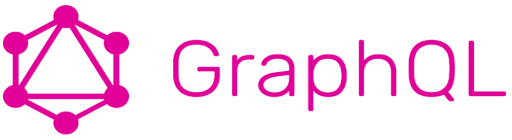
UNIVERSITY OF GRONINGEN
FACULTY OF SCIENCE AND ENGINEERING
DEPARTMENT OF COMPUTING SCIENCE

Information Systems



Group 07

AUTHORS
Frans Simanjuntak (S3038971)
Stefan Cretu(S3048438)

05 December 2017

Contents

1. Introduction

In this document is presented the solution for the assignment 2 of Information Systems course. The main requirement is to build RESTful and a GraphQL APIs that are used to query a database which contains information about people, that could be users of a social network. These users have friends, who are also users of the same social network, and there are cases when it is useful to get the list of friends of an user. Therefore, the APIs presented in this document are also assessed based on their performance to make such nested queries. On top of those, each API is analyzed with respect to some attributes such as security, performance in load balancing, easiness in data access and usability

2. APIs description

In this chapter are presented theoretical aspects with respect to GraphQL and RESTful APIs.

2.1 Restful API

A RESTful API is an application program interface (API) that provides interoperability between computer systems on internet. In a RESTful API, requests made to a resource's URI will elicit a response that may be in XML, HTML, JSON or some other defined format. The response may confirm that some alteration has been made to the stored resource, and it may provide hypertext links to other related resources or collections of resources. There are four HTTP requests that available namely GET, PUT, POST and DELETE.

A RESTful API explicitly takes advantage of HTTP methodologies defined by the RFC 2616 protocol. They use GET to retrieve a resource, PUT to change the state of or update a resource which can be an object, file or block, POST to create that resource, and DELETE to remove it [1].

REST architectures consist of clients and servers. Clients initiate requests to servers and servers process requests and return appropriate responses. Requests and responses are built around the transfer of representations of resources. A resource can be essentially any coherent and meaningful concept that may be addressed [2]. A representation of a resource is typically a document that captures the current or intended state of a resource. The presumption is that all calls are stateless, nothing can be retained by the RESTful service between executions.

2.2 GraphQL

GraphQL is a query language used of APIs. It provides a complete description of the data in the API and allows for an easier API evolution, over time. It’s main advantage comes from the fact it allows the user to get many resources in the same request, as the query follows the references (connections) between the resources (also called objects). That said, its structure makes it suitable when it is desired to find friends of a person, or friends of friends of a person.

The GraphQL engine involves the modelling of the business model of the target application as a graph, by defining a schema which contains nodes, referred to as objects, and relationships between them. Each such a node defines a type and contains one or more fields that can be of another type or of a basic type. This way, the relationships between the defined types are stated, leading to the creation of an inheritance tree, like in object oriented programming. However, in the case of GraphQL the inheritance is not unidirectional (like child class inherits from parent class), but bidirectional, as each type can refer to any other

defined type, which actually involves the creation of a graph between all defined types. Therefore the name GraphQL.

An example is shown below, where 2 objects are created: *Query* and *User*. These objects can be fetched from the application. The first one, has only one field, which is of type *User* and the second one has 2 fields, one of type String and the other one of type ID. Moreover, the first one is a special object in GraphQL as every GraphQL service must have such a type as it defines the entry point of every query (where the engine starts looking from or the root node of the graph).

```
type Query
{
        me: User
}

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

Figure 2.1 Example of types and fields in graphQL [5]

Based on this structure, GraphQL allows to add more types and fields to the API, without changing the existing queries, which explains its easiness in evolution. On top of this, the fields can have multiple types, as they can be of another object type or can be scalars (int, float, string, bool) or enumerations (scalars restricted to a set of values). The last two represent the basic types in GraphQL and they represent the leaves in a GraphQL query as they do not have any fields (they cannot be further decomposed). In addition, when building a query, there can be specified other field types such as null, which means that the query server may not find any data for that field, or list, which tells to the query server that more values are expected for that field, all having same type.

GraphQL allows for building an uniform API for the entire application as the API can be constructed in many programming languages. The way it operates involves creating one function for each existing field in the type system which are called whenever a query that involves those types is addresses. The calling of these functions is optimal when concurrent access is made. Examples of such functions are presented below, as they match some of the field used in Figure 2.1. The first function is used to get the data of the authenticated user, whereas the second one is used to get the name of such an user.

```
function Query_me(request)
{
        return request.auth.user;
```

```
}

function User_name(user)
{
        return user.getName();
}
```

Figure 2.2 Functions implementation used for accessing defined fields [5]

GraphQL service runs on a web service or at a URL and it handles incoming queries. All queries are checked for validation, as they pass the checking when they address only the defined fields. Once a query passes the verification step, then the corresponding function is called. Then, the result is shown in JSON format. An example of a query is illustrated below, alongside the answer to it, in JSON format. It can be noticed that the query specifies the defined fields, whereas the returned answer still contains them together with the afferent values.

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

Figure 2.3 Query example [5]

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

Figure 2.4 JSON result of the above query [5]

The execution of a query starts from the entry point and continues with all nested fields, if any. For each field, as mentioned above, a function is implemented, named resolver, which is called to return the value of the specified field. If the returned value is a basic type, such as int or string, then the call graph stops, but if the returned type is an object it continues until it reaches only basic types.

Since most of the application nowadays use pipelines for handling the requests in order to filter, modify, validate and distribute them (if the case), the GraphQL should be placed after the authentication layer. By this way, all security checks are performed before the query is executed, ensuring access to data in the same HTTP session, which is the most common protocol used for serving GraphQL queries, which always address the root node of the graph, which can be found at the same URL. Nevertheless, the authorization can be included the

query resolver logic, by checking if the current user's id equals the author's id, for instance. This can happen only if the query takes as argument an user object, in order to make the above-mentioned verification.

Amongst the most popular users of GraphQL are GitHub, Facebook and Coursera. [5]

3. Implementation

As mentioned in chapter 1, we focus our experiment on following several quality attributes, as we aim at evaluating the two considered APIs against them. Firstly, we need to define these quality attributes, as well as the way we address them, as follows:

- Security: we look into the possibilities of secure data access offered by Restful and GraphQL APIs
- Data access: it is defined as the easiness in accessing data. More concretely, we refer to it as the easiness of building the queries in terms of the time spent on understanding how they work, the amount of written code and the easiness in integrating the dedicated libraries that allow us to use them in our working environment
- Load balancing: the performance, measured in execution time, when multiple similar requests are made to the same data
- Usability: efforts needed to be made to work with a given API, including the additional efforts when changes are needed to be made. It is closely tied to Data access aspect.

That said, below are the details of implementation in order to simulate the differences between Restful and GraphQL. Firstly, we present the cases used to assess each quality attribute described above. Secondly, we explain the solution we applied, altogether with the libraries and frameworks we used. Afterwards, we address the above-presented quality attributes in correlation to the use cases presented in section 3.1, for each considered API.

3.1 Use Cases

In this assignment, we defined a case as the basis for simulating different scenarios on Restful and GraphQL APIs. The problem that we are going to address was taken from one of facebook features, so-called person and friends of that person. When creating an account on facebook, each user must provide their information detail such as first name, last name, gender, country, and email. After that, she/he can send friend requests to other users, resulting in having none or at least one friend. When 2 users become friends, they can see their information details mentioned above.

Based on the case given above, we define several scenarios in order to simulate how we can retrieve such information using both approaches. There are four scenarios that we need to simulate as listed below:

1. Get all basic information of a person excluding list of his/her friends
2. Get firstname and country of a person excluding list of his/her friends
3. Get all basic information of a person including all basic information of his/her friends
4. Get all basic information of a person including all basic information of his/her friends including all basic information of each of his/her friend’s friends (nested query)

3.2 Solution

Before creating a program for each API, of course the very first thing to do is to create a schema of the data layer which is able to store all information. Then, both APIs will be able to query data from the same data store.

3.2.1 Database Schema

We use MySQL as our database in order to store the information of person and his/her friends. We created two tables which are linked by primary keys as described in figure 1.

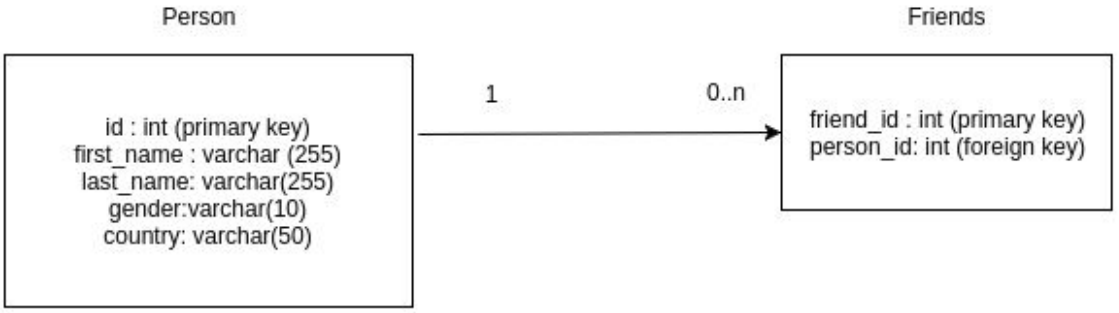


Figure 1. Schema of database

Figure 1 describes the schema of database which contains two tables: person and friends. Table person contains six attributes: id as primary key, first_name, last_name, gender, and country. Table friends contains only two fields: friend_id as primary key and person_id as foreign key. The relationship between these two tables is one to many. The one to many relationship can be intuitively interpreted as one person can have none or many friends. In order to be able to get the information detail of someone's friends, we should query back from table person.

3.2.2 Programming language, ORM, and API framework

Below are the details of technologies used for developing Restful and GraphQL API.

- **Programming language**
  We use python as our programming language since it's rich in terms of library and of course easy to code.

- **ORM**
  ORM stands for object relational mapping. It is a technique that lets us query and manipulate data from a database using an object-oriented paradigm. A python ORM library that we use for this assignment is SQLAlchemy. The SQLAlchemy considers the database to be a relational algebra engine, not just a collection of tables. Rows can

be selected from not only tables but also joins and other select statements; any of these units can be composed into a larger structure [3].

We created two models namely Person and Friends which are mapped automatically by SQLAlchemy with the tables in database.

- **Flask**
  Flask is a web framework that provides tools, libraries and technologies that allows us to build a web application in python. We will make use of flask to create Rest and Graphql API. When creating an Restful API on top of Flask framework, we do not need any additional libraries. However, additional library called flask graphql and graphene should be included when creating Graphql API.

- **Grapehene**
  Graphene is a library for building GraphQL APIs in Python easily. It's main goal is to provide a simple but extendable API for making developers life easier.

- **Apache JMeter**
  Apache JMeter is used to test performance of Web API. It can be used to simulate a heavy load on a server, group of servers, network or object to test its strength or to analyze overall performance under different load types.

3.3 Simulating scenarios on different quality attributes

The details of simulating all scenarios on different quality attributes listed in section 3.1 are described in the following subsections.

3.3.1 Data access (Easiness of implementation) and Usability

In order to assess easiness of implementation for GraphQL and for RESTful API, respectively, we took into account each scenario described in section 3.1

1) **Scenario 1:** "Get all basic information of a person excluding list of his/her friends"

   ➢ ***RESTful API***

   To simulate this scenario, we created a method called getperson which receives a parameter userid as described in figure 2. On top of this method we placed app.route which maps the user request with the corresponding method. In the app route, we define <userid> that indicates the id of the person we are searching for. For example, if we want to search the information of person with id = 1, then the url request should be http://127.0.0.1:5000/person/1.

