GRANT YOUR SPA THE POWER OF GRAPHQL

CHRIS NORING

Google Developer Expert, Cloud Developer Advocate
Microsoft
@chris_noring

Why GraphQL

- AVOIDS multiple roundtrips to fetch hierarchical or related data

- AVOIDS multiple roundtrips to fetch hierarchical or related data
- FASTER development, no need to wait for backend for a new endpoint

- AVOIDS multiple roundtrips to fetch hierarchical or related data
- FASTER development, no need to wait for backend for a new endpoint
- UNIFYING way to ask for data from different endpoints

- AVOIDS multiple roundtrips to fetch hierarchical or related data
- FASTER development, no need to wait for backend for a new endpoint
- UNIFYING way to ask for data from different endpoints
- CUSTOMIZE OUTPUT in the query itself, no need for serializing afterwards

CORE CONCEPTS

- QUERIES, questions

- QUERIES, questions
- SCHEMA, what entities your database consist of

- QUERIES, questions
- SCHEMA, what entities your database consist of
- RESOLVERS, functions that responds to queries

- QUERIES, questions
- SCHEMA, what entities your database consist of
- RESOLVERS, functions that responds to queries
- MUTATIONS, changes data, CREATE, UPDATE, DELETE

QUERIES

A simple query

```
{
  resource: {
    column,
    column2
  }
}
```

results:

```
{
  data: {
    resource: {
      column,
      column2
    }
  }
}
```

TRY IT OUT

TESTING SANDBOXED ENVIRONMENTS

GraphiQL: GraphQL API Explorer

GraphiQL

Prettify

Docs

```
1  {
2    hello
3  }
4
```

```
{
  "data": {
    "hello": "Hello world!"
  }
}
```

QUERY VARIABLES

https://www.graphqlhub.com/

- Hacker News
- Reddit
- GitHub
- Twitter

and more...

GraphQLHub ▶ Prettify

```
{
  graphQLHub
  twitter {
    user(identifier: name, identity: "chris_noring") {
      created_at
      description
      id
      screen_name
      name
      profile_image_url
      url
      tweets_count
      followers_count
      tweets(limit: 2) {
        text
      }
    }
  }
}
```


```
{
  "data": {
    "graphQLHub": "Use GraphQLHub to explore popular APIs with GraphQL! Created by
    "twitter": {
      "user": {
        "created_at": "Tue Jun 19 10:09:09 +0000 2012",
        "description": "Developer 🥑@ Microsoft  GDE Angular1/2 @google, Dev Exp (
        "id": "612518604",
        "screen_name": "chris_noring",
        "name": "christoffer noring",
        "profile_image_url": "http://pbs.twimg.com/profile_images/971392167859163
        "url": "https://t.co/WxiyvrrAyr",
        "tweets_count": 3749,
        "followers_count": 2527,
        "tweets": [
          {
            "text": "@simona_cotin @devstefops @OWASPLondon @MSFTReactor Great, so
          },
          {
            "text": "RT @sherrrylst: If you are based in Copenhagen and interested
          }
        ]
      }
    }
  }
}
```

A typical Schema:

```
User( id, name, age, address, orders)
Order( id, created, items)
OrderItem( id, product, quantity, price)
Product( id, name)
```

How deep can we query this?

Query:

```
Users {
  name,
  orders {
    created,
    items {
      quantity,
      price,
      product {
        name
      }
    }
  }
}
```

Pretty cool right?

```
data: {
  users: [{
    'John',
    orders: [{
      '2017-01-01',
      items: [{
        quantity: 3,
        price: 110,
        product: { name: 'DVD' }
      }]
    }]
  }]
}
```

QUERY WITH PARAMETER

```
Users( email: 'chris@chris.com') {

}
```

Corresponds to:

```
SELECT ...
FROM resource
WHERE email='chris@chris.com'
```

DEFINE A SCHEMA

CREATING A GRAPHQL SERVER

There are many ways to do this

The raw approach using `graphql` lib

```
import {
  graphql,
  GraphQLSchema,
  GraphQLObjectType,
  GraphQLString
} from 'graphql';
let schema = new GraphQLSchema({
  query: new GraphQLObjectType({
    name: 'RootQueryType',
    fields: {
      hello: {
        type: GraphQLString,
        resolve () { return 'world'; }
      }
    }
```

```
let query = '{ hello }';
graphql(schema, query).then(result => {
  console.log(result);
});
```

result:

```
{ data: { 'hello': 'world' } }
```

In short

- query->fields->myQuery, what queries are available
- resolve(), says how we respond to said query

Right now our 'hello' is of type 'string'. What about more complex types?

