From REST to GraphQL

Why a query language is perfect for writing APIs

by **@davidcelis** and **@gjtorikian**

How people build software

David: Hello, everyone! My name is David. I work on our Identity Management team.

Garen: And I'm Garen. I'm part of the Platform Interface team responsible for maintaining our APIs.

The year 2008
Welcome to Welp, Inc.
http://welp.reviews

How people review restaurants

@davidcelis and @gjtorikian

David: We're here to represent Welp, Inc. You know and love us, you use us whenever you think to yourself, "Welp… I'm hungry and indecisive and I have no idea what I'm doing." We've actually been using Welp a lot for the past couple days, as we come to you from opposite ends of the country. I'm here from Austin's beloved thieving sister city…
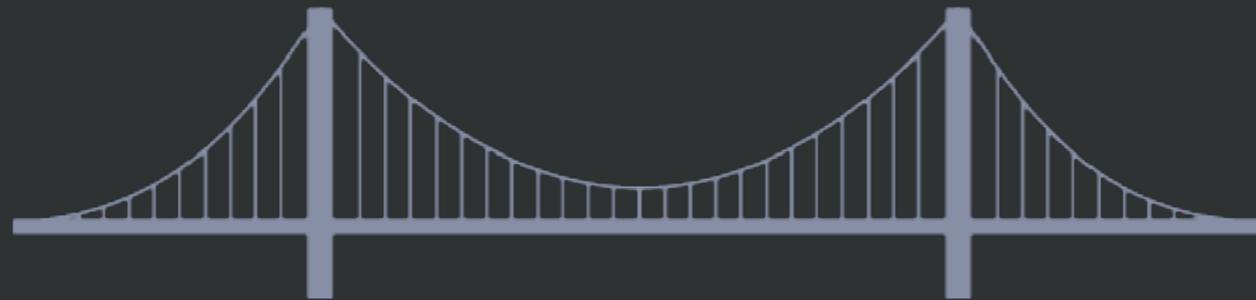
David: Portland. You might be thinking at this point, "Ohhh *that's* why he's wearing that beanie." but you'd be wrong. It's true that you see a lot of beanies around Portland, but in my case it's because I've been neglecting to get a haircut and it's at a really bad length right now.

Garen: And I lived in San Francisco for ten years…

Garen:… but I just moved to Brooklyn. They cost about the same. But the bridge is different! Mostly.

David: We should get back to Welp…

Garen: Okay. Imagine it's the year 2008! At that time, Welp, Inc. was just a humble start-up based out of a Palo Alto garage. We wanted to build a mobile app that can be showcased in Apple's brand new App Store for its brand new cellular phone, the iPhone 3G. In order to do that, we decided to build out a REST API.

David: REST APIs were super popular in 2008. I wanted us to be popular, so I had us build a REST API and hoped desperately that developers would use it too.

David: So we sat down at our favorite locally sourced hand roasted cafe and came up with a set of starting endpoints that we thought we'd need for the app. We want using the app to feel like using the website, only… snappier.

Garen: You can get a list of restaurants or a specific restaurant, you can create or edit your own reviews or tips. You can even remove them entirely!

Garen: A lot of thought had to go into just these endpoints, because we wouldn't be able to change them otherwise. We knew they'd probably be around to stay once people started using them.

David: Yeah, there were a lot of different angles to consider when building out these endpoints.

Garen: Like… What verbs make the most sense for each one? GET and DELETE are usually pretty easy… POST, too. But what about PUT?

David: And what if new verbs get added? *cough cough* Like, in 2008, PUT has always felt weird for updates. I could imagine a new verb getting introduced later for that. But we'll be stuck with PUT.

Which status code to

David: And what status codes should all of our endpoints respond with? There are a ton! There are like… over 500 status codes. I have them in front of me, but I don't have time to read the descriptions of all of them, so I'm just gonna skim through the names… Bear with me…

David: What's the deal with FORBIDDEN or UNAUTHORIZED? Those seem like synonyms to me. Do they *not* mean the same thing? If a user does a bad thing, do we tell them it's NOT ACCEPTABLE? Oh, and I've heard that some people return a 204 NO CONTENT when someone updates something, but other people return a regular 200 with the new representation of that thing in the body. And when are we a teapot?

How do we document it?

Garen: Speaking of reading through things, how are we going to document our API? We want our third-party users of the API to also make use of the API. Should we write code comments and have them parsed out?

Garen: Should we just write separate Markdown documents and hope we can keep them in sync with each other? What language should code samples be written in? Ruby? Python? Java?

```
# Yes. Definitely the friendliest way to consume any API
curl -X POST -H "Content-Type: application/json" \
  -d '{"name":"Mom's Diner","price":"$","primary_category_id":4}' \
  http://welp.reviews/api/v1/restaurants
```

David: I mean, definitely cURL. That's the best way to use any API. Yup, good ol' cURL.

How should hypermedia work?

