EPISODE 486

[INTRODUCTION]

[0:00:00.3] JM: Modern frontend development is about components. Whether we are building an application in React, View or Angular, components are the abstractions that we build our user interfaces out of.

Today this seems obvious, but if you think back five years ago frontend development was much more chaotic. This was partly, because we had not settled around this terminology of the component. React has become the most popular frontend framework. Part of its growth is due to the ease and reusability of components across the community.

It's easy to find building blocks that you can use to piece together your frontend application. You need a video player component, you need a newsfeed component, you need a profile component. All of these things are easy to find.

As you build a React application, you take open source components off the shelf and you build some other components yourself. To keep things looking nice and consistent, you need to style your components. If you're not careful with how you manage your style sheets, you can end up with inconsistent stylings and name space conflicts.

Max Stoiber is the creater of styled dash components; a project to help enforce best practices around styled components. He is also a founder of Spectrum, a system that allows people to build online communities. Spectrum has similar design and engineering challenges to Slack or Facebook. It made for a great discussion of modern software architecture.

In today's episode, Max and I had a wide ranging conversation about frontend frameworks, components and the process of building a product. Max also describes the advantages of using GraphQL and the Apollo toolchain.

[SPONSOR MESSAGE]

[0:01:53.3] JM: G2i is a talent platform built for engineers, by engineers. React Native, React and mobile, the developers on G2i have expertise in the best tools to build your applications. When I need engineers to help me out with my apps, G2i is the first place I go, especially when I'm building with React or React Native.

Contract a G2i developer to help you on a short-term basis, or hire a G2i developer fulltime. If you're looking to build cross-platform applications in React Native, definitely check out G2i. The G2i platform is a community of React Native, React and mobile developers. These engineers can become part of your team.

If you're looking for developers to build your product, check out g2i.co. That's the letter G, the number 2, i.co. You can also send me an e-mail and I'll be happy to tell you more about my experience with G2i. Find your React Native, React and mobile talent by going to g2i.co.

Thanks to G2i for helping me ship my products and thanks for becoming a sponsor of Software Engineering Daily.

[INTERVIEW]

[0:03:15.9] JM: Max Stoiber is a ReactJS developer, an open source maintainer and the founder of spectrum. Max, welcome to Software Engineering Daily.

[0:03:23.7] MS: Hey, Jeff. Thanks for having me.

[0:03:25.5] JM: I want to start by talking about a project that you're well-known for, which is called styled components. In order to talk through this, I want to first start with the idea of the component. We've gotten to this place in frontend development where the abstraction that we build our applications out of is the component. How do you define a component?

[0:03:48.5] MS: A component for me is really any capsulated piece of user interface, I think. In the sense that similar to how you would use Lego blocks to build huge buildings. It all starts with that one tiny block. The component is the most minimal version of a certain piece of user interface. Then we take those components and build them up to complex systems.

What's nice about components is that it becomes easier to understand the system as a whole, because since we have all those small, we captured the pieces that are relatively easy to understand, by understanding all the small pieces we can more easily understand the whole system. Where if you on the other hand, if you were to write spaghetti code and like one huge bowl of mud, right? It becomes really hard to understand, because you'd have to understand the whole system before you could even begin to understand the small pieces. Yeah, I would say that's what I would call a component. I mean, capture the piece of user interface.

[0:04:52.0] JM: Right. How did we get here? You deal with components today and it feels so intuitive, it's hard to imagine how the world used to be, but it did tend toward spaghetti code in many frontend projects. What were the abstraction that we're using before components?

[0:05:11.0] MS: I think the web's always moved towards components. They just weren't always called components. Even mean something like PHP, which has been around for ages. You'd have these includes, which are like encapsulated pieces of user interface. You just include a button of PHP, for example.

I think what's changed is the focus on building a foundational set of components, right? Previously, you just ad hoc create components wherever and have hundreds of thousands. But then, if you have that many components, it becomes understandable again in a huge bowl of mud.

What shifted I think is the mentality from these – we did have encapsulate pieces of user interface, but they have a very big, and it's I think the big mindset shift has been to narrow them down in scope and start with really tiny blocks, rather than a medium or a large box.

Yeah, I think even something like Dojo had – I can't remember what it was called, but there was that modules or components, which something to that effect and knockout and backbone all had a similar concept to components. It's just that in my mind, they were a lot bigger. It wasn't that you created a component for a bot, like that was not very often seen. It was too easy to just render a button, rather than create a button component that you reuse throughout the app.

[0:06:32.0] JM: Okay. You mentioned a couple best practices there basically, starting with very small components. What are some other best practices around building applications around components today?

[0:06:46.2] MS: I think the most important really is keeping a consistent set of them, because still today it's very easy to just create a huge – what in React would be called a component, but would really be a view out of – with 1,500 HTML tax. But ideally, each one of those HTML tax would be its own tiny understandable component, but then you also end up with 1,500 [inaudible 0:07:09.2].

