomes an independent entry point in your obligation. It's a little reminiscent of how PHP used to work, where like each file becomes an entry point, a page, a URL.

What's exciting that's happened lately is each page can now be compiled smartly by Next.JS at build time to, for example, maybe your index is a fully static page. There is no data fetching at the top level. Therefore, Next.JS will output index.HTML. Maybe you have a blog, which needs to come pre-hydrated with data. So like you have blog.JS. Next.JS will give you a hook to obtain data at build time, and maybe you have, again, the need to, when you're browsing a page, get all the data from the client-side, then you use React Hooks. Then it's kind of this very comprehensive model.

Then if you have a very, very specialized page, you say, “I need to use the server lifecycle here.” Next.JS can say, “Oh, this page only does server rendering.” Because of this flexibility, Next.JS has become appealing to lots and lots of different answers. Whether you’re creating your first page or you’re creating a very massive dynamic application.

[00:20:22] JM: It's worth pointing out that you run Zeit, and I just love to get like what’s your perspective, why would you start it – And Next.JS is a JavaScript framework for people who don’t know. It’ a framework for building full-fledged React applications. Why does it make sense for a hosting company to work on a JavaScript framework?

[00:20:43] GR: That is a great question. The mission of our company at Zeit is enabling a workflow for frontend developers in the entire company to develop, preview and ship. We think of this lifecycle as a true game changer, because typically when you develop, it's very hard to like share the progress of that development with your team.

The way that you use our platform is you install a GitHub app and then every push and every PR and every deploy you make gets a preview. This is something that other legacy technologies or even when you use Figma or things like that, you’re used to like previewing your work all the time, like seeing it live. Zeit gives you this incredible feature of every deploy you make, every change you make to your app gets this preview URL and it gives a workflow for the company to now collaborate cross working on a NextJS app, working on even like a simple fully static website. Then you have to ship that, right?

I think we kind of gotten used to separating all these concerns and the frontend developer does some work on local hosts and then hands it off to someone else to ship it or make it real or make it live. But we think that by integrating this technology into a workflow, you get the ability to make software faster and better.

Something we've seen that's emerged as a consequence is you start working on NextJS app, you of course push it to GitHub, or GitLab or Bitbucket and then you import your repo to our platform and then you start working on software differently, because what happens is – And this is broadly enabled by this frontend heavy applications is that instead of focusing so much in the code, you're sort of focusing on how the actual product works.

For example, when we start working on the new Software Engineering Daily website, we say, "Oh, we're going to use NextJS." Funny enough, if I'm working on it, I have it on local host 3000. But how do I actually bring it to the cloud? How do I actually collaborate with you on building that website? I think what's exciting about this is that NextJS is already in this low code category, where like you don't have to write lots and lots of code to make an app.

By integrating into a neat workflow like this that brings NextJS to the cloud, it almost feels like you're working on a no code tool. It just feels like you're just editing the product directly. This is truly how I think products needs to be built. I think, as developers, we tend to over-index so much in abstraction. We tend to think like what's the best way to – Especially in the React world. What's the best way to store a state? What's the best way to make state scale? What's the best way to introspect state?

But this really changes your mindset. I think, combined with NextJS, giving you some good defaults and opinions, it kind of – And going back to why we think this is our responsibility, is like we wanted to work on your product. That part of the developer you shipped lifecycle develop, NextJS is a great answer there to like become really productive and then just work on your product.

[SPONSOR MESSAGE]

[00:23:59] JM: Today's show is sponsored by Datadog, a scalable, full stack monitoring platform. Datadog synthetic API tests help you detect and debug user-facing issues in critical endpoints and applications. Build and deploy self-maintaining browser tests to simulate user journeys from global locations.

If a test fails, get more context by inspecting a waterfall visualization or pivoting to related sources of data for troubleshooting. Plus, Datadog's browser tests automatically update to reflect changes in your UI so you can spend less time fixing tests and more time building features. You can proactively monitor user experiences today with a free 14-day trial of data dog and you will get a free T-shirt.

Go to softwareengineeringdaily.com/datadog to get that free T-shirt and try out Datadog's monitoring solutions today.

[INTERVIEW CONTINUED]

[00:25:05] JM: From a strategic point of view, do you worry at all about what happened with Meteor? With Meteor, they built a technology, MeteorJS, that was awesome to work with, very popular. But in some ways maybe ahead of its time or just went in a certain direction that ended up being perhaps too opinionated and they had constructed the entire company, an entire deployment system around a framework that ended up not being popular enough to support a company.

Now that said, they've pivoted pretty gracefully towards GraphQL infrastructure, so maybe that's not even a cautionary tale. But it seems like you're kind of in the same neighborhood where you're trying to design a frontend framework and a backend deployment platform at the same time. What are the risks there?

[00:26:02] GR: I think that's an interesting story. Like you said, I think there was a lot of learning involved for them, but also for the community at large. I think one of the primary things that I take from that is the need for focus.

From the very get-go, NextJS said, "We don't care about the data layer," whereas Meteor was all about like, "Oh, we're going to give you the entire solution, including MongoDB in the backend." Going back to like how has NextJS evolved and what we've really learned is that in order to serve the frontend developer, you have to be a true frontend technology.

