EPISODE 1174

[INTRODUCTION]

[00:00:00] JM: GitHub manages a large API surface for both internal and external developers. This API surface has been migrated from purely RESTful requests to GraphQL. GraphQL is a newer request language for data fetching and fewer round trips.

Marc-Andre Giroux works at GitHub and is the author of Production Ready GraphQL. He joins the show to talk about GraphQL across the industry and specifically at GitHub.

[INTERVIEW]

[00:00:32] JM: Marc, welcome to the show.

[00:00:34] MAG: Thank you so much for having me. Big fan of the show.

[00:00:36] JM: Great to hear. Everybody listening probably has an idea of what GitHub is, but people may not know the API surface area of GitHub. People can build external apps on top of the API. Give me an idea of the API surface area.

[00:00:50] MAG: Right. So the API, you can kind of think of it as just another way to access GitHub's domain. So anything people usually deal with with the UIs, like creating issues, creating pull requests, you can usually do that with the API as well. The difference is kind of like who's interested in using it. So multiple companies, their whole development workflow works around GitHub and the API is the primary way to access that domain in just a programmatic way.

[00:01:22] JM: So give me a few examples of those API endpoints.

[00:01:25] MAG: Yup. So you've got endpoints from anything to getting an issue, getting metadata for your issue. We've got API straight to Git itself. So creating Git trees, Git objects, and we have things that are a bit more high-level, like our new actions workflows, which are controllable through the API as well.

[00:01:45] JM: And give me some sense of the scale of these different API endpoints.

[00:01:49] MAG: Yeah. So GitHub's API scale is actually quite something. We've got so many companies depending on that API for development workflows. So we've got actually over a billion request data the API serves. We've got into the hundreds of thousands of integrations. So different apps or just clients or even tokens accessing the API. So it's a huge scale. It's a ton of different use cases to support and tons of different clients that have kind of various use cases in mind to support. So it's quite a challenge.

[00:02:25] JM: And maybe you could take me through the life cycle of one particular endpoint just to give an example.

[00:02:31] MAG: Right. So let's take for example just creating a pull request, given a branch. So there's multiple ways to do that with GitHub’s API right now. You can do that with our REST API by emitting a post request to a pull request endpoint. And you can do that with our GraphQL API now by doing a mutation against our create pull request mutation. So is that kind of what you had in mind as a workflow?

[00:02:57] JM: Yes. Why don't you pick one of those and walk through the life cycle?

[00:03:02] MAG: Did you mean kind of like the client life cycle here or –

[00:03:06] JM: Well, the server side life cycle. So like I make a request and then what happens on the server side?

[00:03:11] MAG: Right. So an API request always goes through our load balancer, which is called GLB internally. That load balancer will basically hit our Rails monolith. So GitHub is this a huge Rails monolith that's kind of famous for one of the biggest Rails apps really. And that Rails app serves everything related to the API right now. So if it's the REST API or the GraphQL API, this is a Ruby monolith that's handling the request. We've got these controllers that – The special thing about it is that the API has kind of like a separate little codebase within that monolith that serves these requests. So in terms of REST, we've got a nice way for our developers to define endpoints that has everything that a well-defined GitHub API should have.

So permissions are set at as DSL description where the documentation URL is. So a developer – Really, what our goal is as the API team is for a team to only define their business logic. So whatever the domain is, creating a pull request for example, and we've got the tooling around it what is executed during a workflow like this. So rate limits, middlewares, authorization, authentication all taken care of.

[00:04:27] JM: Tell me a little bit more about what happens at the load balancing layer.

[00:04:31] MAG: Yup. The load balancer level doesn't do much at the moment. It has a little bit of rate limits when it comes to anonymous requests, but a lot of logic is found in our monolith running Ruby. So that little balancer is quite helpful to kind of protect our Ruby processes. But anything that goes in with the API itself is usually right in the monolith.

[00:04:56] JM: What are the canonical problems that come from maintaining an API surface of this scale?

