A GraphQL-first approach to improving your API.

First I’m interested to know, who here has tried out GraphQL? Anyone using GraphQL in production?

And who has heard of GraphQL, but hasn’t had the opportunity to try it out yet?
Today

1. GraphQL vs REST
2. The GraphQL Workflow
3. Lessons Learned
Today

1. GraphQL vs REST
2. The GraphQL Workflow
3. Lessons Learned
Original requirements were simple. No interactivity, one server, one client. Everything in MySQL. Serve static HTML. Forms are the most complex thing.
Complex single page web apps. Microservices, SQL, Redis, external APIs. Lots of clients. And of course, right now the predominant API type people use to transfer data between their services and clients is REST.
Good: flexible, I can build new products without changing the backend.
Bad: lots of roundtrips. One for the list of posts. Then one for the author of each post. Then maybe one for the likes. Also overfetching. Or underfetching (make examples)

That's not going to work.
Good: just the data I need. Fast.
Bad: Extremely inflexible. Development too slow. Lots of code to maintain. Essentially one endpoint for every client!

Neither of these are really great. Most people build hybrids anyways. And they start adding a lot of arguments to prevent overfetching and underfetching. Hybrid becomes a euphemism, frankenstein monster would be a better word.

So people start to cook their own solutions. I've seen half a dozen of these at different companies. RESTish. It's not necessary. Let's introduce a new standard.
My problem: Sometimes we all feel like this guy. We spend a lot of time with plumbing / wiring. We don’t get to focus on the business logic of our app.
So what do we need? What are our requirements?
We want single endpoints, but we also want to join. And we want to select exact fields.

Essentially we need a good abstraction!
What’s a good abstraction?
Surprise! GraphQL give you all these things.

GraphQL original design came from Facebook in 2012, when they were feeling the growing and performance pains of trying to write the newsfeed for native iOS with a RESTish a Frankenstein API monster.
GraphQL is cool because it’s a specification, not an implementation. It’s up to you to implement things how you want, according to these three components.

The first component of your GraphQL API — Schema: the definition of data your API has available, written in a strongly typed way.

Query: Ask for exactly the data you need, one trip, no more no less.

Results: Get predictable results in JSON form that can be consumed by your frontend components.
GraphQL
A specification, not an implementation.

```
type Project {
  name: String
  tagline: String
  contributors: [User]
}

query {
  project(name: "GraphQL") {
    tagline
  }
}
```

```
{ "project": {
  "tagline": "An API query language"
}
}
```

Schema is the heart of your API. Contract between frontend and backend development. We’ll get into this more later.
1. Nested data, congruent with needs for different view hierarchies
2. GraphQL design is unapologetically driven by the requirements of views and the frontend engineers that write them
3. Application specific API type system
4. Client is responsible for requesting it's data and stating what it wants, not the server.
5. Type system is queryable. Powerful platform to build common tools and client software libraries.
Who uses GraphQL?
2012: Initial design at Facebook
July 2015: Announced and open sourced
Sept 2016: Production ready
June 2017: Huge adoption, great tooling, some best practices
The future!
Clear growth, follows the same trend line of Google Trends almost exactly.
npm Downloads
Search term: graphql, redux

Compared to the most popular frontend state management package.
~12% of the most popular frontend framework. This is huge.

You’d normally expect to see an S curve, exponential to logarithmic. None of these curves have tapered off — exciting!

Maybe React growth is slowing a bit.

Other exciting thing, GraphQL is growing faster than React was when it launched. Slope is steeper.
All these companies were having growing pains, and now use GraphQL in production. Reaping the benefits I’ve mentioned of over-fetching and less round trips, etc etc etc.

If anything, these logos should be validation that GraphQL is here to stay. It is growing fast, this list definitely isn’t comprehensive. This list was far less populated last year. You could be on it next year!
I’ll talk first about some of the technology choices you might make when starting to use GraphQL. Then I’ll talk about the general workflow you can approach GraphQL dev with that we’ve seen work really well and think leads to some great results.
I’ve showed several versions of this slide, and talked about how you’ll need to define a schema. Schemas are a meta thing, in order to implement it, we need both our frontends and backends to respect the protocol, and make queries or send responses like so. That’s what makes this a “GraphQL API”.

