EPISODE 793

[INTRODUCTION]

[00:00:00] JM: Modern web development tools have given frontend developers more power. On the frontend, JavaScript frameworks like React and View have become easier to work with. For deployment, tools like Netlify and ZEIT give developers a workflow that is tightly integrated with GitHub. At the database layer, autoscaling documents storage systems, like Firebase and hosted Mongo solutions make it easier to work with objects.

There are also a multitude of APIs that give developers a rich business functionality out of the box making it easy to build applications around SMS, payments and computer vision. If you're building a new application today, you have the option to build around a completely serverless architecture. As the backend and frontend have changed, the middleware to communicate between those layers has also evolved. GraphQL is a modern way of fetching data from disparate data sources.

In previous episodes, we have talked about how GraphQL works and some common patterns for using GraphQL in mature applications. In today's episode, Tanmai Gopal joins the show to describe how to use GraphQL in newer applications. Tanmai is the CEO of Hasura, a company building tools around GraphQL. He discusses the advantages of using serverless functions together with GraphQL and how to architecture an event-based serverless application.

We have a few events coming up. We have a meet up of Software Engineering Daily at Cloudflare on the 3rd of April. Haseeb Qureshi will be sitting down for a conversation with me, and you can sign up for that event by going to softwareengineeringdaily.com/meetup.

Also, the company I'm working on called FindCollabs, is having a $5,000 hackathon. You can join that hackathon by going to softwareengineeringdaily.com/hackathon. We're having both a virtual hackathon, which anybody can come to, and also an in-person hackathon on April 6th at App Academy. We're going to be getting together at App Academy. We're going to be having some food. We're going to be hacking on some cool projects. So if you are looking for

collaborators for your projects, check out Find Collabs and you can enter to win that $5,000 price purse.

Thanks for listening, and let's get on with the episode.

[SPONSOR MESSAGE]

[00:02:39] JM: Triplebyte fast-tracks your path to a great new career. Take the Triplebyte quiz and interview and then skip straight to final interview opportunities with over 450 top tech companies, such as Dropbox, Asana and Reddit. After you're in the Triplebyte system, you stay there, saving you tons of time and energy.

We ran an experiment earlier this year and Software Engineering Daily listeners who have taken the test are three times more likely to be in their top bracket of quiz scores. So take the quiz yourself anytime even just for fun at triplebyte.com/sedaily. It's free for engineers, and as you make it through the process, Triplebyte will even cover the cost of your flights and hotels for final interviews at the hiring companies. That's pretty sweet.

Triplebyte helps engineers identify high-growth opportunities, get a foot in the door and negotiate multiple offers. I recommend checking out triplebyte.com/sedaily, because going through the hiring process is really painful and really time-consuming. So Triplebyte saves you a lot of time. I'm a big fan of what they're doing over there and they're also doing a lot of research. You can check out the Triplebyte blog. You can check out some of the episodes we've done with Triplebyte founders. It's just a fascinating company and I think they're doing something that's really useful to engineers. So check out Triplebyte. That's T-R-I-P-L-E-B-Y-T-E.com/sedaily. Triplebyte. Byte as in 8 bits.

Thanks to Triplebyte, and check it out.

[INTERVIEW]

[00:04:29] JM: Tanmai Gopal, you are the CEO of Hasura. Welcome to Software Engineering Daily.

[00:04:34] TG: Hi. Hi, Jeff. Hi, everyone.

[00:04:37] JM: It's great to have you here, and I want to talk about GraphQL as well as some other trends in web development that surround GraphQL. Let's start there. What are the trends in modern web development that are driving people towards using GraphQL?

[00:04:55] TG: I think there are a bunch. So in order of power perhaps, the most powerful trend is that a lot of work is shifted towards the frontend. So frontend developers are kind of like becoming the new king makers of a business, right? The experience of consuming APIs so far has been terrible for frontend developers. With GraphQL, suddenly their experience becomes really nice. That's kind of one of the major drivers for GraphQL.

Just as an analogy that I sometimes like to think about is when we were looking at the transition from SOAP to REST a while back, why did we move from SOAP to REST. It is not a technical limitation. It was that the kind of development was changing where the new kind of work that a business was doing was not just kind of building code that would talk between a bunch of modules within an org. It was becoming this kind of thing where human beings were integrating APIs that were written by other people and building their own API.

When humans were looking at APIs, looking at like these RESTFul JSON APIs, it just made things so much easier as compared to looking at these XML APIs, and that was one driving force behind kind of switching over to REST and JSON. Similarly today, when a lot of that work is shifting to the frontend, the experience of just using REST JSON APIs, making multiple API calls, etc., etc., a bunch of problems there, is just terribly painful. Replacing that with GraphQL is amazing. So that's kind of one big reason.

The other big reason is the other kind of trend that is driving people to GraphQL is that GraphQL brings in a certain element of typing to your API, and this is again a common thing when kind of systems end up becoming more complicated. You go away from being like rapid prototyping and hacking things together into something that is better typed so that you know the type of what you're going to get when you make an API call. When you make an API call on a REST API, the

REST API can return anything. It is just the kindness of the API developer that they ensure that it returns what the documentation says it did.

But a GraphQL server technically by spec will only return your data that you kind of requested for in the shape that you requested for, which means there is kind of a type to it even at build time for both the server and the client. These are kind of the two large trends that are kind of driving people to use GraphQL. The first trend where it's easier to use as an API is definitely the more powerful one.