There is no backend involved increasingly. There is no, again, data layer preference. There is just a tool that'll occupy the last mile of the application development lifecycle. This is kind of what I think the community at large has really discovered with the rise of Jamstack, where, A, you need a very, very scalable model of development and deployment.

One of the [inaudible 00:27:03] with Meteor was that if you were building a single page, you are running – Even in the Meteor cloud, you're writing this container, running a server, running MongoDB, running all these things. At the end of the day, you would go to the website, for example, in an global market and it'd probably be slow and expensive and it wouldn't scale.

Now, we've shifted the equation entirely towards not deploy to the server, but here's the main distinction, is we're deploying to the edge. We're making frontend development scale from an economic standpoint. It's highly available. It's incredibly cheap to scale, and now it's getting really easy and cheap to develop. The primitives for it are very, very simple.

I think another thing that's been really beneficial for us is been companies had already started this move towards decoupling frontend and backend. I think Meteor was stuck in this in between of like we need to be like Ruby on Rails. So we need to give you backend and frontend altogether. But at the same time the industry was moving towards actually this iOS thing is coming up, and this Android thing is coming up. We're going to need a segregated API. We're going to need Rest. We're going to need GraphQL. We're going to need microservices. I think that's a larger trend that we can't ignore. This is a trend that Jamstack so gracefully adds itself into.

The A of Jamstack is API. Literally, you're just writing the JS and the markup to create an API that already exists. I'm a really big fan of thinking about the broad economic landscape, and I think why would I give someone a tool to rewrite the backend that their companies have already been rewriting for years? Everyone's coming out with a Rest API. Everyone's coming out with a GraphQL API. What is the missing piece?

Well, the missing piece is to produce pages from those APIs and to do so at scale, especially now that so many other people are providing APIs as a service. When you think about headless

CMSs, when you think about a WordPress, the WordPress Rest API, the world at large has decided that the API is now the adapter into the data layer of the world.

I think this is why the Meteor to GraphQL transition made so much sense, because now they're saying, "Okay, we're going to focus on answering that side of the problem and win a true focus manner." There's going to be lots of people that want to expose their data and expose their mutations through an API. Now they fit into this Jamstack model wonderfully where you build and deploy to the edge with NextJS and you're actually not writing the API necessarily, right? You're querying an API that already exists.

It's exciting to see, because now there is also backends as a service that give you a GraphQL API or a Rest API. If you think about focusing on the frontend developer like we are, it's looking really good for them, right? They can produce so much awesome stuff without doing any of the work involved in like actually running a server.

[00:30:12] JM: The distinction between the Ruby on Rails world and the world that we're living in today, well, I guess in some cases we are still living in the Ruby on Rails. Plenty of people are still using Ruby on Rails of course, but the direction that I guess Jamstack represents of less monolithic, less tightly coupled workfl ow. Implicitly, there is less strong opinion about how to do things, but you're building a framework. NextJS must have some strong opinions. What are the strong opinions that you're baking in to NextJS?

[00:30:49] GR: Yeah. When it comes to what the customer ultimately wants is build great apps and websites. That's way Ruby on Rails, even though it's not the best solution available, it still has traction, right? People want to build things easily in Rails.init is a great solution. Now, when you run create next app, which is our equivalent of Rails.init, mpxcreate-next-app, you get an example NextJS application.

What you find inside is this pages that can do their own data fetching. The strong opinion that we have in there is that you can define this data fetching hooks, like get static props, get initial props, or maybe not doing any data fetching at all so that you output HTML. Then you have to query an API there. You can then use fetch and get data from your RESTful API. You can use fetch and get data from a GraphQL. The strong opinion has really been around you're building pages and you’re going to want to be in control of the rendering lifecycle as we mentioned earlier, but that's kind of where the opinions end. We want to stay very, very focused in this frontend specific problem.

Now, where things get really interesting is, okay – And I’d like to remind people that when DHH introduces Ruby on Rails, he builds a blog engine from scratch. He builds a whole like backend for the blog including like he shows off the [inaudible 00:32:22] folding feature. He uses active record to save posts, create posts, lists posts, and then he does the frontend bit, obviously, where like, “Okay, I'm going to render to post.”

If you think about that from the perspective of today, most people using NextJS today didn't have any interest whatsoever in building the admin panel of their blog, the engine of the blog. They probably were using a well-established blog engine. In those data fetching hooks of the NextJS pages, you could query the WordPress API. Then I'm sure like your editors are happier to use the WordPress admin panel than they are using something that you put together in a five minutes Ruby on Rails screencast.

I think the strong opinions have been to actually not fall into the temptation of trying to answer the backend needs strongly that Ruby on Rails provided because we’re living in a new world. This is kind of where NextJS shines, is that like this is kind of what the frontend developer is looking for and coupled with our platform for quickly previewing and deploying. You’re actually beating DHH to not five minutes, but you can do it in a minute and get a live URL of your outcome. I think the best that you could do after the Ruby on Rails screencast is I get like – I don’t know, localhost:3000 server running. With this, you're literally pushing your pages to the edge and just sharing them with anybody.

[00:33:54] JM: To what extent has the Jamstack community consolidated around data fetching workflows involving GraphQL? Does every Jamstack application want to use GraphQL?

