Hochschule Konstanz
Department of Computer Science

Integration of API Data Query Languages in Microservice Architectures

GraphQL and Falcor as Alternatives to REST

Johannes Burghardt

Konstanz, 09.05.2018

Master's Thesis

MASTER'S THESIS

to achieve the academic grade of

Master of Science (M. Sc.)

at

Hochschule Konstanz

University of Applied Sciences

Department of Computer Science

Study Course Software-Engineering

Topic: **Integration of API Data Query Languages in Microservice Architectures**

Master's Candidate: Johannes Burghardt

1. Supervisor: Prof. Dr. Ralf Schimkat

2. Supervisor: M.Sc. Simon Endele

Registration Date: November 9th, 2017

Submission Date: May 9th, 2018

Abstract

In the past few years the internet has seen a shift in usage, volume and device diversity. To keep up with the need for fast and efficient data transfer new technologies like GraphQL and Falcor have emerged, offering new ways of retrieving data by introducing a query language for APIs. Microservices have been proven to be a viable architectural style that provide a decentralized way of developing applications, with REST being the de facto standard for microservice communication. This work shows how integrating a data query language can decouple services to a greater degree and the workflow benefits it entails. By analyzing the drawbacks of REST a comparison with GraphQL and Falcor is carried out and by studying microservice architectures knowledge for a prototype is being established, which shows why GraphQL is a better fit for these architectures and shows how a integration and a migration can be done.

Ehrenwörtliche Erklärung

Hiermit erkläre ich *Johannes Burghardt, geboren am 19.10.1990 in Dsheskasgan*, dass ich

(1) meine Masterarbeit mit dem Titel

Integration of API Data Query Languages in Microservice Architectures

bei der Seitenbau GmbH unter Anleitung von Prof. Dr. Ralf Schimkat selbständig und ohne fremde Hilfe angefertigt und keine anderen als die angeführten Hilfen benutzt habe;

(2) die Übernahme wörtlicher Zitate, von Tabellen, Zeichnungen, Bildern und Programmen aus der Literatur oder anderen Quellen (Internet) sowie die Verwendung der Gedanken anderer Autoren an den entsprechenden Stellen innerhalb der Arbeit gekennzeichnet habe.

Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben wird.

Konstanz, 09.05.2018

(Unterschrift)

Contents

1 Introduction

This chapter gives an introduction into this thesis. It briefly describes the current trend toward efficient data fetching, the drawbacks that come with Representational State Transfer (REST) and why technologies such as *GraphQL* and *Falcor* have emerged. It then describes the underlying problems and contributions and the approach that has been taken in this work. The last section gives an outline by summarizing each chapter.

1.1 Background

In the past few years the internet has seen a shift in usage volume and device diversity. Nowadays more media is consumed, and more sites are visited from mobile devices. In 2016 the mobile market took over the desktop market in terms of usage and this number only as grown ever since [Sta16]. However, as more and more media is consumed from mobile devices, the need for efficient data fetching to increase performance and save bandwidth rises. Therefore, developing efficient communication and data exchange between server-client applications is more important than ever.

With the rise of the microservice architecture and the adoption of this style by well-known companies, the way data is exchanged in software changed [Fow]. REST is currently the de facto standard when developing a communication layer between services [hub17; Alm+17] [Min17, p. 17]. But new technologies place new demands. Mobile and Web applications require various data, that often needs to be tailored to their specific needs. Supplying the required data with REST poses some problems. Server round trips are frequent and over- and underfetching are common problems when fetching data. Roy Fielding, the creator of the REST paradigm, mentioned the following in his dissertation:

> *"The REST interface is designed to be efficient for large-grain hypermedia data transfer, optimizing for the common case of the Web, but resulting in an interface that is not optimal for other forms of architectural interaction" [Fie00, p. 82].*

Facebook and Netflix encountered some of these problems with REST in their software. Facebook was frustrated with the differences between the data they wanted and the server queries they required when rebuilding their mobile applications [Byr15]. Netflix realized, that their REST code became too complicated and they wanted a simpler solution for caching and retrieving data [Hus15]. Both companies started working on a solution independently, with the motivation to tackle the drawbacks they encountered with REST, resulting in two Application Programming Interface (API) data query lan-

guages: Facbook's GraphQL and Netflix's Falcor[1]. The implementation of such API data query languages is not completely new: Feed Item Query Language (FIQL) is a feed item query language proposed back in 2007 and can take arbitrary request [Not07] and Open Data Protocol (OData) is an established standard that defines best practices to consume REST data [Mic15]. However, GraphQL and Falcor provide a modern approach to retrieve and manipulate data.

The Seitenbau Knowledge & Collaboration Solutions (KCS)-Team is currently in the redevelopment phase of Social Office Net (SON) an Intranet portal with a microservice architecture. This platform is build upon many microservices relying on REST communication. The KCS faced the same difficulties that Facebook and Netflix encountered. The question rises, whether the aforementioned drawbacks can be mitigated with the help of an API data query language.

1.2 Problems and Contributions

Implementing new technologies into existing microservice architectures poses problems on the effect such technologies have on the architecture and each service, especially technologies that provide a new approach in their respective area. Such an implementation therefore requires a structured and reproducible approach. One objective of this paper is provide architectural guidelines on how a API data query language, such as provided by GraphQL and Falcor, can be implemented into an existing system. This will furthermore solve questions as to the impact such technologies have on the architecture as well as on the workflow of a team. Therefore the following questions need to be answered:

How can an API data query language be implemented into an already existing microservice architecture and what impact do they have?

1. In which possible ways can a data query language for APIs be implemented into an existing micro-service architecture?
2. Evaluation of GraphQL and Falcor to decide which technology better fits an universal approach.
3. How can existing services be migrated into this schema.
4. Can a data query language for APIs encapsulate microservices even further?
5. What impact does a data query language for APIs have on the workflow?

[1] Both GraphQL and Falcor will be described in detail in chapter 2

1.3 Approach

By researching REST in the context of microservices, drawbacks that are tightly coupled with the usage of REST are highlighted. Analyzing these drawbacks and comparing them to solutions provided by GraphQL and Falcor will give insights on the functionality of these new technologies and how they differ from REST. By researching the microservice architectural style knowledge on how microservices are designed, modelled and how they communicate will be acquired. With the established knowledge on both these topics a prototype will be build. This prototype is modelled after a real-life scenario provided by the KCS SON microservice architecture. A suitable API data query language will be implemented into this prototype with the help of a requirement analysis.

1.4 Outline

This section briefly describes the structure of this thesis. It does so by giving a short summary of each chapter.

Chapter 2 - Terminology

This chapter describes the basic terminology and concepts that are necessary to understand this work.

Chapter 3 - Drawbacks of REST in Comparsion to GraphQL and Falcor

This chapter analyzes the drawbacks of REST and shows how GraphQL and Falcor solve these problems. This chapter shows the different approaches these technologies take.

Chapter 4 - Microservice Architecture

By researching the microservices the principles and benefits behind this style will be made apparent, focusing on proven methods in architecture and API design.

Chapter 5 - Prototype

With the knowledge established in chapter 3 and 4, a requirements analysis for a microservice prototype is carried out. According to these requirements a design concept and a prototype will be created. By means of this prototype the central questions of this thesis will be answered.

Chapter 6 - Evaluation and Discussion

This chapter evaluates the process in Chapter 5 and explains the results in reference to previous research.

2 Terminology

This chapter describes the basic terminology and concepts that are necessary to understand this work. It first describes Service-oriented architecture and microservices. It then describes the definition of an API and elaborates on what REST, GraphQL and Falcor are.

2.1 Service-Oriented Architecture

The Service-oriented architecture (SOA) Manifesto describes SOA as a type of architecture that results from applying service orientation [Ars+09]. It is an architectural style that supports service-orientation by providing a service to other applications or services[2] trough a communication protocol over network. A service can be viewed "as a discrete unit of functionality that can be accessed remotely and acted upon and updated independently." [PF13, p. 346]. According to [Ope16] a service can be described with the following characteristics:

1. a service is a logical representation of a repeatable business activity.
2. services are self-contained.
3. a service may be composed of other services.
4. a service is a "black box" to consumers of the service.

SOA does not provide a single architectural or a technological guideline. There are in fact many ideas on how to interpret[3] SOA. However, it provides an architecture that can be viewed as a technology-neutral approach [Erl+13].

2.2 Microservices

The microservice architecture can be viewed as a modern interpretation of SOA [Dra+16]. Fowler and Lewis describe the architectural style of microservices as "an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms." [LF14]. To further modernize this concept Fowler and Lewis describe characteristics which serve as guidelines for this architectural style, including decentralization of governance and data management, as well as infrastructure automation.

[2] Any application that uses a service, be it a service itself, is commonly called service-consumer
[3] Fowler talks about this ambiguity of the SOA definition in a blog post from 2005, see: [Fow05]

	SOA	Microservices
Scope	Enterprise-wide	One project
Flexibility	by orchestration	by fast deployment and rapid, independent development
Organization	Services are implemented by different organizational units	Services are implemented by teamsin the same project
Deployment	Monolithic deployment of several services	Each microservice can be deployed individually
UI	Portal as universal UI for all services	Service contains UI

Table 2.1: Differences of SOA and microservices by Wolff [Wol16, p. 92]

Although SOA and microservices have synergies, they differ in their scope, flexibility, organization, deployment and in their user interface (UI). Wolff describes these differences between SOA and microservices in [Wol16]. The summary of these differences can be seen in table 2.1. Wolff concludes that while both SOA and microservices split applications into services, SOA aims at flexibility through orchestration of services at the enterprise IT level and microservices focus on deployment and parallel work on different services [Wol16].

2.3 Application Programming Interface (API)

An application programming interface (API) is in general a set of strictly defined methods and rules for communication between software. It provides developers with standard commands that can be used to leverage the functionality a given software exposes. APIs can exist for various systems, including web-based, operating or database systems. While generally conforming to the same purpose, different APIs differ in how they work. To narrow down the scope of this terminology the definition which is relevant to this thesis will be used. In the context of microservices an API is the functionality a service exposes to other services or consumers through an endpoint. This exposed endpoint can be either an REST API, a messaging interface or a modern approach as provided by GraphQL and Falcor.

2.4 Representational State Transfer (REST)

Roy Fielding describes REST as an architectural style for distributed hypermedia systems, which “is a hybrid style derived from several of the network-based architectural styles [...] and combined with additional constraints that define a uniform connector interface“ [Fie00, p. 76]. To derive REST, Fielding uses six constraints. They do not determine the implementation, however, they do determine the characteristics a RESTful

service should have [Fie00]:

1. *Client-Server*: separation of concerns, meaning the separation of the user interface from the data storage concerns
2. *Stateless*: each request from client to server must contain all necessary information to understand the request. Session state is kept on the client
3. *Cache*: data within a response to a request must be implicitly or explicitly labeled as cacheable or noncacheable
4. *Layered System*: allows an architecture to be composed of hierarchical layers by constraining component behavior such that each component cannot "see" beyond the immediate layer with which they are interacting
5. *Code-On-Demand*: client functionality can be extended by downloading and executing code in the form of applets or scripts

The sixth constrain set by Fielding, the Uniform Interface, is the central feature that distinguishes the REST architectural style from other network based styles [Fie00]. It consists of four constraints which are found in a single remark:

> *REST is defined by four interface constraints: identification of resources; manipulation of resources through representations; self-descriptive messages; and, hypermedia as the engine of application state [Fie00, p.82].*

To fully understand the concept of this constraint, the terms **resource** and **representation** need to be explained. In the context of REST a resource can be any information that can be named, e.g a document, a temporal service or a collection of other services [Fie00, p. 88]. Each particular resource must have a Uniform Resource Identifier (URI) to identify its location. Furthermore, a consumer only interacts via a URI with a resource, which is called representation. Amundsen and Richardson further describe the details of these four constraints [RA13] as the following:

1. *Identification of Resources*: resources are identified by a URI. The resource's state may change, but its URI stays the same.
2. *Manipulation of Resources Through Representations*: On the Web, clients and servers manipulate resources by sending representations back and forth using a small set of standardized HTTP methods (GET and POST).
3. *Self-descriptive messages*: A message either contains or links to all information necessary for the recipient to understand it.

4. *Hypermedia as the engine of application state (HATEOAS)*: allows a smart client to automatically adapt to changes on the server side and a server to change its underlying implementation without breaking all of its clients. This constraint is the payoff for obeying the other constraints.

2.5 GraphQL

GraphQL is a data query language internally developed by Facebook in 2012. In 2015 Facebook began open-sourcing GraphQL by drafting a Request For Comments (RFC) specification [Byr15]. Despite its name, GraphQL is not a query language solely for graphs but instead is a language to query application servers that have capabilities defined in the specification. It does not mandate a particular programming language or storage system [Fac16].

GraphQL uses a schema system to describe types, relations and entry points. Clients can declarativly create queries and receive only data that was requested in a single request, even if more than query points have been declared. Figure 2.1 shows a schema which declares the entry point, called the root query, and a queryable user. Starting from the entry point anything the schema exposes can be queried, even deep nested queries are possible.

```
1 type Query {
2     user(id: Int): User
3 }
4
5 type User {
6     id: ID!
7     name: String
8     number: Int
9 }
```

Listing 2.1: GraphQL schema

In contrast to REST, which is a architectural style, GraphQL is a query language with a strict specification. This specification describes strong type system, validation and execution rules. Although still in development, it is the backbone of the GraphQL language. In this specification five design principles are mentioned [Fac16]:

1. *Hierarchical*: Queries are structured hierarchically and are shaped just like the data they return.

2. *Product-centric*: GraphQL is unapologetically driven by the requirements of views and the front-end engineers who write them.

3. *Strong-typing*: Every GraphQL server defines an application-specific type system. Given a query, tools can ensure that the query is both syntactically correct and valid within the GraphQL type system before execution and the server can make certain guarantees about the shape and nature of the response.

4. *Client-specified queries*: Through its type system, a GraphQL server publishes the capabilities that its clients are allowed to consume. It is the client that is responsible for specifying exactly how it will consume those published capabilities.

5. *Introspective*: A GraphQL server's type system must be queryable by the GraphQL language itself.

The GraphQL language and community follows some guidelines not defined in the specification. It is protocol agnostic by specification and does not mention how versioning should be done. However the consensus on this topic is that GraphQL is typically served over Hypertext Transfer Protocol (HTTP) and should remain versionless [Gra18b; Gra18a; Byr16a].

2.6 Falcor

Falcor is a data fetching library developed by Netflix. It allows for modelling backend data as a single JavaScript Object Notation (JSON) object. This JSON object can then be retrieved on the client side and can be queried with familiar JavaScript operations [Net15c]. Falcor, unlike GraphQL, is not a specification but a middleware. A middleware is a function between a raw request and the final response of a route [Exp14].

Falcor implements a Model System, where clients do not use JSON data directly. Instead they work with the JSON data received by a Falcor Model. The primary difference between working with an JSON object and a Falcor Model is that a Falcor Model has a asynchronous API. This model can then be used to query parameters just like a plain JSON object.

A Falcor Model can be made available trough the network via DataSources [Net15a] and handles all network communication with the server [Net15b]. To ensure an efficient communication, Falcor applies following optimization:

1. *Caching*: Maintaining a configurable in-memory cache of values previously queried. If a new request is made for information, which is available in the cache, the data will be retrieved from the cache [Net15b].

2. *Batching*: Batching multiple small requests into a single large request. This can significantly improve performance when using a network protocol with a high connection cost [Net15b].

3. *Request Deduping*: If a request is made for a value, which already has a pending request, no additional request is made [Net15b].

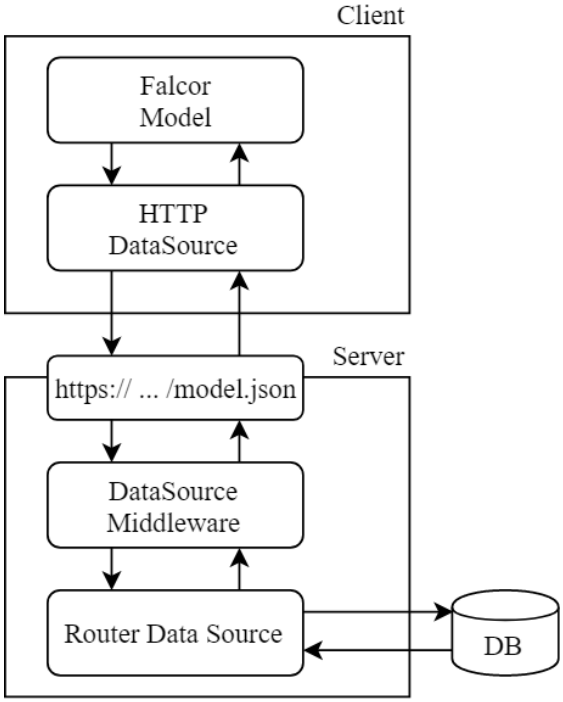

Figure 2.1: Abstraction of the Falcor architecture

Falcor introduces a new convention for modeling graph information in order to map data to real life examples. JSON Graph extends JSON with three new primitive types: references, atoms and errors. A reference, comparable to a symbolic link in the UNIX file system, can point to another location in the same JSON Graph. Atoms allow for metadata to be attached to values. Errors can be placed in the JSON Graph when an error occurs while setting or retrieving data [Net15d]. Modeling application domains as graphs helps to prevent the occurrence of duplicates and to prevent stale data [Net15d].

