EPISODE 985

[INTRODUCTION]

[00:00:00] JM: GraphQL has become a core piece of infrastructure for many software applications. GraphQL is used to make requests that are structured as GraphQL queries and responded to through a GraphQL server. The GraphQL server processes the query and fetches the response from the necessary databases, APIs and backend services. Around 2016 when GraphQL was becoming popular, a company called Meteor was deciding what to do with its business.

Meteor had been started off of the popular framework Meteor.js, which is a system for building real-time JavaScript applications. Meteor.js was loved by many developers, but the meteor company needed to decide if Meteor.js was the most viable opportunity that it could be pursuing with its resources.

From the vantage within the Meteor company, there were some trends in the frontend ecosystem that were potentially disruptive to the viability of the Meteor project. There were also some large potential opportunities. The dramatic changes to the frontend were coming from a downstream effect of Facebook’s open source technologies, specifically React and GraphQL.

Amidst these changes, Meteor, the company shifted its efforts towards GraphQL and renamed the company Apollo. Jeff Schmidt is the CEO of Apollo and he joins the show to talk about the GraphQL ecosystem, the business opportunities around GraphQL and the process of pivoting from Meteor to Apollo.

If you are planning a hackathon, check out FindCollabs hackathons. Whether you are running an internal hackathon for your company or you’re running an open hackathon so that users can try out your product, FindCollabs hackathons are a tool for people to build projects and collaborate with each other. FindCollabs is a company I started to allow people to find collaborators for their software projects, and our new hackathon product allows you to organize your hackathon participants to make your hackathon as productive as possible. Check it out at findcollabs.com.

[SPONSOR MESSAGE]

[00:02:14] JM: Apache Cassandra is an open source distributed database that was first created to meet the scalability and availability needs of Facebook, Amazon and Google. In previous episodes of Software Engineering Daily we have covered Cassandra's architecture and its benefits, and we're happy to have Datastax, the largest contributed to the Cassandra project since day one as a sponsor of Software Engineering Daily.

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

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

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

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

[INTERVIEW]

[00:03:54] JM: Jeff Schmidt, welcome to Software Engineering Daily.

[00:03:56] JS: Well, thank you so much for having me. It's really a pleasure.

[00:03:59] JM: GraphQL consists of a frontend interface and a backend middleware layer, and that backend middleware layer interfaces with existing data sources. Tell me what the frontend developer is doing to interface with that backend middleware layer. Well, the middleware layer and consequently the backends that are connected to that middleware layer.

[00:04:24] JS: GraphQL was all about how you connect your backends to your frontends, and in a modern application, if you go back 5 or 10 years, things used to be a lot simpler. You probably have a webserver somewhere that might talk to a database or a couple of databases, and then the thing consuming that application would be a web browser, which is almost like this passive viewer of the HTML and CSS that's coming down from the server.

Things have gotten a lot more complicated recently. We've entered this world of apps, and so you have not just web browsers but many different things all trying to access the services in your cloud and also instead of having one webserver, you probably now have many microservices, many databases, many APIs. You're trying to combine all of those different resources in the cloud to produce an experience for your user.

Whereas in the past, we used to send HTML maybe over the wire. Now ever app has an API inside of it, and GraphQ really answers the question what's that API that's going to exist inside our application? In the past we used REST to build those APIs, and the problem with REST is that you have to write a ton of custom code every time you turn around. Every time you build a new screen in your app, every time you build a new feature, you probably need a different combination of data from different combination of services.

In the old way, you probably had a backend for frontend team that was having to build a new REST endpoint for every screen in your app. GraphQL gives you a query language so you can flexibly ask for any combination of data and services you need so you can build app features a lot faster without having to write a lot of glue code every time you need to fetch a different combination of things from the cloud.

[00:05:57] JM: Why is GraphQL difficult to setup and operate using just the off-the-shelf open source repository or whatever the open source stuff is out there?

[00:06:09] JS: I think that actually it's got very easy to get started with GraphQL, which is super cool. The thing that takes a little bit more thought is as you start to scale from one or two developers using it to 5, or 10, or 20 people, and as you go from maybe one client talking to one server, how do you scale your graph? Because what we see is it's really easy for people to get started. You can probably get a basic graph up and running in an hour or two with that and you can get that connected to some React components or mobile app.

But what happens is your graph acts almost like this marketplace for data and services and like any marketplace has got this two-sided dynamic where once you built your graph, once you've taken a few different databases and services and created a scheme around it and made it available in this like very flexible query format, you'll find that you start using it for more and more features. You'll find that other teams inside your company also want to use that graph and you also find that you want to pull more and more data from a lot of different services or resources into the graph.

Now it's where you start to want to have some sort of set of workflows or processes or ways of managing the graph. You want to start to thinking about how you secure a graph, because you've got this very powerful query language. You'd never let someone send any SQL query. Every if they had access to the whole database, you wouldn't want to let them run arbitrary queries. You want to figure out how you secure the queries you make. You want to figure out how you do analytics and monitoring. You want to figure out how you – A huge part of the value of GraphQL is the tooling. It's really powerful if you can type that completion of your GraphQL queries right in your editor. You essentially can see the entire math of all your data right there as like IntelliSense tooltips and VS code. Compare that to REST where you're hunting for documentation for every endpoint.

You find that there's a lot of ways you can get more value out of GraphQL and there are a lot of things you want to do to really secure and [inaudible 00:07:51] your GraphQL API in production. That's where people start asking questions. As my schema grows, how am I going to manage the schema? Should I put that on a schema server? Should I version it? The other thing that you encounter is it's so tempting to add so much stuff to your graph that people find that there's this risk of creating a monolithic GraphQL server where you just have many different people all trying to contribute code to this GraphQL server you've built and you can open this problem where everyone owns it. So nobody owns it. That's where approaches like Apollo Federation for example can be helpful. You can federate the implementation of the graph over many teams and just have a central gateway server that acts kind of like a query planner. A query comes in. It's divided into pieces and sent out to these different backend services.

Ultimately, it's all about how you build agile workflows on top of the graph. One of the things that was challenging about REST is like REST really encourages you to think in this waterfall way of building APIs where you build a REST endpoint. You might be able to version your REST API. You might be able to ship a new version of your API every year or two. It's almost impossible to remove a REST API endpoint because you don't know who might be calling it.

Whereas GraphQL is a really good fit for a very agile approach to API development where you build against real user needs, you're constantly able to add and tweak things because you can have great analytics about who's using what, all which comes back to the declarative nature of GraphQL.