In your tech stack, you can implement this protocol however you want. Code you end up writing on you frontend will be known as your GraphQL Client, and similarly on your backend, your GraphQL Server. There are some great implementations of clients and servers that people have made in many languages and already open sourced!
GraphQL as a spec leaves the implementation up to you.

It's what makes it so powerful. People can build, use, and share amazing tools in the ecosystem. GraphQL ecosystem is full of great tools!

It's likely that there are tools or implementations for your tech stack that exist already. Or you could make your own! It depends on your own specific needs.

On the right I've also listed some of my favorite tools in the GraphQL ecosystem. GraphQL improves your app’s performance. These tools are what improve your performance as a developer and enable you to move faster. Each of them could take up an entire talk of its own, but in the next few slides I’ll briefly highlight some.
GraphQL
Not just a tool: an open source spec.

Clients
- Apollo Client
- Relay
- GraphQL Fetcher

Servers
- graphql-express
- graphql-elm
- graphql-elixr
- graphql-python
- graphql-ruby
- graphql-scala
- Backend-as-a-Service

Tools
- API explorer (GraphiQL)
- Server prototyper (Launchpad)
- Devtools (Apollo Client Devtools)
- Performance analysis (Optics)
- Mocking (graphql-tools)
- Code generation (apollo-codegen)

Here are the logos of some clients and servers that exist already!

Clients: Apollo Client, Relay, GraphQL Fetcher.
Servers: graphql-express, graphql-elm, graphql-elixr, graphql-python, graphql-ruby, graphql-scala, BAAS companies
Here are some other logos I found while logo searching. GraphQL-related logos are usually creative derivatives of lines and circles. If you’re interested in writing your own tools for your system’s reqs, you’ll be in good company with logo design.
Once you’ve made the technical choices of what your stack will be, here’s the general approach of a Schema-first GraphQL workflow.
First we'll talk about schema design and why it's really important to do this upfront. The whole workflow is centered around this step.
Schema’s are the heart of GraphQL. We like to say it’s a contract of development between your frontend and backend teams.

With REST, your API is fully controlled by your backend team and by what they want to expose. They tell you what you can ask, and that’s all you can do. With GraphQL, your API is jointly owned. While it’s still the backend team that exposes the data, frontend teams can consume it however they want.

You need to work together to build the schema, not only to make sure that frontend teams are enabled to make reasonable queries for the data they want, but so backend teams can know what queries to expect and make changes accordingly. While frontend devs gain freedom with what they can ask, backend devs lose fine control over how their API is being used. You could theoretically make a monster query that overloads your backend, so you’d want to think about these things ahead of time.
GraphQL Schema

```
# A comment submitted by a user
type Comment {
  # The SQL ID of this comment
  id: Int!
  # The GitHub user who posted the comment
  postedBy: User!
  # The text of the comment
  content: String!
  # The repository which this comment is about
  repoName: String!
}
```

- Contract between frontend and backend
- Declares relationships across backend data sources
- Incrementally adopt - get the API you always wanted!

Best part is incremental adoption. Say you want to improve some part of your API, start with a small schema and a query or two to do that!

You can opt into using GraphQL component by component and go as quickly or as slowly as you want.
In fact, that’s exactly how we brought GraphQL into our commercial product Galaxy. Galaxy is still a hybrid, where about half its data is being fetched through Meteor DDP (similar to REST in protocol), and the other half is fetched through GraphQL and Apollo Client.

Galaxy was the first app we brought GraphQL to for one specific reason, and that was to simply the client logic on this page and improve its performance, because it kept crashing user's browser tabs (showing live data graphs, up to 30 or more at a time, and used to have to maintain connections for each graph and do all the joins on its own.