```python
#restful method that return a person
@app.route('/person/<userid>', methods=['GET'])
def getperson(userid):
    allfields = request.args.get('allfields', default = 1, type = int)
    including_friends = request.args.get('including_friends', default = 1, type = int)

    if (allfields == 1):
        if (including_friends == 0):
            if __name__ == '__main__':
                personobj = db_session.query(PersonModel).\
                    with_entities(PersonModel.id, PersonModel.first_name, PersonModel.last_name, PersonModel.country, PersonModel.gender).\
                    filter(PersonModel.id == userid).first()

            return json.dumps(personobj, cls=new_alchemy_encoder(), check_circular=False);
```

Figure 2. Restful method that returns all basic information of person

Since our intention is to get all basic information of someone excluding his/her friends, therefore we should somehow allow user to define the filters. In order to do so, we must force user to set two boolean parameters called *allfields* and *including_friend*. If the value of allfields = 1 then the API will return all fields. Moreover if the value of including_friend = 0, then the API will return all fields excluding friends. That being said, the url for this scenario would be:

http://127.0.0.1:5000/person/1?allfields=1&including_friends=0

The output of this request is described in figure 3 and it took 0.003806 second to get the response back from the server.

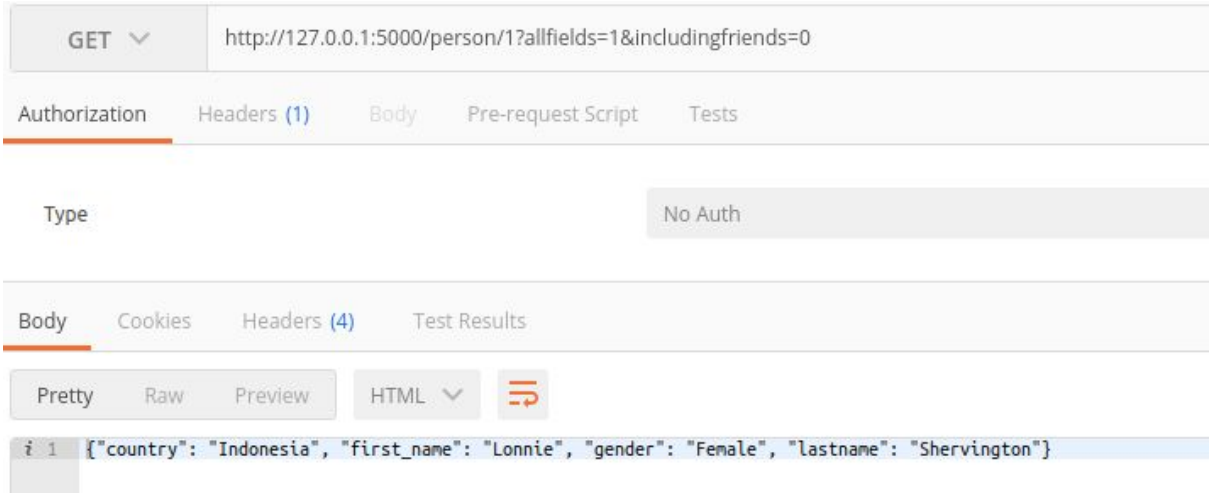


Figure 3. Output of scenario 1 using Restful API

➢ ***GraphQL***

In GraphQL, the implementation is way more complicated than Restful API since the additional library called graphene should be involved. However, we will be beneficial later on because we can reuse this code for the entire scenarios.

In order to create graphql API, firstly we need to create a schema file that handles the query and model. This file consists of three class namely Persons, PersonType, and Query. Class Person describes graphene data model in which a meta class stored inside. The meta class consists of two attributes: model and interfaces. The value of attribute model is *PersonModel* (our data model) and the value of interfaces is relay.Node.

Class PersonType describes the type of our data model when converted to graphene model. In this class, we must specify data types of each attribute in our data model using graphene data type. Moreover, we must create a class called Query which describes all methods of graphql together with its implementation.

To simulate this scenario, we define a method called find_person inside the class Query and in order to solve this method, we created another method called resolve_find_person as described in figure 4. This method receives one parameter called id.

```
import ...

class Persons(SQLAlchemyObjectType):
    class Meta:
        model = PersonModel
        interfaces = (relay.Node, )

class PersonType(graphene.ObjectType):
    name = 'Person'
    description = 'Person Model'

    first_name = graphene.String()
    last_name = graphene.String()
    gender = graphene.String()
    country = graphene.String()

class Query(graphene.ObjectType):
    find_person = graphene.Field(
        PersonType,
        id = graphene.String()
    )

    def resolve_find_person(self, args, context, info):
        query = Persons.get_query(context)
        id = args.get('id')
        return query.filter(PersonModel.id == id).first()

schema = graphene.Schema(query=Query, types=[Persons])
```

Figure 4. Graphql method that returns all basic information of person

Now, lets simulate this scenario by using person id = 1 as the parameter. First, we should open http://127.0.0.1:5000/graphql on our browser in order to be able to simulate graphql. Since we only interested in all basic information of person 1 excluding its friend list, therefore we can filter them on client side without adding additional codes. The output of this scenario is given in figure 5 and it took 0.005964 seconds to get the response back from the server.

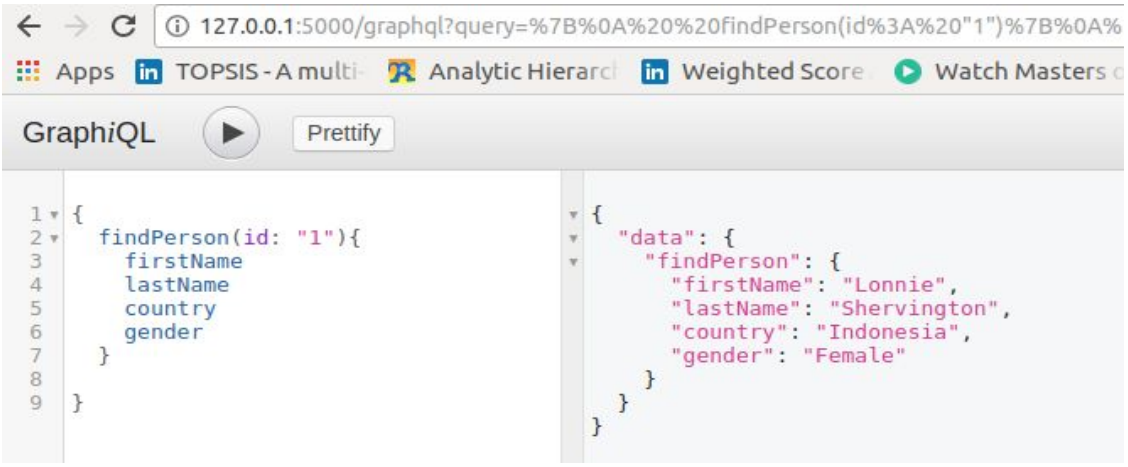


Figure 4. Output of scenario 1 using Graphql

2) Scenario 2: "Get firstname and country of a person excluding list of his/her friends"

➢ ***RESTful API***

To simulate this scenario using Restful API, we have to add new parameter in the url to tell the API which fields we are interested in. Also, we need to tweak existing code to fetch desired attributes accordingly. Below are the url and updated source code for this scenario.

http://127.0.0.1:5000/person/1?allfields=0&includingfriends=0&desiredcolumns=first_name,country

