Tampere University

Ossi Puustinen

# GRAPHQL FOR BUILDING MICRO-SERVICES

Faculty of Information Technology and Communication Sciences
M. Sc. Thesis
April 2020

# ABSTRACT

Ossi Puustinen: GraphQL for building microservices
M.Sc. Thesis
Tampere University
Master's Degree Programme in Computer Science
April 2020

---

Most of the current day web applications follow a strict decoupling of the client and the backend. While REST has become the industry standard for writing APIs, it has its shortcomings - most notably data under- and over-fetching. GraphQL has been hyped as the successor of REST as the new standard for building Web based APIs. GraphQL offers clients a way to describe precisely the data and interactions they are interested in. Its big-name adopters include Shopify, Twitter, Atlassian and PayPal.

This thesis is a case study of a GraphQL microservice, which combines data from multiple different sources and offers it in a single cohesive data model. The objective of the study is to gather information on building GraphQL based microservices, best practices on resolving GraphQL queries and maintainability of a GraphQL server.

The results showed that GraphQL suited well for building microservices, but it is highly oriented towards serving client applications. There are some caveats on resolving hierarchical queries with GraphQL, which are well known and documented and can be avoided. GraphQL provides some benefits in the terms of project maintainability but might require more code to be written when compared to REST.

Key words and terms: GraphQL, REST, APIs, microservices, software development

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

# Contents

## Abbreviations and Acronyms

| | |
|---|---|
| AJAX | Asynchronous JavaScript And XML |
| API | Application Programming Interface |
| DOM | Document Object Model |
| HTML | Hypertext Markup Language |
| HTTP | Hypertext Transfer Protocol |
| JSON | JavaScript Object Notation |
| SPA | Single Page Application |
| PWA | Progressive Web Application |
| GraphQL | Graph Query Language |
| REST | Representational State Transfer |
| URI | Uniform Resource Identifier |
| URL | Uniform Resource Locator |
| XML | Extensible Markup Language |

# 1 Introduction

## 1.1 Problem statement and research questions

Most of the current day web applications follow a strict decoupling of the client and the backend. Architectural styles like REST (Representational State Transfer) have enabled the possibility of building general purpose APIs (Application Programming Interface) serving multiple different clients from browser to native mobile applications. While REST has become the industry standard for writing APIs, it has its shortcomings. Most notably data under- and over-fetching, where the clients have to do multiple subsequent requests or end up fetching excess data.

As web clients are getting more complex, improvements in simplicity of the data retrieval and handling are also becoming more and more important. GraphQL offers clients a way to precisely describe the data they need and fetch it through a single endpoint. Depending on the data model, this comes with the advantage of less data transferred and less requests made. As GraphQL is not tied to any specific data source, even existing REST APIs can be used as a data source for a GraphQL server. The aim of this thesis is to provide answers to the following questions through a case study of a GraphQL server built for a customer company:

- How does GraphQL affect the maintainability of a microservice?
- What are best practices for completing hierarchical data queries with a GraphQL based API?
- How suitable is GraphQL for developing microservices?

The research questions are answered by reflecting on the experiences gained during the development of the case study application. Literature presented in the background chapter and the methods chapter will be used as a basis for this discussion.

## 1.2 Case study details

Our customer wanted to build an application to help their partner stores manage various sales campaigns, promotions, and products. The end users will use the information displayed on the application to make decisions on what products to order in preparation for future sales campaigns. In addition to filtering and viewing the campaigns and products, the users are also able to order more products and view incoming deliveries and past

orders. They need this application to be accessible through tablet devices and to be available throughout Finland.

The data that was to be displayed by the application already existed, but across various different systems mostly accessible through REST APIs or other external sources. While the user interface could be quite easily built with modern SPA (Single Page Application) framework like React, the backend was to be split into an intermediary backend handling client specific tasks and a microservice which would take care of the campaign data handling. Building a general purpose microservice for querying campaign data also came with the advantage of offering it to other yet to be built services as well. In this case study we will be focusing on this microservice.

All data served by the microservice is related to sales campaigns, stores and products. The data sources are not publicly available and are only meant for internal use of other services. While the data is all related to the same domain and can be linked together to form hierarchical data structure, it does not follow a cohesive data model. This issue was mostly related to a monolithic SAP-system that had evolved over time and a REST API that didn't follow the same naming conventions as the SAP-system. Some of the data was going to be received as raw SAP data, and some was going to be fetched in pre-parsed human friendly form. The purpose of the microservice is to retrieve the data from the different sources, parse it into a client friendly data model and offer a way to query it.

Due to potential reductions in transferred data and the general-purpose nature of the microservice, GraphQL was chosen over traditional REST API. However, benefits could be also seen in maintainability of the application. This project would be also used as a place for gathering experiences on using GraphQL in production.

## 1.3 Outline

In this chapter we have offered a short introduction to the topic area and the research questions. In the second chapter we will go through necessary background information on the subject area and web architecture. Most importantly detailed descriptions for REST and GraphQL are provided. We end the second chapter with a look into related studies in the subject area of GraphQL. In Chapter 3 we will review the research questions and discuss how the evaluation of the case study application will be conducted. In Chapter 4 a general description of the case study application and its architecture will be given. In Chapter 5 we will go through the results and finally summarize our learnings in Chapter 6.

# 2 Background

During the past decade the World Wide Web has moved from just serving server-side rendered HTML documents to a more dynamic data-driven form. While in the past browsers were used for viewing web pages, they are now used as an environment for running client applications built with SPA frameworks like Angular and React. These frameworks manipulate the DOM (Document Object Model) according to the user's actions and the data received through HTTP (HyperText Transfer Protocol) requests. User clicking for example a button on the page does not lead into the whole page loading again, but only the necessary parts of the page being updated. More recently service workers enabling offline usage of web pages have reinforced the more application-like user experience - along with other improvements brought by PWAs (Progressive Web Application).

For the past decade REST APIs paired with SPA clients have been the de facto architectural style for building web-based services. The REST APIs are built with generality in mind and can be used by multiple different types of clients, for example native mobile applications and web pages. This generality also comes with the price of not serving the exact needs of a certain client, which may lead to multiple subsequent HTTP requests and excess data retrieval. The amount of excess data transferred and multiple requests done might not cause problems in low-latency environments, but issues my rise with mobile device battery drain and in areas with less than ideal connectivity.

GraphQL takes a similar position in the web architecture as REST, but instead of offering resources behind URIs, it expects clients to define the fields they are interested in. Returning only the exact fields needed minimizes the amount of data transferred between a client and the server. This also enables the server to offer a wide selection of data without worrying excess data transferred, as the responsibility for selecting the fields is left for the clients. REST, GraphQL and web architecture will be discussed in more detail in the following sections.

## 2.1 Modern web, data fetching and client-server model

Before diving into other topics, we should first have a brief look at modern web applications and APIs. At the core of serving data through APIs over the network is the client-server model, which separates the user interface concerns from the data storage concerns.

The client-server model is an architectural model where the application is split into two processes, the client which requests services and the server which provides the requested services [Jia and Zhou, 2005]. After the server has completed a request it sends a response to the client, usually containing a confirmation of a successful completion

along with a result, or an error message. Multiple instances of a server can exist, and a single server can serve multiple clients.

With traditional web pages the entire page is fetched from the server each time the page needs to be updated. SPA is a web application that loads only a single web document and then updates it through JavaScript APIs such as XMLHttpRequest and Fetch [SPA, 2020]. These technologies enable the browser to retrieve data from servers without refreshing the page.

