Bachelor Degree Project

UNIVERSITY OF SKÖVDE

Performance analysis of Web Services

Comparison between RESTful & GraphQL web services

Bachelor Degree Project in Computer Science

Arnar Freyr Helgason

Supervisor: Mikael Berndtsson
Examiner: Yacine Atif

Table of Contents

1. Introduction

In today's interconnected world, we as users constantly demand more information to be accessible from the web. Not only should the data be accessible but a crucial factor is that load times should be fast. With the internet expanding to more devices of different types such as smartphones, tablets, IoT devices and more, this factor becomes even more important.

Another factor is that the flow of information needs to be as responsive and accessible over a whole different range of network platforms and network infrastructure that are in currently use. Devices such as mobile devices on mobile networks, whether it be 4G, 2G, or WiFi all need to be equally considered when building a web application (Cederlund, 2016).

This is where data transfer web services are utilized. These services are present in most websites, applications and other types of internet services, they are there to make the flow of information and the transition between changes in presented information more smooth and more responsive. Without these services, data could not be exchanged in an asynchronous manner and the end products would become more static than dynamic. Since most services on the web today do project some data or information to its end-users, this free-flow of data is extremely important.

All websites that are dynamic and contain dynamic data and information have a database storage at its backend most often in the form of SQL**(S**tructured **Q**uery **L**anguage**)** storage solutions, or NoSQL. Although different database engines, and database types may affect the overall performance of a website, the focus of this study will not be in the differences of these storage solutions, rather the techniques and services that are used to get that data from the server to the client.

Typically server-side scripting languages such as PHP communicate directly with the database and then output the fetched data as a website which will be delivered to the client. This method may not always work as it is lacking in the dynamic feature of data fetching, which makes for a harder *"on the* fly" approach when it comes to the content of web applications.

RESTful services are in widespread use today. These services provide web applications with functions to easily access data between client and server without "*talking"* directly to a database, separating the client from the data storage techniques which the server should manage without any direct interaction from the client itself ("Fielding Dissertation: CHAPTER 5: Representational State Transfer (REST)," n.d.). This gives the option to request a certain data and a specifically formatted and chosen data is returned to the client, allowing the received data to be arranged and utilized on the client itself, and reducing the load on the server itself as it does not have to recreate a full website with the data that should be displayed.

There exist many more services than just the REST API or RESTful services. GraphQL is another data fetching web service that is on the rise, other services such as former technologies as SOAP which was the predecessor to REST, Netflix has its own services of data exchange (Cederlund, 2016). The experiment carried out in this thesis will take a look at the current RESTful services and GraphQL and the performance differences between specifically those two.

A typical response from these services whether it is REST or GraphQL, is in the format of JSON data. Although other response formats can be used such as XML, the JSON format has been adopted as one of the standards for on-the-fly data response format. More independent and detailed explanation of both RESTful API's, GraphQL, and their corresponding technologies will be covered in the following chapter.

A study in the form of an experiment will be conducted that will determine which of the technologies in question will perform better in the developed test cases. The test cases will be designed to mimic real life situations of data transfer between clients and a server.

2. Background

In this chapter, the background of each of the main technologies that we will be using in this study will be covered. The most important information of each of these technologies will be presented for a better understanding of the underlying point of this study and functionalities of the corresponding technologies that all come together to form the basis of the experiment.

2.1. REST API

REST(Representational state transfer) is an **API**(**A**pplication **P**rogramming **I**nterface) which provides client-server communications for Web Applications over HTTP protocol, making it easily acceptable since it is not bound to any particular transfer protocol. The three main design principles of REST are *addressability, uniform interface,* and *statelessness (Belqasmi et al., 2012)*. REST addresses acceptability by defining *endpoints* in a directory structure (Choi, 2012) via different URI for extracting the data. The API works on the principle of **CRUD(C**reate, **R**ead, **U**pdate, **D**elete), which correspond to the most popular functions (Belqasmi et al., 2012) INSERT, SELECT, UPDATE, and DELETE, in persistent data-storages such as SQL.

Calling a RESTful service over the HTTP protocol can be done in multiple programming languages. In jQuery, a framework of Javascript, a REST server can be called from an Ajax query. A common response of data is in the form of JSON(**J**ava**s**cript **O**bject **N**otation) data. Typical JSON response can be seen in **Figure 2.1.** Data can be returned as well in the form of XML(eXtensible **M**arkup **L**anguage) (Choi, 2012).

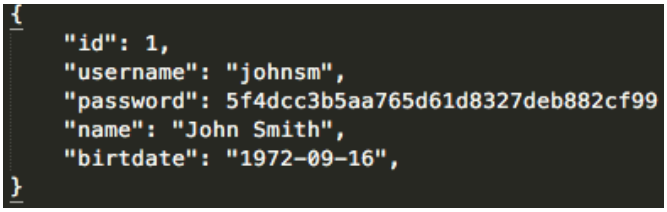


Figure 2.1: Typical JSON response format

Since REST is built on multiple endpoints for specifying the return data, often multiple endpoints need to be called which in turn will increase the number of client-server calls that a Web Application needs for displaying the data to the user. This could possibly result in poorer performance of the service (Cederlund, 2016). In today's society this could have negative effects as Web Services and Applications are being accessed over multiple platforms of devices and networks, ranging from a Desktop Computer with optical fibre to 2G connected Smart Devices.

REST services can be implemented server-side via various technologies. REST can be programmed in most if not all programming languages. For implementing a typical REST server, similar to the one in that will be done in this research, a programming language that is capable of communicating with an SQL database is needed, as well as a way to interact with HTTP requests. For example, a language such as PHP can be used for a REST service since HTTP requests such as GET and PUT are implemented in PHP. Meaning that a client can send a request to a server for a specific endpoint, then the server can interpret that to the actual SQL query statement and communicate with the

backend SQL database. The database returns the data to PHP and in-turn PHP can return the requested data to the client that sent the original request.

Note however that never in the RESTful manifest does it state that for fulfilling the REST constraints do you need to communicate to an SQL server, as the actual storage and the location of the data has can be stored in various ways. Figure 2.2 below gives an example of how an HTTP endpoint is translated into an SQL query on the REST server.

```
router.get("/users",function(req,res){
    var query = "SELECT id, alias, firstName, lastName, email FROM ??";
    var table = ["people"];
    query = mysql.format(query,table);
    connection.query(query,function(err,rows){
        if(err) {
            res.json({"Error" : true, "Message" : "Error executing MySQL query"});
        } else {
            res.json({"people" : rows});
        }
    });
});
```

Figure 2.2: REST endpoint for getting all users

An example of the usage of REST services today could be a case of a company wanting to allow its clients to access some of its data on a RESTful server in a a simple manner. The endpoints can be programmed for the specific cases allowing for a better transaction of data between them. An Irish based company implemented their own RESTful service for their customers to increase the accessibility of their eCommerce platform. The conclusion of that study was met with great success and further work was planned at the time to make the API reach further than just their established customers (Foping et al., 2013).

2.2. GraphQL

GraphQL is an open source query language for API's developed by Facebook Inc. GraphQL executes queries server-side and returns only the data that is defined by a type system in the corresponding Web Service. Available variables and fields for querying are defined in so called *schemas* which are located server-side ("GraphQL," n.d.). Specific queries can be constructed based on the GraphQL services available that have pre-defined the available data for querying allowing for a single endpoint rather than multiple endpoints. By declaring fields we can for example fetch only the name and data of an article instead of receiving everything related to the article and sorting through the data and only displaying the needed fields, this is displayed in Figure 2.3 where a GraphQL query asks for a specific person and specific fields that relate to that person. By eliminating the numbers of queries and the amount of data that is transferred, the transfer speed of the data could potentially be improved over the wide variety of different network connections that are in use of today.

```
{
  people(id:3) {
    firstName
    lastName
    email
  }
}
```

Figure 2.3: Example GraphQL query

GraphQL on the server-side needs to include what has been mentioned before, *schemas*. Those schemas can then be translated into the query answers. In the case of this experiment, the schemas are there to represent the SQL tables and the columns that it contains. A GraphQL object is created for each table representation, containing the fields of the actual tables. Seen in Figure 2.4 is a small example of a schema, where an actual SQL table is represented as a GraphQL object, containing a field for specifying the id field of said table. Further definitions of all schemas used in this experiment can be seen in the appendix.

```
const Product = new GraphQLObjectType({
  name: 'Product',
  description: 'This is a product',
  fields: () => {
    return {
      id: {
        type: GraphQLInt,
        resolve(product) {
          return product.id;
        }
      },
```

Figure 2.4: Definition of a GraphQL Object

GraphQL servers are often installed and configured using the Facebook's React javascript library (“A JavaScript library for building user interfaces - React,” n.d.), as well as the Relay javascript framework (“Thinking in GraphQL | Relay Docs,” n.d.). The description of those two solutions are beyond the scope of this research paper, as not much implementation is needed to get a GraphQL server up and running. The React code that is used in this paper can be seen in the Code sections of the appendix.

2.3. jQuery

jQuery is a Javascript library for use on client-side webpages and web applications. jQuery developers claim that its best features is that it is fast, lightweight and feature-rich library that is based on Javascript code (jquery.org, n.d.). As jQuery is built on Javascript it utilizes original features in the language and simplifies many commands and functions for ease of use for developers and users. The library can be used in various ways such as, traversing the DOM to create, delete, and update elements on websites, which in turn gives the option to easily manipulate data right in-front of the user.

jQuery can do much more than just manipulating the HTML DOM, for example add and remove CSS classes, detect and run code following user interactions with the website. In this experiment, the DOM manipulation property of jQuery and Javascript will mainly be used to present data.

Javascript being client-side, all the content can be interacted with on the fly, meaning that unlike PHP for example which is generally used as a server-side scripting language (php.net, n.d.) , a page refresh is not needed to present the updated site version to the user, possibly improving the User Experience of the website. In this experiment jQuery will be used in combination with its Ajax functionality calls to interact with the RESTful service that will be used.

2.4. Ajax

Ajax (**A**synchronous **J**avascript **a**nd **X**ML) is a technology that is used to fetch data from different servers. As with jQuery and Javascript with manipulation of the DOM, this fetching of data can be done without page refreshes ("Wrox Article : What is Ajax? - Wrox," n.d.). Ajax utilizes the XMLHttpRequests API, which is the basis for the transfer of data between a client and a server ("XMLHttpRequest Level 1," n.d.). Data response is bound to one type, typically the response data can be in the format of JSON, XML, HTML and simple text. This allows the XMLHttpRequest to serve as the basis for Ajax's asynchronous communications and allows RESTful services to respond to a client in a manner of JSON or XML data as is most common.

Ajax itself is not only the requesting of data through the use of XMLHttpRequest. As coined by the author Jesse James Garrett, Ajax is a collection of several technologies ("adaptive path, ajax," 2008). The technologies that Garrett mentions are the following.

- Standards-based presentation of XHTML and CSS
- Dynamic display and interaction using the Document Object Model (DOM)
- Data interchange and manipulation using XML and XLTS (also JSON)
- Asynchronous data retrieval using XMLHttpRequest
- Javascript binding everything together

2.5. SQL

Structured Query Language or SQL as it is known, is a programming language that is used for managing data in relational databases. SQL is one of the most widely used language today regarding databases and data storage solutions. The main structures of SQL are based on the following.

- **Tables**, where all the data is stored, a single database may consist of multiple tables
- **Columns**, representation of each column in a table, this could be such as a column of *usernames* in a table
- **Rows**, represents a row in a table, a row can consist of many columns, such as whole information for a single user

Id	Name	Username	active
1	John Smith	Johnnysm	true
2	Tim B. Lee	TimLee	true

A simplified SQL table example

The connection of tables inside a database can be acquired through the use of **Foreign Keys.** These are specific fields that connect a row in one table to a row in another table. An example of this could be the linking products of a store chain to a certain store, a single product might only be available in certain stores. This constraint can be done by linking the id of a store to that specific product. This allows for searching of specific products in a specific store in the database.

The main statements that can be executed on a SQL database are the following.

- **SELECT**, a select statement will return specifically requested data based on constraints that a query can utilize
- **INSERT**, an insert statement can be used to insert data into a database, either user specific data or auto-generated data such as dates, times, numbers, and so on can be used. The columns or *fields* to be inserted can be specified in the statement itself, allowing for a high degree of flexibility of data inserts
- **UPDATE**, an update statement is used for updating some existing data in the database. This statement usually is utilized with the **WHERE** clause, which gives the option to finely pinpoint for example a specific user with a specific ID number in the database. Calling an update statement on a table without the where clause will allow for a table wide update of columns
- **DELETE**, the delete statement allows for the removal of data from a database. As with the UPDATE statement, this usually is accompanied by a **WHERE** clause to delete a specific record from the database. A where clause is not necessary and is used in cases where a whole set of data in a table is to be removed (“MySQL, MySQL 5.7 Reference Manual 14.2.2 DELETE Syntax,” n.d.)