[00:05:01] MAG: So there's two problems. There's the user scale. So how many requests we're getting? Serving requests that are performant. And then the other scale problem is this internal scale, our developer scale. So the developer scale is actually sometimes even more challenging than the amount of users consuming our API. So we've got hundreds of developers working on different features. They all want to ship their domain not only in the UI, but in the API. So building APIs that can serve that many users, but also serving that many developers is quite a challenge. That's why we focus a lot on making the experience as simple as possible building an experience that's consistent across endpoints when you have dozens of different teams building them. This is where we invest them over. So in terms of user scale, it's also something we focus on obviously, but the developer scale internally is something very challenging as well.

[00:06:02] JM: There's a REST API and GraphQL API. Could you differentiate between those two APIs and how the interaction pattern differs between those two?

[00:06:12] MAG: Right. So that's also a challenge we have in terms of scale, because we trade our GraphQL API and a REST API as basically just two different client experiences to the same business domain. So in theory, anything you could be able to do with a UI, we would want to be accessible through our REST interface and GraphQL interface. In REST, that's kind of like what most people are used to. You consume our domain through a set of resources and manipulate them through HTTP methods. With GraphQL, you consume our domain with GraphQL queries. So it is a lot more flexible. Clients can kind of create their own use case from our exposed GraphQL schema, but it might not be as optimized or coarse-grained as REST.

Both patterns can be useful and we try to support both, but it's definitely a challenge there. Some clients who really need that flexibility will opt for GraphQL. Others who want to get started quickly and can do what are coarse-grained resources will opt for REST.

[00:07:18] JM: Were you involved in building out the GraphQL API?

[00:07:22] MAG: Yeah. So GitHub's GraphQL launched in 2016, I believe. And I joined around in 2017 to help build out a public GraphQL API. Before that I was at Shopify and we were also building one of the first public GraphQL APIs out there. So there's not a lot of public APIs out there for GraphQL just yet. It's a very popular API style for internal APIs, but it's um it's super interesting and quite challenging to go for that pattern for a public API like GitHub’s or Shopify's.

[00:07:56] JM: Can you tell me more about the experience of launching a large-scale GraphQL API?

[00:08:01] MAG: Yeah. So for GraphQL, the main challenge here is that it is a new technology and you can't expect every new client to know very well about it. With REST,HTTP has been around for enough time that a lot of people know how to consume a REST API. With GraphQL, there're a lot of benefits, but you can't assume that everybody knows how to consume it. So the big difference here is when launching a GraphQL API, not only do you have to describe what your API can do, what kind of workflows you can achieve, but there's also a bunch of education about GraphQL itself, which is a challenge. So we try to focus a lot on documentation about GraphQL in general, helping people getting started with building your first GraphQL query and we try to focus a lot on GraphQL strands, so the type schema, and using that to build a documentation where people can discover our use cases as easily as possible.

[00:08:59] JM: And tell me more about how the difference between Shopify – And because you worked on the GraphQL API Shopify, how did that experience differ from working on a GitHub?

[00:09:07] MAG: So, ironically, they're pretty similar. Shopify also is a large Rails shop, a big Rails monolith. So a lot of the same technology were used, but the domain is very different. At Shopify things are oriented around a specific shop while GitHub's domain is a lot more interconnected. So you might have an issue that has an event that points to an issue in a completely other organization or repository. So the data access patterns are very, very different and making that work with GraphQL can be a challenge where everything is scoped to a single shop which can be sharded in its own database, for example. The data access patterns are manageable. With GitHub we have so many different connections between our types or resources that data fetching in a GraphQL world is quite challenging. So we've invested in that a lot.

[00:10:03] JM: How big is the team that work on the GraphQL API?