AJAX (Asynchronous JavaScript and XML) encapsulates these and various other technologies like HTML, JSON (JavaScript Object Notation) and DOM in order to offer a complete feature set for making small incremental updates to the web page without having to do a complete reload of the page [AJAX, 2020]. While one could start building SPAs from scratch with just AJAX, various frameworks for building them exist. Most notable of these being React, Angular and Vue.

## 2.2 Microservice architecture and SOA

In service-oriented architecture (SOA) the application is split into self-contained services that provide a collection capabilities [Thomas, 2016]. Each service acts as an independent application, but they can be composed together in order to automate a specific task or process. A lot of emphasis is put on the reusability of the services. Other key principles of SOA include standardized service contracts, loose coupling, abstraction, statelessness, discoverability.

As the services in SOA are developed as independent applications, they have a great deal of autonomy on their implementation and technology choices [Thomas, 2016]. In order to enable smooth interoperability, the services should follow a standardized service contract, which defines the purpose and the capabilities the service provides. A collection of services handling a meaningful segment of an enterprise is grouped into a service inventory, where all services follow a common standardized service contract.

Microservices can be viewed as a part of service-oriented architecture, a variant of it, or they can be defined completely separately from SOA [Thomas, 2016; Nadareishvili et al., 2016; Fowler and Lewis, 2014]. When defined independently, microservice architectural style splits the application into small independent services, each running as a separate application [Fowler and Lewis, 2014]. The services are built around business capabilities and maintain a high level of autonomy with minimal amount of central management. As the services are independent applications, they can also be scaled independently and built with a technology that best fits the use case. The key characteristics of a microservice can be summarized as follows: limited size and context, autonomous development and independent deployment and decentralization [Nadareishvili et al., 2016].

As a part of service-oriented architecture microservices are classified as one of the non-agnostic service models, where the service is built for a very specific task and it is only reusable in the business context it was developed for [Thomas, 2016]. As a variant of service-oriented architecture, the microservice architecture could be described as a more granular, loosely coupled architecture with bounded context and less focus on reusability [Nadareishvili et al., 2016]. While neither of the descriptions is wrong, it can be desirable to approach microservices without the context of service-oriented architecture in order to avoid confusion and misunderstandings.

While the exact definition of a microservice might vary, their characteristics are best highlighted by comparing them to a monolithic application, which puts all its functionality into a single process. The monolith will be tied to using a single technology stack and a focused development team. Although it might be designed in a modular manner, it will still be more burdened by dependencies between modules. With monolithic applications the whole monolith has to be replicated in order to scale up due to the stress put on one of its modules. The increased size of the code base and the amount of changes targeting it will make also the release and development cycle more burdened.

## 2.3 REST

REST (Representational State Transfer) is an architectural style for building decentralized network-based applications. It was first defined by Fielding in 2000 with a set of constraints that the implementer should adhere to: strong client-server separation, stateless interaction, caching, uniform interface and hierarchical layering of the system [Fielding, 2000]. While REST has evolved overtime, the changes consist more of formulation and articulation improvements rather than drastic changes to the design [Fielding et al., 2017].

In practice a REST based service is serving resources identified by URIs. REST does not set any constraints on the contents of the resources themselves, as it is only focused on the way these resources are accessed. The resources are usually managed through HTTP requests where the HTTP verb matches action that is to be done on the resource. For example, GET will fetch the resource and POST will create a new resource. After completing the request, the server will respond with appropriate HTTP status code and a possible response body. While HTTP is used here as an example, REST is not limited to it, and other protocols can be used in its place.

For example, in order to retrieve information on a user with id 4, the client sends a HTTP GET request to the API endpoint **/api/user/id/4.** On a successful completion the HTTP response sent by the server will contain the user information in its body - as demonstrated in Figure 1. If the user with id 4 does not exist, the server should return a HTTP 404 instead.

```
GET /api/user/id/4

{
  "id": 4,
  "name": "Mark Zuckerberg"
  "department": 1
}
```

Figure 1. Example REST request and response.

As seen in Figure 1, when handling hierarchical data, the sub-resource is often returned just as an identifier. This identifier can be used to retrieve the actual resource from another URI if desired. As a result, the client might have to execute multiple subsequent requests in order to retrieve all desired data. The deeper the data hierarchy the more requests have to be made. This issue is called data under-fetching [Mukhiya et al., 2019].

Similar to data under-fetching, clients may also end up fetching more data than they need. Even if the user of a REST based service would be interested in only a single field in a resource, the server will return the whole resource identified by the URI. This issue goes hand in hand with data under-fetching, as the whole resource for each level in the hierarchy needs to be fetched in order to retrieve the identifier of a sub resource. For example, in order to fetch the department name of the user, first the user resource needs to be fetched in order to acquire the id of the department. Issues like data under and over-fetching can be fixed by creating optimized endpoints for specific use cases, but this results in more maintenance work and overlapping. In reality REST is not always strictly followed and strict enforcing of REST is not really possible [Fielding et al., 2017; Rodríguez et al., 2016]. Implementers can freely diverge from it, but this can be damaging for the overall architecture and maintainability of the service.

In conclusion resource centricity and hierarchically layered system architecture of REST help in splitting the application into modular parts, making it easier to maintain and manage. Handling of each resource will remain separate from others and easy to reason about. The human readable uniform language composed of HTTP verbs and resource URIs is simple and easy to understand. Even when the application grows, the semantics of the API will remain the same and each endpoint will clearly express what resource it is referencing to. But as mentioned before the generality comes with the price of forcing the clients to comply with the API they are accessing.

## 2.4 GraphQL

GraphQL is a query language created by Facebook in 2012 and published as an open standard in 2015. Its key characteristics consist of hierarchical client specified queries, product centricity, strong typing and introspection [GraphQL, 2018]. GraphQL offers clients a flexible system for precisely describing the data and interactions they are interested in. It is important to note that GraphQL itself is not an implementation. It is a specification on a language and its characteristics. GraphQL is agnostic on the programming language and the data sources used, but the reference implementation was written in JavaScript.

GraphQL approach to building APIs is data centric [GraphQL, 2018]. With GraphQL the developers first define their data model and all possible actions that can be done to it in a schema. The clients are free to select only the fields they are interested in as long as the query matches the schema defined by the server. For example, if we want to fetch the name and department of a specific user, we can select the desired fields in the query "user". Only the requested fields will be supplied in the response. This has been demonstrated in Figure 2.

```
query {
  user(id: 4) {
    id
    name
    department {
       name
    }
  }
}

{
  "id": 4,
  "name": "Mark Zuckerberg",
  "department": {
    "name": "HQ"
  }
}
```

Figure 2. Example query and response

As mentioned before, the data available and all possible operations that can applied to it are described in the schema. Figure 3 demonstrates an example schema with object types for user and department, and queries for fetching them. Each field in an object may contain another object type, a scalar type or an array of either. Additionally, one can define interfaces, union types and input types. While interfaces and unions are used for abstraction, input types can be used for passing complex objects as operation parameters. The schema is strongly typed, and introspection enables the schema itself to be queried. This can be used by third party tools to for example generate documentation or to provide diagnostics tools [GraphQL, 2020]. Strong typing also allows the server to make some guarantees on the response as well.

```
type Query{
  user(id: Int!): User
  users: [User]
  department(id: Int!): Department
  departments: [Department]
}

type User{
  id: Int
  name: String
  department: Department
}

type Department{
  id: Int
  name: String
  location: String
  manager: User
}
```

Figure 3. Example schema with two object types and four possible queries.