[00:07:32] JM: How does application architecture change when an app is built around GraphQL instead of the more traditional REST model?

[00:07:42] TG: Yup. I think that's an emerging question. I mean, that's why the answer to that is still kind of evolving. There are different ways of thinking about this and I can speak a little bit from experience from what I'm seeing. So there are three kind of broad approaches, right? The first approach is that it's what Facebook had, which is that, "Hey, we have this monolith. This is a very cool monolith, and it's a very large monolith. It's very complicated. We have one API layer, and this API layer is going to be natively GraphQL." So it sort of exposing RESTish APIs. They expose GraphQL APIs. This happens kind of in their monolith.

This is nice and GraphQL really kind of – If you look at it, was designed for the monolith, because when you build the GraphQL server, there is a whole type system to the GraphQL API. This makes it very convenient to work with a monolith, because with a monolith, what you can do as a – If 20 developers are kind of working together and they're all building a GraphQL schema, they're building the GraphQL types of their API, if there is a clash somewhere, it will conflict at a code level. They'll fix it. It's easier to deal within a monolith, right? It's not delegated out. You have everybody kind of building their own types. Then ultimately the GraphQL types will conflict. It's not one endpoint. The idea of having one GraphQL endpoint, one set of types, works really well with a monolith, and that's the way Facebook designed it.

Now, unfortunately, the rest of the world does not use monoliths, because they find it really hard to scale ownership with monoliths. So it's easier for an organization to kind of scale ownership with microservices. When people have microservices, the approach to using GraphQL is a little

different. In fact, what people are doing also is not refactoring existing microservices, because, for them, these REST APIs already exist.

So the approach that's kind of emerging there is to say, "Let every team build a GraphQL wrapper." So they wrapped the REST API in a GraphQL API and every team kind of exposes a GraphQL API. Then what it also happening is that instead of wrapping over a single service, you build a GraphQL server that wraps over multiple services. So kind of like a gateway. So instead of having a backend for frontend, or instead of having like one REST API endpoint that then talks to 20 other REST services, you have a GraphQL kind of gateway that talks to other REST services. That's kind of another emerging architecture.

The third architecture, which we see with Hasura users and which I'm super excited about is this kind of using it with serverless, right? Which is this other trend that is also happening with modern web development and modern kind of like application development, and there, kind of what is exciting to me is this idea of saying that you have a GraphQL API. Some parts of your GraphQL API can also be written inside serverless functions, and so you have these different serverless functions that expose different sections of your GraphQL API, and then you kind of stitch them all together with a gateway. So your teams or developers can kind of own these individual GraphQL resolvers or pieces of the GraphQL API that are then stitched together, and then the stitching process needs to be a little bit automated or has a process around it to ensure that you don't conflict, and that's your GraphQL API.

But then what you do is to leverage kind of serverless even more and the benefit of serverless. What you do is instead of trying to put a lot of logic in this GraphQL API layer, which is what you typically do in an application, right? Whether GraphQL or REST, you have this tick middle layer. Instead putting more logic there, what you do is that the GraphQL API talks to some kind of a stateful layer, like a database or an event system. Then what that does is that has an eventing setup and that goes in triggers other serverless functions asynchronously that run your business logic.

So the app talks to a GraphQL API. GraphQL API talks to some kind of stateful layer, which then goes and triggers events. Events go and trigger other serverless functions and maybe set off a

workflow, gasket a workflow where there's a lot of business logic happening, and that's kind of one cycle.

But if you think about this cycle, I mean, this is not unique to GraphQL, right? You could have done this with REST APIs also. The cool thing with this is that a lot of your business logic can run asynchronously. But the problem with this architecture historically, if you did not use GraphQL, was always that when stuff happened asynchronously on the backend. So let's say you started off a process to asynchronously run a machine learning algorithm produce something. Maybe you inserted some text in the database and now you want to automatically translate that text, which is a long-running process, and that runs off as a serverless function. That's perfect fit for serverless, perfect fit for eventing. Everything is great.

But when the translated text arrives back into your state, how does the frontend know that the translation work is done? That the file up load is done? That the Google Maps kind of translation of the address into a lat long is done? These are like asynchronous tasks that happen. How does the frontend know that this is done? This is where GraphQL shines, because your GraphQL API can actually be a real-time API, which means that you can subscribe to events as they happen on your stateful layer, on your database or wherever, and the frontend can actually consume those changes as they happen over GraphQL subscriptions.

So to kind of complete this complex loop, the idea is the app goes to a GraphQL API, GraphQL API does some stuff, talks to the state. State triggers events, goes to serverless functions. Serverless functions may modify state again on the backend, and backend state modifications are asynchronously consumed by the frontend app. This is a really cool architecture that is becoming increasingly popular with the kind of more modern developers where they're willing to kind of move almost everything to like 100% serverless stack, right? They don't want to invest in writing monoliths or use like traditional frameworks and stuff like that.

[00:13:59] JM: Mitigating natural disasters is one of the world's greatest challenges. The past decade has been one of the worst periods for natural disasters, and while they may be inevitable, they don't have to be catastrophic. That's why IBM created Code and Response, a

four-year $25 million deployment initiative that puts open source technologies in the communities where they are needed most to win the challenge and help build software solutions to help with natural disaster preparedness. Head to softwareengineeringdaily.com/coderesponse to learn more and access hundreds of code patterns to help you build your solution faster.

Thanks to IBM for being a supporter of Software Engineering Daily.

