Aalto University
School of Science
Master's Programme in Computer, Communication and Information Sciences

Teemu Taskula

Advanced Data Fetching with GraphQL:
Case Bakery Service

Master's Thesis
Espoo, February 12, 2019

Supervisor: Professor Eero Hyvönen
Advisors: Janne Kario M.Sc. (Tech.)
Jukka Keski-Luopa M.Sc. (Tech.)

Aalto University
School of Science

Aalto University
School of Science
Master's Programme in Computer, Communication and Information Sciences

ABSTRACT OF MASTER'S THESIS

Author:	Teemu Taskula		
Title: Advanced Data Fetching with GraphQL: Case Bakery Service			
Date:	February 12, 2019	**Pages:**	56
Major:	Computer Science	**Code:**	SCI3042
Supervisor:	Professor Eero Hyvönen		
Advisors:	Janne Kario M.Sc. (Tech.) Jukka Keski-Luopa M.Sc. (Tech.)		

In today's world building modern applications for multiple environments requires deep knowledge of advanced data fetching solutions. While many longstanding technologies, such as REST and SOAP, have provided robust solutions for most use cases a more advanced data fetching solution is still needed to alleviate issues in efficiency, complexity, and maintainability of current web services.

This thesis studies and compares two data fetching approaches: REST and GraphQL in the context of a case study for a web application, called *Bakery Service*, which possess several issues related to data fetching in its current REST API. An experiment was conducted by first creating separate API implementations for REST and GraphQL. Then the data fetching performance was measured by fetching a single recipe, extracted from the Bakery Service, followed by calculating the total network time of all requests in conjunction with the total size of the response payload and the data utilization percentages based on predetermined set of recipe fields. Additionally, the complexity and maintainability of both technologies were analyzed to gain a better understanding of the non-measurable qualities.

The results of the experiment showed that GraphQL performed considerably better than REST in most test cases. Only REST with field level filtering was able to overcome GraphQL in the data utilization part of the test. Also, the qualitative analyzing hinted that by adopting GraphQL the complexity of an API could be reduced while additionally gaining enhanced development capabilities. These results resemble the findings from other similar studies and strengthen the conclusion that GraphQL can be proposed as an advanced data fetching solution for modern data-heavy applications. However, some future work is needed to broaden the study to take aspects such as caching, mutations, and security into account.

Keywords:	GraphQL, REST, data fetching, performance, complexity, maintainability
Language:	English

Aalto-yliopisto
Perustieteiden korkeakoulu
Master's Programme in Computer, Communication and Information Sciences

DIPLOMITYÖN TIIVISTELMÄ

Tekijä:	Teemu Taskula		
Työn nimi: Edistyksellinen tiedonhaku GraphQL:n avulla: tapaustutkimus Leipuripalvelu			
Päiväys:	12. helmikuuta 2019	**Sivumäärä:**	56
Pääaine:	Computer Science	**Koodi:**	SCI3042
Valvoja:	Professori Eero Hyvönen		
Ohjaajat:	Diplomi-insinööri Janne Kario Diplomi-insinööri Jukka Keski-Luopa		

Nykyajan modernien sovelluksien luominen eri ympäristöille vaatii syvällistä tietämystä kehittyneistä tiedonhakuratkaisuista ja vaikka monet teknologiaratkaisut, kuten REST tai SOAP, ovat tarjonneet tiedonhakuratkaisuja yleisimpiin käyttötarkoituksiin, on kehittyneemmille tiedonhakuratkaisulle siitä huolimatta tarve. Myös verkkopalvelujen kompleksisuuden ja ylläpidettävyyden hallinnointi sekä datan tehokas hakeminen vaativat uusia ratkaisuja.

Tämä työ tutkii ja vertailee kahta tiedonhakumenetelmää: REST ja GraphQL tapaustutkimuksen kontekstissa. Tapaustutkimus kohdistuu verkkopalveluun, jota työssä kutsutaan nimellä *Bakery Service*, ja jonka nykyinen rajapintatoteutus sisältää useita tiedonhakuongelmia. Työn ohella suoritettiin koe, jota varten luotiin omat rajapintaratkaisut molemmille tutkittaville teknologioille. Kokeessa testattiin tiedonhaun tehokkuutta mittaamalla palvelusta otetun reseptin hakunopeutta rajapinnoista. Myös rajapinnan palauttaman datan koko ja käyttöaste sovelluksessa laskettiin esimääriteltyjen reseptikenttien perusteella. Lisäksi molemmat teknologiat analysoitiin kompleksisuuden ja ylläpidettävyyden suhteen tarkemman kokonaiskuvan saamiseksi.

Tutkimuksen tulokset osoittivat sen, että GraphQL suoriutui selkeästi paremmin kuin REST lähes jokaisessa testissä. REST suoriutui paremmin vain data käyttöasteen suhteen, kun haussa hyödynnettiin kenttäkohtaista suodattamista. Laadullinen analyysi antoi myös vihjettä siitä, että GraphQL:n käyttöönotto vähentäisi rajapintatoteutuksen kompleksisuutta antaen samalla käyttöön tehostettuja kehitysominaisuuksia. Saadut tulokset muistuttavat muiden vastaavien tutkimusten tuloksia antaen tukea sille, että GraphQL:n käyttäminen tehokkaaseen tiedonhakuun on suositeltavaa nykyajan dynaamisissa sovelluksissa. Jatkotutkimusta on kuitenkin suoritettava laajemman ymmärryksen saamiseksi ainakin seuraavista osa-alueista: välimuistin käyttö, mutaatiot ja tietoturva.

Asiasanat:	GraphQL, REST, tiedonhaku, tehokkuus, kompleksisuus, ylläpidettävyys
Kieli:	Englanti

Acknowledgements

I would like to thank Taito United for giving me the time and opportunity to write my thesis about a subject that I truly was curious about. Also, huge thanks to Jukka Keski-Luopa who was always supporting me when things felt difficult or when I was not sure how to continue my research. Additional thanks to Janne Kario who helped me finalize my thesis by providing valuable feedback about the work as a whole.

Espoo, February 12, 2019

Teemu Taskula

Abbreviations and Acronyms

AJAX	Asynchronous JavaScript And XML
API	Application Programming Interface
CMS	Content Management System
CRUD	Create-Read-Update-Delete operations
DAO	Data Access Object
DOM	Document Object Model
GraphQL	Graph Query Language
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
JSON	JavaScript Object Notation
REST	Representational State Transfer
SOAP	Simple Object Access Protocol
SPA	Single Page Application
SQL	Structured Query Language
URI	Uniform Resource Identifiers
URL	Uniform Resource Locator
4G	Broadband cellular network, fourth generation
5G	Broadband cellular network, fifth generation

Contents

Abbreviations and Acronyms **5**

1 Introduction **7**
1.1 Problem Statement 8
1.2 Case Study Details 9
1.3 Objectives 10
1.4 Outline 11

2 Background **12**
2.1 REST 13
2.2 GraphQL 15
2.3 Alternatives 18
2.3.1 Falcor 18
2.3.2 JSON API 19
2.4 Related Work 20

3 Methods **23**
3.1 Performance 23
3.2 Complexity and Maintainability 25

4 Implementation **28**
4.1 Experiment 28
4.2 Technical Details 29

5 Evaluation **34**
5.1 Performance 34
5.2 Complexity and Maintainability 42
5.3 Challenges 44

6 Discussion **45**

7 Conclusions **49**

Chapter 1

Introduction

Today's data-driven world pushes us forwards to implement new ways of processing and managing large amounts of data that power our everyday applications. The rise of the Internet has brought forth multiple sophisticated methods for handling this vast amount of data, such as Simple Object Access Protocol (SOAP) [5], Representational State Transfer (REST) [18], and most recently Graph Query Language (GraphQL) [17]. There is a continuing need for efficient management of the ever increasing data consumption of the devices in our pockets. Fortunately, the cellular networks for these devices are improving constantly from today's 4G connections to near future's 5G wireless systems [10] allowing us to consume even more data via our everyday devices. However, not all people have access to these fast networks [32]. Furthermore, the amount of data that modern applications require can be so enormous that advanced methods for data fetching need to be incorporated into an application in order to keep it functional even in fast networks. This issue is evident especially in popular and globally used applications, such as Facebook and Netflix, that provide practically endless feed of data and need efficient solutions for fetching that data in order to provide an unified experience for all their users [6, 30]. Thus, it is not a surprise that many recent data fetching solutions have been developed by these large companies. However, the described data fetching challenges are not restricted to the giants of the web industry. More and more solutions for handling dynamically fetched data is needed as the complexity of today's applications is constantly shifting towards the client-side.

This thesis is a case study about advanced data fetching architectures and methods for a fairly recent commercial Web-based service built for Finnish bakeries and suppliers referred to as Bakery Service for confidentiality purposes. The thesis is implemented in conjunction with a software consulting

company called Taito United[1] that was responsible for building this service. Next, the wider problem of data fetching will be expanded further and then the case study will be introduced in more detail.

1.1 Problem Statement

Many modern client-side leaning applications rely solely on external data sources. Fetching data efficiently over the network becomes a critical task in the system that needs to be solved in a way that latency is minimized as much as possible [34]. Numerous solutions for data fetching exist and new ones are being constantly developed, so it can be difficult to decide which approaches to use without in depth knowledge about the technologies involved. Hence, technologies that have robust built-in capabilities that help solve both the common and more advanced use cases of data fetching are notably valuable. With these technologies the responsibility of deciding what to implement and how to implement various complex elements of a data fetching system is lifted off from the implementer.

For software systems, especially on the Web, it is common that the data requirements of an application evolve over time and can become increasingly complex via new features and growing user base. Fulfilling constantly evolving requirements in today's fast phased development can often lead to sub-optimal solutions to problems that need to be solved quickly in order to satisfy customers or retain certain business advantages. When sub-optimal solutions start to stack on top of each other over time, the maintainability of a system can substantially degrade from the introduced complexity.

Having robust technical solutions that allow efficient data fetching and withstand the inevitable changes in requirements of the system without causing major maintainability problems is a critical necessity for modern applications. In this thesis two technologies, REST and GraphQL, are researched and inspected with these problems in mind. The main research questions for this thesis related to problems described earlier are as follows:

- What are the advantages and disadvantages of GraphQL and REST in the context of data fetching?
- Which one performs better with highly hierarchical data?
- Which one is more maintainable and less complex to implement?

[1] http://taitounited.fi/

These questions will be initially touched on in the next chapter when REST and GraphQL are described in more detail, and finally answered after the evaluation and analysis of the experiment conducted in this thesis. But next, the case study that forms the guiding thread of this thesis is introduced.

1.2 Case Study Details

Bakery Service is a Web-based service that helps bakeries to automate and produce the necessary label information for their products. These product labels are lawfully required to contain information about the ingredients, allergens, and nutrients of the product which can be easily handled with the service. The service involves two user groups: the bakeries that input their recipes into the system and receive the calculated product label information, and the suppliers that offer their product and ingredient information for the use of the bakeries.

For a bakery that is registered in the Bakery Service it is common to have many recipes with multiple sub-recipes and ingredients that then consist of a wide range of different nutrients and allergens. This deep hierarchy of the data means that the final data structure for the whole recipe and the product label that is based on the recipe can be quite large and complex, inevitably introducing challenges for efficient data fetching in the system. More precisely, these large and complex data structures can, depending on the API design choices, either cause a few slow and expensive requests to the server or several smaller requests that add up to an expensive operation as a whole. As a result the client application can become unresponsive while waiting for the requests to resolve thus decreasing the user experience.

