



DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Master's Thesis
for the Attainment of the Degree
Master of Science
at the TUM School of Management
of the Technische Universität München

Identification and Evaluation of a Process for Transitioning from REST APIs to GraphQL APIs in the Context of Microservices Architecture

Author:	Berke Gözneli Heinrich-Wieland-Str. 178 81735 München Matriculation Number 03696708
Course of Study:	Management & Technology
Supervisor:	Prof. Dr. Florian Matthes Chair for Software Engineering for Business Information Systems
Advisor:	Gloria Bondel, M.Sc.
Submission Date:	August 27, 2020



Declaration of Authorship

I hereby declare that the thesis submitted is my own unaided work. All direct or indirect sources used are acknowledged as references.

I am aware that the thesis in digital form can be examined for the use of unauthorized aid and in order to determine whether the thesis as a whole or parts incorporated in it may be deemed as plagiarism. For the comparison of my work with existing sources I agree that it shall be entered in a database where it shall also remain after examination, to enable comparison with future theses submitted. Further rights of reproduction and usage, however, are not granted here.

This paper was not previously presented to another examination board and has not been published.

Munich, August 27, 2020

Berke Gözneli

A handwritten signature in black ink, appearing to be 'Berke Gözneli', with a long horizontal stroke extending to the right.

Acknowledgments

I would first like to recognize my advisor M. Sc. Gloria Bondel who guided me throughout all the phases of this thesis. Her feedback was invaluable, and I would like to thank her for it.

I would also like to thank my supervisor Prof. Dr. Florian Matthes for giving me the opportunity to start this research for Software Engineering for Business Information Systems (SEBIS) chair at the TUM Department of Informatics.

A special thank you goes to the participants of my case study for their valuable contribution to my research.

Last, but not least, I would like to thank all my friends for making my studies fun, and my parents, Ayla and Süreyya Gözneli for their support throughout my whole life, without which I would not have been able to be where I am today.

Abstract

APIs are the driving hubs of the modern service providing software architectures. Over the years, various API technologies have been developed to serve the needs of data sharing, with REST becoming the de facto standard for microservice-based architectures. REST serves the state-of-the-art needs of information sharing, however as the needs of the clients and the complexity of the service providing backend architectures grow, the question of how to optimize this client-provider relationship and whether this could be achieved by migrating the RESTful APIs to new technologies, such as GraphQL arises.

GraphQL is a query language and specification for APIs proposed by Facebook which was officially released in 2015. It is client empowering, as in, it was developed from a front-end point of view to provide a flexible syntax and system for enabling a more convenient way of data querying over network calls. It has many benefits; such as supporting API growth by removing the need of versioning, and eliminating over/under-fetching by allowing the users to only get the data they want.

The potential pay-off of adapting GraphQL led to corporations trying it out in their internal systems. As the interest in GraphQL grew, methods for easing the adaptation and transitioning process emerged. The aim of this work is to provide a snapshot of the current RESTful to GraphQL transition processes, to evaluate them, and to implement such a process along with a GraphQL gateway for a microservices architecture.

Contents

Acknowledgements	v
Abstract	vii
1. Introduction	1
1.1. Motivation	1
1.2. Research Questions	1
1.3. Thesis Structure	2
2. Foundations	5
2.1. API Fundamentals	5
2.1.1. Basic Definitions of API	5
2.1.2. Types of API	5
2.1.2.1. API Types by Availability	6
2.1.2.2. API Types by Use Cases	6
2.1.2.3. API Types by Specifications/Protocols	7
2.2. Microservices Architectures	8
2.3. Change Management	9
3. GraphQL	11
3.1. Description	11
3.2. Design Principles	11
3.3. Operations	12
3.3.1. Query	12
3.3.2. Mutation	13
3.3.3. Subscription	14
3.4. Schemas	14
3.5. Types	15
3.5.1. Scalar	15
3.5.2. Enumeration Types	16
3.5.3. Lists and Non-Null	16
3.5.4. Interfaces	16
3.5.5. Union Types	16
3.5.6. Input Types	17
3.6. Introspection	17

4. Rest to GraphQL Transition	19
4.1. REST vs. GraphQL	19
4.1.1. Overfetching/Underfetching	19
4.1.2. Product Iterations and Versioning	20
4.1.3. Typing and Validation	21
4.1.4. Error handling	21
4.1.5. Performance issues with complex queries	22
4.1.6. Caching	22
4.1.7. File uploading	22
4.2. REST to GraphQL Transition	22
4.2.1. Architectural Transition	22
4.2.1.1. GraphQL server with a connected database	23
4.2.1.2. GraphQL layer that integrates existing systems	23
4.2.1.3. Hybrid approach with connected database and integra- tion of existing system	24
4.2.2. Designing a GraphQL Schema	26
5. Preliminary Expert Interviews	29
5.1. Interviews Summary	29
6. Implementation	31
6.1. Architectural Overview	31
6.2. Development of the Prototype	34
6.2.1. Choosing a Software Library: Apollo-GraphQL	34
6.2.2. Setting Up the GraphQL Server	34
6.2.3. Designing the GraphQL Schema	34
6.2.4. Defining the Data Sources	35
6.2.5. Defining the Resolver Functions	35
6.3. Implementation Challenges	37
7. The Case Study	39
7.1. System Usability Survey Results	39
7.2. Interviews Analysis	40
7.2.1. Identified challenges during the transition	40
7.2.2. Areas where using GraphQL is more convenient than using REST	40
7.2.3. Areas where using REST is more convenient than using GraphQL	41
7.2.4. Acceptance towards GraphQL	41
8. Discussion	43
8.1. Key Findings	43
8.1.1. Technical Findings	43
8.1.2. Organizational Findings	43
8.2. Limitations	44

9. Conclusion	47
9.1. Summary	47
9.2. Future Work	48
A. Evaluation Instructions	49
A.1. Initial Survey	49
A.2. Getting Familiar With The Approaches	49
A.3. Task Description	49
A.4. System Usability Scale	50
A.5. Interview Questions	51
B. Transcripts of the Case Study Interviews	53
B.1. Interview with Participant #1 and Participant #2	53
B.2. Interview with Participant #3	56
B.3. Interview with Participant #4	57
B.4. Interview with Participant #5	59
B.5. Interview with Participant #6	61
B.6. Interview with Participant #7	63
List of Figures	67
List of Tables	69
Bibliography	71

1. Introduction

This chapter is an introduction to the thesis. It explains the motivation behind the research, describes the research questions and provides the outline of this paper.

1.1. Motivation

Nowadays, software technology is ever-changing to make programs run more efficiently and create business value for organizations. Whenever a new technology is released, it takes some years to identify and confirm the proposed values that come along with the technology. They either live up to their potential, or fail to bring the supposed breakthrough. This, of course, applies to APIs as well. A number of API technologies have been developed over the years to serve the needs of data sharing. Among them, REST has claimed a place as the de facto standard protocol for web APIs (Levin, 2015). REST serves the state-of-the-art needs of information sharing, however as the needs of the clients and the complexity of the service providing backend architectures grow, the question of how to optimize this client-provider relationship arises and whether this could be achieved by migrating the RESTful APIs to new technologies. This thesis aims to investigate such a newly emerging API technology, GraphQL, and evaluate whether it brings value or not to transition an API from REST to GraphQL.

1.2. Research Questions

The following section presents the research questions that are based on the motivation of this thesis. They provide basis for the research.

- **RQ-1: What are the existing approaches for a REST to GraphQL transition?**
This research question will be answered through literature review and a series of interviews. The main outcome of this question is the current best practices for transitioning to GraphQL. These findings will form the basis for the following research question.
- **RQ-2: What could be a process for transitioning including stakeholders, activities and artifacts?**
This research question aims to provide a process for a GraphQL transition that includes both the technical and organizational aspects. The question will be answered through a combination of literature review and experimentation. The experiment is a study case where software developers of various backgrounds are

asked to perform transition task. The outcome of this experiment will provide insight to the research question.

- **RQ-3: What are the lessons learned from a transition to GraphQL?** This research question will be answered through evaluating the results of the experiment. The results will be a combination of quantitative analysis in the form of a system usability survey (SUS) and qualitative analysis in the form of participant interviews. The outcome of this research question aims to provide practical insights and lessons learned for future REST to GraphQL transitions.

1.3. Thesis Structure

The following section provides short explanations for the chapters of the thesis.

Chapter 1: Introduction provides the motivation of the thesis and the research questions to be answered. The approach to answering these questions and the expected outcome of them are also explained.

Chapter 2: Foundations introduces the theoretical concepts that are required for a better understanding of this thesis. It describes the fundamentals of APIs in general, then delves into explaining the concept behind the most commonly used "REST" APIs. Also, the microservices-based software architecture pattern is explained here. Finally, this chapter presents software adaptation techniques from a general perspective.

Chapter 3: GraphQL provides the literature review on the technical aspects of GraphQL and explains how this new technology works.

Chapter 4: Rest to GraphQL Transition first compares REST and GraphQL with each other and identifies the advantages/disadvantages of each concept. Then, this chapter provides a literature review on the current REST to GraphQL transition approaches.

Chapter 5: Preliminary Expert Interviews presents the results of the expert interviews that were conducted before the case study of the thesis. These interview results are taken as the basis for the implementation of the GraphQL gateway and the preparation of the case study.

Chapter 6: Implementation first presents the current state of the architecture that is subject to the use case. From there, it continues with explaining the technical development process of the GraphQL gateway that is to be used in the case study. Finally it talks about the challenges faced during this development.

Chapter 7: The Case Study presents the results of the case study in the form of quantitative analysis (systems usability survey) and qualitative analysis (post experiment

interviews).

Chapter 8: Discussion provides the findings the thesis which are divided into technical and organizational categories. Finally, it gives insight to the limitations of this research.

Chapter 9: Conclusion wraps up the work done during this research and proposes ideas for future research.

2. Foundations

The following sections are dedicated to theoretical foundations that are essential understandings for the topics that is going to be discussed in this thesis. The concepts that will be explain here are API fundamentals, microservices architecture and change management.

2.1. API Fundamentals

This section will cover the fundamentals of APIs. We will present the basic definitions associated with APIs and the different types.

2.1.1. Basic Definitions of API

API stands for *Application Programming Interfaces*. They are interfaces that allow developers or systems to interact with programs, websites or parts of systems for data and information obtaining purposes (Palmieri, 2018). The Twitter and Facebook APIs are examples of commonly used APIs. They are open to developers and let them interact with Twitter's and Facebook's softwares.

APIs also allow communications between applications. From a more technical perspective, an API is "a way for two computer applications to talk to each other over a network (predominantly the Internet) using a common language that they both understand" (Jacobson, Brail, and Woods, 2012).

From a business perspective, APIs describe how to connect business processes with consumer applications or other business processes. They provide ways for businesses to leverage new markets and enables businesses to become platforms (Boyd, 2015). Specifically, APIs allow third-party developers and other partners to access the database assets of a business (Boyd, 2015).

2.1.2. Types of API

APIs are typed in different categories based on availability, use cases and specifications/protocols (Altexsoft, 2019a).

2.1.2.1. API Types by Availability

In terms of availability, APIs can be public(open), private and partner, as shown in Figure 2.1 (Boyd, 2014).

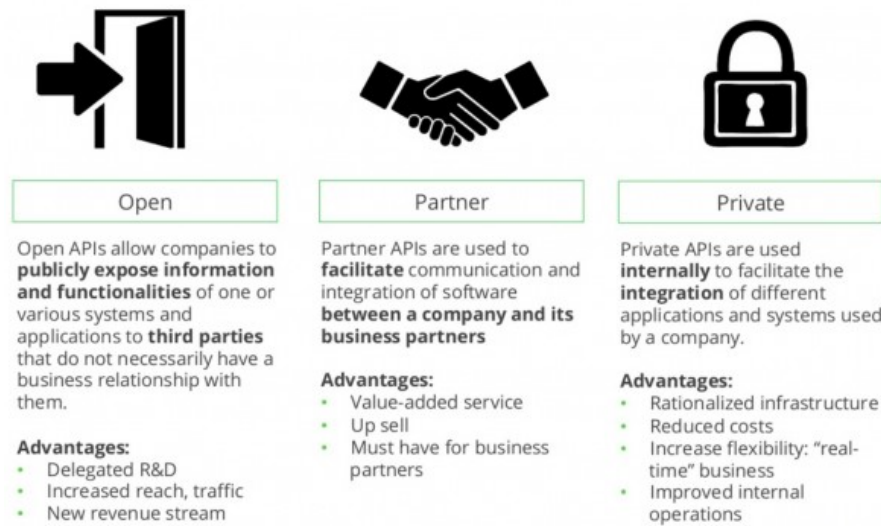


Figure 2.1.: Types of APIs by availability (Boyd, 2014).

- **Public(Open) APIs** - These type of APIs are available to any third-party developers outside the organization, and are also known as "external" or "developer-facing" APIs (Altexsoft, 2019a). The public APIs increase the brand awareness and could provide additional means of income if monetized.
- **Private APIs** - Private APIs make parts of an organization's back-end data and/or application functionality available for use by the developers within the organization (Mitra, 2015). They can significantly reduce the development time and resources required to build new systems or change existing ones (Mitra, 2015).
- **Partner APIs** - Partner APIs are hybrids of public and private APIs. These type of APIs are intended to support the application development of business partner organizations (Mitra, 2015).

2.1.2.2. API Types by Use Cases

In terms of use cases, APIs can be categorized into database APIs, operating systems APIs, remote APIs and web APIs (Altexsoft, 2019a).

- **Database APIs** are designed enable communication between the databases and the applications that access them (Altexsoft, 2019a).

- **Operating systems APIs** define how the resources and services of operating systems are used by applications that access them (Altexsoft, 2019a).
- **Remote APIs** define the rules about how the applications running on different host machines interact with each other (Altexsoft, 2019a).
- **Web APIs** are the most common APIs. They provide data and functionality transfer between web-based systems through a client-server relationship (Altexsoft, 2019a).