3 Drawbacks of REST in Comparison to GraphQL and Falcor

The following chapter describes drawbacks that occur when retrieving data with REST. It will more into detail what these drawback are and how GraphQL and Falcor tackle such problems. This will give insight on how these technologies work and in what way they differ from REST. As mentioned in the Introduction, Fielding states that REST is designed for large-grain hypermedia data transfer, but not for other architectural styles. Newer technologies require a fine grained transfer of data, making REST not the optimal solution for the communication layer.

3.1 Over- and Underfetching

Overfetching occurs when fields are declared in a retrieved resource which are not used. Underfetching occurs when fields are not declared in a resource but are used in the template. This is a common pitfall in REST, as there is often only a certain endpoint for a resource. Listing 3.1 shows a possible result from a REST call.

```
1 {
2   "name": "Bob",
3   "age": "20",
4   "number": 1231234
5 }
```

Listing 3.1: Possible result from a REST call

Calling this route to get the name or the number of the user will result in unnecessary data being sent, as the client has to filter the properties it does not need, resulting in overfetching data. Although in the given example the overfetched data would be mere bytes, in real world examples this can potentially accumulate to a much higher degree. Seeing as routes will change over time, the information they provide is not necessarily static. Endpoints as depicted in listing 3.1 might change to contain the address of the user, resulting in future REST calls fetching more data on this endpoint. Poorly formed queries will therefore have a major impact on performance and data usage.

Underfetching data will result in not all required properties being fetched and thus often requiring another network call to gather the missing information, causing multiple roundtrips before all data is collected. Solutions to restrict over and underfetching are

creating new REST endpoint which delivers the exact information the client needs or add query parameters to the route, which would violate the uniform interface constraints. This requires a close coordination between provider and consumer and changes on both sides.

Declarative Data Fetching with GraphQL

GraphQL exposes a single endpoint through which a client can request and manipulate data. Giving the client the ability to declaratively specify the requested data results in an over- and underfetching resistant design. Listing 3.2 shows a query that requests the **name** and the **age** property.

```
1 {
2   user(id: 4) {
3     name
4     age
5   }
6 }
```

Listing 3.2: GraphQL query specifying user data

Underfetching can be mitigated as the client can specify all required information in a single query, receiving the data in a single request. Adding data to the endpoint will not affect the data that the client will receive. Overfetching data is not possible as only the properties specified in the query will be sent.

Declarative Data Fetching with Falcor

Falcor exposes a single endpoint for clients. By declaring what properties to query in the client, it is resistant to over- and underfetching. Listing 3.3 shows how the same data can be requested with Falcor. Although the syntax is different, the same benefits and mitigation techniques that GraphQL leverages are found in Falcor.

```
1 model.get(["user", 4, ["name", "age"]])
```

Listing 3.3: Falcor model query specifying user data

3.2 Type System

As REST is an architectural style, it does not adhere to a strict type system. Fieldings dissertation does not mention any Create, read, update and delete (CRUD) methods because "the the HTTP methods are not part of the REST architectural style but of the Web's architecture definition" [Fie09]. REST can describe the content it provides in the HTTP Content-Type Header response, but this is limiting. Listing 3.4 shows a possible response from a GET request. The only information the response is providing is what kind of data the request will deliver and the format. In order to know the structure of the response, the data itself has to be viewed.

```
1 GET      /api/user/123     HTTP/1.1
2
3 HTTP/1.1 200 OK
4       Connection:Keep-Alive
5       Content-Encoding:gzip
6       Content-Type:application/json; charset=utf-8
```

Listing 3.4: Response to a REST GET request

There are systems that extend the REST architectural style.Web Application Description Language (WADL) and Web Services Description Language (WSDL) make it possible to map a type system to REST endpoints in the form of Extensible Markup Language (XML). These two description languages are competing, resulting in no definite standard in the REST style. But no standard leads to too many standards[4], making the lacking type system of REST a drawback.

GraphQL's Type System and Schema

GraphQL provides a strongly typed Type system described in the GraphQL specification. Queries have to satisfy the given type system defined by a GraphQL server. GraphQL's type system supports inheritance, interfaces, lists, custom types, enumerated types and has built in scalars [Fac18e], an example is provided in listing 2.1. This enables the consumer of an GraphQL endpoint to know what type of data it will receive and what the shape of the data will look like.

Typeless Faclor System

Falcor uses JSON Graph, an extended JSON notation, to model data. Containing no types, Falcor is schema and typeless by design.

[4] A reason why a lot of REST APIs are called "Restish" or "Restful".

3.3 API Discovery

The uniform interface constraint states that resources are to be identified by an URI. When developing applications that require a granular approach for retrieving data creating multiple endpoints for resource will be the outcome. This however leads to problems with the discoverablity of the endpoints. Listing 3.5 shows a single endpoint with different operations for retrieving, creating and updating data. These operations are not apparent from the routes itself.

```
1 GET     api/users/:username     HTTP/1.1
2 POST    api/users/:username     HTTP/1.1
3 PUT     api/users/:username     HTTP/1.1
```

Listing 3.5: Different meanings of a single REST endpoints

REST is not self documenting therefore every new endpoint has to be manually documented with the according operations, valid inputs and results. When developing large applications this will quickly result in a big overhead.

GraphQL's Introspection

One of GraphQL's design principles included in the specification is the introspection. Due to the nature of the type system it is possible to ask a GraphQL schema for information about what queries it supports, inputs it requires and what results those queries will deliver [Fac18f].

```
1 {
2   __type(name: "user") {
3     fields {
4         name
5     }
6   }
7 }
```

Listing 3.6: Introspecting field information with GraphQL

Listing 3.6 shows how a GraphQL endpoint can be asked to show its root query types, those fields are always available on the root. Discovering what a GraphQL endpoint can return is therefore integrated in the GraphQL specification. This means that documentation can be automated.

Discovery with Falcor Model

Falcor exposes its data through a single endpoint. This is either done by wrapping a Falcor Model as the Datasource or by a Falcor Router, which matches the incoming routes to the Falcor Model data. The client has to know what paths are available. But with its Model and Router Falcor is exposing internal code, it is therefore only suitable for internal use of a team, not a public API.

3.4 Summary

In this chapter the drawbacks of REST have been shown. When developing large applications that need to evolve quick and efficiently and REST lacks in areas of efficiently fetching data, documentation and its type system. Over and Underfetching is a big problem where REST needs alot of fine tuning and requires changes in server and client side to work efficiently. By letting the client decide what data to fetch GraphQL and Falcor solve this problem elegantly. The lack of a type system in REST and Falcor make it difficult to anticipate the data that will be delivered. Furthermore, it is not possible to know the shape of the requested data in REST. GraphQL offers a strongly typed system which enables custom objects. This type systems and its schema enables automated documentation of a GraphQL endpoint, which is not possible with Falcor or REST.

	REST	GraphQL	Falcor
Nature	architectural style	Specification	JavaScript library
Protocol	HTTP	Protocol agnostic	Protocol agnostic
Endpoint	multiple	single	single
Over-/Underfetching	vulnerable	resistant	resistant
Type system	weakly typed	strongly typed	none
Documentation	manual	integrated	manual

Table 3.1: Comparison of REST, GraphQL and Falcor

Table 3.1 shows a comparison of these technologies as described in the terminology and in this chapter. It should be noted that these drawbacks of REST shown here in this chapter are not all problems with REST itself, but constitute from its usage. All drawbacks can be solved with existing tools and technology, as described in this chapter, but require the developer to put more effort into the development and making compromises as there are not always standard solutions. GraphQL and Falcor do not address problems that REST cannot solve but they make these problems easier to solve.

4 Microservice Architecture

This chapter explores microservice architectures and its principles to establish knowledge on how such services can be constructed. By first examining modelling guidelines and core theories the important aspects of such architectures will be shown. Afterwards proven architecture patterns and API designs are analyzed.

4.1 Microservice Principles

Microservice architecture describes a "particular way of designing software applications as suites of independently deployable services" [LF14]. In contrast to a monolithic architecture, where an application is built as a single unit, an application in a microservice architecture consists of a suite of small services, each running its own process and communication with lightweight mechanisms [LF14]. Problems with the monolithic approach, such as scaleability, resilience and deployment lead to the rise of microservices. Figure 4.1 shows the monolithic approach on the left side, where a container always consist of the same big application. The microservice approach on the right side shows how a container can consists of a variation of small services, each of which can be deployed separately.

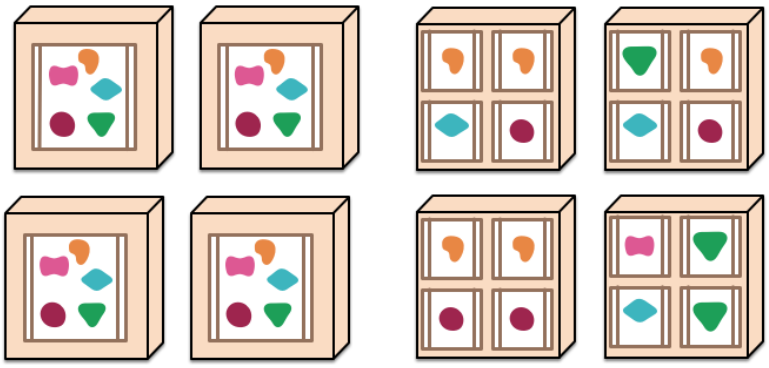

Figure 4.1: Left: monoliths, right: microservices, adopted from [LF14]

There is no formal definition on the architectural style of microservices, but there are common key features that can be observed in architectures that fit the label. Not all characteristics will be met by all architectures, nor do they have to. These characteristics lay the common ground of the microservice architectural style. Fowler and Lewis describe characteristics of microservices in [LF14] and Sam Newman lists characteristics in [New15a]. The relevant characteristics for this work are the following:

1. *Componentization via Services*: Componentization and modularity is achieved via

breaking down systems into services, which then are replaceable, upgradeable, and deployable independently.

2. *Smart Endpoints and Dump Pipes*: Microservices favour the approach of lightweight communication mechanisms that can decouple services as much as possible. The two most commonly used protocols are HTTP request-response and messaging [LF14].
3. *Decentralized Governance*: The microservice architecture does not follow the tendency to standardize the technology stack used in an application.
4. *Evolutionary Design*: Due to service componentization the microservice architecture enables frequent changes in a system.
5. *Autonomous*: by Newman, services need to be able to change independently of each other, and be deployed by themselves without requiring consumers to change.

There are many and varied benefits to microservices. According to Newman, microservices achieve these benefits to a greater degree due to how dedicated they are in achieving these characteristics. Microservices with these characteristics are typically associated with following benefits [New15a]:

1. *Technology heterogeneity*: Allows different technologies to be used by different components in the system.
2. *Resilience*: Componentization provides clear boundaries between services, making them more resistant as failures can be isolated.
3. *Scaling*: Enables scaling in every axis of the scaling cube [AF15].
4. *Ease of deployment*: Changes to a single service can be deployed independently.
5. *Organizational alignment*: smaller teams working on smaller codebases.
6. *Composability*: Enable the creation of new capabilities by composing and re-using existing services.

Microservice architecture therefore follows the UNIX philosophy of "do one thing and do it well" [S03]. Keeping services as small and decentralized as possible is a key principle. The next sections will cover how to model and design microservices.

4.2 Modelling Microservices

To further analyze microservices this section looks deeper into the theory of how microservices are modeled and the proven methods when design. This will provide insights on the important parts when designing the structure of microservices.

4.2.1 High Cohesion and Loose Coupling

A central modeling trait of microservices are high cohesion and loose coupling[5]. These terms are used alot in the context of object oriented systems. In the context of microservices Newman attributes them the following meaning [New15a]:

Loose Coupling

Changes to one service should not require changes to another service. Loose Coupling is a central part of the microservice architecture as microservices should be able to be changed and deployed autonomously. Tight coupling can occur when services are bound together, meaning they have to tightly work with another service which prevents a service from acting as a instance on its own.

High Cohesion

Related behaviour should be grouped together. High cohesion helps when introducing changes, as behavioural changes can be made in one place instead of many. High cohesion helps with refactoring and code management. Low cohesion happens when related behaviour is spread across multiple services.

4.2.2 The Bounded Context

An important task when designing a microservice architecture is the size of the services. Microservices should be small enough so that the benefits of the microservice architecture can come to effect, but should be big enough to not permanently interact with other microservices to complete a task. To determine the size of a microservice, the concept of Bounded Context is often recommended to serve as a point of orientation [Nad+16]. Fowler calls the Bounded Context a central pattern in Domain-Driven Design (DDD), which "is about designing software based on models of the underlying domain" [Fow14].

The goal is to determine boundaries of subsystems, which are called components, in the context of a larger system. This helps in finding the maximum size a component should take, but still can lead to large components. To determine how small a service can be the aggregate pattern, originating from DDD, can be used. Evans describes an aggregate as "a cluster of associated objects that are treated as an unit for the purpose of data changes." [Eva14, p. 511].

[5] Often called low coupling.

4.2.3 Conway's Law

Melvin Conway observed that "Any organization that designs a system (defined more broadly here than just information systems) will inevitably produce a design whose structure is a copy of the organization's communication structure." [Con68].

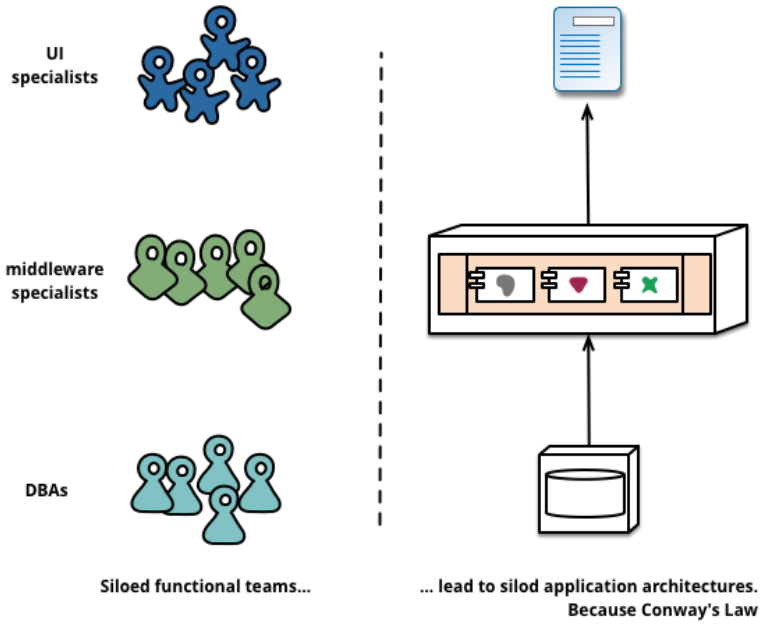

Figure 4.2: Conways Law as depicted by Fowler and Lewis in [LF14]

When designing applications and composing services, these services will inherently look like the organizational structure provided by the teams who develop them. By structuring a team in a specific way it can be ensured that only team members working on the same service or on the same interface need to coordinate together. This allows for domain architecture to not only be implemented by the distribution of microservices, but also by the organizational distribution [Wol16]. This approach comes with difficulties [Wol16]:

- The use of different technologies creates a high entry barrier, having stable teams is therefore encouraged.
- Separating teams per service means that only the team understands the service, and when members leave knowledge is lost.
- Changes that require coordination of multiple teams are difficult.

Wolff further describes that large organizations have numerous communication relationships which can affect the architecture, stating that: "In the end, having too many communication relationships is a real risk for a project." [Wol16, p. 36].

4.3 Microservice Patterns

This section looks into API patterns when designing microservices. The principles described in 4.1 and the theory given in 4.2 already established a fundamental understanding of how such services can be created and that each application has a unique microservice structure. Therefore, only proven architectures that can be applied to a broad range of applications will be mentioned here. When analyzing [Nad+16], [New15a] and [Wol16] one pattern was mentioned in every book, the API gateway pattern. A variation of this pattern, the Backends for Frotnends pattern, is also mentioned by Newman in [New15a].

