

Graph Querying Language

- Data is fetched and updated via a graph-like hierarchy of objects
- Graph is defined by a schema of:
 - Data types
 - Queries
 - Mutations
 - Subscriptions

An alternative to REST APIs

- Shift from HTTP-based concepts to RPC-style
 - Queries - read-only information from your graph
 - Mutations - any modification to data in your graph
 - Subscriptions - read-only queries, refreshed automatically from the server


Language Benefits

- Well-defined schema
- Useful query and analysis tools for developers
- Query/mutate/subscribe to what you need
- Open metadata querying API
- Can be queried even from simple web service calls or curl

A simple GraphQL Schema


```
type Query {  
  getQuizzes: [Quiz!]  
  getQuiz(id: Int!): Quiz!  
}
```

Queries provide data based
on GraphQL Types



```
type Quiz {  
  id: Int!  
  name: String!  
  description: String!  
}
```

Types are defined based
on primitives such as
Int, String, Boolean
and other types



Query with GraphQL

```
import {client} from './apollo-graphql-client';  
import gql from 'graphql-tag';
```

```
export function getQuizzesQuery() {
```

```
  return client.query({  
    fetchPolicy: 'no-cache',  
    query: gql`
```

```
    {
```

```
      getQuizzes {  
        name
```

```
      }  
    }`
```

```
  });
```

```
}
```

This is an Apollo client,
but most of the
differences are in setup,
not in query syntax

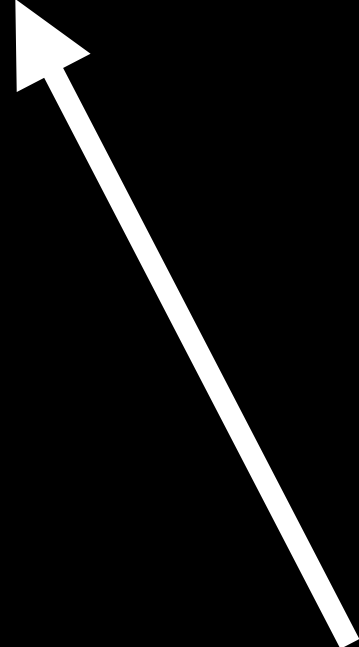
The meat of the call
is just GraphQL
query syntax

Mutations can Return Data

```
type Mutation {  
  voteOnCurrentQuestion(answer: String): ScoreData!  
}
```

*Mutations change your data,
and can also return results
like a query*

```
type ScoreData {  
  correct: Boolean!  
  points: Int  
}
```




Mutations with Complex Input

```
type Mutation {  
  login(credentials: SignInCredentials): String  
}
```

*Special **input** types
are only allowed to send
Parameter sets to a query
or mutation*

```
input SignInCredentials {  
  username: String!  
  password: String!  
}
```



Sample Mutation Call

```
await client.mutate({  
  mutation: gql`
```

```
    mutation answerQuestion($answer: QuestionAnswerInput!) {  
      answerQuestion (input: $answer)  
    }`,
```

```
  variables: {
```

```
    {
```

```
      answer: { key: incomingKey, quiz: incomingQuizId }  
    }
```

```
  }
```

```
});
```

