Tampere University

Markus Ylisiurunen

GRAPHQL QUERY LANGUAGE'S FEASIBILITY IN A MICROSERVICE ARCHITECTURE

Faculty of Information Technology and Communication Sciences
Bachelor's Thesis
April 2019

ABSTRACT

Markus Ylisiurunen: GraphQL query language's feasibility in a microservice architecture
Bachelor's Thesis
Tampere University
Information Technology
April 2019

The goal for this thesis was to introduce a new query language called GraphQL and compare it to a more generally known communication pattern called REST. More specifically, the comparison of the two was done in the context of microservices by evaluating the options based on performance, data consumption, technology maturity and suitability, scalability, and speed of development.

The thesis first discusses the requirements for communication in a microservices architecture and introduces some of the currently used communication protocols, such as Protocol Buffers and Thrift. Then we move on to GraphQL and listing its features mostly based on the official GraphQL specification from 2018. Finally, we compare GraphQL to REST based on the earlier criteria.

We found that GraphQL provides much more flexibility, is better with data consumption, can increase the speed of development and possibly is easier to scale in the microservice environment with schema stitching. On the other hand, we found that REST can perform significantly better at least in comparison to some implementations of GraphQL.

Keywords: GraphQL, REST, microservices, API

The originality of this thesis has been checked using the Turnitin OriginalityCheck service.

CONTENTS

List of Symbols and Abbreviations . . . iii
1 Introduction . . . 1
2 Microservice pattern . . . 3
2.1 Monolithic applications . . . 3
2.2 High-level microservice architecture . . . 4
2.3 Containers . . . 5
2.4 Internal communication . . . 6
2.5 External communication . . . 7
2.6 Communication protocols . . . 8
2.6.1 REST . . . 8
2.6.2 Protocol Buffers . . . 9
2.6.3 Apache Thrift . . . 9
3 GraphQL . . . 11
3.1 Language . . . 11
3.1.1 Syntax . . . 11
3.1.2 Type system . . . 12
3.1.3 Introspection . . . 13
3.2 Operations . . . 13
3.2.1 Queries . . . 14
3.2.2 Mutations . . . 15
3.2.3 Subscriptions . . . 16
4 Comparison to REST . . . 17
4.1 Performance . . . 17
4.2 Data consumption . . . 18
4.3 Technology maturity . . . 19
4.4 Scalability . . . 20
4.5 Speed of development . . . 21
5 Conclusion . . . 22
References . . . 24

LIST OF SYMBOLS AND ABBREVIATIONS

API	Application Programming Interface
JSON	JavaScript Object Notation
REST	Representational State Transfer
RPC	Remote Procedure Call
XML	Extensible Markup Language

1 INTRODUCTION

Microservices have gained popularity in recent years even in the largest and most impactful technology companies out there, such as Netflix, who are running hundreds of microservices in their systems [1]. This thriving architectural pattern can improve the scalability, speed of development and flexibility in making decisions regarding technology stacks or third-party services. Microservices are based on the idea of splitting a single monolithic application into multiple smaller, self-contained, services that work together to provide the functionality of the full application. This separation immediately gives us some obvious advantages such as being able to pick the right technology stack for each service without being tied to a specific programming language. There are also other, more subtle, benefits for splitting the functionality of a larger application to smaller services. At organizational level teams can be responsible for smaller services and therefore making the deployment process easier and much faster as every service can be tested and deployed separately. Naturally, this introduces new challenges of which an important one is communication between multiple services in a way that is performant enough and can scale to different technologies and platforms. The stricter communication requirements force the developers to design the services' Application Programming Interfaces (APIs) so that no new change can break other services. This produces more robust services that can be combined to build a highly available ecosystem of microservices. [2]

Nowadays, one of the most popular communication patterns is Representational State Transfer (REST). It is a quite lightweight set of principles that are used to build consistent, stateless and predictable APIs for various use cases [3]. Other solutions for the same problem space are, for example, Google's Protocol Buffers and Apache Thrift. These are designed to be stricter than REST and they provide additional mechanisms, such as a type system or code generation based on an interface definition file, for improving performance and usability. [4, 5] In addition to the mentioned protocols, there are many others as well but we will ignore them for the most part of this thesis.

Like any technology, microservices have their own challenges as well. As mentioned before, one of the most important ones compared to monolithic applications is the communication paradigm of microservice architecture. As the services are running completely isolated from each other, they have to communicate via a well-defined API over the network [2]. The microservices have to be able to communicate with each other in a reliable and consistent fashion but they also need to expose interfaces to the outside world. The internal communication is crucial because services might depend on each

other, for example, to fetch information about the customers' payments. In addition to internal service-to-service communication, some of the services have to expose public APIs to the outside world. These APIs will be used by third-parties or client applications. It is preferable to make these APIs as easy to use as possible so that they are programming language and platform agnostic. [6] As previously mentioned, Protocol Buffers and Thrift are binary protocols and they are best suited for internal communication between services because they require generated code in order to be usable. This could become hard to achieve in a reliable way if third-party clients would try to connect to an API that uses either Protocol Buffers or Thrift as they would need to have access to the interface definition files for the API and generate the actual code from them. Therefore, we will mainly focus on discussing REST in this thesis.

In this thesis, we introduce and evaluate a relatively new query language called GraphQL which is being developed by Facebook [7]. GraphQL has been quickly gaining popularity and companies have started to rebuild their APIs to support GraphQL. One of these companies, in addition to Facebook itself, is GitHub [8]. GraphQL introduces a new query language syntax that can be used over the well-supported HTTP(S) protocol to query and mutate data. It is designed to be intuitive and flexible to use by letting the client applications define the data they want to query or take an action on. GraphQL is, as the name implies, based on a graph structure of the data making it hierarchical. [7] As GraphQL has gained a lot of interest and it seems to be becoming more and more used in many, even surprising, places, it is important to evaluate its feasibility for the microservice architecture and compare it against REST being one of the most popular approaches at the moment.

