GraphQL
OR *BUST*

TO USE IT, OR NOT TO USE IT?
THAT IS THE QUESTION

AUTHORS

TBA

NORDIC **APIS**
nordicapis.com

GraphQL or Bust

To Use it Or Not to Use It? That is The Question.

Nordic APIs

Also By **Nordic APIs**

This release is dedicated to the over 30 thought leaders that have contributed to the Nordic APIs blog over the past few years!

Contents

CONTENTS

Preface: Introduction to GraphQL

Within the last couple of years, there has been a resurgence of discussion around **API design standards** such as REST, gRPC, GraphQL, and many others. While the Representational State Transfer (REST) methodology has been a perfect fit for many web APIs - for the past decade or more in some cases - some developers see operational improvements within nuanced methods that break from these original constructs.
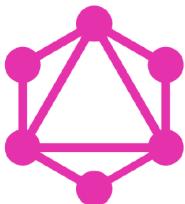
GraphQL, the query language altering how we interact with APIs

Developed by Facebook, **GraphQL** presents a sea change in how API providers enable access to their data, and can bring high usability benefits to consumers. Smart API owners put emphasis into treating their services as a product; this means improving the developer experience and fine-tuning the user onboarding mechanism. As GraphQL enables an unparalleled ability to display API endpoints and test call behaviors, as well as an operational boost in aggregating API responses, it could very well be the tool API owners are searching for to improve their developer experiences.

But we aren't quick to bandwagon on any new technology without first opening the floor to debate. So, over the past year we exhausted many subjects on the blog, throwing GraphQL into the security ring, judging the process of

migrating a REST API to GraphQL, vetting any outsanding licensing issues, and searching for GraphQL APIs in practice today as evidence of its use. We've compared it to other methods of linking API calls, charted industry best practices, and looked at the growing spectrum of GraphQL tooling.

In this volume we've aggregated nearly all of the GraphQL knowledge that has been shared on the Nordic APIs blog and at our conferences. Being a relatively new technology, some may still have questions about it, and we hope to answer those questions as well as open avenues of discussion around new concerns.

The truth is your current systems likely won't *bust* without GraphQL. The title for this eBook represents the fervent community adoption we've seen quickly embrace the technology.

So, please enjoy *GraphQL or Bust*, and let us know how we can improve. If you haven't yet, consider following us, and signing up to our newsletter for curated blog updates and future event announcements. We also accept contributions from the community - if interested please visit our Create With Us page to pitch an article.

Thank you for reading!

    – Bill Doerrfeld, Editor in Chief, Nordic APIs

Connect with Nordic APIs:

Facebook | Twitter | Linkedin | Google+ | YouTube

Blog | Home | Newsletter | Contact

Is GraphQL The End of REST Style APIs?
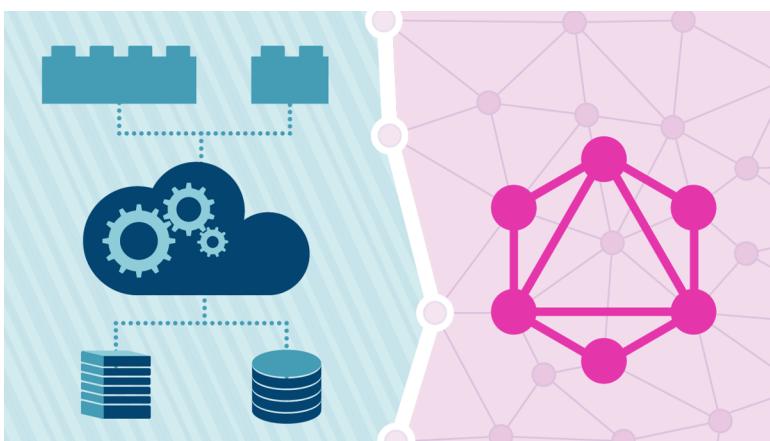
The world of APIs is one of innovation and constant iteration. The technologies that once drove the largest and best solutions across the web have been supplanted and replaced by new, more innovative solutions.

That is why it's surprising, then, that many developers have clung to what they consider the bastions of web API development. Such a bastion is the REST architecture. To some developers, REST is an aging and incompleted proposition that does not meet many of the new development qualifications required by the unique challenges faced by modern development groups.

Today, we're going to look at a technology that is poised to replace, or at the very least, drastically change the way APIs are designed and presented — **GraphQL**. We'll dis-

cuss a little bit of history, what issues REST suffers from, and what GraphQL does differently, tracking a presentation given by Joakim Lundborg at our Platform Summit.

> "The way we design our APIs structures the way we think [about] the tools and applications we build." - Joakim Lundborg, Wrapp

Defining REST and its Limitations

REST or Representational State Transfer, is an API design architecture developed to extend and, in many cases, replace older architectural standards. Objects in REST are defined as addressable **URIs**, and are typically interacted with using the built-in verbs of **HTTP** — specifically, GET, PUT, DELETE, POST, etc. In REST, HATEOAS (Hypermedia As The Engine Of Application State) is an architecture constraint in which the client interacts with hypermedia links, rather than through a specific interface.

With REST, the core concept is that everything is a **resource**. While REST was a great solution when it was first proposed, there are some pretty significant issues that the architecture suffers from. According to Lundberg, the circumstances have changed, giving rise to the need for new technical implementations:

> "Many things have happened. We have a lot of mobile devices with lots of social and very data rich applications being produced…We now have very powerful clients, and we have data that is changing all the time. This brings some new problems."

Here are some issues Lundberg sees with REST:

Round Trip and Repeat Trip Times

REST's defining feature is the ability to reference **resources** — the problem is when those resources are complicated and **relational** in a more complex organization known as a *graph*. Fetching these complicated graphs requires round trips between the client and server, and in some cases, **repeated trips** for network conditions and application types.

What this ultimately results in is a system where **the more useful it is, the slower it is**. In other words, as more relational data is presented, the system chokes on itself.

Over/Under Fetching

Due to the nature of REST and the systems which often use this architecture, REST APIs often result in over/under fetching. **Over fetching** is when more data is fetched than required, whereas **under fetching** is the opposite, when not enough data is delivered upon fetching.

When first crafting a resource URI, everything is fine — the data that is necessary for functionality is delivered, and all is well. As the API grows in complexity, and the resources thus grow in complexity as well, this becomes problematic.

Applications that don't need every field or tag still receive it all as part of the URI. Solutions to fix this, such as versioning, result in duplicate code and "spaghettification" of the code base. Going further, specifically limiting data to a low-content URI that is then extensible results in more complexity and resultant under fetching in poorly formed queries.

Weak Typing and Poor Metadata

REST APIs often unfortunately suffer from **poor typing**. While this issue is argued by many API providers and commentators (often with the caveat that HTTP itself contains a typing system), the fielding system solutions offered simply do not match the vast range and scope of data available to the API.

Specifically, this is an argument in favor of **strong typing** rather than weak typing. While there are solutions that offer typing, the delineation between weak and strong is the issue here, not an argument defused by simply stating "well there *is* typing". The strength and quality of typing *does* matter.

This is more a matter of age and mobility rather than an intrinsic problem, of course, and can be rectified using several solutions (of which GraphQL is one).

Improper Architecture Usage

REST suffers from the fact that it's often used for something it wasn't really designed for, and as a result, it often must be heavily modified. That's not to say that REST doesn't have its place — it's only to say that it may not be the best solution for serving client applications. As Facebook says in its own documentation:

> "These attributes are linked to the fact that "REST is intended for long-lived network-based applications that span multiple organizations" according to its inventor. This is not a requirement for APIs that serve a client app built within the same organization."

All of this is to say that GraphQL is functionally the end of REST — but not in the way that terminology implies. Until now, REST has been seen as the foundational architecture of modern APIs, and in a way, the last bastion of classic API design. The argument here is not made to fully sever REST from our architectural lexicon, but instead to acknowledge that there are several significant issues that are not properly and fully rectified by the solutions often proffered by its proponents. Therefore, the answer to the question of this piece — is GraphQL The End of REST Style APIs? — is quite simple. Yes, using GraphQL is the end of REST style APIs as we know it — specifically through the extension of base functionality and a reconsideration of data relations and functions. ## 4 Things GraphQL Does Better than REST

> "GraphQL declares everything as a graph... You
> say what you want, and then you will get that."

Now that we've seen the issues with REST, how, exactly, does GraphQL solve them?

REST Has Many Roundtrips - GraphQL Has Few

The biggest benefit of GraphQL over REST is the simple fact that GraphQL has **fewer roundtrips** than REST does, and more efficient ones at that. GraphQL unifies data that would otherwise exist in multiple endpoints (or even worse, ad hoc endpoints), and creates packages.

By packaging data, the data delivery is made more efficient, and decreases the amount of resources required for roundtrip calls. This also fundamentally restructures the relationship between client and server, placing more efficiency and control in the hands of GraphQL clients.

REST Has Poor Type Systems - GraphQL Has a Sophisticated One

While REST can have a **type system** through implementations of HTTP, REST itself does not have a very sophisticated typing system. Even in good implementations, you often end up with variants of type settings — for example, *clientdatamobile* and *clientdatadesktop* — to fit REST standard calls.

GraphQL solves this with a very sophisticated typing system, allowing for more specific and powerful queries.

REST Has Poor Discoverability - GraphQL Has Native Support

Discoverability is not native to REST, and requires specific and methodical implementations of HATEOAS, Swagger, and other such solutions in order to be fully discoverable. The key there is "fully discoverable" — yes, REST has HATEOAS as a "native" discovery system, but it lacks some important elements of effective **discoverability** — namely known document structure, server response constraint structures, and an independence from standard, restrictive error mechanisms in HTTP.

While this and many other points of negative consideration towards REST is often answered with "but you can add that functionality!", the fact that it lacks it by default only adds to the complexity we're trying to move away from.

Because GraphQL is based on **relational data** and, when operating on a properly formed schema, is self describing, GraphQL is by design natively discoverable. Discoverability is incredibly important, both in terms of allowing for

extensible third-party functionality and interactions and for on-boarding developers and users with an easy to understand, easy to explore system of functions.

REST Is Thin Client/Fat Server - GraphQL is Fat Client/Fat Server

In REST design, the relationship between client and server is well-defined, but **unbalanced**. REST uses a very **thin client**, depending on processing from the server and the endpoints that have been defined for it. Since the bulk of the processing and control is placed firmly on the **server**, this strips power from the client, and also stresses server side resources. Until now that has been fine, but as devices grow in processing power and ability, this client/server relationship may need rethinking.

GraphQL, however, is different. By offloading specification of expected data format to the client and structuring data around that call on the server side, we have a Fat Client/Fat Server (or even a Thin Client/Thin Server depending on approach) in which both power and control are level across the relationship.

This is very powerful when one considers that the data type being requested will be used for specific purposes as regulated and requested by the Client itself — it makes sense, then, that moving from a Thin/Fat relationship to a Fat/Fat or Thin/Thin relationship would improve this functionality on the Client side while freeing up Server resources. Of course, this assumes that the client is capable of handling this burden.

The End Of The Status Quo

There's a tendency in the tech space for providers and developers of new technologies to proclaim the end of an era with each solution. While it's common to discussion in the field, the fact is that there are very few complete paradigm shifts that signal an irrevocable end to existing technologies.

Innovation depends on prior technologies to create new functionality. Therefore, when a new solution is designed, it's not replacing the solution, but rather iterating. The same is true here. While GraphQL may not be the complete demise of REST, it is the end of the **status quo**. While there are a great many solutions to the issues raised here, they all depend on further integrations and modifications. GraphQL is essentially an overhaul, and one which improves the base level functionality of the API itself.

Conclusion

What we have here is a basic value proposition. GraphQL does what it does well, but the question of integration lies directly on what kind of data you're processing, and what issues your API is creating. For simple APIs, REST works just fine, but as data gets more complex and the needs of the data providers climbs, so too will the need for more complex and powerful systems.

Adopting GraphQL as an adjunct or extension of the REST ideology, while removing REST from the intellectual space of "too big to not use", will directly result in more powerful APIs with easier discoverability and greater manageability

of the data they handle.

5 Potential Benefits of Integrating GraphQL
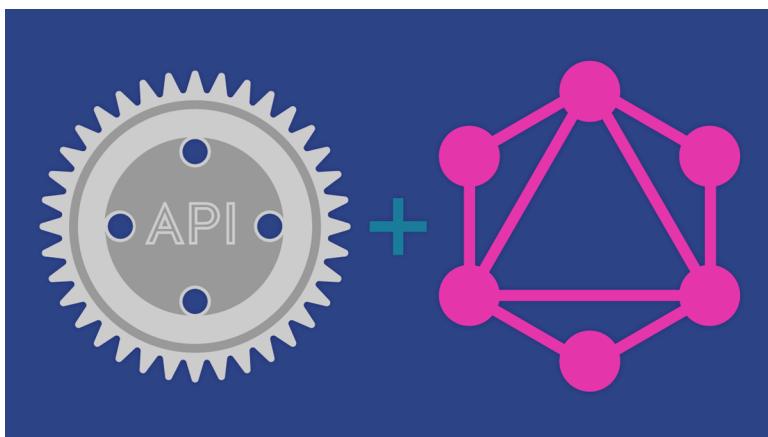
GraphQL is incredibly powerful — so powerful, in fact, that it is used by corporations such as Facebook to drive their large, complex social systems for billions of users. Despite this, the language is still relatively nascent, and its usage has yet to reach the dizzying heights that those languages it replaces and augments occupy.

In this piece, we'll discover what GraphQL is, and what makes it so powerful. We'll give you five compelling reasons to adopt it as part of your system and ecosystem, and highlight some public use cases that demonstrate the success of such adoption and integration.

What is GraphQL

GraphQL is an application layer query language. What this means is that GraphQL is designed to interpret a string from a server or client and return that data in an understandable, stable, and predictable format. As the official website for GraphQL puts it, "Describe your data, ask for what you want, get predictable results."

GraphQL does this via simple, plain to understand requests and statements. A simple example from the official website highlights this simplicity perfectly. This is a valid descriptor within GraphQL:

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

Which, when paired with an effective and simple request:

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

Returns a clean, easy, and simple result:

```
{
  "project": {
    "tagline": "A query language for APIs"
  }
}
```

The simplicity and power behind the language comes from its chief architects. GraphQL is principally an effort from Facebook, with a variety of additional contibutors dedicating their efforts to boost efficency.

GraphQL came from the result of Facebook's transition away from HTML5 applications on mobile to supporting more robust, native applications. As the need for a stronger backend and easier universal interface presented itself, GraphQL quickly emerged as the language of choice.

