What's GraphQL?

Just say what you want, honey

Justin
Mar 15, 2022

The TL;DR

GraphQL is a query language (and runtime) for building and using APIs: developers use it to simplify making API requests.

- Most apps and sites are built on a **network of interconnected APIs**: when you load your profile or feed, they provide the data
- APIs are generally built with a **purpose-per-endpoint**, which means that getting all of the data you need involves **many calls and server trips**
- GraphQL acts as a **wrapper on top of your APIs** so you can say exactly what you want and only need to talk to the server once
- GraphQL also supports **schemas and introspection** to make your data easier to understand

Almost 40% of Javascript developers have used GraphQL before, and it powers pretty high profile APIs like Github's.

How apps work with regular APIs

To understand GraphQL, you need to understand why we needed it in the first place. In What's a Web App and What's an API, we covered the basics of how every app is built:

1. **A frontend** with components that you interact with (the Twitter "app" or site)
2. **A backend** (database) that stores application data like user information
3. **API endpoints** that interface between that frontend and that database

When you load your Twitter feed, your browser is sending a bunch of API requests to the Twitter backend, fetching all of the data it needs (tweet text, who

tweeted them, when they tweeted them, the number of likes, etc.) and then showing that information to you in a nice, pretty format. This same process happens when you click on a specific tweet, run a search, or mute a word - everything happens via API endpoints that Twitter created.

This is the way that web apps have worked basically forever. Unsurprisingly (as this is a pitch, after all) this "architecture" leads to a few frustrating problems:

→ Multiple server trips

Because API endpoints tend to be organized on a *resource* basis, getting *all* of the information we need to populate a whole Twitter page - your profile, your feed, search results - tends to involve multiple API calls to multiple endpoints. I'm just spitballing here, but I'm guessing Twitter has endpoints like these that power your homepage:

- A `/profile` endpoint for profile data
- A `/feed` endpoint for feed data
- A `/trending` endpoint for trending topics

This *resource-based* organization (one endpoint per *thing*) is actually a critical tenet of REST – the protocol that most APIs are built on – but it can make apps a lot less efficient and more complex to manage.

> **🤔 Undefined Terms 🤔**
>
> We'll tackle the REST protocol and explain these ideas (resource-based organization) in a future post. For now, just think of endpoints as very single purpose (which is generally true).
>
> **🤔 Undefined Terms 🤔**

→ Too much data

Endpoints will often return more data than you actually need. Twitter's `/profile` endpoint probably returns a ton of data about a user, and the frontend only actually *uses* a few basic parts of it (number of followers, number of tweets, etc.). This is by design – endpoints can be used for multiple tasks that

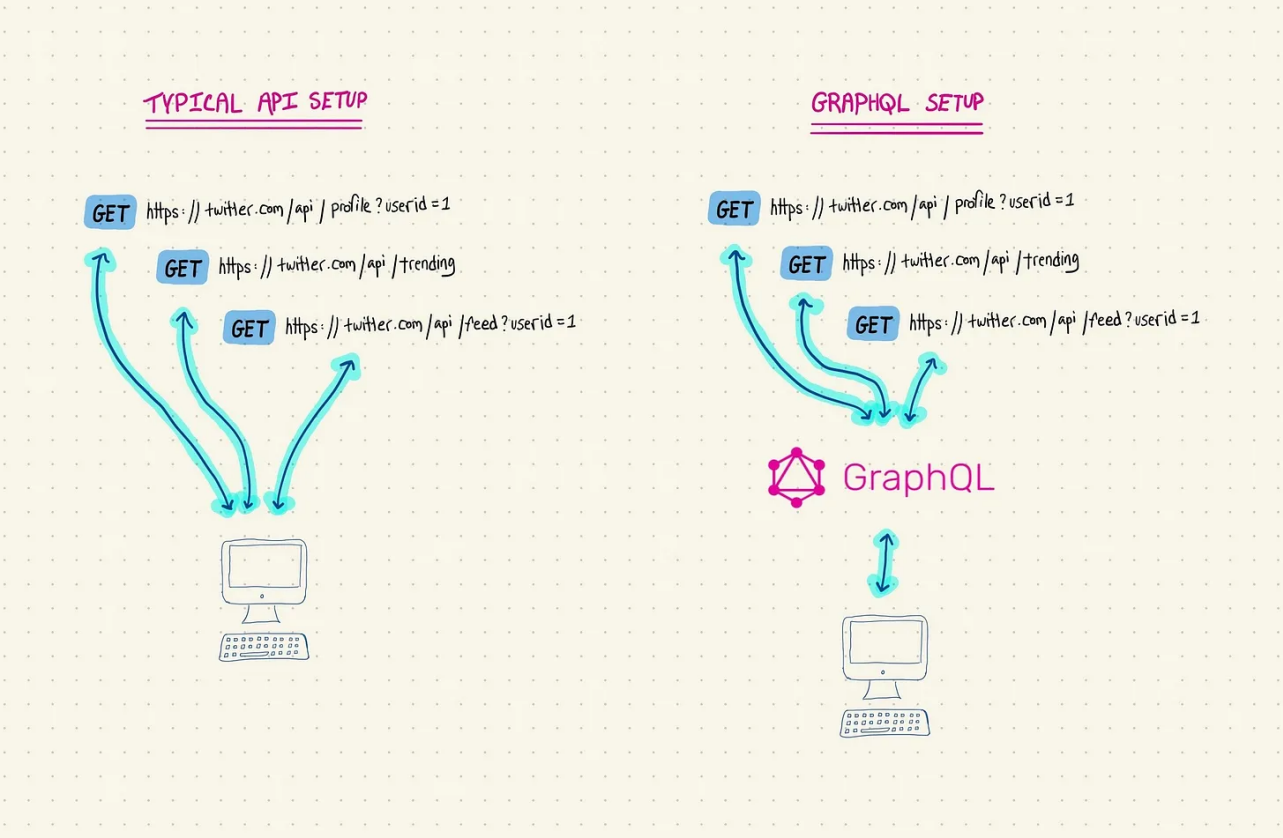
require different data – but it means that you usually need to parse out the specific fields you need, and a bunch of extra data needs to travel across the internet.