As mentioned before, we will introduce GraphQL and compare it against REST especially in the context of microservices. The two communication patterns will be evaluated based on performance, data consumption, technology maturity and suitability, scalability, and speed of development. This thesis first introduces microservices in more detail and discusses their requirements for the communication protocol. Then we move on to briefly introduce some of the currently most used communication protocols and GraphQL in more detail. Finally, we compare GraphQL to REST in the key areas described above and draw a conclusion of whether GraphQL fits the microservice pattern and whether it might be capable of replacing REST in the future.

2 MICROSERVICE PATTERN

Microservices as a whole is a massive and complex topic which is why we will only give a brief introduction to them and some of the building blocks one should focus on when implementing a system using the pattern. We will discuss only the parts of microservices that are relevant to the topic of this thesis, communication between services. This includes the high-level architecture of microservices, communication between services (internal communication) and the communication from the client to the services (external communication).

Microservices is increasingly becoming a common architectural pattern. Using microservices lets the developers move and innovate fast and develop their services in smaller teams with the right technology stack for the problem they are trying to solve. [2] Companies, such as Netflix, have talked a lot about their process for moving to microservices and they have also open-sourced some of their efforts [1]. The fact that microservices have proven to be a good, working, choice at such a large scale strengthens the argument that we are moving away from monolithic applications and increases the need for researching microservices in order to be able to build them to be more robust and efficient.

2.1 Monolithic applications

When talking about microservices, we usually compare them against monolithic applications. A monolithic application is an application that is written in a single code base and is usually composed of three components, user interface, business logic, and a database. [9] User interface is the part of the application that handles visualizing data and letting the user interact with the application. This can be, for example, a website or a mobile application. Business logic is everything required to handle the interactions within the application which includes, but is not limited to, user authentication, handling user actions and interacting with integrated third-party services. Database, in this context, is any place where the customers' data is stored for longer periods of time.

Now, when the monolith is run, everything is included in the single instance of the application. This imposes some problems especially when the application is being scaled up and to which microservices are trying to find a better solution [10]. Microservices can be scaled up independently of each other which might be a major problem for a monolith. If a single monolith consists of multiple services but they are run as one, the monolith has

to be scaled so that each of these integrated services is scaled at the same time. This works if the load for each service is similar but most often this is not the case. One of the key benefits of microservices is that they solve the previous problem by being able to be scaled up independently of each other. [2]

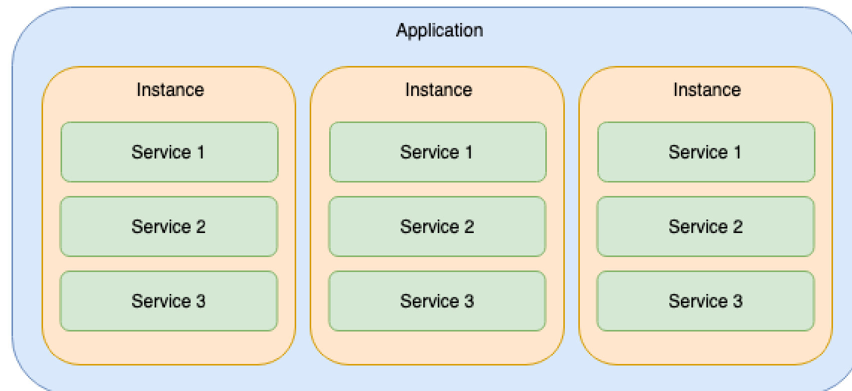

Figure 2.1. *A monolithic application scaled up to three instances. Image is based on [2].*

As one can see from figure 2.1, each service is replicated three times even if there would be no need for that. In reality, the load distribution is very unlikely to be equally distributed to each service making the scaling of all services at the same time use unnecessary resources. [2]

2.2 High-level microservice architecture

As the cloud has become a widely adopted platform for building applications, microservices have become one of the most used ways to build software on top of these platforms. It is an architectural pattern for creating highly scalable software where each service handles a subdomain of the entire application's functionality. Services are independent in a sense that they are completely isolated from each other and only can be interacted with through a predefined API. [6]

Now, let's consider how microservices are built in comparison to monolithic applications. Microservices split the functionality to multiple separate applications that work together by communicating over a predefined API. Each microservice might have its own database and they are able to work in complete isolation from other services. However, some services might depend on other services and therefore need them to be available for them to be able to provide their functionality properly. Sam Newman has put this decoupling issue to words in his book "Building microservices". He says that the golden rule for deciding what each service should do is to think whether you can make a change to the service and deploy it without breaking any other services. If the answer to this is no, then it might be very hard to get the benefits of microservices as the key principle of microservices is to decouple functionality from each other and develop them separately. [2]

A good way of understanding the general idea of what a microservice architecture might look like is to start from a monolithic application and try to break it down to multiple microservices. Even if an application would be a monolith, it contains functionality that is separate from other parts of the application. An example of such a module would be an emailing module that sends emails to the application's users in response to various events. In a monolith, this would be implemented as a module of code and other parts of the application could use the module to send the emails on their behalf. However, we could extract this emailing module as a microservice and access its functionality over the network through an API. Now, instead of calling a method of a code module, the service would make a network call with the required information about the email to be sent and the emailing service would handle the actual sending of the email.

