O'REILLY®

GraphQL for Modern Commerce

Complement Your REST APIs with the Power of Graphs

Kelly Goetsch

GraphQL for Modern Commerce

Complement Your REST APIs with the Power of Graphs

Kelly Goetsch

Beijing • Boston • Farnham • Sebastopol • Tokyo

GraphQL for Modern Commerce
by Kelly Goetsch

Acquisitions Editor: Jennifer Pollock
Development Editor: Angela Rufino
Production Editor: Beth Kelly
Copyeditor: Octal Publishing, LLC
Proofreader: Charles Roumeliotis
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

January 2020: First Edition

Revision History for the First Edition
2020-01-06: First Release

978-1-492-05684-3

[LSI]

To Oleg Ilyenko

Who introduced all of us at commercetools to GraphQL in 2015. His passion for GraphQL and leadership in the open source community served as a model for the rest of us. We are eternally grateful to have had him as a friend, colleague, and collaborator. Rest in peace, @easyangel.

Table of Contents

CHAPTER 1

Introducing GraphQL

Of the thousands of new commercial and open source projects that are released every year, only a handful end up becoming widely used industry standards. GraphQL is one of those technologies for a very simple reason: it brilliantly solves a host of problems for a wide range of constituencies.

This book provides a solid overview of GraphQL and its application to commerce. In this first chapter, we cover GraphQL itself—the problems it solves, how it intersects with REST, and why it's such a natural fit for commerce. In the second chapter, we look at the GraphQL specification itself, including key terminology, how to model your schema, and how to write queries, mutations, and subscriptions. In the third chapter, we explore GraphQL from a front-end developers' standpoint and look at the role that clients can play. Finally, in the fourth chapter, we focus on GraphQL servers, which are actually responsible for executing queries against the schema and performing the requested action.

NOTE

We use the term "queries" to capture all of the operations that you can execute against a GraphQL server, even though GraphQL also supports mutations (changing data) and subscriptions (watching for changes to data).

Commerce Requires More Than REST

In the 1990s, commerce platforms shipped with the frontend and backend as one indivisible unit. In that decade, the web was the only channel and developers needed only to build a website. Given the immaturity of frontend frameworks, the commerce platforms had to offer the frontend as part of the commerce platform. "Headless" didn't exist as a concept. REST hadn't been invented yet and JavaScript first appeared in 1995. These early commerce platforms contributed to or invented much of the software used to dynamically generate static web pages, including JSP, JHTML, PHP, and ASP. The major problem with this approach is that any changes, no matter how small they were, had to be handled by IT and slotted in as part of a formal release. This upset marketers who wanted to be able to make changes on their own.

In the 2000s, content management systems (CMSs) like Day, Interwoven, and Vignette began to offer integrations with commerce platforms, whereby the commerce platforms would serve the backend and the CMS would serve the web frontend. This allowed IT to manage the backend and marketing to manage the frontend. Although this made life easier for marketers, the integration between the frontend and backend was hardcoded with what could only be described as spaghetti code. The two were permanently wedded and inseparable. This was all premobile, so, again, the only channel anyone cared about was the web.

In the 2010s, there was an explosion of consumer electronic devices that put the internet in everyone's pockets. Mobile phones with fully featured browsers and native apps became widely used (see Figure 1-1). The Internet of Things (IoT) became real as the price of semiconductors and internet connectivity fell. Even devices as mundane as TVs started to become connected to the internet and could be used to facilitate commerce.

Commerce has transformed into something that's part of our everyday lives, embedded into the dozens of screens and other internet-connected devices that we interact with on a daily basis.

While commerce was transforming, representational state transfer (REST) APIs were emerging as the default means of exchanging data between disparate systems. When compared to CORBA, SOAP, and the other standards that preceded it, REST was an enormous improvement. Developers could build a REST API, write code to implement the API, and then offer up the API to any system that wanted to call it and had proper access rights.

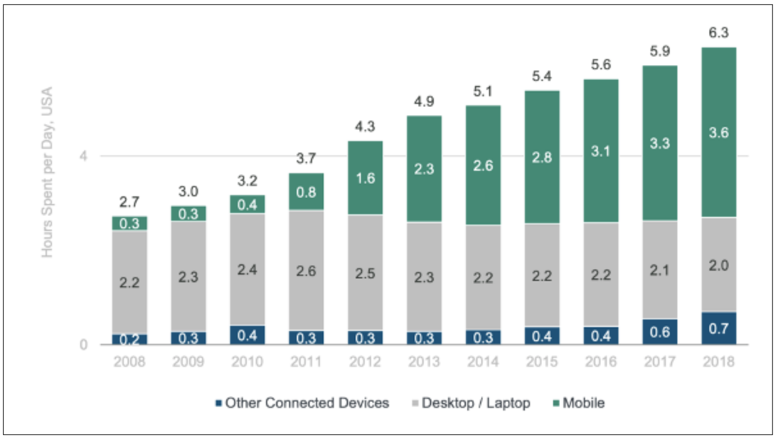


Figure 1-1. Daily hours spent with digital media per adult user, USA (source: Bond)

REST APIs are now the default means of exposing commerce-related data and functionality from applications/microservices. Any application or frontend can now consume that data and functionality to build an experience for a customer, whether on a desktop-based website or an IoT device on a 3G network in an emerging country.

Even though REST APIs are great for backend developers, they pose many challenges for frontend developers. As the number of clients capable of facilitating commerce continues to grow by the week, the power is progressively shifting from backend to frontend developers.

The Challenges of REST APIs

REST APIs are great, but like all transformation technologies the benefits don't come without new challenges. Let's explore.

Lack of a specification

REST doesn't actually have an actual specification, because it's more of an architectural style than a formal standard. The term REST was coined and its principles were put forth in Roy Fielding's 2000 PhD dissertation titled *Architectural Styles and the Design of Network-Based Software Architectures*. Although his dissertation goes a long way toward outlining a vision, it's by no means an actual detailed specification with rules that can be programmatically validated. In his dissertation, Roy explicitly says:

> REST ignores the details of component implementation and protocol syntax in order to focus on the roles of components, the constraints upon their interaction with other components, and their interpretation of significant data elements.

On the other hand, HTTP, OAuth, JSON, and most of the other technologies that developers use on a daily basis are formal documented standards, each with thousands of pages outlining granular rules for what is and what is not considered valid. Take a look at how the JSON specification dictates how arrays should be represented:

> **5. Arrays**
>
> An array structure is represented as square brackets surrounding zero or more values (or elements). Elements are separated by commas.
>
> array = begin-array [value *(value-separator value)] end-array
>
> There is no requirement that the values in an array be of the same type.

There's no ambiguity about this and any JSON-compliant document will represent arrays the exact same way, every time. Standards promote interoperability.

The term "REST" could and often is applied to any human-readable API offered over HTTP that returns JSON or XML as a response type. There could be substantial differences in error handling, authentication, authorization, response types/formats, query parameter format, and so on. Because of these inconsistencies, client-side developers need to work with the idiosyncrasies of each API that they want to call, which in the case of a large data-intensive web page could be dozens.

There are a few standards out like Open API and OData but these standards have differences in what parts of REST they choose to include in their respective specifications. For example, Open API allows for one of four authentication schemes, whereas OData allows for any authentication scheme. Even within the specifications, there are inconsistencies that make lives difficult for frontend developers. For example, the Open API specification allows responses in both JSON and XML formats. Even with a catalog of Open API–compliant APIs, frontend developers might need to embed a lot of frontend logic to account for implementation differences.

Not knowing which API to call

The same data (product, price, inventory, order, etc.) or functionality (decrement inventory, calculate price, mark an order as having been shipped, etc.) might be available in many different systems across a typical enterprise. It's common that an enterprise resource planning (ERP), order management system (OMS), warehouse management system (WMS), commerce platform, and myriad custom applications all touch much of the same data and offer similar if not identical data and functionality, all while claiming to be the "single source of truth." Each of these systems might offer different levels of data freshness, different ways of accessing the data (REST APIs, SOAP APIs, batch feeds, etc.), different performance and uptime guarantees, and so on. Some of the APIs might not even be publicly available.

A common example of this issue is inventory—who actually owns it? Does the commerce platform? The WMS? The OMS? When you query the WMS or the OMS, do those systems take into account inventory that's in users' shopping carts but not yet firmly reserved through a placed order? There are so many nuances that it can be challenging for a frontend developer to discover which APIs to call and what data or functionality each offers and then figure out how to call each API.

To make matters worse, REST doesn't have introspection. The clients and servers aren't working against a fixed contract: clients can send anything and servers can respond with anything.

A common fix for this is an internal wiki page documenting all of the available APIs that clients can call, but those are often stale and their effectiveness is limited by organizational silos.

Overfetching data

A common problem with REST APIs is overfetching data. A typical SKU could have hundreds of attributes, ranging from the standard attributes like display name and long description, all the way to manufacturer-specific codes and logistics-related information. Although the commerce platform has all of these details, each client does not. An Apple Watch client needs only the display name, images, and maybe a short description. Retrieving the entire product is the easiest for a frontend developer:

```
HTTP GET for /products/12345
```

This works great within the confines of a datacenter, but imagine having an Apple Watch fetch an entire two-megabyte payload over a cellular network every time a user wants to look at a product's details? Multiply that across every API needed for an app to function and you could have overfetching that leads to serious performance degradation.

There are two general approaches for solving this problem. The first is to specify in the HTTP request exactly which fields are required:

```
HTTP GET for /products/12345?attributes=display-
Name,images,shortDescription
```

But not all APIs support this, and the format of how those fields are specified tends to differ based on which API you're calling. Again, remember that REST doesn't have any formal standards and the ability to retrieve only certain attributes is not part of that specification. Assuming it's even possible, frontend developers would need to build and maintain these long unruly URLs.

Another approach is to build "backends for frontends," which are little applications that return exactly what's required for a given client. There might be an "AppleWatchProductDetail" microservice that exposes an API that gives the Apple Watch application exactly the few fields it needs. That microservice then calls the full product API.

The challenge with this approach is that you need to maintain hundreds of microservices, and each of those microservices is tightly coupled to the client and to the backend APIs. Every new client now requires a few dozen new microservices, as illustrated in Figure 1-2.

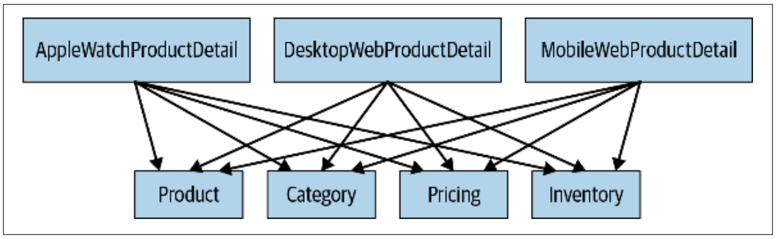


Figure 1-2. Backend for frontend approach

This certainly works, but it comes at the cost of higher maintenance. Many client-related changes and many backend API changes require updating each of the many backends for frontends on which those clients and APIs rely.

Underfetching data

Related to overfetching data, another problem with REST APIs is underfetching data. To render an order history web page, for example, you'll first need to retrieve orders placed by a given customer. Next, you'll need to retrieve the shipment status of each order. Then, you'll need to find out what a customer can do with each order, such as whether an order can be returned. Often times, the requests to the different APIs must be made serially because the data from one API is needed to make the call to the next API, as demonstrated in Figure 1-3.

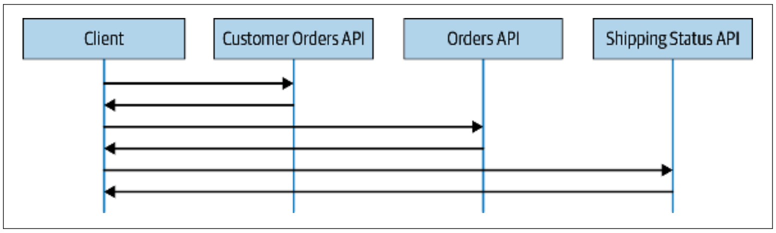


Figure 1-3. Sequential calls

Making multiple, serial round-trip calls from clients (which might have limited computing power, limited bandwidth, and high latency) is very expensive. Devices, especially the lower-power IoT-style devices, can have limited processing capacity, and making so many HTTP requests can consume too much power. Devices like smart watches and mobile phones are often connected through cellular networks, which have low bandwidth and high latency, both of which are very bad for performance.

