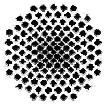 Institute of Architecture of Application Systems 

University of Stuttgart
Universitätsstraße 38
D-70569 Stuttgart

Master Thesis

Transformation of REST API to GraphQL for OpenTOSCA

Eyob Semere Ghebremicael

Course of study:	INFOTECH
First Examiner:	Prof. Dr. Dr. h. c. Frank Leymann
Supervisor:	M.Sc. Kálmán Képes
Commenced:	May 08, 2017
Completed:	November 08, 2017
CR-Classification:	D.2.11, D.2.12

Acknowledgements

I would like to use this opportunity to express my appreciation and thanks to Prof. Dr. Dr. h. c. Frank Leymann and especially to my supervisor M.Sc. Kálmán Képes from the Institute of Architecture of Application Systems (IAAS) at the University of Stuttgart, who believed in me towards achieving the outcome of this thesis work. Your encouragement, guidance and advice have been priceless in accomplishing the objectives of this research work.

I would also like to thank Brotfür die Welt and Diocese of Rottenburg-Stuttgart for their financial support during the thesis period.

Finally I would like to express my deep appreciation to my family and friends for their patience and encouragements. Indeed, I do not have words to explain the encouragement I received from my mother, my father, my brothers and my sister throughout the study period.

Abstract

Software has become ubiquitous in our lives delivering a diversity of functionality. These software applications may have diverse development backgrounds but they need to interact between each other for many reasons. One way to make software communicate between each other is using Application Programming Interfaces (APIs). Therefore, APIs play an important role in the design of application software architectures. Moreover, the design of these software architectures can be described by the architectural style residing behind it. Representational State Transfer (REST) is a well known architectural style that has been used as a guide to the design and development of the architecture of modern web. For simplicity reasons, REST APIs have been adored by most software developers compared to all its previous approaches. But there is concern over its effect on performance when the size of the applications on the client side grows (e.g. multiple REST calls).An alternative approach is needed to prevent or minimize these negative effects. In this research, Graph Query Language (GraphQL) is considered as an alternative for REST API. Furthermore, we developed a generic concept for the transformation of REST API to GraphQL. We also validated our concepts by prototypical implementations.

Table of Contents

List of Figures

List of Listings

List of Tables

List of Abbreviations

API	Application Programming Interface
AST	Abstract Syntax Tree
EAI	Enterprise Application Integration
EJB	Enterprise Java Beans
ESB	Enterprise Service Bus
FTP	File Transfer Protocol
GUI	Graphical User Interface
GraphiQL	Graphical Query language
GraphQL	Graph Query Language
HATEOAS	Hypermedia as the Engine of Application State
HTML	Hyper Text Markup Language
HTTP	Hypertext Transfer Protocol
IPC	Inter-Process Communication
IT	Information Technology
NPM	Node Packaging Module
RPC	Remote Procedure Call
REST	Representational State Transfer
SSH	Secure Shell
SMTP	Simple Mail Transfer Protocol
TOSCA	Topology and Orchestration Specification for Cloud Applications
URL	Universal Resource Locator
UI	User Interface
URI	Universal Resource Identifier
WSDL	Web Services Description Language
WS-*	Web Service Specification
WWW	World Wide Web
W3C	World Wide Web Consortium
XML	Extensible Markup Language
XSD	XML Schema Document

1. INTRODUCTION

At its most basic level, an API allows products or services to talk between each other. For instance an API allows software developers to open the door to the data and functionality of their products or services to other developers, to other businesses or even to other members of a department within a company [Lane2013]. As a consequence of the interaction, companies are increasingly exchanging data, services and complex resources. This exchange can be internally, with external partners or even openly with the public. APIs are widely used for commerce, payments, social, cloud computing and much more. However, mobile phones and tablets in the recent times are the motivating forces for providing APIs and consuming them.

Representational State Transfer (REST) architectural style has been widely adopted by service providers and majority of the software developer community. This is because it is simple to use compared to earlier web service specifications like SOAP and Remote Procedure Call (RPC). Furthermore, the success of the REST architectural style could be related to the constraints that restrict the way service components should be developed [Feilding2000].Those constraints when applied could ensure improvements in the overall system (See chapter 2).

Finding an alternative solution to REST API has been one of the priorities in the software developers' community. The limitations of REST started to impact the software development in many ways. Pioneer companies in the software industry and software developers have worked hard to find an alternative solution for REST API. For instance the impact of REST with the increased mobile usage, low-powered devices and sloppy networks were some of the main reasons behind Facebook's creation to GraphQL [Gcool2017]. Interestingly, other companies like Netflix and Coursera were working on similar ideas, to make API interactions more efficient. Coursera envisioned a similar technology to let a client specify its data requirements and Netflix even open-sourced their solution called Falcor. After GraphQL was open-sourced, Coursera completely cancelled their own efforts and were convinced to use the GraphQL. According to [GQL2017], today GraphQL is used in production by lots of different companies such as GitHub, Twitter, Yelp and Shopify - to name only a few.

1.1 Motivation and Problem Statement

As we can see it, the world of software moves fast. Considering this, software developers and architects always try to balance simplicity vs. complexity; optimization vs. completion. It is true that REST is simple to be utilized and simplicity is good, but the simplicity of REST also leads to some of its limitations. When deploying REST, the service provider determines what data or functionality will be sent down to the service consumer or client application [Wachter2016]. This is because REST grew up in the age where the service provider (server) dominated the web application landscape. This could be fine if the developed client application is small in size but the problem arises when it grows in size and becomes more complex. It is obvious that the application will continue to evolve as long as new requirements arise. Take for example different UI components are on the same page of the client application and each component expects its own response from a certain API endpoint. In the client application's perspective, this could make its code elegant and simple. But it needs to execute *multiple API calls* in order to display what is requested by the user. Indirectly, the service provider has more responsibility to manage the response and the service consumer has to expect its response accordingly. Indeed the service consumer doesn't have much role in managing the response data or functionality. However, as Jonas et al. [Jonas2016] has put it "While just a few years ago most websites used to be rendered on the server and have only relatively little client-side logic, the opposite is true of new apps today. Single-page applications and clients that implement complex logic are the new reality."Therefore, this is the time where the service consumer getsinvolved in deciding aboutwhat response to receive or enabling providers to easily create new API for their clients.

The above simple use case in itself exhibits several shortcomings of using REST API [Samer2017].First and the biggest problem here is the nature of multiple endpoints that force the client applications to undergo multiple round-trips to get the targeted response of their corresponding request. Second, the client application doesn't have its own request language to help it control over what data the service provider will return. There is no language at all or the language available for client application is very limited(e.g. MIME types). The client application may receive huge response with unnecessary data. If the client application has control over the response then this problem may be prevented. Third, it makes the client application to be highly dependent on the service provider hence leads to some problems like inability to independent client application

development. Fourth, the growing number of the endpoints (with different API for the same functionality) in the service providers cause big *versioning* problem [Samer2017]. To support multiple versions means to create new endpoints. This may cause maintenance problem or duplication of code on the provider's side.

The above raised points impact those applications that run on devices with sloppy networks, low-powered devices and in mobile device usage such as smart phones. The problem on those applications of these devices in its turn affects user satisfaction. Therefore, looking an alternative solution that overcomes these and other related problems is the motivation behind this research.

1.2 Thesis Objectives

The general objective of the research is to assess the current available tools in the area of REST API to GraphQL transformation and then the experience attained will be used as an input to the concept development and the prototype implementation. Moreover, the prototype tool will work for the OpenTOSCA (Topology and Orchestration Specification for Cloud Applications) ecosystem.

Specific objectives:

- Identify commonly used existing tools in the field of API transformation in general and in the area of REST to GraphQL API transformation in particular.
- Technically assess the sample of tools and determine the good features as well as the limitations of each tool. The challenges and features experienced from these tools will be used as an introductory to the concept development.
- Develop a generic concept that solves the stated problems
- Validate the concept developed using a prototypical implementation.

1.3 Outline

The remaining document is structured as follows:

Chapter 2: The fundamentals with key concepts and their description for understanding rest of the report.

Chapter 3: The related Work of this thesis such as ANY2API Apification framework and REST to GraphQL transformation tools will be discussed.

Chapter 4: Concept Development with detailed overview of the architecture of the proposed system is discussed here.

Chapter 5: Validation of the concept developed by implementing a prototype realizing our concept.

Chapter 6: Conclusion and Future work summarizes the work done and describes possible areas of future work.

CHAPTER 2
FUNDAMENTALS

This chapter elaborates the concepts and technologies that will frequently appear in the next chapters. In order to make it as a foundation for the understanding of the key components of the proposed concept, many references have been mentioned for better clarity of the main topics. In the first section, the fundamental overview of APIs will be explained and this will be an introduction to the next sections. Next to this will cover about the commonly used REST API. After this, detailed discussion about GraphQL is given. Finally, SOA and Microservices will be discussed.

2.1 APIs

This section explains about a set of functions or procedures that allow the creation of applications that can access the features or data of an operating system, application, or other service. These are known as *Application Programming Interfaces (APIs).*

Nowadays, APIs have become ubiquitous components of software infrastructures. They can be found everywhere; from household equipments such as refrigerator to sophisticated technologies like space stations. Moreover they are part of the dynamically evolving mobile devices; where users are using the devices in their daily lives. Web applications, back-end systems and platforms for mobile apps in particular provide APIs [Richardson2013]. API can be defined as a set of functions or procedures used by computer programs to access operating system services, software libraries, or other systems [BBVA2016]. Just like a user interface which allows interaction and communication between software and an individual, an API facilitates communication between two applications so that functionalities are exchanged between them.

Modern applications have the need to access services (data or functionality) from a remote system. Here comes the responsibility of the API to provide an interface to the stored data or functionality that fits an application’s needs. In this case an API represents a contract between the data or functionality provided by the service provider and the consumers who want to interact with it [Hunter2017]. Hence it determines how a

client can get services from the remote system. As Wettinger et al. [Wettinger2015] has put it, the client uses the library (set of functions and procedures) offered by the remote system, as an abstraction layer on top of an area to access and exchange additional information. Thus, they both use each other's information without compromising their independence. This is the basis for integrating and orchestrating different applications and application components, enabling systematic development and reliable operations of distributed applications, mash-up applications, and mobile applications. APIs can have stakeholders such as API designers, API users and consumers of the resulting product [Stylos2007]. In addition to that APIs are used to create integration of applications with business partners, suppliers, and customers as shown in Figure 2.1[Rudrakshi2014]. As Stylos et al. [Stylos2006] has described it, APIs can improve the development speed, contribute to higher quality software and increase the reusability of software.

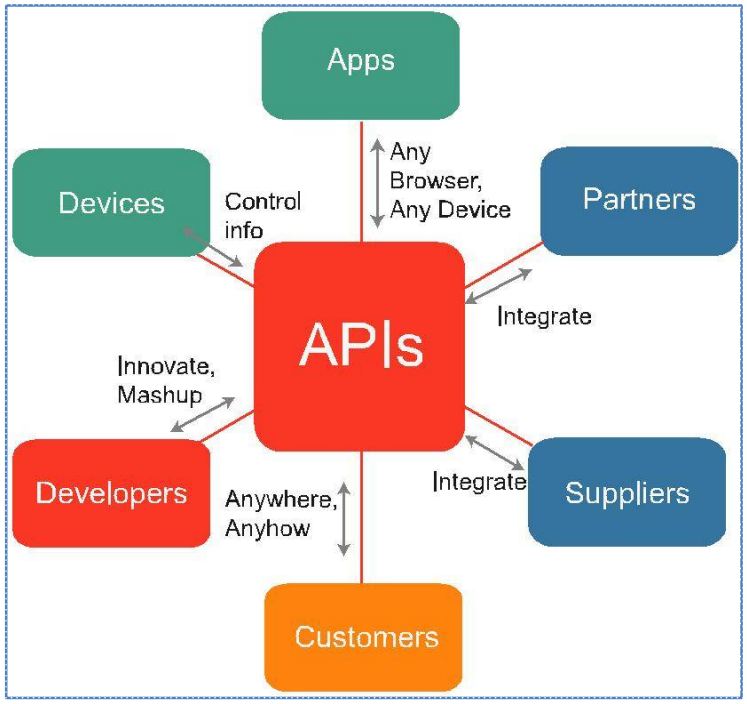

Figure 2.1: Possible stakeholders of an API, taken from [Rudrakshi2014]

According to Granli et al. [Granli2015], APIs can be considered as having three interacting layers; the public interface, the actual implementation of the functionality and an intersecting layer which provides utilities such as error handling, third party libraries and additional auxiliary features. Typically, the interfaces provide the definitions of functions and data structures while the implementation realizes those interfaces.

Technically, APIs can be exposed and utilized in the forms of libraries that are bound to a particular programming language or in the form of language-agnostic Web services [Wettinger2015]. Web-based RESTful APIs [Masse2011] or WSDL/SOAP-based services [W3C2003] are commonly known forms for providing and using APIs. These APIs provide an interface for web applications or for applications that need to connect or communicate each other via the Internet. The number of publicly available Web APIs is constantly growing, especially with the increased innovations in mobile devices [API2017]. These web APIs can be used to do everything from checking traffic and weather, to updating your social media status, or to make payments.

One of the greatest challenges of building an API is building one that will last long and the software developer community always looks at four essential features to rate the quality of an API[BBVA2016]; it must be useful and easy to understand, stable when making improvements, it has to be secured and also has to provide good documentation. Several styles or protocols are being used in building web APIs of which REST is the most popular.

2.2 REST

This section discusses about an architectural style that commonly used to design Web APIs. This style is known as *Representational State Transfer (REST).*

The term "REST" was introduced in 2000 in the doctoral dissertation of Roy Fielding, one of the principal authors of the Hypertext Transfer Protocol (HTTP) specification. REST is an architectural style for distributed hypermedia systems and it is defined based on a set of constraints. According to Fielding et al. [Fielding2000] REST emphasizes on a set of constraints such as scalability of interactions, generality of interfaces, independent deployment and intermediaries of components to

reduceinteraction latency enforce security and encapsulate legacy systems. The REST constraints which are derived from common architectural styles are chosen for the properties they induce on candidate architectures. Any architecture compliant with these constraints can be called REST (or RESTful) architecture [Haupt2014]. It uses the core technologies of the World Wide Web (WWW) such as HTTP together with URIs and MIME type, to promote simplicity, standards-based interoperability, and ubiquitous availability on all kind of platforms [BBVA2016]. To make it clear, REST is any interface between systems using transport protocols like HTTP to obtain service and generate operations on it in all possible formats, such as extensible Markup Language (XML) and Java Script Object Notation (JSON).

Fielding followed a constraint-based approach in the process of discovering for REST. Therefore he identified some constraints and REST is governed by those constraints'. These constraints are as follows [Fielding2000]:

- *Stateless:* Statelessness is key constraint and that's why REST an acronym for **Re**presentational **S**tate**T**ransfer [Fredrich2015]. In the request/response paradigm between the client and server the necessary state to handle the request is contained within the request itself.
- *Uniform interface:* In the case of using HTTP protocol for REST APIs, communication is initiated by the client and it is consisted of a request followed by a response message. Each request message together with the resource identifier includes specific actions or the HTTP verb (e.g., GET, PUT, POST, and DELETE) that define the operation to be performed on the resource [Haupt2015].
- *Client–server*: The uniform interface that separates clients from servers allows clients not to be concerned about the internal affairs of servers and servers also don't care about the user interface or user state of clients. For instance clients are not concerned with data storage details of each server so the performance of the client code is improved and servers can become simpler and more scalable for not concerned about user interface or user state. Therefore Servers and clients can also be replaced and developed independently [Fredrich2015].

- *Resource identification through URI:* It is the Universal Resource Identifier (URI) and no other element that is the sole identifier of each resource in this REST system. The URI allows us to access the information in order to change or delete it, or for example to share its exact location with third parties [Fielding2000].
- *Layer system:* Hierarchical architecture between the components. A client cannot normally tell whether it is directly connected to the end server, or to an intermediary along the way. Intermediary servers may improve system scalability by enabling load-balancing and by providing shared caches [Fredrich2015]. Layers may also enforce security policies. Each layer within the REST system has functionality.
- *Cacheable:* As on the WWW, clients can cache responses. But responses must implicitly or explicitly identify themselves as cacheable or not so that to avoid further requests of clients from reusing stale or inappropriate data in the response. If well managed, caching can partially or completely remove client– server interactions which then further improves scalability and performance [Fredrich2015].
- *Stateful interactions through hyperlinks:* Hypermedia allows the user to browse the set of objects through hypermedia links. In the case of a REST API, the concept of hypermedia explains the capacity of an application development interface to provide the client and the user with the adequate links to run specific actions on the data. To make it genuine, REST APIs should support the Hypermedia as the Engine of Application State (HATEOAS) principle. According to Haupt et al. [Haupt2017], this principle ensures that whenever a request is made and a response is returned from the server, then part of the information contained in the response will be the browsing hyperlinks associated to other client resources These hyperlinks tell the client where it can go next and what actions are possible in the current state of its conversation with the API. It demands that clients of a REST API are guided by the responses they receive from an API.

RESTful technology is based upon characteristic elements known as *resources,* which are sources of specific information. To make it clear, resources are the building blocks of each RESTful Web API and they provide a uniform interface that enables to access and modify their state [Haupt2015]. Each of them is linked to a global identifier, for example a URI. In order to interoperate with a resource, an application must possess

both the resource's identifier and the required method. On the opposite, there is no need to know the services implementation and system configuration, i.e. whether there are caches, proxies, gateways, firewalls, tunnels, or anything else between the application and the server which hosts the resources. However, the application must be capable of interpreting the data format (representation) returned from the resource, which is often an HTML or XML document, though it may also be an image, plain text, or any other content [Webber2010].These resources are accessed by components of the network (user agents and servers) which communicate through a standardized protocol (e.g. HTTP) and exchange content (representations) of these resources.

REST is an increasingly popular alternative to other standard data exchange protocols such as Simple Object Access Protocol (SOAP), which have a high capacity but are also very complex. Sometimes it is preferable to use a simpler data-processing solution such as REST. Table 2.1 summarizes the advantages and disadvantages of using REST [Albreshne2009]. REST has been a popular way to expose data from a server. During the time when the concept of REST was developed, client applications were relatively simple and the development pace wasn't nearly where it is today. REST thus was a good fit for many applications. However, the API landscape has radically changed over the last couple of years. In particular there are three factors that have been challenging the way APIs are designed: Increased mobile usage, Variety of frontend frameworks or platforms and Fast development. These factors in turn lead to some other problems [GQL2017]:

- *Increased mobile usage* lead to the need for efficient data loading or minimal data transfer
- *Variety of different frontend frameworks and platforms:* The heterogeneous landscape of frontend frameworks and platforms that run client applications makes it difficult to build and maintain one API that would fit the requirements of all.
- *Fast development & expectation for rapid feature development:* With REST API modification on the server side leads to changes on the client side.

Advantages and Disadvantages of REST	
Advantages	**Disadvantages**
✓ Simple: applies many existing well-known standards (HTTP, XML, URI, and MIME) ✓ HTTP clients and servers are compatible with all programming languages and operating system/hardware platforms ✓ Small effort is needed to build a client and Services can be tested using simply a mere web browser ✓ Allows discovering Web resources without any discovery or registry repository.	- Encoding a large amount of input data in the resource URI is impossible - May also be challenging to encode complex data structures into URI - Restful web services currently have no standard grammar to describe web services, like what Web Service Description Language (WSDL) do in SOAP. - No standard vocabulary to define the web service interface and an agreement has to be established between the service consumer and service producer.

Table 2.1: Advantages and disadvantages of REST API [Albreshne2009].

2.3 GraphQL

This section explains about one of the current hot topics amongst software developer community. It is thought to be an alternative an alternative to REST. This new API technology is called *Graph Query Language (GraphQL).*

It is true that REST has become the standard for designing web APIs for more than a decade. However, it has also shown to be too inflexible to keep up with the rapidly changing requirements of the clients that access them [Gupta2017]. Particularly, when

REST is used then clients' responsibility on the request and response of the API functionality is nominal. Hence, the client doesn't has much control on what specific functionality to request or what functionality to receive because almost all is provided by the service provider.

GraphQL was developed to cope with the need to give more responsibility for clients to enhance flexibility and efficiency [Bruno2017]. As it has been described by Gupta et al. [Gupta2017], GraphQL provides solution for the many limitations and inefficiencies experienced by developers who interact with REST APIs. For example, GraphQL gives the user an opportunity to request whatever specific information is needed. To the contrary the REST user is forced to do additional requests in order to fetch the specific information needed. This is also possible on the good will of the service provider; if the service provider doesn't provide an endpoint for that request then there is no way to fetch the specific information needed by the user. These are illustrated in Figure 2.2. Therefore, GraphQL tries to improve the way clients communicate with remote systems.

GraphQL is often confused with being a database technology [Stubailo2016].This is a misunderstanding; GraphQL is a query language for APIs and not even for databases. Moreover, GraphQL is database agnostic and it can be suited well in any context where an API is involved. Similar to REST server, a GraphQL server isn't bounded to anyspecific technology or language, and can be implemented using any technology [Bela2015]. GraphQL's power comes from a simple idea, instead of defining the structure of responses on the server; the flexibility is given to the client. As it is clearly illustrated in Figure 2.3, each request specifies what fields and relationships it wants to get back, and GraphQL will construct a response for this particular request. Some of the benefits of GraphQL compared to REST are [Bela2015]:

- *No more Over-fetching- and Under-fetching:*

Over-fetching and under-fetching is very common problem with REST where as it is avoided by GraphQL. As Nilan et al. [Nilan2016] articulated it, "*REST enables semantic data fetching whereas GraphQL enables declarative data fetching."* Unlike that of REST, in GraphQL a client can request specific data it needs from an API. On the other

side, a GraphQL server responds with the precise data a client asked for. This is well illustrated in Figure 2.2.

