Kristianstad University
SE-291 88 Kristianstad
Sweden
+46 44 250 30 00
www.hkr.se

Independent project (degree project), 15 credits, for the degree of Degree Bachelor of Computer Science (180 credits) with a major in Computer Science

Spring Semester 2021

Faculty of Natural Science

REST API vs GraphQL - A literature and experimental study

Tobias Andersson and Håkan Reinholdsson

Authors
Tobias Andersson, Håkan Reinholdsson

Rest API vs GraphQL - A literature and experimental study

Supervisor
Fredrik Stridh

Examiner
Dawit Mengistu

Abstract

The purpose of this study is to compare the two architectural techniques REST and GraphQL. This thesis will compare the two techniques and what defines them. A literature study and experimental study are carried out by the researchers. Four applications have been developed that include the ability to disable and to enable caching for both technologies to test the performance effect of caching. Earlier work has not covered the effects on caching related to these two frameworks. The literature study results point to that REST services are up to date and GraphQL is a technique with a shorter history, but that has declared growth in the industry and is a well suited choice for example when bandwidth matters in mobile phone applications. In the experimental study the tests showed slightly better results on average for REST API in terms of total response time (ms). Depending on the intended project there are many factors that need to be evaluated before making a decision on which framework to use.

Keywords
Rest API, GraphQL, Caching, API,

Table of Contents

Dictionary

Here's a dictionary of abbreviations that appears in this thesis work

HTML - Stands for **H**yper**T**ext **M**arkup **L**anguage and it's the standard markup language for displaying documents as web pages.

CSS - Stands for **C**ascading **S**tyle **S**heets and it is a computer language for laying out and structuring web pages via HTML or XML.

JavaScript - A programming language that can be used both on the client-side and server-side, that makes web pages interactive.

MS - Short Term for a millisecond. Used when measuring response time.

API - Stands for **A**pplication **P**rogramming **I**nterface. A set of functions that allows access/features of an operating system, application, class, or other services to work with one's application.

REST - Stands for **R**epresentational **S**tate **T**ransfer. It relies on client-server, cacheable communications protocol, it is always virtual.

DOM - Stands for **D**ocument **O**bject **M**odel. It's what allows scripts to make dynamic HTML objects on a web page.

GUI - Stands for **G**raphical **U**ser **I**nterface and it is a graphical representation of an interface

IoT - The Internet of things (IoT) is a network of connected smart devices providing rich data.

Front End - A term used in information technology that summarizes the processing done near the user.

Back End - Refers to code whose function is to operate in the background and which cannot be retrieved by the user.

NPM - Stands for **N**ode.js **P**acket **Manager** and is utilized to manage dependencies and packets within Node.js.

JSX - Stands for **J**ava**S**cript XML and is a JavaScript extension utilized within React.js

1 Introduction

This chapter lays the foundation for our thesis, describing the background, the motivation, and finally the research questions.

The purpose of this study is to compare the two architectural techniques REST and GraphQL. This thesis will compare the two techniques and what defines them. Is REST still considered state of the art for building web applications? One thing that these two tech stacks have in common is caching in terms of performance, possibilities, and obstacles. What is the role of caching when these techniques are used? Our answer to this will be presented in the form of an experimental study.

1.1 Background

In today's society, it is difficult to think of an aspect of most people's lives that has not been affected by the Internet. The Internet has changed the way we work, learn, shop, and entertain ourselves. It has also changed the way we interact with friends and family. The Internet, based on its definition, connects different computers and users. Today, the internet does more than just connect computers. It connects people, cultures, and businesses. It is a source of information, a social platform, and a business network. The internet has diverse uses depending on the needs of an individual or the setting. Regardless of the purpose, the importance of the Internet in our daily lives is unquestionable.

The Internet keeps growing every day and today's information technology is evolving rapidly. This makes the development of web technologies and web services important. As web services play a significant role in the development of the Internet, the question arises as to what tools should be used to achieve that performance and availability of information, required by today's users.
Modern applications are usually the type of a distributed system, consisting of a frontend and a backend side. Frontend handles the user interface with graphics, buttons, icons to answer when the user navigates. The backend instead stores and delivers the associated data.

Frontend and backend can be executed on different machines but the data, managed by the backend, needs to be delivered from front-end to back-end or the other way back-end to front-end. To manage this process an API is used. When it comes to web applications, there has historically been a technology that has been more frequently used, which is REST API. In later years, another architecture has been used more and more frequently, GraphQL.

1.1.1 Query standards

First, at hand, it is necessary to define what query standards are concerned. In computer science, there is a broad variety of query standards that apply to many environments, such as, Structured Query Language (SQL), Graph Query Language (GraphQL), Representational State Transfer (REST), and much more. They all have the same task, which is to bring the ability to a system so it can fetch data and access distant data over a standard. This standard can also use different protocols to operate. This speaks for itself, that query standard is a quite broad concept, so the researchers will reduce the scope of the definition to the one that is most interesting for this thesis: the web API standards.

1.1.2 Rest

For the last twenty years, REST has been one of the most frequently used ways to transfer data over networks. The researcher Roy Thomas Fielding at the University of California took the first steps when he, in his doctoral thesis, defined the first concept of REST API. Fielding stated: "REST is a coordinated set of architectural constraints that attempts to minimize latency and network communication while at the same time maximizing the independence and scalability of component implementations [1]. At this time, there were no real standards on how to create and use API. The architectures were not so flexible and as a consequence, the need for a simpler alternative increased hand in hand with the growing web demands.

1.1.3 GraphQL

When the first smartphone ecosystems came, the associated devices were limited by their connectivity. Many applications needed to have a few requests as possible to optimize loading time and speed. A company like Facebook, with many applications and different types of feeds that sends multiple queries to fetch data, saw the need to create a new query standard that would allow them to wrap all of the data into one single query. To solve these problems, the project which became GraphQL was initiated in 2012 for the Facebook platform. It remained within Facebook until 2015 when they released the language specification and an open-source reference implementation of GraphQL. Since its release, it has grown in popularity and its usage has been implemented in most programming languages.

1.2 Motivation

Since information technology is becoming a more important aspect of businesses, it is not possible to diminish the importance of reliable frameworks. REST has been appreciated by software engineers when building their APIs for their clients historically. Actually, it has been clarified as “state of the art” when developing web applications. However, back in 2015 Facebook introduced the new service GraphQL that has been a serious challenger to REST. Since GraphQL has been noticed and at the same time appreciated, evaluating the advantages and disadvantages between these two architectures is inspirational.

The ability to use caching is well recognized with REST but not with GraphQL. GraphQL has no natural query caching to avoid re-fetching resources, and standard HTTP caching will not work for a GraphQL API. This is also demonstrated in the lack of studies carried out on this subject. However, there are solutions to implement caching methods to GraphQL and the researchers see the need to test and experiment how caching can affect performance.

When choosing between REST and GraphQL, it is important to compile the performance of the two technologies to facilitate the decision for the developer. Is it purely an opportunity with GraphQL to save on resources and become more efficient as a software engineer? Both towards the customer but also for our part regarding the lifecycle of software development.

1.3 Research Questions

The researchers have identified and concluded that our study should answer two research questions.

Question 1:

Is REST still considered "state of the art" for building web applications and how do REST API and GraphQL design architectures stack up against each other?