Also, all of those sequential calls amplify any performance issues your APIs might have. Making 10 parallel calls to APIs that respond in an average of 200 milliseconds results in 200 milliseconds of perceived latency to the end customer. But making the same calls sequentially could result in more than two full seconds of perceived latency. When you add in rendering and other overhead, you could easily be rendering pages or screens in seconds rather than milliseconds.

Finally, making all of those calls from the client requires the client to be fairly intelligent. It needs to know which APIs to call and in what order. When multiplied across the dozens of different clients, this could lead to a proliferation of logic in clients that is undesirable for frontend and backend developers alike.

Underfetching can sometimes be solved by having multiple copies of data in each microservice. Going back to the order history example, you could have an order history microservice that has copies of every order, the shipping status and available actions of each order, and the full product details of the products. The major downside to this is the intelligence that each microservice now needs to have. Rather than specializing on one function, each microservice needs to perform tangential functions. Besides the sheer volume of duplicate data (a single order could have multiple megabytes worth of data), this expands the amount of work that each team needs to perform. It can also be difficult for clients to know where to retrieve data when there's so much duplication. The final problem with this approach is that the data each microservice has isn't always the most up to date. For example, the shipping status microservice has the most up-to-date data but if that data is being copied over to the order history microservice, the order history microservice will by definition have out-of-date data.

The other way to solve this is to build a backend for your frontend, like how you'd solve the overfetching problem.

What Is GraphQL?

Although REST is great, its lack of standardization, lack of discoverability, and overfetching and underfetching all cause headaches for client-side developers. GraphQL can solve these problems and many more, all while building on top of—not replacing—REST.

What Are Graphs?

Before we formally introduce GraphQL, we first must explain the concept of a graph. After all, "Graph" is the defining part of the name "GraphQL."

Graphs are a collection of interconnected objects. Here are some common examples in the real world:

- Transportation networks like air, rail, and auto
- Family trees
- Social networks like Facebook, Twitter, and LinkedIn

Graphs also underpin much of IT as well, including the following:

- The World Wide Web (Google built its company on graph theory, which underpins its famous PageRank algorithm)
- XML documents (including HTML documents, which are a subset of XML)
- Networks (routers themselves are nodes)

Any time you have a collection of interconnected objects, you have a graph. When you look around, you'll find them everywhere. Graph theory, a subset of discrete math, is the formal study of the relationships between graphs.

When you look at a social network like Facebook, it's a big graph. "Friends" are bidirectional relationships between two nodes in a graph. Twitter is another example of a graph, but the relationships can be one-way (I follow Martin Fowler but he doesn't follow me). In graph theory, this is known as an undirected graph. Given that Facebook and Twitter are essentially big graphs, it shouldn't be a surprise that Facebook invented GraphQL and Twitter was an early prominent user. Both of their businesses (along with Google) owe their success to having properly understood graph theory.

Commerce Graphs

Commerce is full of graphs. There are objects (customers, orders, products, etc.) and relationships between those objects (customers place orders, which contain products). Figure 1-4 shows a graph with five directed relationships.

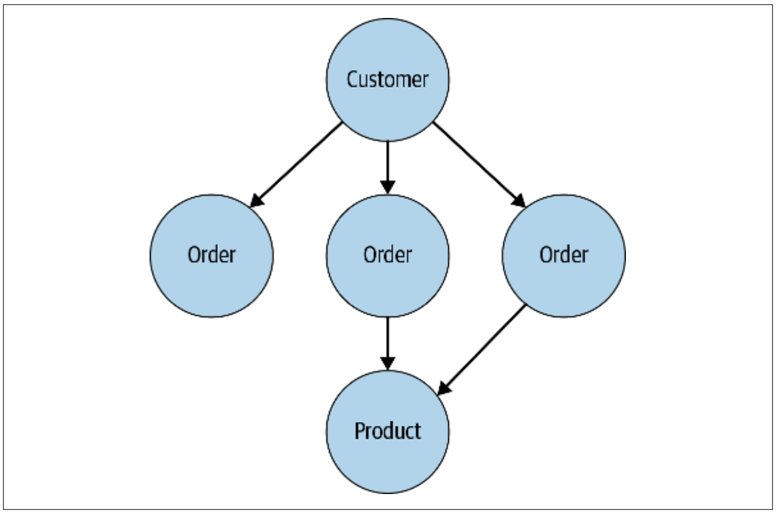


Figure 1-4. Example of a commerce graph

A foreign key in a database is a relationship between two nodes in a graph.

REST is definitively not graph-based. It's great for retrieving single objects (customers, orders, products, etc.) and maybe their dependencies (the products contained in an order, etc.), but that's it. REST would be the equivalent of being able to select data from only one database table at a time, whereas GraphQL allows you to select data from multiple tables in one query.

Now that we've covered what graphs are, it's time to actually explain what GraphQL is.

In the Beginning…

In the 2010s, Facebook began to have the same problems with REST as outlined earlier. The company had more than a billion active users in 2012. Those billion-plus users accessed Facebook from every conceivable device in every conceivable configuration.

At the time, Facebook supported desktop web, mobile web, and iOS/ Android wrappers (which were thin wrappers over mobile web). The iOS and Android applications were widely panned, but Facebook was limited in its ability to create native clients due to the limitations of REST.

In 2012, Facebook set out to build entirely new native mobile apps. Lee Byron, Nick Schrock, and Dan Schafer came together, and by early March they released an initial prototype of GraphQL that was then called SuperGraph. Later that year, Facebook released GraphQL internally in more or less its modern form, with it powering the company's native apps. The desktop web and mobile web clients also switched over to GraphQL after its success was demonstrated with the native apps. Today, every time you pull up Facebook from any client, you're using GraphQL behind the scenes.

Because of GraphQL's success internally, Facebook released its specification externally, including a JavaScript-based reference implementation in July 2015. In September 2016, GraphQL left the "technical preview" stage, meaning that it was mature enough to be considered production ready.

Facebook originally patented GraphQL in 2012 and then licensed it as BSD+Patents when the company publicly released it. Although this license did help to protect Facebook and the GraphQL community from patent trolls, it prohibited users of GraphQL from ever having the legal right to sue Facebook for any patent infringement, even if it wasn't at all related to GraphQL. Bowing to pressure from the GraphQL community, Facebook relicensed the specification under the much more permissive Open Web Foundation Agreement (OWFa), which grants GraphQL users full patent rights and makes it easier for other organizations to contribute. GraphQL.js, the reference implementation, was also relicensed under the extremely permissive MIT license.

In November of 2018, Facebook created the GraphQL Foundation and donated the GraphQL intellectual property to it. The GraphQL Foundation is a top-level member of the Linux Foundation, and a peer to the Cloud Native Computing Foundation (CNCF). The GraphQL Foundation's managing board is composed of senior leaders from Apollo, AWS, IBM, PayPal, Twitter, and others, and their responsibilities are as follows:

- Provide a roadmap for the GraphQL specification
- Oversee changes to the specification
- Provide legal, marketing, and community facilitation support

Today, GraphQL is used by hundreds of enterprise-level organizations. The major cloud vendors along with other software vendors of all stripes (including the commerce platform vendors!) have, or are adding, GraphQL support, as well.

Introducing GraphQL

GraphQL is a formal specification for retrieving and changing data, similar to how SQL is used to change and retrieve data from database tables. At a very high level, GraphQL describes the language and grammar that should be used to define queries, the type system, and the execution engine of that type system.

A very basic example of a GraphQL query is something like the following:

```
query {
        product(id: "94695736", locale: "en_US") {
                brandIconURL
                name
                description
                price
                ableToSell
                averageReview
                numberOfReviews
                maxAllowableQty
                images {
                        url
                        altText
                }
        }
}
```

The GraphQL server would then respond with something like this:

```
{
        "data" {
                "product": {
                        "brandIconURL": "https://
www.legocdn.com/images/disney_icon.png",
                        "name": "The Disney Castle",
                        "description": "Welcome to the magical
Disney Castle!....",
                        "price": "349.99",
                        "ableToSell": true,
                        "averageReview": 4.5,
                        "numberOfReviews": 208,
                        "maxAllowableQty": 5,
                        "images": [
                                { "url": "https://
www.legocdn.com/images/products/94695736/1.png", "altText":
"Fully assembled castle" },
                                { "url": "https://
www.legocdn.com/images/products/94695736/2.png", "altText":
"Castle in the box" }
                        ]
                }
        }
}
```

That response then could be used to render a product detail page as shown in Figure 1-5:

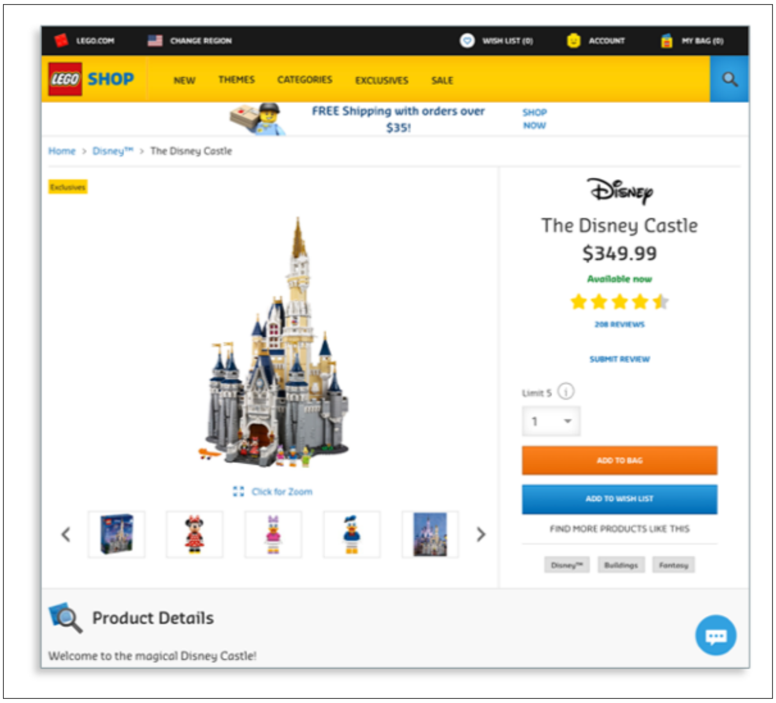


Figure 1-5. Lego's product detail page

Rather than interacting with the multiple APIs that hold this data, the client-side developers can make one GraphQL query against an endpoint that's typically exposed as */graphql*. The GraphQL server then validates the query and calls what are known as "resolvers" to retrieve (or modify) data from the underlying APIs, protocol buffer (protobuf), datastore, or any other source system. GraphQL doesn't take a stance on what programming language you should use, how to retrieve data from resolvers, or much of anything else. GraphQL is a specification, not a specific implementation.

> **NOTE**
> Even though GraphQL is a specification, GraphQL's official reference implementation underlies most of the GraphQL servers on the market, including Apollo and Relay. It was first released by Facebook in 2015 and has emerged as the community standard.

To this point, we've talked only about GraphQL queries (retrieving data). Besides queries, GraphQL supports three other action types:

Mutations
: Creating, updating, or deleting data, then an optional read after data is written

Subscriptions
: Watching for updates to data

Introspections
: Retrieving metadata about what types of queries can be performed

With support for all three operations, client-side developers can build any experience imaginable.

Benefits of GraphQL

GraphQL offers many benefits. Let's explore some of them.