[00:10:06] MAG: We're a pretty small team. We're a four people team. The way we think about it though is that our team doesn't build every GraphQL type or every REST endpoint. So we're here to make sure our API is consistent. That our developers and engineers have the best tools possible to build GraphQL APIs. So that's what we focus on. So I like to think about it in a sense that we are the GraphQL experts. We can talk about how to design a good GraphQL API, but we're not the issue experts or Git experts or any other like domain experts. So I truly believe a great API, an API that's really well designed is kind of a mix between knowing the platform well. So in our case, GraphQL or REST, and knowing your domain really well to expose your use cases in the best way possible. So that's why we try to focus on. We focus on our strength building a great platform for our engineers to build on.

[00:11:06] JM: So you have a surface area of both internal and external APIs. How does the difference between the internal API and the public API management differ?

[00:11:16] MAG: In theory we'd like them to be quite similar. We'd like people to think about internal APIs just as if they were public APIs really, because a well-designed API is a well-designed API. In practice, they differ a bit. For example, in service to service communication or an internal API, you often know your use case very, very well and you have a small set of known customers. So in our case, different teams. That allows us to build APIs that are much more optimized, but maybe less flexible than our public API. And that's a big challenge.

GraphQL kind of aims to solve that bridge, because we know consumers can kind of query for their exact use case. But the flexibility of GraphQL comes at a performance cost that for an internal API where you know exactly what you want and you can optimize it very well for a set of known consumers, we choose maybe something else or where a different technology or even a different design may be more proper.

[00:12:20] JM: What are some common design decisions you see or design mistakes you see people making in their creation of a GraphQL API?

[00:12:30] MAG: The biggest mistake I see is trying to convert an existing source of data into a GraphQL API. So that might be taking database tables and trying to convert them into a GraphQL schema or even taking an existing REST API and trying to convert it into a GraphQL API. That's all things that are possible. And quite frankly they're cool projects to work on, but the end result is rarely what your consumer will want to consume. Every platform has different design concerns. A REST API will be designed with something else in mind than a GraphQL API. So if you try to convert kind of like in a dumb way just one to one from REST to GraphQL or database to GraphQL you'll get often results that match your data, but don't match what your clients actually want to achieve from an API. So that's definitely the number one mistake we try to avoid. And I'd suggest anyone building an API to start from the client first, start from the use case first and then think about the data later.

[00:13:36] JM: What about maintenance of a large GraphQL API? What are the difficulties with maintaining something?

[00:13:44] MAG: Maintaining GraphQL API is challenging, again, kind of in terms of data fetching. Because there are so many ways to interact with a GraphQL API, so many paths to getting the same field, optimizing for all these paths is very hard and that's often shown through the most famous problems with GraphQL API is an N+1 problem where fetching certain queries causes N+1 queries at a database or a data source. And the way we try to avoid that is by building data abstractions that are kind of agnostic to how to query through the GraphQL API. There's this pattern called the data loader pattern that originates from Facebook actually with GraphQL. That kind of like has an interface on your data fetching layer that's all asynchronous or lazy if you want to say. So every time a GraphQL field in our API wants to access a resource, it goes through a data loader that kind of accumulates things to be loaded and resolves them asynchronously as a batch.

[00:14:51] JM: Any other longer term maintenance issues you've seen with large GraphQL APIs?

[00:14:57] MAG: The other issue I see a lot is just the evolution issue. So if you go on the GraphQL website today, you'll see that GraphQL says you don't need to version a GraphQL API. And there's some truth to that, because clients only select what they need. So adding fields or adding use cases doesn't impose an overhead to existing clients. The downside though is that you still need sometimes to remove or modify existing fields, and just like any other API, the solution, that is not always easy. So the way to avoid that as much as possible is first of all design things in a way that is future proof. And the other thing is having a good process for making changes to an API. So API evolution is a huge challenge for any API style, but GraphQL especially, because often people don't opt for versions, opt for a continuous evolution approach, making sure you understand how your API is used and making sure you design things in a future forward way often by, first of all, exposing things that are relevant to your domain, not the data, not the underlying data, is of great acuity. So I would say data fetching and evolving your graph are two of the biggest challenges.

