Intro to GraphQL

Agenda

- Origins of GraphQL
- What is GraphQL?
- Why use GraphQL?
- Querying GraphQL
- Schema Definition Language (SDL)
- GraphQL Resolvers
- Build a GraphQL server to serve a schema
- **Bonus**: Schema stitching

Tyler

Principal Software Engineer @ hyper

GraphQL

Origins

- The shift to mobile
- The Newsfeed & nested, recursive data
- Sequential REST calls (*lots of endpoints*)
- Implicit contract
- Overfetching data

Rest Bonanza

```
GET https://foo.com/api/users/1
// grab post ids and query for each post
GET https://foo.com/api/posts/[id]

OR GET https://foo.com/api/users/1/posts

OR GET https://foo.com/api/users/1?include=posts

OR GET https://foo.com/api/users-posts/1

// Get Friends via id?
// GET as users?
GET https://foo.com/api/users/[id]
// resource on user?
OR GET https://foo.com/api/users/1/friends/[id]

// versions?
GET https://foo.com/api/v1/users/[id]
```

Idea

- One Endpoint
- Server describes all capabilities
- Client describes requirements to the endpoint
- Endpoint fulfills requirements

GraphQL

Server Capabilities: `Schema Definition Language`

Client Requirements: `Graph Query Language`

Fulfillment: `GraphQL Runtime`

This Idea Caught On

Gatsby, Hasura, Apollo, Contentful, Github, Fauna, Relay, OneGraph, Braintree, Arweave, so many more...

The Schema

> The Server Capabilities

Model your business domain as a schema

- Define types and how they relate to each other
  - Think OOP (types reference other types)
  - Fields on types can ref another `Type` or a `Scalar`
- Define *entrypoints* to those business types
  - Just special reserved types!
    - `Query`
    - `Mutation`
    - `Subscription`

```
type Author {
  id: ID!
  name: String!
  posts (criteria: AuthorPostsCriteriaInput): [Post!]!
}

type Post {
  id: ID!
  content: String!
  author: Author!
  relatedPosts (criteria: RelatedPostsCriteriaInput): [Post!]!
  likes: [Author!]!
  createdAt: Date!
}

type Query {
  posts (criteria: PostsCriteria!): [Post!]!
}

type Mutation {
  addPost (addPostInput: AddPostInput!): Post!
}
```

The Query/Mutation

> The Client Requirements

Write queries according to the schema's entrypoints

- Only get what you need
- In the shape that you need it
- Built in validation (the schema is strongly typed!)
- They're just strings! (syntax resembles `JSON`)

```graphql
query GetPostAndRelatedPostsById {
  post (criteria: {
    id: "post-1"
  }) {
    content
    author {
      name
    }
    relatedPosts (criteria: {
      after: "2021-09-10T00:00:00.000Z"
    }) {
      content
      author {
        name
      }
      likes {
        id
      }
    }
    createdAt
  }
}
```

Graph Query Language

aka `GraphQL`

- Fields
- Arguments
- Aliases
- Fragments
- Operation Names
- Variables
- Directives

Let's write some queries!

`https://countries.trevorblades.com/`

Share a query with the class!

`https://github.com/TillaTheHun0/intro-to-graphql-talk/issues/1`

Pain Pain, Go Away

- Shape of the result depends **entirely** on client's query
  - Add new fields on server, without breaking clients!
- **No more** *client* over fetching!
- **No more** multiple REST calls!
- **No more** "mega" REST endpoints
- **No more** "custom" filtering api. The runtime filters fields for you.
- **No more** client side data transformation!
  - just query the shape that you need
  - code to the interface (the schema)

Break

Schema Definition Language (SDL)

- *Very* similar to the GraphQL Query Language
- Language Agnostic (any server in any language can define and resolve it)

- Type
- Field
- Enum
- Scalar
  - Custom
- !
- []
- Input

Let's build a server!

Pokemon!