2.1.2.3. API Types by Specifications/Protocols

The aim of the API specifications is to standardize the exchanges between different services (Altexsoft, 2019a). This standardization allows for the systems that run on different operating systems, written with different programming languages to be able to communicate with each other with ease. There are several types of specifications:

Remote Procedure Call (RPC)

Remote Procedure Call (RPC) is a programming interface for starting procedures on remote computers (Geissler and Ostler, 2018). It is the oldest and simplest among the API protocols (Techolution, 2019). It requires one application to send one or more messages to another application to start these procedures. The recipient application in return replies, once again, by sending one or messages back to the calling application (Merrick, Allen, and Lapp, 2006).

Simple Object Access Protocol (SOAP)

Simple Object Access Protocol (SOAP) was first developed by Microsoft, and it is describe as "a lightweight protocol for exchange of information in a decentralized, distributed environment" (Box, Ehnebuske, Gopal Kakivaya, et al., 2000). It is a protocol based on XML, and is composed of three parts:

- an envelope that defines a framework for describing what is in a message and how to process it
- a set of encoding rules for expressing instances of application-defined data-types
- and a convention for representing remote procedure calls and responses (Box, Ehnebuske, Gopal Kakivaya, et al., 2000).

SOAP is mostly used to ensure high-security during the transmission data (Altexsoft, 2019a). SOAP APIs are usually used by enterprises that provide payment gateways, identity management solutions and financial services (Altexsoft, 2019a).

Representational State Transfer (REST)

Representational State Transfer (REST) was first introduced by the computer scientist Roy Fielding in the year 2000. *REST* is a software architectural style and design pattern for APIs (Avraham, 2017). By Fielding's design, it consists of six architectural constraints for API building (Fielding, 2000):

- **Client-Server:** REST applications should have a client-server architecture and they should be able to operate independently from each other. Separation of concerns is the principle behind this constraint (Fielding, 2000).
- **Stateless:** Communication needs to be stateless in nature, meaning that each request from the client must contain all the necessary information to evaluate the request (Fielding, 2000).
- **Cacheable:** The data inside a server response must be labeled as cacheable or non-cacheable (Fielding, 2000).
- **Uniform Interface:** Irrespective of device or application type, there should be a uniform method of interaction for a given server. This simplifies the overall system architecture and improves the visibility of the interactions (Fielding, 2000).
- **Layered System:** Each component should only be allowed to "see" the layer that they are interacting. with (Fielding, 2000).
- **Code on Demand:** This is an optional constraint. If needed, servers can provide executable code such as applets or scripts to the client (Fielding, 2000).

Any API that adheres to the above principles are referred to as RESTful APIs and they HTTP requests to work with resources (Altexsoft, 2019a).

GraphQL

GraphQL is a query language for APIs and a server-side runtime for executing queries that was designed by Facebook in 2012 and was released to public in 2015. The next chapter is fully dedicated to GraphQL and it will be explained in depth there.

2.2. Microservices Architectures

Microservices software architecture pattern is based on the "single responsibility principle" that was coined by Robert C. Martin, which states that "a class should have only one reason to change" (Martin, 2014). Microservices architectures extends this approach to coupling the services in a system into "micro-services" that can be developed, deployed and maintained independently of each other (Singh, 2018). More simply put, microservices architectures consist of "a collection of small, autonomous services that

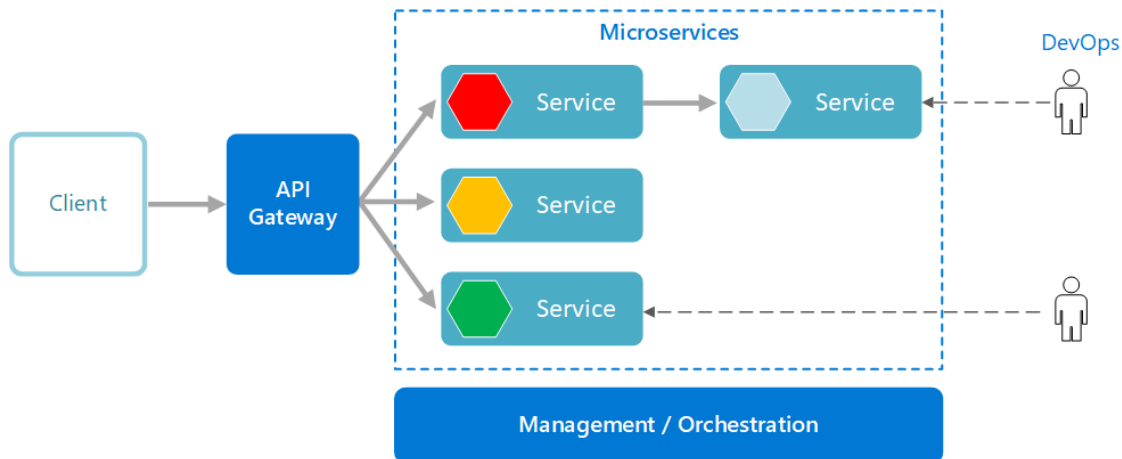


Figure 2.2.: Microservices architecture diagram (Microsoft, 2019).

implement a single business capability" (Microsoft, 2019). Figure 2.2 shows a structural diagram of a simple microservices architecture.

There are a few key items to go over in Figure 2.2:

- **Client** - The front-end or another back-end application client that calls the services.
- **API Gateway** - This is the entry point for the clients. The clients do not call the services directly, but instead they call them through this API. The gateway can be based on REST or GraphQL.
- **Service** - These individual and independent services form the microservices architecture.
- **Management / Orchestration** - This component is responsible for hosting the services.
- **DevOps** - This represents the internal developing operations of the organization.

2.3. Change Management

Change management has been defined as "the process of continually renewing the organization's direction, structure, and capabilities to serve the ever-changing needs of the marketplace, customers and employees" (Moran and Brightman, 2000). According to Gill, change programs often fail due to the poor execution of change management, which requires proper planning, monitoring and control, compatible corporate policies, resources and know-how (Gill, 2003).

The change management capabilities of an organization is also important for API management, since the ability to adapt to or change APIs depends on it. A challenge in

change management of APIs has been attributed to the lack of education of developers. According to Andreo et al., "API design and evolution need a different mindset, awareness and continuous education of development teams to achieve better API quality" (Andreo and Bosch, 2019). Being able to handle the change of APIs is crucial for business, because, according to Macvean, Maly and Daughtry (Macvean, Maly, and Daughtry, 2016), as cited by Murphy et al. (Murphy, Kery, Alliyu, et al., 2018), changing an API after its deployment has a potential to break the software that depend on it.

3. GraphQL

GraphQL is a query language for APIs and a server-side runtime for executing queries (The GraphQL Foundation, n.d.[a]) that was designed by Facebook in 2012 and was released to public in 2015. This section will explain the key aspects of GraphQL.

3.1. Description

"GraphQL is a query language designed to build client applications by providing an intuitive and flexible syntax and system for describing their data requirements and interactions" (Facebook, 2018). It is supported by many different programming languages (The GraphQL Foundation, n.d.[b]), Javascript being the most commonly used one, while GraphQL queries resemble the JavaScript Object Notation (JSON) (ECMA, 2017).

3.2. Design Principles

GraphQL has a number of design principles:

- **Hierarchical:** GraphQL queries are structured hierarchically, meaning that the queries are shaped in the same way as the returned data (Facebook, 2018).
- **Product-centric:** GraphQL is client focused, and driven by the requirements of the front-end engineers. This means that the data provided by GraphQL servers are easily accessible (Facebook, 2018).
- **Strong-typing:** GraphQL servers define an application-specific type system. Each field in the data schema has either a scalar type like integer, string etc. or a class definition, and the queries coming from the front-end can only ask for these types. This ensures that queries are syntactically correct and valid before execution (Facebook, 2018).
- **Client-specified queries:** GraphQL servers present their capabilities to the clients, and it's the clients' responsibility to specify how to utilize these capabilities. GraphQL queries are specified in the field level and return exactly what the clients ask for (Facebook, 2018).
- **Introspective:** GraphQL is introspective, meaning that the GraphQL language itself should be able to query a GraphQL server's typing system. Introspection

will be investigated in more detail in the last section of this chapter (Facebook, 2018).

3.3. Operations

GraphQL operations are queries, mutations or subscriptions that are interpreted by the GraphQL execution engines. (Stubailo, 2017a) There are three parts to a GraphQL operation, as can be seen figure 3.1.

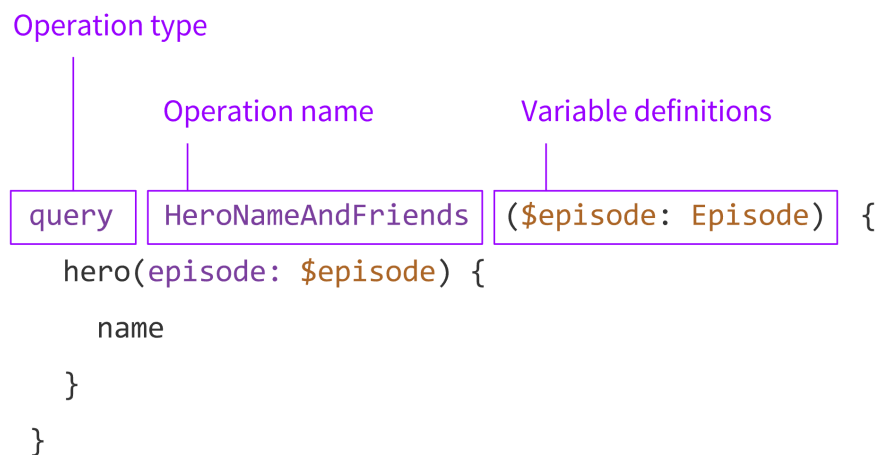


Figure 3.1.: Parts of a GraphQL operation.

- **Operation type:** It describes the type of operation the client is trying to do. It can be either *query*, *mutation*, or *subscription* (Stubailo, 2017a).
- **Operation name:** It's a good practice to give the operations meaningful names, especially for logging and debugging purposes. This is just like naming functions in a programming language (Stubailo, 2017a).
- **Variable definitions:** Queries sent to the GraphQL servers can have dynamic parts that change at each execution, while the query structure stays the same. These are called the *variables* of the query. And the *variable definitions* part of an operation is where these variables are defined (Stubailo, 2017a).

3.3.1. Query

Queries are read only operations that fetch data (Facebook, 2018). Figure 3.2. is an example featuring the Star Wars API, which shows a query and a section of the response.

On the left side is the query that contains the fields that the client wants to fetch, and on the right is the response of the server in JSON notation.

<pre> query HeroNameAndFriends { heroes { name friends { name } } } </pre>	<pre> "data": { "heroes": [{ "name": "R2-D2", "friends": [{ "name": "Luke Skywalker" }, { "name": "Han Solo" }, ...], }, ...] } </pre>
--	--

Figure 3.2.: A GraphQL query and its response.

3.3.2. Mutation

Mutations are write operations followed by fetch (Facebook, 2018), meaning that the data in the database is first overwritten and then fetched back. Figure 3.3. is an example of a mutation operation and the server's response to the client.

<pre> mutation CreateReviewForEpisode(\$ep: Episode, \$review: Review) { createReview(episode: \$ep review: \$review) { stars commentary } } </pre>	<pre> { "data": { "createReview": { "stars": 5, "commentary": "This is a great movie!" }, } } </pre>
---	--

Figure 3.3.: A GraphQL mutation and its response.

This mutation operation first attempts to create a movie review object in the database, and then returns the "stars" and "commentary" fields. The "\$ep" and "\$review" values

are the variables that are used by the mutation operation, and are defined separately in a different part of the query.

3.3.3. Subscription

Subscriptions are GraphQL requests that return data to the client in response to some server-side trigger events (Stubailo, 2017b). Figure 3.4. is an example of a subscription operation and the server response.

<pre>subscription onCommentAdded { commentAdded(reviewId: "sw4rw12345") { comment { id content author { username } } } }</pre>	<pre>{ "data": { "commentAdded": { "id": "123", "content": "Hello!", "author": "Epic Comments Guy" } } }</pre>
--	--

Figure 3.4.: A GraphQL subscription and its response.

This "commentAdded" subscription operation subscribes the client to a review, and receives responses whenever the comment adding event happens. Similar to the mutation operation above, the "\$reviewId" is a variable, but this time it has been defined directly in the query.

3.4. Schemas

A GraphQL service's total type system capabilities are referred to as that service's schema (Facebook, 2018). A schema describes the type of data offered by the service, the relationship between those types and the possible operations that can be performed on them (Wittern, Cha, Davis, et al., 2019). Figure 3.5. shows an example schema.

This schema defines the operation types *query* and *mutation*, respectively, the "review" and "createReviewForEpisode" operations. According to the schema, clients can query for a **Review** object identified with an "id" argument (the ! character means it is a required argument) and they can create a new **Review** object through mutation (The GraphQL Foundation, n.d.[c]).

```
schema {  
  query: Query  
  mutation: Mutation  
}  
type Mutation {  
  createReviewForEpisode(input: ReviewInput!): Review  
}  
type Query {  
  review(id: ID!): Review  
}  
type Review {  
  id: ID!  
  author: String  
}
```

Figure 3.5.: A GraphQL schema.

3.5. Types

GraphQL's type system comprises of objects and scalar types that are defined in service's GraphQL schema.

3.5.1. Scalar

Scalar types are the concrete data points which a GraphQL object's fields resolve to. In the following query in Figure 3.6., the *name* and *appearsIn* resolve to scalar types (The GraphQL Foundation, n.d.[c]):

<pre>{ hero { name appearsIn } }</pre>	<pre>{ "data": { "hero": { "name": "R2-D2", "appearsIn": ["NEWHOPE", "EMPIRE", "JEDI"] } } }</pre>
--	--

Figure 3.6.: A GraphQL query depicting scalar types.

GraphQL has the following default scalar types (The GraphQL Foundation, n.d.[c]):

- **Int:** A signed 32-bit integer.
- **Float:** A signed double-precision floating-point value.
- **String:** A UTF-8 character sequence.
- **Boolean:** True or False.
- **ID:** A unique identifier.

3.5.2. Enumeration Types

Enumerations or shortly referred as *Enums* are a type of scalar that is restricted to a limited set of allowed values (The GraphQL Foundation, n.d.[c]).