4.3.1 API Gateway

The API gateway is a common pattern used in most microservice implementations. It helps in securing endpoints, orchestration and routing [New15a]. Modern API gateways are also used for transformation and service discovery. Figure 4.3 shows the API gateway pattern. A client can connect to the gateway without the knowledge of the underlying microservice infrastructure and its functionality and can only access the endpoints the gateway provides, having to direct connection to any service.

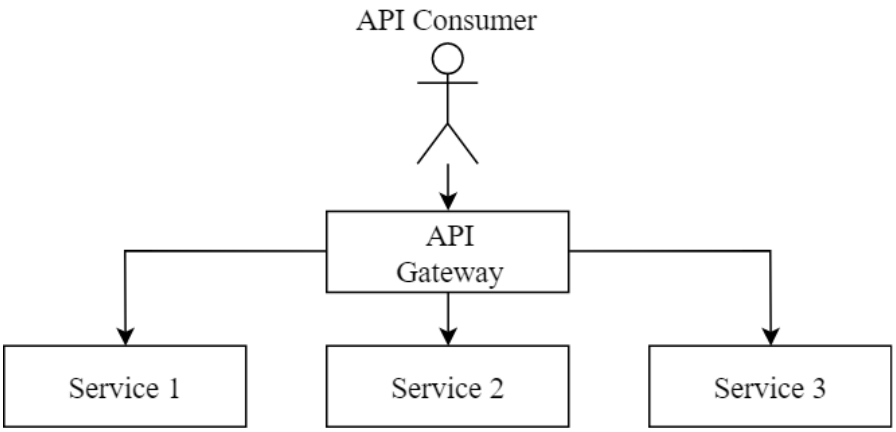

Figure 4.3: API gateway pattern

This separation of external and internal API comes with the following benefits [Nad+16]:

- **Security**: By only letting the API provided by the gateway face the public, it is easier to secure those endpoints.
- **Transformation and Orchestration**: The gateway can serve to transform endpoint calls or batch them together. This is in general what an aggregation layer does.

- **Routing**: Hides the complexity of underlying systems from client applications by routing the client to the correct service when an external URI associated with the microservice is requested.

The gateway pattern is often used to aggregate information. Microsoft calls it the gateway aggregation pattern [NW17]. Applications built around a large number of small services have a higher amount of cross-service calls, aggregating information is therefore one solution to tackle the chattiness between client and server applications.

4.3.2 Backend for Frontend

Newman introduced the Backend for Frontend (BFF) pattern, stating that "Backends For Frontends solve a pressing concern for mobile development when using microservices." [New15b]. This concern comes from requirements of chatty interfaces, which may need content depending on the type of device. An API gateway aggregates data for general purposes, which leads to one layer that provides data for all consumers. But mobile devices and desktop applications often need slightly different data to display the same UI. This can either lead to multiple requests to a service or to an aggregation layer with duplicate code for certain device types. To tackle this problem Newman suggest the Backend for Frontend pattern, where the different device types can connect to a separate gateway which is specifically tailored to the respective device type.

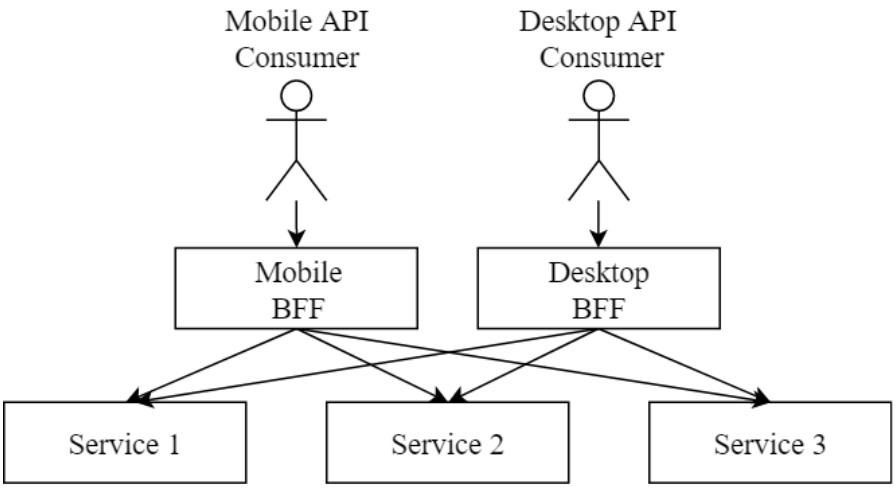

Figure 4.4: Backends for Frontends pattern

Figure 4.4 shows how different device types connect to the services trough different gateways. The BFF pattern separates the generic gateway pattern by creating a gateway for each device. This tackles problems that can occur when having one large layer for all types of consumers, by allowing to decouple functionality for a certain device and allowing separate teams to work on one BFF.

4.4 API Design for Microservices

A central part of a service is its communication and the API it provides for other consumers. Just like the structure of a service, the API a service provides needs to be modular to gain the advantages of the microservice architecture and to independently deployable on its own [Nad+16].

4.4.1 Messaging-Oriented

Messages provide an asynchronous inter-service communication. Irakli Nadareishvili et al. say on this topic: " By adopting a message-oriented approach, developers can expose general entry points into a component (e.g., an IP address and port number) and receive task-specific messages at the same time. This allows for changes in message content as a way of refactoring components safely over time." [Nad+16, p. 67]. Wolff attributes messaging various advantages in distributed systems [Wol16]:

- Messages are resilient to network failures as the system will deliver messages when the network is available again.
- Messaging is asynchronous, making this approach well suited in an architecture where high latency times can occur as a call to another service does not block further processing.
- Messaging is enabled trough a queue. The sender has no knowledge of the recipient, meaning sender and recipient are decoupled.

Messaging is often used for inter-service communication as they provide a fast and lightweight transportation of data. Being asynchronous in terms of inter-service communication is important, as it will not block other services.

4.4.2 Hypermedia-Driven

Hypermedia-driven APIs are often used in the context of microservices. They extend the REST style by not only providing the requested data but by also providing possible actions like links and forms. These Hypermedia-style APIs are also known as Hypermedia as the Engine of Application State (HATEOAS) APIs, as described in section 2.4. It embraces evolvability and loose coupling by allowing as described by the HATEOAS principles [Nad+16]. With the help of these links the client can navigate to other resources by only knowing the URI provided by the response. According to the Richardson Maturity Model a HATEOAS APIs can be considered the final form of REST [Fow10].

```
1 {
2     "name": "Alice",
3     "links": [ {
4         "rel": "friends",
5         "href": "http://localhost:8080/user/3"
6     } ]
7 }
```

Listing 4.1: HATEOAS-based Response

Listing 4.1 shows how a HATEOAS-based response can look like, containing the **rel** attribute indicating a relationship and the **href** attribute that provides a link to the next resource. Clients can progressivly discover the API by following these links. A downside is that this can be data consuming as the client needs to request each link to discover other entry-points.

4.5 Versioning

Handling many microservices poses problems with versioning, as a single breaking change in an interface can potentially move through all depending services. On the topic of versioning Newman and Fowler both state that versioning should be avoided as long as possible and that the best way to reduce breaking changes is to avoid making them [New15a, p. 62] [Fow13]. When talking about versioning external and internal APIs need to be differentiated. Internal APIs that are developed by the same team can be considered special when it comes to versioning, as teams can work closely together when changes need to be implemented [Wol16]. When introducing changes to external interfaces that interact with another team or the public in general, coordination is more complicated. Newman describes techniques when versioning is inevitable, including catching breaking changes early and using semantic versioning [New15a].

4.5.1 Semantic Versioning

To describe changes to an external interface version numbers can be used. Semantic Versioning is a widely used method, which describes how versioning of public APIs can be done. It splits the version number in a "MAJOR.MINOR.PATCH" format [Pre17]:

- MAJOR changes indicate incompatible API changes.
- MINOR version is changed when functionality is added in a backwards-compatible manner.
- PATCH is increased in the case backwards-compatible bug fixes.

Wolff states that when using REST the version should not be encoded in the Uniform Resource Locator (URL) as the URL should represent a resource independent of the version. He suggest that the version can be defined in an Accept header of the request, thus following the principles of HATEOAS [Wol16].

4.5.2 Postel's Law

On the topic of versioning and communication Postel's Law, also known as robustness principle, is often mentioned. Postel's Law is a design guideline for software that states: "Be conservative in what you do, be liberal in what you accept from others." [Pos81, section 2.10]. In the context of microservices this means that a service exposing an API should comply to a strict set of rules, while an API consumer should be as tolerant as possible with regard to what they are accepting. Decoupling can never be completely achieved as services still have to communicate with each other. Following Postel's law when designing interfaces can therefore help with evolving APIs and interoperability [New15a; Fow11].

4.5.3 Coexisting Endpoints

If breaking changes can not be avoided, limiting the impact they have on a running system is important. Forcing API consumers to change or upgrade their API is not ideal. Always maintaining the ability to release microservivces independently is the goal.

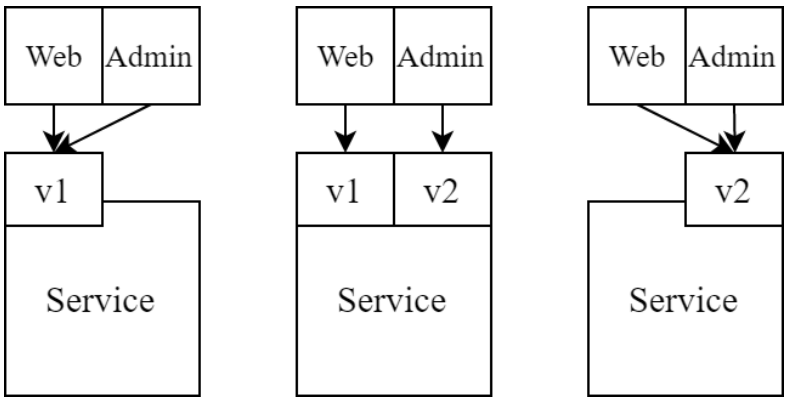

Figure 4.5: Migration with coexisting different endpoint, by Newman [New15a].

Newman describes an approach on how to migrate services with a breaking change. By letting old and new endpoints coexist in the same running service, consumer can gradually migrate, while still maintaining functionality [New15a]. Figure 4.5 shows how such a migration to a newer API can be carried out. Once all consumers are moved to the new endpoint the old endpoint can be removed. This requires that endpoints can be differentiated. When using REST this is either done via a version number in the request header or with a version number in the URL, for example **/v1/user/**.

4.6 Summary

The microservices architectural style is an approach to developing a single application as a suite of small services [LF14]. This chapter showed the proven methods and theories when designing and modelling microservices. By following the characteristics listed in 4.1 certain benefits such as scaling, ease of deployment and composability can be achieved. An integral part of a microservice architecture is decoupling, be it in regard to modelling services or to communication. But complete decoupling can never be achieved as services still have to communicate together. According to Conway's Law the architecture of a software system will resemble the organizational structure, therefore creating well structured teams benefits the architecture design and minimizing unnecessary communication further helps this architecture. REST is currently the de facto standard when developing microservices. But developing large applications with REST poses problems due to the drawbacks described in chapter 3. Seeing as new technologies emerged which tackle these drawbacks, the question rises on how well they integrate in an microservice architecture. The next chapter will look deeper into these problem.

5 Prototype

The previous chapter has shown that a central principle of the microservice architectural style is decoupling. The goal is to achieve decoupling in the modelling process, when designing the architecture and modelling an API. Seeing as REST is currently the de facto standard in microservice communication, the question rises whether an API data query language offers a better solution. The drawbacks of REST and how these new technologies tackle these drawbacks have been shown in chapter 3. To answer the questions of this work, this a prototype has been created, with the help of the knowledge established in the previous chapters. "Prototyping is a technique for providing a reduced functionality or a limited performance version of a software system early in its development [...]." [Sca01]. Nielsen describes two dimensions of prototyping as depicted in figure 5.1. A horizontal prototype is providing a broad view of a system while keeping all the features of a system but eliminates all depth of functionality. A vertical prototype provides a small view of a system that gives full functionality for a few features [Nie93].

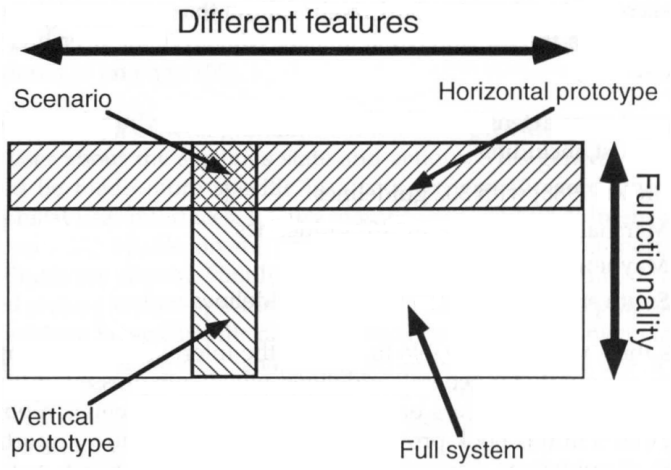

Figure 5.1: Two dimensions of Prototyping, by Nielsen [Nie93]

A horizontal prototype does not deliver the necessary depth of a system to be considered a proof of concept. Therefore, a vertical prototype with a microservice architecture will be created, providing the needed functionality and a set of features that can be analyzed. This prototype will resemble an existing microservice architecture and will be used to integrate an API data query language. To create this prototype, a reference application based on a real-life example will be used. Figure 5.2 shows parts of the SON microservice architecture that focuses on the retrieval of personal messages. The figure shows microservices that manage data of a specific domain. The User-Service contains all information about users. The Page-Service contains different pages, with titles and authors.

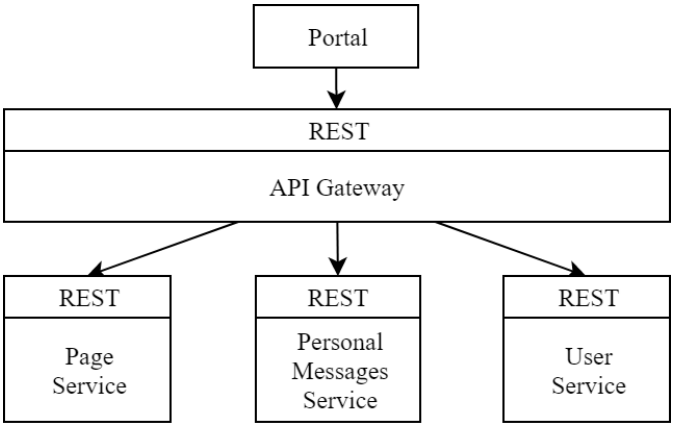

Figure 5.2: Extract of the SON microservice architecture

The personal messages service contains Messages from one user to another, with content, type and a link to a page. The API Gateway is the orchestration layer, bundling all endpoints with REST. The Portal retrieves data trough the Gateway.

5.1 Requirements

This section will carry out a requirements analysis, an important task that gathers all necessary requirements for a project. The importance of this analysis is highlighted by the CHAOS report from 1995 by the Standish Group, where incomplete requirements attribute to 13,1% of failed projects [Gro95]. The ISO-29148 standard describes an iterative process where the stakeholder requirements, a requirements analysis and the architectural design are all applied together to a system [29111]. [Def01] describes the procedure of a requirements analysis with the 15 tasks provided by the Institute of Electrical and Electronics Engineers (IEEE) P1220-standard. It is also noted that: "As with all techniques, the student should be careful to tailor; that is, add or subtract, as suits the particular system being developed." [Def01, p. 41]. A requirements analysis is further used to develop requirements, which can be split into functional and non-functional requirements. Functional requirements specify what a system should do, whereas non-functional requirements specify how a system should perform certain tasks. According to these points, the requirements analysis will be structured into the following parts:

1. Stakeholder identification
2. Determining Goals
3. Determining an API Data Query Language
4. Designing Scenarios with User Stories

5.1.1 Stakeholder Identification

To capture the necessary requirements, the stakeholders of the project need to be identified first. According to ISO29148 a stakeholder is an "individual or organization having a right, share, claim, or interest in a system or in its possession of characteristics that meet their needs and expectations." [29111]. Stakeholders in the context of the SON architecture scenario were identified by consulting with the KCS team. Table 5.1 shows the identified stakeholders in accordance with the KCS team, which includes users of the system, backend and frontend developers, the software architect as well as the product owner.

Identifier	**Stakeholder**
S1	User
S2	Backend Developer
S3	Frontend Developer
S4	Software Architect
S5	Product Owner

Table 5.1: Identified Stakeholder in the context of the SON

After stakeholders are identified their concerns need to be analyzed. Furthermore, they need to be prioritized in order to better base future decisions on certain stakeholders.