[INTERVIEW CONTINUED]

[00:14:51] JM: I share your excitement around the changes in architecture, and to make it more concrete how this would work, I think we should walk through another example. So let's say I'm building a mobile app today. Let's say I want to build SE Daily Instagram. So it's like Instagram, but you can only take pictures of computers, and if you try to upload a picture that is not of a computer, you get your photo rejected. How would I architect that application?

[00:15:22] TG: So in this kind of like super modern way of doing things, right? What you would do is your app will do a streaming upload to a file service, something like S3. I really like this question because this hits upon one of those weaknesses of GraphQL where GraphQL does not have a binary streaming spec built into GraphQL.

So for binary data, uploading and downloading, you typically use traditional kind of APIs, RESTish APIs. You don't actually use GraphQL. In this architecture, what you would have is – and in this kind of like real-time GraphQL server SE, event-driven stack, right? What you would do is you would have your app, and this is the SE Daily Instagram, and then this app now needs to kind of upload a post or create a post.

So what you would do is you would create, you would run a GraphQL kind of mutation. You would go create a post, but this post is not marked as created. What the frontend would also do is upload the file to S3 or to some streaming upload, which is as fast as can be. When the upload finishes with S3, S3 will emit an event thing that the upload has been done. This event will be captured by a serverless function that says, "Oh, cool! The upload is done," and this is

the metadata of the upload, and the metadata of the upload might contain like post information and a bunch of other things, right?

This serverless function will then take this metadata, the URL of the file of S3, and put it in the database and put that in like your primary database wherever you're storing posts and information about posts. The serverless function will put that information there. Once it's there in the database, you might emit one more event for a serverless function to go and compress that image. So you might want to thumbnail that image, because the users uploaded this kind of gigantic 10 MB thing and you want to optimize that or maybe the app has a lot of work done inside the app and maybe the app is optimizing it, whatever.

So the serverless function on the back might go in and further compress this image and save that URL, again, with the same pattern. You go compress it, save it to S3, S3 event, comeback and save the smaller, compressed image in another field in the table and you're kind of updating the table as stuff is happening, as these uploads happen.

Meanwhile, what the app is doing is that the app is subscribing to the post ID to check whether everything is done with a post, right? This is if in case the user wants to see whether the post upload was done successfully or not, or the user can actually navigate away and you don't have to – The user doesn't have to wait for this upload to happen, the thumbnailing to happen.

But in case you want to show that feedback to the user, the frontend app is also subscribing to that post object in the database, and as these fields come in, as the post URL comes in, as the compressed post URL kind of comes in, the app kind of gets this notification real-time that, "Hey! Okay, cool! Post upload is done. Here's the image." "Compression has been," or whatever, "Here's the image," or maybe that's too much detail and you want to show that to the user, so you don't have to show that to the user. So that's what it would look like.

The cool thing here is that this stack is scalable and kind of resilient out of the box. You can have 10,000 users and it won't matter, because S3, which is a managed service for uploading files will scale. These are emitting serverless functions, which are all completely independent of each other, right? It's not like an API server that you need to scale manually. That will keep scaling automatically. These are going and talking to a managed database, which might be

RDS, which is a managed database where you have like easy ways to scale that up and down the way you need to. And your GraphQL API layer, that's the beast that you really need to build, but that's also kind of becoming easier to build, or if you're using a system like Hasura to take care of that for you, which is like a managed GraphQL API. Well, then that's also super easy to build. That's also something that kind of scales automatically and you don't have to manage it.

So that's kind of what's neat. You're building a fairly complicated real-world backend that's doing a bunch of things, but everything kind of keeps working automatically, and it's very nice because this system, if you think about it, is kind of resilient out of the box, right? Imagine a file upload process, which is usually very hard to build if you're building your own API. If you have your own API, you're saving a file to disk or you're saving the file to S3. What is that upload crashes or fails and you already have something on the database? You uploaded something to S3, but then something went wrong. How do you recover from this state where you might have some files in S3, but you don't know what their URLs are in the database? You're in this confused state that you need to kind of make sure that you handle well. If your API restarts, you can go look at S3, find dangling files, either delete them or go update them again in the database. It's painful.

But if you think about this event-driven thing, when S3 upload finishes, the event is emitted. Not before, not after. 100% guarantee given to you by like the vendors of these systems, right? So that means that if the mobile uploads the file, you will have the file. This can happen 10,000 times. This can happen zero times. This can happen 10 times. It's extremely simple for me as a developer to not know backend things, concurrent things, multi-threading things. I just build stuff. I build my app, I build my serverless functions, I model my database, I use these services. That's it. So that's kind of what it looks like. Does that make sense?

[00:20:38] JM: Definitely, and I agree that the benefits of this architecture are numerous, and it makes it quite a good time to be a developer relative to 5, 10 years ago when a lot of these stuff was much, much harder to do. One theme in your explanation is the evented architecture. So, for example, when a photo gets uploaded, we might consider that an event. When the photo gets saved to S3, we might consider that another event and maybe we want to compress the photo or do some other manipulation to it.

In any case, this connection between the serverless world and the event-driven world is – We've explored that on previous episodes. How should this type of plication be managing all these different events that are getting created and triggering other things throughout the system?

[00:21:31] TG: Yup. I mean, that is essentially the question in this, right? I'm guessing that you mean things like how do you manage a workflow of events and stuff like that. Is that kind of what you're asking?