[00:09:12] JM: The frontend developer issues a query to the GraphQL middleware server and that query gets composed from whatever backend data sources and services the middleware layer needs to query, and you mentioned the term query planner, which you often hear associated with distributed SQL databases or just SQL databases in general and there's a lot of richness in SQL infrastructure and there's a lot of historical progress we've made in query planning on the SQL side. Is the processing of a GraphQL query of a similar richness, is it that complex? Is there that much depth to a GraphQL query such that we are going to have advancements in the query planning of GraphQL queries for years to come?

[00:10:13] JS: I think at a core, GraphQL is about an abstraction that you're inserting between your clients and your servers. In the past you used to have this really tight coupling. The core idea is here we're going to let clients describe the data they need. We're going to let servers describe their capabilities and we're going to create this flexible way to map the client's needs to the server's capabilities. Whereas in the past we would have written a bunch of code to do that, and now we have the ability to do that automatically. If you can describe the services capabilities with the schema, if you can describe the client's needs with a query, that's what makes it possible for us to think about this as an algorithm, like q query plan, instead of thinking about it as why I had to write a hundred or a thousand lines of code to get the data from point A to point B.

Yes, to answer your question. I do think there's going to be more and more that's possible in this layer of the stack. The good news is though even simple things with this layer work a lot better than what we were doing previously. While you can imagine a lot of cool things you can, some of which relate to query execution, but some of which relate to, for example, what does it mean to do security at this layer of the stack. What does it mean to think about GDPR and like understand where a data is coming from and where it's going now that we have this rich vocabulary for describing our data? Can we predict the load that's going to exist no our servers? For example, if I see a change that a frontend developer is making, can I work that back towards, "Well, we're going to ship this into this market which has this much traffic on this page or this React component." Can I then go to the backend team and say, "Hey, you might want to consider adding a new GRPC endpoint or something so that these clients will be able to fetch this data more efficiently based on the data we're trying to fetch."

I think there's a lot that we're going to be able to do overtime to make the queries even more efficient than they are today. There's also a lot –

[00:12:00] JM: How it's affecting the frontend developer's interface.

[00:12:03] JS: Exactly. Yeah. Some of this is around fetching data, but some of this is also around there are so many cool things that are possible inside this declarative way of thinking. For example, in a GraphQL, the ability to annotate parts of your query. There is something called at defer. You can actually mark – Something that's very important in any application is page load time or time [inaudible 00:12:27] interaction.

If you're building an application on top of REST, you might have a set of data you fetch first and you might render that and then in the background you might go fetch another set of data. That way you can have the key page elements or the key app elements, like be interactive first while you fetch the other. Maybe elements that are secondary important so the user is going to use second in the background. But that's a whole bunch of code you'd have to write.

In GraphQL, if you have all the layers of your stack set up right, you can do all of that declaratively. You can just mark a part of your GraphQL queries deferred. What will happen in the perfect world if you have everything set up right is, first, you'll get a packet of data that contains all the data that was sort of the non-deferred part of the query. Everything that's necessary to render the page initially. Then as the data that was deferred is available, it can get streamed from the server, like from all the different microservices through the server through down to your client. Using a client like Apollo client, that can get pushed directly into your React component or your iOS rendering components.

You're able to take what used to take a lot of custom code to build this really sophisticated functionality about incremental page loading to drive faster time defers interaction. But instead of writing any code, you just add one word indicating what should come first and what should come second. It's kind of comparable to the richness that existed inside SQL for databases. Before SQL your query planner was a human being. You had to figure out how your data was going to be stored and how you're going to lock the indexes and assemble data from different sources.

SQL gave us this declarative way of thinking. Now just by typing a couple of words, I can get any combination of data I want. That's really I think the place where the comparison to a query planner is really apped, because we have this declarative way of approaching what we're trying to do with data fetching.

But whereas for SQL, a lot of this is around – A lot of the practice is around performance but it's also around the ability to do more advance things like different kinds of aggregates and the different kind of databases that were more performant for certain kinds of workloads. I think there's going to be an analogy set of things for app development, like how do we manage maybe a geographically distributed cache? How do we manage things like at defer? How do we manage life updates? How do we manage security? I think those are some of the directions where now that we have this declarative language for thinking about our data and we're trying to do with it, that's where I think we're going to see over the next like 5 or 10 years a lot of awesome developments in this space.

[00:14:51] JM: The business opportunity for you today at Apollo, you built a platform around GraphQL called Apollo platform, and this has things like security management, metrics, federation. I think you mentioned that's kind of an enterprise.

[00:15:09] JS: No. That’s totally open source. Open standard open source.

[00:15:12] JM: Okay. All right. Well, tell me the developer experience of the Apollo platform. If I am using your web interface in addition to the open source GraphQL tooling, what am I getting out of that?

[00:15:26] JS: We have a SaaS service called Apollo Graph Manager, and as far as like what our commercial software offering is, it's mostly that SaaS service together with the fact that we will carry a pager 24/7/365 for the entire Apollo system, the various other ways we can support, offer expertise. But from a software point of view, everything is open source and then we have this SaaS service that’s complementary to it.

The core thing you get in the SaaS service is a graph management platform. Apollo client, Apollo server, there is lot of great tooling. As you kind of get beyond one client talking to one server or maybe just a couple of developers just building a single feature or two, you find that it's very advantageous to have not just the data plane, not just the place where the query comes in as process, but a control plane, a system for managing and understanding the contents of your graph and the workflows around it. Add a bunch of tooling to help you throughout the development process.

The core to this is a couple of services. One is a schema service. You would check your – You would never think of writing software without checking it into source control. It's very important to understand like not just which files are checked out on different people's laptops, but to have a source of truth about what your code is today. What different branches exist? What you're about to ship into production? Who made which change when?

Our schema server provides a single source of truth for the schema of your graph. So you don't have to use it. But by pushing your schema into a server and understanding the different versions of your schema, understanding the different branches of your schema and how it’s changing over time and understanding what your schema is as supposed to just saying what is a particular process exposing right now? It gives you a really powerful base to build a lot of other tools and workflow and processes.

So we provide the schema service. There is a client registry so you can keep track of all the clients that use your graph and what queries they send and. There is an analytic service that collects execution traces so that you can send it so you can build aggregated statistics or which clients are using which fields. What's driving your P99 latency like for particular screens in your app? It starts with these basic services.

Then on top of that, you can get a fantastic developer experience because now you can be using VS code, typing a GraphQL query, and as you're typing the query, we can show the predicted latency of the quarry right there in your editor based on line production and data, or you can set it up so it's inside your CI. Whenever you commit a change to a service that's in the graph, it'll tell you either this change is safe to push or this change is going to go break maybe a particular client, a particular version. It can do that based on the clients registered queries. It can do that based on actual production traffic. So you can see, "Hey, if I make this change, I'm going to break a client, but it's a client that only spent a set of five queries in the last 90 days and it's probably just an internal dashboard somewhere and you can go find a person who wrote that and go talk to them. That really helps with agility. How do you change your graph multiple times a day rather than once every other year? There's also a set of functionality to help you manage deployment of your GraphQL servers.

