EPISODE 682

[INTRODUCTION]

[0:00:00.3] JM: GraphQL allows developers to communicate with all of their different backends through a consistent query interface. A GraphQL query can be translated into queries to MySQL, MongoDB, Elasticsearch, or whatever kind of API or backend is needed to fulfill the GraphQL query. GraphQL users typically need to setup a GraphQL server to fulfill this query federation to the different data sources.

Prisma is a tool for automatically generating a GraphQL API and resolving and serving GraphQL queries. The developer defines a data model and deploys with Prisma. Prisma generates the necessary GraphQL infrastructure to resolve and serve those queries from the developer's data sources.

Using Prisma can allow the developer to get up and running faster than they would if they had a setup GraphQL infrastructure themselves and define that middleware query layer by hand. Prisma is an open source project, but it's also a company. The opportunities to build a business around a GraphQL infrastructure layer are numerous.

In recent episodes, we've explored the complexities of the data platform, from newer companies like Uber, to older companies like Procter & Gamble. Engineers are struggling to find and access their data sources. Data engineers and data scientists spend months configuring their infrastructure to connect to BI tools and run distributed queries.

GraphQL could simplify data platforms by providing a unified standardized layer. At this layer, you could also offer things like caching virtual datasets, and crowd-sourced queries from across your company. Søren Bramer *Schmidt* is the CTO and cofounder of Prisma and he joins the show to discuss why GraphQL has become so popular, and he also talks about why Prisma works and the opportunities to build developer tooling around GraphQL. This was a great show. I'm really interested in Prisma and the GraphQL community more generally. So I look forward to doing more shows in this space.

Before we get started, I want to mention that we are looking for writers. We're scaling up our written content and we're also looking for podcasters. We are hiring for both of these roles and several other roles. You can find them by going to softwareengineeringdaily.com/jobs. I'd love to see your application.

[SPONSOR MESSAGE]

[00:02:44] JM: DigitalOcean is a reliable, easy to use cloud provider. I've used DigitalOcean for years whenever I want to get an application off the ground quickly, and I've always loved the focus on user experience, the great documentation and the simple user interface. More and more people are finding out about DigitalOcean and realizing that DigitalOcean is perfect for their application workloads.

This year, DigitalOcean is making that even easier with new node types. A $15 flexible droplet that can mix and match different configurations of CPU and RAM to get the perfect amount of resources for your application. There are also CPU optimized droplets, perfect for highly active frontend servers or CICD workloads, and running on the cloud can get expensive, which is why DigitalOcean makes it easy to choose the right size instance. The prices on standard instances have gone down too. You can check out all their new deals by going to do.co/sedaily, and as a bonus to our listeners, you will get $100 in credit to use over 60 days. That's a lot of money to experiment with. You can make a hundred dollars go pretty far on DigitalOcean. You can use the credit for hosting, or infrastructure, and that includes load balancers, object storage. DigitalOcean Spaces is a great new product that provides object storage, of course, computation.

Get your free $100 credit at do.co/sedaily, and thanks to DigitalOcean for being a sponsor. The cofounder of DigitalOcean, Moisey Uretsky, was one of the first people I interviewed, and his interview was really inspirational for me. So I've always thought of DigitalOcean as a pretty inspirational company. So thank you, DigitalOcean.

[INTERVIEW]

[00:04:51] JM: Søren Bramer *Schmidt*, you are the CTO and cofounder of Prisma. Welcome to Software Engineering Daily.

[00:04:57] SBS: Thank you, Jeffrey. It's great to be here.

[00:05:00] JM: So Prisma is an open source project that is related to GraphQL, and I'd like to ease into an explanation of Prisma by starting with GraphQL. I think most people listening to this are faintly aware of what GraphQL is at least, and if they're not, they can listen to some other episodes. But you can start with just explaining why GraphQL became popular. Why is this thing useful?

[00:05:28] SBS: Absolutely. I would start by saying this, if you're not super familiar with GraphQL, go and make yourselves familiar. It's quite a big change to how you interact between clients and servers, and it is raising in popularity at the moment. Where does GraphQL come from? GraphQL came out of Facebook. They had this problem when they were building the mobile app and they were structuring the teams in such a way you had a bunch of different backend teams. Then you had very small frontend team.

What they faced is all the normal issues you have when you query a REST backend. You do multiple round trips. Your applications get slow and you write a lot of boilerplate code to deal with all of these. So most companies, they go through phase where they see, "Oh, it's quite messy what we have here. How do we clean this up?" So they start imposing standard restrictions to the way that they develop their REST API endpoints.