[00:21:42] JM: Right. So there're a number of things. First of all, when a photo gets uploaded, there's going to be an event that's triggered. Where is that event getting triggered? Where is the processing taking place? Then is the event and abstraction that we need to put into a queue? Then if we have this, like you said, with the workflows, if you have something like a banking application where you have a transaction that you want to – Like a banking transaction and first you want to make sure that the money is in the account and then you want to make sure that the money transferred successfully to the other account and then you want to make sure that both accounts are sane and there was no kind of double spend issue. You might have 5 or 6 different steps in a workflow.

If we're talking about building these applications from serverless functions, then we start to get into state management. But before we get to state management, I guess I want to focus on this question of events. How do events get created? Where are the events getting managed? Do we need an event queue or is that somehow handled implicitly? Tell me about event management.

[00:22:48] TG: Got it. Got it. Right. So touches on a few things, like you said, right? But the primary idea is that any stateful that undergoes a change emits an event. It is a responsibility of the stateful system to emit events, right? So in this case, the stateful systems, if you think about it in the SE Daily Instagram app were S3 and was the database. These were the only two stateful pieces. S3 was having files getting uploaded, and then the database was having references saved for those files, or references saved to the thumbnailed version of the file. So these are the two stateful systems.

So these two stateful systems must emit events. So AWS, for example, and most of the cloud vendors now, for all of the stateful abstractions that they give you, they give you events. So they

give you events when things happen. S3 file uploading, file upload finished, bucket created, file created, whatever. They have a bunch of these events. So that's one kind of source of events.

The first problem with eventing is kind of capturing events and the responsibilities on the stateful system to capture events automatically. So the stateful system that you're looking at should have 100% guarantee on event capture. Nothing should happen that should result in an event not being captured. So that is piece one.

The second piece is event delivery, right? Once this event is captured, how is it delivered to your serverless function and how does that process become reliable? Then actually coming to the next part of your question, how was it item potent? How was it exactly once so that you can avoid double spend, right?

So, for example again, if you're looking at these managed state services, for example, S3, S3 emits an event and there you have a guarantee, an exactly once guarantee, that this will call your serverless function exactly once, right? This will happen exactly once.

So you have that, you have kind of that guarantee where you can take that S3 event and then directly use that to call your serverless function, right? You have another alternative where you can take this event and put it into a queue, and something like SQS, your own queue, Kafka, RabbitMQ, whatever, you can ask AWS to put it into a queue for you or you can manually put it into a queue by taking that event and then just doing a simple thing where you just update their database and says – Or update it into a queue, right?

This is useful for those situations where you think that your processing of the event might be failure prone. So if the processing of the event is kind of like a pure function where you're guaranteed that there is no failure that will happen inside the serverless function because it's not dependent on any external kind of other services or API calls. It's a pure kind of transformation and then update. You can get away with it being called directly by your stateful system and you don't need a queue.

But in case you think that this might fail and you want to retry it, right? Maybe, for example, if you look at this image, you might want to run a machine learning algorithm that extracts the

people who are inside this image, right? But when you run that machine learning algorithm and you call that API that is giving you the ML algorithm, that API might have a rate limit. So that means that you are not going to be able to process this once in one shot. You might need to retry this again, right? Because you're hitting a rate limit because of an external dependency. In those cases, you want to take your events and put it into a queue and the queue should kind of give you this kind of retry mechanism.

On the database side, you're looking at the same problem where when the database is kind of doing things, you want events, events should be captured and then delivered. There are different ways of doing this at Hasura, and we have an open source project which does this, which does the eventing on the database on PostgreS today and eventually other databases, where we capture kind of changes that are happening in PostgreS. We store them in a temporary queue and then we do reliable delivery of these events that have been captured.

So you kind of deliver this to a serverless function. There's a retry logic. Your serverless function can reply with the retry after so you can kind of implement this exponential back off and say, "Oh, I'm hitting a rate limit. Try again after 10 minutes." You try again after 10 minutes, I'm hitting another rate limit. Try again after 20 minutes, right? So you can kind of implement that and the queue or the Hasura kind of queue will take care of delivering this and making sure that it's delivered at least one.

The double spend issue is a more complicated issue, and there are a bunch of ways around this, right? One of the ways around this is by doing a transaction. So whenever you are speaking to one stateful system, you can leverage all the properties of that one stateful system. For example, if your serverless function needs to do a transaction on a single instance database where you need to reduce amount from account one, do some checks, increase amount and account to log this transaction and then finish the transaction, right? This whole thing is one transaction happening with one data system, with one database, that gives you the abstraction of a transaction, like a database, like RDS. It gives you a transaction.

So from your serverless function, go ahead and run a transaction. It's absolutely fine, just like you normally would have in code, because the system allows you to do it. But in a more complicated case, you're looking at maybe reducing the amount from a particular user, sending

it off to an external system not knowing if it's processed for a longtime, and that's a more complex scenario.

In that complex scenario, the queue that is doing event delivery to you, the queue must give you exactly one's semantics, right? Because the queue in any kind of error case cannot afford to deliver that same event multiple times, and this is not just failure. This is because most queues that deliver events give you exactly one's semantics, and this is to protect against network failure. Because when you look at eventing, it's complicated.