Especially when you start getting into a federated environment where you have like a gateway and then you have many different backend services, you want every single one of those teams to be able be on a separate development cycle all able to push changes whenever they want to. So you I need to be constantly comparing all the changes that everyone's making, validating that the types they're referencing other services are correct. Then also you want to have a continuous deployment strategy for your GraphQl service, for your central gateway so that even as you're changing all the graphs that go into it, you're keeping your gateway up-to-date.

There is a set of tools for generating the configuration files for those servers, pushing that configuration live and hot into these servers, managing a fleet of gateways. Really, the way I look at it, you really need some way of solving those problems. You need like a registry of allowed queries. You need a registry of your schema. You need some serve sort of analytic service. You could build all that stuff, and some people have. Then on top of that, there is a set of workflows and tools and so on that you want to build. We offer as a managed service and off-the-shelf version of that for people that don't want to make the investment in building all that tooling themselves.

[SPONSOR MESSAGE]

[00:19:40] JM: When I'm building a new product, G2i is the company that I call on to help me find a developer who can build the first version of my product. G2i is a hiring platform run by engineers that matches you with React, React Native, GraphQL and mobile engineers who you can trust. Whether you are a new company building your first product, like me, or an established company that wants additional engineering help, G2i has the talent that you need to accomplish your goals.

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

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

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

[INTERVIEW CONTINUED]

[00:21:28] JM: The idea the Apollo business, it is one of these things that looks pretty obvious in retrospect. You have this particular kind of server. You have a GraphQL middleware server and it has particular characteristics. It would make sense to have an infrastructure provider managing the APIs and the uptime of those servers. Tell me about the day one perspective of what Apollo, the business, was going to be. How did you identify that all of these problems were going to be core to the prototypical GraphQL user? Was there something – I mean, at this point, I guess Facebook had GraphQL widely deployed. Maybe there are some other companies that had GraphQL widely deployed. Were you looking at these other companies who were early adapters of GraphQL and identifying that they all had all these particular infrastructure problems, and therefore if you just built those things as a service you could have a viable business?

[00:22:40] JS: I think when we started there is almost nobody using GraphQL other than Facebook, and this would've been late 2015 I think that we first started looking at it. Let me tell you the story of how we got here. It's definitely some twists and turns. We're the team that originally built Meteor.js, the full stack JavaScript framework.

[00:22:59] JM: Big fan, by the way.

[00:23:00] JS: Oh! Thank you.

[00:23:01] JM: I was a user of Meteor. I loved it.

[00:23:03] JS: Yeah, and it's still going strong. It's still I think the fastest and easiest way to build new applications in JavaScript especially if there's any kind of real-time component. The genesis of Meteor was I guess way back in 2011 when we started writing it, we realized we'd cross some kind of a tipping point in app development where the old way is just weren't cutting it anymore. We used to be pretty happy with Ruby on Rails, or PHP, or ASP.net or fundamentally all these lamp stack derivatives that really trace the origins back to the 90s in some cases. This idea, "Hey, it's going to be – It's cgi-bin, man. We're going to generate some text. Send it over a socket, and that's how we're going to build our apps."

Sometime around within a couple years after 2010, we really cross this point where people were starting to expect a lot more and aim a lot higher. That meant that there was this need to put an API inside an app. You needed this way were you're going to send data over the wire instead of

HTML, and that was the original genesis of Meteor, actually, was a set of libraries to do exactly that data transport and solve many of the same problems that Apollo solves today.

But when we built the first prototype of that in 2011 and we built the examples for it, they didn't make any sense, because like we're trying to have something like Apollo client much less sophisticated back then of course, but we're trying to bind it to like bootstrap. If you remember – Sorry. Not bootstrap. I don't even remember, like these very simple JavaScript frameworks that existed back then, like jQuery, right? It's hard to build an application that has like a great story for fetching data off the Internet and really demonstrate the potential of that if you're in like the world of jQuery.

[00:24:39] JM: I remember that world.

[00:24:41] JS: So Meteor originated as taking all the other parts that you needed to put next to great data fetching to build a modern app development experience. The way that we were able to get that out the door in 2012 was by integrating everything very tightly, because all the stuff was new. By integrating a particular database, a particular fronted experience with a particular set of data fetching protocols and so on, we're able to build this amazing thing.

The limitation that Meteor has is that it's designed for new app development. It has a lot of opinions. You have to take all of its opinions. If you take all of its opinions, it's going to be great. But what happened was over the course of the next – Meteor at its peak was one of the top 10 most hard projects on GitHub. When we launched Meteor 1.0, I think there were local Meteor meet ups at 134 cities around the world on the same day. It got really big.

We had this huge advantage which is we're able to talk to the Meteor community, which included more and more enterprises trying to deploy Meteor at scale with like more and more complex application. Scale not just in terms of how many like people were using the application, but how many people were developing the application. How many data sources there were that the application needed to talk to.

We were seeing these crazy enterprise use cases where people were synchronizing data from many different places around the enterprise into MongoDB just to get into Meteor. They weren't using MongoDB for anything else, but because you had to use MongoD with Meteor.

[00:26:00] JM: I thought it was RethinkDB.

[00:26:03] JS: I always hope that Rethink would come out with great live query functionality that would make Meteor's life a lot easier, but it was –

[00:26:09] JM: It was Mongo. I'm misremembering.

[00:26:10] JS: Yeah. We made a huge investment in Mongo to like read the replication log. We re-implemented the whole MongoDB query engine and JavaScript on the client. Meteor accomplished what it accomplished [inaudible 00:26:22] like really intense computer science at the end of the day.

We were sitting here at the end of 2015, we were starting to think about Meteor 2.0 and we're going through the user requests for what Meteor 2.0 should do. We had heard clearly from the user community, it needs to work with more than just MongoDB. It needs to be able to pull in data from any source and it needs to work with any frontend too, not just JavaScript, because Cordova phone gap, like that had not solved every mobile app developer's problem and we are in a world where people wanted to use a wide variety of things on the frontend, React, React Native iOS, Android.

We thought we had a really clear mandate from the user community to do a second version of the Meteor data system that was data source agnostic and could integrate data from multiple sources. We'd also seen very clearly from the community that we want a declarative approach to data fetching. So we wanted to – In particular, we wanted to co-locate the description of a UI component's data requirements inside the UI component.