In later years, GraphQL has been presented as a revolutionary alternative to REST API:s and fixes many problems with the REST API. Although, several pros and cons need to be evaluated. Which is best suited for a bigger project and database usage?

Question 2:

How does caching affect REST API and GraphQL? Performance, needs, possibilities, and obstacles.

With the use of caching in a network, there are many benefits. Reducing bandwidth and latency, scale down the workload of the server, and have the ability to hide network crashes. But what about caching used with the web architectures REST API and GraphQL? Caching is considered by many developers as one of the GraphQL weaknesses. Is that true? And why is caching so important?

1.4 Ethical and social aspects

This thesis will follow the ground rules set by the European Code of Conduct for Research Integrity.
The data that is planned to be used in the experiment are not real live data and should not contain any sensitive information that could potentially cause harm.
An open-minded mindset has to be acquired and one solution should not be favored over the other.

1.5 Limitations

The conclusions and projections of these researches are limited to a narrow number of general use cases. A massive number of possibilities can be imagined to perform measurements, but it would not be relevant to represent all of these alternatives in this thesis. More complex tests have not been carried out due to the time limit. Most of the tools and frameworks used in the experiments the authors had little previous experience of. This can increase the risk of miss-configuration. The time aspect for the study is limited to 20 weeks. Therefore, the time aspect limits how deeply the two research questions can be explored. Regarding the benchmarks during the experimental study, the fetch requests won't cover larger than 100 000 and not minor than 100 requests. All tests are made locally and not on a remote network. Total response time is measured and the experiments will be limited to comparing those responses. Another limitation is the hardware that the team has worked with. To replace our hardware with a more expensive variant, there could be an opportunity to improve the test results. Based on the time aspect, libraries were used to generate data. The requested data has not been retrieved from different sources or requested nested objects. Our testing does not cover speed comparisons between different databases, which possibly could affect performance results if implemented. A real world application used for these experiments could be a system or server communicating with another system or server, where one acts as a client, and a relatively big amount of requests is sent.

2. Methodology

This chapter is dedicated to the methods that were utilized for this thesis.

2.1 Literature study

The knowledge of the first question is acquired in the form of a literature study where the ambition is to gather broad knowledge related to our first research question. Analysis of the two techniques is important to be able to perform the work. The concrete meaning of this is reading different types of articles and completed studies. As always when it comes to other people's work, it is important to be source-critical, to minimize the risk of lower quality articles. The articles that we have used are taken from Kristianstad University's library and are considered "peer-reviewed" to maintain the highest scientific quality. Master's and bachelor's theses have contributed to our collection of knowledge. There have been some keywords that we used during our research such as REST, API, GraphQL, caching, data fetching, front-end frameworks.

2.2 Experimental study

For the second research question, the authors are going to take advantage of an experimental approach. This means that a back-end side where tests will be run with fetching data from a fake API, by using the technologies REST and GraphQL will be developed. The aim is to do the tests locally on our computers to reduce interference for the best possible results. Along the way here we can retrieve data from our tests to get a better overview of the benefits and limitations of the actual techniques. After the end of the tests, the results will be presented in different types of diagrams. These tests will be with use and no use of caching to evaluate how caching affects these two technologies in terms of performance. Inspiration for this has come from the Rest vs GraphQL - A comparison paper conducted by Oskar Karström and Christian Zaar [12], where they propose to investigate how caching affects these technologies in terms of performance in their future work section.

2.3 Analysis

In the analysis work, it is important to identify key problems and try to understand why these problems exist. Further on, the impacts of these problems on the organization need to be identified and addressed. To address these problems the next step is to uncover possible solutions by research, analysis, and development. Then select the best solution by looking for its pros and cons and checking whether the solution is realistic or not. After this is done it is important to connect our analysis to how we can implement the result with a connection to our research questions.

2.4 Related Work

There have been similar works where the comparison between REST versus GraphQL has been made in different ways. This is beneficial for us as we can take part in these different studies and create an understanding of the subject. The researchers of this thesis will use some of these essays and articles to look for information about previous studies that we think can relate to ours. As GraphQL is a relatively new technology, it is a good idea to carry out more studies on this subject.

A work that the researchers found interesting is “A GraphQL approach to Healthcare Information Exchange with HL7 FHIR” conducted by students in Norway. The key approach here is that the REST provides a comprehensive set of rules. Due to the increase in the complexity of systems being built. But also that the demand in the higher quality of services by the system and users. Therefore, these students conduct a study of what GraphQL offers in the form of query complexity, over fetching, under-fetching, and (n+1) request problems [7].

Another paper is API Design in Distributed Systems: A Comparison between GraphQL and Rest conducted by students in Vienna. This thesis is good in terms of explaining the technologies that are going to be evaluated in our work. Comparison is an interesting part also conducting operation reusability, discoverability, component responsibility, and performance in general. The approach is an experimental study in the first phase and also a literature study in the second phase. Useful for us to take part in and gather opinions about these subjects.

An implementation of GraphQL in .NET Framework with elements of comparison with REST API is another paper that has been useful for us. This paper is conducted at Halmstad University with the main approach to evaluate which techniques are used by web applications. The main differences between these two technologies are discussed, taking advantage of the “query language” being discussed with the conclusion that this API can conduct inquiries against an object.

Oskar Karström and Christian Zaar at Kristianstad University provide a thesis with similar performance measurements. Due to their limitations, they do not implement caching. Instead, they propose in their future work that caching can be investigated and how it can affect these technologies in terms of performance.

3 Literature Study

When GraphQL was introduced in 2015 by Facebook Inc, the conditions for the software industry changed in terms of how Software Engineers could build APIs. This technology challenged the more traditional REST approach in several different aspects. One of the main differences is that GraphQL takes advantage of a query language approach and with that comes a set of tools that only operates over a single endpoint regarding HTTP. Our study shows that it is highly unclear whether REST will still be state-of-the-art when it comes to building APIs in the future.

3.1 REST API

The technology itself is explained as an architecture for various distributed hypermedia systems. What characterizes this is the sets of design concepts, that are based on characteristics and network software. However, there are more areas that REST is associated with such as all things on the internet are declared as resources, it operates on these resources on generic connector interfaces [8]. This technique can be divided as follows:

- A client-server always needs to be implemented. It is based on the concept that the client and server should separate from each other. Data must be able to be manipulated without having an undesirable effect on our design in the database.
- This technique is stateless, which means that calls are made independently of one another. Each call has only the necessary data for that specific call, such as user ID, token, API keys. This approach facilitates stability when we use this structure.
- Another important part is how we handle caching. It is common that when using REST API requests we can get problems with overhead when there are many incoming and outcoming calls. It should therefore be designed to take advantage of cacheable data to avoid this problem.