How people build software   @davidcelis and @gjtorikian 13

David: We also want to be good Netizens and follow every REST guideline we can, so how about Hypermedia? Clients could navigate through resources more effectively if we added navigation links to resources' associations in our API responses. But how can we be sure we're modeling these relationships correctly? Should we just nest a bunch of data instead? Will our API clients even care about links?

How do we deprecate it?

@davidcelis and **@gjtorikian** 14

Garen: And even though Welp, Inc. is going to be around for years our API will almost definitely change during the lifecycle of our business. How can we deprecate endpoints as our platform evolves, or make other breaking changes?

David: Yeah like… I still just have this feeling that someone's gonna come up with a better verb than PUT for updates. How would we add support for something like that down the road?

How do we assemble a view?

How people build software

@**davidcelis** and @**gjtorikian** 15

David: And after bringing it all together, how will we use our API to build Welp's UI?

Garen: We don't want to use two different data fetching systems—in 2008, we have a system for rendering the website and now the API for the mobile app…

David: Yeah. I mean, if our API is gonna power the mobile app, we could use it to power the website too. But… The mobile app and the website are probably going to end up needing different pieces of data for their views, right?

**Garen:** Right….. Welp, let's just figure that out later.

Garen: Fast forward to today.

David: Welp has moved out of its garage and into a real brick-and-mortar glass building overlooking all of San Francisco. Everything the light touches is our domain.

Garen: We're a publicly traded company on the New York Stock Exchange (ticker symbol W E L P) and our REST API has seemingly kept up with our astronomic growth while we change the world, one review at a time.

David: Okay, but let's be real for a second. That kind of amazes me, because it hasn't all been good and rosy, you know. Our API is… kind of a mess.
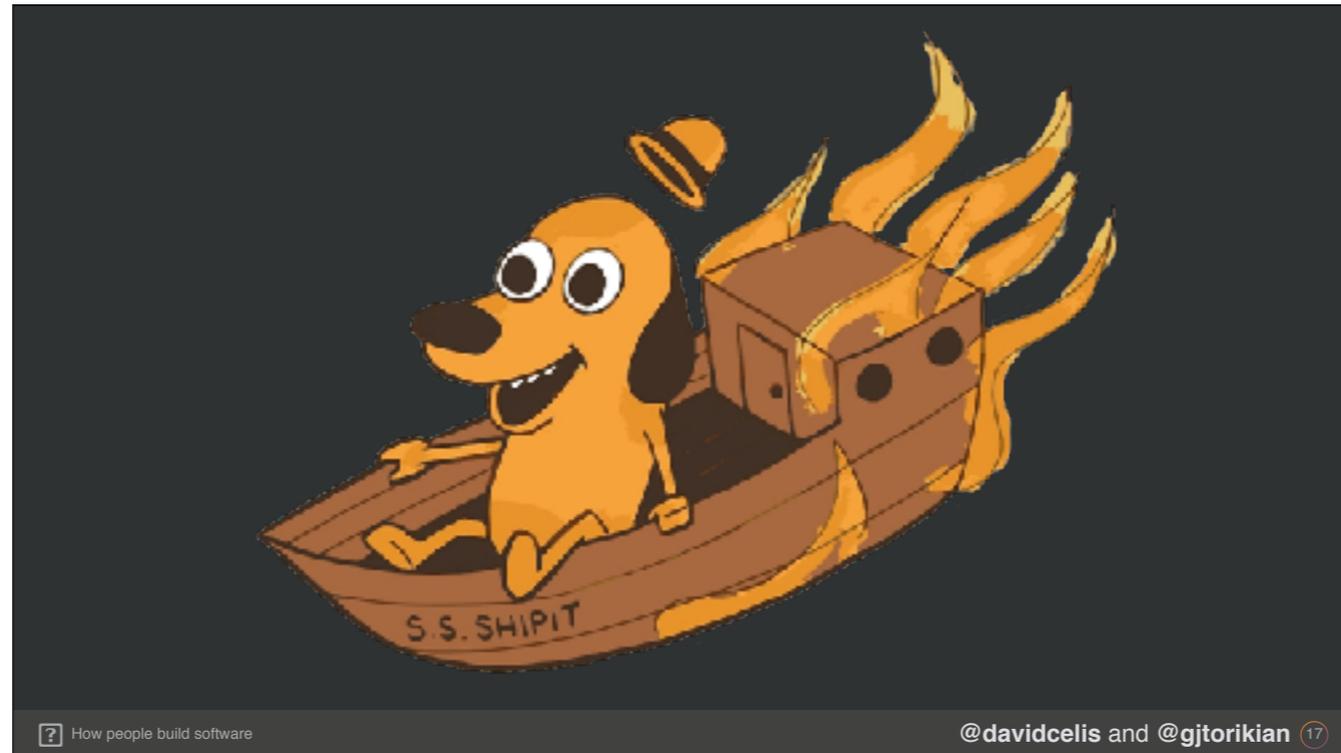
Garen: It's fine. Everything is fine.