In the case of the Bakery Service the API was designed in a way that the number of requests could be minimized in order to reduce the total network time while having idiomatic endpoints for each resource of the system. This was achieved by following the JSON API specification [26] that enabled flexible data fetching via capabilities such as field filtering and resource nesting. However, following such a specification introduced complexity in various sections of the system. Firstly, the data access layer of the API responsible of fulfilling the convoluted data requirements of the client incorporated complex database queries in order to minimize the number of queries to keep the response times as fast as possible and to facilitate the powerful features dictated by the JSON API specification. The JSON API specification will be introduced further when the alternatives to REST and GraphQL are discussed about in the next chapter.

In addition to the server-side complexity, the data format enforced by

the JSON API specification in conjunction with the deep hierarchical nature of the recipe and product label data makes it difficult to manage the state of the client application with high confidence which is why many custom methods and functions have been implemented attempting to handle this complexity in Bakery Service's client application. The current API mostly supports flexible data fetching based on the specific needs of the client but in spite of following the JSON API specification, some data under- and over-fetching issues are present in the client application. In many cases it is simpler to fetch more data than is actually needed because the coordination of the application state becomes difficult to manage when various parts of the application require different subsets of the same data. So, fetching only the required amount of data and ensuring that any custom built utility functions perform properly is a difficult task, and a more preferable solution would be to reduce the complexity of the data or shift the responsibility for handling the complexity to a more robust third-party solution or pass the responsibility to the server all together.

1.3 Objectives

The objective of this thesis is to first research GraphQL as a full or partial replacement for the current REST architecture in the Bakery Service and identify any valuable tools related to using it in practice. Then an experiment with the tools and technologies found will be implemented focusing on only a single page inside the Bakery Service system: the page for showing the details of a recipe. Other pages in the Bakery Service were also considered but only the chosen page is focused on to keep the scope of this case study manageable. Also, the recipe details page is one of the data heaviest pages in the system and thus has an urgent need for data fetching optimization.

The main goal of the experiment is to answer the research questions described earlier and to find a data fetching solution, based on the research made, that compared to the current REST based architecture decreases the load time for all the data required to show the selected page and thus have it be responsive to user activity more quickly. This should be the case even for recipes that have a large number of sub-recipes and ingredients in it. Various aspects of the API requests made in the system will be measured to determine the efficiency of the experimented technologies. Additionally, the inspected solutions will be analyzed from assorted angles related to actually implementing and maintaining them in the practice. These metrics will be introduced in Chapter 3.

1.4 Outline

Now that the problem for this case study has been formulated and the main objectives have been defined the general structure of this thesis can be scanned through. First, the technical details of REST and GraphQL for data fetching are introduced and compared to gain better understanding of the differences between them and the use cases they fit into. Alternatives to GraphQL and REST are also briefly considered and compared. Next some related work will be presented to broaden the academic horizon for this study. Then the selected methods and metrics for evaluating the two technologies are going to be introduced, and after this the implementation for the experiment will be presented. Then the experimented technologies will be evaluated and analyzed, and the results of the experiment will be discussed about in conjunction with going over the challenges encountered during the experiment. Finally, based on the experiment and the analysis results the potential solution for the Bakery Service will be proposed and future work for the study will be discussed.

Chapter 2

Background

The World Wide Web has evolved from a purely document-centric form into a more dynamic and data-driven form where websites are not just server-side rendered static pages anymore but respond to user actions dynamically by combining data from multiple sources. Today many applications are moving the complexity towards the client-side by taking advantage of the modern capabilities of JavaScript for manipulating the HTML Document Object Model (DOM) and making AJAX calls to the server for data needs. In extreme cases, like in the Bakery Service, the whole application logic is contained inside a handful of scripts that are loaded asynchronously when the page is visited by a user. The initial HTML document for the page can consist of only a few lines of minimal HTML code to bootstrap the required resources. Even technologies behind inherently static websites, such as Content Management Systems (CMS), most notably Wordpress that powers a large part of the Web [42], have started to offer API endpoints that expose the same data, that would be normally inlined to the static document, to the clients in order to enable the development of more dynamic applications [47].

For almost a decade REST has been considered as the standard technological solution for dynamic applications on the Web. REST in conjunction with other Web specifications, such as Hyper Text Transport Protocol (HTTP) and Uniform Resource Identifiers (URI), has guided the development and design of the Web into a more resource-centric direction and allowed us to build highly scalable applications [18].

However, while REST has laid down the foundations of the modern Web it still lacks many aspects that are needed in today's data-heavy applications that have to function efficiently in numerous network environments and in an unfathomable number of different devices ranging from cheap low-end mobile devices to extremely powerful high-end desktop computers. For these challenging needs many solutions have been introduced from which GraphQL

is a notable one. Next, the background in conjunction with the strengths and weaknesses of REST and GraphQL are discussed in more detail, and at the end of this section other alternatives to the former solutions are introduced and briefly analyzed.

2.1 REST

REST, first described by Fielding [18], is an architectural style for distributed and scalable Web systems. Fielding introduced five major design factors for REST: cacheability, client-server communication, hierarchical layers, statelessness, and uniform interfaces. Additionally, an optional sixth factor called Code-On-Demand can be considered allowing the clients to enhance their functionality by downloading code as applets or scripts. However, this can be ignored as the modern Web has moved away from applet-based applications [44].

The core idea of REST revolves around resources identified by URIs, and the combination of these URIs form the endpoints of an API. These resources can be fetched and managed by sending HTTP requests, such as GET, POST, or DELETE, that correspond to the actions performed against the resource to the URI of the resource. For example, a response to a GET request made to an endpoint, such as **/recipes/1**, can look something like shown in Listing 1 where the response is in JSON format. Another relatively common response format is XML which, however, is far less popular in Web applications because it is more difficult to parse into a usable form in JavaScript compared to JSON [43]. Notably, the REST specification does not limit what a resource can be so it is possible to have for example images or documents as resources. The specification however enforces that when the client makes a request to the server it must provide all the necessary state that is needed to process the request to keep the interaction context-free [35].

```json
{
    "id": 1,
    "name": "Granny's apple sauce",
    "bakery": "Parker's Bakery",
    "ingredients": [1, 3, 8]
}
```

Listing 1: Example JSON response

In Listing 1 one can also see a common design aspect of REST APIs:

any sub-resources included in a response are represented by their identifiers or by URI strings to the resource instead of the resource data itself. These identifiers can then be used to further request the complete data for the sub-resources. However, having to request N sub-resources one by one requires N additional HTTP requests to the server instead of receiving all the needed data in one round-trip. The requests are handled on the server-side by routes that are bound to the specific URIs of the resources in the system. There is no built in concept of nested URIs in REST meaning that a single request can only invoke one route handler making it inconvenient to include sub-resources.

Nonetheless, REST is not without it's strengths. Having resources in the center of REST guides the design of the whole system into a more modular direction delegating the responsibilities of data handling to each resource entity respectively. By leaning against fairly simple protocols, such as HTTP and URI, REST is simple to adopt and use in any application. There is no need for custom libraries and tools to establish a connection between a client and a server and to send data between them. Also, REST makes it simple to design the API interface in a human-readable manner since after all everything consists of simple URIs and a limited set of HTTP verbs. The design result for an API should be a clear and uniform description of the various endpoints of the system. Furthermore, another advantage of REST related to it's modular nature is maintainability of services. If designed with the REST mindset the responsibilities of different services should not overlap making it straightforward to maintain them separately.

But, as stated by Fielding and discussed earlier, one notable weakness for REST based systems is that a client has to conform to the uniform interface provided by the API [18]. Generally the data is transferred in a standardized form possibly including only the identifier of any sub-resources or the whole requested resource even though only some small portion of that resource is needed by the client application. Consider a view such as the listing of all recipes owned by a bakery in the Bakery Service where only the names of the recipes are shown in the list. With the REST architecture requesting these recipes involves fetching the whole resource for each recipe in order to just list their names. This is highly inefficient, especially for a large number of requested resources, and can create a bottleneck inside the application degrading the user experience. In general, this kind of behaviour is called data over-fetching where most of the requested data is actually not needed by the client. Another common shortcoming of REST APIs is data under-fetching where the client has to first request some resource and then read the identifiers of any sub-resource of that resource and request them one by one to fill in the missing data of the original resource. So, since REST is

a resource-centric architecture it more easily leads to data over- and under-fetching than a more data-centric approach, such as GraphQL, that provides only the data that is requested by the client [4].

An additional minor weakness of REST is that it does not have any notion of session management built into it other than enforcing context-free communication between the client and the server leaving the responsibility of session management to the developer [35]. Common approaches to session management include usage of browser cookies, authentication tokens and offline storage.

At this point it is important to note that many of the weaknesses of REST can be solved via use case specific filters [27] or custom endpoints that fulfill the specific data need for the client application. However, implementing these filters can get cumbersome and the number of custom endpoints can start to grow in an unmaintainable manner increasing the server-side complexity as the data requirements of the application evolve over time [34, 40]. Furthermore, using these kind of sparse field sets to filter the data to the clients needs imposes problems for caching [3]. So, instead of having a dozen custom endpoints that serve only a certain need it would be more manageable to have a built in way to request only the data that the client needs inside the data fetching technology itself. This is what GraphQL does in it's core as can be seen in the next section.

2.2 GraphQL

GraphQL, created in 2012 and standardized in 2015 by Facebook, is a query language that enables client-server applications to precisely describe the capabilities, interactions, and requirements of their data models [17]. Contrary to the resource-centric approach that REST follows, GraphQL is a data-centric specification to building APIs. When a whole resource would be requested with REST, a set of specific fields of a resource can be asked with GraphQL moving the responsibility of selecting the required data to the client from the server [17]. Listing 2 shows a simple GraphQL query that requests the same data shown in Listing 1 in the case of REST. The only difference to that request is that the names of the ingredients are added to the GraphQL query, and the response for this can be seen in Listing 3.

Listings 2 and 3 demonstrate the data-centric approach that GraphQL has for data fetching compared to REST. The fields for sub-resources, here only the name of the ingredient, are included to the query, and then present in the response for this query without having to make additional requests.

The design principles behind GraphQL, as listed in the specification [17],

```
query {
    recipe(id: 1) {
        name,
        bakery,
        ingredients {
            name
        },
    }
}
```

Listing 2: Simple GraphQL query

```
{
    "id": 1,
    "name": "Granny's apple sauce",
    "bakery": "Parker's Bakery",
    "ingredients": [
        { "name": "Sugar" },
        { "name": "Flour" },
        { "name": "Apple" }
    ]
}
```

Listing 3: Response from GraphQL query

are hierarchical structure for queries, driven by a product-centric mindset, safety by strong-typing, client-specific queries and responses, and introspection for enhanced development experience. So, in order to enable many of these principles GraphQL revolves around schemas of different types present in the system instead of resource URIs to define what is available in the API and what kind of actions are possible. Moreover, GraphQL APIs only have one endpoint capturing all queries instead of multiple endpoints as in the case of REST. Listing 4 shows a simple schema constituting of a few truncated types.