You have to define a set – a design system of components that you reuse throughout your app. By having that very narrow, small set of base components, that a very extensible and very flexible, you don't end up creating a ton of complex logic and a ton of ad hoc HTML. I think that's really the most really important thing in what then this trace move towards.

It's just that sense of a component library, or of a design system, or pattern library, whatever you want to call it. Just some base set of components that lives separately from the app. Then style components sort of – the reason we made the library the way it is is because for us, we use components for everything.

Glen and I, we had components, for example a grid. Rather than rendering a div class = grid, like it would do with bootstrap, we would have a grid component, which would maybe do the same thing. Maybe that grid component just renders div class grid under the hood. But really, I don't want to have to care about that. I just want to have that set of components and one of those components just happens to be a grid that I can use to layout my app.

What exactly that does is while building the app itself, while layouting sort of consistently isn't really what I care about. I only care about making everything consistent. Then if I want to change the whole layout of the whole app, I just change my grid component. Start components came from that philosophy of using components for everything and having components be responsible for styling. I think that works out really well, especially in React.

[0:08:46.4] JM: There's a related term called a container, when it comes to frontend development. What is a container in the context of frontend development?

[0:08:55.6] MS: the differentiation between a container and the components came from an article that Dan Abramov wrote – must be three, four years ago now, I think. Which is basically just a pattern of having two different types of components, where one component is responsible for the styling and the rendering of text and the rendering of HTML notes. The other component is mostly responsible for data fetching.

That model where – the one that that data fetch a container, right? It's something that fetches some data from an API and then renders smaller set of components that just care about rendering something.

The nice thing about that is that you end up with these small components that just care about rendering, which are easy to test and easy to verify they work as they should. All of your logic lives in these containers, which themselves only care about fetching data and manipulating data and don't really care about the stylistic aspect of your app, which makes everything much more understandable, because if you have a bug, you can very easily figure out where the bug is.

If you have a bug where an item that was loaded from the API isn't displayed, then you know the bug must be somewhere in the styling. Or if the API pretends to write data, but then the container changes it wrongly, or handles it wrongly, then you also know where that bug is. It got taken a bit – I personally think that article gets taken a bit too much as gospel, and as like a rule, rather than just a nice pattern to use when you have to fetch lots of data from an API.

Where that is taken to the extreme recently was GraphQL, where most GraphQL libraries they use with React have their fetching be based on components. That works out really well, because GraphQL then basically abstracts your entire data fetching logic, right? You don't do any of that. Your component just says, "I want to have the current user, their username, their age and their profile picture."

It doesn't really care where those things come from. It just tells GraphQL to get them and GraphQL gets them. Which works has been nicely for building the frontend, because it makes

everything very streamlined. You don't even have to call them API yourself. You just say, "I want to have this data," and then that data is there.

[0:11:06.2] JM: What else is going on in the – like in the movement towards GraphQL on the frontend, how does somebody refactor their frontend application to use GraphQL?

[0:11:17.9] MS: That's a good question. I think the first step is that you need to have a GraphQL API, which I think is actually the harder part, in the sense that a lot of – or that's not entirely true, but I think frontend developers are more open to GraphQL, because it really makes their lives easier. As a frontend developer, using GraphQL is a joy. I don't want an app where I have to manually orchestrate fastest and fetches and then end up over-fetching data, or under-fetching data.

With GraphQL, you just don't never have to worry about that. It does increase, or it can increase if mismanaged to complexity of backend quite a bit though. The first step will be to convince your backend developers to write a GraphQL API. Once you have a GraphQL API, refactoring to these GraphQL is – it depends on how well your app was architected from the beginning.

If you have those set of base components, that if you have that container components already related strict about it then not much is going to change, because your stylistic components are still stylistic and they still get the same data. Then rather than having containers, you just have GraphQL cords. That works out quite nicely, I think. I've never done it though. I've never refactored that from the rest of API with GraphQL.

[0:12:24.8] JM: We'll talk a little bit more about GraphQL later on. Let's talk about styling and emphasize the component side of things, rather than the data fetching side of things. We've had CSS for a long time, and we've had components for a long time as you alluded to. Whether or not we've been calling them components is up for debate. The best practices around styling our components have certainly changed over time. It's 2017, what are some best practices around styling components today?

[0:13:00.8] MS: The big differentiator to 2016 is that many people now use this concept of styling components, not necessarily with styled components and library, but just styling

components where you have a grid component, or layout components, that just renders some stylistic thing, whatever that might be.

The second big shift in sort of, at least in React ecosystem has been to write your CSS in JavaScript, rather than in separate CSS files, with styled components also those, or lets you do, I guess.

[0:13:32.0] JM: Right. Before you built this project styled components, what were the shortcomings that you saw in how people were managing their styles?