Garen: Sure, some of our clients are still using the older endpoints, but that's nothing to worry about.

David: Dude, they're on v1. That's literally nine years old! I knew we'd probably have to keep supporting our old API versions for a while, but… Nine years?

Garen: I guess so.

Garen: *NEXT BULLET* And our mobile app—now visually optimized for the iPhone 7!—is preeetty slow. Our typical views have to hit a lot of API endpoints to get all of the data they need. We've managed to start fighting some of the slowness by building endpoints to cater specifically to our views…

Garen: *NEXT BULLET* … but they all cobble together completely unrelated data to render those views. It's not very RESTful.

David: *NEXT BULLET* Yeah, and with how quickly the team has grown, we've ended up with some pretty inconsistent practices when building endpoints. We should really get a style guide or something. Soon.

Garen: That would probably help with all of our data integrity issues, too.

Garen: *NEXT BULLET* Developers are sending us null values when they probably shouldn't be. People keep trying to pass in Strings when they ought to be using real Integers.

David: *NEXT BULLET* And we still haven't really gotten around to having well-maintained documentation. Our documentation is… not so good. But hey, at least we

Garen: Yeah, but clearly we've got some issues. What should we do about them for v5? Okay, all right, it's 2017. We could just rebuild our API from scratch using newer REST best practices.

Garen: We could incorporate Swagger to get some better documentation for our APIs…

Garen: *NEXT BULLET* we could use JSON schema to validate our inputs…

Garen: *NEXT BULLET* we could reach out to our integrators and ask them what data they are actually using so we know what's really important for our endpoints, and maybe—

David: Or we could try using something like GraphQL instead.
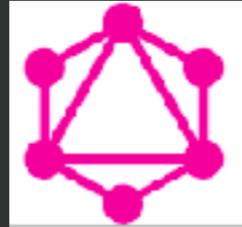
Garen: What?

David: GraphQL. It's this technology that's been making the rounds. It's cool. I started playing around with it after I read a blog post on Medium called "Give it a REST: Use GraphQL for your APIs". Pretty compelling thinkpiece. Here, have a TinyURL.

Garen: …Didn't you write tha--…

David: No.

Garen: Okay, I'll bite. What is GraphQL?

David: It's a technology that was written by Facebook when they were facing some similar issues that we're facing ourselves. It stands for Graph Query Language.

Garen: Query language? Wait, so you want to just… give our users access to our SQL database?

David: No! Nothing like that… Well, kind of like that. But it's safe, I swear. It's a layer on top of our database but we'd build it as part of our application code like anything else. We can even keep using our same ORM. Here let me show you what I've been playing around with.

David: OK so let's take a look at our public REST API. I have it up and running locally on my laptop.

David: *SWITCH TO TERMINAL* I'm gonna make a request to get a list of restaurants. As in most REST APIs, we make a simple GET request to the restaurants resource, and the server responds with an array of restaurants. All of this data is pre-determined by us in the API server based on what we think is important to the client.

Garen: It looks like we're basically returning everything. We think all of this data is important to every client?

David: Well, no… But we're our own main client, and this is what we need to render a single restaurant on the website, so we return everything all the time. But with GraphQL, the paradigm shifts. I've set up a few example queries to fire off to our GraphQL API. Now, imagine we're looking at a list of restaurants in our mobile app. We don't need many details

Garen: Certainly not all of the details we're returning in our REST API. The name, what kind of restaurant it is, its rating, maybe a photo. Anything else can be saved for when a user taps through to the restaurant's detailed view, right?

David: Exactly. So in our REST API, we'd get everything we didn't need anyway, right? But with GraphQL, the client can create a query that gives them only what they need. No more, no less. Let's take a look at that query.

David: *SWITCH TO ATOM* This is a simple GraphQL query. You won't be familiar with the syntax yet, but hopefully you can get a little bit of an idea of what's going on. I'll walk you through it too, of course! At the top, we declare what we're doing and state our operation as a query.

GraphQL: Open source gems

https://www.github.com/facebook/graphql	Open source spec
https://www.github.com/graphql/GraphiQL	Online GraphQL Editor
https://www.github.com/rmosolgo/graphql-ruby	Ruby implementation
https://www.github.com/Shopify/graphql-batch batcher	GraphQL database
https://www.github.com/github/graphql-client	Helpers for Rails views
https://www.github.com/gjtorikian/graphql-docs	Generate static HTML

How people build software

@davidcelis and @gjtorikian 22

Garen: Okay, in all seriousness now… Since this is OSCON, the best part about GraphQL is its massive participation in the open source community. REST is really a set of guidelines for how servers should behave. GraphQL has an open RFC that the web community participates in.