[00:16:19] JM: You've actually written a book on Production Ready GraphQL. What did you learn when you were writing the book?

[00:16:26] SA: So I learned a ton, and this book is basically about everything I've learned at both Shopify and GitHub. And I think while writing this book, one thing that may sound funny to some people is that it's very clear GraphQL is here to stay, but it's very clear GraphQL is not for all use cases. So the book actually doesn't say everybody must be jumping on GraphQL right away. It actually talks quite honestly about GraphQL's tradeoffs, and that's the thing I thought the most about while writing it. And the things I discovered the most is that there are very clear tradeoffs where GraphQL's complexity and flexibility might not be needed, but it's also very needed in some cases in a tradeoff you might be able to accept. So I would say that's the thing I've learned most is how to think about GraphQL in kind of a pragmatic way in a nuanced way.

[00:17:22] JM: Did you spend a lot of time talking to other practitioners who have built a lot of GraphQL stuff?

[00:17:27] SA: Absolutely. So the book actually includes a few interviews I've done with other practitioners. And part of writing a book was also just my experience talking with so many people in the community. A great part of GraphQL is the community around it, the tooling around it, and we're all kind of in the same boat. So everybody is new to that technology especially when it comes to exposing a public API. So I've talked with a lot of people in that boat. So definitely there are great companies doing great things with GraphQL these days. Airbnb comes to mind, Uber comes to mind.

[00:18:05] JM: Not to ask a naive question, but what is advantageous about using GraphQL over Rest from your perspective?

[00:18:12] SA: So I like to answer the question in kind of two ways. The things that are inherently useful about GraphQL that we can start with. GraphQL makes it really easy to support a very large amount of different clients that have different use cases. The classic problems large internal APIs have is that every client wants its own kind of optimized version of an endpoint or a resource or any API that's provided. And over time that leads to the server team trying to kind of adapt to all these use cases and either creating different endpoints for everyone or creating extremely large resources that are kind of tailored to everyone, but not optimized for any single client.

So there're various solutions to that. The backend for front end pattern is one. Netflix did similar things back in 2012 I think with kind of server adapters to different clients' experiences. GraphQL is kind of like Facebook's approach to this problem, where you can in fact define your schema as a way to expose all server possibilities. But then clients are free to choose, pick and choose and create their own use case. I like to almost sometimes think about GraphQL in terms of clients creating server-side resources if the client was creating their endpoint on the backend. So that's kind of the inherently good part about GraphQL and large internal APIs.

Then the other part is kind of the more almost magical part of GraphQL where it just happened so that the community around it, the tooling around it and the fact that it comes with so many things bundled in, like a type schema, like the fact you can select smaller payloads, so like exactly the fields you want. These are all things you could do with other APIs as well, but they're less well-defined. And the GraphQL specification defines all of these good API practices into one. And I think that helped a lot of people just jump on GraphQL because you get all of this for free and you don't need to think about it. So these are not things that only GraphQL does, but it brings a really good package all together.

[00:20:34] JM: Tell me about your perspective on schema design and how that's applied to Github.

[00:20:40] MAG: Schema design is a thing I think a lot about and it's something really important. With a GraphQL API, often people kind of assume, "Hey, I've got a schema. I've got a GraphQL schema. My API is going to be really easy to use compared to my maybe ad hoc HTTP endpoints that I've defined in the past. But the fact is a GraphQL API is not magically well-designed, and thinking about API design is just as important as anything else. So like I was saying a bit earlier is thinking in terms of what your clients actually want to do is such an important part. And I do think GraphQL helps us think that way because it's so query-focused and so client-focused.

So a good way to do that is think about not what your schema should look like, but what your queries would look like and design the backend server with that in mind. And we do that by, yeah, basically just asking what is the client really wanting to do at this point. Does it really want to create a pull request record in the database or does it want something more friendly like merging a pull request or checking the status of a pull request or basically thinking in terms of workflows and use cases, not tables, not services, not anything internal?