```python
#restful method that return a person
@app.route('/person/<userid>', methods=['GET'])
def getperson(userid):
    allfields = request.args.get('allfields', default = 1, type = int)
    including_friends = request.args.get('includingfriends', default = 1, type = int)
    desired_columns = request.args.get('desiredcolumns', type = str)

    if (allfields == 1):
        if (including_friends == 0):
            if __name__ == '__main__':
                personobj = db_session.query(PersonModel).\
                    with_entities(PersonModel.id, PersonModel.first_name, PersonModel.last_name, PersonModel.country, PersonModel.gender).\
                    filter(PersonModel.id == userid).first()

            return json.dumps(personobj, cls=new_alchemy_encoder(), check_circular=False);
    else:
        arr_strcolumns = desired_columns.split(",")
        if "first_name" or "country" in arr_strcolumns:

            personobj = db_session.query(PersonModel).\
                    with_entities(PersonModel.first_name,  PersonModel.country).\
                    filter(PersonModel.id == userid).first()

            return json.dumps(personobj, cls=new_alchemy_encoder(), check_circular=False);
```

Figure 5. Updated Restful method that handles scenario 2

The output of the Restful API is given in figure 6 and it took 0.003958 second to get the response back from the server.

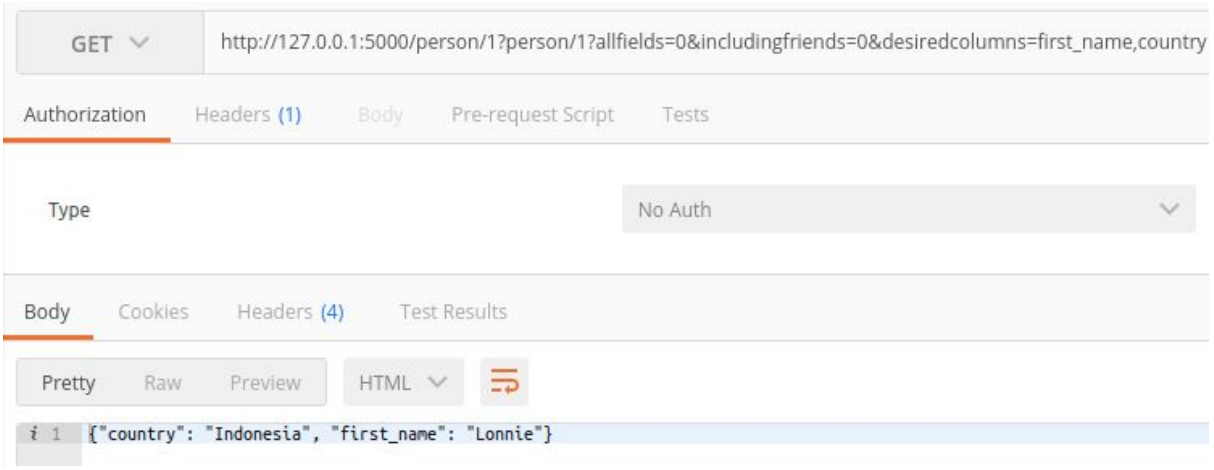


Figure 6. Output of scenario 2 using Restful API

➢ ***GraphQL***

To simulate this scenario using graphql is simple. We just reuse the method in scenario 1 without adding any code and on client side, we just need to specify the desired columns as described in figure 7 and it took 0.005757 second to get the response back from the server.

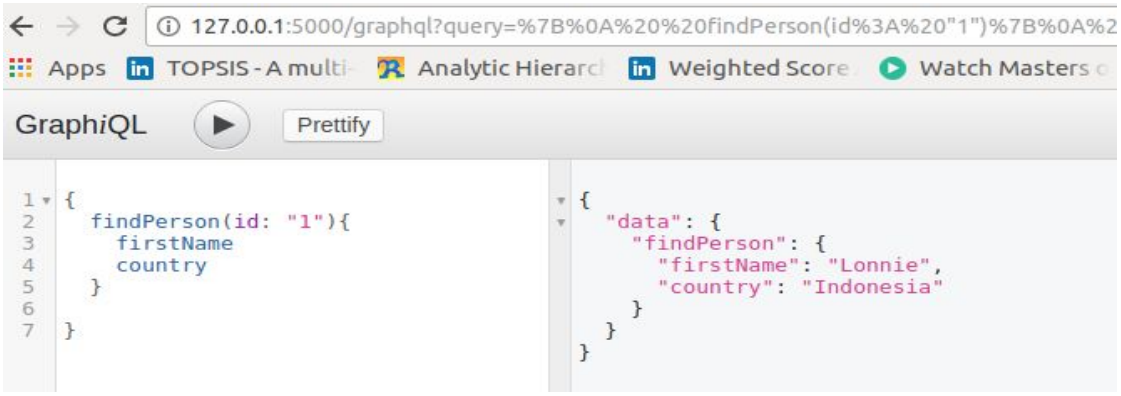


Figure 7. Output of scenario 2 using GraphQL API

3) **Scenario 3**: "Get all basic information of a person including list of his/her friends"

➢ ***RESTful API***

In order to simulate this scenario using restful API, we need to modify the parameter including friends to 1.

http://127.0.0.1:5000/person/1?allfields=1&includingfriends=1

Then we should tweak the code by adding the logic corresponding to this filter as described in figure 8.

```
#restful method that return a person
@app.route('/person/<userid>', methods=['GET'])
def getperson(userid):
    allfields = request.args.get('allfields', default = 1, type = int)
    including_friends = request.args.get('includingfriends', default = 1, type = int)
    desired_columns = request.args.get('desiredcolumns', type = str)

    if (allfields == 1):
        if (including_friends == 0):
            personobj = db_session.query(PersonModel).\
                    with_entities(PersonModel.id, PersonModel.first_name, PersonModel.last_name, PersonModel.country, PersonModel.gender).\
                    filter(PersonModel.id == userid).first()

            return json.dumps(personobj, cls=new_alchemy_encoder(), check_circular=False);

        else:
            personobj = db_session.query(PersonModel).filter(PersonModel.id == userid).first()

            lst_friends = []
            for fr in personobj.friends:
                friendobj = db_session.query(PersonModel).filter(PersonModel.id == fr.friend_id).first()
                friend = NewPersonModel(friendobj.id, friendobj.first_name, friendobj.last_name, friendobj.gender, friendobj.country,[])
                lst_friends.append(friend)

            newperson = NewPersonModel(personobj.id, personobj.first_name, personobj.last_name, personobj.gender, personobj.country, lst_friends)

            json_string = json.dumps(newperson, default=obj_dict)
            return jsonify({'person': json_string})

    else:
        arr_strcolumns = desired_columns.split(",")
        if "first_name" or "country" in arr_strcolumns:

            personobj = db_session.query(PersonModel).\
                    with_entities(PersonModel.first_name,  PersonModel.country).\
                    filter(PersonModel.id == userid).first()

            return json.dumps(personobj, cls=new_alchemy_encoder(), check_circular=False);
```

Figure 8. Updated Restful method that handles scenario 3

The output of this method is described in figure 9 and it took 0.010876 second to get the response back from the server.

Figure 9. Output of scenario 3 using Restful API

➢ *GraphQL*

To simulate this scenario on graphql, we also need to tweak the existing code in scenario 1. However, it is not as complicated as we did in Rest API. We just have to add a new attribute called friends in class PersonType and add corresponding method that handles this attribute. The new implementation is described in figure 10.

```
import ...

class Persons(SQLAlchemyObjectType):
    class Meta:
        model = PersonModel
        interfaces = (relay.Node, )

class PersonType(graphene.ObjectType):
    name = 'Person'
    description = 'Person Model'

    first_name = graphene.String()
    last_name = graphene.String()
    gender = graphene.String()
    country = graphene.String()
    friends = graphene.List(lambda: PersonType)

    def resolve_friends(person, args, context, info):
        query = Persons.get_query(context)
        return [query.filter(PersonModel.id == f.friend_id).first() for f in person.friends]

class Query(graphene.ObjectType):
    find_person = graphene.Field(
        PersonType,
        id = graphene.String()
    )

    def resolve_find_person(self, args, context, info):
        query = Persons.get_query(context)
        id = args.get('id')
        return query.filter(PersonModel.id == id).first()

schema = graphene.Schema(query=Query, types=[Persons])
```

Figure 10. Updated Graphql method that handles scenario 3

The output of this method is described in figure 11 and it took 0.016375 second to get the response back from the server.

```
{
  findPerson(id:"1"){
    firstName
    lastName
    gender
    country
    friends{
      firstName
      lastName
      gender
      country
    }
  }
}
```


```
{
  "data": {
    "findPerson": {
      "firstName": "Lonnie",
      "lastName": "Shervington",
      "gender": "Female",
      "country": "Indonesia",
      "friends": [
        {
          "firstName": "Emlyn",
          "lastName": "Parmiter",
          "gender": "Male",
          "country": "China"
        },
        {
          "firstName": "Justina",
          "lastName": "Marzellano",
          "gender": "Female",
          "country": "Zambia"
        },
        {
          "firstName": "Grannie",
          "lastName": "Bushaway",
          "gender": "Male",
          "country": "France"
        },
        {
          "firstName": "Gradeigh",
          "lastName": "Nuscha",
          "gender": "Male",
          "country": "Indonesia"
        },
        {
          "firstName": "Clem",
          "lastName": "Pierri",
          "gender": "Male",
          "country": "Poland"
        },
        {
          "firstName": "Hermine",
          "lastName": "Ohrtmann",
          "gender": "Female",
          "country": "Peru"
        },
        {
          "firstName": "Blair",
          "lastName": "Tunnadine",
          "gender": "Male",
          "country": "Marshall Islands"
        }
      ]
    }
  }
}
```


Figure 11. Output of scenario 3 using GraphQL API

4) **Scenario 4:** "Get all basic information of a person including all basic information of his/her friends including all basic information of each of his/her friend's friends (nested query)"

➢ ***RESTful API***

In order to be able to simulate this scenario using restful API, we should add one more parameter called friends of friend. The value of this parameter is either 1 or 0. If we set the value to 0, the restful API only returns the detail of a person including the detail of his/her friends (without list of friends). However, if we set the value to 1, the restful API should return the details of a person including his friend's details. The url request for this scenarios as follows:

http://127.0.0.1:5000/person/1?allfields=1&includingfriends=1&friendsoffriend=1

We also need to tweak the existing code by adding query for retrieving friends of person friend's as described in figure 12.