We'd also seen what it takes to scale to the world's largest, most demanding websites, but we needed to accomplish it in terms of just the traffic that something that our system could stand up to. The other thing we'd seen very clearly for Meteor is Meteor was based on this idea of like vertical integration, really. We're going to integrate all the layers of the stack to give you a wonderful experience. Then if you take all the parts for stack and the cost of that, you have less ability to swap out different layers of the stack and you have less ability to use it in existing applications.

we'd seen the rise of React, which competed with Meteor's UI system, Blaze, and we've seen how React was able to – From our point of view, like even though blaze I think was more established earlier and did many of the same things that React did, React was able to grow so much faster than Blaze and then Meteor because React could be used in existing applications. It was incrementally adoptable.

They didn't try to solve every part of the stack. They took one layer of the stack they tried to do it really well. We said for the next thing we do for Meteor, for this data system we're going to build to meet the needs of the Meteor community, we want to make really sure it's incrementally adoptable and compatible with everything and highly scalable both in terms of kind of like query throughput, but also in terms of the size of the teams it can use it.

That started this project to go build a new data system for Meteor, and one of the first things we needed was a query language, because we'd use the MongoDB query language as the core query language for Meteor. But now that we're going to support any number of different data sources, we needed a data source agnostic way to take different data sources and map them all back to a common schema or a language.

That wasn't a new idea for us. We actually almost went that way with Meteor in the beginning. There is a point in time and it was really Meteor versus Derby, and Meteor took this approach of being locked into one database, and Derby took an approach that's more about mapping to existing databases. Anyway, GraphQl was the right thing at the right time for us. I think it was shortly after Facebook had released this spec and we said, "This is perfect. It is a great language. It's really beautiful. It's proven at scale and its database agnostic."

We set out to make GraphQl the basis for the new Meteor data system and we decided to call something other than Meteor. We called it Apollo, because we wanted to clearly communicate that it was something you could use independent with the rest of Meteor. We started from the client out, because the first thing we needed to do was replicate the kind of functionalities inside Meteor, inside the live data system and the GDP wire protocol and a lot of that stuff. We took what we learned from building those systems inside Meteor and built Apollo client.

Our motto for Apollo client was by the community, for the community. We really focused on listening very closely to everyone's use cases and building functionality against like very concrete, very specific needs that people had. We really just continued that exact pattern over the years of trying to build great stuff that people love to use and then listening to those people and asking what problems are you running into. That led us to then invest more in the server-side when we heard about the challenges people are having on the server, which led to Apollo server. That led us to first build a set of management and monitoring tooling not long after that, which was ultimately formed the basis of Graph Manager. Then that led to us starting to build the schema sort of functionality maybe two years ago now. Then that also led to Apollo Federation which we worked on for about a year and released earlier this year. But all that's been driven by really just a dialogue with the user community and listening to people and asking like, “What's the biggest roadblock you’re encountering? How can we help you?” Just trying to do good kind of project management, like stack ranking the things that are the biggest friction points for the community and addressing that.

I think bringing back to your question, you asked like the model of a SaaS service that sits next to some open source is a way of building a business. That existed from day one for us. That was the way Meteor was, for example. I really believe in that model, which is open source all the stuff that should naturally be open source, and the things that should naturally be a managed service make that a managed service. Don't try to force your business model on a problem. Start with like what would be best for your users? Because I think if you don't do that in your business, you're going to get your butt kicked by someone who does.

The example I gave, I call this the complementary product model. You think about everything your users want. You say is there natural an open source piece and naturally a SaaS piece? GitHub is a great example of that. Yeah.

[00:32:03] JM: Beautifully said and there’s so much in what you just said that I want to discuss. But sticking to the subject of Meteor and the gradual product evolution to Apollo, it's a beautiful story because you have this one project that was having, at first, tremendous success but perhaps was undercut by its fully integrated, opinionated, I mean, necessity as supposed to Ruby on Rails where you can sort of take it apart over time. You can iterate one thing away and swap it out for another. I don't think that was as much the case with Meteor. Then you'd kind of tried to get there and you found that one of your pieces was actually more valuable than Meteor in terms of how much resource do we actually have to allocate for a startup, and then you eventually got to a point where you said, "We have to basically focus completely on the Apollo side of things even though the Meteor side of things may be viable as well." Am I understanding that correct?

[00:33:14] JS: Yeah. Yeah. Look. My North Star here, my personal mission in this business is to help out developers help the world. I think there's an enormous upside possible with information technology and I think we're only scratching the surface. Apps are still so time-consuming and so hard to write. I think we're barely 1% of the potential of all these cool stuff we've built, like mobile devices, operating systems, the cloud. The things you can build on top of that are incredible. But apps take so long to write, and I think that for my point of view, there is a huge role that app developers can play in just making a ton of cool things happen in the world if we can empower app developers to not spend so much time on boilerplate code, but have a chance to create awesome things [inaudible 00:33:54]. Because if you think about it, the app in many ways is really the point of value creation. We do all this work on the backend side and on the infrastructure. It's when a user gets to touch that app that it all pays off.

Meteor, that came from a goal of empowering app developers, and we've tried to sort of always ask what's the best thing we can do in that direction. Yeah. I mean, Meteor was a great business for us. It was profitable and it grew nicely and it's a really great solution if you want an integrated platform. But I think if you zoom out and ask how do we do our best job of helping out developers help the world, the answer is you need a coalition. You need a lot of people working together to build a platform that everyone buys into. The design properties inside Meteor that made it such a great integrated experience made it not impossible but harder to swap out the pieces.

Honestly, I think one of the biggest challenges inside the Meteor stack is just that stuff was written so early. It does have pretty clean interfaces inside of it, like Blaze can be used independently of live data. It can be used independently of iso build and like to get the other components of the stack. It's just all that stuff was written like before those ideas were really very widespread and Meteor faced such a difficult education problem in explaining to people like how like Meteor latency compensation or like just all these crazy stuff worked.

The consequence I'd add is I think, with Meteor, we got really far out ahead of the community in terms of what was going on inside of Meteor. You kind of needed to work past a certain point. You kind of needed the work at Meteor development group to understand how some of the technology worked, because there's just so much and it's hard to absorb.

we learned from that and now we think sometimes what's more important and more valuable than shipping this amazing algorithm that's going to make things 10% better for your users, sometimes what's more valuable is building a little bit more of a consensus. Building a little but more of a coalition. Taking things a step at a time. Getting one batch of ideas out there. Getting commentary from the community. Hearing other people's ideas. Letting that all kind of circulate and settle. Then okay, then maybe in three months or six months you take the next step as supposed to racing ahead and trying to ship a ton of futuristic stuff that may be really cool but hasn't really been developed in that like iterative fashion and dialogue with the community.