Monolithic applications had issues with scaling the application up. With microservices, each service can be scaled up independently making them much more flexible in terms of scalability. Each service can also be deployed separately increasing the speed of development. [2]

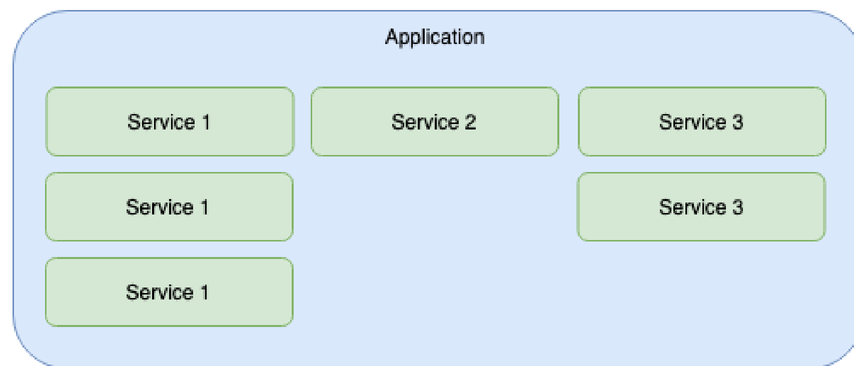

Figure 2.2. *Each service scaled up independently in a microservice architecture. Image is based on [2].*

As shown in figure 2.2, each service can now be scaled independently of each other to respond to the demand for that particular service. For example, the emailing service might not need as many resources as some other service handling core business logic functionality.

Microservices, quite obviously, would have a lot more things that we could discuss here. However, the most important aspect for us is to understand that the communication happens over the network and not through direct method calls within a single program.

2.3 Containers

One might be thinking about how these services can be deployed and ran so that they are able to work together in the desired manner. This is one of the challenges of microservices as now each service has to be managed and deployed separately. However, it is worth mentioning that this is also an advantage of microservices as now each service

can be run wherever it is best suited for the type of workload the service has. One of the fastest growing technologies for managing microservices is Docker and the ecosystem built around it. Docker is a container technology which lets the developers wrap their applications to containers, operating system and machine agnostic units of software. What this means is that the encapsulated services can be run on any machine or a cluster of machines and the platform can manage things like scaling, monitoring and updating the services in a consistent and reliable fashion. One of the most popular projects built around Docker is Kubernetes. [11, 12]

Containers are an advanced topic by themselves but the core idea is to encapsulate some software within its own operating system environment which then can be run on any machine or a cluster of machines. Containers are a flexible, lightweight and scalable way to run simple, or even very complex applications, anywhere. By using containers to encapsulate each service, it allows every service to be written in any language or use any number of third-party libraries or services alongside the actual service. [11] As every service might be written in a different language and they might be run on different physical machines, it means that services can't rely on inter-process communication. Thus, whether the communication is synchronous or asynchronous, it must be implemented over network calls [6].

2.4 Internal communication

As mentioned earlier, microservices have to be able to communicate with each other. This poses some challenges as the services might be located on different physical machines, they might contain multiple instances of the same service or they might be temporarily unavailable. This raises the question that how can two services communicate with each other in a reliable manner without relying on a programming language or operating system specific mechanisms. This problem can be solved by making each service use network calls for all communication. [6]

A common pattern where services have to communicate with each other is the so-called API gateway pattern. In this pattern, there is one service in front of the other services that is exposed to the outside world. The gateway service accepts requests over, for example, HTTP(S) and aggregates the required information from multiple services before sending it back down to the client. In this scenario, the API gateway service has to be able to request information from multiple services in order to be able to construct the final response. [6]

Usually, the services are in a private network where they can reach each other quite fast improving the latency for their communication. For example, Docker Swarm creates a private network spanning over the machines in the cluster to improve the networking between services [11]. As most of the services are built specifically for the application that is being developed, internal communication could use a more performant protocol than

REST. However, one could argue that using the same protocol for both internal and external communication decreases the needed development efforts making the development faster and less error-prone.

Even if each service could implement their own protocol for communications, it is important that they all use the same protocol in order to be consistent and minimize the development effort. It is also important to use a protocol that is not programming language or platform specific or otherwise restrict the use of other technologies in services or clients. This is why the services might use something like REST, Protocol Buffers or Thrift. [6]

2.5 External communication

External communication consists of the interaction between the client applications (web, mobile, IoT, etc.) and the back-end consisting of all the services. The client application might be anything from a smart TV to a mobile application to an IoT device. This sets some requirements as to how communication has to happen in order for it to be efficient and easy enough to develop. The external communication protocol has to support all possible clients as different clients have different requirements. For example, an IoT device might have very limited performance or memory which means that it is not able to include large third-party libraries or do heavy computational processing in order to be able to communicate with the back-end. However, every client is still required to be able to connect to the back-end. This means that communication has to use a protocol that is supported by a wide range of programming languages and is not too computationally heavy for even the most lightweight clients. [6]

Protocols like Protocol Buffers or Thrift require the use of code generation in order for them to be usable. This process takes the interface definition files and generates programming language specific code which can be invoked within the program in order to send and receive requests from other services. [4, 5] This makes the use of these protocols quite difficult because each client would have to have access to the definition file and to support the protocol. This is part of the reason why protocols such as REST have become popular because they are very lightweight and can be used over regular HTTP(S) requests.

It is, of course, not required that the same protocol is used for internal and external communication but it definitely does not harm if it can be used for both. However, nothing comes without downsides. Using the same protocol for both internal and external communication might be a compromise for both. For example, there might be available a better performing option which could be used for internal communication but not for external communication.

2.6 Communication protocols

Even if we will not be discussing other than REST in more detail, it is worth knowing about other protocols as well. It gives context and helps us understand what compromises different protocols have to make in order to be better in some areas. In this section, we will introduce REST in more detail as well as give a brief introduction to a couple of other protocols.