[00:22:06] JM: What's the state of the art of tooling around GraphQL? So that's evolving every day, and that's something we invest a lot in. A lot of our tooling is kind of internal tooling so far, but there're a lot of things I think any GraphQL API should have in place. The first one is a schema linter. So a tool that helps you have a consistent API and an API that respects your API's kind of rules internally and make sure that's done on every change. So that's something we do by checking in the GraphQL schema in Git on every change and having these checks ran on every change. So we have this tool called GraphQL doctor, which analyzes every change to our GraphQL schema and emits recommendations or warnings depending on what the changes. For example, we can detect if someone is removing a field that's used heavily by our integrators. We can then warn them that this change is dangerous and we can even tell them how many times this field was used. So the schema itself kind of like leads itself to great tooling, because it's the interface to your use cases. So any tooling that helps you maintain a consistent API and maintain security and making sure your changes are safe is great.

The other part of tooling I think is really important is tooling that come really naturally with GraphQL is understanding how your schema is used. So if you think about a REST API, the server determines the shape of the response. And when it's sent to the client, you don't actually know which parts of that resource are being used. With GraphQL we know down to every single field argument, enum value. We know what the client has requested and we can assume it's using it. This lets us collect amazing data on how the API is used and use this data in our GraphQL doctor tool, for example, which knows what every single change, what the impact possibly is. We know new features already being used, already being liked. So this is a real power that's often forgotten about GraphQL, is you can know exactly how your API is used down to single leaves of the response.

[00:24:33] JM: What about maintaining a secure API surface area? Any suggestions on security around GraphQL?

[00:24:42] MAG: Yep. So one of the first thing I'll say is that GraphQL does appear less secure to an external eye, but it's not necessarily the fact. So I think security and API security shouldn't only be about the API layer itself. Authorization should be a big part of your domain logic, of your business logic, and no matter how you access it, it should be secure. There are still things you got to be careful of though with GraphQL. The main part is clients can actually send you the queries they want. So that's a benefit for them, but this can be dangerous for a GraphQL server especially a public one.

The way we address this again is by using the schema to analyze income inquiries. We use an analyzer that looks at incoming queries and computes a cost for each query. Instead of rate limiting, for example, a REST API where you would count an amount of requests per minute, we instead compute the complexity of queries and use that to block queries that we deem too expensive and we use that cost to rate limit clients as well.

The main thing we have to be careful about a GraphQL API is, yes, giving the flexibility to clients to design their own use case, but do that within secure bounds. And that's actually possible to do in a static way by analyzing the query against the schema. And there are folks, IBM research actually, that are working on research projects and actual scientific papers on how to analyze a cost for a GraphQL query in the best way possible. That's a big challenge.

[00:26:27] JM: What are you focused on at Github today?

[00:26:31] MAG: Today our main focus is trying to make sure all our use cases and the quality of APIs across REST, across GraphQL, and across our UI is consistent and easily accessible by everyone. So what we're working on is trying to make the REST API and the GraphQL two first-class citizens, just two different experiences for a client, but expose the same functionalities of GitHub at its core. So recently we've released an open API description for a REST API. Kind of bringing our REST API on par with GraphQL in terms of schema. On the GraphQL side, we still have work to do to bring some of these use cases that were possible in REST and not necessarily in GraphQL yet. So we're working on basically making that on equal ground and really treating both of these APIs as first-class citizens and allowing clients to pick the experience they need.

[00:27:32] JM: Do you have any advice for people who are doing large-scale migrations from a purely REST API surface area to GraphQL?

[00:27:42] MAG: I do. I think my first advice would be try to avoid tools that will try to automate this conversion for you as much as possible. So migrating from REST to GraphQL is often an excellent opportunity to kind of rethink a design you didn't like with your REST API and an excellent opportunity to just design things in a GraphQL-first kind of way and not that weird conversion flow we talked about earlier. So that would be my main advice. Try to take that as a new beginning and think of your design again in a GraphQL kind of way.

