EPISODE 348

[INTRODUCTION]

[0:00:00.7] JM: Relay is a JavaScript framework for building data-driven React applications. Facebook open-sourced Relay around the same time the open-sourced GraphQL and Facebook expected Relay to be the more popular of the two projects. However, the reality was reversed. Open-source companies like Meteor began to build GraphQL tools and a few businesses were started around GraphQL.

One year later, the excitement for GraphQL had completely surpassed the excitement for Relay, which had aged poorly in a newborn ecosystem of GraphQL tooling. At the same time, Facebook was also starting to integrate Relay into their React Native apps, but Relay was performing poorly on the low-end android devices. This led the Relay team to the conclusion that they needed to rewrite Relay. This was both to better fit the growing GraphQL ecosystem and to be built with performance in low-end React Native environments at the top of mind.

Relay Modern is the new version of Relay. It was released to the open-source community at this year's F8 Developer Conference for Facebook, and in this episode Caleb Meredith is joined by Lee Byron, the co-creator of GraphicQL, and Joe Savona, a founding member of the Relay Team and they discuss Relay Modern. This discussion includes a conversation about the commercial GraphQL ecosystem, the story of why Facebook decided Relay needed to be rewritten, and a look at the future of UI development from some trends seen in Relay Modern.

[SPONSOR MESSAGE]

[0:01:50.7] JM: For more than 30 years, DNS has been one of the fundamental protocols of the internet. Yet, despite its accepted importance, it has never quite gotten the due that it deserves. Today's dynamic applications, hybrid clouds and volatile internet, demand that you rethink the strategic value and importance of your DNS choices.

Oracle Dyn provides DNS that is as dynamic and intelligent as your applications. Dyn DNS gets your users to the right cloud service, the right CDN, or the right datacenter using intelligent response to steer traffic based on business policies as well as real time internet conditions, like the security and the performance of the network path.

Dyn maps all internet pathways every 24 seconds via more than 500 million traceroutes. This is the equivalent of seven light years of distance, or 1.7 billion times around the circumference of the earth. With over 10 years of experience supporting the likes of Netflix, Twitter, Zappos, Etsy, and Salesforce, Dyn can scale to meet the demand of the largest web applications.

Get started with a free 30-day trial for your application by going to dyn.com/sedaily. After the free trial, Dyn's developer plans start at just $7 a month for world-class DNS. Rethink DNS, go to dyn.com/sedaily to learn more and get your free trial of Dyn DNS.

[INTERVIEW]

[0:03:49.5] CM: I'm here with Lee Byron and Joe Savona who work on the GraphQL JavaScript team at Facebook. The team behind the release and development of Relay Modern. Lee and Joe, welcome to Software Engineering Dailyi.

[0:04:01.8] JS: Thanks for having us.

[0:04:02.7] LB: Thanks for having us.

[0:04:04.8] CM: Let's start with this; just about all users of GraphQL are vocal advocates for the technology, but it is hard to articulate the benefits of GrophQL to someone unfamiliar with the problems that GraphQL solves unlike frontend engineers who have been the most aggressive adopters. Lee, as the co-creator GraphQL, how do you explain why GraphQL is useful to software engineer who is unfamiliar with GraphQL or the problems that it solves?

[0:04:30.7] LB: I usually try to understand where they're coming from. If they are a front and engineer I might speak to the benefits like managing the network connection much better. Typically, when we have either a Rest API or individual API endpoints, you'll need to do multiple rounds of network requests in order to get all the data that you need to have your app work, and especially as we’re building more mobile-centered applications where the network might be slow or spotty, that can be a serious impact on the performance of your app.

If someone is a backend engineer, I often talk about the organizing characteristics that GraphQL can have for your backend code and the different ways that it can help manage and understand the incoming traffic to your website. Often, what happens in a Rest endpoint or custom endpoint style API is your getting hits to these endpoints but you're not a hundred percent sure which fields in those endpoints are being used or not used. Any kind of change that you want to make to that API overtime can be really difficult without actually going and finding the code, the client code that’s actually calling your API and getting to dig into it. If you’re in a public endpoint, that's near impossible, or if you have your very old clients that are still in use, that also may be really difficult to understand which versions of your very old apps are actually doing these things.

With GraphQL, because an individual query specifies exactly what it needs and nothing more, you can evaluate all incoming queries to understand exactly what your backend needs to provide and what it doesn't which makes updating that API overtime much much easier. For example, at Facebook, we've been running the same version of our GraphQL server for five years now without any breaking changes to the API itself, which is pretty unheard-of within the API space.

[0:06:30.2] CM: How do you respond to the common backend developer criticism that GraphQL is very hard to cache on the server side?

[0:06:38.3] LB: It’s very true. It depends on what layer you want to put your caching. For us at Facebook, every request to our API is going to return new data, just because things are changing all the time. We don't put our cache at the network layer, instead we our cache at the data access layers under the hood.

Instead, what we went optimize is every time we’re going to actually go to a database to load data, we want that to be as fast as possible, and that's where we put our caching layer, and that works very very well with GraphQL. When we need to say, “Run a SQL query, or talk to some backend service, we’re almost always going to go to a mem cache style look-aside cash before we actually do that. Often times, that cache data is actually in memory on the very box that you're hitting.

You can still build very well cached services, but using GraphQL does preclude the ability to have cache layers at the network layer since you're going to have a different incoming query to that endpoint every time. That's also the same for most Rest endpoints that offer the ability to sub-select on fields or have some of the characteristics that GraphQL have. Ae as soon as you start introduce those features, you also lose the ability to have network layer caching.

[0:07:53.1] CM: GraphQL lets the client decide what to request with incredible granularity. Joe, how does this change API development?

[0:08:01.4] JS: I think it's getting to a lot of the things that Lee talked about where in traditional client-side development, every time you want to access some new information in a view somewhere, even adding a simple single scaler new scaler field in a reusable UI component, you would have to go and find every possible view, every screen that could possibly include that component, figure out all the endpoints that those screens might fetch data from and make sure those endpoints are now returning that field.

Similarly, if you say, "Actually, we don't need some information in this view anymore. Let's go try to clean this up to not have overhead," you often would end up giving up because you aren't sure that it's safe to remove that data access on your customer endpoints. GraphQL really allows you to make changes to your UI with much more confidence because you're just changing the particular GraphQL query that you're using to fetch your data and you don't have to worry so much about the kind of cascading effects it has on the rest of the system.

