

**Czech University of Life Sciences Prague**

**Faculty of Economics and Management**

**Department of Information Technologies**



**Diploma Thesis**

**Comparison between different API architectures  
(GraphQL, REST)**

**Muhammad Raufur Rahman, B.Sc.**

**© 2020-2021 CULS Prague**

## **Declaration**

I declare that I have worked on my diploma thesis titled "Comparison between different API architectures (GraphQL, REST)" by myself, and I have used only the sources mentioned at the end of the thesis. As the author of the diploma thesis, I declare that the thesis does not break the copyrights of any person.

In Prague on 30.11.2021

---

### **Acknowledgement**

I would like to thank Professor Josef Pavlíček, Ph.D., for his valuable advice and support during my work of this thesis. I would also like to thanks my parents and family, who always encourage my study and provide supports.

# **Comparison between different API architectures (Graphql, REST)**

## **Abstract**

The aim of the work is to compare different API architectures of GraphQL and REST depends on the parameters of implementation, speed, usability, and maintainability. The final result will contain the basic guide of implementation and a detailed comparison list.

**Keywords:** Graphql, REST, API, Comparison, Implementation

# Table of content

<b>1</b>	<b>Introduction.....</b>	<b>11</b>
<b>2</b>	<b>Objectives and Methodology .....</b>	<b>12</b>
2.1	Objectives .....	12
2.2	Methodology.....	12
2.3	Structure.....	12
<b>3</b>	<b>Literature Review .....</b>	<b>13</b>
3.1	REST (Representational State Transfer) .....	13
3.1.1	History .....	13
3.1.2	Constraints.....	13
3.1.3	Resource .....	15
3.1.4	Components.....	16
3.1.5	Connectors.....	16
3.1.6	ROA(Resource oriented architecture).....	17
3.2	GraphQL .....	18
3.2.1	History .....	19
3.2.2	Architectural overview.....	20
3.2.3	Data Types.....	22
3.2.4	Queries.....	23
3.2.5	Mutation .....	25
3.2.6	Subscription.....	27
3.2.7	Introspection.....	27
3.2.8	Schema .....	28
3.3	Result .....	32
<b>4</b>	<b>Practical Part .....</b>	<b>33</b>
4.1	Tools .....	33
4.1.1	Apollo.....	33
4.1.2	Postman .....	34
4.1.3	Github.....	34
4.1.4	Visual Studio Code.....	34
4.1.5	Technologies .....	35
4.2	Practical setup.....	35
4.2.1	Source code .....	35
4.2.2	REST Server.....	36
4.2.3	GraphQL Server .....	39

4.2.4	Test Platform .....	42
4.2.5	Data.....	44
4.3	Workflow .....	45
4.3.1	Initial.....	45
4.3.2	Express Server .....	46
4.3.3	GraphQL Server .....	46
4.3.4	Postman Setup .....	46
4.3.5	Testing parameters.....	46
<b>5</b>	<b>Results and Discussion.....</b>	<b>47</b>
5.1	Implementation .....	47
5.2	Speed.....	49
5.3	Usability .....	53
5.4	Maintainability .....	55
5.5	Conclusion .....	59
<b>6</b>	<b>Bibliography .....</b>	<b>61</b>

## List of figures:

Figure 3.1: Simple client-server data flow architecture .....	(16)
Figure 3.2: A simple graphql query example.....	(17)
Figure 3.3: GraphQL history.....	(18)
Figure 3.4: GraphQL direct DB diagram.....	(19)
Figure 3.5: GraphQL legacy architecture.....	(20)
Figure 3.6: GraphQL hybrid architecture.....	(21)
Figure 3.7: GraphQL query aggregate example.....	(23)
Figure 3.8: Fragment example.....	(23)
Figure 3.9: query for repository.....	(24)
Figure 3.10: mutation for change data.....	(24)
Figure 3.11: query after data update.....	(25)
Figure 3.12: __schemaingithubAPI.....	(26)
Figure 3.13: GraphQL schema type example.....	(27)
Figure 3.14: Schema types.....	(28)
Figure 3.15: One to One connection.....	(29)
Figure 3.16: One to one connection data flow.....	(29)
Figure 3.17: One to many connection query.....	(30)
Figure 3.18: Many to Many connection.....	(30)
Figure 4.1: Package entry file .....	(36)
Figure 4.2: ExpressJs server creation .....	(37)
Figure 4.3: Express 4.0 bin file .....	(38)
Figure 4.4: Express router example .....	(39)
Figure 4.5: GraphQL server config .....	(40)
Figure 4.6: GraphQL schema .....	(40)
Figure 4.7 : GraphQL query resolvers .....	(41)
Figure 4.8: REST integration .....	(42)
Figure 4.9: Postman interface .....	(43)

Figure 4.10: Session data example .....(43)

Figure 4.11: Speaker data example .....(44)

Figure 5.1: Speed test 1 chart.....(48)

Figure 5.2: Data flow from GraphQL to REST.....(49)

Figure 5.3: Speed test 2 chart.....(50)

Figure 5.4: Query granularity.....(51)

Figure 5.5: Maintainability features .....(55)



## List of tables

Table 3.1: REST constraints.....	(14)
Table 3.2: REST resource types [1, 5] .....	(15)
Table 3.3: GraphQL Data types.....	(23)
Table 4.1: Test machine's attributes.....	(43)
Table 5.1: Test case 1 data.....	(50)
Table 5.2: Test case 2 data.....	(51)
Table 5.3: Usability ranking . .....	(55)
Table 5.4: Maintainability ranking.....	(57)

## List of abbreviations

<b>UI</b>	<b>User Interface</b>
<b>SDLC</b>	<b>Software Development Life Cycle</b>
<b>API</b>	<b>Application Programming Interface</b>
<b>REST</b>	<b>Representational State Transfer</b>
<b>JSON</b>	<b>JavaScript Object Notation</b>
<b>HTTP</b>	<b>Hypertext Transfer Protocol</b>
<b>ROA</b>	<b>Return On Assets</b>
<b>SOAP</b>	<b>Simple Object Access Protocol</b>
<b>JWT</b>	<b>JSON Web Token</b>
<b>URI</b>	<b>Uniform Resource Identifier</b>
<b>URL</b>	<b>Uniform Resource Locator</b>
<b>XML</b>	<b>Extensible Markup Language</b>
<b>JS</b>	<b>Javascript</b>
<b>SDL</b>	<b>Schema Definition Language</b>
<b>DDR3</b>	<b>Double Data Rate Type 3</b>
<b>GHZ</b>	<b>Giga Hertz</b>
<b>CRUD</b>	<b>Create Read Update Delete</b>
<b>OOP</b>	<b>Object Oriented Programming</b>
<b>SSO</b>	<b>Single sign on</b>
<b>I/O</b>	<b>Input and output</b>
<b>DB</b>	<b>Database</b>

# 1 Introduction

Software development depends on data flow in various layers. It is an orchestration from the server to UI. All the pieces should work as they should be to maintain an outstanding user experience.

In the modern era of SDLC, data becomes a very complex attribute to manage. Continuously increasing user, role, features are summing up to more data top on previous data. The progressed amount of data is exponential.

Software developers usually manage their data via API and distribute them within a secure API gateway.

REST is the most competitive and used API architecture, and simplicity of usage creates enough user base. However, it has its downsides and gets over-engineered easily in industrial SDLC.

GraphQL is the latest addition in the data flow market, developed by Facebook Inc in 2010 to tackle the downsides of REST, which is not a protocol system like REST but a middleware layer top on the gateway. Enable data flow is more sophisticated than ever.

The comparison is not easy. The technologies are new and, not too many examples are there to analyze. Also, as stated, GraphQL is not an architecture or gateway itself. It is a query language system. This thesis tries to give a technical comparison between REST and GraphQL and choose the best platform for the appropriate usage.

## 2 Objectives and Methodology

### 2.1 Objectives

The objective of this thesis to compare REST and GraphQL in different parameters, like complexity, usability, speed, maintainability, and provides a guideline to both directions via implementation examples.

In this thesis, the following questions should be addressed and solved.

- What is the advantage of using GraphQL?
- When to integrate GraphQL and when to use REST itself?
- GraphQL is a schema language and, REST is an architecture, and why the comparison?
- When not use GraphQL?

### 2.2 Methodology

The methodology for this thesis will reflect a practical approach. Research on selected technologies depends on research findings, design a sustainable and testable solution with proper test-case and used-case. All code and technical assets related to this thesis will share on a public repository, which will be visible and reusable.

Depends on the test result and, other research findings will conclude and regulate the objectives previously defined.

### 2.3 Structure

The rest of the thesis will separate into parts, and Chapter 3 will provide a detailed literature overview for technologies(REST, GraphQL). Chapter 4 will contains the experimental part setup and execution in chapter 5, discussion, and conclusion.

## 3 Literature Review

### 3.1 REST (Representational State Transfer)

REST is a software architectural style that defines a set of constraints to be used for creating web services. REST strength is heavy load-balancing and scalability. It is also user-friendly because of using JSON and HTTP protocols.[1]

REST has three architectural element classes:

- Resource
- Connectors
- Components

#### 3.1.1 History

Back in 2000, Roy Fielding proposed the REST architectural style as his Ph.D. dissertation. He develops REST as a similar service to HTTP. Before that, API architecture was heavily dependent on SOAP(Simple Object Access Protocol) methodology, which was very complex to develop. Compared to SOAP, REST is much lighter and protocol-friendly. [1] [2]

#### 3.1.2 Constraints

Constraints are a set of rules that web service needs to obtain to be defined as REST. There are 5 constraints defined for REST architecture. For data, the payload transfer server should follow all of the constraints to maintain architectural integrity. Those are:

1. Client-Server decoupled, which defines that server and UI should be different layers and can not communicate with each other without proper API authentication.
2. Statelessness, client request should not contain and bind by application state or specific context. Client requests maintain defined properties, those only provide

enough parameters to retrieve exact data which require to fulfil the specific need. No session and token should be passed or maintained by API itself.

3. Caching, the server must include a cacheable or cacheable tag to the request payload, which should not be defined by the client.
4. Layer approach, client, and server should be connected via multiple middleware layers. A client should not have direct access to the server, only load-balancer should enable the client to a specified server. Also, the security layer should be a separate part of the client and the server.
5. Uniform interface, is the most important from a development perspective. It is more likely to maintain integrity across codebase and server requests. The purpose of a uniform interface is visibility and transparency. Below constraints are part of the Uniform interface:

The resource should be identified via request, a server can return JSON, XML,

- a) HTML response depends on request.
- b) The client should have enough information to manipulate the resource itself.
- c) All messages should be self-describing.
- d) Hypermedia as the engine of the application.

Another optional constraint, Code-on-Demand, a server can execute or send code to client ad-hoc. [2] [3]

<b>Constraint</b>	<b>Explanation</b>
Client-Server	UI and server should be different integrity
Statelessness	Client request should not contain/bin any state.
Cacheable	Server should include cacheable tag in payload.
Uniform Interface	Maintain integrity across codebase.

Table 3.1: REST constraints.

### 3.1.3 Resource

To understand REST, familiarization with its elements is essential. From an architectural point, REST consists of 6 elements. All of the elements represent different resource types and the connector between them. At a glance

<b>Element</b>	<b>Description</b>	<b>Example</b>
Resource	URI based data output.	JSON, Image.
Identifier	An URL which identifies resource path.	Czu.com/student.
Metadata	Description of resource, provide controlled payload information, status, location.	Link, Network payload description.
Controller	Logical data distribution	Authentication, authorization.

Table 3.2: REST resource types [1, 5]

Resources work as a representation of any kind of data that a REST service will provide via some controlled system or not. A resource should be identified and used separately from other resources via the identifier. Unique design identifiers act as a different data endpoint in web-based architecture. Generally, the client knows the list of identifiers, which can be called depends on the requirement.

Example: Below identifier URL represent an online e-com data endpoint

- [www.example.com/all\\_customer](http://www.example.com/all_customer)
- [www.example.com/customer/id](http://www.example.com/customer/id)
- [www.example.com/customer/id/cart](http://www.example.com/customer/id/cart)
- [www.example.com/item/id](http://www.example.com/item/id)

Another important feature of a resource is the controller, which defines who can access and how many resources can get based on what identity provides.

Controller acts as a gate-keeper of a resource URI. Different authorization and authentication protocols should be in place to control resources. Control resources use different parameters of credentials and JWT to allocate resources for a specific or extensive time.

### 3.1.4 Components

There are four main classes in the REST components.

