

Why, after 6 years, I'm over GraphQL

May 24, 2024

GraphQL is an incredible piece of technology that has captured a lot of mindshare since I first started slinging it in production in 2018. You won't have to look far back on this (rather inactive) blog to see I have previously championed this technology. After building many a React SPA on top of a hodge podge of untyped JSON REST APIs, I found GraphQL a breath of fresh air. I was truly a GraphQL hype train member.

However, as the years have gone on and I have had the opportunity to deploy to environments where non functional requirements like security, performance, and maintainability were more of a concern, my perspective has changed. In this article I would like to take you through why today, **I would not recommend GraphQL to most people**, and what I think are better alternatives.

Throughout I will use Ruby code with the excellent `graphql-ruby` library for examples, but I believe many of these problems are ubiquitous across choice of language / GraphQL library.

If you know of better solutions and mitigations, please do leave a comment. Now, lets begin...

Attack surface

It was obvious from GraphQL's beginning that exposing a **query language** to untrusted clients increases the attack surface of the application. Nevertheless, the variety of attacks to consider was even broader than I imagined, and mitigating them is quite a burden. Here's the worst I've had to deal with over the years...

Authorisation

I think this is the most widely understood risk of GraphQL, so I won't go into too much depth here. TLDR: if you expose a fully self documenting query API to all clients, you better be damn sure that **every field** is authorised against the current user appropriately to the context in which that field is being fetched. Initially authorising **objects** seems like enough, but this quickly becomes insufficient. For example, say we are the ~~Twitter~~ X 🙄 API:

```
query {  
  user(id: 321) {  
    handle # ✅ I am allowed to view Users public info  
    email # ❌ I shouldn't be able to see their PII just because I can  
  }  
  user(id: 123) {  
    blockedUsers {  
      # ❌ And sometimes I shouldn't even be able to see their public i  
      # because context matters!  
      handle  
    }  
  }  
}
```

One wonders how much GraphQL holds responsibility for Broken Access Control climbing to the [OWASP Top 10's #1 spot](#). One mitigation here is to make your API secure by default by integrating with your GraphQL library's [authorisation framework](#). Every object returned and/or field resolved, your authorisation system is called to confirm that the current user has access.

Compare this to the REST world where generally speaking you would authorise **every endpoint**, a far smaller task.

Rate limiting

With GraphQL we cannot assume that all requests are equally hard on the server. There is no limit to how big a query can be. Even in a completely empty schema, the types exposed for [introspection](#) are cyclical, so its possible to craft a valid query that returns MBs of JSON:

```
query {  
  __schema{  
    types{  
      __typename  
      interfaces {  
        possibleTypes {  
          interfaces {  
            possibleTypes {  
              name  
            }  
          }  
        }  
      }  
    }  
  }  
}
```

```
}
  }
    }
      }
```

I just tested this attack against a **very** popular website's GraphQL API explorer and got a 500 response back after 10 seconds. I just ate 10 seconds of someone's CPU time running this (whitespace removed) **128 byte** query, and it doesn't even require me to be logged in.

A common mitigation¹ for this attack is to

1. Estimate the complexity of resolving **every single field in the schema**, and abandon queries that exceed some maximum complexity value
2. Capture the actual complexity of the run query and take it out of bucket of credits that resets at some interval

This calculation is a **delicate affair to get right**. It gets particularly tricky when you are returning list fields whose length is not known prior to execution. You can make an assumption about the complexity of these, but if you are wrong, you may end up rate limiting valid queries or not rate limiting invalid queries.

To make matters worse, its common for the graph that makes up the schema to contain cycles. Lets say you run a blog with Articles which each have multiple Tags, from which you can see associated Articles.

```
type Article {
  title: String
  tags: [Tag]
}
type Tag {
  name: String
  relatedTags: [Tag]
}
```

When estimating the complexity of `Tag.relatedTags`, you might assume that an article will never have more than 5 tags, so you set this fields complexity to 5 (or 5 * its children's complexity). The problem here is that `Article.relatedTags` can be its own child, so your estimate's inaccuracy can compound exponentially. The formula is $N^5 * 1$. So given this query:

```

query {
  tag(name: "security") {
    relatedTags {
      relatedTags {
        relatedTags {
          relatedTags {
            relatedTags { name }
          }
        }
      }
    }
  }
}

```

You expect a complexity of $5^5 = 3,125$. If an attacker is able to find an Article with 10 tags, they can trigger a query with a “true” complexity of $10^5 = 100,000$, 20x greater than estimated.