What GraphQL is, is really just a formulation of those standards. All the best practices that people have developed overtime in REST APIs, it is written down into a nice cohesive spec that is maintained by [inaudible 00:06:32] from Facebook. Because this is now a standardized effort, you see many companies, many community members rallying around this one standard and creating share tools that everybody can benefit from. That's how you should think about GraphQL. It's all the good part from REST, standardized in a way that the community can work together.

[00:06:53] JM: At a typical company, you have various teams that are working on various parts of the product, and the communications between those different teams may not be as fluid or as friendly as at a company like Facebook. At Facebook, I can see how there could just be this virality of GraphQL propagating through an organization and people latching on to GraphQL. How does the adoption work at a more typical company? When does a typical company start using GraphQL?

[00:07:31] SBS: It's really interesting actually when we go out and talk to some of these bigger companies that have some of the issues you're alluding to. You could call it [inaudible 00:07:39] or probably just really teams trying to protect themselves. What you see is that the adoption of GraphQL is typically driven by frontend teams, teams that are creating mobile apps, or websites, and they are tired of having to deal with this sprawling REST API that's just highs and down. One thing is the performance benefits you get from using GraphQL, but the other thing is just the speed and agility you have, because you can start writing your data requirements in this declarative language instead of having to maintain a bunch of different API calls and synchronize it all.

What you often see is the frontend team will either embed a GraphQL server directly in the frontend and ship it to the browser, or they will set up their own proxy GraphQL server that does just one thing. It accepts an incoming GraphQL query and then it has the responsibility to delegate to all the existing backend services.

[00:08:34] JM: What about the differences between like a big company that's already established and they're trying to onboard with GraphQL versus a startup that is starting a greenfield application and they're having GraphQL from day one. How do the user experiences of those two types of companies compare?

[00:08:52] SBS: One thing is implementing GraphQL when you have a big system already. When that is your perspective, the you typically try to just wrap the existing system. But when you're starting from scratch and you're a little forward looking and you think, "Hey, we're going to do GraphQL. So do we really need to build REST APIs? Do we really need that?" Maybe you do, but many companies find that they don't. So those are the companies we call GraphQL native. They start with a GraphQL mindset from the beginning, and they built their micro-

services and the API endpoints as GraphQL. What we found really useful is a technique called schema delegation that allow you to take a subset of a GraphQL schema and delegate the responsibility of resolving data for that scheme to other services.

One way you can do that is to say a subset of the schema is being delegated to a data service that is implemented on top of Prisma, for example. We see a lot of small companies starting like this, and we get into the details a little bit later. But what that allows you is to get an initial service up and running really, really quickly, and then actually grow in sophistication. You can slowly peel off layers and start adding more complexity into the system. That's probably fairly abstract explanation. But the main point is that when you start from the beginning thinking you want to do GraphQL, you can get a lot of benefits by architecting the application with that in mind from the beginning.

[00:10:15] JM: Let's review quickly what happens in a normal GraphQL query. So if I am a frontend developer and I issue a query to my backend GraphQL system, what happens? What does that query look like and what are the different steps involved in processing and returning the results for that query?

[00:10:38] SBS: First of all, you need to understand what is the structure of a GraphQL query. A REST query is for one particular endpoint, for example, users, get me the user entity with a specific ID. That's a REST API. If you need, let's say, two users and you need all the tools from that user, you end up making at least four calls.

In GraphQL, that is not the case. In GraphQL, you are submitting a three-structured query. You could think of it as a JSON document just containing the keys, not the values. Then it is the responsibility of the server to take this GraphQL query and fill in all the values. So that means you are able to traverse relations in the Query. Let's just look at how does that work on the backend, on the server side.

The server gets this query. It has to parse it, of course, and there are libraries in all the popular languages, very well-maintained libraries. You parse this query and then you have what is called a resolver system. You, as a backend server developer, you don't have to deal with parsing query and figuring how to do all of these efficiently. What you do is you implement a resolver for

every part of your schema. You implement a resolver, knows how to get users. Another resolver knows how to get codes from the user. Then GraphQL serer implementation, resolver system, will take care of executing all of those resolvers in a batched fashion to execute the entire query and get all the data in the most performant way.

[00:12:04] JM: There's a part of setting up a GraphQL system that requires you to set up something called a GraphQL schema, or define a GraphQL schema. What is a GraphQL schema?

[00:12:18] SBS: This is super interesting. In the early days of GraphQL, the GraphQL schema was a little bit abstract. You create the GraphQL schema by essentially writing code. The GraphQL schema was not something you could point at and say, "That's my GraphQL schema." But then about a year ago, a little more maybe, there was an addition to the GraphQL specification called an SDL, it's the schema definition language. SDL is a very concise syntax for expressing that data model essentially. Think about something like UML diagram, but it's text and there's very little visual clutter.