A quick word on GraphQL's implementations — there is a client often used as an entry point to GraphQL known as **Relay** which we will specifically cover in a following piece.

With all this said, what makes GraphQL so powerful?

1 - More Elegant Data Retrieval

GraphQL is all about **simplicity** — and with that simplicity comes a more elegant methodology and experience concerning data retrieval. Because data is collected under a common endpoint or call that is variable concerning the type of data and request as stated in the initial call, several huge benefits are intrinsic to the call system.

First and foremost, using GraphQL eliminates ad hoc endpoints and roundtrip object retrievals. Imagine you were a delivery driver looking at a map of where to deliver a package. On that map, you have fifteen possible entry points to streets near the street you need to get to — each with their own speed limits, limitations of cargo weight, and allowed vehicle type.

The complexity incurred in this situation is absolutely incredible, so much so that it would delay delivery. Now imagine the same situation, but with only a single street and will well-defined, commonly known rules — that's what GraphQL can provide.

2 - More Backend Stability

With simplicity comes **stability** — this is a basic fact of life. The more simple a process is, the less likely there are to be faults in the planning, construction, execution, and continued operation over time. GraphQL makes queries more simple and elegant — and as a result, improves the stability of the entire process.

How GraphQL does this is quite complex, but there's one methodology in particular that is worth mentioning. Because data is delivered in a structured, defined way independent on the client request (as the rules are dictated by GraphQL itself, not the application per se), data can be manipulated, changed, and altered in the backend code base without directly requiring changes in how the client functions.

In the traditional server-client relationship, this is simply not possible — the data is available, queryable, and usable, but the format and method are dictated largely

by the client. As long as the general request fits into a standard methodology as dictated by the language or documentation, that data is delivered successfully. Change how the database works, however, or update the application independent of legacy support systems, and you have a "broken" application.

This is not a problem with GraphQL. The entry point defined by GraphQL is almost like a translatable layer — the structure of request is dictated by the server, and then routed to the necessary resources and systems by the language path itself. Accordingly, an application backend can be overhauled without necessitating a complete re-structuring of the client application or the common call method, as the methodology is controlled directly by the server in the first place.

3 - Better Query Efficiency

GraphQL unifies data that would otherwise require multiple endpoints, or in the worst case scenario ad hoc endpoints and complex repeat retrievals, and gives the requester a **single, simple entry point**.

Because data is defined on the server with a graph-based scheme, data can be delivered as a **package** rather than through multiple calls. For instance, the following code for a content and comment query would have a single endpoint in GraphQL:

```
{
  latestPost {
    _id,
    title,
    content,
    author {
      name
    },
    comments {
      content,
      author {
        name
      }
    }
  }
}
```

In traditional API query languages, this would take at least 8 calls, and would have to point to specific endpoints to differentiate the content of the post itself and the comment content.

This doesn't just increase **efficiency** of data delivery, either — it fundamentally decreases the amount of re-sources required for each data request. Furthermore, retrieval constraints are declared with a declarative-hier-archical query to a single endpoint, reducing the call data demands less for the client.

The biggest net benefit of this entire process is, of course, a fundamental restructuring of the relationship between the client and the server. In GraphQL, the server pub-lishes clear and explicit rules specific to the application, and the client requests that language with common data queries. Simply said — developers are able to impose

data restrictions more naturally and with less impact on the consumer.

4 - GraphQL Is a Specification

Perhaps one of the greatest benefits of GraphQL is that the overhead for adoption is low specifically due to its status as a specification. While there are always new tools on the market, many developers shy away from adopting these solutions because they're "too involved" or "too invested" in their REST architecture.

As a specification, GraphQL is a wrapper that can be defined — you don't have to replace a REST system. This means that developers can reap the entirety of the benefits of GraphQL while ignoring the cost overhead of its incorporation.

A huge benefit of this GraphQL wrapping is also in how it changes the relationship between data and the systems that require said data. Data in GraphQL isn't limited by the language or the datatype, and is instead limited by the server descriptions as defined to the client (so, in effect, not a limitation at all). Data wrapped in GraphQL, regardless of the underlying language or system, can be shared between variations in standard API schemas and the wide range of languages which create them.

More specifically, this also means that GraphQL is fundamentally compatible with anything that REST-centric APIs are compatible with — therefore, the entry point for both GraphQL and its interactiveness with REST dependent systems is relatively low.

5 - GraphQL Improves Understanding and Organization

Perhaps the biggest benefit of implementing GraphQL is one of importance to the API ecosystem as a whole — 99% of the time, an API can be organized into a simple and understandable graph schema, and doing so forces you to better organize and understand your data, the flow of that data, and the inefficiencies and errors in that system.

While this has a net benefit to the developer, it has huge **positive implications for the development ecosystem as a whole**. Because GraphQL attaches types to data, types errors between applications and your server and GraphQL compliant applications to other applications start to disappear.

What we're talking about here is a type of "universal translator", providing data translation between different services while removing the bottleneck typically encountered, Furthermore, this removes pressure towards application developers to ensure compatibility and mutations.

A huge benefit to developers and a more positive interaction in the ecosystem — what's not to love?

Who Uses It

Because GraphQL is extremely powerful, it's been used by several providers who need stable readability with quick speed and indexing. Most of the use cases for GraphQL are therefore those who require **high data throughput** with ease of sorting, which is certainly represented clearly by its most high profile users.

One example of how powerful GraphQL is when it comes to handling high data throughput in a relational matter is Hudl, a sports video analytics provider. Essentially, Hudl takes video of sporting events, practices, and other situations and provides professional feedback for each player and a general overview incorporating this feedback.

Accordingly, the data processed by Hudl is strictly **relational** — unlike other solutions, the data being handled isn't simple one-to-one relational data. Each player belongs to a team, but each player has their own data which is attached to specific activities, skills, and approaches.

In a traditional query language, comparing and contrasting these hugely discrepant data points between each player and then each team would require a massive amount of data combination, or at the very least, a huge range of calls being called multiple times to each server system each second.

With GraphQL, each item is described clearly, and requested specifically by an application. GraphQL allows for a small range of endpoints that provide this data in a consumable, easily understandable method. This also means that the data generated by Hudl isn't strictly tied to their application, and their application alone — since it's described directly, the data can be imported into team planning applications or health tracking applications for tracking of physical health, performance in certain environments, and other such relational queries.

This interactiveness can be seen in another application of GraphQL, AlphaSights. AlphaSights can be seen as a "middle man" of sorts, but one of incredible power — connecting clients with experts and the services that they can provide. While this seems simple on its face value, the underlying data is very valuable, not just to AlphaSights,

but to those they provide this data to.

The engineers at AlphaSights stated themselves the huge value of integrating GraphQL into their system to reduce complexity and improve exchange of data:

> "GraphQL gave us one way in, and one way out. All data was resolved through one object, `Graph::QueryType`. The presence of only one object provided a much easier learning curve for new developers looking at the code base, and the higher level concepts each have their own resolver making it easy to narrow down what each term meant right down to the database level. The mapping of types from a query into ruby objects is very intuitive … By making the server to consumer contract flexible, we get the benefit of much more agile teams, and less debate on how and if the consumer receives that data. We have also found that maintaining the services is much easier. There are no "v2" routes to be added, and the type system allows the consumer to easily discover updates. We have also taken advantage of "GraphApi::Schema.middleware" to monitor which attributes are being used, and help identify legacy types/fields that we can remove, which we are going to open source in the future."

Being able to not only provide data in an easy format, but to make that data easily interactable and digestible by novice developers is a huge benefit, and makes data generated truly extensible.

GraphQL isn't only for huge complex databases, either — it can easily be used to create relatively **simple databases**

with greater efficiency. A great example of this is Beek.io. It's fundamentally a social network, albeit a niche one, focusing on books and those who love them. While the database itself is relatively simple — author, name, genre, etc. — the way this content is handled is supremely simplified under GraphQL.

By simplifying calls to a single entry point, you prevent different endpoints from providing the same data in different ways, and increase the understandability of the data that is delivered. Because of this, while Beek.io is a rather simple incarnation of the social media sphere, it is extremely responsive, and delivers data in a way that book lovers can actually use to better their reading library (and perhaps their friends list).

Conclusion: Assess

While GraphQL is obviously powerful, there are some arguments against it — chiefly that it's still in its infancy. API Evangelist sees GraphQL as a way around "properly getting to know your API resources," doubting whether the majority of non-technical API consumers will find it useful. Others have expressed that an organization shouldn't integrate GraphQL simply because a behemoth like Facebook or Github has a GraphQL API.

ThoughtWork's 2017 Technology Radar, which reaches action verdicts for emerging Techniques, Tools, Platforms, and Languages, labels GraphQL as **ASSES**. Whether or not to adopt GraphQL should at the very least be assessed; For as we've seen through this chapter, it can help ensure data is efficiently generated and accessed, and can make the applications powered by the server that much more powerful and extensive.

How to Wrap a REST API in GraphQL
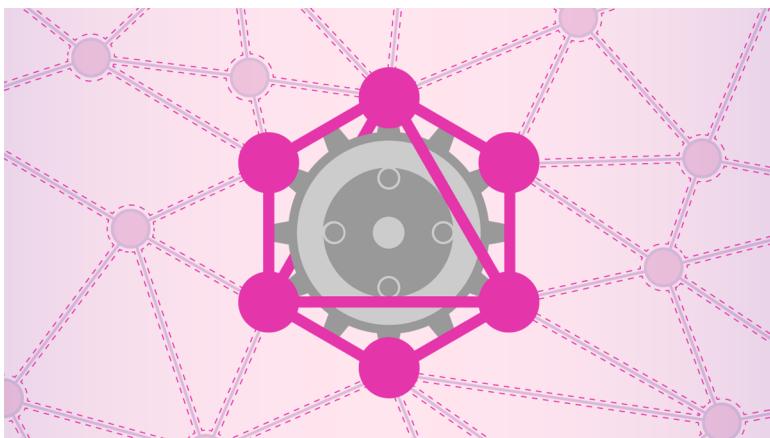
As we saw in the last chapter, GraphQL is a powerful tool. As with any emergent tool in the API space, however, there's some disagreement on exactly how to implement it, and what the best practices for its implementation and use case scenarios are.

Have no fear, dear reader — we're here to sort this all out. We're going to discuss **GraphQL**, where it came from, where it's going, and why you should consider implementing it. We'll focus on how to wrap a **RESTful API** with GraphQL, and how this functions in everyday usage.

What is GraphQL?

For those who are unfamiliar, GraphQL is an application layer **query language**. It interprets a string from a server or client, returning the data in a pre-defined schema as dictated by the requester. As the GraphQL official site states: *"Describe your data, ask for what you want, get predictable results."*

The way data is requested is quite clean and elegant. The following is a valid data descriptor:

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

When this is requested as such:

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

It returns a clean, easy, and simple result:

```
{
  "project": {
    "tagline": "A query language for APIs"
  }
}
```

A GraphQL implementation results in more elegant data retrieval, greater backend stability, more efficient queries, and improved organization with a language that has low adoption overhead. With this being said, let's get into the meat of how exactly we can implement GraphQL in a RESTful API.

Defining a Schema

The first step to wrapping a RESTful API is to define a **schema**. Schemas are essentially like phonebooks — a stated, common methodology of recognizing and organizing your data and the interactions concerning said data.

When properly wrapped, a RESTful API will funnel all influx and outflux of data through the schema itself — this is the main power of the GraphQL system, and is where it gets its **universality**.

In following with the official GraphQL documentation, the implementation we're going to show today is simplified, with some issues in terms of performance against more complex and time-consuming alternative implementations. This solution, however, requires no architectural or server augmentations, and is a perfect stepping off point that we can take to move forward.

The implementation as suggested by the GraphQL documentation is as follows:

```javascript
import {
  GraphQLList,
  GraphQLObjectType,
  GraphQLSchema,
  GraphQLString,
} from 'graphql';

const BASE_URL = 'https://myapp.com/';

function fetchResponseByURL(relativeURL) {
  return fetch(`${BASE_URL}${relativeURL}`).then(res =>\
 res.json());
}

function fetchPeople() {
  return fetchResponseByURL('/people/').then(json => js\
on.people);
}

function fetchPersonByURL(relativeURL) {
  return fetchResponseByURL(relativeURL).then(json => j\
son.person);
}

const PersonType = new GraphQLObjectType({
  /* ... */
  fields: () => ({
    /* ... */
    friends: {
      type: new GraphQLList(PersonType),
      resolve: person => person.friends.map(getPersonBy\
URL),
    },
  }),
});
```

```
const QueryType = new GraphQLObjectType({
  /* ... */
  fields: () => ({
    allPeople: {
      type: new GraphQLList(PersonType),
      resolve: fetchPeople,
    },
    person: {
      type: PersonType,
      args: {
        id: { type: GraphQLString },
      },
      resolve: (root, args) => fetchPersonByURL(`/peopl\
e/${args.id}/`),
    },
  }),
});

export default new GraphQLSchema({
  query: QueryType,
});
```

What this schema is basically doing is attaching **JavaScript** methods to the variables, and establishing the methodology by which the data is returned. The beginning and end is a necessary statement — the import of the GraphQL strictures, and the export of the GraphQL schema proper:

```
import { GraphQLSchema } from 'graphql';

export default new GraphQLSchema({
  query: QueryType,
});
```

By establishing two constants — a **data** type, and a **query** type — data is collated internally through the API, while allowing for fetching given a specific set of arguments given by the requester:

```
const PersonType = new GraphQLObjectType({
  /* ... */
  fields: () => ({
    /* ... */
    friends: {
      type: new GraphQLList(PersonType),
      resolve: person => person.friends.map(getPersonBy\
URL),
    },
  }),
});

const QueryType = new GraphQLObjectType({
  /* ... */
  fields: () => ({
    allPeople: {
      type: new GraphQLList(PersonType),
      resolve: fetchPeople,
    },
    person: {
      type: PersonType,
      args: {
        id: { type: GraphQLString },
```

```
    },
      resolve: (root, args) => fetchPersonByURL(`/peopl\
e/${args.id}/`),
    },
  }),
});
```

What this functionally does is establish the data and accepted query methodology for `email`, `id`, and `username`, and resolves the data by accessing the properties of the `person` object as attached in the code. While this technique relies on some functionality in **Relay**, a companion to GraphQL often considered inseparable, the principle remains the same — predictable, queryable data.