Garen: The GraphQL spec has implementations in a number of languages from JavaScript to Ruby to Scala and Rust. Here's a sampling of all the open source Ruby tooling that we use for our real jobs at GitHub. All of these projects can also be found on GitHub.
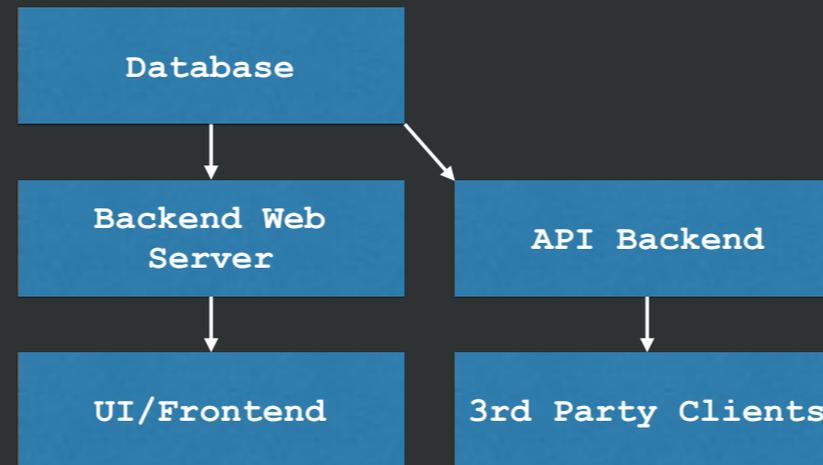
David: Another advantage to GraphQL is its reusability across a typical web stack. Even though this talk is focused primarily on usurping REST, it's just as important to talk about GraphQL's uses for rendering UIs, too.