It's a really nice way to represent what is your domain model? What does it look like? You write type open brace, then all the fields, the types of those fields. If the type of a field is not a scaler, it could be another type. You express relations this way.

What we found is that many companies find it really, really useful to use this schema definition language as a way to communicate between engineering and business people. You can sit down together and very quickly write out, "This is what I think [inaudible 00:13:28] look like." You can look at it and say, "Oh! That makes sense," or "actually, you're missing an intermediary over here. Let's try to move it around a little bit." There is a tooling that can take this schema definition and turn it into short graphs so you can look at all the interactions. There are tools like Prisma that can take an SDL and turn it into a real running GraphQL API backed by a database. This aspect, this standardized tooling makes it super easy to very rapidly iterate on your product and get real feedback while you're sitting there with the domain experts.

[00:14:06] JM: Does the schema definition, does that define relationships between different databases on your backend, or does it just define, is it a schema per database?

[00:14:21] SBS: This is where we need to really distinguish between GraphQL as a technology and Prisma as an open source product. GraphQL in itself has no relation to a databases or anything like that. It is completely agnostic to how the implementation looks like.

GraphQL is, one, the schema that defines relations between abstract entities without knowing what they are backed by on the backend, and it is the resolver system. Now, Prisma is a way to construct a GraphQL API on top of databases. In Prisma, you can say you have an existing SQL database, a PostgreS database, a Mongo database, then you can write SDL that maps this database to the GraphQL API. In Prisma, you create different schemas for individual databases.

Let's say you have a SQL database and a Mongo database together, then you would write two different schemas that would expose those databases as two different GraphQL APIs. In the future – And this is a thing I'm really excited about. In the future, we will make it possible to create relations in the SDL between different databases so that you can link directly between, let's say, a PostgreS database with mid-users and Elasticsearch database, for example, with indexed posts. This is a thing that people very often do when they complex web applications. They need to store data in different kinds of databases, because you need the different [inaudible 00:15:52] capabilities from different databases. You are maintaining syncing between all of these databases. Towards the end of the year, we will start working on features and [inaudible 00:16:00] to make this much more seamless and deal with all of that for you.

[SPONSOR MESSAGE]

[00:16:13] JM: OpenShift is a Kubernetes platform from Red Hat. OpenShift takes the Kubernetes container orchestration system and adds features that let you build software more quickly. OpenShift includes service discovery, CI/CD built-in monitoring and health management, and scalability. With OpenShift, you can avoid being locked into any of the particular large cloud providers. You can move your workloads easily between public and private cloud infrastructure as well as your own on-prim hardware.

OpenShift from Red Hat gives you Kubernetes without the complication. Security, log management, container networking, configuration management, you can focus on your application instead of complex Kubernetes issues.

OpenShift is open source technology built to enable everyone to launch their big ideas. Whether you're an engineer at a large enterprise, or a developer getting your startup off the ground, you can check out OpenShift from Red Hat by going to softwareengineeringdaily.com/redhat. That's softwareengineeringdaily.com/redhat.

I remember the earliest shows I did about Kubernetes and trying to understand its potential and what it was for, and I remember people saying that this is a platform for building platforms. So Kubernetes was not meant to be used from raw Kubernetes to have a platform as a service. It was meant as a lower level infrastructure piece to build platforms as a service on top of, which is why OpenShift came into manifestation.

So you could check it out by going to softwareengineeringdaily.com/redhat and find out about OpenShift.

[INTERVIEW CONTINUED]

[00:18:21] JM: It sounds like today, the primary use case of Prisma is to be some sort of presentation layer in front of – Maybe you could just re-explain what Prisma is. What kind of problem is Prisma trying to solve today?

[00:18:38] SBS: Yeah, absolutely. The way to think about Prisma is as a data layer for your application. Whenever you need to deal with database, is you need to store data. When we go and talk to what we call like rocket ship companies, like modern high-scale technology companies, like Google, Facebook, Twitter, companies of these kind, they all have built a thing internally that you could describe as a data layer.

The purpose of this data layer is to really abstract the database away from the application developer. When you're working with a database directly, you have a lot of power in your hand. You can do a lot of good stuff, but you could also do a lot of bad stuff, and it's actually really

complicated to deal with. So when you want to move really fast, you want to give capabilities to your application developers that is above what normal databases provide, you introduce a middle layer. That's what we call a data layer.

Most normal companies, small companies, or big enterprises, they don't have this kind of thing, because they kind of thought hiring a team of the best engineers in the world to just go and build the data layer is special tailored to your needs. What we're really trying to do with Prisma is to go to all of these big companies, the technology companies, the rocket ships, and learn from them and say, "What did you build into your data layer and why is that useful? Can we find common patterns and abstract that out into an open source product that we can make available to a much wider group of companies?" That's a fairly abstract way to talk about. That does make sense?

