GraphQL

Client & Server Fundamentals

Wyatt Preul

- github.com/geek/cerner
- Joyent - original corporate steward of Node.js.
- Production best practices with Node.js

Outline

- Introduction
- Schema/Query
- Server
- Mutation/Subscription
- Client
- Pitfalls
- Testing
- Summary

Introduction

- Type system for describing data and API
- Query language for interacting with APIs
- Provides extra benefits:
  - Batched requests
  - Req/res validation
  - Usage tracking
  - Responses only contain data client expects

Specification / Libraries

- Facebook project started in 2012 -
- Open spec, free to use, no copyright worries
- GraphQL JS library for execution engine
- GraphQL Foundation (a linux foundation project)

Schema

```
type User {
  id: ID!                 // ! = required
  email: String!
  firstname: String
  lastname: String
}

type Query {
  getUser(email: String!): User
}
```

Schema types/scalars

- Query, Mutation, Subscription
- String, Boolean, ID (String), Int, Float
- ! - denote required
- [] - denote array
- enum, input, interface, union

Query

```
query {
  getUser(email: "test@test.com") {
    firstname
    lastname
  }
}
```

Demo of schema/query

Query Content-Type

- JSON compatible
- Doesn’t require any custom client libraries, can reuse axios, fetch, or even simply cURL from the CLI
- Demo: server1-query.sh

Server Responsibilities

- Load and process schema
- Must be able to parse and execute queries

Server libraries

- hapi - Node.js framework for web apps
- graphi - GraphQL plugin for hapi
- [example usage in server1.js]

Schema Enum

```
type Address {
  lineone: String!
  linetwo: String
  city: String!
  state: StateCode
  zipcode: String
}

enum StateCode {
  MO
  KS
}
```

Sub query example

```
query {
  getUser(email: "test@test.com") {
    email
    firstname
    lastname
    address { lineone }
  }
}
```

Batch query example

```
query {
  user1: getUser(email: "test1@test.com") {
    address {
      lineone
    }
  }
  user2: getUser(email: "test1@test.com") {
    address {
      lineone
    }
  }
}
```

Variables example

```
query getUser($email1: String!, $email2: String!) {
  user1: getUser(email: $email1) {
    email
    firstname
    lastname
  }
  user2: getUser(email: $email2) {
    email
    firstname
    lastname
  }
}
```

Variables JSON

```
{
  “query”: “query getUser($email: String!) {
    getUser(email: $email) {
      email, firstname, lastname
    }
  }”,
  “variables”: {
    “email”: “test@test.com"
  }
}
```

Handlers as resolvers

- Help migrate from REST
- Access the full request object
- Utilize hapi auth at a per handler/resolver
- [server3.js]

Mutations

- Same as a query from clients perspective
- Specified as a different type: Mutation
- Begin to use `Input` type

Mutation Schema

```
input UserInput {
  email: String!
  firstname: String!
  lastname: String!
}

type Mutation {
  createUser(user: UserInput!): User
}
```

Mutation Request

```
mutation {
  createUser(user: {
    email: "test@test.com"
    firstname: "Foo"
    lastname: "Bar"
  }) {
    id
  }
}
```

Subscriptions

- Allow clients to subscribe to server events
- WebSockets are typically used
- Useful for keeping client store in sync with server state changes

Subscription Schema

```
type Subscription {
  userCreated: User
}
```

```
server.plugins.graphi.publish('userCreated', user)
```

Client Requests

- No extra libraries required
- Can ask for only data that is needed
- See example under client/store

Pitfalls

- Assuming that a GraphQL client is required
- Creating more type relations than clients need (can result in a lot of extra work on server)
- Over-stitching schemas, you aren't doing anything wrong to have multiple /graphql routes (admin vs user)

Testing

- test handlers like in REST
- validate schemas with graphql parse
- test queries with easygraphql-tester [see test/]

Summary

- GraphQL is a powerful addition to your toolset
- Supplement or replacement for REST
- Useful when multiple client teams with varying constraints
- Mature ecosystem and helpful community