- Origin server
- User-agent
- Proxy
- Gateway

The origin server works as a resource holder, the server connector gives access to the resource inside the origin server.

User-agent is the client-side class, which responsible for resource calls and display and collect constraints, it calls the client controller to connect with the origin server.

Proxy and Gateway components usually handle requests and connections. Proxy works as a client-side data translator, and security provider. Gateway act similar as proxy from server-side

### 3.1.5 Connectors

The connector represents the middle layer between components. It maintains components communication and provides encapsulation of code implementation. According to Roy Thomas Fielding, there are 5 types of connectors.[1]

1. Client
2. Server
3. Cache
4. Resolver
5. Tunnel

Encapsulation also provides simplicity to the connectors. Connectors hide implementation complexity from components, which enable replacement or agile development.



From top level server and client, connector performs basic operations such, a client sends a request, the server responds with a resource.

The cache can be used both layer, server, or client, which help to store part of a resource for future use.

Resolver, transpose identifier to network address and create a connection to the requested resource.

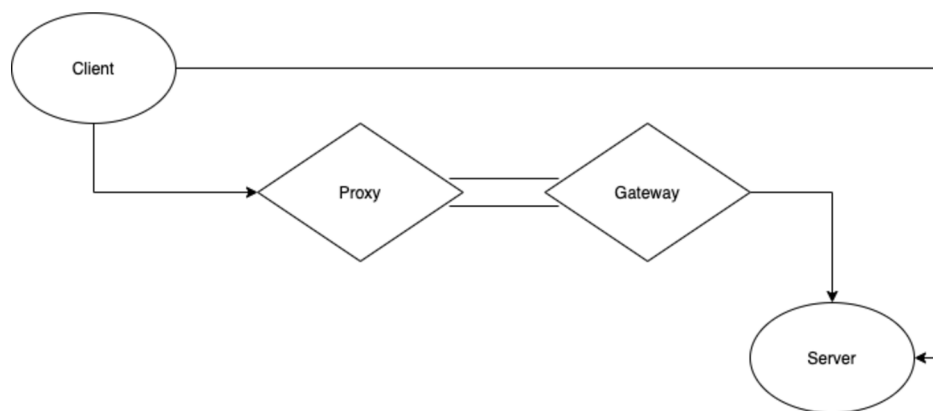


Figure 3.1: Simple client-server data flow architecture

### 3.1.6 ROA(Resource oriented architecture)

Understand REST guidelines can be relevant with ROA architecture understanding. Resource oriented architecture focus on software development resources such as data, piece of code, or specific functionality.

With REST discussion, ROA has to be included due to the interconnectivity of both topics. According to Richardson and Ruby ROA as follows: “The ROA is a way of turning a problem into a RESTful web service: an arrangement of URIs, HTTP, and XML that works like the rest of the Web, and those programmers will enjoy using”[6,7,8]

ROA provides addressability and statelessness to the REST guideline with the resource.

As we discussed before the resource identification, each resource much identifies with address/name. In ROA this address/name identify via URL. If there is no XML to identify the resource, there is no accessibility to that resource.[6]

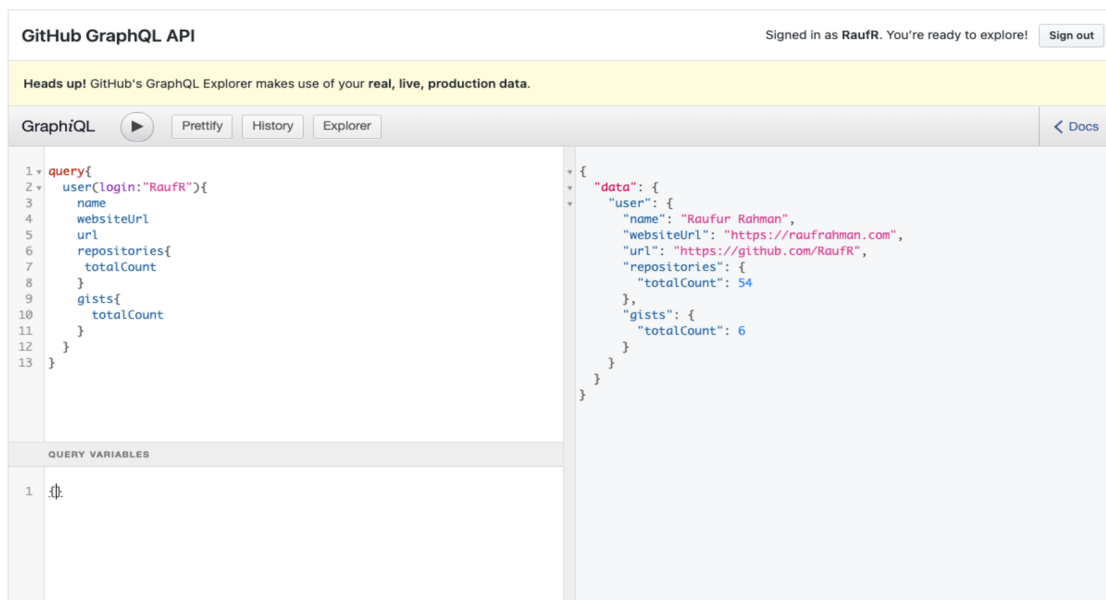
## 3.2 GraphQL

In core, GraphQL is a query language, which can be used in any database even on top of an existing API. [9]

Sometimes, GraphQL term can be flawed by the naming, it is very important to remember: GraphQL is not any Javascript library or binds to the only Javascript, queries define as an object which can be translated to any programming language.

GraphQL is not a data transport protocol, it can be used with any protocol, like HTTP, WebSocket.

And GraphQL does not provide any authentication or authorization or any data manipulation layer.



The screenshot shows the GitHub GraphQL API Explorer interface. At the top, it says "Signed in as RaufR. You're ready to explore!" with a "Sign out" button. Below that is a yellow banner: "Heads up! GitHub's GraphQL Explorer makes use of your real, live, production data." The main area is split into two panes. The left pane shows a GraphQL query:

```
1 query{
2   user(login:"RaufR"){
3     name
4     websiteUrl
5     url
6     repositories{
7       totalCount
8     }
9     gists{
10      totalCount
11    }
12  }
13 }
```

The right pane shows the JSON response:

```
{
  "data": {
    "user": {
      "name": "Raufur Rahman",
      "websiteUrl": "https://raufrahman.com",
      "url": "https://github.com/RaufR",
      "repositories": {
        "totalCount": 54
      },
      "gists": {
        "totalCount": 6
      }
    }
  }
}
```

At the bottom, there is a "QUERY VARIABLES" section with a single variable:

```
1 {
```

Figure 3.2: A simple graphql query example

Above example generated from

<https://docs.github.com/en/free-pro-team@latest/graphql/overview/explorer>

Which is opensource GraphQL API explorer for testing and fetching GitHub data and represent them. In the above example, we query for *user* RaufR via login name, then we fetch *name*, *websiteUrl*, *url* from graphql API, after that, we also requested two other field *repositories* and *gists* with specific *totalCount* argument.

On the right side, we have our data as a JSON object which is ready to use in any web implementation.

### 3.2.1 History

In 2012, Lee Byron, Nick Schrock, and Dan Schafer employees of FaceBook Inc start to rethink their way of data fetching. Because from beginning facebook mobile application both in IOS and Android platform is a wrapper around the website. At that time Facebook had REST server and because of complex queries and very decoupled data architecture, they are facing heavy performance issues and continued crash.

After 3 years, in 2015, they have released the initial production version under name of graphqlJs. Today GraphQL power almost 100% data fetching and many other industry giants like IBM, SAP, Airbnb has started to using GraphQL.

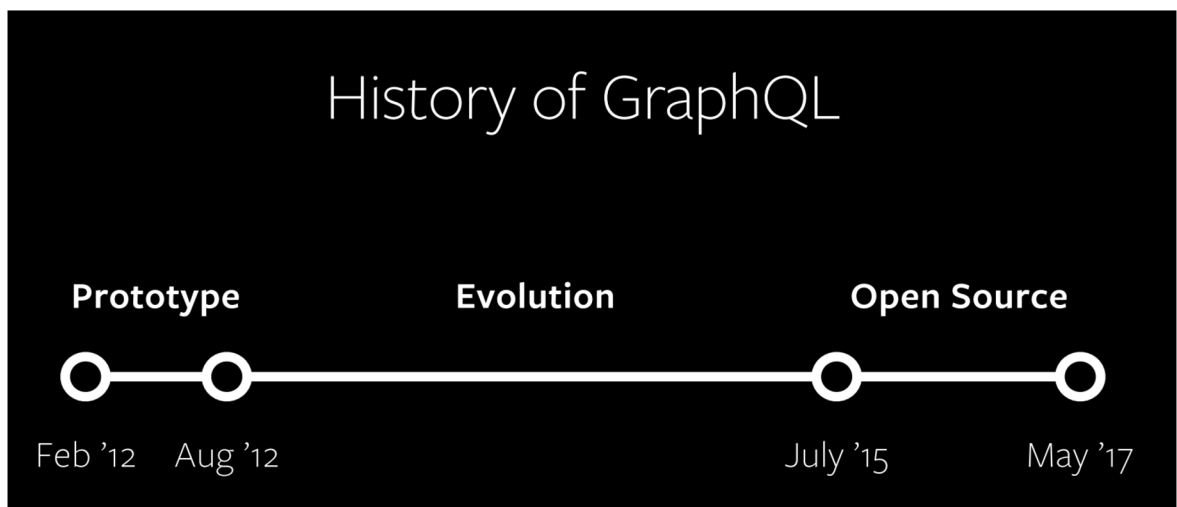


Figure3.3: GraphQL history[11]

### 3.2.2 Architectural overview

GraphQL describes the behaviour of the server, every aspect of data transport, such as how to request and respond should behave, the format of the data server and the client accept, written down and describe in GraphQL layer. The request made by the client is called QUERY. GraphQL also is a transport layer agnostic, which means, it can use in any platform or protocol. As document GraphQL server can be implemented in three ways:

#### a) GraphQL with direct DB connection

This is the simplest architecture, GraphQL Query request data and server read the query and response with data from the direct database. This process is also known as Resolve. The below Figure provides a basic understanding of direct DB connected

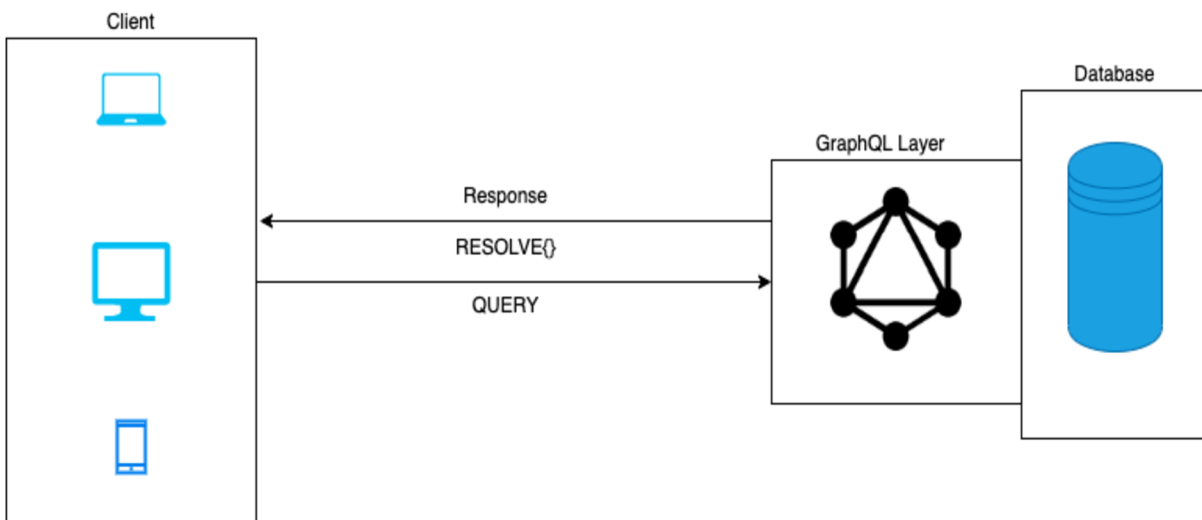


Figure 3.4: GraphQL direct DB diagram

### b) GraphQL on existing system

Usually, application with complex data flow which has a legacy system and very non-linear data architecture can be benefitted by this architecture. Usually, GraphQL servers behave as a layer on different service mesh, each service mesh can consist of microservices, DB, other APIs.