[00:20:09] JM: Yeah. Tell me if this is accurate. My understanding of what Prisma does is if I'm setting up GraphQL without Prisma, I need to define how a GraphQL query gets translated into a MySQL query, or a Mongo query, because the whole idea is that if I'm a frontend developer, all I have to learn is GraphQL, and then the GraphQL middleware infrastructure takes care of translating that GraphQL query into queries, a query for Mongo, or a query for MySQL, or potentially a query for both of those and a way to join those two database queries together. As an application developer, I just get back the data and I don't care about what kinds of query writing had to be done in order to get that data back.

With just normal GraphQL, I would have to write the translations of each of those GraphQL queries into their respective database translations, but Prisma is able to derive the schema of the databases and do translation from GraphQL to the database queries automatically. Is that right?

[00:21:30] SBS: That is right, but it's not the entire story. It's very important to realize that you're not supposed to query Prisma from, let's say, a React app or from a mobile app. You can do that and you can even set up permissions to take that same, but that's not how we envision [inaudible 00:21:46] Prisma and that's not how most of our customers are using Prisma.

What you do instead is you query Prisma from your own GraphQL server. You typically want to have a server that implements business logic or some kind of authorization mechanism. But then because Prisma exposes GraphQL and you're implementing a GraphQL server, you can take advantage of that direct mapping. I mentioned this earlier. It's called scheme delegation. It's an advanced feature that allowed you to basically expose a subset of your GraphQL schema by delegating to another GraphQL schema. This is a very advanced use case, and it's great when you need it.

But most people, even when implementing a GraphQL server, actually just uses Prisma. We might have used an ORM in the past to resolve data. Think of Prisma more as a layer in between your database. Maybe forget for a moment that it uses GraphQL as the wire protocol. But just think about it as a logical component that you'd talk to whenever you need to modify or access data.

[00:22:49] JM: So does Prisma itself need to figure out these database schemas, like the schemas of the underlying database? Does Prisma do introspection into the database just to figure out how to translate GraphQL queries?

[00:23:07] SBS: There are two ways you can work with Prisma. You can either work with Prisma from scratch. Let's say you're building a new application, you don't have any databases already. Then you can use what we call declarative migrations. You basically, instead of doing database migrations, you are writing out the GraphQL SDL, the schema definition language. This is super easy to read and write [inaudible 00:23:30] format that describes your data model. Then Prisma is able to make a diff between your previous data model and your data model and tell you, "The user type has changed. There are two new fields, and this field was renamed. Do you want to execute those changes?" You can use Prisma to migrate your underlying database.

If you have an existing database and you don't want to use the Prisma migration system, then you can let Prisma introspect your database, and Prisma will then generate a data model that fits that database. You're still in control and can remove some part of it. Let's say you have like a legacy, very complex Oracle database, and you don't want all of that exposed to application developers. You can do an initial introspection at the database and you can then go and remove

a bunch of fields or tables and just expose the data model that you want to the application developers.

[00:24:22] JM: Let's look at Prisma through another angle. There is a term that people have used for a long time called ORM, object relational mapper. What is an ORM and how do people typically setup their ORMs?

[00:24:40] SBS: This is a great question. So if you have dealt with databases since forever, right? If you use the relational database, you're typically interfacing with it through SQL. You would write SQL queries. What people found is that it's very tedious to write all of these SQL queries. You don't get a lot of help from your development environment, and there is what you call an impedance mismatch between your typical object-oriented language and the relationship algebra of the database.

What people came up with is the object relational mapper, the ORM, that basically translates your nice object-oriented world in your programming language, let's say, Ruby, to the world of relations in the database. So in Ruby on Rails, for example, you have [inaudible 00:25:26], and it's a fairly sophisticated system that allow you to deal with all your business entities as if they are objects in memory. You can add things to relations by pushing into an array. You just update fields and fills and then when you tell this object to save, or when let's say the request is over, you could configure that [inaudible 00:25:49] automatically [inaudible 00:25:50] to the database.

So this is a very nice model from a developer perspective, but it has some problems. When you use an ORM in a small application, it's great, but then as you grow, you run into all kinds of issues. It's very easy to end up with an architecture where you have these active ORM objects that can talk to the database and you're passing them around and all of a sudden you have calls to a database from all kinds of different places in the application logic. It gets difficult to reason out the performance implications of that.

