EPISODE 04

[INTRODUCTION]

[00:00:00] JM: In 2011, Facebook had begun to focus its efforts on mobile development. Mobile phones did not yet have access to reliable, high-bandwidth connections, and the Facebook engineering team needed to find a solution to improve the request latency between mobile clients and backend Facebook infrastructure.

One source of latency was recursive data fetching. If a mobile application client made a request to the backend for newsfeed, the backend API would return the newsfeed, but some components of that feed would require additional requests to the backend. In practice, this might result in a newsfeed loading partially on a phone, but having a delayed loading time for the comments of a newsfeed item.

GraphQL is a solution that came out of this problem of recursive latent data fetching. A GraphQL server provides middleware to aggregate all of the necessary information to serve a complete request. GraphQL connects the backend data sources and federates the frontend request across these different data sources. GraphQL was open sourced in 2015 and has found many uses in addition to simplifying backend data fetching for mobile clients. Today, GraphQL is used by PayPal, Shoppify, Twitter and hundreds of other companies. Lee Byron is the co-creator of GraphQL and he joins the show to tell the story of GraphQL and how it fit into Facebook's shift to mobile.

We have new apps for Android and iOS. These apps are a great listening experience for Software Engineering Daily. They include all of our old episodes. They’re categorized. They’re related links and greatest hits and social features. You can comment on the episodes and discuss them with members of the community, including me. I will be discussing and commenting on each episode that gets published in the software daily app, and you can join the discussion as well. You can also get ad-free episodes using our app. You can go to softwareengineeringdaily.com/subscribe and skip all the ads.

Also, the company I'm working on is FindCollabs. FindCollabs is a place to find collaborators and build projects, and right now is a great time to start a project, because we're having an online hackathon with $2,500 in prizes. We've got all kinds of cool engineering prizes. If you want to find the details of that hackathon, go to findcollabs.com/open, and I would love to see your projects. I'd love to see whatever you've got to post.

With that, let's get on to today's episode.

[SPONSOR MESSAGE]

[00:02:52] JB: Today's show is sponsored by Datadog, a scalable, full-stack monitoring platform. Datadog synthetic API tests help you detect and debug user-facing issues in critical endpoints and applications. Build and deploy self-maintaining browser tests to simulate user journeys from global locations.

If a test fails, get more context by inspecting a waterfall visualization or pivoting to related sources of data for troubleshooting. Plus, Datadog's browser tests automatically update to reflect changes in your UI so you could spend less time fixing tests and more time building features. You can proactively monitor user experiences today with a free 14-day trial of Datadog and you will get a free t-shirt.

Go to softwareengineeringdaily.com/datadog to get that free t-shirt and try out Datadog's monitoring solutions today.

[INTERVIEW]

[00:04:02] JM: Lee Byraon, you are an engineer at Robinhood and you were a longtime engineer at Facebook. Welcome to Software Engineering Daily.

[00:04:09] LB: Thanks for having me.

[00:04:09] JM: You are part of the initiative to get Facebook on to mobile, and this was a key inflection point in the business. Some people may not remember, but there was a time when

Facebook was only used as a desktop web application. When did engineers at Facebook start to realize how important mobile was?

[00:04:29] LB: I think there are a couple moments over the years where that happened. There wasn't like a single inflection point. Early on, Facebook was definitely just a website that you went to on your desktop computer or maybe laptop, because it started in 2004. That's years before the first real smartphones. Really, two things happened in the really early years of mobile. One was a focus on flip phones and other popular phones that had some very basic access to the internet.

So, Facebook, for a long time had an m.facebook.com mobile website, and for the first few years that was really about extending to a second screen and creating a small version of Facebook that you could take with you. Then when the iPhone first came out in – What's that? 2007 or 2008, there was first a version of it built as a website. If you remember, the very first version of iOS, you couldn't actually make apps for. You could only make websites for. There's one engineer at Facebook who did that, and the following year when the software SDK came out, that engineer sort of abandoned the website that he had built and built a mobile application instead.

That was sort of the very beginning. Then there were a couple of other interesting moments that happened. Another was sort of the recognition that mobile was becoming somewhat important as a second screen. So, I would say this started happen in maybe 2009, and that's really when I started helping out.

I was the designer for mobile at that time. I joined Facebook originally as a product designer, and touch devices were becoming more popular, but were still very much considered a second screen as an alternate experience temporarily while you are away from your desktop computer or laptop. But we really in 2009 took a big refocus on making that experience feel good. The native application for iOS that we had was extremely limited. It was built by a single engineer. That engineer had left the company. It had kind of started to fall very far behind what the rest of our product was doing. So, we built a version of the mobile site that felt good on a touch device. That actually ended up becoming a really important platform to start building products for mobile.

One of the really challenging things that I faced when reaching out to other designers or other product people at Facebook in those earlier years, try to convince them that mobile was something important was the scary scenario where you'd spend all your time and energy focusing on designing and building an experience for a big screen in a browser. Already, the problem of making it work across Internet Explorer, Firefox, Chrome was kind of problem enough.