The top level types of operations in GraphQL are queries (Query), mutations (Mutation), and subscriptions (Subscription) that describe all the possible ways to fetch and alter the data [14]. They act as entry points for the system while the rest of the types describe the structure of the objects which is why they are called object types [15]. Object types, such as the Recipe and Ingredient types in Listing 4, are composed from other object

```
type Query {
  recipe(id: ID!): Recipe
  ingredient(id: ID!): Ingredient
}

type Mutation {
  addRecipe(input: AddRecipeInput): Recipe
}

type Recipe { ... }
type Ingredient { ... }
input AddRecipeInput { ... }
```

Listing 4: Example GraphQL type definitions

types or primitive scalar types such as strings (String) or integers (Int). Additionally, there exists other more higher level types such as interfaces, unions and input types that can be used to handle complex data requirements.

Compared to REST, and many other solutions before REST, GraphQL makes it simple for client to take control of it's own data requirements enabling lighter network communication between the client and server. Having all the necessary data in a lightweight response payload and making less network requests is especially important for people with low-end devices and people that have limited bandwidth and restricted data usage limits. Also, network activity is one of the most battery intensive activity in today's mobile phones which is why mobile phones benefit greatly from GraphQL [45].

Since GraphQL requires the developer to describe all the data structures and interactions with types it allows various tools to verify that a query is syntactically correct and valid within the GraphQL type system before actually executing the query [17]. This can have a considerable positive impact on developer experience and lead into having less bugs in the implementation.

GraphQL also alleviates the pain points related to versioning of an API by following a tactic of always adding new fields instead of removing them [12]. So in practice, there is no need to explicitly version the API since the client can always request only the data it needs by selecting only a specific subset of fields instead of conforming to the data the server decides to return like in the case of REST [12]. But, if needed the developers can add deprecation warnings to the fields of the schema they wish to remove in the future. This approach of always adding fields and never removing them would be problematic in the context of REST since all the fields of an resource are

included in the payload of a request by default, so the payload size would keep increasing over time when new fields are added.

Finally, a notable strength that should not be overlooked is the ability to introspect the exposed API easily with built-in tools provided by GraphQL. This makes exploration of the data and available actions simple, and serves as a powerful development tool for developers that are either building or consuming the API. [13]

Every technology is not without it's weaknesses. Both GraphQL and REST are served over HTTP but since GraphQL has only one entry endpoint for the API it only uses the GET and POST methods instead of the full set of available HTTP methods commonly mapped to Create, Read, Update, and Delete (CRUD) operations. Developers also need to be careful to only use GET method for queries and POST method for everything else [16]. Furthermore, since there is only one URI for the API it is impossible to cache the responses of the API in a traditional manner but instead a custom caching mechanism has to be implemented on the client-side or on the server-side with the help of globally unique identifiers [11]. Lastly, it can be argued that GraphQL is inherently not as simple to use as REST since it requires upfront work for defining the schema of the API with the proper resolvers for various fields of the entitites in the schema. Also, due to the unique query language it can be more difficult to handle various custom edge cases, such as file upload, with GraphQL and usually a combination of REST and GraphQL APIs is introduced to achieve these requirements. However recent work has been done to introduce a specification for multipart form field structure in GraphQL to support file uploads [36].

Now that both REST and GraphQL have been introduced in detail some alternatives to these approaches are briefly discussed in the next section.

2.3 Alternatives

2.3.1 Falcor

GraphQL is not the only data-centric specification for efficient data fetching on the Web. In 2015 Netflix introduced their approach to handle massive amounts of data in their applications called Falcor [29]. Falcor allows the server to model and expose data as a single JSON resource to its clients while supporting queries that request only some parts of that resource. Both GraphQL and Falcor aim to solve the same problem of managing the constant increase of data requirements in modern applications. However, there are some notable differences between the two. While GraphQL has a schema and

static types Falcor does not provide any built-in way to add type checking or schema validation. Moreover, Falcor only allows passing simple arguments, IDs, and specific ranges to queries. Finally, a valuable feature for developers that Falcor is missing compared to GraphQL is a built-in schema exploration tool that makes it simple to understand the API surface of an application.

So, even though Falcor and GraphQL are relatively similar and attempt to solve the same issue presented in this thesis, GraphQL was selected for further research instead of Falcor because of it's powerful features, wider adoption in the web development community and vast offering of tools, libraries, and other resources that make it easier to use GraphQL than Falcor at the moment of writing this thesis.

2.3.2 JSON API

While GraphQL is a specification for efficient data handling with a large number of reference implementations in different programming languages and Falcor is a ready to use solution for the same matter, JSON API is a specification for REST architectures describing how a client should make requests for resources to be fetched or altered and how a server should respond to those requests [26].

JSON API attempts to solve data under- and over-fetching issues by restricting in what format the data should be passed between the client and the server and how that data can be filtered by clients without losing too much discoverability, flexibility, or readability of the data and the API endpoints [26].

The client-server communication for the Bakery Service has been implemented with the rules of JSON API specification in mind from the start. Even though following the specification has helped to mitigate some data fetching issues in the Bakery System, it has generated various other issues, such as maintainability overheads, difficulties of extending existing features, and project-wide complexity, that have started hinder the development speed and reduce developer experience and confidence.

The most notable issues have to do with lack of tooling and libraries, and added complexity in state management on the client-side. For example, requesting only a subset of the fields of some resource is supported by JSON API but keeping track of what fields have already been fetched and stored in the state is a major challenge. Without a built-in way of doing this it can quickly become tedious to handle all the logic for caching and state management for all different subsets of the data.

In addition to the client-side challenges, the requirements enforced by the specification make it extremely difficult to implement robust server-side solu-

tions that satisfy the specification. Especially implementing sparse fieldsets with support for nested resources requires considerable amount of work. For these reasons exploring other solutions, such as GraphQL, for efficient data fetching and management is essential to alleviate the pain of maintaining and extending the Bakery Service in the future.

Next, before diving deeper into the research methods for this case study, some related work is presented to gain better understanding of the academic field for REST, GraphQL, and other data managing approaches.

2.4 Related Work

GraphQL is a fairly recent technology specification in the web development industry [17] especially compared to the REST specification which has been around for almost twenty years [18]. A consequence of GraphQL's young age is that the academic spectrum for it is still rather narrow which makes it difficult to have a comprehensive view of the academic field in the context of this thesis. However, some in depth research has been made solely about GraphQL and also about the adversarial nature of GraphQL and REST as competing approaches for data fetching on the Web.

There have been a few other theses published that compare REST and GraphQL as data fetching solutions. Firstly, Cederlund [7] compared three API implemetation technologies: Falcor, GraphQL with Relay, and REST in his thesis in order to determine which technology provided the fastest data fetching in terms of latency, number of network requests, and the amount of data transferred. The naive initial result was that a custom endpoint specific to the data requirements of a client was the fastest solution from all three. However, Cederlund pointed that when multiple requests were needed to resolve all of the data GraphQL in conjunction with Relay, a client-side library for GraphQL, provided the lowest latency. Finally, the key finding from Cederlund's thesis was that even thought the response payload size could be decreased with GraphQL the size of the HTTP request could mitigate some or all of the benefits of a leaner response due to sending a large query string with the request. Another thesis related to GraphQL, by Eeda [9], proposes and implements GraphQL as a solution for web applications that need to increase their data fetching performance and avoid over- and under-fetching of data. Eeda integrated GraphQL into an existing real-time dashboard system in order to investigate potential performance improvements. While the thesis was mainly user interface architecture specific the key findings showed that GraphQL was able to reduce the total number of network requests considerably achieving a 153% and 245% faster render times for specific views

inside the dashboard application.

Vazquez *et al.* [40] implemented a GraphQL API to replace the former REST API for the Observatory of University Employability and Employment (OEEU). The initially developed REST API was found inflexible to data requirements of new and existing clients that wanted to prototype new analysis methods based on the data from the API. Additionally, due to the nature of the data collected by OEEU the high number of endpoints made it difficult to fetch all the required data by the client without resorting to multiple requests. Also, OEEU's REST API endpoint outputs were tailored to fulfill the needs of all possible clients in the ecosystem causing difficulties in reutilization and maintainability. By moving to GraphQL, OEEU managed to free the clients of the API to evolve independently and reduced client specific complexity related to data restructuring when the data provided by the REST API did not directly fit the needs of the client. Furthermore, maintaining the GraphQL API turned out to be much simpler than maintaining the REST API since changes in the client did not demand changes in the API. All in all, OEEU managed to improve one specific presentation component significantly by decreasing the average network time by 32.26% and request payload size by 47.86%.

Hartig and Pérez [21] studied GraphQL defining the semantics of its query language based on a logical data model to further inspect the complexity of the language. They concluded that many performance related aspects of the language, such as evaluation, enumeration, and size-computation of a query, could be solved efficiently. Since GraphQL queries can potentially be highly nested and recursive the resulting server-side execution of the query might incur a heavy burden on the server and cause denial of service issues. Hartig and Pérez investigated ways to determine the cost of the query without having to fully evaluate it beforehand. They also examined how this issue was handled in popular GraphQL libraries and found out that some libraries significantly overestimated the cost of a query in a frequent manner. In depth analysis of the performance characteristics of GraphQL is vital for any production level API in order to avoid denial of service issues and user experience regressions born from slow data resolution. Yet, the implementation for the experiment related to the Bakery Service examined in this thesis will ignore any query complexity aspects to keep the scope of this study focused. Applying these kind of performance improvements will, however, be mentioned when potential future improvements are discussed later in this thesis.

When adopting GraphQL in existing applications that are built with REST it can be easier to adopt it incrementally by first introducing GraphQL as a gateway API that uses REST endpoints to resolve requested data. Hernandez-Mendez *et al.* [22] introduced a Query Service meta-model that

was based on a GraphQL server implementation communicating with external REST API endpoints to define a unified endpoint for Single Page Applications (SPA). The Query Service can be semi-automatically generated via four steps requiring some hands on work from a developer to refine the output code. The end result of this process is a set of JavaScript files that constitute a well structured and consistent GraphQL server that promoted higher development speed than a manually created server. However, the generated server implementation was not able to merge data from multiple sources without manual modification, which turned out to be a notable limitation to the approach proposed by Hernandez-Mendez *et al.* Also, updating existing code generated by models was an unwieldy process since after the code had been created it was detached from the model that created it, and changes in underlying API specifications required either manual changes to the code or a new model that would be used to generate new code to replace the existing code. The main advantage of a semi-automatically generated API implementation is not directly applicable to the Bakery Service since all the data is provided by a single REST API. Nevertheless, the notion of having a GraphQL server in front of a single or multiple microservices, which might be based on REST or other technologies, is an interesting approach that will be investigated in this thesis by experimenting with a GraphQL layer on top of a REST API.

Now that the academic field for GraphQL and REST related technologies, studies, and solutions has been glanced over it is time to present the methods for evaluating and analyzing the experiment conducted in this thesis.

Chapter 3

Methods

There are various ways one can evaluate and compare technical implementations. Quantitative analysis can be done via measuring various aspects of a system such as performance to gain deep insight of the numeric characteristics of the system under inspection. For the Bakery Service the relevant quantitative metrics include the total number of network requests made, the total network time from the start of first request to the end of the last request, the payload size of the response, and the response data utilization percentage in the application. These metrics describe the performance of the system and they are easily measured and comparable between different technical implementations. Other possible metrics include memory-utilization [38] and HTTP communication throughput [39] but they are not included in the analysis for being unnecessarily low level metrics and not posing an important relevance for the Bakery Service and thus not providing valuable information for technical comparison in this thesis. In addition to quantitative analysis the system can be inspected from qualitative point of view describing the characteristics of the system that are not easily or sensibly measured in a quantitative way. For the Bakery Service these qualitative characteristics include complexity and maintainability. All mentioned metrics were chosen to establish a well rounded analysis of the implemented systems and to answer the technical concerns of Taito United. Next, the chosen metrics are elaborated on and methods for measuring them are introduced.