Our schema in Galaxy is quite small, and it's pretty heavily tied to the queries and data we’re displaying for this view.
Interesting question you'll encounter when designing your schema: which team really owns it?
Teams usually have different priorities when building it.
This is a spectrum. Teams have different values when designing an API. When we designed our schema for Optics, the frontend team took the first pass and we started basically from our wireframes and our knowledge of what our client’s needs would be. We made a proposal and the backend team then came and advocated for all the things I previously mentioned.

After a lot of back and forth, this is about where we landed. I imagine a lot of teams have similar stories, since as I said at the beginning, GraphQL is “unabashedly designed with client developers in mind”.

Who owns the schema, frontend or backend?
Once the contract has been agreed upon, it gives the teams lots of freedom to work independently.
Frontend teams know exactly what queries they can expect to make
Backend teams know exactly what they need to build

Teams can work in parallel for months with a strongly-typed spec to build against.
Using tools in the GraphQL ecosystem….
Frontend teams can use GraphQL mocking tools to prepare the queries they know they’ll have to make and build their components accordingly.
Backend teams can test their API with GraphiQL as they go and add more features.
Building a schema upfront can turn a workflow that traditionally looks like this...
Into something like this.
I want to tell a bit about how we implemented optics and what tools I use every day as I work on it.
In a REST world, this almost can't be the case, because the endpoints the API exposes are usually heavily tied to specific views or components, so teams need to work on these things together constantly throughout the entire implementation process as their understanding of requirements evolves.

Even if you manage to spec your entire API perfectly at the beginning, you not only have to spec out way more things than you would with the GraphQL schema, but your API is also a lot less flexible and subject to change as the frontends change.

As a frontend dev, I can only authentically speak to my own half of the story.

My impression of what this story was for our backend team is that they had a strongly typed spec to build and that was all they needed to be happy.
If you're trying to build an app without a backend: set up mocking.

Luckily, writing and maintaining mocking systems is a lot easier with GraphQL than with REST. ... explain slides ...
There are also some great tools out there to help you mock data even faster.

This is Launchpad, a tool that lets you write a complete GraphQL server in your browser, and deploys that server to a live endpoint which is queryable.

I’m mentioning it because among many other things, a great application of this tool is for GraphQL mocking. You can paste a schema in here, and have a really basic mocking tool that returns “Hello World” for fields that are Strings in less than one minute — with a URL that you can plug directly into your app while it’s running in mock mode.

You don’t even have to write these mocks client side.
Component driven development.

Start with the smallest components, and work up to larger views using mocking tools and building things one at a time.

Build up layouts into larger and larger views and queries.
GraphQL Workflow
How we built Optics.

1. Painlessly adapted to schema changes
2. Connected the frontend and backend in under two days
3. Developed and launched entire app in under 3 months
Cool thing about this workflow is that it doesn’t change for when we’re developing new features — the steps are all the same.

We sit down together and decide on an API, and the frontend team mocks data to develop the component as the backend team is creating the infrastructure to support the new API. It really feels like magic when you bring these two things together and they work perfectly the first time.

How often is it that code works right the first time you run it?
How do you load data to the client?

One question I’ve glossed over so far is how you actually load data into the client?
There are two methods. Like with REST, you can do a plain fetch, get your results, and consume them. This method doesn’t maintain UI consistency though or have any performance features.

You’ll probably want to use a caching client. Though they’re more work to set up, it’s easier to update the UI, they manage all your data in one place, and they enable you to make things like devtools that significantly improve the developer experience.
Why might you want a GraphQL client?

We spend a lot of time in the pipes writing code for our app to handle state management, cache updating, etc. Time spent here in tech debt reduction is time that you’ve lost when you could have been working on that feature users have requested for months.

All that work is repetitive across any app you work on, and you can save yourself from having to worry about the details by using a client that handles it for you and gives you a nice abstraction.

The two most popular clients out there are Apollo Client and Relay.
One the every day tools I do use that I haven’t talked about yet are the GraphQL devtools for the client we use, Apollo Client.

What I wanted to point out specifically is that these are GraphQL specific devtools that have an embedded version of GraphiQL which allows you to query against your endpoint with zero effort — no Postman, no setting of request headers, etc. It’s all handled for you. I use this all the time when experimenting with new queries for new features I want to build.