Client query will execute through GraphQL server and response data from the multi-source format as desire data structure via GraphQL server.

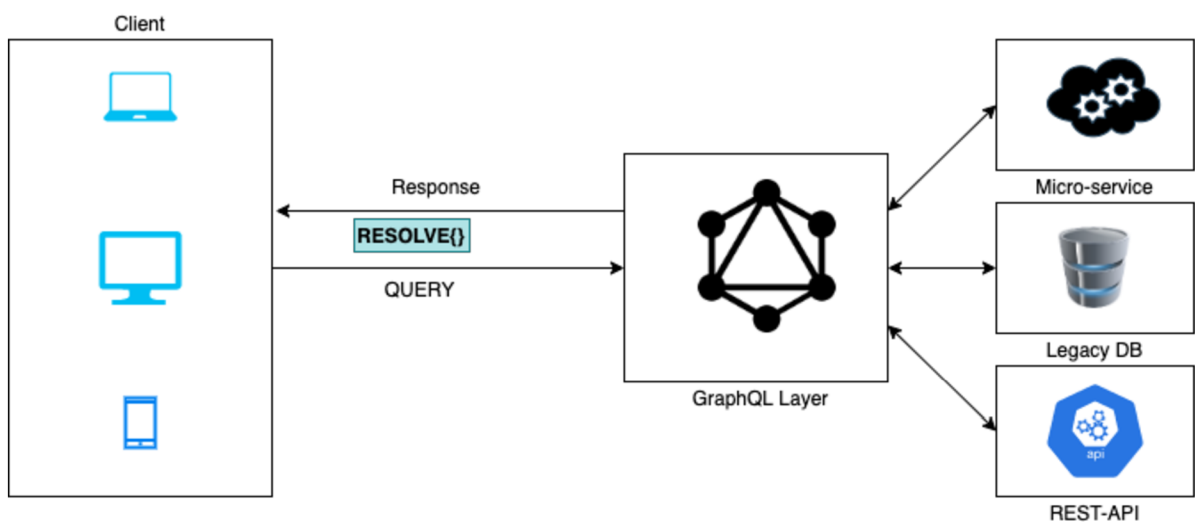


Figure 3.5: GraphQL legacy architecture

### c) Hybrid

This approach combined the previous two approaches. In this approach, the GraphQL server connects top on a DB, which connect to different data transport protocols.

The main challenge to overcome is data integrity.

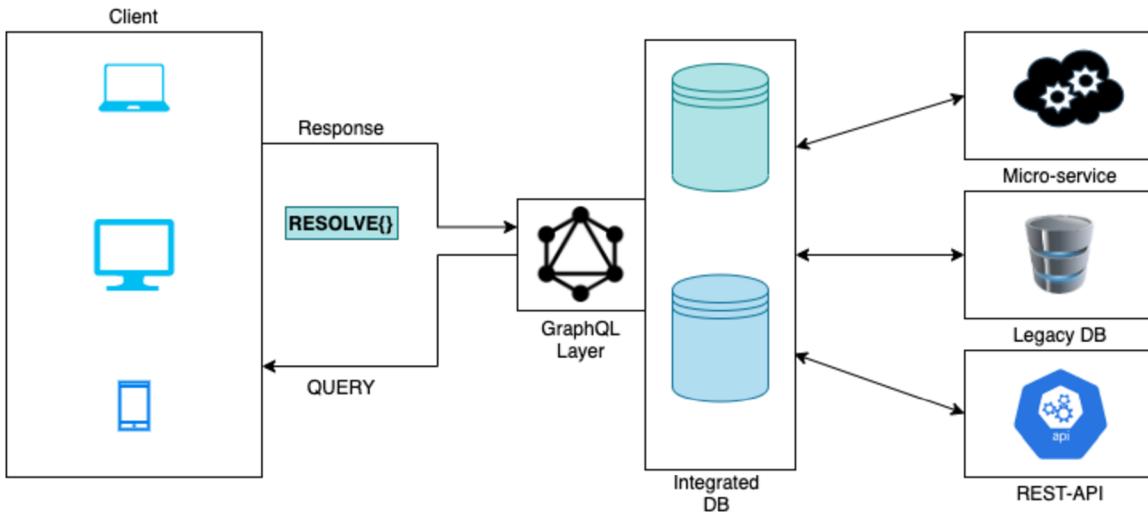


Figure 3.6: GraphQL hybrid architecture

### 3.2.3 Data Types

GraphQL as a language some very strong data types to define objects and response data structure. All 5 types serve different output and their implementation process also different from each other.

Below table represent different data types and there used case in GraphQL

Type	Example	Use case	Example
Scalar	Int, Float, String, Boolean, ID	Scalar can hold single value, so any usual implementation can have multiple scalar.	<i>name: String</i> <i>age: Int</i>
Object	object	Object type can holds other type and consist of multiple fields.	<i>type student {</i> <i>name: String</i> <i>age: Int}</i>
Query	Object	Query used for requesting data.	<i>type Query {</i> <i>name: String}</i>
Mutation	Object	Mutation type used for data manipulation in server, like create, update, delete	<i>type Mutation</i> <i>addStudent{</i> <i>name: String,</i> <i>age: Int}</i>
ENUM	Scalar	ENUM usually a scalar type, with multiple option value. An ENUM data type can consist multiple value but output only single value. ENUM define with snake_case.	<i>type city_select {</i> <i>PRAGUE</i> <i>WSHINGTON</i> <i>DELHI</i> <i>DHAKA</i> <i>}</i>
List	Array	List can hold array of scalar type value.	<i>type Query {</i> <i>students: [student]}</i>
Not-Nullable		This special character type can use to restrict null value in any field.	<i>type student {</i> <i>ID: Int!</i> <i>Name: String</i> <i>Age: Int</i> <i>}</i>

Table 3.3: GraphQL Data types

### 3.2.4 Queries

Queries are the connection layer of the GraphQL server. It is responsible for requesting data and responding with values. A successful query return JSON data and unsuccessful query return a different type of HTTP errors. A query can be executed single-threaded, which means the query can be called at one time. But by default GraphQL will aggregate queries into one.

In figure 3.2.5, left side both queries aggregate into a single query on the right side.

```
1 # Type queries into this side of the screen, and you will
2 # see intelligent typeaheads aware of the current GraphQL type schema,
3 # live syntax, and validation errors highlighted within the text.
4
5 # We'll get you started with a simple query showing your username!
6 query userDetails {
7   user(login:"RaufR"){
8     company
9     bio
10    email
11  }
12 }
13
14 query viewerDetails {
15   viewer {
16     avatarUrl
17     websiteUrl
18     gist: name
19   }
20 }
21 }
```

```
1 # Type queries into this side of the screen, and you will
2 # see intelligent typeaheads aware of the current GraphQL type schema,
3 # live syntax, and validation errors highlighted within the text.
4
5 # We'll get you started with a simple query showing your username!
6 query userDetails {
7   user(login:"RaufR"){
8     company
9     bio
10    email
11  }
12   viewer {
13     avatarUrl
14     websiteUrl
15     gist: name
16   }
17 }
18 }
```

Figure 3.7: GraphQL query aggregate example

Depends on need, queries can be customized. Query structure will define the response structure. A query can also have argument, in figure 3.2.5, the left side query *userDetails* has one argument, login, which define which user data we are requesting.

Another important piece of the query is Fragment.

Fragments are reusable piece of a query, sometimes same data request for different query than fragments are useful. Which reduce development time.



```

1 query userDetails {
2   user(login:"RauFR"){
3     ...data
4   }
5 }
6
7 query viewerDetails {
8   viewer {
9     ...data
10  }
11 }
12
13
14 fragment data on User {
15   avatarUrl
16   bio
17   gists{
18     totalCount
19   }
20 }

```

```

{
  "data": {
    "viewer": {
      "avatarUrl": "https://avatars2.githubusercontent.com/u/14043821?u=81ac5b02eccae79e4a8c3e3cdb281104e5df5f40&v=4",
      "bio": "I do JS",
      "gists": {
        "totalCount": 6
      }
    }
  }
}

```

Figure 3.8: Fragment example

### 3.2.5 Mutation

In GraphQL data manipulation can be done through mutation. Mutation defines procedures exactly similar to query with parameter. A mutation field should have a defined payload structure and not-nullable argument to find the exact record in data.

```

1 query getrepo {
2   repository(name:"protfolio2", owner:"RauFR"){
3     createdAt
4     description
5     sshUrl
6     url
7     id
8   }
9 }
10

```

```

{
  "data": {
    "repository": {
      "createdAt": "2018-10-07T13:05:05Z",
      "description": "Initial Repo description",
      "sshUrl": "git@github.com:RauFR/protfolio2.git",
      "url": "https://github.com/RauFR/protfolio2",
      "id": "MDEwO1JlcG9zaXRvcnkxNTE5NDIxOTE="
    }
  }
}

```

Figure 3.9: query for repository

In above example, we have executed a query for get data for a repository with fields of createdAt, description, sshUrl, url and id

In left side, GraphQL server return exact output for the query.

```

11 mutation changeRepo {
12   updateRepository(
13     input: {repositoryId: "MDEwOlJlcG9zaXRvcnkxNTE5NDIxOTE",
14       description: "Change description for test"}
15   )
16   {
17     repository {
18       description
19     }
20   }
21 }

```

```

{
  "data": {
    "updateRepository": {
      "repository": {
        "description": "Change description for test"
      }
    }
  }
}

```

Figure 3.10: mutation for change data

In the second step, we have written a mutation for update data which takes two arguments in the input field, repositoryId, and description. repositoryId is a mandatory argument but description is the field that contains the data we are going to change. We can put any other editable field as a second argument to change the value of that field.

After that, we define the exact field reference we need to update, in this case, it is description under repository. In GraphQL data manipulation can be done through mutation. Mutation defines procedures exactly similar to query with parameter. A mutation field should have a defined payload structure and not-nullable argument to find the exact record in data.

```

1 query getrepo {
2   repository(name: "protfolio2", owner: "RaufR"){
3     createdAt
4     description
5     sshUrl
6     url
7     id
8   }
9 }

```

```

{
  "data": {
    "repository": {
      "createdAt": "2018-10-07T13:05:05Z",
      "description": "Change description for test",
      "sshUrl": "git@github.com:RaufR/protfolio2.git",
      "url": "https://github.com/RaufR/protfolio2",
      "id": "MDEwOlJlcG9zaXRvcnkxNTE5NDIxOTE="
    }
  }
}

```

Figure 3.11: query after data update

At this time, we can observe the updated description field in query result, which propagate similar as previous mutation.

Another way to handle mutation is using query variable. But this not so much popular in common use due to JS syntax.

### 3.2.6 **Subscription**

After mutation, the third type of operation in GraphQL is subscription, it use for communicate real-time data update similar to web-socket. It is also a root type, and implement in a similar way of mutation.

GraphQL does not specify the data transport protocol, usually it use web-socket, but developer can use any like long-polling and e-mail.

### 3.2.7 **Introspection**

A most powerful feature of GraphQL, which also the documentation generate feature. For understanding and exploring any GraphQL server and queries option, developers have a powerful introspect query name `__schema`. Any given time, `__schema` query can be executed over a GraphQL server, and the developer doesn't have to define this query, this is auto-generated via GraphQL server itself.

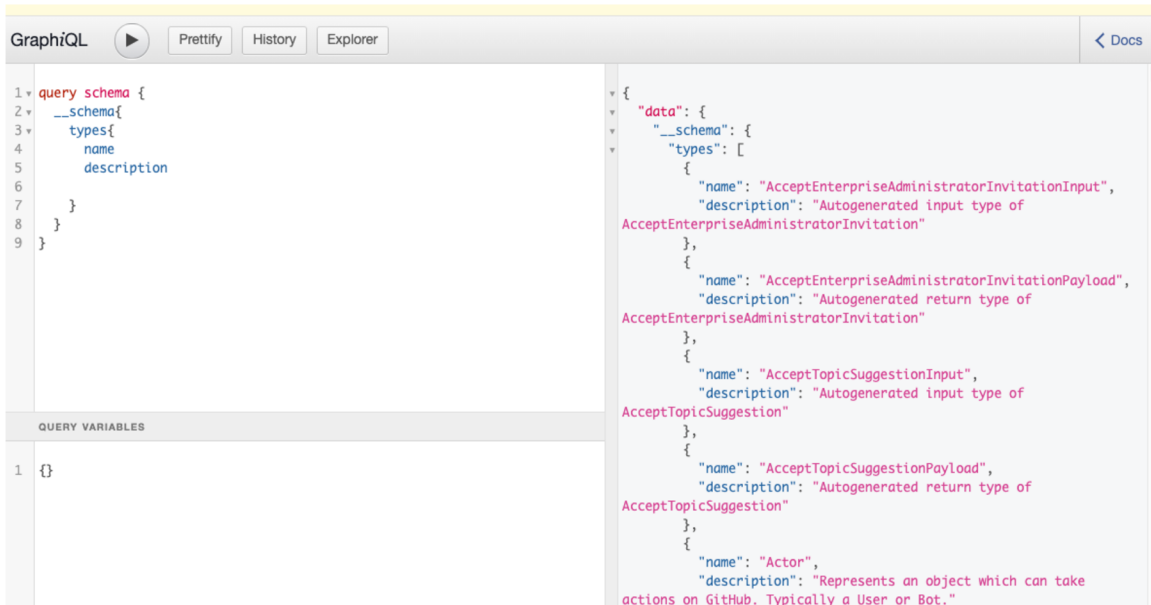


Figure 3.12: \_\_schema in github API

Above introspective query shows as all available query in this server with their description.