Of note for this approach, however, is the fact that the types in question were hand-defined. While this works for small systems, it's not a tenable solution for larger APIs. In such a case, solutions like Swagger can define type definitions automatically, which can then be "typified" for the GraphQL schema with relative ease.

Alternatives to this Method

Thankfully, there are some very enterprising developers who have taken GraphQL to its logical extent, automating the process of creating the schema itself. One such solution is the graphql-rest-wrapper. Designed to easily create wrapped REST APIs, this technique is simple to employ:

```
const wrapper = new gqlRestWrapper('http://localhost:90\
90/restapi', {
    name: 'MyRestAPI',
    generateSchema: true,
    saveSchema: true,
    graphiql: true
})

app.use('/graphql', wrapper.expressMiddleware())
```

This solution is rather simple, but elegant in how it handles the schema production. The "gqlRestWrapper" class creates a GraphQL schema from the REST response. In a way, this is similar to a game of telephone, wherein the middle man takes the data being passed through, and defines it into a usable schema for future interaction.

A few steps need to be taken. First, the npm package must be installed. Then, it needs to be imported. Finally, the code function as stated above needs to be instantiated:

```
npm i graphql-rest-wrapper
```

```
var gqlRestWrapper = require(graphql-rest-wrapper)
```

```
new gqlRestWrapped([apiEndpoint], [variableOptions])
```

Then, middleware, or the interpreter in the telephone game, needs to be attached to the route proper:

```
app.use([ROUTE], wrapper.expressMiddleware())
```

And finally, an HTTP GET/POST request can be made:

```
fetch("http://localhost:9090/graphql",
    {
        headers: {
            'Accept': 'application/json',
            'Content-Type': 'application/json'
        },
        method: "POST",
        body: "{'query': 'query { MyRestAPI { id, name \
} }'}"
    })
    .then(function (res) {
        console.log(res);
    })
```

The benefit of this style of wrapping is that it's entirely automated when properly organized. Whereas the first process is entirely by hand, the second process is entirely handled by an automatic and effective system. This lends itself to problems that the hand-coded method misses, of course — principally, the fact that more complex code can be missed or result in broken schemas.

To Wrap or Not to Wrap

Of course, this begs the question — **should we really be wrapping a RESTful API in GraphQL in the first place?** This assumes that the API in question is being left in REST for the sole purpose that development of a GraphQL compliant endpoint over a series of hundreds of use cases would be an unworthy time sink.

That may not be true, however, when one considers the lengths to which a provider must go in order to get what they want out of their API. It may simply be more useful

to code a GraphQL compliant series of endpoints rather than try to wrap an API in a new skin.

This isn't an all or nothing proposition, either. At the Nordic APIs 2016 Platform Summit, Zane Claes spoke on the movement from an internal, legacy, monolithic API, to a more consistent group of API functionalities that served data given specific devices, specific use cases, and specific requirements.

It is entirely possible to use a legacy API for a time, and slowly migrate to a GraphQL compliant API, rather than wrapping an existing API as a "stop gap". What we're talking about here is the difference between a band-aid and a full-blown hip replacement — the extent of difficulty may not be known until it's actually attempted.

Conclusion: Wrap or Recode

Thankfully, the methodology of wrapping an existing API comes down to how complicated the situation is. For most API providers, a simple wrapping as stated above would work, with the automated solution being entirely acceptable.

For others, however, especially APIs that are simply mono- lithic, the process of re-coding an API to be GraphQL compliant is a more effective choice.

Best Practices for A Healthy GraphQL Implementation
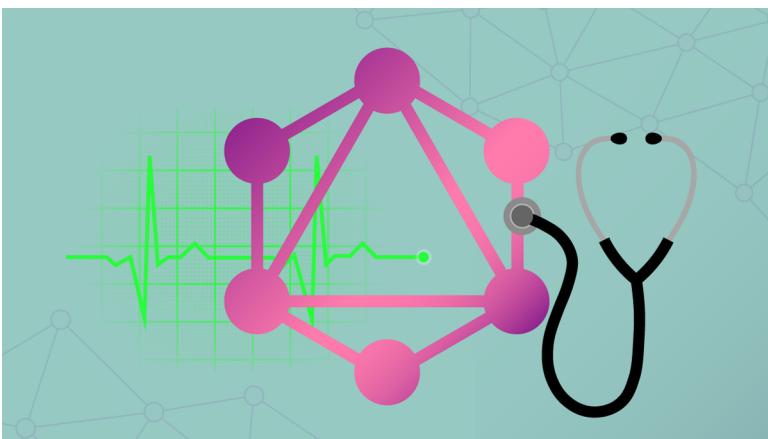
We've discussed **GraphQL** at length previously – and while the discussions on how GraphQL works are obviously very powerful, we've yet to dive into some of the **best practices** that should be adopted when developing a GraphQL-centric API.

Today, we're going to do exactly that. We're going to discuss the best practices for healthy GraphQL implementation, and describe why these practices are so advised. By properly integrating these best practices as a matter of course, your GraphQL-driven implementation and its associated API structure can be more powerful, efficient, and practical.

Dogma vs Practices

Before we dive too heavily into this, it must be said — these are suggested best practices, not absolute dogma. While the majority of the practices herein are applicable to most API installation, use case, and design, there are some situations in which your API design or application will not work well with some aspects of GraphQL, while requiring other aspects of it.

In such cases, these best practices should be considered a **guideline**, not a dogmatic instruction on how GraphQL should be implemented.

URIs and Routes

While GraphQL is fundamentally RESTful, it does drop some elements that have become core to RESTful design, such as **resources**. Wherein most REST APIs use resources as a basic conceptualization for how data is processed and returned, GraphQL eschews this and instead depends on entities as defined by an entity graph. This entity graph drives all traffic to a single URL or endpoint, which is the forward facing entrance into the system.

Accordingly, URIs and routes should all lead back to a single endpoint — this is the fundamental use case of GraphQL, and is principle in its functionality.

Endpoint Collation and HTTP

By its nature, GraphQL serves verb requests over a single endpoint in HTTP. This aspect of GraphQL is fundamental to its design and approach towards endpoint collation

— as such, using a methodology other than a collated endpoint in HTTP is locking a lot of functionality away.

In classic REST format, resources are exposed via a suite of URLs, and each URL has a specified endpoint that is tied into as part of the formatted request. While GraphQL can certainly co-exist with this kind of setup, having such a collection of resources behind URIs defies the purpose of GraphQL implementation in the first place.

Accordingly, if you need to adopt such a practice of **hybrid linking** (aka a single collated point and then a group of URLs representing individual resources), it would be advisable to look into your design, schema, and format, and address whether or not this is actually intended before proceeding with such a system in GraphQL.

API Versioning

GraphQL is a schema, and as such, there's nothing stopping your API from **versioning** however it desires. That being said, GraphQL has stated a strong avoidance of versioning from a philosophical standpoint. GraphQL's stated viewpoint is as follows:

> "Why do most APIs version? When there's limited control over the data that's returned from an API endpoint, any change can be considered a breaking change, and breaking changes require a new version. If adding new features to an API requires a new version, then a trade-off emerges between releasing often and having many incremental versions versus the understandability and maintainability of the API."

Accordingly, GraphQL takes the view that, since the so-
lution only returns explicitly requested data, there's no
such thing as a "breaking change" in properly designed
GraphQL APIs. This is a definite shift from *versioned* API
to *versionless*, and is part of the design ethos of GraphQL
itself.

That's not to say of course that versioning needs to be
an all or nothing practice. It's very easy to simply version
by documenting changes and referring to the API by a
stated version number – this is much more an internal
organizational consideration, though, and as thus falls
outside of the common definition of true versioning. As
such, it's better to call this a revision process rather than
a versioning process.

Nullability and Default Typing

APIs typically handle **nullability** in two steps: with a "null"
common type, and then a "nullable" version of that type,
specifically to separate the two types from one another
while allowing for nullification by specific declaration of
said type. This is all well and good, and in fact is a good
practice in and of itself, but because of how GraphQL is
structured and defined, this actually does not work within
the schema.

Accordingly, the best practice for nullification in GraphQL
is to remember that this null type is in fact a default setting
for every single field. This was integrated into GraphQL
as a methodology for managing failures from a variety of
internal and external causes, and as such, works well for
its intended purposes.

As a corollary to this, keep in mind that if a non-null value
is required, it can be set as a non-null type — regard-

less, it's important to remember this default nullability approach when addressing issues in the return stage of data retrieval.

Pagination

Pagination in GraphQL is almost entirely the purview of the developer. GraphQL allows for listed values as a return, and as those listed values grow in length, how they are returned is dictated by the strictures created by the API owner themselves.

GraphQL allows for a pattern known as "Connections." This element of GraphQL allows for not only discovering information about specific related items, but about the relationships themselves. Adopting Collections is a best practice, as it ties into other GraphQL suites and tools like Relay, which can automatically support client-side pagination in that common format.

JSON and GZIP Dependency

GraphQL is designed to respond to requests using JSON. Despite this, it's not expressly required in the specification, and in fact GraphQL can work with a variety of response types. That being said, it's a best practice to adopt JSON due to its text-centric organization, as GraphQL is designed expressly to work with high-ratio GZIP compression.

Interestingly, GraphQL is designed with the syntax of JSON in mind, and as such, it is a good idea to keep this syntax consistent through both design and response. Likewise, GraphQL suggests that clients respond with the following appended in the header:

```
Accept-Encoding: gzip
```

This will allow for even further **compression** of the response data, and should result in lower overhead and greater network performance. You don't necessarily have to adopt JSON, especially if you are adopting GraphQL after your API has already been designed and given a specified output format, but failure to adopt JSON can result in **decreased efficiency** and confusion amongst the user base when comparing output with the code that generates said output.

Batching to Address Chattiness

GraphQL is, by default, extremely **chatty**. This is simply due to how GraphQL is formed, how the data is collected and returned, and how the server address requests. This chattiness can be a hindrance, especially when doing large scale data requests from a database.

Under GraphQL, the solution is to simply **batch** these requests over a given period of time, collating them, and then submitting the multiple requests as a single, collated request to the system in question.

This in turn eliminates much of the chattiness, as you are no longer issuing multiple packed, collated requests to multiple elements of a GraphQL enabled server — instead, you are bundling bundles of requests, resulting in a passive GraphQL into GraphQL solution that creates a relatively "quiet" system for the amount of data that is being requested.

This does not have to be enabled, of course, and for smaller projects, it might make sense not to do so, so that issues in requesting can be identified and singled out.

That being said, when handling any substantial amount of data, adopting batching is probably a good idea.

GraphQL is designed in a way that allows you to write clean code on the server, where every field on every type has a focused single-purpose function for resolving that value. However without additional consideration, a naive GraphQL service could be very "chatty" or repeatedly load data from your databases.

> Of note: GraphQL suggests that some of this batching can be done using GraphQL tools such as DataLoader. While this is absolutely an acceptable method, adopting batching internally as part of your code will be more powerful than any third party tool can ever be, and if needed, should very much be implemented as part of the base architecture.

Disabling GraphiQL in Production Environments

GraphiQL is a big selling point for many GraphQL adoptees, but even though it's extremely powerful, it should be disabled in production. This recommendation comes from the official GraphQL documentation. GraphiQL should be disabled due to various vulnerabilities that could inadvertently expose internal API functionality by exploring the primary forward-facing endpoint. Enabling GraphiQL essentially negates many of the reasons you are integrating GraphQL in the first place.

GraphQL Authorization Practices

GraphQL specifically notes in its specification documentation that **all authorization must be done in the business logic layer**. This is specified due to how GraphQL functions. Using authorization logic that checks for elements of the entry point validation process, such as "has author ID" or "carries correct token" would mean that, for every possible entry point, this logic would have to be duplicated, which leads to incredibly complex implementations for even small amounts of authorization controls.

Accordingly, GraphQL dictates that this should be done at the business logic layer so that the authorization controls only need to be specified once for the entirety of that specific class. GraphQL notes that this is fundamentally considered having a "single source of truth for authorization."

GraphQL Pipeline Model

As part of this approach towards Authorization, GraphQL advises that all authentication middleware should come before the GraphQL implementation to avoid the same issues with *authentication*. Any filters, plugins, extensions, etc. should all be placed before GraphQL to allow session and user information in its final form for granular and singular control.

Conclusion

As stated at the beginning of this piece, we've aimed to share *advice* rather than *dogma* when we discuss these

best practices. The best practice of all is to **adopt so-
lutions that fit with your specific use case**. Consider
these GraphQL best practices as guidelines rather than
set-in-stone rules, but for the most part, adopting these
best practices and implementing GraphQL as designed
will lead to a more powerful, leaner, and more efficient
system. Failure to adhere can result in many of the ben-
efits to GraphQL disappearing as quickly as your number
of endpoints.

Security Concerns to Consider Before Implementing GraphQL
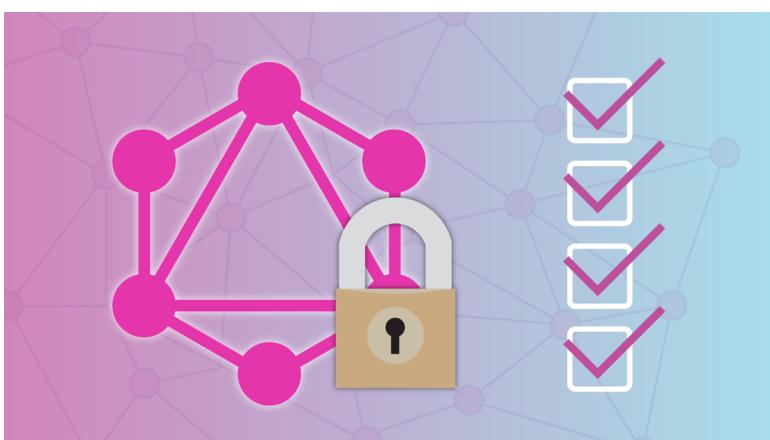
GraphQL is a very powerful query language that does a great many things right. When implemented properly, GraphQL offers an extremely elegant methodology for data retrieval, more backend stability, and increased query efficiency.

The key here though is that simple phrase — "when implemented properly". GraphQL has had somewhat of a gold rush adoption, with smaller developers responding to the media hype and big name early adopters with their own implementations. The problem is, many people aren't considering what adopting GraphQL actually means for their system, and what **security implications** come with this adoption.

GraphQL is a paradigm shift in many ways — and with that, security concerns have changed. While some security concerns have gone away, replaced by architectural differences and nuances, other concerns have been amplified.

