Log in Star ...

Search...

Wed, Oct 13, 2021

GraphQL 101: What is GraphQL?

by Matt Tanner

Most people within the tech world have heard of GraphQL at this point. Adoption is also booming at steady rates. There's a lot of hype around GraphQL technologies and what they could mean for the future of app development.

What is GraphQL?

There's plenty of explanations out there to answer "what is GraphQL?". Many of which are convoluted or lead to further confusion on the subject. In the simplest terms possible, GraphQL gives developers customized access to the data they need. A developer can request the data they need with absolute flexibility so that they know exactly what data is being returned. This eliminates the need to return large payloads from a REST endpoint just to extract 2-3 fields.

A GraphQL server is the main hub. This is where your GraphQL endpoint code runs and receives requests from users. Once a request is received, the code running on the GraphQL server retrieves the data from whatever data sources the GraphQL service is utilizing. This could be a database, web endpoint, or many other different options. It's somewhat similar to a traditional web-based API server setup. Users of the GraphQL service tend to use a GraphQL client in their front-end code.

The client allows them to easily interact with the GraphQL service and also offers some other benefits we will touch on later.

Components of a GraphQL endpoint

THE SCHEMA

A GraphQL endpoint is built from a schema. It is the root of our GraphQL endpoint and core building block of a GraphQL service. The schema defines what data will be available to the client. A basic schema will outline different Types (or objects) that are available, and the fields those Types contain. From this, developers can easily see what is available at their disposal. An example of a GraphQL schema is shown below:

```
type User  {
    id: ID!
    username: String
    posts: [Post]
    comments: [Comment]
}
type Post  {
    id: ID!
    user: User!
    text: String! @search
    comments: [Comment]
    ratings: [Int]
    average_rating: Int
}
type Comment  {
    id: ID!
    post: Post!
    user: User!
    text: String!
}
```

Of course, GraphQL schemas can also support many other features. They can be as simple or as complex as they need to be. For more details on this, you can check out this resource on GraphQL Schemas and Types.

THE RESOLVER

Once you define a schema, you then need to map the data from a data source into the schema. This is what a GraphQL resolver does. The resolver "resolves" the data. It is the link that fills in the gap between the data the user wants and "how to serve that data to the user. In the simplest terms, the resolver knows how to get the data a user is asking for. It also formats the data to fit an appropriate GraphQL response. Usually, this involves the resolver accessing a database, retrieving the data required, and returning it back to the user in the format defined in the GraphQL schema. Instead of a database, the data source can also be a third-party API that the resolver calls. You can read up on how a GraphQL query gets executed, root types, and resolver functions for a more in-depth explanation.

How is GraphQL used?

GraphQL is generally used to retrieve or mutate data. The functions can be done in a few different ways and revolve around queries, mutations, and subscriptions. In order to do these operations, you'll need to use a GraphQL client. A GraphQL client allows your application to make requests to your GraphQL server. You'll likely want to dig a bit further into Graph clients as you become more familiar with the technology. Many options exist!

QUERIES

You use a GraphQL query to retrieve data. The query works very similar to a SQL query in the sense that you get to describe your criteria or filter along with the data you'd like to retrieve. This works similar to a GET request on a RESTful endpoint. The difference with GraphQL is that you get complete flexibility on what data you want to send and retrieve. Based on the schema above, here is a sample query that retrieves some data based on the `Post` type:

```
query {
  queryPost {
    id
    text
  }
}
```

You'll notice that the query keyword at the start of the statement denotes that we are executing a GraphQL query. Once the GraphQL query gets executed, the user would receive back the data they've asked for if it exists. They will only receive the fields they've asked for. No more and no less. Here's what that returned data may look like:

```
{
  "data": {
    "queryPost": [
      {
        "id": "0x3dea15a36",
        "text": "Security in web apps"
      },
      {
        "id": "0x3dea15a39",
        "text": "A crash course on Webpack"
      },
      {
        "id": "0x3dea15a3a",
        "text": "Top 10 GraphQL tools to make development easier"
      },
      {
        "id": "0x3dea15a3f",
        "text": "GraphQL: A beginner's guide"
      },
      {
        "id": "0x3dea15a47",
        "text": "Making sense of browser compatibility"
      },
      {
        "id": "0x3dea15a49",
        "text": "How Turing machines work"
      }
    ]
  }
}
```

The beauty of a query result from a GraphQL endpoint is that the data comes back in the exact shape you requested. Developers can have a more predictable experience because of this. In traditional RESTful APIs, fields come back in a predefined response and lack customization in terms of the response payload.