I might have sent the event over HTTP, but I never got the act back, but the event was delivered to you. You can consumed it, but the 200 status that you responded saying that, "Hey, I've got this event," or whatever, or over the network, the event delivery mechanism didn't receive that. So now it doesn't know where you received the event or not. So it sends you the event again, right? Then you process the event again for no fault of your own, because there's a network in the middle, right? This is a common problem. You have to implement like a two-phase commit kind of thing to solve this problem or other techniques. But your event system needs to give you exactly one's semantics is one way around the problem.

The other way around the problem is that when your serverless function is doing things, you make sure that your serverless function is item potent. So that means that no matter how many times I run the serverless function for the same payload, it will not cause a repetition. It will not cause a double spend. The serverless function itself can be retried like infinite times with the same event and that will count as one spent. That's a little bit harder to build, but again, the tooling is kind of getting better.

One of the ideas that we've been playing around with is making API calls retry proof. So imagine that your function is calling an API call, but instead of making an API call directly to whatever API, whether it's Stripe or A, B, C, whatever your API is. Instead of doing that, you make the API call through another system, and this system actually saves your event payload, like your request payload in a queue, dedupes them and makes the API call only once.

So even if you try making the API call again, you just get the previously saved response for that event, for that request payload. So as long as you have this kind of guarantee that your requests uniquely identifiable, this is not a problem.

So there's active work happening in that area. There are cool things happening there, but that's kind of what the eventing scenario looks like, and the ecosystem is coming up with a bunch of tools. The cloud vendors are coming up with tools. People like us at Hasura are coming up with a bunch of tools to make this eventing more natural.

Does that make sense?

[00:30:42] JM: Definitely. One thing I find interesting about this architecture is it's very new and it seems really, really useful for applications just getting started or ones that want to scale. But we don't really have any large scale case studies. So what I think will be interesting is the applications that are getting built today with this kind of architecture, when they become Uber's scale, or Twitter's scale, or Facebook's scale, what are the architectural challenges they're going to encounter? Have you seen any applications that have really kit scale with this kind of architecture? Do you have a sense for kinds of problems emerge as an application like this scales up?

[00:31:28] TG: I honestly can't say that I have, because I don't think they exist quite yet. So I think maybe nobody has. But there are two interesting facts to this, I think. One is that in a lot of cases, your entire application might not be 100% serverless, but you're adding these components into your system. These kind of managed serverless components into your system, right? That's happening, and that's happening even at scale. Even at scale, a particular kind of workload, right? I talked about this translation example, which is asynchronous, or a thumbnailing example. These are ideally done async, because nobody cares about it happening synchronously. It's not important enough for that, but it's good to have it done eventually.

So that's a very common way to kind of use a little bit of serverless and eventing in your existing stack. But the event-driven system that I'm referring to, all of the companies that you see at scale, they've implemented this kind of system internally. They cannot scale without having events. If they have this one API that is synchronously dependent on other APIs to do things,

like I have an API that talks to service one, and then talks to service two, and then talks to service three to do something, and then service three fails. Oh, shit! I need to rollback service two. I need to rollback service one. But is there a rollback on an API call? No, right?

So most of these web scale companies, their state management is actually evented internally. They've built a lot of that infrastructure to do that eventing which then internally goes, captures that event, puts it into a stateful event log and then you go process that event and do something else. I mean, the rise of Kafka so many years ago, a lot of that – I mean, they initially started off for logs, has become kind of this event system, this event bus in your org, right? So eventing has been known and has been used as the method to solve problems at scale. Are those events then going and talking to serverless functions? No. I mean, because serverless functions weren't around or a thing until very recently. But your own implementation of like these stateless APIs, your own implementation of these microservices, these different microservices and different teams connected to each other via event queues. That is the way of using microservices.

If you're using microservices where APIs are directly dependent on each other, that will stop scaling very fast. I mean, that will become a pain much faster than a monolith will become a pain. When you start seeing network failures with their microservices, you will wonder, "Why don't I just go back to my monolith where at least these stupid network failures didn't happen?"

So the most common way of solving that is by having your microservices talk to each other asynchronously via eventing. And that's already been done, and that's happening for a long time. Every single company and scale has eventing underneath to ensure kind of protection against like these arbitrary failures, against transient failures to be able to scale easily, to be able to retry things easily. Because if an event is captured in the event log, you know that you can process this eventually, right? There might be a small failure, but you have the event. Just go replay the event. Everything will work.

So that's already happening. Bringing serverless into the mix is just making that architecture accessible to everybody. Instead of like these large engineering teams building internal infrastructure at these web scale companies, the vendors and other tooling vendors are kind of doing that for you so that you can leverage the benefits of eventing without the pain of setting up

an eventing system and so that you can leverage the idea of writing these simple, pure-ish kind of serverless functions that then act on these events, and that's making this architecture kind of more accessible. But this has actually been around for a long time.

[00:35:21] JM: So we've been talking about applications that manage to go in this direction from a greenfield, and I want to talk some about adaption of GraphQL and the related set of technologies that we've been discussing. I guess more focused on GraphQL, because I think if you're a bank, for example, that's the kind of non-greenfield application that typically is my go-to, because there are lots of banks that are old, but are still doing quite successful and are updating their architectures overtime. But if I'm a bank and I'm adapting GraphQL and I'm perhaps adapting serverless or cloud technologies, but I think that's a little bit further off. If I'm adapting GraphQL, what is my adaption strategy? How does it compare to the greenfield strategy?

[00:36:10] TG: It's a little bit more tedious. I think going back to one of your first questions about GraphQL, with a greenfield, you can afford to kind of build around GraphQL QPI. You have a native GraphQL API. There's only one layer in the system, instead of a REST API, it's a GraphQL API. I think they're easier.