### 3.2.8 Schema

GraphQL schema is the core part of the runtime, every type and function should be described in schema before use. GraphQL runtime provides a graph-based schema for publishing and manages different data types and structures.

A schema in simple words can be called, collection of types. Schema architecture should be the focus on what data clients need to manage the runtime. Developers need to think about a collection of types that will be used in application development.

GraphQL provides a language to define a schema, called Schema Definition Language(SDL). SDL also a platform-independent language.

Types are the base of a schema. GraphQL types are a custom object. Which holds different attributes for that type. Like a student type can consist of name, age, studentId, etc.

Types can hold not-nullable, another types and array.

```

1 type Student {
2   id:ID!
3   name:String!
4   age: Int
5   department: String!
6   year:String
7 }

```

Figure 3.13: GraphQL schema type example

Developer can define custom scalar type for use within a type object. Also ENUM type can be define as a normal type which can be use in later type definition.

```

1 enum Photo_Category{
2   SMALL
3   LARGE
4   WIDE
5 }
6
7 type Photo {
8   id:ID!
9   url:String!
10  date: String
11  category:Photo_Category!
12 }
13
14
15 type Student {
16   id:ID!
17   name:String!
18   age: Int
19   department: String!
20   year:String
21   photo: Photo
22 }

```

Figure 3.14: Schema types

A types can return multiple type objects as list by putting [] around type field. Which return a JSON array as output. This feature call connection.

In GraphQL there are three type of connections.

1. **One-One**
2. **One-Many**
3. **Many-Many**

Similar with SQL each connect define with a type field which return result within type definition.

- **One-One connection**

In graph theory[10] a connection between two different types is called an edge, a connection between two linear types defined as a one-one connection.

```

type Student {
  name: String
  email:String
  id:ID!
  year:Int
}

type PaperSubmission {
  id:ID!
  name:String
  date:Date
  type:String
  owner:Student!
}

```

Figure 3.15: One to One connection

In figure 3.2.12 Student and PaperSubmission type have a one-one connection and the edge between them is the owner field.

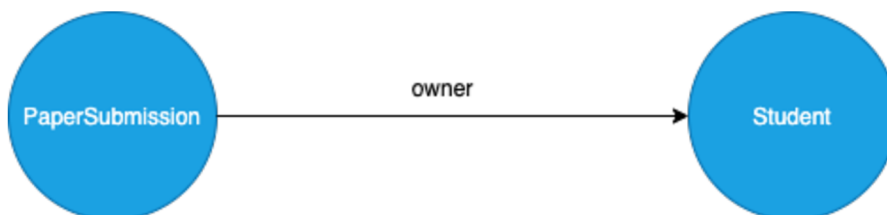


Figure 3.16: One to one connection data flow

One -One connection can represent data flow in linear direction and mutate them within linear approach.

- **One-Many Connection**

GraphQL provides flexibility to the user via providing unidirectional data fetching. Using the same analogy from the previous example, a One-Many connection represents as, if we query a student we should get all the submitter paper from that entity.

```
type Student {  
  name: String  
  email: String  
  id: ID!  
  year: Int  
  submission: [PaperSubmission]  
}
```

Figure 3.17: One to many connection query

For achieving that, we need to add an extra field submission to the Student type. In this way, we can get multiple edges in the same data query.

- **Many-Many Connection**

Often, the application layer needs multiple edges between data, with our previous analogy, an exam can have multiple students and a student can be part of multiple exams. Figure 3.18 represents a many-to-many connection diagram.

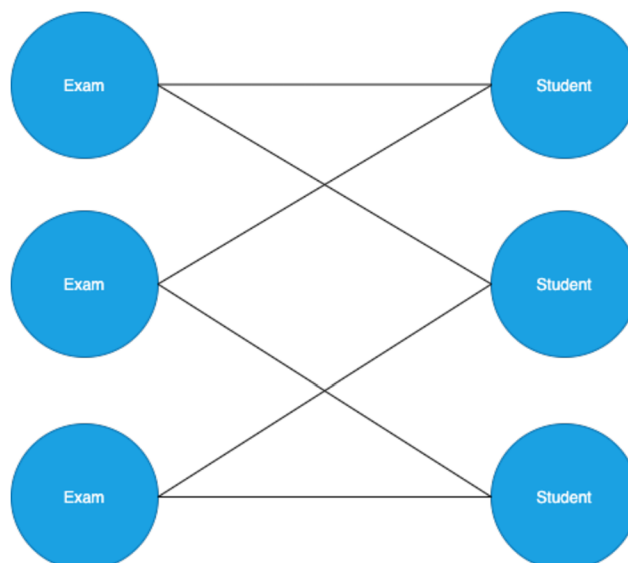


Figure 3.18: Many to Many connection

### 3.3 Result

As I am progressing through the theoretical explanation, it is become clearer that graphql holding an edge over technology and approach. REST architectures intend to serve the best from SOAP world, but the ROA based resource allocation tend toward over data flow, which can be uncontrolled if the DBA did not architect properly from the beginning.

There can be a very extensive difference between them if we compare the approach and integration capability.

In one side we have full framework like REST which provides, structure way to API architecture and resource-based data distribution. Another side graphql is more like a middleware top on the API layer, capability of integrate with any data layer and flexibility of data i/o operation.

In the context of this thesis, my best assumption was checking multiple tools available within opensource platforms. Within a multiple testcase I have decided to choose handful tools, in the choosing parameters, consideration was, ease of use, documentation, adoption and knowledge curve.

The important tool is Apollo client, which is the most sophisticated graphql framework available in current tech stack.

Another important decision was writing the code completely in JavaScript, for the simplicity and linear codebase I decide not use Django(Python based framework) which provides much strong typed and structured approach.

It possible due to mature development cycle of various JavaScript runtime and frameworks like NodeJs, ExpressJs.



Further working opportunity for the theoretical part can be data aggregation and manipulation within different protocols like HTTP, WebSocket. And understand how GrsphQL can be deployed within them.

## 4 Practical Part

This thesis goal is not only to define the difference between GraphQL and REST but also, show a real-world used case scenario with a live server. Which demonstrates the technical implementation and benchmarking.

### 4.1 Tools

For tools selection, my focus was sophistication and redundant less solution. After careful consideration and multiple used-case implementations. I chose the following tools.

#### 4.1.1 Apollo

For simplified GraphQL implementation, there is no other alternative than Apollo clients.

<https://www.apollographql.com/>

I choose this solution because of their vast solution and stable platform performance with huge community support. GraphQL alone is a query language, but there is no structured way for implementation, which introduces so many different implementation examples, it is not a good maintainable approach. Apollo solves that issue, and it wraps the GraphQL in a way that the end developer can handle both server-side and client-side data fetching simultaneously.

Apollo includes various components like Apollo-Server, IDE-Plugin, Apollo-CLI to reduce clutter in the platform.

Another important feature Apollo provides is data federation. Granular microservice data fetching is not the easy implementation in data architecture point, Apollo makes this effortless.

#### 4.1.2 **Postman**

Postman is an API development platform, provides handful API development and maintenance tool. In this thesis we will comprehensively use Postman client to debug and test our API endpoints and there performance.

<https://www.postman.com/product/api-client/>

#### 4.1.3 **Github**

Git is a source code maintenance technology. Vendor like GitHub, Bitbucket, GitLab provides a platform to use this technology.

Git usually a code history mapping technology which works with map each push/commit/merge with a specific tracking number, which can be use in future reference.

Also mentionable features like, rebase, forking.

All of the source code in this thesis has been published and maintained in the GitHub repository. This repository is publicly available so any collaboration can be done without any authorization.

<https://github.com/RaufR/RestGraphComparison>

#### 4.1.4 **Visual Studio Code**

Lightweight IDE for coding and initial debugging. Opensource solution from Microsoft and very user friendly. Can be download and use any platform.

<https://code.visualstudio.com/>

#### 4.1.5 Technologies

For this thesis, I have aligned the latest technology for encouraging new development. A stable and maintainable solution was my priority. All of the technologies can be found in various `package.json` files in the codebase. Some notable example:

- NodeJS
- ExpressJS
- Webpack
- GraphQL

## 4.2 Practical setup

### 4.2.1 Source code

The source code structure is tree-defined multi-tier architecture. All source code consists of a single folder with multiple main folders, this approach makes source code modular. In future use, we can deploy any given application part in any platform with link updates. Server define in two different folders:

- */GraphQLServer*
- */ExpressServer*

Each folder contains its package definition in the *package.json* file, this file defines the necessary dependencies libraries and execution entry point for the server. As an example in figure 4.1 contains a JSON object file with *root*, *scripts*, *dependencies*, and *nodemonConfig* levels.

Each server entry point runs autonomously which enables a multi-thread operation process, with ad-hoc requirements this code can be deployed to multiple PaaS solutions with or without a containerization environment.

Server run time execution is done on `npm`[12] package terminal with command

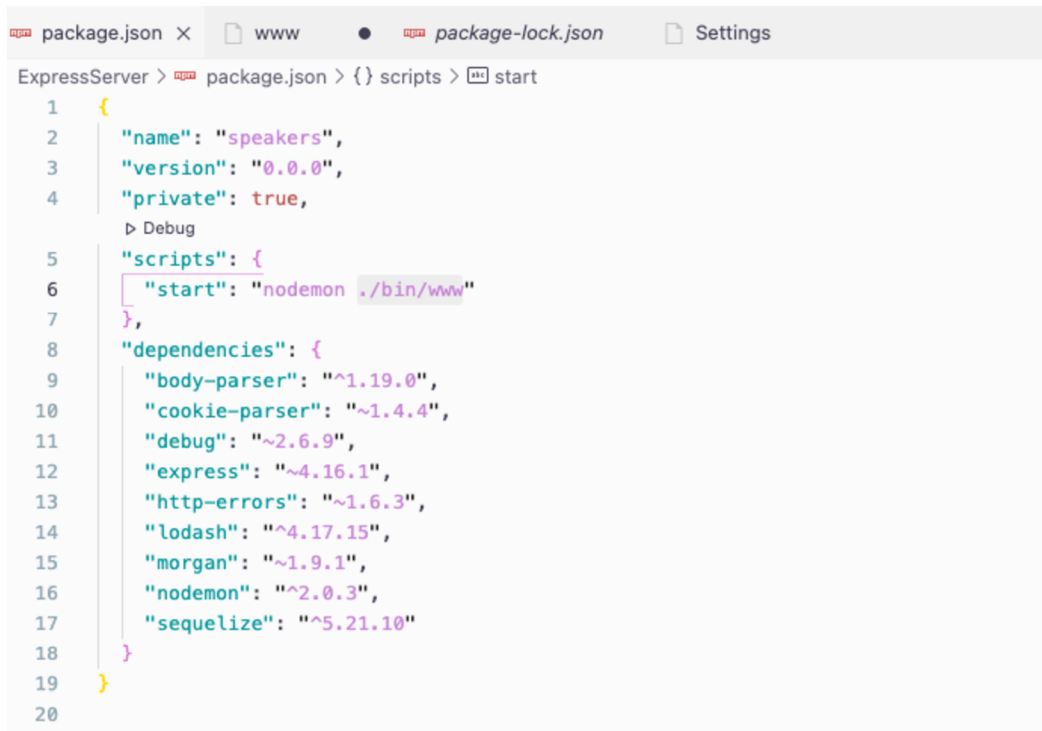
```
"npm start"
```

which triggers nodemon to index.js file and app.js file into a different location and different thread.