```
#restful method that return a person
@app.route('/person/<userid>', methods=['GET'])
def getperson(userid):
    allfields = request.args.get('allfields', default = 1, type = int)
    including_friends = request.args.get('includingfriends', default = 1, type = int)
    desired_columns = request.args.get('desiredcolumns', type = str)
    friends_offriend = request.args.get('friendsoffriend', default = 0, type = int)

    if (allfields == 1):
        if (including_friends == 0):
            personobj = db_session.query(PersonModel).\
                    with_entities(PersonModel.id, PersonModel.first_name, PersonModel.last_name, PersonModel.country, PersonModel.gender).\
                    filter(PersonModel.id == userid).first()

            return json.dumps(personobj, cls=new_alchemy_encoder(), check_circular=False);

        else:
            personobj = db_session.query(PersonModel).filter(PersonModel.id == userid).first()

            lst_friends = []
            for fr in personobj.friends:
                friendobj = db_session.query(PersonModel).filter(PersonModel.id == fr.friend_id).first()

                arr_friends = []
                if (friends_offriend == 1):
                    for fr2 in friendobj.friends:
                        friendobj2 = db_session.query(PersonModel).filter(PersonModel.id == fr2.friend_id).first()
                        friend2 = NewPersonModel(friendobj2.id, friendobj2.first_name, friendobj2.last_name, friendobj2.gender, friendobj2.country,[])
                        arr_friends.append(friend2)

                friend = NewPersonModel(friendobj.id, friendobj.first_name, friendobj.last_name, friendobj.gender, friendobj.country,arr_friends)
                lst_friends.append(friend)

            newperson = NewPersonModel(personobj.id, personobj.first_name, personobj.last_name, personobj.gender, personobj.country, lst_friends)

            json_string = json.dumps(newperson, default=obj_dict)
            return jsonify({'person': json_string})

    else:
        arr_strcolumns = desired_columns.split(",")
        if "first_name" or "country" in arr_strcolumns:

            personobj = db_session.query(PersonModel).\
                    with_entities(PersonModel.first_name,  PersonModel.country).\
                    filter(PersonModel.id == userid).first()

            return json.dumps(personobj, cls=new_alchemy_encoder(), check_circular=False);
```

Figure 12. Updated Restful method that handles scenario 4

The output of scenario 4 is given in figure 13 and it took 0.032961 second to get the response back from the server.

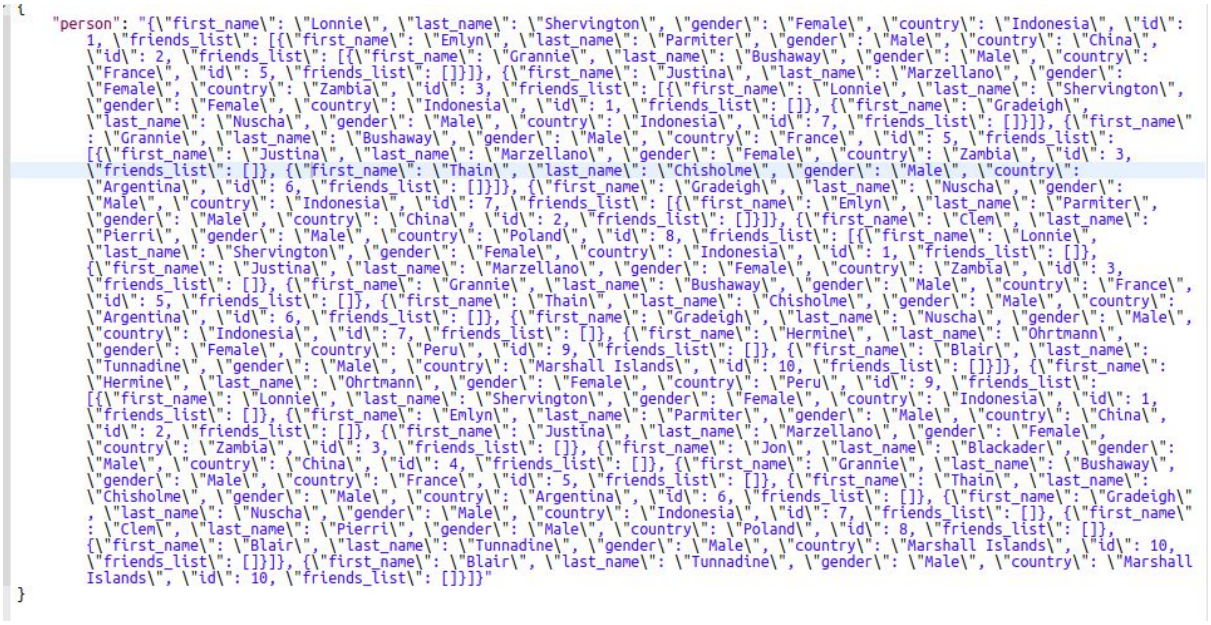


Figure 13. Output of scenario 4 using Restful API

➢ ***GraphQL***

Simulating scenario 4 using graphql is so easy, since we do not need to modify our existing code. All we have to do is to define the parameter properly on the client side as described in figure 14 and it took 0.03865 second to get the response back from the server.

Figure 14. Output of scenario 4 using Graphql API

3.3.2 Security

Security is the essential first step to making sure only API calls with valid, successfully authenticated credentials are able to access API. By controlling the API access, we ensure that calls with authenticated logins are able to access the APIs. In this occasion, we implemented access control on Restful and GraphQL API. The security of each considered API is not assessed for a given scenario, like we presented in subsection 3.3.1, but for any possible query, as this aspect is not query dependent.

➢ ***RESTful API***

To make the restful API secure, we need to add authentication to each method. Since we are using Flask, we can make use of basic HTTP authentication function provided by this framework so-called HTTPBasicAuth.

Since this function is based on HTTP basic authentication therefore we need to override function `@auth.verify_password` in order to tell Flask how should we implement authentication in our API. In this occasion, we just simply define a very basic authentication. If the username is "stefan" or "frans" and password is "testingapi", the method getperson can be accessed, otherwise unauthorized access.

```
#instantiate Flask framework
app = Flask(__name__)
auth = HTTPBasicAuth()

@auth.verify_password
def verify_password(username, password):
    if (username in ["frans","stefan"] and password=="testingapi"):
        return True


#restful method that return a person
@app.route('/person/<userid>', methods=['GET'])
@auth.login_required
def getperson(userid):
    allfields = request.args.get('allfields', default = 1, type = int)
    including_friends = request.args.get('includingfriends', default = 1, type = int)
    desired_columns = request.args.get('desiredcolumns', type = str)
    friends_offriend = request.args.get('friendsoffriend', default = 0, type = int)

    if (allfields == 1):
        if (including_friends == 0):
            personobj = db_session.query(PersonModel).\
                    with_entities(PersonModel.id, PersonModel.first_name, PersonModel.last_name, PersonModel.country, PersonModel.gender).\
                    filter(PersonModel.id == userid).first()