A partial mitigation here is to **prevent deeply nested queries**. However, the example above demonstrates that this is not really a defense, as it's not an unusually deep query. GraphQL Ruby's default maximum depth is 13, this is just 7.

Compare this to rate limiting a REST endpoint, which generally have comparable response times. In this case all you need is a bucketed rate limiter that prevents a user exceeding, say, 200 requests per minute across all endpoints. If you **do** have slower endpoints (say, a CSV report or PDF generator) you can define more aggressive rate limits for these. With some HTTP middleware this is pretty trivial:

```

Rack::Attack.throttle('API v1', limit: 200, period: 60) do |req|
  if req.path =~ '/api/v1/'
    req.env['rack.session']['session_id']
  end
end

```

Query parsing

Before a query is executed, it is first parsed. We once received a pen-test report evidencing that its possible to craft an invalid query string that OOM'd the server. For example:

```
query {  
  __typename @a @b @c @d @e ... # imagine 1k+ more of these  
}
```

This is a **syntactically** valid query, but invalid for our schema. A spec compliant server will parse this and start building an errors response containing thousands of errors which we found consumed **2,000x** more memory than the query string itself. Because of this memory amplification, its not enough to just limit the payload size, as you will have valid queries that are larger than the the smallest dangerous malicious query.

If your server exposes a concept of maximum number of errors to accrue before abandoning parsing, [this can be mitigated](#). If not, you'll have to roll your own solution. There is no REST equivalent to this attack of this severity.

Performance

When it comes to performance in GraphQL people often talk about it's incompatibility with HTTP caching. For me personally, this has not been an issue. For SaaS applications, data is usually highly user specific and serving stale data is unacceptable, so I have not found myself missing response caches (or the cache invalidation bugs they cause...).

The major performance problems I **did** find myself dealing with were...

Data fetching and the N+1 problem

I think this issue is pretty widely understood nowadays. TLDR: if a field resolver hits an external data source such as a DB or HTTP API, and it is nested in a list containing N items, it will do those calls N times.

This is not a unique problem to GraphQL, and actually the strict GraphQL resolution algorithm has allowed most libraries to share a common solution: the [Dataloader pattern](#). Unique to GraphQL though is the fact that since it is a query language, this can **become** a problem with no backend changes when a client modifies a query. As a result, I found you end up having to defensively introduce the Dataloader abstraction everywhere *just in case* a client ends up fetching a field in a list context in the future. This is a lot of boilerplate to write and maintain.

Meanwhile, in REST, we can generally hoist nested N+1 queries up to the controller, which I think is a pattern much easier to wrap your head around:

```
class BlogsController < ApplicationController
  def index
    @latest_blogs = Blog.limit(25).includes(:author, :tags)
    render json: BlogSerializer.render(@latest_blogs)
  end

  def show
    # No prefetching necessary here since N=1
    @blog = Blog.find(params[:id])
    render json: BlogSerializer.render(@blog)
  end
end
```

Authorisation and the N+1 problem

But wait, there's more N+1s! If you followed the advice earlier of integrating with your library's authorisation framework, you've now got a whole new category of N+1 problems to deal with. Lets continue with our X API example from earlier:

```
class UserType < GraphQL::BaseObject
  field :handle, String
  field :birthday, authorize_with: :view_pii
end

class UserPolicy < ApplicationPolicy
  def view_pii?
    # Oh no, I hit the DB to fetch the user's friends
    user.friends_with?(record)
  end
end
```

```
query {
  me {
    friends { # returns N Users
      handle
      birthday # runs UserPolicy#view_pii? N times
    }
  }
}
```

This is actually trickier to deal with than our previous example, because authorisation code is not always run in a GraphQL context. It may for example be run in a background job or an HTML endpoint. That means we can't just reach for a DataLoader naively, because DataLoaders expect to be run from within GraphQL (in the Ruby implementation anyway).

In my experience, this is actually **the biggest source of performance issues**. We would regularly find that our queries were spending more time authorising data than anything else. Again, this problem simply does not exist in the REST world.

I have mitigated this using nasty things like [request level globals](#) to memoise data across policy calls, but its never felt great.