GraphQL supports three types of operations: queries, mutations and subscriptions [GraphQL, 2018]. Each operation is represented by an optional operation name and a selection set. Queries are used for fetching data. Subscriptions are similar to queries, but

they are event based and are usually implemented through web sockets. Mutations are used for modifying existing data or creating new entries.

GraphQL queries are completed with resolver functions [GraphQL, 2020]. A resolver function contains the logic for fetching a specific field or object. As seen in Figure 4, the resolver functions take four arguments: parent, args, context and info. The parent will contain the previous object. In the case of the query in Figure 2 the parent passed to the department resolver would contain the already resolved user fields. Args contains the query arguments. Context contains contextual information, like information on access rights and data sources. Info holds information relevant to the current query and the schema.

```
Query: {
  User: async (parent, args, context, info) => {
    return context.db.loadUser(args.id);
  }
}
```

Figure 4. An example resolver.

If the resolver of a field produces a scalar value, the execution is complete, but in case of an object type the query will contain another selection of fields which will apply that object. In reality trivial single field resolvers do not really exist and most of the GraphQL libraries will let the developers omit them [GraphQL, 2020]. If a resolver isn't provided for a field, the property of the same name is read and returned from the parent object. GraphQL does not specify how the resolver functions actually handle the data resolution, but usually the resolver functions remain slim and call reusable service layer functions to resolve the data.

The requests are made with either HTTP POST or GET [GraphQL, 2018]. As GET requests don't support message bodies, the query has to be supplied using the query params. With POST the queries can be supplied in the request body. Which one to use is entirely up to the developer, but one might favour POST over GET due to better readability, although this might conflict with semantics implied by the HTTP verb. Responses sent by the GraphQL server will contain a JSON body with either requested data or a detailed error message.

GraphQL was primarily built with requirements of views and clients in mind [GraphQL, 2018]. The clients have full freedom to define which data fields they are interested in and omit all excess values, which helps in reducing the total amount of data transferred. While this feature is a result of client centricity, the ability to freely select fields from a cohesive schema is also beneficial for backend services as it reduces the need for writing custom endpoints for specific use cases.

While GraphQL service could be versioned like any other API, it is strongly advised against [GraphQL, 2020]. Traditionally versioning is done in order to signify breaking changes. As GraphQL returns only the data fields that are explicitly requested, new fields can be added without side effects. If existing fields need to be removed from the schema, they can be marked for deprecation before making the actual breaking change. While most of the maintainability of a service comes from quality of implementation and adhering to the best practices of chosen architectural style, lack of versioning provides ease of maintenance for GraphQL services over for example REST based ones.

During the time of writing GraphQL has been around for five years and it's reaching a relative maturity. While the reference implementation of GraphQL was written in JavaScript, open source libraries of it exist for almost every notable language used in web development - from Java to Scala and Clojure [GraphQL, 2020]. Like with all open source libraries, the maturity of these projects may vary, but the availability of them enables viable implementation of a GraphQL server with almost any technology stack.

## 2.5 GraphQL alternatives

While GraphQL has gained a lot of attention since its release, Netflix had a similar project called Falcor in 2015 [Netflix, 2015]. Falcor had a similar idea of letting clients define the fields they are interested in, but it didn't gain as much traction as GraphQL. There are also differences between the two. While GraphQL itself is a specification with various implementations, Falcor is a ready middleware implemented in JavaScript, with focus on optimizing communication between layers of the application [Falcor, 2020]. Falcor is also not a query language and does not support open ended queries like GraphQL does. While Netflix still offers Falcor for download and the GitHub repository is still active, Netflix themselves have started using GraphQL in some projects [Netflix, 2018].

Academic resources for Falcor are extremely sparse and as Falcor seems to be losing its relevance this is unlikely to change. However, Cederlund's study on declarative data fetching from 2016 offers some insight on how it compares to GraphQL [Cederlund, 2016]. The study concluded Falcor to offer inferior performance compared to GraphQL, but it should be noted that client-side libraries for handling the queries were included in the test scenario.

There has also been some community effort in improving upon GraphQL. Deepr is a JavaScript library that offers similar declarative data fetching as GraphQL, but with some design differences [Deepr, 2020]. The developers of Deepr are opting for a slimmer implementation without a built-in type system, new language or advanced features like subscriptions and parallel queries. While features like typing and subscriptions are seen as important, developers of Deepr believe they shouldn't be part of the core implementation. There are also some differences in how collections are handled and other minor alterations, but overall GraphQL and Deepr are still more similar and different.

## 2.6 Related Work

While GraphQL is reaching relative maturity as a technology, academic resources related to it are still quite sparse. It is a common pattern in related studies to draw a comparison between REST and GraphQL, usually through a migration process of an existing REST based service. This approach makes sense as REST has been the de facto architectural style for the past decade and GraphQL has been hyped as the replacer of REST. Nevertheless, some studies focusing purely on GraphQL could be found.

### 2.6.1 GraphQL and graph theory

Knowledge in graph theory is not required for using or understanding GraphQL, but the subjects are still somewhat related as the GraphQL schema essentially contains the application's business model as a graph [GraphQL, 2020]. In simple terms graphs are collections of nodes connected by relationships. Graphs can be used for modelling various scenarios from systems of roads to medical history of populations. For a hands-on approach "Graph databases: new opportunities for connected data" written by Robinson et al. can be used [Robinson et al., 2015]. While the book's main focus is on graph databases, it offers simple introduction to graphs and modelling data with them.

For a graph theory focused resource, Mihalcea Rada and Radev Dragomir cover the basics of graph theory and terminology in their book "Graph-based Natural Language Processing and Information Retrieval" [Mihalcea and Radev, 2011]. But as brought forward before, deep understanding of graph theory is not really relevant in terms of using GraphQL so one should regard the book as a curiosity read and not something essential.

On the subject of graph theory and GraphQL, Hartig et al. did a relevant in-depth study of GraphQL in 2017 [Hartig and Jorge, 2017]. The study covers formalization of GraphQL schema and query syntax, semantics and an analysis of complexity. The approach is quite low level and mathematical and thus the paper would be of interest for people looking for an in-depth theoretical look at the language itself.

### 2.6.2 Migration from REST to GraphQL

Migrating existing REST based services has been a common area of study in academic resources related to GraphQL. Vogel et al. wrote a case study of a migration process of a home IoT service from REST to GraphQL in 2018 [Vogel et al., 2018]. The aim of the migration process was to offer an alternative way for fetching data from the server and to gather experiences from the migration process and GraphQL itself. As the data they wanted to serve remained essentially the same, they were able to derive the GraphQL schema straight from the original REST API, which also enabled Vogel et al. to use the same service-layer as the original REST API. In addition to describing the architecture and migration process, Vogel et al. also did a performance analysis between the two APIs.

While the schema could be derived from the original REST API, Vogel et al. made important notes about the conceptual differences between REST and GraphQL. GraphQL and REST both follow strict client-server separation and statelessness, but the data-centric approach opposed to resource-centric approach is quite different. Substitution of a former REST resource is inevitably going to break the design principles of the original architecture, but if the aim is to incrementally replace the original service, this should come as expected. In the case study conducted by Vogel et al. both the REST and GraphQL API were left running in parallel on the same server, leaving the original REST service completely intact.