5.1.2 Determining and Prioritizing Goals

To determine goals, stakeholders will be interviewed. Their main concerns will be taking into account and goals will be derived from these concerns. As this work focuses on the architecture of microservices, performance will not be considered[6]. To determine goals, the following questions can be used [BD99]:

- What are the tasks that the actor wants the system to perform?
- What information does the actor access? Who creates that data? Can it be modified and by whom?
- Which external changes does the actor need to inform the system about? How often and when?
- Which events does the sytem need to inform the actor about? With what latency?

Interviewing stakeholders will give clarity over their concerns in context of the SON architecture. Arranging them according to the Power/Interest Grid from table 5.3 will

[6]Papers which analyzed performance of GraphQL and Falcor are mentioned in the discussion.

result in a prioritized goal list. Table 5.2 shows each stakeholder with their main concern. Backend Developers need to provide all necessary data to the Frontend team. The Frontend team wants to efficiently retrieve the data that is necessary. The System Architect is aiming for a stable architecture. The User wants the Personal Messages to be displayed, how this is done is not relevant to the User.

Priority	**Stakeholder**	**Concern**
High	Software Architect	Stable architecture
High	Backend Developer	Provide data that is necessary
High	Frontend Developer	Efficiently retrieve data
Normal	Product Owner	Rapid development
Low	User	Retrieve and send personal messages

Table 5.2: Identified Stakeholder goals in the context of the SON architecture

The Power/Interest Grid provides a solution on how different stakeholders can be prioritized [Sha10]. It divides groups into the following categories [Tho]:

- *Manage Closely*: High power, highly interested people.
- *Keep Satisfied*: High power, less interested people.
- *Keep Informed*: Low power, highly interested people
- *Monitor*: Low power, less interested people.

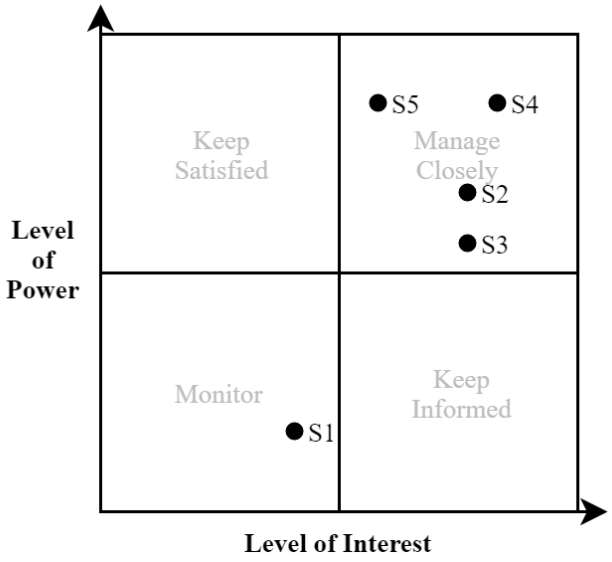

Figure 5.3: Prioritization of Stakeholders

Stakeholders can be prioritized according to these groups. This was done by consulting each stakeholder. Table 5.3 shows the prioritized Power/Interest Grid[7]. Users rank among the *Monitor* group, as they do not have a high level of power considering how the architecture of a system is designed. However, they have an interest in a working product. Backend and frontend developers have high power and a high level of interest in the architecture and the workings of a system, as they bear responsibility for the system. Backend developers however have more power over the system as they provide and manage the data of each service. The Software Architect ranks the highest in power and interest, as they make high-level design choices including the architecture and coding standards. The Product owner, while not as highly interested in the architecture as the developers or the system architect, has interest in a rapid development of features.

5.1.3 Determining an API Data Query Language

Choosing a technology for communication is a vital part in microservices, as it is required for a communication to loosely couple services. This thesis introduced GraphQL and Falcor, new technologies that emerged due to the drawbacks of REST. Table 5.3 shows a comparison of GraphQL and Falcor based on table 3.1.

	GraphQL	Falcor
Nature	Specification	JavaScript library
Protocol	Protocol agnostic	Protocol agnostic
Endpoint	single	single
Over-/Underfetching	resistant	resistant
Type system	strongly typed	none
Documentation	integrated	manual

Table 5.3: Extract of Table 3.1

GraphQL and Falcor are tackling the same problems: querying and managing data. While they both achieve to compensate the drawbacks of REST, there are differences. Falcor does not provide a type system, making it more error prone[8]. GraphQL not only has a type system, but this type system is leveraged to automatically generate documentation about the API. The biggest drawback lies in their nature: while GraphQL is a specification with implementations for a wide variety of languages[9], Falcor is a JavaScript library. Although the microservice architectural style allows for various technologies to be used simultaneously, having a communication technology decide the language of a

[7] It should be noted that this prioritization is influenced by the KCS-Team as their structure provides the point of reference.
[8] Advantages of a type system include safety, optimization and easier refactoring of code.
[9] Including C#, Clojure, Elixir, Erlang, Go, Groovy, Java, JavaScript, PHP, Python, Scala, Ruby.

microservice is not ideal. While both technologies are protocol agnostic, Falcor only provides an implementation written in JavaScript and with no strict specification, whereas GraphQL provides many implementations. Another factor is the community and the available information. After following both of these technologies for the last months, a big trend towards GraphQL can be observed[10]. For the purpose of this thesis the following prototype will be implemented with the reference implementation of GraphQL in JavaScript.

5.1.4 Determining Scenarios with User Stories

To describe requirements, scenarios in the form of user stories are used [Sut98]. User stories are constructed with natural language, making them easy to understand and can describe a concrete example from an user perspective [Sut03]. In this case, a user story is a requirement expressed from the perspective of a stakeholder and describes a functionality that will be valuable to a stakeholder [Coh04]. The format of a User Story often follows the following principle [Con14]:

1. As a *stakeholder/role*
2. I need *requirement/feature*
3. So that *goal/value*

Structuring user stories this way helps to focus on the viewpoint of a specific role. It helps to define high level requirements without going into detail by using language that can be understood by the role it is written for and by the developers that implement them. Besides the description an user story contains acceptance criteria and can be of functional and non-functional nature [Con14]. The following user stories are taken from the SON Platform, thus resembling real specifications.

User Story A	As a user I want the SON Portal to display my messages, so that I can access them.
Acceptance Criteria	A1. Initially only messages of type recommendation will be shown. A2. Data contains Name of the sender, linking to the profile. A3. Data contains title of the Page, linking to the page. A4. Data contains the date the recommendation was sent

Table 5.4: SON User Story A: receiving personal messages

The first user story in table 5.4 is a creation story, building the base of the prototype by implementing the basic feature of receiving personal messages. The second user story in

[10] Activity of both open Github Repositorys, Stackoverflow questions and Google Trends.

User Story B	As a user I want to send personal messages so that I can interact with other users.
Acceptance Criteria	B1. Sending a message with content to another user is possible. B2. Displaying the personal message is possible B3. Displayed messages should contain the sender, content and date.

Table 5.5: SON User Story B: sending personal messages

User Story C	As a software architect I want to migrate the existing services to standardized API.
Acceptance Criteria	C1. All involved services are migrated. C2. Code redundancy should be minimized. C3. API for each service is documented. C4. The number of calls to each service should be minimized.

Table 5.6: SON User Story C: migration of the corresponding services

table 5.5 builds on the first story and extends its functionality with sending messages. The third user story in table 5.6 is a migration task from the existing SON architecture with REST to one with a modern API query language. These user stories have been chosen because they have a combined scope that affects all stakeholders. The first two user stories will show how the backend and frontend will work with GraphQL and if subsequent stories can be developed quicker, which is the concern of the product owner. The third user story is a migration task from the software architect perspective. This feature set allows for a proof of concept to be build.

5.2 Integration

To demonstrate how the previous user stories can be implemented, the integration process of GraphQL will be described in this section. This will provide details on how the integration and migration process of an API data query language, in this instance GraphQL, into the existing SON System can be accomplished. It will go into detail on how this process can be executed, as well as describing the changes, benefits and drawbacks that come with it. The Implementation of GraphQL into a service follows the following steps[11]:

1. *Schema creation*: Creating a schema that resembles the data provided by the service. This can be accomplished with the GraphQL Schema Definition Language (SDL).

[11]The exact code can be viewed in the Attachments

2. *Create resolvers*: A resolver is a function that implements the logic of a root query. They must return the type defined in the schema.

3. *Serving over HTTP*: Define and serve a GraphQL service over a single endpoint.

5.2.1 Schema Creation

Creating a schema in GraphQL can be done pragmatically or with the SDL. Both methods are valid and described in the specification. The SDL is less verbose as the programmatic approach, while offering the same functionality. When creating a GraphQL schema the data a service provides needs to be inspected. Listing 5.1 shows a sample of user data provided by the user service.

```
1 GET     /api/v1/user/:id     HTTP/1.1
2 /* Response */
3 {
4   "userID": 3,
5   "name": "Ester Lindsey",
6   "email": "esterlindsey@recrisys.com",
7   "profilePage": "www.google.de"
8 }
```

Listing 5.1: Data provided by the user service of user with id 3

The provided data from the REST endpoint can be remodelled into a GraphQL schema, while following the GraphQL specification and providing an appropriate type for each field. Types can be either scalar or custom types. Listing 5.2 shows the user type in a GraphQL schema using the SDL.

```
1 type User {
2     userID: ID
3     name: String
4     email: String
5     profilePage: String
6 }
```

Listing 5.2: Modelling the user service in GraphQL SDL

Every GraphQL schema must provide a root query. Root queries are the entry points of a schema and define what functionality a GraphQL service provides and how it will be called. While both queries and mutations are requested with the HTTP POST method, queries provide data and mutations change data. The REST endpoint of the user service contains a single route with a **:id** parameter. This parameter can be modelled into the

GraphQL schema and be taken as an argument by a query. Listing 5.3 shows the root query of the user service, where the field can be queried with a scalar **Int** argument. This will provide information on how to access the endpoint, what information the endpoint requires and what the consumer can expect to be returned.

```
1 type Query {
2     "retrieve single user information"
3     user(id: ID!): User
4 }
5 type User {
6       ....
7 }
```

Listing 5.3: Root Query of the user service

To manipulate data a root mutation field has to be modelled in the SDL. It describes an entry point in the GraphQL service where data can be manipulated. Like queries, they provide all information needed to the consumer. To extend a schema with the ability to manipulate data, a new type is required. When working with existing REST services, they typically use the HTTP POST, PUT or PATCH method to manipulate data. Listing 5.4 shows the POST route of the personal message service.

```
1 POST      /api/v1/personalMessage/      HTTP/1.1
2 /* Request Body */
3
4 {
5   "from": 4,
6   "to": 3,
7   "content": "Hello User 3"
8 }
```

Listing 5.4: POST route for creating a personal message

Listing 5.5 shows how the POST route can be converted into a root mutation field. The exclamation marks imply required arguments. While a REST route typically returns a status code, GraphQL uses JSON based responses for everything. Thus, a boolean type can be returned to indicate a successful manipulation. In case of a failure an error message should be returned in the error field. It is important to differentiate between writing and reading operations, which are split into queries and mutations in GraphQL. “While query fields are executed in parallel, mutation fields run in series, one after the other.” [Fac18g].

```
type Mutation {
    sendPersonalMessage(from: Int!, to: Int!, content: String!): Boolean!
}
```

Listing 5.5: Mutation field for sending a personal message

5.2.2 Create Resolvers

After a schema is created and the root queries and mutations are defined, the logic can be implemented. Resolvers map the entry points of a GraphQL schema (queries and mutations) to functions. These functions are separated from the schema and can execute arbitrary code, making them ideal for the business logic layer. Resolvers must return the type as specified in the Schema, thus forcing the strict type system on to each query. A resolver function receives the following arguments [Fac18d]:

- *obj*: The previous object. Not used for a field on the root query.
- *args*: The arguments provided to the field by the GraphQL query.
- *context*: A value provided to each resolver. Can hold information about the currently logged in user or other authorization tokens.

```
var resolver = {
    personalMessages: ({userID}) => {
        /* business logic */
    } ,
    sendPersonalMessage: ({toUser, content}) => {
      /* business logic */
    }};
```

Listing 5.6: resolvers of the personal message service

With the context argument it is also possible to pass HTTP request parameters to a resolver, allowing for authorization to be implemented for each resolver. As resolvers are responsible for the logic on how data will be retrieved, handling errors is part of their responsibility. It is in the hand of the developer to decide how to return data. The GraphQL specification only describes the resulting body of the response, which should contain all valid data in a data field and end all errors in an error field [Fac16].

5.2.3 Serving over HTTP

GraphQL, although protocol agnostic, is typically served over HTTP. This has advantages when working with existing REST architectures. Authentication does not need to be re-implemented, as it can continue to be done over HTTP [Fac18a]. Where REST often implements authorization on an endpoint level, with GraphQL this logic will move to the resolvers. Another advantage is the usage of *GraphiQL*[12], an integrated browser plugin for exploring GraphQL endpoints. To serve GraphQL through a single endpoint, the schema and its corresponding resolvers are combined. Listing 5.7 shows how a service can be served with express.js[13].

```
1 app.use('/graphql', graphqlHTTP({
2     schema: schema,
3     rootValue: resolver,
4 }));
```

Listing 5.7: Serving GraphQL in express.js

5.2.4 Microservice Overview

An overview of the resulting architecture[14] of a GraphQL service is shown in figure 5.4. The service exposes the queries and mutations over a single endpoint, making them callable over HTTP. Any query or mutation a service exposes can be called.

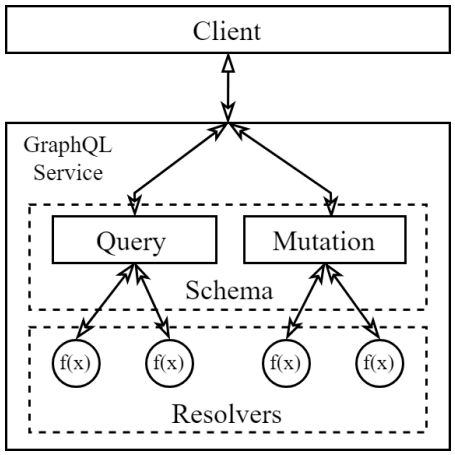

Figure 5.4: Overview of a GraphQL service architecture, adopted from [Rhy17]

The service resolves the clients request by calling the respective resolver function, which will return the type defined in its schema. What can be observed is that the schema

[12]GraphiQL is an Integrated development environment (IDE) that is bundled with GraphQL HTTP frameworks. It can be seen as an automated API documentation.
[13]Implementation in other language are similar.
[14]This overview is valid for any GraphQL service, regardless of the used protocol to serve the service.

and resolvers are separated. While such separation can be accomplished with REST, GraphQL forces developers to separate the API description and the business logic, complying with the high cohesion principle of microservices.

5.3 GraphQL as Gateway

Section 4.3.1 described how the gateway pattern is a core concept in the microservice architectural style, which is also used in the SON architecture. Data provided by several services has to be combined and transformed. Letting service communicate directly with each other would violate the loose coupling principle of microservice, as described in 4.2.1. The task of aggregating data therefore has to be moved to another layer.

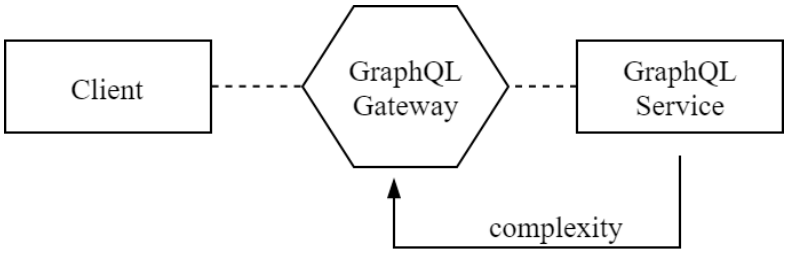

Figure 5.5: Moving complexity from GraphQL microservices to the gateway, adopted from [Stu17a]

Moving the aggregation and transformation of data from the GraphQL services to the gateway will decouple services further, as services can focus on providing data without the need to know how a consumer will consume the API in the end. The following section will take a look on how these tasks can be accomplished.

5.3.1 Combining Schema with Introspection

The creation of a gateway with GraphQL requires all services to expose a valid GraphQL endpoint. GraphQL uses its introspection feature to combine several schemas together. This is an essential part when working with microservices. With the ability to introspect a schema, as described in 2.5, it is possible to receive all necessary information about an endpoint.

- All data can be queried with one request, as data can be arbitrarily combined by the client.
- Introspection offers a central entry point to a service, without the need to know any other endpoints. This also provides documentation of the whole service.

Having a central gateway that exposes one schema that combines all services offers following advantages. Through a central schema the request can be redirected to each

service. When working with microservices each service should map its own domain and only be responsible for its own data. Having a schema and a resolver in each microservice can accomplish that.