My other advice would be to extract any logic that's within your API layer at the REST layer. Extract that elsewhere from the API layer so that both your REST and GraphQL API call in in the same code pass. A very, very common mistake is having very important business logic stuck at that API layer. Commonly, that's found in kind of REST API controllers. And it's tempting for people migrating from GraphQL to REST to have GraphQL call in into the REST API, but that's not always great. And what we found is that not only is your codebase better if you do that if you extract your business logic into reusable logic by both REST and GraphQL, but you also don't get stuck with GraphQL types that oddly resemble your existing REST resources because you kind of leak that internal implementation detail that your GraphQL API calls in your REST API. So that would be my main advice. Isolate your business logic away from any API layer and try to avoid any kind of like automated conversion tools or at least verify what the result of these tools are and make sure they fit your design well.

[00:29:42] JM: Is it worth it to make that kind of migration? How would you judge the costs and benefits of making migration to GraphQL?

[00:29:53] MAG: It can be worth it, but that's definitely something to think about. I think if you have a single client and a single server and you've got a REST API that works well, I don't believe it's necessary to move to GraphQL. You will likely not notice very many benefits especially if you already use something like Open API or JSON schema and you've got typed resources already.

If you do feel the pains though where you've got multiple client teams asking for maybe an endpoint to have an extra field or an endpoint that's growing too large because it tries to do too much for too many clients, that's a point where it might be worth exploring GraphQL or just other strategies that allow evolution of an API server with multiple clients. In a one-to-one scenario where you've got one client that's already optimized with your API, it might be a bit less worth it to move to GraphQL.

The other interesting I'm noticing though if you have a public API is that client-side developers and mobile apps and frontends using React are starting to love integrating with GraphQL more and more. So something funny I've been noticing is that even if GraphQL might not be needed on a server-side, we're getting to a point where technology itself is so appreciated by clients that it may become a requirement for third-party APIs to implement.

So I would say think about the tradeoffs. GraphQL definitely comes with complexity. If you don't need the complexity because you don't have these problems, don't move. But it's interesting to think about the future where maybe clients will want and require GraphQL API because they love the experience so much.

[00:31:47] JM: Are there any other anti-patterns you see of people working with GraphQL?

[00:31:53] MAG: I think the biggest anti-pattern I see is using it in strange contexts where maybe the power of GraphQLs don't apply. I think the main one is public and static data. So if you have an API that exposes a list of countries, for example, do you really need the flexibility that GraphQL, GraphQL server engine and its overhead and a query language? Or would you rather just use HTTP for what it's good for and enable the power of HTTP caching and proxy caches for data that doesn't change often and is not authenticated so it can use caching at its full potential? It's possible to do caching with GraphQL. That's got a common misconception, but that's definitely not the sweet spot. So the biggest mistake I think is just jumping on GraphQL because it sounds like it's something you need to use or learn and not thinking if you actually need it. The public static data is one example I see often. And the other one that's debatable is when you have a single client and a single server and you're just getting started. The overhead of building a GraphQL server might be a little too much for some people.

[00:33:12] JM: So at Github, when you're standing up a new API, what's the process for making it externally available? Do you have to get a REST API in place? Get a GraphQL API in place? Have tests for everything? What's that process like?

[00:33:28] MAG: There're multiple stages. We like to encourage teams to start by literally opening an issue and think about what their API will want to do and what kind of design they have in mind. The next step is moving to the implementation stage where we provide kind of a lot of knobs to release an API maybe behind a feature flag. We have this thing called previews, which are kind of like beta features where people can onboard with a special header. So we've kind of have a progression from this endpoint is not available at all or this type is available behind this feature flag. It's available behind a public beta until it's full GA. And we have that through kind of our internal tools and DSLs where people standing up a new API can choose how to do so.