Easier frontend development

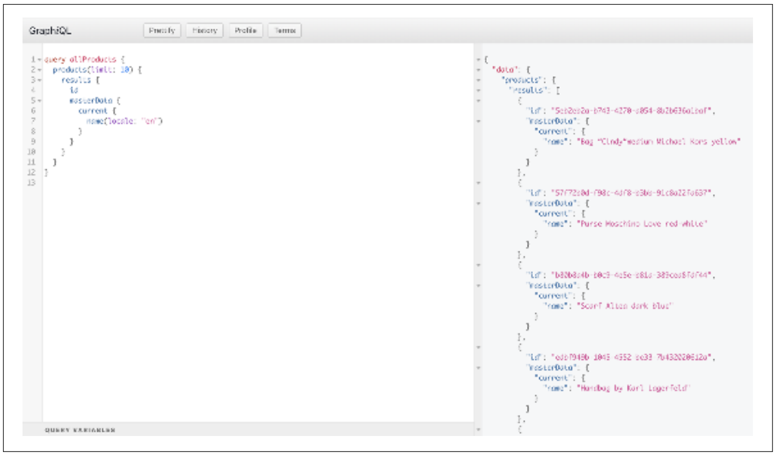
GraphQL's primary benefit is that it empowers frontend developers. Given the wide array of devices and clients you need to support, it's possible to have many more frontend developers than backend developers. Those developers must also iterate much more quickly than the backend. Browsers, operating systems, frontend frameworks, end-customer tastes, and so on all change so frequently that each of these teams need to be releasing constantly. GraphQL is the layer that decouples the frontend teams from the backend teams and allows frontend teams to rapidly innovate.

Frontend developers no longer need to hunt for APIs or try to understand which APIs have the most up-to-date version of data. That complexity is handled by the GraphQL layer. If there are four different APIs holding inventory data, it's the GraphQL layer's responsibility to identify which API should be made available to the clients. Now, each team doesn't need to go through the exercise of discovering each API and hoping they're using the right one.

GraphQL also handles the intricacies of calling each source system (API, protobuf, datastore, legacy system, etc.). Source systems can use different transport protocols, different data formats, and different authentication and authorization schemes, and can have different idiosyncrasies in how they're called. Frontend developers don't want to deal with that. Duplicating all of that logic in every client is difficult, expensive, and slows down overall development velocity. With GraphQL, every time a backend system changes (i.e., from XML to JSON data representation), it's completely transparent to each client. The change can be solely implemented in the GraphQL layer. Clients should be as free of frontend logic as possible.

Another advantage of GraphQL is introspection. Frontend developers can access one endpoint (typically */graphql*), and using an integrated development environment (IDE) or even by calling GraphQL directly (see Figure 1-6), they can quickly understand what types (objects like customers, orders, and products) and fields (attributes like firstName, shippingAddress, and displayName) are available.

Because GraphQL is introspective, every single query is validated before it's executed. Again, it's similar to SQL in that regard. Developers immediately know whether their queries are valid, even if the source systems aren't available.



```
query allProducts {
  products(limit: 10) {
    results {
      id
      masterData {
        current {
          name(locale: "en")
        }
      }
    }
  }
}
```

```
{
  "data": {
    "products": {
      "results": [
        {
          "id": "5eb2ed2a-b743-4278-a854-8b2b636a1baf",
          "masterData": {
            "current": {
              "name": "Bag "Cindy"medium Michael Kors yellow"
            }
          }
        },
        {
          "id": "57f72d0d-f98c-4df8-a3bd-91c8a22fa637",
          "masterData": {
            "current": {
              "name": "Purse Moschino Love red-white"
            }
          }
        },
        {
          "id": "b82b8d4b-b0c9-4e5e-a81a-389cad8fdf44",
          "masterData": {
            "current": {
              "name": "Scarf Alton dark blue"
            }
          }
        },
        {
          "id": "edbf948b-1043-4352-ae33-7b4322208612a",
          "masterData": {
            "current": {
              "name": "Handbag by Karl Lagerfeld"
            }
          }
        },
        {
```

Figure 1-6. GraphiQL, the GraphQL IDE

Improved performance

Next, GraphQL dramatically improves the performance for end customers.

GraphQL makes data from multiple source systems available in one JSON document. Frontend developers specify exactly the data that they need, and GraphQL provides it by making parallel requests to the source systems that have that data. Within a datacenter, latency, bandwidth, and computing power are basically unlimited. It's far more advantageous from a performance standpoint to make all of those requests within a datacenter and then offer up the client a single document containing that data. Clients are often connected over mobile networks, which are bandwidth and latency constrained. Client devices are often constrained by their computing power. Making all those HTTP requests has a cost, and small IoT-style clients, wearables, and other devices don't have that much computing power to work with.

Also, GraphQL completely eliminates the problems of over- and underfetching. Frontend developers specify exactly what they need and GraphQL provides it. There is likely to be some over- and underfetching when the GraphQL layer calls the source systems, but again, within a datacenter resources like latency and bandwidth are essentially unlimited.

Less code to maintain

Finally, GraphQL leads to there being far less code to maintain. As previously discussed, clients don't need to replicate the logic required to call different source systems. Clients can execute queries against a GraphQL server rather than having to write complex authentication code for each source system. Any changes with how source systems are called can be made in one location.

Also, GraphQL completely eliminates the need for backends for frontends. Clients can query the GraphQL layer for everything they need. Not having that intermediary layer dramatically cuts down on how much code needs to be written, tested, maintained, and run.

Drawbacks of GraphQL

Like all new technology, GraphQL isn't a silver bullet that magically fixes all of your problems.

GraphQL is another layer that must be maintained with its own architecture, development, operations, and maintenance needs, though it does eliminate the need for dozens or hundreds of backends for frontends.

Also, security, as we discuss in Chapter 4, can be challenging with GraphQL. The GraphQL specification leaves out security entirely, leaving it up to each vendor.

The final challenge with GraphQL is that it can be difficult to combine multiple GraphQL endpoints and schemas. Frontend developers want one endpoint (*/graphql*) with one schema, but different teams and different vendors will all have their own endpoints and schemas. The commerce platform vendor can expose its own */graphql* endpoint and schema, for example. Your cloud vendor, CMS vendor, search vendor, various internal teams, and so on might all expose their own endpoints and schemas. Frontend developers then need to access multiple endpoints. At that point, why even bother with GraphQL when REST is already available? Fortunately GraphQL server vendors do offer a way to offer your frontend developers a single GraphQL endpoint and schema, which we discuss in detail in Chapter 4.

GraphQL Compared to REST APIs

Part of the reason REST has become so popular is because microservices have emerged as the default architecture for building many cloud-based applications and commerce applications in particular. As Figure 1-7 shows, microservices are individual pieces of business functionality that are independently developed, deployed, and managed by a small team of people from different disciplines. For more on microservices, take a look at *Microservices for Modern Commerce* (O'Reilly, 2017).

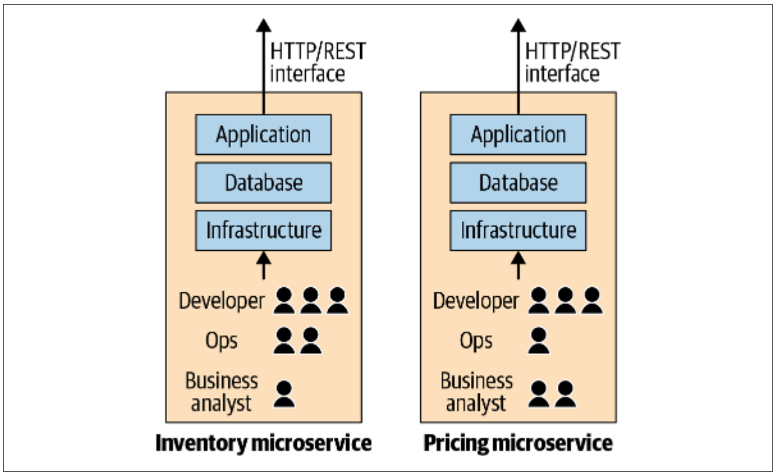


Figure 1-7. Microservice architecture

Microservices by definition require an API, given that a microservice's data can be accessed *only* through an API. In other words, a client or another application cannot directly read or write to a microservice's datastore. REST APIs are what most choose to expose.

With REST, developers are optimizing for east/west communication (communication within a datacenter between applications), not north/south communication (communication from a datacenter to a client, as demonstrated in Figure 1-8). Developers don't care much about the latency/bandwidth/computing power constraints of their clients. Their scope is within a datacenter, where all of those resources are basically unlimited.

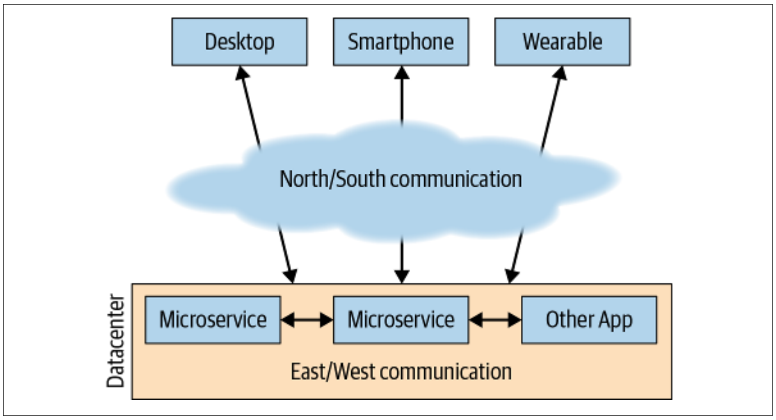


Figure 1-8. North/south versus east/west communication

Too much focus on east/west communication leads to developers automatically returning all data from an API by default. If a product has 200 attributes, any given developer is likely to write the application to return all 200 of those attributes. A product could be hundreds of kilobytes or even megabytes. The client has very little say in what data is returned. Developers don't have much visibility or concern for what happens after the data is exposed in an API. With GraphQL, developers specify exactly what they need. GraphQL shifts the power from the backend developers to the frontend developers.

As previously discussed, REST is more of an architectural style than a formal standard. GraphQL, on the other hand, has a very strict specification, which we cover in greater detail in Chapter 2. GraphQL's strict standard supports introspection, which allows frontend developers to quickly see which types and fields are available. REST can allow developers to discover types through Hypermedia as the Engine of Application State (HATEOAS).

```
<product id="{id}">
        <link rel = "inventory" uri = "/Inventory/product/
{id}"/>
</order>
```

HATEOAS helps frontend developers discover new types, but it's not widely used. Because REST doesn't have a formal schema, field-level introspection is not possible.

Although GraphQL and REST do share the same transport protocol (typically HTTP) and often the same data response format (JSON), the two approaches for retrieving data are substantially different and were designed for different use cases. If REST is working for you now, stick with it.

If you need more than REST can provide, consider adopting GraphQL as a complement to REST, not a replacement.

Final Thoughts

In this chapter, we discussed the challenges of REST and why commerce, specifically, requires more than REST on its own. Then, we introduced GraphQL and discussed its pros and cons. Finally, we discussed why you should view GraphQL as a complement to REST, not a replacement for it.

In Chapter 2, we cover the GraphQL specification.

CHAPTER 2

The GraphQL Specification

A specification allows all parties to work together with a common language and syntax. Specifications are everywhere. Motorists drive their cars on the same side of the road, obey stop lights, obey posted road signs, and follow other local regulations. Without clearly defined rules, traffic collisions would happen everywhere. Written languages are an example of another specification, with Arabic and Hebrew specifications calling for text to be written right to left, whereas English and German specifications call for text to be written left to right.

In the technology realm, some common technologies have a specification and some do not. XML has a very strict standard. From the formal specification, you'll find many examples like this:

> The formal grammar of XML is given in this specification using a simple Extended Backus-Naur Form (EBNF) notation. Each rule in the grammar defines one symbol, in the form
>
> symbol ::= expression
>
> Symbols are written with an initial capital letter if they are the start symbol of a regular language, otherwise with an initial lowercase letter. Literal strings are quoted.

Any human or software that writes or reads an XML document can make sure the document is valid because there's a strict specification.

As previously discussed, REST doesn't have a specification. A REST endpoint can return a response in a specific standards-compliant version of JSON or XML, but you'll find little agreement across REST endpoints on how response bodies should be structured or how errors are logged.

Introducing the GraphQL Specification

With a better understanding of what a specification is, let's now apply that to the GraphQL specification.

What's in the GraphQL Specification?

The GraphQL specification is a lengthly set of rules that govern how the schema should be defined and what happens after a query is passed to the GraphQL server.

The schema itself starts with the Schema Definition Language, which is commonly abbreviated as SDL. The SDL defines the basic syntax for the language. Think of it like punctuation in a written language. For example, here's what the GraphQL definition says about line terminators:

> LineTerminator
>
> New Line (U+000A) Carriage Return (U+000D)New Line (U+000A) Carriage Return (U+000D)New Line (U+000A)
>
> Like white space, line terminators are used to improve the legibility of source text, any amount may appear before or after any other token and have no significance to the semantic meaning of a GraphQL Document. Line terminators are not found within any other token.