            return json.dumps(personobj, cls=new_alchemy_encoder(), check_circular=False);
```

Figure 15. Implementation of security on Restful API on top of Flask framework

The output of Restful API when the username and password are incorrect and vice versa are described in figure 16 and 17 respectively.

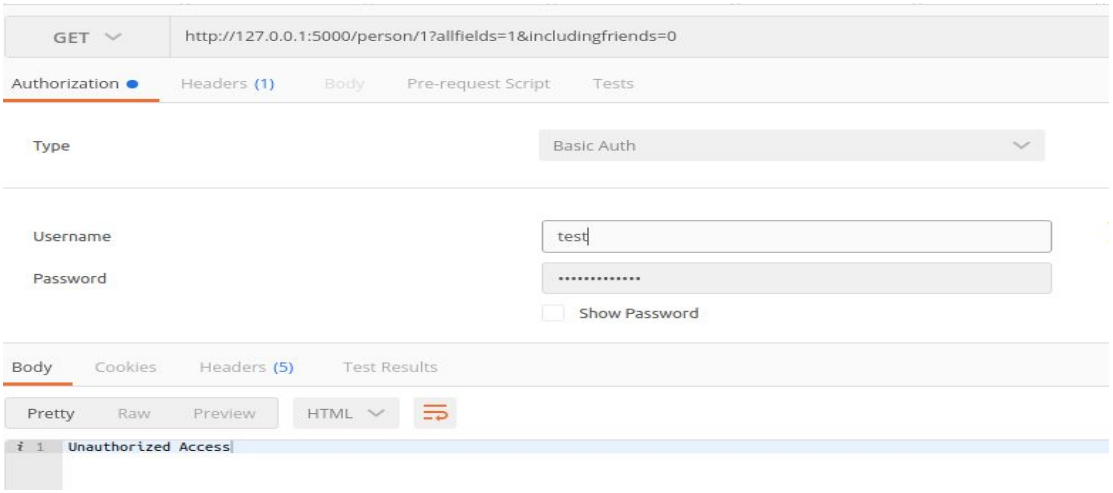


Figure 16. Authentication with incorrect username and password

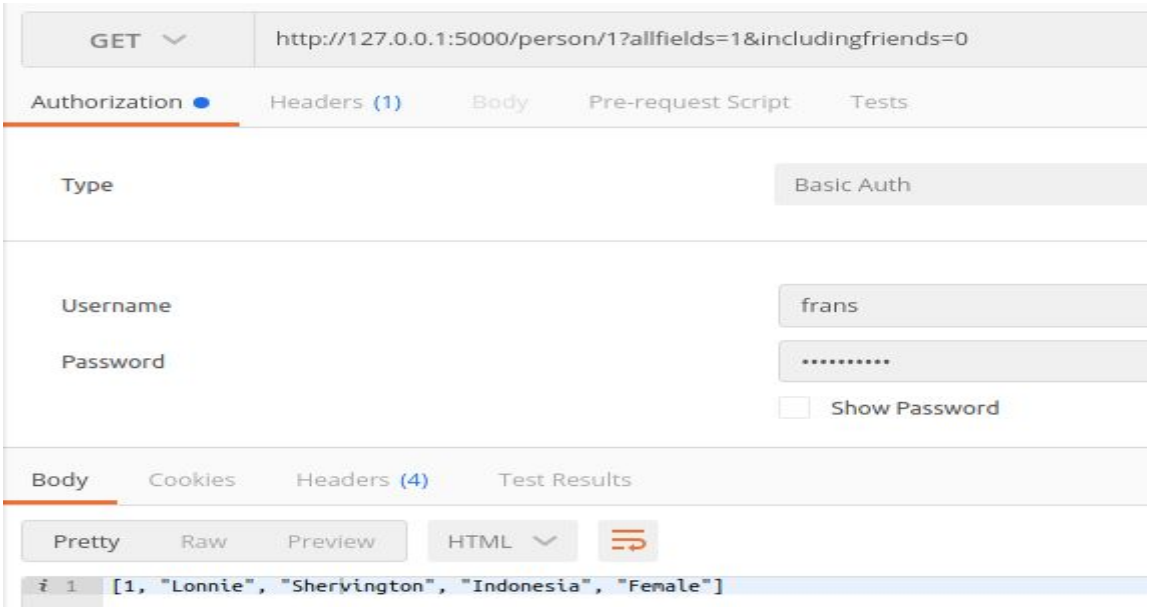


Figure 17. Authentication with correct username and password

➢ ***Graphql***

There are several techniques that can be used to implement security on Graphql. One technique that works well is by using JWT. **JSON Web Token** (**JWT**) is an open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. This information can be verified and trusted because it is digitally signed.

However the implementation of JWT is not easy since its structure consists of three parts: header, payload, and signature. The header typically consists of two parts: type of the token, which is JWT, and the hashing algorithm being used, such as HMAC SHA256 or RSA. The second part of the token is the payload, which contains the claims. Claims are statements about an entity (typically, the user) and additional metadata. The last part is signature which can be created by taking the encoded header, the encoded payload, a secret, the algorithm specified in the header, and finally sign that.

The implementation of security on GraphQL can be seen in Figure 18. From this figure, we can see a new class called User is created. This class have three attributes namely id, username, and password. For testing purpose only, we created two users, frans and stefan in which each user has password testingapi.

We also created two new methods called authenticate and identity. The method authenticate will validate the user account while identity returns the user object. We placed function jwt_required() inside graphql_view to authorize each request to API.

```
class User(object):
    def __init__(self, id, username, password):
        self.id = id
        self.username = username
        self.password = password

    def __str__(self):
        return "User(id='%s')" % self.id

users = [
    User(1, 'frans', 'testingapi'),
    User(2, 'stefan', 'testingapi'),
]

username_table = {u.username: u for u in users}
userid_table = {u.id: u for u in users}

def authenticate(username, password):
    user = username_table.get(username, None)
    if user and safe_str_cmp(user.password.encode('utf-8'), password.encode('utf-8')):
        return user

def identity(payload):
    user_id = payload['identity']
    return userid_table.get(user_id, None)


app = Flask(__name__)
app.debug = True
# auth = HTTPBasicAuth()
app.config['SECRET_KEY'] = 'super-secret'
jwt = JWT(app, authenticate, identity)

def graphql_view():
    # print ("verify_password", auth.verify_password)
    view = GraphQLView.as_view('graphql', schema=schema, context={'session': db_session}, graphiql=bool(app.config.get("DEBUG", False)))
    view = jwt_required()(view)
    return view

app.add_url_rule(
    '/graphql',
    view_func=graphql_view()
)

@app.route('/protected')
@jwt_required()
def protected():
    return '%s' % current_identity


@app.route('/')
# @auth.login_required
def index():
    return ("Go to /graphql")

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

Figure 18. Implementation of security on Graphql API on top of Flask framework

When accessing method on Graphql API, firstly we should request a token by attaching username and password and then the API will reply with an access token as described in figure 19 and figure 20 respectively.

```python
import json
import urllib2
import requests


data = {
        "username": "frans",
    "password": "testingapi"
}

response = requests.post('http://127.0.0.1:5000/auth', json=data)

# For successful API call, response code will be 200 (OK)
if(response.ok):
    json_data = json.loads(response.content)
    print(json_data)
else:
  # If response code is not ok (200), print the resulting http error code with description
    response.raise_for_status()
```

Figure 19. Implementation of requesting token from Graphql API


```
{u'access_token': u'eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZGVudGl0eSI6MSwiaWF
0IjoxNTEyNDg0ODc0LCJuYmYiOjE1MTI0ODQ4NzQsImV4cCI6MTUxMjQ4NTE3NH0.WUowHiOOwnN0dj4
iPIANVvLYCLi1w9RUSnNEveArazs'}
```


Figure 20. Access token from Graphql API

Moreover, we used the above access token to access specific Graphql API. Figure 21 and figure 22 show authentication with valid and invalid token respectively.

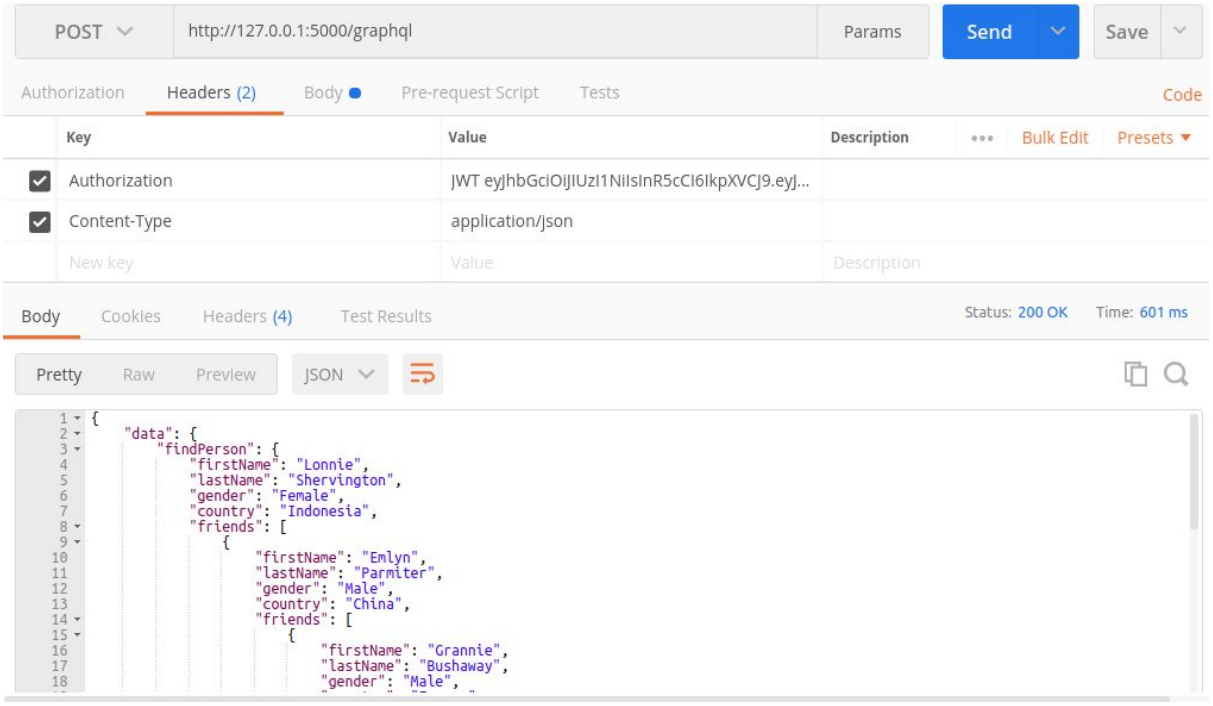


Figure 21. Authentication with valid token

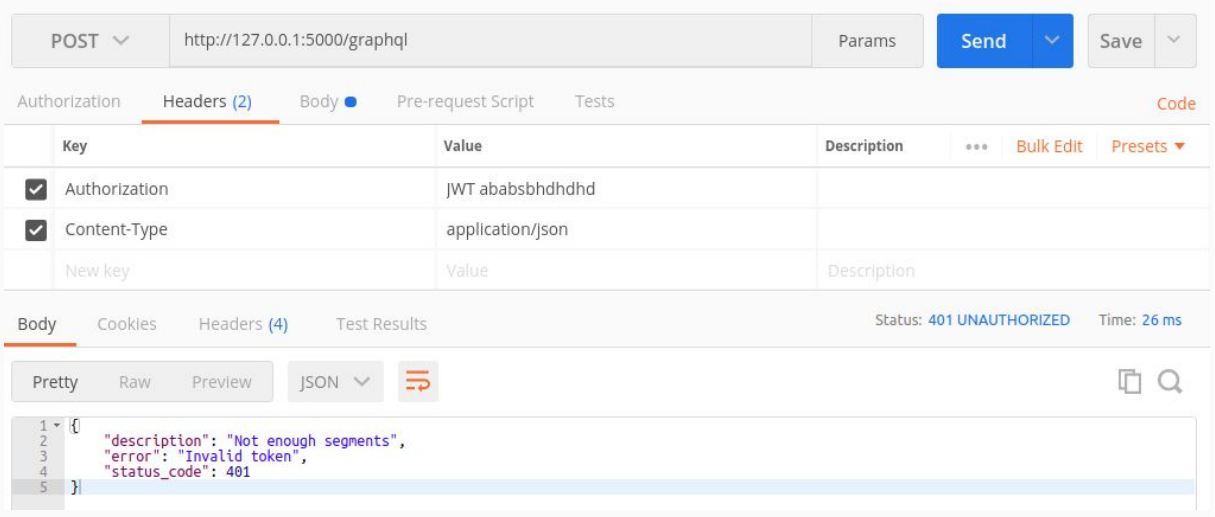


Figure 22. Authentication with invalid token

3.3.3 Load Balancing

The load balancing is tested for only one scenario, as the strategy behind assessing it is to perform multiple similar queries on the same data, and afterwards to measure the response time. By this way, the performance in responding to multiple queries can be deducted. However, in real cases, the multiple queries can be of different types, but we only considered those of the same time for easiness of implementation of the test classes. Also, by applying this strategy, we can observe the behaviour and measure performance of the APIs when reading the same data in a relatively short time. This implies calling same method, addressing the same object model and, subsequently, reading from the same location in the database.

