A Seminar Report On

GRAPHQL:A DATA QUERY LANGUAGE

SEMINAR REPORT SUBMITTED TO

SAVITRIBAI PHULE PUNE UNIVERSITY,PUNE

In partial fulfillment for the degree of

BACHELOR OF ENGINEERING

(COMPUTER ENGINEERING)

BY:
OMKAR PRASHANT SHELAR

EXAM SEAT NO:T150394390

VISHWAKARMA INSTITUTE OF INFORMATION TECHNOLOGY,PUNE COMPUTER ENGINEERING DEPARTMENT APRIL-MAY 2018

Certificate

This is to certify that the following student of T.E. Computer, Vishwakarma Institute of Information Technology, Pune

OMKAR PRASHANT SHELAR

has successfully completed the Seminar and Technical Communication Report on

GraphQL : A data query language

in the partial fulfillment of the requirement for the completion of T.E. in Computer Engineering in 2018 as prescribed by the Savitribai Phule Pune University.

A.A.BHOSALE
Guide

Dr.S.R.SAKHARE
H.O.D.

Dr.B.S.KARKARE
Director

Date:- 23/3/2018

Place:-Pune

Abstract

In the past few years, the data has been increasing. With the increase in data, distribution of data has become a problem.

Today, to distribute data and resources among programs over the Internet, we use REST and SOAP standards. Though these systems work, they are not efficient or simply do not satisfy the ever-increasing requirements of current systems.

Hence Facebook in 2012, started working on GraphQL which solved some of the issues of REST systems. GraphQL though dissimilar to REST can be used in querying data on remote servers in a more interactive and efficient way.

This report gives a brief introduction to GraphQL and compares it with a typical REST system. This report highlights the advantages and disadvantages of both REST and GraphQL. Helping understand the use cases for each.

Acknowledgment

It is matter of great pleasure for me to submit this seminar report on ”GraphQL: A data query language”, as a part of curriculum for “Bachelor in Engineering (Computer Engineering)” of University of Pune, I am thankful to my guide Prof. Amol Bhosale, Assistant Professor in Computer Engg Department for his constant encouragement and able guidance. I am also thankful to Dr.B.S.Karkare, Director of VIIT Pune, Dr. S.R. Sakhare Head of Computer Department for their valuable support. I take this opportunity to express my deep sense of gratitude towards those, who have helped us in various ways, for preparing my seminar. At the last but not least, I am thankful to my parents, who had encouraged and inspired me with their blessings.

Omkar Shelar

Contents

1 Introduction

1.1 The data distribution problem

Data in today's day and age is ever increasing. The relationships between data are also increasing. Problems related to archival, retrieval, interoperability, discovery and distribution is getting complicated. Interoperability of data among organizations and applications is getting complex. Large technology companies are moving towards distributed systems because it helps them scale and manage effectively. The frequency with which organizations are collecting and making data available is increasing. Data collected is used by users/developers via **Application Programming Interfaces**(APIs).

APIs help in querying the database of another user/organization and retrieving the data that program is interested in. APIs help computer programs communicate with each other. APIs also refer to performing some task in some remote computing environment.

API documentation provides the set of endpoints and query parameter to query another program. This program retrieves, translates and sends the data to the requesting program.

1.2 Traditional Solutions

Traditional and most widely used solutions to the communications and data distributions problem are using **REepresentational State Transfer**(REST) and **Simple Object Access Protocol**(SOAP). These solutions(especially REST) are widely used for communication between programs and exchanging data over the Internet.

1.3 Modern Solution - GraphQL

API landscape has changed over the last few years. Increase in low-powered, IoT and mobile devices with low network capabilities lead to the development of GraphQL. GraphQL is a data query language and an API standard. Provides a more powerful, flexible alternative to REST. Initially developed for internal use in Facebook Inc. in 2012. Facebook publicly talked about GraphQL in React.js Conference 2015.

It was created in response to performance problems of Facebook's mobile native API's. It enables declarative data fetching by the client. The client describes the variety and structure of the data. Hence preventing over fetching and under fetching of data. This client-centric approach is one of the key factors for the success of GraphQL.