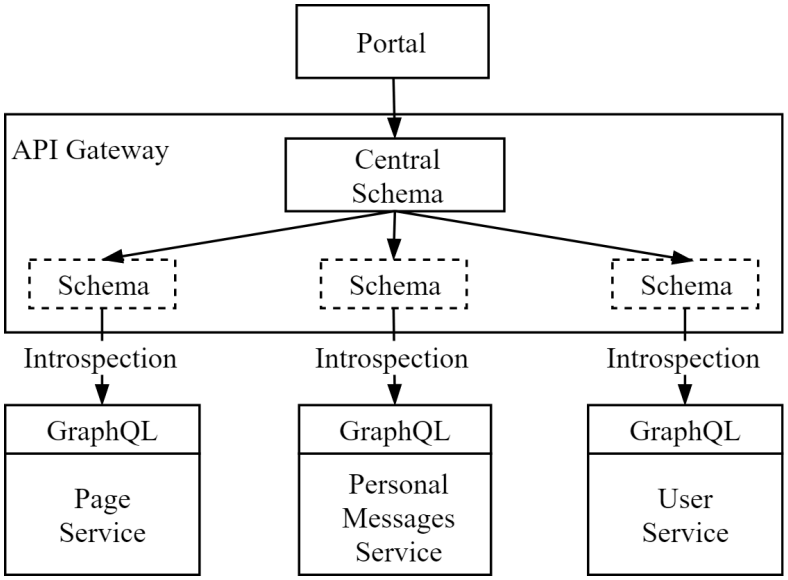

Figure 5.6: Creating a central GraphQL schema with introspection

In REST new endpoints need to be manually documented and redirects need to be implemented in the gateway. GraphQL takes advantage of its introspection system to combine and discover schemas. When querying an endpoint, GraphQL can merge different schema together, allowing for any root queries, mutations and types to be combined into a schema. There is no need to redirect to routes from the gateway. A significant advantage of this technique is that changes of a schema in a microservice will be registered in the gateway[15]. This results in only one place where changes have to be made for a new feature to be made available and discoverable.

5.3.2 Schema Stitching

In the GraphQL community the term **schema stitching** was coined by Apollo[16] and is often used for the process of merging several schema into one [Gra17]. In contrast to the combination of schema by introspection, schema stitching goes further by manipulating the resulting schema and providing the ability to link fields and types of schema together. This allows for fields or types, that are provided by a service, to be resolved by another service and can effectively reduce calls needed by the client to resolve requested information. Figure 5.7 shows how the author field of the page service, which returns a scalar of type Int, will be linked to the User Type. Creating such a link between objects in the GraphQL gateway enables the client to request the Page and query user information

[15]This requires the gateway to execute the introspection step again.
[16]Apollo is an open source company for JavaScript and GraphQL.

with only one request. This is done by redirecting the call of the author, which provides the ID, to the query of the user service. This would become inefficient when requesting data that contains more than one user, as each request would query the user service independently. A mitigation technique is to create an entry in the user service that will return a list of users, allowing for batching requests to each service. With the ability to efficiently batch request, it is possible to only call each service once for a request that requires data from several services. Many schema stitching technologies use Facebook's DataLoader[17] library to batch requests.

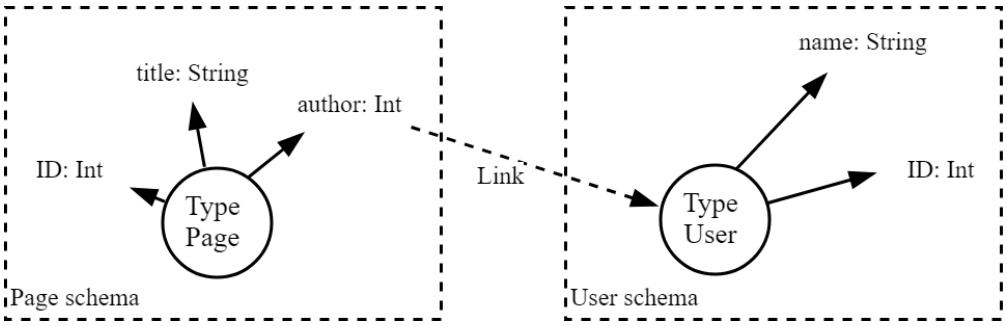

Figure 5.7: Linking schema together

The GraphQL specification offers another way to stitch schemas together. By allowing types to be extended, the central schema of the gateway can delegate calls to the necessary queries. Linking types or fields together has a disadvantage. The central schema will be reliant on the services that were stitched together. Stitching schema together is currently the reason GraphQL gateways are so powerful. The prototype validation will therefore be evaluated with a stitched schema with extending types.

5.4 Migration

This chapter gives an overview of the procedures that have to be taken when migrating from an existing REST architecture to a GraphQL based architecture. These steps have to be followed for a migration of services to be possible:

1. Creating a schema and resolvers for each service. This is the main part of the migration step and has been shown in 5.2.

2. Creating a Gateway that combines the schema of each service. How this can be accomplished has been shown in 4.3.1.

Section 4.5.3 describes how a migration can be carried out when breaking changes are introduced. Implementing a new API into an architecture certainly qualifies as a breaking change. When serving GraphQL over HTTP, any existing REST endpoints can stay

[17] A utility library for the data fetching layer, can also be used with REST. See: [Fac18c]

available for the consumer. This allows for two API's to coexist in the same service, making a migration by coexisting endpoints possible. Having made the changes in the backend, there are changes to the frontend part that have to be made. Every request to a REST endpoint has to be rewritten in as a GraphQL query.

5.4.1 Wrapping REST endpoints

Working with an established microservice architecture, the possibility of REST endpoints that can not be removed exists. This might be due to legacy reasons, that API consumers can not yet move to the new API or that the endpoint can not be refactored. By wrapping existing REST endpoints with a GraphQL resolver, the advantages of GraphQL's ability to aggregate data can still be used. This can be accomplished by calling the REST route in a resolver function and mapping the incoming data to the desired type. This will keep the functionality of the REST route intact while allowing for querying single fields via the GraphQL schema. Listing 5.8 shows a REST route. The URI is the identifier for a specific user, returning user data depending on the parameter provided. The status of the response will be set depending on whether a user has been found.

```
1 router.get('/api/v1/user/:id', (req, res) => {
2     const user = getUserByID(req.params.id);
3     if(user === null) {
4         return res.status(404).send("User Not found")
5     }
6     return res.status(200).send(user);
7 });
```

Listing 5.8: REST route to retrieve a user

As resolvers can be asynchronous, it is valid to call the route inside the resolver function by passing the arguments received from the query. A successful call will return the requested user data. If the response of the REST call fails to retrieve a user, an error message can be thrown. This will not terminate the GraphQL request, but will add an error field in the response. By wrapping REST endpoints a GraphQL service can be quickly implemented. However, this comes with disadvantages as the service will be dependent on the REST route. Therefore, implementing resolvers without wrapping REST endpoints is preferable.

```
1 /* New query for the user-service */
```

```
2  type Query {
3      ...
4      userFromREST(id: ID): User
5  }
6
7  /* Wrapping the REST endpoint */
8  function getUserFromREST(id) {
9      return http.get(`http://localhost:4445/api/v1/user/${id}`)
10         .then(response => response.data)
11         .catch(error => new Error('could not retrieve user from REST') )
12 }
```

Listing 5.9: GraphQL resolver for the REST route

5.4.2 Versioning

By serving a GraphQL endpoint and the REST endpoints simultaneously, consumers can gradually move to the new endpoint. Following the GraphQL philosophy of not versioning, as described in section 2.5, the new endpoint will not contain a version number. While it is possible and valid to version the GraphQL endpoint, it is encouraged to follow the GraphQL add-only approach to schema evolution. This has trade-offs, as major changes to an existing schema can potentially break clients. Adding additional fields to a schema is always safe, as clients will not receive data that has not been requested [Byr16b].

5.5 Validation

The validation of the prototype will be carried out by analyzing the goals of the scenarios that were created in section 5.1.4. This will give a detailed view on how the scenarios have been accomplished with the GraphQL microservice architecture. By analyzing the acceptance criteria of each scenario it will be shown to what degree a scenario has been completed and show potential risks of such architecture. To summarize the findings of each scenario in accordance with the acceptance criteria, the notation in table 5.7 will be used.

Identifier	Explanation
●	Criteria can be fully fulfilled
◐	Criteria can be partially fulfilled
○	Criteria can not be fulfilled

Table 5.7: Identifier explanation

This will provide an overview of each scenario by showing to what degree a given criteria can be fulfilled.

5.5.1 Scenario A: Receiving Personal Messages

The first scenario in table 5.4 refers to the retrieval of personal messages. To implement this feature, a schema in the personal message service has to be implemented. The schema will contain the required fields. A root query to retrieve the messages of a single user will be created. To separate messages by type, an argument can be passed to the message query. Listing 5.10 shows the root query and the documented message query, which will return messages depending on the type.

```
1 type Query {
2     personalMessages(userID: Int!): PersonalMessages!
3 }
4 type PersonalMessages {
5       ...
6       "Type = 'personal' OR 'recommendation'"
7       messages(type: String): [Message]
8 }
```

Listing 5.10: Documented Personal Message Schema

The messages provided by the personal-messages-service contain the ID of sender and the ID of the page that was recommended. These types need to be linked inside of the message type, making them resolve a page and a user. This will allow for a single call to retrieve all information needed. Stitching together these schema is done in the GraphQL gateway. By introspection, the schema of each service will be called. Before all schema are merged into a central schema, the message type will be extended with the necessary links. Listing 5.11 shows how to extend the schema with the SDL. These extended fields will use the information provided in the message fields, the page ID and the user ID. By delegating these fields to a root query of the respective service, information about the user and the page can be resolved.

```
1 /* Extend Type in Gateway */
2 extend type Message  {
3       fromUser_UserService: User
4       recommendedPage_PageService: Page
5 }
```

Listing 5.11: Extending the message type

The gateway will combine the schema of each service and extend the types that have been extended. This will make the message type aware of the newly added fields. With GraphQL's introspection the combined schema can validate the query, having knowledge about the user and page type from other services. This will result in a schema that can retrieve all messages of a certain type, with resolved information about pages and sender. A big advantage is the readability of this approach. Listing 5.12 shows the final query to gather the necessary information for scenario A.

```
1 query scenarioA {
2   personalMessages(userID: 42) {
3     messages(type: "recommendation") {
4       date
5       from_UserService {name, profilePage}
6       recommendedPage_PageService {title, link}
7 }}
```

Listing 5.12: Query A: GraphQL query for scenario A

Figure 5.8 shows a sequence diagram, depicting the calls between services to resolve query A. The gateway receives the query and forwards it to the personal message service. The personal message service will resolve all messages from user with ID 42. As the message type has been extended, the personal message service will call the user service

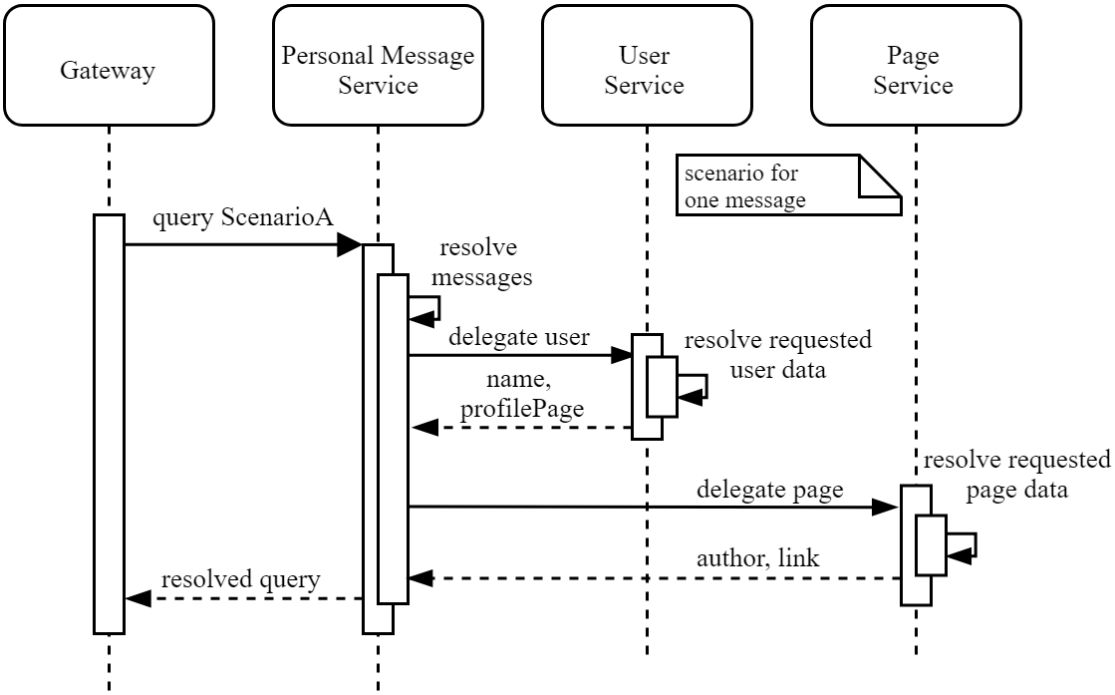

Figure 5.8: Sequence of service calls for Scenario A

and the page-service for every available message. What can be observed is that each service will only return the requested fields in the data, minimizing the data that will be sent from each service. By stitching the schema together, a readable query can be produced that can resolve all required information. All messages of type recommendation will be returned with the content and the date. The user service resolves the name and profile page of a user, and the page service will resolve the author and the link, thus resulting in all acceptance criteria to be fulfillable.

Acceptance Criteria	A1	A2	A3	A4
Fulfillable degree	●	●	●	●

Table 5.8: Results of prototyping Scenario A

5.5.2 Scenario B: Sending Personal Messages

The second scenario, which is shown in table 5.5, will forward a message from one user to another. This will be done with a mutation. The newly created data will then be retrieved by a query. A new mutation, that will accept the arguments required to send messages from one user to another, has to be created in the personal message service. Listing 5.5 shows the mutation that is used for this scenario. Implementing and using mutations follow the same principle as queries. Listing 5.13 shows the usage of the mutation, by sending out a message from user 4 to user 3 with a content.

```
1 mutation send_scenarioB {
2   sendPersonalMessage(from: 4, to: 3, content: "Hello number 3")
3 }
```

Listing 5.13: Using mutations

The business logic behind this mutation is written into a resolver function, as shown in section 5.2.2. The next step to this scenario is using the newly created messages in a query to retrieve the data. Listing 5.14 shows the query for scenario B. This query will retrieve all messages of type personal, the content, the date and information about the sender of the message. The queries of scenario A and B both use the same message type defined in the schema, but they slightly differ in the fields that have been required. What can be observed is that, despite developing the mutation, there is no need to develop a new endpoint that will expose the required data. A consumer of a GraphQL service can state which fields to query and reuse already existing types and fields to combine them and create new information.

```
query scenarioB {
    personalMessages(userID: 3) {
        messages(type: "personal") {
            content
            date
            from_UserService {name, profilePage}
    }}}
```

Listing 5.14: Query Scenario B

This can potentially lead to less changes that have to be made for a feature to be implemented[18]. With mutations it is possible to execute write operations that are run in series [Fac18g]. Retrieving the newly created information, with every field that is required, can be done without adding a new endpoint or changing how data is exposed. Therefore, all acceptance criteria of the second scenario can be fulfilled.

Acceptance Criteria B	B1	B2	B3
Fulfillable degree	●	●	●

Table 5.9: Results of prototyping Scenario B

5.5.3 Scenario C: Migration of the corresponding services

Section 5.5.1 and 5.5.2 showed that it is possible to integrate GraphQL into an existing microservice architecture built in restful style and how to implement features that require read and write operations. The third scenario, as described in table 5.6, refers to the migration of each service, in context of the SON microservice architecture. The criteria describes a system that is fully migrated. All services, including the gateway, have been migrated and are able to expose a GraphQL and a REST endpoint simultaneously, thus fulfilling the first criteria. The second criteria is to reduce code redundancy. This criteria can be fulfilled to a higher degree than with an API data query language. A RESTful architecture would implement new endpoints to efficiently fetch certain data, often resulting in duplicate code to aggregate data. GraphQL does not need to create new endpoints as the consumer dictates what fields have to be returned. This creates an architecture that is suited to minimize code redundancy, while being efficient in the data it provides. This fulfills the second criteria of this scenario.

[18]This will be elaborated in detail in section 6.3

The third criteria is the documentation for each service. The way a GraphQL endpoint has to be described by a schema, which defines the read and write operations and the types it requires, a GraphQL service will always be documented. With the introspection, the documentation can be consumed by any consumer and can be passed to another layer, for this instance the gateway.