We used a very popular software so-called Apache JMeter to test the performance of each API when being accessed by 100 parallel requests[1].

➢ ***RESTful API***

We simulated this API using 100 concurrent threads HTTP Request in which the duration of time that JMeter will distribute the start of the threads over is 1 second. The detail of the test can be seen in figure 23.

[1] http://jmeter.apache.org/

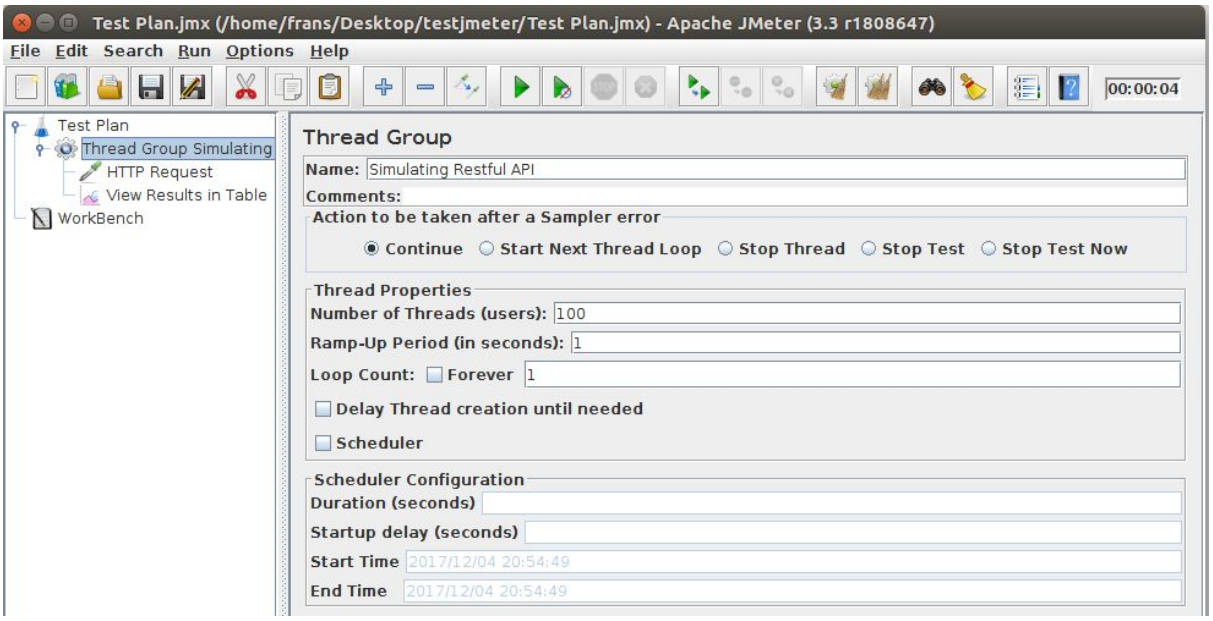


Figure 23. Configuration of Restful thread groups on JMeter

We create HTTP Request Sampler in order to tell JMeter which API should it tests. Here, we can define the IP of Restful API including port, request method, and path as shown in figure 24. From this figure, it can be seen that the url which was being simulated is http://127.0.01:5000/person/1?allfield=1&includingfriends=0 (scenario 1).

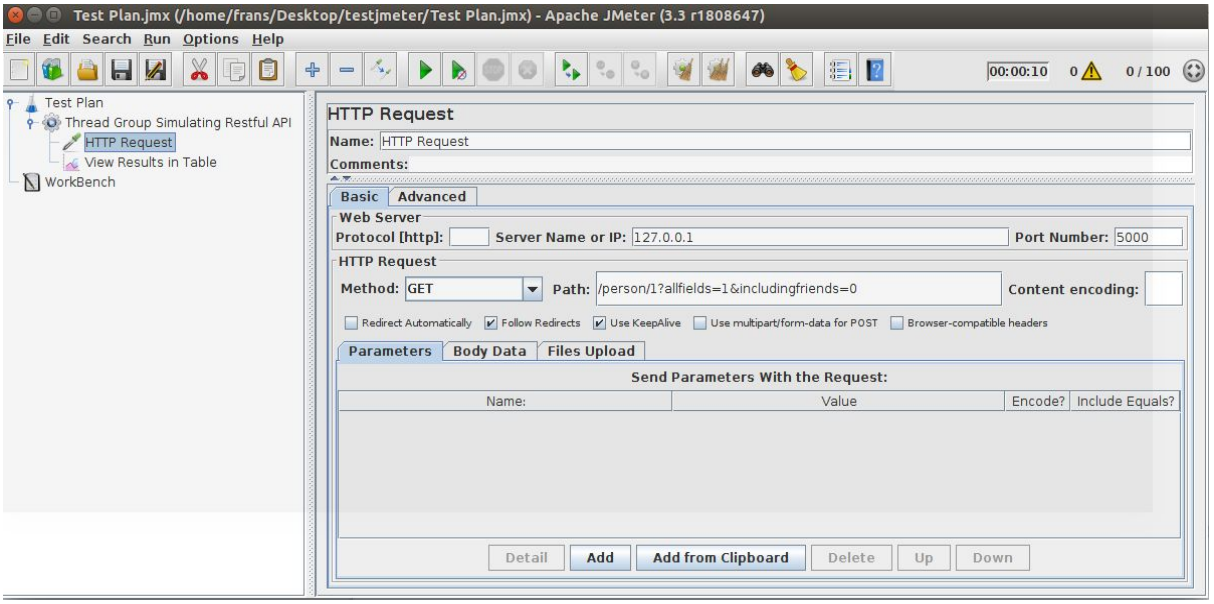


Figure 24. Configuration of HTTP Request Sampler Restful API

The output of this test is a table which shows **sample time** (the number of milliseconds that the server took to fully serve the request (response + latency)) and **latency** (the number of milliseconds that elapsed between when JMeter sent the request and when an initial response was received). From figure 25, we can see that the average time of Restful API to serve request on each thread is 62 ms.

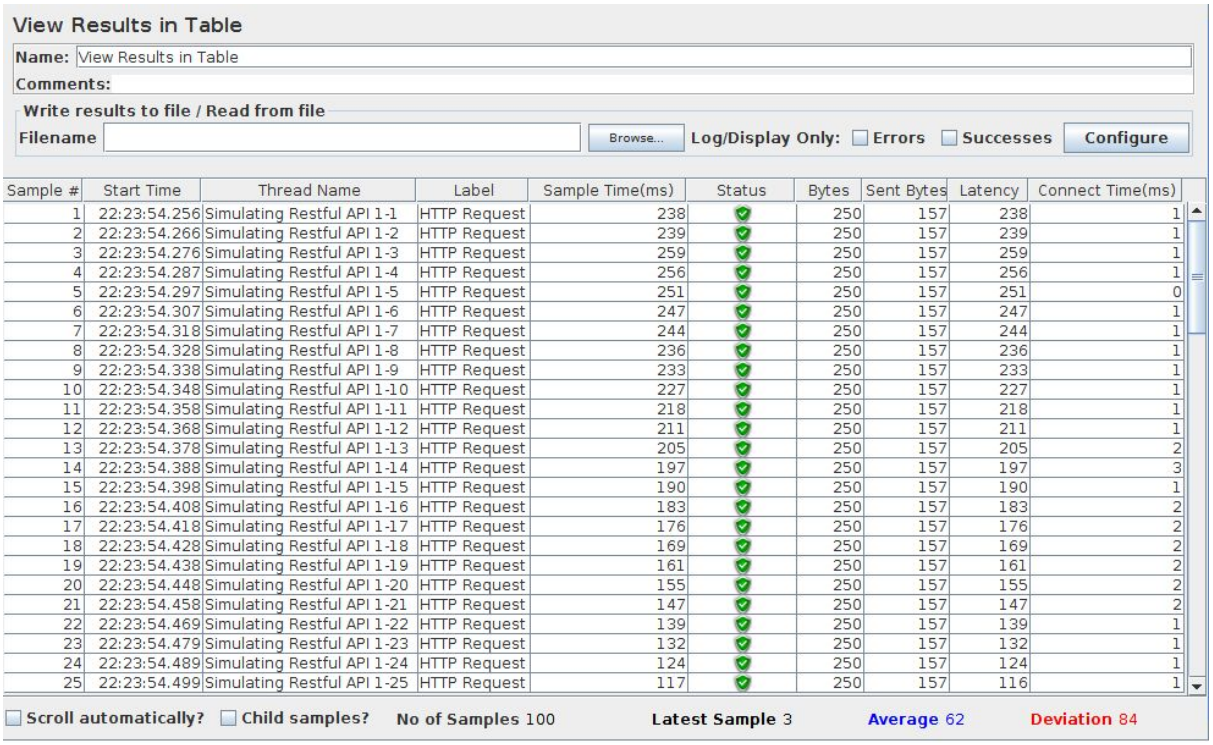


Figure 25. The output of JMeter with 100 concurrent threads

➢ *Graphql API*

We also simulate the performance of Graphql using API using 100 concurrent threads HTTP Request in which the duration of time that JMeter will distribute the start of the threads over is 1 second. The detail of the test can be seen in figure 26.