In this piece, we're going to talk about those issues, highlighting some general concerns in regards to security in an API system supporting GraphQL. While GraphQL itself is not the primary driver of these concerns, these issues should be addressed within the greater frame of a GraphQL system, and all of the implications that suggests.

GraphQL - A Summary

Quickly, let's summarize what GraphQL is, and how it does what it does. Simply put, GraphQL is an application layer query language designed to interpret a string from a server or client, and return that data to the requesting client in the form that they request.

GraphQL was developed by Facebook as a means to transition away from HTML5 applications on mobile towards robust, native applications. This was facilitated by allowing easier backend queries through the unification of multiple interior endpoints to a single forward facing endpoint.

With that in mind, what are some concerns in regards to security and best practices in terms of GraphQL?

Implied Documentation vs. Actual Documentation

Another serious concern when implementing GraphQL is how documentation is handled between versions. GraphQL does not have versioning support in the same way other systems do. That's not to say that you can't version in GraphQL, but simply that by design, GraphQL is designed to API evolution without version control being required.

While this is certainly a fair approach, there are a good number of developers who depend on versioning to communicate changes in field values, endpoints, and declarations. While this is not good practice, that doesn't stop it from being relatively common — and unfortunately, when GraphQL is brought into the mix, the issue only becomes worse.

Let's say an API changes a fundamental endpoint or internal construct. In a traditional API, this change would be documented as a version change, with in-built documentation as such. In GraphQL implementations, this isn't the case, and so many developers might learn about this deprecated endpoint or construct by attempting to use it, and being turned away.

So what's the security issue here? The problem is that, without proper documentation, you can very quickly run into a situation where an endpoint is no longer valid, but data is still being sent and requested from that endpoint. This can result in collisions and unintended functionality. More to the point, if your application is still attempting to poll a non-existent GraphQL endpoint, an emulated endpoint from a man-in-the-middle attack could theoretically step almost seamlessly into the data stream.

There are many ways to mitigate this, and adopting a continuous versioning strategy is absolutely paramount as a solution. That being said, this is much more a problem with developer practices than GraphQL itself. That comment aside, the issue dramatically magnified by GraphQL, and should be considered prior to any GraphQL implementation.

Unified Failures

Perhaps the biggest issue with a GraphQL implementation is inherent in the approach itself. GraphQL has the extremely powerful ability to unify multiple endpoints into a single queryable point — this is the entire crux of what makes GraphQL powerful. The problem here, though, is in the fact that a single endpoint, even if it connects to multiple internal endpoints, functions as a single point to the consumer.

We often tend to think of APIs from an "interior to edge" way — we think from the code base out to the consumer experience. Typically, this is fine, as the consumer is a single point accessing multiple endpoints, which then call multiple functions, which might relay to even more databases or resources. When we're talking about GraphQL, however, we're actually creating an "interior to edge to interior" style system, in which the codebase narrows to a single point or collection of single points, which then expands to the user.

In a poorly defined GraphQL request from the consumer (or, for that matter, a poorly formed GraphQL endpoint defined by the developer), a single failure means a total failure to access internal resources. If improperly abstracted, documented, and specified, you're essentially

putting all your eggs in a basket.

Functionally speaking, this entire issue could be summed up thusly: call failures equate to insecurity for the dev consumer. Minimizing these failures is an entire industry in and of itself, and poorly adopting GraphQL negates much of the efforts developers have made to this end.

Data and Server Transaction Volumes

GraphQL has another fundamental problem related to its foundational aspects — queries are often larger and more complex than in traditional REST APIs. In GraphQL, a single request might combine multiple requests to multiple endpoints, resulting in an unpredictable amount of data for each request over time.

Consumers are in control of their requests — and in some ways, this is dangerous. Not every provider is going to have huge scalable infrastructure or cloud servers on retainer, and thus not every provider is going to be comfortable with the idea of allowing for variable content requests based on the whim of a user.

There's also the concern of limiting the actual request in a reasonable way. Consumers aren't always the most judicious with their requests — sometimes, a consumer may request more data than they really need, and over a long time and large amount of requests, this adds up to significant overhead that did not exist in a non-GraphQL solution.

Information Hiding and Chattiness

Let's get this out of the way first — neither GraphQL or REST enforces any significant amount of data hiding. The problem with GraphQL specifically comes when people rush to adopt it, and instead of logically thinking out the mapping of public endpoints to private models, they simply map a 1:1 relationship.

While this is functional, it's incredibly dangerous. The entire point of GraphQL is to allow the client to request the data the way that they want it to be requested — but this also means that GraphQL enables data in a way that may not be intended, and when 1:1 relationships are established, you lose some very important control of internal assets.

This is, just as with any REST API, entirely manageable with proper GraphQL schema design. Third party clients and APIs should be able to interact with your GraphQL endpoint without knowing the internal workings of the API.

GraphQL can be done in a way to have this be a limitation of the endpoint, but this comes with the threat of a rushed adoption, possibly resulting in exposing much more than was intended. Depending on how GraphQL is set up and how queries are handled, the actual endpoints defined can insist on "chattiness", rather than allow it.

This isn't just a matter of schema revelation, either. When an endpoint unifies multiple internal endpoints and functionalities, this endpoint can be improperly implemented to retrieve far more data than was ever intended. This has some serious implications.

First, there is the server implication. If the data is being

retrieved first and then stripped of irrelevant data before being transferred to the client, we're being extremely wasteful. Imagine if every single time you want to drive, you had to burn an entire tank of petrol, regardless of distance travelled.

In some cases, this would make sense — a two hundred mile trip would easily eat through your tank, and possibly more. For a short trip, this would be absolutely wasteful, and over time, would lead to excessive wear and tear.

The same is true on the server side — limiting this chattiness is incredibly important.

Then there is the aforementioned security internal exposure issue. This has already been somewhat discussed, but it bears repeating — allowing for unlimited access to a variety of endpoints in a manipulatable way is the perfect storm for penetration testing, and makes identifying the structure and scheme of the backend trivial in many cases.

Authorization and GraphQL

Another point of potential issue is improper implementation of authorization in GraphQL-dependent APIs. Again, this is much less an issue with GraphQL itself, and more an issue with adoption.

In GraphQL, authentication can be handled handled using "query context". The basic idea here is that the request can be injected with arbitrary code that is then passed through to the request, which allows very fine, granular control of authorization by the developer proper. This is a very secure system – it's arguably much stronger

than other solutions when it comes to SQL injection and Langsec issues.

The problem comes with developers not actually understanding that this is a possibility. Developers might look at their current authorization logic, decide they don't want to change anything fundamentally, and instead insert this logic into the GraphQL layer itself rather than the business logic layer.

This is fundamentally flawed, and is advised against in the GraphQL specification itself. Placing this logic in the GraphQL layer opens the security system up to code injection, sniffing, and other attacks that could easily expose the internal authorization structure, thereby rendering it null.

Measured Optimism

This isn't to say that GraphQL is a poor choice, or that you should be wary of implementing it. Quite the opposite, in fact — proper implementation of GraphQL is possibly one of the best things that can be done for a large API with a variety of data on the backend that must be delivered in a sensible, consumer-controlled way.

What this is to say, though, is that our optimism should be tempered. As with any new technology or implementation, we need to prove that the theoretical is provable in the actual implementation. With so many potential issues sprouting from improper implementation, it's incredibly important for developers to look at GraphQL in the frame of "how do I make sure I do this right" rather than "move fast and break things".

We're not alone in this advice for measured optimism, either.

ThoughtWorks, a software company which delivers verdicts to the industry on new technologies, advised that developers should "assess" whether or not GraphQL is the correct solution given their use case. API Evanelist Kin Lane was likewise cautious, stating that during a conversation in the API Evangelist Slack channe:

> "the consensus seemed to be that GraphQL is a way to avoid the hard work involved with properly getting to know your API resources, and it is just opening up a technical window to the often messy backend of our database-driven worlds."

The simple fact is that, as with any solution, we need to be cautious and ensure the following:

We are using GraphQL for an appropriate use, and not adopting it in the "flavor of the week" mentality; Our endpoints are well documented, with secondary paths in the case of failure; We are properly hiding the internal schema and structure of our server; Finally, we are limiting the amount of interactions allowed to a "reasonable" measure.

If and when these situations are validated, GraphQL makes for an amazing implementation.

7 Unique Benefits of Using GraphQL in Microservices
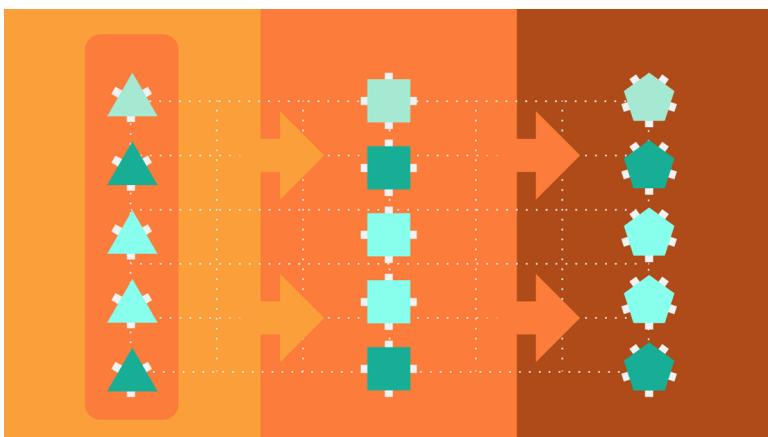
We've talked about GraphQL at length previously, and for very good reason – GraphQL is, in many ways, one of the more powerful tools an API provider has in terms of providing singular endpoints to the consumer and controlling data flow. While this value has been proven time over time, however, it seems that some of the more salient and special benefits of adopting GraphQL are often lost in the conversation.

Today, we're going to talk about these unique benefits, and what they actually mean to a production API. Many are familiar with **microservices**, so in this piece we'll discover positive impacts GraphQL brings a microservices arrangement, such as data owner separation, granular data control, parallel execution, service caching, and more.

Clearly Separated Data Owners

One of the main benefits of having everything behind a single endpoint is that data can be routed more effectively than if each request had its own service. While this is the often touted value of GraphQL, a reduction in complexity and service creep, the resultant data structure also allows data ownership to be extremely well defined, and clearly delineated.

This is in large part because the request itself is pointed towards a single endpoint, and that request must package the expected data format and the resource owner designation, whether it be for a specific function or a specific data type, is thus intrinsically tied to that request.

This is different from a typical REST API, in which the request to each microservice might be requesting data that the microservice endpoint doesn't support or cannot deliver, which would then be passed on to another endpoint, and so on. This ends up creating a web of requests and passed communication, and to the average viewer, who actually owns the requested resource is thus hard to decipher.

Data Load Control Granularity

Another benefit of adopting GraphQL is the fact that you can fundamentally assert greater control over the data loading process. Because the process for data loaders goes into its own endpoint, you can either honor the request partially, fully, or with caveats, and thereby control in an extremely granular way how data is transferred.

In many ways, this granularity is what REST has tried to

achieve with some degree of success when implementing specific schemas and request forms. That being said, GraphQL integrates this as a specific structural element of how it works, and as such, does so much more effectively than most other solutions.

Parallel Execution

Because a single GraphQL request can be composed of many requested resources, and because GraphQL can choose dynamically when, how, and to what extent a response is issued to such a request bundle, GraphQL can leverage a sort of parallel execution for resource requests. What this functionally results in is the ability for GraphQL requests to partially fail, but for the response to deliver more useful data to more requests in a single issuance.

This ultimately has the benefit of allowing a single request, even when partially failed, to serve the place of what would traditionally be multiple requests to multiple services over multiple servers. This also allows a single request to use a relatively more restrained amount of processor time and power to deliver the same amount of information that would otherwise be required of those multiple requests, thereby delivering greater power with less requirements.

> "Instead of having six sequential calls, what [a parallel dataloader] will do is give you all six IDs in one go, so you can make one request instead of six. This will make your downstream databases and APIs way happier – no one will

be on fire, no one getting paid at 2 am in the morning." -Tomer Elmalem

Request Budgeting

In GraphQL, requests can be given a "maximum execution time", and this value can then be used to budget requests. Each request is given a value, and from that, the server budget is calculated and calls are prioritized. As an example, let's assume our server has a total budget of 2 seconds, and look at a sample batch of requests.

We receive four requests – one is a single second, two of them half a second, and the final request a full two seconds. When budgeting our requests, the GraphQL server can accept the first three requests, and either delay or refuse the last request, as it would exceed the allotted time that the server has open for requests of its nature.

What this in effect does is allow the server to prioritize requests and grant them where appropriate, which ends up reducing timed out requests in the long run. Additionally, this system can return information to the requester – such as would be the case with a 6 second request, for instance – that can then inform the user to break their requests into smaller pieces or wait for a low-budget time to make the request.

> "[As an example] we set the maximum budget to one second. Some sleep function takes about half a second, and we've got about half a second in budget remaining. [...] Our budget is one second, but we have some function that takes a second and a half, so we're actually

negative 500 milliseconds in terms of budget – so everything downstream fails. If this happens on the first service call, then we can skip executing the next 6 calls [which avoids] a lot of wasted processing power when requests are timing out."

Powerful Query Planning

Combining two of these major benefits – parallel requests and budgeting – allows us to more effectively plan out our query schedules. By being able to send some queries to an endpoint, and have others execute in parallel at a later time per their weight and processor demand, you can effectively plan out those queries over a relatively broad set of criteria and per the time allotted.

While this seems simple, the ability to plan out queries over time and address per priority is one of the elements of GraphQL that is so incredibly powerful.

Service Caching

GraphQL utilizes Object Identifiers for one of the biggest savers in terms of server processing demands – caching. By being able to cache resource for services which request them, GraphQL can essentially build a cache of often-requested data and save processor time and effort.

According to Elmalen, this is rather easy to implement, in fact – GraphQL documentation suggests utilizing a system of globally unique IDs in order to note the resources being cached, thereby providing them with minimal processing demand.

> "Since you're handling a lot of network requests, you don't want to have to keep making those requests over and over and over again if you're seeing the same data frequently."

Easy Failure Handling and Retries

GraphQL is unique, in that, in a normal connection and request cycle, there is no real "fail" or "succeed" – everything is either succeed or partially fail. GraphQL requests can partially succeed, returning only elements of the data requested or specific datasets as allowed, and this control can be very granular.

As a result of this, a resolver to a request can be more than "all or nothing", partially resolving as a single resolver delivers content, and the rest of the request is dumped. This means that GraphQL is in a position to handle failure rather gracefully, notifying the requester of the failure but returning useful, actionable data alongside the noted failure and what caused said failure.