When you have an existing system, the typical way of – There are two ways of moving to GraphQL. One is – And this is the most popular method that's easy for people to reason about, is that they take their REST APIs, they add another layer before the REST API. This is frequently owned by maybe the backend team if the backend team is empathetic and cares about the frontend people, or sometimes in a lot of cases it's the frontend team that actually takes up this ownership of saying, "We'll write our own GraphQL service on the back."

This GraphQL service that's on the backend is stateless, doesn't do much. Is deliberately thin on business logic, is deliberately thin on doing things that might make it hard to think about performance or scaling, and this service actually just exposes a GraphQL API, but internally delegates that to existing REST endpoints, right?

So the actual business logic, the work, the authentication, the security, as far as possible is done by these upstream REST services which have existing code in them and you don't bother

those back in developers, right? You let them work as they've been working in their own microservices and you're completely independent of that.

But what you do is instead of consuming those APIs in an app, you consume those APIs in a GraphQL service. Then this GraphQL service exposes an API to your app, right? This is a common migration method. It's not nice. It seems almost like this is a step while we're transitioning, and maybe this is not the final step of what the world will look like when everybody is on GraphQL, because you're introducing another layer. One more layer is one more component to manage, one more component that can fail. One more network hub, one more thing to worry about when it comes to scaling your system up, and that's a common problem.

But if you look at a lot of legacy systems that were on SOAP that wanted REST and the SOAP developers didn't want to change, you added a REST interface on top, like a REST layer on top of it as an external service. That used to be very common, sometimes still is. So that's kind of one method.

The other method that we see with Hasura users because of what the Hasura product does, like you basically look at – You take a database and you expose a GraphQL API automatically on top of that database and you have these kind of access control semantics to control how your GraphQL API does security or does application security and exposes the underlying database models, right? Whether you want to tweak them a little bit, you want to change the names, you want to do a little bit of restructuring on this GraphQL API that you expose, which is automatically real-time and does all the hard things about performance and has a bunch of all of that work done, right?

So very commonly we've seen that users have existing databases. They run their existing logic on these REST APIs, but they read from the GraphQL layer, because GraphQL is optimized for reads like nothing else, and it's mostly a read problem that GraphQL solves really well. That ends up being a good intermediate path to say, "I have this high-performance GraphQL read layer that I can use for my app. For writes, I can continue using the existing APIs. I can even wrap them in GraphQL mutations and then stitch that together with the read API. So my GraphQL mutations will go to REST APIs, which will go and modify state. If I want to read stuff or subscribe to stuff or have real-time stuff, then I'm going to use this auto-generated GraphQL

layer that is secure that I can talk to." So we're seeing that is also a common kind of extension to existing applications that we're seeing, but these are kind of the two broad approaches.

[SPONSOR MESSAGE]

[00:40:10] JM: Today's episode of Software Engineering Daily is sponsored by Datadog, a monitoring platform for cloud scale infrastructure and applications. Datadog provides dashboarding, alerting, application performance monitoring and log management in one tightly integrated platform so you can get end-to-end visibility quickly, and it integrates seamlessly with AWS so you can start monitoring EC2, RDS, ECS and all of your other AWS services in minutes. Visualize key metrics, set alerts to identify anomalies, and collaborate with your team to troubleshoot and fix issues fast.

Try it yourself by starting a free 14-day trial today. Listeners of this podcast will also receive a free Datadog t-shirt. Go to softwareengineeringdaily.com/datadog to get that t-shirt. That's softwareengineeringdaily.com/datadog.

[INTERVIEW CONTINUED]

[00:41:16] JM: So you are building a company around GraphQL. In thinking about these two kinds of application developer types, you've got greenfield applications and applications that have been around for a while. Do you have a sense for where the market is for building a business around GraphQL?

[00:41:37] TG: I think it's early to say. I think all of the companies that are kind of in the GraphQL space are actively like trying to figure this out. I think there are a bunch of different ways to think about this just like any other business, like building a business around REST, for example. What were the businesses that people built around REST? That is a good – Whatever problems they solved there, that has a good analog for what you could theoretically build a business around when it comes to GraphQL, right? So if you think about just REST itself with all of the stuff around API management, an API security and API collaboration, right? There's so much stuff that was happening there, which have resulted in large businesses, performance monitoring, analytics, collaboration tooling, and a lot of that can't potentially come into the

GraphQL world where people who are moving to GraphQL will lead these systems to help them deal with GraphQL, or deal with GraphQL APIs. So there's definitely an opportunity of that kind.

I think it's hard to build businesses that help you build GraphQL APIs, which are like frameworks. Developer frameworks are not a good business, because, I mean, they should be open source. I think if you look back over the last maybe decade or two, cases of being able to monetize a successful framework, or successfully monetizing a framework even if the framework is widely successful, has been quite hard.

So being able to monetize that is not easy, and that's in fact what the Meteor folks who are now doing a lot of the Apollo work, who've kind of championed the accessibility of GraphQL, who are doing – Still do. That's been hard. That is hard, or at least that's what it seems like on the outside, and I am not sure what the internal details are.