The SDL also covers topics such as reserved names, how to comment, directives, the available operands, and so on.

Next, the specification outlines the root operations that are available. As discussed in Chapter 1, those operations are query, mutation, and subscription. Think of these as verbs that define which operations can be executed. The specification then outlines how to create types (objects like customers, orders, and products) and fields (attributes like firstName, shippingAddress, and displayName). Think of types as nouns. Similar to many other programming languages, GraphQL offers the ability to define interfaces, unions, fragments, and so on. The specification then defines how to tie together operations (like a query) with a type (like a product).

Finally, the specification goes into depth on how GraphQL servers should execute and respond to queries. For example, the specification details how the server should respond to successful and unsuccessful queries. The specification also details the rules for when operations can be executed serially or concurrently, for example.

The specification is about 216 printed pages of dense technical writing, but this level of detail is what makes GraphQL so valuable. The prescriptiveness of the specification guarantees interoperability within the ecosystem, which makes the entire ecosystem more valuable.

The specification is written for software developers writing GraphQL clients and servers. Users of GraphQL would be better served by HowToGraphQL.com or any of the GraphQL books available on the market.

What's Not in the GraphQL Specification?

Although the GraphQL specification is very particular about defining the syntax of queries and how the server should execute and respond to queries, the specification is not particular about the following:

- How queries are passed to GraphQL. Often, queries are routed through an HTTP server like Express, but they could also be passed to GraphQL through a Functions as a Service (FaaS) platform like AWS Lambda.
- Which language the servers and clients are written in. There are at least a dozen (and counting!) language-specific GraphQL server implementations and a few dozen client implementations. So long as the GraphQL specification is adhered to, you could build a client or server in any language.
- How multiple GraphQL schemas are combined (schema stitching or schema federation).
- Where the data is and how it's retrieved by the GraphQL resolvers.

- Anything related to security. Authentication, authorization, limits, and so on are all completely beyond the scope of the GraphQL specification.
- Caching, whether on the client side or the server side.

These topics are where the various GraphQL implementations can differentiate themselves. Commercial vendors, especially, are building entire companies around solving many of these problems.

GraphQL Specification Governance and History

The GraphQL specification and GraphQL.js reference implementation were first published externally by GraphQL's creators at Facebook in July 2015. The specification has been updated once or twice each year since then.

Originally the specification was posted to Facebook's GitHub, but in 2018, Facebook turned the governance and hosting of the specification and the reference implementation over to the GraphQL Foundation (as discussed in Chapter 1). Even though Facebook is a member of the GraphQL Foundation, it has as much say over the future of the specification as other members, which include Apollo, AWS, IBM, PayPal, Twitter, and more.

Principles of the GraphQL Specification

GraphQL was designed with a strict set of principles in mind. Let's examine them here.

Evolvable

With change being the only constant in software development, it's inevitable that your GraphQL schema will need to change. Maybe you add a new product type, add a new promotion attribute, or must deprecate a payment method. Your GraphQL schema will change.

When designing GraphQL, its founders strongly opted for evolving schemas rather than versioning them. Evolving schemas means adding nonbreaking changes to the schema, thereby slowly evolving it over time. The GraphQL specification says:

> GraphQL services and schema maintainers are encouraged to avoid breaking changes; however, to be more resilient to these breaking changes, sophisticated GraphQL systems might still allow for the execution of requests which at some point were known to be free of any validation errors, and have not changed since.

Versioning, as is often practiced when designing REST APIs, requires that clients specify the version of the REST API in the HTTP request. Clients make requests to *https://api.domain.com/v1.1/product* or v2, or whatever version they want. But the client must keep track of which version it needs to call, and the producer of the API needs to run multiple versions of the application concurrently, each with a different API. For more information about versioning versus evolving APIs, have a look at *APIs for Modern Commerce* (O'Reilly, 2017). Although it's specific to REST APIs, the same principles hold true for GraphQL.

When fields do need to be deprecated, GraphQL offers the `@deprecated` directive as follows:

```
type Category {
  displayName: String
  name: String @deprecated(reason: "Use `displayName`.")
}
```

In this example, clients would be gradually moved from using the `name` field to the `displayName` field. After it's confirmed (via usage monitoring) that all clients have moved, the GraphQL schema definition can be updated to drop the deprecated field entirely. Some commercial GraphQL server vendors have features that allow you to replay all transactions from a past period of time (usually a few days) against your changed GraphQL schema to make sure nothing broke. These tests can even be integrated into your continuous integration (CI) pipeline with every change to the schema.

You can avoid many schema changes by putting ample time and thought into naming your types and fields properly. Rather than a field name like `quantity`, use a name like `quantityAvailableToSell`. If you later need to add a different type of quantity, such as `quantityFromDropShipper`, you can do that without having to rename one or more fields.

Data Oriented

Within any commerce platform, you have two types of operations:

Create/read/update/delete (CRUD) data operations
: Querying products, updating inventory, and so on.

Execution of functionality
: Adding to a shopping cart, creating customers, and so forth.

GraphQL was built for CRUD operations and therefore easily handles them. It is fundamentally a data access layer.

GraphQL wasn't built for the execution of functionality. Adding to a shopping cart, creating a customer, executing a search, and so on all require datastore-level CRUD operations but also the execution of a lot of business logic. Even though technically you could put that business logic in your GraphQL server, it was never intended to be there and shouldn't be there. GraphQL is a data access layer, not a platform for executing business functionality.

Client Centric

GraphQL is unapologetically built for frontend developers who are building experiences for clients. The first sentence of the actual specification (June 2018 version) says:

> 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.

The server side is obviously important, but GraphQL was built for and by frontend developers. Most of the GraphQL ecosystem, including Facebook's reference implementation, which underpins many of the commercial and open source offerings, is written in JavaScript. Facebook also developed and released the wildly popular React frontend framework alongside GraphQL, with the two often being used together.

Strongly Typed

A defining characteristic of the GraphQL specification is that it's strongly typed, meaning that the author of the GraphQL schema precisely defines the types (objects like customers, orders, and products), fields (attributes like firstName, shippingAddress, and displayName), and data integrity constraints (data types, whether fields are required, etc.).

Here is a very simple example of how you'd define an `Order` type in GraphQL:

```
type Order {
        date: Date!
        orderTotal: Float!
        products: [Product]
}
```

If you tried to set `orderTotal` as a String like `foo`, you'd immediately get an error. That's strong typing.

In JavaScript, you wouldn't even be able to statically define a type. You'd construct an `order` object on the fly:

```
var order = {
        date: "2019-09-05T09:22:16Z",
        orderTotal: 86.82,
        products: ["p96698497", "p49360022", "p87891741"]
};
```

In this example, you could set `orderTotal` to `foo` and JavaScript would still allow it because there's nothing that says it can't be a String. Strong typing allows for query validation, introspection, autodocumentation, and a host of other benefits.

GraphQL Terminology

Before we get too far, we need to explain in more detail some GraphQL terms that you've seen earlier.

Types

A *type* in GraphQL is an object, like a customer, order, or product. It's a "thing"—a noun.

Here's an example:

```
type Product {
        name: String!
        description: String!
        price: Float!
        ...
}
```

In this example, `Product` is the type. Types are the building blocks of GraphQL.

When modeling types, don't fall into the trap of mapping your REST APIs each to their own type. REST APIs, backed by microservices, are often extremely limited in scope to one type of data and/or one piece of functionality. Because separate teams build each microservice, and one of the central tenets of microservices is to avoid dependencies between teams, you are unlikely to find too many relationships between REST APIs. REST APIs are typically designed so that they are entirely self-contained, with few or any links to other REST APIs. The whole point of GraphQL, however, is to allow frontend developers to traverse a graph of connected objects. When you design your types, make sure that you have as many type-based references as possible. For example, you'd want your customer type to refer to your orders type, and your order type to refer to your customer type:

```
type Customer {
        orders: [Order!]!
}
type Order {
        customer: Customer!
}
```

Developers then can traverse the graph by referencing `customer.orders` or `order.customer` in their code.

Fields

A *field* is an attribute like name, description, price, and so on. Fields belong to types. Here's an example of some of `Product`'s fields:

```
type Product {
        name: String!
        description: String!
        price: Float!
        parentCategory: Category!
        reviews: [Review]
        ...
}
```

Following the name of the field, you'll find its data type. Data types can be scalars (primitives) of the following type:

- Strings (`String`)
- Integers (`Int`)

- Floats (`Float`)
- Booleans (`Boolean`)
- Unique identifiers (`ID`)

In the previous example, you can see that `name` and `description` are `String`s, whereas `price` is a `Float`.

Fields can also have a data type of another type within the same GraphQL schema. In the previous example, the `parentCategory` field points to a `Category` type. When you build your query, you can retrieve field values from the referenced types:

```
query {
        product(id: "94695736") {
                category {
                        name
                }
        }
}
```

This would return something like the following:

```
{
        "data" {
                "product": {
                        "category": {
                                        "name": "Men's Belts"
                        }
                }
        }
}
```

Fields can be marked as required by adding an exclamation point after the data type. Here's an example of a required field:

```
type Product {
        name: String!
}
```

Here's an example of an optional field:

```
type Product {
        reviews: [Review]
}
```

Rather than having a singular value, GraphQL also supports fields having multiple values through the use of brackets ([]). A field defining a single String would be represented as follows:

```
type Product {
        review: Review
}
```

This singular field calls for one `Review` type.

To allow mutliple reviews, you'd simply wrap `Review` in brackets:

```
type Product {
        reviews: [Review]
}
```

Now your clients can iterate through multiple reviews, retrieving only the values from each review that are necessary to render the page.

Arguments

You can pass arguments to any GraphQL operation, and they are often used for retrieving specific nodes in the graph. Going back to the example we've been using in this chapter, the query begins with an argument to retrieve the `product` whose ID is `"94695736"`:

```
query {
        product(id: "94695736") {
                name
        }
}
```

Notice how the argument is named, unlike in many other programming languages. In most others, you'd pass in `"94695736"` but GraphQL requires both the parameter name and value, which in this case is `id: "94695736"`

You can pass in as many arguments as you'd like. Multiple arguments are often passed in for pagination, or for mutations. Here's how you'd create a new product and then retrieve its ID and name after it has been created:

```
mutation {
  createProduct(
        name: "The Disney Castle",
        description: "Welcome to the magical Disney Cas-
tle!...."
  ) {
    id
    name
  }
}
```

The data type of the argument(s) can be a scalar or even a type.

Arguments do not need to be hardcoded.

Variables

GraphQL allows you to pass in named variables to queries. Variables are prefixed with a `$` and are immediately proceeded by the data type and then an exclamation point if it's required. Queries can have an unlimited number of variables. Here's an example of a query with two variables:

```
query orderHistory ($id: ID! $year: Int){}
```

This query requires the customer's ID as the `id` variable and then optionally accepts `year` as an input.

All GraphQL servers allow you to pass key/value inputs, with the key being the name of the variable and the value being the value of the variable.

```
{
  "id": "12345",
  "year": 2019
}
```

If you're accessing GraphQL through an HTTP server, you can pass in variable names/values as HTTP GET arguments.

Fragments

Sometimes, GraphQL queries can become repetitive by having to call out the same fields over and over. Let's take an address, for example. Retrieving an address would look something like this:

```
query {
        customer(id: "47937102") {
                firstName
                lastName
                addresses {
                        type
                        address1
                        address2
                        address3
                        city
                        state
                        zip
                        country
                        phone
                }
        }
}
```

Rather than typing those field names throughout your queries, you can define a fragment as follows:

```
fragment addressInfo on Address {
        type
        address1
        address2
        address3
        city
        state
        zip
        country
        phone
}
```

Fragments can belong to only a single type, in this case the `Address` type:

```
query {
        customer(id: "47937102") {
                firstName
                lastName
                addresses {
                        ...addressInfo
                }
        }
}
```

Wherever you need to retrieve those specific fields from the `Address` type, you can now use `...addressInfo` and spare yourself from having to type all those fields manually. An added benefit of using fragments is that if you add additional fields to types, you can update the fragment directly rather than having to update each occurrence manually.