Meteor still does a lot of stuff that you can't get anywhere else, like automatic live queries for example. GraphQL will have that eventually. You can wire it up yourself today with a lot of effort, but Meteor had it from day one. That's the advantage. But I think that the advantage that we have on the Apollo side is by focusing on one layer of the stack and really pacing ourselves in terms of we are talking early about all the potential in the data graph. We want to build that together with people. We don't want to race ahead and figure it all by ourselves. We want to integrate with all the other stuff around us in the stack to create this really healthy ecosystem around app development.

[00:36:58] JM: Can you take me inside the company at the moment where you decided to sell off Meteor and focus completely on Apollo?

[00:37:09] JS: Let's see. That was quite a few years after we started working on Apollo, because our original vision was, “Hey, these things can coexist. Meteor will be a great –”

[00:37:20] JM: Difficult for a startup.

[00:37:21] JS: Yeah. It is hard. I mean, we also had – Meteor was increasingly mature, and the hard part looked like the Apollo side, not the Meteor side. We thought that Meteor could be a great out-of-the-box toolkit for building Apollo apps. If you want to bring this technology into existing application, you use Apollo. If you want an off-the-shelf preconfigured set of things that work great, that's Meteor. We were gradually making all the other – We've made a ton of like improvements and modernizations in Meteor to make it like work with NPM and React and all these sorts of things to keep it up-to-date with the development of technology.

The key learning for us though was Apollo started growing like really fast, like Meteor had grown fast, but Apollo grew really fast, because Apollo – The thesis about incremental adaption was actually right, and also Apollo got a lot more traction in the enterprise a lot faster precisely because the problem that Apollo and GraphQL solves about integrating multiple data sources at scale is it's a challenge that we all face, but it's especially a challenging phase in the enterprise.

We got to the point where we thought, "Well, we'll raise more money, hire more people," or all these ways you have as a startup and getting more resources. But we had this realization that as long as Meteor and Apollo existed inside the same company, Apollo was going to start Meteor for resources, because every time we had an incremental dollar, every time we had an incremental hour, we were going to spend it on the thing that was growing so much faster where we had like so many more users with so many feature request and so many more immediate things we could do that'd be so valuable with people. That's what finally led us to the search to say, "Hey, we need to find a new steward for Meteor. Something that's going to make this their number one thing and someone that's going to invest time and money in it because I really do believe in the power and potential of that platform and I know it's going to be a long time before we can bring all these stuff that Meteor has to Apollo."

We ultimately did find that partner in tiny capital who wants to invest in a platform, has a great history of building community. Has a great history of working with things that have a strong design-led component. I'm really excited about how that's all gotten wrapped up and what it means for the Meteor community.

[00:39:37] JM: I have to admit, I misunderstood what happened at Meteor from a distance. I thought that meteor had lost enough traction that your company had to search for another business, and that's not really what happened at all.

[00:39:57] JS: That’s not really what happened. It's more that –

[00:40:00] JM: But what I will say, it did feel like Meteor got perhaps disrupted by React. React became such a disruptive force. There is almost dangerous to be doing a lot of work on a frontend framework.

[00:40:14] JS: Meteor is great for rapid application development, but Meteor – I think many developers – I think there are times in history where people write software by starting with integrated frameworks, and there are times in history where people write software by bringing together lots of libraries because they want that flexibility and choice. They want to build this, I think to use MBA terminology, like a best-of-breed solution to their stack.

What happened is in the wake of 2011, 2012, I like to think like maybe partially inspired by some of the stuff we do with Meteor, there is just this explosion of like activity and creativity in the JavaScript ecosystem and so many new things were getting created. The idea that you would – For the most ambitious developers maybe or developers with the most resources or experience, the idea that you'd be tied down to a particular set of modules that is fixed, that it only changes slowly, means you'd be giving up the upside of like all these cool things that were coming out like every week that you could slot in. So that was really the challenge that Meteor faced. For the people that really liked to move fast and use cutting edge technology who are, if not the most influential, certainly the loudest people in the industry, like the people you hear about on Twitter. Meteor really lost the attention of those people that wanted to try the latest and greatest to combine lots of different pieces, because the whole idea of Meteor was to give you this very easy to use interface, and for us that meant stability.

That same property though made Meteor really appealing to a different group of people, people who could trust that the APIs were going to stay the same and that like the people who thought that JavaScript fatigue was a bug rather than a feature. It just meant that – It’s almost more of a maturity curve. Meteor moved on to kind of like the late majority part of the market instead of the early adopter part of the market.

It's a great, mature technology for people that don't want their framework to change every year. That's another way to tell the story, because one of the other key moments for me was we added React support to Meteor. We saw the React was getting bigger and bigger and we said, “Okay. Great. We’ll add React alongside Blaze as another option aside Meteor.”

One piece of feedback we got from quite a few people was I'd rather you hadn’t done that, because I like that Meteor has one standard UI library. I like that we have this whole ecosystem of packages that all can assume that you’re using Blaze. I like that we have these amazing full stack form processing packages that combine like client-side validation and server-side validation and like link to your database schema. It’s like all the pieces fitting so beautifully together to give this beautiful modular, high-level functionality. That's only possible when you depend on blaze and live data and many Mongo and like all these components.

One of the take homes there is like there're pros and cons to each approach. The fully integrated approach gives you things that you can't get if you have the interchangeable parts approach. But the interchangeable parts approach is what the early adopters are going to be using when technology is changing really quickly. I think that's kind of what the fork was. There are people that are very happy with Meteor. There are people that are very happy with the modern JavaScript, like combine lots of different libraries approach.

My prediction is that over some period of time, we are going to see in JavaScript more of a convergence toward standard frameworks. I think that day may be some ways away yet because there are still so much change that's happening. In the meantime, I think – The pendulum may yet shift toward kind of these like fully integrated approaches, kind of like how probably in the late 90s there was also a very turbulent period where people are inventing all sorts of new ways to build web applications, and maybe by the mid 200os, a lot of that had settled down and best practice has been figured out and you start to see the emergence of more things like Rails or more opinionated and more structured.

I think we're still in a period where like the thing you hear about the most on Twitter at least is going to be the approaches where you are combining lots of different pieces and keeping your options open. But I'm also hopeful that over the next I don't how many years, as we do find the right patterns that Apollo will ultimately be a part of whatever that standard stack is.

[SPONSOR MESSAGE]

[00:44:24] JM: Today's show is sponsored by Datadog, a monitoring and analytics platform that integrates with more than 250 technologies, including AWS, Kubernetes, and Lambda. Datadog unites metrics, traces and logs in one platform so that you can get full visibility into your infrastructure and your application.