A criteria that can not be completely fulfilled is minimizing the calls to each service. While GraphQL offers a way to expose data in a single endpoint, resolving required data from other services in a single call can not be accomplished without using extensions. With the use of the gateway pattern, a client will always get a single response. And with the ability to define multiple queries in one request, fetching data on the client side becomes easier. However, reducing the number of calls to one, from one service to another, is not possible in a feasible way with the current GraphQL technology. Schema stitching allows for linking types, but section 5.5.1 showed that by linking the types services will be called multiple times. Although it would be possible to query all users once, this would have to be done in another query, destroying the readability of the query. Table 5.10 shows the acceptance criteria of scenario C.

Acceptance Criteria	C1	C2	C3	C4
Fulfillable degree	●	●	●	◐

Table 5.10: Results of prototyping Scenario C

5.6 Summary

The previous chapters showed user scenarios and described the prototype of this thesis. To determine the needs and conditions of the prototype, a requirements analysis has been carried out, that identified stakeholders and goals. The identified goals have been depicted in form of user stories, that have been taken from the SON backlog and have been the basis of the developed prototype. These stories cover the integration and migration scope of the SON architecture. The decision was made to use GraphQL as the API data query language, as it is offering a more promising and strict approach compared to Falcor. According to the defined goals, it has been shown how to integrate and how to migrate GraphQL into an existing REST architecture. The integration process explained how the resulting services work with a GraphQL microservice architecture. In particular the gateway pattern has been shown to be a viable pattern when working with GraphQL, due to the nature of GraphQL's introspection ability. By validating the prototype against all acceptance criteria, an overview for each story was given. This demonstrated that an integration and a migration into a REST architecture is not only possible, but has

benefits in regard to discovery and over- and underfetching. The ability to stitch schema together allows for clients to fetch data in a single response. The only acceptance criteria that can not be fulfilled by the GraphQL services is drastically minimizing the number of calls made from one service to other services.

6 Evaluation

This chapter will evaluate the findings that have been presented in this thesis and will elaborate in detail on the drawbacks and benefits of working with GraphQL. Furthermore, the architectural and workflow changes will be presented.

6.1 Benefits and Drawbacks

Offering a new approach on how to tackle the problems with REST, GraphQL offers some interesting benefits and drawbacks. The prototype gave a detailed look into GraphQL and and how it affects a microservice architecture. This section will present benefits and drawbacks that stood out while working on the prototype.

Over- and Underfetching

Over- and underfetching are one of the drawbacks when working with REST. Section 3.1 described how GraphQL can mitigate these cases in theory. The prototype showed that this theory holds true. Scenarios A and B, described in 5.5.1 and 5.5.2, showed that the responseonly contains the data requested in a query, thus mitigating the problem of over- and underfetching.

Type System

The lack of a strict type system is the second drawback that is described in section 3.2. The advantages of an API that follows a strict type system became apparent in scenario A, described in section 5.5.1. The queries the client can form contain all information about what types the response will contain. Furthermore, the type system gives the ability to extend types and therefore stitch schema together.

API Discovery

The manual discovery is another drawback of REST, that is described in section 3.3. This drawback is tackled by GraphQL with the type system and the introspection ability, which results in a powerful tool set provided by GraphQL. Section 5.3 showed how a gateway can be constructed and how the introspection can combine all schema together. This results in a gateway that is documented with all known schema and types. Documentation happens automatically and can be used directly with the built-in API explorer.

Decoupling with GraphQL

Section 4.2.1 described high cohesion and loose coupling, a central concept in a microservice architecture. GraphQL forces a high cohesion on a microservice architecture. This is done by separating the API description part, which is the schema, and the business logic, the resolvers. Furthermore, GraphQL helps with implementing the *interface-segregation principle*[19], which states that "Clients should not be forced to depend on methods that they do not use." [Mar02]. As clients will only use requested mutations and queries, this helps in decoupling services from clients. Loose coupling states that changes to one service should not require changes to another service. While REST achieves loose coupling with the uniform interface principle, as described in section 2.4, GraphQL achieves the same result with a strict specification and client specific queries. GraphQL's API design therefore follows Postel's law, as described in 4.5.2. The prototype showed that stitching schemas together in the gateway results in an aggregation layer that is dependent on each service. While this does not directly violate loose coupling, as each service is still independent, this will create a tighter coupling of the services in the gateway.

Obsolete Backend for Frontend

Section 4.3.2 describes the Backend for Frontend pattern by Newman, often differentiating between desktop and mobile clients. This pattern is used when aggregating data needs to be tailored to a specific device type. Especially for mobile devices, it is necessary to create endpoints that only supply the required data, which can be problematic with REST, as many endpoints have to be created and managed. GraphQL's declarative data fetching approach moves the responsibility of which data to fetch to the client, thus making the BFF pattern obsolete.

Caching

Newman describes that caching in too many places can get too complicated, which is the case when caching is done in each microservice [New15a]. Caching in REST usually happens over the HTTP layer. As GraphQL does not follow the concept of URIs, identifying resources is handled by providing a globally unique identifier. Clients can then use these identifiers to build a cache [Fac18b]. Caching in enpoint based APIs can be done in the following places [Stu17b]:

1. *Client Caching*: The client keeps track of the data that was requested.
2. *Network Caching*: Intercepting requests that are similar and returning a response early out of memory.

[19] Part of the SOLID design principles by Martin Roberts, which can help in achieving high cohesion and low coupling [Mar02]

3. *Application Caching*: Caching responses in-memory to quicker generate responses.

As GraphQL's requests are done via a POST method, tools for network caching can not be used. Application caching needs to be implemented by hand, as GraphQL does not provide an out of the box solution, but this holds true for REST as well. However, GraphQL provides a strict specification, that not only services exposing an API must follow, but also one that clients must follow. There are several client implementations available that help with caching requests. But caching server side remains a challenge in GraphQL.

Batching requests

The prototype showed that batching requests with GraphQL is not a trivial task. By stitching schemas together it is possible to link certain types, and the ability to send out multiple queries in one request can reduce the number of calls. Batching request is currently done with the use of Facebook's DataLoader, as described in 5.3.1, which can be used in REST as well. GraphQL has several client implementations that help with batching requests, batching requests from one service to another is not in the client scope.

Serving large files

The prototype has shown that working with a lot of small data samples works well with GraphQL, as all data is returned in JSON. As long as the response contains small data, that is represented in scalar types or with strings, this has no disadvantage. When working with big data files, which are typically images or other media files, GraphQL's one response approach can be a disadvantage. Big files have to be decoded and served as a string or a custom type. However, this can quickly lead to an enormous response, as each queried image field will be returned in a single response. When working with static files, GraphQL should respond with the URL to the image[20]. This can potentially require another call to receive the image.

6.2 Architectural Changes

The prototype has shown that a GraphQL microservice architecture is identical to a REST microservice architecture and patterns like the API Gateway work well in this constellation. While an API data query language like GraphQL does not completely change the architecture, it changes some aspects of the API's of a service and how clients and server interact.

[20]The Github GraphQL API returnes the URL for media files.

- *Clearly Separated Data Owners*: Having everything behind a single endpoint results in a well defined ownership, as no complex routing is required [SAN18].
- *Specified Query Language*: GraphQL is strict data query language that specifies API capabilities of server and client side [Stu17a].
- *Decoupling*: GraphQL decouples API consumers from API providers.In REST the shape of the response is determined by the server, whereas with GraphQL the shape is defined by the consumer that uses it.
- *Empowering Clients*: GraphQL's type system and introspection allows for powerful clients. Batching and Caching is implemented in the clients. Sashko Stubailo describes a GraphQL query as a "unit of data fetching", since its query is attached to the code that uses it [Stu17a].

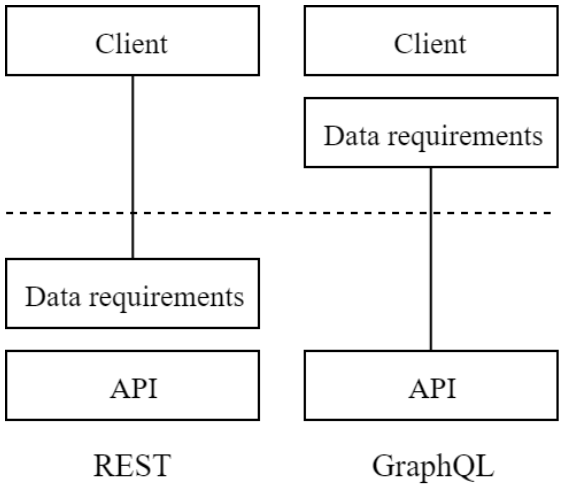

Figure 6.1: Moving Data Requirements to the Client, adopted from [Stu17a]

The declarative approach of GraphQL not only gives clients more control over the data, but with the type system and introspection there are more possibilities as to what a client can do, for example batching, caching and validation. The data requirements are identified within a query, which is defined and executed by a client. Changes to types and fields of a query on the server will not change the data that has been requested, as only the specified fields will be required. Figure 6.1 shows the architectural change of an API data query language, as the data requirements move to the client.

6.3 Workflow Changes

While developing the SON portal the KCS team realized, that dividing stories into backend and frontend is not ideal. Depicting a full feature in a user story is better suited for the KCS development team. This requires stories to be cut vertically, which

means a user story contains a backend and a frontend task. This vertical cut results in a close interaction between frontend and backend developers. When with a architecture based on REST, the backend needs to know what information to provide and create a new endpoint that provides the data, even if the data already exists in another form in one service. The frontend then can consume the newly created route. Figure 6.2 shows this scenario. Changes for a new feature are typically required in two places.

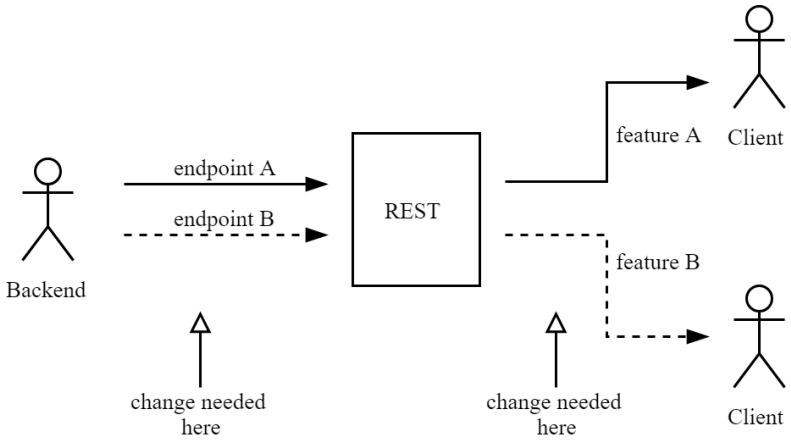

Figure 6.2: REST workflow, adopted from [Cla16]

With GraphQL, a service exposes all data in a single endpoint and the client defines what data to fetch. Backend developers can therefore focus on each service, without the need to know the exact shape of a specific response. Scenario B, described in 5.5.2, showed that the new message type did not require a new field or new types to be added. Figure 6.3 shows that in a GraphQL architecture, changes for a new feature only have to be made on the client side.

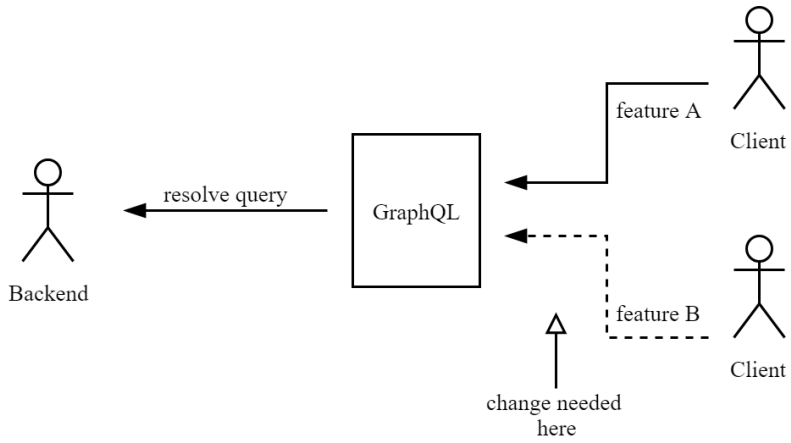

Figure 6.3: GraphQL workflow, adopted from [Cla16]

This results in less interaction between frontend and backend, as shapes and data do not need to be specified by both teams, which is a big step towards an independent workflow.

6.4 Summary

The evaluation in regards to the prototype showed that GraphQL can mitigate the drawbacks that were described in section 3. Over- and underfetching are tackled by the declarative approach. The strict type system brings advantages when forming queries and developing services, as the shape of the returned response is always known. This also impacts the architecture, as there is no need for the BFF pattern anymore. The API discovery is where GraphQL shines. Endpoints are documented automatically and when working with a gateway, documentation of all services is combined together. Especially with GraphiQL and the type system, any endpoint can be easily explored. Decoupling is done via a strict separation of schema and resolvers and also by adhering to Postel's law for its API design, which results in a robust interface. But there are also drawbacks when using GraphQL. As every request is a POST method, most tools for network caching do not work. Caching can therefore be either implemented in the client or in the application itself. Batching request is another non trivial task. Although the single response based design can minimize calls and parallel execution of queries can be done, batching is currently often implemented with the *DataLoader*, which is an API library that can also be used in RESTful services. Due to the nature of the queries, batching has to be implemented into the GraphQL language before it becomes viable. Another drawback is media contents. GraphQL is designed to serve a lot of small data sets. Requiring binary content can quickly bloat the response and lead to a lot of data being returned. The prototype showed that a GraphQL microservice architecture can be implemented and that a migration from REST is possible. The gateway pattern is a good fit for a GraphQL service, as combining several endpoints can be done with ease, offering the advantages of the API discovery and making the BFF pattern obsolete. An architectural change that can be observed is that with the declarative data fetching approach, GraphQL is moving the data requirements from the server to the client. Decoupling is also achieved in the workflow, separating the need to decide what data endpoints will return, which is usually a combined task by frontend and backend.

7 Discussion and Conclusion

This chapter discusses the findings of this thesis. It will answer the problems of this work with help of the established knowledge, look into possible benefits and drawbacks and will discuss the use of GraphQL in accordance to the stakeholders.

7.1 Discussion

This thesis focused on the integration of API data query languages in microservice architectures and looked at two major technologies, GraphQL and Falcor. Falcor has been deemed the more user friendly and easier to learn. Working with both of these technologies, this has not been the case. GraphQL has been found to be easier to work with. With the type system, the ability to introspect schema and the automatic API discovery, GraphQL offers a more powerful technology to design interfaces. Another reason for choosing GraphQL over Falcor is the community. While both technologies are open-sourced, GraphQL's community is overwhelmingly bigger and is still growing. GitHub even switched from REST to a GrapQL based API [Dai+16]. The availability of a variety of tools, ranging from clients to custom libraries, make GraphQL the better choice.

The prototype showed that a migration from a REST service is possible, while also being able to let the endpoints coexist. The resulting architecture did not include major changes, although complexity was moved to the gateway. GraphQL has proved to be a good fit for the gateway pattern, leveraging the API discovery and schema stitching to create a powerful, yet easy to understand gateway. This is especially useful for any consumer of the API, as data can be queried from one single endpoint with a meaningful query language. However, this requires services to be written in GraphQL, which in return forces consumers to use a GraphQL client.

With its declarative approach, GraphQL puts the data requirements into the hands of the client. Backend and frontend developers do not need to define and maintain many interfaces anymore. The backend developers can focus on each service by exposing all data, and frontend developers and clients can state what data to consume. Shifting the data requirements also has impacts on the workflow. New features or changes to features often require the backend to adjust or create endpoints and the frontend to consume or adjust to the endpoint. Now these changes only have to be made in the client. This of course, only holds true if the data that is required has been already exposed by a service, but future changes and adjustments should be easier to implement. Separating the need to define interfaces is how GraphQL gives developers of both teams more free-

dom and how the resulting microservices are decoupled, when compared to Conway's law.

GraphQL is no silver bullet and has its drawbacks. For small applications, the overhead of GraphQL might be to complex. When creating services that serve large files, there is no way around a RESTful API. Especially as REST is widely used and can be understood by every browser, it is better for the general public.

The KCS team was interested in an alternative to REST for its microservice architecture. With the established knowledge, this can be answered in regards to the stakeholders. For a User there are no changes when migrating to a GraphQL architecture. Although performance has not been the focus of this work, it is a crucial criteria for a user. Petter Johanson analyzed the response time of GraphQL and REST in his master thesis and concluded that REST has a lower response time than GraphQL [Joh17]. These findings would suggest not to use GraphQL in an application where response time is important. From the perspective of the software architect a stable system is demanded. Seeing that GraphQL is a rather new technology, it can be debated whether GraphQL is mature enough to be production ready. The findings of this paper suggest it is, although there are some drawbacks as caching. Kit Gustavsson and Erik Stenlund analyzed GraphQL in their master thesis project and concluded that GraphQL is to be preferred when "a large amount of requests/responses and performance in terms of page-speed and network use is important." [GS16]. The most benefiting stakeholders are the frontend and backend developers, as GraphQL gives both parties more freedom while providing strict rules to design an API. However, this comes with the cost of complexity, but can have a positive effect on the development speed of features. Especially in a large team, rules for a strict API design would be recommended.