Interfaces

Like other object-oriented programming languages, GraphQL allows you to define interfaces. An interface is basically a type that can be implemented by other types or queried.

Suppose that you have an order type and want to add subtypes for B2C and B2B orders. Most of the fields are generic enough to belong to the order type, but some are specific to B2C orders and some are specific to B2B orders. Here's an example of what the order type would look like:

```
interface Order {
        date: String!
        products: [Product!]!
        merchandiseTotal: Float!
        shippingTotal: Float!
        orderTotal: Float!
        payment: Payment!
}
```

Then, you'd define new types (in this case `B2COrder` and `B2BOrder`) that implement the `Order` interface:

```
type B2COrder implements Order {
        facebookHandle: String
}

type B2BOrder implements Order {
        approvers: [User]
}
```

In the case of a B2C order, you'll want to capture the customer's Facebook handle and in the case of a B2B order, you'll probably have one or more approvers. These fields are each unique to their specific types of orders, yet the vast majority of fields would be shared by all types implementing the order interface.

The order itself can be queried, with specific fields requested from specific implementations of the `Order` interface (via the `...on` notation):

```
query allOrders {
  orders {
        date
        products {
                name
        }
        ...on B2COrder {
                facebookHandle
          }
        ...on B2BOrder {
          approver {
                firstName
                lastName
          }
        }
  }
}
```

Common examples of interfaces in commerce include orders, products, customers, payment methods, and so on.

Inputs

When working with mutations, the number of inputs can get to be excessive. Going back to the order example, let's define mutations to create B2C and B2B orders:

```
type Mutation {
        createB2COrder(date: String!, products: [Product!]!,
merchandiseTotal: Float!, shippingTotal: Float!, orderTotal:
Float!, payment: Payment!, facebookHandle: String): B2COrder

        createB2BOrder(date: String!, products: [Product!]!,
merchandiseTotal: Float!, shippingTotal: Float!, orderTotal:
Float!, payment: Payment!, approvers: [User]): B2BOrder
}
```

This is ugly already, and in a real production system, an order could have dozens if not hundreds of fields. Orders could be created across many different mutations as well. Every time you update your fields, you don't want to be forced to go through your mutations to make sure you're not forgetting something.

GraphQL allows you to define "input" types to solve this exact problem. The definition of an input looks the exact same as a type definition but with the `type` keyword being changed to `input`:

```
input OrderInput {
        date: String!
        products: [Product!]!
        merchandiseTotal: Float!
        shippingTotal: Float!
        orderTotal: Float!
        payment: Payment!
}
```

You can then pass the `OrderInput` object to your mutations:

```
type Mutation {
        createB2COrder(order: OrderInput!, facebookHandle:
String): B2COrder
        createB2BOrder(order: OrderInput!, approvers: [User]):
B2BOrder
}
```

Your GraphQL schema will be far more readable and maintainable with inputs defined.

GraphQL Operations

Now that we've covered some basic terminology, it's time to cover the four types of operations that GraphQL supports.

Queries

The vast majority of operations executed against a GraphQL server are queries. A query is a simple retrieval of data, analogous to an HTTP GET with REST or a SELECT with a database. Data is not changed, it's simply retrieved. Retrieving data is what GraphQL was built and optimized for.

A query follows this structure:

```
query ProductDetailWebPage { # some query }
```

The `query` keyword is optional because GraphQL assumes everything to be a query by default.

Next, you'll see `ProductDetailWebPage`, which is the name of the query. Naming your queries is optional and helps with the readability of your queries and with debugging. An advantage of naming your queries is that GraphQL will force you to select one query to execute if you pass multiple queries to a GraphQL server.

This wouldn't work, for example:

```
query ProductDetailWebPage { # some query }
query ProductDetailAndroid { # some query }
```

Through your GraphQL IDE or through code, you could have one large document with related queries and then specify which one you'd like the server to execute.

Queries work with GraphQL concepts such as arguments, variables, fragments, and interfaces.

When formatted, queries look almost exactly like JSON, but with only the keys. GraphQL responses perfectly mirror the queries, but with the keys and the values. The types and fields you retrieve are also the types and fields you'll get back. This mirroring on multiple levels was deliberate by the GraphQL authors and has been an integral part of GraphQL's rise in popularity.

Successful responses always contain a `"data"` key as follows:

```
{
        "data" {
                "product": {
                        "brandIconURL": "https://
www.legocdn.com/images/disney_icon.png"
        ...
}
```

Failed queries always begin with an "errors" key, as follows:

```
{
  "errors": [
    {
      "message": "price for product with id 94695736 could not
be fetched.",
      "locations": [ { "line": 6, "column": 9 } ],
      "path": [ "Product", 1, "price" ]
    }
  ]
```

A response could have both "data" and "errors" in the same JSON document.

Mutations

Mutations are changes to data (create, update, and/or delete) followed by an optional query. Think of them as HTTP POST/PUT/PATCH/DELETE methods or INSERT/UPDATE/DELETE commands in a database.

As mentioned earlier in the design principles of GraphQL, GraphQL is intended to be a data access layer. It is not intended to be a layer that contains much, if any, business-level functionality. Many mutations require executing business functionality. For example, when you create a customer, you'll also want to do the following:

- See whether the customer already has an account
- Create a record for that customer in your loyalty system
- Create a record for that customer in your marketing system

And so on.

None of this code should be in GraphQL. The code should be in your microservices layer. Mutations should be used selectively for modifying data that doesn't require much business logic. Product catalog–related data is probably best for mutations, but anything related to customers or orders is probably too complex for GraphQL to handle on its own.

NOTE

This is why datastores are largely void of any business logic. The functionality to put the data into the datastores should reside in the application, not in the datastore or the datastore access layer.

Mutations are structured like queries. Start with the operation name (in this case `mutation`) followed by an optional name for your mutation (in this case `createDisneyCastle`). Then, the underlying mutation (in this case `createProduct`) is invoked. Finally, the mutation calls for the retrieval of the `id` and `name` of the product that was just created:

```
          mutation createDisneyCastle {
            createProduct(
             name: "The Disney Castle",
             description: "Welcome to the magical Disney
    Castle!....",
             price: "349.99") {
                          id
                          name
            }
          }
```

Mutations work with GraphQL concepts such as arguments, variables, fragments, interfaces, and inputs. Inputs can be used only with mutations.

Mutations can easily lead to unintended bulk changes of data. A developer could define a `deleteAllProducts` mutation for testing locally and forget to remove it in production. Unless properly secured by role, a developer could easily invoke this in production.

Subscriptions

Subscriptions are real-time streams of data that allow bi-directional communication over a single Transmission Control Protocol (TCP) socket. Facebook originally built subscriptions to allow its customers to see real-time "Likes" without having to refresh the page.

Suppose that you want to be notified every time your inventory is changed. Your subscription would look something like this:

```
subscription {
  inventoryChange (productId: "94695736", loca-
tion:"FulfillmentCenter32") {
        inventoryCount
  }
}
```

Every time the underlying inventory is changed, the new value for `inventoryCount` will be pushed over a web socket or some other type of persistent connection.

The only way a subscription differs from a query is the use of the operation name `subscription` and the ongoing push nature rather than a one-time pull, as with a query.

Introspection

As discussed earlier, a defining characteristic of GraphQL is that it's introspective, meaning it's possible to query the schema to understand the following:

- What types are available
- What fields are available
- The metadata (data types [whether required or not], arguments, directives, etc.)

With few exceptions, everything in your schema definition can be retrieved by querying the schema:

```
query {
  __schema {
        types {
          name
          description
        }
  }
}
```

Introspection is what allows GraphQL servers to validate queries before they're executed. It also allows for a rich ecosystem of GraphQL tooling, including IDEs.

Final Thoughts

In this chapter, we discussed the GraphQL specification and its history, core architectural principles, and terminology, as well as the operations it supports. It should be clear by now how well thought out GraphQL is and how much Facebook improved it between its internal launch in 2012 and its public release in 2015.

Next, we explore clients.

CHAPTER 3
GraphQL Clients

A GraphQL client is software that runs in each client (web browser, native application, etc.) that handles the life cycle of connecting to the GraphQL server, executing queries, and receiving responses.

Technically, *any* code that allows you to make an HTTP request is a client, from cURL on the command line to any of the thousands of JavaScript libraries that are available. Every programming language, framework, operating system, and so on has the means of making HTTP requests because HTTP-based API calls underpin modern IT.

NOTE Even though GraphQL is technically agnostic to the underlying transport protocol, HTTP is the only one you'll see out in the real world.

Formal GraphQL clients offer basic HTTP request handling plus the following:

- Low-level networking
- Batching
- Authentication
- Caching
- Language-specific bindings
- Frontend framework integration

Let's explore each one of these in greater depth.

Low-Level Networking

Making an HTTP request sounds simple, but things can go wrong in the long, complicated journey from a client to the GraphQL server and back. The GraphQL server might not respond, the response might be slow, there might be errors calling the underlying datastore, the response size could be very large, and so on. The real world is full of problems. The networking stack of your client can greatly help with these issues by allowing you to configure the following:

- Retry policies (how many times to retry, how long between each retry, etc.)
- Limits on response sizes
- Timeout limits

Depending on your client, you can even swap out HTTP for another protocol.

Batching

The entire point of GraphQL is to allow your clients to retrieve everything needed to render a page or other experience with one request. In an ideal world, you'd have one HTTP request per page.

Frontend frameworks like React encourage modularization. Each "component" in React should be more or less self-contained, including fetching data. Here's an example of a very simple component:

```
import React from 'react'
import {Query} from 'react-apollo'
import gql from 'graphql-tag'

const Price = () => (
  <Query
    query={gql`
      {
                  product(id: $id) {
                          price
        }
      }
    `}
  >
```

```
      {({ loading, error, data }) => {
        if (loading) {
         return "Loading..."
        }
        if (error) {
         return "Error!"
        }

        return "{data.product.price}"
      }}
    </Query>
  )
```

Imagine having a product detail page composed of 10 of these components, each with their own GraphQL query. Even though this makes the frontend code more manageable from a development standpoint, you're back to the same problem as REST APIs.

Some clients allow you to batch together multiple GraphQL queries so that you can still have the modularity, but multiple queries are sent to the GraphQL server in one batch. The client will gather all of the GraphQL requests over a period of tens of milliseconds and then submit one request to the server.

Authentication

All GraphQL clients require that you authenticate with the GraphQL server before being able to execute queries. Because GraphQL is served almost exclusively over HTTP, the authentication scheme you've been using to secure your REST APIs (typically OAuth 2.0) can easily be reused for GraphQL. Chapter 4 examines this in greater depth.

Authentication typically requires inserting an HTTP header, as follows:

```
const httpLink = new HttpLink({
        uri: "https://api.myserver.com/graphql",
        headers: {
                authorization: `Bearer ${token}`
        }
});
```

All HTTP clients, whether GraphQL focused or not, allow you to insert custom HTTP headers.

Caching

Most of the requests handled by a GraphQL server are for queries. As discussed in Chapter 1, queries simply *retrieve* data; they do not change it. Therefore, many of these queries can be cached locally.

One of the advantages of REST is that it uses the HTTP ecosystem around caching. You could make an HTTP GET to */ProductCatalog/Product/12345* with an HTTP request header of "max-age=180" and your client (or an intermediary) could cache the results. Every piece of software touching the HTTP request between the client and server knows how to handle that HTTP request. With GraphQL, all operations are typically submitted over HTTP GET or POST (though, as we'll discuss later, GraphQL is transport layer agnostic). HTTP is used as a tunneling mechanism and you're not able to use the native verbs and caching mechanisms. It's a different paradigm entirely.

Let's go back to the very first GraphQL response from Chapter 1:

```
{
        "data" {
                "product": {
                        "brandIconURL": "https://
www.legocdn.com/images/disney_icon.png",
                        "name": "The Disney Castle",
                        "description": "Welcome to the magical
Disney Castle!....",
                        "price": "349.99",
                        "ableToSell": true,
                        "averageReview": 4.5,
                        "numberOfReviews": 208,
                        "maxAllowableQty": 5,
                        "images": [
                                { "url": "https://
www.legocdn.com/images/products/94695736/1.png", "altText":
"Fully assembled castle" },
                                { "url": "https://
www.legocdn.com/images/products/94695736/2.png", "altText":
"Castle in the box" }
                        ]
                }
        }
}
```

In this example, the fields `brandIconURL`, `name`, `description`, `max AllowableQty`, and `images` are unlikely to change frequently and can therefore be cached. However, `price`, `ableToSell`, `averageReview`, and `numberOfReviews` are likely to change constantly.