3.5.3. Lists and Non-Null

Lists and non-nulls are type modifiers that affect the validation of the returned values. Lists are denoted by square brackets, [and], and indicates that the field will return an array of the specified type. Whereas the non-nulls are denoted by an exclamation mark ! and indicates that the field cannot return an empty value (The GraphQL Foundation, n.d.[c]). Figure 3.7. is an example of a non-null "name" field with scalar type and a non-null list "appearsIn" with an *Enum* type:

```
type Character {  
  name: String!  
  appearsIn: [Episode]!  
}
```

Figure 3.7.: A GraphQL query depicting enum and non-null types.

3.5.4. Interfaces

An interface is an abstract type that contains certain sets of fields that a type must include to implement the interface (The GraphQL Foundation, n.d.[c]). Any type that implements the interfaces need to have the exact fields along with the correct argument and return types (The GraphQL Foundation, n.d.[c]).

3.5.5. Union Types

Union Types indicate that a field can return more than one object type, but leaves the specific fields of the objects undefined (Apollo GraphQL, n.d.[a]).

3.5.6. Input Types

Input types allow objects to be passed as arguments to a GraphQL query. They are most often used in the mutation operations where the clients can pass a whole object to be created (The GraphQL Foundation, n.d.[c]). Input types look identical to regular object types, but instead uses the keyword *input* rather than *type*.

3.6. Introspection

Introspection is a technique that allows the clients to ask a GraphQL schema for information on what kind of queries it supports (The GraphQL Foundation, n.d.[d]). Because of this powerful utility, GraphQL servers provide their own documentation on-the-go.

Introspection can be done by querying the `__schema` field, which is always available as default in GraphQL schemas. Figure 3.8. is an example of introspection and the server response:

<pre>{ __schema { types { name } } }</pre>	<pre>{ "data": { "__schema": { "types": [{ "name": "Query" }, { "name": "Episode" }, { "name": "Mutation" }], } } }</pre>
--	---

Figure 3.8.: Introspection and server response.

The introspection on the left asks the schema what types are available to operate with. The response tells the client that they have the *Query*, *Episode* and *Mutation* types.

4. Rest to GraphQL Transition

It's important to note that GraphQL is an alternative to REST, and it's not an absolute replacement for REST. Even though it fixes a lot of problems that developers experience with RESTful architectures, it also brings a new set of challenges. This chapter will first evaluate and compare these two APIs, and then go over how to transition from REST to GraphQL.

4.1. REST vs. GraphQL

The following sections compare REST and GraphQL in certain major areas.

4.1.1. Overfetching/Underfetching

One of the limitations of REST today is the problem of over/under-fetching. RESTful services, by design, allow the requesting clients to access and utilize the web resources by using a "uniform and predefined set of stateless operations" (Eschweiler, 2018). Meaning, RESTful APIs provide, through the implemented endpoints, the data that server developers designed, and nothing more or less. This leads to two points:

- **Overfetching:** To access the required data field, the clients calls an *http* endpoint that returns this field. However, the endpoint is designed in a way that returns multiple data fields in the server response, which are not required by the client but are there nonetheless (GraphQL Community, n.d.[a]). This is called overfetching. In the end the client ends up with more data than needed, and has to ignore them.
- **Underfetching:** Underfetching roughly means that a specific endpoint doesn't provide enough of the required information (GraphQL Community, n.d.[a]). This leads to client making additional requests to fetch all of the required data. This can escalate to a situation where a client needs to first fetch a list of fields, and then needs to make additional requests to get the relevant data concerning those fields.

Figure 4.1 shows a typical REST fetch.

In the first request, the `/users/<id>` endpoint has been accessed to fetch the initial user data. In the second request, `/users/<id>/posts` endpoint has been called to get the posts of the initially fetched user. And finally the third endpoint, `/users/<id>/followers`, is accessed to fetch the followers data of the user. This is an example of both underfetching



Figure 4.1.: Three REST requests to fetch data.

and overfetching. Initially there have been not enough data, and in the end there is more data than required.

In contrast to this, GraphQL allows the exact data to be fetched using a single query, because as seen in the previous chapter, every GraphQL object is connected, can be fetched through these connections. The example in Figure 4.2. shows the same request done with a GraphQL fetch instead.

Here it can be seen that the exact same data can be fetched with a single request. A query is sent to the GraphQL server, in the shape of the desired schema, and it is exactly what's returned in the response body. This is one of the major advantages of GraphQL over REST. The problem of over/underfetching is eliminated.

4.1.2. Product Iterations and Versioning

Endpoints using REST APIs are most commonly structured in a way that reflects to views of the front-end applications (**GraphQLREST**). This is quite useful since it allows the clients to get all the required information, in the shape they desire, by simply accessing one endpoint.

The biggest drawback with this approach is that doesn't allow fast product iterations on

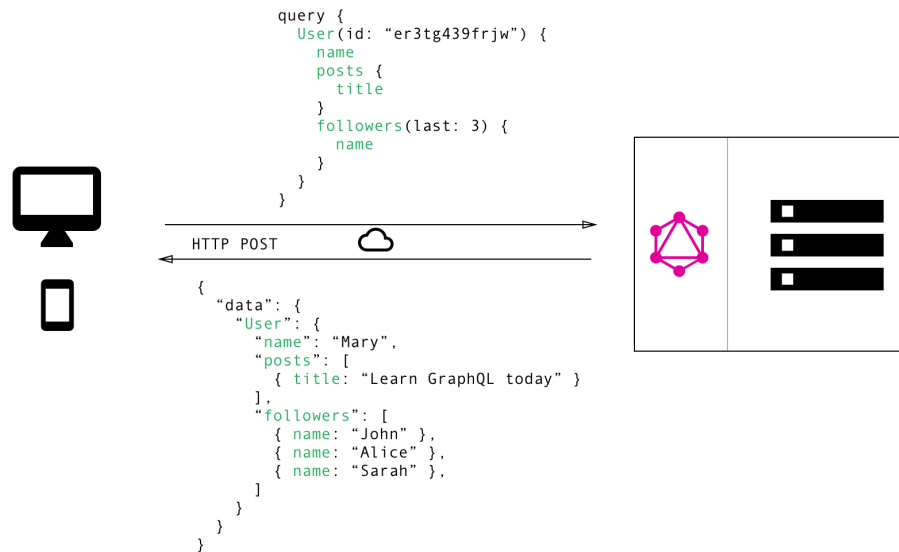


Figure 4.2.: A single GraphQL request to fetch data.

the front-end. Meaning that with every change that's made to the UI, it's highly likely that a change will also be required on the endpoint to alter the data that it's providing. This slows down the development process, and requires constant versioning on the back-end side.

GraphQL addresses this problem by allowing the front-end to dynamically and spontaneously change their queries. Simply put, the requirement changes of the new UI views can be solved by altering the query that's sent to the GraphQL servers, with no extra work required by the back-end engineers. This eliminates the need for versioning on the back-end and allows for a faster product development (Altexsoft, 2019b).

4.1.3. Typing and Validation

GraphQL's introspection feature allows for the front-end clients to delve into the typing of the data and ensures that the applications only ask for what's possible and in the appropriate format (Altexsoft, 2019b). Therefore, there is no need to develop additional validators. In contrast, clients using endpoints built on REST APIs do not know about the typing of the data before they receive the payload and need to build validators to ensure it is in the required format.

4.1.4. Error handling

When there is an error while processing queries, the GraphQL servers provide a detailed description of what kind of error occurred and to which part of the query it is related (Altexsoft, 2019b). Contrarily in REST, the HTTP headers are checked for the status of response, and then further inspection is carried on investigate the error.

4.1.5. Performance issues with complex queries

Basic GraphQL queries are quicker than their REST counterparts, but as the queries grow more complex, i.e. asking for too many nested fields at once, performance issues arise (Altexsoft, 2019b). And in these kind of cases, retrieving the data by means of multiple REST endpoints, with fine-tuned and specific queries is more efficient (Herrera, 2018).

4.1.6. Caching

Caching helps reduce the amount of traffic to a back-end server by storing the most commonly accessed information close to the client. Since GraphQL doesn't rely on HTTP caching methods, storing the contents of a request, *caching*, is not enabled. In contrast, by providing many endpoints, "a REST API enables easy web caching configurations to match certain URL patterns, HTTP methods, or specific resources" (Altexsoft, 2019b). Due to having only one endpoint with many different queries, it's much harder to use this type of caching with a GraphQL API.

4.1.7. File uploading

It is not possible to upload files with the current specifications of GraphQL, whereas REST allows file uploading via *POST* or *PUT* operations (Altexsoft, 2019b).

4.2. REST to GraphQL Transition

The following section is a collection of transition processes undergone by the leading GraphQL adapters. It is focused on best practices gained by practical experiences rather than theoretical assumptions and experiments.

4.2.1. Architectural Transition

The traditional front-end applications had been interacting with relational databases. As the applications grew more complex, the data sources became varied and the applications stopped interacting with them directly but instead through APIs; and these being RESTful APIs most of the time (Stubailo and Man, 2016). Figure 4.3. shows a diagram depicting a front-end application's communication with back-end services.

In this example, the front-end application is using four REST endpoints to connect to three different databases. The first two endpoints connect to the same database, but return different data to be used in different components of the front-end application. The other two endpoints respectively connect to other databases to return other kinds of data. The first step in a transition to GraphQL is to transform this architecture. According to the GraphQL community (GraphQL Community, n.d.[b]), there are three commonly used GraphQL architectures.

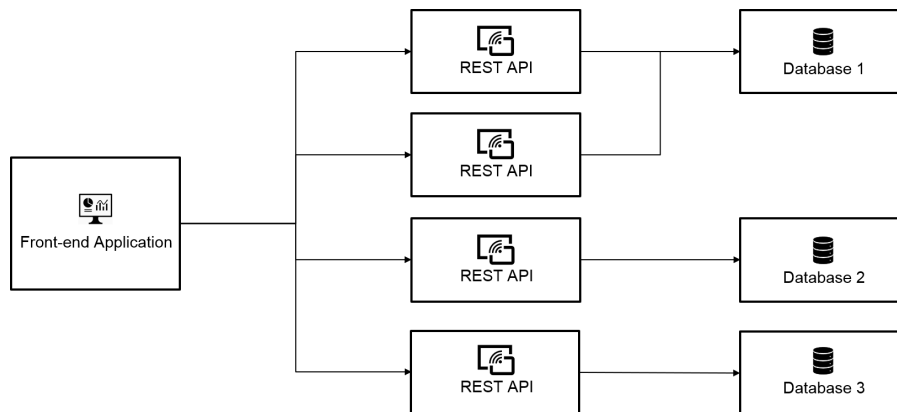


Figure 4.3.: A front-end application communicating with its back-end services.

4.2.1.1. GraphQL server with a connected database

In these kind of architectures, there is a single web server that implements GraphQL. The front-end application is directly connected to the database through a GraphQL layer, and Whenever a query is sent to the GraphQL server, the server reads the query payload and proceeds to fetch the required information from the connected database. It constructs the response as a JSON object notation, as seen in the previous chapter, and returns it to the client. GraphQL doesn't rely on operating with any specific database, so what was there with the previous REST architecture can still be used without a problem. Figure 4.4. shows a a standard architecture featuring a front-end application connecting to a single database through a GraphQL server. This design would replace the first two REST APIs seen previously on figure 4.3.

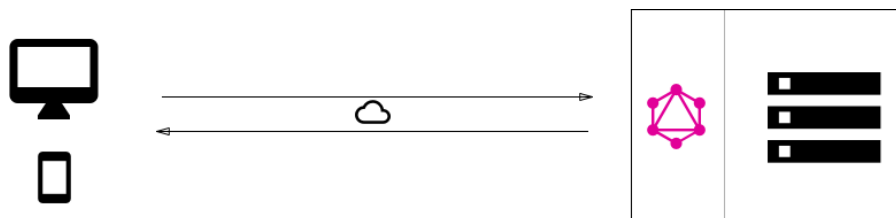


Figure 4.4.: A standard architecture with one GraphQL server that connects to a single database.

4.2.1.2. GraphQL layer that integrates existing systems

Another common practice is to have GraphQL, instead of directly connecting the application to the database, integrate multiple existing services behind a single endpoint. This practice is especially suited for companies that use a plethora of microservices through numerous APIs. It is also compelling for companies that are unwilling to alter their long existing legacy systems, because it's usually practically impossible to integrate

new technologies into these type of systems (GraphQL Community, n.d.[b]).

In this transition context, GraphQL can be used to unify the existing systems, be it legacy systems, other microservices or third party APIs, by acting as a gateway between the front-end application and the back-end services.

Figure 4.5 depicts this case. It can be seen there that the front-end client only talks to the GraphQL server. Again, the GraphQL server receives a query from the front-end client, reads it, and then fetches data. For this transition, no changes to the existing APIs are necessary. The legacy services and the previous APIs are still there. GraphQL aggregates them and provides their data to the front-end application through one unified endpoint. If we look back to Figure 4.3, this design would come in between the front-end application and the REST API nodes.

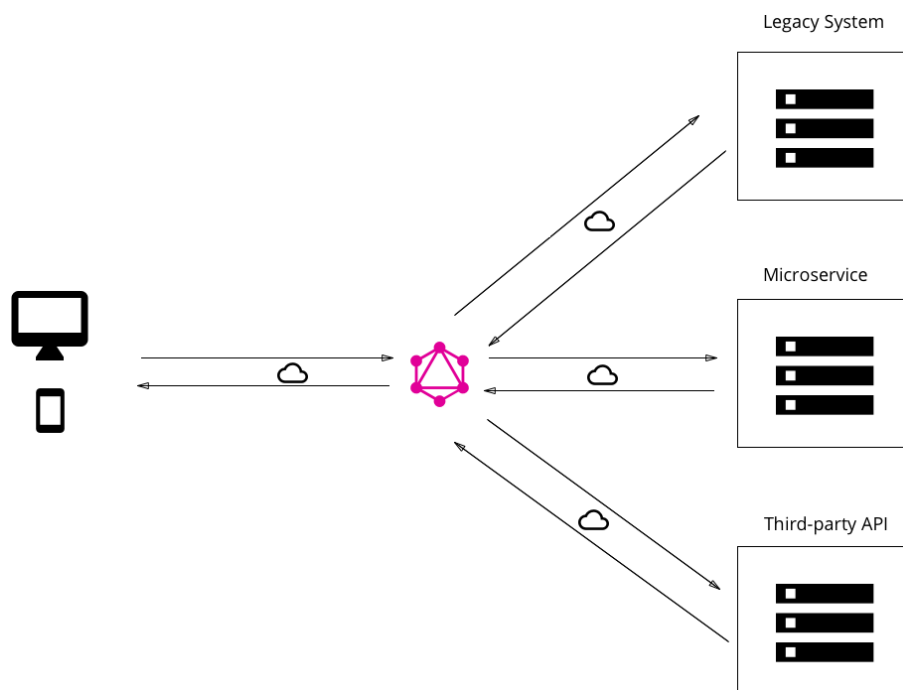


Figure 4.5.: GraphQL acting as a gateway.