3.1 Performance

As already stated, the performance metrics relevant to this thesis are the number of network requests and their duration, the response payload size, and the response payload utilization percentage. Especially the total du-

ration of all network requests is of interest since it is possible to request resources in parallel, instead of in sequence, to some extent by utilizing the browser capabilities for multiple simultaneous connections [37]. In recent years modern web applications have adopted the SPA approach for client-side implementations making the network layer one of the main bottlenecks for an application since all required data needs to be dynamically fetched from external sources during runtime. This means that the faster the application can resolve its data requirements the better the application works in practice. So, making multiple API requests that hit the network will degrade user experience. Also, it is beneficial to avoid slow requests that arise from requesting large data payloads. So, in order to have satisfactory performance for the system at whole it is essential to make as few network requests as possible and request only the necessary data needed by the client.

The methods for recording performance metrics in the Bakery Service experiment are straight forward. Firstly, the number network requests made by the client will be recorded and a label of the resource in question (recipe, ingredient, bakery, supplier) will be attached to that recording. Additionally, the network time for each recorded request will be measured to see which requests took the longest. Secondly, the size of the response payload will be calculated with the estimation function shown in Listing 5.

```
const estimatedSize = JSON.stringify(jsonPayload).length;
```

Listing 5: Estimated request payload size

Finally, the data utilization percentage for the response payload will be calculated based on the list of object fields that are actually used in the hypothetical user interface defined for the experiment. A resource such as a recipe or an ingredient can have numerous fields stored in the database but only a subset of these fields are used on the client-side. The fields of this experiment are selected to reflect the real-world user interface in the Bakery Service for displaying details of a single recipe. The selected fields used for both technical solutions are shown in Listings 6 and 7 represented as GraphQL queries. Some fields and resource entities, such as nutrients and allergens, are neglected since they are only used in special occasions in the Bakery Service and in order to keep the experiment straightforward to implement. Also, the body size of the requests is not taken into account for the payload size calculation since there exists ways to minimize the effect of it for the network request duration. These solutions will be discussed more in Chapter 6.

```
query recipe {
  recipe(id: "1") {
    modified
    nameFi
    netWeight
    domesticity

    bakery {
      companyName
    }

    ingredients {
      amount
      ingredient {
        eanCode
        generalNameFi
        supplier {
          companyName
        }
      }
    }
  }
}
```

Listing 6: A GraphQL query for a recipe without sub-recipes

3.2 Complexity and Maintainability

Code complexity has been studied extensively in the past decade to get valuable insight about the elements that contribute to code quality. Measuring and evaluating the complexity of a software implementation enables developers to make insightful decisions about the implementation details that can help make improvements to the code quality and decrease maintainability time [2]. Various metrics for calculating numeric values for code complexity exists, such as Halstead [20] and McCabe [28] complexity measures, which can be incorporated into tools for developers to use. One such tool is *complexity-report*[1] which is used to programmatically analyze Node.js programs written in JavaScript by running the tool against the files of the program. The output for this analysis is a report listing complexity measures, such as Halstead

[1] https://github.com/escomplex/complexity-report

and McCabe, for each file of the program and for each function in that file. From there on the results can be used to identify potentially problematic sections in the program for more detailed manual analysis.

In addition to numerical analysis of complexity it is beneficial to inspect non-measurable qualities of a software system. Especially when it comes to web development industry, which constantly moves forward in a rapid pace [31], qualities that improve maintainability of given software are highly valuable. For software in general and especially for technologies such as REST and GraphQL used to create APIs, a non-exhausting list of valuable qualities include: solid security, fast development speed, evolutionary robustness, and high cohesion and low coupling between the client and the server. It is favorable for these qualities to be built-in to the core technologies of the API in order to establish standardized ways of utilizing them. Yet, often these same qualities can be attained via external libraries that supplement the core technologies. So, later in this thesis the implemented experiments will be evaluated in the context of the stated qualities to gain a well rounded understanding of the involved technologies.

```
fragment RecipeIngredients on Recipe {
  ingredients {
    amount
    ingredient {
      eanCode
      generalNameFi
      supplier {
        companyName
      }
    }
  }
}

query recipe {
  recipe(id: "1") {
    modified
    nameFi
    netWeight
    domesticity

    bakery {
      companyName
    }

    ...RecipeIngredients

    subrecipes {
      ...RecipeIngredients
    }
  }
}
```

Listing 7: A GraphQL query for a recipe with sub-recipes

Chapter 4

Implementation

Now that the chosen analysis methods have been introduced it is time to examine the experiment conducted in this thesis. First, the general structure of the experiment is presented, then the technical details of the final implementation are briefly displayed, and finally the challenges encountered during the implementation phase are discussed about.

4.1 Experiment

In order to compare GraphQL and REST in the context of the Bakery Service a set of experiment cases was constructed to explore various conditions where the API could be used in real life. When the case study's problem statement was introduced in Chapter 1 the main challenges mentioned were complex data handling and state management in conjunction with data over- and under-fetching issues on the client-side and severe technical complexity on the server-side causing maintainability overhead. Therefore, any constructed test case should take these issues into account to meaningfully challenge the technical implementations for sound comparison and analysis. Table 4.1 lists all the test cases included in the study.

As seen from Table 4.1 the experiment is split into two main categories based on the type of data involved: a recipe without sub-recipes and a recipe with its sub-recipes. A recipe without any sub-recipes represents a simple base case inside the Bakery Service. On the other side, a recipe with sub-recipes represents a highly nested data structure present in the Bakery Service. Both types of recipes are used in a hypothetical user interface for showing the most important details of given recipe. These details include the bakery of the recipe, all ingredients of the recipe including the supplier of the ingredient, and any sub-recipes of the recipe recursively including its

Data type	Technology	Source of data
No sub-recipes	REST	Local
No sub-recipes	GraphQL	Local
No sub-recipes	GraphQL *(in front of REST)*	External API
With sub-recipes	REST	Local
With sub-recipes	REST *(with field filtering)*	Local
With sub-recipes	GraphQL	Local
With sub-recipes	GraphQL *(in front of REST)*	External API

Table 4.1: Experiment test cases

ingredient data.

After defining the two guiding categories for the test cases it is necessary to define the technical variations included in this study. Firstly, the simulated REST API for the Bakery Service is tested with and without field filtering which was discussed in Chapter 2 when REST was introduced. Additionally, all REST implementations in the experiment use a local source of static data simulated as a database. Furthermore, the implementation of the REST API will follow an idiomatic approach meaning that no custom endpoints will be used to satisfy client specific needs but only client-agnostic endpoints for each resource will be implemented. Secondly, a GraphQL API is implemented with two sources of data: local static data and the REST API described earlier. The reason for this is that implementing a GraphQL API as a complete replacement for the current REST API is most likely not a viable option for Taito United. Instead an incremental adoption of GraphQL would be more feasible. This can be done by initially having the GraphQL API in front of the existing REST API using it as the data source. A potential downside for this approach is that the network requests made between the GraphQL API and the REST API add up to substantial amount of latency degrading the client-side user experience. Finally, all test cases will be executed in the Chrome browser in two different network conditions: fast WiFi and slow 3G mobile network. Conducting the experiment in both fast and slow network environments is critical to ensure that the results are not distorted by a non-realistically optimal environment.

4.2 Technical Details

The technical details of the implementations for both REST and GraphQL will only be shortly described and the description for the client-side implementation is totally skipped due to it not being relevant for this study.

The technologies and libraries used to implement APIs for both REST and GraphQL were selected based on a literature review [1, 25, 46]. Figures 4.1, 4.2 and 4.3 illustrate the different architectures of the implemented API servers. The different actors are color coded as follows: green indicates a client that makes requests for data, blue indicates the API server that fulfills the data needs of the client, and yellow represents the data source of the API server.

A simple REST API was implemented with a Node.js library called *express*[1] having separate routes for each resource entity in the system. Each route defines its sub-routes that form the final endpoints of the system. Clients can send HTTP request to these endpoints to fetch data. The coarse architecture of the REST API can be seen in Figure 4.1 where in addition to the endpoints there exists Data Access Objects (DAO) that are used via Data Access Manager in the routes for data access. Using DAOs makes it simple to switch between various data sources behind the scenes without having to modify the routes in any way.

The basic structure of the GraphQL API server is mostly the same as of REST API server's as can be seen in Figure 4.2. The server was bootstrapped with *express* just like in the case of REST but the GraphQL part of the server was implemented with *apollo-server-express*[2] library. Unlike with REST which has multiple endpoints for the API there only exists one endpoint for the GraphQL API that the client can use to send GraphQL queries to fetch data. These queries are validated against the GraphQL schema and if the query is valid it will be executed against the schema filling the requested data with the help of resolvers. Each entity in the system: recipe, ingredient, etc. has its own resolver functions that are responsible of resolving a specific field in the schema. The resolvers have access to the DAOs in order to access the requested data. The only difference between the GraphQL API using the local data, shown in Figure 4.2, and the GraphQL API using the REST API as its data source, shown in Figure 4.3, is the data access layer. In practice only a single GraphQL API implementation was created and it was possible to change the data source of the API on the fly via a DAO specific route.

Both described API servers were deployed to Google Cloud Platform[3] in the *europe-west1* zone in the same Kubernetes[4] cluster to have them physically close to the location where the experiment was conducted and to guarantee that all HTTP requests between the APIs were executed inside the same cluster ensuring that no unnecessary latency was introduced to the ex-

[1] `https://www.npmjs.com/package/express`
[2] `https://www.npmjs.com/package/apollo-server-express`
[3] `https://cloud.google.com/`
[4] `https://kubernetes.io/`

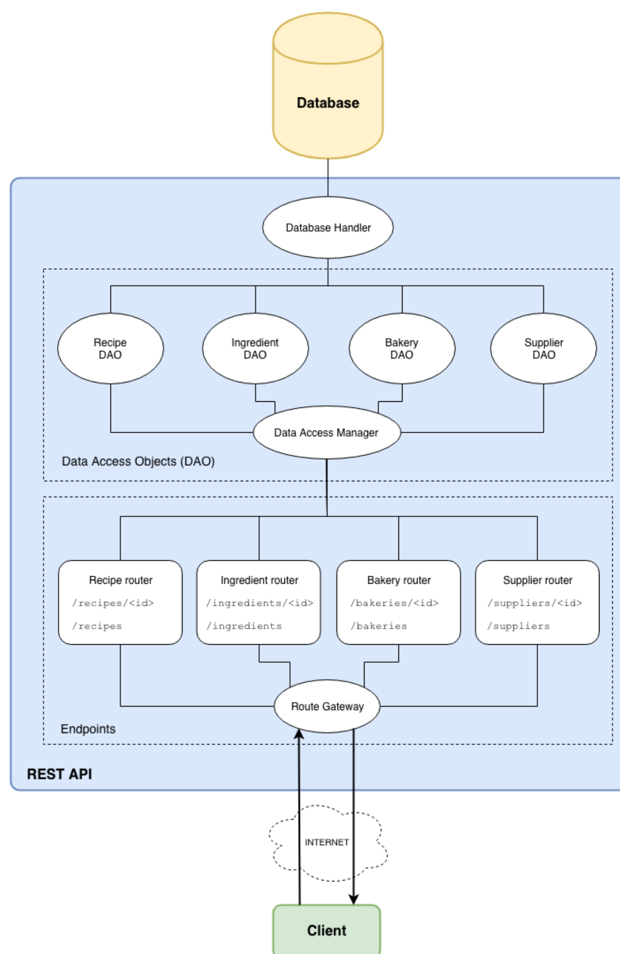

Figure 4.1: The structure of the REST API with a local data source.

periment. And to conclude, the experiment was run one hundred times and an average of the measured results was taken to mitigate random network issues and establish valid measurements.