When it came time to move things to mobile, the prospect of supporting an iOS native app, an Android native app, a touch optimized website and a flip phone optimized website sounded like it would just make the amount of engineering work 5X.

So you could imagine that, especially if you don't quite know how important mobile is. It's totally reasonable that your instinct would be, “Maybe we should look into that later. That's not the right priority now.” So, as a consequence, the mobile product constantly lagged behind.

So, when we did this effort to make the touch optimized website just really good, just much better than anything else that we had built for mobile up until that point, we kind of reframed that conversation. Instead of going from one platform to five platforms, we were able to make going from one platform to two platforms. Because if you built for that mobile website, you would automatically get flip phone and basic phone support and we would deliver that directly to iOS and Android apps via essentially a glorified web browser that we had built. So our iOS and Android apps were these glorified web browsers with a little bit of extra sauce on top. They let you upload photos. They let you sort of include your location for check-ins. But other than that, they were really just a thin wrapper around our website.

So that really simplify product development, and that was an early accelerant for the Facebook product team to start thinking about what building for mobile would look like and start making that more of a priority.

Then, I would say the next inflection point was in 2012, and there are a lot of really interesting things happened in 2012. One is that we saw the chart of mobile-only users take up dramatically, where it now became clear that mobile was not a second screen. There was not

just going to be a significant number of people who only use mobile soon and very soon. There was going to be a majority of people who only use mobile. In fact, desktop would become not only use less than mobile, but desktop would become the second screen.

So, we saw this inflection point coming. We realized that we are quite far behind on product. 2012 was also the year that Facebook did its IPO. I remember in the early filings and Mark Zuckerberg's tour around where he's talking to potential investors. He had to highlight all the risks for the company, and one of the top risks was that we saw this giant shift to mobile devices and that we haven't figured out good product for mobile yet. That's also when within Facebook we were sort of revisiting this technical decision to build everything on web and bring things back to native development on both iOS and Android, which sort of unfolded into a lot of super interesting engineering work that we had to do.

[00:10:03] JM: As you were noticing the increased use of mobile as the primary device that Facebook was consumed on, were you noticing different use cases for people using Facebook? Was the mobile use case different than the desktop use case?

[00:10:22] LB: To a degree that it was. In fact, that was kind of our earliest hypothesis, was when you would take Facebook with you on your mobile device, that you weren't just doing the same things that you would do on the desktop, but there was a small set of things that you would do really differently.

So, we knew that chat was going to be really important, because you wanted to be able to keep up with your friends while you're away from your computer. We knew that location services were going to be really important, because that was just a unique quality to mobile, that you just couldn't quite get on a desktop browser. Photos were going to be important.

In fact, I think we even underestimated that, because Instagram later really proved just how important photos were to the mobile experience over desktop. So, we kind of knew from the very beginning that mobile usage was going to be different, and that certainly did play out to a degree. But I think where we really missed the mark was trying to get a better sense of how people would use the real core experiences.

So, for example, the newsfeed on mobile was really just a shrunken version of the desktop newsfeed. We just made it fit on the smaller screen. Similar with groups and events, they were just kind of the same product, but shrunken down. We missed a lot of the nuances on how people would engage with those things differently.

Now, actually, the conversation is the other way around. Now that mobile is the dominant platform, the default designed experience is how people use their phones. Now we actually think about the desktop as the second screens and now we're actually looking at how people use desktop browsers differently than they use their mobile phones. In fact, sometimes there're tasks where you really just want to sit down at a bigger screen to get them done. Those are appropriate for the desktop to do differently.

But the big difference was sort of before 2012, mobile was always a subset of the features that you could get to. There're always cases where you just couldn't get it done on a mobile phone. You had to go to a computer to get it done. After 2012, it became clear that that was just not acceptable. But there were some people for whom that wasn't an option. So, the mobile experience had to have all features, and the desktop browser could just sort of extend and augment those to do things that the desktop itself is good at.

[00:12:34] JM: The first effort to solve Facebook for mobile was this HTML5 solution. What were the deficiencies of the HTML5-based approach to Facebook mobile?

[00:12:46] LB: There were definitely pros and cons to that approach. As I was talking about before, the real benefit was that it unified all the platforms, and that was the dream. Just to dramatically reduce the amount of work. In fact, we ended up having this phrase that turned into sort of a bad language, which is right once, run anywhere. Now, anytime anyone hears that, a red flag immediately goes up, because that approach just didn't work.

The reason that it didn't work, the deficiencies were really the technology on mobile browsers in particular. You don't think there's anything inherently wrong with the web platform or HTML as a technology. In fact, a lot of the UI tech that we've built on top of the web has extended itself into mobile domain really well. I think, React as the pattern has both extended itself into natively implemented versions and component kit and emulated versions like React Native.

But, our early bed that ended up being wrong was that the vendors, the mobile vendors, would compete to have higher quality mobile browsers. Because I can remember when Steve Jobs went up on stage to introduce the original iPhone, he held it up and he said, “It's a phone. It's an internet communicator.” He really focused on it being a portal to the web.