Wittern et al. conducted a case study where they wrote a tool for generating GraphQL wrappers for any REST-like API with an Open API specification (OAS) [Wittern et al., 2018]. The proof of concept tool worked, but its success rate depended greatly on the quality of OAS specification. Additionally, nested queries worked only if the OAS had link definitions between object types. It is important to note that these wrappers use the original REST API to complete the requests, so the approach is not nearly as efficient as directly accessing the original service layer from resolvers.

While migration of REST based systems has been a common topic in GraphQL related studies, one should consider if GraphQL will actually benefit the project before embarking on to a migration project. Also as covered by Vogel et al, a partial migration of a service where GraphQL endpoint is offered alongside the traditional REST API is also a completely valid option [Vogel et al., 2018].

### 2.6.3 Performance and security

The performance comparison between GraphQL and REST run by Vogel et al. measured single round trip performance and performance difference in situations where REST required multiple round trips to be made [Vogel et al., 2018]. Also, the effects of selecting only a single field in the GraphQL query were measured. The results showed that GraphQL and REST had similar performance in query execution for atomic resources. In

cases where multiple REST requests had to be made, GraphQL performed noticeably better, but if an optimized REST endpoint which didn't require multiple round trips was used, the performance was again equivalent. Choosing only one field from the dataset didn't have a noticeable performance hit.

The results bring forth the fact that there isn't a meaningful performance gain when GraphQL is compared to a well optimized REST server, but with GraphQL there is no need to write endpoints for special use cases. It should also be noted that these tests were run in an environment with ideal connectivity. This excludes possible performance improvements gained from reduced data transfer sizes. But as GraphQL requests tend to be larger in size than REST requests, the data reductions in the server responses need to be large enough for GraphQL to provide a meaningful improvement, as noted by Cederlund in 2016 [Cederlund, 2016].

In practice writing specialized REST API endpoints might not be possible due to higher maintenance costs and increased workload. Mukhiya et al. did a study in migrating a healthcare related service to GraphQL in 2019 [Mukhiya et al., 2019]. They found GraphQL to nearly halve the request times and the amount of data transferred in some scenarios. The results highlight the issues related to the data under fetching and improved performance GraphQL can offer in cases where writing optimized REST endpoints is not an option.

The issues with data under- and over-fetching were also brought forward by Bryant in 2017 [Bryant, 2017]. While Bryant didn't conduct a performance analysis, he described the issues through use cases of a GraphQL service built for retrieving archival metadata. The benefits of precise single-request data fetching offered by GraphQL become apparent when accessing 10-level deep contextual data hierarchies. With the traditional REST approach, the clients have to go through nine requests in order to reach the desired data - retrieving excess information along the way. While introspection offered by GraphQL seems to be less talked about, its benefits for third-party tooling and more precise error messages are discussed in Bryants paper.

As we have covered so far, GraphQL offers improvements in some scenarios, but it also comes with some shortcomings. Both Mukhiya et al. and Vogel et al. brought up GraphQL's vulnerability to denial of service attacks arising from the clients' ability to freely define overly complex queries [Mukhiya et al., 2019; Vogel et al., 2018]. Vogel et al. presented limiting the query size and pre-defining all permitted queries and storing them on the server as naive solutions to the problem. While these are easy to implement, the query size alone might not be enough to identify harmful requests and limiting usage of the service to predefined queries would severely handicap the services ability to serve different types of clients. Resolver restriction and static analysis of queries prior execu-

tion were presented as more sophisticated solutions, but they take more time to implement. Hence these security features could be best integrated into libraries implementing GraphQL.

# 3 Methods

This thesis covers a case study, where GraphQL was used in a customer project. The aim of the thesis is to evaluate suitability of the GraphQL for building microservices, gather knowledge on best practices on completing hierarchical GraphQL queries and evaluate the maintainability of a GraphQL server. In this chapter we will go through how these research questions are answered.

In simplified terms maintainability describes the ability of the software to undergo changes. Various models for measuring software maintainability have been proposed from traditional qualitative metrics to statistical code analysis and even artificial neural network -based models [Shafiabady et al., 2016]. While there are tools and models for quantitative maintainability analysis of a codebase, this type of analysis would not be beneficial in terms of this case study. GraphQL exists only on a specific layer of the application and there are various other areas affecting the general maintainability of the codebase. As a result, qualitative metrics will be used as a basis for the maintainability evaluation.

Traditional qualitative approaches for measuring software maintainability were originally defined as a part of software quality models. The McCall model, developed in 1976, defined three major perspectives for quality measurement: product revision, product transition, and product operations [Al-Badareen et al., 2011]. Product revision describes products ability to undergo changes, transition describes its adaptability to new environments and operations describes its operation characteristics. In the McCall's model maintainability is classified as only one of the parts of product revision. The criteria for measuring maintainability are as follows: consistency, simplicity, conciseness, modularity and self-descriptiveness [McCall et al., 1977].

The criteria defined by McCall will be used as the basis for the discussion regarding maintainability. New and existing queries should be easy to add and modify, as well as the schema defining the data on offer. GraphQL connects to the service and data access layer of the application through resolver functions. The modularity of these resolvers and their effect on the service layer will be discussed in the evaluation. As much of the maintainability will rise from the overall quality of the software, as well as architectural styles followed, how well GraphQL suits principles like modularity and generality will be taken into consideration.

The best practices for completing hierarchical queries will be discussed through findings made during the development of the microservice. The focus of this question is on finding the red flags and patterns that should be avoided and on evaluating the design decisions made when building the microservice. The users of the service should be able to query the schema without worrying about the actual source of the data. This comes

with the requirement of query execution time remaining plausible regardless of query or result size.

Suitability of GraphQL for building microservices will be evaluated by reflecting on the definition of a microservice found in the second chapter. In addition, the application's suitability for the primary use case will be discussed. While the case study application is built with primarily one client in mind, its usability for other services will be discussed. In this regard REST will be used as a point of comparison. Although GraphQL is agnostic on the actual source of the data, its capability of handling multiple different data sources will be discussed.

In conclusion quantitative analysis was found not to be beneficial for this case study. Quantitative evaluation would be more fitting in a scenario where two technologies are compared with each other. Instead of quantitative analysis, the research questions will be evaluated through discussing learnings made during the development of the case study application.

# 4 Implementation

## 4.1 Background

This case study is focusing on a microservice built as a part of a customer project. The case study application was still in early development at the time of writing, so the implementation described here does not represent a final version. The implementation described here is also streamlined from the actual one and some services and details were cut away as they were not relevant in terms of this study. Before diving into the details of the case study application we should have a look at the problem we are solving.

Our customer needs to pass information about upcoming sales campaigns, promotions and products to their partner retail stores in order for the shopkeepers to be able to prepare for them. In practice for the stores this means making supply orders prior to the start of the sales campaigns and updating the product prices when the campaign starts. Currently this is done by passing the relevant information in excel sheets, which is prone to mistakes and does not always offer the most complete and up to date information. It is also cumbersome for the shopkeepers.

The requirements set for the application stated that it had to be accessible through tablet devices throughout Finland. Creating a SPA client was a clear choice as no installation would be required and creating responsive applications accessible by any device was relatively easy. React was chosen as the SPA framework. Apollo GraphQL also provided a client side React library for handling client-side caching and GraphQL requests. The requirement of being accessible throughout Finland put some restrictions on data transfer, as some areas might have less than ideal connectivity.