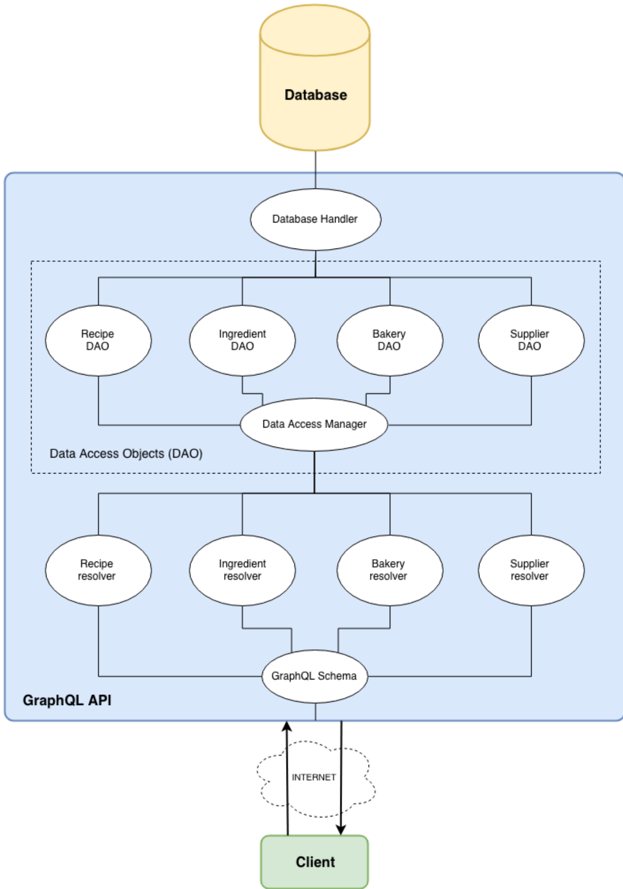

Figure 4.2: The structure of the GraphQL API with a local data source.

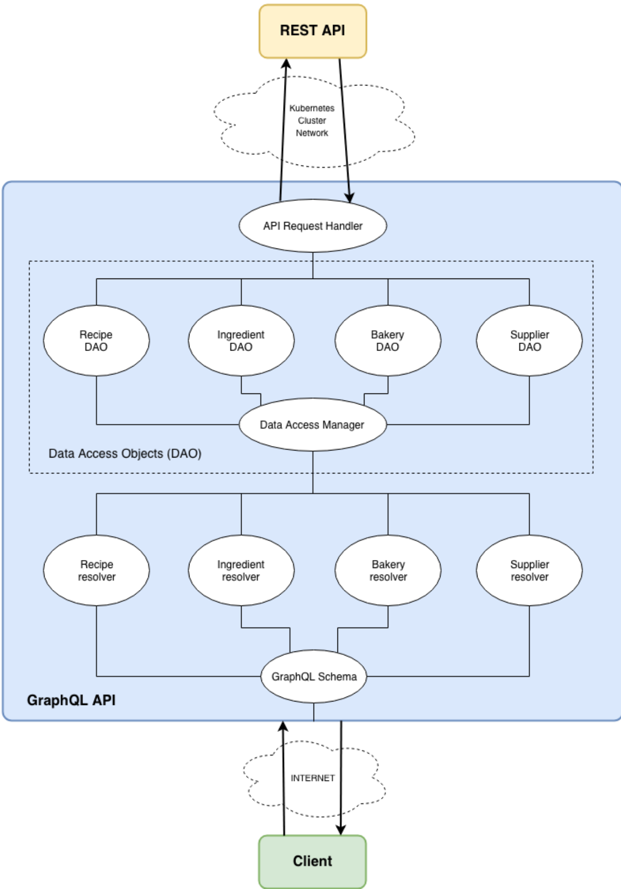

Figure 4.3: The structure of the GraphQL API communicating with the REST API.

Chapter 5

Evaluation

Now that the test cases and the high level technical implementation details of the experiment have been described it is time to evaluate how successful the conducted experiment was by first going through the results of the experiment, and then analyzing them based on the criteria defined in Chapter 3, and finally talk about the challenges that were encountered during the execution of the experiment. The results and the analysis of various aspects of the experiment are split into two sections where the first section introduces the results of the numerical performance measurements of the experiment visualized with graphs, and the second section talks about the qualitative characteristics of GraphQL and REST in the context of the experiment and in a general manner with respect to complexity and maintainability.

5.1 Performance

As stated in Chapter 3 the recorded performance metrics were the number of network requests made, and the total duration of these network requests, the payload size of the responses, and the data utilization percentage of the response data. For each REST related test case: a recipe without sub-recipes, a recipe with sub-recipes, and a recipe with sub-recipes combined with field filtering the results of the measured payload sizes are shown in Figures 5.1 - 5.3 and results of the measured network times are shown in Figures 5.4 - 5.6. The results for the GraphQL test cases are not separately visualized but instead they are combined with the aggregated results of the REST test cases to gain a more useful perspective of the results. Aggregating the REST results is done by simply summing the response payload sizes together and calculating the average data utilization percentage. Also, the total network duration for each REST test case is presented in Figure 5.8 accompanied with

the network duration results of the GraphQL test cases. The total network duration is calculated from the start of the first request to the fulfillment of the last request.

The first test case to inspect is the fetching of a recipe without any sub-recipes from the implemented REST API which represents a simple data fetching task in the Bakery Service as described in Chapter 4. The Figure 5.1 shows the response payload size and the data utilization percentage for each resource of the recipe. Without including any sub-recipes the recipe contains in total seven resources that were separately fetched from the REST API. The estimated response payload sizes for the full resource without any field filtering ranges from 250 bytes to almost two kilobytes based on the requested resource. For all resources the utilization percentage is extremely low ranging from five percentage to twelve percentage which reflects the harsh reality of modern single page applications where the amount of data requested overshoots the actual usage of that data. This kind of behaviour was defined earlier as data over-fetching. The Figure 5.2, showing the results for fetching a recipe with sub-recipes from the REST API, reveals the issue of data over-fetching even more clearly than Figure 5.1 since the amount of requested resources is almost twice the number of resources in the first test case due to the included sub-recipes. So, the severity of data over-fetching increases with the number of requested resources. However, by having a mechanism such as field filtering implemented in the REST API it is possible to mitigate data over-fetching all together. Figure 5.3 shows how the size of the response payload was minimized to include only the necessary data achieving a full one hundred percent data utilization score for all resources when fetching a recipe with sub-recipes from the REST API.

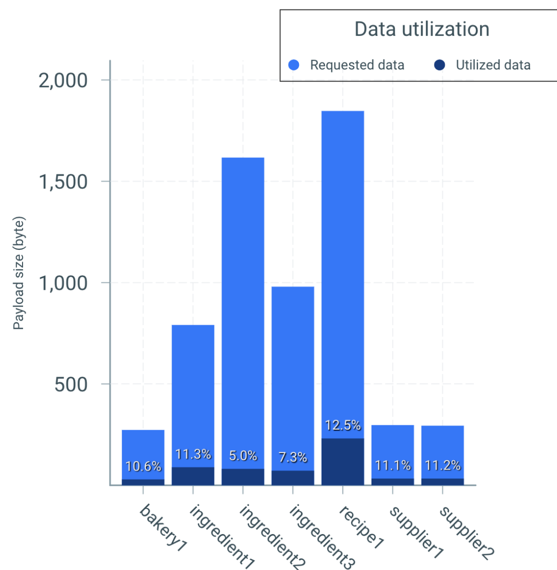

Figure 5.1: REST response payload sizes and utilization (no sub-recipes)

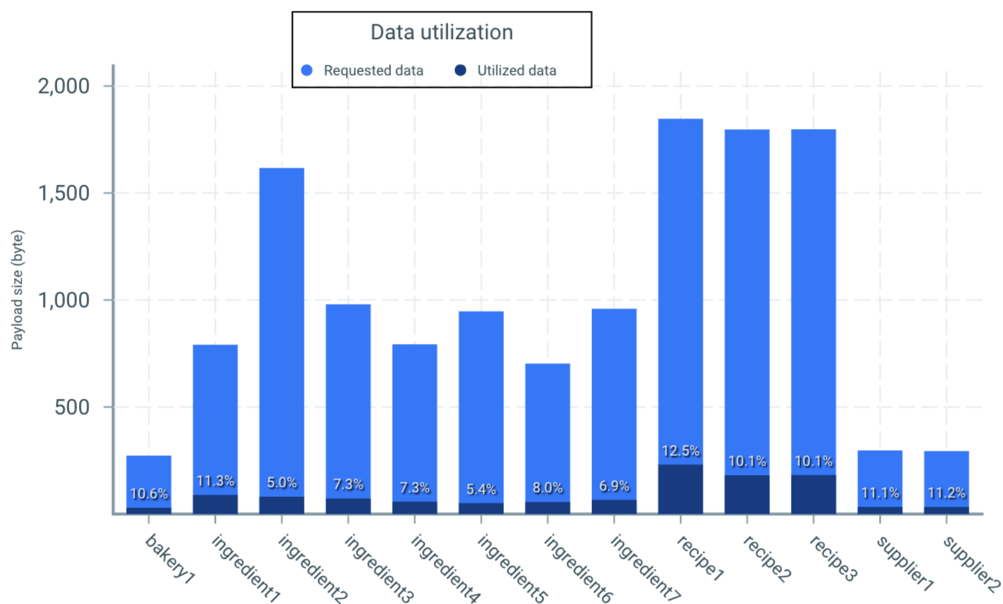

Figure 5.2: REST response payload sizes and utilization (with sub-recipes)

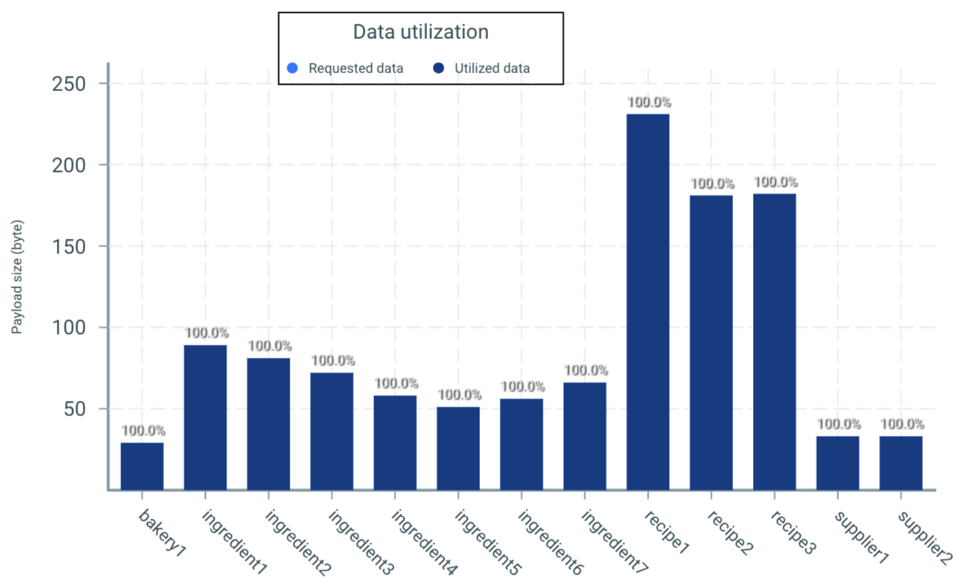

Figure 5.3: REST response payload sizes and utilization (with sub-recipes and field filtering)

