GraphQL vs REST: Overview

A few months back I wrote a comparison between RPC and REST for Smashing Magazine, and now I want to talk about the differences between REST and GraphQL: the new kid on the block.

GraphQL is incorrectly considered by some to be a "replacement" to REST. GraphQL is a newer concept, being released by Facebook publicly in 2015, whereas REST was a dissertation published by Roy Fielding in 2000, popularized by companies like Twitter (quite inaccurately) in 2006.

This article aims to cover a few notable differences, and make the following points:

1. REST and GraphQL are totally different
2. GraphQL isn't a magic bullet, nor is it "better"
3. You can definitely use both at the same time
4. GraphQL is dope *if used for the right thing*

A few quick differences

REST is an architectural concept for network-based software, has no official set of tools, has no specification, doesn't care if you use HTTP, AMQP, etc., and is designed to decouple an API from the client. The focus is on making APIs last for decades, instead of optimizing for performance.

GraphQL is a query language, specification, and collection of tools, designed to operate over a single endpoint via HTTP, optimizing for performance and flexibility.

One of the main tenets of REST is to utilize the uniform interface of the protocols it exists in. When utilizing HTTP, REST can leverage HTTP content-types, caching, status codes, etc., whereas GraphQL invents its own conventions.

Another main focus for REST is hypermedia controls (a.k.a HATEOAS), which lets a well designed client run around an API like a human runs around the Internet; starting with a search for "How to complete my tax returns", reading a perfectly relevant article, and after a few clicks ending up on BuzzFeed article about Miley Cyrus throwing Liam Hemsworth a "Weed-Themed" birthday party.

If your API is not using hypermedia controls, then GraphQL could be a more relevant approach, because you weren't really using REST anyway.

This article will not attempt to point out a winner, but we're going to look at a few areas where the two differ. Don't get mad that a lot of the sections say "it depends", because the winner in each section really depends on what your API is doing, and how. GraphQL comes out stronger in some areas, REST in others, and sometimes they're both kinda terrible. Let's dig in!

Is the API More Than Data Transfer?

One of the most common tasks REST APIs provide is CRUD via JSON, but it can do plenty more than that, such as file uploads.

Uploading an image in the HTTP body can look a little something like this:

```
POST /avatars HTTP/1.1
Host: localhost:3000
Content-Type: image/jpeg
Content-Length: 284

raw image content
```

Leveraging a cool part of HTTP (and therefore REST), API developers can support `application/json` requests on the same endpoint to handle the upload slightly differently, and offer URL-based uploads too:

```
POST /avatars HTTP/1.1
Host: localhost:3000
Content-Type: application/json
```

```
{
  "image_url" : "https://example.org/pic.png"
}
```

Generally the APIs I have worked on have enjoyed both, which is super handy as iOS often sends photos directly from local files, and web clients often send a URL to the user's Facebook display picture.

If we were talking about uploading videos or other large files, I would (as this article suggests) switch to another approach and have a dedicated service which handles the upload, leaving the main API to only accept metadata; title, description, tags, etc.

This is the approach you are forced to take with GraphQL, because you can only speak to GraphQL in terms of fields:

