MASTER THESIS

Thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Engineering at the University of Applied Sciences Technikum Wien - Degree Program Software Engineering

Using REST and GraphQL as IPC Mechanism in a Microservices Architecture

By: Markus Hösel, BSc

Student Number: 1510299022

Supervisors: Oliver Wana, MSc
Christian Hager, MSc

Vienna, May 18, 2017

FH University of Applied Sciences
TECHNIKUM WIEN

Declaration

“As author and creator of this work to hand, I confirm with my signature knowledge of the relevant copyright regulations governed by higher education acts (for example see §§21, 42f and 57 UrhG (Austrian copyright law) as amended as well as §14 of the Statute on Studies Act Provisions / Examination Regulations of the UAS Technikum Wien).

I hereby declare that I completed the present work independently and that any ideas, whether written by others or by myself, have been fully sourced and referenced. I am aware of any consequences I may face on the part of the degree program director if there should be evidence of missing autonomy and independence or evidence of any intent to fraudulently achieve a pass mark for this work (see §11 para. 1 Statute on Studies Act Provisions / Examination Regulations of the UAS Technikum Wien).

I further declare that up to this date I have not published the work to hand nor have I presented it to another examination board in the same or similar form. I affirm that the version submitted matches the version in the upload tool.“

Vienna, May 18, 2017 Signature

Kurzfassung

Das Ziel dieser Thesis ist REST und GraphQL als Inter-Prozess-Kommunikationsmechanismen (IPC-Mechanismen) zwischen Microservices zu evaluieren. Für diesen Zweck wird ein bereits existierendes Projekt welches eine RESTful API implementiert in drei Microservices geteilt, wobei jedes dieser Microservices einen bestimmten Task erfüllt. Diese Microservices werden anschließend geklont und der REST API Layer wird mit einen GraphQL API Layer ausgetauscht um eine zweite Microservice Architektur zu erhalten welche auf GraphQL basiert. Beide Architekturen werden dann hinsichtlich ihrer Performance und ihren Eigenschaften bezüglich Schemavalidierung untersucht. Des weiteren werden Swagger und GraphiQL, welche als die populärsten Developer Tools gelten, analysiert.

Die Ergebnisse weisen mehrere interessante Punkte auf. Hinsichtlich Performance zeigt GraphQL aufgrund der implementieren Abfragesprache teilweise beträchtliche Vorteile im Vergleich zu REST. Vorallem in Fällen wo GraphQL mehrere Resourcen innherhalb eines Requests abrufen kann während REST hierfür mehrere Requests benötigt wird ein wesentlicher Performanceunterschied zugunsten von GraphQL erreicht.

Ein weiterer Vorteil von GraphQL ist das Type System welches unter anderen dazu verwendet wird die Payload eines Requests zu validieren. Im Vergleich dazu spezifiziert REST als Architekturstil weder ein Type System noch eine Validierung. Diese Thesis greift aufgrund dessen auf eine externe Bibliothek mit den Name Joi zurück, welche sowohl für eine konservative als auch für eine liberale Validierung einer Payload verwendet werden kann. GraphQLs Validierung im Gegenzug lässt sich standardmäßig nicht anpassen und kann als konservativ angesehen werden. Diese Einschränkung hat auch gewisse Auswirkungen auf die Microservices, da eine konservative Validierung eine Payload konform zum Schema erwartet während eine liberale Validierung in bestimmten Fällen auch eine Payload mit Abweichungen zum Schema als gülltig validiert, was Voraussetzung ist um das Robustheitsprinzip implementieren zu können. Um nun eine liberale Validierung mit GraphQL durchzuführen wurden Änderungen an der Bibliothek selbst vorgenommen, durch welche es ermöglicht wird das Robustheitsprinzip auch unter GraphQL zu implementieren.

Die Analyse der Developer Tools zeigt das sowohl Swagger für RESTful APIs als auch GraphiQL für GraphQL Endpunkte eine automatische API Dokumentation generieren und weiters für das Testen der API verwendet werden können. Im Gegensatz zu Swagger ermöglicht GraphiQL jedoch nicht das Setzen eines Tokens um authentifizierte Requests durchzuführen. Es wurden daher Änderungen an der Bibliothek selbst durchgeführt um dies zu ermöglichen.

Schlagworte: Microservices, Monolith, REST, GraphQL, IPC Mechanismus

Abstract

The goal of this thesis is to evaluate REST and GraphQL as inter-process communication (IPC) mechanism between microservices. For this purpose an existing project which already implements a RESTful API is divided into three microservices with each of those microservices fulfilling one particular task. These microservices are cloned and the REST API layer is replaced with a GraphQL API layer in order to form a second microservices architecture based on GraphQL. Both architectures are then analyzed in respect to their performance and their capabilities to perform schema validation. In addition Swagger and GraphiQL - which are the most popular developer tools for RESTful APIs and GraphQL respectively - are investigated.

The outcomes show several interesting findings. In terms of performance GraphQLs query capabilities can be a huge advantage compared to REST. While its overall response time is slightly lower compared to the RESTful API, GraphQL especially outperforms REST in certain cases as the number of necessary requests can be reduced significantly.

Another advantage of GraphQL is its type system which is used to validate the payload of each request. In contrast REST as an architectural style does not specify any type system nor any validation process. This thesis therefore uses Joi as external library to support validation of payloads within a REST environment. The findings show that Joi can be used to implement a conservative as well as a liberal validation strategy while GraphQL enforces a conservative validation of the payloads. This has a major impact on microservices architectures due to the reason that the conservative validation strategy enforces the payload to be conform with a defined schema whereas the liberal validation strategy allows the payload to diverge from the defined schema in order to fulfill the robustness principle. However, by modifying the GraphQL library itself the thesis proofs that GraphQL can be adapted to enforce a liberal validation of the payloads as well.