As part of this, GraphQL requests can be parsed to find the null fields returning in a failed request, and note what the error actually was. If the error was a simple misconfiguration or poorly formed call, it can be made in a better format upon instruction from the GraphQL layer itself – if the data is prohibited or absent, the requester can also ascertain this. This means that automatic retries are possible, as are granular levels of call termination and auto-failure handling.

> "In GraphQL we have a lot of flexibility with how the resolvers work. Query execution doesn't

have to be all or nothing like it might be in a REST API, where if one endpoint call throws an exception everything might blow up. Resolvers execute independently of everything else, so one resolver can fail but the entire query can still resolve."

Case Study of Microservices In Action: How GraphQL Benefits Yelp

To see what GraphQL is so useful in a microservices approach, we can look at Tomer's argument in favor of integration at Yelp. Yelp is foundationally a business that ties to additional businesses, and as such, utilizes a public API rather heavily. Their public API is supposed to give clients data connections by which the resources can be collected, reviewed, and collated.

In their original Yelp API, this external demand for data causes a significant problem. The API was too large, had too many endpoints, and as more data was requested from outside partners, the API was expanded either with more bloated endpoints or with additional endpoints to feed out the data. This resulted in an extremely large solution that was hard to maintain, hard to iterate upon, and generally less agile and was desired.

As a solution, Yelp began to implement a new design paradigm in GraphQL. Their chief reasoning was the fact that, for their outside businesses, those partners wanted to make a single request for the data they needed, and nothing more. GraphQL fit this requirement perfectly, as a single endpoint could now serve a multitude of resources in a defined and predictable way. More important, only

a single request would have to be made, as that re-
quest could be formed in the way that the data requester
needed the data and it could be served based upon the
agreed data served by the provider.

This process and choice really encapsulates the purpose
of GraphQL in a microservices architecture. While it is
true that Yelp could have simply added more endpoints,
created more microservices, expanded the resource han-
dling, and progressed the bloat process of their API. All
this would have done, however, is increased complexity,
bloat, and the cost to maintain the API.

By adopting GraphQL, however, their microservice-ori-
ented design functions more agile, more responsively,
and, perhaps most importantly, more adherent to the
design requirement at hand – a single request delivering
the data as requested.

Final Thoughts

It should be noted that GraphQL is extremely powerful,
and that the benefits at hand are especially powerful for
most microservice structures – though not all. In some
cases, especially in cases in which the data does not
change, new endpoints are not added, and additional
functions are not required, GraphQL may add additional
unwarranted complexity.

GraphQL has often been sold as a perfect solution for
every problem, but the reality is that it meets one require-
ment better than most other, and if that's not part of your
requirements, it may not be the best solution for your
implementation.

For situations like that faced by Yelp, however, GraphQL

fits perfectly and solves the major issues at hand. For this reason alone, should developers find themselves in a microservice architecture and requiring a greater flexibility in data delivery and structuring, GraphQL should absolutely be a top consideration.

Comparing GraphQL With Other Methods to Tether API Calls
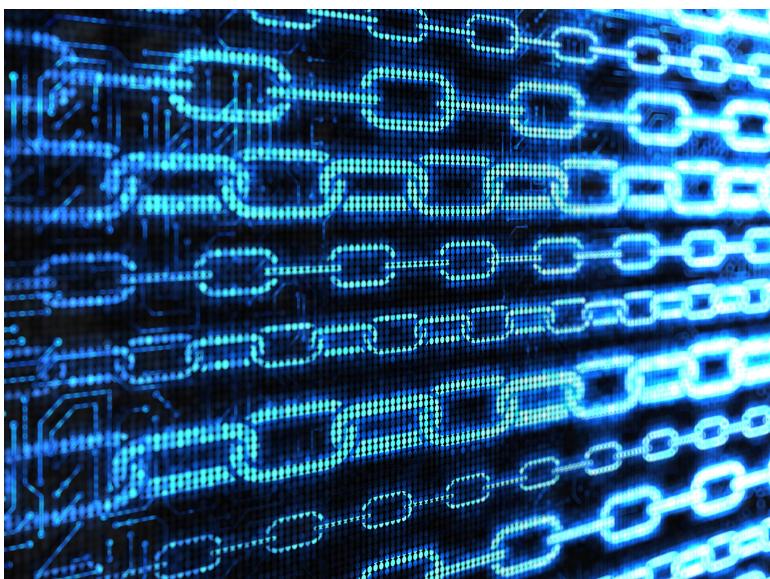
From the very beginning, developers have sought to make their systems run more efficiently, delivering greater amounts of data and functionality in a slimmer, more useful methodology. Increased efficiency usually equates to money saved, so there is a very real drive behind the constant need for condensing and code tweaking.

Though there are a variety of approaches taken towards this end, one method remains a popular suggestion — **tethering API calls**. A simple process in theory becomes

one more complex in application — with interesting results in certain use cases.

In this chapter, we'll discuss some general methods used to combine API calls, and how to implement them. We'll take a look at the various strengths and weaknesses of this approach, as well as highlight some use cases that would benefit from its implementation. We'll also compare it to a chief alternative competitor, GraphQL, and examine whether or not one comes out on top.

What Do We Mean By Tethering API Calls?

Tethering is the act of initiating multiple data requests to various parameters, APIs, or microservices, in a **single call**. The idea is that by combining these requests, your platform gains efficiency and speed. However, there are some caveats that we'll discuss shortly.

One way to consider tethering is to think about how mail is handled via the hub-and-spoke model. When you write a letter, this letter is dropped off in a post box. The postman then collects these letters and parcels, binding them together for processing. At the processing shop, these bundles from everywhere in your city are bundled together into even larger crates depending on their destination and purpose.

Tethering works much the same. The big benefit of this method arises in that postal metaphor — the **joining of unrelated resources within a single action**.

Let's say you have a collection of APIs in a microservice architecture, each manipulating data across a suite of

applications. While there is no need for a music discovery application to access the contacts on an iOS device regularly, a new update has enabled a significant social networking service on the discovery application.

Luckily, your users also largely use another one of your products, a video sharing app akin to Vine, with users collated from their contact list. While pulling straight from the contact list and comparing to accounts might get you some users, you are likely to have a much greater chance of adding new users to your new social platform that already actively use your suite.

With linking API calls, a request can be done to the video sharing application API to issue a friend request on behalf of the user, while avoiding sending new requests to users who are not represented on any of your suite's applications.

In this case, what would typically take several calls — a "is this user on the video app?" call, a "are they on the user's contact list?" call, and finally a "friend request sent!" call — can be condensed to a singular call.

Benefits of Tethering API Calls

There's quite a number of benefits to this approach in particular. First and foremost is **speed** — when an API only has to issue a single call that is handled by the relational processing between microservice APIs, calls are returned quicker and with greater accuracy. Light can only travel so fast, and thus processing is fundamentally limited — reducing the number of calls that must be made reduces latency, making chaining a powerful way to optimize APIs for mobile apps.

In the same vein, chaining is much more **efficient** than traditional calls. By using the connection between APIs rather than creating a singular "filter" or "intersection" style API that attempts to combine all calls after they're made, you can perform the same function while not creating bloat inside your ecosystem. Chaining is the world's greatest game of telephone — you ask your neighbor for sugar, milk, and eggs, and they in turn ask their neighbor for what they don't have, and so on, until you get what you need — without this, you'd have to walk to each neighbor individually.

There is of course an implicit increase in **security** for this function as well. While it seems counter-intuitive, what with more resources being accessed with only a single call, the end result is a reduction in attack vector for the simple reason of having less calls.If you had 100 armored vans transporting money through a high-crime area, or a single armored van transporting money, you would have greater security through reduction in trips over time, even if the actual security of the single result is the same.

This mode of combined communication also has the benefit of making the API more friendly to **automation**. By moving the logic of the call to a higher level, and opening the call itself to shared resources, the ultimate result is an easier to deploy, more automatic, less code-heavy, and faster solution.

What this all means is that, as long as it's properly implemented, both the provider and consumer can have a much better, **streamlined experience**. While consumers benefit from combined calls and more simple call structures for large amounts of data, the provider has a much reduced amount of documentation and data bandwidth implications.

Drawbacks of Tethering API Calls

That's not to say, however, that chaining is perfect — there are some caveats to consider before implementing this methodology.

The first and most dramatic possible drawback is the fact that chaining lends itself to **circular logic**, if you're not careful. Because multiple resources are being called, and many of these resources can be augmented by calls, it is possible to call a chain that results in an infinitely changing result.

These unexpected results can be negated, of course, and much of the drawback of circular references can be converted into a net benefit with proper redirects and interceptors, but the threat is very real, and easy to fall into if not mitigated from the onset.

Another is **lack of support**. Unfortunately, chaining is simply not supported by common and publically used schemas and API patterns. Because of this facet, where communication logic and business logic are separate and architecturally limited, an API must be redesigned at worse, or augmented with abstraction layers at best, which can in turn negate many of the benefits inherent to the system.

Adding **complexity** to the call can mean a greater barrier to users and additional developers. With this also comes the risk of easier **denial of service attacks** and other such attacks that can use the increased ability to make combined calls to break the system.

This can be negated widely by simply enabling rate limiting, but all in all, it may be a negative that makes integration not worth it.

Use Cases

Chaining is a preferred methodology in some limited circumstances. Since the circumstances themselves are often specific to the application, we must speak broadly as to the qualities that make API tethering a good choice.

- **Unrelated Databases**: When multiple databases are present with typically unrelated data, and data must be pulled from both to form a relation, combining API calls is a good idea. While it would be best in some cases to create a relational link between these databases, the problem arises in the frequency — if the data is not expected to constantly be referenced in a concrete way, API Chaining can get around this problem with relatively low overhead.
- **Unexpected Data Requirements**: When the specific requirements for a function or the given evolution of a product are unknown, tethering can cover this facet. Because not every facet of evolution in an API can be predicted, developers often suggest microservices as a solution. This is fine, but it limits the choice of development tracks into the realm of what already exists and forces concrete development relations. Tethering calls can eschew this.
- **High Volume Calls**: When high volume calls are the name of the game, chaining can reduce the total impact of the simple volumetric issues by combining calls into single returns. This also has the added benefit of more secure communications, as the number of calls is reduced. While this could be touted as a feature, unfortunately, reception of tethering has not been as widespread as before, meaning that

security in the API space doesn't always flow over into chained requests. This can be negated with training, however.

Alternatives — GraphQL

This isn't the only solution to the problem of such complex calls, of course. GraphQL can be used to replicate much of the effect gained alternative strategies, albeit via a different methodology. Because data can be defined via the GraphQL query, multiple datapoints can be collated into a single database or collection and then returned in the requested format for easier parsing.

For example, this call queries multiple endpoints through a singular call:

```
{
  latestPost {
    _id,
    title,
    content,
    author {
      name
    },
    comments {
      content,
      author {
        name
      }
    }
  }
}
```

This is done via a completely different methodology, though, so some of the benefits of tethering — specifically the reduction in call volume and processing — are fundamentally lost. The architecture itself is simply being masked with GraphQL — none of the processing is changed on a base level, and the traditionally poor processing architecture marches on unbeknownst to the user.

While multiple endpoints are brought into a single endpoint, the result is still functionally the same — multiple calls are still called. The only difference is in the perception of the returned data and latency in returning that date.

There is a big benefit in integrating GraphQL, however, in the fact that the data can be pre-formed and pre-determined by the requesting entity. Whereas tethering simply returns the data for interpretation, GraphQL specifically issues a schema for data display, thus controlling the output more effectively.

Conclusions

API tethering is very powerful, but quite problematic when not properly implemented. Because of this, it's often said that using tethering is outside of best practices, both due to lack of training and the avoidance to call through a proxy loopback. However, it is within "best practices" for an untrained pilot to not fly a 747 — that doesn't mean the 747 isn't useful to those who know how to utilize it. It's also less efficient to transport the Space Shuttle inside a 747 — but in some use cases, it's had to be done. Neither argument is really one against implementation — they're best categorized as perhaps "too strong" precautions against poor implementation.

Though many of the criticisms against API tethering are valid, they can be negated by proper implementation. The choice to combine API calls will depend entirely on your circumstance and the needs to the application itself. In the right situation, with the right user base, API tethering is incredibly powerful.

The Power of Relay: The Entry Point to GraphQL
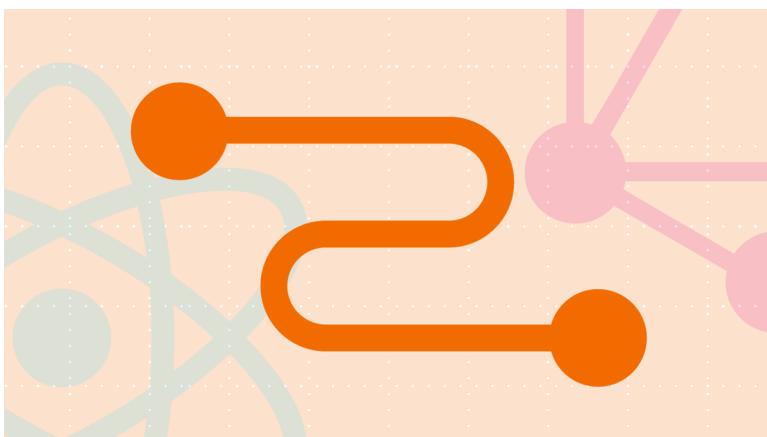
In many ways, GraphQL is a futuristic approach to dealing with all the headaches surrounding high-data transfer, large-volume relational content. As more is written about the technology and as its implementation is discussed, it goes without saying that related components are becoming increasingly more interesting as well.

One of these components, **Relay**, often falls to the wayside in the conversation — and that's a shame, given that Relay is incredibly powerful, useful, and interesting, given the right use cases.

Accordingly, this piece will focus on Relay as an extension of GraphQL per Facebook's stated development guidelines and documentation. We'll discuss how Relay does what it does, what specifically makes it special, and why

pairing Relay with some — but not all — GraphQL imple-
mentations is a good idea.

What's the Difference Between GraphQL and Relay?

VS
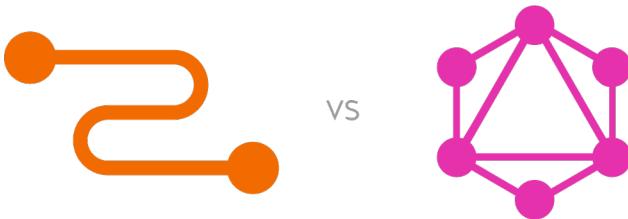
While GraphQL and Relay are often pushed in the same
sentence (and are treated like a package by Facebook
and many other advocates), they are actually two very
different parts of a greater mechanism.