2.6. Node.js

Node.js is an event driven JavaScript runtime environment designed to give the programmer a way to construct scalable network applications (“About | Node.js,” n.d.). Designed with latency in mind and focusing heavily on HTTP applications makes Node.js a great contender for developers that are building network applications today. As touched on later in this research, node provides a vast library of additions that can be used with Node.js for improving functionality, making it very scaleable. As node is event driven, it does not focus on the useage of the I/O model thus allowing it to be non-blocking and a lightweight runtime environment (“Node.js,” n.d.).

Node.js is a runtime server which apps and websites can be built on. In this experiment Node.js is used to serve as the backend. Chosen for its compatibility with JavaScript and GraphQL (“GraphQL,” n.d.). Although GraphQL itself can be used with many other programming languages, Node.js was chosen since the test environment will contain a simple way of testing webpages rather than an application like such that could be written in Java, or other languages.

As both REST and GraphQL are available on a multitude of programming languages, it allows for this same experience to be replecated on other programming platforms.

Node.js takes advantage of the **npm** javascript package management system. This system allows developers to include libraries and other packages in a simple way in their projects (“npm,” n.d.). Various packages are included in this experiment, such as nessicary packages to access GraphQL's code library (“GraphQL,” n.d.).

3. Problem Description

3.1. Problem

As websites, applications, and services are getting more and more popular in today's inter-connected world through the internet a good standard is needed for transmitting data efficiently over a variety of platforms. The massive amount of data that is transferred every minute over the Internet is stunning, a speculation in 2014 suggested that around 300.000 tweets are tweeted over Twitter, 200 million emails are sent, and close to 220.000 photos are shared on Instagram (James, 2014).

Although these numbers are not from scientific sources, we can just imagine how much data is transmitted and shared between users over the Internet every minute. With sharing data becoming increasingly more popular on dynamic sites such as Facebook, we as developers need to implement new ways to get this data to the end-user as efficiently as possible.

This is where RESTful services come to play. Data can be transmitted in an efficient way and returned to either server-side or client-side in XML, JSON, or even other types of formats upon request in a stateless manner. This allows for many services such as Facebook or Google to implement their own RESTful services to serve their customers, for example other companies can implement Google search in their own Web Application ("Custom Search JSON/Atom API | Custom Search," n.d.).

Are developers and companies sharing data by using RESTful services as efficiently as they possibly as possible?

Take for example an eCommerce company that implements a REST API for its customers for getting detailed information about their products. A client wants to implement their API for being able to list a quick view of their products and it should only contain an image, name of the product, and a price. Many eCommerce companies offer some kind of an API service today, one of them is eBay ("eBay Developers Program," n.d.).

The problem is the response from a REST request gives the client extra information that is not needed at the time such as number of units in stock, reviews by customers who bought the product, and a description. This addition of data creates an extra overhead by transferring unneeded data along with the data that the client needs.

The simplest solution to this would be for the eCommerce company to develop different endpoints that meet all of its clients need for different data. This however could quickly become tedious and expensive as clients come in all shapes and sizes, requesting different responses and different combinations of data, which in turn have to server their own customers that most likely are making requests from different devices and from networks that have a wide range in performance.

3.2. Hypothesis

The solution that this thesis is going to test and explore is the newly arrived GraphQL. This technology will be compared with the older but widely used REST API. Both the GraphQL and its REST counterpart will utilize a combination of jQuery and Ajax to interact with their respected server-side services. The results of this experiment will reveal the difference between these services in question.

Can the more query oriented language of GraphQL increase the performance of data-flow in the form of JSON responses over the network, and what are the differences in latency and package size compared to the widely used RESTful services that are in use today?

By eliminating overhead of data and letting developers choose the data that is required a possible increase in performance could be seen.

By utilizing the single endpoint design of GraphQL in comparison to the multiple endpoints of REST for combining data from different sources, will the network performance increase?

Those are the main questions that this study and its corresponding experiments will set out to investigate.

3.3. Related Work

GraphQL has been tested against REST and other services, the performance was tested with a well established Swedish news platform, Aftonbladet (Cederlund, 2016). With the purpose of finding out if implementing GraphQL and related services could improve the performance of the platform. With the conclusion being that there is no one solution for all the different test cases that Cederlund tested against. In the case of his research the different cases involved fetching data from different endpoints, returning different amounts of data for each endpoint . Based on the data that is being requested different services performed differently, this difference in data requested could be a simple query for one field such as a heading of an article, or the whole article itself (Cederlund, 2016). This related study does explore different datasets to fetch, with different complexities. However the author did the research in cooperation with a known news agency based in Sweden, this limits the test data to a certain degree not to mention that by collaborating with a private agency, the data that is being used and its structure is private and not directly available for further replication for the exact study with the same data.

Not much more of a similar work has been done on the relation between GraphQL and REST services. Locating scientific sources on GraphQL is very difficult at the time of writing, as GraphQL is still a relatively new service.

4. Method

4.1. The Method

The method that is going to be used to test the performance differences between GraphQL and REST is an experimental method. An experiment is to be designed in the form of a database which contains realistic data, as well as realistic SQL relational database structures such as multiple tables combined with *foreign keys* allowing for more complicated table joins and single table selects which will favour both of the "to be tested" technologies. The same database structure will be used for both technologies so by having the same database with the same data for both, a certain degree of fairness can be ensured, as well as the back-end services should perform the same work on the data to maximise the fairness between the two.

The experiment in question is strictly the performance of the technologies in question. As in, data fetching. This does not include the creation, or displaying of a whole complete website as the goal is to measure the differences between the data fetching services and not the load time of a webpage per say.

The performance differences of the proposed solution will be tested against its counterpart by measuring latency as in how long it takes to fetch the requested data. Fetching data of various sizes and complexities will give a greater scope of functionality. The package size of each individual test will be recorded for a better clarification of where each of the technologies to be tested will outperform the other. Multiple iterations of tests will be conducted on both the technologies for eliminating any odd data such as spikes in network latency.

For the data and conclusion to be realistic multiple tests of multiple iterations will be conducted over a different type of networks. Just by testing on ethernet over a local network will drastically reduce the significance of the results in the real world on real live data.

The break-down of steps of how the experiment will be conducted is the following.

- Measure *latency* of delivery
- Measure *data size* of each response
- Run multiple *iterations* of individual tests for reliable data collection
- Run each iteration over *different types of networks*

The design of this experiment should yield results closer to real live data transfer in real world situations.

A conclusion will be made when all the data has been collected and analysed. The conclusions that this experiment provides will be visualized with charts and tables that will scientifically show the performance differences between the technologies in question.

4.2. Reliability

By measuring the latency of response between the proposed solution and the others an overview can be produced that shows the differences, by reducing overhead of unneeded data per request the network latency should be smaller since less data is needed to be transferred between the client and the server.

Same goes for experimenting if a single endpoint can outperform multiple endpoints by eliminating the number of calls. The single endpoint design of GraphQL is a big factor for the developers of the language and their claims of performance against its RESTful counterpart ("GraphQL," n.d.).With this experiment the claims of the GraphQL developers that it should perform faster than other services in on the market today can be tested.

By conducting variety of tests with different sets of data many types of scenarios can be tested and compared. A similar research was conducted in a former study into this matter, where the conclusions gave that in certain cases the proposed solution outperformed its counterpart while underperforming in others (Cederlund, 2016).

4.3. Ethics

As many of these technologies are open-source and no extensive study on them could be found, it is hard to determine their state of functionality and reliability. Various problems might be encountered during the process of this experimentation.

The data that is planned to be used is not real live data and should not contain any sensitive information that could potentially cause harm if released.

All code that is to be written for the experiment will be published with this study, and should not contain any privately owned code that is not under one of the open-source licenses such as the GNU license family.

A completely unbiased mindset has to be acquired and one solution should not be favoured over another.

5. Experiment

The experiment was designed in the way that will best represent the differences of the two technologies of fetching data that lies server side from a client. By further researching articles and past studies about performance measurements of experiments similar to this one, the conclusion was to measure two major factors that will be able to set these technologies apart.

Latency and *data volume, i.e. size* will be measured. This way of testing especially draws heavy influence from the similar research conducted by (Cederlund, 2016) where he too measured latency and data volume to distinguish the differences in performance of GraphQL, REST API, and other services. By conducting the research in this manner we can backup the hypothesis made earlier about the performance of GraphQL and REST API data fetching, and later on back that hypothesis up with scientific data.

5.1. Design

The main structure of the experiment is to measure with the factors mentioned before, and down below in further in-depth manner. The experiment will be set up in the way that we can efficiently measure the performance between GraphQL and REST API by putting them to the test inside a well defined and controlled environment.

Both the pilot, and final experiment will be conducted locally on the same computer, i.e. both the server and and the client are located on the same computer and will communicate to each other internally over the *loopback* interface of the computer.

As traffic is very low on a Localhost as well as on a Local Area Network, a traffic generator of some sort can be introduced to generate and emulate actual traffic to the server. This way, different tests can be conducted based on different traffic load, which in turn should have some effects on the latency between the client and the server. This generation of traffic will however not been used in this experiment, but could be utilized in future experiments and testing.

5.1.1. Latency

Latency is often used when measuring performance differences over of data flowing over a network. Measuring latency or *response time* could be a good indicator for the performance of each technology. In this experiment the response time will be measured from when a request is made client side up until a response has reached back to the client. A conference paper by *Zhou* where the goal was to measure the latency/response time of a REST API service, the experiment was conducted by running multiple iterations with each iteration lasting 1 hour. In each iteration random requests are sent repeatedly at 500ms intervals. The average response time is then recorded at 5 minutes intervals (Zhou et al., 2014).

This approach of testing will be taken as it is very interesting to see the whole time scale of a request-response from client-server-client environment. No further calculations of trying to separate different times such as the server request process time, network delivery time and other times will be taken as the though would be that the response time that will represent the most real situation is the total time of sending a request up to getting a response back. This broad time scale does certainly involve many different components that could be measured independently, however for this experiment the total time will be the defining factor of difference for the technologies that are under testing.

5.1.2. Data Volume

Measuring data volume can be done by measuring the actual size of the response package. As data is received over the network as an HTTP packet, the actual sizes of these packages can be measured and compared. A single HTTP response consists of a header and the message response itself, followed by a message trailer if there are any. The headers themselves consist of useful information about the payload such as the HTTP Status code, i.e. (200,404,500 etc.), entity headers, response headers and so on (Kozierok, 2017). By eliminating these headers, we can calculate the size of the actual data that was transferred from the server to our client, that will be used for the purposes of the application.

A great tool for measuring the raw HTTP package sizes is a packet sniffer. A packet sniffer is program that monitors all the data that goes through a device's network adapter, such as Ethernet, WiFi, or even the loopback address*(127.0.0.1, ::1).* Various tools for packet sniffing or packet analysing exist. Wireshark itself is just one of the available softwares out on the market to analyse packet traffic. The reason why Wireshark was chosen for this experiment, was because of the ease of filtering packets via the graphical user interface.

The sniffer stores both incoming and outgoing packets of the device which in turn can be used for analysis purposes later on (Qadeer et al., 2010). This proves as a powerful tool and it can give all the information needed for this experiment about a transferred packets size and structure. An example of a packet sniffer at work can be seen below in Figure 5.1.