What you very often see is, as applications grow, people try to push back the ORM to create a new logical layer in the application that you could a data access layer or something like that and make sure that the ORM updates never spread from there. When you get to that point, many people say, "Huh! Why do we need the ORM? Could we just write SQL? It'd be much simpler." I

think ORMs are really used a lot in simple applications, but as people get more and more sophisticated, they pull back from the ORM.

[00:26:54] JM: How does Prisma compare to the usage of an ORM?

[00:27:00] SBS: On a high level, it's fairly reasonable to think of the Prisma client as an ORM. It's the same basic functionality. You retrieve data and you store data. Because the Prisma client is modeled to GraphQL, you are – Think about when you load data and when you store data, and when you do load data, it's very simple for you to load all the data you need. You can traverse relations and make sure that you get all the data that is required for the computation you're about to perform in one go.

That avoids the typical problem of ORMs where you would either fetch too much data, or not fetch enough so that you need to go to the database in multiple rounds. The Prisma client makes it really easy for you to think through what it needs to do and basically layout your computation in the right order. First, load your data, do your computation, then store your data. From that perspective, Prisma really helps you enforce a sane architecture.

[00:27:56] JM: Why is it called Prisma?

[00:27:59] SBS: I think the initial idea is Prisma is a general word for prism, and what does a prism do? You have a beam of light coming into the prism and it spreads out so that you have this one GraphQL query coming in and Prisma spreads it out and sends it to the correct databases and then send these all back together again.

[00:28:21] JM: Now that we have revisited several different areas in Prisma in more detail and different areas of GraphQL in more detail, let's revisit one of the questions that I asked you earlier. So a GraphQL query without Prisma versus a GraphQL query with Prisma. Contrast those two things.

[00:28:43] SBS: Building GraphQL servers is actually pretty hard. I mentioned in the beginning that you would do batching of different resolvers to make sure that you resolve the query in the most effective way.

[00:28:56] JM: By the way, what's a resolver?

[00:28:58] SBS: A resolver is a piece of code that you create to return the data required for a subset of the query. So let's say you have a data model schema definition language model that contains users and posts. You would then write a resolver to get users and you would write another resolver to get posts. Let's say you have a query that gets a specific user in the first 10 posts. The most naïve implementation of this would first resolve the user, get the user, then go and get all the posts for these user. What happens then if for each of the post you also need to get comments? Then for the 10 posts, you might go out individually and do an additional request. This is bad for performance. You should not do this.

The way you deal with this, the way all GraphQL APIs deal with this is that they implement what is called a data loader pattern. So instead of going to the database immediately when you see that you need to request some new data, you put the data loading request on a queue. Then after a little while, you look in that queue and you say, "Oh! I need to get 10 comments for 10 different posts. Let's batch all of these together and go to the database with one request."

So normally when you implement a GraphQL API, you need to do all of these yourself. There are libraries to help you deal with it, but you need to do this correctly in your resolvers. If you use Prisma, you use the Prisma client to implement your resolvers, it already implements the data loader patterns. That means when you implement a resolver, you only need to think about how do I [inaudible 00:30:39] the correct data here and what kind of transformations do I want to do? You don't need to think about any of the infrastructure to make sure that your GraphQL API, your data access is fast. That is built into the Prisma client. That comes for free basically.

[00:30:54] JM: So in terms of translating the query into the target database, what is Prisma doing to help make that translation easier?

[00:31:07] SBS: So what Prisma does is it exposes this very rich cloud API as a GraphQL schema. So you can use a subset of these API in your resolvers, and that means you can basically expose just a subset of the API exposed by Prisma in your application GraphQL API.

So by giving you all of these building blocks that is a very direct match to the GraphQL API you want to implement, you can very often take shortcuts.

So remember, I talked about the impedance mismatch before, the mismatch between the relational model in the database and your [inaudible 00:31:47] model in your programing language. When you have GraphQL on both ends, you don't have the same kind of mismatch. The GraphQL resolves map very neatly to objects anyway, because the result of a GraphQL query, it's just a tree. It's a tree of entities. So it's very easy to represent that as objects in your programming language. It frees you from thinking about all of the stuff you need to think about when you have a relational database.

[00:32:14] JM: Such as what? What's an example of something that I would have to think about with using a relational database that I would not have to think about if I was using Prisma?

[00:32:23] SBS: So if you write raw SQL, then you have multiple different tables and you probably have relations between those tables. That's how you make them useful. So you need to now join those tables. What happens when you do a single join is that you have one table. You have a key in the other table. Let's say you have the user table and you have the comments table. Then in the comments table, you would probably have a user ID that creates a reference from comments back to users. To get a user and all of that user's comments, you now join those two tables.