2.6.1 REST

REST itself isn't a technology but rather a set of principles that defines how the API endpoints should be designed and used. One of the most important principles of a RESTful API is that it should be completely stateless. Every request should be idempotent, meaning that whatever request you send to the server, it should always, regardless of how many times you send it, return the same response. REST was designed to be used for web applications over HTTP(S) and given the application implements REST properly, it can use a lot of the web's architecture in its advantage. [3]

Another important principle of REST is the use of HTTP request methods. HTTP request methods represent the wanted action that the client wants to perform. The most common values for the HTTP request method are *GET*, *POST*, *DELETE* and *PUT*. These are for retrieving, inserting, deleting and updating a resource. [13]

In addition to the use of HTTP request methods, another equally important principle is the use of identifiers and constructing the URI schema in a proper way. REST proposes a consistent and predictable way for constructing the URIs of each individual resource as well as a collection of resources. [3] To explain this, let's consider the following scenario. Let's decide that we have an online store for books with hundreds of books. Following the REST's principles for constructing the URI for the entire collection of books would result in a URI of */books*. Sending a *GET* request to this URI would return the full list of available books. If the client wanted to get a single book, it would send a *GET* request to */books/123* where *123* is the ID of the book. The same URIs can be used with other HTTP request methods, such as *DELETE* which would delete the resource for the book with the ID of *123*.

The URI schema can extend to include nested information such as information about the author of the book. The URI for such a resource would be */books/123/author*. Another way the previously introduced URI schema could be extended is by adding query parameters to the endpoints. [3] In the books store scenario, this could be added as a query parameter for getting the list of books only in a given year, for example, */books?year=2019*.

As said previously, and as one might be able to gather, REST can be easy to understand and implement. It is also very predictable lowering the learning curve of implementing

your own APIs but also understanding third-party APIs. However, REST can also become quite bloated as the number of endpoints can grow into a very large number. It might also not be flexible enough for multiple client types increasing the number of needed endpoints to cover all the special needs of various clients. Depending on the framework or type of implementation, REST can also be a bit risky as it can require some boilerplate code for each endpoint. In my own experience, this might open security issues such as unauthenticated endpoints.

In addition to the backend side of issues, there are also issues in the frontend as well. One of these is the fact that the client has to send multiple requests to aggregate all the data it needs. These requests might be sent in parallel but they might also have to be sent sequentially if the next request needs information from the response of the previous request.

Overall, REST is more of a collection of rules and principles than a complete protocol which will always be implemented in the same way. In my opinion, this is one of the weak points of REST as it is quite a loose specification.

2.6.2 Protocol Buffers

Protocol buffers are developed by Google for serializing structured data to be used in communication purposes. It is a language- and platform-neutral solution for the problem and it works by defining the wanted data structures and types, compiling it down to automatically generated source code in a specific language and by using the generated code to read and write data to the protocol buffer format. [4]

In short, protocol buffers are meant for efficient data transfer between two processes where each process can read and write the data in an efficient manner. In addition to pure data structure serialization, protocol buffers also provide a way to define remote procedure calls (RPCs). By defining the service and its available methods in a proto file, the protocol buffer library can take care of executing the procedure when requested. [4]

Therefore, protocol buffers can be considered a polished REST API where it takes care of data structure serialization, data types, and remote procedure calls. However, it doesn't really solve many of the REST's problems. An additional challenge might turn out to be the fact that now every client also has to be able to use the protocol buffer library's generated source code in order to be able to communicate with the API.

2.6.3 Apache Thrift

Apache Thrift was originally developed by Facebook and it was open sourced in 2007. Facebook had been having difficulties with scaling up its system to meet the demand. They also had been developing their services in multiple languages and were on the

lookout for an open source solution that would provide a good solution for communication between services at their scale. However, they did not find any existing solutions that would satisfy their criteria so they decided to build their own. [5]

Thrift's core concept is to let the developers write the API definition in a simple definition file in a language-neutral syntax. This file is then taken by the Thrift compiler system which lets it generate programming language specific code. This generated code can then be used in any supported language to interact with any service using the same definition regardless of the language of the other service. However, Thrift protocol defines some restrictions for the transport layer of the protocol but these are implementation details and are not interesting to us in this thesis. [5]

One might notice that Thrift is very similar to Protocol Buffers. All communication protocols come down to being able to write some state to a transmittable object and being able to rebuild that in the receiving service. The difference between a text-based protocol, such as Extensible Markup Language (XML) or JavaScript Object Notation (JSON), to a binary protocol, is that a text-based protocol does not have the prior knowledge of the incoming data and they have to parse the payload character by character. On the other hand, binary protocols have the type information and structure of the incoming data prior to receiving the data and they can use this in their advantage and this way improve their deserialization performance significantly. [14]

3 GRAPHQL

GraphQL is a query language designed for building flexible API endpoints for a variety of clients and giving the client the ability to query the data however they see fit the best. GraphQL has a new syntax built for supporting the design of the query language. It is not tied to any specific programming language nor is it a programming language itself but rather an abstraction for querying the data from an API endpoint by a client. [7] The underlying programming language which implements the GraphQL API endpoint can decide how they want to construct the program and what are the abstractions for developers to use while building the GraphQL server. Most often the API is built by using a third-party library that takes care of most of the GraphQL server's requirements.