- To keep in mind when implementing this type of solution is that it should provide standardized means of communication between the client and server-side. It needs to take advantage of HTTP URI resources such as Create, Read, Update, Delete, and JSON (see figure 1.1).

```
<html>
    <head>
        <title>REST/JSON + jQuery Example</title>
        <script src="//code.jquery.com/jquery-1.11.2.min.js"></script>
        <script>
            function callRest() {
                    console.log("call rest");
                    $.ajax({
                        type: 'PUT',
                        url: '/api/user',
                        dataType: "json",
                        contentType: "application/json",
                        data: JSON.stringify({
                         "firstname": "John",
                         "surname": "Doe",
                         "email": "jdoe@example.com",
                         "role": "admin"
                        }),
                        success: function (data) {
                            console.log("success");
                        }
                    });
                    console.log("call rest DONE");
            }
        </script>
    </head>
    <body>
        <button onclick="callRest()">Click me</button>
    </body>
</html>
```

Figure 1.1 - Example of a REST API

3.1.1 Advantages

There are several benefits to using REST as a developer and we are going to name a few here. These are often the reasons why the technology is chosen from the perspective of a software engineer. First out is the aspect that it is cacheable, which means that server response can contain information about whether or not the client can access the cache. If this is possible in any way the response should come with a version number and if the client makes the request twice the server does not need to double the workload to retrieve the same type of data. Advantage number two is that it is stateless which is explained that the server never saves any of the previous requests or responses. It works also with a layered system that means that every component within the system that uses REST does not have access beyond the specific one that it is working against. It also allows a greater variety of data formats and thanks to the JSON format it also has better support for browser clients which we are reviewing. The bandwidth request also using this approach is limited, as it does not need much when the requests are sent to the server.

3.1.2 Disadvantages

When we examine the negative effects of using REST, there is one thing that stands out. This is that this technique has a lack of state. Well, what does that mean concrete? It is explained that many applications on the web use stateful actions. One example is when the client visits an e-commerce website, he needs to know the number of items before the actual purchase goes through. This in turn leads to the maintaining state lying on the client, which leads to the application being slow navigated. Another sensitive thing we want to highlight is that when we use the REST API, there is a security risk. Concrete this means that it is not appropriate to have data communication between client and server when the information is confidential.

3.2 GraphQL

Biehl [2] describes GraphQL as a query language that provides a new philosophy for building API:s that helps the user to structure the interaction between the frontend and the backend.

GraphQL enables the user to build APIs:s for retrieving data, writing data, get and getting notified when the data changes. This is true but it is also worth mentioning that GraphQL is rather a conceptual model for what should be included, but which does not provide clear guidelines on how to implement the API. A GraphQL API basically does not have a specification for how to implement it.

A GraphQL operation can resemble a "CRUD", Create, Read, Update, Delete query. Hence the reason it is called a query language. The difference is that a GraphQL process is either a query (read), mutation (write), or subscription (continuous read). Each one of these operations is a string that needs to be composed according to GraphQL:s query language specification (see figure 1.2). As GraphQL is evolving regularly, there may be more operations in the future[3]. First, at hand, both the client and the server need to understand GraphQL to achieve good communication. Wieruch [3] states that GraphQL is a declarative data retrieval language that follows a number of principles. Three of these principles are that the language should be hierarchical, client-specific, and strongly typed. The GraphQL queries are hierarchical because the queries consist of a hierarchical set of fields. The questions are client-specific because the client defines the questions posed to the server and the response format is described in the question. GraphQL is strongly typed, which ensures that the GraphQL queries are both syntactically correct and valid by the so-called type system before the queries are executed. This type of system acts as a contract between the client and the server. The server receives the GraphQL-queries and validates their correctness with the help of the type system. It also defines the schema where the queries are detailed. If the query is properly formatted, the server can provide data for that query.

Default types are the following [6]:

- Int: A 32-bits signed integer.
- Float: signed double-precision floating-point value.
- String: UTF-8 character sequence.
- Boolean: true or false.
- ID: unique id that is serialized

The schema, on the other hand, is the key to the data. The GraphQL schema is usually defined on the server-side, but clients can read (query) and write (mutations) data based on the schema. On the server-side application, all information about what is available is offered, and the client-side application requests parts of it by performing GraphQL queries, or alters parts by using GraphQL mutations [5].

Figure 1.2 Example of GraphQL query

3.2.1 Advantages

A common problem with REST is the factor of over-and underfecthing. Overfetching means that a client downloads more data than required, which can be information that is useless for the actual request. Underfetching is the other way around and happens when a specific endpoint does not provide the correct information and additional requests need to be carried out. The main reason this occurs is the way the API:s are designed. To download data, an endpoint needs to be reached and return the data. GraphQL solves this problem by fetching only the exact and specific data in a single request. Overfetching is handled by the client as it specifies which data that needs to be returned [3]. Underfetching is handled by GraphQL by defining a nested query and therefore asking for all the necessary and related data in a single query. The form of the answer corresponds to the form of the query and thus only the necessary data is retrieved. Using only one endpoint makes it easier to organize data, unlike REST, where developers have to maintain and adapt new endpoints depending on user requests. GraphQL on the other hand provides an efficient data retrieval because the client can query for nested objects without defining new endpoints, as the data structure is not hard-coded as in REST.

The reason why GraphQL only needs to query a single endpoint is that GraphQL queries can traverse related objects and fields. In this way, the client can get all the relevant data requested in a single request, instead of going through several endpoints, which is the case of REST.

3.2.2 Disadvantages

GraphQL can be mistaken for a server-side database or a replacement for it. That is not its purpose as it is a query language. When a query is requested, the server performs database access. To access multiple fields in one query whether it is requested in a RESTful architecture or GraphQL, the varied resources and fields still have to be retrieved from a data source. This shows the same problems when a client requests too many nested field data at a single time. Therefore there must be mechanisms like maximum query depths, query complexity weighting, avoiding recursion, or persistent queries to stop inefficient requests from the client side.

3.3 Caching

Cache memory or memory caching is a technology used by applications to store data that has been newly accessed. This data is stored in the computer's Random Access Memory (RAM) for retrieving the data faster. The cache makes applications more efficient by getting the data from for example a database. This data can be saved and reused in those cases where queries from the client and the server are done repeatedly. The cache memory itself is limited in size and therefore the saved data eventually needs to be removed so that more current data can be saved [4].

How do the caching mechanisms relate to the techniques that we investigate? One disadvantage about GraphQL is caching, in terms of decreased bandwidth and bad latency, and it is hard to implement. The reason for this is that it stores copies of frequently accessed data in several places along with the request-response. The main point of caching is that we reduce the pressure on the server and streamline the performance of the application. This is becoming a problem for GraphQL as it is only about one endpoint when we want to fetch the data.

However, there are solutions for this in terms of batching techniques and persisted queries. REST on the other hand has no problems with caching and can take full advantage of HTTP caching. What HTTP caching does, is that it allows resources to be copied all along the request path such as local cache and proxy cache. This is also a good reason why REST is usually implemented in HTTP.