7.2 Conclusion

This thesis has shown how to integrate and migrate an existing REST microservice architecture with an API data query language. Falcor and GraphQL have been evaluated and GraphQL has been found to be more fitting for an universal microservice approach. The impact on the workflow and on the architecture have been described, and the effects were identified to be of positive nature. While REST is designed to be efficient for large-grain data transfer, GraphQL has been found to be designed for efficient fine-grain data transfer. It was created out of the need for efficient mobile communication and is still focused on the needs of User-Interface developers. In a microservice architecture, where many services offer small data, a technology that provides control over the requested data can be a benefiting tool. GraphQL is still being developed, but the ever growing use and community indicates that it is on its way to be an established alternative to REST.

List of abbreviations

API Application Programming Interface

BFF Backend for Frontend

SOA Service-oriented architecture

HTTP Hypertext Transfer Protocol

KCS Knowledge & Collaboration Solutions

URI Uniform Resource Identifier

URL Uniform Resource Locator

REST Representational State Transfer

RFC Request For Comments

UI user interface

IEEE Institute of Electrical and Electronics Engineers

HTTP Hypertext Transfer Protocol

JSON JavaScript Object Notation

XML Extensible Markup Language

CRUD Create, read, update and delete

DDD Domain-Driven Design

WSDL Web Services Description Language

WADL Web Application Description Language

HATEOAS Hypermedia as the Engine of Application State

OData Open Data Protocol

SDL Schema Definition Language

SON Social Office Net

FIQL Feed Item Query Language

IDE Integrated development environment

List of Figures

Listings

List of Tables

Bibliography

[29111] ISO/IEC: Std 29148:2011. *Systems and software engineering — Life cycle processes — Requirements engineering*. Tech. rep. International Organization for Standardization, 2011.

[AF15] Martin L. Abbott and Michael T. Fisher. *The Art of Scalability: Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise (2nd Edition)*. 2015. ISBN: 978-0134032801.

[Alm+17] Washington Henrique Carvalho Almeida et al. *Survey on Microservice Architecture - Security, Privacy and Standardization on Cloud Computing Environment*. Tech. rep. 2017.

[Ars+09] Ali Arsanjani et al. *SOA Manifesto*. 2009. URL: `http://www.soa-manifesto.org/`. [Online; accessed 16-January-2018].

[BD99] Bernd Bruegge and Allen A. Dutoit. *Object-Oriented Software Engineering; Conquering Complex and Changing Systems*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1999. ISBN: 0134897250.

[Byr15] Lee Byron. *GraphQL: A data query language*. 2015. URL: `https://code.facebook.com/posts/1691455094417024/graphql-a-data-query-language/`. [Online; accessed 10-January-2018].

[Byr16a] Lee Byron. 2016. URL: `https://github.com/facebook/graphql/issues/175`. [Online; accessed 30-January-2018].

[Byr16b] Lee Byron. *How to really solve version problem*. 2016. URL: `https://github.com/facebook/graphql/issues/134`. [Online; accessed 22-04-2018].

[Cla16] Lin Clark. *Execution*. 2016. URL: `https://code-cartoons.com/a-cartoon-intro-to-facebook-s-relay-part-1-3ec1a127bca5`. [Online; accessed 03-05-2018].

[Coh04] Mike Cohn. *User Stories Applied: For Agile Software Development*. Pearson Education, Inc., 2004. ISBN: 0-321-20568-5.

[Con14] Agile Business Consortium. *The DSDM Agile Project Framework (2014 Onwards) - Requirements and User Stories*. 2014. URL: `https://www.agilebusiness.org/content/requirements-and-user-stories`. [Online; accessed 16-April-2018].

[Con68] Melvin E. Conway. *How Do Committees Invent?* Tech. rep. 1968. URL: `http://www.melconway.com/Home/Committees_Paper.html`.

[Dai+16] Kyle Daigle et al. *The GitHub GraphQL API*. 2016. URL: https://githubengineering.com/the-github-graphql-api/. [Online; accessed 06-05-2018].

[Def01] Department Of Defense. *Systems Engineering Fundamentals*. Defense Acquisition University Press, 2001.

[Dra+16] Nicola Dragoni et al. 2016. URL: https://arxiv.org/pdf/1606.04036v1.pdf. [Online; accessed 16-January-2018].

[Erl+13] Thomas Erl et al. *SOA with REST - Principles, Patterns and Constraints for Building Enterprise Solutions with REST*. The Prentice Hall service technology series. Prentice Hall, 2013, pp. I–XXXII, 1–577. ISBN: 978-0-13-701251-0.

[Eva14] Eric Evans. *Domain-driven design : tackling complexity in the heart of software*. 1st ed. Addison-Wesley, Aug. 2014. ISBN: 0321125215. URL: Hardcover.

[Exp14] Express. *Using middleware*. Express, 2014. URL: http://expressjs.com/en/guide/using-middleware.html. [Online; accessed 22-January-2018].

[Fac16] Facebook. *GraphQL*. Internet-Draft. 2016. URL: http://facebook.github.io/graphql/October2016/. [Online; accessed 18-January-2018].

[Fac18a] Facebook. *Authentication and Express Middleware*. 2018. URL: http://graphql.org/graphql-js/authentication-and-express-middleware/. [Online; accessed 20-04-2018].

[Fac18b] Facebook. *Caching*. 2018. URL: http://graphql.org/learn/caching/. [Online; accessed 21-04-2018].

[Fac18c] Facebook. *Dataloader*. 2018. URL: https://github.com/facebook/dataloader. [Online; accessed 24-04-2018].

[Fac18d] Facebook. *Execution*. 2018. URL: http://graphql.org/learn/execution/. [Online; accessed 01-05-2018].

[Fac18e] Facebook. *GraphQL*. 2018. URL: http://graphql.org/learn/schema/#type-system. [Working Draft; accessed 19-02-2018].

[Fac18f] Facebook. *GraphQL*. 2018. URL: http://graphql.org/learn/introspection/. [Working Draft; accessed 13-02-2018].

[Fac18g] Facebook. *Queries and Mutations*. 2018. URL: http://graphql.org/learn/queries/#mutations. [Online; accessed 26-04-2018].

[Fie00] Roy Fielding. *Architectural styles and the design of network-based software architectures*. 2000. URL: https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf.

[Fie09] Roy Fielding. *It is okay to use POST*. 2009. URL: http://roy.gbiv.com/untangled/2009/it-is-okay-to-use-post. [Online; accessed 12.02.2018].

[Fow10] Martin Fowler. *Richardson Maturity Model - steps toward the glory of REST*. 2010. URL: `https://martinfowler.com/articles/richardsonMaturityModel.html`. [Online; accessed 13-March-2018].

[Fow05] Marting Fowler. 2005. URL: `https://martinfowler.com/bliki/ServiceOrientedAmbiguity.html`. [Online; accessed 24-January-2018].

[Fow11] Marting Fowler. *TolerantReader*. 2011. URL: `https://martinfowler.com/bliki/TolerantReader.html`. [Online; accessed 19-March-2018].

[Fow13] Marting Fowler. *Enterprise Integration Using REST*. 2013. URL: `https://martinfowler.com/articles/enterpriseREST.html#versioning`. [Online; accessed 01-March-2018].

[Fow14] Marting Fowler. 2014. URL: `https://martinfowler.com/bliki/BoundedContext.html`. [Online; accessed 19-02-2018].

[Fow] Marting Fowler. *Microservices Resource Guide*. URL: `https://martinfowler.com/microservices/`. [Online; accessed 20-March-2018].

[Gra18a] GraphQL. *GraphQL Best Practices - Versioning*. 2018. URL: `http://graphql.org/learn/best-practices/#versioning`. [Online; accessed 24-January-2018].

[Gra18b] GraphQL. *Serving over HTTP*. 2018. URL: `http://graphql.org/learn/serving-over-http`. [Online; accessed 24-January-2018].

[Gra17] Apollo GraphQL. *Schema stitching - Combining multiple GraphQL APIs into one*. 2017. URL: `https://www.apollographql.com/docs/graphql-tools/schema-stitching.html`. [Online; accessed 18-04-2018].

[Gro95] Standish Group. *CHAOS Report*. 1995. URL: `https://www.projectsmart.co.uk/white-papers/chaos-report.pdf`. [Online; accessed 04-April-2018].

[GS16] Kit Gustavsson and Erik Stenlund. "Efficient data communication between a webclient and a cloud environment". MA thesis. 2016.

[hub17] hubrix.co. *2017 API Survey Results*. 2017. URL: `https://www.hubrix.co/2017/12/2017-api-survey-results/`.

[Hus15] Jafar Husain. *JSON Graph: Reactive REST at Netflix*. Netflix, 2015. URL: `https://dl.acm.org/citation.cfm?doid=2742580.2742640`. [Online; accessed 10-January-2018].

[Joh17] Petter Johansson. "Efficient Communication With Microservices". MA thesis. SE-901 87 UMEA SWEDEN: Umea University, 2017.

[LF14] James Lewis and Marting Fowler. 2014. URL: `https://martinfowler.com/articles/microservices.html`. [Online; accessed 16-January-2018].

[Mar02] Robert C. Martin. *Agile Software Development, Principles, Patterns, and Practices*. Pearson Education, Inc, 2002. ISBN: 0-13-597444-5.

[Mic15] Microsoft. *OData – The Protocol for REST APIs*. 2015. URL: http://www.odata.org/. [Online; accessed 14-April-2018].

[Min17] Martin Helmich Mina Andrawos. *Cloud Native programming with Golang: Develop microservice-based high performance web apps for the cloud with Go*. Packt Publishing, 2017. ISBN: 978-1-78712-598-8.

[Nad+16] Irakli Nadareishvili et al. *Microservice Architecture: Aligning Principles, Practices, and Culture*. 2016. ISBN: 978-1491956250.

[NW17] Masashi Narumoto and Mike Wasson. *Gateway Aggregation pattern*. 2017. URL: https://docs.microsoft.com/en-us/azure/architecture/patterns/gateway-aggregation. [Online; accessed 23-02-2018].

[Net15a] Netflix. *Data Sources*. Netflix, 2015. URL: http://netflix.github.io/falcor/documentation/datasources.html. [Online; accessed 22-January-2018].

[Net15b] Netflix. *How Does Falcor Work?* Netflix, 2015. URL: http://netflix.github.io/falcor/starter/how-does-falcor-work.html. [Online; accessed 22-January-2018].

[Net15c] Netflix. *What is Falcor?* Netflix, 2015. URL: https://netflix.github.io/falcor/starter/what-is-falcor.html. [Online; accessed 22-January-2018].

[Net15d] Netflix. *What is JSON Graph?* Netflix, 2015. URL: http://netflix.github.io/falcor/documentation/jsongraph.html. [Online; accessed 22-January-2018].

[New15a] Sam Newman. *Building Microservices: Designing Fine-Grained Systems*. 1st. O'Reilly Media, Feb. 2015, p. 280. ISBN: 978-1491950357.

[New15b] Sam Newman. *Pattern: Backends For Frontends*. 2015. URL: https://samnewman.io/patterns/architectural/bff/. [Online; accessed 27-02-2018].

[Nie93] Jakob Nielsen. *Usability Engineering*. Academic Press, 1993. ISBN: 1-12-518406-9.

[Not07] Mark Nottingham. *FIQL: The Feed Item Query Language*. Internet-Draft draft-nottingham-atompub-fiql-00. IETF Secretariat, Dec. 2007. URL: http://www.ietf.org/internet-drafts/draft-nottingham-atompub-fiql-00.txt.

[Ope16] Opengroup. *What Is SOA?* 2016. URL: https://web.archive.org/web/20160819141303/opengroup.org/soa/source-book/soa/soa.htm. [Archived from the original on August 19, 2016, accessed 14-January-2018].

[PF13] Sandro Pasquali and Kevin Faaborg. *Mastering Node.js*. Packt Publishing, 2013, pp. I–XXXII, 1–577. ISBN: 978-1-78588-896-0.

[Pos81] Jon Postel. *Transmission Control Protocol*. STD 7. RFC Editor, Sept. 1981. URL: http://www.rfc-editor.org/rfc/rfc793.txt.

[Pre17] Tom Preston-Werner. *Semantic Versioning 2.0.0*. 2017. URL: https://semver.org/#semantic-versioning-200. [Online; accessed 14-March-2018].

[Rhy17] Andrew E. Rhyne. *GraphQL Tutorial - Getting Started*. 2017. URL: https://medium.com/@thebigredgeek/graphql-tutorial-getting-started-f97c8e03156e. [Online; accessed 20-04-2018].

[RA13] Leonard Richardson and Mike Amundsen. *Restful Web APIs*. 2013. ISBN: 978-1-449-35806-8.

[S03] Raymond Eric S. *Basics of the Unix Philosophy*. 2003. URL: http://www.catb.org/~esr/writings/taoup/html/ch01s06.html. [Online; accessed 14-April-2018].

[SAN18] KRISTOPHER SANDOVAL. *7 Unique Benefits of Using GraphQL in Microservices*. 2018. URL: https://nordicapis.com/7-unique-benefits-of-using-graphql-in-microservices/. [Online; accessed 02-05-2018].

[Sca01] Walt Scacchi. *Process Models in Software Engineering*. Tech. rep. Institute for Software Research, University of California, 2001.

[Sha10] Rupen Sharma. *What is the Power/Interest Grid*. 2010. URL: https://www.brighthubpm.com/resource-management/80523-what-is-the-powerinterest-grid/#imgn_0. [Online; accessed 07-April-2018].

[Sta16] Statcounter. *Mobile and tablet internet usage exceeds desktop for first time worldwide*. 2016. URL: http://gs.statcounter.com/press/mobile-and-tablet-internet-usage-exceeds-desktop-for-first-time-worldwide. [Online; accessed 27-December-2017].

[Stu17a] Sashko Stubailo. *The GraphQL stack: How everything fits together*. 2017. URL: https://dev-blog.apollodata.com/the-graphql-stack-how-everything-fits-together-35f8bf34f841. [Online; accessed 16-04-2018].

[Stu17b] Phil Sturgeon. *GraphQL vs REST: Caching*. 2017. URL: https://philsturgeon.uk/api/2017/01/26/graphql-vs-rest-caching/. [Online; accessed 22-04-2018].

[Sut03] A. Sutcliffe. "Scenario-based requirements engineering". In: *Proceedings. 11th IEEE International Requirements Engineering Conference, 2003.* Sept. 2003, pp. 320–329. DOI: 10.1109/ICRE.2003.1232776.

[Sut98] Alistair Sutcliffe. "Scenario-based requirements analysis". In: *Requirements Engineering* 3.1 (Mar. 1998), pp. 48–65. ISSN: 1432-010X. DOI: 10.1007/BF02802920. URL: https://doi.org/10.1007/BF02802920.

[Tho] Rachel Thompson. *Stakeholder Analysis Winning Support for Your Projects.* URL: https://www.mindtools.com/pages/article/newPPM_07.htm. [Online; accessed 12-April-2018].

[Wol16] Eberhard Wolff. *Microservices Flexible Software Architecture.* 2016. ISBN: 978-0-13-214301-1.