[00:34:07] GR: I do not believe so. I think the flexibility of data fetching strategies remains incredibly useful. As an example, our dashboard and marketing website and blog and everything, one NextJS obligation fully Jamstack 99.99% of it is a static and I believe we have like a couple serverless functions. When you come to our dashboard, we give you this shell that it begins doing data fetching in parallel with React Hooks actually. We query a bunch of different REST endpoints that we already had built that are the same endpoints that we expose to our enterprise customers when they use an API for advanced deployment needs or the same APIs that we use for to GitHub and GitLab and Bitbucket integrations. We already had this RESTful API and we had to solve the challenge of how do we present the data to the user really, really fast. Hooks actually were an incredibly suitable solution. We do data fetching in parallel. For certain parts of the application, we subscribed to WebSocket subscriptions for accelerating events. It's super real-time, and this is kind of how Twitter.com works. A lot of people think, "Oh! In order for Twitter come to be real-time, for example, like I'm sure they use a GraphQL subscription." No. They use REST and then they re-fetch the data upon certain actions, like focusing the page, coming back to the app, overtime. You can get really, really far to build applications that are real-time engaging in incredibly fast like twitter.com or whatever app you're building with, let's call it good old REST.

I think what we're going to see over the next few years is obviously a combination of GraphQL. I believe we're going to see a return to some extent to just using functions for data fetching where you think about like – I don't know, like use get user session, and that you think of your Hook as a discrete function that does data fetching and the implementation details of that Hook don't matter. They could use anything underneath.

You mentioned service workers earlier, and I that's a nice thing about abstracting data fetching and not marrying yourself only to one way of doing data fetching. Because, for example, you can use hooks that get data from the offline storage of the device or your computer. You can subscribe to changes over time inside that Hook.

I've been really excited about hooks in React specifically because of this encapsulation ability. They kind of look – What's fascinating about hooks to me is that they can look synchronous. You notice that there's is no async/await, but they're the ultimate form of a synchrony. When you use the React Hook, you say get user session, or something like that. Let's say that that's backed by Auth0. You go and you make an API call or it goes through your REST backend or you or you mix a GraphQL query.

Then not only is it then returning the data. It can stay subscribed to the changes of that data point over time. Not only does it not have the await in front. It also goes beyond what a promise can do. It goes into the terrain of what an observable can do, which is you stay subscribed to changes and there's a cancellation function.

I get excited about that primitive, more so than a prescription for a certain protocol, HTTP gateway design and so on. I just like the frontend developer thinking in terms of I need this data. By the way, I need updates to the data over time. That's what's exciting, because like with just React Hooks, you can create the most compelling real-time alive web application.

You asked me about Meteor, and I think that Meteor was trying to deliver on this stream with a coupling to MongoDB, with a coupling to a WebSocket server, with a coupling to a green thread approach to NodeJS. It was like I think that they were right about what they wanted in the outcome, but there were such simple solutions that were actually client-only to get there. We're really happy that like we've been able to get all this goodness with very simple technology that runs in the client.

[00:38:30] JM: Can describe the implementation details of hooks a little bit more? I don't know much you've looked into it.

[00:38:34] GR: Yeah. Going back to that image of like I create my pages directory, I create index.js. What does index.js contain? Well, you export a React component. You go expert default, and let's say div, hello world div.

That doesn't do anything, right? If you run next build, you get index.html and just says hello world. How can I actually bring data fetching to this? This is where hooks are so exciting. The function that returns your JSX, your page, can now say – In this case let's say [inaudible 00:39:07] user whatever equals use is logged in. Now that Hook will return the value of that data point at a given point in time.

When you first call it, most likely what are you going to get? Nothing. You're going to get loading state and then you're going to fire an effect and you're going to go and get the data. But inside the Hook, there is this technology for being able to return values overtime. The Hook can do fetch. The Hook can look at local storage. The Hook can look at IndexDB. The hook can do GraphQL. The hook can do WebSocket. The implementation details don’t matter.

But every time it returns the value, the whole page will – And this is the magic of React. You don't worry about like we did with jQuery, “Oh, when I return from the WebSocket, you said DOM selector, whatever. There is this like mind-liberating idea that when the Hook gets more data, the full thing re-renders. Then you now paint your template with a user being logged in or not.

This is just so simple and so compelling because, again, you might start with a very simple implementation. Maybe you had an API for getting the logged in state or whatnot, but then you can get really ambitious with what that Hook can do. One of the things that we use it for the Zeit dashboard that's really exciting is hooks can fetch data in parallel. The data needs are kind of somewhat co-located with components.

Interestingly enough, something we do that’s really cool is at build time, we know that we query like three or four APIs and then we will hoist those data fetching requirements and we create in the fixed HTML that comes with the page, we have link preload texts. As the HTML begins to be streamed in, the browsers that are parsing it, the link tax will start making fetch requests to those data portions that are then going to be needed by the hooks.

You asked a good question earlier, like what are the challenges that React developers face. One that I found that is really interesting is you have to wait for React to boot up to do all your data fetching. Think about that, right? You know that when you go to /shop/cart, you're going to go and get the cart state from a server. Now all you have is an HTML page that then has to download a pretty sizable JS bundle. React needs to boot. Your Hook needs to execute.