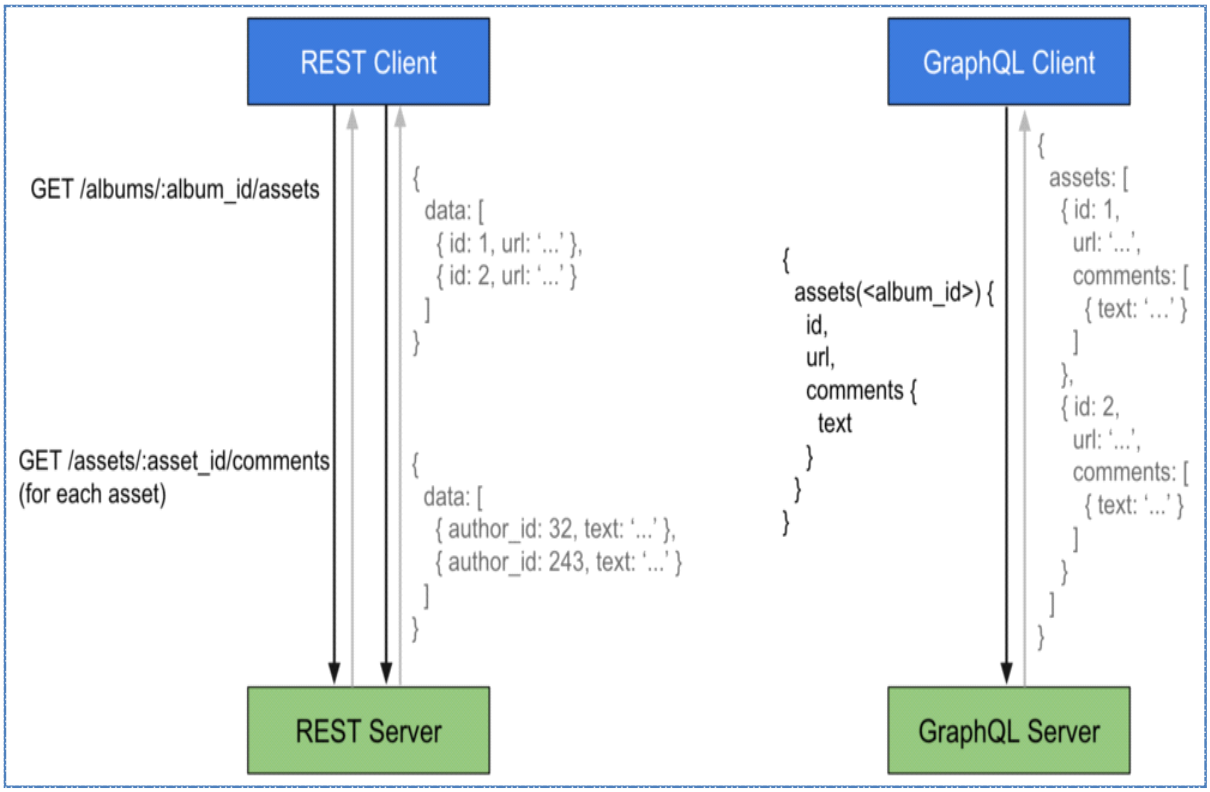

Figure 2.2: Illustration of Data fetching: Left side is using REST where as right side using GraphQL, adopted from [Bela2015].

- *Rapid Frontend development:*

The major limitation of REST is that it doesn't allow for rapid development on the frontend [GQL2017]. With every change that is made to the UI, there is a high probability that more data may be needed now than before. Consequently, the backend needs to be adjusted as well to account for the new data needs. This kills productivity and notably slows down the ability to incorporate user feedback into a product.

- *Intuitive Analytics on the Backend:*

Whenever the client requests data, GraphQL allows the client to have a good understanding of the data at the backend. This is because each client has the power to

request specific information it is interested in. Thus, it is possible to have deep understanding of how the available data is being used at the backend. This has some benefits; the client can have a role in evolving the API or deprecating any fields that are not requested by any clients any more.

- *Benefits of a Schema & Type System:*

GraphQL uses a strong type system to define the capabilities of an API [Gupta2017]. GraphQL has schema which is written down using the GraphQL Schema Definition Language (SDL) and all the types of the API are part of the schema. This schema acts as a *contract* between the server and the client [GQL2017]. Indirectly, the schema defines how a client can access the data. Once the schema is constructed, the frontend and backend teams can do their tasks independently and without further communication since both are aware of the structure of the data that's transferred over the network.

Furthermore, working with a GraphQL API on the frontend is good opportunity to implement further abstractions so that to help implement common functionality on the client-side. For example queries and mutations can be sent without constructing HTTP requests; there is no need to deal with low-level networking details. In addition to that validation and optimization of the queries based on the schema can be done.

GraphQL services define a set of types that are used to describe the set of possible data that can be queried from the service. These data types are part of the GraphQL schema. Whenever requests arrive at the GraphQL service, then they are validated and executed against the GraphQL schema [GQLorg2017].

A) Commonly used terms in GraphQL

- *GraphQL schema*: It is the backbone of the query or mutation execution. The request is always executed according to the structure and context of the schema. A GraphQL schema is composed of special root types or entry points*: query*, *mutation* and *subscriptions* [GQLorg2017, GQLspecs2016].
- *Query:* commonly used entry point/root type and composed of fields and data types. It is used to fetch data and not associated with manipulation of data at the backend. It fetches data according to the *fields* underneath it [GQLorg2017, GQLspecs2016].

- *Mutation:* A way for the client to speak with the server. It is used to manipulate the data at the backend. It creates and updates entries at the backend according to the *fields* underneath it [GQLorg2017] [GQLspecs2016].
- *Subscription:* This is only used when real time interaction with the server is needed; in order to get informed immediately about important events [GQLorg2017][GQLspecs2016].
- *Object Types:* These are the most basic component of GraphQL schema which determines the kind of object to fetch from the service. They also determine what fields the GraphQL service offers. The object type has a name and fields; those fields have to be resolved at some point. The resolved concrete data is scalar type and represents the leaves of the query. Mutations and Queries are special object types that act as an *entry point* of every GraphQL query. The GraphQL schema language supports the scalar types of *String, Int, Float, Boolean* and *ID.* [GQLorg2017].
- *Fields:* GraphQL is about asking for specific fields on objects. These fields can represent either scalar data types or objects. Each field is executed according to the *resolvers* underneath it. [GQLorg2017].
- *Arguments:* In GraphQL, every field and nested object can have its own set of arguments and this helps for making diverse API fetches [GQLorg2017][GQLspecs2016].
- *Resolvers:* These are functions used to fetch the data of the fields. Each function corresponds to exactly one field of the payload [GQLorg2017].

B) Client Application

It is common to see GraphQL backbends expose their API over HTTP where queries and mutations can be sent in the *body* of a POST request [GQLorg2017]. For example with *express-graphql*, an endpoint can be mounted on a GraphQL server and HTTP POST request can be sent on to it. This operation can be done in a variety of ways; using developer console from browser or with *curl* from the command prompt. The GraphQL query is passed as the *query* field in a JSON payload.

However, as it is well explained in [Graphcool2017], using these procedures leads to some challenges in working with a GraphQL backend. For example problem arises in caching data that is returned by the server, difficulty in UI framework integration, inability to keep the local cache consistent after a mutation, difficulty in managing up web sockets for GraphQL subscriptions (which enables real-time updates) and also difficulty in applying pagination for collections. This can become complex operation with the increased size of the query. Therefore switching to a standard client application becomes imminent.

A GraphQL client should have at kind of functionality that doesn't force clients to handle the above challenges [Graphcool2017]. Instead, the client has to completely concentrate on the domain of the application or on implementing the specific requirements of the application. As explained before, GraphQL API has a more interesting structure compared to the REST API. Therefore, these GraphQL clients are expected to exploit this feature of GraphQL. There are several powerful clients that exploit the underlying structure of GraphQL API of which Relay and Apollo client are the most famous [GQLorg2017]. These GraphQL clients have the capability to handle batching, caching and other features automatically.

- ***Apollo Client*** is developed by the effort of a community and it is a powerful and flexible GraphQL client which can work for major development platforms. It is framework agnostic and can supports variety of frameworks such as React and Angular.
- ***Relay*** is developed by Facebook and mostly focuses on performance optimization. It is based on JavaScript framework and used for building data-driven React applications

To summarize their differences, one major difference between Relay and Apollo is in the flexibility of the two approaches [Graphcool2017]. Relay doesn't give a lot of freedom to developers on want to structure of the application whereas Apollo gives variety options ranging from lightweight integrations to much more sophisticated approaches. Therefore, Relay is preferable for large-scale applications that have complex data requirements and many dependencies between different parts of the

application. In which maintaining these dependencies by hand would be cumbersome and error-prone. On the other hand, Apollo provides a much more lightweight and flexible approach that works in any platform or environment.

GraphiQL (note the i, “graphical”) is also an alternative to client applications. It is commonly used during testing and development but should be disabled in production by default. GraphiQL is aware of the semantics of the data and it provides exploring and debugging means where the other alternatives like *Curl* don’t have [Allsopp2016]. It supports debugging by giving hints and pointing out to errors as the user types. Furthermore, GraphQL is good in documentation which GraphiQL can leverage it. The response of GraphQL doesn’t have to be JSON only but GraphiQL comes with a JSON viewer which is preferred one. GraphiQL is well used during the prototype and validation of this research work and an example of it is shown in Figure 2.4.

GraphQL Introspection is used to understand what fields and types of a GraphQL schema are available [Alligator2017, GQLorg2017]. It is through this introspection system that GraphiQL has the ability to provide documentation about the schema. For example to know what types are available then it is possible to ask GraphQL by typing __schema (double underscore) which will provide the type definitions starting from the root. It will output mix of custom types as well as built-in scalar types. __schema, __Type, __TypeKind, __Field, inputValue, _EnumValue and __Directive (all with double underscore) are part of the introspection system.

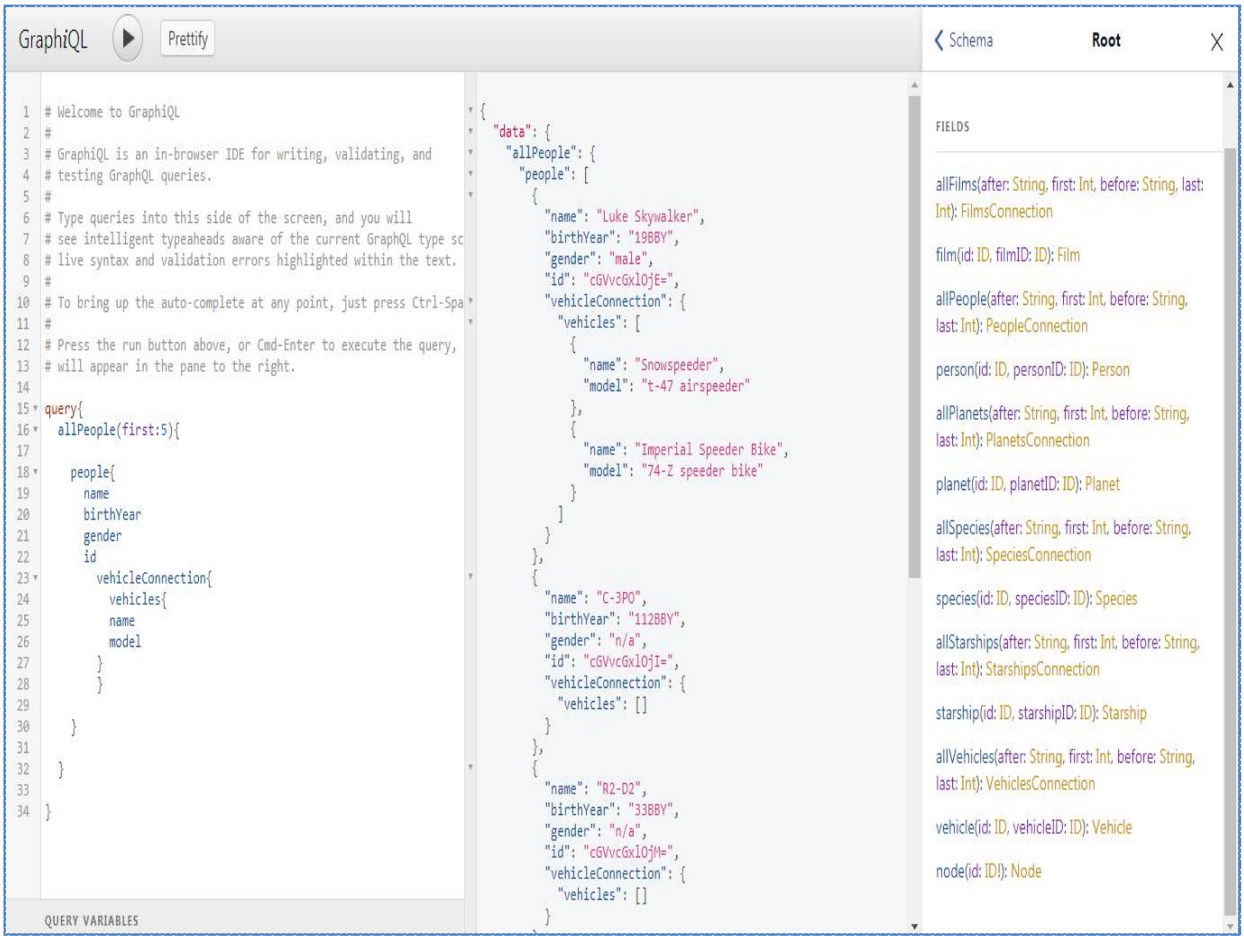

Figure 2.3: An example of GraphiQL, the left panel represents the query, the middle panel is for the response and the right panel is the structure of the GraphQL schema, adopted from swapi-to-GraphQL tool while in execution.

2.4 SOA

This section introduces an architectural style widely used for building distributed applications. This architectural style is called *Service Oriented Architecture (SOA).*

One of the fascinating aspects of software engineering is how great concepts continue, but their execution and application are regularly reinvented using current tools and practices. The rise of service based architectures in general and SOA in particular is a great example of this process [Richards2015]. To make the discussion effective, a clear understanding of the basic term *service* is needed. A service is a function that is well-defined, self-contained, and does not depend on the context or state of other services [Barry2003]. To make it clear, a service is an activity or a task always made available to its consumers.

The consumers of the service may not need to implement and maintain its functionality. Consumers use it without any concern about it and they treat is as a “black box”. It could be offered to the consumer through different transport systems, qualities and representations which can help consumers decide what is best for their needs.

The needs of the consumer usually couldn’t be accomplished by only one service. Therefore these services need to be connected by some means so that to communicate with each other. The communication can involve either simple data passing or it could also involve two or more services coordinating for some activity. From a software architectural point-of view, this is what is known as service-oriented architecture (SOA). As [Barry2003] has clarified it, SOA is basically a collection of services that needs to communicate between each other. To make it concise, SOA can be described as an architectural style that determines how applications can be built based on services which represent the components the application. The definition of SOA can be more elaborated by the following three principles:

- *Reusable components:* It is essential to decompose business applications into business components in such a way as many components as possible are general

purpose (reusable) and as few as possible are special purpose [Umer2009].

- *Web-services enablement:* The components must have well defined service interfaces that can be stored in a directory so that service consumer can query an interface directory to discover and invoke the needed service providers. Web service (WS) is the favored enabling technology at present. WS provides a widely accepted mechanism for service definition through WSDL that can be defined and discovered through a universal, description, discovery and integration (UDDI) directory by exchanging XML messages using HTTP over the Internet [Umer2010].
- *Enterprise Service Bus (ESB):* Instead of point to point communications between participants, a loosely coupled common middleware infrastructure must be used for communications, brokerage, security, directory and administration services needed throughout the enterprise. Although such an infrastructure can be provided by the existing Enterprise Application Integration (EAI) platforms, the SOA patterns strongly suggest WSenabled ESBs for SOA [Geza2017].

From the consumers' point SOA causes a great positive impact by the features and characteristics it offers when constructing applications [Albreshne2009]. For example, SOA could offer loose-coupling, service reusability and heterogeneous interoperability. Considering these benefits, a consumer can invoke a function without the need to know about the location, platform or framework of the service. This can be achieved by using certain middleware that hides all the complexities needed to complete an interaction successfully. Web Service (WS) technology is an example of SOA technology which enables construction of distributed applications. Next section discusses about web services.

2.5 Web Services

Different books provide different definitions for web services. However, I will stick to the definition given by Cerami et al. [Cerami2002] where it is defined as "any piece of software that makes itself available over the Internet and uses a standardized XML messaging system. XML is used to encode all communications to a Web service. For example, a client invokes a Web service by sending an XML message, then waits for a

corresponding XML response. Because all communication is in XML, Web services are not tied to any one operating system or programming language--Java can talk with Perl; Windows applications can talk with UNIX applications.” Therefore, web services are platform-independent and based on XML messages. The idea is to distribute services over the Internet and to make the services available for consumers. These services can be invoked, composed and implemented with any language. Moreover WS technologies enable the development of large scales of distributed systems. WS-* stack which is a set of specifications for WS, could be used to implement SOA applications. Some of the characteristics of WS technologies are:

- *XML-based:* WS technologies rely on XML as a standard for data representation and transportation. XML avoids any network, operating system or platform binding [*Point2017*].
- *Loose coupling:* There is no direct tie between a web service and its user. This in turn facilitates software system management and helps the integration of different systems which is contrary to tightly coupled system where the client and server logic are closely bound to each other, implementing a loosely coupled architecture [Papazoglou2008].
- *Ability to be synchronous or asynchronous:* The interaction style between the client and the execution of the service can be synchronous or asynchronous which later one is crucial factor to implement loosely coupled systems.
- *Supports Remote Procedure Calls (RPCs):* Web services enable clients to invoke methods and operations on remote objects using an XML-based protocol (SOAP). Service supports/implements RPC either by providing services of its own, or by translating incoming invocations into an invocation of an EJB or a .NET component [Albreshne2009**]**.
- *Supports document exchange:* XML is capable to represent data, simple and even complex documents in a generic way [Papazoglou2008].

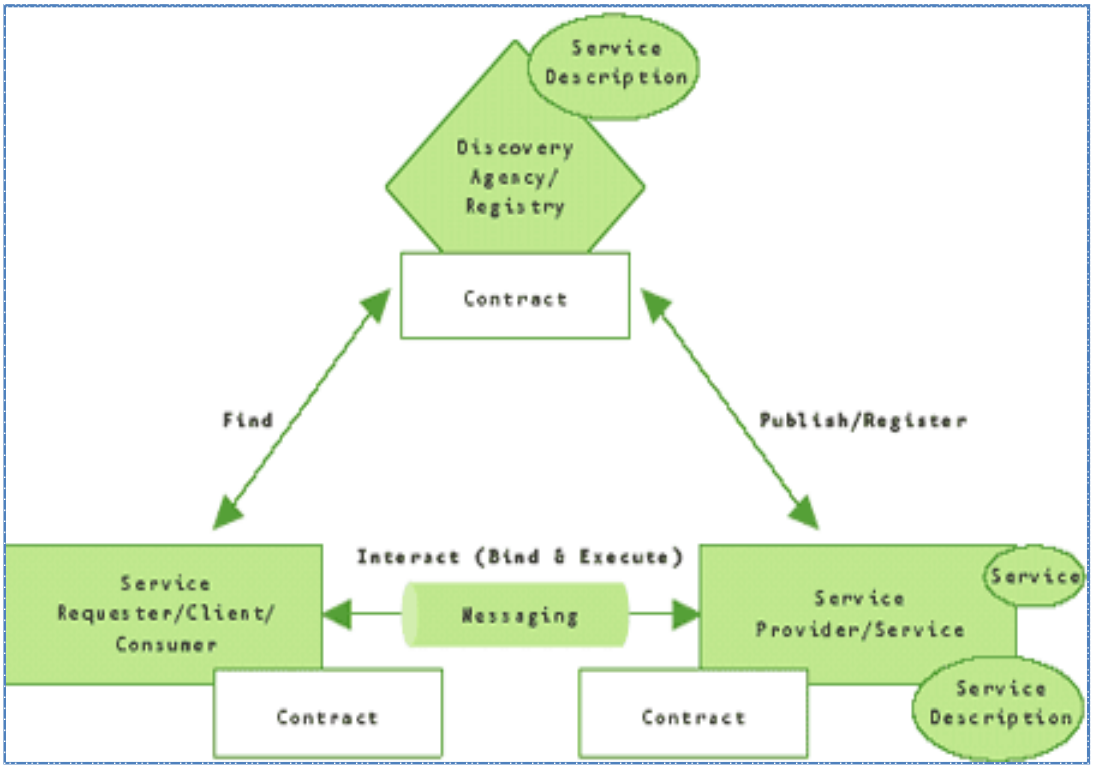

Figure 2.4: Relationships between operations and roles of web service, Adopted from [Chatterjee2004]

Some of the benefits of using WS technologies are well explained by Albreshne et al. [Albreshne2009]; reusability (the program's functionalities can be invoked by other applications), Interoperability (platform and technology independent), Standardized Protocol (wide range of choices, competition leads to increased quality and reduction in the cost, automatic discovery by the service provider and consumer.

The basic WS architecture includes WS technologies capable of exchanging messages, describing Web services and publishing and discovering WS descriptions [W3C2004]. To clarify it, the basic WS architecture models the interactions between three major roles [Point2017]:

- *Service provider:* This one is provider of the web service; builds the service and makes it available on the Internet for consumers.

- *Service requestor:* This is any consumer of the web service. The requestor invokes an existing web service by opening a network connection and sending a request.
- *Service registry:* Centralized directory of services. It is used as a central place where providers or developers can publish new services or find existing ones. It therefore serves as a centralized clearing house for companies and their services [Chatterjee2004].

The relationship between the web services roles and operations is illustrated in Figure 2.5. As it can be seen from the Figure, the web service provider publishes its web services with the discovery agency. The web service consumer looks for the desired web services using the registry of the discovery agency. Finally, the web services client invokes the web services by using the information obtained from the service discovery agency. In order to achieve this, the interfaces of a Web service's functionalities need to be described in a description language as well as a messaging protocol are needed. WSDL and SOAP are two familiar WS technologies deployed as description language and as a messaging protocol respectively [Albreshne2009]. These two will be discussed in the next sections.

2.5.1 SOAP

SOAP is a messaging protocol widely deployed by WS technologies and it also is an alternative to REST and JSON [Barry2003]. It does not define a standard Transport Protocol to carry the messages between providers and consumers and it is used on top of many transport protocols, but *HTTP* is mainly used and *Simple Mail Transfer Protocol (SMTP)* can also be used to carry SOAP messages. Indeed HTTP is efficient transport protocol in sending and receiving SOAP messages [Papazoglou2008]. HTTP in turn is famously used by web browsers to access web resources. However, other protocols such as SMTP or FTP may be also used. The components of distributed applications also can use SOAP as an option to exchange data and information over a network. SOAP can be described as architecture to exchange messages in distributed environments. It is mainly used by WS technologies to facilitate the interaction between service providers and consumers paradigm.

The responsibility of SOAP is to define how a message is formatted but not how the message is delivered. A SOAP message is encoded as *XML document* and Figure 2.5 shows its structure [IBM2017]. The document is consisted of a root element called Envelope, which can contain an optional *Header element* and a mandatory *Body* element. The Header element is used to pass application-related information to be processed by SOAP nodes along the message path. The Header provides information on authentication, encoding of data, or how a recipient of a SOAP message should process the message [Barry2003]. The Body element contains information intended for the ultimate recipient of the message. The Fault element, contained in the Body, is used for reporting errors. The XML elements in the header and the body are defined by the applications that make use of them. However, the SOAP specification imposes some constraints on their structure.

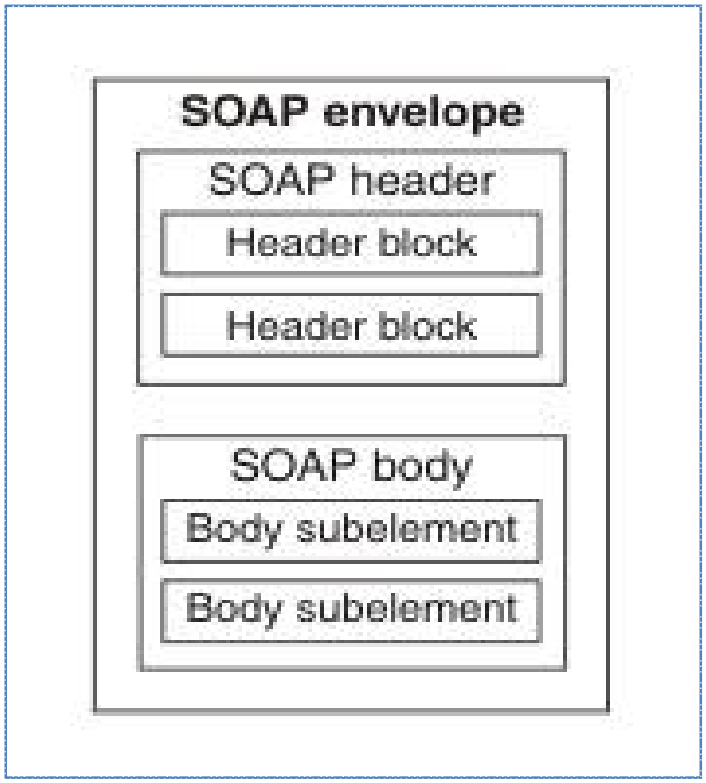

Figure 2.5: SOAP message structure, adopted from [IBM2017]

According to [Papazoglou2008], SOAP communication model is determined by its communication style and its encoding style. Accordingly, SOAP supports two possible communication styles: RPC Style and Document (message) style.