[0:13:43.5] MS: I'm going to answer this question in a very roundabout way. People when they see something like CSS and JS library, sometimes go, "Why would I ever do this?" See, that works perfectly fine for me. I don't really need to use anything else. Why would I use this? Then I go, "If what you're using right now works for you, then that's perfect. Don't use anything else, because why change a running system?"

At the same time, the reason styled components and other libraries like styled components exist is because CSS isn't – wasn't invented for components. It was really invented for documents, for laying long pieces of text. Where today with something like React, you probably aren't building a long document, you probably aren't rendering – you might be rendering a long piece of text, but many people use it for more dynamic things, for dynamic applications rather than just rendering a document.

CSS has this gap between it can work in component-based systems but it requires a lot of work to get right, because it has things like it will name space, every class name has to be globally unique, which is hard when you have these encapsulated pieces of user interface. But then you have to sign them globally unique variables, or names basically, but just doesn't make any sense.

The react community went into this direction of why don't' we just not care about class names anymore? They mentioned with CSS modules, class names were essentially just taken out of your hand entirely, right? You would have a build process like webpack take your CSS and uniquefy every single class name, so that you would never have a class name patch. Which is

super nice, because it's just avoids a whole bunch of bugs that you never want to have to deal with.

I really have more important things to do than to figure out whether the class name I'm trying to use is globally unique or not. Why do I have – as a human have to take care of that? I'm not very good at it. I have to search through an entire code base, I'd have to make sure that people don't use variables and stuff.

Then it's just tedious. Whereas a compute is really good at it, because they already have – they can have – they can know which class names there are. They can have a global list of all the class names there are, and then they can make sure that they're unique. Why don't we let computers do all the stupid work and we can focus on the more important parts of our app?

That thing taken to an extreme with styled components where you don't even ever assign a class name anymore to anything. With CSS modules you would still import class names from your CSS files. Without components, you never ever see a class name, or at least you shouldn't depending on third party. When you write out components, you never handle class names directly. You only write components, which have a certain style fragment, a certain piece of styling associated with it.

Which is super nice, because now I can focus on actually styling my app, rather than having to focus on, "Well, do I make my styles dynamic? How do I make a button primary? What's the overriding mechanism? How do I yada, yada, yada?" You just render a button, or a button primary, and that just looks either normal all the time.

[SPONSOR MESSAGE]

[0:16:55.4] JM: For more than 30 years, DNS has been one of the fundamental protocols of the internet. Yet, despite its accepted importance it has never quite gotten the due that it deserves. Today's dynamic applications, hybrid clouds and volatile internet demand that you rethink the strategic value and importance of your DNS choices.

[INTERVIEW CONTINUED]

[0:18:53.4] JM: To underscore this, explain why namespace clashes often occur in CSS and how you are avoiding that with styled components?

[0:19:03.5] MS: Every person that's worked on any big frontend, or the client side – well, any big frontend really has probably ran into a bug where I think yourself, or some other developer, or a third party library uses a global class name that you've already used.

Then you have to figure out, "How do I override this? How do I make it different?" You have to either use important, or you have to choose a different class name. But then when you have to choose a different class name, what do you choose? Because it's probably already a good name for what you're trying to do.

Then if you have to choose a different name, it might not be the optimal name anymore. It's just super annoying. It's just a problem that you don't want to have to deal with where your head are overwrite a button styling, because it has a head or button declaration.

That gets even worse when you have domestic class names, when you have high specificity, because then overriding might not even be possible. If somebody uses inland styles, you suddenly can't override it anymore. All these annoying problems with CSS you just don't have them anymore. Just gets rid of that entire class of problems, but just not having you worry about class names at all ever. I don't know if this makes any sense without having actually seen what it looks like, but it's very nice to work with.

[0:20:11.8] JM: Right. Let's describe that in more details. How do styled components work under the hood?

[0:20:14.8] MS: What styled components as under the hood is you tell it which HTML type you wanted to tell it which styling you should have. For example you could say, you want to have a button HTML type that is a color of red. Under the hood, styled components, that component that has a certain automatically generated class name, and then it injects your color red declaration, or look at styles associated with that HTML tag into a style tag with that unique class name.

All you ever see as a user of styled components is components, which is why it's called styled components. You just see a button component and that button component renders a button HTML tag with a certain piece of styling. Then under the hood, styled components would take that and inject it into the head and generates a unique class name that can't be duplicated, so you won't ever run into annoying bugs, where some class names overwrite each other or anything like that.

[0:21:10.0] JM: Maybe you want to describe some of the things that people are often confused about when you discuss styled components with them.

[0:21:17.3] MS: The biggest source of confusion is probably that people think it's inline styles. A first wave of building more encapsulate CSS, or a first wave of it mentions around building encapsulate CSS were focused on inline styles. They would assign a style attribute onto text and then pass the styles as a string.

The string with inline styles of course is that they're in performing when you render a large application with inline styles. It takes the browser a while to get through all of those inline styles. Then on top of that you can't have media inquiries, you can't have pseudo selectors, so you can't hover styles or focus styles or anything like that.