Many companies including Facebook, GitHub, Coursera, Shopify, New York Times have been using GraphQL in production ever since.

2 Literature Survey

The IEEE paper titled ”GraphQL for Archival Metadata: An Overview of the EHRI GraphQL API”, gives the history of GraphQL technology and why Facebook Inc. developed the technology to increase the performance of their mobile users.

The IEEE paper gives an overview of the technology and how is it implemented in EHRI for archival metadata. The author, Mike Bryant gives a brief introduction of GraphQL and it's queries and mutations. The author gives example queries in the paper with the example outputs.

The author describes why GraphQL has advantages over REST. He describes why GraphQL was a better choice for EHRI data.

The author compares GraphQL with Structured Query Language(SQL) and SPARQL. He describes how SQL queries are a form of relational calculus as compared to GraphQL which returns a hierarchical response.

The paper also gives a brief introduction to debugging and testing tools and documentation generation tools for GraphQL endpoints.

The paper highlights, particularly how GraphQL is important for extracting relationships between data. The paper describes relationships as ”first-class” data items.

The papers also highlight how EHRI is using the GraphQL API for bulk data retrieval and retrieval of contextual data. The API has been applied to a graph database namely Neo4j in EHRI.

The master thesis titled ”API Design in Distributed Systems: A Comparison between GraphQL and REST” discussed GraphQL in a very detailed manner and provides a fair comparison of GraphQL and REST. The thesis highlights the advantages and disadvantages of both systems. It discusses how GraphQL will be useful in certain distributed systems implementations.

The thesis compares both GraphQL and REST based on the following criteria:

1. Operation Reusability
2. Discoverability
3. Component Responsibility
4. Simplicity
5. Performance
6. Interaction Visibility
7. Customization

Looking at the comparison of the criteria, we understand what kinds of applications both REST and GraphQL are suitable for.

3 Motivation

In 2012, Facebook Inc. saw a huge growth in mobile native users. Their REST API system experienced a high number of requests from mobile applications. The performance of their mobile applications started dropping.
This led to the creation of the GraphQL specification and run-time.

Many other companies were working on similar projects(Eg: Netflix was working on Falcor) to solve the complexity of data requested by clients from servers.

4 REST API systems

4.1 Introduction

REST APIs are primary means of interaction with web applications. They basically provide a layer of abstraction between the core functions of the systems and the user. REST standard was developed by Roy Fielding in 2000. It was adopted as a standard by most web applications used today.

4.2 Advantages of REST systems

4.2.1 Stateless:

REST systems are completely stateless. They follow a stateless protocol at the application layer(HTTP). Every request in a REST system is treated independently. This feature makes it highly scalable.

4.2.2 Structured access to resources

REST systems has a definite structure(JSON) by which they produce responses. Structured data helps the client to parse data easily.

4.2.3 REST specification is strict

REST specification is very strict in regard to what endpoints are to be exposed and the way in which they are to be exposed.

4.3 Problems with REST API systems

The applications in the early 2000s(when REST standard was developed) did not have the need to exchange such huge amounts of information we exchange today. Certain limitations of REST APIs make them less efficient and affect the performance of the client and server systems.

4.3.1 Multiple Endpoints:

A REST API system has multiple endpoints. the developers have to manage all the endpoints separately, version the endpoints properly and protect these endpoints against network attacks.

4.3.2 Over Fetching of data

In a REST API system the server decides the data and structure of data in the response body. This might lead to sending data with is not required to the client. This increases the bandwidth costs of the server as well as the client. There is also a network latency increase.

4.3.3 Multiple Requests

To query a REST API system, the client might need to send multiple requests to the server at multiple different endpoints. This decreases the performance of the client as it has to wait for all requests to complete.

This performance can decrease even more if the next request depends on the response to the previous request. This will lead to the client sending sequential requests which will increase the latency rapidly as each request has its own round-trip latency.