We can create our own type with 'GraphQLObjectType'

```
let humanType = new GraphQLObjectType({
  name: 'Human',
  fields: () => ({
    id: { type: GraphQLString },
    description: { type: GraphQLString },
    name: { type: GraphQLString }
  })
})
```

Creating a resolver function, matching the schema

```
return Promise.resolve(
  {
    id: '1',
    description: 'a description',
    name: 'john'
  }
)
```

back to our schema

```
human: {
  type: humanType,
  resolve: () => getHuman()
}
```

We can create our own type with 'GraphQLList'

```
humans: {
  type: GraphQLList(humanType),
  resolve() {
    return getHumans();
  }
}
```

Our resolver function

```
const getHumans = () => {
  return Promise.all([{
    id: 1, name: 'chris'
  }]);
}
```

What about parameters in our queries?

- Add args to our resolve function
- Dig out the args and query properly

The second argument in our resolve function contains our `args`

```
getHuman: {
  type: humanType,
  resolve: (root, { id }) => getHumanById(id)
}
```

And now query like so

```
getHuman(id: 1) { name }
```

What about resolving deep queries? e.g User->Orders->OrderItem etc..

- We add a field for deeper/nested data
- We need to create a resolver for the field itself, on the type

Let's start with the data structure

```
const humans = [{
  id: 1, name: 'chris', friends: [1,2,3]
}]
```

We need to resolve 1,2,3

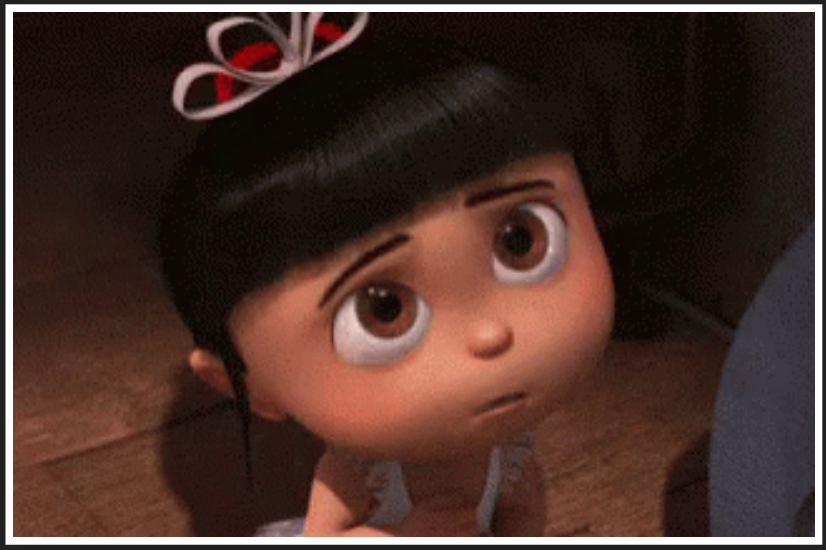
Now the schema, let's add that to humanType

```
const humanType = GraphQLObjectType({
  name: 'Human',
  fields: () => ({
    id: { type: GraphQLString },
    description: { type: GraphQLString }:,
    name: { type: GraphQLString }
    friends: {
      type: GraphQLList(humanType),
      resolve: human => getFriends(human)
    }
  })
})
```

Lookup each human by `id`

```
getFriends = (character) => {
  return character.friends.map(f => getHuman(f))
}
```

Isn't there a better way?



YES, there is

You can be using **buildSchema()** method from the **graphql** lib

It uses something called *GraphQL Schema Definition Language*

```
var schema = buildSchema(`
type Product {
  name: String,
  id: Int
},
type Query {
  hello: String,
  products: [Product],
  rollDice(numDice: Int!, numSides: Int): [Int]
}
          `);
```

A lot easier to read right?