I remember being really excited about that, because that was our bread-and-butter. That's what I understood really well, and it felt like a great open platform. Then later when Google acquired Android and started really investing in Android, we thought, “Hey! Google builds Chrome. Google is a web company. They're going to try to compete with Apple on having a much higher quality browser, and Apple is going to be forced to compete and continue to improve their browser as well.” We really hope to see this future where developers would build websites as web applications, and Google and Apple would compete with each other to see who could run those things better.

But, boy, were we wrong. I mean, first of all, the Android team had the worst browser out of all. We didn't really understand the divisions within Google between the Android team and the Chrome team. While versions of mobile Chrome later appeared – And these days, the Android browser is actually quite good and it is actually Chrome-based. For years, the version of the Android mobile browser was just years behind desktop browsers and suffered from critical bugs, which meant there was no pressure, competition pressure, on Apple.

Then, later as Apple is trying to figure out how to help developers build the best experiences on their platform, they bent strongly in favor of native applications and interesting SDKs to expose and really left mobile Safari to rot. Every year, I think we’ve seen a real renaissance in desktop browsers over the last 10 years. I really credit Chrome with that, but there's been just a lot of awesome activity out of the W3C and the WHATWG of just introducing more and more super interesting technology into the we platform.

But we would see in mobile Safari and Android mobile even worse, just multiyear lags for those technologies to be adopted. Then when they were adopted, there were critical bugs. So we couldn't even use them. So, that just put a huge damper on what we could build in the first place.

Then the second was just accepting the constraints of the device. I think we really underestimated just how many clever tricks that the native SDKs employed to get acceptable performance for native applications.

So, for example, one of the real innovations of iOS in its very first version was this idea of a scroll view where you would take elements of the top of the scroll view that went out of view. Move them to the bottom of the scroll view, repaint them. So that as you're scrolling, it feels like you're scrolling through an infinite list of things, when really you're scrolling through the same set of 10 objects where you're just sort of repainting them as they go around. That trick is just not readily available to the web. We tried. We had extremely smart engineers build many prototypes of this exact model, and all of them failed in some respect toward some edge case.

In fact, when they do fail, what happens is as you're scrolling, you sort of fill the graphics buffer up with rendered content on the web until the point where you exhaust all memory on the machine. Then the operating system is forced to kill you.

So, in the case of the browser, you might see your browser tab freeze up, or the browser itself crash. In the case of a natively wrapped application, you would see that browser or you'd see your entire app crash. Then another issue was JavaScript performance. So, in the in the browsers, JavaScript was accelerated. You would see what really fact performance due to sort of multistage GIT acceleration of the JavaScript.

But because of some security issues, both iOS and Android were very limiting in how they would apply that same acceleration to a native app, because they were concerned that a native app would exploit those things to have bad behavior on the platform.

So, just performance of just actually doing business logic was quite a bit slower in these wrapped native apps. So, really, it played out sort of exactly opposite to how we predicted that it would, which is why ultimately we had to admit defeat and sort of publicly admit that HTML5 was the wrong bet to build a mobile app and start over.

[SPONSOR MESSAGE]

[00:18:21] JM: There are so many good podcasts to listen to these days, and it can be hard to make time to sit down and read a full-length book, and there are more good books than ever. I like business books, and self-help books, and history books, but I don't have time to get through all the books that I want to.

Blinkist gives you the best takeaways, the need to know information, and the important points of thousands of nonfiction books condensed into 15 minutes that you can read or listen to. I like Blinkist, because I like audio, and Blinkist is an innovative, useful audio format. You can get a free 7-day trial and support Software Engineering Daily by going to blinkist.com/sedaily and signing up.

On Blinkiest, I've listened to a few very long books about China in their 15-minute form, and I also use Blinkiest to review great books that I've read in the past, such as *Principles* by Ray Daio, or *Being Mortal* by Atul Gawande.

Try out your free 7-day trial by going to blinkest.com/sedaily and signing up. That's blinkiest.com/sedaily,and get those books condensed into 15 minutes, and get more throughput in your book reading activities.

Thanks to Blinkiest for being a sponsor of Software Engineering Daily.

[INTERVIEW]

[00:20:00] JM: Facebook was not the only company to go down this path. There were many people who thought that HTML5 would be good enough, as you articulated. Eventually, most of these companies, if not all of these companies, figured out that they didn't need to build native mobile experiences for Android and for iOS, and this creates a management question. How do you set up the teams for Android and iOS? What's their relationship to the HTML5 mobile app that you've already built that you now want to deprecate? What's their relationship to the desktop team? What's their relationship to the design team? How did you define the team strategy for these different mobile apps?

[00:20:47] LB: I think this is a super interesting topic, because there's no good answer. In fact, at Facebook, we went through a lot of different forms of that especially as we grew. So, first, we didn't actually ever deprecate the mobile site. In fact, today if you go to m.facebook.com, you'll see actually the exact same architecture that we originally built in 2009 has actually scaled quite well. It still works very well as a mobile website, and there is in fact some set of people for whom either they're not on iOS and Android. Less the case these days, but was the case for many years, or they just prefer to use the web for over a native application. So we never really deprecated that, but we certainly diverted some energy away from it.