RPC style: These are used as remote objects on the client application side. Clients send their request as a method call and the method returns a response message. This information is formatted as sets of XML elements loaded into a SOAP message as shown in Listing 2.1.

```
1 <Envelope xmlns="http://www.w3.org/2001/12/soap-envelope"
2 <Header>
3 ...
4 </Header>
5 <Body>
6 <GetProductPrice>
7 <product-id>10</product-id>
8 </GetProductPrice>
9 </Body>
10 </Envelope>
```

Listing 2.1: Example of RPC style

Document (message) style: Thanks to XMLs features, SOAP supports documents exchange for any kind of XML data. The client sends the whole document to the provider instead of sending a set of arguments [Papazoglou2008]. Listing 2.2 shows an example of a SOAP document message.

```
1 <soap:Envelopexmlns:SOAP=http://www.w3.org/2001/12/soap-envelope>
2 <soap:Body>
3 <pourchaseOrderorderDate=”2017-09-20” xmlnso=http://www.amzon.com/POs>
4 <po:accountName>Eyob</po:accountName>
5 <po:accountNumber>1234</po:accountNumber>
6 <po:book>
7 <po:title>J2EE web services</po:title>
8 <po:quantity>30</po:quantity>
9 <po:price>1000</po:price>
10 </p:book>
11 </pourchaseOrder>
12 </soap:Body>
13 </soap:Envelope>
```

Listing 2.2: Example of Document style

2.5.2 WSDL

Web services need to be described in a consistent manner so that they can be published by service providers. Once described, they can be discovered by service clients and developers, and assembled in a manageable hierarchy of composite services that are orchestrated to deliver value-added service solutions and composite application assemblies [Papazoglou2008]. This is very important to develop service-based applications and business processes, which comprise service assemblies? In order to accomplish this, consumers must precisely determine the XML interface of a Web service along with other miscellaneous message details. The good thing about this is that XML Schema is verbose and this can partially help because it allows developers to describe the structure of XML messages understood by Web services. Unfortunately, XML Schema alone is not enough because it may not describe important additional details involved in communicating with a Web service such as service functional and non-functional characteristics or service policies [Fakorede2007].

As mentioned before, service description is a key to making the SOA loosely coupled and reducing the amount of required common understanding, custom programming, and integration between the service provider and the service requestor’s applications

[Papazoglou2008]. Service description is understandable by machine and can describe the operational characteristics, structure, and non-functional properties of a Web service. In order a WS to expose the functionality, format and transport protocol has to be described. Furthermore, it can describe the payload data using a type system. The service description combined with the underlying SOAP infrastructure sufficiently hides all the technical details (e.g., machine and implementation-language specific elements), from the service consumer's application and the service provider's WS.

When a WS uses a SOAP then it would require some documentation explaining the structure of SOAP messages, which protocol will be employed (E.g. HTTP, SMTP), operations exposed along with their parameters in a machine-understandable standard format, and the Internet address of the Web service in question [Papazoglou2008]. WSDL realizes the benefits of SOAP by providing a way for Web services providers and consumers of such services to work together easily [Tapang2001]. WSDL is the service representation language used to describe the details of the complete interfaces exposed by Web services and thus is the means of accessing a WS. It is by means of this service description that the service provider communicates with service consumer by providing specifications that allows invoking of a particular WS. Furthermore, neither the service consumer nor the provider should be aware of each other's technical infrastructure, programming language, or distributed object framework [Papazoglou2008].

WSDL is a format for describing the public interface of a Web service. It is a way to describe services and how they should be bound to specific network addresses. This public interface may include operational information relating to a Web service such as all publicly available operations, the XML message protocols supported by the Web service, data type information for messages, binding information about the specific transport protocol to be used, and address information for locating the Web service [W3C2001]. Although a Web service description in WSDL is written exclusively from the point of view of the Web service (or the service provider that publishes that service), WSDL is inherently intended to constrain both the service provider and the service consumer that use of the service [Papazoglou2008]. Consequently, WSDL represents a

contract between the service consumer and the service provider. This scenario is depicted in Figure 2.7. Therefore, the Web service description focuses only with information that both parties must agree upon, but not on information that is only relevant to one party (e.g. internal implementation details). Essentially, WSDL precisely describes the following aspects:

- What a service does: including the operations the service provides.
- Where it resides: its location using protocol specific address details(e.g.URL)
- How to invoke it: details of the data formats and protocols necessary to access the service's operations.

According to [Barry2003], WSDL has three parts; *Definitions*, *Operations* and *Service bindings.* Definitions are usually expressed in XML format and include both data *type* definitions and *message* definitions that will definitely use the data type definitions. These definitions are usually based upon some agreed XML vocabulary. *Operations* describe actions for the messages supported by a Web service. Operations are grouped into *port types.* Port types define a set of operations supported by the Web service. *Service bindings* connect port types to a *Port*. A port is defined by associating a network address with a *port type*. A *collection of ports* define a *service*. This binding is usually created using SOAP.

Therefore a WSDL document uses the following elements in the definition of network services [W3C2001]:

Types are container for data type definitions using some type system (such as XSD).
Message is abstract of typed definition for the data being communicated.
Operation is an abstract description of the action supported by the service.
Port Type is an abstract set of operations supported by one or more endpoints.
Binding is a concrete protocol and data format specification for a particular port type.
Port is a single endpoint defined as a combination of a binding and a network address.
Service is a collection of related endpoints.

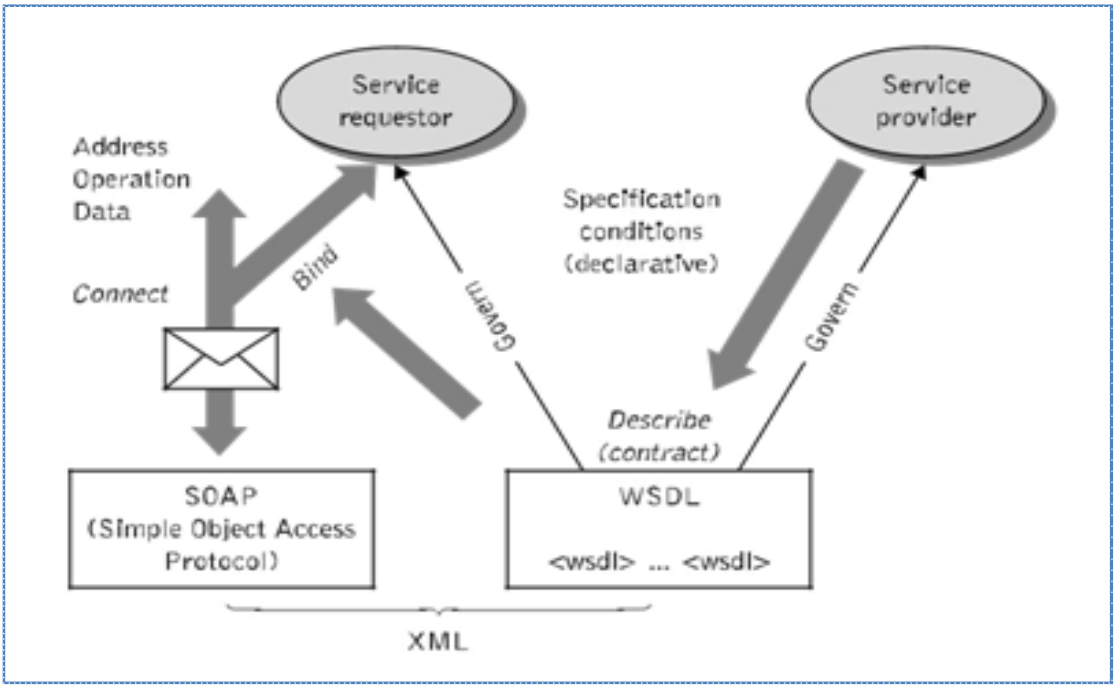

Figure 2.6: WSDL governing interaction between a service consumer and a service provider, adopted from [Papazoglou2008].

2.6 Microservices

This section introduces *Microservices Architectural style* which is gaining much attention among software developers' community.

The mainstream languages such as Java, C/C++, and Python are commonly used for the development of server-side applications. They provide abstractions to take down complexity of programs into pieces of modules. Unfortunately, these modules of a monolith are tightly coupled and they can't be executed independently [Dragoni2016]. Consequently this creates difficulty when using monoliths in distributed systems such as difficulty to evolve and maintain large-size monoliths due to their complexity, changes or updates of a particular modules leads to rebooting of the whole application which in-turn may cause considerable downtime, dependency is high within monoliths hence adding

or updating of libraries results in inconsistent systems, undergoing continuous deployment of monolithic applications is difficult due to conflicting resources' requirements , monoliths limit scalability and creating new instances of same application causes increased traffic and using monoliths causes technology lock-in in the developers point of view. The rise of service based architectures in general and *Microservice Architectural Style* in particular is to avoid these problems.

Microservices is an architecture style, in which large complex software applications are collection of loosely coupled services (also called microservice). Each microservice can be deployed independently of one another and focuses on completing one task only i.e. It does that one task really well. In all cases, that one task represents a small business capability [IBM2016]. Martin Fowler et al. [Fowler2016] describe Microservice Architectural Style 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, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies."Some of the characteristic features of microservices are the following:

- *Small and focused:* Microservices focus on a unit of work hence they are small. There are no rules to determine how small should be a microservice, but *Two-Pizza Team rule* is typically used a reference guideline. It can be stated as; if two pizzas cannot feed the team building a microservice then the microservice is too big [IBM2016]. The microservice needs to be small enough so that it will not create problem incase rewriting or maintenance of the entire microservice is needed. A microservice also needs to be treated like an application or a product with its own source code management repository and its own delivery pipeline for builds and deployment. Other than reuse, microservices can boot localized optimizations such as UI responsiveness in which in its return leads to customer satisfaction.
- *Loosely coupled*: There must be zero coordination between microservices and that is necessary for independent deployment of each one. Loose coupling enables frequent

and rapid deployments which allow the consumers to get much-needed features and capabilities [Richardson2016].

- *Language-neutral*: Microservices need to be implemented using the programming language and technology that makes sense for the specific task at hand. This is because to use the correct tool for the correct job is important [IBM2016]. The Microservices which are composed together to form a complex application need not be written with the same programming language. For example Java might be the correct language for some cases whereas Python could be best for others. Furthermore, communication between microservices is using language-neutral APIs, and typically HTTP-based resource API (such as REST). Language-neutral makes it easier to use the most existing optimal language skills.
- *Bounded context*: When saying bounded context, it means that a particular microservice does not “know” anything about underlying implementation details of other microservices around it. Microservice with correctly bounded context is self-contained and its code can be updated and understood and no need to know anything about the internal details of its peers. This is because the microservices and its peers can interact strictly through APIs and so no need of sharing data structures, database schema, or other internal representations of objects [Richardson2016].

Now it is obvious that Microservice architectural style is different from monolithic architecture and it has some benefits compared to monolithic. But why is microservices needed while SOA is still there? SOA and Microservices have one thing in common; they are Service based architectures and generally are distributed architectures [Richards2015]. This means that service components remotely interact through some sort of remote access protocol (e.g. REST, SOAP or other kinds of protocols). But there are also skeptics in the software community who dismiss microservices as nothing new but just rebranding the idea of SOA. Superficially, Microservices is similar to SOA because both approaches are service based. One way to clarify this is by considering Microservices Architecture pattern as SOA without both WS-* and ESB. In which both of them are amongst the main feature of SOA. Microservice based applications favor simpler, lightweight protocols such as REST instead of WS-*. In order to avoid using ESBs, Microservices use functionality similar to ESB which is within them

[Richards2015]. Furthermore, the Microservices Architecture pattern also rejects concepts such canonical schema which are parts of SOA. Currently many organizations including Netflix, eBay, Amazon, the UK Government Digital Service, realestate.com.au, Forward, Twitter, PayPal, Gilt, Bluemix, Sound cloud, The Guardian, and many other large-scale websites and applications have all evolved from monolithic to microservices architectures [Flowgica2017]. Figure 2.8 shows an example of Microservice architecture.

As it can be seen in Figure2.8, API Gateways are commonly associated with microservice architecture. Richardson et al. [Richardson2016] has put it this way, “An API Gateway is a server that is the single entry point into the system. It is similar to the Facade pattern from object-oriented design. The API Gateway encapsulates the internal system architecture and provides an API that is tailored to each client.”It is commonly associated with microservices. All requests from the service consumers’ first go through the API Gateway. It then routes requests to the appropriate micro (service).

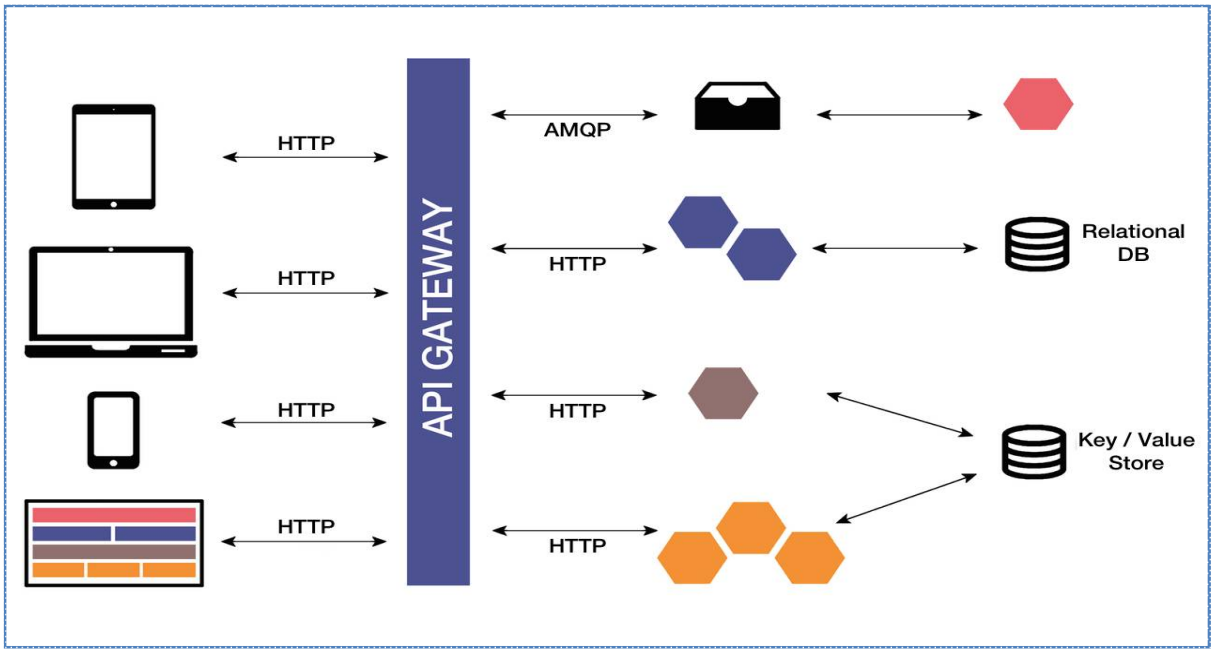

Figure 2.8: Example of Microservice Architecture, adopted from [Flowgica2017].

The API Gateway will often handle a request by invoking multiple micro (services) and aggregating the results. This gateway provides specific APIs and it reduces the number of round-trips between the service consumer and service provider which reduces network latency and it also simplifies the service consumer's code.

Typically an API gateway is a piece of software which will provide some or all of the following security and management features [Chauhan2017]:

- API creation : sometimes even offering visual editors
- API management : API lifecycle - draft, publish, upgrade etc
- Portal features : allowing users to discover and use your APIs
- Security : Authentication & authorization (Threat protection)
- Protocol transformation, routing & orchestration
- Analytics & monitoring: who is using your APIs, when and how?
- Contract & SLA management

Using API Gateways also brings some concerns [Richards2015]:

- Needs to be highly available component that must be developed, deployed, and managed.
- There is also a risk that the API Gateway becomes a development bottleneck. Developers must update the API Gateway in order to expose each microservice's endpoints.

Furthermore API Gateways have similar functionality with Enterprise Service Buses (ESBs) and also complement each other very well [Oracle2017]. API Gateways and ESBs typically both perform the following similar tasks:

- Protocol mediation
- Message routing and transformation+
- Service composition
- Message processing

Lastly, service based architectures including SOA and Microservices have introduced significant improvements; however they are complex compared to monolithic. The reason for the complexity is because they involve many considerations including service contracts, availability, security, and transactions (to name a few) [Richards2015]. However, with added complexity come additional characteristics and capabilities that will make the development teams more productive. The point to be underlined here is that moving to service based architectures shouldn’t be a must unless you are ready and willing to address the many issues facing distributed computing.

CHAPTER 3
RELATED WORKS

This chapter discovers some of the related works already done in the area of building and transforming APIs. Several tools or frameworks have been developed that help in transforming or wrapping one type of API to another type of API as well as in building APIs from executables. Here we look for Any2API framework which is used to build APIs from arbitrary executables, and also some tools that wrap REST API to GraphQL will be discussed.

3.1 ANY2API Framework

As explained in chapter two, APIs are versatile in integrating and orchestrating different applications and application components. It also enables systematic development and reliable operations of distributed applications, mash-up applications, and mobile applications. Although common protocols (e.g. HTTP) can be used to easily orchestrate APIs, the technical integration with different artifacts and heterogeneous management systems is a very error-prone, time-consuming and challenging. Reusable artifacts could be scripts such as Chef Cookbooks, Juju charms from UNIX and also templates like Docker container images [Wettinger2015].These are shared and reused by open-source communities in conjunction with provider supplied services. Therefore it is important to ease the invocation of these different artifacts, technologies, and service providers in a technically uniform manner.

Several frameworks based on different programming languages and technologies are available to develop and create APIs. However, most of these development frameworks allow individual API to be implemented manually [Wettinger2015]. This may not be feasible or could be even impossible for some individual development of APIs. This is due to scaling issues (e.g., creating APIs for a huge amount of individual executables) or missing expertise, meaning the person, who needs to utilize certain functionality, is not able to develop a corresponding API. Moreover these artifacts are executables and many of these frameworks require a central middleware (e.g. service bus) so that to utilize them through an API [Wettinger 2014].

Unfortunately depending on a central middleware has the following drawback [Wettinger2015]:

- The individual artifacts are not packaged with their API to be utilized at runtime,thus they are not self-contained.
- The central middleware component results in additional costs and maintenance effort.
- When new kind of executable comes in, the central middleware has to be adapted, extended and redeployed. Accordingly this leads to potential risks such as downtime, functional failures, and unintended side effects.

ANY2API is a generic approach and it avoids the three drawbacks mentioned above. It automatically generates API implementations (APIfication) for arbitrary executables such as scripts and compiled programs, which are not natively exposed as APIs. According to Pepple et al. [Pepple2011], APIs can be utilized either as provider-hosted APIs or as self-hosted APIs. To clarify it, the provider hosted APIs are offered by Cloud providers to provision virtual servers, storage, and other resources whereas the self-hosted APIs are offered by open-source Cloud management platforms such as Open Stack. As it has been clarified in [Jojow2017], ANY2API can generate self-hosted API implementations by transforming existing individual executables. Furthermore, it broadens the potential variety of tools and artifacts because their implementation-specific differences are completely hidden by using the generated API implementations. The generated API implementations would simplify the orchestration and integration of different kinds of artifacts with existing provider-hosted APIs. As a result of this, full deployment automation can be achieved by integrating and orchestrating provider-hosted and self-hosted APIs by hiding the specific details (abstraction) of different kinds of executables.

The architectural design of ANY2API is depicted in Figure3.1. The upper part of the diagram shows all user interactions which are performed using a corresponding interface. Although the command-line interface is the most powerful option available, a web-based user interface is also on the plan to further simplify the usage of the framework. All the interfaces use the core and utility modules to interact with available scanners, invokers, and generators [Jojow2017] .Whereas the lower part of the

architecture diagram represents the logical workflow of the framework.

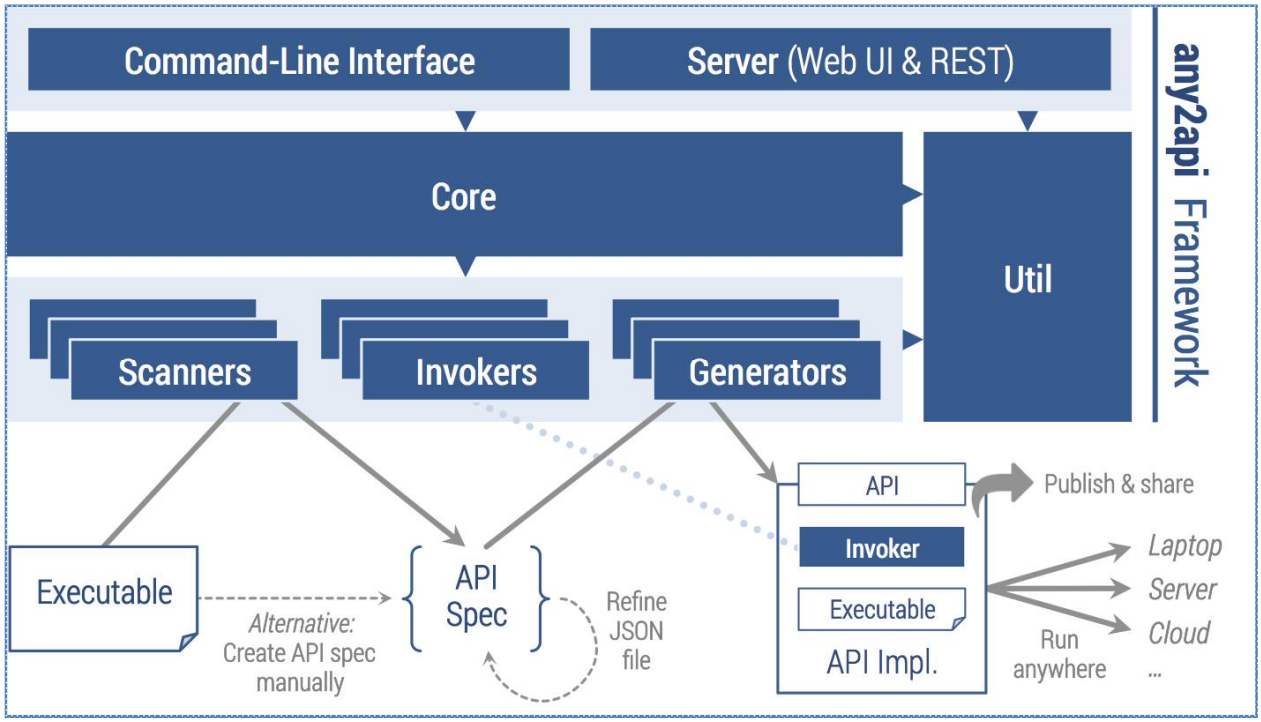

Figure3.1: Architecture of ANY2API adopted from [jojow2017]

The flow chart in Figure3.2 illustrates the ANY2API APIfication process [Wettinger2015]. It shows the individual steps undergone to create the API implementation in automated manner. An executable (e.g., Juju Charm) including its metadata, targeted by the APIfication method, is selected (step 1). At this step available invokers are checked from the invoker registry and the corresponding invoker (Juju Charm invoker) capable of running the given type of executable is selected (indicated as A). Note that each invoker supports at least one executable type. The interface type (e.g., RESTful API) and the API implementation type (e.g., Node.js or Java) are then selected using corresponding generator registry from generator registry (indicated as B). For example a generator that provides HTTP+REST as interface type and Node.js as implementation type can be selected .The executable with it metadata is analyzed by a corresponding scanner module selected from the scanner registry (e.g., Juju scanner) to discover the input and output parameters from the executable (indicated as C). Furthermore, these input and

output parameters for the generated API can be more refined (indicated as D) but this is optional and only done if the scan could not discover all the parameters. This produces an API I/O specification (*API spec*); that contains the input and output parameter names, their data types, and the mapping information to properly map between API parameters to the executable parameters at runtime. The API spec is then rendered to the respective format (e.g. JSON file).