- **type Query**, to create query (Query reserved)

- **type Query**, to create query (Query reserved)
- **type "Your type"**, for creating a custom type

- **type Query**, to create query (Query reserved)
- **type "Your type"**, for creating a custom type
- **hello: String**, to create a field of type primitive

- **type Query**, to create query (Query reserved)
- **type "Your type"**, for creating a custom type
- **hello: String**, to create a field of type primitive
- **products: [Product]**, to create a list type

- **type Query**, to create query (Query reserved)
- **type "Your type"**, for creating a custom type
- **hello: String**, to create a field of type primitive
- **products: [Product]**, to create a list type
- **rollDice(numDice: Int!, numSides: Int): [Int]**, creates parameterized query definition

- **type Query**, to create query (Query reserved)
- **type "Your type"**, for creating a custom type
- **hello: String**, to create a field of type primitive
- **products: [Product]**, to create a list type
- **rollDice(numDice: Int!, numSides: Int): [Int]**, creates parameterized query definition
- Int!, mandatory parameter

- **type Query**, to create query (Query reserved)
- **type "Your type"**, for creating a custom type
- **hello: String**, to create a field of type primitive
- **products: [Product]**, to create a list type
- **rollDice(numDice: Int!, numSides: Int): [Int]**, creates parameterized query definition
- Int!, mandatory parameter
- **input MyType {}** only for Mutation inputs

MUTATIONS

CHANGES THE DATA

```
const mutationType = new GraphQLObjectType({
  name: 'DeletionExample',
  description: 'Will perform a deletion',
  fields: () => ({
    deleteArticle: {
      type: Article,
      description: 'Delete an article given an id',
      args: {
        id: { type: GraphQLNonNull(GraphQLInt) }
      },
      resolve: (value, { id }) => {
        return deleteArticle(id)
      }
    }
  })
```

Calling the mutation

```
mutation 'Example' {
  deleteArticle(id: 1) {
    status
  }
}
```

A complex mutator example, pass an object

Schema definition

```
const mutationType = new GraphQLObjectType({
  name: 'CreationExample',
  description: 'Will perform a creation',
  fields: () => ({
    createArticle: {
      type: Article,
      description: 'Create an article',
      args: {
        article: { type: Article }
      },
      resolve: (value, { article }) => {
        return createArticle(article)
      }
    }
  })
```

Calling the mutation

```
mutation 'Example' {
  createArticle(article: {
    name: 'tomato',
    description: 'it is red'
  }) {
    status
  }
}
```

Now to do this with **buildSchema**

```
var schema = buildSchema(`
  type Mutation {
    createProduct(name: String!, description: String!): String
  }
`);
```

and defining it

```
var root = {
  createProduct: args => {
    const { name, description } = args;
    const newProduct = addProduct(name, description);
    return `Created: ${newProduct.id} ${newProduct.name} - ${
      newProduct.description
    }`;
  }
};
```

Ok, much much simpler, why are you showing us the long version at all?

It's good to know what happens underneath :)

ADDITIONAL FEATURES

Alias

an Alias allows you to rename the field you are querying so the output is to your liking

`userchris` is our alias

```
github {
  userchris: users(userid:'softchris') {
    name, userid
  }
}
```

output:

```
{ userchris: { name: 'chris', userid: 'softchris' } }
```

Fragment, creating a reusable query piece

Great for:

- Keeping it DRY, a reusable code snippet that removes repitition
- Update once, Adding/removing fields to a query leads to updating in ONLY one place

all queried fields goes into our `fragment`

```
fragment UserDetail on GithubUser {
  company,
  avatar_url,
  repos {
    name,
    commits {
      message
    }
  }
}
```

```
query GitHub($user: String!) {
  github {
    user(username: $user) {
      ...UserDetail
    }
  }
}
```

Directives, the ability to include/exclude fields

`skip`, does not include the field

```
query SkipExample($var: Boolean) {
  otherField,
  someOtherField @skip(if: $var)
}
```

`include`, includes the field

```
query IncludeExample($var: Boolean) {
  otherField,
  someOtherField @include(if: $var)
}
```

CONSUMING A GRAPHQL SERVER

You don't actually need a client !