4.3.4 Error Handling

In REST systems, the client needs to check the HTTP response headers to find the errors received. In a GraphQL API however, the HTTP response status is always 200 OK but has the specific error in the response body.

4.3.5 Versioning

REST APIs are inflexible to server-side changes. Server-side code is changing rapidly and is continuously deployed. This rapid development might break the APIs.

In REST systems, different versions of APIs are maintained. This can lead increase of endpoints, duplication of endpoints. The API endpoints have to be maintained. In some cases, switching versions can lead to changes in the client programs.

5 GraphQL: An Introduction

GraphQL was developed to overcome many of the shortcomings and limitations of client-server communications using REST. GraphQL is described as ”a query language for your API”. GraphQL also is a server run-time for executing queries by the type system.

GraphQL can be used with any database system. Any existing REST system can be implemented as a GraphQL server without rewriting the data layer.

For example the query:

```
{
  me {
    name
  }
}
```

Could produce the JSON result:

```
{
  "me": {
    "name": "Luke Skywalker"
  }
}
```

6 GraphQL Terminologies

6.1 Schema:

Schema is central to GraphQL. A schema describes the available types, queries, and mutations. A schema element is divided into query and mutations. Each query contains objects. Objects contain fields. Fields have specific data types. GraphQL is strongly typed. The data types of GraphQL are described further. Exclamation mark after a field represents a non-nullable field. Enclosed square braces represent an array of enclosed types. Order of query and mutations can be changed when requesting a GraphQL endpoint.

Example Schema:

```
schema {
    query:Query
    mutation:Mutation
}

type User {
    id:ID!
    nickname:String
    post:[Post]!
    followees: [User]!
    followers: [User]!
}
type Post {
id: ID!
author: User
content: String
# The ISO representation of the date when the post was created.
createdAt: String
replies: [Post]!
replyTo: Post
}

type Query {
    timeline(of: String): [Post]!
    user(nickName: String): User
    users: [User]!
}
type Mutation {
writePost(authorNick: String, content: String, replyTo: String = null):
     Post
newUser(nickName: String): User
followUser(me: String, other: String): User
```

```
}
```

6.2 Queries:

Queries are used to retrieve data from a GraphQL service. The structure of the query is similar to the structure of the reponse generated by the query.

Example Query:

```
{
  hero {
    name
    # Queries can have comments!
    friends {
      name
    }
  }
}
```

Result of the above query:

```
{
  "data": {
    "hero": {
      "name": "R2-D2",
      "friends": [
        {
          "name": "Luke Skywalker"
        },
        {
          "name": "Han Solo"
        },
        {
          "name": "Leia Organa"
        }
      ]
    }
  }
}
```

The above example shows use relation between the structure of the query and the response generated by the GraphQL server. This has a significant advantage over REST API system because the structure of the response is in control of the client.

Also, note that the ”friends” field is of the array type. The query will remain the same even if one object is present in the response. The client has to handle such type of anomalies.

6.3 Arguments:

Arguments in GraphQL are used for filtering data in the GraphQL server. They are similar to the `?id=1` in a REST system. Arguments can also be given to specific scalar fields.

Example of Arguments:

```
{
  human(id: "1000") {
    name
    height(unit: FOOT)
  }
}
```

Will produce the result:

```
{
  "data": {
    "human": {
      "name": "Luke Skywalker",
      "height": 5.6430448
    }
  }
}
```

6.4 Alias:

The client program can also give alias to fields in a GraphQL query.

Example of Alias in GraphQL:

```
{
  empireHero: hero(episode: EMPIRE) {
    name
  }
  jediHero: hero(episode: JEDI) {
    name
  }
}
```

Will produce the result:

```
{
  "data": {
    "empireHero": {
      "name": "Luke Skywalker"
    },
```

```
    "jediHero": {
      "name": "R2-D2"
    }
  }
}
```

6.5 Query Naming:

Each query can be given a name. Though naming of queries is optional, query naming is recommended for production applications. It helps in debugging and logging purposes. **Example:**

```
query HeroNameAndFriends {
  hero {
    name
    friends {
      name
    }
  }
}
```

The above query will produce the result:

```
{
  "data": {
    "hero": {
      "name": "R2-D2",
      "friends": [
        {
          "name": "Luke Skywalker"
        },
        {
          "name": "Han Solo"
        },
        {
          "name": "Leia Organa"
        }
      ]
    }
  }
}
```

Note: Naming the query does not have any effect on the response. It is just used as a debugging tool on the server-side.