It's just overall not perfect. When people first see styled components and see, "Hey, this is encapsulated styling." They think, "Oh, this is another one of those inline styles libraries that generates inline styles. But I don't like inline styles, so I'm not going to use this." When really it isn't.

We inject CSS into a style tag. Meaning, it's just CSS. There's nothing fancy about it. It's literally just CSS. You can use media inquiries, pseudo selectors, whatever you want, whatever you used to from CSS, you can just use in styled components, because it is just CSS. That's probably the number one confusion that people have.

I actually wrote a blog post about this a while back and people still ask me that question every other time and go, "Why would you use inline styles?" I'm like, "No, it's not inline styles." Yeah, if you were thinking I want to use inline styles, it's not inline styles.

[0:22:43.7] JM: Okay. Well, I put that confusion, disambiguation article in the show notes of this episode. I'm glad we got to talk a little bit about the styled components side of things and some of your beliefs around components and frontend development, because I want to use that as a jumping off point into Spectrum, which is a product that you have built which allows people to build online communities.

I'd like to take a top-down approach to this and discuss first the product and then how you built it and then how the architecture you built Spectrum out of reflects your beliefs around frontend development. Let's start with the product itself. Explain what Spectrum is.

[0:23:32.8] MS: With my open source projects like styled components, I always had this issue of I can't personally answer every single question of every single user that ever comes in. That's just too much, right? No maintain of any popular [inaudible 0:23:44.4] can do that. You want to

have a place where the users of your library can connect with each other, can answer each other's questions, so you have to do less work.

The issue is that all of the existing places are very badly optimized for long-form discussions. The most common tools that I think is like get to Slack, but they all use IOC style chat, where you just have one large chatroom.

[0:24:11.9] JM: What about discourse?

[0:24:13.2] MS: Discourse is a forum, so you also don't want to have a forum, because then it's static again. You want to feel like you're a part of something. Discourse is just how oftentimes do you visit forums realistically.
It's' just not the same feeling as a chatroom. I see chat is nice in some aspects. It's real-time. You feel like there is warm bodies. You feel like there is people around. Forums also are nice because they are great for long-form discussions, when you talk about something more in-depth that has a space, a forum is great for that.

Where I see chat is horrible for that, because people are there and there is five conversations opened at the same time and it just scrolls up and up and up. If you want to try and talk about something that people talked about yesterday, you basically can't. It's just not very well made for this kind of in-depth discussion purpose.

[0:25:01.2] JM: We have a Stack Overflow, or Quora.

[0:25:04.4] MS: In comparison to Stack Overflow or Quora, ,those are – Stack Overflow does just question on the sites. They don't want a more open-ended discussions. You don't feel like you're part of the React community just because you answer questions, or ask questions about React and cycle flow. You're just there to get help and then you leave again.

Spectrum is this - [inaudible 0:25:26.3] Brian, my two co-founders take on what we think online community should be, which is basically a mixture between chat in the forum, where you have policies like you have in the forums. Every conversation is threaded by default. Every

conversation that you have has its own place and you can refer back to old threads and you can talk about them. They have some topic. Then “comments underneath it,” are real-time chat.

It has this mixture of it’s good long-form discussion, because the discussions don’t get lost after that. They’re not gone. There’s still the link. You can just visit the conversation from two years ago. Well, not two years ago because we went there out, but from eight months ago and it will still be there. Then because it’s real-time chat, it still feels like people are in the room, still feels like there is warm bodies around. Yeah.

[0:26:20.5] JM: What’s the difference between – I’ve messed around with Spectrum, have used it a little bit. It’s like if you imagine like Slack, but instead of making channels, you just make new threads. Because the problem with Slack is that it’s built in a way where the channels that you create are supposed to persist over time.

Whereas, in Spectrum it feels like there is less pressure to think long-term about the channels that you’re starting, because you’re just starting threads, and so people can start a thread and then have this forum-style thread that is persistent, but it’s also lightweight because if people stopped talking about it, it’s going to fade away into the background. Whereas, in Slack you just get this channel bloat where even if nobody is using a channel, just sticks around.

[0:27:14.2] MS: Yeah. you just described everything much better than I did. I think another important property there is that the threads can still be found, because it’s a web app, because it’s our first goal was to build a web app. You can actually search for it. If somebody searches for it a styled components question, they might find the conversation that somebody had on Spectrum about their exact problem and they might be able to solve it.

Or in Slack, or in Gitter, or on Discord, that knowledge is lost forever. If at all, these tools have built-in search functions and those are probably not very great, because it’s still one long stream of text. But Spectrum, you can just Google it. You can Google any conversation that happen on Spectrum and you’ll find it on Google, or Bing, or whatever search engine you use. The knowledge doesn’t get lost forever.