When you think about that, it's like you create a giant spreadsheet with a lot of duplicated data. Many application developers probably don't think about this, but it's kind of weird. When you start doing relations across more than two tables, it becomes pretty messy. That's why you have this entire category of ORMs that try to deal with all of that. Try to do these joins, then look at all the data, remove all the duplicate data, get it back into a same object structure that you can reason about as a developer. If [inaudible 00:33:31] and you go directly to the database, it's pretty complex to reason about.

Now, contrast that with the world of GraphQL. The mental model that GraphQL that puts in you is exactly the mental model you care about. You want to specify a tree structure of a data you care about and you want to get that data back as a tree structure. So you get the one user, you

get the comments, the comments on array. From the comments, you can then traverse other things. Those other things would be objects or arrays. It's a very natural model compared to relational databases.

[00:34:05] JM: Prisma is compatible with some particular databases. So you have to write some kind of integration with each of the databases that it's compatible with, right? You have to write some compatibility layer between Prisma and MySQL, and with PostgreS, and with Mongo. What is that compatibility that you write with these integrations?

[00:34:29] SBS: That's correct. Prisma is open source. So you can have a look in the repository. There's a lot of code in there. One major push we have right now is implement connectors for more databases. You're correct, we currently have connectors for MySQL, for PostgreS and for Mongo. What does a connector do? A connector knows how to implement those resolvers that I talked about before and for a specific database.

Let's say you have a simple table called user, then there would be one resolver to go and get a particular user. There would be a resolver to go and get all users. There would be a resolver to get posts that are related from that user. So that's what the connectors do. They resolve data from a database.

What they also do is they migrate a database. Remember, I said we have the declarative migration system that basically produces a diff on your data model to see what has changed and then performs those migrations on the database in the most efficient way.

Of course, those connectors do all of these in a very efficient way. So all the data loader patterns, all the chunking and grouping happens in Prisma. What we are working on over the next four or five months is to add much, many, many more database connectors. An example I gave earlier was Elasticsearch, which I'm personally very excited about, because so many companies we go and speak to, they do this exact thing. They store most of the data in a relational database, but they need to have a storage data fast, sophisticated, full text search. How do you do that? Well, you copy all of the data to Elasticsearch and then you need to maintain this entire syncing system and now your engineers need to not just deal with relational databases, but also deal with Elasticsearch. You add an entire new variable to the mix.

What is so great about mapping all of these to one paradigm to GraphQL is that not just is the mental model better. It's the same mental model that you can apply to all the databases. So in the future when we have storage databases, we have relational databases, we have graph databases, you as an application developer can switch between all of them without having to learn a new drive or learn a new mental model and about the quirks of hosting a particular database. You can just go and use the GraphQL playground and you can explore the API that those databases provide in a very familiar, very easy to explore way with auto completion just by writing GraphQL of course.

[SPONSOR MESSAGE]

[00:37:12] JM: Your audience is most likely global. Your customers are everywhere. They're in different countries speaking different languages. For your product or service to reach these new markets, you'll need a reliable solution to localize your digital content quickly. Transifex is a SaaS based localization and translation platform that easily integrates with your Agile development process.

Your software, your websites, your games, apps, video subtitles and more can all be translated with Transifex. You can use Transifex with in-house translation teams, language service providers. You can even crowd source your translations. If you're a developer who is ready to reach a global audience, check out Transifex. You can visit transifex.com/sedaily and sign up for a free 15-day trial.

With Transifex, source content and translations are automatically synced to a global content repository that's accessible at any time. Translators work on live content within the development cycle, eliminating the need for freezes or batched translations. Whether you are translating a website, a game, a mobile app or even video subtitles, Transifex gives developers the powerful tools needed to manage the software localization process.

Sign up for a free 15 day trial and support Software Engineering Daily by going to transifex.com/sedaily. That's transifex.com/sedaily.

[INTERVIEW CONTINUED]

[00:39:01] JM: I can imagine in a multi database infrastructure, you might have several different databases that could potentially satisfy your query, and if GraphQL and Prisma are taking care of translating your query into each of those different databases, you could have the GraphQL Prisma infrastructure take care of figuring out which database is actually the best fit for a query.

[00:39:36] SBS: Now it gets really exciting, right? Because Prisma is a stateful component that sits between the application and your databases. It knows about everything that is going on. It sees all the queries. It sees all the traffic. It can keep statics overtime and see how those particular queries perform. It can look at the amount of data. It can look at indexes in a database and say, "I'm recognizing some queries that are slower than they should be. Maybe we could suggest to the application developers to apply particular indexes, or maybe we could suggest to the application developer that they introduce a new kind of database. Use Prisma to pre-aggregate result into this database and perform these queries to that database instead."