MUTATIONS

When we want to update or delete data, we use a GraphQL mutation. Similar to a POST or PUT in RESTful terms, this operation allows us to manipulate data. Unlike a POST or PUT, GraphQL mutations come with more flexibility and allow for a more refined way to interact with the data. With a mutation, it is possible to create or update one or many entries in the dataset.

A mutation request starts with the mutation keyword. You'll also pass the data that you want to add to your server-side data. Based on the query we saw earlier, here is what a mutation would look like to add some data in:

```
mutation {
  addUser(input: [{username: "albert"}]) {
    user {
      username
      id
    }
  }
}
```

When a mutation gets executed and completed, you can also return the newly created or updated data. This is useful since it allows us to mutate the data and then return that piece of data for use or confirmation. Instead of issuing a subsequent query manually, you can do all of this in a single request.

Below is the response after executing the above mutation:

```
{
  "data": {
    "addUser": {
      "user": [
        {
          "username": "albert",
          "id": "0x3dea15c60"
        }
      ]
    }
  }
}
```

You can also send multiple mutations in a single request. The important thing to note is that they will be executed in order. If we send two mutations in a single request, before the second one starts, the first one is guaranteed to finish.

SUBSCRIPTIONS

Subscriptions are queries that respond to changes in the data. A user can set up a subscription by specifying the data they want to watch. A subscription is defined on the server-side and the client-side. When a change occurs, either through an update or the creation of new data, the subscriber will be notified and sent the updated data.

It's important to be cautious about the usage of subscriptions. They should not be used in every scenario where data needs to be kept in sync. Instead, it may be better to simply poll data or update the data based on user action (like the click of a button). A few cases where you may want to apply subscriptions would be to keep track of small changes in large objects. Polling large objects is resource-intensive and not efficient compared to subscribing to changes through a subscription. Another great use is when low-latency, real-time updates are required, for example, notifications or live updates. Something like a real-time chat app would greatly benefit from this type of usage.

There are many ways to implement subscriptions in a GraphQL service. A great place to dig deeper is in the Apollo docs on using subscriptions in React. You might be interested in our blog post that shows how to integrate subscriptions using Apollo in a toy Instagram clone.

How does it compare to REST?

Many of the conversations around GraphQL involve the premise of "GraphQL versus REST". I think much of the time this leads to confusion. Both technologies can be used separately, together, or not at all. It's not simply a matter of one versus the other.

It is, however, acceptable to use REST in order to understand how GraphQL fits into the equation in a modern development stack. It allows those unfamiliar with GraphQL to see the path to adoption. It also shows how GraphQL may remedy some of the long-standing problems that have come with RESTful API adoption.

In a typical RESTful CRUD setup, we would see a few technology layers. This would include the endpoint itself, possibly some business logic, and a database. Each time a RESTful request comes in it needs to be translated or transformed into a database query. For every operation, this setup will need to be repeated.

Since a REST endpoint generally has limited scope, lots of endpoints need to be created for each operation an application needs. This has led to an API problem. Every time we need new functionality, we are either forced to edit the existing REST endpoint or create a new one. This can have many side effects, including duplication of code, confusion for developers, and so on. In the case of updating the endpoint, this could also lead to breakage for current consumers of the endpoint.

This is where GraphQL really shines. Imagine, a single endpoint where you can query and mutate any of the available data. Gone are the days of waiting for the backend team to build out individual APIs.

Is a unified data graph what we should all aim for?

This is a very important question in terms of system architecture. The premise of a unified data graph is to move away from having individual data sources and GraphQL APIs. Instead, all data is exposed through a single interface.

Plenty of ways are available to achieve this, like schema stitching and federation. Both are ways that individual services can be combined into a single, unified data graph.

Of course, GraphQL fanatics love the idea of having an entire organization's data exposed through a single interface. It brings plenty of advantages in terms of ease of data access. It allows developers to develop applications more quickly without having to search for the data they need.

There are drawbacks, though. Including authorization and security issues that need to be thought about very holistically. It's not as simple as just not allowing a developer to only have access to specific REST endpoints that they require.

So without going into extreme detail, it depends on your organization's needs. The data you store and its sensitivity will determine if you should strive to expose all of your data through a single, unified data graph.

What are the advantages and disadvantages of GraphQL?