The SPA client should have access to pre-parsed data as data handling can be performance intensive. The responsibility of the client is to display the upcoming and past campaigns, product information and order information. The data displayed can also be filtered, which is done at the client level for now, but this might change in the future. Client level caching is not implemented at the time being as the development is still in early stage.

The GraphQL microservice will be receiving data from multiple different external services. The data received is all related to the same domain and can be linked together to form a complete hierarchical data model, but in order to accomplish this some parsing had to be done. Most of the data is coming from event-based sources and is persisted to a local database and some additional information related to products is retrieved through an external REST API. The data retrieved from the REST API needs to be as up to date as possible. This prevents storing the data in a local database or implementing long term caching. These could have been desirable options due to performance reasons.

## 4.2 The GraphQL microservice

The GraphQL server was implemented using Apollo GraphQL, which is developed by the Meteor Development Group [Apollo, 2020]. In addition to the GraphQL implementation written in TypeScript, Apollo offers additional tooling and client-side libraries for making GraphQL requests. The most noteworthy of this additional tooling is the playground which enables easy testing of a running GraphQL server.

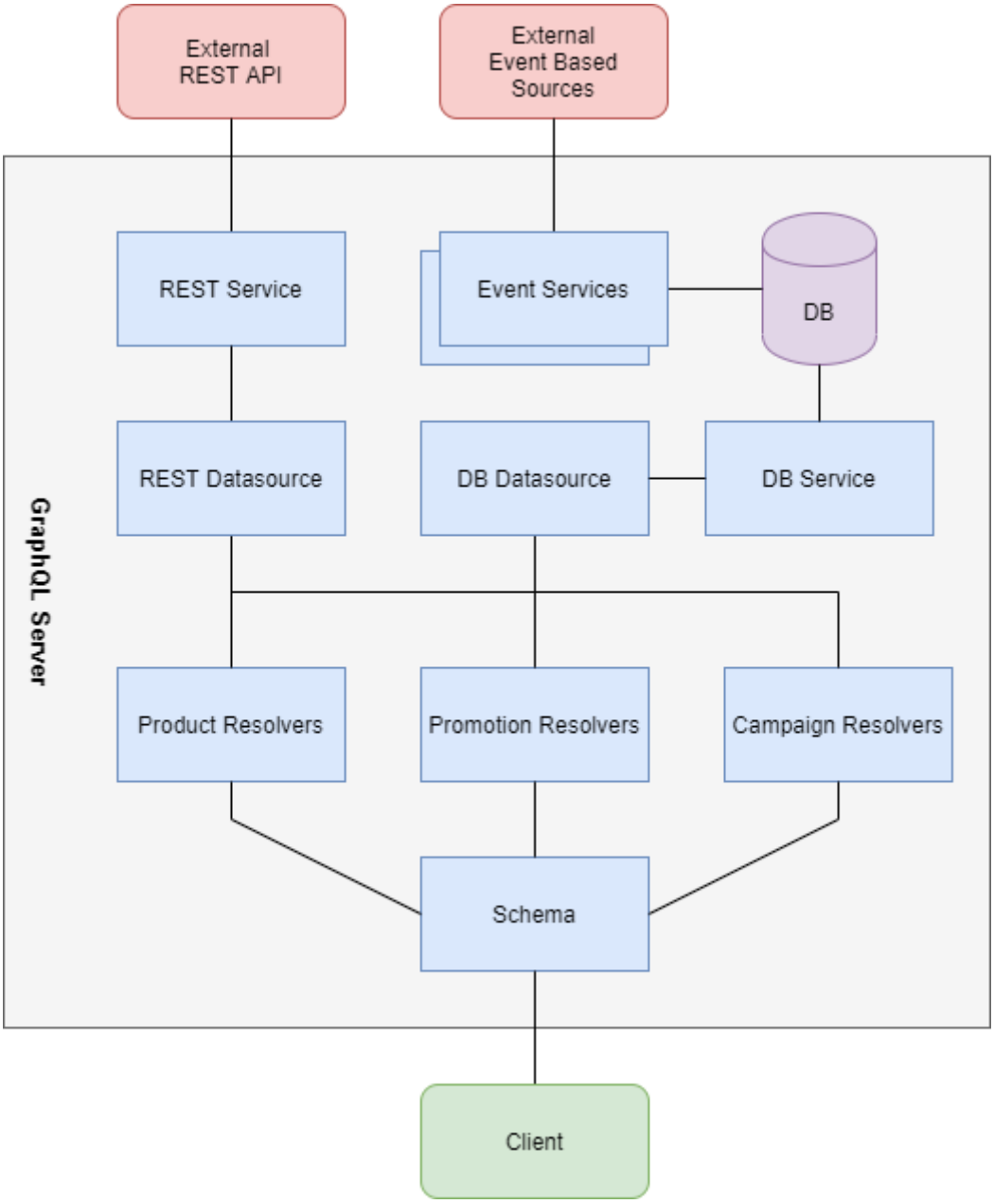


Figure 5. Outline of the architecture.

Figure 5 offers an overview of the case study application architecture. The grey square contains the GraphQL server. The lines between objects represent communication between parts of the case study application and external services. Inside the GraphQL microservice this means function calls and database queries. The communication between the microservice and the clients and external services takes place over HTTP.

The red squares represent external sources, from which the data is received from. The REST- and event-services handle communication with these external data sources. The data from the event-based services is persisted to the database. The DB-service is used for handling data retrieval from the database.

The REST- and DB-datasources encapsulate data fetching from the corresponding service with added support for caching and deduplication. Resolvers complete the queries by calling these datasources. All data offered by the microservice and the queries used for accessing it are described in the schema. The green client square represents the React clients described in the beginning of this chapter.

Most of the data resides in the local database that is populated through event-based services, but some additional information on products needs to be retrieved from an external REST API. When a client is querying data from the GraphQL server, only the schema will be exposed, and the client will remain completely unaware of the actual source of the data and how the queries are resolved.

The resolvers in Figure 5 are split into three groups by the data they are accessing. Each of these groups can contain multiple resolver functions. For example, in order to find a specific campaign and an array of campaigns different resolvers need to be written. Each resolver can technically have access to both the external REST API and the local database containing the data form event-based sources. In the case of the case study application only product information needs to be patched together from multiple sources.

```
type Campaign{
  id: Int
  ...
  promotions: [Promotion]
  products: [Products]
}

type Promotion{
  id: Int
  ...
  products: [String]
}

type Product{
  id: Int
  ...
}

type Query{
  campaigns(): [Campaign]
  campaign(id: Int): Campaign
  product(id: Int): Product
  ...
}
```

Figure 6. Outline of some of the object types and queries defined in the schema.

Figure 6 demonstrates a simplified version of the object types and queries offered by the GraphQL server. Three dots are used for marking n-number of omitted fields. In addition to campaign information, the campaign object contains information on the related promotions and products. This information is stored in the corresponding fields, each containing an array of objects defined in the schema. The query object defines only some of the campaign and product queries as an example.

```
query {
  campaign(id: 4) {
    id
    name
    promotions {
      name
      startdate
      enddate
    }
    products {
      id
      name
    }
  }
}
```

Figure 7. An example query.