[0:09:02.6] CM: This is a question for both of you, or either view. In the span of about two years, the release of the GraphQL specification has created an ecosystem of commercial tools aiming to serve the GraphQL market. How are commercial tools like those provide by Apollo, Graphcool, or Scaffold useful to GraphQL users?

[0:09:23.9] LB: I think they're incredibly useful just because having those options for tools, most obviously, you can just go use those thing. For example, Graphcool, gives you an entire backend as a service and manages everything for you. If you are a new company trying to get up off the ground or you want to build a side project, it's now extremely easy to go from nothingness to a full backend that speaks fluent GraphQL as an API service.

Then once you're getting going and then you encounter more of these tools like Apollo's Optics which lets you use analysis on all the incoming GraphQL queries to understand better where the performance bottlenecks in your application are and what kind of axis load that you have on your GraphQL server. That’s exactly the kind of thing that API designers would dream of having on their backend services even in moderate size companies.

That’s a really difficult thing to build and it's pretty awesome that you can take advantage of some of the core principles of GraphQL to make building that kind of thing really useful. Even if you don't end up using the tools from these companies, what really excites me is that these companies are illustrating all of the different kinds of things that can be done with the GraphQL server. Even if you're not to go use those things, you can kind of see this the shape of things that can be built and maybe even try to build similar versions of those things yourself within your own company if you end up using GraphQL.

[0:10:52.9] CM: Yeah. All right. Two more kind of quick speculative questions and then we’ll dive in in talking about Relay. How big, if you could speculate, is the market for commercial GraphQL services today?

[0:11:04.3] LB: I have no idea, but if we can maybe judge by the size of audiences that we see at the GraphQL conferences which seem to get bigger every year. Last year we had almost 500 people at the GraphQL Summit Conference in San Francisco, that was hosted by the Meteor Team. This year, we’re going to have a similar conference that should be even bigger. We’ve now extended that to a GraphQL Europe Conference, that will be in Berlin this year. There are GraphQL meet-ups all over the world many of which have more than 100 attendees every time they meet.

I'm seeing the spread in terms of a groundswell effort or impacts from people all over the world and then I personally meet with a lot of medium to large size companies as they’re starting to adopt GraphQL just so I can impart some lessons that we’ve learned at Facebook, and that includes all kinds of companies within the Valley as well. I think it's still relatively early. GraphQL is young technology, but it's been pretty inspiring to me to see such an aggressive uptick in usage.

[0:12:09.4] CM: Yup, it is definitely on. If we look five years or more into the future, how big do you think that the market can be for commercial GraphQL services?

[0:12:17.7] LB: I think we can look at similar style of services that are already out there. I think maybe at a most optimistic angle we might look at SQL and ask yourself how many people are using SQL and one of its flavors to build something. I don't see that — The same that SQL ended up helping us have a common language for talking about relational databases, I see GraphQL as potentially being that common language to talk about networked API data. It could be as big as SQL perhaps, maybe not in five years, but over the long term.

[0:12:52.9] CM: Given that there are a couple of startups whose success depends on the success of GraphQL, how political is process for adding new features to the GraphQL specification today?

[0:13:05.0] LB: Hopefully, not much at all, although that is actually one of the topics that's pretty close to some of the things I'm working on now. What we really want from GraphQL is a stable base. I'll go back to the SQL analogy. When was the last time you were just like counting down the weeks until a new feature got added into the SQL specification? We've been using the same base specification for SQL for decades because SQL ended up providing the right base pieces to let us do those things. Now, every variant of SQL has their own minor additional flavor to that, but the core of it is all very much the same.

I see us going through this kind of post-initial release of GraphQL where we're identifying all the things that we knew would be the things we would want to tackle next and we're now starting to identify and evaluate those and hopefully we'll fold in solutions into specified GraphQL so that everyone using it can agree on a similar thing.

What I hope is not to see like an acceleration of new changes getting out of the GraphQL as more people start using it and more companies that rely on it start jumping in. Instead, to focus our energy on coming up at the core set of abstractions that can be composed together to create lots and lots of variation. That way, we can arrive at a stable base that people can assume will be stable for the long term rather than something that constantly changes under their feet.

[0:14:37.0] CM: That sounds like you have to say know a lot. How happy does not make some of Facebook's partners and growing GraphQL?

[0:14:44.5] LB: It actually hasn't been a problem yet because the vast majority of time, we end up in one of two buckets. It's either, “Oh, that's a really exciting idea and it's along the same lines as things that we've been thinking about as well and let's get together and figure out what the right solution to add to this should be,” or its, “Yes, let's figure out what this change might do to GraphQL holistically. Look at bumps into lots of really gnarly edge cases. Maybe we should rethink this.” In fact, actually, you can do everything we proposed here with what we already have a GraphQL via something that's may be slightly less and tactically brilliant, but just as expressive. I'd much rather end up with something that has a large amount of expressivity than something that has your beautiful syntax for every possible edge case.

There actually haven't been any instances where there’s been some critical feature that has to get in or else some company will totally fold where we have disagreed within the collective of people who work on the GraphQL spec. That hasn't happened yet. Hopefully, it won't. If it does, I'm pretty convinced that we’ve built the right set of people across many companies that have a say in how the spec evolves, that we’ll be able to manage that pretty well.

[SPONSOR MESSAGE]

[0:16:06.7] JM: For years, when I started building a new app, I would use MongoDB. Now, I use MongoDB Atlas. MongoDB Atlas is the easiest way to use MongoDB in the cloud. It’s never been easier to hit the ground running. MongoDB Atlas is the only database as a service from the engineers who built MongoDB. The dashboard is simple and intuitive, but it provides all the functionality that you need.

The customer service is staffed by people who can respond to your technical questions about Mongo. With continuous back-up, VPC peering, monitoring, and security features, MongoDB Atlas gives you everything you need from MongoDB in an easy-to-use service. You could forget about needing to patch your Mongo instances and keep it up-to-date, because Atlas automatically updates its version.

