LEVVEL

Can My Business Benefit from GraphQL?

2019

Author: Nicholas McHenry

LEVVEL

Overview

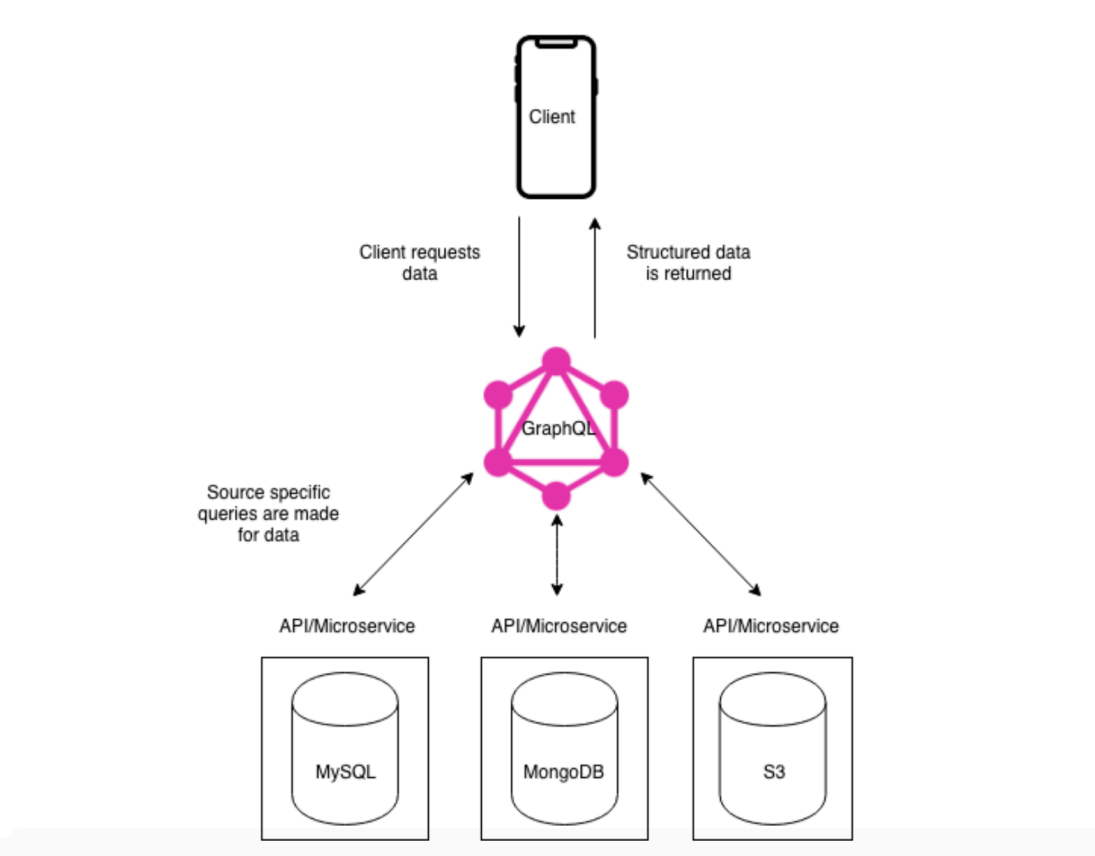
The purpose of this guide is to determine whether your organization can benefit from the inclusion of GraphQL in your API and microservices architecture. If you can answer yes to any of the questions below, you can benefit from using GraphQL.

- Does your engineering team maintain rapidly changing internal or external APIs? Are these APIs using a form of versioning?
- Is the data stored and made available by your organization highly complex, correlated, or nested?
- Do you have information that needs to be accessed by internal and external developers stored across multiple microservices
- Do you have an engineering team focused on front-end technologies like React or Angular? Does this team have trouble retrieving data from your internal APIs
- Is there confusion regarding the data that is available through your APIs
- Does your organization spend a significant amount of time revising API or microservice endpoint architecture given new requirements?

GraphQL Introduction

As per its design specification, GraphQL is “a query language designed to build client applications by providing an intuitive and flexible syntax and system for describing their data requirements and interactions.” Another way to think about it is as a query language for you APIs, much like SQL is a query language for a relational database.

More than just simple data retrieval though, GraphQL offers a host of features and benefits such as a declarative, strongly typed syntax and providing only the data that is specified by the client. Additionally, GraphQL supports a wide range of server-side languages and therefore can be implemented on different microservice to provide a universal query interface. Originally, GraphQL began as an internal project at Facebook but is now maintained as a hugely successful open source project on Github. The diagram below illustrates GraphQL architecture at a high level.



Why Does GraphQL Exist?

As noted above, GraphQL was initially developed at Facebook during the re-development of their native mobile applications in 2012. One of the first problems they faced, and many other data-intensive businesses now face, was the complex and nested nature of their data as well as its incompatibility with traditional RESTful API design. We will attempt to illustrate these phenomena with a brief example involving the modeling of information typical of a social media application.

To begin, we can think of a simple set of RESTful endpoints for retrieving specific users and some of their data shown below:

```
/users  -> returns all users

/users/21 -> returns user with an id of 21

/posts -> returns all posts

/posts/21 -> returns all posts by the user with id 21
```

What if we want to retrieve the friends of a specific user and their data? We would likely use a set of endpoints like these:

```
/users/21/friends -> Returns the friends of the user with id 21

/users/21/friends/posts -> Returns the posts of friends of the user
with id 21
```

The complexity is growing, but still manageable. However, what if we wanted to make available data about the interests of a specific friend and all of the posts that they liked? Unfortunately, it may look something like this:

```
/users/21/friends/25/liked_posts_and_interests -> Returns the liked
posts and interests

of the friend with id 25 of the user with id 21
```

The specificity and non-uniform nature of these endpoints break with RESTful principles. Not only this, but continuing to design an API in this way isn't scalable when we consider the number of custom endpoints that must be developed, tested, and documented to make available all the necessary data for the application. Taken together, this all equates to reduced development velocity, wasted man-hours, and wasted money.