So on that front, enabling GraphQL or solving very specific problems that use GraphQL that exploit or kind of bring off the value of GraphQL, I think there are opportunities there. I mean, honestly, this is surprising for our team as well. When we open sourced Hasura 7, 8 months ago and we kind of launched it, we expected most for users to be building greenfield applications, like building new applications, because you put a tool like this out, it becomes very common. That gives you a GraphQL API that's real-time. It's most common for people who are kind of building new things to use this. Of course, that's a huge chunk of our user base. But within this short amount of time, we've seen a fairly large number of enterprise use cases running this in production where they're basically using Hasura to modernize their legacy database.

They have an existing database. There's data inside it. Nobody wants to use the existing APIs to deal with that. So they just kind of throw this layer. It's super high-performance, tiny footprint, but it's optimized just for solving this kind of tiny use case around being able to read, write and do real-time stuff really fast and really well. So they kind of just use this layer and they get a GraphQL API. The real benefit to that is that the frontend teams are freed up and can work faster, right?

In a sense, it's a business built around GraphQL, because we're exposing GraphQL APIs. But in a sense, it's built around like maybe another kind of business that you would have seen analogs

with before where it's just speeding up stuff and automating components that you don't have to maintain anymore, which is fairly common in an enterprise technology, that you have a bunch of these components that you don't build, that you don't maintain that do it for you.

So that's the opportunity that we're exploring, but I'm sure there'll be kind of lots more as the GraphQL ecosystem opens up, as adaption widens. As a lot of the early adapters today are people who can build their own stuff, build their own infrastructure, build their own tools, do their own thing. But the world a few years later will be – A lot of people will want to use GraphQL, but they don't want to invest the resources in building the infrastructure surrounding GraphQL. That's where a lot of businesses will probably emerge just like what we saw with the REST world.

I mean, just to be clear. Today, if you think about it, there's no REST business, right? There's no business which was that company monetized REST. That's a little bit hard to say. I think the same thing will happen with GraphQL or should happen with GraphQL, because it's an open spec. So many years later people will look back at it and say, "This was the GraphQL company." I don't think people will say, "This was the GraphQL company." I think people will say, "This is amazing. I mean, GraphQL is amazing, and these are the product and the different businesses in the GraphQL space that help you get proactive with GraphQL," and that's kind of what should happen and what I think will happen.

[00:46:23] JM: Now that I have some perspective on your vision for how the business might unfold, let's talk about what you're actually working on. So describe what Hasura does, what the product offers right now, or I guess I should say the open source project and your vision for turning that into a business?

[00:46:41] TG: So the open source project, I mean, I think the way we think about it is there's the open source project that we have and then there's the business that we build around the open source project, which is hard on a good day. But the open source project today, what it does is we call ourselves a real-time GraphQL engine and a serverless eventing engine.

So what we do is you point us to database, this might be a new database or an existing database, and we look at the database and generate a GraphQL API for you. We give you

semantics to – I mean, we give you a rule engine to configure application security to say, "If a user has a rule called user, a rule called editor, a rule called collaborator, they can access this data if ID is equal to their session ID, or ID is equal to the cookie session ID, wherever the authentication systems is coming from."

You set up this kind of access control, you get this GraphQL API, and we solve the two hard problems with GraphQL. One, giving you good performance, because with GraphQL what happens is that you can make an arbitrary query, and this might absolutely like destroy your database. It's like the N+1 query problem, where your GraphQL API will end up making multiple calls with the database.

So we solve that problem by taking a very unique approach of compiling. We compile a GraphQL query into a single SQL query. We inject kind of like the write access control things and we compile that to a single query. You query for user and addresses. We'll compile that to a select star from user common addresses where ID equal to whatever, where the access controls are right. Then we do that kind of in one query, which is what you would have done if a backend developer was writing this and wanted to give you an efficient API.

The second problem is real-time. So we listen to changes on the database, and every time there's a change, we refresh the subscription that you have and then you get the latest result on the client, right? So on the client, it's kind of no nonsense, super easy to real-time, and we've done the hard work on the backend to make sure that it's scalable that it doesn't destroy the database when you start subscribing to change with the database. So that's what the open source project does.

When we launched this like 7 months ago, that's what it did. We added support for eventing soon after, because that's kind of what we were seeing with our users. So now whenever something changes in the database, we also call out the serverless functions or web books on the backend. This has nothing to do with subscriptions anymore. So that you can go and run asynchronous logic, and that's kind of what the project did.

But now when we think about the business, what we really want to do is – And the way we think about our role in the ecosystem is what we want to do is we want to make all of the data that

you have in your organization and in your team. We want to make that as accessible and as easy to consume as possible for the next generation of modern web developers, right?

So people who are building these rich applications on React, and Angular, and View, we want to make your data, your existing data, your existing APIs, databases, all of that, we want to make that accessible as quickly as possible to these modern web developers so that they can get productive quickly. These modern developers, not only are they using GraphQL in the frontend, but they're going to be writing their business logic in serverless functions.

So we need to make data accessible to these users who're building inside serverless functions as well, and that's kind of where our eventing piece comes in to say that this is how we expose your data safely to the stakeholders who're kind of building the next generation of technology and web experiences for you. That's really our role, where we're kind of making data more accessible for GraphQL and serverless. That's kind of where we think about like the business value that we're unlocking, and we do that with GraphQL and serverless today, and today just PostgreS, but eventually with other databases and with more different kind of API sources and stuff like that.

[00:50:43] JM: We started off the conversation talking about this SE Daily Instagram app type, and what's interesting to me is I really like building new applications and I really like the ease of use that modern software development offers, especially relative to like when I started getting involved in software like 10 or 12 years when it was not as trivial as it is today to spin up your own application fairly cheaply and easily.