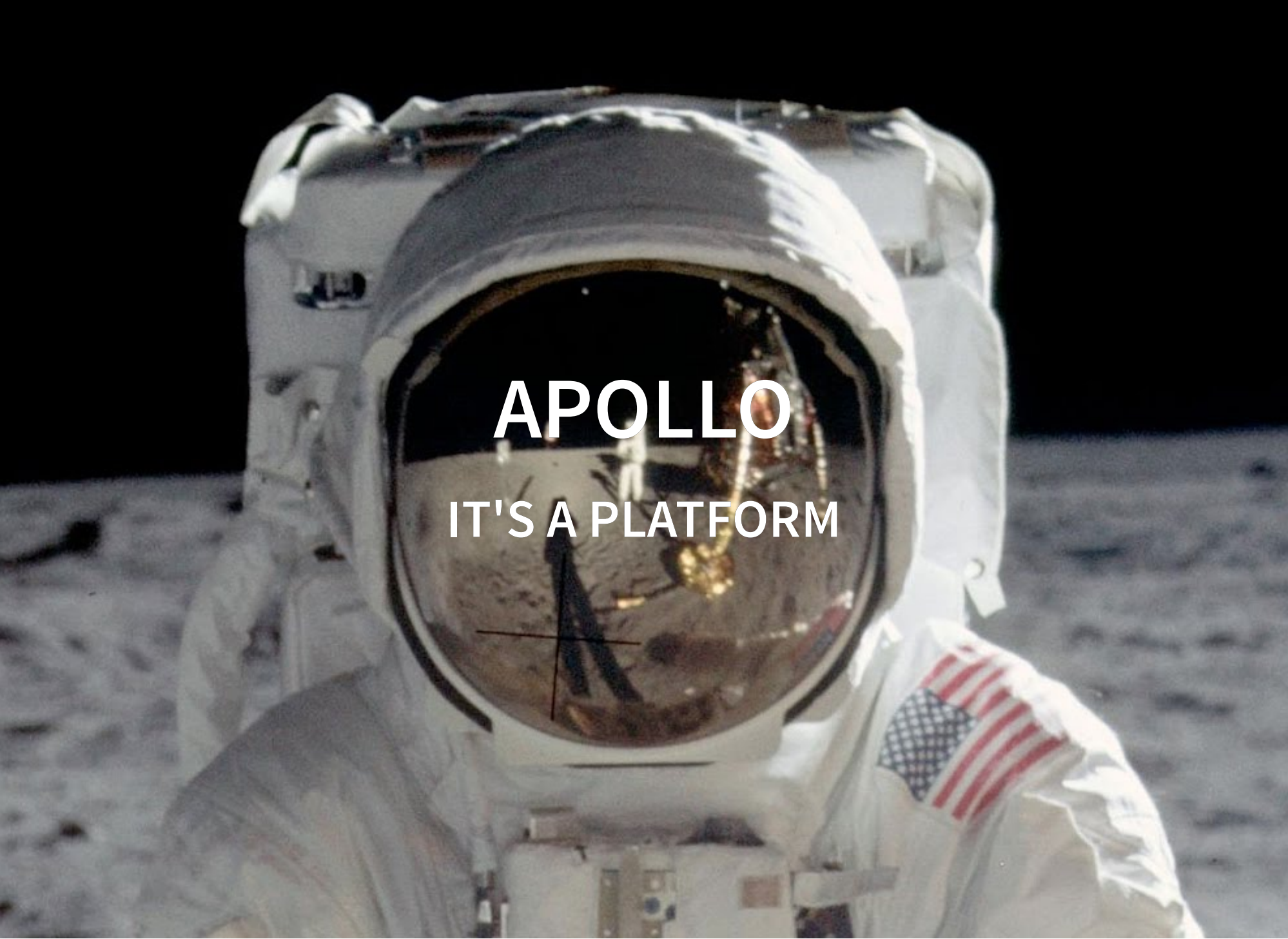
```
fetch(
  'http://localhost:4000/graphql?query={ humans { name }}',
  { headers: { 'content-type': 'application/graphql' } }
).then(response => // do something)
```