4.2.1.3. Hybrid approach with connected database and integration of existing system

The third architectural design choice is to combine the first two approaches and build a GraphQL server that has both a directly connected database and still communicates with other APIs and legacy systems (GraphQL Community, n.d.[b]). Figure 4.6. illustrates this case.

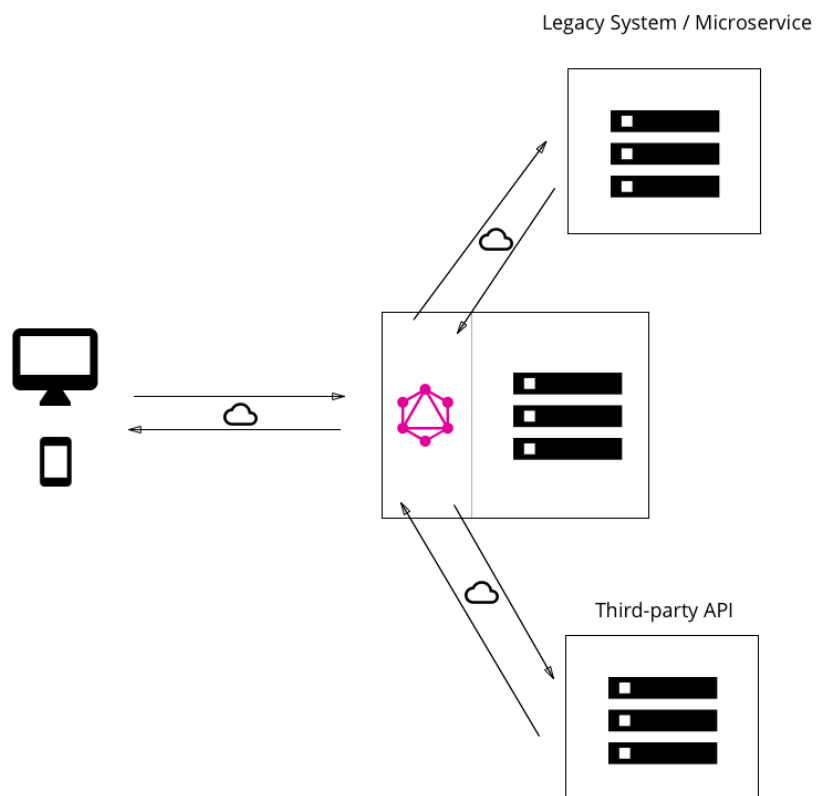


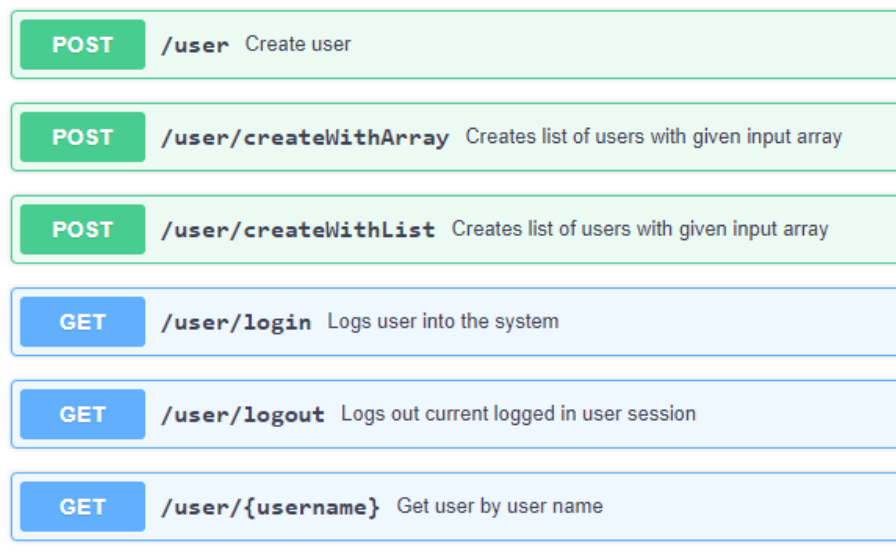
Figure 4.6.: GraphQL acting as a hybrid gateway.

This approach allows increased flexibility and pushes all the data and information management complexity to the GraphQL server. When the server receives a query, it will resolve it and either reach out to the connected database or the integrated APIs / legacy systems to gather and return the required data.

The following section describes the next transition step, which is generating a schema from existing REST documentation that maps all the above-mentioned API calls to the correct data sources.

4.2.2. Designing a GraphQL Schema

Most of the REST APIs today use the *Open API Specification* (OAS) to document the functionalities and responses of the API calls. Formerly known as "Swagger", the OAS is "a standard, programming language-agnostic interface description for REST APIs" (OpenAPI Initiative, 2017). The OAS is a format used by the developers of the API to describe and document them in an understandable and organized manner. It breaks down the API into operations identified by unique combinations of URL path and HTTP methods, required input parameters for queries and their responses. Figure 4.7 shows an interface containing a simple list of operations that were, under the hood, generated from an OAS schema. It shows the HTTP methods that can be performed on certain URL paths. If clicked on these boxes, the interface further expands to show query parameters and responses.



POST	/user	Create user
POST	/user/createWithArray	Creates list of users with given input array
POST	/user/createWithList	Creates list of users with given input array
GET	/user/login	Logs user into the system
GET	/user/logout	Logs out current logged in user session
GET	/user/{username}	Get user by user name

Figure 4.7.: Operations about "user" on an Open-API Specification schema interface.

Most often, the OAS schema objects can be directly translated into GraphQL types (Wittern, Cha, and Laredo, 2018). For instance:

- Query parameters map to **GraphQL Input Types**.
- Arrays map to **GraphQL List Types**
- Enums map to **GraphQL Enum Types**
- Scalars map to **GraphQL Scalar Types**

For a REST to GraphQL transition, the schema describing these objects and operations described with URL and HTTP methods, need to be translated into a single GraphQL schema. This transition approach relies on "taking as input an OAS describing the target API and outputting a GraphQL schema that, once deployed, forms a GraphQL wrapper around the target API" (Wittern, Cha, and Laredo, 2018). The GraphQL wrapper translates the queries to corresponding requests against this target API. This is in parallel with the approach described in the previous section, where a GraphQL gateway integrates the existing systems(see Figure 4.5.). Meaning that the existing systems are wrapped by a GraphQL gateway and the GraphQL schema is generated from each individual API's OAS schema.

5. Preliminary Expert Interviews

This chapter summarizes the expert interviews that were conducted before the implementation. The interviews give insight to the requirements of the implementation, and the potential challenges that will come along the way based on the experiences of the interviewees.

5.1. Interviews Summary

We have asked experts, who are the developers with GraphQL experience that are working for the company which is subject to the case study and also developers from outside, some questions regarding GraphQL itself and the transition process from REST APIs to GraphQL APIs. This section summarizes these questions and the answers received in an analytically grouped way.

- **How GraphQL can adress the current challenges in the dashboard application -** Currently many REST APIs are being called from the dashboard application. This causes a lot of complexity because all of the APIs have to be defined within the application and the structure of the API requests and responses need to be known by the developers. GraphQL can address this issue by aggregating the REST APIs through a gateway implementation and only exposing the GraphQL endpoint to the front-end application. Another challenge is caching data and refreshing front-end components individually. At the moment, an extra implementation in the form of another front-end component is required when calling the REST APIs in order to make the calls happen on a periodic basis(also known as "polling"). GraphQL can address this with it's prominent polling option.
- **Potential technical challenges in transitioning to a GraphQL gateway and how to overcome them -** Deciding on which tools and third party libraries to use when developing a GraphQL server is a crucial technical challenge. However, based on the experiences of the interviewees, the current state of the art library to use is the "Apollo-GraphQL". Another challenge is handling *authentication* and *authorization*. *Authentication* is the verification of the identity of users through validating their credentials such as username and password, whereas authorization is the verification of access rights of the user (Siddiqui, 2018). The interviews suggest that the best way to handle this is via having the authentication built in to the GraphQL gateway and keeping the authorization in the REST APIs.

- **Potential organizational challenges in transitioning to a GraphQL gateway and how to overcome them** - Almost all of the web developers inside and outside the company have a certain expertise in the REST APIs. It is safe to say that a majority of the developers today have not worked with GraphQL. The first step of overcoming this problem is making people interested in GraphQL by showing to benefits of using it. Next, diving deep into the concept by giving tech talks would allow the developers to familiarize themselves with GraphQL. Finally, "hand-holding" the developers throughout an implementation process is essential in making them comfortable with using GraphQL. It is a good moment now to note that the before-mentioned *introspection* property of GraphQL is very useful in this familiarization process.
- **Expectations from the new GraphQL implementation of the dashboard application** - The expectation of the interviewed developers is to have flexible data requests for the front-end components, to be able to know the data structure beforehand easily and to not care about dozens of different URLs when doing API calls.

6. Implementation

This section describes the REST to GraphQL transition and the implementation process of the GraphQL gateway for the case study of this research. It is divided into three parts:

- **Architectural Overview:** This section gives an overview of the current systems architecture and explains the proposed architectural change that will come with the implementation of the gateway.
- **Development of the Prototype:** This section talks about the GraphQL server prototype that's designed corresponding to the proposed architecture.
- **Implementation Challenges:** This section will go over some of the important challenges faced along the implementation process.

6.1. Architectural Overview

The case study of the research features a dashboard application that multiple different developer teams use within the company. The dashboard application is a front-end system (henceforth, the "dashboard application" and "front-end" phrases will be used interchangeably, because the dashboard application is the front-end of the case study) that monitors the back-end systems and any changes in the databases, and it's customizable for each team. Meaning that the application calls different APIs in different cases.

The current architecture of the dashboard application is microservices-based. There is one one main front-end application and from there 87 REST API calls along with 18 database queries are made. Figure 5.1. shows the current architectural diagram featuring all the services connected to the front-end application and the API calls.

From the diagram, it can be seen that currently multiple data sources are being used to get the required data to run the front-end application, and in order to connect to those data sources that there are 87 endpoints that are being utilized. Whenever additional information is required in the front-end, for example, a new endpoint needs to be created to reach out to the databases and return the data.

The proposed architectural change involves a GraphQL gateway as seen in Figure 5.2. The gateway will aggregate all the API calls and the database queries, and provide their data to the front-end through one unified endpoint.

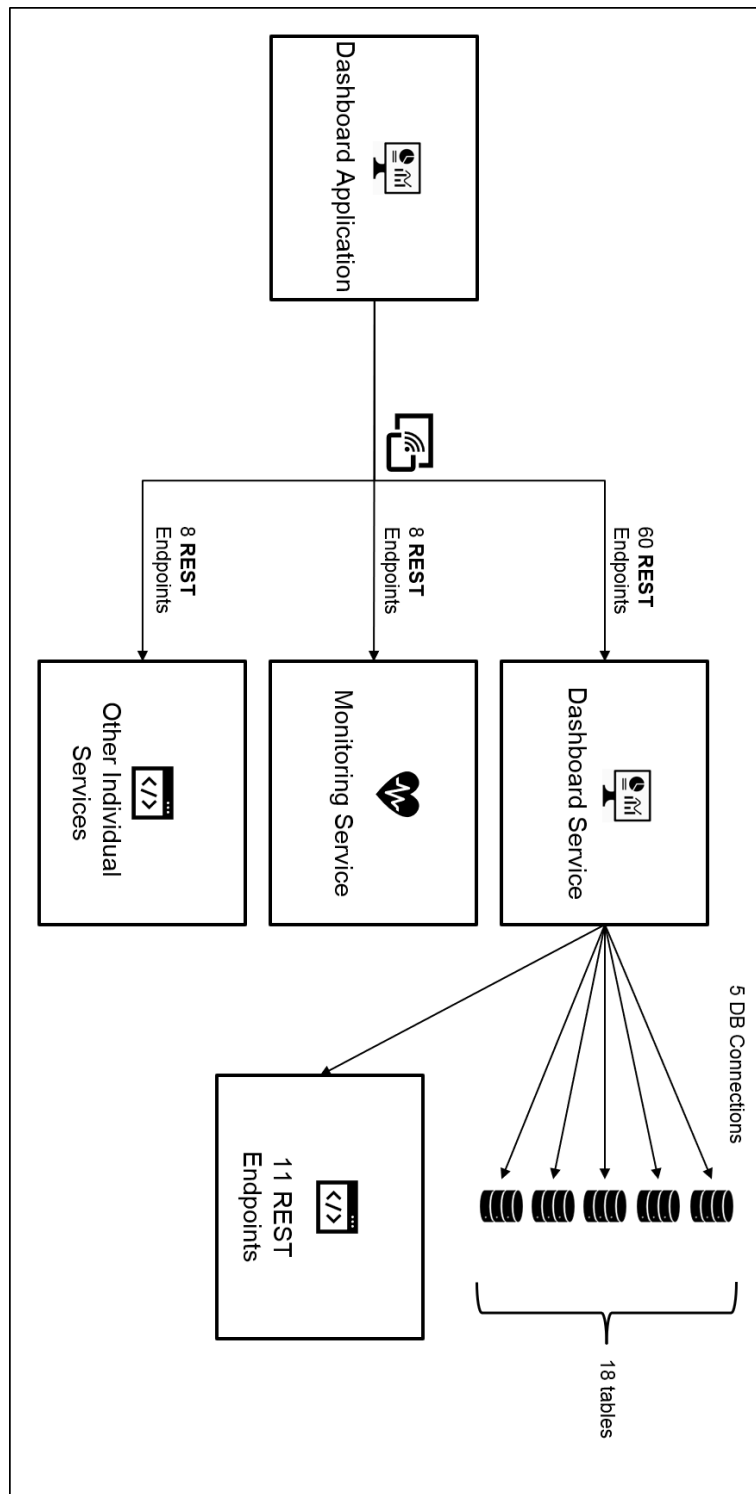


Figure 6.1.: Current state of the microservices architecture.