6.6 Variables:

GraphQL supports variables in queries. This is particularly helpful because most of the time programs send GraphQL queries. The structure of the query remains same. Variables can be used by passing the GraphQL query a dictionary.

Here's a sample query:

```
query HeroNameAndFriends($episode: Episode) {
  hero(episode: $episode) {
    name
    friends {
      name
    }
  }
}
```

```
{
  "episode": "JEDI"
}
```

6.7 Directives:

Directives allow the client to dynamically change the structure of the query based on values of variables. Two types of directives are supported:

`@include(if: Boolean)` : This only includes the field when the condition is `true`.

`@skip(if:Boolean)` : This skips the field if the condition is `true`.

Example Query:

```
query Hero($episode: Episode, $withFriends: Boolean!) {
  hero(episode: $episode) {
    name
    friends @include(if: $withFriends) {
      name
    }
  }
}
```

Corresponding dictionary:

```
{
  "episode": "JEDI",
  "withFriends": false
}
```

Result:

```
{
  "data": {
    "hero": {
      "name": "R2-D2"
    }
  }
}
```

6.8 Mutations:

Mutations help in modifying the data on the server-side. It is not unlike a POST request in REST APIs. There is no restriction on using an HTTP POST request to modify server data but following the standard is expected. **Example Mutation:**

```
mutation CreateReviewForEpisode($ep: Episode!, $review: ReviewInput!) {
  createReview(episode: $ep, review: $review) {
    stars
    commentary
  }
}
```

Example dictionary:

```
{
  "ep": "JEDI",
  "review": {
    "stars": 5,
    "commentary": "This is a great movie!"
  }
}
```

Result:

```
{
  "data": {
    "createReview": {
      "stars": 5,
      "commentary": "This is a great movie!"
    }
  }
}
```

7 Type system of GraphQL

GraphQL implements an independent type system. This helps in implementing the specification in multiple languages.

7.1 Objects and fields

Object type is essentials to GraphQL systems. Each object is a JSON with keys representing the attributes or fields of the object.

7.2 Scalar type system of GraphQL

GraphQL is strongly typed. This helps in maintaining data integrity in strongly typed programming languages.

- `Int` : A signed 32-bit integer.
- `Float` : A signed double-precision floating-point value.
- `String` : A UTF-8 character sequence.
- `Boolean` : `true` or `false`.
- `ID` : is a scalar type that represents a unique identifier. Used to re-fetch the object from cache.

8 Tools and libraries

8.1 Graphiql

Graphiql is a tool most widely used for testing and debugging GraphQL endpoints.

8.2 Libraries

8.2.1 Server Libraries

Many popular programming languages have libraries for implementing a GraphQL server. Examples:

- Python - Graphene
- JavaScript - GraphQL.js
- PHP - graphql-php
- Java - graphql-java
- Scala - Sangria
- Ruby - graphql-ruby

8.2.2 Client Libraries

Though requests for queries and mutations can be sent without using libraries, the user can use a client side library for convenience. Examples:

- Java/Android - Apollo
- Python - GQL
- JavaScript - Relay/Apollo Client

9 Advantages of GraphQL over REST

9.1 Single Endpoint Multiple Resources:

GraphQL queries are made to a single endpoint and all the resources are available through the single endpoint. This helps the API provider manage fewer endpoints.

9.2 No Versioning required:

The major reason why REST APIs have versions is that new features need to be exposed. Versioning breaks the structure of the data in REST systems.

In GraphQL however, the client decides the structure of the data, preventing the need to version the APIs. This again leads to fewer endpoints on the API providers end making it easier to manage the endpoints.