Like any technology, there are advantages and disadvantages to using GraphQL as well. As the technology grows in popularity, many of the disadvantages are being solved for. In time, as GraphQL matures, I believe we will have solved most of the disadvantages in some way or another. Here are a few of the advantages and disadvantages to note for GraphQL:

ADVANTAGES

- A GraphQL client can request data from the server and dictate the format of the response.
  - Eliminates the problem of "over-fetching". This is where a response contains fields that the user didn't ask for.
- Retrieve many resources in a single request.
  - This is unlike REST where we may need to make multiple calls to different endpoints to retrieve the data we require. This is also known as "under-fetching".
- API is self-documenting so that users can see exactly what data is available and know how to easily create a request.

DISADVANTAGES

- GraphQL responses always return a 200. This is regardless of whether the request was successful or not.
- GraphQL has a lack of built-in caching support.
  - Most solutions do have some sort of assistance to help with this issue though.
- GraphQL can add complexity.
  - A simple REST API with data that is unlikely to change can be much more simple to implement and maintain

compared to GraphQL.

- Deeply nested queries and recursive queries, when not discouraged or stopped, can lead to service issues including DDoS attacks on an endpoint.
- Rate limiting is more tricky to do, especially when all data is exposed through a single GraphQL endpoint.
    - With REST, you may rate limit individual endpoints but this is tougher to break up in GraphQL

Knowing some of the above complexities, the decision to adopt GraphQL should be a pragmatic one. We can easily see the benefits to developers by looking at the mass adoption of GraphQL throughout many small and large organizations' technical stacks. The technology is here to stay and can be a great tool to add to any organization's toolbox.

How can I adopt GraphQL?

Compared to REST and other alternatives, GraphQL offers an extremely wide array of options. There are ways to convert your existing services into GraphQL, ways to leverage existing RESTful endpoints to resolve data in your GraphQL service, and a multitude of platforms that support GraphQL out of the box. You can also build services from scratch with a framework, similar to how you would build RESTful services. In short, adoption is simple to get started with. The toughest part is figuring out exactly what approach you want to take.

If you build from the ground up, choosing a language and framework or library to build GraphQL endpoints, the learning curve may be steep. This depends on the complexity of what you are building as well. The benefit is that this approach offers the most customization, the trade-off is the amount of work it takes to get up and running.

Using a platform, like Dgraph, mitigates much of the learning curve. Although it is easier to use a platform, you may lose some customization that you'd get from a "ground up" approach. However, most platforms are heavily focused on customization and those capabilities are growing very quickly. Almost to the point where they are as customizable as other options for building GraphQL.

There are further considerations for enterprise adoption of GraphQL. Luckily, those have been already conveniently mapped out in another blog post!

Adopting GraphQL starts by taking the first step: trying it out for yourself!

Try out Dgraph Cloud today!

We are confident that Dgraph is the easiest way to get started with GraphQL. Simply create a schema, click Deploy, and have an instant GraphQL backend. Our solution also gives you instant data persistence, no external database is needed. Since our solution includes a GraphQL API and graph database powered by a single GraphQL schema, you'll have an instant backend ready to serve GraphQL requests from your app. Dgraph isn't a GraphQL code generator, it's an all-in-one platform to serve all of your GraphQL needs!

Getting started is as easy as creating a Dgraph Cloud account and launching your first backend in a matter of minutes. Take our A Tour of Dgraph that'll give you a headstart with interactive, hands-on lessons.

Try it out today to experience all the benefits of GraphQL without all the hassle!

Tags

graphql graphql-101 api tutorial database dgraph cloud dgraph-cloud

Matt Tanner, Author

Matt is a Developer Relations Lead at Dgraph passionate about all things related to data, development, and architecture. Previously, Matt has worked as a developer, tech lead, and architect for some of the largest financial institutions and insurers in Canada. Most recently, he has focused on working with global startups to help with Developer Relations. He is always dabbling in the latest tech and applying this to his own ventures in technology

Start Discussion 0 replies

The #1 graph database on GitHub is easier in the Cloud

Get Dgraph Cloud

START FREE →

But wait, there's more...

Fri, Feb 17, 2023

Dynamic AutoScaling of GitHub Runners

by Sudhish KR

Wed, Jul 20, 2022

Dgraph raises $6M in Seed Round

by Dgraph Labs, Inc.

Tue, Dec 14, 2021

Dgraph v21.12: Zion - The Last City Standing

by Manish Rai Jain

PRODUCTS	LEARN	CUSTOMERS	COMPANY
Dgraph Cloud	Interactive Tour	BLOG	Team