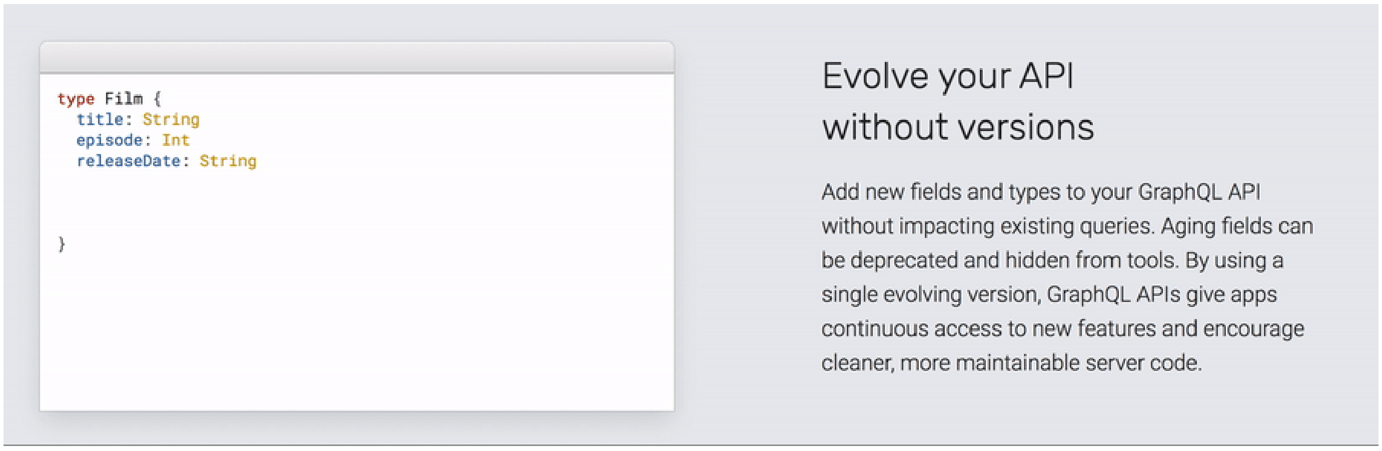
```
POST /graphql HTTP/1.1
Host: localhost:3000
Content-Type: application/graphql

mutation addAvatar {
  addAvatarFromUrl(image_url: "https://example.org/pic.png") {
    id,
    image_url
  }
}
```

Some will argue that this is more "clean", and it is, it's very clean, but being forced to create another service is overkill for smaller images, especially early on. Another approach is to upload directly to Amazon S3, forcing a dependency on clients and potentially letting your tokens leak, or... use multipart uploads, which are a super hacky approach that depends on if the server and various clients can even support it.

This is one area where REST holds strong. Some would say that REST handling CRUD and arbitrary stuff is confusing, but this is a core tenet of what makes REST so useful. A REST API can do anything, not just send fields backwards and forwards - even if that is how REST is often used.

Both GraphQL and REST Prefer Evolution

One false advertised benefit of GraphQL I've seen suggested (in quite a few locations) is that you "never have to version anything."



```
type Film {
  title: String
  episode: Int
  releaseDate: String



}
```

Evolve your API without versions

Add new fields and types to your GraphQL API without impacting existing queries. Aging fields can be deprecated and hidden from tools. By using a single evolving version, GraphQL APIs give apps continuous access to new features and encourage cleaner, more maintainable server code.

How versioning works in GraphQL. (source: http://graphql.org/)

The suggested approach is to add new fields and deprecate old ones, which is a concept well known in REST as evolution.

Deprecating, communicating to third-parties, monitoring usage, and removing at an acceptable time, is exactly what many REST APIs have been doing forever.

> The reason to make a real REST API is to get evolvability … a "v1" is a middle finger to your API customers, indicating RPC/HTTP (not REST)
>
> — Roy T. Fielding (@fielding) September 8, 2013

Although GraphQL and REST can (and should) version via evolution just as easily, GraphQL really helps API developers out when it comes to deprecations.

GraphQL makes Deprecations Awesome

One area where GraphQL excels is to make monitoring field usage incredibly easy at a technical level. GraphQL clients are forced to specify the fields they want returned in the query:

```
POST /graphql HTTP/1.1
Host: localhost:3000
Content-Type: application/graphql

{
  turtles(id: "123") {
    length,
    width,
    intelligence
  }
}
```

Tracking this would be trivial, but a REST API acts a little differently. Whilst all REST APIs make the base endpoint available via `/turtles/123`, not all APIs offer sparse fieldsets: `/turtles/123?fields=length,width,intelligence`. Of those that do offer it, it's almost always optional.

If a REST API client calls `/turtles/123`, then they could be using *any* field in that response. Imagine the API plans to get rid of `intelligence` (because all turtles are geniuses), how do the API developers know which clients are using that field?