Disclaimer

This is just one way to build a GraphQL Server

Setup

NodeJS@v16

(https://nodejs.org/en/)

--

Hyper Cloud Application `https://dashboard.hyper.io`

- Sign-in with Github
- Create an app

Setup

`https://github.com/TillaTheHun0/intro-to-graphql-talk`

--

Create `.env` file with `HYPER` set to your connection string

```
npm i
npm run setup
npm run list
```

Let's mount graphql on express

Use Cases

- Fetch a list of pokemon
- Fetch whether they are a starter pokemon

GraphQL Resolver

A function that tells GraphQL how to fulfill a field

```
const myResolver = (parent, args, context, info) => {
  ...
}
```

```
const typeDefs = gql`
  type MyType {
    id: ID!
    someBoolean: Boolean!
    someArrOfInts: [Int!]!
  }
`


const resolvers = {
  MyType: {
    id: (parent, args, context, info) => 'some-id',
    someBoolean: (parent, args, context, info) => true,
    someArrOfInts: async (parent, args, context, info) =>
      Promise.resolve([1, 2, 3])
  }
}
```

Resolver Signature

```
const myResolver = (parent, args, context, info) => {
  ...
}
```

- `parent` : what was returned from the **previous** type's resolvers
- `args` : arguments provided directly to this field in the query
- `context` : object that is passed to *every* GraphQL resolver
- `info` : field that contains metadata about the query

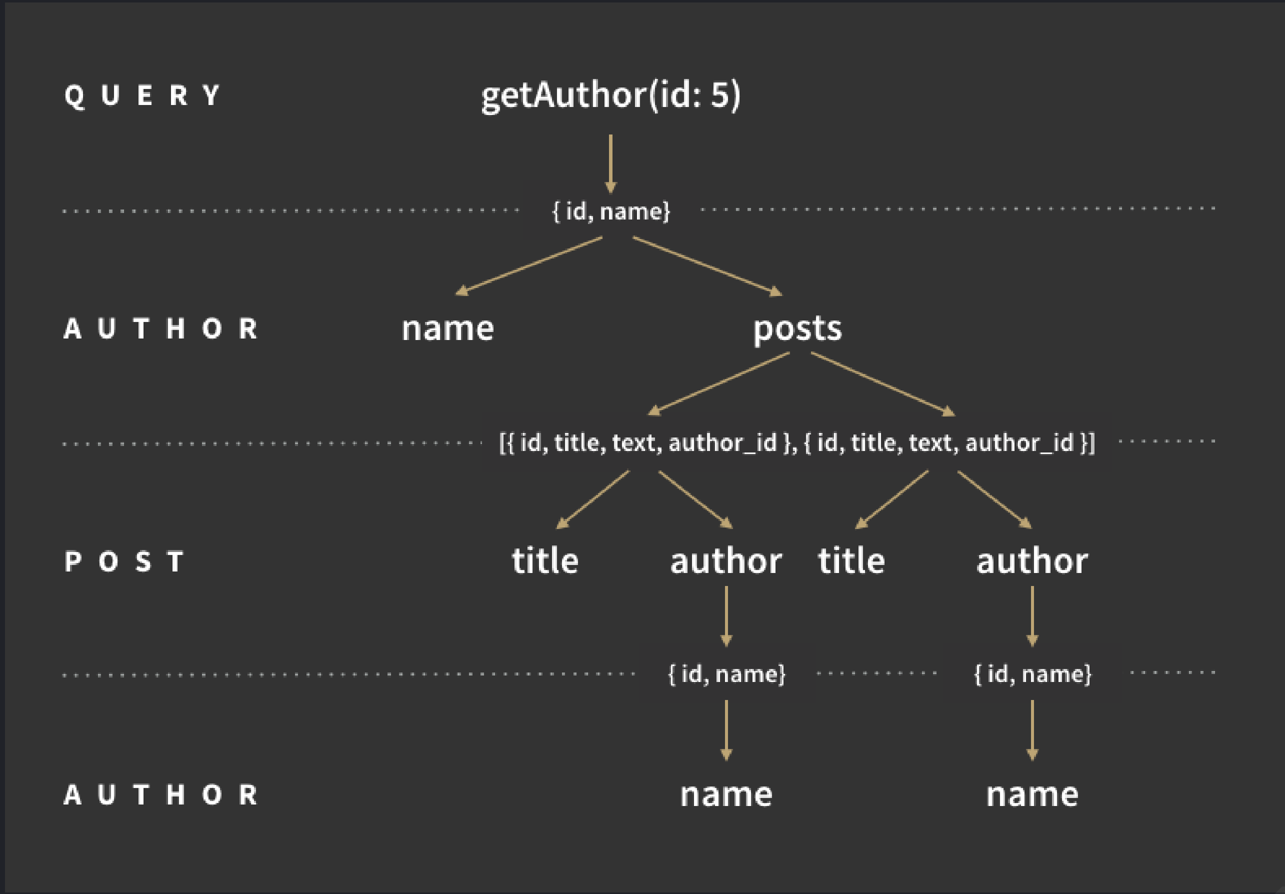
Tyler's Opinion: you won't need `info` most of the time. It's for advanced use cases (perhaps we will get to some)