[0:27:58.9] JM: If I'm a user and I go to Spectrum, am I typically going to Spectrum because I have intent? Am I going to Spectrum and saying, "I want to find information about styled components. I want to find the answer to a specific question around styled components." Or are people going there with just like the – I'm going to Hacker News, or I'm going to Facebook because I am bored, and I'm just looking for some jolt of adrenaline.

[0:28:27.9] MS: I think it's both. I think what makes it different from Slack over full is that it's both. You can go there and you can just ask your question, but then because it's real-time you suddenly feel like you're in a conversation and you feel like you want to stick around. Then our goal is that people want to come back, even if they don't necessarily have the question, they just want to come back and hang out and chat with other people who use styled components.

Even if it's not necessarily about styled components, they just want to talk about something, and make it feel like they are placed to be. As a user also it's nice, because you don't have 15 different logins. It's one login and it's all of your communities, which is pretty great.

[0:29:08.6] JM: Yeah. I guess, that's another issue with discourse. Well, some people might call it an issue as discourse has never really become like a network multi-forum forum. But I guess that's okay. It was built to be a narrow-specific forum. I don't mean to insult discourse at all. It's very useful. Plenty of people use it. It is fantastic for what it's meant to do. It's not a real-time chat platform, it's a forum.

Spectrum, it's this real-time chat, I guess if people – if you get a chance, or just check it out real quick so you can understand what we're talking about here, because I'd like to discuss the tech stack now. You've got basically a platform that is like a real-time chat forum, so it's going to have similar architectural issues as a forum, similar architectural issues as a Slack-type of application. Describe the tech stack for Spectrum.

[0:30:10.6] MS: It's not with a backend, because that's more of a question. We use [inaudible 0:30:13.7] as our main database, as our main persistence layer, because of its really nice real-time features, I guess. We use GraphQL for API and we use Node.js in the backend, just because we're three jobs with developers, so that was an obvious choice. Then we use Redis for some job stuff.

Then on the frontend, we use React, we use obviously GraphQL again, because you have to use a GraphQL client if you want to create GraphQL. We use styled components obviously, we use Redux; that's a course overview, I guess.

[0:30:46.6] JM: Now RethinkDB is gives you push, basically like – well, explain why you use RethinkDB, because that's a differentiated aspect of your architecture.

[0:30:58.6] MS: Yeah. RethinkDB is built for real-time. They have this concept of change feeds, where you can take basically any query you want and in real-time listen to changes to it. Which is perfect for a chat app, because we can just list to changes on anything and everything, right? We have change feeds for new messages and threads, we have change feeds for new threads in communities, we have change feeds for basically everything; modifications, messages, online statuses, all sorts of stuff.

We don't have to have a separate layer that we use just for real-time. For example, a common architecture to architect something in real-time would be to have pulse credited persistence layer, and then use Redis for real-time modifications. But RethinkDB just does both in one.

It also has very nice jobs and support. It works super well with Node.js. It works very well with sharding and replication, so it just overall is a nice database. We decided to use it and it's been fairly great so far.

[0:31:57.1] JM: The change feed is pushing changes to – is there the GraphQL server, in the GraphQL server –

[0:32:05.4] MS: Yes, exactly. In the GraphQL server we just have change feeds on for example, a new message. Then whenever a new message comes in, we use subscribe via GraphQL's subscriptions, then the subscription just sends down the new message.

[0:32:18.8] JM: Is there any concern around using RethinkDB, because the company that was built around RethinkDb doesn't really exist? I'm sure you've gotten this question before. I've just got to ask it anyway.

[0:32:29.9] MS: Of course. Of course. There are a lot of people who might not know. This is a very common question I get when I say I use RethinkDB. People go, "Wait. Didn't they shut down?" But thankfully, the RethinkDB, the IP was acquired by the Linux Foundation and it's very still very, very actively developed. The open source committee just took it and ran with it. It's still –

[0:32:50.6] JM: That's fantastic.

[0:32:51.9] MS: - under very heavy development. Since the last motion there have been like, I think a thousand commits now. It's under very, very active development. The only thing that doesn't exist anymore is the enterprise support for it, which to be honest I don't really care about anyway.

Am I sad that the company shut down? Yes, because it's a great database. I would've loved to see then by doing a great database make an awesome company out of it. Unfortunately, they didn't quite succeed. The great database still exist.

[0:33:17.0] JM: I believe Slava and most of the team, or some of the team went to Stripe. They're going to get rich anyway.

[0:33:25.2] MS: Yes.

[0:33:28.2] JM: Do you know if that team – are they still involved with RethinkDB? Is the original core team still hacking on it?

[0:33:36.8] MS: As far as I know they're still around, but I don't think they work in it. Obviously, I don't think they work in it as often as they used to before. I think it's mainly just a community-driven project.