An RPC approach is to make a new `/getTurtle2` endpoint (or `/v2/turtles/123` in a RPC API pretending to be a REST API), and tell people to use that new one.

One approach in REST APIs is to email warnings to whatever email address was entered when the client signed up for OAuth tokens (like Facebook previously did), maybe offering a feature flag so various clients can flip the switch when they are ready.

Another approach would be to create a new version of the resource, meaning instead of calling `GET` with `Accept: application/vnd.turtlefans.com+v1+json` they should start calling with `Accept: application/vnd.turtlefans.com+v2+json`.

All of the above approaches suffer the same issue, and that is that an entire version can be overkill for a simple change, and you might be forcing the developers to look into a version upgrade without needing to.

For example, if the client is requesting `application/vnd.turtlefans.com+v1+json` and the API is removing `intelligence` in v2, the API developers know not to drop v1 until the last of the clients are upgraded. Sadly, if clients are calling v1 *but not using the `intelligence` field, the API developers have no idea!* The API developers only know that the clients want v1, not what they're using of that v1 resource.

GraphQL makes it easy to track specific field usage to a client, meaning API owners can reach out to only those clients using fields that are heading out, or for internal projects you could have errors thrown in development/staging environments. I had a vague brain fart about doing this latter option for non-GraphQL HTTP APIs, but have not had a chance to see it through *yet*.

> Thinking about a HTTP format for handling deprecations, because global and resource versioning is a PITA. Early draft pic.twitter.com/KTqPhevVU8
>
> — Phil 🏅 (@philsturgeon) August 12, 2015

That could help in some situations, but would certainly be baked into things the same way it is in GraphQL.

GraphQL making field deprecation easier was a point brought to my attention by Tom Clark, super-smart Head of Devops at WeWork.

GraphQL Puts Client Performance First

GraphQL is always the smallest possible request, whilst REST generally defaults to the fullest. It's common practice to offer options like `?fields=foo,bar` or partials. Google recommend doing this for HTTP APIs, whatever that's worth.

Even if a REST API returns only a basic partial by default, there are still more bits being transferred over the wire by default, than with the GraphQL approach. If a client needs a field, they request it, and if the API adds a new field, clients don't get it, unless they discover that field in a blog post or whatever and add it to the GraphQL query.

REST Makes Caching Easier At All Levels

Caching for HTTP, common usage in REST APIs, and different types of caching are huge topics, and something that I had to split out of this article. I'll post a followup shortly, which will be posted on the APIs You Won't Hate Newsletter.

> In an endpoint-based API, clients can use HTTP caching to easily avoid refetching resources, and for identifying when two resources are the same. The URL in these APIs is a globally unique identifier that the client can leverage to build a cache. In GraphQL, though, there's no URL-like primitive that provides this globally unique identifier for a given object. It's hence a best practice for the API to expose such an identifier for clients to use. – Source: graphql.org

REST over HTTP uses a whole pile of HTTP conventions that make existing HTTP clients, HTTP cache proxies, etc., all work easily to benefit both API clients and API servers, but with GraphQL... tough. Reorganize your data stores, use a bunch of Redis, and hope clients are caching too.

GraphQL is a Query Language First

A very fundamental difference here of course is that only one of these is a query language. REST APIs are often created initially simple, then slowly more and more query language-like features are tacked on over time.

The most reasonable way to provide arguments for queries in REST is to shove them in the query string. Maybe a `?status=active` to filter by status, then probably `sort=created`, but a client needs sort direction so `sort-dir=desc` is added.

Some APIs use also end up with arguments in the query string that are not related to filtering, more like options. In the GraphQL example they specify the unit they'd like to see a `height` returned in:

```
{
  human(id: "1000") {
    name
    height(unit: FOOT)
  }
}
```

```
{
  "data": {
    "human": {
      "name": "Luke Skywalker",
      "height": 5.6430448
    }
  }
}
```

That's pretty darn handy, and it stops the confusion of `?status=active` being a filter but `?unit=foot` being a display option. I have seen `?filter:status=active`, `?filter[status]=active`, etc., but this is still a bit of a mess.