Coupling

In my experience, in a mature GraphQL codebase, your business logic is forced into the [transport layer](#). This happens through a number of mechanisms, some of which we've already talked about:

- Solving data authorisation leads to peppering authorisation rules throughout your GraphQL types
- Solving mutation / argument authorisation leads to peppering authorisation rules throughout your GraphQL arguments
- Solving resolver data fetching N+1s leads to moving this logic into GraphQL specific dataloaders
- Leveraging the (lovely) [Relay Connection](#) pattern leads to moving data fetching logic into GraphQL specific [custom connection objects](#)

The net effect of all of this is to meaningfully test your application you **must** extensively test at the integration layer, i.e. by running GraphQL queries. I have found this makes for a painful experience. Any errors encountered are captured by the framework, leading to the fun task of reading stack traces in JSON GraphQL error responses. Since so much around authorisation and DataLoaders happens inside the framework, debugging is often much harder as the breakpoint you want is not in application code.

And of course, again, since its a query language you're going to be writing a lot more tests to confirm that all those argument and field level behaviours we mentioned are working correctly.

Complexity

Taken in aggregate, the various mitigations to security and performance issues we've gone through add **significant** complexity to a codebase. It's not that REST does not have these problems (though it certainly has fewer), it's just that the REST solutions are generally much simpler for a backend developer to implement and understand.

And more...

So those are the major reasons I am, for the most part, over GraphQL. I have a few more peeves, but to keep this article growing further I'll summarise them here..

- GraphQL discourages breaking changes and provides no tools to deal with them. This adds needless complexity for those who control all their clients, who will have to [find workarounds](#).
- Reliance on HTTP response codes turns up everywhere in tooling, so dealing with the fact that 200 can mean everything from everything is Ok through to everything is down can be quite annoying.
- Fetching all your data in one query in the HTTP 2+ age is often not beneficial to response time, in fact it will worsen it if your server is not parallelised, vs sending separate requests to separate servers to process in parallel.

Alternatives

Ok, end of the rant. What would I recommend instead? To be up front, I am definitely early in the [hype cycle](#) here, but right now my view is that if you:

1. Control all your clients
2. Have ≤ 3 clients
3. Have a client written in a statically typed language
4. Are using >1 language across the server and clients²

You are probably better off exposing an **OpenAPI 3.0+** compliant JSON REST API. If, as in my experience, the main thing your frontend devs like about GraphQL is its self documenting type safe nature, I think this will work well for you. Tooling in this area has improved a lot since GraphQL came on the scene; there are many options for generating typed client code even down to [framework specific data fetching libraries](#). My experience so far is pretty close to "the best parts of what I used GraphQL for, without the complexity Facebook needed".

As with GraphQL there's a couple of implementation approach...

Implementation first tooling generates OpenAPI specs from a typed / type hinted server. [FastAPI](#) in Python and [tsoa](#) in TypeScript are good examples of this approach³.

This is the approach I have the most experience with, and I think it works well.

Specification first is equivalent to "schema first" in GraphQL. Spec first tooling generates code from a hand written spec. I can't say I've ever looked at an OpenAPI YAML file and thought "I would love to have written that myself", but the recent release of [TypeSpec](#) changes things entirely. With it could come a quite elegant schema first workflow:

1. Write a succinct human readable TypeSpec schema
2. Generate an OpenAPI YAML spec from it
3. Generate statically typed API client for your frontend language of choice (e.g. [TypeScript](#))
4. Generate statically typed server handlers for your backend language & server framework (e.g. [TypeScript + Express](#), [Python + FastAPI](#), [Go + Echo](#))
5. Write an implementation for that handler that compiles, safe in the knowledge that it will be type safe

This approach is less mature but I think has a lot of promise.

To me, it seems like powerful **and** simpler options are here, and I'm excited to learn their drawbacks next 😊.

Thanks for reading! See [Hacker News](#) and [Reddit](#) for more discussion on this article.

1. Persisted queries are also a mitigation for this and many attacks, but if you actually want to expose a customer facing GraphQL API, persisted queries are not an option. ↩
2. Otherwise a language specific solution like [tRPC](#) might be a better fit. ↩
3. In Ruby, I guess because type hints are not popular, there is no equivalent approach. Instead we have [rswag](#) which generates OpenAPI specs from request specs. It would be cool if we could build an OpenAPI spec from Sorbet / RBS typed endpoints! ↩

39 Comments

 Login ▼

G

Join the discussion...

LOG IN WITH

OR SIGN UP WITH DISQUS 

Name

♡ 7

Share

Best Newest Oldest

Subscribe


Privacy

Do Not Sell My Data

Bessey's Blog

Bessey's Blog
bessey@gmail.com

[Subscribe via RSS](#)
[View Archives](#)

 [bessey](#)
 [mjhbessey](#)

Technical blog of a full stack software engineer, working on all things web since 2013