[0:33:48.0] JM: Speaking of companies that didn't really succeed in their original open source mission, but have seen some success anyway. The Meteor company, a company built around

the Meteor framework didn't exactly pan out in terms of Meteor, although I got to say I used Meteor and it was awesome. It just was –

Meteor is such an interesting story, because I guess it came up around the same time that React was becoming popular. There was just so much churn in the frontend world, and they just circumstantially – things didn't work out for Meteor, and then the Meteor company ended up pivoting to focusing on GraphQL.

You used the Apollo toolchain, which comes out of the company that was Meteor and is now I think called Apollo. Actually, I was planning to ask this, but why don't you give your perspective? What happened with Meteor? When you think about it from a historical context of how frontend development evolved, what went on with Meteor? Why didn't it work out?

[0:34:57.9] MS: That's a good question. I don't know any inside stories there. I think it was just monolithic in an era where people tried to get more modular. When you chose Meteor, you chose a whole stack, you chose everything and you couldn't really get out of it, You couldn't just replace a part of it. You used Meteor where you didn't.

[0:35:17.5] JM: People didn't want a new rails.

[0:35:18.9] MS: Yeah, exactly. I guess it was like rails, except for JavaScript. I guess, it's actually not too bad if it parallel to drop. Except in JavaScript, everybody nowadays has everything in modular pieces. I think that's a big reason by React exceeded is because it doesn't do much. Then the community stepped up and innovated around React. Where something like styled components would've never happened in a monolithic system like Meteor, because I wouldn't just never had the idea to make it and it would just never happen.

I think that's one of the big reasons that Meteor, even though it's a great system I think, I think the whole stack is pretty great. I'm sure people that use it love it. It's just the wrong thing at the wrong – the right thing at the wrong time. If they did that five years earlier, I think it would've been mind-boggling to us.

[SPONSOR MESSAGE]

[0:36:13.5] JM: Do you have a product that is sold to software engineers? Are you looking to hire software engineers? Become a sponsor of Software Engineering Daily and support the show while getting your company into the ears of 24,000 developers around the world.

Developers listen to Software Engineering Daily to find out about the latest strategies and tools for building software. Send me an e-mail to find out more, jeff@softwareengineeringdaily.com. The sponsors of Software Engineering Daily make this show possible, and I have enjoyed advertising for some of the brands that I personally love using in my software projects.

If you're curious about becoming a sponsor, send me an e-mail, or e-mail your marketing director and tell them that they should send me an e-mail, jeff@softwareengineeringdaily.com.

Thanks as always for listening and supporting the show. Let's get on with the show.

[INTERVIEW CONTINUED]

[0:37:15.6] JM: What is the Apollo toolchain? This is what the company that was formerly called Meteor – I believe it's now called Apollo. It's the thing that they're focused on. Apollo is this toolchain around GraphQL. What does the Apollo toolchain give you?

[0:37:31.1] MS: We use the entire Apollo toolchain. The nice thing about it is that Facebook as they did with React, they released GraphQL as a standalone thing. They released GraphQL with the traffic library with reference implementation. Then they've released data load, which is just – and you use GraphQL.

Most of the rest of the things were left out in the open, and then there was relay, which is a client sideline that you used to query GraphQL. They left many things out in the open, like how do you do caching? How do you do performance analysis? They weren't just a bunch of things around Graph QL. It weren't built out.

Apollo just went, pivoted from Meteor to GraphQL; at least that's as far as I know. Just built all of the tooling around GraphQL. Now they have a server implementation, which is called GraphQL

Express, where you can just add GraphQL to a express server in two lines of code, and then they have this tooling called GraphQL tools, which is a collection of tools as the name suggest around GraphQL, so that you can write the missing [inaudible 0:38:29.4] language and you can have yourself as separately. You don't have to use the standard, sort of bit annoying object notation.

Then they have a pool of client, which is a client side replacement for relay that allows people to relike, because it just was influenced by relay, but needed – I don't want to say beta, but they had some different opinions that were doubt very well. Now with Apollo client 2.0, everything is again more modular, and you can plug to gather your own plan, and you can have better cache implementation and all sorts of stuff.

Overall, it's just a toolchain, an open source toolchain to own that – covers that whole stack if you want to use GraphQL. I think for the company that their goal, or their idea is to own the entire stack of building something with GraphQL, and then building commercial tooling around that and giving them supply support and giving you this tool called engine, which does caching and performance analysis and giving you air tracking and giving you the more sass style enhancements that you would have on top of any production API. They just give you that and you pay for it.

[0:39:37.1] JM: I want to switch back to talking about Spectrum more broadly. I look at Spectrum and I see a big complicated web app, like Facebook, or Slack, or a Quora type of thing. Is it hard to find memory leaks in a complicated frontend app?

[0:39:56.3] MS: I hate that you ask that, because we actually haven't really. I've been trying to find it for the past two days. Yes, the answer is – The problem is that with Spectrum –

[0:40:10.3] JM: By the way, I asked because I get memory leaks in even small apps and complicated apps.