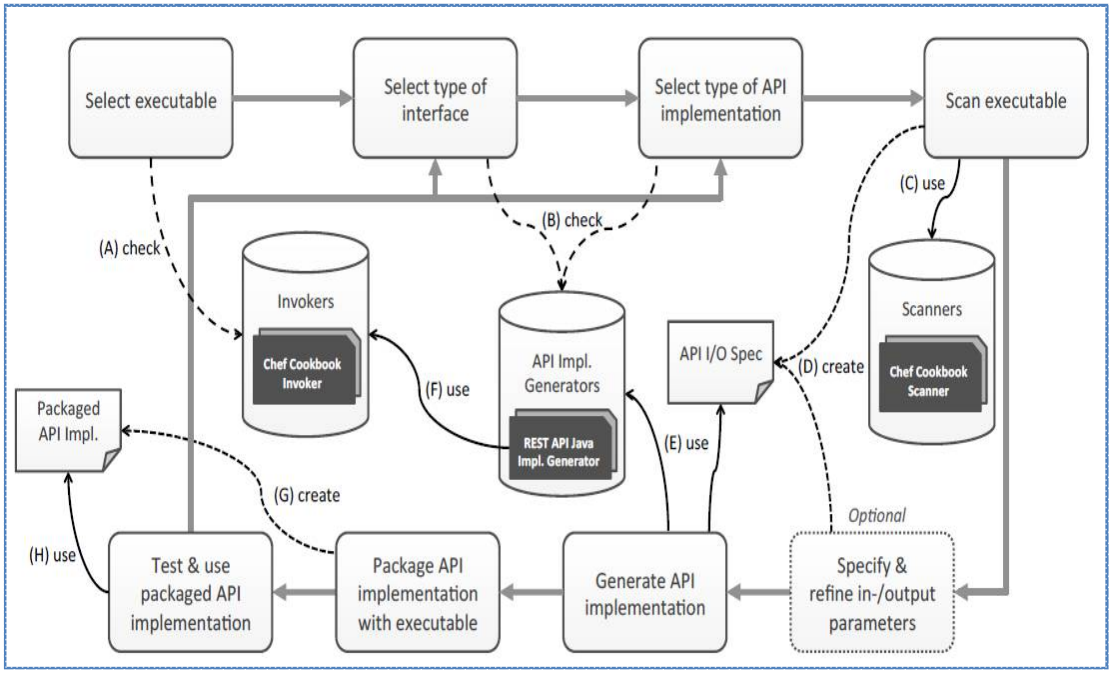

Figure 3.2: Flow chart of the ANY2API APIfication process adopted from [Wettinger2015].

Consequently, the API spec tells the corresponding generator to create a proper API implementation (indicated as E). The corresponding invoker, given by the invoker registry to run the executable, is provided by the invoker (indicated as F). The generator module from generator registry (e.g., REST API generator) receives the API spec and then builds a packaged, self-contained API implementation (indicated as G) and it can be tested and used accordingly (H).

The package includes the selected executable(s), the generated API endpoint and the selected invokers. Each API implementation can be packaged as node packaging

module (npm) or docker container. The package can run anywhere (laptop, server or cloud). Furthermore the generated API implementation is used to enable the invocation of the corresponding executable through a well-defined interface, independent of any underlying technology stack. The invocation of the corresponding executable can be done either in local environment or in remote environments (using SSH or PowerShell).The later one helps to decouple the environment of an API implementation instance from the environment of the actual executable that is exposed by the API. Now the generated API implementation can enable the invocation of the corresponding executable through a well-defined interface, independent from the underlying technology stack. An example of generated API implementation is shown in Figure 3.3.

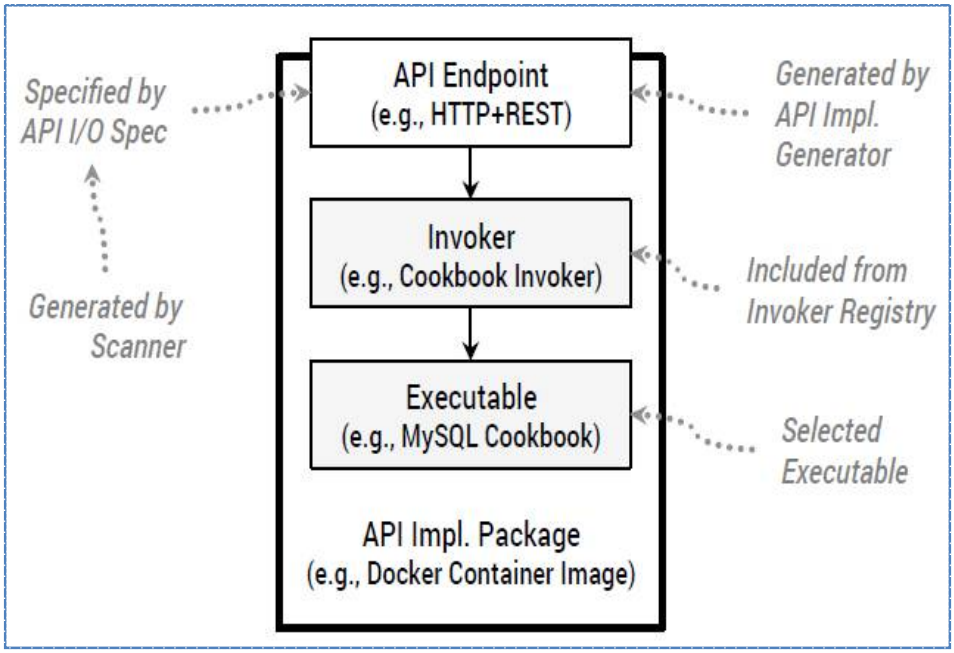

Figure3.3: Sample generated API implementation adopted from [Wettinger2015]

These are the terms commonly associated with ANY2API framework [Jojow2017]:

- *Executable:* typically expects inputs and produces outputs through different channels. It can be a code snippet, a script, a binary or a configuration definition.
- *Scanner:* Specialized module that receives and scans an executable and then outputs API spec. Scanner registry keeps track of the corresponding scanner.
- *API Spec:* Describes the interface for one or multiple executables

- *Generator:* This specialized module receives an API spec and then generates and packages an API implementation for the executables described in the given API spec. The Generator registry keeps track of available generators.
- *API Implementation:* This is portable and self-contained package containing the API spec, all executables, all required invokers (to invoke the executables) and the generated implementation of the API endpoint.
- *Invoker:* This module is packaged as part of the API implementation and used to invoke executables at runtime.

The evolution of ANY2API leads into an open ecosystem composed of API bricks (generic, reusable and configurable), including reusable wrappers, adapters and plug-ins. This will avoid usage of basic API development frameworks to build specific kinds of APIs. Furthermore, the API bricks; the wrappers, adapters and plug-ins will lead to API diversity which is very important: as Johannes et al. [Johannes2016] has put it “API diversity is the key because there is no ‘one-fits-all’ kind of API. In certain cases, REST is a good choice, but sometimes messaging, RPC or streaming APIs work much better.”The ANY2API framework is used for the concept development of this thesis as discussed in chapter four.

3.2 GraphQL Approachs

In the previous chapter we discussed that GraphQL has some benefits compared to REST APIs. GraphQL solves many of the shortcomings of REST APIs. But now we have a challenge; what approach can we use to deploy GraphQL? Depending upon the existing practices and resources, there are three approaches to deploy GraphQL [GQL2017]:

A. GraphQL server connected to a database

This approach is very common if the existing system doesn’t have constraints to work on it. In this method the existing API files are transformed into GraphQL. Existing endpoints and their attributes as well as their association to other endpoints need to be constructed into GraphQL schema models. Furthermore the existing endpoint methods

are also converted into GraphQL resolvers and mutations. The GraphQL server is also connected to the database. Therefore detailed understanding of the existing system is required and as it can be imagined, this method may demand plenty of time and efforts if the existing API project is big. Hence the API will be completely moved into GraphQL. Requests arrived to the GraphQL server is fetched from the database. Figure3.4 illustrates this approach.

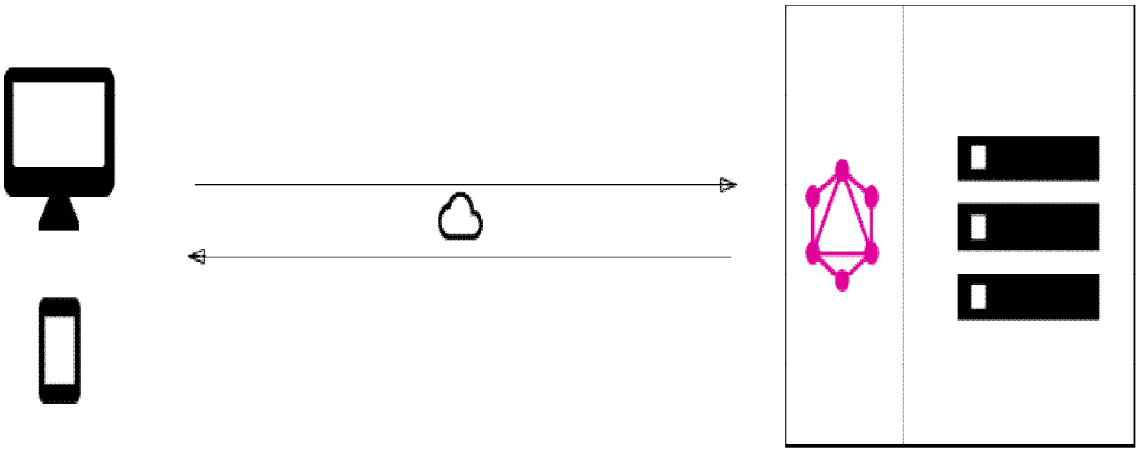

Figure 3.4: GraphQL connected to database taken from [Graphcool2017].

B. Wrapping: GraphQL Serves as Mediator

In this approach, existing systems are integrated behind a single and coherent GraphQL server. This is good for companies with legacy infrastructure and many different existing APIs. They could have been developed over years and complete transformation or maintenance is not easy. This is also good choice for those companies who want to practice new technologies without impacting their existing system systems [Kristsov2016]. Therefore GraphQL in particular is used in unifying these existing systems and hide their complexity somewhere behind. For example GraphQL can be built on top of existing REST API. The GraphQL will be responsible in fetching data and providing the response to the client. The GraphQL doesn't care about the details of the data stores behind the existing. Examples of this approach are discussed in section 3.3.

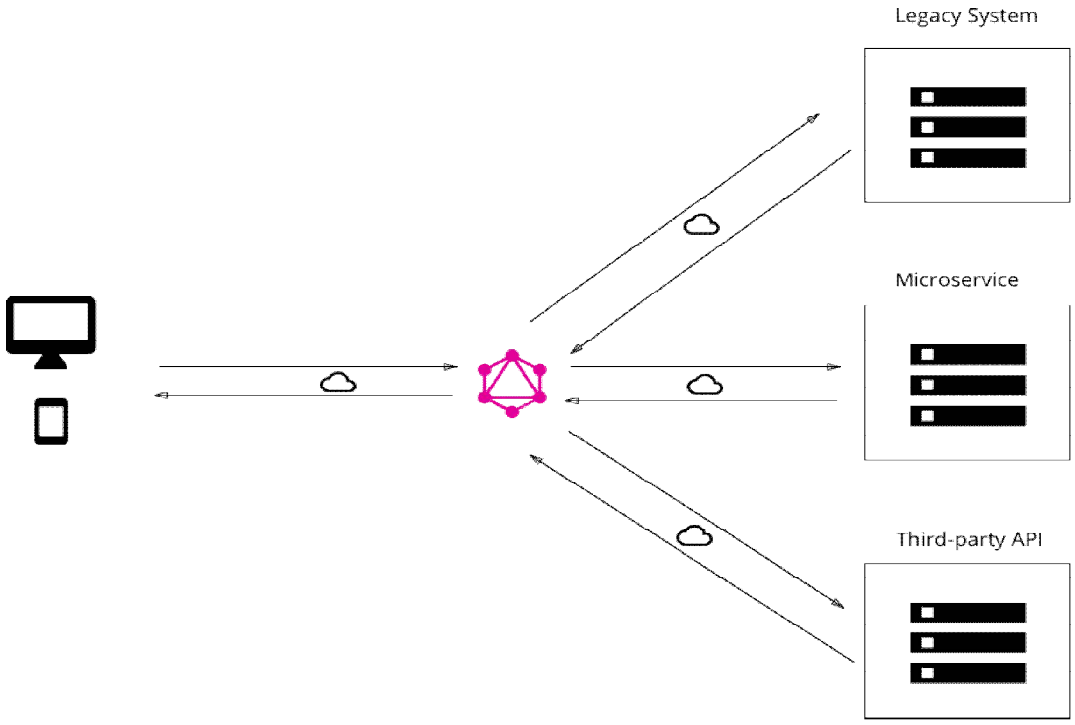

Figure 3.5: GraphQL as a mediator taken from [Graphcool2017].

C. Hybrid

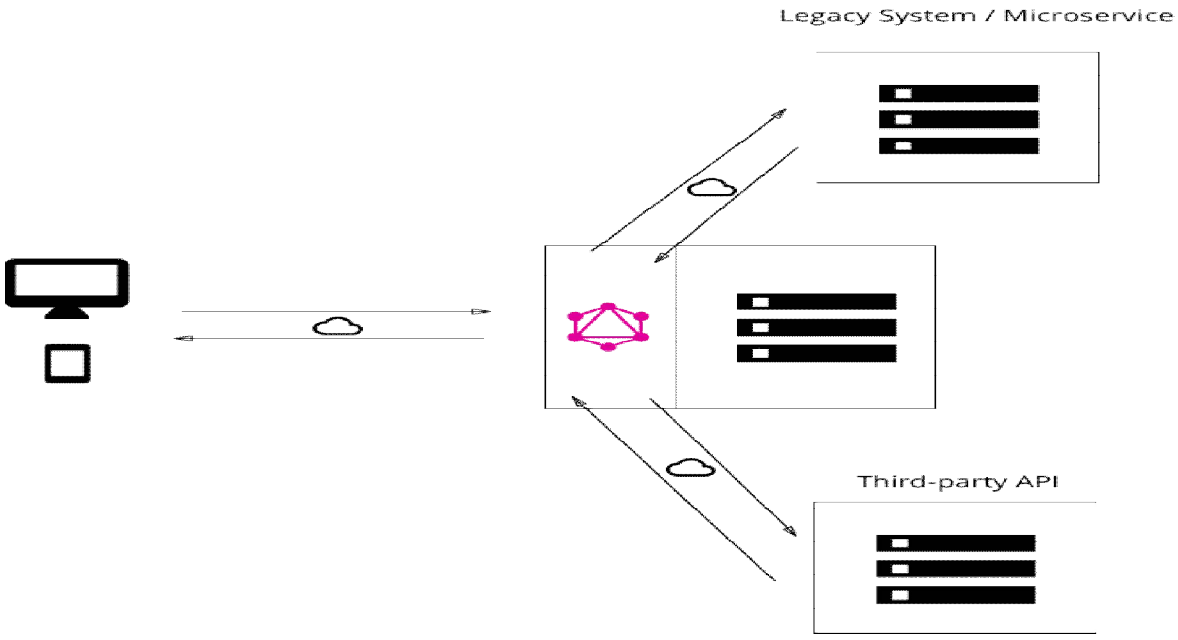

Figure 3.6: GraphQL in a hybrid system taken from [Graphcool2017]

It is possible to combine the two approaches where a GraphQL API connected to database can be made to talk with other existing APIs. Requests from clients are resolved either from the database or from the APIs. The hybrid approach is depicted in Figure 3.6.

3.3 REST to GraphQL Tools

The schema of the GraphQL is built based on the existing API providers and it wraps calls to them. As Steve et al. [Steve2016] articulated it, although it is possible to build the schema on the client side, it is better to move it to the server side for performance reasons. There are tools that wrap REST APIs into GraphQL but many of them deploy similar approaches. Depending upon their familiarity amongst the web developers' community, three REST to GraphQL wrappers are selected for this thesis. The tools are GraphQL-RESTWrapper [Alon2016] Swapi–to-GraphQL [Stubailo2016] and Swagger-to-GraphQL [Yarax2016].

The general architecture of the REST to GraphQL wrapping tools is similar as described in Figure 3.7. Their main difference is on how they generate the GraphQL schema. All the three tools are implemented using Node.js and express to construct the GraphQL server. The GraphiQL is used as the client application in the service consumer side. As Figure 3.7 shows the service consumer (GraphiQL or client application) sends its request query to the GraphQL server (1, 2, and 3). Then the GraphQL server receives the query from the client (4). To process this query request, a GraphQL schema has to be generated at runtime using one of the wrapping tools. Once the schema is generated, the graphql creates an instance of it and the GraphQL server calls the instance of the generated schema (5,6). After this, the server builds the REST calls using the query and the resolvers of the fields of the schema. Note that, Base_URL of the service provider is used in constructing the resolvers. Then the server sends the REST calls (e.g. using HTTP) to the service provider (7). The service provider processes the REST calls and sends the response back to the GraphQL server (8). After this, the GraphQL server receives the response from the service provider and customizes it according to the content of the query received against the generated

schema (9). Lastly, the customized response is sent back to the client (10). The response received by the client could be either a data or an error message (11). For instance an error message can be received if the client requested non-existing fields, but the client is well informed (13) about the error. Whereas the data received has similar pattern with the requested data and the client can match them easily (12).

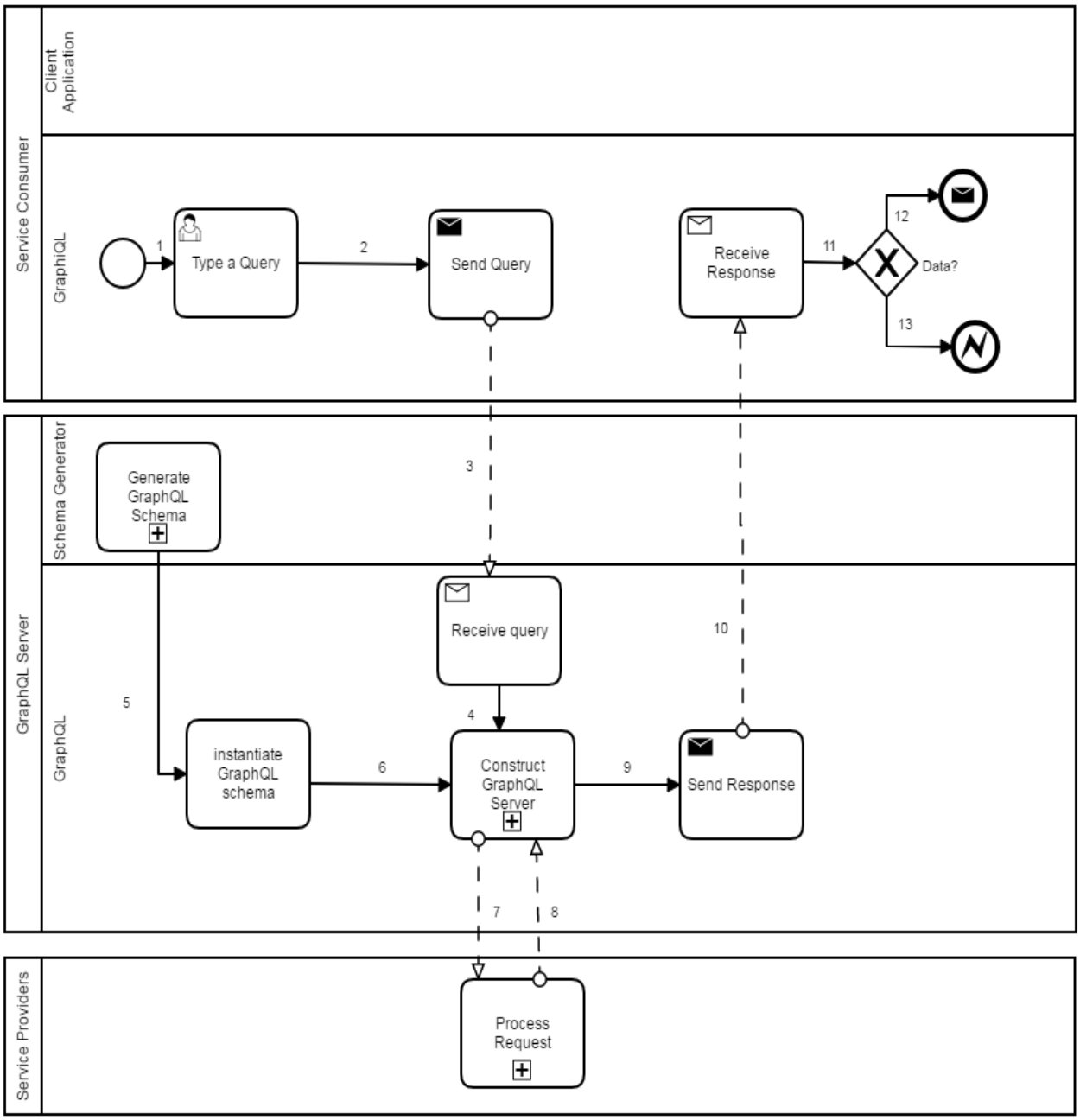

Figure 3.7: GraphQL as a wrapper: General architecture

3.3.1 GraphQL-REST-Wrapper

GraphQL-REST wrapper allows use GraphQL on top of an existing REST API very easily using express-graphql[Alon2016]. The tool will fetch the REST API response, constructs a GraphQL schema from it and then expose the data from a GraphQL server. It either builds a schema by parsing the service provider's response (usually JSON) or uses a given schema. When the client sends an HTTP request to the route with GraphQL query, GraphQL server will fetch the response from the REST API and sends only the data requested.