Using a client brings a lot of benefits though

- rich eco system of features and services
- caching
- binding
- consistency
- security
- connection with other data source

Two major options

- Apollo, https://www.apollographql.com/
- Relay, by Facebook, https://facebook.github.io/relay/



APOLLO

IT'S A PLATFORM

In numbers

- 300 000 downloads / week
- 15 000 devs in their slack channel
- GraphQL Summit, 800+ attendees

CORE

- Apollo server
- Apollo client, it works for all the major SPA frameworks
- iOS and Android clients
- Apollo CLI, for usage with cloud services

CLOUD SERVICES

- Schema registry,a registry for GraphQL schemas that acts as a central source of truth for a schema
- Client registry, a registry to track each known consumer of a schema, which can include both pre-registered and ad-hoc clients
- Operation registry, a registry of all the known operations against the schema
- Trace warehouse,

GATEWAY

A configuration of Apollo Server and additional plugins that functions as a GraphQL gateway. The gateway composes separately deployed “micro-schemas” that reference each other into a single master schema

- Apollo
  gateway,

WORKFLOW

Apollo implements some useful workflows for managing a GraphQL API.

- Schema change validation, tells you if schema changed, can check against type definitions, warns of incompatabilities
- Safe listing, Apollo provides an end-to-end mechanism for *safelisting* known clients and queries, a recommended best practice that limits production use of a GraphQL API to specific pre-arranged operations

Enough with the sales pitch, let's create something

Creating an Apollo server

- Define a schema
- Create resolvers
- Instantiate and launch our server

`gql` allows us to use GraphQL schema definition

```
const { gql } = require("apollo-server");

const typeDefs = gql`
type Query {
  users: [User],
  user(id: Int!): User
}

type User {
  id: ID!
  name: String
}`
```

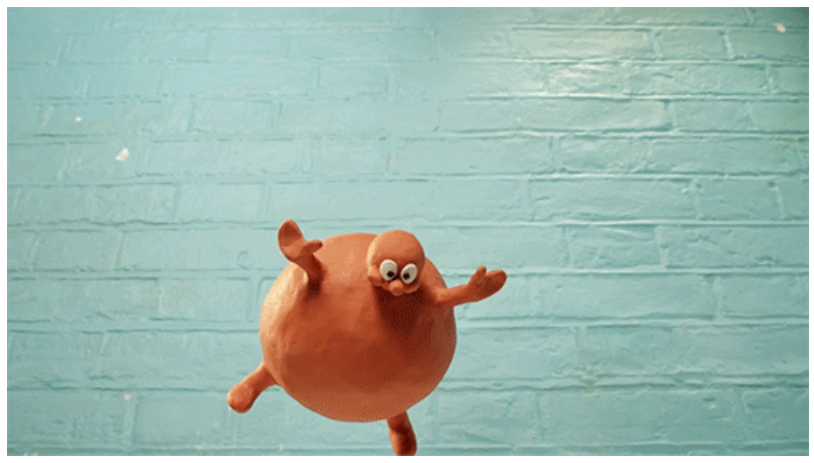
Define resolvers

```
const users = [{ id: 1, name: 'chris'}];
const getUsers = () => users;
const getUserById = (id) => users.find(u => u.id === id)

module.exports = {
  Query: {
    users: async () => getUsers(),
    user: async (_, { id }) => getUserById(id),
  }
};
```

instantiate and run your server

```
const server = new ApolloServer({ typeDefs, resolvers });

server.listen().then(({ url }) => {
  console.log(`🚀 Server ready at ${url}`);
});
```

Houston, we have a lift off



Playground

Q M GetLaunches

PRETTIFY HISTORY http://localhost:4000/ COPY

```
query GetLaunches {
  launches {
    site
  }
}

query GetLaunchById($id: ID!) {
  launch(id: $id) {
    id
    rocket {
      id
      type
    }
  }
}

mutation Test {
  bookTrips(launchIds: [1,2,3]) {
    success
  }
}
```

```
{
  "data": {
    "bookTrips": {
      "success": true
    }
  }
}
```