Both servers can be run concurrently.

REST server does not have dependencies on GraphQL server for any data I/O operation.

GraphQL server certainly fetches a portion of the data object from running the express server. This is by design to settle much more operational costs to the GraphQL server.



```
package.json x  www  package-lock.json  Settings
ExpressServer > package.json > {} scripts > start
1  {
2    "name": "speakers",
3    "version": "0.0.0",
4    "private": true,
5    "scripts": {
6      "start": "nodemon ./bin/www"
7    },
8    "dependencies": {
9      "body-parser": "^1.19.0",
10     "cookie-parser": "~1.4.4",
11     "debug": "~2.6.9",
12     "express": "~4.16.1",
13     "http-errors": "~1.6.3",
14     "lodash": "^4.17.15",
15     "morgan": "~1.9.1",
16     "nodemon": "^2.0.3",
17     "sequelize": "^5.21.10"
18   }
19 }
20
```

Figure 4.1: package entry file

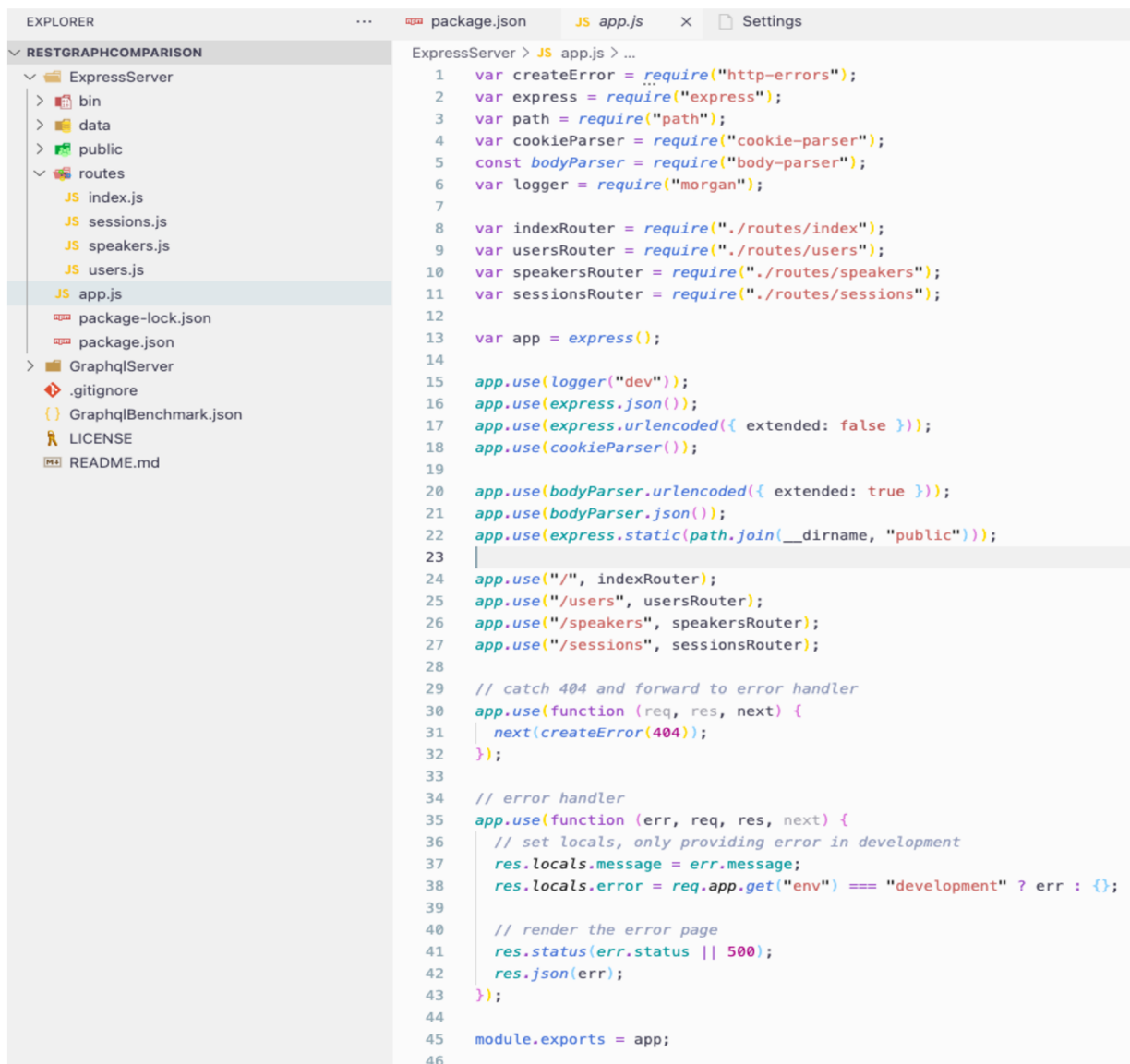
One level up to the root, node\_modules folders hold necessary libraries for executing runtime and build time.

#### 4.2.2 REST Server

For REST implementation I have choose ExpressJs, which is a framework for NodeJs platform and a very popular server side framework. Theoretical discussion contains more arguments and details regarding ExpressJs.

Figure 4.2 have the code snippet of ExpressJs server entry code. From line 1-11 specifically calling other dependencies, line 15-27 assign the dependencies to Express module, line 24-27 contain different route address for the server, routes are used as identifier which support ROA pattern. Each route have specific data exposure for specific resource.

Line 30-45, error handler for serve requests.



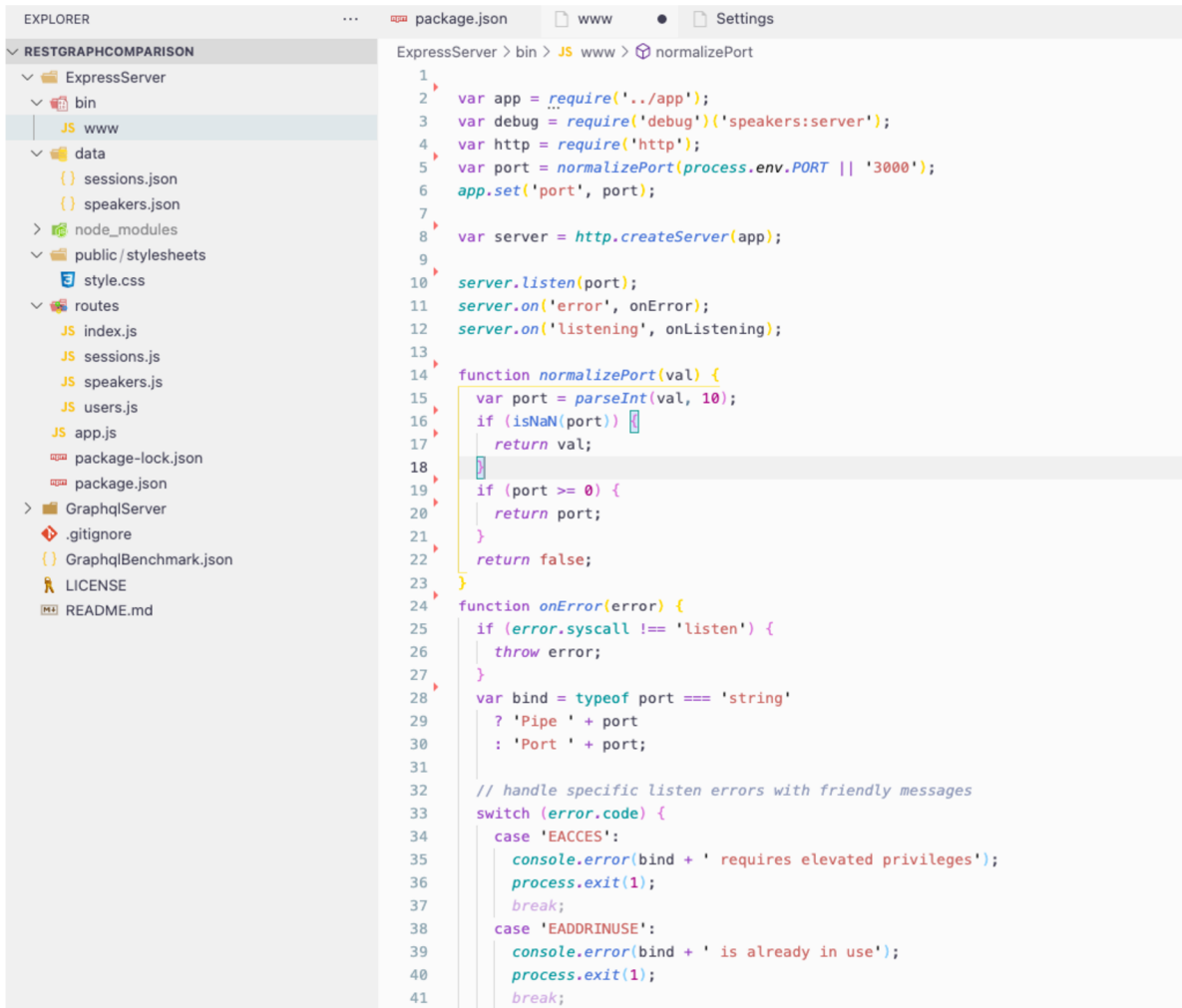
```
EXPLOSER package.json JS app.js X Settings
RESTGRAPHCOMPARISON
  ExpressServer
    bin
    data
    public
    routes
      JS index.js
      JS sessions.js
      JS speakers.js
      JS users.js
      JS app.js
      package-lock.json
      package.json
    GraphQLServer
    .gitignore
    GraphQLBenchmark.json
    LICENSE
    README.md

ExpressServer > JS app.js > ...
1  var createError = require("http-errors");
2  var express = require("express");
3  var path = require("path");
4  var cookieParser = require("cookie-parser");
5  const bodyParser = require("body-parser");
6  var logger = require("morgan");
7
8  var indexRouter = require("./routes/index");
9  var usersRouter = require("./routes/users");
10 var speakersRouter = require("./routes/speakers");
11 var sessionsRouter = require("./routes/sessions");
12
13 var app = express();
14
15 app.use(logger("dev"));
16 app.use(express.json());
17 app.use(express.urlencoded({ extended: false }));
18 app.use(cookieParser());
19
20 app.use(bodyParser.urlencoded({ extended: true }));
21 app.use(bodyParser.json());
22 app.use(express.static(path.join(__dirname, "public")));
23
24 app.use("/", indexRouter);
25 app.use("/users", usersRouter);
26 app.use("/speakers", speakersRouter);
27 app.use("/sessions", sessionsRouter);
28
29 // catch 404 and forward to error handler
30 app.use(function (req, res, next) {
31   next(createError(404));
32 });
33
34 // error handler
35 app.use(function (err, req, res, next) {
36   // set locals, only providing error in development
37   res.locals.message = err.message;
38   res.locals.error = req.app.get("env") === "development" ? err : {};
39
40   // render the error page
41   res.status(err.status || 500);
42   res.json(err);
43 });
44
45 module.exports = app;
46
```

Figure 4.2: ExpressJs server creation

This file use as a abstraction to the *bin/www.js* file, which allocate port and trigger

Server run. Figure 4.3 include the ww.js file snippet, line 8-12 create and run the server on port 3000.



```
1
2 var app = require('../app');
3 var debug = require('debug')('speakers:server');
4 var http = require('http');
5 var port = normalizePort(process.env.PORT || '3000');
6 app.set('port', port);
7
8 var server = http.createServer(app);
9
10 server.listen(port);
11 server.on('error', onError);
12 server.on('listening', onListening);
13
14 function normalizePort(val) {
15   var port = parseInt(val, 10);
16   if (isNaN(port))
17     return val;
18
19   if (port >= 0) {
20     return port;
21   }
22   return false;
23 }
24
25 function onError(error) {
26   if (error.syscall !== 'listen') {
27     throw error;
28   }
29
30   var bind = typeof port === 'string'
31     ? 'Pipe ' + port
32     : 'Port ' + port;
33
34   // handle specific listen errors with friendly messages
35   switch (error.code) {
36     case 'EACCES':
37       console.error(bind + ' requires elevated privileges');
38       process.exit(1);
39       break;
40     case 'EADDRINUSE':
41       console.error(bind + ' is already in use');
42       process.exit(1);
43       break;
```

Figure 4.3: express 4.0 bin file

Express server expose the data within some routes to accessibility, all router must be included within server implementation, which responsible to create different URI and send related data to that route. As such

/users routes hold all the user data

/user/{:id} route take id as parameter and map only specific user data.

ExpressJs have its own module to handle this process.