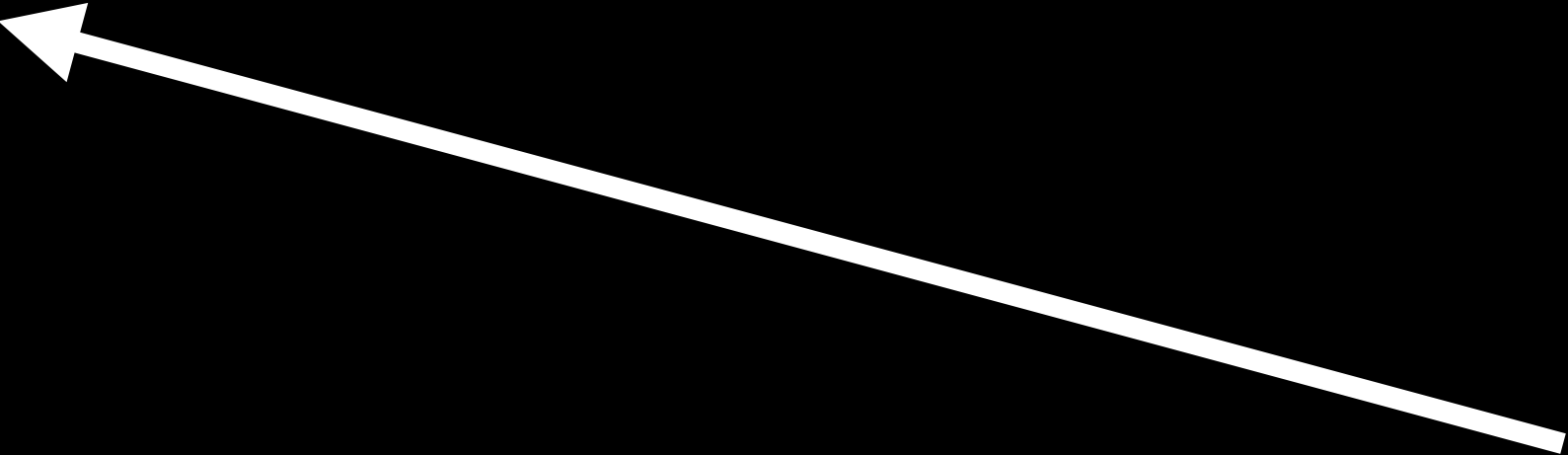
Query signature includes
param name and type



Actual schema mutation param
name and substitution variable



Map to variables incomingKey
and incomingQuizId from
JavaScript function params



GraphQL Subscriptions

GraphQL Subscriptions

- Act as server-side push messages
- Typically via WebSockets (but can be set up with polling in theory)
- Auto refresh themselves and notify the client when changed
- Typically require additional GraphQL endpoint and connection configuration
- Newest feature of GraphQL (2018) and hence support is all over the map

Defining Subscriptions

```
type Subscription {  
  nextMessage: GameplayMessage  
}
```

← Same semantics as
queries but they
are refreshed by
the server

```
type GameplayMessage {  
  gameOver: Boolean  
  leaderboard: [LeaderBoardEntry!]!  
}
```

(can have input parameters
and return responses)

```
type LeaderBoardEntry {  
  playerId: String!  
  nickName: String!  
  score: Int!  
}
```

Summary of Type Syntax Oddities

- **Thing** - A **Thing**. It is **not** required
- **Thing!** - A **Thing**. It is required
- **[Thing]** - An **array** of **things**. The array isn't required, nor are the individual entries (which is odd, but a GraphQL-ism)
- **[Thing!]** - An **array** of **things**. The array isn't required, but if supplied, each element must be a **Thing** and not **null**
- **[Thing!]!** - A **required array** of required **Things**

GraphQL Tools

Schema Mapping Tools

Code Generation - gql-gen

- <https://github.com/dotansimha/graphql-code-generator>
- Can generate TypeScript, Flow, Java, even integrate to client APIs such as Apollo
- Can generate a raw Schema AST
- JavaScript functions via *resolvers*
- Data mappings via servers like Prisma (think data remoting)

Popular GraphQL Distributions

- The official GraphQL reference implementation, **graphql-js**
- Apollo GraphQL
- Apollo GraphQL Yoga (<https://github.com/prisma/yoga2>)
- Facebook's Relay (React-based API for GraphQL) - <https://facebook.github.io/relay/>

Lots of other GraphQL APIs

- Clients and servers exist for
 - JavaScript
 - Java
 - Scala
 - Python
 - Ruby
 - C#/.NET
 - Go
 - Groovy
 - Erlang
 - Clojure
 - Elixir...

Gr

PRETTIFYHISTORY

http://localhost:3000/graphql

COPY CURL

1 query {
2 getQuizzes {
3 name
4 |
5 }
6 }
7

id
name
description
questions
Int!

▶

{
 "data": {
 "getQuizzes": [
 {
 "name": "First Quiz"
 }
]
 }
}

QUERY

HTTP HEADERS

TRACING

SCHEMA

GraphQL Voyager

React App

reactjs - How

Using the view

Using Media C

A Complete G

flex-direction

mini.css - Doc

GraphQL Voy

Web App Mar

localhost:3000/voyager

< Type List

Question

No Description

POSSIBLE TYPES

MultipleChoiceQuestion

No Description

TrueFalseQuestion

No Description

FillInBlankQuestion

No Description

quiz:questions

Question<<union>>

possible types

MultipleChoiceQuestion

TrueFalseQuestion

FillInBlankQuestion

MultipleChoiceQuestion

idInt!

textString!

options[ChoiceOption!]!