Well, at least one layer that I think there's a lot of, I guess, subjectivity in right now is which backend as a service to choose? So if I'm architecting one of these projects, I could do it entirely on S3. I could do it across AWS, but you also have these rich backend as a service tools, like you have ZEIT, Netlify, Firebase, Heroku. For a developer that's playing in this world, I think they're looking closely at these different backend as a service products. How should a developer evaluate these different backend as a service products? Do you have a sense for how they compare in your mind?

[00:51:48] TG: Yeah, that's a very good question and that's something that we've thought about deeply as well. I think the backend as a service landscape has shifted a little bit from being more Parse and Firebase-like, which was kind of like an all or nothing to more ZEIT and Netlify-like, which is they solve tiny and most specific problems and more platformish than the actual API.

So that's kind of what's happening, and I think the way we think about when we build our systems and we tried to leverage as much of like the serverless stuff as possible when we set up our internal infrastructure and whatnot, the heuristics that we have are whatever code we write, the local development workflow, the CI/CD and where we deploy it should be vendor agnostic, right?

The code that I'm writing should not we written away that can only be run on AWS. Even though AWS has a specific format the way the serverless function runs, right? Because it's very easy to kind of just do this wrapping to – There's common tooling across your entire team, so that whenever you're writing stuff, it's not something that can only be deployed to AWS, or to ZEIT, or to Netlify, or to whatever. So ensuring that is important.

Whatever backend systems that you're using should either have APIs that are replaceable, or the component itself is open source. So that means that you're either using like a PostgreS-like database, which is open source where you have this guarantee that I'm not locked into RDS. I'm not looked into – I'm not even locked into PostgreS, because it's open source. PostgreS community has so much tooling that you can use to kind of migrate to and from a PostgreS database. So having these infrastructure components being open source is very good.

The good thing with stuff like S3, which is one of these other gigantic components in your stack, the great thing with S3 is that S3 has become kind of like a standard API. So now it doesn't matter if only AWS has S3, because almost all of the cloud vendors have S3 compatible API, and then there are companies like Minio, or MinIO, where they're basically saying, "Hey, on your own storage inside enterprise or on cloud or wherever, you can run Minio, and that will do the storage for you," but it will still give you an S3 compatible API.

So I think as long as the APIs that you're using are either replaceable or the infrastructure components themselves are open source and have a community around it, you're fine. You don't have to be too worried about what choice you make, because you can probably switch from one to the other fairly easily. At that point, the decision becomes more about what cloud vendor do you prefer because of other reasons. But that's the way I think about it.

[00:54:20] JM: Okay. Well, it's been really fun talking to you. Do you have any other predictions for what the next five years looks like for the world of GraphQL? What's going to change? What's going to improve?

[00:54:33] TG: I think the experience around using GraphQL and kind of consuming GraphQL and using GraphQL APIs is definitely going to become better and more accessible. That's going to happen. The tooling to kind of build your own GraphQL systems should also gradually become easier, and we'll see kind of different patterns around GraphQL.

I think one of the other things that we'll see is that the use cases for GraphQL will evolve significantly. Right now, it's kind of been born in this fairly specific use case the way Facebook had it. It's not gradually making its way to the rest of the world, and people have already started using very interesting ways where they use GraphQL as a data source for doing things. So GraphQL is becoming a declarative data source for creating static sites, and that's an approach that Gatsby is taking.

We just released an open source tool that uses GraphQL as a data source for building charts, for doing data visualization. So instead of thinking about data for that chart, you specify that as a GraphQL query, which is declarative, and then the chart just gets generated. There's going to be that approach where GraphQL becomes a data source kind of language, and then you have a lot of automated tooling once the data source is kind of specified, and there's going to be a lot of tooling around that because the productivity benefit of that is insane. You specify a GraphQL data source, stuff magically happens. Not just this kind of API use case, where I make the API call like a data. So that's going to happen. But I would be weary of saying anything more than that, because I think the ecosystem is rapidly evolving.

[00:56:10] JM: Okay. Well, Tanmai, thanks for coming on Software Engineering Daily. It's been really fun talking to you.

[00:56:14] TG: Absolutely. Thanks for having me, Jeff.

[END OF INTERVIEW]

[00:56:19] JM: This podcast is brought to you by wix.com. Build your website quickly with Wix. Wix code unites design features with advanced code capabilities, so you can build data-driven websites and professional web apps very quickly. You can store and manage unlimited data, you can create hundreds of dynamic pages, you can add repeating layouts, make custom forms, call external APIs and take full control of your sites functionality using Wix Code APIs and your own JavaScript. You don't need HTML or CSS.

With Wix codes, built-in database and IDE, you've got one click deployment that instantly updates all the content on your site and everything is SEO friendly. What about security and hosting and maintenance? Wix has you covered, so you can spend more time focusing on yourself and your clients.

If you're not a developer, it's not a problem. There's plenty that you can do without writing a lot of code, although of course if you are a developer, then you can do much more. You can explore all the resources on the Wix Code's site to learn more about web development wherever you are in your developer career. You can discover video tutorials, articles, code snippets, API references and a lively forum where you can get advanced tips from Wix Code experts.

Check it out for yourself at wicks.com/sed. That's wix.com/sed. You can get 10% off your premium plan while developing a website quickly for the web. To get that 10% off the premium plan and support Software Engineering Daily, go to wix.com/sed and see what you can do with Wix Code today.

[END]