So, initially, when we decided that it was time to build a native app, the real management concern that we faced was that we just didn't have enough iOS and Android engineers at the company. We had have this multiyear sprints building mobile experiences on the web, which never really challenged the engineering expertise that we hired for.

So, when it came time to build, we started with an iOS native app first and then moved to an Android native app after that. We first really struggled to hire really high-quality iOS engineers fast enough, because the product surface area was quite large. So, we started on just one element of the app. We just started on newsfeed. So, we wanted to make sure that when you opened the app, the immediate thing that you saw was native rendered.

Then as you sort of moved to the app, you actually would go back into the previous architecture that we had of the wrapped web view. Actually, to this day, there are still some edge pieces of the Facebook app where you'll find yourself in a web view, and you probably won't even notice that you're on a web view, because devices are so powerful these days that the kind of limitations we face in 2012 aren't quite as painful now as they were then.

So, we built up this iOS team to focus on building iOS version of newsfeed as kind of like a SWAT team, very specific goal. Then once that was proved to be somewhat successful, the team started to expand to take on more of the application. So, they moved into working on profile, then photos, then groups, then events, and more and more of the core functionality of Facebook until the point where we had a pretty sizable team of iOS engineers that operated in a bubble.

So, we had more than 20 iOS engineers that were responsible for the entire iOS app end-to-end. How they related to the other teams was somewhat strange, and that they would really consult with those other teams, consult the designers, consult with other engineers and product managers working on these various products, because elsewhere throughout Facebook, there's a team dedicated to news feed or a team dedicated to profile or a team dedicated to photos. That was a mix of engineers and designers and PMs to be able to move really fast in that one particular product are. So this was a weird quirk, but we decided that it was the right thing to keep those engineers together so that they could work together to build a high-quality app, which we decided was more important than building individual high-quality product experiences that we were concerned with software if we had zero, one or two iOS engineers per product team.

Eventually though, we did have to make that shift. We did it on both platforms, iOS and Android. Maybe a year and a half or so after we started it on this transition, and it actually started in a way that in retrospect didn't work quite so well. So, we understood that we just couldn't get iOS and Android engineers fast enough. So we decide the next best thing would be to train internal engineers, existing engineers at Facebook, to learn iOS or Android if they were interested in that

A lot of engineers sort of saw the writing on the wall. They saw that iOS and Android were going to be really important platforms. They saw as a huge opportunity for themselves personally. So, that was actually pretty successful, and we brought in some external consultants who did a great job at setting up an education schedule for everybody, and there's these two weeks sort of intensive programs you could go in to learn.

But you can imagine, even for a really smart and experienced engineer, only two weeks on a new platform really only gives you the basics. So, a lot of the folks that came out of this were beginners, and the set of people who were working on this original core iOS team were a lot of the folks who had come in externally who had many years of iOS experience. So, there is this experience in balance where the core set of folks had a lot of experience, and these people who were out on these independent product teams had very little experience.

So, the next set of problems that we faced from sort of a big picture point of view was how to deal with bugs. So, when we're looking at the web platform, bug tracking is very different,

because you do a release, and the release goes out to all customers immediately, and any bug that happens has a PHP stack trace or a JavaScript stack trace that you can immediately match back to a commit that probably happen sometime in the last couple of days, because Facebook tended to release at worst, weekly and typically daily.

Mobile is quite different. We were releasing closer to once a month. So, there is a huge amount of stuff that could've gone wrong, and it's quite hard to build back a stack trace of a crash and then link that stack trace to an actual commit that went in that caused that bug. So, typically what happened is there were more and more of these inexperienced iOS and Android engineers throughout the company building products. They would introduce bugs. Weeks later, those bugs would get shipped out in a build. Customers would face those bugs. We would see crash reports or complaints. Because there is no way to sort of divvy out those issues to the various product teams, because it wasn't always immediately clear just looking at a stack trace. Those would go to that core iOS team, which ended up being this toxic relationship, right? Because you have this set of really experienced folks who used to be focused on building all of products, and now they're really focused on building the network layer and the data layer and this sort of core infrastructural pieces that are shared across all these teams.

But, really, now they're focusing on fixing bugs. That's not whatever everyone wants to spend all of their engineering time on. So, they start to be upset with all these other engineers, say point blame, and they say, "Oh! These inexperienced engineers are causing all these bugs, and we have to fix them. This sucks!"

So, we tried a lot of different things to solve this problem. We have this core team start focusing on building tools to help better identify blame commits responsible for these bugs. But, ultimately, the right thing was to just blow up that central team and send experienced engineers out to each of these individual product teams. Luckily, by that point, we had hired enough iOS and Android engineers and we had trained enough iOS engineers internally, that each of the core product teams could have sort of enough of iOS or Android specific engineers that they could sit next to somebody that had that kind of expertise that they had and ended up in healthy teams. But, man! It was a rough go from the very initial [inaudible 00:28:03] around 2012 until we ha sort of healthy teams that had per technology product teams. That was really 2015.