One way of sot of circumventing this problem is because NextJS is building the page, it can now embed in the HTML the data requirements of those hooks later on. We've really fallen in love with this technique because when we got rid of SSR for the dashboard, we actually found that our dashboard became faster, because we started thinking about like what can the browser do if all you have, if all you’re armed with is an HTML page. What can the browser start doing ahead of everything booting up?

We can get data. We can render the shell. We can render some animations and so on. Then when React kicks in, boom! You get all your data. The beauty of this too is that that data because of hooks can update over time. So not only did you get that discrete initial data load, but you kind of accomplished the "Meteor dream" of fully real-time applications with basically no technology just like SmartJS.

[00:42:42] JM: Since we're talking about React internals at this point, we should just talk about the newer features of React. Something I know you had some part in the – At least the conversations around is React Suspense. Suspense gives the developer some control over rendering items that have not yet finished loading some asynchronous data. What are the ways in which React Suspense addresses asynchronous data handling that has historically been difficult to work with?

[00:43:16] GR: Yes. The main idea behind Suspense is – There are intervals of time within which you know that you can do lots of work without necessarily committing a change to the Vue layer. So the best example here is when you're transitioning between pages, you might want to coalesce changes together, and there is an API called Suspense List that sort of enables this to say, "Well, if I have three portions of the page that need data, can we sort of aggregate them within a threshold so that they all render at once and they can provide a much better experience?

That's kind of one of their big product needs that people have had over long periods of time, which is one example of this that we've seen is when you go and load a new page and it triggers some data fetching, let's say that your APIs are actually really fast. You might be used to the experience of like you click on something and it shows something and then immediately changes into something else.

What's interesting about this is that as your company or project evolves, you start getting a really good handle on how long data fetching takes from the server of choice that you have. For example, for our case, because our API's enterprise grade, we look at the P99 times. We basically obsess over how fast are all these APIs responding to ensure great quality of customer service? We actually alert on APIs becoming slower and so on.

When you think about, “Okay, now I'm in the frontend and I’m going to get data from the – I don't get, get projects API.” How long does that on average take and what’s the worst-case scenario and what's the best case scenario? What we found with our own experiences that we can answer that really, really quickly. Sometimes in under 100 milliseconds once the initial page load happened.

What we’re currently investigating is what are the opportunities to like take advantage of this knowledge? Not only the knowledge that we have a data fetching layer that’s really performant, but also that the user's perception of performance is also not as granular as Lake 50 milliseconds, rights? Like a user will feel something as instantaneous if it happens in like, let's say, 200 milliseconds, right?

Suspense in combination with hooks is something we’re researching right now, which would allow us to say, “Hey, I'm going to transition to this page, but let's only commit aggregated changes that wants to provide a better experience to the user.”

Another example of this is when you're dealing with dependencies between components where maybe showing you, let's say, that you're rendering your bank’s dashboard. Maybe you have two independent data fetching hooks. One for getting the balance and one for getting the user information or something like that.

Could you suspend easily between both so that you rendered those two pieces of information at once? Well, a lot of product designers will tell you yes. I think Suspense will become this other tool in the toolset of the frontend developer to coordinate with the product designer on like what is the expectation around what we should present on the screen, what we should block on? What we can do ahead of time as well so that the product feels great.

It's interesting that I mentioned earlier that before we started the podcast, what I love about Jamstack, what I love about NextJS is that it’s really shifting the conversation into how can I build a better product as supposed to like how can I do better state management or what router should I use and things like that? As a result, you start like going deeper into these concerns that perhaps earlier you were oblivious to because you were just happy that you actually were reliably data fetching.

Okay. So now we've solved that problem. We can reliably and fast data fetch. What is next? I think what's next is that we're thinking about ways that we can transition between pages in ways – At least for my perspective, to feel more app-like. What do I mean by app-like? I mean, in a lot of case when you go to a native app and you tap on something, the app has already fetched the data for that. There's no so much an expectation of like all, "Oh, you're going to wait or you're going to render a spinner." No, because they thought a lot about the experience.

Another one is animation. Until today, NextJS provides you a model for switching between pages and you can actually retain some sort of layout between pages as well, but kind of animating a lot is on you. I think what's interesting about Suspense, what's interesting about like where we're headed is we're now elevating the bar of product quality and we're thinking about things like data fetching ahead of time. We're thinking about aggregating data fetch together so that they commit all at once to the screen. We're thinking about animating transitions, and this kind of the next big mile. I think Suspense is still early. We are not using it yet.

One of the things we're looking into now to solve the problem of how do I transition to a page and it already has the data that I need, is we're looking into – just like I said, we hoist the data requirements of a page so that we can preload it even with just HTML. We're looking into applying the same for transitions between pages so that when you're going to click on settings, even when you hover on settings, we can already kickoff the data fetching needs of the settings page. That actually is kind of solving a problem that, as I mentioned already, it's pretty big today, which is like we want instantaneous page transitions, but we want the page to do not have this jitter of like skeleton like move around and like then do data fetching. It seems very promising so far. We're currently experimenting with it. Our data fetching hooks library for those that are curious is called SWR. It's swr.now.sh and it plays really well with NextJS because is literally just React Hooks. This React Hook library is really enabling a lot of this cool use cases in a way that, again, is like data back in Agnostic, Jamstack friendly. You can do lots of interesting things with preloading data. We've built our entire dashboard with it and it's just really phenomenal.