Figure 3.8 shows the architecture on how the GraphQL-Rest-Wrapper tool generates works. Normally the tool starts when the GraphQL server gets a query from the client (1, 2). The GraphQL server application requires an instance of a GraphQL schema in order to process the query. Therefore, the server calls the GraphQL-REST schema generator so that to get the schema (3, 4). Now there are two choices and it checks whether a GraphQL schema is given or not (5).

GraphQL schema is given*:* If GraphQL schema is already available then the graphql creates an instance of it and the server processes the query according to the given schema definition (15, 17).

GraphQL schema is not given*:* Otherwise the tool has to generate new GraphQL schema from a given REST response which is in JSON format. The REST wrapper is instantiated and the Abstract Syntax Tree (AST) of the REST response is parsed (6, 7). While parsing the AST explorer, the data types of the AST will be mapped to the GraphQL data types and this continues until the end of the explorer (8, 9). The type token of the AST will be built once parsing the AST explorer is done. After that, the GraphQL schema is built from the type token map AST. The GraphQL schema is saved to a file (for next time) and an instance of the schema is created by graphql (12, 14 and 16). Finally, the GraphQL server application receives the instance of the schema and the query is processed according to the definition of the generated GraphQL schema (17,18).

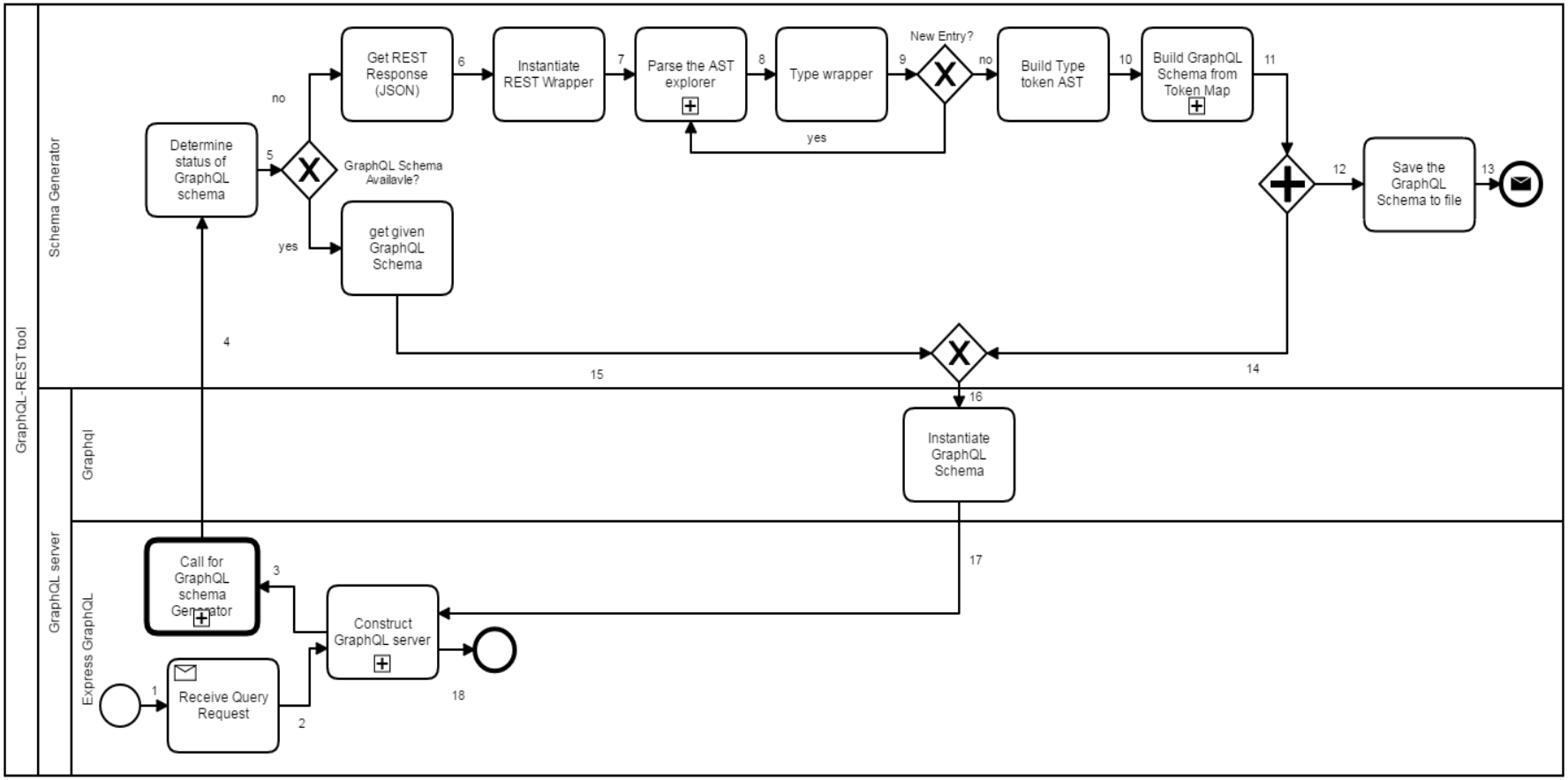

Figure 3.8: Building GraphQL Schema by GraphQL-REST wrapper

Table3.1 illustrates the advantages and disadvantages of using REST to GraphQL tool.

Advantages and Disadvantages of GraphQL-REST tool	
Advantages	**Disadvantages**
▪ Partial auto-generation of GraphQL schema ▪ It allows user to get exact data requested	▪ Content Type is restricted to JSON format ▪ Tightly coupled system ▪ Batching, caching and pagination not supported ▪ It can crash when used for big projects and ▪ It runs on the client side

Table 3.1: Advantages and disadvantages of using GraphQL-REST wrapper

3.3.2 Swapi-to-GraphQL-Wrapper

A description file (known as Swapi) for the service provider's endpoints is already prepared and the GraphQL schema is generated based up on the file definitions. Particularly, the file holds the list of schemas (in JSON format) of the provider's endpoints (in this case the star wars film).

Figure 3.9 illustrates how the Swapi–Rest-GraphQL tool generates the GraphQL schema. Similar to the GraphQL-REST tool, the process starts when the GraphQL server gets a query from the client (1, 2).The GraphQL server application requires an instance of a GraphQL schema in order to process the query. Therefore, the server calls the swapi–to-GraphQL schema generator so that to get the needed schema (3).After this, the schema generator gets the swapi file (list of schemas) and it loads the file to check its validity (4, 5 and 6). If the swapi file is not valid then the process terminates with an error message. Otherwise the GraphQL schema's root is created and the list of schemas with in swapi file are parsed one after another (7, 9). A GraphQL query is created from the title of each schema and the properties of each schema are

also parsed (13). Query fields with their resolvers are constructed from each property of the schema (14, 15).The parsing of the properties stops when every property is assessed and a GraphQL query with its corresponding fields is created (12, 16 and 17). If the data type of the property is primitive then data type of the property will be converted to its equivalent GraphQL data type and it will be field of the already created query field (through line 11). Otherwise the data type of the property is array or object type and another query field willbe created. Its contents are repeatedly parsed to create its fields. The parsing ends when all the outcomes are primitive data types hence a GraphQL query with its subsequent fields will be created. All necessary resolvers with their arguments will be also constructed for each field. However, parsing of the schemas continues until each and every schema is parsed (19). Therefore all GraphQL queries with their corresponding fields will be created. These Queries intern will act as the fields of the parent root query (8, 20). The root query will get all the queries as its fields and then a GraphQL schema is built (21). Finally an instance of the GraphQL schema is created and GraphQL server will continue to process the query according to the definition of the generated schema (21, 23 and 24). Table 3.2 illustrates the advantages and disadvantages of using GraphQL-REST tool.

Advantages and Disadvantages of swapi-to-GraphQL tool	
Advantages	**Disadvantages**
▪ Partial auto-generation of GraphQL schema with its queries ▪ Enables users to get specific data they requested.	▪ The tool is not generic and exclusively considers the star wars film as the service provider. ▪ Content Type and the swapi file must be in JSON format ▪ Tightly coupled system and generates run time code ▪ Mutations are not considered ▪ Security is not considered

Table 3.2: Advantages and disadvantages of using Swapi-to-GraphQL wrapper

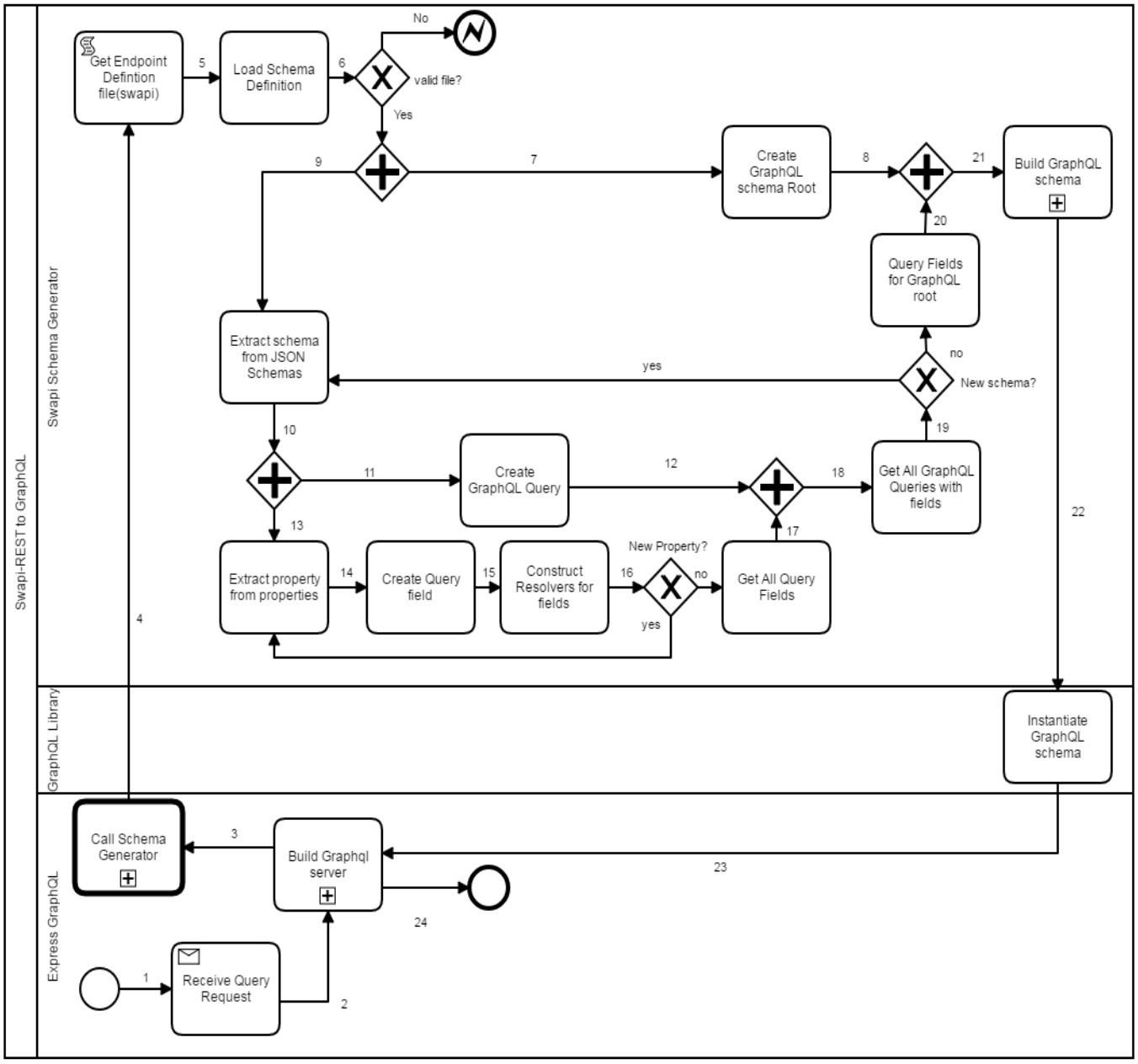

Figure 3.9: Generating GraphQL Schema from Swapi-to-GraphQL Wrapper

3.3.3 Swagger-to-GraphQL

As swapi is required for the Swapi-to-GraphQL tool to operate, the Swagger-to-GraphQL tool also requires an API definition which is swagger file in JSON format.

Similar to the other two tools, the Swagger to GraphQL tool executes once a query is received from the client (1, 2). A schema is needed to build the GraphQL server hence the schema generator is called (3). The given swagger schema is loaded and it is checked whether it is valid or not (5, 6). If the swagger file is not valid then the execution will terminate with an error message. Otherwise, a GraphQL Schema root will be created and the schemas of the valid swagger file will be parsed (7, 8). The endpoints will be created from each path and its methods (9).In addition to that each endpoint is also consisted of corresponding parameters and requests (created from the Base URL of the service provider).

The endpoints will be repeatedly (loop) parsed and the method of each endpoint will be checked (11). If the method is GET then a GraphQL query will be created and the corresponding parameters of the endpoint will become the fields of the Query (12, 14 , 15 and 16). Otherwise the method is used to manipulate the service (PUT, UPDATE and DELETE) hence a mutation with its fields and resolvers is created (13, 14, 15 and 16). The parameters of the corresponding endpoint will be the fields of the mutation. The resolver of each field will be created from the parameters and the requests of the corresponding endpoint. Therefore a query or mutation of each endpoint with its respective fields is created.

The parsing stops once each and every endpoint is parsed and all queries or mutations with their corresponding fields will be generated (17). Consequently all these mutations and queries will be the fields of the parent GraphQL query or root (18, 19). The GraphQL schema is built and an instance of the schema will be executed by the GraphQL server (20, 21). Finally, the GraphQL server executes the query according to the generated schema (22, 23).
Table 3.3 describes the advantages and the disadvantages of using swagger to GraphQL tool.

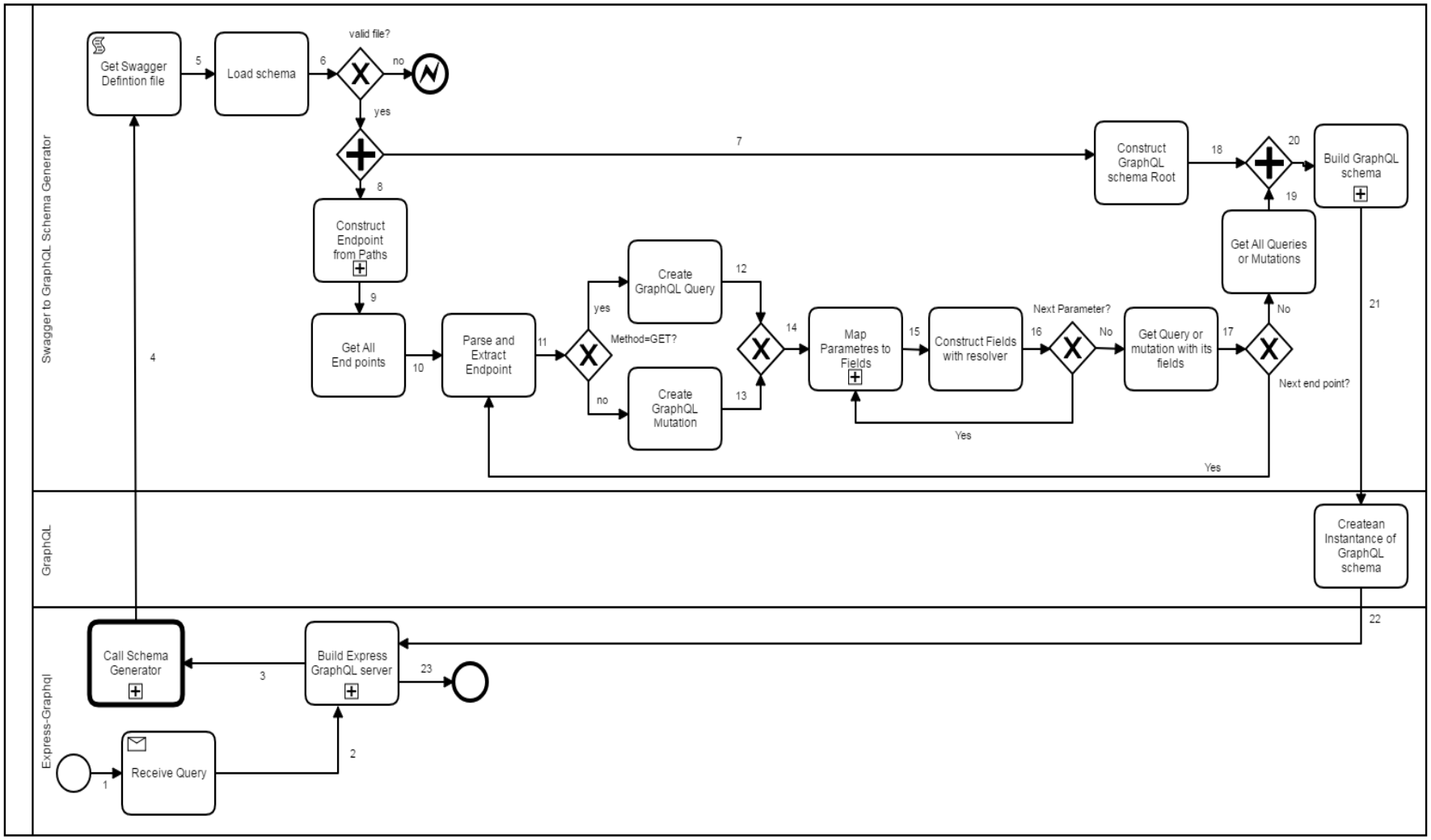

Figure 3.10: Building GraphQL Schema from Swagger file

Advantages and Disadvantages of swagger to GraphQL tool	
Advantages	**Disadvantages**
▪ Auto generation of GraphQL schema with its queries and mutations ▪ Security measures are available ▪ Enables users to get specific data they requested.	▪ Content Type and the swagger file must be in JSON format ▪ Tightly coupled system and generates run time code ▪ Batching, caching and pagination not available

Table 3.3: Advantages and disadvantages of using Swagger-to-GraphQL wrapper

To conclude our discussion with the wrapping tools; the three tools have many similarities and typically they all depend on some references from the service provider. This is true unless GraphQL schema of a corresponding service provider is already available. This reference is swapi file in GraphQL-REST, swagger in swagger to GraphQL and the REST response in GraphQL-REST. Moreover all the three tools depend on the GraphiQL (See chapter 2) hence they don't have production ready client application. Consequently many features that affect API usability are lacking. However, the tools are capable of doing the basic objective, i.e. they wrap REST calls to get the needed responses.

CHAPTER 4
CONCEPT

In the previous two chapters, the fundamental concepts as well as solutions and approaches towards creating and transforming APIs have been introduced and discussed. They are the basis for this thesis .The purpose of this chapter is to explain theoretically the concept that has been developed in this research so that the reader gets a conceptual understanding for the recommended system.

Basically the concept is developed based upon the related works discussed in chapter three. Therefore in the first step of the concept development, the wrapping tools are analyzed according to some criteria. The second step is to propose the system design taking into consideration the outcome of the first step. Thus, this chapter is divided into two main sections: Analysis of closely related works and proposed system design. The first section has two subsections; the criteria used to analyze the tools are discussed. In the next subsection the GraphQL-REST wrapping tools are compared and analyzed. The second section by itself is also consisted of three subsections; the proposed abstract architectural design, the service consumer architectural design view and service provider architectural design view will be elaborated and discussed.

4.1 Requirements

This section explains about an important stage of software development and that is known as *Requirements* of the research.

Chapter three elaborated the three wrapping tools associated with REST and GraphQL with their respective flow charts. Generally the tools have similar structure and their major difference is on the approach they deploy in generating the GraphQL schema as shown in Figure 4.1. The Schema Generator acts as a bridge between the service provider and the service consumer. It builds the GraphQL schema and then customizes

responses according to the schema structure. If these tools are capable of transforming REST API to GraphQL, then why don't we just deploy one of them?

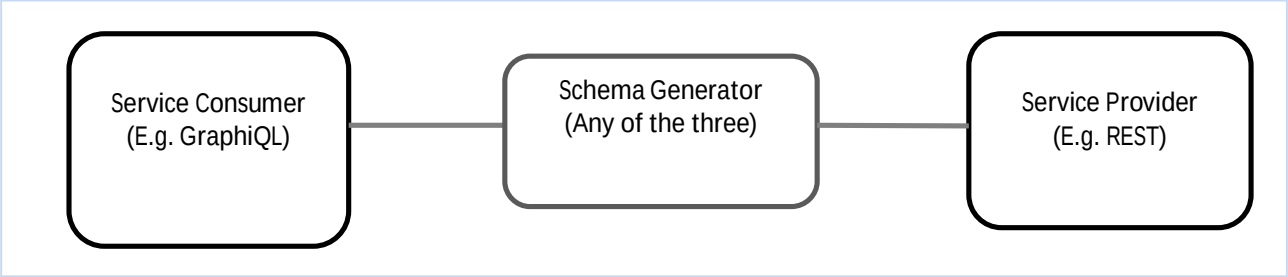

Figure 4.1: Schema Generator as a bridge

Even though the tools can address at least the basic problem of this research, they have major drawbacks associated with performance. Therefore, the process of identifying the drawbacks of the tools and looking a solution for them is the foundation for the concept development.

A. Structural Problems

The tools are built based upon simple structures; practically there is no any clear distinction between components. It is true that simple structures (similar to monolithic structures) are simple to deploy, simple to test and simple to scale horizontally [Kharenko2015]. Regardless of these advantages, simple structure leads to some critical problems that have to be addressed.

Some of the challenges of deploying the tools associated with their structures are [Chauhan2017]:

- Run time execution: This is the major problem with the three tools because they all generate a huge unnecessary run time code. As can be seen from the flow charts of the tools, they all generate unnecessary huge code during the process of schema generation.
- The size of the application can slow down the start-up time.
- The entire system must be redeployed on each update and continuous deployment is difficult.

- The system is *tightly coupled* and any change usually impacts the whole system and could lead to extensive manual testing. It can also have severe problem during system maintenance. Moreover there is difficulty of reusing components.
- The system could have scaling difficulty if different modules have conflicting resource requirements.
- Reliability problems: Bug in any module (e.g. memory leak) can potentially bring down the entire process. Furthermore, the bug will impact the availability of the entire system since all instances of the application are identical.
- The structure could become a barrier to adopting new technologies. Since changes in frameworks or languages will affect an entire system and it is extremely expensive in both time and cost.

B. Limited Capability

Basically the structure of tools could have an impact on the capability of the tools. The tools are not generic because the options they cope with are very limited.

- *Client Application:* The structure of the tools is highly dependent on the GraphiQL client application. Although GraphiQL has some good features that help to develop new abstractions and help implement common functionality on the client-side(chapter two), it is purely used in the testing and development stages [GQLorg2017].What about other service consumer applications? Or other services?
- *Content Type:* GraphQL is transport agnostic but HTTP is commonly used for client server protocol because of its ubiquity [GQLorg2017]. Most modern web frameworks use a pipeline model where requests are passed through a stack of middleware (filters/plug-in). The request can be inspected, transformed, modified, or terminated with a response as it flows through the pipeline. HTTP is commonly associated with REST, which uses resources as its core concept. GraphQL server operates on a single URL/endpoint (usually /graphql) and all GraphQL requests for a given service should be directed at this endpoint. The content type of the structure is associated with GraphiQL which means only application/graphql or application/JSON content types are supported. What about other content types like application/XML?

