Embracing the power of GraphQL

Pierre Carrier (pierre@meteor.com), MDG (meteor.io)

GraphQL Toronto, 2017-07-10

GraphQL @ MDG

- Apollo clients (web, iOS, Android)
- Apollo Developer Tools
- graphql-tools, graphql-server
- Launchpad
- Optics
- Articles & (better) talks

Hiring in Toronto

pierre@meteor.com

My background

Tech support @ Red Hat

SRE @ Spotify, Airbnb

Software engineer @ Twitter (Core Storage)

Core Engineer @ MDG (Apollo)

I (candidly) speak for myself!

My GraphQL story

First introduced when interviewing for MDG (Oct 2015).

Took months to feel truly comfortable.

Part of the team that built Optics in 2016, more to come.

Current understanding from March 2017. It helped to work full time on tooling.

GraphQL is a spec, not code

1. Overview
2. Language
3. Type System
4. Introspection
5. Validation
6. Execution
7. Response

A. Appendix: Notation Conventions

B. Appendix: Grammar Summary

Confusing terminology

A request refers to:

- A query document
  - Named fragments
  - Either:
    - One or more named operation(s): `query`, `mutation`, `subscription`, etc.
    - One anonymous operation (`query` if type unspecified)
- Optionally, an operation name
- Optionally, variable values (`Map<String, Any>`)

Query language

- Describe what you want, get it back.
- Independent of transport.
- Independent of encoding (needs JSON/+, ordering).
- No inherent overhead:
  - No need to transmit the query document on the wire.
  - Parsing and validation can trivially be cached.
  - Execution **could** be compiled (AOT or JIT).

Schemas are great

- SQL schemas are about the structure of the data.
- GraphQL schemas are both data model and API.
- Introspection is baked in every schema per specification.
  Tools can offer static analysis, autocompletion, etc. as mere clients.
- Use the IDL as a living document. Treat it like a contract and amend it as such.

Schema first

- Model features into API.
- Open a PR that changes the IDL. Backend CI goes red.
- In parallel:
  - Mock and build frontend
  - Iterate on backend until CI goes green.
- Stage. Ship. Rinse and repeat.

Backwards compatibility is hard

- Very few operations are backwards compatible:
  - Add new types
  - Add fields to existing types
  - Add nullable parameters
  - Wrap field types in `NonNull`
  - *(Arguably)* Add implementations on interfaces and types in union
- Use Optics? :)

Great tooling

- Given an endpoint, we know the schema.
- Given a query and the schema, we know the exact shape of the response.
- GraphiQL was The Killer App. IDE support is emerging and unequal.
- https://apis.guru/graphql-voyager/ is fun.
- Code generation increasingly accessible. Great for clients. Less clear to me for servers.
- GraphQL doesn't require a client library, but they help. A lot.

A few common topics

- Authentication
  - We don't think it should be GraphQL-specific.
  - Many modern authentication systems involve HTTP flows (eg OAuth).
- Authorization
  - Cannot think of endpoints anymore. It's about **edge traversal**.
  - Thanks to explicit mutation APIs and field accesses, Role-Based Access Control is a breeze.
- Real-time updates
  - I don't think we have the right solution yet. Subscriptions are usually not the right answer.
  - Luckily much of GraphQL is not tied to request/reply.

Custom scalars are needed

- Few standard scalar types (`Int`, `Float`, `String`, `Boolean`, `ID`).
  Community could use more uniformity and/or declarative encoding for scalars, eg:
  - `Nothing`;
  - Timestamps;
  - Blobs;
  - Arbitrary JSON values(?).
- Weak numerical tower (no specific “signedness” nor precision, no range constraints, etc.).

The type system is limited

- No generics. We get `List<User>`, but cannot build `PaginatedList<User>`.
- No identity for object types (nor standard mechanism to lookup objects directly).
- Flat namespace.
- Parameters are hard to specify (neither abstract types nor constraints between parameters).
- No error taxonomy (eg transient vs permanent, `403` vs `404` vs `500`), no contract on error sources.

Misc. limitations

- Selection sets cannot be empty.
- Execution is a pure “top-bottom” traversal, no dependencies between subtrees.
- Cannot **generally** mix query and mutation operations.
- Cannot explode arrays. If we can only look up one user at a time (eg `user(id:ID!):User`) and want to read N users at once, we must use multiple queries or a “dynamic” query, eg:

```
query ReadMultipleLastNames {
  u1: user(id:"pcarrier") { lastName }
  u2: user(id:"xav_cz")   { lastName }
}
```

Mutations: bolted on?

- Only the top-level selection set of a mutation is executed serially, which encourages a flat mutation space. Clients allowing, expose a hierarchy of mutations, and handle constraints on execution order yourself.
- The only maintainable way to keep some isolation between “mutation types” and “query types” is to:
  - Discriminate between them (through naming or directives),
  - Have fields of mutation types be either, but fields of “query types” never be “mutation” types.

I WISH WE COULD

```
mutation AttendMyOwnTalks {
  query { ← Mutations and queries cannot be mixed-and-matched out-of-the-box
    me { id @as($myId) }
    meetup(id: "graphql-toronto") {
      talks(author: $myId) @as($events) { id title startTime endTime }
                    ↑ Cannot use bits of the response as parameters
    }
  }
  #each($event:$events) { ← Cannot explode arrays
    rsvp(coming: YES, eventId: $event{id}) ← Must have non-empty selection set if an object
    ↓ No namespacing
    calendar.google.com/myCalendar {
      upsertEvent(name: $event{title}, ← Mutations are intended to be "flat"
                  from: $event{startTime},
                  to:   $event{endTime})
    }
  }
}
```

Q&A