4 Experimental Study

In this section, the purpose is to describe below the types of tools we use when doing our experimental study. The researchers intend to argue here why the chosen techniques are relevant according to our study. The idea has always been to do modern scientific research and therefore the following technologies below have been chosen. 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, measured in total response time with an increasing number of requests. To answer research question two, the authors had to collect amounts of data to analyze different measurement times. When it comes to the REST API, Express.js worked as a fetch collaborator, which is an open-source framework for Node.js. In the GraphQL case, we took advantage of the Apollo Client which served as a comprehensive state management library. Objects that were collected are students that contain four different parameters such as id, first name, last name, and email. The fetching requests are divided into four different categories in terms of the number of requests. The numbers are 100, 1000, 10000, 50000, 100000 and contain the actual requests against the server-side. These values were chosen to test how an exponential increase in requests can affect the techniques. With only one or two requests there will be no caching effect and the purpose is to see where the effect of caching really shows its benefits. Each test consists of ten iterations and an average time in milliseconds is calculated and presented in diagrams below. The tests were carried out on a laptop with the following hardware and software:

- **Processor:** Intel(R) Core(™) i5-7200 CPU @ 2.50GHz 2.70GHz
- **RAM-memory:** 8,00 GB
- **Operating System:** Windows 10 PRO

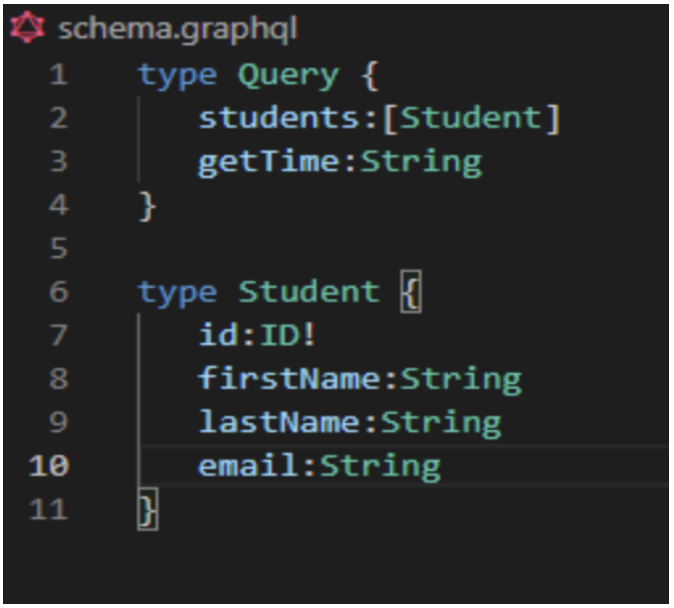

Figure 5.1 - GraphQL schema being defined

For each request, 100 different students are picked up. To create these different objects, Faker.js was used to generate fake data. This is useful when doing performance tests but also smooth so that we could save time and not have to mutate the data by ourselves. The data size of the 100 students was 11.14 kilobytes. To reduce the probability of overhead during the measurements, three pre-measurements iterations have been run before every test.

4.1.1 GraphQL

This technique differs from the following in several different ways. It is a query language for reading and mutating data created by Facebook Inc. The goal was to rethink the way that we are fetching data for any device, especially when we are low on network bandwidth. It is important to point out that GraphQL will never replace REST, but should be seen as a good alternative. It is a very powerful framework and is used by large companies such as Twitter, Shopify, and The New York Times. When we have analyzed the technology, we have been impressed by the simplicity of the technology.
The way GraphQL operates, it does not follow the HTTP specification for caching and instead uses a single endpoint. This allows the developer to make sure that caching is implemented properly for non-mutable queries.

To enable caching with GraphQL, the researchers have used Apollo Client. Apollo Client is a thorough state management library for JavaScript that is used to manage both remote and local data with GraphQL. Apollo includes a default cache implementation for Apollo Client 2.0. It is called InMemoryCache and it is a normalized data storage that supports all of Apollo Client 1.0's features [11].

4.1.2 REST API

Looking back historically when it comes to developing various APIs, REST stands out as a much-appreciated technology. REST stands for representational state transfer and was created by computer scientist Roy Fielding. Software engineers have been developing application programming interfaces for decades and the use of this technique has been huge. Interacting with a computer or system to retrieve information to perform a function, an API helps us to communicate what you want to do and fulfill this request. In recent times, this technology has been challenged by the above.

4.1.3 Visual Studio Code

The choice of code editor fell on Visual Studio Code as it felt most modern for developing web applications. This tool has built-in git commands and is extensible and customizable. It is built for developing against the web with technologies such as HTML, CSS, JavaScript. But it has support for hundreds of different languages. It has shown solid growth in the web development field, in times when the need of developers of having a lightweight well-done editor is huge.

4.1.4 Faker.js

Instead of implementing our own database, we took advantage of a JavaScript library that is named faker.js. This is useful when building and in our case testing the performance of our application. This library can generate fake data for many different areas and some examples are addresses, names, dates, images to name some.

4.1.5 Apollo Client

With the use of the class InMemoryCache, provided by the Apollo Client, it is possible to implement caching to the GraphQL API. Apollo Client stores the results of its GraphQL queries in a normalized, in-memory cache.

This enables the client to respond to future queries for the same data without sending unnecessary network requests. When GraphQL makes a query, by default Apollo caches the response in what it refers to as a flat, normalized lookup table. It constructs a unique identifier for each object returned from the query, by combining its id or properties with the type name defined in the schema. Apollo will also store the results of each of those individual queries, in case it makes the same exact query again later. That data is stored in Apollo's cache, based on the name of the query it executes [11].

4.1.6 Implementation

The implementation of the tests is done with the tools described above. To execute the tests, four applications were developed. Two using REST API and two using GraphQL with one of each technique having the ability to use cache and the other one not. To fairly compare the performance between REST and GraphQL, the same dataset and underlying implementation for communication between the client and the server will be used. The server and the client will also be executed locally on the same computer so that the network latency does not affect the result. As the server and client are two different processes, they do not share the same memory space, even if they are running on the same computer. The four implementations will undergo in total 20 different test cases. Every implementation was built with JavaScript and Node.js.

The implementation of GraphQL required a schedule and types. The queries are defined with their fields in the schedule and can not be validated without a type scheme. To implement caching, Apollo Client and associated InMemoryCache were used. REST with cache was implemented in the same way as REST without cache, but with the addition of a cache implementation, node-cache, which is a Node.js package.
All of the traffic during the testing, both to the REST API and the GraphQL servers goes through a Node.js server script.

5 Results

This chapter presents the results that were generated during the thesis work.

5.1 Results of literature study

This literature study started with the perspective of answering the question of whether REST is still considered state of the art or if it has been overcome by the later GraphQL. Our first conclusion is that an API:s ability to outshine the other one in these areas depends more on how it is designed and less on whether it is designed with GraphQL or REST. Each one of GraphQL and REST has its own advantages, disadvantages, features, and characteristics. A REST API is an architectural concept for network-based software, which has been used to create new APIs:s. GraphQL is a schema language to describe an API, a query language, and a server-side run-time for executing queries. It is a tool to work over a single endpoint, whose focus has been to optimize performance and flexibility.

GraphQL shows its strengths where bandwidth matters. This is especially considered in apps for mobile phones, IoT devices, and smartwatches. For applications where nested data needs to be obtained as well. An example of this can be a blog platform where posts need to be collected along with nested comments from users. The way GraphQL handles over-and-under fetching and how it can reduce the need for making round trip responses significantly makes it more efficient to REST in terms of querying functionality. GraphQL can tailor each request to exactly what is needed, from multiple objects down to specific fields. As long as you tell it what you want.

For modern applications and how they are built, GraphQL is very suitable. An application may only have one backend, but many different clients may depend on its data. Clients such as web, phones, and smartwatches. GraphQL can be used to connect those, but also to accomplish the requirements of each client application–network usage requirements, nested relationships of data, fetching only the needed data without a dedicated API for each client. On the server-side, there can also be one backend and a group of microservices that provide their specific functionalities.