9.3 Prevents problems of over-fetching/under-fetching:

In GraphQL, the client decides the structure and the variety of data that it wants. This leads to the response being exactly as the client needs it to be.

9.4 Prevents multiple requests:

In GraphQL, related data can be queried on the same endpoint. This means that the client need not send multiple requests at multiple endpoints to the server. Hence decreasing the latency of task the client wants to perform as each task takes only one round-trip to the server.

9.5 Better Error Handling:

In GraphQL, the actual error message is transmitted in the response body. This is particularly helpful to the client as it knows exactly what error occurred on the server. This is much better than the REST systems where the client has to rely on HTTP status codes for determining the error.

10 Disadvantages of GraphQL

10.1 Caching

This is a single most important disadvantage for GraphQL systems. Most caching systems are HTTP based caching systems, which cache content from multiple endpoints.

GraphQL, however, exposes only one single endpoint. This makes it difficult to cache using existing HTTP caching tools.

GraphQL does implement caching in a certain way but the client has to take care of all the caching needs.

10.2 Expensive Queries

The flexibility that GraphQL offers also has certain disadvantages. The depth of the requests by the client can increase to a large amount. This can make that processing that GraphQL operation expensive.

Enterprises while using GraphQL in production have to run tests on each and every query and make sure that the query is not expensive on the database and servers.

10.3 Relatively a new specification

Relative to REST API systems, GraphQL is a new technology. It is to be seen as to whether enterprises make the switch to GraphQL specifications from existing tested and robust REST systems.

10.4 Needs dependencies on server-side

GraphQL need dependencies on the server-side. Example ”graphene” is needed on the server side to implement GraphQL in Python.

11 Source Code

```
# /app.py
from flask import Flask, request, jsonify
from flask_graphql import GraphQLView
from models import db_session, Users as UsersModel, Follows as
    FollowsModel
import json
from schema import schema

app = Flask(__name__)
app.debug = True

app.add_url_rule(
    '/graphql',
    view_func=GraphQLView.as_view(
        'graphql',
        schema=schema,
        graphiql=True # for having the GraphiQL interface
    )
)

@app.route('/rest/user/', methods=["GET"])
def rest_endpoint_handler():
    id = request.args.get('id')
    if id is not None:
        user = db_session.query(UsersModel).filter(UsersModel.id ==
            id).first()
        followers =
            db_session.query(FollowsModel.follow_by).filter(FollowsModel.follow_to
            == id).all()
        d = dict()
        d['firstName'] = user.firstName
        d['lastName'] = user.lastName
        d['username'] = user.username
        d['email'] = user.email
        l = list()
        for follower in followers:
            l.append('/rest/user/?id='+str(follower[0]))

        d['followers'] = l
        return jsonify(resultSet = d)
```

```
    else:
        users = db_session.query(UsersModel).filter().all()
        l1 = list()
        for user in users:
            followers =
                db_session.query(FollowsModel.follow_by).filter(FollowsModel.follow_to
                == user.id).all()
            d = dict()
            d['firstName'] = user.firstName
            d['lastName'] = user.lastName
            d['username'] = user.username
            d['email'] = user.email
            l = list()
            for follower in followers:
                l.append('/rest/user/?id='+str(follower[0]))
            d['followers'] = l
            l1.append(d)
        return jsonify(resultSet = l1)


@app.teardown_appcontext
def shutdown_session(exception=None):
    db_session.remove()

if __name__ == '__main__':
    app.run()
```

```
# /models.py
from sqlalchemy import *
from sqlalchemy.orm import (scoped_session, sessionmaker, relationship,
                            backref)
from sqlalchemy.ext.declarative import declarative_base

engine = create_engine('sqlite:///database.sqlite3',
     convert_unicode=True)
db_session = scoped_session(sessionmaker(autocommit=False,
                                         autoflush=False,
                                         bind=engine))

Base = declarative_base()
Base.query = db_session.query_property()

class Users(Base):
```

```
    __tablename__='users'
    id = Column(Integer, primary_key=True)
    firstName = Column(String)
    lastName = Column(String)
    username = Column(String)
    email = Column(String)

class Follows(Base):
    __tablename__='follows'
    id = Column(Integer, primary_key=True)
    follow_by = Column(Integer, ForeignKey('users.id'))
    follow_to = Column(Integer, ForeignKey('users.id'))
```

```
# /schema.py
import graphene
from graphene import relay
from graphene_sqlalchemy import SQLAlchemyObjectType,
    SQLAlchemyConnectionField
from models import db_session, Users as UsersModel, Follows as
    FollowsModel

class UserType(graphene.ObjectType):
    name = "UserType"
    description = '...'

    email = graphene.String()
    firstName = graphene.String()
    id = graphene.Int()
    lastName = graphene.String()
    username = graphene.String(description='Something you forget often')
    followers = graphene.List(lambda:UserType)


    def resolve_followers(self, info, **args):
        followers_list =
            db_session.query(FollowsModel.follow_by).filter(FollowsModel.follow_to
            == self.id).all()
        l = list()

        for follower in followers_list:
            l.append(follower[0])

        l = tuple(l)
```

```
        userList =
            db_session.query(UsersModel).filter(UsersModel.id.in_(l)).all()
        return userList

    db_session.query(PostsModel).filter(PostsModel.post_by ==
         self.id).all()


class QueryType(graphene.ObjectType):
    name='query'
    description='...'

    user = graphene.Field(
        UserType,
        id = graphene.Int()
        )

    def resolve_user(self, info, **args):
        id = args.get('id')
        requiredUser = db_session.query(UsersModel).filter(UsersModel.id
            == id)
        print(requiredUser.first())
        return requiredUser.first()

schema = graphene.Schema(query=QueryType)
```