GraphQL APIs expose a single endpoint where the client can send its queries to. This single endpoint has the knowledge of all the data that the system or microservice is capable of retrieving or manipulating. The endpoint is constructed from a graph of data, as one can figure from the name GraphQL. This graph of data can then be queried in a very flexible fashion. [7]

3.1 Language

GraphQL introduces a new query language syntax which is built to be intuitive and flexible to use and giving the developers the tools to describe their actions against the underlying data. GraphQL is hierarchical, strongly typed, introspective and the queries are client specified. [7]

3.1.1 Syntax

GraphQL is used via a completely new syntax which was built just GraphQL in mind. One of the most important aspects of the new query language syntax is the fact that the queries are very similar to the JSON output which makes them intuitive to use and understand [15]. The following example is a query to retrieve a person's first name and their parents' names.

```
{
  person(id: 1) {
```

```
    firstName
    parents {
      firstName
    }
  }
}
```

This example demonstrates GraphQL's syntax quite well. The hierarchical aspect becomes clear by seeing the way the client is capable of querying the person and their parents. In fact, the client could choose to go as deep into the hierarchy as it wants just by adding the *parents* field to the person's parent similar to what now is done to the person itself [7]. Another aspect of GraphQL this demonstrates is the use of arguments in the language. When the client queries a person, it includes an argument for the *id* of that person. This lets the GraphQL API know that the client is interested only in a specific person with the *id* of *1*.

As previously mentioned, the GraphQL syntax strongly resembles JSON. The response to the earlier query could be the following.

```
{
  "person": {
    "firstName": "Jack",
    "parents": [
      { "firstName": "Robert" },
      { "firstName": "Anne" }
    ]
  }
}
```

If compared side by side, one can see that the two are very much alike. This is intentional and it makes the queries intuitive and understandable [15]. Naturally, there is a lot more to GraphQL's features that could be covered here. However, it is most important for us to understand the basics of how a GraphQL could be used and we will not go more into detail to other language features of GraphQL.

3.1.2 Type system

Programming languages can usually be divided into two subcategories, strongly and weakly typed languages. Strong typing means that every variable has to have a specific type that has to be known beforehand. In contrast, weak typing means even if the variables have to be declared beforehand, their types do not. The type of a variable is dynamically determined at runtime. Both have their advantages and disadvantages but it is quite universally agreed that strong typing gives the developers extra assurance that

everything works as they should. [16]

Even if GraphQL is not a programming language, it has a concept of data types. There are some primitive types, such as integer and string, and user-defined types that are composed of the primitive types. The types are used to determine whether a query is valid and, in addition, the information about the types of each field can be used for giving intelligent suggestions or auto-complete features where suitable. [7]

```
type Person {
  firstName: String!
  lastName: String!
  parents: [Person]
}
```

The example above defines a *Person* type which was used in the earlier query. It defines the first and last name as non-nullable strings as well as an array of other persons as its parents.

3.1.3 Introspection

As the GraphQL API has information about all possible types and their relations defined in its schema, this information can be exposed and used by third-party clients or services to provide additional features for the API. Introspection means that the API exposes its schema that can be queried similar to any other data by a GraphQL client. [7] This additional information enables a few interesting use cases.

The first use case for using the introspection information is automatically generating documentation for the API. The client is able to get information about each type and their fields as well as a description for each field [7]. This information can be used to automatically generate documentation for the API which lists all possible data that can be queried, parameters for each query and all return types and their fields.

Another use case for the introspection information would be to stitch multiple schemas together. This is not part of the GraphQL specification but third-party tools, such as Apollo, have built tooling to accomplish this [17]. By being able to stitch multiple subschemas into one, we are able to build multiple microservices and expose all the public endpoints as a single API endpoint. This can make both development and using the API much easier.

3.2 Operations

GraphQL provides flexible operations for fetching and mutating the underlying data in various different ways. In GraphQL there are two concepts, queries, and mutations. Queries are for fetch-only actions where the client wants some part of the data and mutations

are when the client wants to mutate the data in some way. In addition to queries and mutations, GraphQL provides a third operation called a subscription. Subscriptions are long-lived requests that fetch new data as it comes into the server. [7]

3.2.1 Queries

Probably the most familiar and used feature of any query language is the way the client can retrieve any piece of the data. This is called a query in GraphQL. As mentioned before, GraphQL is based on a graph of data which means that queries are hierarchical. In this case, hierarchical means that the client can decide how deep into the graph they want to go within each query. [7] As an example, let's say that the endpoint has information about people and their parents, who are people themselves. The client might want to retrieve some person's parents' and grandparents' names in a single query. This can be achieved by sending the following query to the GraphQL endpoint.

```
{
  person(id: 1) {
    firstName
    parents {
      firstName
      parents {
        firstName
      }
    }
  }
}
```

The previous query would result in the following response.

```
{
  "person": {
    "firstName": "Jack",
    "parents": [
      {
        "firstName": "Roberta",
        "parents": [{"firstName": "Treva"}, {"firstName": "Amber"}]
      },
      {
        "firstName": "Willis",
        "parents": [{"firstName": "Don"}, {"firstName": "Anne"}]
      }
    ]
  }
```

```
}
```

There are a couple of interesting things worth mentioning in the query above. The first thing is how the client can decide how deep they want to go in the hierarchy. This is done by using the *parents* attribute on a person which retrieves the current person's parents and it can be done as deep as needed. Another thing here is the parameter *id*. This is used to identify the person we are interested in in the first place.

Queries have many more features but they are not interesting to us in this thesis. We will focus on the basic features and concepts and go only as much into details as needed.

3.2.2 Mutations

In GraphQL, queries let the client retrieve a piece of the data. This is only the first half of what a usual API has to be able to do. Generally, the clients have to be able to manipulate the data in one way or another as well. In GraphQL, this is done with something called mutations. Mutations are operations that first does a write operation and is then followed by a fetch operation [7]. What this means is that the write operation part of the request is an action that modifies a part of the API's data and then is able to retrieve the modified data in a single request. Hence, mutations can be thought of as an action followed by a query.