This defines the perfect use for GraphQL schema stitching, which lets you aggregate all functionalities into one GraphQL schema[3].

REST, which has been the industry standard for many years, has many benefits when it comes to the widespread knowledge about it and the ecosystem of libraries and tools surrounding it. Thanks to REST:s separation of the client and the server, its scalability helps development in many approaches. If REST is relevant today, the summary answer will be yes, but here are some factors that need consideration before we make the decision. There is a strong history with the technology itself and it has long been appreciated by developers. Therefore, there is a large knowledge base where developers can help each other and the information bank thus becomes large. After our study, the perception is that we see a great challenger to REST in GraphQL that is growing at a high pace. The perception is that Facebook has created a framework that they themselves requested in order not to have to be dependent on old technologies that feel outdated. What supports our thesis that GraphQL challenges are that it has not existed for a particularly long time (introduced 2015). But it already has a user share of 19% [9], which shall be considered strong. The answer is largely dependent on the intended specific project requirements.

If the project is a social or blog platform with nested data and a modern style that does not require multiple round trips to fetch data, GraphQL is a very suitable choice.

When the project is relying on robust native caching, capabilities in authentication, and proven techniques with a large user base, REST is a good choice.

When there are advantages, disadvantages, constraints and tradeoffs with each API design, it is easier to choose the option best suited to the individual case. The choice should always be the API product that meets the needs of all the participants in the API value chain[10], the API provider, the API consumer, and the end-user.

5.2 Results of experimental study

As figure 5.2 shows (the first test), there is a noticeable difference if the caching is on or off. It becomes clear that the difference becomes greater the more requests that are being assigned. When caching is turned on, GraphQL works efficiently with short response times.

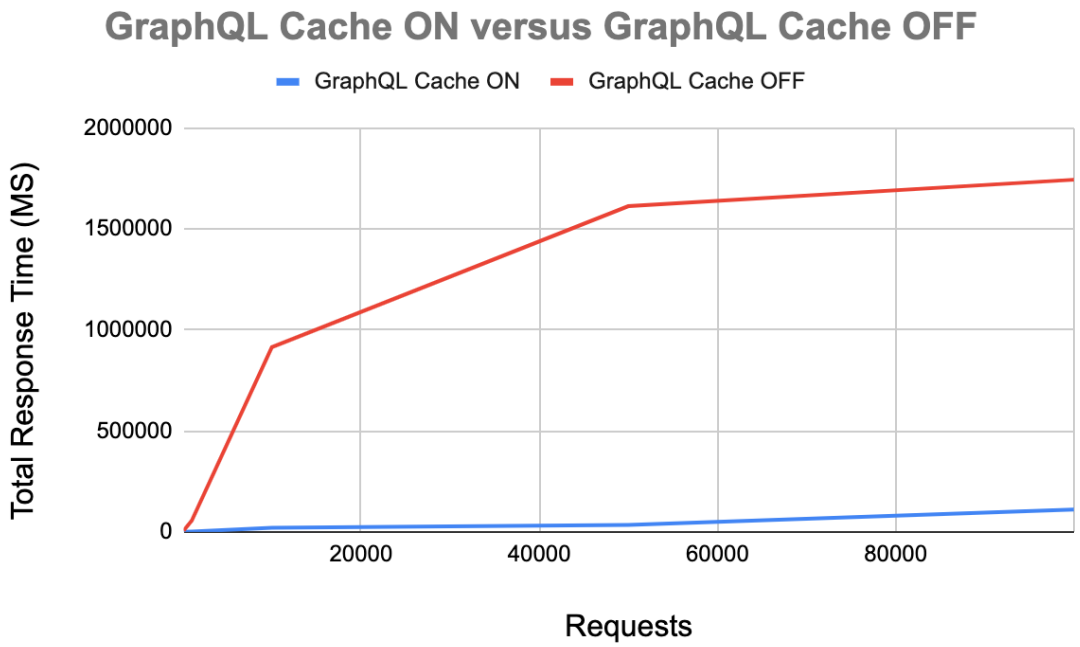

Figure 5.2 - Caching outperforms non caching in GraphQL with an average increase of 2016,35%

The second test is being made in figure 5.3 but switching technology to REST. The difference is remarkable even here, if REST is allowed to work with caching the response times remain more efficient.

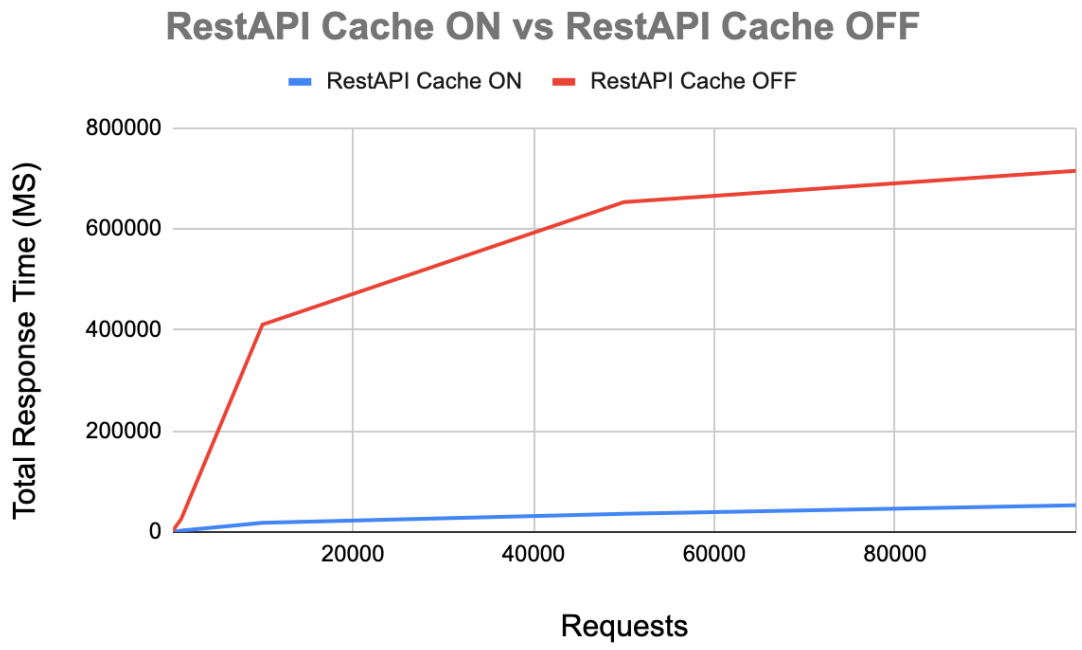

Figure 5.3 - Caching outperforms non caching in REST API with an average increase of 1386.87%

In the third test as seen in figure 5.4, GraphQL is compared to REST with caching turned on. Interesting here is how the lines correlate during a little more than half of the requests, then to separate completely when the requests are getting bigger.