[SPONSOR MESSAGE]

[00:49:56] JM: As a company grows, the software infrastructure becomes a large complex distributed system. Without standardized applications or security policies, it can become diffi cult to oversee all the vulnerabilities that might exist across all of your physical machines, virtual machines, containers and cloud services. ExtraHop is a cloud-native security company that detects threats across your hybrid infrastructure. ExtraHop has vulnerability detection running up and down your networking stock from L2 to L7 and it helps you spot, investigate and respond to anomalous behavior using more than 100 machine learning models.

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

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

[INTERVIEW CONTINUED]

[00:51:30] JM: Just to connect the conversation about hooks and the conversation about Suspense – correct me if I'm wrong. It sounds like hooks are a way of decoupling the application that you write from the way that variables are getting their data, whether they're getting it from a CDN, or an API, or the cache, or whatever implementation details you want to add now or later. Suspense is a way of accounting for whether or not the data has successfully been loaded into your application.

[00:52:10] GR: Correct. They're addressing different needs today. In the future they might merge more closely. For example, one of the options that we have in our data fetching hooks is we can say suspend true or suspense true, I believe, and then we can then link it into the suspense subsystem, which works on this idea of promises that are thrown and waiting on them and blocking on them and so on. But today they are strictly separate. If I had to give you the recipe today for data fetching in a Jamstack application, absolutely you just use a hook. Again, you can get really far with just hooks on all these data needs like caching real-time, subscriptions, pulling overtime, tolerating errors, retrying on errors is sort of what SWR gives you out-of-the-box.

But then we're still dealing with layouts and UI issues, right? As I mentioned, one that happens is that there is this idea of transitions that are not as graceful as they could be, because hooks are so fast. They're trying to like react immediately to every change and they make coalescing changes a little bit harder. In some cases, it's very desirable.

For example, when I used twitter.com, I just care about the time they're loading as fast as possible, right? Then, for example, there is sidebar of recommendations that can load later and it would never "suspend" on both coming together. It doesn't benefit me as a product designer, but there is an alternative universe where, again, there is two components that are doing data fetching and they both need to work in unison.

They could still use different data fetching hooks but they need to kind of all render at once, and I think React can evolve to make this an easier task, like the coordination of blocking. I think it doesn't even need to deal necessarily with data fetching, right? You can suspend on any asynchronous flow, but broadly that's the separation that we have today, and I guess that suspense is still sort of experimental, but that shouldn't stop you from actually doing data fetching with modern techniques like hooks.

[00:54:25] JM: JavaScript is single-threaded. If the React team puts enough effort into building the right APIs and abstractions, can we attain an experience that actually feels multithreaded in React?

[00:54:40] GR: Absolutely. I think the idea of JS as single-threaded is – Well, first of all, it has stopped being true for a while because there are ways of – When you delegate data fetching to a service worker, for example, or even when you use fetch, there's so much just happening behind-the-scenes in different threads. This is similar to how NodeJS is a thread pool for all the heavy asynchronous blocking – Sorry. To turn synchronous blocking work into asynchronous work and delegating off the main thread.

I think the fact that React is doing all the scheduling and it's sort of managing your application will allow developers to even have this like multithreaded world under the hood or even if it's the illusion of a multithreaded world under the hood for free. You're just thinking about components and you're thinking about side effects. I think hooks are the right step in the right direction, because they started including constraints that allow for the React scheduler to do smarter optimizations under the hood.

Ultimately, I think it's also helping platforms like React Native where there is more interop with the operating system. There is more interop with UI components that are not governed by the same system. I am really happy that React is starting to introduce constraints to ensure that over long periods of time, it can be optimally parallel for every step of the painting, rendering, layout, execution.

I think what happens today, frankly, is developers tend to be oblivious to the fact that certain APIs are just slowing down the rendering. The most common that I see is the usage of local storage. Local storage is a blocking API it synchronously reads from the file system.

If your visitor has a hard drive, you're blocking on spinning a physical disk until you can continue rendering. Keep in mind, you can put local storage anywhere. Even because of muscle memory [inaudible 00:56:54], "Oh, you can use a hook with local storage." No. You shouldn't. But the interesting kind of lesson is, as you mentioned, the fact that JS still has all these blocking APIs are easily accessible and then the can invoke them from the main, let's call it the "UI thread" has really noticeable drawbacks even today.

I saw recently a bunch of hooks being open source that use local storage extensively, and that's something that basically shouldn't happen. I think it shouldn't happen from an API perspective. I think one way or another, the engine itself should make this an impossibility, almost like an impossible state, like, "Oh, local storage within the UI thread. Oh, delegated to some other time, to some other thread."

Likewise, I think the web has been pushing forward for asynchronous APIs existing, but I think discipline is not quite enough. In this case, you're prescribed to use IndexDB instead. I think this is something which is broadly a very, very interesting thing to think about and this is also why I get so excited about deploying NextJS through the edge network and networks becoming so fast and 5G and just like sort of like becoming one with a computer. I'm just kidding.

[00:58:14] JM: But not really kidding.