[0:40:16.9] MS: Yeah. Spectrum has gotten over the past eight months very – a very complicated frontend application that does a lot of things. It's just organically grown into somewhat of a ball of mud. As much as we try to structure everything properly and architect it

very well, we don't have a set plan for what we want Spectrum to look like in a year, or even a month. We're constantly experimenting on.

When you're experimenting and moving as fast as we do, it's very hard to stop and reconsider how you structure something, because you just want to ship it, because you have no idea if users are going to like it. Like we've shipped several things that we've had to scrap again, because we realize no direction, not what we want.

[0:41:02.7] JM: How important is testing in that process?

[0:41:05.8] MS: We've started to test – I'm going to say we started to testing and we used started typing useful type for our – both for the frontend and for the backend. More so for the backend, because it really helps there. It really helps us make sure we don't have any bugs and it's multiple times already called bugs that would've always been introduced.

I mean, we also test our backend API a bit. We've started doing end-to-end testing for the frontend. But again, because we constantly change what things look like, it's sort of very hard to – Because you don't want to test implementation details. It's like, I don't want to test that specific component, because I know that in the next month that component might look just totally different.

For us, it doesn't really make sense to test this specific component, but it also is tedious to test everything, because if you ran into and test, what exactly do you verify? Where do you even start? You do want to know some things very specifically, but you also want to know more broadly does the app work?

We use end-to-end. For most are going to be used just to verify that basic things are displayed, like that when you visit a community you see a list of threads and you see the member can and stuff like that. Just to make sure we don't break the broad usage. Unfortunately, it means that we have to find a few buggy parts in our end. That's just the way it is when you –

I feel like, so this whole design system, pattern library, ideology, or philosophy is super great when you have a specific plan for what you're trying to do. The caveat to it is that it was strict by

restricting to a common set of components, and sort of restricted in what you can do. Because all this you can adding more components, but if you keep adding more components you end up with a mess again.

So we have a set of common components, but then we also very often just have ad hoc components, because we're trying something new, or because it's just easy in that case. Overtime, this code base has gotten not as nice as I would like it to be, as it always is.

[0:43:03.3] JM: Yeah. Well, rules are made to be broken.

[0:43:05.5] MS: Yeah. I think it's a matter of practicality. We just want to figure out, more like we don't even know what we're building. We know what we're building, but we don't know specifically. We don't know what the ideal community platform looks like. We have some broad ideas and we're trying to experiment toward a good solution, but obviously we're very far from it. We have a lot of experimentation in front of us.

It doesn't make sense for us to just restrict ourselves to a single set of components and then never change anything, because it's just unrealistic. We're not going to hit the perfect community platform on this track. We're going to keep operating and keep changing things. That unfortunately had the side effect that our cup is a bit messy at the moment, which is why I've been trying to figure out why we have a memory leak. Because we use service at rendering, and so our server keeps crashing, because we have a memory leak in our frontend, which is annoying.

[0:43:56.2] JM: Yeah. Well, I'm sure you'll figure it out and –

[0:43:58.1] MS: Yeah, of course. Of course.

[0:44:00.6] JM: I mean, I respect your decision to move fast and develop memory leaks. I have the same philosophy. I want to talk about some more broad topics. It's 2017, how do React and Angular and View compare today? I realized you're mostly working in React, but you got your **[0:44:21.9]** around at the frontend community. What's your perspective on these other ecosystems?

[0:44:27.6] MS: Let's start with View. View is very interesting, because I think it takes the views that React had and makes them accomplishes a similar thing in a slightly different manner. It works I think very nicely. I think if you know how View works, it works pretty well. It's very similar to React. I would probably at the moment not choose it for building and production, because obviously I know React and certain apps. But also because it's just not as mature yet.

I think that due to their focus on a learning curve, they're going to keep growing. I think they will be a very dominant player in the coming years. React has I think become the default choice if you want to build any sort of dynamic web application nowadays at least. Maybe that's just my bubble speaking. Maybe in just inside of a huge filter bubble. To me, it seems like you likely choose React today if you were going to build anything complex, dynamic, web app, progressive web app thing.

Angular, I think is interesting because it's contrary to React and I also think View. Angular is very monolithic again, right? Angular does everything and it does everything out of the box. Where React really does nothing, or a very limited set of things and then you have to plug together your own stack based on which state management you want, how you want to fetch data and all sorts of stuff.

I still think Angular, especially with Angular 2 and now 4 and 5 went in a good direction with changing to more component-based model. It's very widely used still. I don't know – again, this might be my filter bubble speaking, but I don't know many people that would choose Angular over React today, except those that really know Angular. But again, maybe that's just my filter bubble, because I hang out around React developers all the time. That's totally possible.

[0:46:18.5] JM: We are using View for this new site that we're building for Software Engineering Daily. It's called softwaredaily.com. The big penalty that I have seen so far, I mean this is based in my first developed frontend app – help very significantly develop frontend app. I've had a really easy time learning View. I've really enjoyed it, just because of the same reason what you said, the easy onboarding.