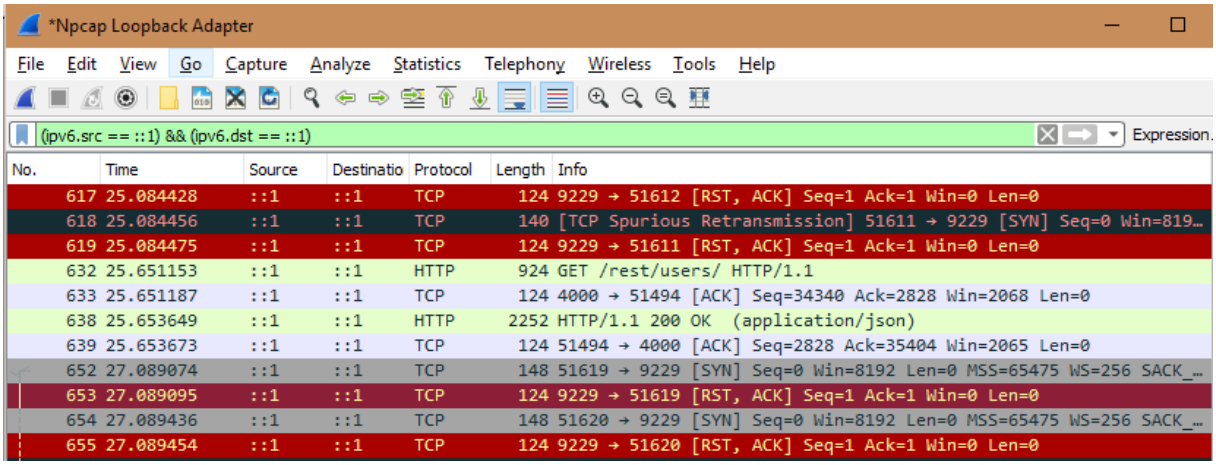


Figure 5.1: Example of a packet sniffer intercepting traffic on loopback

5.2. Implementation

The implementation of the tests will be done via the the various tools described in the *Background* chapter at the beginning of this research, with these tools come various smaller tools or tool-kits that will be used to set up the testing environment. Those smaller tools that are more connected to the implementation itself will be described in this chapter.

The steps that will be taken to implement the testing environment are as follow:

- Setup a Node.js server environment that will handle the connections to the individual servers (*i.e. REST and GraphQL*).
- Setup a GraphQL server that can handle GET requests.
- Setup a RESTful API server that will handle REST related GET requests.
- Setup a MySQL server with tables and datasets that should represent a *normal* database.
- Generate a decent amount of data for the MySQL database for the pilot test, Later on more data can be added for a larger test scale.
- Setup a testing endpoint where the tests can be run and the response time can be measured, store the data for later analysis.
- Use a packet sniffer tool to identify the incoming packets during each test, and store the data for later analysis.

After these steps the pilot testing can be performed to get a good overview of performance differences on a smaller scale than the experiment itself. The pilot testing serves as a base for the experiment itself thus also serving the purpose of testing out all the interconnected elements and to see if everything works as it should before the experiment is conducted.

The final experiment itself will be tested in much the same way as the pilot test. Test variables with different and more complex set of data will be used, while keeping the test environment itself in the same structure. The testing environment will use Ajax in combination with jQuery to query both technologies, REST and GraphQL.

5.3. Environment

5.3.1. Server

All of the traffic during the testing, both to the REST API and the GraphQL servers goes through a Node.js server script. This serves the function of routing the traffic for the specific requests to the right place. If a request comes in from a user that wants to fetch information with the REST API, then the node server looks at the URL and directs the request based on that. The routes for the testing environment is as following.

- /graphql (the root of the graphql server handler)
- /rest (the root of the REST API server handler)
- /pilot (the root of the environment used for testing)

Both the GraphQL root and the REST API root handle direct querying through a raw URL. This allows for great flexibility and allows developers to use GraphQL queries on multiple platforms for example through PHP, ASP.NET and so on. An example of a server route can be seen below in Figure 5.2. The whole code for the server script can be found in Appendix A – Code

```
app.use('/pilot', function (req, res){
  res.sendFile(Path.join(__dirname+'/pilot_test.html'));
});

app.use('/rest', ROUTER);
```

Figure 5.2: Route to 'pilot' and 'rest' interface on the server

As much of Node.js is built and driven on packages, there exists massive amount of packages for use with Node.js, all of which can be installed using their '*npm*' package system (“Node.js,” n.d.). The main packages of the server script have to do with GraphQL and its dependencies, which is used in connection with GraphQL and to create the routes that lead to the API's themselves. Another package that is needed for the testing environment is for the MySQL connection. In total the server thus needs three packages to function in addition to other individual scripts that are the core of each API.

5.3.2. GraphQL Server

The GraphQL server itself is separated into two different scripts, a *schema* script and a *db* script (Database definitions used to server the GraphQL server).

In the schema script the queries for the server are defined, there the available fields are laid out, what data can be fetched and how it all connects to each other. Specifically it is split into queries and GraphQL objects. The tables of the database are described in objects which in-turn are made available through the use of a query. An example schema of a *person* can be seen in Figure 5.3.

```
const Person = new GraphQLObjectType({
  name: 'Person',
  description: 'This represents a person',
  fields: () => {
    return {
      id: {
        type: GraphQLInt,
        resolve(person) {
          return person.id;
        }
      },
      firstName: {
        type: GraphQLString,
        resolve(person) {
          return person.firstName;
        }
      },
      lastName: {
        type: GraphQLString,
        resolve(person) {
          return person.lastName;
        }
      },
      email: {
        type: GraphQLString,
        resolve(person) {
          return person.email;
        }
      }
    };
  }
});
```

Figure 5.3: GraphQL schema description example

As with the server script, the schema scripts depends on packages. The package that is used for the schema itself is the official GraphQL package which was made available for Node.js through the *npm* library. The *db* script also uses packages for functionality, *Sequelize* and *lodash* are used, but rather more for more flexible programming of the code itself. Sequelize is used for the connection to the MySQL database in the case of GraphQL. Sequalize is a promised-based **ORM** *(Object-relational mapping)* for Node.js. Sequelize supports transaction making such as **CRUD** that is mentioned in Chapter 1. (“Sequelize | The Node.js / io.js ORM for PostgreSQL, MySQL, SQLite and MSSQL,” n.d.).

5.3.3. REST Server

The REST server itself is initiated within the server script itself. Apart from that the REST API uses another script that handles the functions of the database connections via the *mysql* package. In the REST API script itself the queries and routes are constructed, and allowing for addtion of extra routes.

For the pilot test the two routes that are defined are the root for fetching all the users as well as a route for fetching a specific user from the database. An example of the route that fetches all the users from the database can be seen below in Figure 5.4.

```
13      router.get("/users",function(req,res){
14          var query = "SELECT id, firstName, lastName, email FROM ??";
15          var table = ["people"];
16          query = mysql.format(query,table);
17          connection.query(query,function(err,rows){
18              if(err) {
19                  res.json({"Error" : true, "Message" : "Error executing MySQL query"});
20              } else {
21                  res.json({"people" : rows});
22              }
23          });
24      });
```

Figure 5.4: REST route for the "/user" endpoint

These routes themselves become the endpoints at which the data is requested from. A call can be made for example with the code above via jQuery or raw HTTP GET request, and the users should be returned to the client in the form of JSON data.

The complete code for the REST server can be found in Appendix A – Code .

5.3.4. Measurements

As mentioned before in chapters 5.1.1 and 5.1.2. The performance will be evaluated by measuring latency and data volume.

The latency will be measured in the way of running two different timers in Javascript, specifically the **new Date().getTime()** function will be used. The first timer will be started when the request will be made on the client and the second timer will be started when the response was been received by the client. As Cederlund mentioned in his research, it is highly interesting to measure the full stack of the request. This means that there is a risk of latency fluctuations at any point during the transmission between the client and the server. Included in his measurements and implemented in this experiment as well are factors such as the time it takes the client hardware to handle the request, the server handling of the request, all the way up until the handling of the response by the client (Cederlund, 2016).

The latency data will be stored in the form of *CSV (see Figure 5.5 for exact format)*, storing the number of the request, start time, end time, time difference, as well as the endpoint URL for future references and accessibility.

	A	B	C	D	E	F
1	Iteration	Start time	End time	REST	Endpoint	
2	1	149167361654	149167361656	14	http://localhost:4000/rest/users	
3	2	149167361706	149167361707	12	http://localhost:4000/rest/users	
4	3	149167361757	149167361758	13	http://localhost:4000/rest/users	
5	4	149167361808	149167361810	15	http://localhost:4000/rest/users	
6	5	149167361860	149167361861	13	http://localhost:4000/rest/users	
7	6	1491673619115	149167361913	18	http://localhost:4000/rest/users	
8	7	149167361963	149167361965	14	http://localhost:4000/rest/users	
9	8	149167362015	149167362016	14	http://localhost:4000/rest/users	
10	9	149167362067	149167362068	14	http://localhost:4000/rest/users	
11	10	149167362118	149167362120	17	http://localhost:4000/rest/users	

Figure 5.5: Example of latency data

The data volume/packet size will be measured by using previously mentioned tool, the packet sniffer. In the case of this experiment the packet sniffer *Wireshark* will be used. By utilizing that tool we can look for certain packages by the help of the filter system. By applying the filter as exampled below in Figure 5.6 we can filter out just the traffic on the loopback network, and filter down to the exact type of requests that we will be making. In the instance of this experiment since both the API's return data in JSON object format, we can filter based on that. As Figure 5.6 shows, filtering out the get requests themselves we end up with only the response packets instead of the combination of the response and requests.

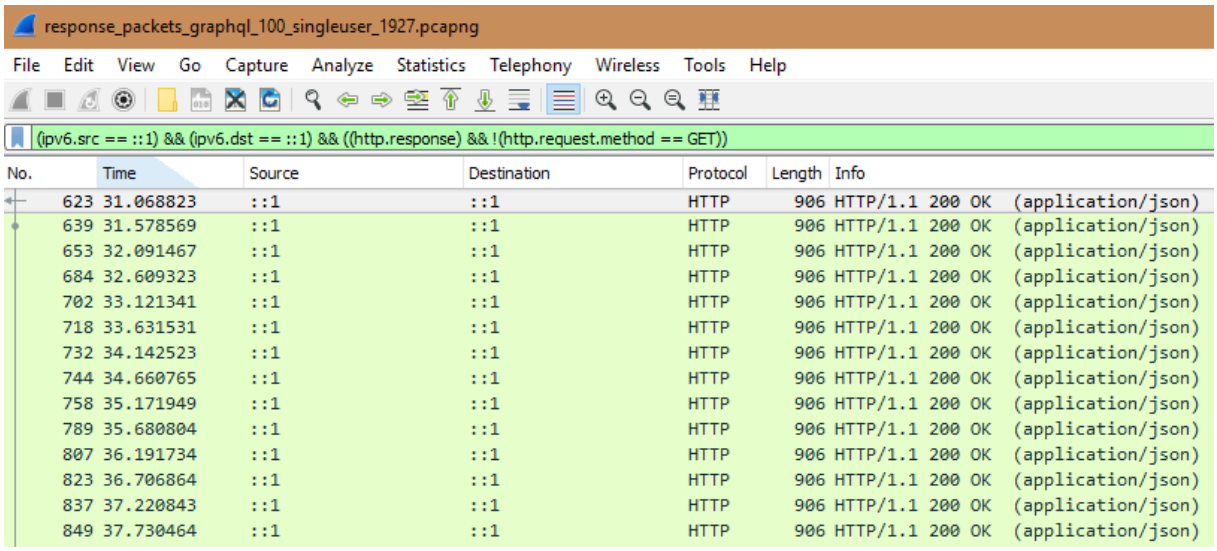


Figure 5.6: Example of the filters used to narrow down the amount of packages

After a test has been conducted, the packet sniffer can be stopped and the data can be exported from the program as a *CSV* file, which allows for easy reading and analysis of the data.

Both the latency and the data volume will be represented in the form of graphs which will compare and clearly display the differences of the API's in question.

5.3.5. Hardware

As the pilot test will run both the client and the server on the same client, they both have the same hardware available. The *central processing unit (CPU)* is an *Intel Core i7 860* which runs at 2.80GHz. The operating system is *Windows 10 Educational 64-bit,* and the *random access memory (RAM)* is 12GB DDR3 running at 666MHz.

Technological specifications for further research and experiment is yet to come as the experiment has yet to be tested on different hardware.

5.3.6. Generated Datasets

The database schema for the pilot test will consist of one table, this table can be used in various ways to try out the measurements and difference performance of the REST API and GraphQL. The table schema can be seen in Figure 5.7.

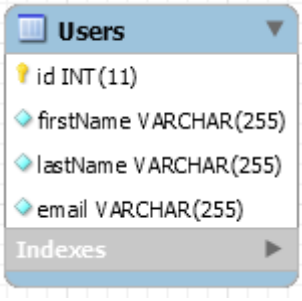


Figure 5.7: Pilot test table structure

The schema that will be used for the experiment itself will contain multiple tables and dependency between them, which will allow for more complex relational database querying and more complex testing of the API's of this experiment.