You also have the ability to explore the current state of your live schema through this tool. We were actually moving so quickly when building Optics that there were a lot of instances where I went to someone on our backend team ask if a new field was ready, or if they’d deployed a specific change, and their response was always “Check GraphiQL”.
GraphQL Workflow

- Schema Design
- Implementation
- Monitoring

@danimman
Show of hands for people who do monitoring of their production services? This is mostly to see if you’re all still paying attention, because I’d hope this is everyone!

Being able to do production monitoring of **GraphQL** is an important part of the GraphQL workflow.
GraphQL Monitoring
What can you get?

1. Know exactly what fields are used and how
2. Measure field resolver performance
3. Detect breaking schema changes
4. Field-level caching across queries
   … and more!

GraphQL opens up a world of questions you could never even ask about your API before.
Back in the early days of us adopting GraphQL. This is a screenshot from some rudimentary instrumentation we did to trace query resolvers.
Query performance improving from 500ms to 300ms resolution time.

This is the type of information you can only understand from being able to dive deeply into how your fields resolve, which something you can only do with GraphQL.
If you’re interested in seeing what some GraphQL instrumentation might look like, Optics is a free tool that you can use for up to 10K requests per month.

GraphQL opens up a world of questions you could never even ask about your API before through looking at the usage of your schema. Again, this ties into Schema-first design because the way in which you design your schema originally will affect the way in which you monitor it later, and the way in which your queries resolve later.
With that, I’ve wrapped up a lot of the elements that go into a GraphQL workflow. The key point is really: take the time to invest in your schema up front, and don’t end up with a Frankenstein monster of an API. It’s an easy trap to fall into if you started with the specific prop hierarchy for a component you were building, or a specific query you knew you wanted to make for example.
I’d like to wrap up my talk by sharing some lessons we’ve learned from using GraphQL, interacting with so many people in the community, and building tools for other developers.
There's value in spending time doing extra communication and documentation.

When we survey our open source community on what we could be doing better, people always come back and tell us we should write more content, tutorials, documentation...

I've also spent a lot of time making the point today investing upfront time in your schema design, and doing that communication between your development teams early on, is what's going to enable you to fully leverage all the benefits of GraphQL later down the road.
There’s value in investing in tooling. Productivity is a product of the tools we have. These tools are what enable us to work efficiently.

I’ve been so surprised, through working in this ecosystem, with how much the tools I’ve used have saved me time. I have a good perspective, because I’ve been working with GraphQL since before a lot of the tools existed, so I know what it was like before and after.

When I say invest in tooling, I mean, don’t be afraid to spend a week working on a productivity tool that might feel like a distraction at the time, but will save you many months in the future.

The most powerful tool for me day-to-day is the Apollo Client devtools. They were originally built by three engineers as a one week long experiment, and they’ve paid off many times over and over, not only for us, but for the community of people who use Apollo Client.
There's value in prioritizing the developer experience.

I've talked a lot about the benefits of GraphQL for frontend developers. When I talked about monitoring, I went over some of the interesting insights backend teams get with GraphQL.

My favorite quote when asking my teammates why they liked GraphQL came from one of our backend engineers. He said… “…”

I liked this “in a nutshell” phrase, because we’re all humans who are writing this code. GraphQL has great performance benefits for the code we’re writing, that’s why it was originally dreamed up. But it also has great benefits for our velocity as developers. It helps us move faster and focus on what matters, and I think this is even more important because it helps us move onto the next bigger, more important things faster.
Today

1. GraphQL vs REST
2. The GraphQL Workflow
3. Lessons Learned

How GraphQL came about and its benefits over its predecessors.
What's involved with the GraphQL workflow.
How to get the most out of GraphQL.
This talk has mostly been an introduction/overview, but GraphQL Summit will be a great place for you to see deeply technical talks on GraphQL, hear case studies, and meet other people using it.

Know anyone that wants to work on GraphQL full-time? Questions? DM me on Twitter!

Thanks for the time!