David: In a typical web architecture, the logic to serve the user facing functionality is often separate from the logic used to serve the API's needs. This leads to separate codebases behaving in different ways.
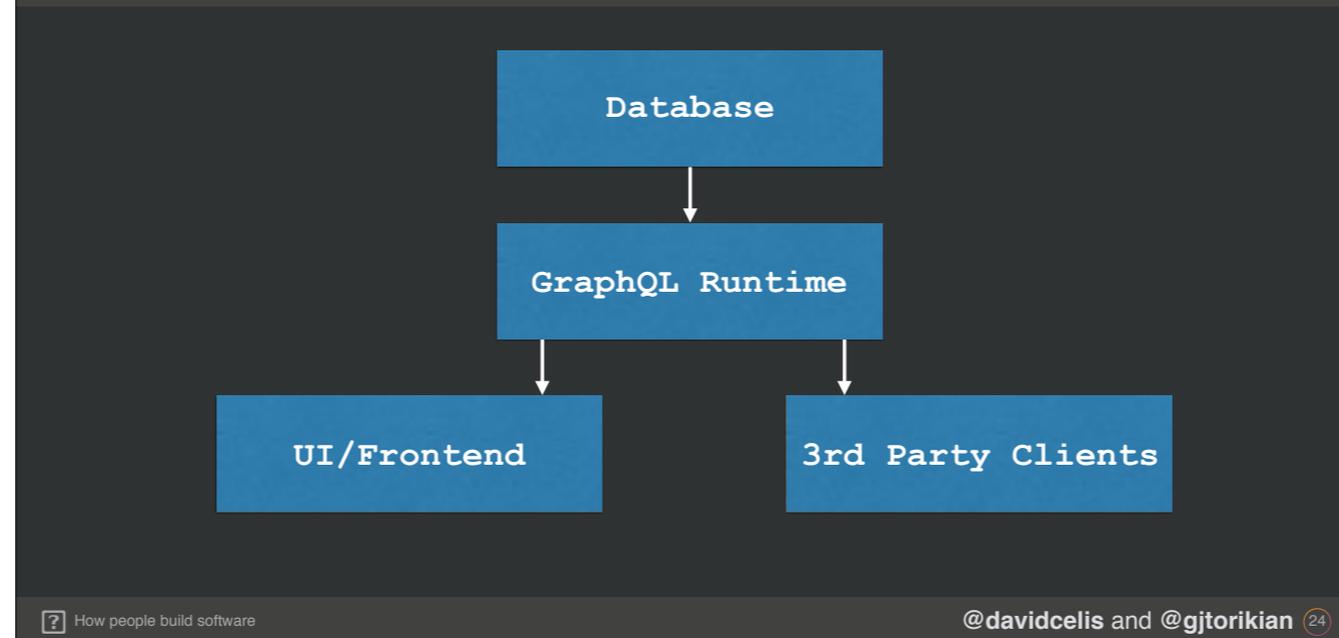
David: With GraphQL, these concerns are combined into one interface. The UI can make the same queries that a third-party client does, and can behave in a way that's consistent with how a third-party client might consume the data.
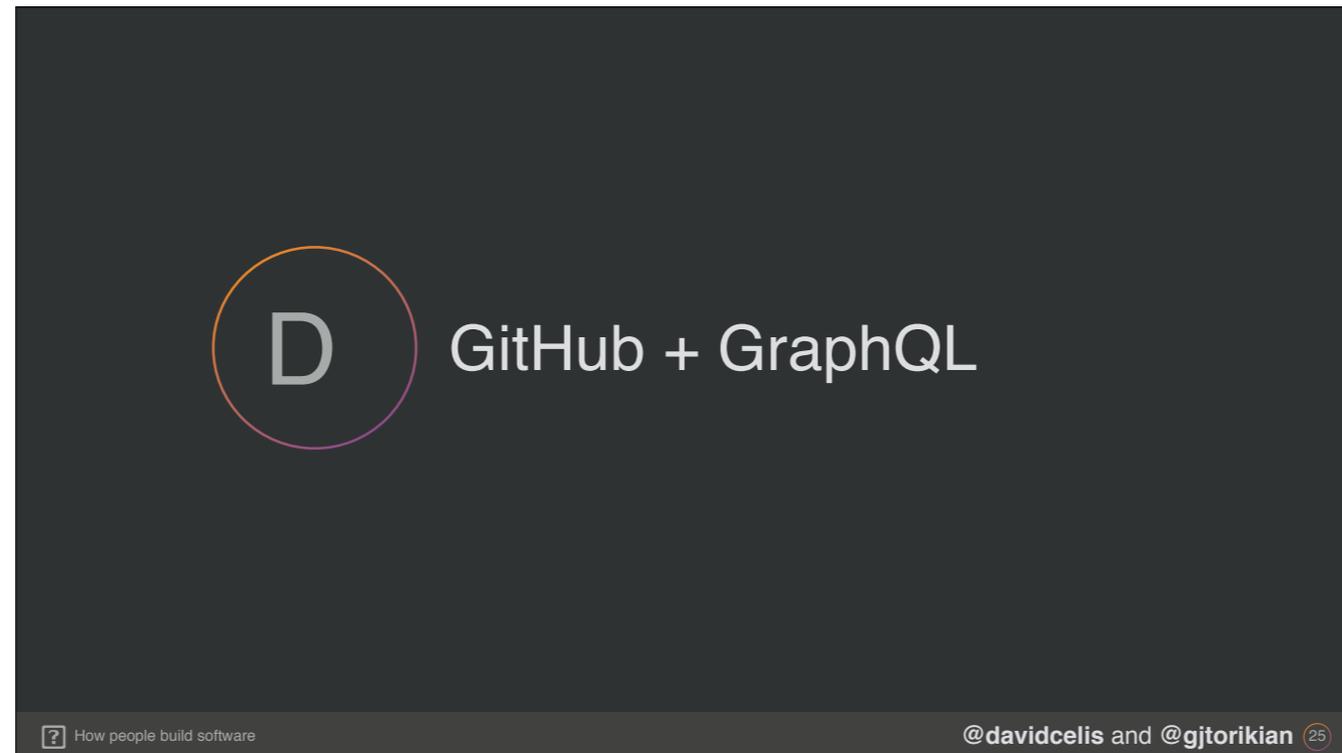
Garen: And since we're clearly done talking about Welp, Inc., we'd like to show you some of the ways that GitHub is migrating from REST to GraphQL.