[00:28:12] JM: Your experience building the early mobile applications for Facebook, this led to discoveries of constraints on mobile applications due to connectivity, due to device speed. What were the constraints of developing a complex mobile experience back in those days when we're talking about middleware and backend infrastructure?

[00:28:41] LB: There were a ton of constraints. There're constraints all the way from the frontend to the actual built software, to the release process, to how we talk to the network. Every single one of those introduced new constraints that were new to Facebook as an engineering and a product organization.

From the frontend, I already talked about some of them on the web platform being very limited in terms of what we could render. A problem that we faced really quickly as we shifted to native applications is just accepting the size of the product. Facebook does so many different things, and most of the web product, all of the business logic happens on the server. The server sends back out HTML.

These days, we talk a lot about building web apps and having very rich web clients, but that's a pretty new phenomenon, especially in 2012 and before. Only small amounts of JavaScript were sprinkled on the page to add interactivity. Most business logic across most applications, including Facebook, were on the server.

Moving to mobile apps really shifted that equation, where all of the business logic had to be on the client in order to be snappy and responsive. But that also meant that building all these products meant a many, many megabyte sized file. Knowing that people would need to download all these of megabytes in order to upgrade their applications alone was a constraint. There is a constraint that both iOS and Android place on just the maximum possible size of a binary that it will load into memory to run, which we faced multiple times.

There were limitations on the Android platform in particular about how many different classes and class methods that you could have in the built application. We actually had to build brand-new technology from the ground up on top of the Android platform to allow us to go past those limits. So, we sort of constantly rode the edge of what the biggest application could be. In fact, I remember a conversation with an Apple app reviewer who is reviewing the Facebook

application who had some insight into a lot of applications across the market and let us know that Facebook was not only the biggest app in the market. It was bigger than any internal apps at Apple, and that included Launchpad, or springboard. Sorry. Springboard, which is the actual – Sort of the equivalent to the Finder on the Mac. It's the thing that runs – It's the UI for iOS, which is super sophisticated, and the Facebook app is actually even bigger than that. So, we were really getting Apple some – The other side of constraint, and they ended up actually building little bits and pieces of iOS to help the Facebook app load correctly. So, those were constraints.

Then as we talk about the network, we felt little pieces of this as we moved to mobile web. But when we're on desktop browsers, you're on Wi-Fi, or you might even have a hard line connection to the internet. Even on pretty crappy Wi-Fi, you're still talking about you're going to hop from your laptop to the Wi-Fi hub within 100 yards. Then that's got a wired connection to the internet even if it's a crappy one. Your latency is going to be decent and your speeds are going to be okay. When we moved to mobile web, we started to interact with 3G networks.

3G networks have pretty poor upload and download speeds, but really the pain point is latency. When you want to make a connection, the latency can often be over an entire second, and that's for a single round trip. When you're talking about the network, you have to think about the actual network protocols that are going on.

Facebook, at that time was just nearing the end of a transition to HTTPS, and that requires a TSL handshake, which has multiple round trips back and forth, and that can alone take multiple seconds. So, you're talking about in mediocre or poor conditions a page load time of 15, 20, 35, 40 seconds in the worst possible cases, and that's just for loading a webpage, where you just need to download HTML, maybe a couple of JavaScript files.

For mobile applications, we saw that performance get even worse, because as we shifted business logic from the server to the client, that meant that data needs, which previously were entirely encapsulated on the server where the business logic would say, "Okay. I need a user object. Okay. I need a newsfeed object. Okay. Now, I need a photo object."

All of those transactions happened on the server where they were very fast. But when you move that business logic from the server to the client, now those data accesses have to go over the

3G network. Each one of those is a round trip to grab a particular object. Now we're not just talking about loading an HTML file and a couple of JavaScript files. We're talking about loading every single object that you need in order to draw the screen, which in some cases is thousands, if not hundreds of objects, which can add up to be extremely poor on the network.

So, some of our early prototypes of that native iOS app before we launched it, even just walking around parts of the Facebook campus in Menlo Park, California, we were seeing well over 45-second load times for pages. It's just abysmal. So, we really had to look into how to optimize our use of the network to reduce those things.

[00:34:01] JM: And that study of the mobile infrastructure led to the projects that became GraphQL. Can you explain how these issues with mobile infrastructure led to GraphQL?

[00:34:14] LB: Yeah. GraphQL definitely came out of exactly this problem and a couple other problems. Really, this was a secondary problem that GraphQL is solving. The other problem that it was solving was since all of the data access previously was done on the server, all the APIs for doing those data access were PHP APIs. They weren't internet APIs. They weren't like REST.