Check you mongodb.com/sedaily to get started with MongoDB Atlas and get $10 credit for free. Even if you're already running MongoDB in the cloud, Atlas makes migrating your deployment from another cloud service provider trivial with its live import feature. Get started with a free three-node replica set, no credit card is required. As an inclusive offer for Software Engineering Daily listeners, use code "sedaily" for $10 credit when you're ready to scale up. Go to mongodb.com/sedaily to check it out.

Thanks to MongoDB for being a repeat sponsor of Software Engineering Daily. It means a whole lot to us.

[INTERVIEW CONTINUED]

[0:18:12.1] CM: Yeah. I guess this is a good transition to our conversation about Relay, because what happens when you want to add a feature to the GraphQL specification for the Relay framework specifically. What's the process there? Joe, you can pitch into it if you want.

[0:18:26.6] JS: Sure. Yeah. This has come up like a couple of times during the development of Relay and Relay Modern, and it really actually goes through the process that Lee just described where oftentimes we encounter a use case. A concrete examples is object identification. How are we going to uniquely identify objects within the cache and how much control do we want to give the schema developer versus flexibility and control versus performance, for example. If we can assume a particular field name, things might be faster.

Really, those type of things go through the process that Lee just described. Myself and others in the Relay team kind of met with Lee and Dan. This is back a year ago for this particular example, and talked about what are the different approaches that we can do here, and that's something that a lot of people are asking for. Hopefully, we're moving forward with that.

There’s other things that we’ve come up with where initially it seems like, “Oh, I’d be really great if we could add some feature to this spec.” Then upon digging into it, we realize actually it has these kind of complications that really aren’t as simple as we might've first thought. It's really actually in the end ends up not being a good idea and we find a way around it. I really think Lee did a great job describing it. I think, so far, really, it’s fallen into those two categories.

[0:19:38.1] CM: Now, I want to walk down the history of Relay, from the first release or even prerelease to the point where you decided you might need to rewrite it and on. Through that discussion we can talk about important features in both Relay Classic and Relay Modern. I guess before we get started, there are two versions of Relay we’re going to be talking about; Relay Classic, which was the first version of Relay that Facebook released alongside GraphQL, and then Relay Modern, which is a rewrite of Relay which is more recently released and which we want to spend the most time talking about because that's the most new, shiny, and interesting technology today, if you will.

Okay. What was the original vision behind Relay when you first released it?

[0:20:21.3] JS: Yeah. Just for a bit of context, I’ve been working on Relay for about 2-1/2 years now, basically since I joined Facebook. Obviously, Lee has been part of GraphQL since the beginning which kind of predates that.

Relay actually started out — Relay is not the original name of the framework. We had an internal name. The original idea was really actually around routing and it was basically an attempt to build a framework for URL routing where tackling the question of; how do we deal with nested routes and fetching the — Given that you’re transitioning from one route to another, what is the sort of code you need to download in order to be able to render the next route?

As part of that, we realized, “Well, you also need the data for that next route too,” and so GraphQL was a great way to say, “Yeah, actually express the dependencies and fetch that data efficiently from the server,” and so that's why GraphQL was added into this framework. Overtime, the GraphQL portion became kind of the main aspect.

Relay Classic, we open-sourced it about a year and a half ago, I guess. I guess of 2015. Relay Classic was kind of at a point where we've largely moved away from the routing emphasis, but that legacy routing aspect was still there and something that developers of Relay had to contend with. That's kind of Relay Classic. Also, the other thing too there is that Relay Classic was designed based on the original version of the internal version of GraphQL.

As part of open-sourcing Relay, we're open-sourcing GraphQL at the same time as we're open sourcing Relay. We kind of got Relay adopted to open-source GraphQL, but we hadn't really fully adopted it. You can see that with things like having to have multiple queries, one for root field, where the open-source specification allows you have multiple root fields in a single query. Relay had some kind of work around some limitations there.

Relay modern was basically — That's the whole show. I don't know if you have any follow up questions about GraphQL before we dive into that.

[0:22:23.0] CM: Yeah, a couple of questions. Why was GraphQL released around the same time as Relay? Because GraphQL is like a dependency of Relay, so it might make sense to release GraphQL first and then Relay. Why were they released around the same time?

[0:22:37.9] LB: I can answer that a little bit.

[0:22:39.4] CM: Yeah, go ahead.

[0:22:40.3] LB: Originally, we weren't planning on open-sourcing either of these projects, and GraphQL has been around at Facebook for over five years now and the first few years that we are working on it, we thought about it but we didn't have nearly as much confidence that it would be important beyond the walls of Facebook then than we are now, of course. The first time that we talked about this stuff publicly was at the very first React Conference, which was February of 2015. Jing Chen and Dan Schafer gave a talk about Relay and GraphQL not to announce that we would be open-sourcing anything but just to talk about how we were using React inside Facebook that we thought was interesting.

We got such an overwhelming response to that talk where people were saying, “This is amazing. When are you going to open-source this technology?” We’d be like, “We were just telling you what we were working on. We weren't making any plans to open-source anything.” The team was so excited about the response that we were getting that we immediately started trying to figure out what it would mean to actually open source both of these technologies.

That was kind of the spark for a desire to open-source for both of them. It was because we ended up talking about them in the context of one presentation. Really, at that point, they kind of went their own ways to try to figure out what open-sourcing would mean.

The process for GraphQL finished first which is why we open-sourced first, but we also we’re talking to each other and we knew that, obviously, open-sourcing Relay without open-sourcing GraphQL would make no sense because it is a dependency of how that thing works, and so it just ended up kind of working out such that GraphQL was open-sourced I think like three or four months before Relay was open-sourced.

To give people a little bit of a heads up to start learning about this new technology, trying to figure out how it fits into the things that they were building, and then later Relay was released, then people already had a stable base on top of which they could introduce Relay. That was really — Relay and GraphQL don't have a common story within Facebook. As Joe is mentioning, really started as a completely separate project from GraphQL internally to solve a totally different problem, and only later as we’re building these things together internally did we realize that by using GraphQL we could make Relay even better, and then that was before we talked about it publicly and then that led us to the whole open-source journey.

[0:25:05.2] CM: Since GraphQL and Relay, the spark for that release was the React development community. Did you expect such a strong response to GraphQL, or were you expecting a stronger response and reaction to Relay?