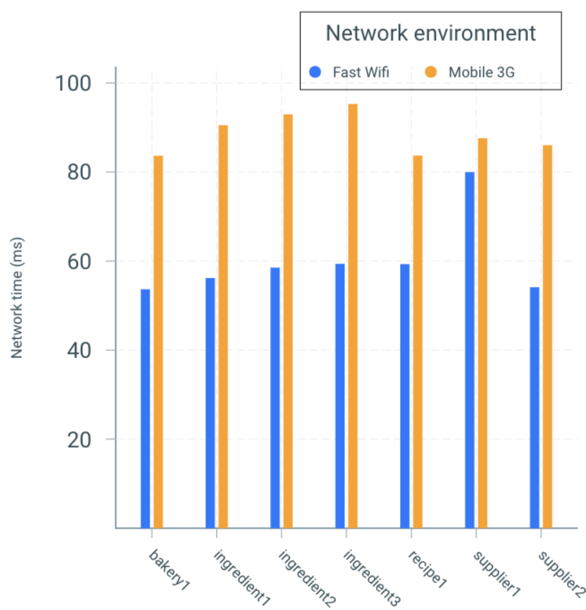

Figure 5.4: REST network duration (no sub-recipes)

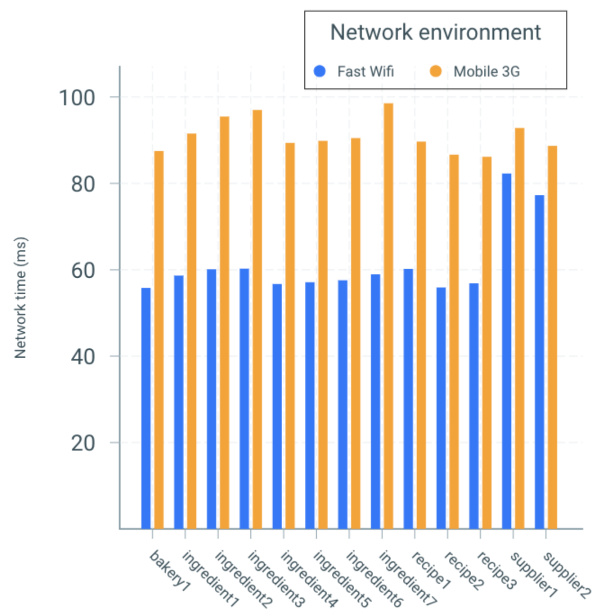

Figure 5.5: REST network duration (with sub-recipes)

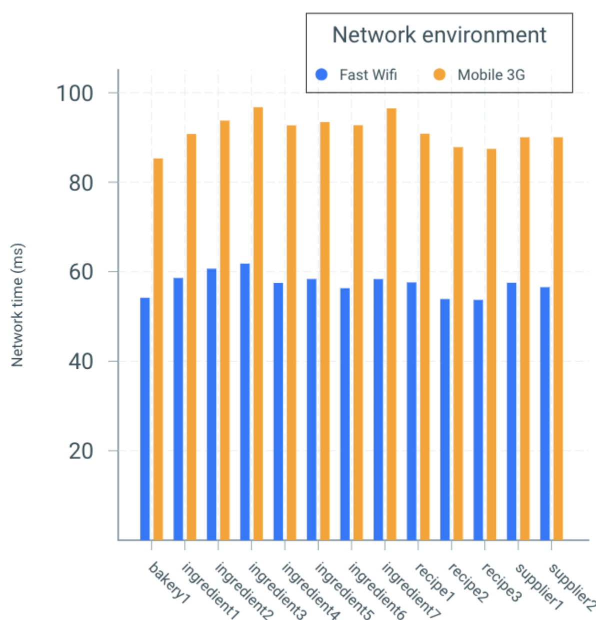

Figure 5.6: REST network duration (with sub-recipes and field filtering)

Network related measurements were conducted in two different network environments: fast WiFi and mobile 3G in order to get an accurate view how the implementations perform in a realistic environments where the Bakery Service would be used in practice. Figures 5.4 - 5.6 show the individual network round-trip times for each resource of a recipe in various test cases. In the REST implementation some resources were fetched in parallel, such as the ingredients of the recipe, to maximize the fetching efficiency. Most modern browsers can have up to six simultaneous open connections [24] which explains the spread between the different network times. Some resources are fetched in parallel while some resources are put in a queue to wait for a free connection. It is also possible to fetch all resources sequentially which would result into a more balanced graph, however, by fetching resources in parallel whenever possible the aggregated network time will be noticeably less than the aggregated network time when fetching resources in sequence. It is important to notice that the possibility of parallel fetching depends on the data hierarchy, so for example, it is not possible to fetch the bakery of a recipe at the same time as the recipe itself since in order to fetch the bakery it's identifier has to be first known. Comparing Figures 5.5 and 5.6 shows that the effect of field filtering has no substantial impact, like it had on the payload data utilization percentage, on the network duration when fetching

individual resources. So, even though the client is fetching less data per request it still has to make the same number of requests to receive all the necessary data it needs.

By only looking at the results of each resource separately it is not possible to see the big picture of the experiment results which is why Figures 5.7 and 5.8 show the aggregated results for data and network results for both REST and GraphQL. In Figure 5.7 it can be seen that GraphQL has the most consistent data utilization percentage, around 65%, for all test cases. The reason why GraphQL does not have full 100% data utilization, like REST with field filtering has, is because the response payload has some additional meta fields that are needed by the client-side library that manages the state of the GraphQL queries and their data. The remaining REST test cases with and without sub-recipes have a very low data utilization percentage meaning that almost 90% percentage of the received data is irrelevant. The data utilization percentage for the aggregated payloads for the field filtering case is 100% as expected based on the earlier analysis.

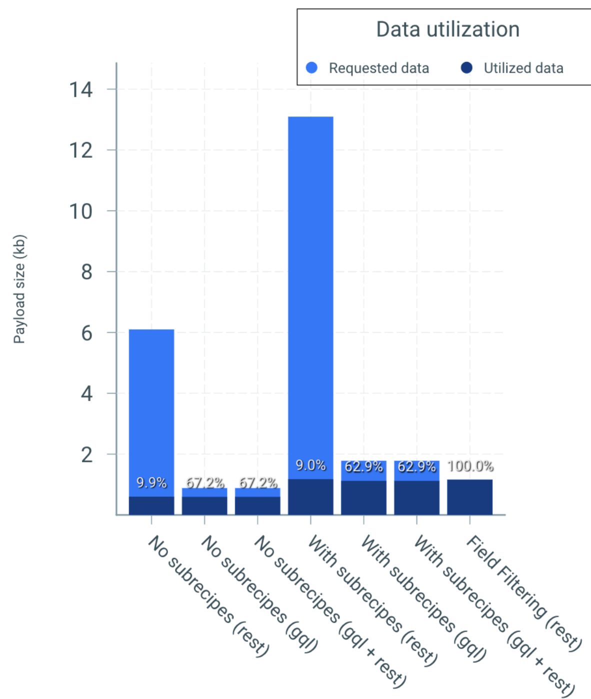

Figure 5.7: REST (aggregated) and GraphQL response payload sizes.

The Figure 5.8 for the total network duration results clearly shows the major difference between idiomatic REST and GraphQL. Firstly, for all test cases without sub-recipes the GraphQL implementation is more than twice faster than the REST implementation. Secondly, when it comes to recipes with sub-recipes the difference is even larger having the GraphQL implementation be from six to almost ten times faster than the REST implementation. Even on mobile 3G network environment the data is loaded under 200 milliseconds in the slowest test case where the GraphQL API uses the REST API to resolve the fields of the query. The difference between WiFi and mobile 3G is much more noticeable for REST than it is for GraphQL since GraphQL only needs one HTTP request to query the data where as REST has to make multiple HTTP requests to achieve the same result. On fast connections the REST API might be able to respond quickly enough to ensure a satisfying user experience but on slow connections the loading times will increase dramatically degrading the user experience.

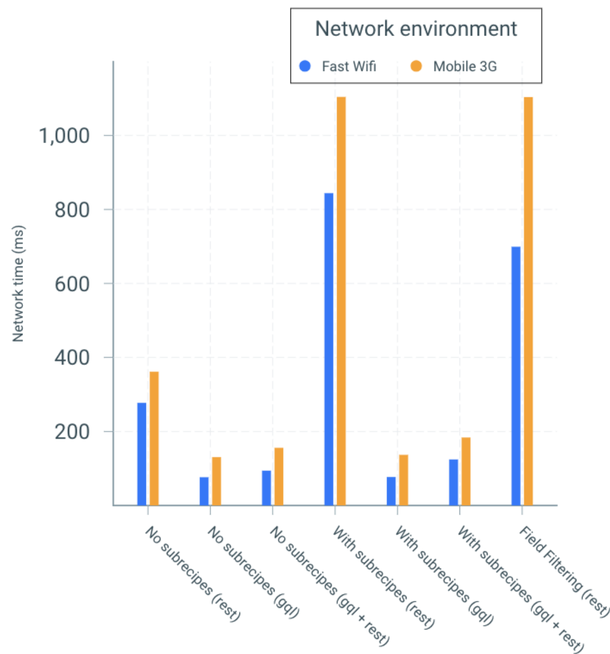

Figure 5.8: REST (aggregated) and GraphQL network duration.

5.2 Complexity and Maintainability

Designing robust, flexible, and efficient REST APIs is a challenging task which is why design specifications such as JSON API [26] have been published to help and guide developers. Following these kind of specifications is however often difficult and cumbersome because of a lack of official tooling around the specification forcing developers to either lean on unofficial, and many times untested or unmaintained, libraries or to create their own specification compliant tooling. In the case of the Bakery Service, following the JSON API specification enabled highly flexible and efficient data fetching between the client and the server but created a massive amount of technical complexity from various custom tools and utilities around the JSON API specification. Additionally, the complexity from the specification leaked into the database layer which was responsible for parsing and evaluating the requests from the client with respect to the data fulfillment. Also, since the JSON API specification allows both filtering of the response fields and inclusion of any nested resources to the response the database queries had to take these facts into account when building the SQL query strings. Having the data resolution logic in the database layer ensured optimized data access and excellent performance but lead to extremely complex and verbose SQL queries that are burdensome to maintain.