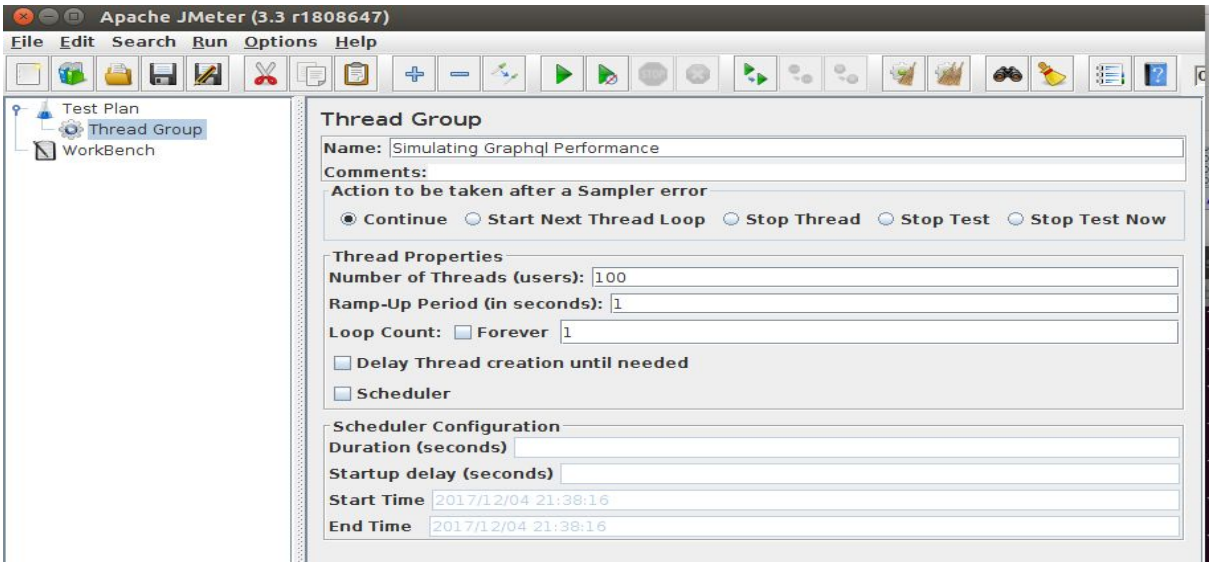


Figure 26. Configuration of Graphql thread groups on JMeter

Again, we create HTTP Request Sampler in order to tell JMeter which API should it tests. Here, we can define the IP of Graphql API including port, request method, and path as shown in figure 27. From this figure, it can be seen that the url which was being simulated is http://127.0.0.1:5000/graphql with query of scenario 1 as body of the request.

```
{
 findPerson(id:"1"){
        firstName
        lastName
        gender
        country

 }
}
```

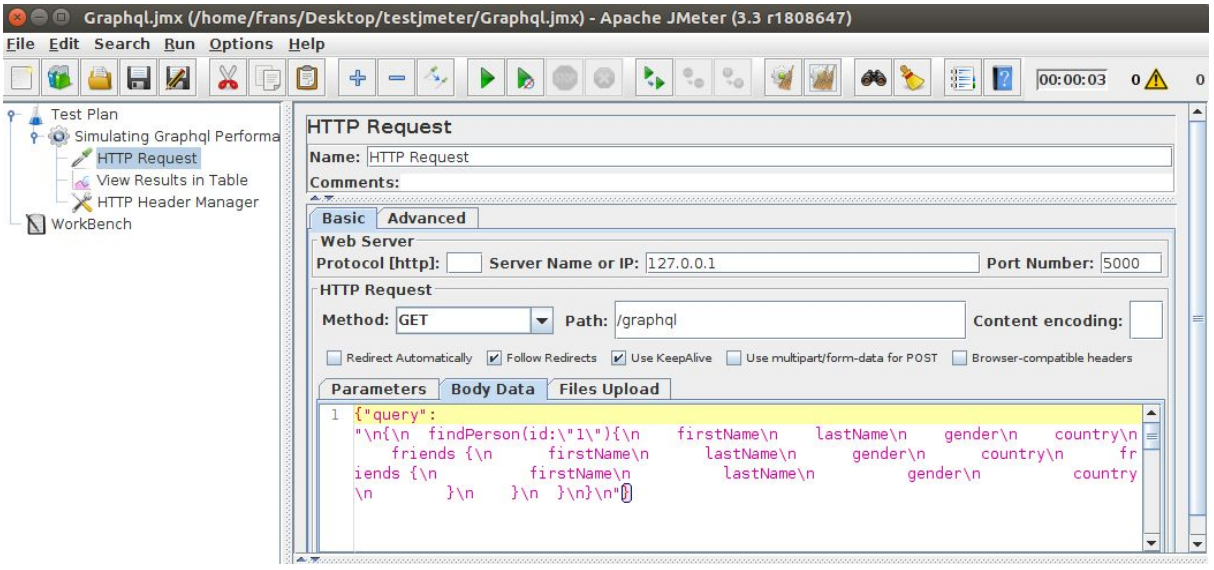
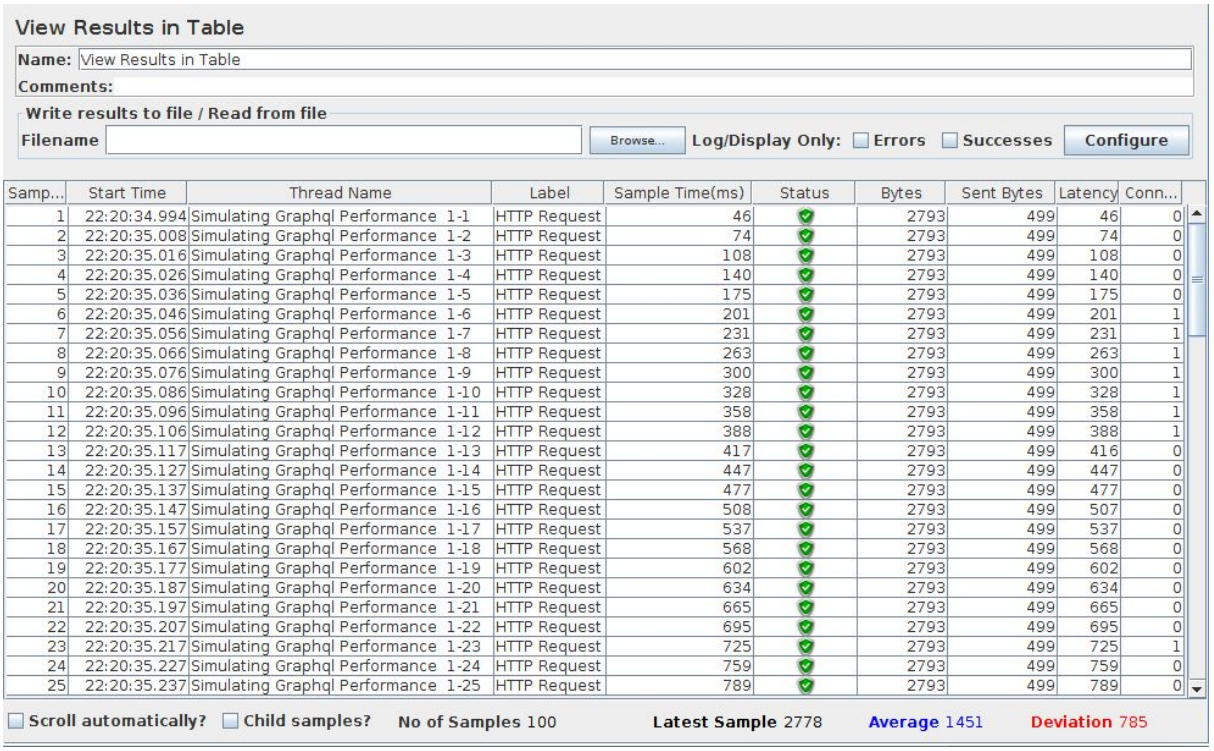


Figure 27. Configuration of HTTP Request Sampler Graphql API

The output of this test is a table which shows **sample time** (in millisecond) and **latency**. From figure 28, we can see that the average time of Graphql API to serve request on each thread is 1451 ms.