In these scenarios, GraphQL beats the pants off of REST APIs that try to hand-roll their own query language functionality, but I would suggest hand-rolling your own query language is a bad idea anyway. GraphQL gives you a query language syntax, and SDKs for your programming language to fetch the right models for that, but so do things like OData, a project with the slogan "A Better Way to REST".

This is another example of folks saying that REST cannot do something, just because many REST APIs don't do it, or because its implemented poorly by many that do. Saying that, remember that the more customisation an endpoint-based API adds to the request, the fewer cache hits are likely at a network caching level, making the REST/HTTP approach less rewarding, and forcing you down the route of the application caching and database reorganisation that GraphQL forces anyway.

If your API is highly customizable, it's another ticked box for considering GraphQL.

GraphQL Removes "Include vs Endpoint" Indecision

Another customisation consideration that comes up a lot is when to offer included relationships, and when to use another endpoint. This can be a difficult design choice, as you want your API to be flexible and performant, but includes used past the most trivial uses can be the opposite of that.

You start off with overly simplistic examples like `/users?include=comments,posts` but end up on `/trips?include=driver,passengers,passengers.avatar,passengers.itineraries` and worse.

I call this the "Mega Include of DOOOOOOM", and it is crap solution to a genuine concern, when clients strive for performance over all else. Includes are a common convention found in REST APIs, recommended in specs like JSON API, but actually bends the rules of REST a bit.

REST would call for a HATEOAS approach, which would have you make one call to the `/trips` endpoint, then hit `"links": { "driver": "https://example.com/drivers/123" }`, and again for passengers, and again for child data of each of those passengers. In this case, it would be worse than the dreaded `n+1`, and more like `n+(2+(num passengers*2))`!

Includes start off with the best of intentions, but can grow to be a bottleneck in a REST API, with unwieldy queries happening against the data store. GraphQL not only removes this design question, but forces the include approach, and forces you to consider efficient means of fetching this data.

This is a big win for GraphQL, as forcing the include approach, and forcing efficient "Mega Includes of Doom", the GraphQL will be both efficient and consistent. Trying to make a REST API be include-only would be bizarre and unusable, so consistency will never work there.

GraphQL will not help with your database queries, so you still need to fine tune those indexes and cache fragments of the data intelligently. If you skip this, clients will surprise you. At a previous job we had a poorly-tuned mega include from the iOS app taking about 20s+ to respond, which was bloody terrifying. We almost just had them curl down a `latest.sqlite` to process locally. 🤣

Scoped Includes Suck in Both

Using that trip example again, a client may be including passengers but realize that includes historical passengers. Don't want to see people who left the carpool? The client has to iterate through passengers locally, removing any models where `model.status != "active"`, which is a waste of processing on the client side.

The REST way to do this would be to use `/passengers?trip=123&status=active` which is obviously more flexible, but clients will skip this due to the extra requests required.

With client-side filtering being a poor choice, and the extra request not being ideal, REST API developers are often forced to add a new include: `/trips?include=activePassengers`. These are tough to pre-empt, so I was hoping GraphQL could help clients define their own scopes, filtering these includes to be the appropriate data themselves, which would help identify the scoped includes the API should add as convenience methods.

It seems like GraphQL doesn't help API developers in this instance, but there does seem to be talk of adding `@filter` to do this in the future.

GraphQL Devolves Power to Clients

Another question that comes up a lot for REST API developers, is:

> Our iOS app, Android app and web app are very different from each other. How would we return different data for each client?

We all start off trying to make our REST APIs so generic they can be used by anything, but as the mega-include problem indicated, clients want and need a lot, and they're always trying to reduce calls.

Some basic solutions for outputting different data per-client in a REST API are:

Create custom endpoints: `/iphone_snapshot`. We've done this at a past company where the mobile data was completely different to anything else. It felt dirty, was almost definitely RPC, but it got the job done. Having the supposedly generic REST API know so much about a specific client defeats the purpose a little bit, but we needed to get the data to the iPhone and that was what happened.