[0:25:19.5] LB: We were totally expecting a stronger response to Relay. We got a very strong response to both. You can still see remnants of the very first commit to the public GraphQL repo which was this hilariously awesome ASCII character that says, “Give Relay,” that’s because people were like tweeting at us and sending us messages with that think. I don't know how it became a meme, but there is like palpable excitement about Relay.

We were really thinking about — We were excited to open-source GraphQL, but we were kind of thinking about it as like, "Well, the most important thing is that we get this thing out before Relay so that people can use Relay, because that's what they're really excited about.

I think it was really — At least to me, it was surprising to see how much excitement and uptick we got around GraphQL in isolation outside of Relay. In fact, we he had to kind of rethink how we are talking about these projects because we had spent so much time talking about them in the same breath that people would get confused and say, "I would love to use GraphQL but I have an iOS app, or my backend is written in Ruby." Like something that we're like, "Well, that's fine. You can still use it." They're like, "Oh, yeah, but isn't it a JavaScript framework?" We're like, "No. No. No. No. No."

We then had the kind of backup and reintroduce GraphQL as we would talk about it at conferences and meet- ups as a standalone technology and spent a lot more time focusing on the spec and the interface of the network. That part was super surprising. I don't think we expected that.

[0:26:47.9] CM: Yeah, looking back in hindsight, it's definitely an interesting story, but given that you've now rewritten Relay to Relay Modern, was it a mistake to open-source Relay Classic?

[0:26:59.6] JS: I don't think so. I think one of the things we've tried to do, and Lee can add to this, but I think we've tried to — part of the idea of even talking about these things at the first React Conference was, "Hey, these are the things that we're working on. These are the ideas that we've — The things that we've explored and the ideas that we found to be useful," and wanting to share those ideas and concepts with the community. Sharing code — We like to share a code when we can, and we share ideas otherwise in order to let other developers benefit from what we've learned.

I think that getting Relay out there in its first form was — People used it. It was valuable to them, and we got great feedback from the community. It was valuable to them. We got great feedback about the framework and how it could be better which kind of folded into Relay Modern. Overall, I think it was — I'd say it was a success to open-source it.

[0:27:55.1] CM: Relay Classic has been described as highly dynamic. Whereas one of the appeals of Relay Modern is that it is static. What do these terms mean in the context of a GraphQL client?

[0:28:08.2] JS: Yeah. By dynamic in Relay Classic, what we're really talking about is the way that queries are constructed. How and when that step occurs? Stepping back a little bit, as we were using Relay Classic internally, we started using it in more and more — It's originally built with a focus on web. As we began to develop React Native apps, we started using — Relay kind of became the de facto way to do data fetching in React Native apps which then once we had React Native android, we started using it on android and using Relay on lower end devices.

We started looking out what is taking up time in a Relay application and what parts of Relay can we optimize. One of the things that we saw, we've dug into systraces looking at profiles and where time was spent, and it turns out that a lot of the time we spent basically constructing queries, that's basically building a large in-memory object AST just to describe a query. Time spent traversing over that AST to convert it back to a GraphQL string to upload to the server. Then traversing over that AST in order to do things like write results from the server into the store and read results from a store back out and give them to views, basically constructing and walking over trees was a lot of the time spent in a nutshell.

In Relay Classic, that's happening at runtime because the queries themselves are dynamic. You write basically what looks like an ESX template string and it supports interpolation so you can — Basically, at runtime, we're constructing this AST that has values that you might've decided at runtime and just to directly injecting them into the query structure. This is very expressive. It allows us to kind of simulate the appearance of fragments having locally scoped variables which is kind of convenient in some cases and it goes back to, again, the React oriented roots from Relay.

Again, doing that just takes up a lot of work at runtime. Once we saw this, we said, “Okay. One of the biggest levers that we have for improving performance is to attempt to remove a lot of these work constructing and traversing queries.”

We looked at how the native application at Facebook are built and they, for a long time – They actually also started out doing this runtime query construction and later switched to statically constructing a query. That means static text with no interpolation of anything. It’s just straight up GraphQL text, and at build time, you traverse the entire application, find all the fragments and queries, you construct individual queries with all the fragments they need.

You send each of those queries to the server, save it, bet back and ID. Then at runtime, you use that ID to actually fetch the data for that query. This is kind of the core idea of Relay Modern was basically rebuilding a system around persistent queries. That’s kind of step one, and then we said, “If we’re going to do that, then that actually opens up the door to all these other optimizations that we can do to,” and so we explored a lot of those things.

[0:31:03.5] CM: Do you have any, I guess, off the hand numbers for what moving to a more static system. Moving from runtime to compile time for GraphQL query generation, how much time did that save you?

[0:31:15.0] LB: Yeah. We talked about this in the blog post. The marketplace tab in the Facebook application is built using React Native and Relay and we converted it incrementally from Relay Classic to Relay Modern using the compatibility layer. When we switched it to — Basically, compatibility mode is we’re using the newer APIs where you’re actually running the Relay Classic runtime under the hood. You’re actually not getting all these performance benefits.

When we actually finally made the switch and turned on Relay Modern runtime and use static persistent queries, we saw 900 millisecond drop on the average TTI on the android side. That’s example of where we see this having an effect on products.

[0:31:57.7] LB: That wasn't just turning on the core. Turning the core was — It did make it much faster in the hundreds of milliseconds, but a lot of that 900 milliseconds was how static queries enabled us to do certain kinds of optimizations that would just be impossible if we didn't know the queries until runtime. For example, with the marketplace tab, we know the GraphQL query associated with that view.

It takes a little bit of time to boot up the React Native environment, parse and run all the JavaScript and then get something rendered on the screen. Only at that point can Relay Code exist and start executing and do its job. We had to wait those couple of hundred milliseconds to actually send a request over the network. Since we can know those queries ahead of time, we can actually send the network, or send the query over the network the instant that you tap that tab and then the whole handful of hundreds of milliseconds that we're actually initializing the UI were busy on the network as well.

That allows us to do this kind of parallelization trick, and that's just one of many different kinds of tricks that we can do that all in aggregate were almost an entire second of time saved between tapping that icon and seeing the final UI with the data in it, in front of you.