About REST and GraphQL, we kind of have these internal personas about what kind of API might be useful to one or both. Ideally, we would want any use case to be accessible through all of our interfaces. That's not exactly the case today and it's something we're working on right now and trying to make it way easier for teams to expose functionality everywhere.

[00:34:47] JM: Are there any major differences between how Shopify managed things and how Github manages things in terms of GraphQL?

[00:34:55] MAG: It's actually quite similar. The process is actually quite similar except I think the main difference here is that Shopify is kind of split into two APIs. They have the admin version of the API as kind of the back office and the storefront API. We don't really have that. So we do have an internal API and a public API, but we build it as one and annotates parts of the schema using kind of a tooling, again, in our DSLs we provide our engineers to annotate, "Is this field available publicly or is this part of an internal schema?"

And then at runtime we're kind of able to mask certain parts of the schema depending if you're using the API in an internal way or a public way. We actually do the same for anything that's a preview feature or feature flag. These are all designed as one schema internally, which simplifies kind of the cognitive overload for engineers. But at runtime, our kind of a GraphQL server is able to modify the schema a client might see depending on the permission. So do you have a feature flag? Are you an internal consumer of the schema? Do you have a preview-enabled? All these kinds of things. So the approach here is that we don't actually – Although they look like different APIs, they're built using kind of the same foundations and we annotate things to generate multiple APIs from one basically.

[00:36:25] JM: What's the process for deploying and rolling out updates to a GraphQL server at Github?

[00:36:33] MAG: So the deployment workflow is the same whether you're deploying a UI change, a GraphQL change or a REST API change. Our deployment flow is actually quite fun. It's based a lot on ChatOps. So to deploy a GraphQL change, you open a pull request with your changes. Our automation runs on the schema to make sure you're designing things in a correct way that you're not making a breaking change. And when it's your turn to deploy, you deploy using Slack. So we've got this deploy command where you're passing your PR. We deploy so much that we actually have a deploy queue that we call trains. You hop on one of these trains and you PR gets deployed automatically. So an engineer actually doesn't need to be involved into the details of a deploy. We basically interact through those very useful ChatOps.

[00:37:22] JM: When you're building out your external GraphQL API for Github, did you roll out everything all at once? Di you have to have everything ready before a big bang release for the external API or did you gradually roll out the API surface area?

[00:37:38] MAG: So right before I joined, Github released kind of a small beta GraphQL API, which contained quite a lot of things to be honest, but quite a small set of the whole Github functionality tree. So that was kind of a small bang, a small initial beta. But since then we've kind of been evolving it slowly by adding things. And the way we do things is by exposing functionality first internally. So often our UI or mobile app will be using features before it's in the public API. So that lets us kind of test and see if we did a good job with the design of the API. Then we might roll it to partners in a private way through feature flags. We had these preview features where we could expose it to anyone who wants to try it, but knowing it's a better feature. And then we release it fully.

So when somebody makes a GraphQL change, maybe adding a field or even a REST change actually, adding a new resource, it might not immediately be available in the public API. It goes kind of through this pipeline of confidence starting from internal only to available to everyone.

[00:38:50] JM: What do you see in the near future for the GraphQL ecosystem as a whole? Where are improvements coming and where are the most broad changes coming?

[00:39:00] MAG: I think what I'm most excited about is GraphQL being more of a mature player into the API landscape. So I'm excited for the time where there's less blogs about here's the difference between REST and GraphQL and why is GraphQL better. Or is REST better? And focus more on here's when you would use REST or when you will use GraphQL and what are the tradeoffs. And get into a place where everyone is in a place where the technology is mature enough where it has nothing to prove anymore. There's no more of that API fight between different styles and we can focus on the strengths of all the APIs.

For GraphQL itself, one thing I'm very excited about is just different performance improvements. The overhead of a GraphQL server where it has to execute everything the client requested, asked for, is not something to take for granted. So it does have an overhead for that flexibility. And I'm excited for anything new that would change how our GraphQL query would be executed. Right now most graphical servers execute GraphQL queries in a fairly naive way. And there are exciting things coming like pre-compiled queries and persistent queries that I think could change that game a lot.