There is no distinction between mutations that insert, update or delete part of the data in GraphQL. They are all considered similar mutations. Each mutation can have a name that should describe what it is going to do when invoked. Similar to queries, mutations can also accept parameters such as the id of the person to delete. [7]

Below is an example of inserting a new person to the database.

```
mutation {
  insertPerson(firstName: "Pearl", lastName: "Kohler") {
    firstName
    lastName
  }
}
```

As seen in the previous example, mutations are called similarly to queries and can accept parameters as well. In fact, mutations are queries as they will only first run the mutation with possible side-effects and then run the defined query for *firstName* and *lastName* against the returned piece of data.

3.2.3 Subscriptions

Subscriptions are long-lived requests that can constantly receive new data as it comes into the GraphQL service. The response for a subscription request is not a single response but rather a response stream. It is a result of an underlying source stream which is a stream of some type of events. The response stream consists of the source stream events after an operation function has been applied to each one of those events. [7]

A subscription may be easier to understand through an example. Let's consider a chat application that has the concept of rooms. Rooms can have as many persons as wanted and each one can send messages to that room. Traditionally, there have been at least two methods for achieving a near real-time message delivery system. The first method would be to use long-polling where the client is constantly polling the service for new messages and will return in case there are new messages in the queue waiting for it to receive. The second method has been to use some type of persistent connection systems such as WebSocket or MQTT. This method requires an open connection to the server where it can receive or send data between the service and the client. GraphQL subscription is similar to the long-living connection such as WebSocket and MQTT. It opens a connection with a selection set on a subscription field [7].

Below is an example of the chat room concept [7].

```
subscription NewMessages {
  newMessage(roomId: 1) {
    sender
    text
  }
}
```

As one can see, subscriptions look very similar to queries and rightfully so. They act very similar to queries the only difference being that they will not complete after the response but will keep receiving events over time as long as the connection is kept open. [7]

At the time of writing this paper, subscriptions are the least supported feature of GraphQL. Partly because they are hard to implement in scale and require statefulness. Statefulness means that the server has to know about the connection and its attributes in addition to the entire dataset which the schema supports. In case the server is scaled horizontally, it can become very hard to accomplish this.

4 COMPARISON TO REST

Now that we have introduced microservices, REST, and GraphQL, we take a look at some of the differences and compare the two against each other. More specifically, we evaluate both options in the context of microservices.

4.1 Performance

Performance is obviously a big topic to take into account when deciding which technology should be chosen for communication in a microservice architecture. One important thing to keep in mind here is that, as mentioned in the previous chapter, GraphQL depends on the tools built for the specific language the GraphQL server is built on top of. The library has to parse the query, travel through the graph and invoke the necessary handlers for the nodes [7]. This is more work than what a typical REST library has to do and therefore can have a bigger effect on the performance. Hence, the implementation quality of the GraphQL server framework plays a quite significant role here.

In the paper "Efficient Communication With Microservices", Petter Johansson has built a test application for benchmarking the performance of GraphQL and REST. The services were implemented in C# using the ASP.NET framework and therefore had to use a .NET specific GraphQL library. Both services were designed to do the exact same operations and to not do any calculations for the retrieved data. [6]

Johansson did three separate experiments sending 100 requests each time and measuring the round trip time for each request. In order to minimize the load on the servers from affecting the results, he had a second-long pause between each request. The first test set was designed to fetch a fairly small amount of data from both services. In this test, REST had an average round trip time of 78.94 ms and GraphQL had an average of 142.92 ms. [6]

Johansson ran two other test sets with 25 rows and 50 rows of data. The procedure was the same as in the first test set and the results for all of the three test sets are provided below. [6]

As can be seen from table 4.1, GraphQL performed considerably worse in comparison to REST. The most active community for GraphQL is around JavaScript and Node.js so it might be possible that GraphQL implementations have improved quite a bit since this per-

Table 4.1. *Round trip performance test results [6].*

	Test 1	Test 2	Test 3
REST	78.94 ms	100.35 ms	84 ms
GraphQL	142.92 ms	256.80 ms	357.89 ms

formance test was made. In the future, it might be worth to redo the experiment described before and use another implementation of GraphQL. However, we must remember that not all services are built with the best technology for GraphQL and these results give a good understanding of the performance differences.

4.2 Data consumption

As we discussed in the GraphQL section, the client can specify what the query should return in response to a request. The client uses the special GraphQL syntax to describe its needs and it can request recursive data in a single network call. [7] This makes GraphQL very efficient in the way it almost never requires fetching too little or too much data.

One problem I have noticed with REST APIs is that it usually requires the client to send multiple requests in order to aggregate the needed data. Let's use the book store from before as an example. A website is trying to get a list of the best-selling authors and show their best-selling books in the same view. The client could first send a request to */authors?sort=bestselling&count=10* to retrieve the best-selling authors and their information. Now that the client has information about each author, it could send a new request for each author to */authors/<id>/books?sort=bestselling&count=10* to retrieve the author's best-selling books. This requires *authors_count + 1* network calls.

Now let's consider the previous example but this time using GraphQL. We are able to retrieve the same data in a single query and a single network call.

```
{
  author(sort: "bestselling", count: 10) {
    firstName
    lastName
    books(sort: "bestselling", count: 10) {
      name
    }
  }
}
```

The GraphQL query above would return the authors' names as well as their books' names in a single request. This uses GraphQL's graph-like structure to its benefit in order to retrieve the books' information simultaneously.

What comes to data consumption efficiency, GraphQL is a clear winner in this category. The client is able to decide what data it needs and can request exactly that and nothing else. It can also combine what needs multiple requests using REST into one request. Overall, GraphQL is much stronger in data consumption than REST.