GraphQL is, fundamentally, a way to model and expose
data in the native application. That is to say that GraphQL
is the methodology by which all of its extended function-
ality is prepared for fetching and interaction.

Relay, on the other hand, is the **client-side data-fetch-
ing solution** that ties into this stated model to render
data efficiently for the end user. It ties into the GraphQL
schema, it uses the GraphQL schema, and with further
server-side additions, it augments GraphQL schema, but
to say they're one in the same is like saying "gasoline" and
"tires" are one in the same because they're both used to
power a car.

To further drive home the point, GraphQL is able to be

used entirely independently of Relay, while Relay depends on GraphQL (or, at least, GraphQL-like schemas) to function. GraphQL can be used with any **fetching technology** that is designed to handle the query in question (which is easily done with most modern solutions).

The GraphQL specification describes how Relay makes three core assumptions on what a GraphQL server provides:

1. A mechanism for refetching an object.
2. A description of how to page through connections.
3. Structure around mutations to make them predictable.

What is Relay?

So what exactly is Relay? At its most base level, Relay is a **JavaScript** framework crafted to build React.js services. It was designed as a component to Facebook's GraphQL, expressly crafted to handle high data throughput and output the requested data in a dynamically stated way.

A big power point behind Relay is how it handles this data fetching. Relay handles data through declarative statements in GraphQL, composing the data query into efficient batches while keeping to the stated data structuring. Because of this, Relay is very **fast**, very **efficient**, and more important, **extensible** to the application demands in a dynamic manner.

That's not the only thing that makes Relay users sing its praises, of course. **Colocation** is present in Relay, allowing for aggregate queries and limited fetching. **Mutations** are supported widely as well, and provides optimistic updates to create a more seamless user experience by

presenting data as a positive throughput even while the server is still managing the request.

Essentially, Relay does what it does well, in very specific applications, and more efficiently than other solutions.

The Good

So with all this in mind, why use Relay at all? If GraphQL does so much, and operates outside of Relay, why do we need it? Well, GraphQL isn't perfect. It lacks the ability to poll and reactively update, and it has some built-in inefficiencies that make the system less than optimum.

Relay, on the other hand, fixes many of these issues, extending its usefulness into new heights. With Relay, data requirements are expressly stated and fetched much more efficiently than just standard fetching in GraphQL. This increase in efficiency stems largely from the **data caching** built into Relay, allowing existing data to be **reused** instead of forcing a new fetch for each round trip on the server.

Part of this boost in efficiency comes from aggregation and colocation of queries into single, streamlined data requests. While this has a huge benefit in terms of logic, the main benefit is in the network traffic and pure volumetric reduction.

The improvements don't stop there. Relay offers efficient mutations, and provides for cataloguing, caching, and altering these mutations after the fact, including dynamic column/value alteration.

A huge benefit here is the support for **optimistic updates**. Optimistic updates are an interesting methodology of client mutations wherein the client simulates the

mutation as the server commits the mutation to the back-end, allowing the user to interact with the changes they made and simulate their experience without waiting for the server to commit.

As part of the support for optimistic updating, Relay provides a system for Relay mutation updates, status reporting, and rollback, which allows for more seamless management of client and server states. Relay also supports rich pagination, easing the heavy burden of large data returns and making them easier to consume, further improving user experience.

We can see the effectiveness of Relay by looking at its implementation. Messaging application Drift is designed to tie customers and providers together using real-time messaging natively on the provider's website. Because of previous experiences with multiple endpoints and large data requests, the team at Drift knew that speed would be affected — and dramatically.

When they started a new company, Drift, and saw themselves falling into the same hole they previously did, they made the decision to fix the issue early, and integrated GraphQL into their services. When faced with the following complex data set:

> "Customer attributes come from a single request that returns name, title, location, time zone and avatar. But in order to display the account owner, we'll need to query the organization's team endpoint to fetch that name and avatar. To render those colorful tags, we need to fetch the tag definitions from another endpoint. The list of all contact avatars requires a search for all customers in the same company

against our ElasticSearch backend. The chat count and last contact requires multiple calls to our Conversation API and one last call to fetch that user's online status or last active timestamp."

They coded the following data query:

```
{
user(id:1) {
   name
   title
   avatarUrl
   timezone
   locale
   lastSeenOnline
   email
   phone
   Location
  accountOwner {
     name
     avatarUrl
   }
   tags {
     edges {
       node {
         label
         color
       }
     }
   }
   accountUsers(first:10) {
     edges {
       node {
```

```
          id
          avatarUrl
        }
      }
      pageInfo {
        totalAccountUsers
      }
    }
    recentConversations(first:10) {
      edges {
        node {
          lastMessage
          updatedAt
          status
        }
        pageInfo {
          totalConversationCount
        }
      }
    }
  }
 }
}
```

Their reaction?

> "We were able to expand our query based
> on the needs of the client and request a ton
> of information that usually would have taken
> multiple requests, a lot of boilerplate and un-
> necessary code written on both the client and
> the server.The payload now conforms to ex-
> actly what the customer wanted, and it gives
> the server the ability to optimize the resources
> necessary to compute the answer. Best of all,

this was all done in a single request. Unbeliev-
able. Welcome to the future."

Welcome to the future, indeed.

The Bad

There's a lot of good things about Relay, but there's some
underlying issues behind each benefit. Take, for instance,
the idea of mutation handling within Relay itself. When
mutations occur, especially when they're done in an op-
timistic update paradigm, you run into some significant
issues.

For instance, when querying a database with multiple
fields in the GraphQL schema, you're essentially updating
the client multiple times, the backend multiple times, and
hoping the related graph part each updates properly.
Nine times out of ten, they do — but even a single failure
could have a rolling effect on the backend at large.

Further, the idea of optimistic updates is great in theory,
but adds some logic responsibility on the client-side de-
veloper that may or may not be useful in all use cases.
While large edits would definitely benefit from such an
update scheme, simple updates and mutations do not
require simulation. What this means is a ton of logic
required to be implemented in the client-side team, with
the server-side team having very little responsibility for
ensuring cross compatibility.

There is, of course, the concern over loss of data and
validation in such a system as well. With each increasing
level of complexity in this situation, you're reducing the

efficiency that makes Relay such a good sell in the first place.

The main issue raised towards Relay, however, is that it's not technically necessary — while the functionality of Relay is impressive, there's a bevy of solutions both unique and already integrated into common languages and architectures that mirror the functionality such that Relay might simply be reinventing the wheel.

Standard GraphQL can be used without Relay, especially on projects with a smaller scope than Facebook, without much loss of functionality. Other solutions like Cashay mirror the cache storage solution for domain states in a simpler, more user-friendly format.

A REST Replacement

There's been some contention over exactly why we need Relay — or even GraphQL, for that matter. The development of GraphQL and Relay comes from the idea that there's something fundamentally wrong or done poorly in REST, a prospect that not everyone agrees with.

Let's state upfront that almost everything that GraphQL does can be done in REST, though perhaps less efficiently. Fetching complex object graphs is easier done in GraphQL, but the same functionality can be replicated with constructing an endpoint around the given data sets as a subset of the greater whole.

These complex requests are made easier with the fact that HTTP can send parallel network requests, though with greater overhead. And what limitations exist in this current solution, HTTP/2 is attempting to solve them, and (in some opinions) to great effect.

There's also the issue of just what Relay and GraphQL are designed to handle. The two were developed by Facebook for Facebook — a site that deals with thousands of meta-data points and relations that the average developer may never come across. In this case, for many people, it's a case of killing weeds with a flamethrower — yes, it's a solution, but it's an over-engineered one.

A good portion of criticism towards GraphQL and Relay in principle is the fact that many of the issues people have with REST aren't issues with RESTful architecture, but with the common implementations. REST supports content negotiation, is feature rich with everything HTTP has to offer, and has basic over and under fetching prevention solutions.

Essentially speaking, the problems that Facebook set out to fix are, in the eyes of many people issues of poor REST implementations and improper coding techniques, not the actual REST architecture itself.

Regardless of which side you fall on, it basically boils down to this — is adopting GraphQL and Relay worth the effort when considering that many of its features can be replicated somewhat in REST? What are your specific needs? Do you have Facebook level (and style) data to manage? If not, GraphQL, and thereby Relay, may not be the best choice in the world.

Conclusion

Relay is powerful — but like GraphQL, it's not a magic bullet. High-volume data in Relay is the gold standard for efficient data handling, but for standard data handling, you really have to question whether or not it is truly better

than proper RESTful design.

That being said, Relay is still in its infancy. It came from Flux, and just like Flux, is constantly being iterated upon and expanded into bigger, better things. As time goes on, much of the concern about Relay will likely be assuaged, with its functionality expanded further while being made more efficient.

10 GraphQL Consoles in Action
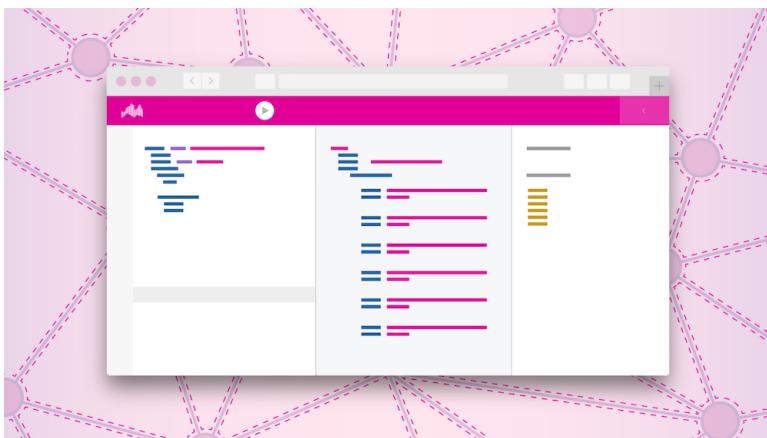
Most web API providers probably already know about **GraphQL** by now. It's the API query modeling language making waves throughout the API industry, allowing developer users to interact with a web API in an arguably cleaner method. The industry is still divided, though. While some venture to say it will replace REST in its entirety, others question whether GraphQL is worth it. Outside of Facebook, the founders of GraphQL, there are actually quite a few API providers actually implementing GraphQL in practice. Usage keeps popping up in new environments throughout many public APIs, often utilizing Graph*i*QL, the official open source visual **console** for interacting with API calls. These **visual interactive aids** are designed to help developer users query data and perform mutations. So, to add a little validation of the technology, let's check

out some companies officially implementing it with their **public API** programs. We'll list 10 APIs using GraphQL, and test out their **GraphiQL** consoles with mock use cases to sample the behavior.

Graph*i*QL: GraphQL API Explorer

Graph*i*QL is an open source IDE console for exploring a GraphQL server. It displays a code editor with autocomplete on the far left, comes with error highlighting, query results in the middle column, and a documentation explorer on the far right. You can demo the Star Wars API to test it out. The **Query Variables** on the bottom left show what is actually being manipulated in the request. For a developer consumer, a Graph_i_QL interface could be helpful for building GraphQL queries, viewing the types of data that are available, and more.
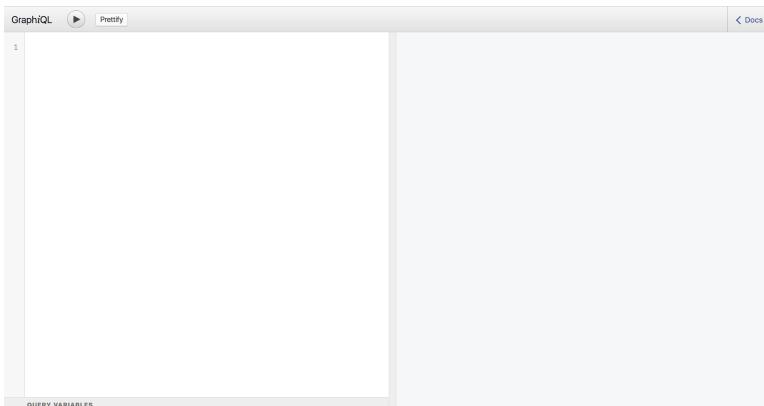
An empty GraphiQL console

1: GraphQL Hub

First off, let's consider the GraphQLHub aggregation. Created by Clay Allsopp, you can use it to explore **5 popular APIs**: Hacker News, Reddit, GitHub, Twitter, and Giphy using GraphQL. Though it's technically an unofficial proxy — not manufactured by these companies — it's a unique demonstration of the power of GraphQL to combine disparate elements into the same query style.

Query of recent Hacker News posts
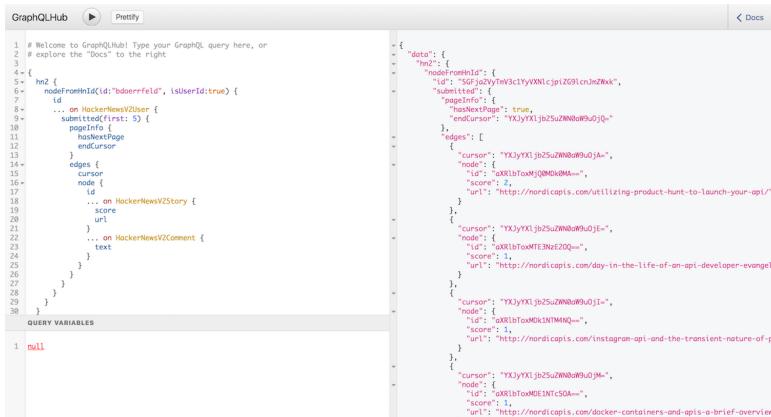
Say I wanted machine readable data of my last 5 posts to Hacker News, with the URL, score, and unique identifier. Using the GraphQL Hub interface, we can easily query this and read the results in JSON. Reminiscent of a Postman API collection, a GraphQL "hub" could be used by a team to easily interact with their own unique API dependencies.

2: Brandfolder

Brandfolder is an **asset management** service. With their API, developers can programmatically upload or retrieve brand asset files like images or fonts. Their GraphQL playground uses GraphQLify to provide a GraphQL server and schema to act as a middlemen between the Brandfolder backend and frontend, translating a JSON API to interact with React/Relay applications. To play around here you'll need an account and authorization.