[0:33:23.1] CM: Hopefully we'll have a bit of time to go more into some of the native integration. One of the things that the Relay compiler does its static time is optimize GraphQL queries. I'm curious, looking at a GraphQL query, it's kind of hard to imagine how you can optimize a GraphQL query. It's a pretty simple structure. How does that Relay compiler statically optimize GraphQL queries before they're even run?

[0:33:48.6] JS: Yeah. I guess optimization — Yeah, like I said. At first you look at it and you say, "Well, it's already only fetching the fields that I want. Where can I add optimization there?"

The thing that we're really optimizing is not so much the GraphQL query that we persist on the server and execute there, but the way that we process the results on a client. Generally speaking, when you're writing a GraphQL client, if you do normalization, which is basically having a mapping of identifiers to records so that you can keep your data consistent, what you have to do is basically walk the JSON response that you get back from the server together in parallel with the query structure in order to understand what arguments you use in order to fetch some data. If you have an array of users, was that the first 10 items? Is that the first 10 users?

Is that the first 10 users after some cursor? You have to understand what is being fetched in order to understand the response.

What we're really looking at is, "Okay, how can we reduce the time spent in traversing that response and query in parallel?" One of the first things that we realized was when we're writing GraphQL, we have encouraged as a best practice to co-locate GraphQL fragments with the UI that uses that data. What that ends up meaning is that lots of opponents might ask for the user's ID and the user's name or profile picture, and that's good. Each fragment should ask for the fields that it needs.

In aggregate, when you look at the query for a given user, the ID field kind of appears multiple times. If you naïvely construct a runtime presentation of the query and traverse it, you'd process the same field for an object, or you could potentially process each field multiple times, and you're just doing the exact same work over and over again, like, "Let's grab the ID from the JSON response and write it into the store," or "Grab the ID from the JSON response, write it into store." It's the same field.

The biggest thing was basically removing duplication in that data structure in order to process each field once if possible. There's some obvious flattening tricks you can do. We have some additional tricks of a field, basically fields that are kind of — We know that they must have been fetched via some other fragments, so get rid of them. We look at adding [inaudible 0:36:03.0] and we're sometimes able to statically know that they'll will be true or false and actually remove some trees.

Things like that allow us — In a benchmark, we took kind of a feed query, a sample feed query with some example data and compare the time spent to write that data into the store for the same query, same data with Relay Classic and Relay Modern. In a benchmark on a phone, that was not 10 times faster in Relay Modern, thanks to those types of optimization.

[SPONSOR MESSAGE]

[0:36:37.7] JM: Dice.com will help you accelerate your tech career. Whether you're actively looking for a job or need insights to grow in your current role, Dice has the resources that you need. Dice's mobile app is the fastest and easiest way to get ahead. Search thousands of jobs from top companies. Discover your market value based on your unique skill set. Uncover new opportunities with Dice's new career-pathing tool, which can give you insights about the best types of roles to transition to.

Dice will even suggest the new skills that you'll need to make the move. Manage your tech career and download the Dice Careers App on Android or iOS today. To check out the Dice website and support Software Engineering Daily, go to dice.com/sedaily. You can find information about the Dice Careers App on dice.com/sedaily and you'll support Software Engineering Daily.

Thanks to Dice for being a loyal sponsor of Software Engineering Daily. If you want to find out more about Dice Careers, go to dice.com/sedaily.

[INTERVIEW CONTINUED]

[0:38:00.6] CM: Interesting. Yeah, that's something I didn't even imagine that all the time savings are actually happening on the client. I was expecting the time savings to probably happen more on the server, but yeah, that makes a lot of sense. We've talked a bit about the static versus dynamic tradeoff in Relay Modern, but what are some of the other flagship features in Relay Modern that warranted rewriting Relay Classic?

[0:38:23.7] JS: Yeah, great question. Basically, when we started working on Relay Modern — Yeah, again, the main impetus for doing this was to achieve static queries. We looked at things that the community had been asking for, kind of feedback that we'd gotten from the community, feedback that we've gotten from internal developers and also just kind of long-standing ideas that we'd had of, "It'd be really nice if — Achieve a more elegant architecture if we did certain things."

One of things that we knew was having — We worked on garbage collection, or a.k.a, cache eviction in Relay Classic, but it hadn't really been designed in at the beginning and it wasn't super easy to build on afterwards. We really wanted it to make that core part of the framework going in. Garbage collection is basically the idea of; we know the GraphQL queries that are active allow us to basically — Given that the cache of data that we have, we can sort of — You can imagine like if you drew the graph of data that you have in the client on the board and the queries kind of tell you which parts to mark as still necessary and then take everything that wasn't marked and just get rid of it. That's the idea of garbage collection. That's as kind of important in longer running apps.

[0:39:34.3] LB: Overall, all of the features that we added were motivated by the shift from — We originally built Relay to help us build really great apps for the desktop web and as our attention shifted to building stuff for React Native on the context of pretty low powered, especially android devices on mediocre networks, that just kind of reframed all the prioritization around the things that we are building.

[0:40:01.7] JS: Yeah. In terms of low-end devices, memory is obviously a constraint on these devices, and so being able to preemptively free up memory from the cache hopefully means that you're running into JavaScript garbage collection less often, and that can take some time. It's a tradeoff of — Basically, the JavaScript engine doesn't understand the cache structure, it just sees a bunch of objects that are all referenced. It doesn't know what to free up. We have more knowledge and so we can kind of use that to attempts to play more nicely with the memory usage.

Although things are — A lot of people kind of ask, "How do we do local data management in combination with Relay?" For example, if you're wanting to use Redux and Relay together, is that necessary? What we saw was that once — We heard from people that we're using both, and what they told us was that most of their data ended up in Relay and that they ended up with very little left in Redux, really, jus for the truly local state.

Rather than attempt to make Relay integrate into Redux, we thought, "Okay, let's just make it possible to express that little bit of extra local state inside Relay." This is something we're kind of still experimenting with, but we now support basically client only extensions of the schema and then ability to just write the data into those — Write data for those fields and then read it back out as using plain old GraphQL fragments.

[0:41:21.7] CM: Yeah, I know the local state features of Relay Modern are definitely something I want to talk a bit about later. One of the major criticisms of Relay Classic is that it was very aimed at Facebook's needs. It had connections and mutations in a very specific style, but I'm Relay Modern you've done some work to allow extension of some of these patterns. Could you talk about how you built extensibility into Relay Modern so that you aren't tied just to some of the core specifications that mapped to Facebook's needs?