4.3 Technology maturity

Technology maturity is a very important aspect to keep in mind when deciding what technology to use in an application. As one of the key principles of microservices is that each service is able to pick any technology stack they want, it is important to evaluate the state of GraphQL implementations available for the most common languages. REST does not require any other library than a basic HTTP server that is capable of receiving HTTP(S) requests over the network. Therefore, any programming language is able to support REST APIs. On the other hand, GraphQL requires much more from the library in order to be able to provide an API that fulfills the GraphQL specification's requirements for a GraphQL API.

Quite likely the most active community around GraphQL is the JavaScript community. Apollo is one of the most popular GraphQL server and client implementations that are used quite widely already. The GraphQL reference implementation is also written in JavaScript [18]. There is a repository that collects many of the GraphQL implementations for different languages and it shows quite an extensive list of programming languages that already support GraphQL [19].

However, REST is a quite safe choice because of its popularity and wide usage. It is also very lightweight about what dependencies it needs in order to be used. Any web framework is able to implement a REST API without other libraries as REST is mostly just a set of rules to follow. The same cannot be said about GraphQL. GraphQL consists of two parts, the server and the client. The server is considerably more complex than REST and it requires a lot of tooling to build the API with. The client is a different story. The client only needs to send basic HTTP(S) requests with the right type of request body [7]. Any programming language or device that is capable of supporting REST can also support GraphQL.

Ultimately, GraphQL's usability comes down to the implementations for different programming languages. Fortunately, the list of these implementations is quite extensive and it only grows as time goes by [19]. It is also possible to wrap other types of APIs into GraphQL entities enabling us to use even some smaller languages, such as Rust, which doesn't have a GraphQL implementation quite yet as it can be exposed as a GraphQL API just by building a lightweight wrapper around it. This isn't a perfect solution by any means and therefore the library availability is definitely an important thing to consider when deciding to go with GraphQL.

4.4 Scalability

When designing large software systems, scalability is one of the top concerns. If done incorrectly, it can lead to a complete redesign of the architecture which can be extremely costly. It is important to evaluate the available options and pick the one that will solve the problem in hand the best possible way. In this context, scalability means the ability to respond to an increasing amount of requests as well as supporting an increasing amount of services without too much overhead. The performance part was discussed in the section 4.1 already so we will focus on discussing the ability to support an increasing amount of services.

A popular pattern for REST APIs built with microservices architecture is to use an API gateway. The idea with an API gateway is to build a service that aggregates other services into a single endpoint making the use of the API easier and faster for the clients. [6] This can be quite scalable but it requires manual work to add new and modify existing endpoints. It is also an extra layer of complexity that can make development harder.

On the other hand, GraphQL can support schema stitching. Schema stitching is a process of combining multiple GraphQL APIs into one schema and exposing that as a GraphQL API to the outside world. Schema stitching can be used with microservices to automatically combine multiple service's schemas into one. [17] It is important to note that schema stitching is not part of the GraphQL specification but rather something a GraphQL library can implement on their own thanks to the graph-like structure of GraphQL [7].

There are some potential issues with schema stitching as well. One of these is if the developers want to combine different schemas' types together at the combined schema. Let's consider an example. We have two services, one handling books and other handling authors. Now, each author may have books associated with them as well but as the schemas come from different services, we cannot use the `Book` type within the `Author` type. Fortunately, Apollo's solution has a way to extend the subschemas to introduce new fields to their types [17]. This way we could extend our `Author` type to include the specific author's books and delegate that query to our book service's GraphQL API.

As mentioned before, schema stitching is library specific and might not be supported by all of the available libraries for different languages. Apollo is one of the supporting libraries and it can be used to build the schema merger service which then is exposed to the outside world. Apollo supports any GraphQL API as one of the schemas to be merged and therefore it does not matter in which language the other services are built with [17].

In conclusion, GraphQL schema stitching can be a very powerful method for building automatically scaling APIs without much extra effort apart from building the actual services themselves. By using schema stitching, the clients can still take advantage of all the GraphQL's features. There is nothing comparable to schema stitching for REST.

4.5 Speed of development

When thinking about the value what software can bring, it is clear that it is linked to the capability of the software to solve problems. The faster and cheaper these problems can be solved the better. Hence, the speed of development is a critical metric to consider before making decisions on what technology to use.

Both of these communication protocols are quite lightweight and the learning curve isn't considerably steeper in either one of these. REST is more lightweight and, according to my experience, can make the implementation require duplication of logic. It can also encourage to use not-so-optimal solutions to implement client specific features to compensate for the problems of under- or over-fetching. At an extreme, this can lead to building multiple very similar endpoints just for the sake of client specific requirements. On the other hand, GraphQL brings many of these features to the query language as first-class citizens [7].

GraphQL requires a bit more boilerplate code but it can pay off quickly. Once the types have been decided and declared, it is quite fast to implement the resolvers for each type. It is important to note here that each type has to implement its resolvers only once and the type can be used in as many places as needed. [20] This gives lots of flexibility to the developers as the framework takes care of much of the heavy lifting.

Documentation is hard to keep up to date which can lead to having deprecated documentation for a REST API. GraphQL is, for the most part, self-documenting meaning that anyone can use the introspection types to explore the capabilities the API provides [7]. These are always up-to-date as the introspection types come from the live API. It can be a large boost in productivity when the developers are always able to see the correct information about other services' APIs.

GraphQL provides a great framework for building high-quality APIs fast. In addition to making the development of the actual API fast, it can make it easier for the API's users to explore the API and build clients for it. REST is simple, maybe even a bit too simple. It doesn't give many tools for building APIs that need minimal boilerplate code but rather leaves that to the developers.

5 CONCLUSION

We have discussed microservices in general, introduced some of the most widely used communication protocols, and introduced GraphQL quite extensively. In addition, we have compared REST and GraphQL to each other specifically in the microservice context and evaluated them based on a few of the most important criteria.