Figure 7 demonstrates what a query targeting a specific campaign and related products and promotions would look like. As can be seen the objects defined in the schema form a hierarchy, where promotions and products are located under the campaign as child objects. The query takes a similar hierarchical structure. Clients are free to choose the fields they desire from the selection offered in the schema.

The queries are resolved using resolver functions, as seen in Figure 8. The resolver functions are usually slim, in some scenarios only calling a single service layer function to fetch the relevant data. The context passed in arguments contains information on query context, but most importantly it will provide access to datasources. In Apollo GraphQL the datasources are classes that encapsulate data fetching from a specific service with added support for caching, deduplication and error handling.

```
Campaign: async(parent, args, context) {
  return context.datasources.sql.getCampaign(args.id)
}
```

Figure 8. An example resolver.

Each field in a query is technically resolved using a separate resolver function, but in most cases a complete object is retrieved from the datasource. Apollo GraphQL allows the developers to omit the most trivial resolvers by defaulting to reading the property of the same name from the parent object if a resolver isn't provided. As a result, resolvers are written for each query and for each child object in the context of the parent.

It is possible to use a single resolver to fetch all data under for example campaigns along with all child promotions and products, but this results in data related to child objects being fetched from the source even when it isn't needed. Although the excess data does not end up in the final response, this is far from optimal as retrieving data from the source comes with a performance cost. Especially so if the source is external, for example a REST API.

While resolving child objects only when needed results in a more modular query resolution, it also causes an issue related to data under-fetching. Each resolver will make a round trip to the origin source of the data, creating as many calls as there are objects. For example, when resolving a list of campaigns, the campaigns list can be resolved in a single call, but the promotions resolver will be called as many times as there are campaigns. Creating this many calls causes needless stress to the origin source of the data.

In order to fetch all child objects in a single round trip batching needs to be used. In the case of GraphQL this is usually done with a dataloader. A dataloader batches multiple data fetches targeting the same model and datasource into a single call. In practice this means adding an extra layer between the resolver and the datasource. When a resolver needs to fetch data, it will pass the information required for completing the fetch to a loader. The loader will collect the information and fetch the desired data in a batch.

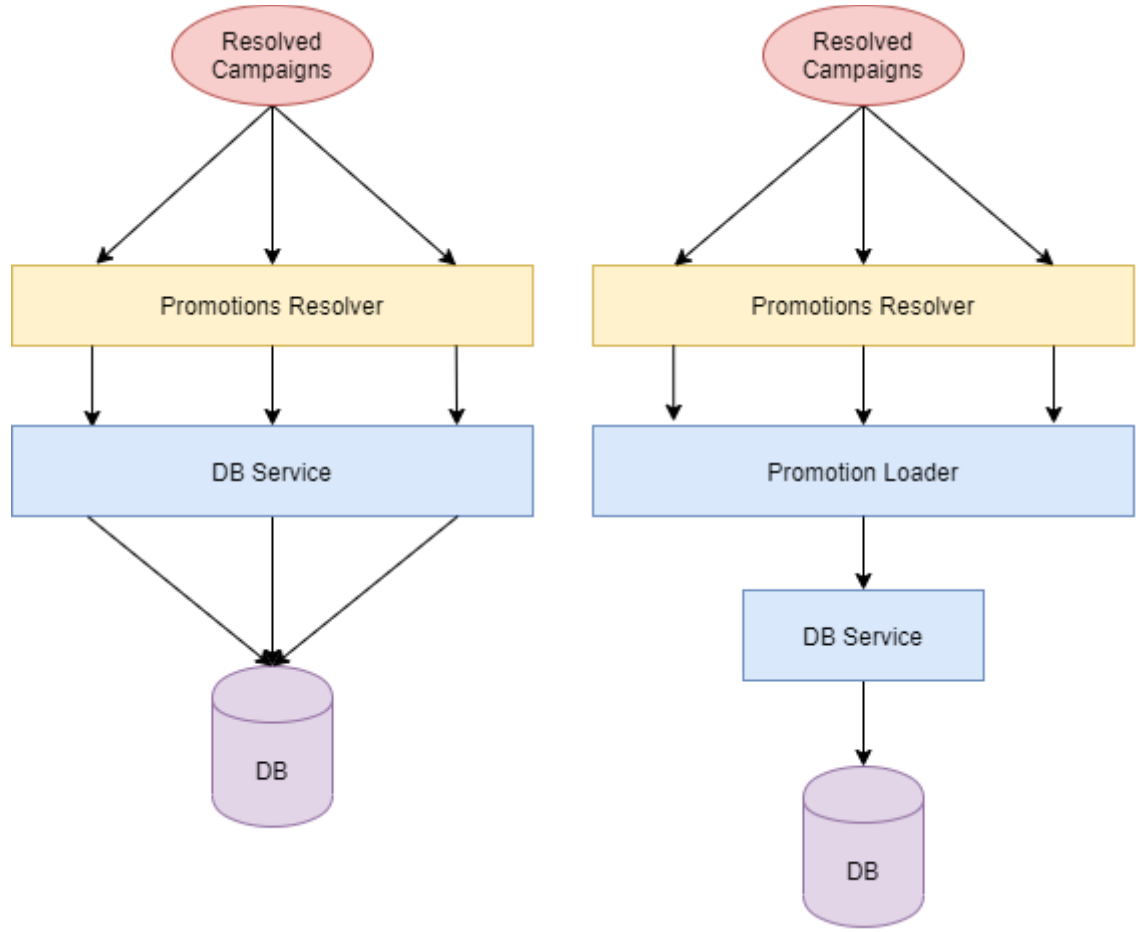


Figure 9. Resolving promotions related to campaigns without a dataloader (left) and with a dataloader (right).

Figure 9 demonstrates how promotions related to an already resolved list of campaigns would be fetched with and without a dataloader. The red circle represents a resolved list of campaigns, which results in three calls to the promotions resolver. So, in this example there are three campaigns with unknown amount of related promotions each. The arrows represent function calls and database requests.

Without a dataloader each set of promotions related to a campaign will create a separate call to the DB-service, which in turn will fetch the promotions related to each campaign from the database separately. While in this example only three round trips to the database will be made, resolving the promotions this way will create as many calls to the database as there are campaigns. In the case study application this may potentially result in hundreds of round trips to the database resulting from a single GraphQL query.

With a dataloader, the resolvers will call promotion loader instead. The loader will collect the requests and call a DB-service function, which will fetch all promotions related

to different campaigns in a single round trip. As the promotions related to different campaigns are fetched in the same DB-service function call, they need to be linked back to the related campaign. This will result in some extra data manipulation that needs to be done by the dataloader.

# 5 Evaluation

The evaluation of the research questions is done by reflecting on the learnings, challenges and successes made while developing the study application. The key factors used as the basis of this discussion were defined in the Chapter 3. Each research question is evaluated separately in its own sub-chapter.

## 5.1 Maintainability

The GraphQL approach to data fetching is client-centric, but the hierarchical queries remove the need for writing optimized queries for specific use cases. This helps in keeping the server as general as possible with less client specific optimization. Still GraphQL might come with a slightly larger codebase, as it usually needs to be paired with a data-loader pattern in order to load data in batches. The dataloaders similar to resolvers are usually small in size, but they will still add an extra layer to the architecture resulting in more code. A larger codebase also means more maintenance.