Depending on which client you use, you can specify various cache directives for both the type and the field.

NOTE Caching is entirely absent from the GraphQL specification.

Here's how you'd cache an entire image type for a whole day using the Apollo GraphQL implementation:

```
type image @cacheControl(maxAge: 86400) {
        id: ID!
        url: String!
        altText: String!
}
```

Here's how you'd cache individual fields within a type:

```
type product {
        id: ID!
        brandIconURL: String @cacheControl(maxAge: 300)
        name: String! @cacheControl(maxAge: 300)
        description: String! @cacheControl(maxAge: 300)
        price: Float!
        ableToSell: Boolean!
        averageReview: Float
        numberOfReviews: Int
        maxAllowableQty: Int @cacheControl(maxAge: 300)
        images: [Image] @cacheControl(maxAge: 300)
}
```

When a client sees a `cacheControl` directive, it serves up the data from its local client-side cache. If it doesn't have the data, it then must call the GraphQL server. The mechanics of caching are entirely dependent upon the client and server vendor, and the client/server often need to work together. As mentioned earlier, caching isn't in any way part of the formal GraphQL specification. The previous examples were all from the Apollo GraphQL implementation, though other implementations are similar.

It's extremely unlikely that entire GraphQL queries can be cached for commerce. There's always at least one field or type that can't be cached. Therefore, you really need to use an actual GraphQL client, rather than a traditional HTTP client.

Language-Specific Bindings

Frontend developers want to be able to call the GraphQL server using the programming language in which they're writing the clients. JavaScript developers want a GraphQL client written in JavaScript. Swift developers want a GraphQL client written in Swift. And so on. Even within the programming languages, there are "flavors." In the JavaScript world, there's Angular, Vue, Meteor, Ember, and others.

A good client offers the same functionality regardless of the programming language the developer chooses.

Frontend Framework Integration

There are entire frontend frameworks built around GraphQL that include GraphQL client functionality. For example, Facebook's Relay/Relay Modern are built entirely around React and GraphQL.

Although GraphQL is fast, there's still a few hundred millisecond delay before data in a user interface (UI) is retrieved and rendered. Popular GraphQL clients can display placeholder data while the actual data is being retrieved. Figure 3-1 presents an example from Facebook, which is using Relay Modern.

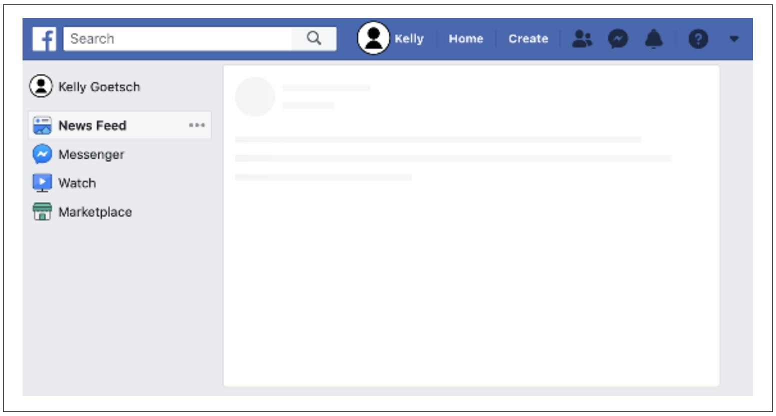
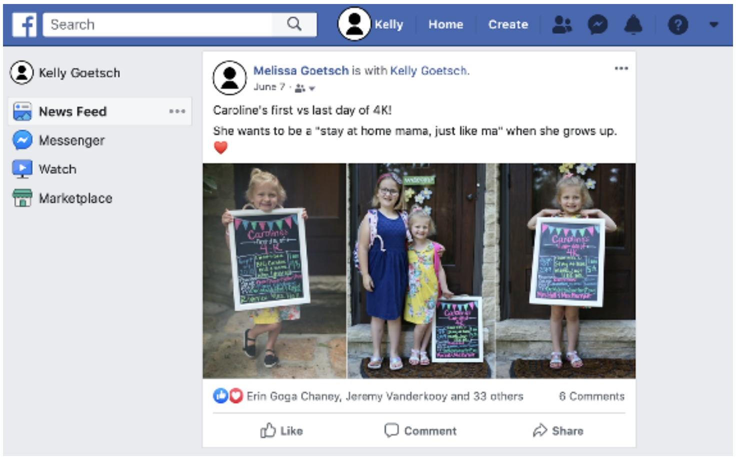


Figure 3-1. Progressive rendering

After the content is retrieved over GraphQL and rendered, the screen looks as depicted in Figure 3-2:



Melissa Goetsch is with Kelly Goetsch.
June 7 ·

Caroline's first vs last day of 4K!

She wants to be a "stay at home mama, just like ma" when she grows up.

Erin Goga Chaney, Jeremy Vanderkooy and 33 others 6 Comments

Figure 3-2. Fully rendered

Rendering the UI and then painting the data on it dramatically improves the perceived performance.

These frameworks can also get creative with how mutations are handled from a user experience (UX) standpoint. Rather than executing the mutation and repainting the entire page, these frameworks make it possible to update the UI first and then asynchronously execute the mutation against the server.

Final Thoughts

In this chapter, we explained what GraphQL clients are, why they're better than clients that just handle HTTP requests, and what value specifically GraphQL clients offer.

In Chapter 4, we discuss GraphQL servers.

CHAPTER 4

GraphQL Servers

At a high level, a GraphQL server is responsible for responding to queries from clients. It's typically fronted by an HTTP server and listens at *https://api.myserver.com/graphql*. Clients (whether GraphQL or otherwise) make requests to that endpoint, and the server responds.

A GraphQL server is composed of two parts: an HTTP server and a GraphQL engine, as shown in Figure 4-1.

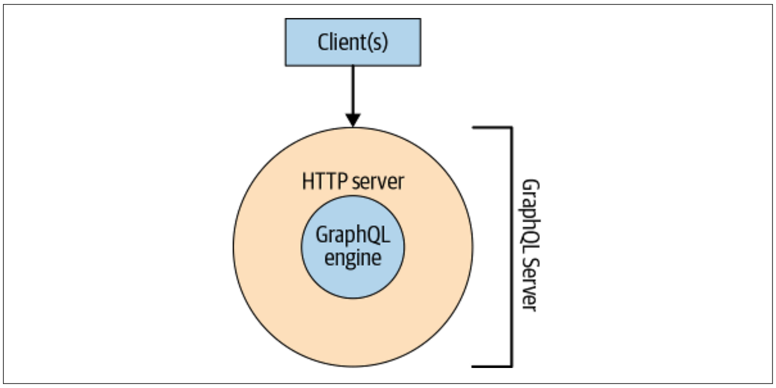


Figure 4-1. GraphQL server

The core GraphQL engine accepts the schema definition upon instantiation, builds the type schema, and allows you to execute queries against that schema. This is a library of code implemented in many common programming languages.

The HTTP server accepts the GraphQL queries and then passes them to the core GraphQL engine. When the engine responds, the HTTP server then passes the JSON response back to the client.

Let's discuss this in more detail.

Building a Type Schema

The GraphQL schema definition is purely some text. It's like code that hasn't been compiled yet. It's pretty useless on its own sitting in a text file or as a string in memory somewhere. For the GraphQL schema to be of any use, it must be loaded into a GraphQL server upon instantiation. Here's how you'd do it with GraphQL.js, which is Facebook's reference implementation and the near standard:

```
var { graphql, buildSchema } = require('graphql');

// Construct a schema, using GraphQL schema language
var schema = buildSchema(`
        type Product {
                id: ID!
                brandIconURL: String
                name: String!
                description: String!
                price: Float!
                ableToSell: Boolean!
                averageReview: Float
                numberOfReviews: Int
                maxAllowableQty: Int
                images: [Image]
        }
`);

// The root provides a resolver function for each API endpoint
var root = {
  id: () => { return '94695736'; },
  brandIconURL: () => { return 'https://www.legocdn.com/images/
disney_icon.png'; },
  name: () => { return 'The Disney Castle'; },
  description: () => { return 'Welcome to the magical Disney
Castle!....'; },
  price: () => { return 349.99; },
  ableToSell: () => { return true; },
  averageReview: () => { return 4.5; },
  numberOfReviews: () => { return 208; },
  maxAllowableQty: () => { return 5; },
  images: () => { return [ {url: 'https://www.legocdn.com/
images/products/94695736/1.png', altText: 'Fully assembled cas
tle'}, {url: 'https://www.legocdn.com/images/products/
```

```
    94695736/2.png', altText: 'Castle in the box'} ] },
    };
```

(based somewhat on the example found in the GraphQL.js tutorial).

In this very simple example, we instantiated a GraphQL server with a single type defined. We cover this shortly, but each field must have a corresponding "resolver," which is a function that provides the value for the field. In this example, the data is hardcoded but in real-world code you'd call the various REST APIs to retrieve the data needed for each field.

At this point, the server is instantiated but is not yet ready to accept queries.

HTTP Request Handling

The GraphQL engine is responsible for building the type schema and parsing/validating/executing queries, but it doesn't provide the functionality required to accept and respond to queries. You need a way to pass queries to the library and for the responses to be passed back to the clients given that the clients are always physically separated from the servers. Although you could front your GraphQL engine with anything that can be used to communicate with a client, an HTTP server makes the most sense and is by far the default within the GraphQL community.

NOTE

The GraphQL specification doesn't include anything in front of the GraphQL engine. HTTP servers, authentication, authorization, caching, and so on are all completely beyond the scope of the specification.

An HTTP server automatically gets you the following:

- Authentication
- Authorization
- Protection from malicious queries
- Monitoring
- Metrics collection
- Health checking

And so on. There are hundreds of feature-rich, mature HTTP servers available, so it makes sense to take advantage of that ecosystem to front your GraphQL engine.

Most GraphQL servers are written in JavaScript and therefore require a JavaScript-based HTTP server such as Express, Koa, and Hapi. You can easily embed these HTTP servers in your application. You can also layer your HTTP servers with other intermediaries that are capable of working with HTTP. For example, you could put AWS Elastic Load Balancer (ELB) in front and have that route HTTP requests down to Express. AWS ELB could have the logic for security, monitoring, and more, and Express could serve as a pass-through to the GraphQL engine.

By custom, GraphQL is exposed as */graphql*. All requests are posted to that single URI, typically as HTTP POST, though HTTP GET is often used, as well.

Here are the variables that you'll need to post:

query
: The actual query, like "`{product(id: $id) {price}}.`" This is required.

operationName
: The name of the query to execute, in case there are multiple queries provided. Recall that in Chapter 2 we discussed that multiple queries are possible in the same string. This is required only if multiple queries are provided.

variables
: A map of key/value pairs that are used as variables.

Here's an example of what you'd post to a */graphql* URI over HTTP POST:

```
{
  "query": "{product(id: $id) {price}}",
  "variables": { "id": "94695736"}
}
```

Next, it's time for the GraphQL engine to parse the query.

Parsing Queries

Upon receiving a query, the GraphQL engine immediately parses it to an abstract syntax tree (AST). An AST is basically a parsed version of the query with extra metadata around arguments, data types, location of code, and so on. Visually, an AST looks like a graph, as demonstrated in Figure 4-2.