12 Future Work

1. **Reusability of operations**: The dynamic way in which GraphQL fetches data makes it difficult for safely caching the data. This makes network caching tools essentially ineffective.
   While it is true that GraphQL has a unique ID for identification and caching, but the client has to handle the caching and network caching tools and web browsers cannot handle single endpoint HTTP caching.
2. **Relatively a new technology:** GraphQL is relatively a new technology. It is still evolving. The GraphQL schema definition language was made part of the specification in February 2018.

13 Conclusion

GraphQL has really made data querying to the server much more flexible and powerful. The responsibilities on the client side are also increasing because of the flexibility that GraphQL provides.

GraphQL does have a few disadvantages, but it will be interesting to see how the future work on the specification will tackle these problems.

GraphQL though a new technology and specification, will really be crucial to solve network latency and performance issues on the API consuming systems. It will also be helpful in distributed micro-services architectures.

As more and more enterprises are implementing GraphQL endpoints over REST endpoints, the flexibility and efficiency of these systems both on the client and server side will improve drastically.

Over the coming years it will be interesting to see how enterprise APIs evolve with GraphQL. It will also be interesting to see how enterprises implement defenses against expensive GraphQL queries.

14 References

1. Mike Bryant, ”GraphQL for Archival Metadata: An Overview of the EHRI GraphQL API” IEEE, 2017 IEEE International Conference on Big Data (BIGDATA).
2. Thomas Eizinger, ”API Design in Distributed Systems: A Comparison between GraphQL and REST” Master’s Thesis(University of Applied Sciences Technikum Wien)
3. Eyob Semere Ghebremicael, ”Transformation of REST API to GraphQL for OpenTOSCA” Master’s Thesis (University of Stuttgart)
4. https://graphql.org/
5. https://www.howtographql.com/
6. https://blog.pusher.com/rest-versus-graphql/
7. https://medium.freecodecamp.org/rest-apis-are-rest-in-peace-apis-long-live-graphql-d412e559d8e4
8. https://www.moesif.com/blog/technical/graphql/REST-vs-GraphQL-APIs-the-good-the-bad-the-ugly/’
9. https://github.com/facebook/graphql
10. https://github.com/graphql-python/flask-graphql