Create custom representations: Using `Content-Type: application/vnd.turtlefans.com+v1+iphone+json` which had its own custom serializer would supposedly be a bit more REST in that it's just another representation, but equally odd.

Custom APIs! This concept was/is in use by Netflix if I remember rightly, and the idea is to have one API for each of their clients. There is an Android REST API, a iOS REST API, Web REST API, etc. Each of these is tailored to perfectly match the needs of their specific teams, and makes requests back to the central Generic REST API. Requiring multiple APIs, multiple development teams, etc, is certainly out of the reach of many organizations, but it does solve the issue nicely.

Or... **GraphQL!** Instead of making custom endpoints, custom representations, or custom APIs, the clients simply write their own queries. This moves the responsibility out of the hands of the API developers, and into the clients, who then get to write in their language of choice, instead of bugging the API team to write it in a different language.

Whether you need this paradigm shift or not is entirely up to how similar your clients are. If you're a private/internal API and your clients are all practically identical, then you certainly don't want them all handling things.

But, if you have a multitude of different clients, or are public (therefore have no idea how the API will be used), GraphQL quickly starts to seem more appealing.

Why Not Use Both?

The biggest oddity I notice in the "GraphQL vs REST" conversation, is the falsehood that *you must pick one*.

In a world of SoA, you are likely to have multiple services, which expose multiple APIs. In the RPC vs REST article I point out that some services might be REST and some might be RPC, and you can absolutely throw some GraphQL in with your REST.

One mix of REST and GraphQL could just be adding a `/graphql` endpoint to `api.whatever.com` and having that as your GraphQL endpoint on an REST API.

Another, is one that has been vaguely tossed around at work, which is the idea of having one GraphQL API, acting as a gateway to our other multiple REST APIs.

Having one GraphQL server act as a sort of data proxy, giving one entry point for mixed data, one Authentication scheme despite each REST API having its own "unique" approach to tokens, one HTTP call for clients - despite it hitting multiple actual REST APIs, etc., would be a damn powerful thing.

Summary

Ask yourself - at the very least - these following questions:

- How different are your clients from each other?
- Do you trust your clients to handle caching?
- Do you want "dumb clients" that act like crawlers - knowing very little about the API, or clients that own a lot of the logic and know a lot about the API, using it simply as a data transfer?
- Are you ok letting go of HTTP debugging proxies, cache proxies, all the knowledge your team have around HTTP, etc?
- Are you just doing basic CRUD with simple JSON documents, or will your API need file upload/download too?

If your REST API is following good practices, like allowing careful evolution instead of global versioning, serializing data instead of returning directly from data store, implementing sparse fieldsets to allow slimming down response sizes, GZiping contents, outlining data structures with JSON Schema, offering binary alternatives to JSON like Protobuff or BSON, etc., then the advertised advantages of GraphQL seem to fall a bit short.

If you need a highly query-able API, expect an array of clients that need small and different data, and can restructure your data to be inexpensive to query, then GraphQL is likely to fit your needs.

Beyond these various pros and cons for GraphQL listed above, what I really enjoy about GraphQL being an option, is having a new alternative to REST when considering an API. An alternative that is well documented, with a full specification, with a lovely marketing page, with an official reference implementation in JavaScript, and which avoids some of the tricky design choices REST forces you to make.

I like that many folks will no longer treat REST like a shiny unicorn, struggle to implement REST properly, then call it REST just so they look good and have a ✅ for their marketing. I just hope that not too many people treat GraphQL like a shiny unicorn instead.

Swapping one false idol for another isn't going to make the API world a better place.

More on GraphQL:

- GraphQL vs REST: Overview
- GraphQL vs REST: Caching
- You Might Not Need GraphQL
- Representing State in REST and GraphQL

Previous: Building APIs with Rails: Handling Errors Nicely

Next: GraphQL vs REST: Caching