Some of these more complex topics, like the Redux style store. I was so confused by them until I saw them in the context of View, and View really simplified this data flow and all the code management that goes along with the modern frontend app. But again, the big penalty that we've paid is you can't take very many View components off the shelf, because there just aren't many.

With React, like let's take the example of a media player. This is a podcast, so we wanted to develop this – an audio player that you could use in the browser. React probably has 50 of those that you can take off the shelf and import into your app. View have two or three, and one of them was buggy. Another one, we had to manipulate a little bit. Are there any other penalties for using View? Because you were saying, maybe it's not ready for production yet. What is it that makes you say that?

[0:47:44.6] MS: I think that the library, or the – I want to call it library, or the framework itself is ready for production. I think it's just that the ecosystem and community on React is just unparalleled. Whatever you look for, you find 50 different implementations for it, which is good and bad, because you have to choose between them. But it's good because it saves you a lot of work.

If you were searching for a media player, you'd probably find 50 different media player implementations and one of them will probably fit your purpose. That's pretty great. If you have a question, you can just go on Slack and there is, I don't know. There must be hundreds of thousands of questions by now about React and you can just – Somebody has probably had the problem before that you're having.

I think that's a good thing. I think that you can't go wrong with React, but I also think that you – currently this has a leg up on React in terms of the onboarding and the learning curve and stuff, as you see it's easy to learn. It doesn't feel very complex. I think they've proven that you can have a similar – you can build something similar to React in a way that makes it much easier to understand.

[0:48:49.1] JM: The last question. I follow Pete Hunt on Twitter. He's been tweeting a lot about web assembly recently. We've done a few shows about web assembly, but Pete Hunt is

tweeting very excitedly about web assembly. What's the state of web assembly and what does web assembly enable on the frontend? What are we going to see from it in the near future?

[0:49:15.0] MS: I think the most interesting implication is that you can write languages other than JavaScript and run them in the brand, because those languages might have different traders. JavaScript is a very flexible language. It does lots of things, but it doesn't excel in any. Like you can program JavaScript functionally, you can program JavaScript opted oriented, you can – it depends on what you want to do, which is nice, but it's also annoying, and it doesn't have static tanking, for example. There is different language with different tradeoffs.

We have compared to JavaScript languages, like elementaries and closer script and a bunch of others, that let you write your code in a different language and then compiled onto JavaScript with a same compiler type safety and functional style as the original languages. Imagine where you could optimize if you could run those languages natively.

That would enable, I think a whole new cost optimizations, where the browser could suddenly apply type optimizations and what have you, from older and much more mature languages that have types, and ever user could be running them in the browser without having its intermediary representation of JavaScript in.

I think that's very exciting, and I think – as far as I know, the one missing part there is that you can't access them from web assembly, as far as I know. You probably won't be seeing React being re-implemented in C++ anytime in the near future. But it's exciting. I mean, I think it's going to change the way the web works really. I don't think that's an overstatement to say, because someday you can – you no longer restrict it to three languages. You can write any language you want.

[0:50:53.3] JM: Okay. Well, Max I want to thank you for taking the time to come on the show, and I'm really looking forward to seeing how Spectrum evolves.

[0:50:59.9] MS: Thank you for having me.

[END OF INTERVIEW]

[0:51:03.6] JM: DigitalOcean Spaces gives you simple object storage with a beautiful user interface. You need an easy way to host objects like images and videos. Your users need to upload objects like PDFs and music files. DigitalOcean built spaces, because every application uses objects storage. Spaces simplifies object storage with automatic scalability, reliability and low cost. But the user interface takes it over the top.

I've built a lot of web applications and I always use some kind of object storage. The other object storage dashboards that I've used are confusing, they're painful, and they feel like they were built 10 years ago. DigitalOcean Spaces is modern object storage with a modern UI that you will love to use. It's like the UI for Dropbox, but with the pricing of a raw object storage. I almost want to use it like a consumer product.

To try DigitalOcean Spaces, go to do.co/sedaily and get two months of spaces plus a $10 credit to use on any other DigitalOcean products. You get this credit, even if you have been with DigitalOcean for a while. You could spend it on spaces or you could spend it on anything else in DigitalOcean. It's a nice added bonus just for trying out spaces.

The pricing is simple; $5 per month, which includes 250 gigabytes of storage and 1 terabyte of outbound bandwidth. There are no cost per request and additional storage is priced at the lowest rate available. Just a cent per gigabyte transferred and 2 cents per gigabyte stored. There won't be any surprises on your bill.

DigitalOcean simplifies the Cloud. They look for every opportunity to remove friction from a developer's experience. I'm already using DigitalOcean Spaces to host music and video files for a product that I'm building, and I love it. I think you will too. Check it out at do.co/sedaily and get that free $10 credit in addition to two months of spaces for free. That's do.co/sedaily.

[END]