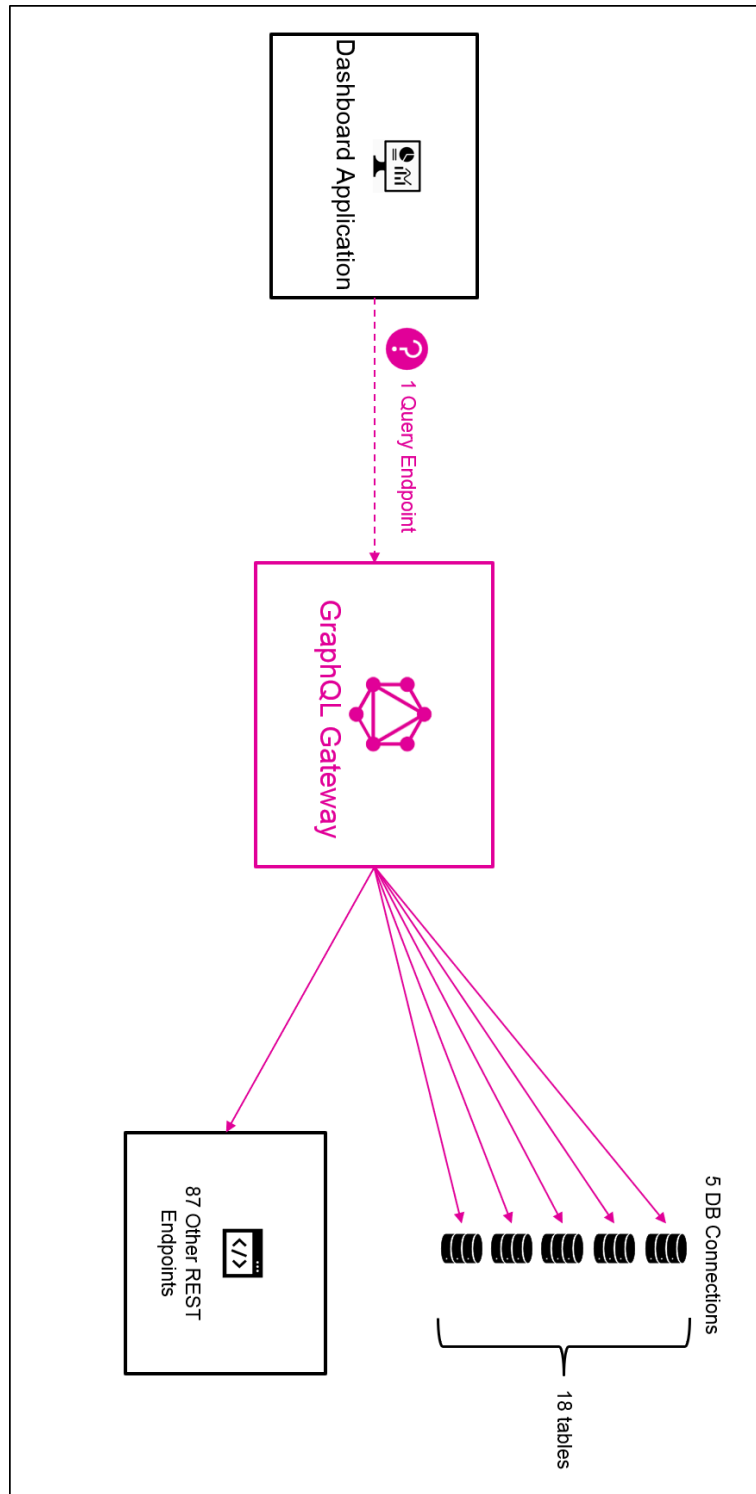


Figure 6.2.: Proposed architectural change featuring a GraphQL gateway.

6.2. Development of the Prototype

This section will describe the development process of the GraphQL gateway server. We have begun the development by categorizing the monolithic dashboard application into smaller meaningful sections. The application serves multiple teams in the company and they each have separate dedicated web pages. Instead of migrating the whole RESTful back-end that the application uses, we have isolated and focused on a single one of these pages. This page, that is the subject of this paper's case study, is composed of 8 widget components that call a total of 11 REST endpoints. With these calls, the application gets data from both the internal APIs of the company and public APIs such as weather data providers.

For this paper's purpose, and to be used for the case study, we have aggregated these REST endpoints in a GraphQL gateway. In the end, the front-end application only makes a single call to the GraphQL server, and depending on what is requested, the GraphQL server accesses one or more of the 11 REST endpoints to get the data. The rest of this section describes the transition step-by-step.

6.2.1. Choosing a Software Library: Apollo-GraphQL

The first step of the server implementation was choosing a library. There are quite a few GraphQL libraries available today, such as *Apollo*, *Bit*, *Relay*, *Prisma* being among the most prominent ones (Saring, 2019), and they make the development of a server much easier by providing the essential functionalities. For our research, we have chosen the *Apollo* library, as it is suggested on the official GraphQL web-page (The GraphQL Foundation, n.d.[b]) for Javascript based development. *Apollo* has an extensive documentation on developing a GraphQL server (Apollo GraphQL, n.d.[b]), and our implementation followed their suggested best practices as can be seen in the following sections.

6.2.2. Setting Up the GraphQL Server

The GraphQL server was created using the afore-mentioned Apollo-GraphQL library. Figure 5.3 shows the declaration of the server using Apollo-GraphQL. Essentially, this library allows us to create a GraphQL server by defining the *schema*, *data sources* and *resolver functions* and then passing them to the *ApolloServer* constructor as seen in the figure.

The following sections describe how the *schema*, *data sources* and *resolver functions* are defined.

6.2.3. Designing the GraphQL Schema

The first and the most crucial step of the transition was creating a GraphQL schema. This schema was defined by checking which data fields the front-end components

```
const graphQLServer = new ApolloServer({  
  schema,  
  resolvers,  
  dataSources,  
});
```

Figure 6.3.: Declaration of the GraphQL server.

needed. Once this was figured out, sub-schemas were defined for each component and they were "stitched" together in the end to create the ultimate schema of the GraphQL server. Figure 5.4 shows an example of one these sub-schemas. This schema belongs to component that was used for monitoring the status of pull-requests in the company's code repository.

In the schema, it can be seen that there is a query *pullRequestsLegacy* that returns an array of type *PullRequest*. The type *PullRequest* is defined below and it contains various fields of scalar types and object types such as *User* that depicts the type of the *reviewers* field. These object types are also defined in the schema as can be seen further down. All of these types that have been defined in the schema resulted from our initial inspection of the status of the front-end widget. Once we determined which data the application used to display on its widget, we have defined the schema. We have repeated the same process for all of the widgets and created multiple schemas. In the end, all of these schemas were stitched together to create the total schema of the GraphQL server.

6.2.4. Defining the Data Sources

After the schema was defined, we knew what data we needed, but not how to get this data. To achieve this, data sources need to be defined which then will get mapped to the queries through resolver functions that will be explained in the next section.

Since our GraphQL server acts as a gateway that aggregates multiple REST APIs, our data sources will be the REST APIs that the application normally used before the migration. Figure 5.5 shows the code that defines the data source which corresponds to the repository example in Figure 5.4.

The *BitbucketAPI* is constructed as a *RESTDataSource* class. The *getPullRequests* method makes a request to a URL that is essentially an endpoint that belongs to a REST API, and returns the response after some custom filtering.

6.2.5. Defining the Resolver Functions

At this we have the schema and the data sources defined, but these two are not connected to each other yet. The resolver functions do exactly that. Figure 5.6 shows how the query

```
extend type Query {  
  pullRequests(project: String): [PullRequest]  
}  
  
type Build {  
  name: String  
  url: String  
}  
  
type PullRequest {  
  repository: String  
  title: String  
  branch: String  
  createdAt: Int  
  hasApproval: Boolean  
  commentCount: Int  
  hasMergeConflict: Boolean  
  needsWork: Boolean  
  reviewers: [User]  
  author: User  
}  
  
type User {  
  user: UserDetails  
  role: String  
  approved: Boolean  
  status: String  
}  
  
type UserDetails {  
  name: String  
  emailAddress: String  
  id: ID!  
  displayName: String  
  active: Boolean  
  slug: String  
  type: String  
  links: JSON  
}
```

Figure 6.4.: A sub-schema for monitoring pull-requests in the code repository.


```
class PullRequestsAPI extends RESTDataSource {
  constructor() {
    super();
    this.baseUrl = config.api.monitoringServiceUrl;
  }

  async getPullRequests(project) {
    const pullRequests = await this.get('/repos/pull-requests');
    return filterForProject(project, pullRequests);
  }
}
```

Figure 6.5.: Definition of the data source for the pull requests widget.

pullRequests, that is defined as a query type in the schema, resolves when executed. Whenever the *pullRequests* query is sent to the server, the server will connect to the *PullRequestsAPI* data source and return the received data to the client.

```
const monitoringResolvers = {
  pullRequests: async (root, args, { dataSources }) => {
    const pullRequestsLegacy = await dataSources.pullRequestsAPI.getPullRequestsLegacy(
      args.project,
    );
    return pullRequests;
  },
};
```

Figure 6.6.: Definition of the data source for the pull requests widget.

In the end, the above process was applied to the whole front-end component. Meaning, for each widget's requirements, a *schema*, one or more *data sources* and one or more *resolver functions* were defined. Once this was finished, we had a running GraphQL server that was used for the case study of this research. The following chapter will delve deeper into the case study.

6.3. Implementation Challenges

During the implementation, there weren't any major challenges besides those of technical nature that arose due to unfamiliarity with GraphQL. There are listed and briefly explained below:

- **Passing headers through the gateway** - Some of the REST endpoints among the data-sources required headers to be able to respond to the requests. The initial implementation weren't using any headers and once this case was encountered, the server failed to retrieve data. After doing some research and inspecting other use cases, this problem was solved using one Apollo's libraries with a built-in header support (Apollo GraphQL, n.d.[c]). This library allows the GraphQL server to access the headers in the client request and pass them over to the REST endpoints defined in the data sources.
- **The schema growing larger** - As we gradually developed the server and defined more queries and data types, the GraphQL schema started getting larger and larger. This did not result in a clean and easy to extend code structure. To fix this, we have divided the schema into different parts corresponding to different data sources, and then "stitched" (Wawhal, 2018) them back together in the end.

7. The Case Study

This chapter will present the important findings of the qualitative analysis that is based on interviews and system usability surveys following a case study. For this case study, a total of 7 developers participated in a 2 hour long experiment where they were asked to perform a REST to GraphQL transition task. Table 6.1 shows the list of participants. For anonymity, the participants' names are not shown and they will henceforth be referenced by their Ids. The structure of the case study and the interviews can be found on Appendix A.

The case study task was moderately complex, and it was designed to be finished in 2 hours by developers who were not familiar with GraphQL. Specifically, the task was about modifying the front-end of the dashboard application to use the GraphQL gateway (that was described in the previous chapter) instead of its current REST back-end to request data. The task described how to formulate a query, explained how to progress through the implementation step-by-step and pointed towards online resources to learn more about setting up a GraphQL server.

7.1. System Usability Survey Results

The GraphQL gateway server that the participants used in their task had a usability score of 81 (Brooke, 2013). This puts the system above the average score of 68 (in over 500 studies) and into the top 10% percentile the most "usable" systems (Sauro, 2011).

Participant Id	Role	Front-end experience	Back-end experience	GraphQL familiarity (1 Least - 5 Most)
P1	Junior Developer	4 years	6 years	2/5
P2	Junior Developer	3 years	2 years	1/5
P3	Senior Developer	5 years	20 years	3/5
P4	Senior Developer	5 years	8 years	1/5
P5	Team Lead	5 years	10 years	2/5
P6	Junior Developer	1 year	1 year	1/5
P7	Senior Developer	6 years	6 years	1/5

Table 7.1.: Case Study Participants.

7.2. Interviews Analysis

This section will present the results of the interviews grouped into different categories.

7.2.1. Identified challenges during the transition

The participants were asked to identify any kind of transition-related challenges they might have faced during the experiment. These challenges are listed below:

- **Unfamiliar Query Syntax** - 6 of the participants (P1, P2, P4, P5, P6, P7) reported that they had troubles getting adjusted to the query syntax of GraphQL. 2 among these participants (P4, P6) stated that this was not a major issue and that it was only a challenge because it was their first time using it. After a few more tries they would get adjusted to the syntax. Whereas another participant (P7) said that even though they knew they would get used to the syntax eventually, they still found it unintuitive and difficult to understand without consulting the official GraphQL documentation. Furthermore, 2 participants (P1, P5) reported that the most difficult part of the query was including dynamic variables and that it took them multiple executions to understand the concept.
- **Paradigm Shift** - 1 participant (P3) stated that the current code of the task was written for RESTful API consumption and that they had to change some of the programming paradigms to be able to work with GraphQL. However, they also reported that after this issue was overcome, the resulting code looked cleaner and was able to achieve more.

7.2.2. Areas where using GraphQL is more convenient than using REST

The participants were asked to state in which areas using GraphQL APIs seemed to be better than using REST APIs. These are listed below:

- **Introspection** - All of the participants said that they found the introspection feature of GraphQL very useful. One participant (P3) stated that being able to form the queries and test them on the fly was very valuable to them, because they were able to see the exact response structure on the go.
- **Data complexity** - 3 participants (P1, P5, P7) said that GraphQL becomes more useful as request complexity increases. One of these participants (P1) claimed that using GraphQL seemed to be more suited for cases where multiple data sources exist. The other participants (P5, P7) said that for complex queries GraphQL would be more useful because to be able to achieve complex data transfer, REST APIs need more custom logic written in the code.
- **Flexibility** - 3 participants (P2, P3, P6) reported that if there was a need for managing data on the fly and a need for flexibility, GraphQL provided more

advantage. Having a schema defined in the server proved helpful when it came to flexibility.