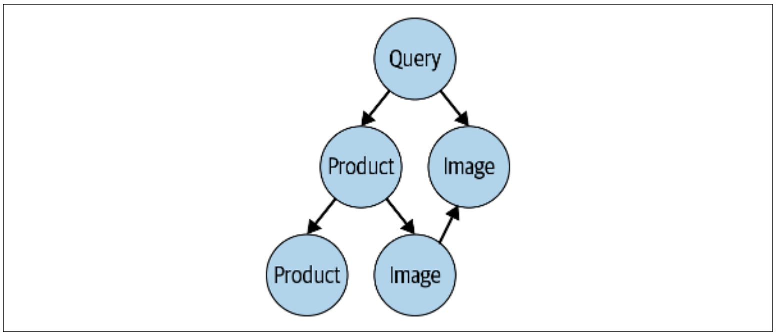


Figure 4-2. AST

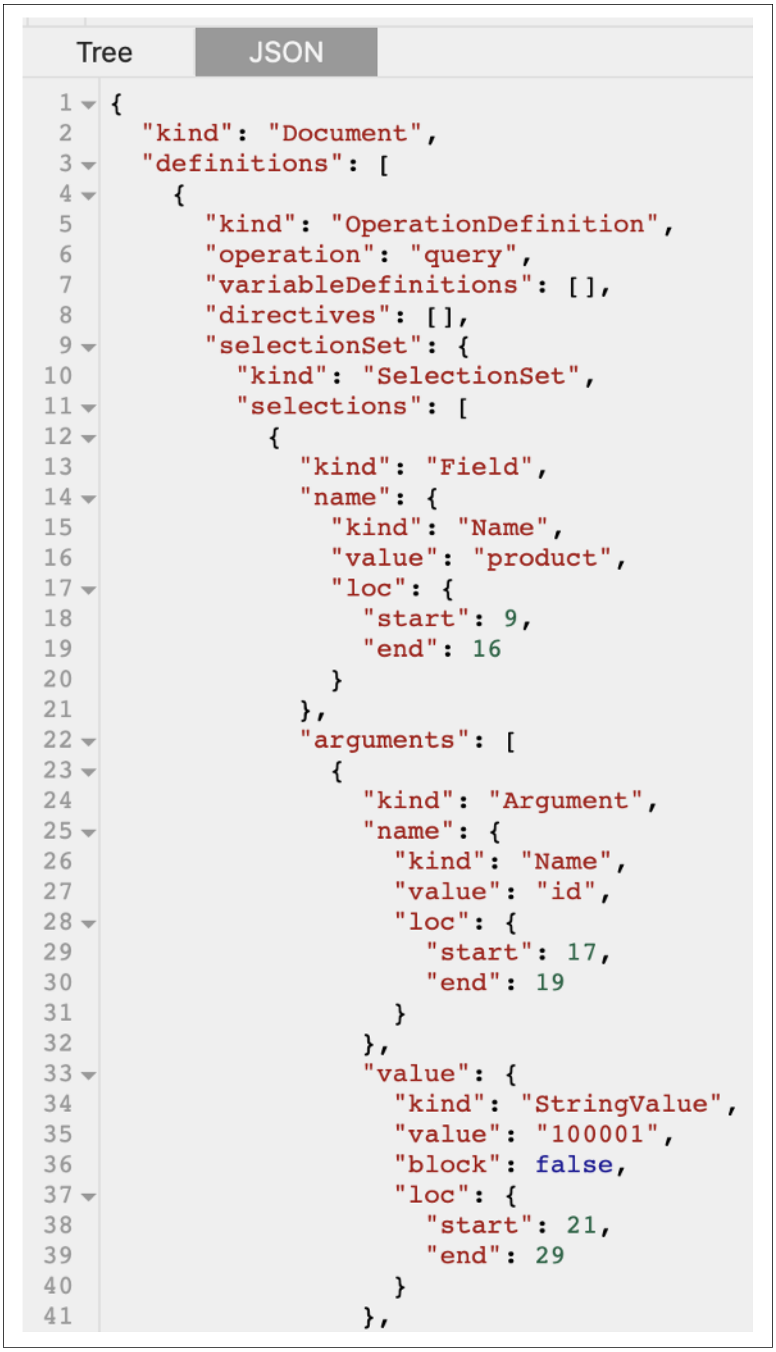
Conversion to AST is required by the GraphQL specification.

A very simple GraphQL query looks something like this:

```
query {
        product(id: "100001", locale: "en_US") {
                name
        }
}
```

In an AST, it would be represented as shown in Figure 4-3.

The full AST representation of this simple three-line query is 120 lines long. Conversion to an AST is something that the GraphQL server automatically does. You will never need to see or deal with an AST. It's strictly internal to the GraphQL server.



```
{
  "kind": "Document",
  "definitions": [
    {
      "kind": "OperationDefinition",
      "operation": "query",
      "variableDefinitions": [],
      "directives": [],
      "selectionSet": {
        "kind": "SelectionSet",
        "selections": [
          {
            "kind": "Field",
            "name": {
              "kind": "Name",
              "value": "product",
              "loc": {
                "start": 9,
                "end": 16
              }
            },
            "arguments": [
              {
                "kind": "Argument",
                "name": {
                  "kind": "Name",
                  "value": "id",
                  "loc": {
                    "start": 17,
                    "end": 19
                  }
                },
                "value": {
                  "kind": "StringValue",
                  "value": "100001",
                  "block": false,
                  "loc": {
                    "start": 21,
                    "end": 29
                  }
                },
```

Figure 4-3. An AST representation of a sample query (from https://astexplorer.net)

Validating

Next, the GraphQL server takes the newly generated AST and checks it for errors. The GraphQL specification is very particular about validation. It starts out the validation section by stating the following:

> GraphQL does not just verify if a request is syntactically correct, but also ensures that it is unambiguous and mistake-free in the context of a given GraphQL schema.

Common errors include:

- Referencing types or fields that don't exist
- Missing required parameters or fields
- Data in the wrong format (e.g., a `String` when the schema called for a `Float`)

The GraphQL specification is explicit in stating that only queries that have been fully validated should be executed.

NOTE

This is another example of how GraphQL is different from REST. Unless you implement custom validation for each REST API, most applications will execute any request that passes XML or JSON validation.

Executing Queries

Now that the server has been started and the query has been parsed and validated, it's time to actually execute it.

The GraphQL server starts by crawling the AST it produced earlier. It then executes what are called "resolver" functions for each type and field to retrieve the data from the underlying source. Finally, the GraphQL server constructs a single JSON response that is passed back to the HTTP server in front.

Let's spend some time on resolvers because they're really the heart of GraphQL. A resolver is the function that calls the underlying REST APIs, databases, legacy backends, or any other source of data.

Resolvers are what actually call the underlying source of data and return the data in the proper format. They shouldn't have any business logic, because GraphQL is strictly an intermediary.

Here's an example of a simple query:

```
query {
        product(id: "94695736", locale: "en_US") {
                name
        }
}
```

And here's a simple response:

```
{
        "data" {
                "product": {
                        "name": "The Disney Castle"
                }
}
```

In the GraphQL server, there's a resolver function for the "name" field that returns the actual name of the product. Here's what that function would look like in GraphQL.js, the JavaScript-based GraphQL reference implementation from Facebook:

```
name(obj, args, context, info) {
        if (context.product == null) {
                fetch('https://api.myserver.com/product/
94695736')
                        .then(resp => resp.json())
                        .then(context.product = resp)
        }
        return context.product.name;
)
```

In this example, `name` is the field name, `obj` is the parent object (in this example, the query), `args` are any arguments provided to the field (like `(locale='en_US')`), and `context` is a catch-all object that can be used to store long-lived objects of value to other resolver functions. You wouldn't want to call a REST API for every field.

Server Implementations

GraphQL is a specification, not a specific implementation. Anyone can write a GraphQL server using whatever programming language and implementation methodology, so long as it adheres to the specification. As we've discussed, the GraphQL specification is fairly silent on how the internals of a GraphQL server should work.

Much of the GraphQL community uses GraphQL.js either directly or indirectly as the GraphQL server. GraphQL.js was released by Facebook (along with the original specification) in 2015 and is actively maintained by a large community of individual and corporate contributors. The entire Apollo ecosystem, which is the largest and most popular within the GraphQL community, is built around GraphQL.js. There really aren't any other JavaScript-based implementations of GraphQL servers, and with JavaScript being the default client- and server-side programming language in the GraphQL community, GraphQL.js is the de facto standard.

Many organizations do not run much or any JavaScript on the server side, yet they still choose GraphQL.js because of how strong of a GraphQL server implementation it is. Remember, there shouldn't be any business logic in your GraphQL server. GraphQL is a layer over the top of your REST APIs, databases, legacy backends, or any other source of data. Most of the time, you call an API, parse the results, and return the data specified by each type or field. The programming language you use is mostly irrelevant. What matters is that you're able to find developers and that you're supported by a rich ecosystem of tooling. JavaScript checks both of those boxes.

If you want a non-JavaScript GraphQL server, there are implementations available in many programming languages including Scala, Go, Java, Ruby, Python, Haskell, and more.

When possible, it's best to use a GraphQL client and server provided by the same vendor because of the additional functionality provided outside of the GraphQL specification. Remember that caching, security, monitoring, testing, and other aspects are completely beyond the scope of the GraphQL specification. Many of those features require making changes to both the client and the server, with both pieces working together. Furthermore, some frontend frameworks like Relay require that the GraphQL server have additional functionality that's outside the GraphQL specification. For example, users of Sangria, the GraphQL implementation written in Scala, must add an additional library to support Relay.

Now that we've covered the fundamentals of GraphQL servers, let's discuss some of the additional features that are available in various implementations.

Monitoring

Like all services in production, you must monitor GraphQL for availability, functionality, and performance. Unlike REST APIs, for which each endpoint can be monitored separately, all GraphQL requests are GETs or POSTs to */graphql*. A query can be for one field or for an entire product catalog.

The key to monitoring GraphQL is to monitor your resolver functions because that's where the real work happens. The GraphQL server itself is very unlikely to fail. Within each resolver function, it's best practice to log a correlation ID, the operation type, and the query complexity (which we discuss soon) to a log file and/or a time series database like InfluxDB or Prometheus. You then can layer an analytics platform like Grafana on top to aggregate and analyze the data.

Per the GraphQL specification, GraphQL servers execute resolvers concurrently in the case of queries, and sequentially in the case of mutations. Therefore, the performance of any given GraphQL query is a function of the slowest resolver and the performance of any given GraphQL mutation is the sum of all resolvers.

Commercial GraphQL vendors have full monitoring solutions in place already, so it's best to use that functionality if it's available.

Testing

Every time you touch your schema, resolvers, or underlying data sources (REST APIs, databases, legacy backends, or any other source of data) you need to retest your entire GraphQL layer to make sure you didn't introduce any errors.

Fortunately, GraphQL is easy to test in local environments as well as in integration or QA environments. With GraphQL, you have a fixed set of inputs (queries, mutations, and variables) and a fixed set of outputs (in nicely formatted JSON). It's easy to write test cases with real or mocked data that exercise every type and field in your schema. Writing tests should be mandatory for every new type and field introduced to your schema.

If you want to provide your frontend developers with some mocked data so that they can build their frontends as the backend is being built in parallel, it's easy to have each resolver return some mocked data. If you want to test resolver functionality in isolation, outside of the GraphQL server, you can call each resolver independently because they're self-contained functions with a few inputs and a fixed output.

Some commercial GraphQL server vendors have features that allow you to replay all transactions from a past period of time (usually a few days) against your changed GraphQL schema to make sure nothing broke. You can even integrate these tests into your CI pipeline with every change to the schema.

Everyone's testing needs are unique, and there are enough commercial products and open source tools available to meet your needs.

Security

A topic of particular importance to those adopting GraphQL is security. GraphQL's centralization makes security easier (authentication, authorization, etc.) but also more challenging (expensive queries, destructive mutations, etc.).

Let's explore the security-related topics that you'll need to address.

Authentication

Though the */graphql* URI is often publicly available, you don't want anyone to be able to call it. Users should be required to properly authenticate. Authentication ensures that a user, whether a human or another system, is who he/she/it purports to be.

Authentication should be performed in a layer that sits atop your GraphQL server. The HTTP servers embedded within a typical GraphQL server tend to be fairly minimalist and therefore might not support the additional authentication-related features that a more robust HTTP server/load balancer/reverse proxy would be able to support. You also want to shield your GraphQL server from abusive queries and denial of service attacks.

Because GraphQL is served over HTTP, you can take advantage of all of the common authentication schemes and tooling available for traditional REST APIs. See *APIs for Modern Commerce* (O'Reilly 2017) for more information.

Authorization

After you've authenticated your client, you must now authorize that client to call specific operations (query, mutation, subscription, introspection), specific types (products, orders, inventory, etc.) and specific fields (`price, quantity, availableToSell`).