Check out new features like trace search and analytics for rapid insights into high-cardinality data; and Watchdog, an auto-detection engine that alerts you to performance its anomalies across your applications. Datadog makes it easy for teams to monitor every layer of their stack in one place, but don't take our word for it, you can start a free trial today and Datadog will send you a t-shirt for free at softwareengineeringdaily.com/datadog. To get that t-shirt in your free Datadog trial, go to softwareengineeringdaily.com/datadog.

[INTERVIEW CONTINUED]

[00:45:28] JM: While we're on the subject of stuff in the future, you may not have any perspective on this, but if you do have a perspective, you're somebody who I really want to know your perspective on. Do you have any beliefs about how the no code or low code platforms might merge or intersect or interface with this quickly evolving choose your own adventure world of React and Apollo and JavaScript stuff on the frontend?

[00:46:01] JS: Yeah, absolutely. I think about what's an app. I think about that in kind of like this very broad way. I think of an app as anything we do that takes all this great stuff that exists in the cloud and makes it useful to people. Often, apps are about two things. They're about combining functionality in they're about distributing functionality.

If I think about something like Lyft or Uber, there's a whole bunch of services behind that. There is a real-time marketplace of writers and drivers and there's a navigation service and there's a review service and there's a payment service. Each of those things independently doesn't solve any problem that I as a user have, but if you combine them together, then it solves a problem I do have, which is I'm here and I want to be there and all these things like combine give you that great experience.

It's about combining capabilities into a solution and then it's about taking that solution and bringing it to wherever people are. If I could only use Lyft or Uber on my desktop, it wouldn't be that useful. It's really key that I be able to have it on my phone so it's in my pocket whenever I am. I think in the same way, what's going to happen in the future if we think about apps broadly is we’re going to find more and more ways to take all this cool stuff that exists in the cloud, and the cloud is like this like world of genies that can answer any wish you have if you combine the wishes in the right way so it's actually useful to you.

We’re going to find more and more ways, more and more platforms are pushing this stuff into the real world, whether that’s like mobile platforms, whether that’s Apple Watch, whether that's home assistance, whether that's like IoT or even just whether it's API so that I can take the functionality of company A and integrate it into website B so that whenever I want to listen to music, Spotify is there. Whenever I want to like travel, Expedia is there. I think that’s –

[00:47:42] JM: Like a service or a building block or whatever.

[00:47:44] JS: Yeah, as a building block, because fundamentally like what businesses do is they develop these capabilities and they want to figure how they can get people to use these capabilities, that distribution so they have to go to where the users are with like solutions work for the user.

Today, we do all that by writing a bunch a code. Every time we want to use cloud services in a different way, we have to go build a new app, and that just seems natural to us because it's the only way we know. But I think in the future, it's going to look crazy. I think we’re going to write – Still going to write apps in the future. I think we’re going to write more and more and more apps.

But I think in the future, you're not going to need to write a new app every time you want to do something in the cloud. We're going to find more flexible ways to do that.

I think GraphQL and the data graph is one of the enabling technologies for that, because right now if you try to solve that problem, you get immediately blocked by the fact that you have all these like very confusing custom ad hoc REST APIs out there. It's like a whole bunch of endpoints. Only a human can know what those REST end points do.

But with a data graph, suddenly the stuff in the cloud is introspectable, like you can query it and say, “What's the schema? What's available?” Now you can go into like your low code, no code editor and you can like visually view the schema and you can like bind it to UI components. There should be a beautiful drag-and-drop, like visual builder type interface possible here when we're able to build on top of a data graph.

Looking a little bit farther out, why can't I just ask my voice assistant for data and services in the cloud and like why can't there be a little bit of semantic metadata to help understand like what's out there? Do I have to write code every time I need to – I want to use things in the cloud?

The missing thing is this declarative layer. It's this abstraction layer so that we can kind a properly tag and expose all the services in the cloud, and that's exactly what the data graph is. I think we're going to see the concept of apps be broadened to be not just when we write code, but a lot of other low code, no code ways of interacting with data and services in the cloud. I think we're going to see the ways that those services are distributed, the way they come into our lives are going to be broader and broader.

Whether it's like AR and VR or like voice assistance, how many of these technologies will survive and flourish and what new things have we not yet even invented. I don't know. But it's going to become more and more ubiquitous in our lives. I think all of that's going to be powered by – One of the building blocks for that is better APIs instead of 20-year-old REST API technology that was designed to do something totally different.

[00:50:13] JM: Do you see this as a marketplace opportunity for you? As in Uber plugs in their marketplace API somehow and you expose a GraphQL thing and somebody can consume it using GraphQL, or am I just way off base here?

[00:50:29] JS: More and more companies are making their APIs be GraphQL-based. The reason for this is really interesting. All the reasons why you'd want to have a GraphQL API internally are even 10X more true if you think about the API you want to expose to a third party.

[00:50:46] JM: GitHub was the first one to do this, right?

[00:50:49] JS: Yeah. GitHub was an early one. Shopify, Yelp, some more examples. If you think about either GitHub or Shopify, let's look at the problem they were trying to solve. They said we want to be a platform company. We want GitHub to be the center of software development. We want Shopify to be the center of commerce. That means we need lots of different people available to plug into us. So we need to make it very easy for people to build on top of us and we need to make sure that those users have great experiences.

If you're Shopify, it's really important that if I have some plug-in that's going to sit inside my checkout flow that that thing load instantly, because every millisecond delay is going to hurt conversion. I've got to give them a good API. If you look at REST APIs, if you think about what your development cadence is like, when you're building an application, you're constantly building new REST endpoints, because the odds are good that the REST endpoints you have today aren't the REST endpoints you need to build the future you're building today. You need a slightly different combination of fields, slightly different combination of data. Your backend for frontend team is always buildings these endpoints for you or your backend team – You have to go to your backend team and ask them for these endpoints. It's a ton of code that can't make your app good. It can only make it bad. It could harm security, performance, maintainability. Just a bunch of glue code and there's a ton of it, but at least you'll work at the same company and at least somewhere up the management chain there's someone who you all report to and who thinks this future should get build and can help straighten out like everyone's priorities. That's not the case when you're talking about your public API where you're trying to enable your partners.

Your partners are like, “Okay. Hey, you guessed at some REST endpoints I was going to need. The odds that you got that right are slim,” and I'm going to be stuck trying to cobble together whatever I was trying to build on top of your scientific wild guesses about what I was going to need.” If I need a different endpoint, I'm going to have to go back to you and ask for it. It might take six months for you to ship that new version of your API with that endpoint.