As a resolver function is written for each object type in the schema, the developers are mostly free from worrying how the data is stitched together into a hierarchy. Each resolver containing the resolution of the specific object type also supports overall separation of concerns in the application architecture. While there is a need for writing a resolver for each child object in the context of the parent, the resolver functions are usually quite slim as they are mostly just calling service layer functions to fetch the data.

Versioning or publishing different versions of API is not really necessary with GraphQL. All queries are handled through a single endpoint. By following a tactic of always adding new fields instead and removing them prevents breaking changes. This is possible because clients receive only the fields they have specified in the query. If old fields really need to be removed, they can be marked for deprecation in order to warn clients on future changes. The data model and all operations that can be applied to it are defined and documented in the schema. As a result, GraphQL based API will inherently document itself by design at least to some extent.

Introspection offered by GraphQL makes it possible to query the schema itself. In addition to documenting the API, it enables a variety of third-party libraries which can for example help in diagnostics and debugging. A good example of a third-party extension is the playground offered by Apollo GraphQL. In the playground developers can view the schema and test queries freely, assisted by automatic validation and completion of fields. This makes the cycle of developing and testing new queries much faster.

Making changes to the GraphQL server should be about as easy as making changes to a REST API. There is some extra work in updating the schema and additional layers in the architecture, like the resolvers and dataloaders, but the service layer containing the actual data fetching will quite likely look almost identical to a service layer of a REST API.

In conclusion GraphQL provides various benefits regarding maintainability of the application. Especially introspection and lack of versioning can be seen as benefits over for example REST. GraphQL comes with some extra work in the form of having to write a resolver for each child object in the context of the parent and in having to use a dataloader or similar solution in order to maintain plausible performance. While these are relatively small additions, they both will add an extra layer to the architecture that needs to be maintained.

## 5.2 Best practices for data resolution with GraphQL

In the implementation resolvers were written for each query and each child object in the context of the parent object. GraphQL does not specify how the resolver functions should fetch the data, but care should be used as fetching data too naively can cause the application to call the origin source of the data once for every object in the query. In the case study application a dataloader library was used to load data in batches in order to avoid this.

Before landing on the current implementation more naive approaches to resolving hierarchical data were also used, mostly due to limited time and a need to enable easier development of the client application. These approaches consisted of fetching all data accessible in the query with a single resolver and at a later phase resolving the child objects separately, but without a dataloader. The naive approaches worked in early phases of development, but issues with them were clear and the need for using a dataloader is also brought forward in the Apollo GraphQL documentation.

While it might be justified to take aforementioned shortcuts in the early development, their detrimental effect caused by fetching excess data or stressing the origin source of the data with multiple subsequent requests is clear. These issues seem to be the biggest caveats related to data retrieval in GraphQL. If they are avoided GraphQL should provide a robust and performant API with declarative data fetching.

### 5.3 Using GraphQL for building microservices

As described in the second chapter, the microservice architecture splits the application into small independent services, each running as a separate application. Microservices are built around a specific set of business capabilities, remaining limited in size and context. They are decentralized, can be deployed independently. GraphQL is used for developing APIs, and some of the described features are more related to the overall architecture than just the API layer. The focus of the discussion in this section is on how suitable GraphQL is for building services focusing on a specific set of business capabilities, and its ability to serve multitude of different types of clients, and to function as single service in a group of services.

GraphQL is agnostic on the source of the data. In the case study application external REST API and event-based sources were used. Fetching data from external services and a local database worked as it would with for example a REST based microservice and this is not really the domain GraphQL affects, as data retrieval from source usually happens at the service layer of the application. While requesting data from other services does not bring anything new to the table, what should be taken into consideration is that GraphQL was built primarily with clients in mind and fetching data from a GraphQL API differs from REST somewhat.

GraphQL and REST have a different approach to the data they offer. Resource based approach of REST makes clearer division in the data, while GraphQL offers the data in a hierarchical schema. When serving clients, fetching only the required fields, and minimizing the number of requests makes sense. The client may lack ideal connectivity, due to for example being in a remote location, making the reductions in data transfer meaningful. As the client's user interface is the end of the road for the data, there is no need to retrieve anything more than what is required by the view.

For service-to-service communication the reductions in data transfer sizes might not be as important. Both services are quite likely deployed in cloud with optimal internet connection available. In this scenario the services are also more likely to be interested in fetching complete resources, as they probably will not be the end consumer of the data. So, if the purpose of the microservice is to mainly serve other services, the resource-based approach of REST might be the better option.

In the point of view of a client or service accessing the GraphQL API, the requests will take larger form than for example REST based requests, as the GraphQL query needs to specify all fields that are to be fetched. This will result in some extra work for the application making the requests. If the service or client making the requests is going to always fetch all possible fields of an object from a flat data model, GraphQL will not benefit the project.

Ultimately the suitability of GraphQL for building a microservice will depend on the data model and the needs of the clients and services the microservice is going to serve. GraphQL shines in scenarios with a large hierarchical data model and multitude of different clients and/or services with different data needs. While microservices are limited in scope and business domain, the data they serve might still be highly hierarchical and contain excessive number of fields. In cases like these GraphQL will fit the project well.

# 6 Results and discussion

While the case study application is going to remain in development for around three years, the work so far was enough to provide a base for answering the research questions of the thesis. GraphQL was found to offer various benefits related to maintainability of the application, notably in documentation, generality and lack of versioning. Still it should be noted that a huge part of the maintainability of the application will rise from the overall software quality and following of the selected architectural style.

GraphQL itself is agnostic on how the data is resolved and does not really enforce writing resolvers for each object type in the context of the parent - opposed to just fetching all data available for the query in a single request. Still writing resolvers in this manner should be regarded as a necessity for getting the most out of GraphQL. Using a solution for loading data in batches instead of making a new round trip in each resolver should also be regarded more as a mandatory design decision than a novelty. As a result, GraphQL API will likely come with a larger codebase than a REST based one.

While a GraphQL based service can be accessed by other services, GraphQL has been built with primarily clients in mind. If the aim of the project is to build a microservice geared towards serving other services, the more rigid resource-based approach of REST can be more attractive. Still nothing prevents one from using GraphQL in this scenario and there are no technical limitations for offering both a REST and GraphQL endpoints. The service layer of the application should be usable by both, although GraphQL requires some extra work discussed earlier.

GraphQL has been hyped as the new de facto standard for writing web-based APIs. GraphQL does provide a more client friendly and precise way to fetch data and has a potential to greatly reduce the number of requests made and data transferred, but it still might not be the right choice for every project. Especially smaller projects with trivial non-hierarchical data models won’t stand to gain much from GraphQL. If the services and clients accessing the API are always going to fetch complete resources, the declarative data fetching will just end up causing more work for the developers.

GraphQL has great potential for building general purpose APIs with multitude of different clients. The clients won’t have to worry about performance or fetching too much data and the server itself can be built in a generic manner. Each client is free to select only the fields they are interested in without worrying how the data is actually resolved. Many of the problems related to data under- and over-fetching do not exist. While GraphQL has been built with primarily clients in mind, nothing prevents it from also being used in server to server communication.

# 7 Conclusion

The purpose of this study was to gather experiences from GraphQL, its effects on maintainability, suitability for building microservices and best practices of resolving the data queries. The evaluation was done through a case study of an application built for a customer company. While the case study application provided a solid base for evaluating GraphQL, it wasn't really tailored for the research questions we set in the thesis. There were also some added variables rising from case study application being in early development, most notably constant changes in the implementation, but this in turn helped in the maintainability evaluation.