An alternative approach to the flexible but complex API design, driven by the adopted specification, is to create simple and idiomatic endpoints such as the endpoints in the conducted experiment or to create custom endpoints that satisfy the specific needs of the client. Multiple simple endpoints require the client to make multiple HTTP request to fetch the required data which causes performance problems as discussed earlier. With custom endpoints the required data can be fetched via one HTTP request and the client does not need to provide any parameters for field filtering or nested resource inclusion reducing the technical complexity on the client-side. However, maintaining and scaling custom endpoints is problematic since over time more and more API endpoints will have to be implemented when new client-side functionality is added to the application. Furthermore, the opposite case where the client removes some functionality from the application will cause over-fetching until the server-side implementation for the related custom endpoint is updated [25]. Also, there can be multiple clients for different environments such as web, mobile, and TV which might require their own custom endpoints since the data is represented differently in each environment.

On the contrary to REST, any GraphQL backend only has one endpoint that represents the whole API. Also, all data provided by the API and ways

to alter it is declared in a strongly typed schema promoting an API design that reduces tight coupling between the client and the server since the server does not have to know the data requirements of the client. The client is able request any data present in the schema via custom queries that conform to the data requirements of that client. In most cases any data that the client needs for any specific use case can be fetched via a single HTTP request. Additionally, the complexity of data merging and filtering is hidden from the developer so there is no need to implement custom tools or complex data access functions that would hinder the maintainability of the application. Thus, GraphQL enables increased efficiency, flexibility, and maintainability compared to REST APIs [40].

Maintaining an API comprises of additions, deletions, or improvements of functionality in the endpoints or the schema of the API over time. Handling these modifications in a way that the clients of the API stay functional while the quality of the code remains high is a difficult task. Various versioning approaches can be used to gracefully modify existing APIs. In the case of REST a common approach is to provide multiple versions of the same API under different URI prefixes, such as **/api/v1** and **/api/v2**, or declare the desired version in the request headers [19]. In order to avoid explicit versioning the developers of an API can adopt an approach where new features, such as new fields to resources or new endpoints, are always only added and never removed to make sure no existing clients break due to changes. Without advanced field filtering methods this approach clearly leads to over-fetching since the client does not know that new fields have been added and the server has no way of knowing which fields the client actually wants. On the other hand with GraphQL versioning is not necessarily needed since the client is always responsible of querying exactly the data it needs so the server has comprehensive information about field level usage. Thus, the server is able to add deprecation warnings on a field level and remove the field from schema when the usage of it is low enough [46]. All in all, for the Bakery Service versioning is not an important topic since the API is used only by one client and all parts of the service are controlled by one entity, but, in the general case versioning can be an important part of an API design.

Other important qualities not yet discussed that make the development and the maintenance of an API easier include documentation and introspection. For both REST and GraphQL it is possible to automatically generate documentation and provide an introspection playground for the given API. With REST this process requires external tools, such as Swagger[1], that require defining descriptions of the API endpoints in inline comments or in

[1] `https://swagger.io/`

separate files. Having to change the API description every time something is modified is a laborious task and could be considered as a maintainability burden. GraphQL however, has built-in introspection with extensive documentation about the schema [13]. Having this kind of development playground available out of the box reduces development time and ensures that the documentation stays up to date when the implementation of the API changes.

5.3 Challenges

While conducting the complexity measurements with the *complexity-report* tool, mentioned in the previous chapter, it became evident that the measurements obtained did not reflect the complexity of the system in a valuable way. First of all, the size of the server-side implementations for both GraphQL and REST were fairly small compared to real-world production level implementations which skewed the measurements to a great extent making them irrelevant to this study. Avoiding this issue was not possible since implementing a large scale system similar to the existing Bakery Service was not feasible for this thesis. Secondly, the complexity measures in general were deemed as non-decisive metrics for the comparison of GraphQL and REST since they mostly reflected the subjective technical decisions of the implementer instead of innate characteristics of the technologies under inspection. So, in the end it was decided that the numeric evaluation of various complexity metrics should be left out of the evaluation of the implemented experiment.

Another challenge encountered early on during the implementation process was the question of how the server-side data should be stored and accessed. The real Bakery Service uses SQL database to store all the data in the system. Due to the nature of the current REST API implementation, accessing that data involves overly complex SQL queries that has been identified as critical maintainability issue by Taito United. Implementing a similar data management system for the experiment was not reasonable in the scope of this study which is why the technical decision to not use a real database was made in favor of static JSON data representing structurally the same data as that is stored in the Bakery Service. Furthermore, by adopting GraphQL on the server-side it is possible to reduce the complexity of the data access methods by delegating the data resolution to GraphQL.

Chapter 6

Discussion

Having evaluated the performance metrics of the conducted experiment and analyzed the complexity and the maintainability characteristics of both REST and GraphQL it is now time to sum up the major strengths and weaknesses of both technologies, and then propose a path forward for the Bakery Service, and finally discuss about future work.

Firstly, the numerical measurements illuminated some performance characteristics of REST and GraphQL showing that GraphQL performed up to six times faster in mobile 3G network and almost ten times faster in a WiFi network. Even when the server-side data resolution was done by fetching the data from the REST API, the GraphQL implementation managed to load the data under 200 milliseconds in all test cases while similar REST test cases took over one second to resolve. Additionally, the amount of data requested by the client was considerably less with GraphQL in all test cases except for REST with field filtering which was the only one that managed to utilize all of the data from the server. Even though field filtering for REST utilized the data perfectly it still had to make multiple API request to various endpoints to receive all the data needed. A highly positive finding was that GraphQL was able resolve data almost equally fast with local data sources and with the REST API as a data source. This finding makes it feasible to incrementally replace parts of the current Bakery Service REST API endpoints with GraphQL API in a way that the current REST API can be utilized as the data source.

Secondly, the complexity of both technologies was attempted to measure but the results of this process turned out to be non-relevant for this study. However, analyzing the non-measurable characteristics related to complexity and maintainability gave a deeper insight into GraphQL and REST as technical solutions for API development. REST was found to be simpler than GraphQL for smaller applications that do not have complex data needs

that involve field level filtering and nesting of sub-resources. However, many modern applications depend on a large amount of data being fetched from APIs, so in order to provide these kinds of high level features for REST APIs various custom endpoints would have to be implemented. Custom REST endpoints then again were found to be a source of complexity and a common cause of maintenance issues in general and for the Bakery Service. GraphQL on the other hand hides most of the complexity in the technology itself and shifts the specification of the data requirements to the client while providing only one endpoint for the whole API. A GraphQL schema defines all the possible ways how clients are able to interact with the API enabling an enhanced playground for fast development with built-in introspection and automatically generated documentation which is guaranteed to stay up to date when any API related code is updated. While a solid support for versioning, or more specifically in the case of GraphQL the possibility to avoid it, was not found to be of importance for the Bakery Service, it was still regarded as a supporting factor for GraphQL.

So, based on the discussed advantages it can be proposed that GraphQL should be adopted on the server-side for the Bakery Service. However, the level of adoption is not entirely definite based on the study conducted in this thesis since certain aspects of interest were left out of the experiment due to scope management. The implementation was scoped to exclude custom endpoints for maintainability reasons, usage of a real database for being non-relevant for the study, and some recipe related resources, such as nutrients and allergens, which are only used in special occasions in the Bakery Service. But, based on the experiment the path of least resistance and highest utility would be to build a GraphQL API on top of the existing REST API and incrementally migrate features from the old REST implementation to the new GraphQL implementation. Additionally, the infrastructure needed for efficient communication between two server-side endpoints already exists which lessens the initial investment required for a GraphQL API using a REST API as a data source. Moreover, migrating the most data heavy endpoints of the current API first would make it straightforward to validate the technology right away without investing too much effort in to the migration. Also, since the data layer of the current Bakery Service is overly coupled with the adopted JSON API specification, causing notable maintenance complications, it should be refactored to support the field level data resolution of GraphQL. Moving the filtering and nesting logic out of the data layer and delegating it to the GraphQL libraries should notably simplify the code complexity on the server-side alleviating the current maintenance issues.

Finally, the conducted experiment was not without its shortcomings so some future work would be appropriate to deepen the knowledge on GraphQL

as a potential replacement for REST and solidify the findings for a more comprehensive proposal. Future work could include adding a cache layer to the experiment implementation in order to see how both technologies perform with and without a cache. Caching is a common performance improvement technique used in many modern web applications to reduce the time for fetching data from the server. For REST APIs caching can be done via the endpoint URLs by using simple HTTP caching mechanisms [34]. Since GraphQL API only has one URL and every request from the client is done via a HTTP POST method it is not possible to use generic HTTP caching mechanisms for GraphQL queries. However, most common GraphQL client-side libraries such as Apollo, which was used in the experiment, provide a built-in caching mechanism based on browser offline storage [41]. More advanced caching can be achieved by utilizing globally unique object IDs within queries [41]. Today's single page applications rely heavily on data fetching from APIs so having an efficient caching layer in place can improve the loading times for requests immensely especially in slow network conditions.

Another potential improvement to the experiment is to add automatic persistent queries[1] to the GraphQL implementation. GraphQL queries that are sent in the body of a HTTP POST request can be relatively large, up to ten kilobytes [23], which can impose a performance overhead for client-server communication. Automatic query persistence involves hashing the query string before it is sent to the server and using the generated hash for any following requests to reduce the request body size [23]. In the conducted experiment the query strings used amounted to around 500 bytes which is negligible but in real-world usage of the Bakery Service the size of the query strings could be much larger.

Even though GraphQL enables client applications to describe their data needs with a rich query language that is capable of fetching all the required data in one request the trend for many modern web frameworks is to split the application into small components that are themselves responsible of fetching the data they need. Having multiple components with multiple queries can lead into a situation where the application is making too many network requests causing a heavy load on the server. Query batching on the client-side is a process where multiple queries are grouped together into a single list of queries that is sent via a single HTTP request. The batching process is usually implemented with a timing threshold in a way that when a new query is made it will be withheld until the timer ends and if other queries are made during that time they will be grouped together to be sent via one request. Query batching can however cause performance issues if one

[1] `https://github.com/apollographql/apollo-link-persisted-queries`

of the grouped queries takes a long time to execute causing the client to wait possible longer for the results of the other queries than it would have had to if the queries would have been made individually. [8]

Also, the security aspects of both technologies should be studied to better understand the possible attack vectors when using these technologies in production. The security of REST is more known due to it being almost twenty years old but the security of GraphQL still needs more research. The most apparent security problem with GraphQL is the fact that clients are able to send large queries to the API that can take too much time and resources to process. However, there exists various strategies and tools to counter these kinds of attacks against the server. The first and the simplest strategy is to add a timeout to the execution of each query and if that timeout is passed the execution will be terminated [33]. Another strategy is to calculate the complexity of the query by assigning complexity costs to the fields of the query and disallow queries that have too high total query complexity [33]. Many other strategies exists and investigating them and the ones mentioned would give valuable insight on how to achieve robust security for the Bakery Service.

In summary, analyzing an experiment that simulates a real-world implementation cannot give an absolute conclusion on whether GraphQL is a better solution for building APIs than REST, but it can give a well-educated conclusion that can be used to initiate prototyping with GraphQL in order to validate it in a production environment and to perform additional research.

Chapter 7

Conclusions

Creating applications today involve much more attention to the performance of data fetching than before. Many applications dynamically fetch the data they need on the client-side instead of rendering the whole page on the server-side with the data and then serving the static page to the client. Additional trouble is added when multiple network conditions and countless different devices have to be taken into account. At the moment, a common and a popular way to implement APIs is to use REST which promotes defining idiomatic endpoints for the data resources. However, it was found that following the principles of REST can easily lead to performance issues where the client is either over-fetching or under-fetching data from the API. These issues are then usually circumvented by implementing custom endpoints that return only the specific data that is needed by the client. Custom endpoints were, however, found to be a source of complexity and maintainability problems so adopting them in a large scale manner is not desirable. Alternatives to REST have been implemented in recent years from which Falcor and GraphQL were considered in this study and GraphQL was chosen for closer inspection.