GraphQL has become popular query language with a wide range of use cases and it has an active community of developers building libraries and other projects around it. It has solved or improved some of the REST's problems quite significantly. One of the areas GraphQL is better than REST is, undoubtedly, flexibility and data consumption. GraphQL's unique way of querying data enables the clients to fetch just the right amount of data saving on the overall amount of traffic sent over the network. This not only improves latency but can decrease the number of round-trips required for the user interface to visualize the data to the user.

REST is still a strong communication protocol and is most likely not going away in the near future. It is very easy to use as it is a lightweight set of rules the API implementation has to follow. Its strengths, in comparison to GraphQL, are performance and technology maturity. As REST does not require extra libraries in order to be used, it is very performant. Further, REST can be used with any web server framework making it one of the most widely supported protocols available.

Ultimately, the decision of whether to use REST or GraphQL depends on the use case. If performance is absolutely crucial, then maybe REST, or even something else, might be the way to go. On the other hand, if data consumption, scalability, and speed of development are important, GraphQL might be something to consider. GraphQL most likely is not going to replace REST in the near future but rather live alongside REST. It offers a more flexible alternative with interesting benefits and a thriving ecosystem of third-party libraries and projects to take its capabilities even further.

In the future, it would be interesting to extend this thesis and explore the different implementations of GraphQL and compare their performance and other attributes. As mentioned earlier, GraphQL relies heavily on the implementation. Hence, it would be beneficial to compare them against each other as it would bring clarity to the current maturity of the GraphQL ecosystem and help with the decision of the protocol to use.

In general, this thesis answers to the presented research problem quite extensively and well. However, communication protocols are a large topic and would require more work

to do comprehensive and more in-depth research. Building an example application using GraphQL would have been very beneficial and it is most likely the single one thing I would do differently. However, the scope of this thesis wasn't to do that and taking it into account, I'm happy with the result and extensiveness of this thesis. Communication protocols are an important aspect to consider and it can make a big difference to the resulting application and even business goals. There weren't many similar papers about microservices and GraphQL available, making this a quite unique and significant thesis.

REFERENCES

[1] T. Mauro. *Adopting Microservices at Netflix: Lessons for Architectural Design*. Nginx. 2015. URL: https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/ (visited on 03/21/2019).

[2] S. Newman. *Building microservices*. O'Reilly, 2015, 282.

[3] S. Tilkov. A Brief Introduction to REST. In: InfoQ, 2007, 11. URL: http://espinosa-oviedo.com/web-programming/wp-content/uploads/sites/5/2014/02/A-Brief-Introduction-to-REST.pdf (visited on 02/13/2019).

[4] *Protocol Buffers*. Google. URL: https://developers.google.com/protocol-buffers/ (visited on 02/13/2019).

[5] M. Slee, A. Agarwal, and M. Kwiatkowski. Thrift: Scalable Cross-Language Services Implementation. In: Facebook, 2007, 8. URL: https://users.cs.jmu.edu/bernstdh/Web/CS462/thrift-20070401.pdf (visited on 04/05/2019).

[6] P. Johansson. Efficient Communication With Microservices. In: Umeå University, 2017, 45. URL: http://www8.cs.umu.se/education/examina/Rapporter/PetterJohansson2017.pdf (visited on 01/19/2019).

[7] *GraphQL specification*. Facebook. 2018. URL: https://graphql.github.io/graphql-spec/June2018/ (visited on 02/13/2019).

[8] *GitHub GraphQL API documentation*. GitHub. URL: https://developer.github.com/v4/ (visited on 04/06/2019).

[9] A. Messina, R. Rizzo, P. Storniolo, M. Tripiciano, and A. Urso. The Database-is-the-Service Pattern for Microservice Architectures. In: Information Technology in Bio- and Medical Informatics, 2016, 11. URL: https://link.springer.com/chapter/10.1007/978-3-319-43949-5_18 (visited on 01/19/2019).

[10] N. Dragoni, I. Lanese, S. T. Larsen, M. Mazzara, R. Mustafin, and L. Safina. Microservices: How To Make Your Application Scale. In: Perspectives of System Informatics, 2018, 10. URL: https://link.springer.com/chapter/10.1007/978-3-319-74313-4_8 (visited on 04/06/2019).

[11] *Docker documentation*. Docker. URL: https://docs.docker.com/ (visited on 03/17/2019).

[12] *Kubernetes documentation*. Kubernetes. URL: https://kubernetes.io/docs/home/ (visited on 03/17/2019).

[13] *HTTP request methods*. Mozilla. 2019. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods (visited on 03/28/2019).

[14] A. Sumaray and S. K. Makki. A Comparison of Data Serialization Formats For Optimal Efficiency on a Mobile Platform. In: Proceedings of the 6th International Conference on Ubiquitous Information Management and Communication, 2012, 6. URL: https://dl.acm.org/citation.cfm?id=2184810 (visited on 04/06/2019).

[15] *Introduction to GraphQL*. Facebook. URL: https://graphql.github.io/learn/ (visited on 03/16/2019).

[16] L. D. Paulson. Developers Shift to Dynamic Programming Languages. In: Computer, 2007, 4. URL: https://ieeexplore.ieee.org/abstract/document/4085614 (visited on 04/10/2019).

[17] *Apollo schema stitching*. Meteor Development Group Inc. URL: https://www.apollographql.com/docs/graphql-tools/schema-stitching.html (visited on 03/17/2019).

[18] *GraphQL reference implementation*. Facebook. URL: https://github.com/graphql/graphql-js (visited on 04/06/2019).

[19] C. T. Lin. *awesome-graphql*. URL: https://github.com/chentsulin/awesome-graphql (visited on 04/06/2019).

[20] *Apollo documentation*. Meteor Development Group Inc. URL: https://www.apollographql.com/docs/ (visited on 03/17/2019).