It gets so bad that what you often find when people have APIs that are really important to their business, they end up assigning – They end up staffing those APIs with developers that are paired with their particular partners and their only job is to build the REST endpoints that those partners need. Just like you’d have a backend for fronted team inside your company, but you have to do it, like you have to do it with all your API partners.

The fact is that this idea that you're going to invest in an API and sure with your partners when you’re going to be done is a myth, at least with REST. Because with REST – REST APIs are inherently point-to-point. They're not like this flexible marketplace. It's like you can build point-to-point connections, but that's probably not going to be enough to enable your partners.

That's what's driven people to want to share their data graph to adapt public GraphQL APIs or GraphQL APIs for their partners, and we have a few people that have gone first. I know that there are quite a few companies that have this on their 2020 roadmap, and I think it's going to be really exciting time because it means that all the APIs we use, we’re going to have a much more flexible and fluid way to get at those data and services, and if we have an API, it means that we’re going to see – We’re going to unlock a whole lot more amazing use cases on top of the functionality we’ve already built. It’s going to be more ways to take all of our cool stuff and reach end-users not just directly, but through all these other people's applications.

If an app is all about combining related functionality to solve our problem than distributing it, like these public data graph APIs are such an important part of that because suddenly I can combine my functionality with another company's functionality to reach users.

We’re at the beginning of this, and I think the key thing about it is, to digress for one minute, people who have been fantasizing about this data graph thing for a long time. I think there’s a Tim Berners-Lee article in like 2001 where he lays out his vision for the semantic web, and his vision for the semantic web is like everything inside the data graph and then like 90% more stuff, right? Here we are in 2019, there're been so many attempts, whether it's like – So many different attempts to build this that have happened.

[00:54:52] JM: Diffbot is pretty good. I don't know if you've seen Diffbot.

[00:54:55] JS: Yeah, it's awesome.

[00:54:56] JM: Google had one, Google Knowledge, whatever, or something.

[00:54:59] JS: Well, also, there is RDF, there's Sparkle. There is an enormous investment by W3C around this. People have been trying to do this for a while.

[00:55:09] JM: By this, in case people have lost the thread, basically a queriable way of targeting any object on the way.

[00:55:18] JS: Yeah. A queriable API. So first I'm going to have some kind of a schema, a map of your data, and then I'm going to give you a query language.

[00:55:26] JM: That is something richer than HTML.

[00:55:28] JS: Yeah, exactly. Objects with fields of some kind at minimum, yeah. In some way I can query that and say, "I want a little bit of this. A little bit of that. Join it this other thing and wrap it up in a bow and give it to me."

The key insight that's made it work this time in 2019, it's about the fact that the users have to lead the way. Here's what I mean by that. There is one point of view on how you would build a data graph or semantic web or whatever terminology you want to use that says we should bring in a bunch of analysts, and the analyst should interview every one of my business and they should draw the layout of our date on a whiteboard.

[00:56:01] JM: Oh no.

[00:56:03] JS: This is really appealing if you're coming from a consulting back or an academic background. There are a lot of problems that can be solved this way. What happens at the end of a year? You've got a whole bunch of whiteboards full of diagrams. If you implement that, you're going to find that it’s not what users want and the users will give you valuable feedback about why it stinks. You maybe spend another year. Now you’ve got another set of whiteboards, and again you’re a year out of date.

The approach of having it be driven by sort of analysts doesn't work because you never capture like what's really going on fast enough or in a complete enough way. It's a waterfall approach instead of an agile approach and for all the reasons that doesn't work in other areas of the software. It doesn't work to build the map of all your data.

The other approach that people try sometimes is say, “Okay. Well, let’s just go to the backend team. Let's go to the people that maintain the databases. They know the land of the data, right? They should build the map of the data.” This was sort of the approach that something like OData tends to take. The problem with that is they know how they implemented, yes, but do they know how it's used? You end up with – The value of the data graph is it serves as an abstraction layer between the users and the suppliers of something so that you can refactor how your backend works without disturbing all the clients in the field.

If you start from the backend or you start from the database layout, you end up with something that’s really tightly coupled to your implementation and you lose a lot of the benefits because you’ve effectively just thrown away the biggest benefit of the system, which is that the clients are independent of the implementation of your backend. You've also probably made it more fiction than there needs to be for the users of your API, the app developers, to use it because, well, you never asked their opinion. You just started with, “Hey, I already know like the layout of my GRPC or my Thrift endpoints. I know the layout of my SQL tables.”

The thing that makes the data graph work so well is that it’s an agile approach and it's focused on the users. Like in any endeavor in life, if you start with the users in mind, things often go better. Data graphs are typically built by product engineering teams that take this very incremental approach, “I'm trying to build this feature. I'm going to take – I need pieces of data 1, 2 and 3. I need that in my graph so I can power my future okay.” Tools like Apollo server make it very easy to like in the space of hours or even minutes like just define a type, like write a couple lines of JavaScript to go fetch data to whatever other API you have and put it into this form and you're up and running with the graph.

You keep running that loop, whereas you're building features, each feature motivates what you add to the graph. That means that the graph is always being built from the point of view of the end user and it's always in the structure that's going to create the most value for that end-user. Then together with tools like a polygraph manager or having a schema server or having good CICD, you can get to the point where you don't have to version this and release a new version every year. You can be constantly changing it as you learn. It's those two things that have made it possible to solve this problem that a lot of people had given up on, which is building this like larger organizational map of your data.

Now, I'm going to bring it back to what we need to do to get to this world of public APIs, because we can see the value to be there. We need to figure out how we're going to apply these same agile practices to public APIs. You as the user of an API, you need an easy way to give feedback about what's working for you and what's not and feeding that back into the people that are providing the API. You as the person that's providing an API, you need to have a really fine-grained understanding of like how are people using your API. What's working for them? What's not? What are they building? What's driving demand?"

I think it's really building that tooling. An API isn't like a waterfall, like throw it over the wall and run away. See you in a year kind of thing, but it's a constant like workflow or collaboration or dialogue. That's what we need to do to get to the point where APIs on the Internet work right so that we can really build all these – Realize all these cool possibilities that should come from being able to connect all of our services on all of our capabilities.

[01:00:02] JM: What's the hardest engineering problem you've had to solve at Apollo in recent memory?

[01:00:07] JS: I think the hardest problem we've had to solve recently was Apollo Federation. Apollo Federation is our technology that lets you instead of having one monolithic data graph, you can build many different separate GraphQL services and combine them all into one graph. I think the reason why it would've been easy to build something that sounded really cool and that you could build a cool demo or two against. I think that the nature of something like Federation though is that it has to come from real-world use cases.