The data itself is generated via the *Faker* package that is available through the npm library of Node.js. Faker is a useful tool for generating fake data for testing purposes. Faker provides a huge selection of data types that can be generated, from names, addresses, email addresses, and more. For the testing purposes and the consistency of the data in the database, Faker provides a seed functionality where a seed can be inserted and the same data should be generated each time the data in the database is regenerated (“faker,” n.d.). An example of some generated data can be seen in Figure 5.8 below.

id	firstName	lastName	email
1	Garnett	Schinner	Christelle_Renner59@hotmail.com
2	Estella	Miller	Melissa5@gmail.com
3	Waino	Reichert	Trevion18@hotmail.com
4	Zachary	Deckow	Elliott.Monahan@yahoo.com
5	Carli	Kilback	Bailey_Beer@gmail.com
6	Holden	Bahringer	Judge.Moore@gmail.com
7	Alycia	Greenfelder	Frederick.Feeney@hotmail.com
8	Ozella	Mosciski	Geovanni_Langworth46@gmail.com
9	Alyce	Stokes	Marilou.Dach@yahoo.com
10	Adonis	White	Virgil.MacGyver30@hotmail.com

Figure 5.8: Data generated with Faker

Although this data is branded as fake or made up data, there exists some ethical risks by generating data in this way, for example by generating email addresses we can not be totally sure if the address is made up or if possibly it could have stumpled on to an address that actually exists.

5.4. Pilot Test

The pilot test will be implemented through the use of the structure and files mentioned before. Pilot test specific code is availale in Appendix A – Code .

5.4.1. Aim

The aim of the pilot test is to set out and test the possible code solutions for the experiment itself. It will provide a small scale overview of comparison between REST API and GraphQL and its performance differences in unison with providing a small set of graphs and data.

5.4.2. Testing Criteria

Randomly generated data was generated via the Node.js package *Faker*. The random seed used for the pilot testing was the number *42*. The code for setting the seed can be seen in Figure 5.9. A dataset totalling of 100 rows was generated for the singe table pilot test approach mentioned before.

```
57
58
59  // Add a Faker seed number for a consistent dataset.
60  Faker.seed(42);
61
```

Figure 5.9: Faker seed being set

The test was conducted using a single html file that runs a script for either of the API's, providing the option to input the URL of the endpoint. For safety measurements and load balance, although not much load is present on the local server, a delay of 500ms was taken between requests as similar approch was taken by Zhou (Zhou et al., 2014). Each endpoint for each test was requested 100 times.

The following cases were tested.

- Fetching all users
- Fetching a specific user
- Fetching only email (no-endpoint available for REST API)

The last point sets out to try and see determine if the more flexible GraphQL queries can be quicker to respond if only providing the information that is needed. As no endpoint was configured for only fetching users emails from the REST API, the call needed to be made to fetch all the users.

Testing will be conducted as well on the data volume and the packets and sizes will be recorded during the tests for analysis and comparison, which will then be represented in a graphical way.

5.4.3. Results

After running the initial pilot test it became more clear that Cederlunds observations (Cederlund, 2016) that GraphQL performance less efficiently than its rest counterpart when dealing with a single-endpoint .

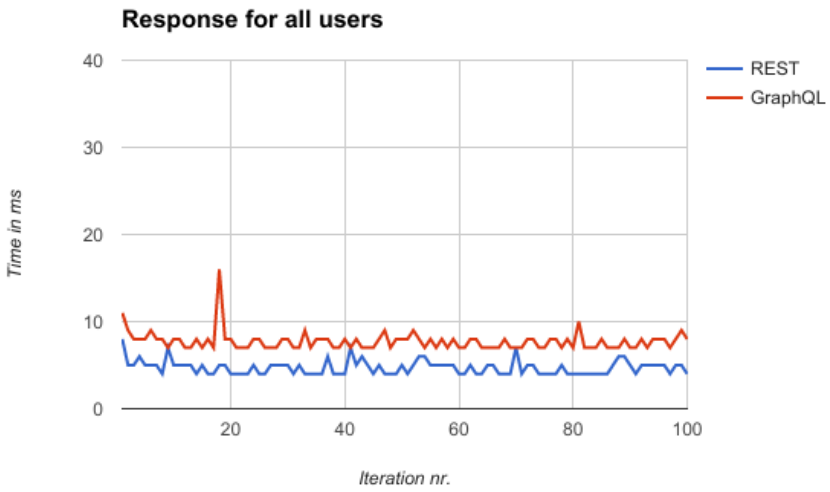


Figure 5.10: Difference in response time when fetching all users

Shown in Figure 5.10 above is the difference of performance between REST and GraphQL while fetching all the users in the database and their corresponding data. The performance difference itself is not utterly different between the two API's in this situation and under this load, with a difference of ≈3*ms* in favor of the REST API. Comparing the two averages of both the technologies will give the results of ~64.3% increase in response time for GraphQL, compared to REST.

When tested for difference by fethings only the email of all the users via GraphQL, and all users and their data via the REST API (*As mentioned before, a test case if a dedicated REST endpoint didn't exists*), similar results are produces. This is shown in Figure 5.11 below.

In this case of fetching emails, the response time difference is even greater. Proving that for a single endpoint the more static REST does perform faster by a margin of ≈7.8*ms.* This is not a whole lot of difference when considered in milliseconds, however it can be noted that although not a great margin in milliseconds, percentage vice it is a significant increase by ~114.6%.

These results can be quantifed in the cases when a clients request is for simple single-table SQL fetch, rather than more complex joining of multiple tables and data. The static RESTful service will serve the response with a request in a significantly lower percentage time than its GraphQL counterpart.

To further quantify these findings. One final test was run with 500 iterations of requests for the database used later for the final experiment. Only the emails of all users in the database were fetched via GraphQL, compared to fetching all user and their data via REST. The trend seems to be followed as the difference was 83.25% increase in response time for GraphQL compared to REST.

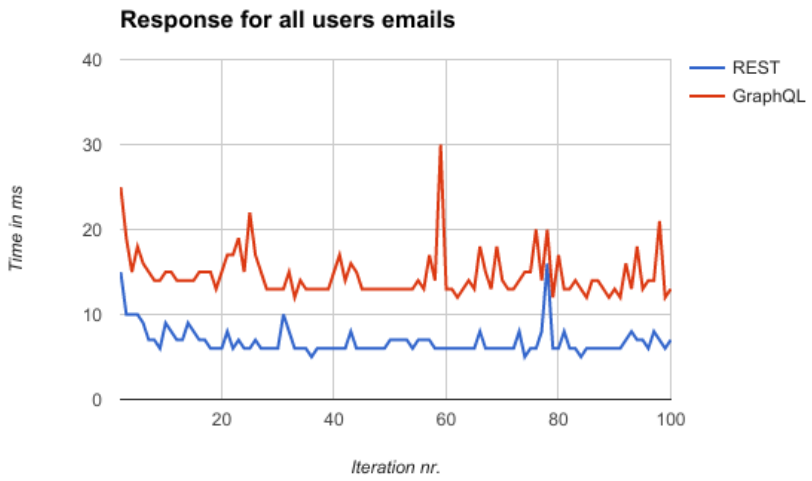


Figure 5.11: Difference in response time when fetching all users emails

Getting to the actual data volume or *packet size* of the previous requests, we can see that in both instances the REST API packet size is much smaller in size compared to the GraphQL response packet size in both the case of fetching all users and just by fetching the users emails, as can be seen in Figure 5.12.

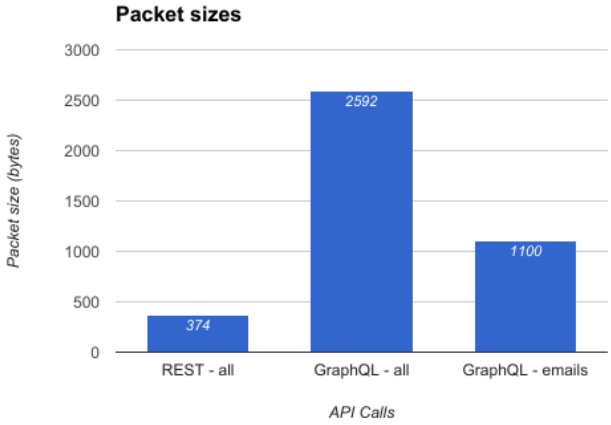


Figure 5.12: Difference in packet size in relation to different endpoints

With these observations we can establish that the pilot test did function well and did return valuable data that the experiment can build upon in the future. Under these specific circumstances the REST API outperforms the GraphQL counterpart in both response time and data volume that is returned to the client from the server.

All graphs and data collected during the pilot tests can be found under Appendix B – Graphs & Data.

5.5. Final Experiment

5.5.1. Structure

The structure of the experiment that will be used to test the hypothesis is in the form of endpoints that can be called via Ajax, just as in the pilot test before. Another database was implemented, new endpoints created and more data added for a more realistic testing environment.

The endpoints for the experiment are four.

- /graphql (GraphQL)
- /users/order/$id (REST, Get a certain user from an order id)
- /orders/user/$id (REST, Get a certain order from a user id)
- /products/order/$id (REST, Get products based on the order id)

The GraphQL will continue to use the same endpoint as in the pilot test since GraphQL is built on a single endpoint. There we can simply pass in a different query that should contain the fields that are requested.

The REST API however will need multiple endpoints for this experiment. The pilot test tested the two techniques against each other by fetching data from a single table. The final experiment will take this step further and will merge multiple tables together. This way we can test for single simple queries as well as more complex queries, which in turn should give a fair comparison between the two. Comparing the two against fetching from multiple tables should also backup or disprove the claims that the developers of GraphQL make (“GraphQL,” n.d.).

For testing the multi-table approach, a new database had to be created. The database contains tables that are keyed together using combinations of foreign keys. The tables are either linked together in a 1:N relationship or a N:M relationship. This gives us a very flexible relational database on the 3rd normal form. A diagram of the database that was constructed for this test can be seen in Figure 5.13 below.

The fields that were fetched were chosen in GraphQL, fields such as *password* for user were left out in both the REST and GraphQL calls. Since GraphQL is based on a single endpoint with a more dynamic approach of selecting which data the user wants back from the server, this was the main focus of the experiment. GraphQL could be expected to perform better than its REST counterpart when it comes to choosing data, as REST will return all the data that is previously defined for the endpoint in question, on the backend server.

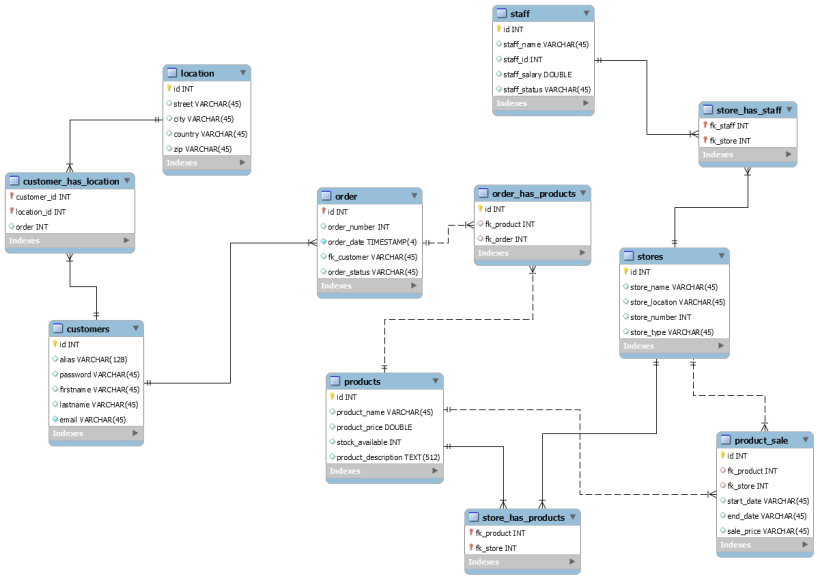


Figure 5.13: EER diagram of the testing database

The size and structure of the database is suppose to represent a minimal requirements of tables and fields that a database for a store should need to have, either this could be a physical store, online store, or a hybrid of the two. Filling and querying the whole database is not the scope of this research, thus we will narrow down the part of the database that will be used for the testing purposes in this experiment.

Specifically the tables *customer, order, order_has_products,* and *products*. These are the tables that will be populated with test data and the queries will be performed on these corresponding tables.

A more detailed view of the tables used can be seen in Figure 5.14.