TrueFalseQuestion

idInt!

textString!

FillInBlankQuestion

idInt!

textString!

Query

☐ Sort by Alphabet

☒ Skip Relay

☒ Show leaf fields

Powered by GraphQL Voyager

+

RESET

-

GraphQL Implementations

- Relay - Facebook's JS RI
- GraphQL-js - a reference implementation
- Apollo - JavaScript clients and Servers, maintained by Prisma (our focus today)
- Lightweight express-GraphQL, HAPI-GraphQL projects
- Clients/Servers for many other languages / platforms including Java/Spring, Scala, Clojure, Ruby, Python, PHP, Go, .NET, Elixir, Haskell and more
- Good research on JavaScript clients at <https://medium.com/open-graphql/exploring-different-graphql-clients-d1bc69de305f>

You can also post / curl a query...

- Some query tools provide curl examples to the clipboard... (I didn't type this in!)

```
curl 'http://localhost:8080/graphql' -H 'Accept-Encoding: gzip, deflate, br' -H 'Content-Type: application/json' -H 'Accept: application/json' -H 'Connection: keep-alive' -H 'Origin: altair://-' -H 'Authorization: Bearer eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJrZW5ueSIsImF1dGgiOiJ0eSI6ImlJPTU5gR3jwdlP7yYk21uUN8Lab2xm-orRZPKoQ' --data-binary '{"query":"query {\n  \n  gameStatus {\n    gameMode\n    gameDescription\n    currentQuestion {\n      text\n    }\n    questionScore {\n      nickName\n      score\n    }\n    finalScore\n  }\n}","variables":{}}' --compressed
```

Batteries Not Included

- Things you'll need to wrestle with
 - Security (CORS or Proxy, authentication, authorization)
 - Client configuration for subscriptions / web sockets
 - Generating client code for schema objects
 - Monitoring (Apollo has a potential metrics engine available)
- You need to decide what level of granularity your API will support and you need to implement each query/mutation/subscription, or delegate it to an API for processing (like PRISMA)

Compared to JSON

GraphQL		REST/JSON	
Pros	Cons	Pros	Cons
<p>Query-based</p> <p>Fetch only what you need</p> <p>Create types from schema</p> <p>Subscription Model</p>	<p>APIs difficult for newbies</p> <p>Config is somewhat complex</p> <p>Security is a challenge and can be non-trivial</p> <p>Varied API quality</p>	<p>Ubiquitous</p> <p>Easily tested</p> <p>Native to SPA apps</p> <p>Agnostic to API tooling</p>	<p>No standard query API</p> <p>Metadata models also not a standard</p> <p>Subscriptions not native to REST</p>

Demonstrations

GraphQL Potential Uses

- Back-end of app with large semantic data structures
- Provide queryable API for customer with their choice of query tool and language
- App-defined ad-hoc or canned queries of content by graph of data
- Push-based messaging can be used on SPA clients

NOTE: GraphQL is not as big of a payoff if data model is trivial

Questions?

Don't miss this Philly Cloud Computing Event!

IoT on AWS

Wednesday, November 6th — 9AM to 5PM
Science History Museum, Philadelphia, PA

Cloud computing is a natural counterpart to smart devices: it's available anywhere, scalable to meet your needs, and generally more reliable than self-hosted hardware. Amazon Web Services provides a start-to-finish IoT solution: from gathering data, to storing it securely, to analyzing it and providing the results to your users.

In this one-day event we'll present a high-level overview of the steps involved in building an IoT data pipeline. In short sessions, our speakers will follow data from an IoT device as it is ingested, analyzed, secured, and used to make decisions. By the end of the day you will have a basic understanding of the complete pipeline.

Afternoon Hands-On Workshop:

There will be an optional afternoon workshop, limited to 50 participants, providing hands-on experience with connecting IoT devices and working with the data they produce.