Why GraphQL at GitHub?

· Get Integrators to parity with site UI

· Reduce our own inefficient stack

· More control over how our data is queried

How people build software

@**davidcelis** and @**gjtorikian** 26

David: When we started looking at adopting GraphQL at GitHub, we had a few goals in mind. First, we wanted our third-party integrators to have the same functionality as the rest of GitHub. We understand that all of GitHub's UI functionality may not satisfy the needs of every user, and that's fine. By moving over to building our UI with GraphQL, we could make sure that our API had the same capabilities as the rest of the site on the day a new feature shipped.

Garen: *NEXT BULLET* Our Rails and REST codebase was also getting to be pretty gnarly. After nine years as a monolithic Rails app, we wanted to start breaking up parts of the site into separate services. By using a single, consistent querying language like GraphQL, we could do this much more easily.

David: *NEXT BULLET* And finally, we wanted clients to be able to request the data they needed, rather than sending massive JSON blobs. This benefits both us and our users. First, it reduces the strain of querying and serving a bunch of data off of our servers. And second, because the payloads are scoped to what a user requested, it ensures that the client can get the data they need much quicker.
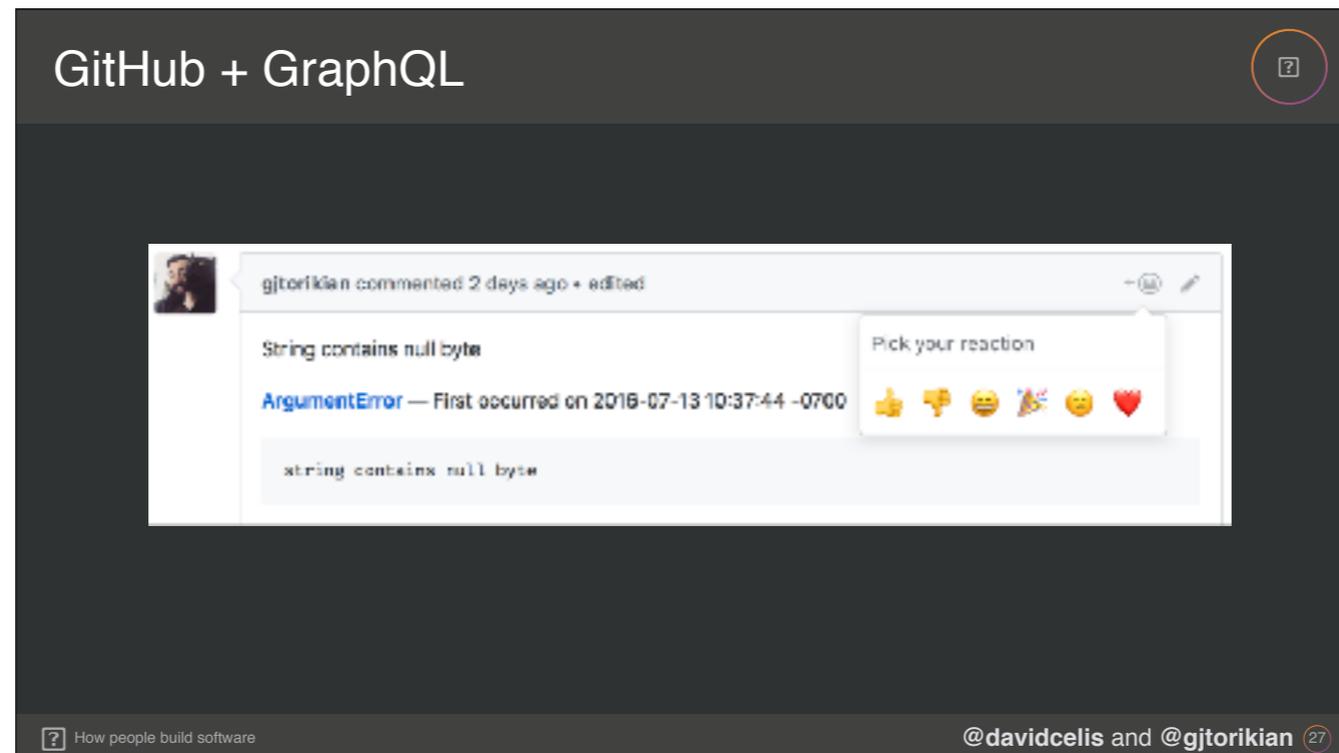
Garen: To start converting some of GitHub to GraphQL, we picked a relatively innocuous feature like reactions. Reactions was great both because they were new and because the feature was one of our simplest.

Garen: Despite the simplicity, it combined querying an issue comment to get a list of reactions, and some mutations in order to add and remove reactions. We could test the full range of GraphQL's capabilities in our one simple Reactions feature. We also figured that if we screwed up the process of converting it into GraphQL, it would be okay, since it wasn't a main component to using GitHub.

David: But did we screw it up?

Garen: Well… if a bug ships to production and no one notices, can you really call it a bug?

David: Fair.

David: Focusing on reactions also gave us a pretty good start for the entirety of the GraphQL schema. One option for us was to begin modeling our schema by starting with the Repository type.

David: However, as you can imagine, Repository is one of the biggest objects we have in our system. There are tons of pieces of data and a lot of associated types that hang off of a repository. Knowing where to go next would have been daunting.

David: Reactions, on the other hand… If you think about it, a reaction belongs to a comment, which belongs to an issue, which belongs to a repository. So instead of starting from a repository and working down to its various associations, we worked upwards, and built out only the GraphQL pieces that were necessary to power Reactions.

Garen: Since we were already rewriting the interface, one aspect that became apparent early on was that the way we had written the code previously was fairly inefficient. As we migrated more systems over to GraphQL, we also started to focus on and eliminate performance issues like N + 1 queries. As the project progressed, we began to add instrumentation to observe how our GraphQL resolvers were performing.

Garen: This graph here charts our top offenders over time. The y-axis represents the number of N + 1 calls made during a given day, so, the higher the number, the worse the performance.

Garen: To fix this, we made use of an open source Ruby gem from Shopify called graphql-batch. It provides an interface to batch all the queries that a GraphQL query requests, and then asynchronously resolves all of them. Here, you can see that we were able to drastically reduce, if not entirely eliminate, these excess DB calls, for both our UI and our API clients.

David: Another way that we began to build our GraphQL schema was to observe how integrators were making use of our REST API. Through years of communique, we knew that third-parties had to piece together multiple API calls to implement critical functionality.

David: For example, say you have a deployment system. In order to check on what's deployed to all of your repository's environments, here's a list of requests that system might need to make in order to check on the overall state of your deployments.

David: First, your client would need to get a listing of all the repositories that belong to your organization.