Attachment

```
{
  "name": "router",
  "version": "0.0.1-SNAPSHOT",
  "private": true,
  "main": "app.js",
  "scripts": {
    "serve": "nodemon app.js"
  },
  "dependencies": {
    "apollo-link-context": "^1.0.8",
    "apollo-link-http": "^1.3.2",
    "apollo-server-express": "^1.3.4",
    "cors": "^2.8.4",
    "express": "^4.16.2",
    "express-graphql": "^0.6.12",
    "express-jwt": "^5.3.1",
    "express-session": "^1.15.6",
    "graphql": "^0.13.2",
    "graphql-tools": "^2.20.0",
    "graphql-weaver": "^0.11.7",
    "node-fetch": "^2.0.0"
  },
  "devDependencies": {
    "nodemon": "^1.13.3"
  }
}
```

Listing 7.1: Gateway dependencies

```
const express = require('express');
const cors = require('cors');
const bodyParser = require('body-parser');
const graphqlHTTP = require('express-graphql');
const {graphqlExpress, graphiqlExpress} = require('apollo-server-express');
const {weaver} = require('./src/weave');
const {apollo, apolloNoStitch} = require('./src/apollo');
const jwt = require('express-jwt');
const app = express();

const endpoints = [
    'http://localhost:4445/graphql', // User service
```

```
13      'http://localhost:4444/graphql', // Page service
14      'http://localhost:4446/graphql'  // personal messages
15  ];
16
17  const startGateway = (schema) => {
18      app.use('/graphql', jwt({
19          secret: 'my-secret',
20          requestProperty: 'auth',
21          credentialsRequired: false,
22      }));
23      app.use('/graphql',
24          graphqlHTTP(request => ({
25              schema: schema,
26              graphiql: true,
27              context: request
28          })));
29      app.listen(3000, () => console.log('Gateway on 3000'));
30  };
31
32  /* weaver for simple schema stitching with DataLoader*/
33  // weaver(endpoints).then(schema => startGateway(schema));
34
35  /* apollo for sophisticated schema manipulation and authorization */
36  apollo(endpoints).then(schema => startGateway(schema));
37
38  /* No Sitch for resilient gateway */
39  // apolloNoStitch(endpoints).then(schema => startGateway(schema));
```

Listing 7.2: Gateway entry file

```
1   const {mergeSchemas, makeRemoteExecutableSchema, introspectSchema} =
2       require('graphql-tools');
3   const fetch = require('node-fetch');
4   const {HttpLink} = require('apollo-link-http');
5   const {setContext} = require('apollo-link-context');
6
7   async function getIntrospectSchema(url) {
8
9       const http = new HttpLink({uri: url, fetch});
10
11      const link = setContext((request, previousContext) => ({
12          headers: {
13              'Authentication': "Bearer ..." ,
14          }
15      })).concat(http);
```

```
16      try {
17          const schema = await introspectSchema(link);
18          return makeRemoteExecutableSchema({schema, link,});
19      } catch (e) {
20          console.log(e)
21      }
22  }
23
24  async function apolloNoStitch(endpoints) {
25      const allSchemas = await Promise.all(endpoints.map(ep =>
26          getIntrospectSchema(ep)));
27      return mergeSchemas({
28          schemas: allSchemas
29      })
30  }
31
32  async function apollo(endpoints) {
33      const allSchemas = await Promise.all(endpoints.map(ep =>
34          getIntrospectSchema(ep)));
35      const linkSchemas = `
36              extend type Page {
37                  authorFromUserService: User
38              }
39
40              extend type Message  {
41                  from_UserService: User
42                  recommendedPage_PageService: Page
43              }
44          `;
45      return mergeSchemas({
46      schemas: [...allSchemas, linkSchemas],
47      resolvers: mergeInfo => ({
48          Page: {
49              authorFromUserService: {
50                  fragment: `fragment ExtendedUser on Page { author }`,
51                  resolve: (p, a, c, i) => {
52                      return mergeInfo.delegate('query', 'user',
53                          {id: p.author}, c, i)
54                  }
55              }
56          },
57          Message: {
58              from_UserService: {
59                  fragment: `fragment ExtendedUser on Message { from }`,
60                  resolve: (parent, args, context, info) => {
61                      return mergeInfo.delegate('query', 'user',
62                          {id: parent.from}, context, info);
63                  }
```

```
64              },
65              recommendedPage_PageService: {
66              fragment:`fragment ExtendPage on Message {recommendedPage}`,
67              resolve: (parent, args, context, info) => {
68                  return mergeInfo.delegate('query', 'getPage',
69                      {id: parent.recommendedPage}, context, info);
70              }
71              }
72          }
73      })
74      });
75  }
76
77  module.exports = {apollo, apolloNoStitch};
```

Listing 7.3: Gateway schema stitching with Apollo

```
1   const {weaveSchemas} = require('graphql-weaver');
2
3   async function weaver(endpoints) {
4       return await weaveSchemas({
5           endpoints: [
6               {
7                   //namespace: 'UserService',
8                   url: endpoints[0], //UserService
9               },
10              {
11                  //namespace: 'PageService',
12                  url: endpoints[1],
13                  fieldMetadata: {
14                      'Page.author': {
15                          link: {
16                              field: 'user',
17                              argument: 'id',
18                              batchMode: false
19                          }
20                      },
21                  }
22              },
23              {
24                  //namespace: 'Personal Messages',
25                  url: endpoints[2],
26                  fieldMetadata: {
27                      'Message.fromUser': {
28                          link: {
```

```
                                field: 'users',
                                argument: 'id',
                                batchMode: true,
                            }
                        },
                        'Message.recommendedPage': {
                            link: {
                                field: 'getPage',
                                argument: 'id',
                                batchMode: false
                            }
                        }
                    }
                }
            ]
        });
}
module.exports = {weaver};
```

Listing 7.4: Gateway schema stitching with graphql weaver

```
{
  "name": "user-test",
  "version": "0.0.1-SNAPSHOT",
  "private": true,
  "main": "app.js",
  "scripts": {
    "serve": "nodemon app.js"
  },
  "dependencies": {
    "axios": "^0.18.0",
    "express": "^4.16.2",
    "express-graphql": "^0.6.12",
    "graphql": "^0.13.2"
  },
  "devDependencies": {
    "nodemon": "^1.13.3"
  }
}
```

Listing 7.5: User service dependencies

```
const express = require('express');
```

```
2  const expressGraphQL = require('express-graphql');
3  const {schema, resolver} = require('./src/buildSchema');
4  const app = express();
5  app.use('/', require('./src/restRoutes'));
6  app.use('/graphql', expressGraphQL({
7      schema: schema,
8      rootValue: resolver,
9      graphiql: true
10 }));
11 app.listen(4445, console.log('UserService on port 4445'));
```

Listing 7.6: User service entry file

```
1  const {buildSchema} = require('graphql');
2  const userData = require('../resource/userData.json');
3  const http = require('axios');
4  const fs = require('fs');
5
6  const schema = buildSchema(`
7
8      type Query {
9         "retrieve single user information"
10        user(id: Int!): User!
11        "retrieve many users"
12        users(id: [Int]): [User]
13        userFromREST(id: ID): User
14     }
15
16     type User {
17         userID: ID!
18         name: String!
19         email: String!
20         profilePage: String
21         image: String
22      }
23 `);
24
25 class User {
26     constructor(obj) {
27         this.userID = obj.userID;
28         this.name = obj.name;
29         this.email = obj.email;
30         this.profilePage = obj.profilePage;
31     }
32
```

```
33      image() {
34          let pr = new Promise((res, rej) => {
35              fs.readFile('./resource/default.png', (error, data) =>
36              res(new Buffer(data, 'base64')));
37          })
38          return pr.then(r => r)
39      }
40  }
41
42  const resolver = {
43      user: ({id}, context) => getUserByID(id, context),
44      users: ({id}) => getUsersByID(id),
45      userFromREST: ({id}) => getUserFromREST(id)
46  };
47
48  //////// retrieval logic //////////
49  function getUserByID(id, context) {
50      console.log('calling userservice GraphQL with id', id);
51      let found = userData.filter(p => p.userID === parseInt(id));
52      if (found.length === 0) {
53          context.res.status(404);
54          throw new Error('User Not Found');
55      }
56      return new User(found[0])
57  }
58
59  function getUsersByID(id) {
60      console.log('Userservice : users(', id, ')');
61      const idsToInt = id.map(x => parseInt(x));
62      let found = userData.filter(p => idsToInt.includes(p.userID));
63      return found
64  }
65
66  function getUserFromREST(id) {
67      return http.get(`http://localhost:4445/api/v1/user/${id}`)
68          .then(response => response.data)
69          .catch(error => new Error('User Not Found'))
70  }
71  module.exports = {schema, resolver};
```

Listing 7.7: User service schema and resolvers

```
1  const express = require('express');
2  const userData = require('../resource/userData.json');
3  const router = express.Router();
```

```
4  const path = require('path');
5  const fs = require('fs');
6
7  router.get('/api/v1/user/:id', (req, res) => {
8      const user = getUserByID(req.params.id);
9      if (user === null) {
10         return res.status(404).send("User Not found")
11     }
12     return res.status(200).send(user);
13 });
14
15 router.get('/api/v1/user/:id/image', (req, res) => {
16     fs.readFile('./resource/default.png', (error, data) => {
17         let img = new Buffer(data, 'base64');
18         res.set('Content-Type', 'image/jpeg');
19         res.send(img)
20     })
21 });
22
23 function getUserByID(id) {
24     const user = userData.filter(p => p.userID === parseInt(id));
25     return user.length === 0 ? null : user[0];
26 }
27
28 module.exports = router;
```

Listing 7.8: User service REST endpoints

```
1  [
2    {
3      "userID": 1,
4      "name": "Gretchen Wiley",
5      "email": "gretchenwiley@recrisys.com",
6      "profilePage": "www.google.de"
7    },
8    {
9      "userID": 2,
10     "name": "Cathleen Garrison",
11     "email": "cathleengarrison@recrisys.com",
12     "profilePage": null
13   },
14   {
15     "userID": 3,
16     "name": "Ester Lindsey",
17     "email": "esterlindsey@recrisys.com",
```

```
18      "profilePage": "www.google.de"
19    }
20  ]
```

Listing 7.9: User service extract of user data

```
1  {
2    "name": "service-test",
3    "version": "0.0.1-SNAPSHOT",
4    "private": true,
5    "main": "app.js",
6    "scripts": {
7      "serve": "nodemon app.js"
8    },
9    "dependencies": {
10     "body-parser": "^1.18.2",
11     "express": "^4.16.2",
12     "express-graphql": "^0.6.11",
13     "graphql": "^0.12.3",
14   },
15   "devDependencies": {
16     "nodemon": "^1.13.3"
17   }
18 }
```

Listing 7.10: Personal Message Service dependencies

```
1  const express = require('express');
2  const graphqlHTTP = require('express-graphql');
3  const {schema, resolver} = require('./src/buildSchema');
4  const app = express();
5  app.use('/graphql', graphqlHTTP( req => ({
6      schema: schema,
7      rootValue: resolver,
8      context: req,
9      graphiql: true
10 })));
11 app.use('/', require('./src/restRoutes'));
12 app.listen(4446, () => console.log('PersonalMessages Service Port 4446'));
```

Listing 7.11: Personal Message Service entry file

```
1  var {buildSchema} = require('graphql');
2  const testdata = require('../resource/testdata.json');
3
4  const schema = buildSchema(`
5      ## root Query
6      type Query {
7          personalMessages(userID: Int!): PersonalMessages!
8      }
9
10     type Mutation {
11         "send out a message of type 'personal'"
12         sendPersonalMessage(from:Int!,to:Int!,content:String!):Boolean!
13         recommendPage(to: Int!, content: String!, page: Int!):Boolean!
14     }
15
16      type PersonalMessages {
17          userID: Int
18          messageCount: Int!
19          "Type = 'personal' OR 'recommendation'"
20          messages(type: String): [Message]
21      }
22
23      type Message {
24          content: String
25          type: String
26          date: Date
27          from: Int
28          recommendedPage: Int
29      }
30      scalar Date
31
32      enum MessageType {
33            recommendation
34            personal
35      }
36 `);
37
38 class PersonalMessages {
39     constructor(pm) {
40         this.pm = pm;
41         this.userID = pm.userID;
42         this.messageCount = pm.messages.length
43     }
44     messages(input) {
45         let allMessages = this.pm.messages.map(m => new Message(m));
46         if (input.type) {
47             allMessages = allMessages.filter(mess =>
```

```
48                mess.type === input.type);
49            }
50            return allMessages;
51        }
52    }
53
54    class Message {
55        constructor(m) {
56            console.log(m)
57            this.content = m.content;
58            this.type = m.type;
59            this.date = new Date(m.date);
60            this.from = m.fromUser || m.from;
61            this.recommendedPage = m.recommendedPage;
62        }
63    }
64
65    var resolver = {
66        personalMessages: ({userID}, context, r) =>
67            personalMessages(userID),
68        sendPersonalMessage: ({from, to, content}) =>
69            sendPersonalMessage(from, to, content),
70        recommendPage: ({to, content, recommendedPage}) =>
71            recommendPage(to, content, recommendedPage)
72    };
73
74
75    function personalMessages(userID) {
76        console.log('Personal Message with id', userID);
77        let a = testdata.filter(p => p.userID === userID);
78        return new PersonalMessages(a[0])
79    }
80
81    function sendPersonalMessage(from, to, content) {
82        let user = testdata.find(p => p.userID === to);
83        if (!user) return new Error('User nor found');
84        user.messages.push({
85            content: content,
86            date: new Date(),
87            from: from,
88            type: "personal"
89        });
90        return true
91    }
92
93    function recommendPage(to, content, recommendedPage) {
94        let a = testdata.filter(p => p.userID === to);
95        a[0].messages.push({
```

```
96          content: content,
97          date: new Date(),
98          from: 1,
99          type: "recommendation",
100         recommendedPage: recommendedPage
101     });
102     return true
103 }
104
105 module.exports = {schema, resolver};
```

Listing 7.12: Personal Message Service schema and resolvers

```
1  const express = require('express');
2  const bodyParser = require('body-parser');
3  const data = require('../resource/testdata.json');
4
5  const router = express.Router();
6
7  router.use(bodyParser.urlencoded({ extended: false }));
8
9  router.get('/api/v1/personalMessage/:id', (req, res) => {
10     let personalMessages = data.find(p =>
11         p.userID === parseInt(req.params.id));
12     return res.status(200).send(personalMessages)
13 });
14
15 router.post('/api/v1/sendPersonalMessage/', (req, res) => {
16     let a = testdata.filter(p => p.userID === toUser);
17     a[0].messages.push({content: content, date: new Date(),
18         from: 1, type: "personal"});
19     return true
20 });
21
22 router.post('/api/v1/personalMessage/', (req, res) => {
23     let a = data.filter(p => p.userID === parseInt(req.body.toUser));
24     a[0].messages.push({content: req.body.content, date: new Date(),
25         from: 1, recommendedPage: req.body.recommendedPage});
26     return res.status(200).send('ok')
27 });
28
29 module.exports = router;
```

Listing 7.13: Personal Message Service REST endpoints

```
[
  {
    "userID": 1,
    "messages": [
      {
        "content": "sint qui consectetur",
        "date": "Wed Apr 11 2018 14:35:28 GMT+0200",
        "type": "recommendation",
        "fromUser": 1333,
        "recommendedPage": 8
      },
      {
        "content": "aute mollit excepteur",
        "date": "Wed Apr 11 2018 14:35:28 GMT+0200",
        "type": "recommendation",
        "fromUser": 7,
        "recommendedPage": 1
      },
      {
        "content": "ut pariatur mollit",
        "date": "Wed Apr 11 2018 14:35:28 GMT+0200",
        "type": "recommendation",
        "fromUser": 16,
        "recommendedPage": 5
      },
      {
        "content": "enim cillum cillum",
        "date": "Wed Apr 11 2018 14:35:28 GMT+0200",
        "type": "recommendation",
        "fromUser": 2,
        "recommendedPage": 1222
      }
    ]
  }
]
```

Listing 7.14: Personal Message Service extract of message data

```
{
  "name": "Page-Service",
  "version": "0.0.1-SNAPSHOT",
  "private": true,
  "main": "app.js",
  "scripts": {
    "serve": "nodemon app.js"
```

```
 8    },
 9    "dependencies": {
10      "express": "^4.16.2",
11      "express-graphql": "^0.6.12",
12      "graphql": "^0.12.3"
13    },
14    "devDependencies": {
15      "nodemon": "^1.13.3"
16    }
17  }
```

Listing 7.15: Page service dependencies

```
1  const express = require('express');
2  const graphqlHTTP = require('express-graphql');
3  const {schema, resolver} = require('./src/buildSchema');
4  const app = express();
5  app.use('/graphql', graphqlHTTP({
6      schema: schema,
7      rootValue: resolver,
8      graphiql: true
9  }));
10 app.listen(4444);
```

Listing 7.16: Page service entry file

```
1  const {buildSchema} = require('graphql');
2  const pageData = require('../resource/testdata.json');
3  const schema = buildSchema(`
4
5      type Query {
6          getPage(id: Int): Page!
7      }
8       type Page {
9          ID: ID!
10         title: String
11         author: Int!
12         link: String
13      }
14 `);
15
16 class Page {
17     constructor(data) {
```

```
18          this.ID = data.pageID;
19          this.title = data.pageTitle;
20          this.author = data.author;
21          this.link = data.self.url;
22      }
23  }
24  var resolver = {
25      getPage: ({id}) => {
26          console.log('calling page service with id:', id)
27          let a = pageData.filter(p => p.pageID === id);
28          return new Page(a[0])
29      }
30  };
31  module.exports = {schema, resolver};
```

Listing 7.17: Page service schema and resolvers

```
1  [
2    {
3      "pageID": 1,
4      "pageTitle": "Talkalot",
5      "author": 8,
6      "self": {
7        "url": "bar.de"
8      }
9    },
10   {
11     "pageID": 2,
12     "pageTitle": "Lingoage",
13     "author": 30,
14     "self": {
15       "url": "bar.de"
16     }
17   }
18 ]
```

Listing 7.18: Page service extract of page data