Here are some common business rules you'd want to implement:

- The merchandising team should be able to view only product catalog–related data. Any data related to an individual customer shouldn't be viewable.
- Only connections made from the CSR application should be allowed to add credits to a customer's account.
- Administrators should be able to do anything.
- Nobody should be allowed to call the `deleteAllOrders` mutation in a production environment.

If you recall from earlier in the chapter, each resolver is passed a context object of some sort, which is used to access long-lived objects, database connections, and other objects that are of value to all resolvers. That context object can also be instantiated with a user object. That user object could have the following:

- User ID/name
- Role(s)
- Organization

Within each resolver, you can then apply limited business logic. Here's a very simple example of how you'd prevent merchandising team members from viewing the `products` field of an order:

```
products(obj, args, context, info) {
        if (context.user == null || context.user.role == "mer
chandising") {
                return null;
        }
        return context.order.products;
)
```

If the user isn't attached to context or the user has the wrong role, you could return null (as in this example), an empty value/array, or throw an error to the client. It's up to you.

In this example, we're using GraphQL.js, but the concepts are the same regardless of your GraphQL server implementation.

Expensive Queries

One of the benefits of GraphQL is that it allows you to query and mutate large amounts of data with a single line of text. Can you imagine the amount of work a GraphQL server would need in order to serve this response?

```
query lotsOfData {
        allProducts
        allSKUs
        allCategories
        allCustomers
        allOrders
}
```

That single query would quickly peg any server's CPU. The response size would be well into the gigabytes. If someone were to accidentally run that a few times, it could very quickly bring down an entire GraphQL server.

Another challenge with GraphQL is that its ability to traverse a graph can lead to deep recursions that burn valuable resources. You could define a product with a reference to category and a category with a reference to all of its products as follows:

```
type Product {
        category: Category!
}

type Category {
        products: [Product]
}
```

Now imagine a query like this:

```
query somethingMalicious {
        allProducts {
                category {
                        products {
                                category {
                                        products {
                                                category
                                        }
                                }
                        }
                }
```

```
            }
    }
```

Fortunately, there are well-established ways to protect your GraphQL server from overly complex queries, whether malicious or not.

Query size limits

Before the query even hits your GraphQL server, you can filter out unusually large HTTP requests. Large could be defined as follows:

- Number of characters
- Number of bytes
- Number of unique types and/or fields requested

This filtering can be done in the HTTP server above your GraphQL server.

Clearly this isn't very effective, but queries that are dramatically larger than others should be filtered. You might want to block all HTTP requests that are larger than 250 kilobytes, for example.

Timeouts

Another way to protect your GraphQL server is to kill queries that are taking too long to execute. If a query is running for 10 seconds, something is probably wrong and the query should be terminated.

It's best to terminate long-running queries at both the HTTP server above the GraphQL server as well as within the GraphQL server itself. Depending on the implementation, it is possible to halt the execution of the resolvers, but there might be unexpected errors as connections to data sources are abruptly terminated.

Allowlists

Rather than allow all GraphQL queries, another approach is to create an allowlist of acceptable queries. Only queries on the list are allowed to be executed.

If you have control over your clients, you can run tools like PersistGraphQL to analyze your code and pull out any GraphQL queries. Those queries are then added to the list. If you were to pull up GraphiQL and arbitrarily execute queries, they wouldn't work.

Another option is to watch the queries executed over the course of a week and build an allowlist from that. Subsequent queries not on that allowlist wouldn't be allowed to be executed.

Query depth

Another option for protecting your GraphQL server is to check the depth of your queries before you execute them. Using something like graphql-depth-limit, you can see how many levels deep your queries are.

For example, this query is one level deep:

```
query somethingMalicious {
        allProducts
}
```

This query is two levels deep:

```
query somethingMalicious {
        allProducts {
                category
        }
}
```

And so on.

Part of the value of GraphQL is that you can form these large, nested queries. But there needs to be a limit. A query five levels deep is verging on abusive. A query 10 levels deep is definitely abusive.

An advantage of checking query depth is that it forces your developers to come up with more elegant solutions. Nobody wants to write or debug a query that's five levels deep.

Query complexity/cost

Similar to query depth, you can estimate the cost of a query before it's executed by looking at its complexity. Here are some factors that influence a query's complexity:

- Nesting depth
- How many fields are requested
- How many types are requested

Assuming defaults from graphql-query-complexity, the following query would have a cost score of 2:

```
query {
        product(id: "94695736", locale: "en_US") { # cost=1
                name                               # cost=1
        }
}
```

You could assign a higher weight to specific fields. For example, retrieving a product and its attributes is pretty straightforward. Now let's add in price:

```
query {
        product(id: "94695736", locale: "en_US") { # cost=1
                name                               # cost=1
                description,                       # cost=1
                brandIconURL,                      # cost=1
                price                              # cost=5
        }
}
```

This query comes out with a cost of 9. Price has a higher complexity score because retrieving it requires another call to a different REST API. Network hops are always more expensive and therefore cost more.

You then can set a maximum cost for a query. For example, you could set a cost limit of 100. As with query depth, enforcing a maximum cost forces your developers to come up with more elegant solutions.

Merging Schemas

The greatest value of GraphQL is that there's a single schema for frontend developers to execute queries against. Developers can get any data they want from any backend datasource with one query. The GraphQL schema and the interconnectedness of the types and fields is what enables this. Unfortunately, the GraphQL layer can quickly become a monolith. Imagine having 25 different microservice teams, each trying to contribute to a single GraphQL schema as depicted in Figure 4-4. It quickly becomes complicated. The whole "monolith in the pipes" problem is what led to service-oriented architecture's decline.

Given this tension between centralization and decentralization, how do you allow individual teams to work in parallel while exposing a single cohesive GraphQL schema to your clients?

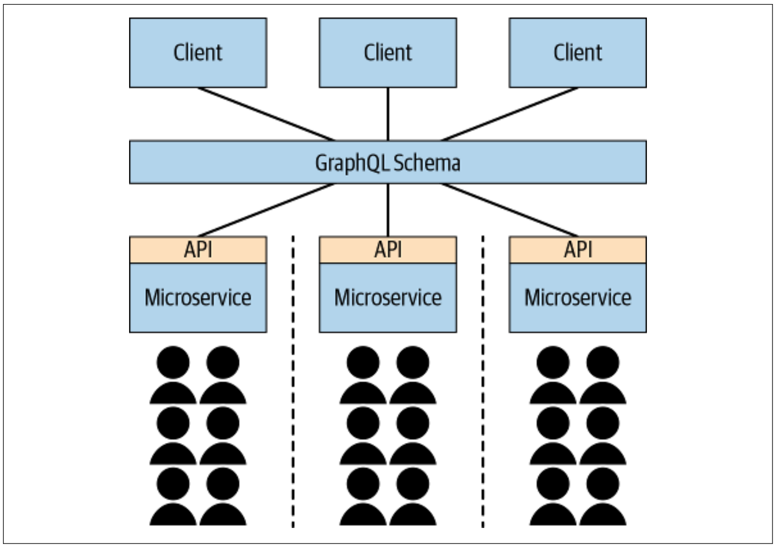


Figure 4-4. Centralization of GraphQL versus decentralization of clients/microservices

Separate Files

A very simple yet effective way of distributing the ownership of your GraphQL schema is to break apart your schema into separate physical files. When you instantiate your GraphQL server, you need only to pass it a string containing your schema definition. That string could be retrieved from 1 file or 100 files, it doesn't matter to your server. It needs a single string. At runtime or at build time, you could easily combine multiple files. Popular libraries like graphql-import can automate this process for you.

In this model, you would have your pricing microservice team own `pricing.graphql`, while your product catalog team would own `product_catalog.graphql`, and so on. Different teams will be touching other team's schema definitions, but at least there's some ownership and physical separation of definitions.

Schema Stitching

To this point, we've only discussed having different microservice teams within an organization contributing to the same schema. With many software vendors exposing their own GraphQL endpoints, your frontend developers could end up having to call different */graphql* endpoints based on what data they want to retrieve. Suppose that you have a commerce platform vendor, a CMS vendor, and some microservices you've built in-house. Each organization exposes its own */graphql* endpoint as follows:

Commerce platform: *http://commerce-platform-vendor.com/graphql*

```
query product {
        product(id: "94695736") {
                displayName
        }
}
```

CMS: *http://cms-vendor.com/graphql*

```
query content {
        content(productId: "94695736") {
                longDescription
                images
        }
}
```

Your own microservices: *http://your-company.com/graphql*

```
query inventory {
        inventory(productId: "94695736") {
                quantity
        }
}
```

Imagine the challenges your frontend developers would have calling each of those endpoints, each with their own authentication and authorization schemes. You might as well go back to calling individual REST endpoints. You can run into the same issue if you have multiple teams within an organization each exposing their own GraphQL endpoint.

With GraphQL schema stitching (again, not part of the GraphQL specification), you can combine those three queries into one:

```
query productDetailPage {
        product(id: "94695736") {
                displayName
        }

        inventory(productId: "94695736") {
                quantity
        }

        content(productId: "94695736") {
                longDescription
                images
        }
}
```

To set it up, you need a GraphQL gateway that exposes its own */graphql* endpoint. That gateway then connects to and merges the schemas from the other */graphql* endpoints.

The big drawback with schema stitching is that you must query each type. You can't have one query that accesses one type (like "product" in this example). Instead, you need to have one query that accesses the `product`, `inventory`, and `productContent` types. Another drawback is that there can be naming conflicts. For example, what happens if two schemas both define a type named `product`?

Schema Federation

Increasingly, schema federation is replacing schema stitching. Schema federation allows multiple teams/vendors to contribute to a single type, so that clients need to query only one type.

Here's an example of how you'd instantiate your Apollo-based GraphQL server with different types from different sources:

```
const gateway = new ApolloGateway({
        serviceList: [
                { name: 'product', url: 'http://commerce-
platform-vendor.com/graphql' },
                { name: 'content', url: 'http://cms-vendor.com/
graphql' },
                { name: 'inventory', url: 'http://your-
company.com/graphql' }
        ]
});

(async () => {
        const { schema, executor } = await gateway.load();
        const server = new ApolloServer({ schema, executor });
        server.listen();
})();
```

You can then query a single type (in this case `product`) and it will magically pull fields from the appropriate types from the appropriate */graphql* endpoints provided the @key and @external directives are properly used:

```
query productDetailPage {
        product(id: "94695736") {
                displayName       # from http://commerce-
platform-vendor.com/graphql
                longDescription   # from http://cms-vendor.com/
graphql
                images            # from http://cms-vendor.com/
graphql
                inventory         # from http://your-
company.com/graphql
        }
}
```

Schema federation at scale is difficult but worth it due to the autonomy it gives each team.

Final Thoughts

By now you should have a firm understanding of the shortcomings of using plain REST for commerce, graphs and how they are used in everyday life, the origins of GraphQL, the GraphQL specification, and how GraphQL clients and servers work.

Adopting GraphQL will give your frontend developers a substantial competitive advantage by allowing them to get exactly the data they want, without the burden of trying to find and retrieve data from various REST APIs. Even though GraphQL is an additional layer to maintain, many will find that not having to support many diverse frontends that rapidly change will far outweigh the overhead incurred by adopting it.

Happy GraphQL'ing!

About the Author

Kelly Goetsch is chief product officer at commercetools. He came to commercetools from Oracle, where he led product management for their microservices initiatives. Kelly previously held senior-level product development and go-to-market responsibilities for key Oracle cloud products representing revenue in the nine-figure range. Prior to Oracle, he was a senior architect at ATG (acquired by Oracle), where he was instrumental to 31 large-scale ATG implementations. In his last years at ATG, Kelly oversaw all of Walmart's implementations of ATG around the world.

Kelly has expertise in commerce, microservices, and distributed computing, having spoken and published extensively on these topics. He is the author of three books: *APIs for Modern Commerce* (O'Reilly, 2017), *Microservices for Modern Commerce* (O'Reilly, 2016), and *E-Commerce in the Cloud* (O'Reilly, 2014).

He holds a bachelor's degree in entrepreneurship and a master's degree in management information systems, both from the University of Illinois at Chicago. Kelly also holds three patents, including one key to distributed computing.

Acknowledgments

Thank you to Tori Hall, Stephanie Sprinkle, Rob Senn, Laura Luiz, Yann Simon, Tamas Piros, and Camilo Jimenez.