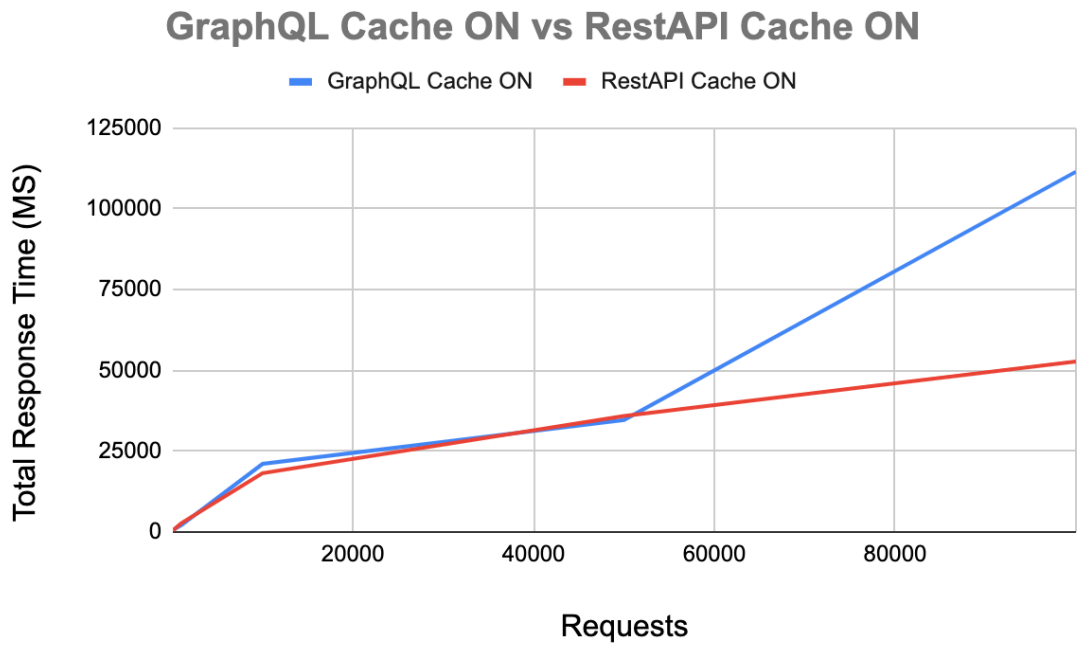

Figure 5.4 - The two techniques are set against each other with caching on

The final test involves caching turned off and is comparing GraphQL vs REST. A direct impact from the start is that REST can handle the lack of caching in a better way than GraphQL.

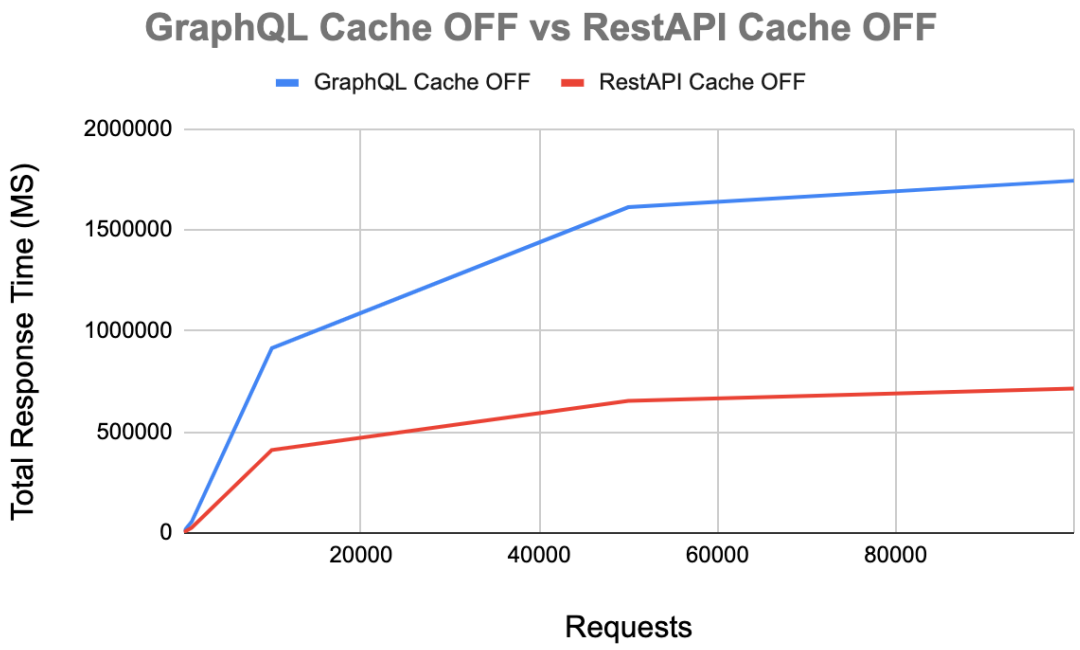

Figure 5.5 - The two techniques are set against each other with non caching

6 Discussion

In total, the researchers did four different tests with 5 different request values. When compiled, the number will be a total of 20. The use of caching that is implemented in JavaScript code had a big impact on our results in all tests more or less. When the caching was turned on, we saw increased performance both in the REST case and also in the GraphQL case. In general, both techniques worked better with caching turned on (which we assumed it would), but it was nice to see some concrete numbers in terms of milliseconds. That REST works with shorter total response times than its competitor is answered in our tests but also confirms the result of our literature study that we made.

Research Question 1: *Is REST still considered "state of the art" for building web applications and how do REST API and GraphQL design architectures stack up against each other?*

Looking only at results from the literature study combined with the concrete numbers from the experimental study, the answer should be that REST is considered state of the art for fetching data from web applications. However, other parameters speak for GraphQL that need to be discussed. It allows us to avoid over and under fetching by doing our calls, it has only one single endpoint and is also very easy to update. The advantage of a solution with REST is that there is a long history of usage. This provides a broad knowledge base when seeking answers to questions about the technology. A major advantage of the REST API is that the application becomes more scalable. Typically we use JSON data as an interchange format, which is more compact and smaller in size compared to the XML file (to name an example). It is also important to notice that it is stateless on the server-side. Therefore for every request that we make will be processed independently from the previous ones.

Research Question 2: *How does caching affect REST API and GraphQL? Performance, needs, possibilities, and obstacles.*

When caching was used, it positively impacted both of the technologies. Performance clearly improved, which can be seen in the figures. Possibilities with the REST compared to GraphQL is that when we are making GET requests, REST takes advantage of caching in a better way, since it provides many endpoints it is possible to configure a web cache to match certain URL patterns. Speaking of GraphQL we only have one single endpoint where most of the queries are sent. Since each query can be different, it becomes harder to use this type of caching using GraphQL, therefore it becomes an obstacle. Possibilities are that those two techniques have much in common, in our view, it shall not be considered a replacement to the other. The advantage of REST is that you can specify each endpoint for the specific need. What is the best solution then? It is a difficult question to seek answers to since it depends on various things. Let's take two examples that with REST you can make an API call that returns a list of users. This involves setting up some kind of dispatcher on the back end that listens to the actual model being updated. On the other hand, with GraphQL you can render and pull data directly from the actual subscription.