4.2 Proposed Solution

The major shortcoming of the tools lies on the structural problem. Therefore the first step is to propose a structure that avoids the above mentioned drawbacks. Moreover, the proposed solution should be generic that can anticipate for so many options. As already discussed in chapter two, service based architectures plays a big role here and can avoid majority of the challenges. This means all the tasks of the components should be transformed into services. In the next sections, service based architecture will be elaborated.

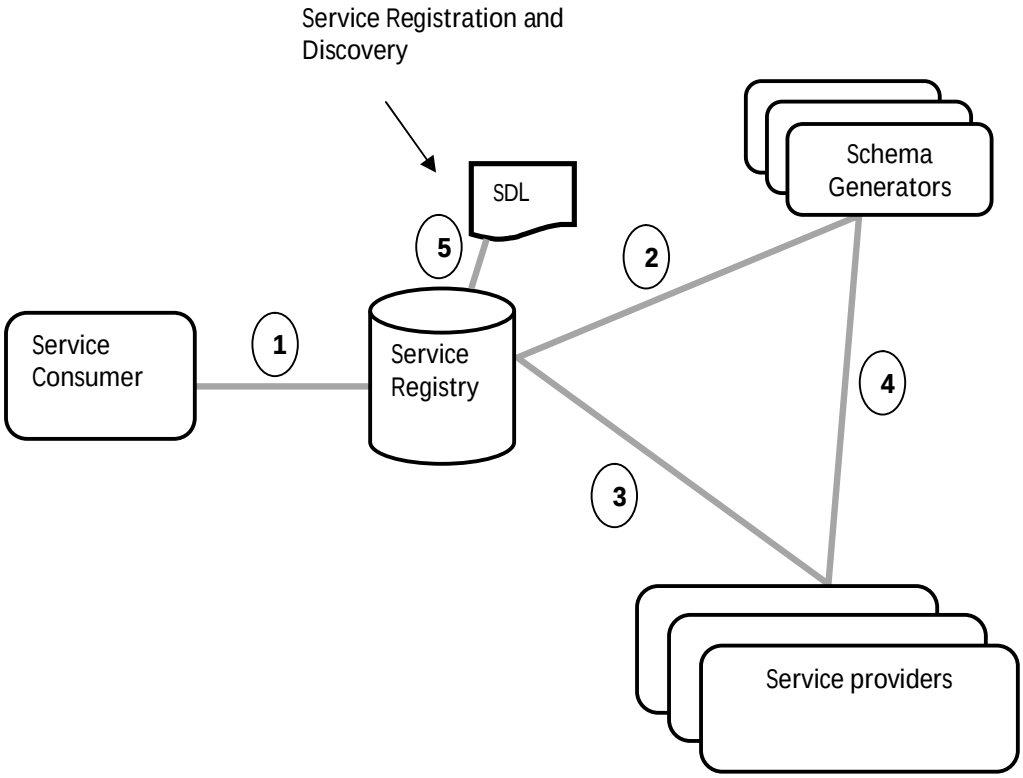

Figure 4.2: Abstract architecture of the proposed system

4.2.1 Architecture

The abstract architecture shows the big picture of the proposed architecture. It is comprised of components and connection links from one to another. The proposed system has five major components, and each component in turn may have sub components. Figure 4.2 clearly shows all the major components. Here below is an explanation of each major components of the architecture.

A. Service Consumer

Alkkiomäki et al. [Alkkiomäki2012] defines service consumer as “an application, service, or some other type of software component that requires the functionality of the service. The service consumer executes the service by sending it a request according to the service interface.”The requestor utilizes an existing web service by opening a network connection and sending a request. Furthermore, the role of service consumer requires certain requirements and needs that are fulfilled by one or more web services available over the Internet [Granell2009].The *service contract* is consisted of the requirements that have to be fulfilled by both the service provider and service consumer (already discussed in chapter two). Notice that the service consumer can be also named as client or service requester.

The service consumer consults the service registry any service that can process its request (1) is available. When the service is available in the registry then the service consumer proceeds with its request. If the request information from the service consumer requires API wrapping then the request will be sent to the schema generators and to the service providers. Otherwise it will be sent to the service provider only. The service Registry knows well about these information.

B. Service Provider

The service provider is an application that executes requests from service consumers in accordance with the service interface. The role of service provider is to implement the

service and makes it available on the public registries or Internet by creating functional descriptions [Granell2009]. A service provider can also act as a service consumer.

The service provider should constantly publish its service description to the *service registry* (3) so that the information in registry will be up-to-date. After doing the binding process, the service provider may receive a request from the service consumer. After that the service provider processes the request and prepares the response accordingly. If the response requires customization then the service provider searches for the required wrapper or schema generator from the registry (3). Once the needed schema generator is found then the service provider acts as a service consumer and then they both undergo the binding process (4). After that, the service provider sends the response to the schema generator (4). Other alternative, if customization of the response is not needed then the service provider sends the processed response back to the service consumer.

C. Service Registration and Discovery

Service registry is a centralized directory of services. The registry is used as a central place where providers or developers can publish new services or find existing ones. It therefore serves as a centralized clearinghouse for companies and their services**.** Moreover it allows efficient communication by creating a link between service providers and service customers. The primary Objective of Service Registry is to provide fast, easy access to communication, and to operate among different applications with a limited human intervention. Some of the benefits of using service Registry are [Wishworks2015] [Richardson2016]:

- It is constantly evolving catalog of information about the available services and it helps in managing service located in different places (internally, externally).
- Service Repository is where metadata of services and related artifacts, such as policies can be stored.
- It provides an integrated Governance Solution management.
- Provides access to search facilities, notification services, and optimizes service reuse.
- It Manages the Service life-cycle and visibility

Service Discovery is the process of identifying web service providers, and retrieving web services descriptions that have been previously published. The primary mechanism involved in performing of Service Discovery is a service registry, which contains relevant metadata about available and upcoming services as well as pointers to the corresponding service contract documents that can include Service License Agreements (SLAs). In modern architectures individual service instances need to be decoupled from the knowledge of the deployment topology of the architecture. After the discovery process is completed, the service developer of the service consumer exactly knows the location of a service (URI), its capabilities, and how to interface with it. Some of the benefits associated with Service Discovery are [Wishworks2015] [Richardson2016]:

- Discovery of the service, its status, and its owner will be helpful for service reusability.
- Dynamic service registration and discovery allows avoiding service interruption.
- It helps in handling fail over of service instances
- It allows load balancing across multiple instances of a Service

Therefore the service Registration and Discovery process is consisted of three parties; the service consumer, the service provider and the service Registry as discussed in chapter two. Services need to be described using some standard description language so that to be discovered easily. In the architecture above, the service registry receives the consumer request (1) and then searches the service description file (5) to check if the required service provider is available. If it is available then it sends back the service provider's description to the consumer (1). In addition to that, the service registry constantly updates the information of the service providers' (2 and 3). The schema generator services also use the service registry and service discovery in building the new schema and when they interact with service consumer as well as the service provider.

Talking about the technology; the *AWS Elastic Load Balancer (ELB)* is an example of a server-side discovery router. *Netflix OSS* provides a great example of discovery [Richardson2016]. *Netflix Eureka* is a service registry and it provides a REST API for

managing service instance registration and for querying available instances. *Etcd* is a highly available, distributed, consistent, key value store that is used for shared configuration and service discovery. Two notable projects that use etcd are Kubernetes and Cloud Foundry. *Consul* is a tool for discovering and configuring services. It provides an API that allows clients to register and discover services and it can perform health checks to determine service availability. *Apache Zookeeper* is a widely used, high-performance coordination service for distributed applications. It was originally a subproject of Hadoop but is now a top-level project. WSDL, RSDL (RESTful Description Language), Swagger, YAML …are some examples of service description languages. WSDL is well explained in chapter two.

D. Schema Generators

The schema generators could be components of services or modules that transform existing service providers' schema to another type of schema. The new created schema is used to customize the processed API response from the service provider to the needs of the service consumer. Note that there could be multiple schema generators thus the service consumer or service provider has to look for the appropriate one.

In Figure 4.2 the schema generators constantly publish their services to the registry (2). If the request from the consumer requires wrapping or customization of the API response, then the selected schema generator will receive the request and then builds a new schema according to its specifications. The service provider also sends the processed response to the schema generator. After that the schema generator customizes the response from the service provider according to the new schema. Lastly, the customized response is sent back to the service consumer. *Note that, the schema generators execute only when customization of the API response is required.* Examples of schema generators include those discussed in chapter three; *GraphQL-REST*, *Swapi-to-GraphQL* and *Swagger-to-GraphQL.* The proposed schema generator known to be *service based proposed schema generator* is also a good example.

4.2.2 Schema Generator

As previously noted, the good features as well as the challenges experienced on the related works discussed in chapter three are the motivation behind the concept development of this research. Particularly, the proposed schema generator considers the features and challenges experienced by the schema generator of the swagger-to-GraphQL tool. Why this tool and not the others? The research done on these three tools shows that the swagger-to-GraphQL provides better features than the other two. Moreover, the schema generator of the swagger-to-GraphQL operates better than the two as shown in Table 4.1 and it shows that Swagger-to-GraphQL has clear advantages above the other two.

Tools	**How is schema generated?**		
	Query	**Mutations**	**Resolvers**
GraphQL-REST	Automated	Not Supported	Automated
Swapi-to-GraphQL	Partially automated	Not Supported	Manual
Swagger-to-GraphQL	Automated	Automated	Automated

Table 4.1: Comparison of the wrapping tools

In addition to that swagger is becoming famous API description language (for REST). Now taking Swagger-to-GraphQL's schema generator as a foundation for this concept is feasible. Figure 4.3 shows the tasks or services within the proposed schema generator. Basically there could be many options on how to design this proposed schema generator. For example the three parts of the schema generator could be treated as one component (monolithic) of the system or all the three parts could be treated as services. When the later one is deployed then system will be service based architecture.

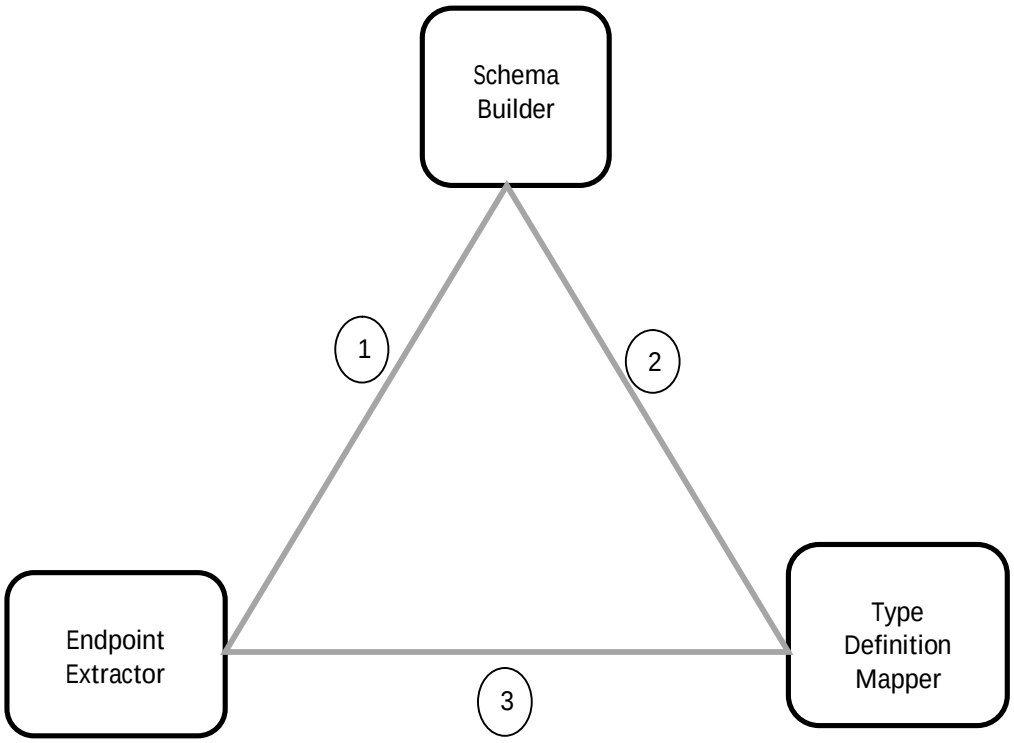

Figure 4.3: proposed service based schema generator

Schema Builder: This is the service that has multi functions.

- Initially loads the provider's native schema definition (e.g. swagger) so that it will be ready to be processed by the other components (loads schema)
- It sends the loaded schema to the other components (1,2)
- It integrates the outputs from the other two services and builds the intended schema.
- It can also act as the router for the whole component.

Endpoint Extractor: This one has limited scope

- It receives the loaded schema from schema builder (1)
- It extracts endpoints from the schema
- Then sends the extracted endpoints to the TypeDefinitionMapper (3)

TypeDefinitionMapper: This one has to do some tasks

- It receives the adopted schema from the schema builder (2)
- It receives the endpoints from Endpoint Extractor(3)

- It translates the data types of the native schema to a given languages types(e.g. GraphQL data types)
- Sends the processed or typed schema to the schema builder

Figure 4.4 shows the process of building schema using the service based schema generator of the architecture in Figure 4.3.The service based schema generator initiates after binding is done with a service consumer and a request arrives accordingly(1). Besides the usual query, the request may contain the Base_URL and the schema definition of the service provider. If these two are not available within the request then the Schema Builder uses its default configured schema and Base_URL (2). In either way, the Schema Builder loads the available schema and its validity is checked (3, 4). If the definition fie is not supported or not valid then the process terminates with an error message before going long way (5). Otherwise, the loaded schema is sent to both the Endpoint Extractor and the Type Definition Mapper (7, 12). The sending of the loaded schema could be at the same time or at different times; it depends upon the configuration of the Schema Builder. The Endpoint Extractor receives the loaded schema and it extracts all the available endpoints from the schema (8, 9).The endpoints are consisted of all corresponding parameters and attributes. After that, the list of endpoints is sent to the Type Definition Mapper (9). The Type Definition Mapper receives both the list of endpoints and the loaded schema and then wraps the types of the endpoints to the data type of a targeted language such as GraphQL (10, 11, 12, 13 and 14). The loaded schema and the extracted end points do not necessarily arrive at the same time and the parallel gateway is used just for convenience. Moreover the necessary libraries of the targeted language are used in mapping the types from one to another. The type mapped endpoint definitions are sent back to the Schema Builder (15). Finally the type mapped endpoint definitions are received by the Schema Builder and the necessary schema of the targeted language is built (16, 17 and 18). Here again the libraries of the targeted language are used to build the schema.

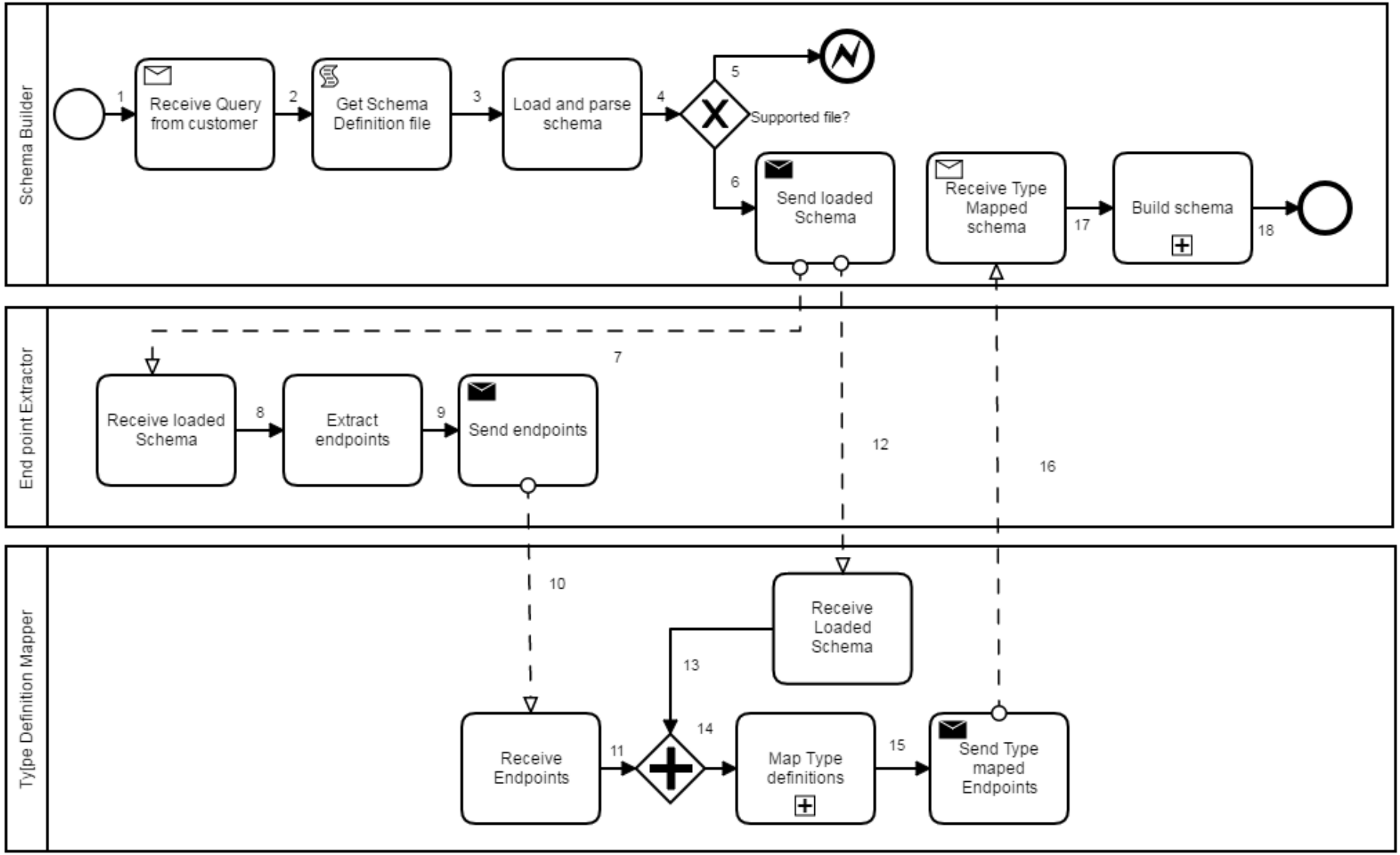

Figure 4.4: Generating schema using the service based schema generator

Interaction: the service based component can execute in distributed systems, running on multiple machines and each service instance is typically a process. Consequently, services must interact using an *inter-process communication (IPC)* mechanism [Richards2015]. Typically, each service uses a combination of these interaction styles.

For one-to-one interactions:

- *Request/response*: A service consumer waits for a response once it makes a request to a service. The client expects the response to arrive in a timely fashion.
- *Notification or one-way request:* A service consumer sends a request to a service and it does not expect to receive response from service provider.
- *Request/async response*: A service consumer sends a request to a service, which replies asynchronously. The service consumer does not block and it can accomplish other tasks while waiting.

For one-to-many interactions:

- *Publish/subscribe*: A service consumer publishes a notification message, which is consumed by zero or more interested services.
- *Publish/async responses*: A service consumer publishes a request message, and then waits until other services responds to it.

In the overall system, the services communicate between each other using messaging protocol (such as SOAP or REST) and the massage is carried by transport protocol (e.g. HTTP).

4.2.3 Service Consumer View

Figure 4.5shows the flow of the request from the service consumer to the service provider and the schema generator. The service consumer consults for the registry if any service that can process its request is available (1, 2, and 3). The service registry searches its registry to find the required service (4). The service registry searches for available services. If customization of the response is required then the service registry sends service information to available schema generator/wrapper, otherwise it sends

information to the service provider (5, 6). The service consumer receives the information and after that binding with either service provider or wrapper is done (7).

The service consumer which can be a client application (or even a service) prepares a request and makes it ready to be sent to those services where binding is already done (8). In addition to the query, the request could contain additional optional parameters. For example the request could be consisted of the proxy URL or the schema of the intended service provider. The former one is very important when multiple service providers exist. But these are Optional variable since the schema generator or wrapping tool can have default proxy URL and schema file. The direction of the request has two options, once the service consumer makes it ready to be sent (8, XOR gateway or 9).

Customization of response is required: The request goes to available wrapper or schema generator (yes).The schema generator receives the request (11). After that, the schema generator builds the new schema according to section 4.2.2 and then sends a request call to a service provider (12, 13 and 14). After that, it waits for the response to come from the service provider. The service provider sends back the processed response to the schema generator (15). The schema generator or wrapper customizes the response against the definition within the generated schema. Lastly, the schema generator sends the customized response back to the service consumer (17).

The service provider can consult the registry to find available wrapper or schema generator (22). In addition to that services of the schema generator can consult the service registry when interacting between each other to build schema (23). The consultation is done the same as what the service consumer has done.

Customization of response is not required: The service consumer sends the request to the service provider only. Once the service provider processed the request then it sends the response back to the service consumer (19). The service consumer receives either the customized response from the schema generator (18) or the response from the service provider (19). It processes the received response accordingly (20, 21). The pseudo code in Listing 4.1 summarizes these all processes.