- **Type Validation** - 3 participants (P1, P5, P7) reported that GraphQL was perfectly suited for type validation through its requirement for a typed schema (Wittern, Cha, Davis, et al., 2019).
- **Microservice-based architectures** - According to 1 participant (P5), using GraphQL as a gateway was more convenient for microservice-based architectures, through leveraging multiple services in one GraphQL API.

7.2.3. Areas where using REST is more convenient than using GraphQL

The participants were asked to state in which areas using REST APIs seemed to be better than using GraphQL APIs. These are listed below:

- **Simple Applications** - 1 participant (P5) said that for applications that only connect to one API to get its data, REST APIs would be enough and using GraphQL would be over-engineering.
- **Communication between APIs** - 1 participant (P1) claimed that for back-end APIs to communicate with each other, REST APIs are more useful. GraphQL is more suited for front-end to back-end interactions.

7.2.4. Acceptance towards GraphQL

Overall, all of the participants found GraphQL easy to use and stated that they would like to consider using it during future projects. 2 participants (P3, P7) said that they would prefer starting a project from scratch with GraphQL rather than migrating a complex project from REST. From a business point of view, 1 participant (P3) said that GraphQL was still a young technology, and undertaking a major transition to it from REST would not provide any additional business value at this stage. Furthermore, they said that if the REST APIs of the system worked properly, there was no reason to switch to GraphQL. The same participant concluded that with no business pressure and more available time, they would use GraphQL over REST.

8. Discussion

In this chapter, we briefly summarize the key findings of this research and identify areas for further research and improvement.

8.1. Key Findings

In this section, we describe the most important findings of this work.

8.1.1. Technical Findings

This section focuses on the technical aspects of the findings.

- **Use GraphQL as a gateway** - Based on both the preliminary research and the case study, we found that using GraphQL as a gateway between the front-end and the RESTful back-end is the most convenient way for transitioning, and in the end the resulting system utilizes the advantages of both REST and GraphQL. The advantage of REST in this case is to allow more customizable logic on the back-end, and the advantage of GraphQL is the flexibility in asking for data and triggering the REST endpoints independently.
- **Keep query complexity low** - Based on the query execution tryouts of the participants, it is better to avoid nesting queries within each other and keep the queries simple and straight-forward.
- **Step by step transition** - We found out that dividing the application into smaller and meaningful parts, and then deriving a GraphQL schema from these distinct logical parts makes the transition easier.

8.1.2. Organizational Findings

This section focuses on the organizational aspects of the findings.

- **"Handholding" the developers during the initial phases of the transition** - We found that the best way to maintain the developers' interest in the new and unfamiliar concept of GraphQL was to guide them throughout the process. There is plenty of resources and documentation available for GraphQL, but it takes time to go through and understand them for a timely transition. For this reason, one appointed person acquiring knowledge (in the case study this person was me),

and then teaching others while doing parts of the actual transition is faster and more efficient. This ensures a seamless progress through the transition.

- **No substantial business value in transitioning** - Most developers are already familiar with REST APIs and that has been the case for a lot of years. Given that, most systems have been designed to comply with REST APIs and they function perfectly well in accordance. A transition from REST to GraphQL would take a lot of time to accomplish, and in the end we would essentially have a system that has the same functionality and provides the same business value as the previous system, only with GraphQL under the hood. Therefore, even though GraphQL has a potential to improve the systems technically, there is not a significant added business value in undergoing such huge transitions. It is more pragmatic to opt for GraphQL while starting a project from scratch.

8.2. Limitations

The following section describes the limitations, in the form of threats to validity based on the guidelines proposed by Wohlin et al. (Claes Wohlin, Per Runeson, Martin Bost, et al., 1996), of our study.

- **Threat to construct validity** - The construct validity is concerned with correctly evaluating the dependent variables. In this study, the threat was hypothesis guessing by the participants. The participants were aware that the aim of the experiment was to evaluate GraphQL over REST, and their expectancy was for GraphQL to be "better". This made them likely to have a positive attitude towards GraphQL when doing the development and answering the interview questions. We have tried to mitigate this threat by preparing the interview questions in the most objective way with little emphasis on the positive aspects of GraphQL.
- **Threat to internal validity** - The internal validity is concerned with the uncontrolled aspects that may affect the experimental results, such as the participants' experience with GraphQL and programming in general. We have tried to mitigate this effect by providing all the necessary guidance and information to everyone equally, so even the most inexperienced participants would attain a level close to the experienced ones and everybody would start on an equal ground.
- **Threat to external validity** - The external validity is concerned with the possibility of generalizing this research's results. The experiment was conducted with 7 subjects. Therefore, it is possible that this number of subjects is not a representative sample. Still, our sample is diversified; the subjects have different roles in the company, different backgrounds, various years of experience in programming and a different familiarity with GraphQL. Also, the number of tasks in the case study could be a threat. Since there is only one task to accomplish, and only one type of GraphQL query to formulate, it might not be a perfect coverage of the GraphQL

use cases. The dashboard application that was the subject of the case study, by nature of dashboards, only reads and displays data. Meaning that no database writing operations were done in the case study, and that functionality of GraphQL and its effect on a transition was not evaluated. Furthermore, there was a limited time period (2 hours) given to the participants, so the task content and boundaries were designed to be completed in this time window.

9. Conclusion

This chapter summarizes the work done for the thesis and suggests follow-up research.

9.1. Summary

As the summary, we answer the research questions presented in the introduction of this paper.

RQ-1: What are the existing approaches for a REST to GraphQL transition?

This question was answered through literature review and a series of expert interviews. The current approaches for REST to GraphQL transition has two focuses that relate to schema design and architecture. As an answer to this question, we identified three ways of architectural design for the transition through literature review:

- A standard architecture with one GraphQL server that connects to a single database
- Using GraphQL as a gateway that integrates existing systems
- A hybrid approach where a GraphQL acts as a gateway that connects to database and also integrates existing systems

RQ-2: What could be a process for transitioning including stakeholders, activities and artifacts?

We answered this question through literature review and conducting an experiment where the participant developers were asked to do a transition task. Based on the analyses of the experiment, the following results were discovered as an answer to this research question:

- The process should start with tech-talks concerning the core concepts of GraphQL given to the stakeholders who would be undertaking the transition task.
- The process should continue with preparing an incremental transition plan where the micro-services are narrowly grouped in related categories.
- The developers undertaking the incremental transitions should be "handheld" throughout the completion of the transition process.

RQ-3: What are the lessons learned from a transition to GraphQL?

This research question was answered through further evaluation of the results of the experiment. After the experiment a quantitative analysis in the form of a system usability survey and a qualitative analysis in the form of interviews were done. Based on these analyses, the following were identified:

- GraphQL is easy to understand and use for the developers regardless of their background.
- Currently, REST APIs serve the needs of the modern front-end to back-end communication, and switching to GraphQL is does not provide any additional value.
- It is more pragmatic to use GraphQL when starting a project from scratch than undergoing a full-scale REST to GraphQL transition, since the effort and time required for a transition doesn't justify the potential added value. By using GraphQL on a fresh project, the double work of building up REST and then doing the transition is avoided.

9.2. Future Work

The time limitation of this research made it only possible to investigate rather simple transition cases. For example, the case study task didn't involve all the functionalities of GraphQL, such as writing and subscribing to data. Only the reading of data was tested and investigated. Therefore, as a potential future work, the impact of these complex functionalities of GraphQL on the transition process could be investigated. It is highly likely that the inclusion of the complex cases would affect the developers' ease of adjustment to GraphQL. Furthermore, this thesis only investigates the transition process from REST to GraphQL. It would also be an area of research to investigate transitions from other API standards, such as *SOAP*.

Even though these kind of future research is necessary, the work done in this thesis provides valuable first insights into how well GraphQL is perceived by organizations and how a transition process could be performed on a larger scale.

A. Evaluation Instructions

This evaluation session will be about using a GraphQL server to ask for data instead of the traditional HTTP requests over REST APIs. We'd like to ask you to re-design and migrate a front-end component of our dashboard application that is currently using REST APIs to get its data to use our GraphQL back-end server instead.

A.1. Initial Survey

- What is your current role?
- How many years of experience do you have as a front-end developer?
- How many years of experience do you have as a back-end developer?
- How familiar are you with Javascript? (1-5)
- How familiar are you with React? (1-5)
- How familiar are you with GraphQL? (1-5)
- How familiar are you with Apollo-GraphQL? (1-5)

A.2. Getting Familiar With The Approaches

In this section we will give you an oral tutorial on how to use the GraphQL server and how to write queries. If you still feel that you need more information to get familiarized with implementation of a GraphQL, please refer to the Apollo-Client and Apollo-Server documentations that can be found on Github. For the moment, Figure A.1. shows how a simple query and its response structure looks like.

A.3. Task Description

Step 1: Please go to our GraphQL server playground and familiarize yourself with the schema and query operations.

Step 2: Go to the "Backoffice Summary" page of the dashboard application. Familiarize yourself with the front-end structure and the network calls.

Step 3: Clone the "dashboard-app" repository to your local machine. Navigate to the "bo-summary page" section of the codebase.

<pre> query HeroNameAndFriends { heroes { name friends { name } } } </pre>	<pre> "data": { "heroes": [{ "name": "R2-D2", "friends": [{ "name": "Luke Skywalker" }, { "name": "Han Solo" }, ...], }, ...] } </pre>
--	--

Figure A.1.: A GraphQL query and its response.

Step 4: Migrate the "Pull Requests Widget" to using our GraphQL server as the data source. Our advice here is to transfer the stateful component to a functional component. The current GraphQL client best practices is along this line. However you are still free to utilize the GraphQL server as you wish.

A.4. System Usability Scale

	Strongly Disagree			Strongly Agree		
1. I think that I would like to use this system frequently	1	2	3	4	5	
2. I found the system unnecessarily complex	1	2	3	4	5	
3. I thought the system was easy to use	1	2	3	4	5	
4. I think that I would need the support of a technical person to be able to use this system	1	2	3	4	5	
5. I found the various functions in this system were well integrated	1	2	3	4	5	
6. I thought there was too much inconsistency in this system	1	2	3	4	5	

- | | | | | | |
|---|---|---|---|---|---|
| 7. I would imagine that most people would learn to use this system very quickly | 1 | 2 | 3 | 4 | 5 |
| 8. I found the system very cumbersome to use | 1 | 2 | 3 | 4 | 5 |
| 9. I felt very confident using the system | 1 | 2 | 3 | 4 | 5 |
| 10. I needed to learn a lot of things before I could get going with this system | 1 | 2 | 3 | 4 | 5 |

A.5. Interview Questions

- Describe the usefulness of schema introspection in solving the tasks.
- What challenges have you faced during the migration process?
- Have you encountered any problems that you couldn't solve that you could otherwise have solved using RESTful APIs?
- In what areas do you think using GraphQL is **MORE** convenient than using RESTful APIs?
- In what areas do you think using GraphQL is **LESS** convenient than using RESTful APIs?
- Would you use GraphQL in your future tasks?
- Do you have any additional remarks/thoughts on using GraphQL?

B. Transcripts of the Case Study Interviews

B.1. Interview with Participant #1 and Participant #2

Interviewer:

Recording. Perfect. Okay. Uh, thank you for doing this today. I appreciate it. So, let's just briefly ask you the questions now. Uh, starting from the first one, could you describe the usefulness of the schema introspection, uh, while you were doing this task? This is the GraphQL playground essentially. Having the access to documentation and schema, do you think it was useful?

Participant #2:

The playground itself, I think, was very useful. Um, the schema descriptions and the docs itself, I did not use so much today because I could come up with most of the things just playing around on the left side, in the playground thing with auto completion and everything.

Participant #1:

It was for me also very useful for solving this task because I just knew what I could ask for and whatnot. Yeah. The same for me since I knew what properties I can expect to be there. So I just played around and created my query and then just pasted the query into the source code and that helped a lot.

Interviewer:

Good to know, okay. So next, did you face any challenges while doing this? If so, what was it and what did you spend the most time on?

Participant #2:

For me it was really just implementation, no challenges that are related to GraphQL itself. I would say the biggest challenge, although it was not a big challenge was the syntax thing for me when I wanted to name the query or I wanted to pass variables, but this is just, I mean, even that went pretty well, um, considering that it was more or less the first time for me to use it. So yeah, the implementation challenges are related to the Apollo library or applying. So I cannot really say that it's, it has to do with GraphQL itself.

Interviewer:

Okay.

Participant #1:

Um, on my side, the GraphQL part was easy to do and since I use the upper level, uh, client, and it was almost ready to use out of the box. And then I just gave the client, the URL to the GraphQL server. It was just ready for use. And I just had some problems with refactoring, the existing code, uh, and, and, and, uh, I understood the, uh, component and its connection to the parent component broadly first. And by the way, it wasn't related to GraphQL. I didn't know that how the implementation, uh, was there, so normally I could familiarize myself with the code before jumping into GraphQL.

Interviewer:

Yeah. Yeah. Okay. That makes sense. So, have you encountered any problems that you couldn't solve that you could otherwise have solved using RESTful APIs?

Participant #2:

Not in this particular example. If I, if we keep thinking about this, of course, we might come up with some things, but in our today's one and a half hour experience, no, you know, no problems that I couldn't solve. And it was kind of a perfect example for using GraphQL, you know, everything was set up for it.

Interviewer:

Okay, then maybe you can answer this one. Uh, so in what areas would you think using this GraphQL server would be more convenient than using REST endpoints?

Participant #2:

Well, naturally in all the areas with multiple data sources. Of course. Yeah. So that's also why the dashboard app is perfect use case. Um, I mean, I don't know if that would make sense. Probably not, but you could probably write one query for the whole back office page, um, would be interesting to see the underlying optimizations that the server maybe does for it not to be like, wait too long for all the data to resolve. I don't know how that works because right now the widgets are loading independently. And if one widget has, has its data, then it's being rendered. What would happen if this would be a huge query? I don't know, but yeah. Multiple data sources for sure. Um, that's, that's the one thing that obviously comes to mind. I don't know. Um, I guess menu, uh, can, uh, let the client to be flexible with asking for queries and filter them, but sometimes you, you cannot do that, but if you are able to give the client flexible, uh, in terms of, uh, providing different queries for different use cases, gradually is the best fit, uh, or it's better actually than REST because you have to write different endpoints or different use cases.