[00:58:17] GR: Becoming one with the network is what I'm excited about. We run this experiment with SWR where – The API for SWR, you can go to swr.now.sh or another similar project, it's called React Query by Tanner Linsley. You basically say, "Use SWR and use as a key any API," like you say like /tweets, and then a function to get that, like fetch. The easiest way in the world to get data is use SWR/tweets, fetch, but you can use any sort of asynchronous function there as the sort of data fetching function.

We built our entire dashboard off of this in our entire website, even when you like go to our marketing pages and on the top right you see your logged in state, we're using a hook called – And we wrap them. So easily hook called like use logged in or whatever.

One of the things we realized is, as I mentioned earlier, hooks are so – I would call them anxious. They're like anxious to return and give you what they have, right? The first time the hook executes, it has nothing. We started thinking about like there are a lot of cases where we could be returning something, especially now that we're equipped with IndexDB.

We're like, "Okay, let's say that you have a list of your cart state for a shopping cart or a list, a wish list," like your saved books in audible.com. We're like, "What if we could just do like use SWR/books and then find asynchronous function called fetch with cache?" so that what we do is we essentially read from a cache like IndexDB or whatever and then fetch the real, like most up-to-date data and then continue to fetch it overtime?

But what we found, this is like what's surprising going back to like why I'm so excited about the global network, is that we shipped Telemetry on the client side and we found that the cache was actually kind of like not helping us that much, because of a lot of our users were either on like discs or local devices that were busy. So perhaps IndexDB was taking a long time, we found the lots of IndexDB reads concurrently. We're kind of getting throttled. Where like some people had just really slow disks. So then we're like, "Oh, maybe we have do a data race between the disk," and then we realized, "Wait, we're using data hooks to get fresh data," and the network is actually really fast and becoming faster with HTTP 3, with databases that are global, with databases that can shard customers by country. We realized just go to the networks as fast as possible. The network is a computer, like Sun used to say.

Funny enough, we undid all those efforts in doing like smart like local data caching and we moved to just hitting the network really quickly. The kind of sort of story here is it's amazing what clients had apps that download and load really fast from this edge network can do, and I see the future as being getting data and committing data to the edge network incredibly fast and then we'll truly build applications that are still very easy to write and deliver but that just render and load data instantly. I think React is so well- positioned to sort of capture that feature.

[01:01:38] JM: You've spent some time talking to the React team about some of this concurrent stuff, I believe. Partly I think that's because you used to work with them, right? Because you were in MuTools.

[01:01:50] GR: Yeah.

[01:01:51] JM: How come you never joined Facebook?

[01:01:54] GR: The story is really funny, because when I was working on MuTools, I believe I started when I was 16 and I was in Argentina. I've always made it my mission to not share my age publicly because I've always truly believed in results over anything else, and I think I always saw my age as being like sometimes overblowing results out of proportion, like the stories of like, "Oh, wow! He's doing that and he's only N-years-old." But also sometimes it was kind of like the opposite, like it would drag me down, "Oh, he cannot be hired. He is 15." It was like both things in one.

But what happened that was really funny is when the MuTools team sort of started moving into Facebook, I remember getting pinged for jobs at Facebook many times, and one serious offer involved like a recruiter being involved, like probably in fact one of the early Facebook recruiters. Then they kind of found out, "Oh, not only is he in Argentina and he would have to move to San Francisco," which was like a core tenant at the time. I think obviously there's one office. But also I believe that when I did get that offer, I was probably 16 or 17 still. It's basically impossible and it was really funny.

I mean, it was really – Over the years it's been really flattering and makes me really proud that I've been close to those stories and I obviously could have been a bigger part. But now being and creating an independent project and an independent company, I also see the value of growing the entire React ecosystem. I see the value on building a more opinionated takes on software development. I see the value on closing the loop.

I think one of the most interesting loops to close is that, "Okay, you get excited about this open source library. How do I actually publish it to the world?" This is why I look always to Apple for inspiration, because they built a platform and we all agree, "Okay, super close, and it's not great," but they're giving you the ability to reach the world in a way that they've basically removed almost every question from your head, right?

Apple actually downloads all the app blogs from a CDN. Sure, they're not JS yet, but they download these native apps from a CDN. Developers don't know that, that they've worked really hard in creating this edge network for application delivery. They're given the tools that they recommend and they're giving a fair amount of flexibility. They've giving ways to build and publish them. They've given feedback, right? If you publish a bad app, this is what happened to Mark Zuckerberg. He published a bad app and he got one-star reviews.

I look to dad for inspiration. I think by having a more opinionated take on the software development lifecycle, but still being very close to the open source community, we can push the web forward to get to that level of end-to-end amazing experience.

Nowadays, when I think about what I would love to improve in the React ecosystem, I look to native for his inspiration. I look to animations. I look to compelling experiences. I look to fast data fetching. I look to simple things, like when you go back and forth in an app, you don't re-fetch data, and that's one of things that you can do with hooks today.

Alex Russell had this funny bio where he said add Google and Chrome – Sorry. In Web Standards actually, and he had this funny phrase for many years. He's like, "Every day working hard to push the web into the 90s," or something like that, because his thing was what you could do with C++ or in GTK or QT or whatever. In the 90s, the web always would take longer to reach that level of performance or API access or whatever it is.