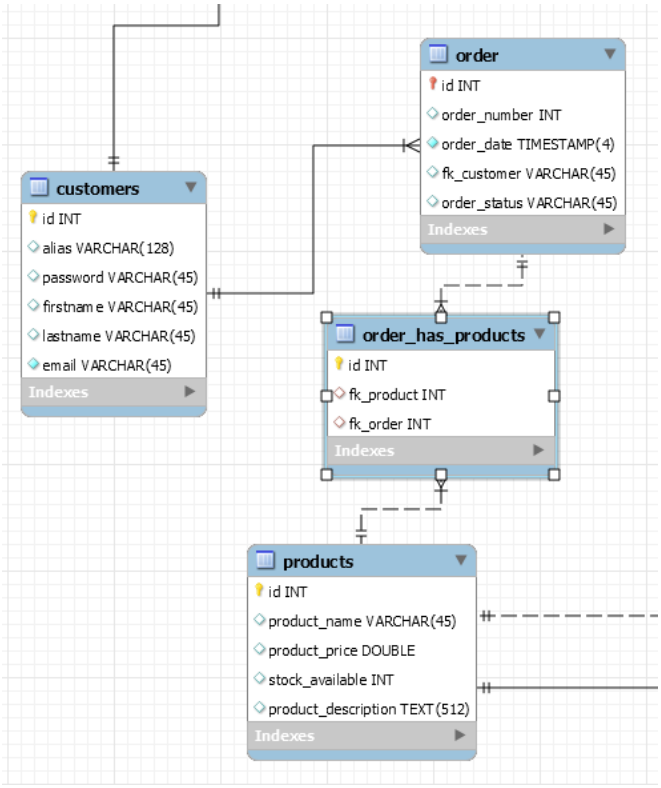


Figure 5.14: EER diagram - detailed view

The relations between these tables are as follows.

	Customer	Order	Order_has_Products	Products
Customer	x	1:N	-	1:N
Order	1:N	x	-	N:M
Order_has_Products	-	-	x	-
Products	1:N	N:M	-	x

This tells us that many customers can have many orders, while only one order can have one customer. An order made by a customer is directly linked to the customer himself, while being linked through a join table to many products. The connections and fields are visible in Figure 5.13 and Figure 5.14.

5.5.2. Data

As with the previous pilot test, the tables will be filled with data using *Faker*, the npm available package for nodejs. Data was generated for all the related tables in fair numbers. The amount of data that was generated and inserted into the tables can be seen in the table below.

Table	**Amount of Data (rows)**
Customers	*10000*
Orders	*8000*
Order_has_Products	*16000*
Products	*50000*

The generation of this data was done totally at random. The fields for the join table *order_has_products* were generated by a random number. This had to be done to not break any foreign key restraints, such as trying to insert an id of a product that does not exists in the database.

An example response from both GraphQL and REST API querying the data in the database can be seen below in Figure 5.15 and Figure 5.16. Queries can be tested with GraphQL on the *graphql* endpoint, as a GraphQL editor comes with GraphQL when it is installed. This allows for testing of queries, data, and in-depth query structure information can be found.


```
{
  "data": {
    "Order": [
      {
        "id": 45,
        "order_number": 29214,
        "order_status": 6,
        "fk_customer": [
          {
            "id": 5814,
            "firstName": "Wiley",
            "lastName": "Mosciski",
            "email": "Odie40@gmail.com",
            "alias": "Murray90"
          }
        ],
        "products": [
          {
            "id": 34582,
            "product_name": "grey Mouse Handmade",
            "product_description": "Reprehenderit fuga ea voluptatem alias.
Perferendis et soluta voluptates iure ut praesentium sunt.",
            "product_price": 26847,
            "stock_available": 2067
```


Figure 5.15: GraphQL query of the test data

To be able to test out the queries of REST easily, the program Postman was used. Postman is an easy tool to test out API calls. An URL is inserted into a query field, as well as needed HTTP headers if there is a need for authentication or other variables specified by the server. In the background Postman runs these queries just like Ajax. Note that API calls can even be tested in the developer consoles of various browsers, simply by entering the code needed to invoke a request.

Postman was utilized in this research since the testing environment page that was programmed for measuring the performance differences, does not display the actual server response, as that data is not needed for the analysis.

There *GET* queries can be passed in and the response can be views easily, along with HTTP headers, and further information of the query that was made (“Postman,” n.d.).


```
{
  "Products": [
    {
      "id": 34582,
      "product_name": "grey Mouse Handmade"
    },
    {
      "id": 7585,
      "product_name": "Investment Account yellow"
    },
    {
      "id": 23262,
      "product_name": "interface South Dakota"
    },
    {
      "id": 47278,
      "product_name": "Lakes HTTP hack"
    }
  ]
}
```


Figure 5.16: REST query of the test data

6. Results

An iteration of 100x was run on each technology in question. The data that is needed can be all requested in a single call with GraphQL, while REST has to make three database calls, to three different endpoints to get the total amount of data that is needed. The endpoints are described in the chapter before in more details.

The query for getting the corresponding information about an order, including the user that owns the order, and the products that are a part of the order was carried out on both GraphQL and REST.

From the data that was gathered it can be seen that indeed GraphQL is quicker than its REST counterpart. This supports the findings of Cederlund in his research (Cederlund, 2016). Cederlund discovered there that as has been established in the pilot test. Smaller queries for example from one table are quicker with REST. However when it comes to more complex queries, it seems that the new GraphQL does outperform the REST service. The results can be seen in Figure 6.1.

Figure 6.1: Getting relational database query

The data-points collected were cleaned of unusually high spikes which ranged all the way up to 200+ milliseconds. The data was cleaned in the way that the value was removed and the next value was copied into the field instead. This was done both for REST and GraphQL, for every point that went over 40 milliseconds in response time. This way of removing spikes that skew the results should not have dramatic effects on the final results. As the points were removed in a fair way, not favouring any technology in question over another. Another factor is that changing relatively few *points(under 5 points)* in a dataset of hundred points, should not affect the results greatly.

When it comes to the size of the packages that are returned, similar pattern can be seen. By running this more complex query, it can be expected that the more static REST service will return more unneeded data, as the data to be returned is all predefined on the backend server.

Seen in Figure 6.2 is the total significant difference between GraphQL and REST when it comes to the packet size returned to the client. The biggest packet size by far is the return of the REST service when it fetches the product information, as this particular order id that was chosen had several products linked to it. This includes the name, price, and description which takes a fair amount of bytes, since a description is a chunk of text. The packets in Figure 6.2 are represented as **one for each call**, since REST has to make three API calls to the server to fetch all the corresponding information needed.

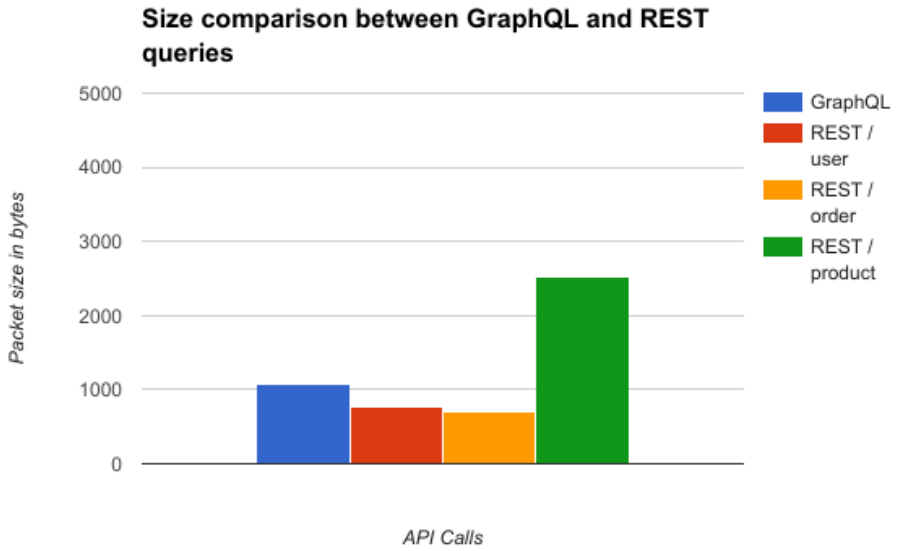


Figure 6.2: Difference in packet sizes between calls

If all the REST calls (*individual REST calls in Figure 6.2*) are combined together to form the whole structure of the data, the packet size will look like in Figure 5.6. This points to the fact that by choosing the data that is wanted, we can indeed reduce the total data size, and by reducing it down to one call just like GraphQL is structured to do, it should decrease packet size as well as bandwidth usage in whole. This can of course change between the data that is being requested and how complex the queries are.

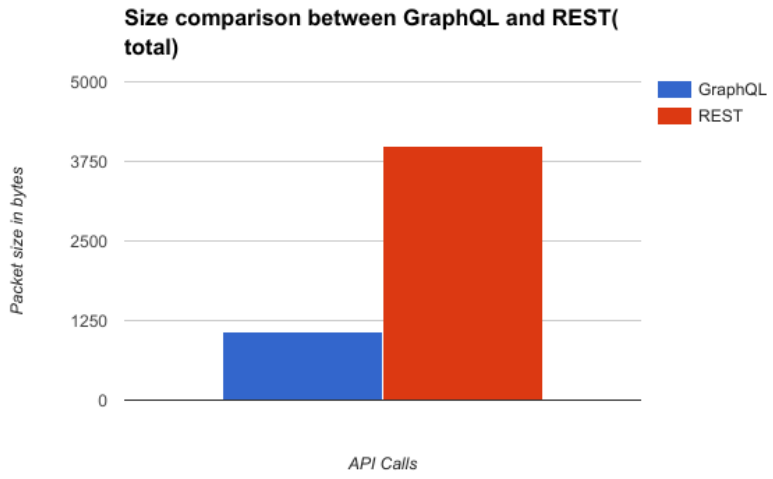


Figure 6.3: Total packet sizes

7. Conclusions

In conclusion. By running the pilot test we did establish that REST is faster for the more simple structured queries, such as fetching information from only one source or table. The difference in performance regarding response time grows exponentially with the size of the database. By fetching more information, GraphQL will produce slower answers. The difference measured in these two experiments conducted in this research suggest that a difference between 64-115% is at least expected.

When it comes to construct bigger databases and in return bigger and more complex queries, there is where GraphQL can come into its own. In the second experiment of this research, a difference of 25% was detected when fetching from the more complex query structure. The average response time can be seen pictured below in Figure 7.1.

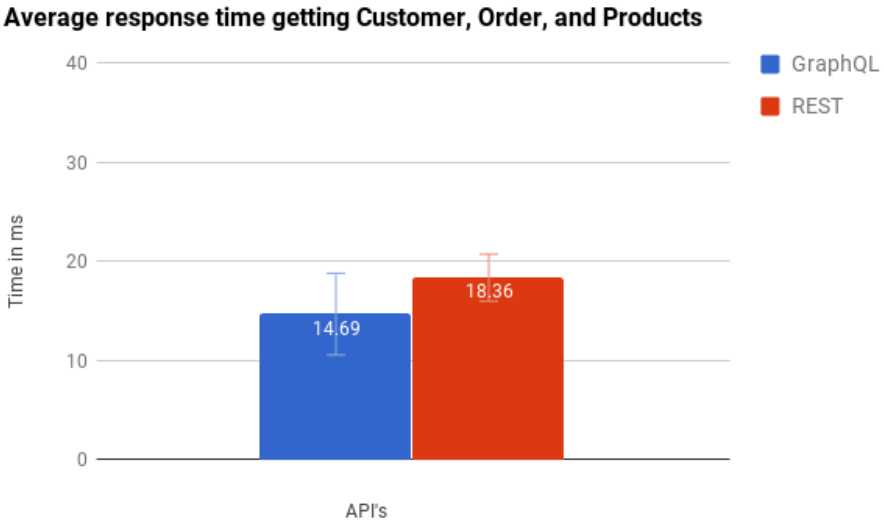


Figure 7.1: Average response time fetching complex data

A second test to confirm these findings was run. This test implemented the same data fetch procedures, however instead of iterating 100 times, this test was iterated 1000 times. The results followed the same pattern that has been detected before. Showing GraphQL to be 51.75% faster than the RESTful service.

By being able to choose the data with much more precision, the total packet sizes sent between the clients and the server can be reduced, possibly making loading times faster. Choosing the data specifically, could also reduce the work of the client, as not as much logic has to go into processing the possible vast amount of data that comes from the server, and rather process just the data that is needed at that point in time.

While GraphQL seems to create a larger overhead in the HTTP request, it can be justified when the data to be fetched is of more complex nature. Such as demonstrated in the experiment with multiple joining tables.

As stated in the beginning of this experiment, nothing stops the developers from creating more endpoints for the REST service that is running. This is however some sort of a hybrid static solution compared to implementing GraphQL on a server. Since as developers we might not know at what points certain endpoints will be needed. Thus GraphQL could be implemented once with its high flexibility and dynamic nature, allowing the clients to pick and choose the data without having have to worry if an endpoint exists or not for the specific query.

The code for both the pilot test as well as the final experiment can be found in Appendix A – Code .