Testing Brandfolder's GraphQL console will require account authorization

3: Buildkite

Buildkite specializes in helping companies with **continuous deployments**. They've been driving front end development through the `https://graphql.buildkite.com` endpoint. Their GraphQL API has even leapfrogged the web UI, as you can use it to **schedule builds**. Also built on the official open source GraphiQL, their GraphQL console is accompanied by a thorough step-by-step walkthrough; a good thing to consider to help new developers. As Tim Lucas of Buildkite states: "GraphQL was made to be human friendly and empower people to easily use, explore, integrate with APIs."

Buildkite demonstrates how to use GraphiQL to perform mutations on a deployment schedule

4: EHRI

The European Holocaust Research Infrastructure (EHRI) provides access to holocaust related data and **archive materials** held in institutions throughout Europe. In addition to offering a Search API to access the database, The EHRI portal provides an experimental GraphQL API. As EHRI describes, GraphQL is a "faster and more efficient option when the data required in known in advance." This echoes issues that some have with GraphQL being only accessible by developers familiar with the API.

5: GDOM

GDOM is a web parser open sourced on GitHub. Using the Graphene framework, GDOM uses GraphQL syntax

to **traverse** and **scrape** content off web pages. Another Hacker News example, here we use GraphQL syntax to traverse the Y Combinator home page and return top stories:

Graph*i*QL in the context of DOM traversal and scraping

6: GitHub

From using ChatOps to initiating other company cultural innovations, **GitHub** is quick to embrace cutting edge operational improvements. They've taken a stab at GraphQL as well. Currently in an early access alpha stage, you'll have to register for the GitHub pre-release program to use it (which just means signing in via your GitHub account and accepting a terms of use). Let's use the GitHub GraphQL API to retrieve some basic account information:

Retrieving account information with the GitHub Graph*i*QL API sandbox

When will this become a full release? Well, GitHub places their GraphQL on the long-term side of their Platform Roadmap.

7: HIV Drug Resistance Database

GraphQL may also be useful in the context of **academic research**. Stanford University is currently using Graph*i*QL to visually query their Sierra Web Service, which interacts with their curated database of HIV drug resistance data. Below is a GraphQL call to retrieve a **genomic** sequence from a virus with a specific mutation:

Sequence analysis on Stanford's GraphiQL console

8: Helsinki Open Data

HSL is the **public transportation **authority in the Helsinki, Finland metropolitan region. They produce data on routes, train stations, and more, which is all publicly accessible as open data on the HSL Developer Community. Say we

wanted to retrieve basic information on all transit stations in the Helsinki area; using the HSL Graph*i*QL console we can use a single GraphQL query to list stations and fields such as station name, specific location, and more:

A GraphQL query to retrieve information on transit stations in the Helsinki area. No "Prettify" option?! Oh well.

9: melodyCLI

MelodyCLI is a **dependency manager** for Go. It helps save time by caching Go repositories, using a GraphQL API to routinely expose metadata on dependencies. Below we've used the melodyRepo Playground Graph*i*QL interface to retrieve dependency information on a specific GitHub package:

MelodyCLI GraphQL playground call to sample dependency management functionality

10: SuperChargers.io

Superchargers.io is a web app map that provides the locations of Tesla stores, powering stations, and other service centers. Wholly based on a GraphQL API, the site does a nice job of onboarding developers to understand how queries behave.

Query to Superchargers.io retrieves all Supercharger stations in Europe.

11: Microsoft

Some teams at Microsoft are considering Graph*i*QL adoption. Dimitry Pimenov, a Program Manager at Microsoft Graph shared with Nordic APIs a test of their live GraphQL console. Essentially, how the tooling works is it takes in an OData API description, parses it, and creates a GraphQL schema and code generates the necessary resolvers. According to Pimenov, though there are some performance bottlenecks with this approach, it does provide a much better developer experience as far as resource discovery goes.

Does Increased Usage Validate GraphQL?

Now used within asset management, academic study, public transportation, and more, you can see that GraphQL is getting around. However, trends alone shouldn't validate the adoption of technology. So, *why* are more and more groups appreciating GraphQL? Well, as we've discussed before, there are some potential benefits:

- **Better query efficiency**: You can query multiple API calls simultaneously.
- **More elegant data retrieval**: Chaining calls can return a lot of data with a single request.
- **Low adoption overhead**: Since GraphQL is a wrapper that can be defined, you don't have to replace a REST system.

As the team behind melodyCLI put it:

> "We chose GraphQL to give us greater flexibility in how metadata is structured while giving clients more flexibility in how it's queried"

GraphQL should not be viewed as *the* stand-alone end all solution for everyone — many APIs still need developer documentation and strong API versioning strategies. A pain point is a lack of onboarding material in the Graph*i*QL interface; for developers unfamiliar with API calls, knowing how to structure a query is difficult as you will have to dig into documentation for the correct ids, **field** names, **Strings**, etc.

Tips on Making GraphQL / Graph*i*QL Awesome

With that in mind, for API providers considering providing a Graph*i*QL style IDE, here are some tips for getting the most out of it:

- **Introduce what GraphQL is**: The technology is still new to many, and they would love your help in structuring their first queries.
- **Have example GraphQL queries**: Auto-fill the Graph*i*QL skeleton with sample queries to help users get started. The Superchargers.io FAQ, for example, links to 4 different sample queries that populate a Graph*i*QL console.
- **Have supplementary tutorials or FAQs** on the side that explain your GraphQL schema and fields.
- **Brand it**: Lastly, it doesn't hurt to include your own skins. Include branding on top of the blank Graph*i*QL layout to make it your own.

More Resources:

- Facebook cites roughly 70 companies implementing GraphQL here
- "GraphQL — A Legit Reason to Use It," Edge Coders
- APIs.guru's ongoing Collection of GraphQL APIs
- To help you get started, see this comprehensive list of community GraphQL libraries and tools
- Bruno Pedro of Hitch hq provides the counter argument: 3 reasons not to use GraphQL

- Learn GraphQL: Step by step tutorial introduction to GraphQL

10 Tools and Extensions For GraphQL APIs
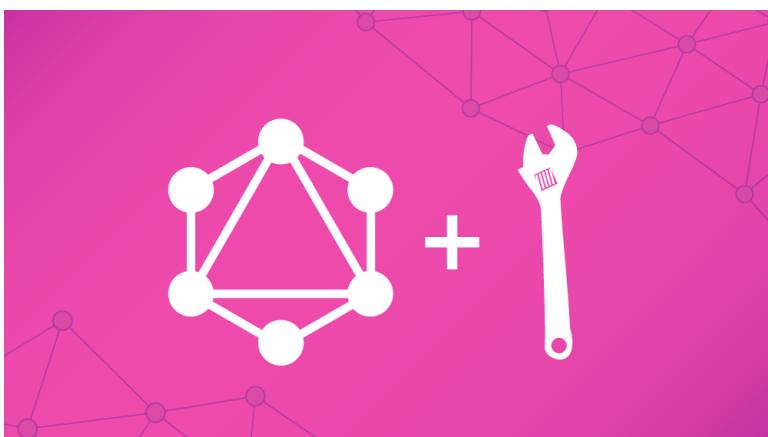
With the surge of interest in **GraphQL**, a vibrant new ecosystem of supplementary software has quickly emerged. Open source communities and entrepreneuring startups alike are validating new GraphQL use cases, filling in GraphQL implementation gaps, and enabling more and more developers to adopt GraphQL practices with decreased overhead through the use of some pretty awesome **tools**.

This is fantastic news for the query language, because Lee Byron, a designer of GraphQL at Facebook, recently noted that in order for GraphQL to reach ubiquity, the ecosystem desperately needs additional tooling:

> "Where we definitely still need focus is in building and improving the tools for using GraphQL

and lots of different environments"

Whether these tools handle **performance analysis** of graphical services, are pre-built GraphQL **servers**, or offer **IDEs** for exploring GraphQL schemas, a lot can be done to improve the GraphQL usability and adoption experience. In this article, we'll feature a short curated selection of various kinds of tools that can be used to explore GraphQL relationships, **document** a GraphQL API, **optimize** and monitor a GraphQL API, **map a REST API** to GraphQL, and much more. If you have undergone a GraphQL transformation, or are planning one out, you may want to consider whether extensions like these may be a helpful addition to your API's stack.

List of 10+ Tools & Extensions for GraphQL APIs

1: GraphiQL

An in-browser IDE for exploring GraphQL APIs Repo | Demo

As we've previously discussed, many GraphQL APIs use this open source console as an interactive API playground. Graph*i*QL is the popular Integrated Development Environment (IDE) for interacting with GraphQL API calls, enabling developers to query data and perform mutations.

This IDE is relatively easy to implement; for Node.js servers, express-GraphQL can automatically generate Graph*i*QL. Since it's built on React, Graph*i*QL can also be injected with unique CSS for custom branding.

The Graph*i*QL left column provides a space to enter queries with a syntax editor to search for relevant schema using autocomplete. When the request is run, the response is displayed on the right column

Having visual aid in the form of an API sandbox, playground, CLI, or other interactive means is an important facet of supporting a living, breathing API program that caters to the needs of your audience.

2: GraphQL Voyager by APIs.guru

 Represent any GraphQL API as an interactive graph Repo | Demo

If you're hoping to visually see how relational your data is, running it through GraphQL Voyager can make for a cool experiment. Voyager takes a GraphQL API and turns it into a visual **graph**; after setting a root schema, you can visually view how fields are connected to types. Voyager is interactive, too — selecting a type highlights the fields it is comprised of, and links to relevant data within the graph.

Echoing Lee Byron, APIs.guru founder Ivan Goncharov

recently told Nordic APIs:

> "Writing advanced tooling which works with any existing GraphQL API is possibly one of the critical things that REST APIs are missing."

GraphQL Voyager provides a left column that describes field information, and a visual interface that provides quick navigation. Users can also simplify the graph by eliminating Relay wrapper classes. Beyond being a sweet visualization, the Voyager tool could help companies envision their data model, and spark conversations on relational data — finally, we can view the "graph" behind GraphQL.
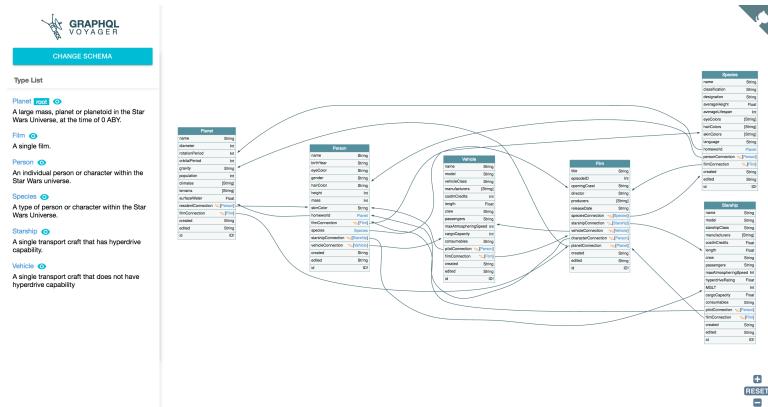
GraphQL Voyager, by APIs.guru

3: GraphCMS

Build a GraphQL Content API in Minutes

GraphCMS is an API-centric Content Management System (CMS) that is intimately tied with GraphQL. It lets you build

a hosted GraphQL backed for web apps, providing tooling to manage content. Users define data structures, validate them in a GraphQL console, and see the representation in the user interface, all within the same platform.

While GraphCMS is perhaps not a good fit for an existing API platform, it would cater well to a blog, web app, or other content structures that require the ability to programmatically share data. A GraphQL-based CMS would be an interesting alternative to traditional CMSs like Wordpress or Drupal, and would enable a more futuristic content management framework that comes API-equipped and is thus a more flexible management layer for end user interfaces.
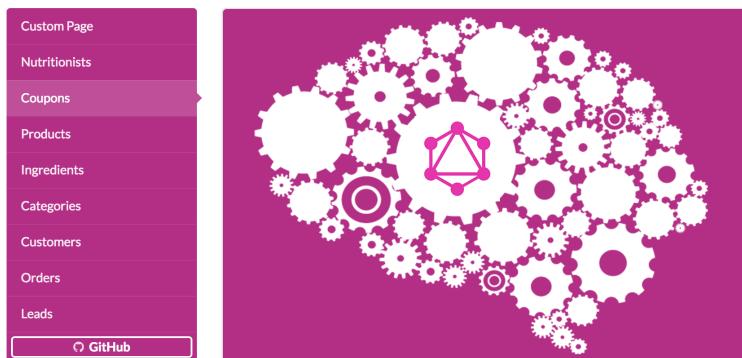
GraphCMS

4: GraphQL Docs by Scaphold.io

Clean, minimalist documentation for GraphQL APIs Repo | Demo

Need static documentation for a GraphQL API schema? Look no further than GraphQL Docs. The site will generate

simple, functional documentation in under 10 seconds given a GraphQL endpoint URL. Visitors can choose to catalogue their API documentation publicly, or to keep it private.

An open source equivalent for static documentation is graphdoc — fork this to generate and host GraphQL docs on your own. For both, the result is a clean interface, search mnenu, and links to schema definitions for object, and more.

![GraphQL schema definition for the Album Object in the Spotify API)

5: GraphQL Faker

Mock or extend your GraphQL API with faked data. Repo

If you are mocking up a barebones API, why not add some *lorem ipsum* data to test things out? With GraphQL Faker, GraphQL API developers can insert realistic data to mimic real life results. It's powered by faker.js, enabling developers to mock over 60 kinds of realistic data, like street address, first and last name, avatar images, and more. All you need to initiate it is to write a GraphQL IDL, and GraphQL faker provides some examples to get started within the IDL editor. It's as simple as adding a directive to a field or custom scalar definition:

```
type Person {
  name: String @fake(type: firstName)
  gender: String @examples(values: ["male", "female"])
}
```

6: Swagger to GraphQL

Migrate from REST to GraphQL in 5 minutes Repo

For those API providers entrenched with the traditional REST model, they may be indifferent to try out GraphQL. This **mapper** takes the pain away, taking a Swagger schema and automatically wrapping it to GraphQL. Tooling like this could be leveraged to decrease migration headaches, and enable a provider to maintain both a REST and GraphQL facade. For more, read the creator's thoughts on moving an existing API from REST to GraphQL.

7: GraphQL IDE

An extensive IDE for exploring GraphQL APIs Repo

An Integrated Development Environment, or IDE is typically a source code editor that maximizes developer productivity with things like debugging, code completion, compiling, and interpreting abilities. The GraphQL IDE is an alternative to Graph*i*QL, but the keyword here is *extensive*. It offers additional project management features, custom and dynamic headers, import/export abilities, and the ability to store queries and view a query history.