→ **General disorganization**

API endpoints are independent of each other – the only way to get a bird's eye view of what's available to use and how to use it is documentation (engineers writing the answers down). Endpoints change all the time, and I have literally never worked at a company where engineers were excited about writing and maintaining updated documentation (although I'm told these people do exist).

GraphQL, our savior

Rewind to 2012: Facebook had just passed 1 billion users, and was dealing with all of the problems we just outlined, at hyper scale. They started working on an internal framework to solve them, and open sourced / released it in 2015. It's called GraphQL, and it's a wrapper that sits on top of your APIs; instead of querying all of your endpoints separately and directly, you query a **single, ergonomic GraphQL endpoint**, and it takes care of the legwork of actually getting you the data you need.



You can think of a GraphQL server as a sort of personal shopper. Instead of going to 10 different stores and shopping at each, you tell your shopper what you're looking for, and they come back to you with the finished haul. You just need to make sure you tell them exactly what you want.

Here's an example of what a standard GraphQL query might look like:

```
query {
  allStarships(first: 3) {
     starships {
      name
      model
      crew
      passengers
    }
  }
}
```

We're querying our Star Wars endpoint (yes, seriously) for the first 3 starship models, and getting their names, models, crews, and passengers. Here's what we'll get back:

```
{
  "data": {
    "allStarships": {
      "starships": [
        {
          "name": "CR90 corvette",
          "model": "CR90 corvette",
          "crew": "30-165",
          "passengers": "600"
        },
        {
          "name": "Star Destroyer",
          "model": "Imperial I-class Star Destroyer",
          "crew": "47,060",
          "passengers": "n/a"
        },
        {
          "name": "Sentinel-class landing craft",
          "model": "Sentinel-class landing craft",
          "crew": "5",
          "passengers": "75"
        }
      ]
    }
  }
}
```


I got this from a working demo that GraphQL provides here – play around with it to get a better sense of how this really works.

GraphQL solves the problems that we identified with typical API endpoints:

- **Multiple server trips**: you define exactly what you want, and GraphQL takes care of which endpoints need to be used to return that data, all in one server trip
- **Too much data**: GraphQL only returns that fields (name, passengers, etc.) that you ask for, even if the original endpoints give you more than that
- **General disorganization**: GraphQL endpoints are built on user defined schemas that clearly show what fields are available and how to work with them

This is obviously the simple version of the story, but the ideas hold: GraphQL is growing *really* fast, and an entire ecosystem of supporting tools has been developing over the past few years.

Schemas, resolvers, and introspection

GraphQL doesn't magically learn how your endpoints work: you need to explicitly define how it's supposed to find the data it needs. If you've got an endpoint that returns a user's profile data (like in our Twitter example), you tell GraphQL that when you ask for "name" and "followers" it should go to that endpoint to get them. These series of functions are called **Resolvers** and you need to set them up for any fields you want to access.

As you set up Resolvers for all of your data points, you're basically documenting how your endpoints work - and GraphQL takes advantage of that with **Schemas** and **Introspection**. When you're building your GraphQL endpoint, you define a schema: it's just a list of all of the available fields, what types they are, and how they relate to each other. Here's what a piece of a GraphQL schema might look like:

```
type Starship {
  id: ID!
  name: String!
  length(unit: LengthUnit = METER): Float
}
```

A full GraphQL implementation will have this information for every available field, and it's super clutch: it's basically like an automatically evolving set of documentation. If you want to get details on any data, you can introspect with a query that returns the available data types and fields:

```
{
  __type(name: "Starship") {
    name
    fields {
      name
      type {
```

```
        name
        kind
      }
    }
  }
}
```

> **🚨 Confusion Alert 🚨**
>
> Schemas and introspection can get confusing: the way it works is that you define a schema, and then can introspect that schema (i.e. use it, look at it) through GraphQL queries.
>
> **🚨 Confusion Alert 🚨**

There's a lot going on here, which is why GraphQL typically needs to run on its own server. Popular JS frameworks and middleware like Express.js already have GraphQL extensions that make GraphQL simpler to add to your existing setup. But as GraphQL has gotten more and more popular, developers are starting to reorganize their entire architectures around it. Fauna is a database that supports GraphQL natively (this is a big deal), and Apollo and Prisma provide layers above databases and APIs for smoother GraphQL integration. It's still early, but things are moving fast.

Terms and concepts covered

`GraphQL`

`Resolvers`

`Schema`

`Introspection`

Further reading

- Because using GraphQL spans the whole stack, tools like Apollo help with a lot, and they can be hard to understand. This primer can help
- The State of JS 2019 Survey has a lot of interesting data about GraphQL, Apollo, and JS in general
- You can query GraphQL in specialized IDEs that help format queries and introspect schemas

---

3 Comments

Write a comment...

James Mar 19, 2022 Liked by Justin

GraphQL seems to reduce the volume of network calls required but would you still need to update resolvers every time there's a change in underlying API?

♡ 4 Reply Collapse •••

1 reply by Justin

B whiteside May 22, 2022

What would I need this for baby cakes?

♡ 1 Reply Gift a subscription Collapse •••

1 more comment...