7 Conclusions and Future work

7.1 Conclusion

First of all, it is important to point out that these are both incredibly strong techniques. Where both REST and GraphQL have their advantages but also disadvantages to consider before choosing which to use for your project. What we noticed early on is that a REST API requires more experience from you who will implement it, to create it in the best possible way to make it perform in a good way. REST is scalable which means that you can scale your applications without too much difficulty. Positive is also that it can handle different types of calls to return different data formats. This means that software engineers can build services that meet the specific needs of a diverse base of users. GraphQL is however more straightforward in terms of implementation, it creates fast iterations to the front-end possible. Even though REST performs faster in these tests, the authors believe that it can be more beneficial and effective in many projects to implement a system in GraphQL, if the system needs to ask questions as in REST means that two or more endpoints are contacted. Worth mentioning is that it can be time-consuming to replace working REST services with equivalent GraphQL services, but migration from REST to GraphQL can be of value if the system will be used for a long time to come as GraphQL may continue to grow. The conclusion here is that REST is the best solution over GraphQL in the tests we have done, measuring total response time in milliseconds. To summarize, the authors believe that REST is still considered state of the art when it comes to web applications. At least according to the studies carried out. As soon as that caching is involved, REST puts it into effect where GraphQL has no automatic caching system and this shows in the measurements. Implementing caching to GraphQL also adds a level of complexity. A complexity that may not be needed when there is a need for a simple API. Although, the tests carried out clearly display the benefits of such implementation. When caching is implemented the total response time is significantly improved.

When it comes to choosing GraphQL over REST, there are many more factors to look at than just performance criteria. Factors that could be for example bandwidth usage for a mobile application, where GraphQL is the better choice. In terms of built-in caching and monitoring facilities, REST is the better choice.

7.2 Future work

There can always be more testing carried out within these conditions. Different types of data variations and input sizes are altered. The tests have been executed on a controlled local network and future testing could be on a remote network to see potential effects. The tests have been carried out with the use of only one type of implementation on each technique, due to the time limitation. It would be interesting to compare different types of implementations with different types of programming languages on both techniques.

8 References

[1] Oggier, Camille. *How fast GraphQL is compared to REST APIs.* Haaga-Helia University of Applied Sciences, Helsinki. 2020. [Cited 2021- 02-24].

[2] Biehl, Matthias. *GraphQL API Design, Volume 5*. API-University. 2018.

[3] Wieruch, Robin. *The road to GraphQL: 2018 Edition*. Leanpub. 2018.

[4] Handy, Jim. *The Cache memory book: Second Edition.* Academic Press, INC. 1998.

[5] Hartig O, Pérez J. Semantics and Complexity of GraphQL. WWW '18: Proceedings of the 2018 World Wide Web Conference [Internet]. Lyon: International World Wide Web Conferences Steering Committee Republic and Canton of Geneva, Switzerland; 2018 Available from: https://dl-acm-org.ezproxy.hkr.se/doi/abs/10.1145/3178876.3186014

[6] GraphQL Foundation. Schemas and Types. [Internet][Cited 2021-04-15] Available from: https://graphql.org/learn/schema/

[7] Suresh Kumar Mukhiya, Fazle Rabbit, Violet Ka I Pun, Adrian Rutle, Yngve Lamo. A GraphQL approach to Healthcare Information Exchange with HL7 FHIR, Elsevier. 2019

[8] Ying, Guo. Fang, Deng. Xiudong, Yang. *Design and Implementation of Real-Time Management System Architecture based on GraphQL.* Nanjing. 2018. [Internet] [Cited 2021-04-15] Available from: https://iopscience.iop.org/article/10.1088/1757-899X/466/1/012015/meta

[9] Charboneau, Tyler. Breaking Down SmartBear's 2020 State of API Report. [Internet] 2020-08-27 [Cited: 2021-04-15] Available from: https://nordicapis.com/breaking-down-smartbears-2020-state-of-api-report/

[10] The New Technical Writer. Understanding the API Value Chain 2015.

[Internet] 2015-08-21 [Cited: 2021-04-13] Available from:

https://thenewtechnicalwriter.wordpress.com/2015/08/10/understanding-the-api-value-chain/

[11] Apollo Documentation Home. Configuring the cache. [Internet]
[Cited: 2021-04-16] Available from:
https://www.apollographql.com/docs/react/caching/cache-configuration/

[12] Karlstöm, Oskar. Zaar, Christian. *REST vs GraphQL: A comparison*. Kristianstad University, 2020. [Cited 2021-05-05]

Appendices

Appendix 1: GraphQL Server, Schema and StudentCreator code

```
JS server.js > ...
1   const bodyParser = require('body-parser');
2   const cors = require('cors');
3   const express = require('express');
4   const db = require('./db');
5
6   const port = process.env.PORT || 9000;
7   const app = express();
8
9   const fs = require('fs')
10  const typeDefs = fs.readFileSync('./schema.graphql',{encoding:'utf-8'})
11  const resolvers = require('./resolvers')
12
13  const {makeExecutableSchema} = require('graphql-tools')
14  const schema = makeExecutableSchema({typeDefs, resolvers})
15
16  app.use(cors(), bodyParser.json());
17
18  const  {graphiqlExpress,graphqlExpress} = require('apollo-server-express')
19  app.use('/graphql',graphqlExpress({schema}))
20  app.use('/graphiql',graphiqlExpress({endpointURL:'/graphql'}))
21
22  app.listen(
23     port, () => console.info(
24        `Server started on port ${port}`
25     )
26  );
```

Figure 6.1: GraphQL Server code

```
schema.graphql
1   type Query {
2      students:[Student]
3      getTime:String
4   }
5
6   type Student {
7      id:ID!
8      firstName:String
9      lastName:String
10     email:String
11  }
```

Figure 6.2: GraphQL Schema, which represents the kind of object to fetch from the service, and what fields it has.