So, when it came time to build the equivalent product as an all client-side business logic iOS and Android app, we realized that all of the tools that we were used to using as an engineering organization were no longer available. In fact, it was kind of even more interesting than that, because the set of people who were really building this new iOS app were really new to Facebook, because like I was talking about before, we didn't have that many iOS engineers. So, we got started on this project after having a couple really key hires during the company. So we had a lot of folks who were pretty new to Facebook. Really, we're giving a lot a license to operate in a bubble and just work as quickly as they could. So when it came time to finding data, they just kind of looked around to see what they could find, and they've found themselves a newsfeed API.

I remember when they came to me with the first prototype, and it worked pretty well. It loaded decently fast. It scrolled smoothly, but it was missing newsfeed stories. As I compared it to my newsfeed on my laptop, there is clearly some pieces missing.

So, when I started asking questions digging into what was going on, they're saying, "Well, we're actually rendering all of the newsfeed stories that the API is returning," and that's when I sort of turned flushed white for a moment. I was like, "Wait a minute. What API? Oh, right! Where are you getting the data to load these stores? I didn't even think of that."

Then they pointed me to this newsfeed API that we had built for external business partners years earlier, I think in 2009, and had really kind of abandoned. It still works. The business immigrations that we had built were still being used to some degree, but no one had really touched it since then, which meant that it was limited to the features of the 2009 version of newsfeed, and Facebook in that time was going through dramatic product improvements to all of its products, newsfeed in particular. So, by 2012, there was a ton of new capabilities in newsfeed that that API just this didn't quite capture.

First of all, I was kind of surprised that it even works at all. But, really, we had to think about what would an API look like that actually did everything that newsfeed needed to do and sort of fit the model of this over the internet API. We realized really quickly that a REST API wouldn't work. We kind of tried to figure out how to do that. The critical problem that we faced was newsfeed stories can be arbitrarily complex and arbitrarily nested.

One of the features that we had launched relatively recently was aggregated stories. So we would say three of your friends are talking about whatever, and then there'd be within that story three more stories, or so-and-so commented on so-and-so's post, which is a story about a story. Those things can be arbitrarily deep.

So, if in a REST model, you would imagine loading just the data object for a newsfeed story, which then might have a URL reference to another newsfeed story and you would need to do that recursively. Because the network conditions are pretty poor, that meant that you can't start loading the nested story until the original story has completed its whole sort of journey through the network, which means that if you have a story that has no one newsfeed story, which wraps another newsfeed story, which wraps a news article, then you're talking about potentially three stacked round trips to the network, which could add up to be multiple seconds just to load that one piece of information.

So, we really wanted to look for a way to get all of the information that we would need for a particular story in a single round trip recognizing that that nested story might actually appear later in the newsfeed, because you might see – Scroll way down and you see the original post, and then you scroll up in the newsfeed and you see a story about somebody engaging with that post. That caused us to sort of backup a bit and think about the data model for newsfeeds.

So, we did this exercise as we are trying to explore different API options. We explored Thrift, which was a technology originally built at Facebook that it's now an Apache project. So we wrote a Thrift definition for all the different data objects in newsfeed. We also looked at FQL, which is a Facebook API technology that had been built the year prior. That allows you to have a SQL-like interface to objects.

We actually built a version of Facebook newsfeed that used FQL, and it was somewhat successful. It allowed us to describe all the data that we needed in a SQL expression and load that in a single round trip from the network. But the problem was that the SQL expression was extremely complicated. Describing recursive joints is already a pretty challenging thing to do in SQL, and that's just one of many joins that we needed to do across the people writing the post, the newsfeeds in the post, potential attachments, like photos, and events, and groups in the post, and events and groups could have their own sort of metadata attached to those.

So, the SQL expression that we were using ended up being I think like many hundreds, if not well over a thousand lines and was really hard to read and hard to maintain. So it was kind of a dead end as well. So, that lead us on sort of trying to figure out what to do next. One of my colleagues at Facebook, Nick Schrock, was partly responsible for having built FQL. He was also the critical person responsible for these PHP APIs for data access and was aware of sort of some of the problems that we are facing in mobile and APIs, and had actually built this prototype. He called the prototype Supergraph, and it was – The very first form of it was almost akin to taking like literal PHP code and writing it like a string on a mobile app and then sending that string up to the server.

Now, it definitely wasn't just about [inaudible 00:40:32] PHP code. That would be dangerous, but we're taking a lot of inspiration from what the actual underlying PHP data access API look like

and try to figure out what this simplest version of that syntax could be on client, and that was really interesting. It borrowed from this idea of using syntax to do an API request from FQL. Rather than using SQL established language, we refer now in the space of exploring our own language. I thought that was a super interesting idea. I was really excited to help on that. So, I took the Thrift definitions that I had been working on and then worked with Nick on this prototype, and then because we're working on newsfeed particular, we needed in newsfeed expert. So, the third co-creator of GraphQL is Dan Schafer, who was really the most broadly knowledgeable newsfeed engineer at the time who was excited about helping us with this as well.

So, Dan made sure that all of the newsfeed APIs fit the sort of internal data access model that we needed, and Nick helped expand his prototype, and then I helped make sure that that prototype expanded to actually match the needs that their mobile app had. So, all that work together sort of in a very rapid couple of months turned into the first version of GraphQL that helped solve both of those two problems, both how to load all the data that you needed over the network to avoid all the latency and slow network problems while also sort of describing this like multi-nested join and recursive data structure that was the super complex newsfeed.