Samp...	Start Time	Thread Name	Label	Sample Time(ms)	Status	Bytes	Sent Bytes	Latency	Conn...
1	22:20:34.994	Simulating Graphql Performance 1-1	HTTP Request	46		2793	499	46	0
2	22:20:35.008	Simulating Graphql Performance 1-2	HTTP Request	74		2793	499	74	0
3	22:20:35.016	Simulating Graphql Performance 1-3	HTTP Request	108		2793	499	108	0
4	22:20:35.026	Simulating Graphql Performance 1-4	HTTP Request	140		2793	499	140	0
5	22:20:35.036	Simulating Graphql Performance 1-5	HTTP Request	175		2793	499	175	0
6	22:20:35.046	Simulating Graphql Performance 1-6	HTTP Request	201		2793	499	201	1
7	22:20:35.056	Simulating Graphql Performance 1-7	HTTP Request	231		2793	499	231	1
8	22:20:35.066	Simulating Graphql Performance 1-8	HTTP Request	263		2793	499	263	1
9	22:20:35.076	Simulating Graphql Performance 1-9	HTTP Request	300		2793	499	300	1
10	22:20:35.086	Simulating Graphql Performance 1-10	HTTP Request	328		2793	499	328	1
11	22:20:35.096	Simulating Graphql Performance 1-11	HTTP Request	358		2793	499	358	1
12	22:20:35.106	Simulating Graphql Performance 1-12	HTTP Request	388		2793	499	388	1
13	22:20:35.117	Simulating Graphql Performance 1-13	HTTP Request	417		2793	499	416	0
14	22:20:35.127	Simulating Graphql Performance 1-14	HTTP Request	447		2793	499	447	0
15	22:20:35.137	Simulating Graphql Performance 1-15	HTTP Request	477		2793	499	477	0
16	22:20:35.147	Simulating Graphql Performance 1-16	HTTP Request	508		2793	499	507	0
17	22:20:35.157	Simulating Graphql Performance 1-17	HTTP Request	537		2793	499	537	0
18	22:20:35.167	Simulating Graphql Performance 1-18	HTTP Request	568		2793	499	568	0
19	22:20:35.177	Simulating Graphql Performance 1-19	HTTP Request	602		2793	499	602	0
20	22:20:35.187	Simulating Graphql Performance 1-20	HTTP Request	634		2793	499	634	0
21	22:20:35.197	Simulating Graphql Performance 1-21	HTTP Request	665		2793	499	665	0
22	22:20:35.207	Simulating Graphql Performance 1-22	HTTP Request	695		2793	499	695	0
23	22:20:35.217	Simulating Graphql Performance 1-23	HTTP Request	725		2793	499	725	1
24	22:20:35.227	Simulating Graphql Performance 1-24	HTTP Request	759		2793	499	759	0
25	22:20:35.237	Simulating Graphql Performance 1-25	HTTP Request	789		2793	499	789	0

Figure 28. The output of JMeter with 100 concurrent threads

4. SWOT Analysis

SWOT stands for Strengths, Weaknesses, Opportunities and Threats. The first 2 categories contain internal factors., whereas the last two contain external ones. Also, the first and the third categories highlight the positive aspects of a possible solution to a problem, that are helpful to achieving the objectives, whereas the second and the fourth represent factors that might be harm the solution's implementation and execution [4].
That said, in this chapter we aim at assessing those factors with respect to the two APIs we used for experiments described in the previous chapter. These factors are meant to correlate the theoretical description of these technologies to the result we present in chapter 3. The purpose of doing this analysis is to find the advantages and the drawbacks of these APIs and to determine which is better to use and why.

4.1 RESTful SWOT analysis

	Helpful to achieve the objective	Harmful to achieve the objective
Internal origin	Strengths: ● The REST API totally separates the user interface from the server and the data storage which improves the portability of the interface to other types of platforms and allows the different components of the developments to be evolved independently. ● It's easy to implement since it does not require additional libraries or other dependencies ● It can be secured using authentication so only those who are eligible to access the API ● Implementing security is not as painful as Graphql. Normally the implementation of security already covered by framework. Hence, we just need to adapt it accordingly	Weaknesses: ● It is less suitable for applications that change very often ● It's less scalable compared to Graphql since it involves doing a lot of work when dealing with a new requirement such as tweaking existing methods and updating parameters request accordingly ● Not all of existing methods can be reused. Sometimes we need to create a new method in order to satisfy a requirement ● It relies on parameter request to distinguish different queries which tend to be set incorrectly by requester. ● The parameters query are not as flexible as Graphql

	• Restful API is faster than Graphql. It can be seen from its performance during load balance testing in section 3.3.3. • It is cacheable, so if we were worried about latency we can save bandwidth caching responses from the server.	
External origin	Opportunities: • It is compatible with multiple programming languages • RESTful API always adapts to the type of syntax or platforms being used, which gives considerable freedom when changing or testing new environments within the development	Threats: • The security part is mostly delegated to external services, but is easy to implement • Parameter requests tend to be maliciously attacked by hacker. Encrypted parameters are needed for sensitive information. • Since Graphql comes with a lot of improvements, therefore Restful API might be less popular in the future

Table 4.1 Restful SWOT analysis

4.2 GraphQL SWOT analysis

	Helpful to achieve the objective	Harmful to achieve the objective
Internal origin	Strengths: • Allows for an easier evolution of the API • Does not involve changes in URL, as it always starts looking from the same URL, which represents the entry point in the graph and function calling • Defining types allows for a clear structure of the API • Its graph structure easily allows for creating simple queries when searching for nested data • Queries can be served through HTTP, which is widely known and used,	Weaknesses: • Involves defining a structure which includes types and their fields, as well as the functions (resolvers) used for retrieving the values for the desired fields, which might take some time to learn to do and to implement • The defined structure allows for searching only for defined types and not necessarily for all existing data in the database • Allows for trivial security checks, which might not be enough and which is not seen as a best practice • Involves doing a lot of work by defining schema and integrate the

	making the API development easier ● It may allow for secure access, through a certain type of verification, described in section 2.2	libraries, which is not time efficient when the client want to perform only simple queries
External origin	Opportunities: ● It is compatible with multiple programming languages ● Delegating the security task to external services might be good as there can be used specialized services for this, which normally perform more thorough verifications than GraphQL would do as a consequence of the separation of concerns (low coupling, high cohesion)	Threats: ● The security part is mostly delegated to external services, and the one we presented JWT proved to be not that easy to implement and integrate ● Lower speeds in data access when compared to RESTful

Table 4.1 GraphQL SWOT analysis

Which API should be chosen and why?

When compared to RESTful API, GraphQL involves the using and the integration of an additional library called graphene, which might create dependencies problems in the cases of some programming languages and platforms. However, it was not the case for python and actually, it made it easier to use GraphQL afterwards. Moreover, as mentioned in its SWOT analysis, it involves creating a schema, with types and resolvers, that can be time consuming, especially when only simple queries are meant to be performed.

Regarding data accessibility, in case of the 1st scenario more work was done in the case of GraphQL, by implementing the schema and the resolvers. The execution time for GraphQL was almost 6 milliseconds, whereas for RESTful it was ~4 milliseconds. In the second scenario, the execution time when using RESTful API slightly increased by almost 0.2 ms whereas it decreased by the same amount for GraphQL. Both these scenarios did not include any nested queries. In the case of the 3rd scenario, the response given by RESTful API took 10 ms, whereas GraphQL response came in 16 ms. Finally, in the last scenario, RESTful API returned the response within almost 33 ms, in comparison to the 38 ms of GraphQL.

Nevertheless, for each new scenario there was not needed to modify existing code or schema, whereas in the case of RESTful API we had to add new parameters or to modify existing

ones for the used URLs. Also, in each of these 4 cases GraphQL did not involve changes in URL. Thus, these aspects make it more usable.

In addition, RESTful API's performance when serving multiple simultaneous requests proved to be better, as in the same scenario, it executed thea task in 62 ms, whereas GraphQL needed 1451 ms, that makes the former 23 times faster. It can be noticed that constantly GraphQL was slower that RESTful API, both when testing the 4 scenarios and when testing load balancing performance.An explanation for this can lie in the time spent on traversing the graph, in the case of latter. So, it can be said that in terms of load balancing, RESTful is the winner.

When it comes to security, it is easier to implement it with RESTful, but it is over HTTP which might not be always enough. Though delegating this responsibility to another service might be a hassle in terms of development and integration, this separation of concerns can lead to a more powerful security layer in the case of GraphQL.

That said, we would recommend the usage of RESTful API only when it is known from the beginning that only unnested queries shall be made. However, most of the queries nowadays are often embedded, also, in many cases, it is hard for a developer to estimate what kind of queries will an user do. Therefore, the best choice would be GraphQL, mainly for its increased usability, even though its time in data accessing is, in general, higher.

5. Learning Experiences

Based on experiment we performed during this assignment, we found out developing Restful API is much easier than GraphQL since we do not need to add any additional libraries and to create another model as we did for GraphQL API. However when it comes to scaling the API, we have to do a lot of works on Restful API such as tweaking existing method or maybe creating a new one for a very complex query. We also need to adjust the url by adding some parameters which were used to query data. On the other hand, Graphql comes in handy. Even though building GraphQL API from the scratch is not as simple as Restful API, however we can benefit from it later on especially when scaling the API. We do not need to modify the url parameters since it does not need so. Also, we had less work on modifying existing method which was not as painful as Restful API. GraphQL is an excellent choice if our application consists of nested queries.

Moreover, we found it a bit difficult when implementing security on GraphQL API. In the first place we could not find a proper library and example on internet as guidance. We had difficulties to follow several tutorials on internet since none of them succeed. However eventually we managed to implement security with many trials and errors. Conversely, implementing security on Restful API was easy. We just have to extend existing HTTP Basic authentication provided by the Flask framework and put an annotation `@auth.verify_password` on top of each API method.

References

[1] http://searchmicroservices.techtarget.com/definition/RESTful-API accessed on December 3rd 2017
[2] http://gtjourney.gatech.edu/gt-devhub/documentation/restful-api-structure accessed on December 3rd 2017
[3] https://www.sqlalchemy.org/ accessed on December 3rd 2017
[4] http://ctb.ku.edu/en/table-of-contents/assessment/assessing-community-needs-and-resources/swot-analysis/main. Accessed on 3rd of December
[5] http://graphql.org/. Accessed on 4th of December.