Participant #1:

Uh, what else can I say having the schema just integrated with GraphQL environment is like a game changer, when we compare it to rest and having the documentation on the client side. I don't know if this is an advantage for GraphQL or for the whole, platform, I don't know if you're just comparing GraphQL and REST and it's not, maybe this is the schema part.

Interviewer:

It's a part of GraphQL schema yes.

Participant #1:

All right. If it comes with GraphQL, um, and I think as soon as you have a schema, you can run some kind of fancy UI on top of that. Be it the one that we saw today or Graphical, or, yeah.

Interviewer:

What about the other way around? In what areas do you think using GraphQL is less convenient than using RESTful APIs?

Participant #2:

So in general, what about, I mean, you have a lot of boiler plate, you have a lot of setup going on and yeah, for easy REST APIs with multiple types of data with multiple rest end points. So this might be overkill. Um, you have to come up with a schema and everything, you know, in general, in the area, I don't know about sensitive data authentication, things like this. I have, I don't, I just don't know, but I could also imagine that having such a, an open end point with, uh, playgrounds in the public more or less could also, yeah, at least have potential for some breaches, because it's easier to find out what data you can get and whatnot. Um, I don't know if you can, how easy it is to authorize, um, certain clients to get only certain data from the schema. You know what I mean? To, to not make the whole GraphQL thing, public, but yeah. Sort of user role management, the admin can see everything. Some other clients can only see some data, etc. I don't know if you can even define this.

Interviewer:

You can define this somehow but I don't know how to do it. We can also disable the playground on production.

Participant #1:

I think when we have services want to talk to each other, GraphQL is not handy, but for example, if you want to have a database and you want to generate some reports, then the GraphQL comes really handy, then you can customize the fields and types.

Interviewer:

Would you use GraphQL your future tasks?

Participant #1:

Why not in a personal project? I already did it too. Also, it has potential in this dashboard app.

Participant #2:

Yeah. Same for me. I also already used it a little. Yeah. Um, although in my part, I was just a consumer. I did not mingle around with schema and resolvers, but I would definitely use it in my next experience.

Interviewer:

Then it is time for the final question. Do you have any additional remarks on using GraphQL?

Participant #1:

Nothing specific, no.

Participant #2:

I guess if misused, it could lead to very bad results here and overload the servers, I guess. You really require some training, probably, for the people for getting used to it. Yeah, because it's so easy, you know, it's so prone to being misused, people can just add more fields to the query and apply heavier load to the server side.

B.2. Interview with Participant #3

Interviewer:

Could you please describe the usefulness of the schema introspection, meaning the GraphQL playground in doing this implementation?

Participant #3:

So, yeah, it looks much modern and fashion, API for building request over HTTP. Yeah. And, uh, I would say there are a lot of pluses and minuses, so pros and cons to use it or not. But, the main idea of GraphQL is, clear, it allows the users of the API to manage the requests as they need on the fly. And this is the main, uh, positive point from GraphQL. And, uh, I've seen it absolutely. When I build a request I've seen the list of data that I need and the response. So yest I see the data and I do not need any additional back-end to have access to the schema.

Interviewer:

I'm taking the question. Have you faced any challenges during this task?

Participant #3:

Not so much, but, uh, the main challenge was that, uh, our current code on the front end side was, written exactly under the structure of REST API. So I needed to change some paradigms of the code to work with the data from the GraphQL server. Not so much changes, but some of them were critical when I started working.

Interviewer:

In what areas do you think using GraphQL is more convenient than using RESTful APIs?

Participant #3:

No, both of these mechanisms are absolutely hundred percent on the market and they work. It means that I might absolutely on both APIs do the same. I might solve any issues I have using GraphQL or REST API. Um, the it's it's mean that I might use, I

might absolutely on both, uh, API APIs. I might do the same.

Interviewer:

Okay. Yes. Actually, you answered the next question too.

Participant #3:

Yeah, no, there is some direction of course where they differ from each other. There is some cases that GraphQL looks more interesting, sometimes not. GraphQL is very interesting when you need to manage data on the fly, in a flexible way. The classic example is the home page of Facebook, depending on the conditions of search, the data requested is flexible. On the other hand, when the data received will never be changed, maybe using REST APIs is more interesting in this case. Also, not a big problem but GraphQL can only POST requests and sometimes you might need to do something on the browser, so how do we receive data?

Interviewer:

With these points in mind, would you use GraphQL in the future?

Participant #3:

I would if I see there is an exact task for GraphQL features. For example if I need to be more flexible with data, where there are a lot of conditions when asking for data. It depends on the business. I do not think a transition needs to be done because it does not bring business value. As I said, it depends on the business processes and how much time I have, but I would rather use it on a fresh task if I absolutely need GraphQL features.

Interviewer:

Do you have any additional comments on using GraphQL?

Participant #3:

Uh, let's say GraphQL is still a young technology. My current employee enforces the rule that immature technologies should not be used on production. The technologies that are not checked by hundred percent of the market. We might use a little legacy technology but they are checked. They have stability, don't know exactly how to say it, but it's very important. Still, myself, I would absolutely use GraphQL if I start a project from scratch with no business pressure.

B.3. Interview with Participant #4

Interviewer:

Recording. Yes. Could you please answer the first question, and describe the usefulness of schema introspection in solving the tasks, meaning the GraphQL playground. So seeing the documentation and being able to write the query there, that's schema

introspection.

Participant #4:

Um, yeah, I find it very useful that I can, um, in the one window, or in the same place I can see schema. Um, also whenever I write the query I get already offered with all the possible arguments I can have or the fields of one entity. And I see, I find it very useful so that I can already try it. Because for example, if I do it within a normal environment, then I have to go to the model to see which fields it has in a separate file. Maybe a separate project. We don't have the playground at all normally, so yeah, I like it.

Interviewer:

Good. Thank you. So have you faced any challenges while doing this?

Participant #4:

Well, yes, but I would say this is just because I did it, like, the very first time in my life. I had to ask for help or search for help in Google. So, the queries was not very intuitive, but after I did it once, then I was able to do it quickly next time. So I didn't find any problems where I got stucked or where I couldn't continue. It was just something new to me also with, a little bit of query with the filter in general.

Interviewer:

Cool. Okay. Have you encountered any problems that you couldn't solve that you could otherwise have solved using RESTful APIs?

Participant #4:

Yeah, I didn't find any problems like that I couldn't solve. I cannot imagine this problem for now. Maybe if I use this more often, I will come up with something, but currently I didn't find any problems which I couldn't using GraphQL druing this small task.

Interviewer:

Okay. Uh, so now I'll ask you to compare GraphQL with REST. In what areas do you think using GraphQL is convenient than using RESTful APIs?

Participant #4:

Yes. For example, if we are talking about, let's say microservices, like our architecture, and if you have a UI, which is combining data from different sources and different microservices, then I really see benefit of GraphQL where we can define for front-end something like a view model, or whichever data we want. And I really liked that I can define the query in the playground. I can test it, try it and just copy into the code. It's really cool to see already which data source I'm using already in the view. So in the component, for example, so I don't have to go to service and then I see, I'm pulling this endpoint and I have to open the service separately to see what the endpoint is returning. Then I'd have to go to model to see. So, it's really clear what I have while using GraphQL. And I find it also useful that I can sometimes minimize the amount of fields I'm retrieving. With REST APIs, you just sometimes receive more than what you

need. So here I can really form the queries easily for a use case. I can just start for any use case, different things that I, yeah, I would say if I have, um, UI or at the front end, who is, uh, combining more data sources or who is, uh, using only smaller subset of, of bigger data source or some data formed in some specific way, I would, I would be happy to use graph QL. Also what I liked this a hook, which provides me this refresh, for example, if I need to regularly refresh, then I don't need to implement that myself.

Interviewer:

In what areas do you think using GraphQL less convenient than using RESTful APIs?

Participant #4:

Let me think about it. If you have a specific application which is really connecting to one backend, and you really don't want to expose to this application more data, REST would be better. So if you have a front-end and back-end where back-end is exposing RESTful endpoints for front-end, then I would say it is enough to use. So when you don't need to use data from multiple sources. Using GraphQL here would be over-engineering.

Interviewer:

Would you use GraphQL in your future tasks?

Participant #4:

Yeah. I like it. I would like to use it. We can try, we can think about using it in one of our future applications. In one of the new projects or existing small ones. I really like it. I think we don't need much time to get comfortable with that.

Interviewer:

Okay. Any additional remarks on using GraphQL?

Participant #4:

Updating data and not just reading data would be interesting to try. But maybe let's just try using it first, really. So we can get more ideas. Also it's interesting that it could be used for type safety. Right? So we can define the types. So this is another, let's say positive thing, because currently we don't have it on our RESTful endpoints.

B.4. Interview with Participant #5

Interviewer:

First of all, I would like to ask you to describe the usefulness of schema introspection while solving the task.

Participant #5:

Well, pretty useful, especially if you consider some people that never checked this project, that don't know how this entity looks like. So, this is already quite a good documentation. The names are more or less self explanatory. You basically don't actually need any kind of extra communication for this case. And still I saw that there was a

possibility to provide an extra documentation for the schema. So it was quite useful and at the same time you can also play there, as in, you can also execute queries and not just read the documentation.

Interviewer:

Have you faced any challenges while doing the task?

Participant #5:

Passing the arguments was not so straightforward to these functions. Like, for example, I was asking the question how it would be like if I want to execute several such functions where they have several arguments. What if they are conflicting and stuff like this. It was also not very straightforward with the schema explorer, but well, copying it into the code afterwards was fast. And then, I spent some time to understand how this hook function was working, but this is nothing special about GraphQL. Challenging was also like to understand, like, "Oh, okay. This object is nested. This is actually not a string, but a complex object". It took me a few executions to understand these kind of field structures in the schema.

Interviewer:

Did you see any problems that you could have solved if you used REST API to get data instead of GraphQL?

Participant #5:

Not a problem but, yeah with REST I would just ask the endpoint to give me all the data, so I would not have to think about what fields to ask for.

Interviewer:

Next, could you please compare REST APIs to GraphQL APIs with the task in mind?

Participant #5:

Um, okay, let me think. Interesting question. Of course they are comparable. It's not that I can say, in a certain case, one should definitely need to go for GraphQL, and because it cannot be done with REST. And also the other way around. They are more or less interchangeable I think. So, I'm just now trying to feel from this experience, yeah, so, probably for some, like, really simple entities, it would be easier to go with REST. If I had maybe like three or four fields that were not changing often, I would say using REST is better. For GraphQL I would say it would be probably easier if you have some kind of complex query to do, like, I don't know, five, six arguments, and maybe some kind of complex search which was not part of this example. In our applications when you need to pass several arguments of different types and of different structures, then you need to have a custom logic for REST to check them, to validate them that they are correct and allowed at all. With GraphQL we'd be able to easily do this validation without the need for extra code. And theoretically, not the case for the task that I just did, if you have different use cases for the same entities, it would be cool to use GraphQL to have these different combination of fields created according to what the applications require.

Interviewer:

Would you use GraphQL in your future tasks?

Participant #5:

Depends on the tasks, as the shift is quite big. It's not about just writing the queries, but more about how to approach problems, how to develop a system and dynamics to build features that just should be thought through differently. Uh, so I would try it in some simple projects to just have a look, if it will be not too much of an extra work to formalize these queries. I don't think it will be that much extra work on the server side but it might be extra work on the client side. I think you can generate the schema automatically and then just fix the mistakes.

Interviewer:

And any additional remarks on GraphQL?

Participant #5:

No, not at the moment, maybe later.

B.5. Interview with Participant #6

Interviewer:

Can you please describe the usefulness of schema introspection, meaning the GraphQL playground?

Participant #6:

I think it's very useful because you can solve the task quickly, just search for the different objects that are exposed to you. Hm. So it's really a playground, so trial and error, so you can just construct a query on the fly and then afterwards, once your query is working, you basically just need to copy and paste it into the string that you defined in your application. And you get some kind of, I mean, so it's kind of documentation done for you, right? I think it's already automatically produced by the server, right?

Interviewer:

Yeah, exactly

Participant #6:

So that's cool because it has a nice documentation interface that you can use, which is already produced when you develop the code.

Interviewer:

Have you face any challenges during this migration process?

Participant #6:

I mean, I needed to get a little bit familiar with how to use the processes of libraries and the query I had to look up myself how to do. So just taking more time because it

was the first time with it. So once we get it once or two times then there would be a challenge to do a similar migration. The query is quite specific. It's typed.

Interviewer:

Do you think you could have done anything better if you used REST instead of GraphQL in this task?

Participant #6:

I don't think so. Uh, nothing pops up into my mind. Maybe when you just think about the request, you said it's this only POST, so it might be more costly to the server. Because POST requests cost more than GET requests.

Interviewer:

Okay. So, in what areas do you think using GraphQL would be more convenient than using REST? And vice versa?

Participant #6:

I'm not sure I'm able to answer this question. I think when you just want to get data, it's really cool. I mean, you should have some, some complex logic behind that. So it may be easier to have a RESTful API because you want to be in control of what you have and what you do on the server side, but if you just want to get data, I think GraphQL is way more open to just getting data. I guess, you can do any kind of data manipulation that you want, right? You can sort the data, you can limit the data that you get. So, basically from the query side, you can control everything. Exactly. So, I think if you need something, which is equivalent to a GET request, it's very good. You can have either queries and mutations in GraphQL right?

Interviewer:

Yes, and subscription. But in our case, the dashboard application only uses GET so you never do a mutation. But yes, you can essentially to more than GET.

Participant #6:

Okay.

Interviewer:

Next, would you use GraphQL in the future?

Participant #6:

Yes. Good. I think it's specifically really good for a mobile app or a web app. So, both connected on the same endpoints. I think it makes things easy. And I think that was one of the reasons why it was invented for Facebook.