The other part of it is observability of a GraphQL server. A graphical server can often be kind of a black box where you send in a query, something happens, and you get exactly the response you want. But how that query is computed, how that GraphQL engine computes it is quite important for performance reason and something you want to detect kind of slow nest in and it's not always easy out of the box right now. So that's something we also spend a lot of time internally on. We're making sure we understand how certain fields execute, but also the relationship of different query shapes giving different performance results. So that's a big challenge trying to optimize for all these use cases.

If I look at our GraphQL API and if I come back to what I was saying earlier where every GraphQL query is kind of a client constructed server endpoint, well, we serve so many different shapes of GraphQL queries that you can almost kind of think of it as if we were supporting millions of endpoints. So it's definitely a challenge to make sure that all these use cases execute in a performant way, in a consistent way.

[00:41:37] JM: Let's say I've got my GraphQL schema built and I want to evolve that schema over time, what are the best practices for evolving a GraphQL schema?

[00:41:47] MAG: So the state of the art is using a continuous evolution approach, meaning you've always got one version of your API running in production and you use deprecations to kind of warn clients about upcoming changes. So the great thing about GraphQL is it includes deprecations as a first-class citizen. So there's a deprecated directive you can apply to certain fields, values for example, to make sure clients are aware that an upcoming change is coming. So we use that heavily when wanting to make changes. We deprecate a field and start communicating that it's going to go away.

The key here though is to focus on changing things in an additive way always. So if I'm deprecating a field, it's because there's something else a client should use instead. If you deprecate something without an alternative, clients are not going to move away. So continuous evolution is great because clients don't need to hop from version to version and kind of grasp all the changes that are contained within versions, but it also comes with great responsibility where you have to communicate changes and offer alternatives that are great for clients.

One way we do that is by using a technique we call brownouts. So communicating changes are great, but you always have a long tail of clients that either don't care or didn't see your communications about a field going away. The way we try to reach those people is by using brownouts. And brownouts are basically periods where we'll disable that field as if we've removed it just for a minute or two hoping that the client systems using our API notice maybe errors or anything where they would be able to realize something's going on and then notice our communications.

So in order, I think, the first thing you want to do is deprecate your schema members that are going away. Start communicating these changes. And as I was saying earlier, with GraphQL, we've got the potential to know exactly who and how are they using our API. So you can track which clients would be affected by a certain deprecation and email them directly. Hide these fields from your docs so no new clients are getting on-boarded. And finally if that doesn't work, using brownouts to kind of wake up people who haven't seen the communications is a great way to do it.

I must say though that versioning GraphQL is possible even though if it's not a common approach. The folks at Shopify use kind of a filtering approach like we do at GitHub up for internal or public to create these versions, these calendar versions between GraphQL versions. And that's been working great for them. So I think they actually got a blog post about that. So common approach is use continuous evolution as long as you can with great communication, but versioning is possible even though that's not the most commonly used approach.

[00:44:58] JM: Mar, is there anything else you want to add about Github or GraphQL or just your thoughts on engineering in general?

[00:45:06] MAG: I think the only thing I would want to add here is that there is a lot of talk about GraphQL versus REST versus GRPC or whatever API style there is, and I think the important thing here to remember is that we're building APIs for users to build, to build things, to use features. And for them the API style is not always the most important thing. It's often the thing we like to think about as engineers building an API. But the reality here is that we have to pick the one that enables our clients to access the use cases in the ways they want. And I think no matter what the technology, designing a great API is similar, whether that's a code API, a Twirp, GRPC, GraphQL or REST, thinking about the client is key. And finally I think there's nuance in everything and carefully examining the tradeoffs. In our case, we've picked to support both, is really important.

[00:46:09] JM: Okay, Marc. Well, thanks for coming on the show. It's been a real pleasure talking to you.

[00:46:12] MAG: Thank you so much, Jeff.

[END]