Figure 4.5 snippet show a common implementation of router, which include get, patch and post http requests.

```
ExpressServer > routes > JS sessions.js > ...
1  var express = require("express");
2  var router = express.Router();
3  var sessions = require("../data/sessions.json");
4  var _ = require("lodash");
5
6  router.use("/:id", (req, res, next) => {
7    let session = _.filter(sessions, { id: req.params.id })[0];
8    if (session) {
9      req.session = session;
10     return next();
11   }
12   return res.sendStatus(404);
13 });
14 /* GET users listing. */
15 router.get("/", function (req, res, next) {
16   res.json(sessions);
17 });
18
19 router.post("/", function (req, res, next) {
20   res.json({ message: "not implemented" });
21 });
22
23 router.patch("/:id", function (req, res, next) {
24   Object.entries(req.body).forEach((item) => {
25     const key = item[0];
26     const value = item[1];
27     req.session[key] = value;
28     console.log(req.session);
29   });
30 });
```

Figure 4.4: Express router example

### 4.2.3 GraphQL Server

GraphQL server also has a similar code structure as the express server, entry script defined in package.json file and app has node-based runtime. The main difference is in the implementation of the server.

Figure 4.4 snippet explains the implementation for a GraphQL server. Which import resolvers, schema, and data sources in lines 1-12. Define the server and trigger the server executed in lines 14-18.

```
GraphQLServer > JS index.js > ...
1  const { ApolloServer, gql } = require("apollo-server");
2  const SessionAPI = require("../dataSources/sessions");
3  const SpeakerAPI = require("../dataSources/speakers");
4
5  const typeDefs = require("../schema");
6
7  const resolvers = require("../resolvers");
8
9  const dataSources = () => ({
10   sessionAPI: new SessionAPI(),
11   speakerAPI: new SpeakerAPI(),
12 });
13
14 const server = new ApolloServer({ typeDefs, resolvers, dataSources });
15
16 server.listen({ port: process.env.PORT || 4000 }).then((url) => {
17   console.log(`GraphQL is running" at ${url}`);
18 });
19
```

Figure 4.5: GraphQL server config

Very important part of the server is the schema, which defines the data type and pattern for the mutation and resolvers to works with. Schema usually define the architecture of data objects. As GraphQL more focus over data object throught the query/mutation resolvers. Schema defines how the query will look like. Fire 4.5 have the overview of the schema inside GraphQL server.

<https://github.com/RaufR/RestGraphComparison/blob/main/GraphQLServer/schema.js>

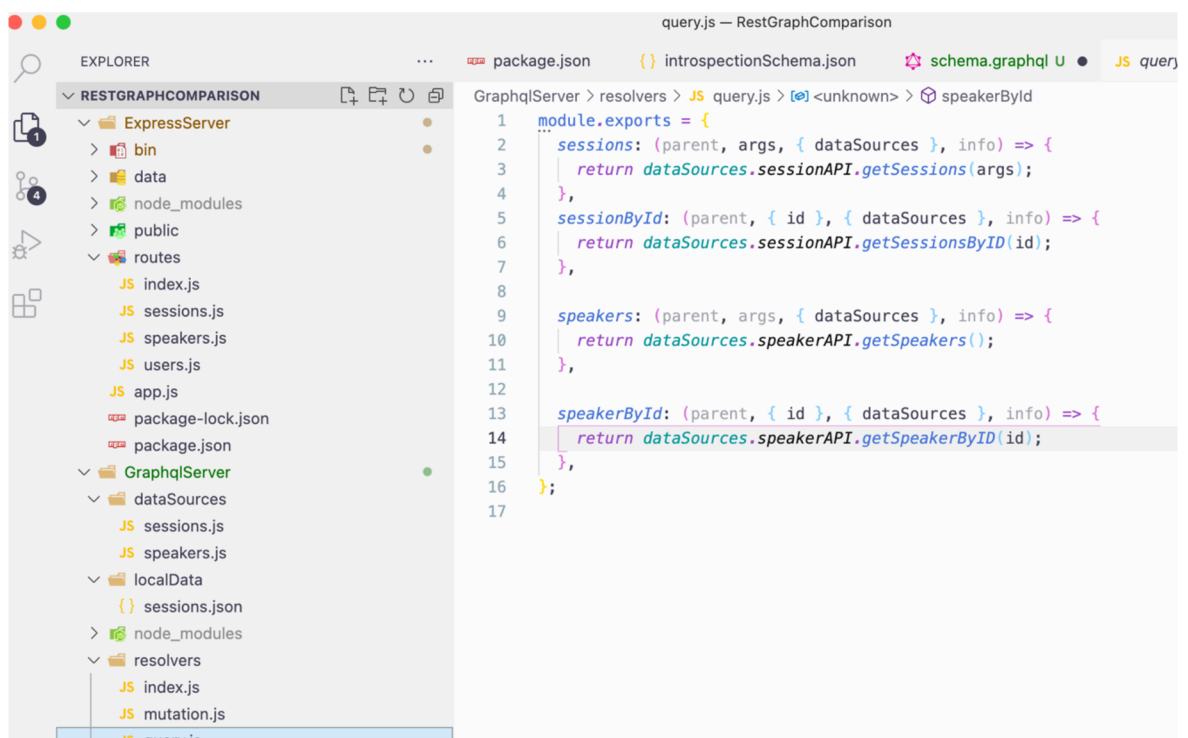
```
GraphQLServer > JS schema.js > ...
1  const { gql } = require("apollo-server");
2
3  module.exports = gql`
4    type Query {
5      sessions(
6        id: ID
7        title: String
8        description: String
9        startsAt: String
10       endsAt: String
11       room: Room
12       day: String
13       format: String
14       track: String
15       level: String
16     ): [Session]
17     sessionById(id: ID): Session
18     speakers: [Speaker]
19     speakerById(id: ID): Speaker
20   }
21
22   enum Room {
23     Europa
24     Sol
25     Saturn
26   }
27
28   type Mutation {
29     toggleFavoriteSession(id: ID!): Session
30     addNewSession(session: SessionInput): Session
31   }
32
33   input SessionInput {
```

Figure 4.6: GraphQL schema



GraphQL server depends on the resolvers to handle data manipulation, format and mutation.

Each field in Query is a function which takes a type as parameter and return another type. This functional operation done by resolvers. When a field executed, the corresponding resolver return next value. Figure 4.5 represents resolvers for query operations. 4 different queries take datasource and id as parameter and returning four different query callback objects.



The screenshot shows an IDE window titled 'query.js — RestGraphComparison'. On the left is an Explorer pane showing a project structure with folders like 'ExpressServer', 'GraphQLServer', and 'resolvers'. The 'resolvers' folder is expanded, showing 'index.js', 'mutation.js', and 'querv.is'. The main editor displays the code for 'speakerById' in 'query.js'. The code defines a 'module.exports' object with four resolver functions: 'sessions', 'sessionById', 'speakers', and 'speakerById'. Each function takes 'parent', 'args', 'dataSources', and 'info' as parameters and returns a result from a corresponding API method.

```
1 module.exports = {
2   sessions: (parent, args, { dataSources }, info) => {
3     return dataSources.sessionAPI.getSessions(args);
4   },
5   sessionById: (parent, { id }, { dataSources }, info) => {
6     return dataSources.sessionAPI.getSessionsByID(id);
7   },
8
9   speakers: (parent, args, { dataSources }, info) => {
10    return dataSources.speakerAPI.getSpeakers();
11  },
12
13  speakerById: (parent, { id }, { dataSources }, info) => {
14    return dataSources.speakerAPI.getSpeakerByID(id);
15  },
16 };
17
```

Figure 4.7 : GraphQL query resolvers

Resolvers for this thesis implement to the public repo for further work.

<https://github.com/RaufR/RestGraphComparison/tree/main/GraphQLServer/resolvers>

GraphQL server has 2 different data sources one is local file another data source is the express server which running on localhost:300

This integration have done via apollo-datasource-rest library, which is a very sophisticated wrapper function around the rest json object. Figure 4.6 has the coding implementation of this integration.

```
GraphQLServer > dataSources > JS speakers.js > ...
1  const { RESTDataSource } = require("apollo-datasource-rest");
2
3  class SpeakerAPI extends RESTDataSource {
4    constructor() {
5      super();
6      this.baseUrl = "http://localhost:3000/speakers";
7    }
8
9    async getSpeakers() {
10     const data = await this.get("/");
11     return data;
12   }
13
14   async getSpeakerByID(id) {
15     const data = await this.getSpeakerByID(`/${id}`);
16     return data;
17   }
18 }
19
20 module.exports = SpeakerAPI;
21
```

Figure 4.8: REST integration

#### 4.2.4 Test Platform

The main objective of this thesis consists 4 different methods:

- Implementation
- Speed
- Usability
- Maintainability

Implementation tools are common IDE and public repository with enough authorization granularity.

For speed benchmark, all of them test against the local development environment. Which have the below attributes.

Model	Apple Notebook 2015 pro 13 inch
RAM	8GB DDR3
Processor	Intel core i5 2.7 GHZ
Operating system	macOS Big Sur 11.2.1

Table 4.1: Test machine's attributes

I used Postman API client. I have defined and share one workspace with multiple endpoints. With no other env-setup or external server setup, both of the run in local env on http.

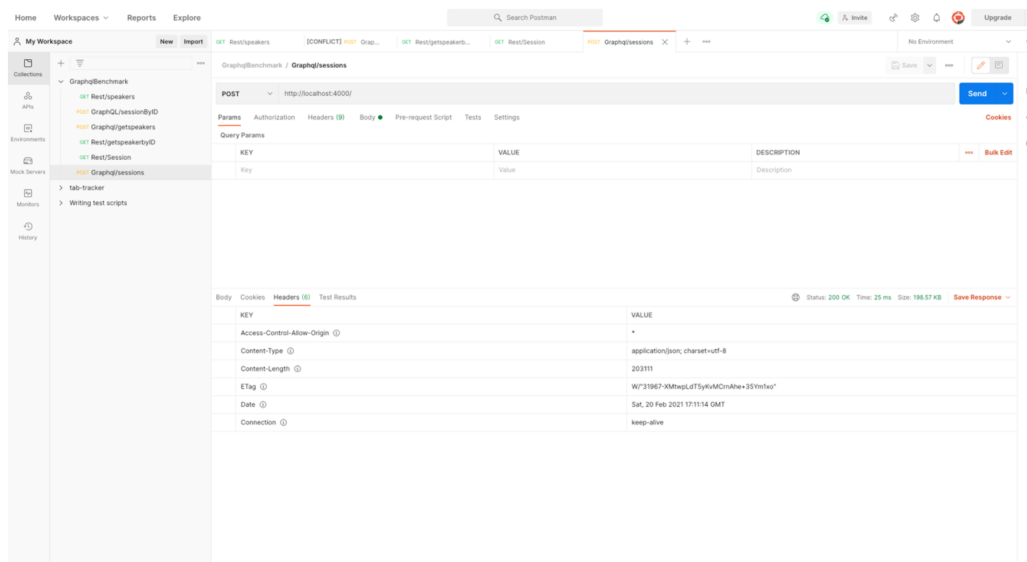


Figure 4.9: Postman interface

With this approach, our test cases are direct and fewer setup steps.

Usability used cases define as standard SDLC used case scenario and I have used Postman to demonstrate data fetching levels.

For maintainability demonstration, an abstract software development team scenario have used.

#### 4.2.5 Data

For the speed test, seed data. Have collected from public repository. Data are mostly by relational and integrated within a single object value. For external data sources both data REST and GraphQL have similarity and common architecture

```
1  [
2    {
3      "id": 84473,
4      "title": "Secure Programming for the Enterprise",
5      "description": "Est sunt nostrud officia fugiat sunt reprehenderit cupidatat. Et incid",
6      "startsAt": "8:00",
7      "endsAt": "5:00",
8      "speakers": [
9        {
10       "id": "2bda8276-b7b6-4653-a7c5-1bcc59d11a49",
11       "name": "Jean Ryan"
12     }
13   ],
14   "room": "Europa",
15   "day": "Wednesday",
16   "format": "FullDay Workshop",
17   "track": ".NET",
18   "level": "Intermediate"
19 },
```

Figure 4.10: Session data example

Data consist main to segments session and speakers. Session data include the details of the session and another level depth object which contains the pointers toward speakers with a reference id and name.

This data response over both server within a high density recursive request. Below link contains the full data source.