[0:41:52.8] JS: Sure, yeah. This was another thing that we kind of had in mind kind of going in is basically; let's take all the restrictions that we've kind of had in Relay an attempt to figure out a way around them. Some of them were pretty easy. For example, Relay Classic, kind of — Mutations at Facebook have a client mutation ID field and we kind of have acquired that on Relay Classic, and that was pretty easy one to get rid of Relay Modern. Relaxing the arguments to mutations was something we had a specific — You'd have a single mutation input argument in Relay Classic. Again, we just removed that constraint. Supporting arbitrary root fields with arbitrary arguments was also something we wanted to do for Relay Modern.

The one kind of big schema restriction that we still have in Relay Modern that we're hoping to work with the community on is object identification. We still rely upon schemas implanting the node interface with a globally unique ID field in order to know how to find identify an object. That still in Relay Modern. We'd like to find some way to kind of make that more flexible and hopefully addressing those as part of the GraphQL spec.

You touched on connections which I think is a really important topic. In Relay Classic, connections that was — The Facebook connection pattern was built into the framework, so you didn't have to use it, but it was sort of there and it was definitely the most convenient way to do pagination. What we wanted to do was make it possible to do that type of pagination but using an arbitrary approach, and so we kind of took pagination out of the core and basically enabled the core primitives to build those types of things in user space.

If you look at the way connections were handled in Relay Modern, it's actually effectively a plug-in. It's a compiler plug-in and a runtime plug-in. It's included by default with the framework, but it's still something you have to actually enable when you create an instance of Relay. We need to actually go to the process of documenting this and making it easier for other people to create similar types of things, but it's easy to imagine, for example, a windowed pagination or limit offset style pagination that uses the same core primitives that we've built for the Facebook connection model.

[0:44:00.2] LB: I'll also add that while we no longer require using that connection model for Relay Modern, we'd still think it's an API best practice. In fact, as we were having meetings with GitHub as they were building towards the beta release of their public GraphQL API, at first we were suggesting that they use a much simpler way of exposing lists of things. They actually came to us and pointed out a bunch of cases where the connection model was actually a best fit for them not motivated by the desire to use Relay, but just motivated by the actual shape of the data that they needed to expose.

I think it's maybe not surprising that the thing that we landed on at Facebook for the best way for us to page over sets of information ended up being also a pretty reasonable API design pattern for other companies as well.

[0:44:48.4] JS: Yeah, I certainly agree that independent of GraphQL connecting pagination is a pretty good idea.

[0:44:54.0] CM: For about two years, say, you had Facebook employees writing Relay Classic code. When you rewrote Relay you obviously couldn't ignore them. What was the support or upgrade plan for products and teams at Facebook using Relay Classic?

[0:45:09.8] JS: Yeah, it's a great question. We started off by, again, rewriting Relay, kind of tested it out, kind of building some. We had a fairly simple view that was using Relay Classic and it's was a standalone view that we converted completely to Relay Modern just as a kind of experiment. Once we were confident enough that the framework was ready to kind of really start taking really expand to other products, we developed what we are referring to as a compatibility layer. The basic idea is that we back-ported the Relay Modern kind of React API, so simpler, kind of simpler and more streamlined components for fetching a query and rendering its results and defining containers that have fragments and React views collocated together.

We have a streamline versions of those in Relay Modern and we basically implemented that API in a way that you could write your components using that new API but still run them inside of a Relay Classic app. You could incrementally convert from the bottom-up your components and they'd still be running using the Relay Classic runtime, but once you had converted an entire view, you could switch it over and actually use the Relay Modern runtime.

This is the process that we followed for a few internal apps such as market place, which I mentioned. We also have JS codeshift scripts that will, given a Relay Classic container, automatically go through and attempt to rewrite it. Simple things that don't do any re-fetching can certainly be rewritten completely by this, by the script. If you're doing re-fetching, for example, paginating or fetching more data, it kind of gives you some to-dos about and suggest ways to achieve that in the new API.

[0:46:54.5] CM: Why did you decide to build a compatibility layer instead, say, taking a couple of weeks and rewriting these products using Relay —

[0:47:05.7] LB: No, it would be a never ending process. Yeah, this is the biggest mistake that any company can make, is to — We've actually made it at Facebook a couple of times in our history which is why I can be confident in saying that's a terrible thing to do, is stop what you're doing, stop building product features. Everybody, let's focus on migrating from this old thing to this new thing.

At any point at Facebook, there are multiple of these efforts going on. You might be undergoing a migration for the data that you're getting from API, the migration for the tools that you're using to fetch that, like to be a Relay. It might be migrating between different backend services to serve up a service. It might be migrating between all kinds of different things that we're using. If we had to do all those is like a tight lock. We'd never get anything done.

We've tried this in the past where we've said, "Okay. No new features until we finish some big technical migration project," and it was a massive mistake for us in the past. We stopped building product for almost an entire year on some platforms when we decided to go try to do these kinds of things. It doesn't work out well for us. I don't think that that's unique to Facebook. As I talk to people from other companies, everything is changing at their companies all the time as well. They’re building new products. They’re solving technical problems and they need to be doing all these things at the same time. It's my opinion that any software that goes through major breaking changes that doesn't give people an incremental path to handle those breaking changes one file at a time will just be the kind of software that gets left behind on old versions for most people.

[0:48:44.4] CM: What role did the Apollo GraphQL client play in Facebook's decision to build Relay Modern?

[0:48:51.9] LB: I don’t think that it did, have a role it all. I mean, we were super excited to see the Apollo client come out because more tools for people is always a great thing. Also, a lot of the great ideas that we had built into the first version of Relay and the kinds of things we were talking about both through GitHub on the issues and in poll requests and individually with people when we’d go to meet-ups, a lot of those were showing up in the Apollo client, which was super exciting, because our goal when we open-source things is not to write the software that everyone uses. Actually, there's only so much value we get out of having people actually use actual lines of code that we ship. The thing that we really want to do is build communities, build ecosystems, and share ideas. We were super excited to see Apollo build all those things.