7.1. Future Work

Future work involves experimenting on REST and GraphQL on an even bigger scale, with a larger experiment, larger dataset, and on different networks, which has been mentioned before in this experiment. Knowing whether GraphQL could significantly improve the access and availability to all types of devices.

Further work could also be done on the caching of data. Instead of calling a database for the same data multiple times, a caching service could be implemented which could reduce the load of the server and the database, while at the same time possibly returning data even faster to the client.

References

A JavaScript library for building user interfaces - React [WWW Document], n.d. URL https://facebook.github.io/react/ (accessed 12.11.16).

About | Node.js [WWW Document], n.d. URL https://nodejs.org/en/about/ (accessed 4.9.17).

adaptive path » ajax: a new approach to web applications [WWW Document], 2008. URL https://web.archive.org/web/20080702075113/http://www.adaptivepath.com/ideas/essays/archives/000385.php (accessed 2.14.17).

Belqasmi, F., Singh, J., Melhem, S.Y.B., Glitho, R.H., 2012. SOAP-Based vs. RESTful Web Services: A Case Study for Multimedia Conferencing. IEEE Internet Comput. 16, 54–63. doi:10.1109/MIC.2012.62

Cederlund, M., 2016. Performance of frameworks for declarative data fetching : An evaluation of Falcor and Relay+GraphQL.

Choi, M., 2012. A Performance Analysis of RESTful Open API Information System, in: Kim, T., Lee, Y., Fang, W. (Eds.), Future Generation Information Technology, Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 59–64. doi:10.1007/978-3-642-35585-1_8

Custom Search JSON/Atom API | Custom Search [WWW Document], n.d. . Google Dev. URL https://developers.google.com/custom-search/json-api/v1/overview (accessed 12.11.16).

eBay Developers Program [WWW Document], n.d. URL https://go.developer.ebay.com/ (accessed 4.9.17).

faker [WWW Document], n.d. . npm. URL https://www.npmjs.com/package/faker (accessed 4.8.17).

Fielding Dissertation: CHAPTER 5: Representational State Transfer (REST) [WWW Document], n.d. URL http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm (accessed 4.9.17).

Foping, F., Walsh, J., Roche, D., 2013. Design and Implementation of a Private RESTful API to Leverage the Power of an eCommerce Platform, in: Proceedings of International Conference on Information Integration and Web-Based Applications & Services, IIWAS '13. ACM, New York, NY, USA, p. 681:681–681:685. doi:10.1145/2539150.2539251

GraphQL: A query language for APIs. [WWW Document], n.d. URL http://graphql.org/ (accessed 12.11.16).

James, J., 2014. Data Never Sleeps 2.0 | Domo Blog. Domosphere.

jquery.org, jQuery F.-, n.d. jQuery.

Kozierok, C., 2017. The TCP/IP Guide - HTTP Response Message Format [WWW Document]. TCPIP Guide. URL http://www.tcpipguide.com/free/t_HTTPResponseMessageFormat.htm (accessed 4.8.17).

MySQL :: MySQL 5.7 Reference Manual :: 14.2.2 DELETE Syntax [WWW Document], n.d. URL https://dev.mysql.com/doc/refman/5.7/en/delete.html (accessed 2.14.17).

Node.js [WWW Document], n.d. URL https://nodejs.org/en/ (accessed 4.8.17).

npm [WWW Document], n.d. URL https://www.npmjs.com/ (accessed 6.8.17).

php.net, n.d. PHP: What is PHP? - Manual [WWW Document]. PHP.net. URL http://php.net/manual/en/intro-whatis.php (accessed 2.12.17).

Postman [WWW Document], n.d. . Postman. URL https://www.getpostman.com/ (accessed 5.26.17).

Qadeer, M.A., Iqbal, A., Zahid, M., Siddiqui, M.R., 2010. Network Traffic Analysis and Intrusion Detection Using Packet Sniffer, in: 2010 Second International Conference on Communication Software and Networks. Presented at the 2010 Second International Conference on Communication Software and Networks, pp. 313–317. doi:10.1109/ICCSN.2010.104

Sequelize | The Node.js / io.js ORM for PostgreSQL, MySQL, SQLite and MSSQL [WWW Document], n.d. URL http://docs.sequelizejs.com/en/v3/ (accessed 4.9.17).

Thinking in GraphQL | Relay Docs [WWW Document], n.d. URL http://facebook.github.io/relay/index.html (accessed 12.11.16).

Wrox Article : What is Ajax? - Wrox [WWW Document], n.d. URL http://www.wrox.com/WileyCDA/Section/id-303217.html (accessed 4.9.17).

XMLHttpRequest Level 1 [WWW Document], n.d. URL https://www.w3.org/TR/2014/WD-XMLHttpRequest-20140130/ (accessed 4.9.17).

Zhou, W., Li, L., Luo, M., Chou, W., 2014. REST API Design Patterns for SDN Northbound API, in: 2014 28th International Conference on Advanced Information Networking and Applications Workshops. Presented at the 2014 28th International Conference on Advanced Information Networking and Applications Workshops, pp. 358–365. doi:10.1109/WAINA.2014.153

Appendix A – Code

Pilot Test

Server.js

```
import Express from 'express';
import GraphHTTP from 'express-graphql';
import Schema from './schema';
import Path from 'path';
import mysql from 'mysql';
import rest from './REST';

const APP_PORT = 4000;

const app = Express();

const ROUTER = Express.Router();

function REST(){
  var self = this;
  self.connectMysql();
};

REST.prototype.connectMysql = function() {
  var self = this;
  var pool = mysql.createPool({
    connectionLimit : 100,
    host : 'localhost',
    user : 'root',
    password : '',
    database : 'thesis-db',
    debug : false
  });
  pool.getConnection(function(err,connection){
    if(err) {
      exit(1);
    } else {
      connection.connect();
      var rest_router = new rest(ROUTER,connection);
    }
  });
}

new REST();
```

```
app.use(Express.static(__dirname));
app.use('/pilot', function (req, res){
  res.sendFile(Path.join(__dirname+'/pilot_test.html'));
});

app.use('/rest', ROUTER);

app.use('/graphql', GraphHTTP({
  schema: Schema,
  pretty: true,
  graphiql: true
}));

app.listen(APP_PORT, ()=>{
  console.log(`App listening on port ${APP_PORT}`);
});
```

Schema.js

```
import {
  GraphQLObjectType,
  GraphQLInt,
  GraphQLString,
  GraphQLList,
  GraphQLSchema
} from 'graphql';
import DB from './db';

const Person = new GraphQLObjectType({
  name: 'Person',
  description: 'This represents a person',
  fields: () => {
    return {
      id: {
        type: GraphQLInt,
        resolve(person) {
          return person.id;
        }
      },
      firstName: {
        type: GraphQLString,
        resolve(person) {
          return person.firstName;
        }
      },
      lastName: {
        type: GraphQLString,
```

```
        resolve(person) {
          return person.lastName;
        }
      },
      email: {
        type: GraphQLString,
        resolve(person) {
          return person.email;
        }
      }
    };
  }
});

const Query = new GraphQLObjectType({
  name: 'Query',
  description: 'This is the root query',
  fields: () => {
    return {
      people: {
        type: new GraphQLList(Person),
        args: {
          id: {
            type: GraphQLInt
          },
          email: {
            type: GraphQLString
          }
        },
        resolve(root, args) {
          return DB.models.person.findAll({where: args});
        }
      }
    };
  }
});


const Schema = new GraphQLSchema({
  query: Query
});

export default Schema;
```

db.js

```
import Sequelize from 'sequelize';
import _ from 'lodash';
import Faker from 'faker';

// Init db connection.

/*
 * Constant with connection settings.
 * DB name. username, password
 * DB lang, host
*/

const Connection = new Sequelize(
  'thesis-db',
  'root',
  '',
  {
    dialect: 'mysql',
    host: 'localhost'
  }
);

// Add the table structures.

const Person = Connection.define('person', {
  firstName: {
    type: Sequelize.STRING,
    allowNull: false
  },
  lastName: {
    type: Sequelize.STRING,
    allowNull: false
  },
  email: {
    type: Sequelize.STRING,
    allowNull: false,
    validate: {
      isEmail: true
    }
  }
});
```

```
// Add a Faker seed number for a consistent dataset.
/*Faker.seed(42);

// Add some Faker test data to the database.
Connection.sync({force: true}).then(()=>{
  _.times(100, ()=>{
    return Person.create({
      firstName: Faker.name.firstName(),
      lastName: Faker.name.lastName(),
      email: Faker.internet.email()
    });
  });
});*/
export default Connection;
```

REST.js

```
import mysql from 'mysql';

function REST_ROUTER(router,connection) {
    var self = this;
    self.handleRoutes(router,connection);
}

REST_ROUTER.prototype.handleRoutes= function(router,connection) {
    router.get("/users",function(req,res){
        var query = "SELECT id, firstName, lastName, email FROM ??";
        var table = ["people"];
        query = mysql.format(query,table);
        connection.query(query,function(err,rows){
            if(err) {
                res.json({"Error" : true, "Message" : "Error executing
MySQL query"});
            } else {
                res.json({"people" : rows});
            }
        });
    });

    router.get("/users/:id",function(req,res){
           var query = "SELECT id, firstName, lastName, email FROM ?? WHERE
??=?";
           var table = ["people","id", req.params.id];
           query = mysql.format(query,table);
           connection.query(query, function(err,rows){
                      if(err) {
```

```
                res.json({"Error" : true, "Message" : "Error executing
MySQL query"});
            } else {
                res.json({"User" : rows});
            }
           });
    });
}
module.exports = REST_ROUTER;
```

pilot_test.html

Most of the code for the actual pilot test HTML file has been left out and the main focal point of the Ajax call to the API's is shown here.

```
var timer1 = new Date().getTime();
$.ajax({
           url: url,
           dataType: "json",
           method: "GET"
}).done(function(data){
var timer2 = new Date().getTime();
var total_time = timer2 - timer1;
```

Final Experiment

server.js

This file stayed the same as from the pilot test.

schema.js

```
import {
  GraphQLObjectType,
  GraphQLInt,
  GraphQLString,
  GraphQLList,
  GraphQLSchema,
  GraphQLFloat
} from 'graphql';
import DB from './db';
```

```
const Person = new GraphQLObjectType({
  name: 'Person',
  description: 'This represents a person',
  fields: () => {
    return {
      id: {
        type: GraphQLInt,
        resolve(person) {
          return person.id;
        }
      },
      alias: {
        type: GraphQLString,
        resolve(person) {
          return person.alias;
        }
      },
      password: {
        type: GraphQLString,
        resolve(person) {
          return person.password;
        }
      },
      firstName: {
        type: GraphQLString,
        resolve(person) {
          return person.firstName;
        }
      },
      lastName: {
        type: GraphQLString,
        resolve(person) {
          return person.lastName;
        }
```

```
      },
      email: {
        type: GraphQLString,
        resolve(person) {
          return person.email;
        }
      }
    };
  }
});

const Product = new GraphQLObjectType({
  name: 'Product',
  description: 'This is a product',
  fields: () => {
    return {
      id: {
        type: GraphQLInt,
        resolve(product) {
          return product.id;
        }
      },
      product_name: {
        type: GraphQLString,
        resolve(product) {
          return product.product_name;
        }
      },
      product_price: {
        type: GraphQLInt,
        resolve(product) {
          return product.product_price;
        }
      },
```

```
      stock_available: {
        type: GraphQLInt,
        resolve(product) {
          return product.stock_available;
        }
      },
      product_description: {
        type: GraphQLString,
        resolve(product) {
          return product.product_description;
        }
      }
    };
  }
});

const Order = new GraphQLObjectType({
    name: 'Order',
    description: 'This is the order object',
    fields: () => {
      return {
        id: {
          type: GraphQLInt,
          resolve(order) {
            return order.id;
          }
        },
        order_number: {
          type: GraphQLInt,
          resolve(order) {
            return order.order_number;
          }
        },
        fk_customer : {
```

```
          type: new GraphQLList(Person),
          resolve(order) {
            return DB.models.person.findAll( {where: {id:
order.fk_customer}}).then( e => {
              return e;
            });
          }
        },
        products : {
          type: new GraphQLList(Product),
          resolve(order) {
            return DB.models.order_has_product.findAll({
              where: {'fk_order': order.id},
              include: [{
                model: DB.models.product,
                attributes:
['id','product_name','product_price','product_description',
'stock_available']
              }]
            }).then( e => {
              var arr = [];
              e.forEach(function(item) {
                arr.push(item.product);
              });
              return arr;
            });
          }
        },
        order_status: {
          type: GraphQLInt,
          resolve(order) {
            return order.order_status;
          }
        }
```

```
      };
    }
});

const Query = new GraphQLObjectType({
  name: 'Query',
  description: 'This is the root query',
  fields: () => {
    return {
      Person: {
        type: new GraphQLList(Person),
        args: {
          id: {
            type: GraphQLInt
          },
          alias: {
            type: GraphQLString
          },
          password: {
            type: GraphQLString
          },
          firstName: {
            type: GraphQLString
          },
          lastName: {
            type: GraphQLString
          },
          email: {
            type: GraphQLString
          }
        },
        resolve(root, args) {
          return DB.models.person.findAll({where: args});
        }
```

```
},
Product: {
  type: new GraphQLList(Product),
  args: {
    id: {
      type: GraphQLInt
    },
    product_name: {
      type: GraphQLString
    },
    product_price: {
      type: GraphQLInt
    },
    stock_available: {
      type: GraphQLInt
    },
    product_description: {
      type: GraphQLString
    }
  },
  resolve(root, args) {
    return DB.models.product.findAll({where: args});
  }
},
Order: {
  type: new GraphQLList(Order),
  args: {
    id: {
      type: GraphQLInt
    },
    order_number: {
      type: GraphQLInt
    },
    fk_customer: {
```

```
            type: GraphQLInt,
            resolve(id) {
              return DB.models.person.findAll({where: id});
            }
          },
          order_status: {
            type: GraphQLInt
          }
        },
        resolve(root, args) {
          return DB.models.order.findAll({where: args});
        }
      }
    };
  }
});

const Schema = new GraphQLSchema({
  query: Query
});

export default Schema;
```