The web historically has been lagging behind a little bit on like the APIs that you get, for example. But what the web has that is truly amazing to me is just how broad the ecosystem is. How much of the building blocks are open and accessible by everybody? But most importantly, the fact that you can reach so much further than what Apple can do with iOS. These URLs that you publish, these deploys that you make can truly reach the entire world. Then they download a code, tiny bits of code lazily, and that code can be updated much faster than native app could.

I think what I kind of see happening this decade is we're going to continue to make the software development loop really, really amazing. I believe strongly that React will play a major, major role in making that happen. Then we'll get to this sort of point where we're building the highest quality experiences when you compare JS and React to any other benchmark that you could imagine. Today, I still we're lagging behind and we're getting there.

[01:06:46] JM: Do you think it eventually goes as far as React Native and totally pervades the mobile device?

[01:06:51] GR: We're already seeing it happening. There is plenty of very popular apps that are shipping React Native with a combination of even exact same React codebase wrapped into like a React Native WebView to a hybrid mechanism of you boot into React Native and some things are web and some things are calling into the native component APIs to fully React Native where like every single component is using the operating system component, like the native UI component that Apple or Android have already built.

In my opinion, this is only going to accelerate because of the underlying economic advantage that you get – And just like product development and product design and engineering advantage that you get from one unified codebase and one unified ecosystem. Like I said, it took a long time for Electron to be the way that a lot of companies like Slack have chosen to ride their clients. Nowadays, it would be unthinkable that they would ride to other apps, one in like the Windows UI APIs and one in Mac OS ones and so on, or Linux for that matter.

I think the same is going to happen to mobile. I think, little by little, like mobile websites are getting better. PWAs are getting better. Google has great data that PWAs can be even more engaging and compelling than native counterparts. What we need to work hard as a community is on being very cognizant of our benchmarks of performance, right? What is the experience on the first page load like? What is the experience on how long it takes to render? Is the experience as compelling from a user interaction perspective? I think once we start working on those things more consciously, we're definitely going to get there.

[01:08:39] JM: How will the Jamstack look in five years?

[01:08:41] GR: I think Jamstack is going to be the enabling technology, because when I tell someone like, "Hey, hop on React," and they've never used React or maybe they're still running on a JVM monolith that does server rendering, and like React is too specific of a prescription, but it's also not complete enough of a solution. It's not a stack. You're being told, "Use a library," right? That's not how you build software, is if I tell you, "Okay, you're going to build the biggest next big social network and you're like, "Okay, I'm going to build an iOS app." You kind of get this idea of there's an entire sort of prescription for it.

With React it's more of a – This is kind of the thing that we're solving with NextJS. We're saying NextJS is a framework for Jamstack really, because we wanted to take and adapt the best way to build software. By saying Jamstack, we say implicitly CDN network. We say implicitly no overhead on operating and running your code. We say very rich experiences, because the presence of JavaScript includes, "Oh, you're going to do a lot of interesting stuff in the client-side." We're saying optimize your markup. We're saying take advantage of the APIs that you've already built.

Imagine that I just tell you use React. You're like, "Okay. Do I write my APIs in React?" It's not enough of that solution. What I like about Jamstack is saying like, "This is a very concrete answer to what the future looks like to how I build my next app. Even if frameworks change over time, like I don't start with NextJS, or I use Vue, or I use Svelte or Sapper. What's more important is that you are still using that batter programming model and that as a result, your users will get a better experience.

[01:10:37] JM: Next often gets categorized side-by-side with Gatsby. How do you see Nnext and Gatsby diverging or converging?

[01:10:49] GR: I think NextJS is less opinionated and more flexible. Gatsby has always prescribed the usage of GraphQL, for example. Gatsby has a constant of themes. NextJS has always been more deliberate and careful on staying closer to the bare metal of React.

For example, we thought about creating a theme abstraction, but then we realized, "Well, components provide a solution for theming. React context provides a solution for theming. Even CSS provides a solution for theming." We want to stay closer to the native web as well in that regard, and the abstractions that we present are as a result lower level. We think this is a better bet longer-term with regards to how industries evolve, with regards to how React itself evolves. For example, that's why I was saying like, for most – Actually I would say for all our client-side data fetching, we use hooks. We don't like some new way of doing data fetching that is unique to NextJS.

When we use static-side generation or we need to fetch data at the top of the page, we do have a NextJS specific cook, but that data fetching hook that NextJs provides, the only contract is that it's a promise gets resolved. You get the data however you want. I think that's a very important pillar because we see NextJS as a platform for the future of React development, and platforms needs to stay very, very flexible and the opinions that they provide have to be battle tested.

For example, when we started NextJS, Create React App didn't exist, but we were dog-fooding NextJS internally a lot. I remember we're starting to strive to do a lot everything with the React. So funny enough, the first page that I built for my company was the logo, "Welcome. Leave your email." Then I remember later on I was like, "Okay, if people are going to put in personal information, they have to at least have some terms of service and some privacy information and so on."

I thought, “Okay, what is the minimum viable company website?” and it had like those three or four pages. When I was building with React I was like, “Whoa! I am shipping the content of the terms of service in the JS bundle together with the first page and the input for that page.” I was like, “There has to be a better way.” I went back to the principles of the web of like the web is just a bunch of URLs that download HTML. When you go to cnn.com, you’re not downloading the terms of service.