QUERY VARIABLES HTTP HEADERS

```
{ "id": 1 }
```

Apollo client

```
import ApolloClient from "apollo-boost";
import { ApolloProvider } from "react-apollo";

const localGraphQL = "http://localhost:4000/graphql";
const client = new ApolloClient({
  uri: localGraphQL
});

class App extends Component {
  render() {
    return (
      <ApolloProvider client={client}>
       // define app here
      </ApolloProvider>
    );
```

defining a component and define `Query` with `gql`

```
import React from "react";
import { Query } from "react-apollo";
import gql from "graphql-tag";

const Local = () => (
  <Query
    query={gql`
    // your query here
  `}
  >
  {({ loading, error, data }) => {
    if (loading) return <p>Loading...</p>;
    if (error) return <p>Error :(</p>;
    return data.products.map(({ name, id }) => (
      <div key={id}><p>{`${id}: ${name}`}</p></div>
```

How do we work with parameterized queries?

define a gql query that takes a parameter

```
const GET_PERSON = gql`
  query Person($id: Int!) {
    person(id: $id) {
      id
      name
    }
  }
`;
```

Set up `Query` component, populate `variables`

```
const PersonDetail = ({ id }) => (
  <Query query={GET_PERSON} variables={{ id }}>
    {({ loading, error, data }) => {
      if (loading) return null;
      if (error) return `Error!: ${error}`;

      return (
       <div>{{ data.person.name }}</div>
      );
    }}
  </Query>
);
```

- Use **Query** component
- Set **query** to your gql
- provide a function to **this.props.children**

Apollo puts all data in a built in cache

Data retrieval works like this:

- If data in cache, use cache
- If not in cache, do actual call

Sometimes you want to see the latest updates though.
For that we have three choices:

Sometimes you want to see the latest updates though.
For that we have three choices:

- Polling, for most cases

Sometimes you want to see the latest updates though.
For that we have three choices:

- Polling, for most cases
- Refetch, for when you want the latest data, now now

Sometimes you want to see the latest updates though.
For that we have three choices:

- Polling, for most cases
- Refetch, for when you want the latest data, now now
- Subscriptions, for when low-latency is super important or you have a large initial state

POLLING

- **pollInterval**, fetch at certain interval
- **startPolling**, start the polling when you want
- **stopPolling**, stop the polling when you want

```
<Query query={GET_DATA}, pollInterval={500}>
{(loading, error, data, startPolling, stopPolling)
  if (loading) return null;
  if (error) return `Error!: ${error}`;
  return (
    <div>{data.prop}</div>
  )
}}
</Query>
```

Refetch, when you actively want to fetch the latest data

```
<Query query={GET_DATA}>
  {(loading, error, data, refetch) => {
    if (loading) return null;
    if (error) return `Error!: ${error}`;
    return (
      <React.Fragment>
        <div>
          {data.prop}
          <button onClick={() => refetch()}>Fetch</button>
        </div>
      </React.Fragment>
    )
    }}
</Query>
```

What about mutations?

Write the gql

```
const ADD_PERSON = gql`
  mutation AddPerson($person: Person!) {
    addPerson(person: $person) {
      id
    }
  }
`;
```

Call `Mutation` component

```
<Mutation mutation={ADD_PERSON}>
    {(addPerson, { { data, loading, error } }) => (
      <div>
        <form onSubmit={e => {
          e.preventDefault();
          addPerson({ variables: { person: { name: input.value
          input.value = "";
        }} >
          <input ref={node => { input = node;}} />
          <button type="submit">Add Person</button>
        </form>
      </div>
    )}
  </Mutation>
);
```

- Use **Mutation** component
- Set **mutation** to your gql
- provide a function to **this.props.children**

SUMMARY

- https://graphql.org/learn/
- https://www.howtographql.com
- https://www.graphqlhub.com/
- https://www.apollographql.com
- aka.ms/graphql-blog-posts
- aka.ms/graphql-serverless

THANK YOU FOR LISTENING

@chris_noring