<https://github.com/RaufR/RestGraphComparison/blob/main/ExpressServer/data/sessions.js>  
[on](#)

Another data set is the speakers data which use mostly GraphQL layer for integration purpose, In figure 4.6 speaker data consist with attribute related with individual speaker personal information. With unique identifier.

Speaker details initially render within ExpressJs layer and then integrate with GraphQL layer with a map call back function which return granular speaker details with related session.

Speaker data can be seed from below link:

<https://github.com/RaufR/RestGraphComparison/blob/main/ExpressServer/data/speakers.json>

```
1
2
3   "id": "381b010e-f51d-4fca-a249-271f72a6a5b9",
4   "bio": "Anim anim cupidatat cupidatat consequat nisi mollit velit officia nulla et eiusm
5   "sessions": [
6     {
7       "id": 129718,
8       "name": "Batching Vs Streaming"
9     }
10  ],
11  "name": "Macey Duncan"
12 },
13 {
14   "id": "c5e306ae-3f40-4ff0-92a2-503e2f1dc6a1",
15   "bio": "Ex amet elit anim qui consectetur fugiat consequat dolor occaecat. Pariatur eu d
16   "sessions": [
17     {
18       "id": 85318,
19       "name": "Azure you want to use AWS IoT?"
20     }
21  ],
22  "name": "Jerome Parker"
23 },
```

Figure 4.11: Speaker data example

## 4.3 Workflow

I came across several solutions to implement the workflow, which consists, cloud test, SSR, or third-party testbed. After careful consideration and completing the benchmark step, I conclude the workflow step as below

### 4.3.1 Initial

- Install nodeJS in local machine. Latest LTS version should be enough.
- Download and install GIT

- Clone public repository from <https://github.com/RaufR/RestGraphComparison>

#### 4.3.2 Express Server

- Navigate to *./Expressserver* folder
- Run *npm install* in terminal and then run *npm start*
- Successful execution should start an express server in <http://localhost:3000/>

#### 4.3.3 GraphQL Server

- Navigate to *./GraphqlServer* folder
- Run *npm install* in terminal and then run *npm start*
- Successful execution should start an graphql-playground server in <http://localhost:4000/>

#### 4.3.4 Postman Setup

- Download and install Postman Api client
- Sign and open
- Click on import on left side bar
- Upload *GraphqlBenchmark.json* file, which can be found in repository.

All above workflow must be done before any benchmark step.

#### 4.3.5 Testing parameters

Before reaching out to the final conclusion, I have decided to granular test parameters to relatively linear.

Testing in Postman requires no specific setup on its own but to make this test more aligned with real-time usage I have set network throttle to mid-low, which provides a higher response time for each request.

Postman also provides specific recursion mechanism to an URI which will be extensively used in a

Some parameters I have to consider especially in our TestBed.

- Bandwidth(generate via network throttle)
- Processing time
- CPU usage
- Memory usage

## 5 Results and Discussion

### 5.1 Implementation

Implementation comparison can be defined by the complexity and learning curve between Rest and GraphQL architecture.

Both of the parameters contain very dynamics attributes and differ on developer experience and exposure towards aligned technologies.

From my personal perspective, both of the architecture have positive and negative sides when it comes to implementation steps.

#### **REST**

- Structured architecture. Developer with strong CRUD experience can easily adopt, also this introduce heavily depends on similar architecture and rigid data manipulation step.
- Widely used, examples are more available.
- Strong community support, helped to smooth the learning curve.
- Codebase moderately followed years long MVC architecture, easy to understand.
- Client have very low control over data fetching, resource output size defined via server side, and without calling whole response client can not do any other calculation, which push runtime higher.
- More granularity means more endpoints, which become very complex in heavy data manipulation.

- Middleware connectivity and usage of multiple data source is possible but introduce more complexity within codebase.

## GraphQL

- Non-linear flexible architecture, developers with functional programming experience adopt easily. Which can be overwhelming for OOP focus developers.
- Newer technology, much more strong prospectus, have support from big-techs.
- Well-written documentation, fewer but very defined tutorials, learning curve can be smooth but comparatively still higher.
- Very flexible data fetching from client side, don't need to write whole endpoint for each type of fetching. Client have full control of how much data needed. Which reduce runtime significantly.
- Support multiple languages and modern frameworks. Easy to adopt within different codebase.
- Very easy and native support for multiple data connector.

Implementation learning curve higher for GraphQL also implementation flexibility is much higher in GraphQL. While REST force developer to maintain the structure and give zero control to the client, GraphQL gives total flexibility over implementation and response size.

I have come to the final conclusion that GraphQL is a much more sophisticated solution when the developer is experienced, and REST will be very developer-friendly at the beginning but a rigid CRUD system can be a big issue in future development, I have discussed more in maintainability section.



## 5.2 Speed

As the previous discussion define, GraphQL has some clear edges over REST. Also, data properties and multiple foreign key connections within data show more clear sign of fetch time difference.

Gradual data load increase, our graphql server show performance downgrade but hold the key parameters when the data have multiple levels of depth.

The testbed was predefined; the initial data connector contain local preload data, which contains several object layers, and both datasets integrated.

The endpoints were implemented with specific resources and URI from REST API divided into two URI, speakers and speakerById. Both sections have different payload properties and different exposure URLs.

From GraphQL, there are several depth layers sessions; the session includes a speaker, and the speaker includes multiple sessions.

For demonstration more details, two HTTP fetch operations on both API with a similar data payload create a fair comparison point.

In first fetch operation, data payload identical in both API and there is no integration between both API.

I used 0ms throttle, payload of 131382 with 2 level depth connection for each request. I have run 5 iteration of the test. As result show, GraphQL have 23ms to 33ms range and Rest has 39ms 5ms range.

Rest is the winner in lowest data fetch result but runtime consistence is not there, while GraphQL show clear sign of runtime declaration over each run. I would also identify that REST use server cache policy more often than GraphQL. Which probably the lower the time cost in continues data fetching operation. Hence both server can be modify to use cache as much operation needed.

Throttle(ms)	Payload	Depth	Rest		GraphQL	
			Time(ms)	Size(kb)	Time(ms)	Size(kb)
0	131382	2	39	131.601	33	131.601
			19		23	
			5		30	
			35		29	
			35		27	

Table 5.1: Test case 1 data

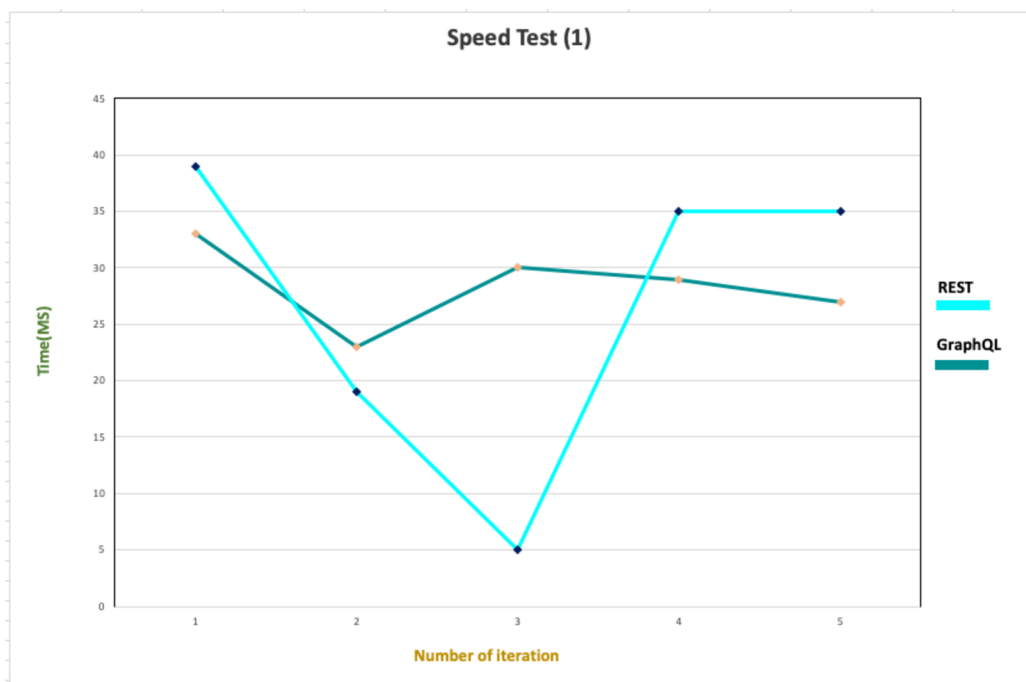


Figure 5.1: Speed test chart

For the second operation, data layers are integrated into a singular point. This fetch call executes from the GraphQL layer and connects to REST with GraphQL. GraphQL layer passes session.id to REST middleware and fetches a specific speaker with the given session.id param.

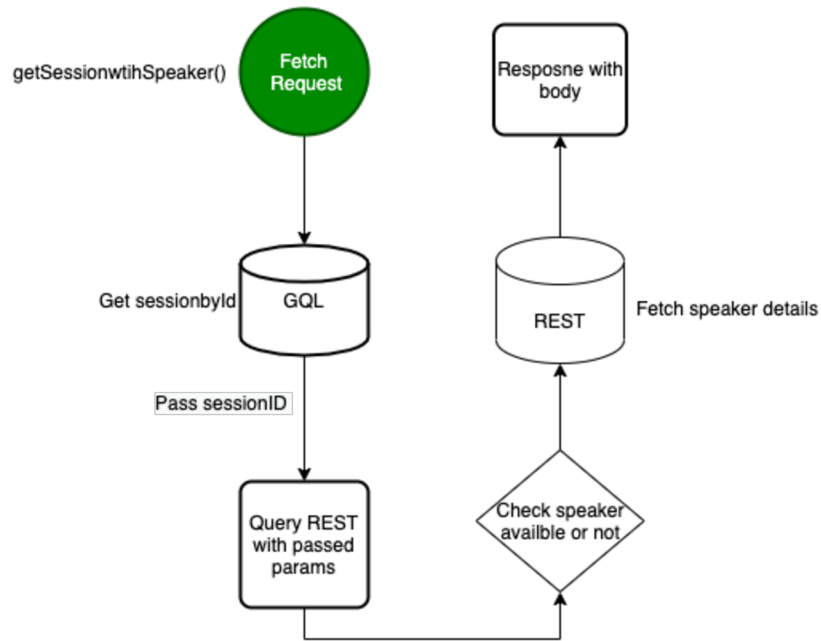


Figure 5.2: Data flow from GraphQL to REST

This test's expectation is needed to justify the GraphQL layer requesting data from the REST layer, but the REST layer does not. As both data have 90% similar attributes so for the comparison, deducted REST fetch time from GraphQL fetch time and observed.

Opertion speed counting formula looks like:

$$\text{Total response time} - \text{REST response time} = \text{GraphQL response time}$$

For this operation I used 10ms throttle, payload of 1 with 4 level depth connection for each request. I have run 5 iteration of the test. As result show, GraphQL have 25ms to 21ms range and Rest has 19ms to 15ms range.

Throttle(ms)	Payload	Depth	Rest		GraphQL	
			Time(ms)	Size(kb)	Time(ms)	Size(kb)
10	1	4	25	1.33	17	1.33
			24		17	
			21		19	
			21		15	
			24		17	

Table 5.2: Test case 2 data

Test Result in chart overview.

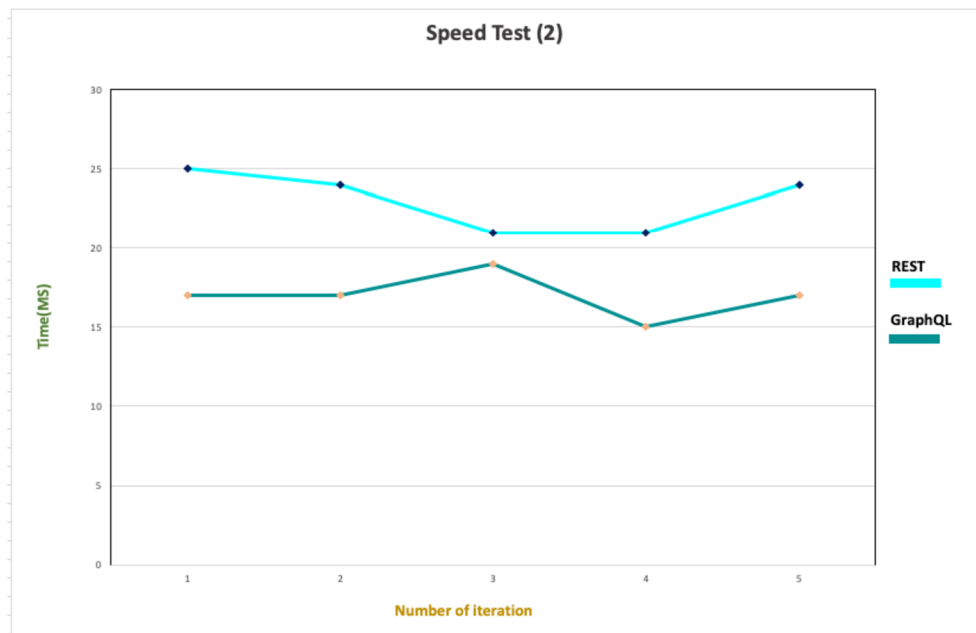


Figure 5.3: Speed test 2 chart

Second test case pattern is similar between two set. GraphQL pattern have range between 15-19ms and REST pattern 21-25ms. Due to similar data depth and linear connectivity GraphQL show a potential 4ms threshold with REST.