The evaluation methods proved to be somewhat hard to define as quantitative metrics were not useful here. Still open discussion and reflection on the case study application aided by predefined key points proved to provide a valid base for answering the research questions. While GraphQL was found to provide some benefits regarding maintainability of the application and also to support modularity and separation of concerns in the overall architecture, it also might come with a larger codebase compared to for example REST. While the focus of the study was not in comparison of REST and GraphQL this subject was also touched as REST was closely related to the subject area.

The benefits gained from declarative data fetching GraphQL offers will depend a lot on the use case, as has been noted by various other studies in the topic area. The biggest benefits are found in scenarios with deep data hierarchies and multitude of different types of clients with different data needs. As is with many other decisions made in software development, the developers should focus on using the right tools for the right job. While GraphQL has been hyped as the replacer or the next step after REST and while it does fix some issues often found in REST based systems, it is not a silver bullet.

For suggestions on future research, it would be great to have more studies focusing on security aspects of GraphQL. Open ended queries offered by GraphQL are known to make the server vulnerable to for example denial of service attacks, but studies focusing on this area seem to be non-existent. Also, most of the GraphQL related studies are focusing purely on the data retrieval. Data mutation and event-based data fetching through subscriptions are usually only mentioned in the introduction chapters of studies. As GraphQL is only one technology in the field of declarative data fetching, a study on the general subject area could prove interesting and offer a needed look into the picker picture.

## References

[AJAX, 2020] Mozilla. AJAX. https://developer.mozilla.org/en-US/docs/Web/Guide/AJAX. Accessed: 28. March 2020.

[Al-Badareen et al., 2011] Anas Bassam Al-Badareen, Mohd Hasan Selamat, Marzanah A. Jabar, Jamilah Din, and Sherzod Turaev. 2011. Software quality models: A comparative study. In: Mohamad Zain J., Wan Mohd W.M., El-Qawasmeh E. (eds), *Software Engineering and Computer Systems. ICSECS 2011. Communications in Computer and Information Science, vol 179*. Springer, Berlin, Heidelberg, 46–55.

[Apollo, 2020] Meteor Development Group. Apollo. https://www.apollographql.com/. Accessed: 28. April 2020.

[Bryant, 2017] Mike Bryant. 2017. GraphQL for archival metadata: An overview of the EHRI GraphQL API. In: *IEEE International Conference on Big Data (Big Data)*, 2225-2230.

[Cederlund, 2016] Cederlund Mattias. 2016. *Performance of Frameworks for Declarative Data Fetching: An Evaluation of Falcor and Relay+GraphQL*. M. Sc. Thesis. School of Electrical Engineering and Computer Science, KTH Royal Institute of Technology.

[Deepr, 2020] Github. Deepr. https://github.com/deeprjs/deepr Accessed: 17. March 2020

[Falcor, 2020] Netflix. What is Falcor? https://netflix.github.io/falcor/starter/what-is-falcor.html. Accessed: 17. March 2020.

[Fielding, 2000] Roy Thomas Fielding. 2000. *Architectural Styles and the Design of Network-based Software Architectures.* Ph. D. Dissertation, Information and Computer Science, University of California, Irvine.

[Fielding et al., 2017] Roy T. Fielding, Richard N. Taylor, Justin R. Erenkrantz, Michael M. Gorlick, Jim Whitehead, Rohit Khare, and Peyman Oreizy. Reflections on the REST architectural style and "principled design of the modern web architecture"

(impact paper award). 2017. In: *Proceedings of the 2017 11th Joint Meeting on foundations of software engineering,* 4-14.

[Fowler and Lewis, 2014] Martin Fowler and James Lewis. Microservices. https://martinfowler.com/articles/microservices.html. Accessed: 13. March 2020.

[GraphQL, 2018] The GraphQL Foundation. GraphQL Specification. https://graphql.github.io/graphql-spec/. Accessed: 1. March 2020.

[GraphQL, 2020] The GraphQL Foundation. Learn. https://graphql.org/learn/. Accessed: 28. March 2020.

[Hartig and Jorge, 2017] Hartig Olaf and Jorge Pérez. 2017. An Initial Analysis of Facebook's GraphQL Language. In: *Proceedings Of The 11th Alberto Mendelzon International Workshop On Foundations Of Data Management And The Web*, 10 pages.

[Jia and Wanlei, 2005] Jia Weijia and Zhou Wanlei. 2005. *Distributed Network Systems From Concepts to Implementations*. 1st edition. New York, NY: Springer US; 2005.

[McCall et al., 1977] McCall Jim, Richards Paul, and Walters Gene. 1977. *Factors in Software Quality. Volume I. Concepts and Definitions of Software Quality.* General Electric CO Sunnyvale CA. 1977.

[Mihalcea and Radev, 2011] Mihalcea Rada and Radev Dragomir. 2011. *Graph-Based Natural Language Processing and Information Retrieval*. Cambridge University Press, 2011.

[Mukhiya et al., 2019] Mukhiya Suresh Kumar, Fazle Rabbi, Violet Ka I Pun, Adrian Rutle, and Yngve Lamo. 2019. *A GraphQL Approach to Healthcare Information Exchange with HL7 FHIR. In: Procedia Computer Science Volume 160*, 338-345.

[Nadareishvili et al., 2016] Nadareishvili Irakli, Ronnie Mitra, Matt McLarty, and Mike Amundsen. 2016. *Microservice Architecture: Aligning Principles, Practices, and Culture.* First edition. O'Reilly, 2016.

[Netflix, 2015] Netflix. Netflix Releases Falcor Developer Preview. https://netflixtech-blog.com/netflix-releases-falcor-developer-preview-aefc033df7a7 Accessed: 17. March 2020.

[Netflix, 2018] Netflix. Our learnings from adopting GraphQL. https://netflixtech-blog.com/our-learnings-from-adopting-graphql-f099de39ae5f. Accessed: 17. March 2020.

[Robinson et al., 2015] Robinson Ian, Jim Webber, and Emil Eifrem. 2015. *Graph Databases: New Opportunities for Connected Data.* Second edition. Sebastopol, California: O'Reilly, 2015.

[Shafiabady et al., 2016] Shafiabady Aida, Mohd Naz'ri Mahrin, and Samadi Masoud. 2016. Investigation of software maintainability prediction models. In: *2016 18th International Conference on Advanced Communication Technology (ICACT)*, 783-786.

[SPA, 2020] Mozilla. Glossary, SPA. https://developer.mozilla.org/en-US/docs/Glossary/SPA. Accessed: 28. March 2020.

[Thomas, 2016] Erl Thomas. 2016. *Service-Oriented Architecture: Analysis and Design for Services and Microservices*. Prentice Hall, 2016.

[Vogel et al., 2018] Vogel Maximilian, Weber Sebastian, and Zirpins Christian. 2018. Experiences on Migrating RESTful Web Services to GraphQL. In: Braubach L. et al. (eds), *Service-Oriented Computing – ICSOC 2017 Workshops. ICSOC 2017. Lecture Notes in Computer Science, vol 10797*, 283-296.

[Wittern et al., 2018] Wittern Erik, Alan Cha, and Jim Laredo. 2018. Generating GraphQL-Wrappers for REST(-Like) APIs. In: Mikkonen T., Klamma R., Hernandez J. (eds), *Web Engineering. ICWE 2018. Lecture Notes in Computer Science, vol 10845,* Springer, Cham, 65–83.