Interviewer:

Yes. Any additional remarks?

Participant #6:

I think it was a nice exercise. I would love to use GraphQL more and talk more extensive on it. I think the task was the right format for the interview, for two hours it was good and fun. Maybe I could have done it faster.

B.6. Interview with Participant #7

Interviewer:

Is it working? Yes. Perfect. Okay. So thank you. First of all, I would like to ask you to describe the usefulness of schema introspection, meaning the GraphQL playground, while doing the task today.

Participant #7:

Well, I don't know what kind of an answer you expect? I can say it's very useful to use this playground thing, it's just without this thing, it's much harder to do anything there. Right? I don't know where to find this kind of information, but this just shows you immediately what you're going to get there. So it's very, very easy.

Interviewer:

Yeah.

Participant #7:

Absolutely. It's a crucial part of this system.

Interviewer:

Um, have you faced any challenges while doing this?

Participant #7:

First of all, I needed to understand like how this single bit works, how the query syntax works and looks like. So there were some challenges there. So you need to understand a little bit from this whole thing to be able to use it. It's not that intuitive, at least from my point of view, even though I use JavaScript and I use these libraries. I needed to understand it a little bit. I think that's, that's a little bit looking at the outside of this. Right? So it's like, it gives you a little bit more complexity.

Interviewer:

Okay. Do you think you could have solves some challenges that you faced today by using REST instead of GraphQL?

Participant #7:

It would maybe have been a little bit more complicated. Right? So this, uh, Apollo GraphQL library gives you all these possibilities. You don't have to code yourself, right? And that's why, if you ask this question this way, I would say using GraphQL is simpler. It wouldn't be simple with the REST APIs. But overall, to answer your question, I would say no.

Interviewer:

Okay. Next, based on what you saw today, in which areas do you think using GraphQL would be more convenient than using REST? And vice versa?

Participant #7:

I would say for something that is a little bit more complex and a little bit more complicated, I would see some benefits of using GraphQL, and not for the very simple, "asking for two data fields" cases. It doesn't make so much sense to have this kind of, I don't know, critical capabilities. Right? But if you have complicated entities and you want to return the complicated data there, it might make more sense to query it different ways and things like that. I think that's what GraphQL was created for when you want, when you have more complicated model and you don't want to create all these end-points just to query it. You want things to be a little more dynamic right there. I think it's pretty useful. I think Facebook created it, right?

Interviewer:

Yes they did. So, what about the other way around? When would using REST be more convenient?

Participant #7:

So maybe it would be where you just expect some, compatibility. Right? So let's say you have all the clients and you have to integrate with someone. Maybe that someone doesn't know GraphQL, and now for those cases, you need to fall back to something what they know and maybe they know REST. And that's the easiest solution. Even for the systems, let's say there is a system, but it doesn't speak GraphQL. It just speaks REST. And in that system, you can just use REST for like legacy problems, legacy issues. But other than that, I would say, if you start to GraphQL, then use it like everywhere and just simply use it.

Interviewer:

Would you use GraphQL in your future tasks?

Participant #7:

I would like to use it, but I'm afraid that we don't have, at least for now, so complicated usages or APIs that are that complicated in any way that I see this kind of need to use something else. We barely have endpoints there. So it's like, not that of a big domain or something, but maybe later. Yes. But it's hard to see.

Interviewer:

Any additional remarks on GraphQL?

Participant #7:

I think this is an interesting experiment to see what are the options there. Especially for something like this dashboard thing for now, maybe I would like to see it somewhere else. But of course, I mean, if you think about this as always with every new technology,

it gives you something, but also gives you this complexity right now that you have to learn it and you have to explain it to people and things like that. And the real question is, are the benefits more than the disadvantages, right? Like, does it fulfill this whole thing? And then are the costs are less than the benefits? That's the question.

List of Figures

2.1. Types of APIs by availability (Boyd, 2014).	6
2.2. Microservices architecture diagram (Microsoft, 2019).	9
3.1. Parts of a GraphQL operation.	12
3.2. A GraphQL query and its response.	13
3.3. A GraphQL mutation and its response.	13
3.4. A GraphQL subscription and its response.	14
3.5. A GraphQL schema.	15
3.6. A GraphQL query depicting scalar types.	15
3.7. A GraphQL query depicting enum and non-null types.	16
3.8. Introspection and server response.	17
4.1. Three REST requests to fetch data.	20
4.2. A single GraphQL request to fetch data.	21
4.3. A front-end application communicating with its back-end services.	23
4.4. A standard architecture with one GraphQL server that connects to a single database.	23
4.5. GraphQL acting as a gateway.	24
4.6. GraphQL acting as a hybrid gateway.	25
4.7. Operations about "user" on an Open-API Specification schema interface.	26
6.1. Current state of the microservices architecture.	32
6.2. Proposed architectural change featuring a GraphQL gateway.	33
6.3. Declaration of the GraphQL server.	35
6.4. A sub-schema for monitoring pull-requests in the code repository.	36
6.5. Definition of the data source for the pull requests widget.	37
6.6. Definition of the data source for the pull requests widget.	37
A.1. A GraphQL query and its response.	50

List of Tables

7.1. Case Study Participants 39

Bibliography

- Levin, G. (Oct. 2015). *The Rise of REST API*. URL: <https://blog.restcase.com/the-rise-of-rest-api/>.
- Palmieri, D. (May 2018). *API, a sustainable competitive advantage through connection*. URL: <https://medium.com/yourapi/api-a-sustainable-competitive-advantage-through-connection-c966fc8895f7>.
- Jacobson, D., G. Brail, and D. Woods (2012). *APIs: A Strategy Guide*. Ed. by M. Treseler. Sebastopol: O'Reilly Media, p. 148. ISBN: 978-1-449-30892-6.
- Boyd, M. (2015). *Developing the API Mindset: Preparing Your Business for Private, Partner, and Public APIs*. Tech. rep. Nordic APIs, p. 96.
- Altexsoft (June 2019a). *What is API: Definition, Specifications, Types, Documentation*. URL: <https://www.altexsoft.com/blog/engineering/what-is-api-definition-types-specifications-documentation/>.
- Boyd, M. (Feb. 2014). *Private, Partner or Public: Which API Strategy Is Best For Business?* URL: <https://www.programmableweb.com/news/private-partner-or-public-which-api-strategy-best-business/2014/02/21>.
- Mitra, R. (Apr. 2015). *API Strategy 201: Private APIs vs. Open APIs*. URL: <https://apiacademy.co/2015/04/api-strategy-201-private-apis-vs-open-apis/>.
- Geissler, O. and U. Ostler (Nov. 2018). *What is a Remote Procedure Call - RPC?* URL: <https://www.datacenter-insider.de/was-ist-ein-remote-procedure-call--rpc-a-712873/>.
- Techolution (July 2019). *What are the Different Types of APIs?* URL: <https://techolution.com/types-of-apis/>.
- Merrick, P., S. Allen, and J. Lapp (2006). *XML REMOTE PROCEDURE CALL*.
- Box, D., D. Ehnebuske, I. Gopal Kakivaya, A. Layman, N. Mendelsohn, L. Development, C. Henrik, F. Nielsen, S. Thatte, and D. Winer (2000). *Simple Object Access Protocol (SOAP) 1.1*. Tech. rep. Microsoft. URL: <http://www.w3.org/TR/2000/NOTE-SOAP-20000508Latestversion>:<http://www.w3.org/TR/SOAPAuthors>.
- Avraham, S. B. (Sept. 2017). *What is REST — A Simple Explanation for Beginners, Part 1: Introduction*. URL: <https://medium.com/extend/what-is-rest-a-simple-explanation-for-beginners-part-1-introduction-b4a072f8740f>.
- Fielding, R. T. (2000). "Architectural Styles and the Design of Network-based Software Architectures". PhD thesis. UNIVERSITY OF CALIFORNIA.
- Martin, R. C. (2014). *Agile software development, principles, patterns, and practices*. Pearson Education Limited, p. 524. ISBN: 1292025948.

- Singh, J. (2018). *The What, Why, and How of a Microservices Architecture*. URL: <https://medium.com/hashmapinc/the-what-why-and-how-of-a-microservices-architecture-4179579423a9>.
- Microsoft (Oct. 2019). *Microservices architecture style*. URL: <https://docs.microsoft.com/en-us/azure/architecture/guide/architecture-styles/microservices>.
- Moran, J. W. and B. K. Brightman (2000). "Leading organizational change". In: *Journal of Workplace Learning* 12.2, p. 9. URL: <http://www.emerald-library.com>.
- Gill, R. (2003). "Change management — or change leadership?" In: *Journal of Change Management* 3.4.
- Andreo, S. and J. Bosch (2019). "API management challenges in ecosystems". In: *Lecture Notes in Business Information Processing*. Vol. 370 LNBIP. Springer, pp. 86–93. ISBN: 9783030337414. doi: 10.1007/978-3-030-33742-1_{_}8.
- Macvean, A., M. Maly, and J. Daughtry (May 2016). "API design reviews at scale". In: *Conference on Human Factors in Computing Systems - Proceedings*. Vol. 07-12-May-2016. Association for Computing Machinery, pp. 849–858. ISBN: 9781450340823. doi: 10.1145/2851581.2851602.
- Murphy, L., M. B. Kery, O. Alliyu, A. Macvean, and B. A. Myers (2018). "API Designers in the Field: Design Practices and Challenges for Creating Usable APIs". In: IEEE. ISBN: 9781538642351.
- The GraphQL Foundation (n.d.[a]). *Introduction to GraphQL | GraphQL*. URL: <https://graphql.org/learn/>.
- Facebook (2018). *GraphQL Specification*. URL: <https://spec.graphql.org/June2018/>.
- The GraphQL Foundation (n.d.[b]). *Code | GraphQL*. URL: <https://graphql.org/code/>.
- ECMA (2017). *ECMA-404: The JSON Data Interchange Syntax*. Tech. rep. Geneva, Switzerland: ECMA International.
- Stubailo, S. (2017a). *The Anatomy of a GraphQL Query*. URL: <https://blog.apollographql.com/the-anatomy-of-a-graphql-query-6dffa9e9e747>.
- (2017b). *The next step for realtime data in GraphQL - Apollo GraphQL*. URL: <https://blog.apollographql.com/the-next-step-for-realtime-data-in-graphql-b564b72eb07b>.
- Wittern, E., A. Cha, J. C. Davis, G. Baudart, and L. Mandel (July 2019). "An Empirical Study of GraphQL Schemas". In: *IBM Research*. URL: <http://arxiv.org/abs/1907.13012>.
- The GraphQL Foundation (n.d.[c]). *Schemas and Types | GraphQL*. URL: <https://graphql.org/learn/schema/>.
- Apollo GraphQL (n.d.[a]). *Unions and interfaces | GraphQL*. URL: <https://www.apollographql.com/docs/apollo-server/schema/unions-interfaces/>.
- The GraphQL Foundation (n.d.[d]). *Introspection | GraphQL*. URL: <https://graphql.org/learn/introspection/>.
- Eschweiler, S. (Mar. 2018). *REST vs. GraphQL Blog - Medium*. URL: <https://medium.com/codingthesmartway-com-blog/rest-vs-graphql-418eac2e3083>.

- GraphQL Community (n.d.[a]). *GraphQL is the better REST*. URL: <https://www.howtographql.com/basics/1-graphql-is-the-better-rest/>.
- Altexsoft (Mar. 2019b). *GraphQL: Core Features, Architecture, Pros and Cons*. URL: <https://www.altexsoft.com/blog/engineering/graphql-core-features-architecture-pros-and-cons/>.
- Herrera, E. (Sept. 2018). *5 reasons you shouldn't be using GraphQL*. URL: <https://blog.logrocket.com/5-reasons-you-shouldnt-be-using-graphql-61c7846e7ed3/>.
- Stubailo, S. and D. Man (Oct. 2016). *Navigating your transition to GraphQL with GraphQL first development*. URL: <https://www.slideshare.net/sashko1/navigating-your-transition-to-graphql-with-graphql-first-development>.
- GraphQL Community (n.d.[b]). *How to GraphQL - The Fullstack Tutorial for GraphQL*. URL: <https://www.howtographql.com/>.
- OpenAPI Initiative (July 2017). *OpenAPI Specification (Version 3.0.0)*. URL: [https://github.com/OAI/OpenAPI-Specification/blob/master/versions/3.0.0.md/](https://github.com/OAI/OpenAPI-Specification/blob/master/versions/3.0.0.md).
- Wittern, E., A. Cha, and J. A. Laredo (2018). "Generating GraphQL-wrappers for REST(-like) APIs". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 10845 LNCS. Springer Verlag, pp. 65–83. ISBN: 9783319916613. DOI: 10.1007/978-3-319-91662-0{_}5.
- Siddiqui, A. (Sept. 2018). *Authentication vs Authorization*. URL: <https://medium.com/datadriveninvestor/authentication-vs-authorization-716fea914d55>.
- Saring, J. (May 2019). *13 GraphQL Tools and Libraries You Should Know in 2019*. URL: <https://blog.bitsrc.io/13-graphql-tools-and-libraries-you-should-know-in-2019-e4b9005f6fc2>.
- Apollo GraphQL (n.d.[b]). *Introduction to Apollo Server*. URL: <https://www.apollographql.com/docs/apollo-server/>.
- (n.d.[c]). *Authentication*. URL: <https://www.apollographql.com/docs/react/networking/authentication/#header>.
- Wawhal, R. (Aug. 2018). *The ultimate guide to Schema Stitching in GraphQL*. URL: <https://hasura.io/blog/the-ultimate-guide-to-schema-stitching-in-graphql-f30178ac0072/>.
- Brooke, J. (2013). "SUS: A Retrospective". In: *Journal of Usability Studies* 8.2, pp. 29–40.
- Sauro, J. (Feb. 2011). *Measuring Usability with the System Usability Scale (SUS)*. URL: <https://measuringu.com/sus/>.
- Claes Wohlin, Per Runeson, Martin Bost, Magnus c. Ohlsson, Bjorn Regnell, and Anders Wesslin (1996). *Experimentation In Software Engineering - An Introduction*. Kluwer Academic Publishers, p. 213. ISBN: 0792396537.