Resolver Rules

- Resolvers are executed "breadth-firstly"
    - Siblings are executed in parallel
    - A resolver on a child type is excuted only after it's parent type fully resolves
- If an **object** is returned, then execution continues
- If a **scalar** is returned, execution completes

Query Resolution

```
query {
  getAuthor(id: 5){
    name
    posts {
      title
      author {
        name
      }
    }
  }
}
```



```
query {
  getAuthor(id: 5){
    name
    posts {
      title
      author {
        name
      }
    }
  }
}
```

1. run Query.getAuthor
2. run Author.name and Author.posts (for Author returned in 1)
3. run Post.title and Post.author (for each Post returned in 2)
4. run Author.name (for each Author returned in 3)

GraphQL resolves a query until it has received a scalar value for each field in the query

Resolver Context

The 3rd argument passed to each resolver

Mutable (do not recommend mutating)

Can be built on each query received

Resolver Context

Tyler's Opinion: Use `context` for dependency injection!

Trivial Resolvers

```
const resolvers = {
  id: (parent) => parent.id,
  name: (parent) => parent.name
}
```

Most GraphQL runtime implementations provide you these out of the box, including `graphql.js`

Use Case

- fetch whether a pokemon is a `favorite`
- filter pokemon based on `favorite`

What do we have so far

- Input Validation
- Filtering
- Output Validation
- *Documentation*
- Explicit Contract

Use Case

- Fetch the moves each pokemon can learn
  - move name
  - elemental type

Problem

Overfetching on the **server**

Solution

Only load the data when it is asked for!

How? With a separate resolver!

What did we gain?

- Merging data from two data sources
- Only loading from data source when client asks for it
- Automatic field mapping (DTO)
    - Our schema describes the shape and graphql runtime fulfills it
- Separation of concerns

Use Case

- Fetch the Pokemon that can learn each move

Pokemon can be resolved from two different places

- `Query.pokemon`
- `Moves.pokemon`

Furthermore, Pokemon can come from different data sources:

- `Query.pokemon` from hyper data (`metaClient`)
- `Moves.pokemon` from PokeApi (`pokeClient`)

`favorite` only comes from hyper. Oh no!

`name` is capitalized in hyper data, but not in PokeApi 😞

Solution

Resolver for the `favorite` field!

Resolver for the `name` field!

Problem

Serving data via a hierarchical can cause issues:

- Loading the same Pokemon multiple times
- Loading the same Move multiple times

N+1 Problem

Solutions?

- Load data in the parent that the child will need?
- Don't let data be cyclical on the graph?

💩 (Both of these take away benefits of graphql!)

dataloader

Automatic batching, deduping, and caching from datasources

Tyler's Opinion:

- Parent should always provide the primary identifier of the type being resolved.
- each field level resolver on that type loads its own data.
- Let `dataloader` dedupe, batch, and cache requests to data sources
- TL;DR: **use dataloader**

Break

Mutations

- All mutations should be top level
- Unike `Query` GraphQL executes mutation siblings sequentially
- Allows sending atomic operations to the server that contains multiple mutations

Use Case

- Add a new favorite Pokemon

Bonus

Schema Stitching

FIN