Federation was a very interesting experience in talking to a lot of people who are using one of our previous solutions for this, which is called schema stitching, and hearing what was working for them and what wasn't working for them. It turns out though the final spec is pretty simple. There's a ton of complexity inside there that captures, like it kind of mirrors the complexity that exists in real-world environments. How are you going to reference an object, for example? Is it going to be sufficient to have like say that every object has an ID and a global namespace of IDs? That's something that was tried previous in the GraphQL community. It was like challenging for a lot of people in production for a lot of reasons. It worked very well at Facebook because like they actually do have like one graph and every graph has a node. GraphQL at Facebook is almost a query language for a database they have, is this like bit graph database.

[01:01:29] JM: Right. They are a semantic web.

[01:01:31] JS: Yeah. It's a little bit of a different use case in some ways on the backend ide. On the frontend side, it's the same use case. But on the backend side, like the way a lot of that's implemented is actually a pretty different perspective. As you look at the complexity that exists in the real-world, you discover that sometimes people – Like they're going to want to have different primary keys for different objects. They're going to sometimes going to want to have primary – Compound primary keys. They're sometimes going to want to have compound primary keys for which some of the elements that go into the key exist not in the object in question, but inside an object that's related to the object.

For example, if I have user and a user has a username, but the username is unique only inside of a particular country. Then maybe I need to recurs into like the address object and to the country code. That's not quite the right example, but you see what I mean, or like if we think about execution of a GraphQL query being distributed across several servers, because the alternative is a monolithic server which we talked to so many people where there graph was getting so big that having one codebase maintained by one team that contained their entire graph was like just not working anymore. You have to think about the actual way, like in a perfect world, the data layout is super clean and beautiful. In the real-world, different services reference objects with different keys. Data isn't de-normalized in a different way. Different services need inputs from other services in order to compute the things that they need to compute.

Federation was a really satisfying experience of talking with a lot of users and thinking, “Wow! These problems look really complex,” but it turns out that like all the complex problems really were just in many ways pointing a finger at a couple of things that had to be true about the design, and when we got those things right, the rest fell out really naturally.

It's been very gratifying to see more and more people adapt Federation and to see how it's also a very simple spec in the end, and so it's been so much fun to see this get adapted across the GraphQL community from servers in many different languages. I think that was a hard problem that we got right.

The other thing I'd say about the design is there's a lot of other cool stuff we want to do there. We left probably 80% of it on the cutting room floor. Following up on what I said earlier, you take this stuff a step at a time. Ship a little bit, get feedback, see if you got it right. Then you're in a much better position. The next step you want to take, you can bring that to people and say, “What do you think about this?” If the community has already sort of internalized and critiqued and responded to and gained some production experience with the first step, you're going to get a lot better directional input and course correction and prioritization of like that next chunk of functionality.

[01:04:04] JM: We’re already up against time, but on the note of Federation, I think the Kubernetes community is also very early in its days of how Kubernetes Federation is going to be done. But this idea of you have this infrastructure that today because of all these legacy croft is very heterogenous, we are starting to have a vision for how it becomes more homogenous. As it becomes more homogenous, the way of providing some consistency and some control is is that Federation model, whether we’re talking about GraphQL servers or just servers more generally with containers.

Last question, you founded Monument, which is a live work event space in the city. This is one side project you have. I am sure you have other side projects. Do you find that side projects, if you're thinking – If you're objectively called about it, are side projects useful for thinking laterally about your business or do you think they're ultimately distractions?

[01:05:15] JS: I think it depends on the side project. I think it depends on you too. I think for me, peak performance is about getting the right balance of divergent and convergent thinking. Divergent thinking is when do you go broad? When do you generate lots of ideas and kind of explore the ramifications in lots of different directions and like open lots of doors. See what's behind the doors.

Then convergent thinking is, “Okay. I've got this whole universe of possibilities, but now we have to do something. Now we have to finish something.” Let's select and edit and window that down to the point that we have something really crisp that we can execute on.

I think that, for me, I really try to invest in divergent thinking. You have to put an equal amount of energy. I think at the convergent thinking, if you're going to invest in divergent thinking because otherwise you end up like diverged. You've got a thousand ideas, like you're sitting in front of a whiteboard –

[01:06:02] JM: I've been there. [inaudible 01:06:03]. You'd probably been there too.

[01:06:05] JS: Yeah, or another saying I have, little freedom is the ability to do anything you want. Big freedom is the ability to do exactly what you want. Take your pick. You have to make some commitments if you want to good deep. You have to give up a little bit, a little freedom to get some big freedom.

Yeah, I think it's all about kind of having the right portfolio. Think of things that broaden your horizons and things that you really focus on and get good at. For me, so much of what I do is motivated by people. What I really personally love is when I can share a magical experience for someone, and that was part of what drove the early days of Meteor. It's part of why I love being a startup founder, because when I see people join our company and they get great career outcomes and they get to talk at a conference in front of hundreds or thousands of people and talk about the future software development and like have that moment in their career, like that's a magical experience for me.

When app developers are able to build things faster than they otherwise could and you can cut out all that like boring stuff that makes you hate your job and build the stuff that like you find really like energizing and empowering and exciting when you get to do something that just like blows a user’s mind. I love giving that to people. I get a lot of energy from that.

A big reason why I do Monument, I just love meeting people. I love meeting people on a broader San Francisco community outside of tech, and it's gotten so hard for people that don't have tech jobs survive here. So a big reason why we do that project is we want to provide a low-cost event space for events that can afford the venues that are so expensive now. Some lower cost places to live for people that they're doing something with their life that maybe is just as valuable for society but doesn't pay in the same way as some of the awesome jobs we have in tech right now. For me, to have those people around me and be a little bit more multidimensional, it helps to remind me like why I work so hard. It's energizing for me.

[01:07:47] JM: Jeff Schmidt, thanks for coming on the show. Really great conversation.

[01:07:51] JS: Thank you so much for having me. It's such an exciting time to be alive if you’re a software engineer with all of these planetary-scale systems we’re trying to figure how to build. I look back at where we were 10 or 20 years ago and it feels like things are moving so slow, but like so much has happened and it's always awesome to get a chance to sit down with you and reflect on like some of the things that are changing our world.

[END OF INTERVIEW]

[01:08:19] JM: As a programmer, you think an object. With MongoDB, so does your database. MongoDB is the most popular document-based database built for modern application developers and the cloud area. Millions of developers use MongoDB to power the world's most innovative products and services, from crypto currency, to online gaming, IoT and more. Try Mongo DB today with Atlas, the global cloud database service that runs on AWS, Azure and

Google Cloud. Configure, deploy and connect to your database in just a few minutes. Check it out at mongodb.com/atlas. That's mongodb.com/atlas.

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

[END]