Our test cases produce a very clear judgement which lean toward GraphQL speed execution. The main reason is caching capability and linear JSON object based data manifestation. Both test cases have significant difference between payload to understand the both technologies capability with workload variety.

Future working opportunity and enhancement can be done with a live server in a PaaS solution within different protocol to understand the much more complex test cases.

Also Authentication is another very strong point which can be tested.

One of the objective of this thesis was test the execution speed, and I can abbreviate that GraphQL and REST both has very similar execution speed on low payload, with complex multiple depth data fetch operation GraphQL is bit faster.

### 5.3 Usability

This term reflects the usage of different architecture in different SDLC. Usability traces the amount of granularity of data manipulation inside an implemented API and also measures the total number of application support toward API lifecycle.

As a comparison of granular data manipulation GraphQL has an edge over REST because of schema-based data fetching and mutation architecture.

```
QUERY
1 # Write your query or mutation here
2 query {
3   speakers {
4     id
5     bio
6     name
7     Session {
8       id
9       title
10    }
11  }
12 }
```

```
1 # Write your query or mutation here
2 query {
3   speakers {
4     id
5     bio
6     name
7   }
8 }
```

Figure 5.4: Query granularity

Like the above example, both queries executing in the same schema object left query calling speakers with Session object and right query just fetching speakers with its respective fields.

From the data quality perspective, this granular fetching provides much more clean data output and the amount of data gets reduced. The main objective here is “only fetch the data application need to load that component” which supports the AOT compilation method and widely supports via modern Javascript frontend frameworks like React, Angular.

Similar data manipulation can not be done in REST without creating another endpoint and extends URI links. Which overloads the development time for both ends and introduces non-DRY code inside the codebase.

I do not completely agree that data granularity is better in GraphQL, hence REST also provides, much more sophisticated way to handle data manipulation. But in the simplicity of granular operation GraphQL of course ahead of the comparison.

And the second part, application support. From a development perspective, both have a wide range of language choices and flexibility of coding style. GraphQL has its own dependability toward Javascript because it was primarily developed for frontend data granularity purposes, at the same time all other modern languages like python, C#, Java can have the same opportunity.

Another hand, REST supports almost all modern languages and has more mature support and user base.

In this case, REST should be used in lower relational data architecture where structured data has more priority.

Usability describes also, the capability of the tool/tools to perform specific tasks for the users effectively. Human interaction with both technologies within SDLC defines the testing ground for usability. Usability elaborates the satisfaction rate of the tool and the overall quality of the tool.

As per standard **ISO/IEC 9126-1:2001** software engineering product quality the main attribute of the software usability with human interaction can be judged by:

- Understandability
- Learnability
- Operability
- Attractiveness

According to my finding and analyze, I prepare a simple table with rating system from 1 to 5.

Standard	REST	GraphQL
Understandability	Score : 5 Reason: Very easy to understand for beginner level. Complexity spikes with the design depth itself.	Score: 3 Reason: Very complex in the beginning, but the complexity reduce with further understanding.
Learnability	Score: 4 Enough resources available.	Score: 3 Enough resources available.
Operability	Score: 3 Reason: Operation with flexibility is not so simple due to very structure data flow.	Score: 5 Reason: Very easy to operate, high flexibility and integration capability.
Attractiveness	Score: 0 Not Applicable	Score: 0 Not Applicable
<b>Total</b>	<b>12</b>	<b>11</b>

Table 5.3: Usability ranking

From the perspective of usability both technologies have almost same position. It is depending on user preferences and use case.

## 5.4 Maintainability

Modern API maintenance consists of layers of service maintenance. A normal API can consist of at least 3 layers:

1. Connector layer (DB, other services)
2. Security (SSO, Token)
3. Endpoints/Resources

So maintaining an API is not a linear codebase maintain. The good news is in the current technological structure cloud computing providers like Azure, AWS has a very sophisticated solution for API management.

API call numbers can vary on application architecture which is not measurable at the beginning, also developer experience and coding skills can affect the total number of API calls.

Other than call, all other layers can be predefined and pre-estimated on architecture. The security layer is the most important maintenance layer, which does not affect the performance but of course, affects usability and access. GraphQL and REST both support token and SS.O implementation. Also, modern security frameworks like, OAuth support seamlessly with any architecture.

Certainly, GraphQL or REST does have a direct connector layer. A REST or GraphQL can have multiple data input resource which calls and execute in code and give output to endpoints or schema. Both architectures have their connector functions, REST attached with resource within a single monolith call or microservice, also GraphQL has flexibility over connectors integration. The data layer does not care about the connector layer but rather depends on it.

Maintainability refer the ease of maintain the currently build complete product in order to

- Patch release.
- Bug finding and fix.
- Protect product competitiveness.
- Maximize efficiency.
- New requirements.
- Future adoption
- Integration integrity.

According to **ISO/IEC 9126**

Maintainability contains 5 specific features.



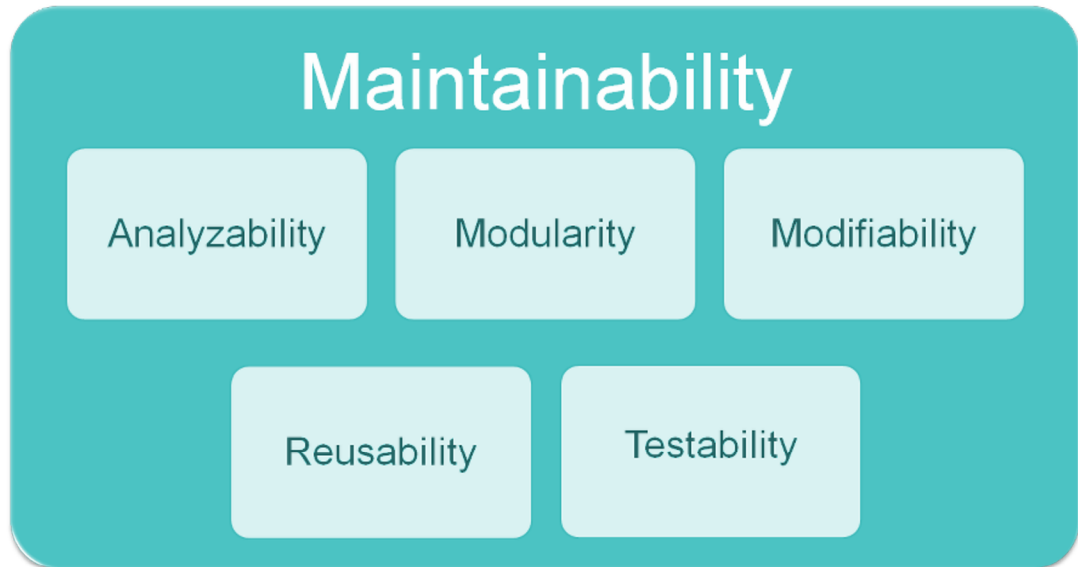


Figure 5.5: Maintainability features

For the purpose of showing the difference between of our selected technologies I used table method with ranking which similar as usability.

Rank between 1 to 5.

Standard	REST	GraphQL
<b>Analyzability</b>	Score: 5 Reason: Analyzing a data structure is very simple due to structured approach.	Score: 4 Reason: Analyzing a schema can be bit difficult, with some external tooling this can be done.
<b>Modularity</b>	Score: 2 Reason: Not modular due to ROA based data flow. Specific identifier can only handle specific set of data	Score: 5 Reason: Modularity is the key of GraphQL. Robust inter connectivity and object based approach make this more easy
<b>Modifiability</b>	Score: 3 Reason: Easily modifiable, due to linear code structure.	Score: 3 Reason: Initially complex to modify existing parts, if developer do not have architectural overview
<b>Reusability</b>	Score: 4 Reason: REST endpoint can be reusable but not the resource within different request.	Score: 5 Reason: Very strong reusable functionality. With DRY approach codebase can be very redundant free.
<b>Testability</b>	Score: 5 Reason: Various tools and methods.	Score: 5 Reason: Various tools and methods.
<b>Total</b>	<b>19</b>	<b>22</b>

Table 5.4: Maintainability ranking

From the ranking, the output lean toward GraphQL due to the characteristic of the technology. Also, I would like to mention that, all of this standards can impact various project in different ways. In case of project, where modularity is not so important as testability.

In verdict, I can say that hence the GraphQL show much positive result toward maintainability, it can also be REST if the usage perspective is different.

Software projects need to very well evaluate before going for specific solution. In my opinion, all require standard related with **ISO/IEC 9126** should be individually consider as a parameter, not put in a group.

This thesis objective on maintainability conducted and provides specific result with sufficient reasoning and valuation.

## 5.5 Conclusion

The main objective of the thesis was to analyse both technology and the architectural approach and provides a clear differential value between them. Modern applications and services became very complex and data became more and more non-linear relational. From data generate to fetching and manipulation through application layers getting complex and costly for development. On the other hand, application, frontend layer depends on browser engines, which have limited memory capacity and limited internet speed. So, any data manipulation cost can have an effect on usability which affects the direct SDLC chain.

The theoretical and practical parts describe the current state and minimal capability of the technologies and present an open-source codebase to manipulate and reuse.

In the second part, I have defined the main difference between REST and GraphQL with clear recommendations on a case basis.

The goal was to demonstrate and catalyst the decision of API architecture for software development. To achieving that, several tests have been done with technological separation. Also, Broad theoretical differences have been established with a detailed overview.

As a software development process gets more complex and technologically jargon, this thesis clarifies the API architecture point of view, when to use GraphQL and when to use REST.

This thesis can be extended to cloud-based deployment to test both architecture performance in the different cloud platforms, which is a significant next step. Another improvement can be made to implement real-time DB with more industrial microservices to test WebSocket protocol performances.

I had enjoyed the time of the thesis and exploration of different technologies. A warm thanks to my faculty for continuous support and available resources.

Base on the study, this thesis can be concluded that GraphQL is a better solution when the data relation is non-linear, data fetching size needs to be reduced, and when the frontend of the application has complex representation logic.

In the case of REST can be very useful on, any linear relational data with minimum representational complexity with high data fetching capacity.

However, the methodological procedure and research conclusions have great potential to explain, analyse and evaluate current API architectural distinction between GraphQL and REST and clarify the technological decision for software development requirements.

## 6 Bibliography

- [1] <https://searchapparchitecture.techtarget.com/definition/REST-REpresentational-State-Transfer>
- [2] Fielding, R. T. REST: Architectural Styles and the Design of Network-based Software Architectures. Doctoral dissertation, University of California, Irvine, 2000.
- [3] Jian Meng, Shujun Mei, Zhao Yan. RESTful Web Services: a Solution for Distributed Data Integration. International Conference on Computational Intelligence and Software Engineering. December, 2009.
- [4] S. Vinoski. RESTful Web Services Development Checklist. IEEE Internet Computing. November, 2008.
- [5] Haibo Zhao, Prashant Doshi. Towards Automated RESTful Web Service Composition. IEEE International Conference on Web Services. 2009.
- [6] RESTful Web Services by Leonard Richardson, Sam Ruby.
- [7] Richardson, L., and Ruby, S. Restful web services, 1.st ed. O'Reilly,2007
- [8] Erik Wilde, Cesaro Pautasso. REST: From Research to Practice. Springer Science+Business Media. 2011.
- [9] Learning GraphQL: Declarative Data Fetching for Modern Web Apps Book by Alex Banks and Eve Porcello
- [10] Introduction to Graph Theory Richard J Trudeau
- [11] <https://about.sourcegraph.com/graphql/graphql-client-driven-development/>
- [12] <https://docs.npmjs.com/about-npm>