[SPONSOR MESSAGE]

[00:42:13] JM: When I'm building a new product, G2i is the company that I call on to help me find a developer who can build the first version of my product. G2i is a hiring platform run by engineers that matches you with React, React Native, GraphQL and mobile engineers who you can trust. Whether you are a new company building your first product, like me, or an established company that wants additional engineering help, G2i has the talent that you need to accomplish your goals.

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

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

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

[INTERVIEW CONTINUED]

[00:44:05] JM: Facebook was at the leading edge of dealing with some of these problems, but it was far from the only company. Amazon was dealing with these problems. Netflix was dealing with these problems. Basically, any company that was trying to make the shift to mobile and deal with some of the other trends that you mentioned were dealing with this problem, and they all were able to see. Eventually that GraphQL was a great style of solution, and that's partly because Facebook started releasing big open source projects, like GraphQL. GraphQL was open sourced around the same time as React and some other open source projects, and there was a real push to have open source be part of Facebook strategy. Why did Facebook start releasing big open source projects?

[00:44:56] LB: There's a couple of reasons, and really it boils down to a couple of key people within Facebook who believed in it. So, Facebook's actually long had a relationship with open source way back to its early, early days. Mark Zuckerberg spent a bunch of time contributing to Memcached in 2004 and 2005 when trying to scale Facebook to, at that point, millions of users, and Memcached needed to sort of change in important ways to support that.

So, even from the very beginning, we're doing that. We also sort of have this internal recognition that a lot of companies, they're built on the technology that's popular at the time, and then they sort of perpetuate that technology. So, at the time in 2004 when Facebook was getting put together by Mark, the LAMP stack was the best way to build something quickly, Linux, Apache,

MySQL and PHP, and that's exactly the set of technologies that make Facebook work. Even today, variations on the LAMP stack as sort of the core architecture of Facebook.

I think it's really important to the remember that all four of those things are open source projects. So, companies including Facebook wouldn't exist without open source. So, in many ways, we always sort of felt this moral obligation to give back, because Facebook wouldn't exist without open source.

So, in the early years, we had the balance contributing to open source with other internal priorities. When you're a small company, it's hard to turn to open source and spend a lot of time, because it really does take quite a lot of effort. But there were plenty of effort, probably projects that were really open source projects even sort of predating GitHub, including Thrift. Thrift was open sourced, eventually became part of the Apache Foundation including actually a lot of the work that went into that very early version of iOS.

It was called the 320 Library, and a lot of really iOS engineers actually use the 320 Library that was a Facebook open source projects, and that was in 2007 and 2008. So, really dating back early on, open source has been an important piece of Facebook engineering. But I think as Facebook started to get big enough that rather than just focusing on the product problems in front of it, they could start looking at slightly bigger pictures, like what is the right way to build a good web application and what is the right way to make high-quality JavaScript UI? We were sort of starting to address these problems from first principles, because we had such a huge number of engineers at the company building products that any sort of incremental improvement to those problems would be really leveraged in the impact that would happen across all those engineers. So, that's when we started investing in a lot of the projects that would ultimately become open source.

I think another important piece – There're certainly open source projects across all of Facebook, but I think a lot of the folks in the frontend community in particular who see open source, important open source projects coming out of Facebook, those open source projects were all coming out of one particular team. That team was run by Adam Wolf. It called the product infrastructure team, and that team was really charged with coming up with tools, technology and patterns that would allow us to build higher quality products faster. That's the team that is

responsible for React, GraphQL, yarn. It was the team that contributed in on Babel and when Babel became temporarily a Facebook project when Sebastian joined it. It was part of product infrastructure.

So, really, that set of people is not particularly big. But because of sort of cultural leadership from Adam, who always supported any effort to open source things, we really kind of got behind that. I think a lot of the open source efforts at Facebook were not always well received or not always paid attention to even until React. I think React was the inflection point. A lot of people looked at Facebook engineering as a little bit of a clown show, especially after we didn't necessarily set the best reputation for ourselves with our foray into mobile.

The best engineers in the industry didn't see Facebook as the holder of the all the best ideas. In fact, we often got laughed at in conferences, sometimes for a good reason. So, when we launched React, React went through its own sort of interesting first couple years within Facebook before it became an open source projects. But when it was open sourced in 2014 or 2015, it was not well received at all. Early criticism of JSX as a syntax, complaints that it was billing itself as reactive programming, but it was a poor man's version of that, if that all, and just pointing out that there's no way that it could be fast and it’s a bad idea and it should be ignored.

Maybe a year later, people started coming around. They started to realize once they used it, the positive qualities that were there. I think you know now the rest is history. React has left a really important mark on product engineering even beyond JavaScript now, because its impact on the way people think about iOS and Android product engineering as well.