Of course, there is the halfway solution of piecing together a set of calls to different REST endpoints to get the information needed by your frontend developers and users. This introduces the common problem of over-fetching and under-fetching. We can think of this simply as merely more or less data than we need in a single HTTP request. In the case of over-fetching, an engineer may only need a small amount of data but is forced to retrieve a more substantial amount of data than needed given the inflexibility of traditional RESTful endpoints.

Conversely, under-fetching occurs when an engineer must make more than a single request given that a particular set of data is not available from a single endpoint. In both the case of over-fetching and under-fetching there are performance issues, as you are either taking up unnecessary network bandwidth for excess data or making more HTTP requests than necessary.

Reasons to Use GraphQL

While it didn't necessarily catalyze its creation at Facebook, the rise in popularity of loosely-coupled microservice architecture has undoubtedly fueled the increased adoption of GraphQL as well as its popularity among companies like Github and Paypal. Particularly, a microservice based architecture is an excellent use case for GraphQL. This is due to the fact that it is common for technologies, including newer databases and programming languages, to be adopted to overcome evolving business challenges. This causes an organization's microservices to be deprecated, updated, and generally change over time.

However, one thing that doesn't change is the fairly universal business need of retrieving and aggregating information stored in different microservices. In an older RESTful architecture, a different HTTP call would have been made to each microservice with subsequent logic performed once all the data had been retrieved.

A better, more efficient solution to this would be to query all of these microservices through one universal interface and a single HTTP call. GraphQL enables this functionality, and as previously mentioned, it has implementations in a variety of server-side languages so even if you have legacy microservices implemented in Java and .Net and modern ones implemented in Go, GraphQL offers a unified querying mechanism.

Another steadfast benefit of GraphQL deals with API versioning. While noted on the official GraphQL website that there is nothing that inherently stops a GraphQL service being versioned like any other API, the framework does take an opinionated stance on it. Specifically, the organization and implementation of GraphQL allows for API schemas to continuously evolve instead of only during new version releases.

See our Guide to Subscriptions in GraphQL with Apollo

With a traditional semantically versioned RESTful API, a significant amount of time and money is spent planning new endpoints to make more data externally or internally available. In addition to this investment, there is always the concern that upcoming API changes may unintentionally break client-side code or other services that rely on a previous version. If a traditional API version switch is botched and corresponding endpoint changes aren't communicated to all parties involved, the aforementioned concern often becomes a reality which results in further costs in terms of developer hours and system downtime.

On the other hand, in the context of a GraphQL service, a good amount of these issues can be avoided in conjunction with a robust CI/CD pipeline. If new data needs to be made available to solve a business problem, a backend developer can add to a GraphQL schema rather than writing a new endpoint or altering an existing one. Once the changes are rigorously tested and verified, they can be deployed to the appropriate environments and used by other developers. Functionally speaking, more data can be added to a GraphQL schema in this way without running the risk of breaking dependent code. Overall, this allows for a much more organic evolution of an API schema that is more in line with the reality of new and sometimes unpredictable business requirements.

On the topic of data availability and visibility, GraphQL comes with built-in tools that allow anyone to visualize the data available from a GraphQL service. This dramatically simplifies the job of a front-end developer or any stakeholder who is expected to use the information provided by the service. Furthermore, GraphQL is also designed with front-end and design teams in mind as first-class citizens in its ecosystem.

Primarily, the flexible nature of GraphQL lends itself well to rapid UI iteration given that client-side developers can work relatively independently of the server-side developers. This is due to the fact that frontend developers don't need

custom endpoints created every time a UI tweak is made that requires new or slightly different data. Even in the rare cases where an unexpected change needs to be made to a GraphQL schema to make more data available, it is generally simpler to do so than in a traditional RESTful API.

In turn, this has the effect of freeing up backend developers to work on delivering valuable features instead of wasting time context switching and patching existing APIs. On the other hand, frontend developers can continue focusing on the user experience instead of losing velocity communicating with and waiting on backend teams. All of this ultimately allows an organization to deliver products faster, cheaper, and with fewer headaches.-

Conclusion

This checklist covered some of the features and benefits of GraphQL at a high level, though a significant number of aspects of the technology weren't included. GraphQL isn't a silver bullet for solving API design problems, but it does alleviate some of the pain points that modern, data-intensive enterprises face. Properly used, GraphQL can be used to build scalable, easy-to-use APIs that can adapt rapidly to changing business needs.

LEVVEL

About Levvel

We are an IT, strategy, and design consulting firm that combines the innovative DNA of a startup with the wisdom, scalability, and process rigor of a Fortune 100 company.

Levvel's proven DevOps Assessment methodology delivers a precise strategy to enable your team to deliver on key practices that help organizations innovate faster through automating and streamlining the software development and infrastructure management processes.

We firmly believe that mentoring can be integrated with delivery. Our main focus is on saving our partners as much as possible on the lifetime total cost of ownership and maintainability of their systems. For more information, contact us at hello@levvel.io.