One of the invariants that we found was, “Well, there will always exist multiple pages, multiple URLs and multiple entry points into your application.” Then we started evolving in that direction. I remember when Create React App came out, Create React App was even lower level than Next, which is a great thing, and it’s a great on-ramp into React. However, I decided it was still open source NextJS because it is like, “You know what? I know that people will figure out how to solve this, but I know that every single developer will want that as specific feature.” The same happened with a few other things like importing CSS, the router, the router API. All the APIs that we’ve built into Next have been carefully considered in terms of those invariants, things that are not going to change.

I think, broadly speaking, that's what’s going to determine the evolution of Next and the evolution of Gatsby where NextJS is going to become more of a platform. Whereas Gatsby is going to become more opinionated around a use case, like I need to render a specific blog post, and I know that I want to use GraphQl. But I think as companies evolve, they become a lot more dynamic and versatile, and I think Gatsby is kind of optimizing more for the less dynamic and versatile use case, which is also fine, and I think there will have a lot of usage for those things. But it’s just strictly less flexible.

[01:14:50] JM: The deployment medium of a container that may be executing serverless functions or perhaps a web assembly module, these are different architectures. They’re exemplifi ed by AWS Lambda and Cloudfl are workers respectively. Tell me your perspective for how the deployment medium for serverless applications is going to look in the limit as far as we know.

[01:15:19] GR: Yup. What I mentioned about Jamstack that I like is that – And if you've heard previous sort of summaries of our platform, we’ve always said we’re serverless. Serverless as a term is super loaded, right? What is it? No servers, or servers, but they're abstracted and hidden, or like what is it?

What I like about Jamstack, it's a more opinionated take on how you are serverless. Jamstack is obviously serverless, but what I really, really like is this idea that you don't deploy to a specific location. You deploy to an edge network. As a result – I'm addressing the first side of the comparison, like how is it different for a container. It's cost prohibitive and I would say nearly impossible to say, "I'm going to deploy my server to 20 locations immediately in a way that's fast, cheap, etc."

Even then, when the request comes in, do you always wanting make it go through a server? Especially since we just discussed there are so many types of pages that can be basically narrowed down to a bunch of HTML that can be returned immediately. So you probably don't want to use a server or a container even if the container is serving static files. This is a funny thing to carefully consider. If you have a server that is serving static files, and even if you throw a CDN on top, you're just adding more steps in more places where things can go wrong and most likely you're not actually optimizing correctly for the fact that you're dealing with static files. I love this idea of push to the edge, especially for all the parts that can be decidedly static where we already know ahead of time that they're static. That's just like the beauty of that model.

Further, going to like the worker type abstraction, I think that you should avoid that. I think that if you're receiving a request from a visitor, why would you want to run code at the edge if you can avoid it? I mentioned some strictly tactical and operational reasons for doing so, like, "Oh! Now you have to worry about code that executes in a server context and code that executes in a client context, because a code will run in both, because the app needs to become alive later.

I'm running exception handling with Sentry, for example. I am putting it in my worker, but then I'm putting it in my page. I am doing telemetry. I'm doing goal analytics in both sides. I'm increasing the surface for security problems, like a worker, and this why I actually don't love the name of it, is sharing traffic and putting it all in one worker, right? It's bringing back this process abstraction. I strictly dislike that from a security standpoint. I don't like the shared memory architecture. Basically, the idea of a V8 isolate is let's share a memory space with everybody else, which can basically create lots of bleeding of sensitive data.

I just love my edge to give me the thing really fast without executing any circuitry. I look at it as strictly a cost-saving mechanism for transferring the job that is inevitably going to happen on the device to only the device. It's going to save me money. It's going to make me sleep better at night. Td this is why I love the idea of NextJS becoming the Jamstack framework, because it's hard to capture all this subtle nuance and all these technical alignment of the pieces. But if you just use this framework, you're going to get all this cost-saving, simpler engineering, simpler deployment all in one package.

Going back to why I really want to invest in making NextJS the platform, I still think anybody in the world should be able to take NextJS and run it as a container if they want. Is that the way they think the future is headed? I really don't think so. Even the present is not there, but you should be able to do so. We need to retain this ability to take your NextJS project, run it locally, run it remotely and what I like to say is not run it. I like the idea of not running code at the edge and running lots of tiny little pieces of code in the client so that your page loads fast and starts doing things really fast.

[01:19:37] JM: Guillermo, thanks for coming on the show. It's been great talking.

[01:19:38] GR: Thank you so much.

[END OF INTERVIEW]

[01:19:48] JM: Apache Cassandra is an open source distributed database that was first created to meet the scalability and availability needs of Facebook, Amazon and Google. In previous episodes of Software Engineering Daily we have covered Cassandra's architecture and its benefits, and we're happy to have Datastax, the largest contributed to the Cassandra project since day one as a sponsor of Software Engineering Daily.

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

More than 400 companies including Cisco, Capital One, and eBay run Datastax to modernize their database infrastructure, improve scalability and security, and deliver on projects such as customer analytics, IoT and e-commerce. To learn more about Apache Cassandra and Datastax's enterprise, go to datastax.com/sedaily. That's Datastax with an X, D-A-T-A-S-T-A-X, @datastax.com/sedaily.

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

[END]