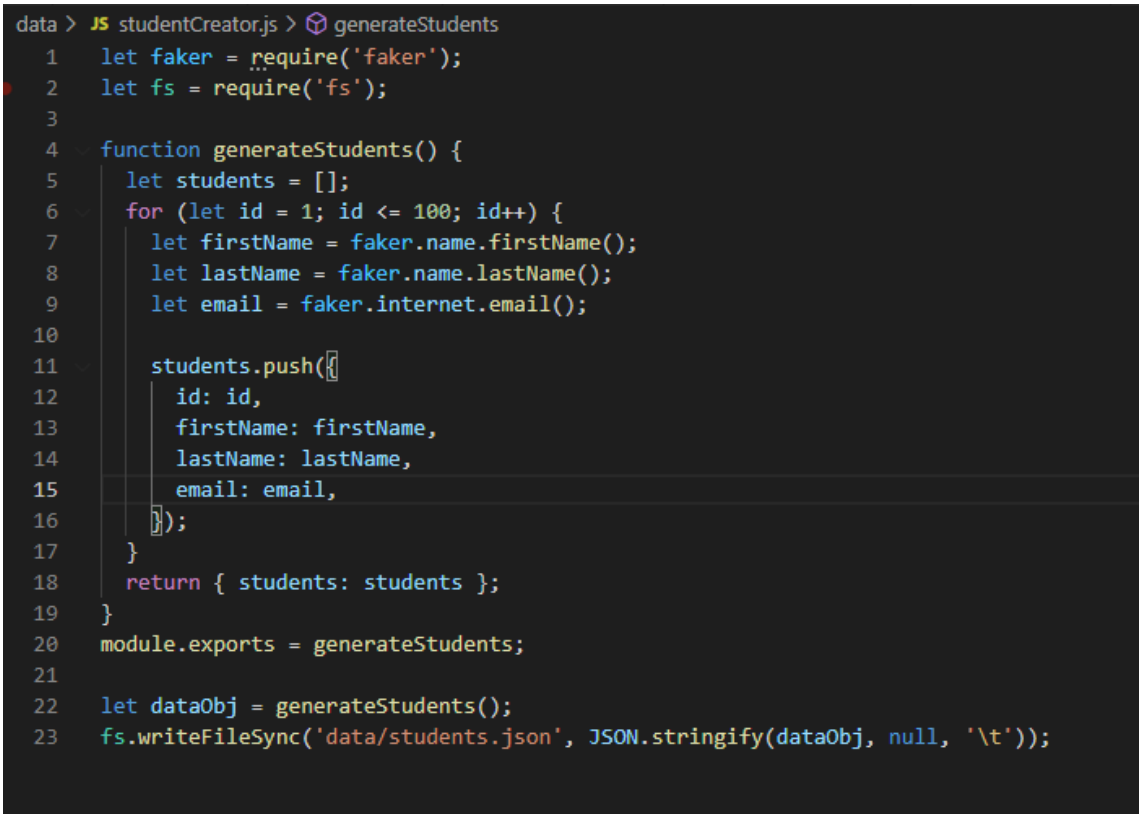

Figure 6.3: Function to generate students to the request.

Appendix 2: GraphQL Client

src > JS graphql-client.js > loadWithApolloClient > query

```
1
2   const ApolloClient = require("apollo-boost").ApolloClient
3   const createHttpLink = require("apollo-link-http").createHttpLink
4   const InMemoryCache = require("apollo-cache-inmemory").InMemoryCache
5   const crossFetch = require("cross-fetch/polyfill")
6   const gql = require('graphql-tag')
7
8   const client = new ApolloClient({
9      link: createHttpLink({uri:`http://localhost:9000/graphql`, fetch: crossFetch}),
10     cache: new InMemoryCache()
11  })
12
13  const loadWithApolloClient = async () => {
14      console.log("inside apollo client function")
15        const query = gql`{
16           getTime
17           students {
18              id
19              firstName
20              lastName
21              email
22           }
23        }`;
24        const {data} = await client.query({query})
25        return data;
26  }
27
```

Figure 6.4: Client side with Apollo-client function to enable caching.

```
27
28  const loadWithoutCache = async () => {
29      const response = await fetch('http://localhost:9000/graphql', {
30          method:'POST',
31          headers:{'content-type':'application/json'},
32          body:JSON.stringify({query:`{
33              getTime
34              students {
35                  id
36                  firstName
37                  lastName
38                  email
39              }
40          }`})
41      })
42      const rsponseBody =  await response.json();
43      console.log(rsponseBody.data)
44      return rsponseBody.data;
45  }
46
47  let i;
48  for (i = 0; i <100000; i++) {
49      loadWithoutCache().then(data => {
50          console.log(data)
51      })
52  }
```

Figure 6.5: Client side with function to disable caching.

Appendix 3: Rest Client and Server sides

Client.js > ...

```
1   const http = require('http');
2
3   for (i = 0; i < 100000; i++) {
4       var url = 'http://127.0.0.1:3000/getStudents';
5       http.get(url, (resp) => {
6       let data = '';
7
8       resp.on('data', (chunk) => {
9           data += chunk;
10        });
11
12      resp.on('end', () => {
13          console.log(data);
14          console.log("Response received");
15      });
16
17
18      }).on("error", (err) => {
19      console.log("Error: " + err.message);
20      });
21
22  }
23
```

Figure 6.6: Rest client side

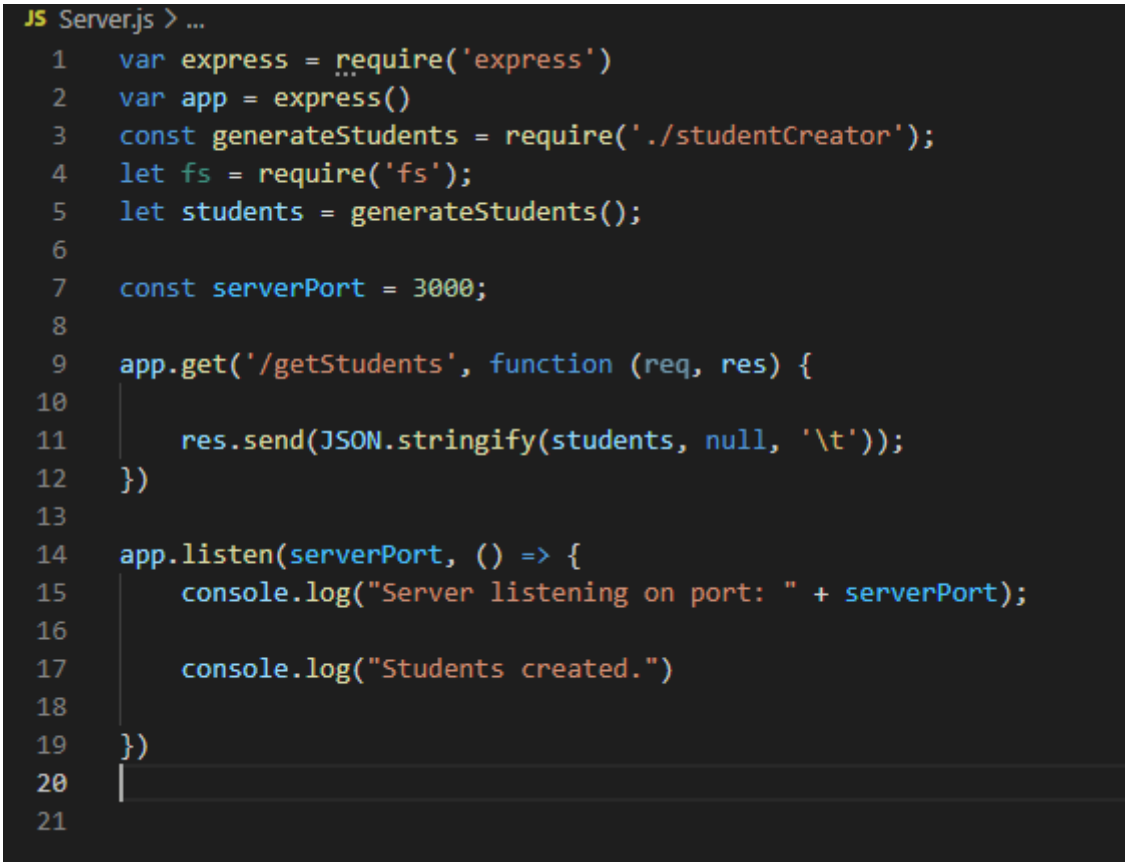

Figure 6.7: Rest Server side no caching

```
let students = generateStudents();
const NodeCache = require( "node-cache" );
const myCache = new NodeCache();

const serverPort = 3000;

app.get('/getStudents', function (req, res) {

    if(myCache.has('uniqueKey')){
        console.log('Retrieved value from cache!')

        res.send(JSON.stringify(students, null, '\t'));
    }
    else{

     myCache.set('uniqueKey', req.query)

     console.log('Value not present in cache,'
          + ' performing computation')
     res.send("Result: " + req.query)


     console.log(req.query);

    }
})

app.listen(serverPort, () => {
    console.log("Server listening on port: " + serverPort);
    console.log("Students created.")
})

```

Figure 6.8: Rest Server side with caching