We saw a ton of extremely positive things happen because of the open source effort overtime. Most notably was the shift in the perception of Facebook engineering as an organization, where before people really thought of Facebook engineering as a bunch of clowns who didn't know what they were doing. Now they said, “Hey! Well, if they're smart enough to come up with React, maybe we should pay attention,” and that’s when a lot of the other Facebook open source projects started to get a lot more notoriety and adaptability and we really started to attract a lot of extremely smart people.

I think it was a year or two later that one of the folks who's working on open sources at Facebook decided to run a poll and she polled all of engineers that had joined Facebook in the previous six months and asked them if they knew about Facebook open source before they joined the company. If Facebook open source was a significant factor and their decision to join the company. The answer to both of those two questions were sort of off the charts, way higher than we predicted.

It turned out that like well over 80% of new Facebook engineering members not even within the frontend team just like across all of Facebook. Well over 80% were aware of Facebook open source projects. Well over 60%, close to 70%, marked Facebook open source as an important reason for why they joined the company.

So, we started to realize that, wow, not only was React having great impact on the frontend community, but it was having this outsized impact on Facebook's ability to recruit, to attract great talent, to retain talent and to establish itself as a well-known brand.

So, once we saw that, we started looking at other projects that lined up well between sort of three important things. One being that the team was excited about open sourcing it. They thought that it had a good fit with the community. That we felt that there was clear overlap between internal priorities and community priorities. In other words, that if we were to continue working on things that were important to Facebook, that it would have tangential positive impact to the community as well.

Then further it is that it would tie back clearly to this recruiting angle, that by doing it, it would have a positive impact on Facebook’s engineering reputation and ability to attract good people. I think that ended up playing out extremely well in a lot of projects, including GraphQl. I think GraphQL actually succeeded well beyond what we thought it would, including yarn, ingests and a ton of other really important projects.

[00:52:36] JM: The change to an emphasis on mobile is a shift in the market that could have killed the company if Facebook wasn't paying attention to that shift in user preferences. Facebook does seem notable for its ability to detect these fundamental inflection points in how

the market wants to use its product. Are there any other key inflection points that you remember from your time at Facebook as being formative to your experience as an engineer there?

[00:53:10] LB: think that was the most important one. In fact, I just want to reemphasize what you said and just – Not only was it a threat to the company. It was the threat to the company. In fact, now in hindsight, it's kind of insane that three of us who are focusing on this crazy bet on GraphQL right at the critical moment where Facebook was in the middle of figuring out the IPO, we realized mobile was totally messed up and probably the right management decision would be to take a proven technology off-the-shelf and run as fast as possible with it, and yet they trusted us and allowed us to dig in and figure out what the right technology was even if it costs us a couple extra months on the path.

I think that played out really well. GraphQL ended up serving Facebook for years and still does, but it's really true. That shifts between platforms has killed other companies and it came very close to killing Facebook, and I think we pulled it around just in the nick of time. I think other inflection points that have been really sort of formative to Facebook that I've witnessed are the shift to sort of a multi-app platform.

This initial shift from a web-based app to a native app really still focused on Facebook being a single icon on the screen, a single destination. Then once you open that, then you are in the land of Facebook and you could do things.

There are some relatively early pressure to break that the pieces to have – I think the most successful version of that was the separate Messenger app, and there was a lot of tough internal conversation about whether that was the right thing to do. People were nervous that if you would move the app out, the just raw fire hose of attention that newsfeed got strongly benefited all these other products and people were concerned that they wouldn’t be able to stand on their own. But we saw this shift coming of applications, especially mobile applications feeling like lightweight single-focused utilities and started to shift in that direction.

Some cases actually shift a little bit too late. I think in one case, we were working on folding photos out of the app, and there was this relatively short-lived app called Facebook Camera,

which was really a competitive take against Instagram. It did okay, but not super well, and that was part of the impetus for the Instagram acquisition.

As I remember, it was this realization that Facebook Camera was just getting reamed, or at least it looked like it was possible in the future that it would really get creamed. The same goes with sort of the other sort of suite of mobile applications companies that Facebook acquired over the years. That's really certain sort of that second shift on mobile to sort of single-focused applications.

There have been a couple of other ones over the years, but they're really more about technology shifts rather than market shifts, but those are sort of critical moments and it's always impressive in retrospect when you see that it kind of worked out for the best.

[00:56:07] JM: Lee Byron, thank you for coming on Software Engineering Daily. It's been really fun talking to you.

[00:56:10] LB: Absolutely. It's been my pleasure.

[END OF INTERVIEW]

[00:56:15] JM: GoCD is a continuous delivery tool from ThoughtWorks. If you have heard about continuous delivery, but you don't know what it looks like in action, try the GoCD Test Drive at gocd.org/sedaily. GoCD's Test Drive will set up example pipelines for you to see how GoCD manages your continuous delivery workflows. Visualize your deployment pipelines and understand which tests are passing in which tests are failing. Continuous delivery helps you release your software faster and more reliably. Check out GoCD by going to gocd.org/sedaily and try out GoCD to get continuous delivery for your next project.

[END]