8: GraphQL Network

Chrome Devtool that provides a "network"-style tab to allow developers to debug more easily. Repo | Demo

Many of us techies are Google Chrome power users, so a GraphQL browser tab is a hot ticket or any serious

GraphQL API developer. Called GraphQL Network, this helpful tab is similar to viewing network requests in the Chrome DevTool — which is great for debugging RESTful API calls, but falls short when working with GraphQL, since "/graphql" is usually displayed as the endpoint, meaning that differentiating seperate requests is a pain.

The tab shows a concise list of recent GraphQL requests, listing HTTP method name, status, and type of request. In addition, GraphQL Network also gives a raw view of the string of GraphQL being sent, as well as a computed view as the server interprets it. Having separate entries laid out as well as being able to view the machine-readable fragments in this way could be helpful to **monitor** and **debug** GraphQL queries.

9: Graphcool

Flexible backend platform combining GraphQL+ AWS Lambda Site

With flexible, scalable, serverless architecture the rage, having an agile backend ready to go is an interesting prospect, especially when it's GraphQL compatible out-of-the-box. Graphcool is a platform to aid **GraphQL schema design** and app backend development, coming with a visual console to design and edit your your data schema, the ability to create advanced GraphQL data models, custom fields, and relations between data. With many integrations with popular tech like AWS Lambda, Algolia, and Auth0, Graphcool looks to be a powerful tool for modern database management.

Screenshot of the Graphcool console, where you can add new fields and data relations.

10: Optics by Apollo

Optimize GraphQL queries Website | Apollo on Github

To round off our list of GraphQL tooling, last but certainly not least is Optics, a product for monitoring GraphQL APIs. Optics is an analytics solution for GraphQL APIs that traces how queries run, helps you see what types of queries are being performed and their frequencies. ny As we've discussed before, API metrics are crucial, and any web API can benefit from adding an analytics solution to their platform. Seeing request volumes displayed visually, as well as having a better grasp on bottlenecks like latency issues is necessary for optimizing performance and improving page load times.

Optics is developed by Apollo, who are big contributors to the GraphQL ecosystem — their Apollo Client is a flexible, production ready GraphQL client for React and native apps, and Apollo also hosts events to spread GraphQL knowledge.
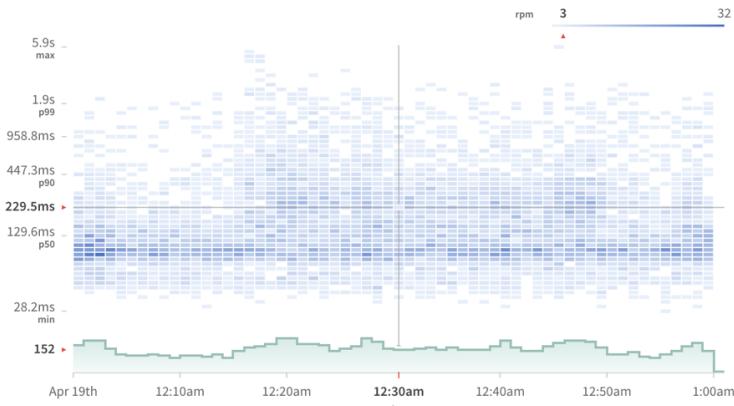
Example Optics visualization on latency trends over time

Final Thoughts

Other significant tools include Thunder, a GraphQL server for Go, Join Monster, an NPM package for arbitrating issues between GraphQL and SQL database, or Dgraph, a fast database that works with GraphQL.

> "Great tools are will what will make GraphQL ubiquitous" -Lee Byron, GraphQL/Facebook

There are many more that seem to be emerging daily; hopefully with the advent of tools like the ones mentioned in this article, more providers can reap the benefits of GraphQL and increase general extensibility with expending little effort.

Did we leave out any awesome GraphQL tools? Please comment below!

Resources

- Facebook's Draft RFC Specification for GraphQL
- Visit the Awesome GraphQL List for many more GraphQL tools.

What The GraphQL Patent Release Means For the API Industry

GraphQL has driven much of the conversation around modern API web design, and for good reason — it's powerful, extensible, and very useful for high data query applications. The ability to request data in a predetermined, knowable format, and the ability to collate endpoints into a single external point, has made GraphQL something that powers some pretty huge projects.

Unfortunately, reliance on a near-ubiquitous format comes with a negative – when a change is made to GraphQL, its effects are wide-ranging and significant. This can be seen in the recent troubles concerning GraphQL **patent licensing** and the disputes that have arisen – a simple change of license format and the resultant understanding

(and in some cases, misunderstanding) in the community has sent shockwaves through the API landscape.

Today, we're going to talk about some purported issues related to the **GraphQL patent** release to provide context and a bit of clarity. We'll take a look at whether or not the GraphQL license issue is as significant as first thought, and more to the point, what the newly released patent scheme from GraphQL actually means for most providers.

Background

For those who don't know what GraphQL is, it can broadly be summarized as an application layer query language. GraphQL interprets strings from the client, and returns data in an understandable, predictable, pre-defined manner. This is a very short, summarized explanation of what GraphQL does, but there is so much more that makes it special – for a more complete summary, check out our piece on the potential benefits of GraphQL adoption.

What is more important to this discussion is how **GraphQL** was created. After internal development began in 2012 at Facebook, GraphQL was released publicly in **2015**, offering an alternative to the dominant architectures of the API space, notably REST.

GraphQL was initially developed to help Facebook cope with challenges in **fetching** specific data from their collection of services without introducing bloat and complexity. By allowing the client to define the expected data format, Facebook, through GraphQL, was able to design a methodology to deliver data in a more usable, efficient manner.

This usability and efficiency was very attractive to many early adopters. Companies like Pinterest, GitHub, and Shopify quickly began to use GraphQL either wholesale or in part. During the rush to adopt GraphQL, however, there were looming, unresolved questions arising from GraphQL's creation. There was no licensing language in the public release, or information on whether or not the language was patented, and whether or not legal ramifications were possible. Many questions were left unanswered.

Developers Express Concern

In September of 2017, **GitLab** officially froze their GraphQL project due to these concerns. In a GitLab issues post, senior director of legal affairs Jamie Hurewitz stated that the BSD+Patents license, which Facebook had adopted to try and alleviate patent concerns, was concerning.

> "If we were to allow this license, it could lead to potential future conflicts with software licensed under Apache. Also, we could be impairing the future rights of our customers. Essentially, this is **not really an open source product** based on the implications of the license. While there is no payment of cash, payment is in the form of giving up future rights."

The license that had been inserted into the GraphQL release allowed for the use of GraphQL under certain terms that removes your license to use GraphQL under the following terms:

> "The [patent] license granted ... will terminate ... if you ... initiate directly or indirectly, or take a direct financial interest in, any Patent Assertion: (i) against Facebook ... (ii) against any party if such Patent Assertion arises ... from any software... of Facebook ... or (iii) against any party relating to the Software."

In other words, developers were free to use GraphQL, as long as they never challenged Facebook over patent infringement from anything built upon their system by Facebook. Unresolved issues caused outrage among programmers. It was very significant for the community of GraphQL adoptees, and led to many developers pulling away from Facebook and its new patent.

Developer and attorney Dennis Walsh subsequently discussed this issue in a lengthy analysis on Medium, wherein he posited that "*Whether Facebook wants to assert these patents is the province of gut feelings and lore. I don't believe that Facebook ever offensively litigated a patent, but the potential for litigation is more than theoretical — it's very real if they choose that path*." It was clear that for developers like Walsh, the idea that you had a license to GraphQL as long as you didn't try to patent or protect your own processes and developments was troublesome.

On August 30th, the co-inventor of GraphQL at Facebook, Lee Byron, addressed concerns on GitHub, and stated that he was "aware" of the patent problem, and that it was currently being worked on. He stated:

> "I'll bring this to the attention of our legal council for their suggestion on how to resolve this issue. We definitely want to ensure the community has all necessary rights to be able to

use GraphQL! I'll make sure we get a speedy resolution."

Finally, on September 26th, 2017, Facebook relicensed GraphQL under an **OWFa 1.0 license**, granting a perpetual license to users of GraphQL. While this definitely improves the situation, easing many of the developer issues surrounding the issue, there may still be some cause for concern.

Out of the Frying Pan...

There's still some cause for concern around the new licensing scheme, however. Facebook has broadly adopted the **MIT license**, which doesn't expressly include a patent grant. There was some concern expressed, such as that of RedMonk founder Stephen O'Grady, that adopting MIT over the Apache license, which gave a more clear patent situation to developers, created new concerns:

> "The problem is that by choosing this approach, Facebook does not convey with the MIT license any patent grants as they would have under the Apache. If Facebook has patents that read on React, in other words, users of that software are not given an explicit license to them via MIT, only an untested implicit license."

To O'Grady, this essentially means that Facebook has resolved one patent issue by introducing a second.

Many of these issues are ones that must be tested to be confirmed – in other words, until a patent infringement

suit is drawn against or by Facebook, **most of the concerns are simply conjecture**. Even so, the fact remains that this continues to be, for some, a **concerning development**. This is certainly an artifact of the API development world as a whole, and is something that will have to be tested, vetted, deprecated, and rebuilt as time goes on.

Patents in the United States were originally envisioned during a time where reproduction could take months at great expense to the creator. Patents were put in place to "promote the progress of science and useful arts" according to its Constitutional call. At the time, the idea of rapidly developing upon an open source standard and forking these projects into additional sub-developments was simply impossible to imagine, and as such, applying patents to the modern web world is resulting in some significant complexity.

Is This a Concern?

That being said, there are just as many pundits suggesting that the issue has **largely been blown out of proportion**. The termination of a patent due to legal proceedings is common in some spaces, and in fact has been practiced by policies applied to such juggernauts as the WebM codec from Google. Such patent terminations, termed "defensive termination provisions", are largely common in enterprise implementations, and in many cases, their full teeth have yet to be significantly bared in a legal proceeding.

It should also be noted that the defensive termination provision as noted in Facebook's GraphQL documentation was not nearly as extreme as some others. The

Mozilla Public License, for instance, not only strips your patent license in response to legal challenges, but your copyright license as well – without the copyright license, though you may lose the patent license, you could still in theory continue to use the code. With a copyright termination, however, you would be legally obligated to stop using your derivative code entirely.

That is obviously a much more extreme type of license termination, but the issue remains at the forefront of many people's discussions when adopting GraphQL.

Oil and Water

While some have tried to connect some sort of nefarious purpose to the patent issues at hand, the truth is that what we're witnessing is a **collision of open source ethos with corporate reality**. Facebook is a corporation, and as such, it's to be expected that they would do everything they can to look after their best interest. This is an unfortunate fact of the corporate world, especially for a publicly traded company.

On the other hand, we have open-source philosophy; the idea that code should be shared, developed upon, refined, and extended, with minimal if any protections applied to the foundational code base. While this is, in theory, benefits everyone – resulting in increase security, more refined code, and wider adoption – the fact is that a corporation like Facebook would likely find this sort of approach dangerous considering how many of their core functions are driven by the codebase that is being patented.

Perhaps the most clear reason for much of this concern

is the fact that **intent** can't really be communicated in patent legalese – whether or not Facebook would use such a patent agreement as a means to attack developers is an entirely different conversation from whether or not the actual patent license as it currently is would be acceptable to most developers. Unfortunately, the two topics are being conflated, leading to the issue at hand.

Final Thoughts

So what does this all mean for the API space? It's easy to assign nefarious intent to Facebook, but the reality is that Facebook is a public company – as many companies in the API community now are. This means that they have certain concerns that they need to address, and certain expectations concerning the use of their applications and codebase.

That being said, stifling open source implementations is a significant issue – and while that's not what's happening in this case, the reactions of some providers to the relicensing (such as dropping GraphQL in fear of possible legal issues) is understandable. That should be the takeaway to all of this – while there are concerns about patent licenses within the GraphQL license language, **much of the fear is based on conjecture**, and if the possibility of future concern is significant enough to worry an organization, they should consider moving away from GraphQL and into a more open-source friendly, freely licensed alternative.

If you do not intend on creating something that is monetized, or the new re-licensing scheme proposed is an acceptable mitigation, then by all means, continue to use GraphQL. If you still fear the possible issues in the future, ensure that your API can behave for a time without

GraphQL or any GraphQL-derived functionality. For those somewhere on the fence, do keep in mind that until there is court action taken, these fears all conjecture.

The GraphQL Community At Large

The community surrounding GraphQL has grown exponentially since developers caught wind. Now, with the evergrowing proof of its use in the case studies cited in this volume, we see GraphQL moving from fad status into enterprise adoption.

If you would like to participate in the economy forming around GraphQL tooling, or to learn more about GraphQL strategy, there are events solely realted to GraphQL. We'll surely cover the topic on the blog, but in-person GraphQL-specific events are listed on this event page site.

Stay Connected

Thank you again to our readers, event attendees, and event sponsors and partners. If you appreciate what we're doing, consider following @NordicAPIs and signing up to our newsletter for curated blog updates and future event announcements.

Nordic APIs Resources

 Visit our Youtube Page for Deep Dives

Watch full videos of high impact API-related talks on our Youtube Channel. In 2018 and beyond, we will be featuring seminars on niche API topics in our **LiveCasts**; hour-long webinars on advanved topics given by core community members.

More eBooks by Nordic APIs:

Visit our eBook page to download these eBooks for free! If you choose to purchase through LeanPub or Amazon, all proceeds go to the Red Cross's efforts in Sweden.

How to Successfully Market an API: The bible for project managers, technical evangelists, or marketing aficionados in the process of promoting an API program.

The API Economy: APIs have given birth to a variety of unprecedented services. Learn how to get the most out of this new economy.

API Driven-DevOps: One important development in recent years has been the emergence of DevOps â€" a discipline at the crossroads between application development and system administration.

Securing the API Stronghold: The most comprehensive freely available deep dive into the core tenants of modern web API security, identity control, and access management.

Developing The API Mindset: Distinguishes Public, Private, and Partner API business strategies with use cases from Nordic APIs events.

Endnotes

Nordic APIs is an independent blog and this publication has not been authorized, sponsored, or otherwise approved by any company mentioned in it. All trademarks, servicemarks, registered trademarks, and registered servicemarks are the property of their respective owners.

- Select icons made by Freepik and are licensed by CC BY 3.0
- Select images are copyright Twobo Technologies and used by permission.

Facebook | Twitter | Linkedin | Google+ | YouTube

Blog | Home | Newsletter | Contact