db.js

```
import Sequelize from 'sequelize';
import _ from 'lodash';
import Faker from 'faker';

// Init db connection.

/*
 * Constant with connection settings.
 * DB name. username, password
 * DB lang, host
```

```
*/

const Connection = new Sequelize(
  'thesis-db',
  'root',
  '',
  {
    dialect: 'mysql',
    host: 'localhost'
  }
);

// Add the table structures.

const Person = Connection.define('person', {
  alias: {
    type: Sequelize.STRING,
    allowNull: false
  },
  password: {
    type: Sequelize.STRING,
    allowNull: false
  },
  firstName: {
    type: Sequelize.STRING,
    allowNull: false
  },
  lastName: {
    type: Sequelize.STRING,
    allowNull: false
  },
  email: {
    type: Sequelize.STRING,
    allowNull: false,
```

```
    validate: {
      isEmail: true
    }
  }
});

const Order = Connection.define('order', {
  order_number: {
    type: Sequelize.INTEGER
  },
  fk_customer: {
    type: Sequelize.INTEGER,
    references: {
      model: Person,
      key: 'id'
    }
  },
  order_status: {
    type: Sequelize.INTEGER
  }
});

const Product = Connection.define('product', {
    product_name: {
      type: Sequelize.STRING,
      allowNull: false
    },
    product_price: {
      type: Sequelize.DOUBLE,
      allowNull: false
    },
    stock_available: {
      type: Sequelize.INTEGER,
      allowNull: false
```

```
    },
    product_description: {
      type: Sequelize.TEXT,
      allowNull: true
    }
});

const Order_has_Product = Connection.define('order_has_product', {
  fk_product: {
    type: Sequelize.INTEGER,
    references: {
      model: Product,
      key: 'id'
    }
  },
  fk_order: {
    type: Sequelize.INTEGER,
    references: {
      model: Order,
      key: 'id'
    }
  }
});

// Table relations.
Person.hasMany(Order, {foreignKey: 'fk_customer', as: 'Order'});
Order.belongsTo(Person, {foreignKey: 'fk_customer', as: 'Customer'});
Order.belongsToMany(Product, {foreignKey: 'fk_order', through:
Order_has_Product});
Product.belongsToMany(Order, {foreignKey: 'fk_product', through:
Order_has_Product});

Order_has_Product.belongsTo(Order, {foreignKey: 'fk_order'});
Order_has_Product.belongsTo(Product, {foreignKey: 'fk_product'});
```

```
export default Connection;
```

REST.js

```
import mysql from 'mysql';

function REST_ROUTER(router,connection) {
    var self = this;
    self.handleRoutes(router,connection);
}

REST_ROUTER.prototype.handleRoutes= function(router,connection) {
    router.get("/",function(req,res){
        res.json({
          "Message" : "This is the root of the REST API",
          "REST PATHS" : {
            "/users" : "Get all users",
            "/users/id" : "Get a certain user from user id",
            "/users/order/id" : "Get a certain user from an order id",
            "/orders" : "Get all orders",
            "/orders/id" : "Get a certain order from the id",
            "/orders/user/id" : "Get a certain order from a user id",
            "/products" : "Get all products",
            "/products/id" : "Get a product from its id",
            "products/order/id" : "Get products based on the order id"
          }
        });
    });

    router.get("/users",function(req,res){
        var query = "SELECT id, alias, firstName, lastName, email
FROM ??";
        var table = ["people"];
        query = mysql.format(query,table);
```

```
        connection.query(query,function(err,rows){
            if(err) {
                res.json({"Error" : true, "Message" : "Error executing
MySQL query"});
            } else {
                res.json({"people" : rows});
            }
        });
    });

    router.get("/users/:id",function(req,res){
          var query = "SELECT id, alias, firstName, lastName, email
FROM ?? WHERE ??=?";
          var table = ["people","id", req.params.id];
          query = mysql.format(query,table);
          connection.query(query, function(err,rows){
                    if(err) {
                res.json({"Error" : true, "Message" : "Error executing
MySQL query"});
            } else {
                res.json({"User" : rows});
            }
          });
    });

    router.get("/users/order/:id",function(req,res){
          var query = "SELECT `people`.`id`, `people`.`alias`,
`people`.`firstName`, `people`.`lastName`, `people`.`email` FROM ?? INNER
JOIN `people` ON `people`.`id` = `orders`.`fk_customer` WHERE
`orders`.`id` = ? AND `people`.`id` = `orders`.`fk_customer` LIMIT 1";
          var table = ["orders", req.params.id];
          query = mysql.format(query,table);
          connection.query(query, function(err,rows){
                    if(err) {
                res.json({"Error" : true, "Message" : "Error executing
MySQL query"});
```

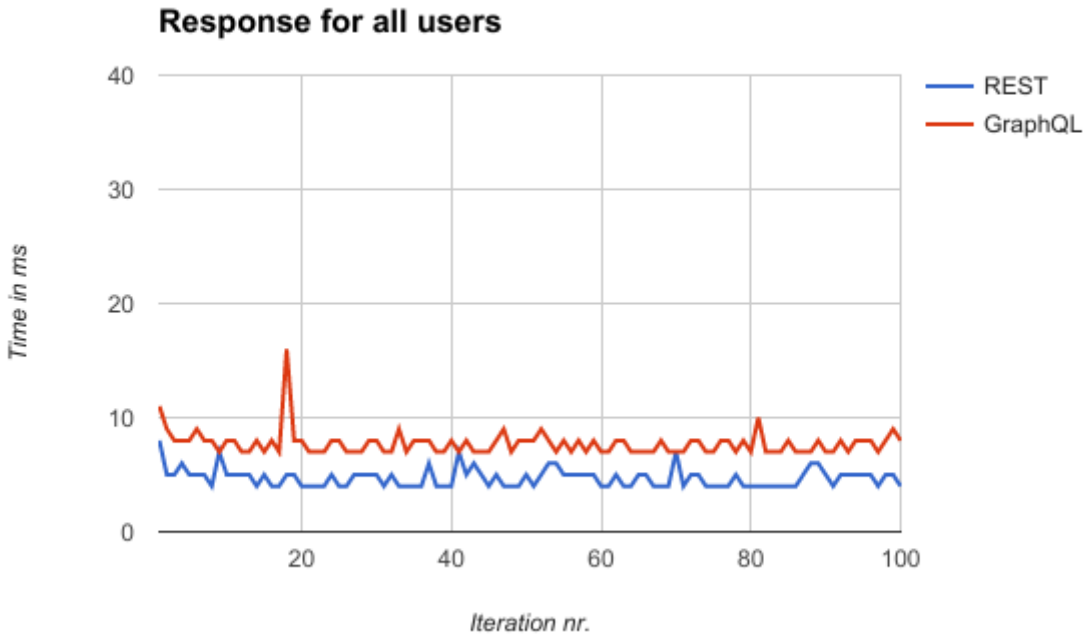
```
            } else {
                res.json({"User" : rows});
            }
           });
    });

    router.get("/orders",function(req,res){
      var query = "SELECT id, order_number, fk_customer, order_status FROM
??";
      var table = ["orders"];
      query = mysql.format(query,table);
      connection.query(query, function(err,rows){
        if(err) {
          res.json({"Error" : true, "Message" : "Error executing MySQL
query"});
        } else {
          res.json({"Order" : rows});
        }
      });
    });

    router.get("/orders/:id",function(req,res){
           var query = "SELECT id, order_number, fk_customer, order_status
FROM ?? WHERE ??=?";
           var table = ["orders","id", req.params.id];
           query = mysql.format(query,table);
           connection.query(query, function(err,rows){
                     if(err) {
                res.json({"Error" : true, "Message" : "Error executing
MySQL query"});
            } else {
                res.json({"Orders" : rows});
            }
           });
    });
```

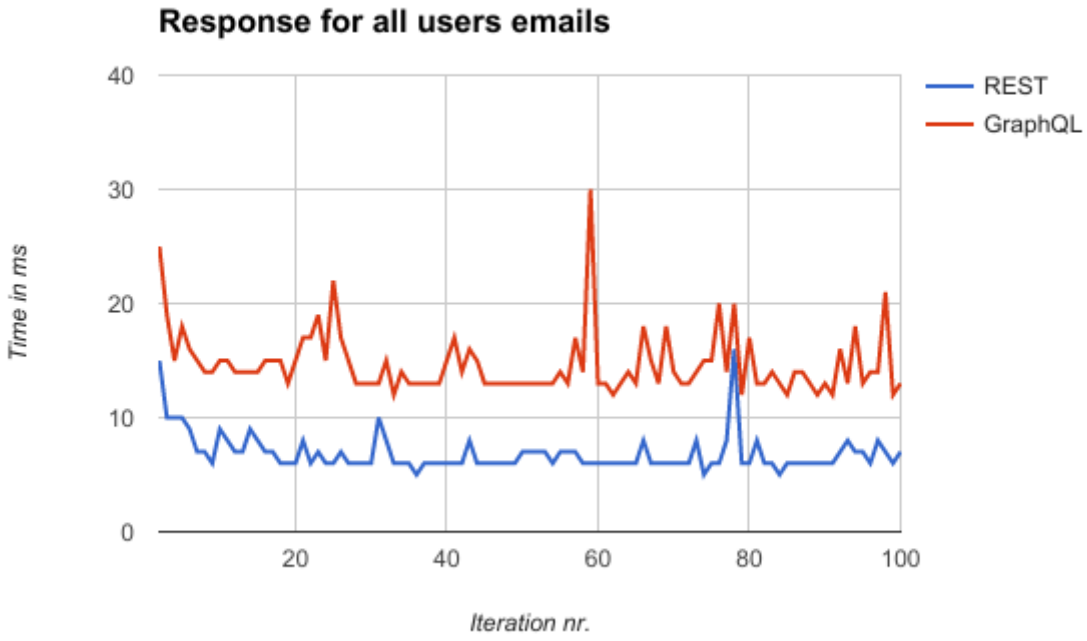
```
    router.get("/orders/user/:id",function(req,res){
          var query = "SELECT id, order_number, fk_customer, order_status
FROM ?? WHERE ??=?";
          var table = ["orders","fk_customer", req.params.id];
          query = mysql.format(query,table);
          connection.query(query, function(err,rows){
                    if(err) {
               res.json({"Error" : true, "Message" : "Error executing
MySQL query"});
           } else {
               res.json({"Order" : rows});
           }
          });
    });

    router.get("/products",function(req,res){
          var query = "SELECT id, product_name, product_price,
stock_available, product_description FROM ??";
          var table = ["products"];
          query = mysql.format(query,table);
          connection.query(query, function(err,rows){
                    if(err) {
               res.json({"Error" : true, "Message" : "Error executing
MySQL query"});
           } else {
               res.json({"Products" : rows});
           }
          });
    });

    router.get("/products/:id",function(req,res){
          var query = "SELECT id, product_name, product_price,
stock_available, product_description FROM ?? WHERE ??=?";
          var table = ["products","id", req.params.id];
```