As Apollo sort of worked through the first versions of their client, we spent a lot of time talking to them as they would stumble into problems. We’d talk about the solutions that we had to similar problems on other platforms, and I think it's no surprise now that you'll see Relay Modern and the latest version of the Apollo client actually similar in more ways than they are different. That's a direct result of us as a community meeting at these meet-ups talking about what the best practices should be for these kinds of clients as we learn.

[0:50:15.3] JS: Basically, it really all goes back to us looking at Relay Classic and the ways that we’re using it in the scenarios that we’re using it in and basically deciding, “Okay, we think there’s room for improvement here,” and kind of going after it. It’ really great that we’ve ended up in kind of a similar place, and I think it opens up the door for hopefully more collaboration now that, as we mentioned, we’re more similar than we are different, which is great.

[0:50:41.9] LB: Yeah. The biggest inspiration, I think, for Relay Modern, was how we use GraphQL on our native apps. In the same way that I was mentioning before, the biggest motivation for rethinking the Relay core was trying to figure out how to build super high quality software for really underpowered mobile devices. Of course, first, we went and asked the iOS and the android teams at Facebook what are they doing that we're not doing. I think, Joe, in particular, spent a lot of time with our iOS team to learn the best practices and lessons that they had learned from years of using GraphQL.

When we first built GraphQL at Facebook, the iOS team was the very first team to start building client software that would use it, and so you can actually see a lot of the decisions that our iOS GraphQL teams have made through the lens of the decisions that we made in Relay Modern.

[0:51:36.3] JS: The store basically was myself and a couple others from the team kind of spending time in New York, the iOS GrpahQL team, and then kind of brainstorming, "Okay, this is how the native version works. How does that translate to JavaScript? Does it translate?"

There are certain things that do and certain things that don't, particularly obviously all the stuff around memory management. There's kind of some different approaches for managing memory and doing garbage collection. That has to be a bit more manual in JavaScript. A lot of really great lessons there. It's really exciting to bring that to the JavaScript community and hopefully we'll have a chance to talk more about the technical approach there, because I think there are lessons to be learned. Certainly, weaknesses of the language, that would be great to see addressed.

[0:52:21.3] CM: Yeah, certainly. Now, I want to talk about a couple of the high-level themes in Relay Modern and software in general and kind to discuss them and what their place is with Relay Modern.

The first thing we're going to talk about was address a bit before. The first formal state management system for React released by Facebook was flux. After that release, a flurry of different state management solutions were unleashed by the React community with the "winners" appearing to be Redux and MobX at least, for today. Relay Modern has client fields feature which allows users to extend their GraphQL schemas with extra fields that are local to the product. How do client fields in Relay compared to state management system like Redux?

[0:53:08.7] JS: Yeah, it's a good question. First, we should make clear that this is kind of an experimental feature. Our native clients have used these types of schema extensions for small amounts of local data. This is not a brand-new idea for the GraphQL ecosystem within Facebook. I think in terms of — I would hesitate to say that you could gotten completely replaced Redux with Relay or the same for MobX. This is something that we see this use case of sometimes you need to have a little bit of extra bit of extra data relating to objects that come from the server and these type of schema extensions are a pretty clear way to address that use case. Whether it also extends during larger amounts or completely local state is I think something that I'd to kind of see the community experiment with.

Yeah, I definitely don't — Our goal is not to replace Redux or MobX, but just to kind of off — We're really targeting the applications that fetch a large amount of data from the server where GraphQL is a great use case and that's the main focus for Relay.

[0:54:17.6] CM: In the future, what do you think about the possibility of perhaps using Relay without a GraphQL server to manage state entirely on the client side with the client side defined types?

[0:54:30.0] JS: I'd love to see somebody try it and report back at how well it worked out. The interesting thing is that in the Redux community, and I'm less familiar with MobX, what the best practices are there. Certainly, for Redux, the best practice is to normalize your data and then you have selectors that sort of take this normalized data to read it back out into an object that you then use in your view function.

Then you look at what Relay is doing, and a fragment is effectively a selector that takes normalized data and reads it back out into a convenient shape for rendering your UI. The process of taking a GraphQL query and its response from the server and putting it into a normalized cache is all the work that you would've done manually to normalize that's happening automatically.

It's not hard to imagine somebody defining a local schema, using it to put data into the store, read it back out from the store, but it's something that we haven't done. It's not the use case, the primary use case that we're targeting. It'd be good to see if you'll try it out and see where it works, whether it works. I imagine at least at some parts of the Relay runtime could be useful for that whether you end up using all of Relay are not.

[0:55:35.3] CM: Yeah, I definitely think it's an interesting thought, so it'd be fun to see someone try. Relay Modern was impart, and you mentioned this, designed from the ground up to be fast in React native and especially on low-end android devices. A large part of the React Native narrative is that React Native can be used right beside native code. You don't have to get 100% JavaScript, you can use React Native beside Java or Objective- C code that you already have.

In these hybrid React Native apps, engineers may have Relay running in JavaScript with its own data store and another GraphQL client running in native iOS or android code with its own data store. How important is it that Relay and native GraphQL clients share the same data?

[0:56:20.2] JS: Yeah, that's a great question. How important? I think I actually depend upon the application. That's something where — Even within, for example, a purely native app that uses GraphQL or a purely React Native Relay app, the degree to which — Basically, I guess the point I'm trying to make here is that data consistency is something that we've found to be — There's a spectrum, and it's not necessarily that 100% full data consistency is always exactly what you want.

Just like in databases, people have discovered the value of eventual consistency for certain cases. I think that eventually a consistent UI is also sort of often a good tradeoff. The degree to which — A simple example there being a mutation. You might not know the full effects of a mutation when you execute it on a client, but it's enough to kind of apply — Temporarily applied part of the change optimistically while you wait for the full change to come from the server. In that intermediate sate while you're waiting for the server response, you're sort of eventually consistent. It's partially consistent. It's not totally correct, but it's good enough. That's an example of what I'm talking about.