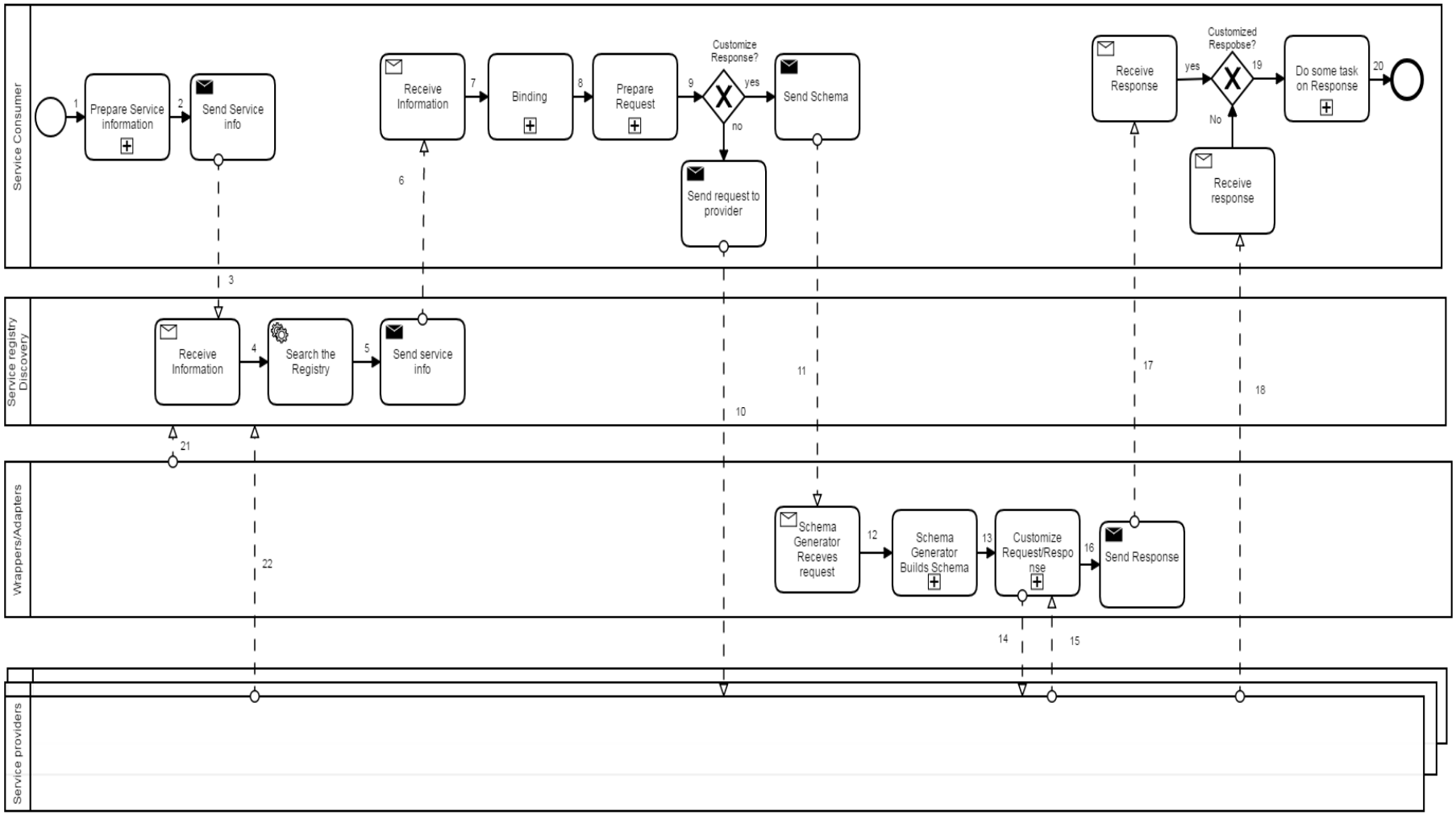

Figure 4.5: Sending Request from Service Consumer to Service Provider

1. *serviceConsumer* SENDS Request
2. IF Customized_Response REQUIRED
3. *schemaGenerator* RECEIVE Request
4. *schemaGenerator* BUILDS *schema*
5. *schemaGenerator* CUSTOMIZES Request
6. *schemaGenerator* SENDS Request
7. s*erviceProvide*r RECEIVES Request
8. s*erviceProvide*r PROCESSES Request
9. s*erviceProvide*r SENDS Response
10. *schemaGenerator* RECEIVES Response
11. *schemaGenerator* CUSTOMIZES Response
12. *schemaGenerator* SENDS Response
13. ELSE
14. *serviceProvider* RECEIVES Request
15. *serviceProvide*r PROCESSES Request
16. *serviceProvider* PREPARES Response
17. ENDIF
18. *serviceConsumer* RECEIVES Response

Listing 4.1: The pseudocode for the flow of request from service consumer to service provider

4.2.4 Service Provider View

The Service provider describes its service using any of the service description languages (e.g. WSDL, YAML and Swagger) and sends the service information to the registry (1, 2). The service registry receives the service information and it registers or updates the service in its entry (3, 4). After that it publishes the service so that it will be discoverable by service consumer or other services (5, 6).

The service provider receives a request from a service consumer or schema generator after the necessary binding procedures (7, 11). After that it processes the request according to its service specifications. Once processed, the response from the service provider has two directions (XOR Gate or 9).

Customization of response is required: The processed response is sent to the schema generator (yes).The schema generator receives the response and customizes it according to the definition in the generated schema (16, 18). The customized response is sent back to the service consumer (19, 20). The services of the schema generator can describe and publish their service information the same as the service provider has done (21). Once the service information is published in the service registry, other service members of the schema generator can discover it easily. This way, the schema is built by the coordination of the services of the schema generator.

Customization of response is not required: The service provider normally sends the processed response back to the service consumer (no decision of the XOR gateway). The service consumer receives the response from the service provider (17). The pseudocode for the service provider architecture view is in Listing 4.2.

```
1. serviceProvider RECEIVES Request from schemaGenerator OR  serviceConsumer
2. serviceProvider PROCESSES Request
3. IF Customized_Response  REQUIRED
4.       serviceProvider SENDS Response // to schemaGenerator
5.       schemaGenerator RECEIVESResponse
6.       schemaGenerator CUSTOMIZES Response
7.       schemaGenerator PREPARES Response
8.       schemaGenerator SENDS Response // to serviceConsumer
9.    ELSE
10.       serviceProvide rSENDS Response    // to serviceConsumer
11.   ENDIF
12.   serviceConsumer RECEIVESResponse
```

Listing 4.2: The pseudocode for the flow of Response

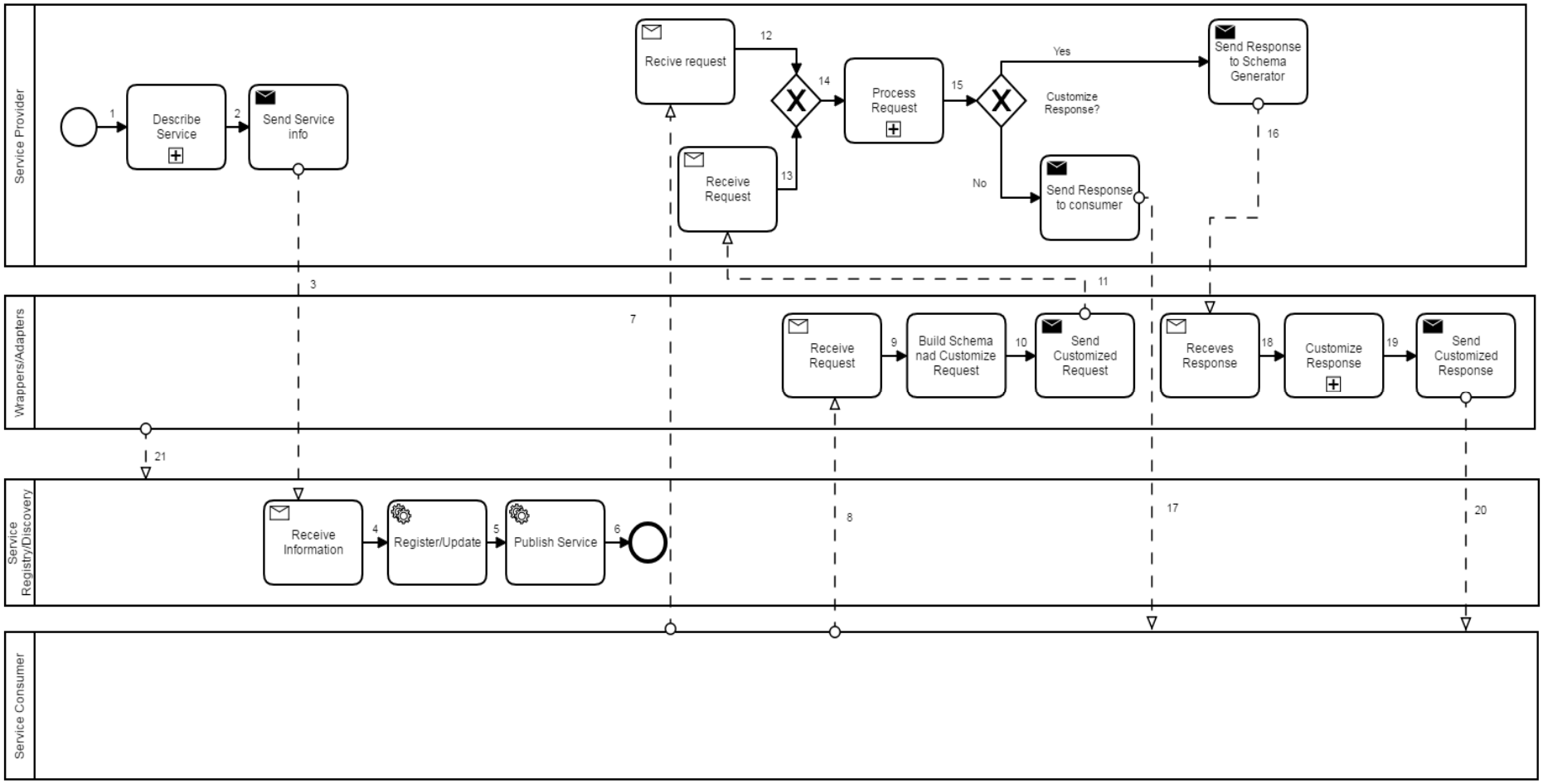

Figure 4.6: Sending Response from Service Provider to Service consumer

CHAPTER 5
VALIDATION

In the previous chapter, the concept development of the proposed system is discussed. Now the concept has to be realized using a prototype. The prototype in its turn has to be validated using real world data. Therefore this chapter discusses about validation of the developed concept.

A service based prototype known as *REST2GraphQL* is developed to realize the concept developed in chapter four. The main task of the REST2GraphQL is to wrap REST calls (from service provider) into GraphQL API. In order to validate this prototype, *Petstore* [PetsoreV2] will be considered as the service provider. Petstore is one of the famous swagger based RESTful service providers. It was developed by the swagger team and it is also well known amongst software developer community. A use case will be also provided to show the differences of using REST2GraphQL prototype against another existing client application that uses the Petstore as its service provider [PetstoreClient]. During the use case experiment, the request and response of both the REST2GraphQL and the Petstore client application will be shown. Furthermore, the flexibility exhibited during the request/ response samples of each application will be observed and analyzed. How easy is it to get the response of the requested information? Does the application provide flexibility towards building the needed request? On the other hand, does the application provide the exact needed response data? How easy is it to build a request or to receive a response?

5.1 REST2GraphQL Prototype

In the last chapter, concept was developed into a proposed system. But the verbose proposed system has to be converted into a working technical system. The technical realization of the proposed system is what we call the Prototype.

This section discusses about the *prototype* of the proposed concept. This prototype (REST2GraphQL) is a service based implementation or prototype of the concept developed in chapter four.

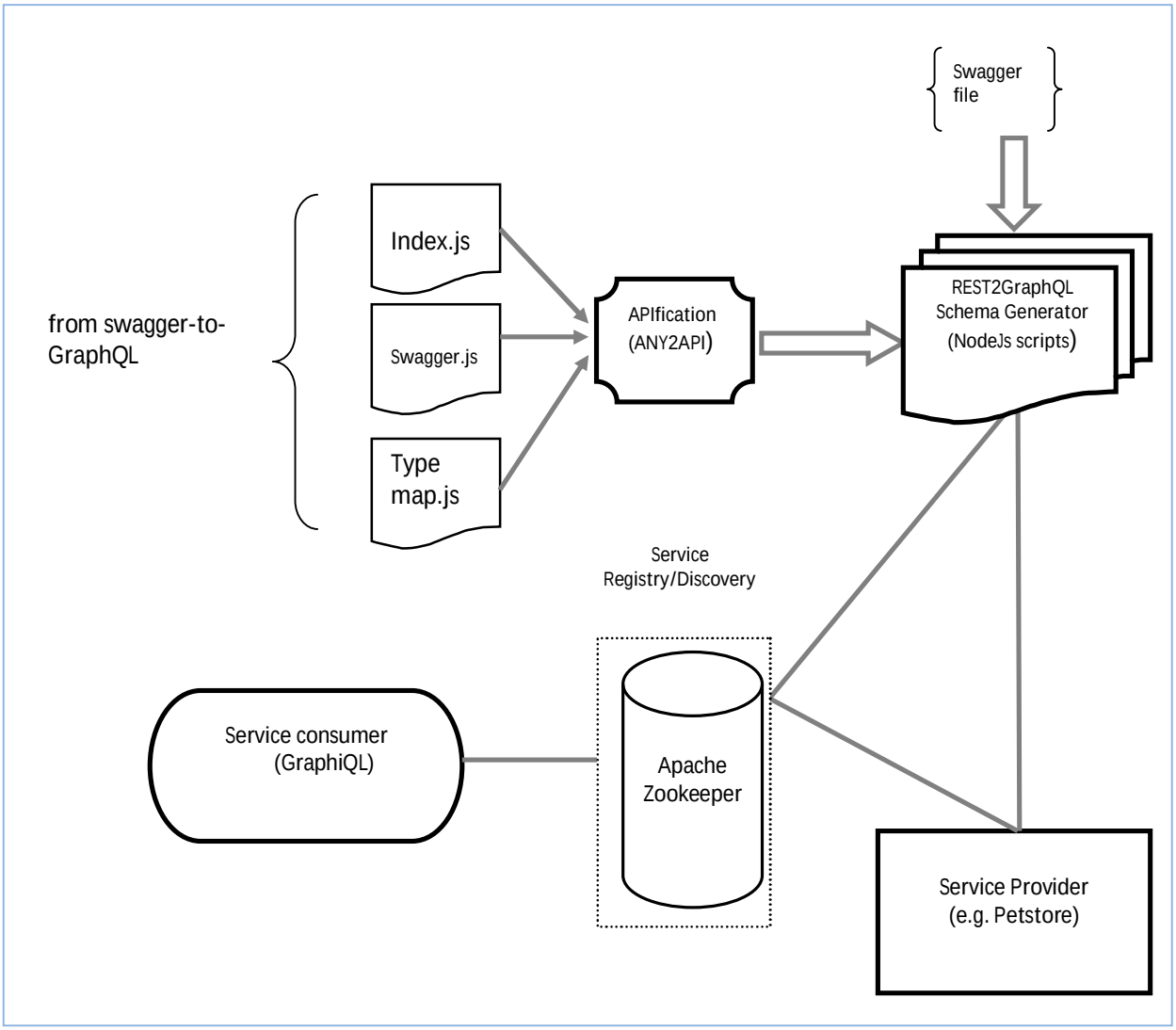

Figure 5.1: Components of the REST2GraphQL prototype

The components of the REST2GraphQL prototype mapped to the tools used are as shown in Figure 5.1. Below is the explanation of each component of the prototype.

A. Service Consumer

The concept developed and the prototype is considered as a development stage. Therefore, GraphiQL will be deployed as the client application of the REST2GraphQL . As discussed in section 2.3, it has good features and it is feasible to use it for this prototype.

B. REST2GraphQL schema Generator

The swagger-to-GraphQL schema generator from chapter three is used as a foundation for this prototype. Similar to the Swagger-to-GraphQL tool, this prototype also expects a swagger file. It is good to remind that, the swagger-to-GraphQL is built based on simple architecture hence couldn't comply with the proposed concept. Therefore the simple architecture of the schema generator of the swagger-to-graphql tool has to be transformed into service based architecture. Moreover, the swagger-to-GraphQL tool lacks some important features like *application/XML* content type and these features have to be added to the prototype. The implementation of the prototype for the REST2GraphQLschemagenerator is also based on Node.js and express. Actually Node.js version 6.10.3 and express version 4.15are used to build the proposed schema generator.

To proceed with the transformation, the first thing to do is to convert the scripts of swagger-to-GraphQL (*swagger.js, index.js and typeMap.js)* into service APIs. Already developed tools like ANY2API (from chapter three) can be deployed. An API implementation of each service that exposes a RESTful interface is generated. Alternatively, the transformer (e.g. ANY2API) can provide a *Docker file* which is used for each generated API implementations. Using the Docker file, a self-contained and portable container image can be created [Wettinger2015]. In addition to that, Docker registries (private or public) can be employed to store, manage, and retrieve variety of pre-built API implementations versions.

Once the scripts are APIfied, communication or interaction using RESTful interfaces of the converted scripts can be established. When the interaction is established then the services will coordinate to build a GraphQL schema from a given *swagger file.* This REST2GraphQLschema generator of the prototype expects a swagger file and Base_URL of the service provider.

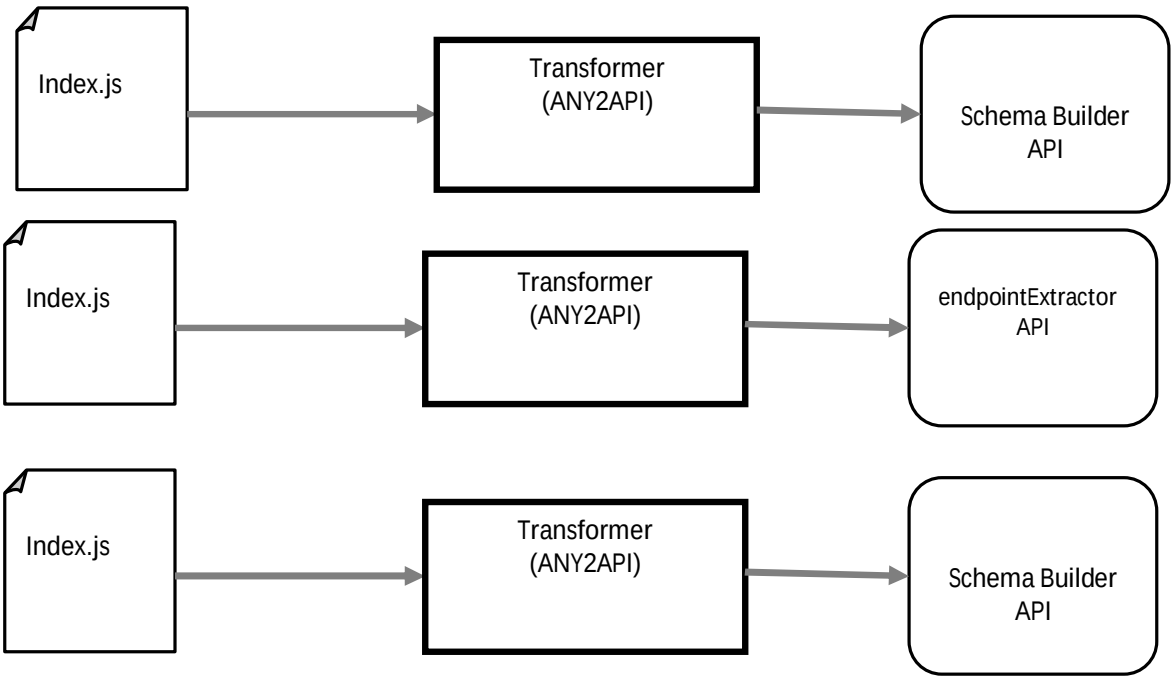

Figure5.2: Transformation of existing scripts into service with REST API interfaces

The GraphQL schema will be built by the coordination of the three services of the schema generator component; *endpointExtractor.js, schemaBuilder.js and typeDefMapper.js.* All the three formed services are supposed to provide REST APIs. Therefore, the interaction between these services is done through their REST API interfaces.

scheamaBuilder.js: This service also acts as a router for the schema Generator component. The Input and output to the component passes through this service. Once it receives the request from the GraphiQL, the swagger file is loaded and prepared for usage by the other two services. Since the swagger file is in JSON format, *REST* on HTTP is used to transport the schema to the two services. On the other hand, it

receives the *GraphQL type mapped schem*a from the *typeDefmapper.js*. It finalizes building of the GraphQL Schema and makes it ready to be used to customize the response from the service provider. The schema Builder customizes the response from the service provider once it receives the response. The response from the service provider is also expected to the schemaBuilder is also transported using REST on HTTP.

endpointExtractor.js: This is the service that receives the loaded schema from the schemaBuilder service and it extracts the endpoints by parsing the loaded schema. After that it makes the extracted endpoints ready to be consumed by the typeDefMapper service.

typeDefMapper.js: This service receives the loaded schema and the extracted endpoints from the schemaBuilder and the endpointExtractor services respectively. It maps the schema to GraphQL native data type definitions and then sends it to the schemaBuilder service. The interaction between endpointExtractor and TypeDefMapper is request/response style and it is done using REST.

To summarize the process of GraphQL schema Generator:
Input: Swagger file and proxy URL are required by the schema generator. For this prototype, these are configured as default at the router or schemaBuilder service of the schema generator.
Output: The swagger file is parsed and a new GraphQL schema is generated.

C. Service Provider

The service provider of this prototype is supposed to be any RESTful provider where its service is described in swagger. Normally the swagger file is JSON formatted and this prototype expects such format. The exclusive swagger file will be provided to the REST2GraphQL schema generator when customization of the response from the provider is required.

D. Service Registry/Discovery

The interaction between all the components of the REST2GraphQL is coordinated using *Apache Zookeeper service registry*.

- The user of the GraphiQL discoveries for service provider or schema generator using this tool.
- The service provider and the schema generator use this tool to publish their service information.
- Services of the schema generator use this tool to publish or discover service information.
- The service provider can use this tool to discover the required schema generator.

5.2 Runtime Scenarios

Below are some of the scripts collected during the run time of the prototype. To make the validation meaningful, the customized or minimal swagger file of the petstore is considered. The original petstore swagger is large, thus it will not make sense to use it here for practical reasons. Listing 5.1 shows the minimal swagger file. This minimal petstore swagger file is built for *application/JSON* content type only. Therefore it will consume and produce JSON contents only.

A. Petstore Swagger file

```
{  "swagger": "2.0",
  "info": { "version": "1.0.0", "title": "Swagger Petstore",
    "description": "minimal petstore swagger",
    "contact": { "name": "Swagger API Team" },
    "license": { "name": "MIT"}
  },
  "host": "petstore.swagger.io",
  "basePath": "/api",
  "schemes": ["http"],
  "consumes": ["application/json"],"produces": ["application/json"],
  "paths": {
    "/pets": {
      "get": {
        "description": "Returns all pets from the system that the user has access to",
        "produces": ["application/json"],
        "responses": {
          "200": { "description": "A list of pets.", "schema": { "type": "array",
                "items": {"$ref": "#/definitions/Pet"}
          }
        }
      }
    },
    "post":{
          "tags":["pet"], "description":"Add a new pet to the store",
          "operationId":"addPet", "consumes":["application/json"],
          "produces":["application/json"],
          "parameters":[{"in":"body","name":"body",
             "description":"Pet object that needs to be added to the store",
             "required":true, "schema":{"$ref":"#/definitions/Pet"}
                  }],
          "responses":{"405":{"description":"Invalid input"}},
          "security":[{"petstore_auth":["write:pets","read:pets"]}]
        }
    }
  },
  "definitions": {
    "Pet": { "type": "object",
      "required": ["id","name"],
      "properties": {
        "id": {"type": "integer","format": "int64"},
        "name": { "type": "string"},
        "tag": {"type": "string"}
      }
    }
  }
}
```

Listing 5.1: Minimal swagger file of the petstore adopted from [OAI2017]

B. Extracted EndPoints

As discussed during the concept development and the implementation of the prototype, the endpointExtractor of the schema Generator extracts the endpoints from the swagger definitions. This schema of endpoints will be provided to the typeDefMapper service so that the native data types will be converted to the GraphQL data types. Figure 6.1shows the endpoints produced from the loaded schema (minimal-petstore swagger file) of schemaBuilder service.

```
{ get_pets:
   { parameters: [],
     description: 'Returns all pets from the system that the user has access to',
     response: { type: 'array', items: [Object] },
     request: [Function: request],
     mutation: false },
  addPet:
   { parameters: [ [Object] ],
     description: 'Add a new pet to the store',
     response: undefined,
     request: [Function: request],
     mutation: true } }
```

Figure5.4: The end points extracted from the minimal-petstore taken from command line

C. Swagger to GraphQL type Mapping

The native data types of the petstore swagger file has to be converted into GraphQL data types. The code snippet in Listing 5.2 shows how the swagger types are converted into their equivalent GraphQL data types.

```
1. Const primitiveTypes={
2. String:graphql.GraphQLString,
3. date: graphql.GraphQLDate,
4. integer:graphql.GraphQLInt,
5. number: graphql.GraphQLInt,
6. boolean:graphql.GraphQLBoolean
7. };
```

Listing 5.2: Code snippet from typeDefMapper.js that maps native swagger types to GraphQL data types

D. Building the schema

Once the data types are mapped, the GraphQL schema is built as shown in Figure 5.5. The response from the petstore service provider will be customized according to the definitions in this file.