GitHub + GraphQL: Consolidation of actions

```
GET /orgs/:org/repos
  for :repo in [repos]
    GET /repos/:owner/:repo/deployments
      for :deployment in [deployments]
        GET /repos/:owner/:repo/deployments/:id/statuses
```

How people build software

@davidcelis and **@gjtorikian** 32

David: Then, for each of those repositories, your client would need to get a list of the repository's deployments.

David: Then, for each of those deployments, your client needs to make a request to get a list of the deployment's statuses. It's exponentially inefficient.

Garen: Instead, with GraphQL, we began building relationships like these slowly. Since we had added repositories as part of our work on building reactions, we moved onto building deployments off of repositories, and then deployment statuses off of deployments.

Garen: Here, you can see that a client can make a single GraphQL request to fetch multiple resources, such as the first 10 repositories off an organization, the first 10 deployments of those repositories, and the first 10 statuses of those deployments.

Garen: Building relationships like these also matches the goal of hypermedia navigation we alluded to earlier. Except instead of the server providing that information, the client is able to freely navigate through the objects.
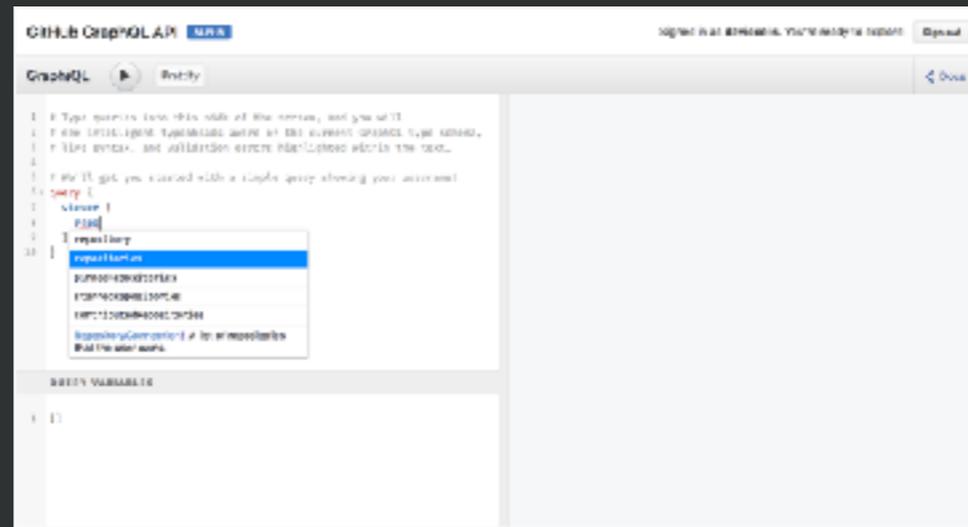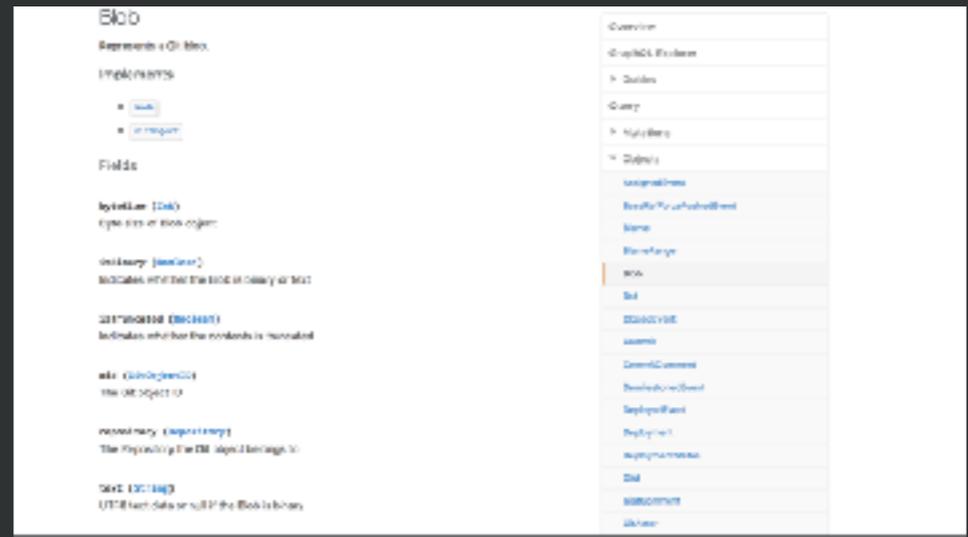
David: We also embedded GraphiQL directly into our documentation site at developer.github.com and provided an OAuth hook to sign in via GitHub. You can see in the upper right hand corner that I've authenticated as myself, and I'd be able to perform live queries against actual GitHub.com production data that I own.

Garen: Speaking of documentation, whenever a GraphQL schema is updated, we also automatically update the reference material. We have a cron job that runs every ten minutes that takes a look at GitHub's schema and generates new HTML pages if any of the descriptions have changed

Garen: This eliminates the human hassle of having to keep the reference material in sync with the code.