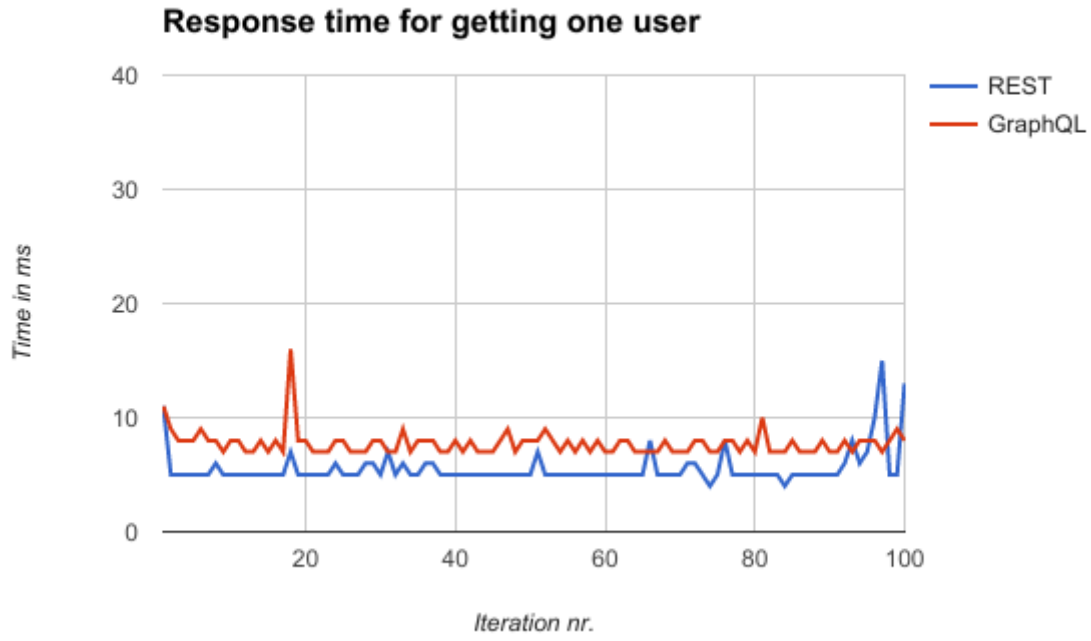
```
            query = mysql.format(query,table);
            connection.query(query, function(err,rows){
                        if(err) {
                    res.json({"Error" : true, "Message" : "Error executing
MySQL query"});
             } else {
                    res.json({"Product" : rows});
             }
            });
    });

    router.get("/products/order/:id",function(req,res){
            var query = "SELECT `products`.`id`, `products`.`product_name`,
`products`.`product_price`, `products`.`stock_available`,
`products`.`product_description` FROM ?? INNER JOIN `products` ON
`products`.`id` =  `order_has_products`.`fk_product` WHERE
`order_has_products`.`fk_order` = ?";
            var table = ["order_has_products", req.params.id];
            query = mysql.format(query,table);
            connection.query(query, function(err,rows){
                        if(err) {
                    res.json({"Error" : true, "Message" : "Error executing
MySQL query, " + err});
             } else {
                    res.json({"Products" : rows});
             }
            });
    });
}

module.exports = REST_ROUTER;
```

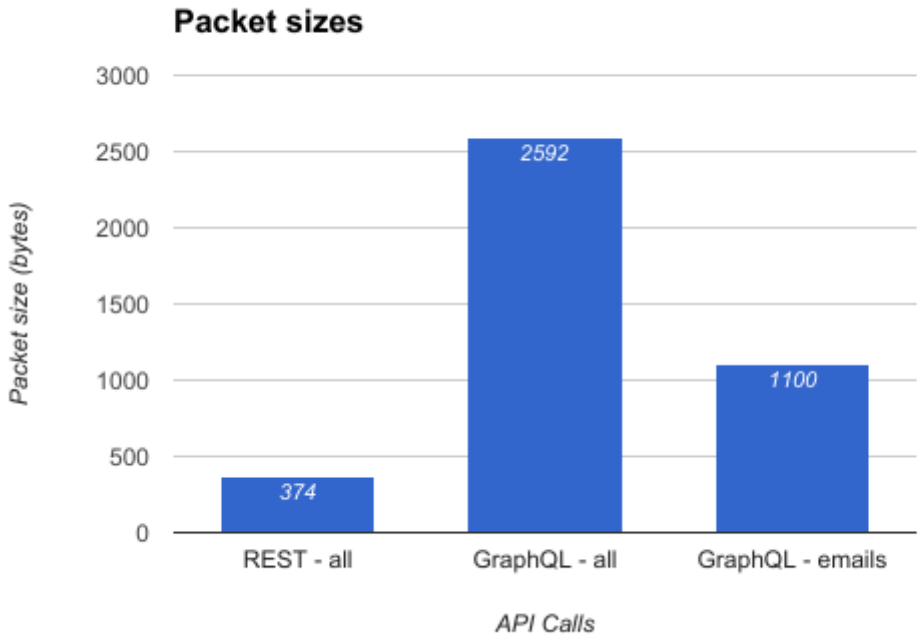
Appendix B – Graphs & Data

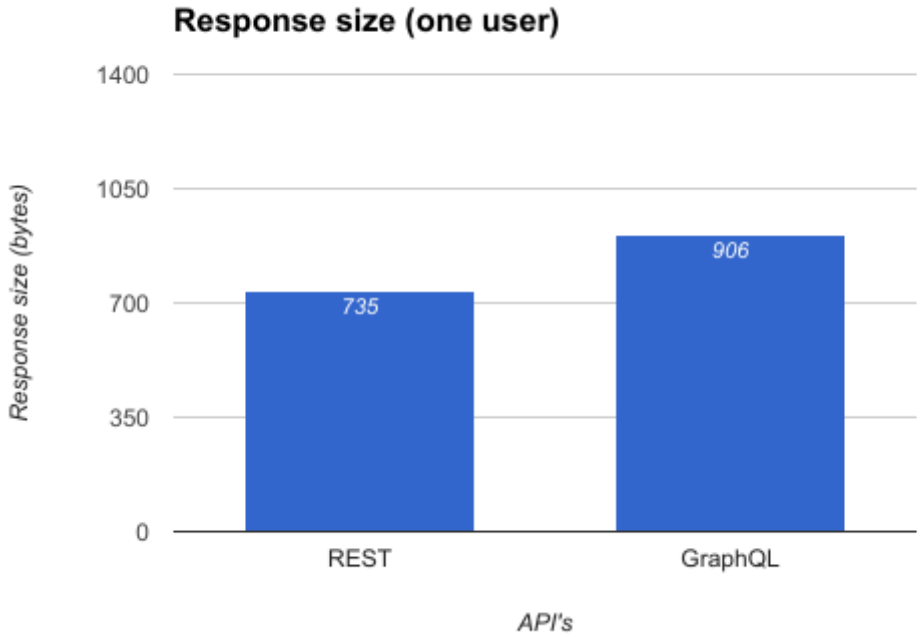
Pilot Test



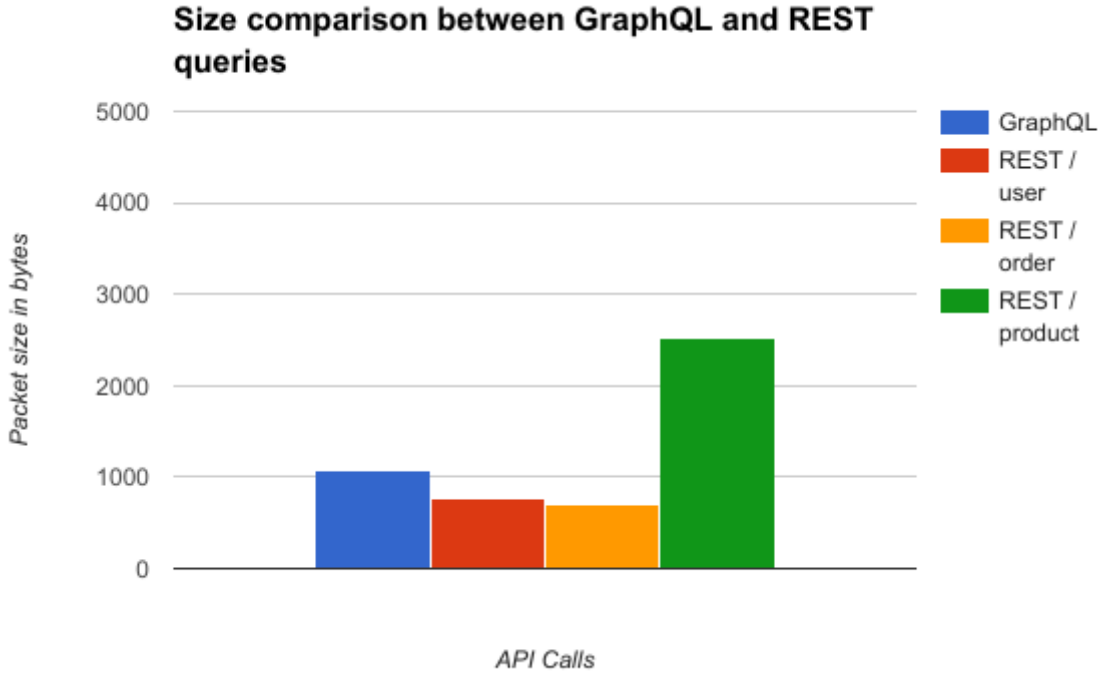


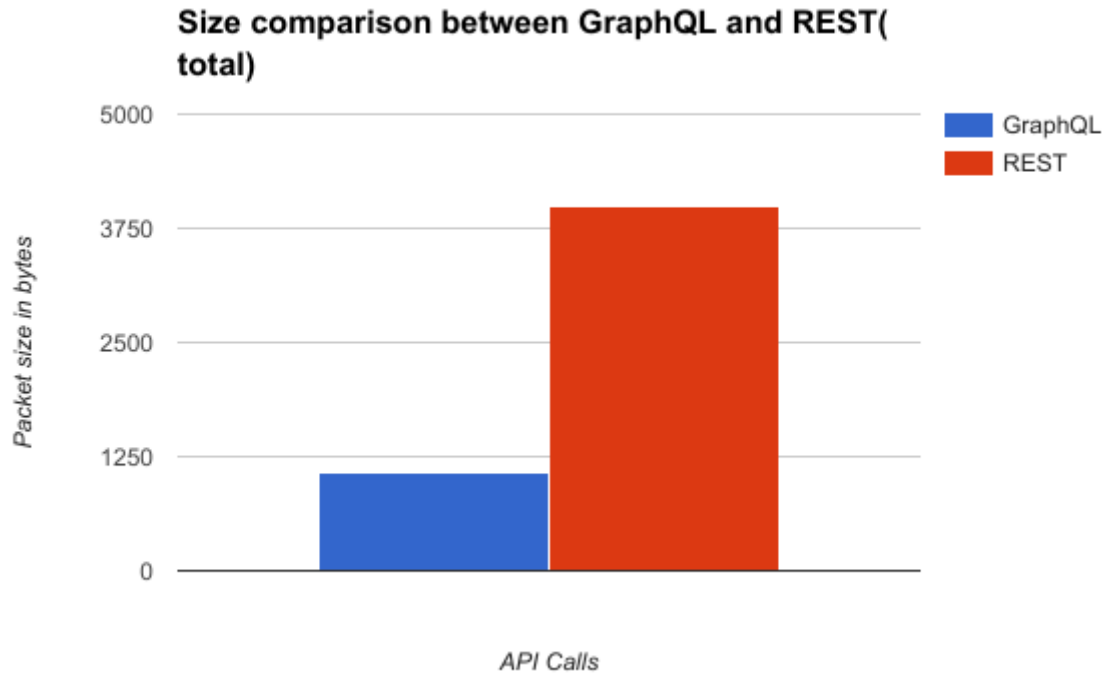

Response time for getting one user
40
30
20
10
0
Time in ms
20
40
60
80
100
Iteration nr.
REST
GraphQL


Packet sizes
3000
2500
2000
1500
1000
500
0
Packet size (bytes)
2592
1100
374
REST - all
GraphQL - all
GraphQL - emails
API Calls



Final Experiment


Size comparison between GraphQL and REST queries
5000
4000
3000
2000
1000
0
Packet size in bytes
GraphQL
REST / user
REST / order
REST / product
API Calls


Size comparison between GraphQL and REST(total)
5000
3750
2500
1250
0
Packet size in bytes
GraphQL
REST
API Calls


Average response time getting Customer, Order, and Products
40
30
20
10
0
Time in ms
14.69
18.36
API's
GraphQL
REST