I think the same thing applies with JavaScript and native data access. You have to consider how consistent do you really need to be. This is a problem that we've worked on at Facebook because, obviously, we have React Native views, we have native views that have different data stores. We have some solutions in place that get us some degree of data consistency. We're working to achieve greater data consistency overtime, but it's definitely something that — It's interesting to see — Apollo had a blog post about this a couple of days ago where they're working to align their client APIs across the different platforms, and that's really great, great to see, and that's' something where, because they have clients for all the platforms, in open-source, they can work to iterate on that, and that's really really awesome to see, and I'm excited to see what they come up with there. Obviously, we haven't open-sourced our native client infrastructure. Yeah, looking forward to seeing what Apollo does there.

[0:58:24.1] CM: Yeah, is there a future where Facebook will release or perhaps re-factor some their native client infrastructure to provide this feature in open-source?

[0:58:34.3] LB: We've talked about it, but at this point a lot of the value that we would reap from doing that actually is actually already out there. Like I was saying before, the main value we'd get from open-sourcing this stuff isn't getting people to use the lines of code that we write It's sharing the ideas that we're using to make building these apps successful in the first place.

The large majority of our iOS and android infrastructure that deals with GraphQL is really tightly tuned to a very nonstandard iOS and nonstandard android app that we've built. Even if we did share a lot of those things, they wouldn't really be immediately usable by most people. It's all the more reason why we're excited to see Apollo go down a very similar path, is this yet another channel where we can share some of those ideas and then, by proxy through the Apollo project, share a lot of the actual lines of code that will work for people?

[0:59:28.1] CM: Okay. Then this last kind of general topic is I know something, Lee, you've tweeted a bit about, webpack, being more the future. I'm sure, Joe, you have some ideas as well. Looking at the software field today, it feels like the era of interpreted languages is coming to end. The cross-platform promise of interpreted languages, like Java, Ruby, Python, and JavaScript are not fulfilled by projects like LLVM in the future web assembly. Facebook is even investing in a compiled language for front-end development, Reason. Relay Modern is moving more work to build time and out of the interpreted runtime. Given this, are interpreted languages like JavaScript doomed in the long run?

[1:00:11.0] LB: Maybe. Although I think it's important to look at why we spent so much time and energy on interpreted languages in the first place. The massive win that you get from those is just as amazing incremental development time where to the first real interpreted language that I used was PHP and sort of the early days of learning how to do web development myself. It was amazing to me that I could go to my editor, change a line in the file, hit save, reload my browser, and immediately see that change. I didn’t need to wait for a compiler to run. I didn't need to do anything. It was almost instant. These days, we even have tools on top of that such that whenever you save a file, the web browser reload for you and it might — Like Hot load, or it does some amazing things to make it so that as soon as you save a file exactly that part of your updates.

I think interpreted languages showed us just how fast the development cycle could be and after doing that there’s just no way that I could go back to a C++ code base and feel like I'm moving quickly. I think what's really the most fascinating to me as we’re looking at the most — The newest, the most modern languages, like Swift, Reason, and their like, is that they're trying as hard as they can to maintain as much of that incremental compilation and fast development cycle as possible while still maintaining the benefits of a compiled language that I think we mostly did not succeed at achieving with interpreted languages which is the ability to spend a lot of computer power up front such that at run time you spend a little computing power as possible. That's really important for systems engineering. It's not nearly as important for UI engineering. I think we’ve spent decades building — A decade and a half building JavaScript applications at this point now. Basically, can prove that, yes, you can build pretty awesome stuff in interpreted language.

Now, we’re seeing the need to have a little bit of that, like a mix. I think a mix is really really interesting, where you can have interpreted stuff in addition to bits and pieces of compiled stuff to reap somewhat of a best of the both worlds. I think that'll they'll take us another couple of years for sure. Some more stuff to learn from.

[1:02:28.4] JS: Can I add one thing there?

[1:02:29.4] CM: Go for it, please.

[1:02:30.1] JS: Which is Relay compiler is kind of doing a little bit of what Lee is talking about there, where we want the benefit of compiled ahead of time optimized queries, but we want to make the development cycle as fast as possible. Having — The compiler is not fully incremental right now, but we're working to kind of always — We're working to improve the speed and make it more focused on doing the minimal amount of work necessary given whatever files you changed, kind of making of more incremental.

I think, an example, that type where it doesn't have to be one or the other, you can kind of get some of the best of both with those types of tooling. Yeah, I'm really excited to see that type of thing. For me, if you're looking at the Relay runtime codebase, it's flow-typed. It looks a lot like — It is very strongly typed for much of the code, and so it's really interesting to see what can languages going forward do to take advantage of the fact that the code actually is so static and is strongly typed to get the best of the nice developer experience while you're writing the code and kind of clean, easy to understand a code, but also also making that go faster at runtime. I'm just excited to see what happens in this space.

[1:03:44.5] CM: Certainly. How do you think React and Relay will change in a world with web assembly, to take advantage of web assembly?

[1:03:51.0] LB: We have no idea yet. Web assembly is still super new. At this point, web assembly gives you a bite buffer and it gives you some assembly style tools on top of that, but what it doesn't give you yet is access to the JavaScript object model, the garbage compiler, the Dom, all the pieces that you need to actually build a front-facing UI application. Those things will come in with time, but until they do I think it's kind of too soon to speculate on what we'll be able to make with that kind of thing. I think once they come, Relay is not the thing that I would be looking at as an exciting thing where changes might come once we get web assembly. That can interact with all these pieces.

What we'll see is languages like Rust, or Swift, or Reason having compiled to whatever the future web assembly looks like to give us the ability to eke out even more wins on a memory footprint or a CPU footprint, which honestly might not matter at all for a desktop browser but might be really significant to get that last bit of performance. That could be the difference between a mediocre experience and a great one for a very terrible multiyear old android phone that someone is stuck using.

[1:05:04.8] CM: Yeah, definitely. It was super great talking to you, Joe and Lee. Thank you for joining me on Software Engineering Daily.

[1:05:10.1] LB: Sure, it was a pleasure.

[1:05:10.9] JS: Yeah, thank you.

[END OF INTERVIEW]

[1:05:15.5] JM: Thanks to Symphono for sponsoring Software Engineering Daily. Symphono is a custom engineering shop where senior engineers tackle big tech challenges while learning from each other. Check it out at symphono.com/sedaily. That's symphono.com/sedaily.

Thanks again to Symphono for being a sponsor of Software Engineering Daily for almost a year now. Your continued support allows us to deliver this content to the listeners on a regular basis.

[END]