If you go and talk to many of these last scale intern companies, you'll see that they often have something similar to CQRS. CQRS, the idea is that you separate your read model from your write model. You basically have two completely different data paths. One is to write [inaudible 00:40:38] to a transaction or a system to make sure everything is nice and consistent. But then you have the read part. The purpose of spreading up those two paths is that you recognize that most of your traffic often like are factor of a hundred or a thousand. Most of your traffic is read traffic, not write traffic. It makes sense to optimize those two paths separately.

The system you often come up with is you have the relational database on the right path, and then you pre-aggregate views of this data including summations and joins and all these stuff, and you'd take that complex result and you store it in maybe a document database, like Mongo, where you can basically store all the data you need to serve a particular query.

What we found is when we go and talk to all of these companies that do these, it's massively complex, and there are no good systems to deal with it. They all build their own complex systems internally that they are now the only people in the world who know how to operate. Then they hire new people, they need to train them from the beginning. Really, they're all quite

different, but they're doing the same thing. There should be a standardized way to deal with all of these.

Down the line, when Prisma has support for many more databases, when we start seeing adoption, it makes sense for us to really try to make this [inaudible 00:41:56] much more accessible to not just big technology companies who can build their own infrastructure, but to all applications developers. This is a tool that should be available to everybody building internet applications.

[00:42:09] JM: So in terms of the checkboxes that we've talked about Prisma potentially satisfying. We've got simplifying the queries for application developers that are writing services. We've got allowing joins between different – Simplifying and allowing joins between different databases. We've got query recommendations. We've got potentially caching. You didn't really explicitly mentioned caching, but caching could – I guess you did kind of, because in the suggestion, like Prisma says, "Hey, this query would run a lot faster if you inserted this Redis and then you started caching your stuff, and we can do all that for you. Would you like us to set that up for you? Just click this button," and you do that, and that would be a really cool, smart, futuristic data layer.

The thing that we haven't really touched on is data science, data infrastructure, machine learning. These things are also tied to the data problems that companies have. We've had shows with companies like Dremio and other companies that are tackling this data infrastructure, or an Uber. A recent show that I did that has really stuck in my mind was about Uber's data infrastructure and their ETL pipeline and they're getting their transactional data, like my rides that are going on right now. Translating that, pulling that data into a data lake and then pulling data from that data lake into Elasticsearch, or pulling in into Presto and making it available to data scientists and data engineers that are building on top of Uber's data platform, because there's so many people at Uber. They need this big data platform to be able to not only write microservices and have, like you said, that ability to have a really fast and responsive read pattern, but the ability to do ad hoc machine learning jobs, and analytics, and stuff related to billing, and there's so much that goes on on top of the data platform that is not directly related to the sort of microservices applications that we're talking about. So how are you evaluating the

opportunities of the data platform of the machine learning and the data science suite of applications?

[00:44:35] SBS: I like that term data platform. First of all, it's important that you're not building a database. We're definitely not building something that would sit directly on a desk and know how to store data. We layer on top of existing databases. That's super important to keep in mind. The other thing is we are definitely not building something for Uber to consume. They're too big. They have very specific needs and they can hire teams to work on something, and it makes sense for them to hire teams that work on something that is very specific to their needs.

What we're rather doing is we are looking at the patterns that emerge at big companies, like Uber, and we try to productize it. We build an open source product around that, that may be less sophisticated is not the right way to say it, but more normal software teams can use to build out their infrastructure.

I think data platform is interesting, because you don't really want to just be dealing with databases. You don't actually care that much about databases. You care about your data and you care about that data being available to your application engineers, your data scientists, whoever are in your organization and should have access to this data.

Next to Prisma, the open source product, we are also working on related software we call Prisma Cloud, and Prisma Enterprise. So those are basically a platform on top of the open source Prisma. What this platform gives you is a way to manage access to all of your databases, to manage access rise who can change data structures. You have a history of what the different employees are doing. This is also where we will introduce many more aspects in the future [inaudible 00:46:23], like analytics, the kind of suggestions we talked about before.

I'm quite curious though why you create data platform to machine learning and these kinds of things. I obviously haven't listened to that front [inaudible 00:46:37]. Maybe you have something particular [inaudible 00:46:40].

[00:46:41] JM: Well, if you have to train a model and you want to train that model on the last four years of rides at Uber, you need a way to get all that data from your data lake into your training system and so –

[00:47:02] SBS: Right. Okay. I get what you're getting at. If you have transactional applications, like a web server, and it can do a lot of requests, reads and writes a lot of data. What is important for a web server is low latency. Not so much [inaudible 00:47:16]. When you're dealing with tons and tons of historic data, because you want to do machine learning, you want to train your models, then it's the exact opposite. Latency is not really that important, but throughput is important, because you have just terabytes and terabytes of data to push through.