David: Which, by the way, is a very real hassle we have with our REST API…
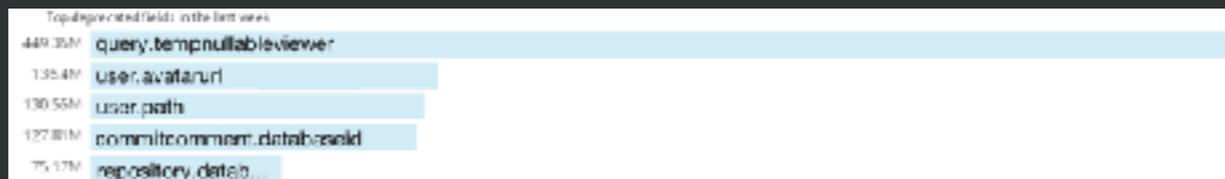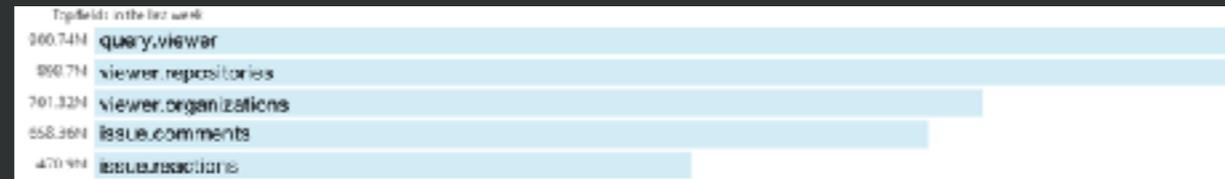
David: Since GraphQL also provides introspective functionality, we are also able to graph precisely which fields our clients are requesting. The top graph here shows the total sums for our top ten most requested fields.

David: The bottom graph shows the total sum of our top ten requested fields that we have marked as deprecated. Insights like these allow us to better understand what our users are interested in fetching through the API. It helps us prioritize where we ought to be focusing on for performance improvements or even new schema additions.

David: One of the best things about GraphQL is that it's still evolving. The specification is open to proposals from the community, and there are loads of upcoming features being considered.

Garen: *NEXT BULLET* There's a lot more functionality that we unfortunately don't have time to cover, but we can't resist discussing some advanced or upcoming features, which are aliases *NEXT BULLET* and directives *NEXT BULLET*.

Advanced features: aliases and directives

```
POST /graphql


query {
  repository(id: 3) {
    name
    KB: diskUsage
    MB: diskUsage
  }
}
```

```
Response

{
  "data":{
    "name": "GitHub",
    "KB": "3380000",
    "MB": "3380000"
  }
}
```

How people build software   **@davidcelis** and **@gjtorikian** (39)

Garen: In this query, we're fetching how much disk space a particular repository takes up. This is represented by the diskUsage field, which we store as an integer of bytes in our backend. You may have noticed we're requesting the same field twice, but with different bits before the field name. Those bits with colons act as aliases. We're requesting the same field twice, but with a KB and an MB alias. In our response, we'll get keys named KB and MB instead of diskUsage.

David: Of course, right now, the aliases are technically wrong because diskUsage is really stored in terms of bytes…

David: … but aliases can go hand-in-hand with directives, which provide a way to format the data that comes back. Now we're asking the KB alias to come back calculated to kilobytes, and the MB alias to come back as megabytes. So while in the database we have disk usage stored as bytes, in this example the server has defined that it supports this @unit directive which we can use to format the data. It's the server's responsibility to convert diskUsage into the unit that the directive is requesting.

David: And the client ultimately controls the format of the data it wants to receive.

David: Now, keep in mind that this query is just an example! This doesn't actually work on the GitHub GraphQL API. So don't go and try this query. Please.

GraphQL: The Future™

- The spec is still evolving!
- Many advanced/upcoming

  features:

  - Aliases

  - Custom Directives

  - Deferrals/Streams

  - Subscriptions & Live updates

How people build software

@davidcelis and **@gjtorikian** 41

Garen: There are also proposals to allow you to defer parts of a query if they're lower priority. ***NEXT BULLET*** The response comes in as a stream, and deferred fields stream in later. With streams, you can receive items in a connection one by one.

David: *NEXT BULLET* We've also got subscriptions, which will add pub/sub functionality to GraphQL, and the "live" directive would let you receive patches to a specific field. Your query would return a normal response at first, but as the field you requested as "live" is updated in the backend, you receive those updates.

David: So keep your eyes on GraphQL, it's been a blast using it to build out APIs and there are a lot of exciting features coming up. Finally, we'll leave you with our silly Welp app. You can find it at github.com/welp/welp.reviews if you want to look at an example GraphQL schema or even run it locally. The README has setup instructions, or you can tweet at us with angry complaints of it being broken.

Garen: And that's all, folks!