The premise of this study was to determine the major advantages and disadvantages of both GraphQL and REST, understand which one performs better with hierarchical data, such as the one used in the Bakery Service, and to conclude which one is more maintainable and less complex to implement. The study was conducted as a case study for a Bakery Service in order to find potential improved alternatives to its current REST API, focusing on GraphQL. The main pain points of the current implementation included complex custom made data handling and state management on the client-side, convoluted data layer logic tightly coupled with the adopted JSON API specification, various custom endpoints that satisfy only specific client needs, and lastly, as a combination of many factors, difficulties in maintaining the API implementation. So, the objective of this thesis was to investigate

whether GraphQL could be utilized as complete or as a partial replacement for the current REST API in order to solve the mentioned issues.

An experiment was conducted to compare REST and GraphQL after researching and selecting the tools and technologies suggested by recent literature related to both technologies. The methods used to compare these technologies included performance analysis of the experiment implementations and qualitative analysis of complexity and maintainability. The experiment was implemented in a context of a single page inside the Bakery Service which was identified to be data intensive and in need for data fetching improvements. All the implemented variations of REST and GraphQL APIs were tested in both fast and slow network conditions measuring the number of network request and the total times of all requests made to the API in conjunction with the response data sizes and utilization percentages inside the client application.

The results of the conducted experiment showed that GraphQL performed generally better than REST in all test cases in both network conditions. Although the highest data utilization percentage was achieved with field filtering for REST, the corresponding total network time performed considerably worse compared to GraphQL. In all test cases the total network time for GraphQL requests were under 200 milliseconds whereas for REST the slowest times measured were over one second. So, on average GraphQL was able to fetch less data quicker than REST making it more suitable for fetching data with complex hierarchical structures. Qualitative analysis on both technologies showed that REST was lacking built-in support for field level filtering and nesting of sub-resources which lead into complex custom solutions in the Bakery Service that hindered the maintainability of the API implementation. GraphQL however, provided built-in way to select the fields of interest and include any sub-resources via GraphQL queries on the client-side moving the responsibility of defining the data requirements to the client. A strongly typed schema was used for declaring all the possible ways to interact with the GraphQL API enabling high confidence for using the API via query validation in conjunction with enhanced developer experience via introspection and automatically generated documentation from the schema.

In summary, this thesis was focused on advanced data fetching solutions for modern applications that dynamically fulfill their data needs during runtime. Some challenges were encountered during the experiment phase of the study when the complexity metrics that were tried to be numerically measured were determined to be irrelevant for this study. Future work for this study includes concepts such as caching, automatic persistent queries, query batching, and security. Also, one aspect that was not studied was the concept of mutating data in REST and GraphQL. So, in order to have a full picture of

the inspected technologies some future work is needed to study the mutation operations available in REST and GraphQL. However, the findings from the conducted experiment provide valuable cues for potential improvements in the API implementation illuminating a path forward for the Bakery Service.

Bibliography

[1] Alex Banks, E. P. *Learning GraphQL: Declarative Data Fetching for Modern Web Apps.* O'Reilly Media, 2018.

[2] Antinyan, V., Staron, M., and Sandberg, A. Evaluating code complexity triggers, use of complexity measures and the influence of code complexity on maintenance time. *Empirical Software Engineering 22*, 6 (2017), 3057–3087.

[3] Baer, E. A GraphQL Primer: The Evolution Of API Design (Part 1). https://www.smashingmagazine.com/2018/01/graphql-primer-new-api-part-1/, Jan 2018. Accessed: 2018-5-19.

[4] Baer, E. A GraphQL Primer: The Evolution Of API Design (Part 2). https://www.smashingmagazine.com/2018/01/graphql-primer-new-api-part-2/, Jan 2018. Accessed: 2018-5-19.

[5] Box, D., Ehnebuske, D., Kakivaya, G., Layman, A., Mendelsohn, N., Nielsen, H., Thatte, S., and Winer, D. Simple Object Access Protocol (SOAP) 1.1. Tech. rep., W3C, May 2000. https://www.w3.org/TR/2000/NOTE-SOAP-20000508/.

[6] Byron, L. GraphQL: A data query language. https://code.fb.com/core-data/graphql-a-data-query-language/. Accessed: 2018-12-17.

[7] Cederlund, M. Performance of frameworks for declarative data fetching: an evaluation of falcor and relay+ graphql. Master's thesis, KTH Royal Institute of Technology, 2016.

[8] Dawkins, J. Batching Client GraphQL Queries. https://blog.apollographql.com/batching-client-graphql-queries-a685f5bcd41b. Accessed: 2018-12-07.

[9] Eeda, N. Rendering real-time dashboards using a graphql-based ui architecture. Master's thesis, The University of Western Ontario, 2017.

[10] EUROPEAN COMMISSION. EU and Brazil to work together on 5G mobile technology. `http://europa.eu/rapid/press-release_IP-16-382_en.htm`, 2016. Accessed: 2018-01-26.

[11] FACEBOOK. Caching. `http://graphql.org/learn/caching/`. Accessed: 2018-12-01.

[12] FACEBOOK. GraphQL Best Practices. `http://graphql.org/learn/best-practices/`. Accessed: 2018-10-15.

[13] FACEBOOK. Introspection. `https://graphql.org/learn/introspection/`. Accessed: 2018-12-01.

[14] FACEBOOK. Queries and Mutations. `http://graphql.org/learn/queries/`. Accessed: 2018-4-10.

[15] FACEBOOK. Schemas and Types. `http://graphql.org/learn/schema/`. Accessed: 2018-4-12.

[16] FACEBOOK. Serving over HTTP. `https://graphql.org/learn/serving-over-http/`. Accessed: 2018-12-09.

[17] FACEBOOK. GraphQL. Tech. rep., Facebook, Inc, Oct 2018.

[18] FIELDING, R. T. *Architectural Styles and the Design of Network-based Software Architectures.* PhD thesis, University of California, Irvine, 2000. `http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm`.

[19] GIESSLER, P., GEBHART, M., SARANCIN, D., STEINEGGER, R., AND ABECK, S. Best practices for the design of restful web services. In *International Conferences of Software Advances (ICSEA)* (2015), pp. 392–397.

[20] HALSTEAD, M. H., ET AL. *Elements of Software Science (Operating and programming systems series).* Elsevier Science Inc., New York, NY, 1977.

[21] HARTIG, O., AND PÉREZ, J. Semantics and complexity of graphql. In *Proceedings of the 2018 World Wide Web Conference* (Republic and Canton of Geneva, Switzerland, 2018), WWW '18, International World Wide Web Conferences Steering Committee, pp. 1155–1164.

[22] Hernandez-Mendez, A., Scholz, N., and Matthes, F. A Model-driven Approach for Generating RESTful Web Services in Single-Page Applications. In *MODELSWARD* (2018), pp. 480–487.

[23] Hingston, T. Improve GraphQL Performance with Automatic Persisted Queries. https://blog.apollographql.com/improve-graphql-performance-with-automatic-persisted-queries-c31d27b8e6ea. Accessed: 2018-12-07.

[24] Hogan, L. C. *Designing for Performance: Weighing Aesthetics and Speed*. O'Reilly Media, Inc., 2014.

[25] John Resig, L. S.-R. The GraphQL Guide. https://graphql.guide/, 2018.

[26] Klabnik, S., Katz, Y., Gebhardt, D., Kellen, T., and Resnick, E. JSON API Specification v1.0 (Archived Copy). http://jsonapi.org/format/1.0/. Accessed: 2018-01-23.

[27] Masse, M. *REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces*. O'Reilly Media, Inc., 2011.

[28] McCabe, T. J. A complexity measure. *IEEE Transactions on software Engineering*, 4 (1976), 308–320.

[29] Netflix, I. Falcor, A JavaScript library for efficient data fetching. https://netflix.github.io/falcor/. Accessed: 2018-02-05.

[30] Netflix, I. Why Falcor? https://netflix.github.io/falcor/starter/why-falcor.html. Accessed: 2018-12-17.

[31] npm. This year in JavaScript: 2018 in review and npm's predictions for 2019. https://blog.npmjs.org/post/180868064080/this-year-in-javascript-2018-in-review-and-npms. Accessed: 2018-12-09.

[32] OpenSignal. The State of LTE (November 2016). https://opensignal.com/reports/2016/11/state-of-lte. Accessed: 2018-01-23.

[33] Prisma. Security. https://www.howtographql.com/advanced/4-security/. Accessed: 2018-12-07.

[34] Riva, C., and Laitkorpi, M. Designing web-based mobile services with REST. In *International Conference on Service-Oriented Computing* (2007), Springer, pp. 439–450.

[35] Roy T. Fielding, Richard N. Taylor, J. R. E. M. M. G. J. W. R. K., and Oreizy, P. Reflections on the REST Architectural Style and “Principled Design of the Modern Web Architecture”. In *Proceedings of 2017 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering* (2017), pp. 4–14.

[36] Seric, J. GraphQL multipart request specification. `https://github.com/jaydenseric/graphql-multipart-request-spec`. Accessed: 2018-12-09.

[37] Smith, P. G. *Professional Website Performance: Optimizing the Front-End and Back-End*. John Wiley & Sons, 2012.

[38] Sun, H., Bonetta, D., Humer, C., and Binder, W. Efficient dynamic analysis for node.js. In *Proceedings of the 27th International Conference on Compiler Construction* (New York, NY, USA, 2018), CC 2018, ACM, pp. 196–206.

[39] Ueda, T., Nakaike, T., and Ohara, M. Workload characterization for microservices. In *Workload Characterization (IISWC), 2016 IEEE International Symposium on* (2016), IEEE, pp. 1–10.

[40] Vázquez-Ingelmo, A., Cruz-Benito, J., and García-Peñalvo, F. J. Improving the OEEU's Data-driven Technological Ecosystem's Interoperability with GraphQL. In *Proceedings of the 5th International Conference on Technological Ecosystems for Enhancing Multiculturality* (New York, NY, USA, 2017), TEEM 2017, ACM, pp. 89:1–89:8.

[41] Vogel, M., Weber, S., and Zirpins, C. Experiences on Migrating RESTful Web Services to GraphQL. In *International Conference on Service-Oriented Computing* (2017), Springer, pp. 283–295.

[42] w3 Techs. Usage of content management systems for websites. `https://w3techs.com/technologies/overview/content_management/all`. Accessed: 2018-02-06.

[43] W3School. JSON vs. XML. W3School tutorial, `https://www.w3schools.com/js/js_json_xml.asp`. Accessed: 2018-02-06.

[44] Warren, T. Oracle's finally killing its terrible Java browser plugin. `https://www.theverge.com/2016/1/28/10858250/oracle-java-plugin-deprecation-jdk-9`, Jan 2016. Accessed: 2018-02-06.

[45] WEYL, E. *Mobile HTML5: Using the Latest Today.* O'Reilly Media, Inc., 2013.

[46] WIERUCH, R. The Road to GraphQL. `https://www.robinwieruch.de/the-road-to-graphql-book/`, 2018.

[47] WORDPRESS. REST API Handbook. `https://developer.wordpress.org/rest-api/`. Accessed: 2018-5-06.