[00:47:32] JM: Also, this is a false dichotomy, because in the future you're going to want models that update really quickly and models that update really quickly even based on large influxes of recent data. So if you have 5,000 cars on the road, 5,000 self-driving cars on the road, they're constantly pulling in lots of data and you want the models that are being built off of those data to be updated really quickly. Then you actually have high volumes of data where you want those to be piped into different services really quickly. So it's like a false dichotomy that exists today, but probably will get changed in the future.

[00:48:08] SBS: Okay. Yeah, that makes sense. So one thing we're definitely seeing when we talk to bigger companies is that they – New systems, they don't really think about batch processing anymore. You think about stream processing. So that means all the data that is coming in, all the changes is made available on a system like Kafka, something like that. Maybe you use some other modern stream processing systems [inaudible 00:48:31] update your mostly machine learning models.

The two aspects of this right there is the consumer. You as a data science is to have to train your machine learning models. Do you even support a streaming data, or do you need to update those models in a batched fashion? If you're very sophisticated, you will build models that can be updated incrementally and you can provide these updated models to users much faster.

The other side of it is does your data infrastructure, does it support streaming all of these data, or does it only support a batched fashion? That's probably the aspect that is most interesting for

us. We care about streaming data. We care about synchronizing data between different kind of data systems. Right now we care most about databases. We care about streaming data from relational database to an Elasticsearch database. But I could definitely see us in the future caring about streaming to other kinds of data systems, such as a data lake, or probably more directly to whatever technology you're using to update your machine learning models. I think that that's going to be supper interesting.

[00:49:36] JM: Yeah. I know we're going to wrap up, but related to what you just said, this is something I didn't really touched on. Well, we didn't touched on real-time. That would have been interesting, because I know that's a challenge, like real-time syncing between different data systems is not straightforward. Also, the element of if you have this common layer of GraphQL between your different databases, it unlocks the ability to do pretty straightforward ETL between those databases.

If you have a common schema between your SQL database and your Mongo database and your Elasticsearch cluster with GraphQL and Prisma, then it's easier to mirror data, or import data from different system. Is that an accurate way of putting it?

[00:50:30] SBS: This is one of the biggest challenges in enterprises, that you have all of these disparate systems that all have data that is kind of similar, but the shape doesn't match. Let's say you have 10 different systems that all know about an order, but they all contain a subset of an order. They might not even have the keys that allow you to kind of match them altogether. That is exactly – There's an entire category of big complex enterprise software and expensive consultants who help model out your data structure.

I think GraphQL and the schema model has a role to play there, because it does exactly what you do. You map out your domain model, and when you have that, you can start mapping between them and linking all of these up. I'm really excited about that part.

[00:51:20] JM: Okay, Søren. Well, it's been really fun talking to you. We covered a lot of ground and opened a lot of doors that hopefully we can re-explore in the future. I'm excited about what you guys are doing with Prisma. So nice work and we'll talk soon.

[00:51:35] SBS: Thanks a lot. It was awesome to be on your show.

[END OF INTERVIEW]

[00:51:40] JM: Cloud computing can get expensive. If you're spending too much money on your cloud infrastructure, check out DoIt International. DoIt International helps startups optimize the cost of their workloads across Google Cloud and AWS so that the startups can spend more time building their new software and less time reducing their cost.

DoIt international helps clients optimize their costs, and if your cloud bill is over $10,000 per month, you can get a free cost optimization assessment by going to D-O-I-T-I-N-T-L.com/sedaily. That's a D-O-I-T-I-N-T-L.com/sedaily. This assessment will show you how you can save money on your cloud, and DoIt International is offering it to our listeners for free. They normally charge $5,000 for this assessment, but DoIt International is offering it free to listeners of the show with more than $10,000 in monthly spend. If you don't know whether or not you're spending $10,000, if your company is that big, there's a good chance you're spending $10,000. So maybe go ask somebody else in the finance department.

DoIt International is a company that's made up of experts in cloud engineering and optimization. They can help you run your infrastructure more efficiently by helping you use commitments, spot instances, rightsizing and unique purchasing techniques. This to me sounds extremely domain-specific. So it makes sense to me from that perspective to hire a team of people who can help you figure out how to implement these techniques.

DoIt International can help you write more efficient code. They can help you build more efficient infrastructure. They also have their own custom software that they've written, which is a complete cost optimization platform for Google cloud, and that's available at reoptimize.io as a free service if you want check out what DoIt International is capable of building.

DoIt International art experts in cloud cost optimization, and if you're spending more than $10,000, you can get a free assessment by going to D-O-I-T-I-N-T-L.com/sedaily and see how much money you can save on your cloud deployment.

[END]