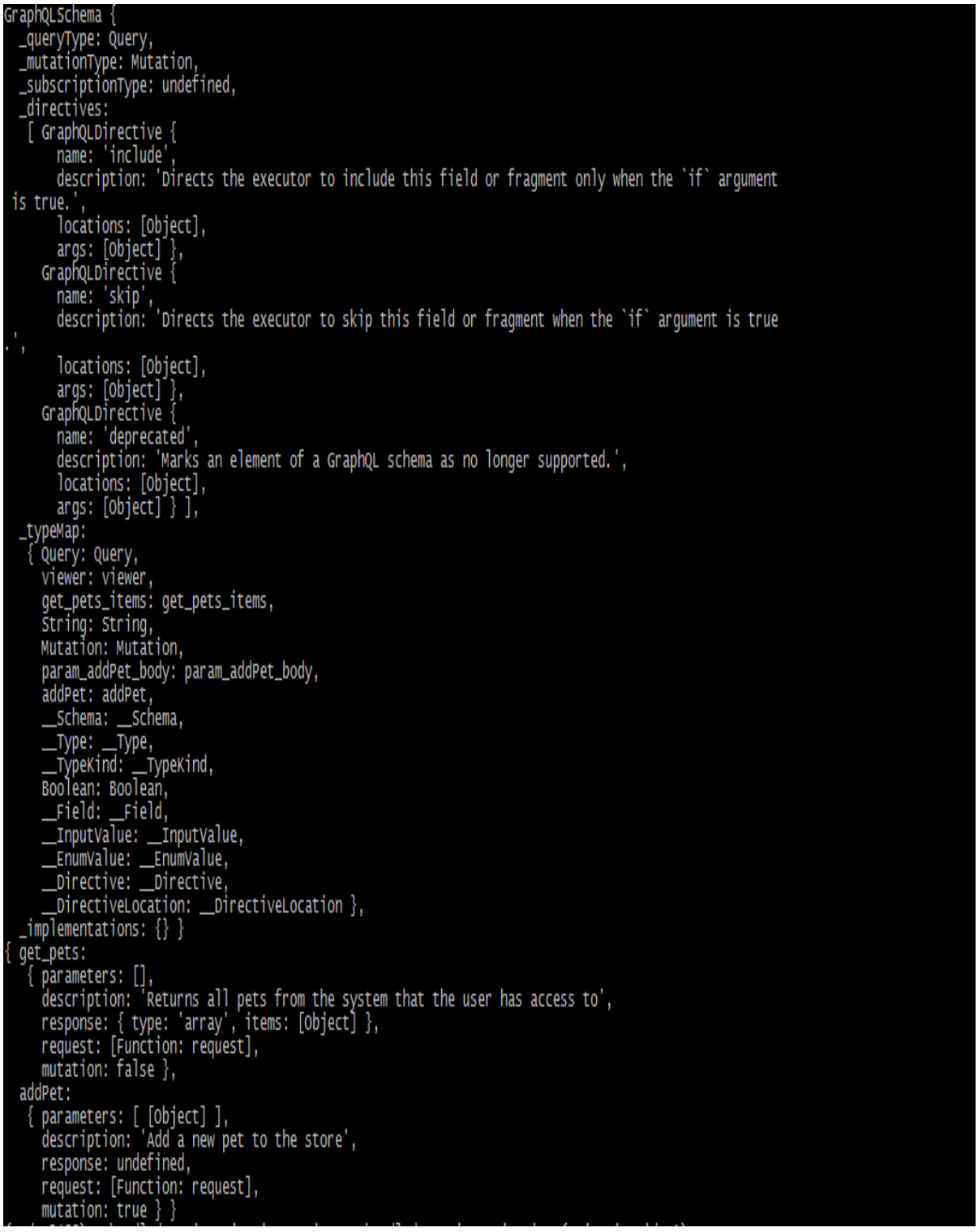

Figure 5.5: GraphQL Schema created from the REST2GraphQL taken from command line

5.3 Use Case

To validate the proposed prototype against the given service provider, an example will be provided. The example is to get a pet with certain ID from the petstore database.

Get the name and status of a pet with Id =33

As it can be seen from the screen shot of the GraphiQL in Figure 5.4, the user of the REST2GraphQL application requests (left panel) some specific information of the pet (with id=33) from the petstore API service. The same Figure shows that the user receives response from the service provider in style similar to the Requested information.

Figure5.4: The Request/Response from REST2GraphQL

On the other hand the screen shot of Figure 5.5 shows the request sent and the response received from the petstore client application. The Request URL is the request made to get the information of the Pet with ID=33. The Response body is the response received from the petstore service provider.

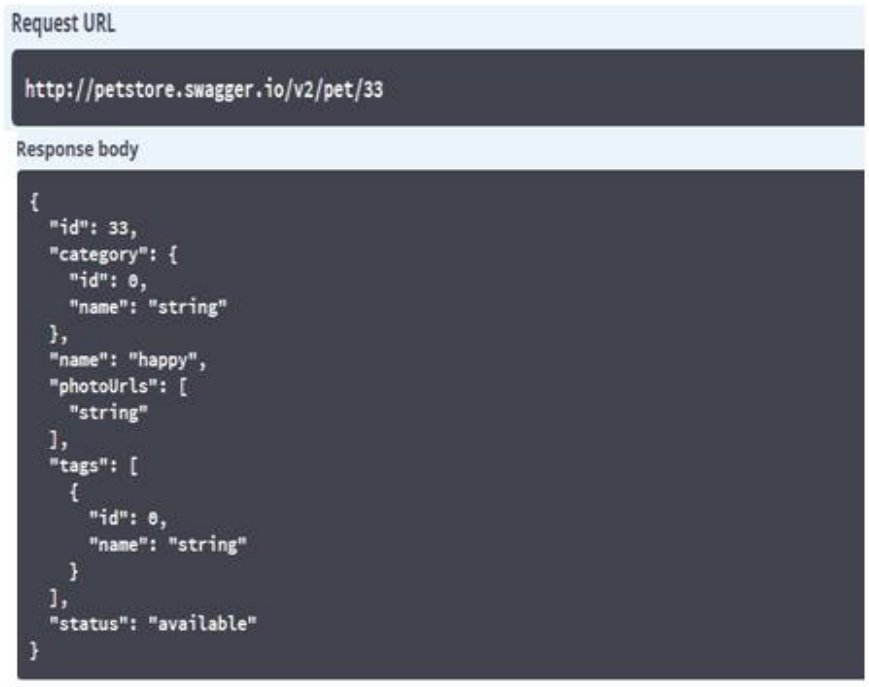

Figure 5.5: The Request/Response from the petstore client application

CHAPTER 6
CONCLUSION AND Future Work

6.1 Conclusion

This master thesis has come out with a generic concept to enable wrapping of REST to GraphQL API. Particularly, OpenTOSCA ecosystem is the long term target of the concept developed. Several concepts and technologies were discussed and researched in chapter two and they were good ingredients to the concept development. Furthermore, the deeper understanding and research on projects related to the thesis was a foundation to the concept development. In chapter three, a framework that is used to generate API implementation from executables or scripts was discussed. In addition to that several tools and approaches that generate GraphQL schema were also discussed.

The generic concept developed during concept development has to be transformed into technical ideas. Therefore, a prototype was developed to show that the concept works. As discussed earlier, the research done on the related works of chapter three is the driving force behind the prototype. Particularly, swagger-to-GraphQL was used as a foundation for constructing the prototype. The ANY2API framework was also intensively used to create API implementations from the executable scripts.

The validation stage shows that the objectives of the thesis work are achievable. Moreover, the validation result shows that the problems stated in chapter one are solvable using this concept and by a prototypical implementation.

6.2 Future Work

First of all, it is important to remember that it is not long time since GraphQL was introduced to the world as an API tool. It was just by September 2016 officially announced as production ready [GQLorg2017]. Many of the software developer communities are still working hard to enrich it. Approaches and tools that could enhance the usage of GraphQL are yet to come out. Moreover, the scope of the thesis topic covers wide areas and it couldn't be perfectly done for known reasons. In which time constraint and scarcity of available tools are some of the factors. Therefore, this research work has to be considered as an important step to the long journey.

GraphiQL is used as a client application for GraphQL during the prototype and the validation stages of this research work. As discussed in chapter two, GraphiQL can be considered as a client application for GraphQL. Although GraphiQL has some fantastic features, it is yet considered a tool for development stage. Usually, it is turned off during deployment or production time. Therefore, a client application for GraphQL has to be developed using either of the tools discussed in chapter two (GraphQL client applications). Once the client application is developed then many features that affect performance can be implemented easily. For example it would become easy to implement Caching, Pagination and Batching. Moreover, with the introduction of client application, the limited content type support of the prototype can be expanded for others like XML.

Lastly, it is good to evaluate the performance of the prototype and the concept developed in this research. Based upon the evaluation, a decision can be reached on whether to deploy this system for production or not.

Bibliography

[Albreshne2009] Albreshne, A. B. D. A. L. D. H. E. M., Fuhrer, P. A. T. R. I. K., &Pasquier, J. A. C. Q. U. E. S. (2009). Web services technologies: State of the art. *Definitions, Standards, Case Study*.

[Alkkiomäki2012] Alkkiomäki, V., & Smolander, K. (2012). Service elicitation method using applied qualitative research procedures. In *Advanced Design Approaches to Emerging Software Systems: Principles, Methodologies and Tools* (pp. 1-17). IGI Global.

[Alligator2017] Alligator. Introspection Queries with GraphQL. [Online]. Available: https://alligator.io/graphql/introspection-queries/ (Last visited 10.10.2017)

[Allsopp2016] Clay Allsopp. GraphiQL: GraphQL's Killer App**.** [online]. Available:https://medium.com/the-graphqlhub/graphiql-graphql-s-killer-app-9896242b2125 (Last visited 1.10.2017)

[Alon2016] Alon, Graphql-REST-Wrapper 2016. [Online]. Available: https://github.com/alonp99/graphql-rest-wrapper (Last visited 15.06.2017)

[API2017] ProgrammableWeb. 9000APIs: Mobile gets serious| Programmable Web. [Online]. Available: https://www.programmableweb.com/news/9000-apis-mobile-gets-serious/2013/04/30 (Last visited 25.10.2017).

[Barry2003] Barry, D. K. (2003). *Web services, service-oriented architectures, and cloud computing*. Morgan Kaufmann.

[BBVA2016] BBVA API_MARKET. 101: Introduction to the world of APIs. [Online].Available:https://bbvaopen4u.com/en/content/ebook-101-introduction-world-apis (Last visited 20.10.2017)

[Bela2015] Petr Bela. GraphQL in the age of REST APIs. [Online]. Available: https://medium.com/chute-engineering/graphql-in-the-age-of-rest-apis-b10f2bf09bba (Last visited 25.08.2017).

[Bruno2017] Bruno Reis . GraphQL Tutorial: How to build an App in PHP. [online]. Available: https://www.scalablepath.com/blog/how-to-build-graphql-app-php/ (Last visited 15.10.2017)

[Buna2017] Samer Buna. REST APIs are REST-in-Peace APIs. Long Live GraphQL. [Online]. Available: https://medium.freecodecamp.org/rest-apis-are-rest-in-peace-apis-long-live-graphql-d412e559d8e4 (Last visited 22.09.2017)

[Cerami2002] Cerami, E. (2002). Top ten FAQs for Web services. *Disponível na Internet. http://web.oreillynet.com/lpt/a//webservices/2002/02/12/webservicefaqs.html*, *27*. [online].Available: http://archive.oreilly.com/pub/a/Webservices/2002/02/12/Webservicefaqs.html (Last visited: 25.08.2017)

[Chauhan2017] Anshul Chauhan. Designing scalable backend infrastructures from scratch. [Online]. Available: https://medium.com/@helloansh/designing-scalable-backend-infrastructures-from-scratch-af80f5767ccc (Last visited: 15.09.2017)

[Chatterjee2004] Soumen Chatterjee, Cap Gemini Ernst and Young. Messaging Patterns in Service-Oriented Architecture, Part 1. [Online]. Available:https://msdn.microsoft.com/en-us/library/aa480027.aspx (Last visited 22.09.2017)

[Dragoni2016] Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R., & Safina, L. (2016). Microservices: yesterday, today, and tomorrow. *arXiv preprint arXiv:1606.04036*.

[Fakorede2007] Fakorede, O. (2007). *An investigation into the implementation issues and challenges of service oriented architecture* (Doctoral dissertation, Bournemouth University).

[Fielding2000] Fielding, R. T., & Taylor, R. N. (2000). *Architectural styles and the design of network-based software architectures* (p. 151). Doctoral dissertation: University of California, Irvine.pp 94-106

[Flowgica2017] Flowgica Technologies. Microservices architecture. [online]. Available:http://flowgica.com/microservice-architecture/ (Last visited 19.10.2017)

[Fowler2016] Fowler, M., & Lewis, J. (2016). Microservices a definition of this new architectural term (2014). *Saatavissa (viitattu 29.3. 2016): http://martinfowler.com/articles/microservices.html*.

[Fredrich2015] Fredrich, T. RESTful Service Best Practices Recommendations for Creating Web Services, 2015.

[Geza2017] NGE GEZA Global Entrepreneur Zone for All.Service Oriented Architecture (SOA). [online]. Available: http://www.ngegeza.com/SOA.htm. (Last visited 11.10.2017).

[Granell2009] Granell, C., Gould, M., & Esbrí, M. A. (2009). Geospatial web service chaining. *Handbook of Research on Geoinformatics. Information Science Reference*.

[Granli2015] Granli, W., Burchell, J., Hammouda, I., & Knauss, E. (2015, August). The driving forces of API evolution. In *Proceedings of the 14th International Workshop on Principles of Software Evolution* (pp. 28-37). ACM.

[Graphcool2017] Graphcool Blog. Relay vs Apollo. [Online]. Available:https://blog.graph.cool/relay-vs-apollo-comparing-graphql-clients-for-react-apps-b40af58c1534 (Last visited 24.10.2017)

[GQL2017] GraphQL community. How to GraphQL: The Full stack Tutorial for GraphQL. [Online]. Avaiable: https://www.howtographql.com/ (Last visited 25.10.2017)

[GQLorg2017] GraphQL. GraphQL: A query language for your API. [Online]. Availabe:www.graphQL.org/ (Last visited 25.10.2017)

[GQLspecs2016] Facebook. GraphQL: Working draft October 2016. [online]. Available: http://facebook.github.io/graphql/October2016/#sec-Language.Fields (Last visited 25.10.2017)

[Gupta2017] Shivangi Gupta. Introduction to GraphQL – A Query Language for APIs. [Online]. Available: https://blog.knoldus.com/2017/09/03/introduction-to-graphql-a-query-language-for-apis/ (Last visited 25.09.2017)

[Haupt2014] Haupt, F., Fischer, M., Karastoyanova, D., Leymann, F., & Vukojevic-Haupt, K. (2014, September). Service composition for REST. In *Enterprise Distributed Object Computing Conference (EDOC), 2014 IEEE 18th International* (pp. 110-119). IEEE.

[Haupt2015] Haupt, F., Leymann, F., &Pautasso, C. (2015, May). A conversation based approach for modeling REST APIs. In *Software Architecture (WICSA), 2015 12th Working IEEE/IFIP Conference on* (pp. 165-174). IEEE.

[Haupt2017] Haupt, F., Leymann, F., Scherer, A., & Vukojevic-Haupt, K. (2017, April). A Framework for the Structural Analysis of REST APIs. In *Software Architecture (ICSA), 2017 IEEE International Conference on* (pp. 55-58). IEEE.

[Hunter II2017] Hunter II, T. (2017). HTTP API Design. In *Advanced Microservices*(pp. 13-54). Apress.

[IBM2017] IBM Knowledge Centre. Structure of a SOAP message. [online].Available:https://www.ibm.com/support/knowledgecenter/en/SSGMCP_5.3.0/com.ibm.cics.ts.webservices.doc/concepts/soap/dfhws_message.html (Last visited 20.09.2017)

[IBM2016] Daya, S., Van Duy, N., Eati, K., Ferreira, C. M., Glozic, D., Gucer, V., & Narain, S. (2016). *Microservices from Theory to Practice: Creating Applications in IBM Bluemix Using the Microservices Approach*. IBM Redbooks

[Jojow2017] Johannes Wettinger .any2api. [Online] . Available: https://github.com/any2api/any2api.github.io/blob/master/legacy/home.md (Last visited 21.10.2017)

[Johannes2016] Johannes Wettinger “APIfication using any2api,” 2016. https://medium.any2api.org/apification-using-any2api-69265dcafb0d

[Kharenko2015] Anton Kharenko. Monolithic vs. Microservices Architecture. [online]. Available: https://articles.microservices.com/monolithic-vs-microservices-architecture-5c4848858f59 (Last visited 11.10.2017)

[Kristsov2016] Roman Kristsov. *Moving existing API from REST to GraphQL.* [online].Available:https://medium.com/@raxwunter/moving-existing-api-from-GraphQL-REST-205bab22c184 (Last visited 27.07.2017)

[Lane2013] LANE, K. API 101, 2013. *E-book retrieved from< https://s3. amazonaws. com/kinlane-productions/whitepapers/API+ Evangelist+-+ API*, *101*.

[Luscher2016] Steven Luscher. *Wrapping a REST API in GraphQL*.[online]. Available: *http://graphql.org/blog/rest-api-graphql-wrapper/* (Last visited 27.07.2017)

[Masse2011] Masse, M. (2011). *REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces*. "O'Reilly Media, Inc.".

[Nilan2016] NilanMarktanner *Declarative Data Fetching with GraphQL.* [online]. Available:https://css-tricks.com/declarative-data-fetching-graphql/ (Last visited 13.10.2017)

[OAI2017] Open API Initiative. [Online]. Available: https://github.com/OAI/OpenAPI-Specification (Last visited 25.10.2017)

[Oracle2017] Oracle® Fusion Middleware. *How API Gateway interacts with existing infrastructure.*[online]. Available:https://docs.oracle.com/cd/E55956_01/doc.11123/administrator_guide/content/admin_existing.html (Last visited 09.10.2017)

[Papazoglou2008] Papazoglou, M. (2008). *Web services: principles and technology*. Pearson Education.Petstore service provider based on swagger.

[Petstore] Swagger community. [Online]. Available: http://petstore.swagger.io/v2 (Last visited 27.10.2017)

[PetstoreClient] Petstore client application based on REST. [Online]. Available: http://petstore.swagger.io (Last visited 27.10.2017)

[Pepple2011] Pepple, K. (2011). *Deploying openstack*. “O’Reilly Media, Inc.". Pp 1-15.

[Point2017] TutorialsPoint. Web services Architecture. [Online]. Available:https://www.tutorialspoint.com/webservices/web_services_architecture.htm (Last visited 27.08.2017)

[Richards2015] Richards, M. (2015). Microservices vs. service-oriented architecture.

[Richardson2013] Richardson, L., Amundsen, M., & Ruby, S. (2013). *RESTful Web APIs: Services for a Changing World*. "O'Reilly Media, Inc.".

[Richardson2016] Richardson, C., & Smith, F. (2016). Microservices from Design to Deployment. NGINX.

[Rudrakshi2014] Rudrakshi, C., Varshney, A., Yadla, B., Kanneganti, R., &Somalwar, K. (2014). *API-fication-core building block of the digital enterprise*. Technical report, HCL Technologies.

[Stubailo2016] SashkoStubailo. *GraphQL: The next generation of API design*.[Online]. Available:https://dev-blog.apollodata.com/graphql-the-next-generation-of-api-design-f24b1689756a (Last visited 11.09.2017)

[Stylos2006] Stylos, J., Clarke, S., & Myers, B. (2006). Comparing API design choices with usability studies: A case study and future directions. In *Proceedings of the 18th PPIG Workshop*.

[Stylos2007] Stylos, J., & Myers, B. (2007, September). Mapping the space of API design decisions. In *Visual Languages and Human-Centric Computing, 2007. VL/HCC 2007. IEEE Symposium on* (pp. 50-60). IEEE.

[Tapang2001] Tapang, C. C. (2001). Web Services Description Language (WSDL) Explained. *Microsoft Developer Network*.

[Stubailo2017] Meteor Development Group, Inc.Swapi–*Rest-GraphQL*.[Online]. Available: https://github.com/apollographql/swapi-rest-graphql (Last visited 12.06.2017)

[Umer2009] Umar, A., & Zordan, A. (2009). Reengineering for service oriented architectures: A strategic decision model for integration versus migration. *Journal of Systems and Software*, *82*(3), 448-462.

[Umer2010] A. Umar, “Enterprise Architectures and Integration for Strategic IS Planning,” extracted from the book enterprise Architectures and Integration Using SOA.2010.

[Wachter2016] Otto von Wachter. *REST API downfalls and dawn of GraphQL.* [Online]. Available: https://medium.com/@ottovw/rest-api-downfalls-and-dawn-of-graphql-dd00991a0eb8 (Last visited 21.06.2017)

[Webber2010] Webber, J., Parastatidis, S., & Robinson, I. (2010). *REST in practice: Hypermedia and systems architecture*. “O’Reilly Media, Inc.".

[Wettinger2014] Wettinger, J., Binz, T., Breitenbücher, U., Kopp, O., Leymann, F., & Zimmermann, M. (2014, April). Unified Invocation of Scripts and Services for Provisioning, Deployment, and Management of Cloud Applications Based on TOSCA. In *CLOSER* (pp. 559-568).

[Wettinger2015] Wettinger, J., Breitenbücher, U., & Leymann, F. (2015). Any2API-Automated APIfication. In *Proceedings of the 5th International Conference on Cloud Computing and Services Science. SciTePress*.

[Wishworks2015] WishWorks™, “.*SOA Service Registry/Repository and Service Discovery.*[Online].Available: http://www.whishworks.com/blog/digital transformation/soa-service-registryrepository-and-service-discovery (Last visited 11.10.2017)

[W3C2001] Christensen, E., Curbera, F., Meredith, G., &Weerawarana, S. (2001). Web services description language (WSDL) 1.1.

[W3C2004] World Wide Web Consortium (W3C). *Web Services Architecture.* [Online]. Available:https://www.w3.org/TR/ws-arch/ (Last visited 19.09.2017)

[W3C2003] Mitra, N., & Lafon, Y. (2003). Soap version 1.2 part 0: Primer. *W3C recommendation*, *24*, 12.

[Yarax2016] *Swagger-to-GraphQL.* [Online]. Available:https://github.com/yarax/swagger-to-graphql (Last visited 22.08.2017)

Declaration:

I certify that this work contains no material which has been accepted for the award of any other degree or diploma in my name, in any university or other tertiary institution and, to the best of my knowledge and belief, contains no material previously published or written by another person, except where due reference has been made in the text.

In addition, I certify that no part of this work will, in the future, be used in a submission in my name for any other degree or diploma in any university or other tertiary institution without the prior approval of the University of Stuttgart.

Signature: Eyob Semere

Stuttgart