The investigation on the developer tools shows that Swagger as well as GraphiQL can be used to auto generate a documentation for the developed APIs and to consume those APIs. Swagger in addition also allows to set an authentication token in order to perform authenticated requests while GraphiQL by default has no such option. Necessary changes to the GraphiQL library are therefore performed in order to add support for authenticated requests to this library as well.

Keywords: microservices, monolith, REST, GraphQL, IPC mechanism

Acknowledgements

I want to thank aaa - all about apps GmbH for giving me the opportunity to use their cloud infrastructure in order to perform measurements necessary to scientifically document results, Christian Hager, MSc for his support as technical supervisors, Oliver Wana, MSc for his support as organizational supervisor and DI Mario Ranftl, BSc for his technical advices.

Contents

1 Introduction

Microservices describes a way to design and deploy applications in a loosely coupled way [1]. Its main difference compared to monolithic applications is the division of a software into a suite of small, independent services. These services run on their own process - mostly on their own machine - and communicate with each other using inter-process communication methods and well defined APIs [5] [3].

Dividing a single software product into multiple individual microservices has several advantages, such as strong modularization, easy replaceability, independent scaling, free choice of technology and continuous delivery. However, such a distributed system also has challenges. The overall architecture is more complex compared to a single monolith and a focus on the domain architecture must be given to ensure the independent development of such services [4]. When setting up a microservices architecture several decisions are necessary. These decisions include the inter-process communication (IPC), service discovery, data management and deployment strategies [2].

This thesis is written for aaa - all about apps GmbH, a software company particularly interested in microservices because of their scalability and their independent deployment possibilities when using a continuous delivery pipeline. As a company with a focus on mobile applications aaa - all about apps GmbH has a lot of experience on REST for client server communication and is currently working on the integration of GraphQL for client server communication as well. REST is also a popular protocol for the communication between microservices and is used by several microservice implementations such as [6] and [7]. In terms of GraphQL, a new library which quickly becomes more popular [8], no scientifically documented results as IPC mechanism have been published yet. The company therefore has a special interest on a comparison of REST and GraphQL as IPC methods within microservices architectures.

The goal of this thesis is to compare REST and GraphQL as IPC mechanism in terms of performance, schema validation and developer tools. For this task an already existing project is first split into three individual services which make use of a RESTful API for the inter-process communication. Figure 1 outlines the services and the inter-process communication between those services. The three services which are implemented are a user service which manages all user related tasks such as registration and authentication, a project service which manages project related tasks and an API gateway which acts as a single entry point to the microservices architecture and distributes the requests from the client to either the user service or the project service. The project service in addition implements Githubs REST API [16] in order to use Github as a third party.

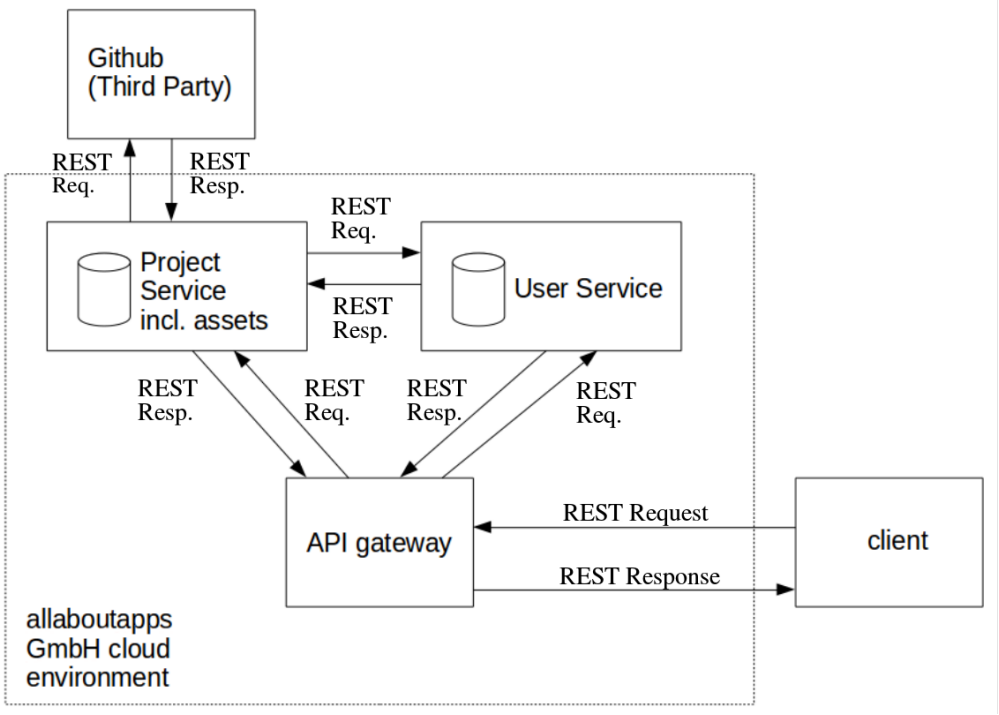

Figure 1: Microservices architecture with REST as IPC mechanism

The microservices architecture outlined in Figure 1 is then cloned and REST as IPC mechanism is replaced with GraphQL in order to form a second microservices architecture. This is illustrated in Figure 2.

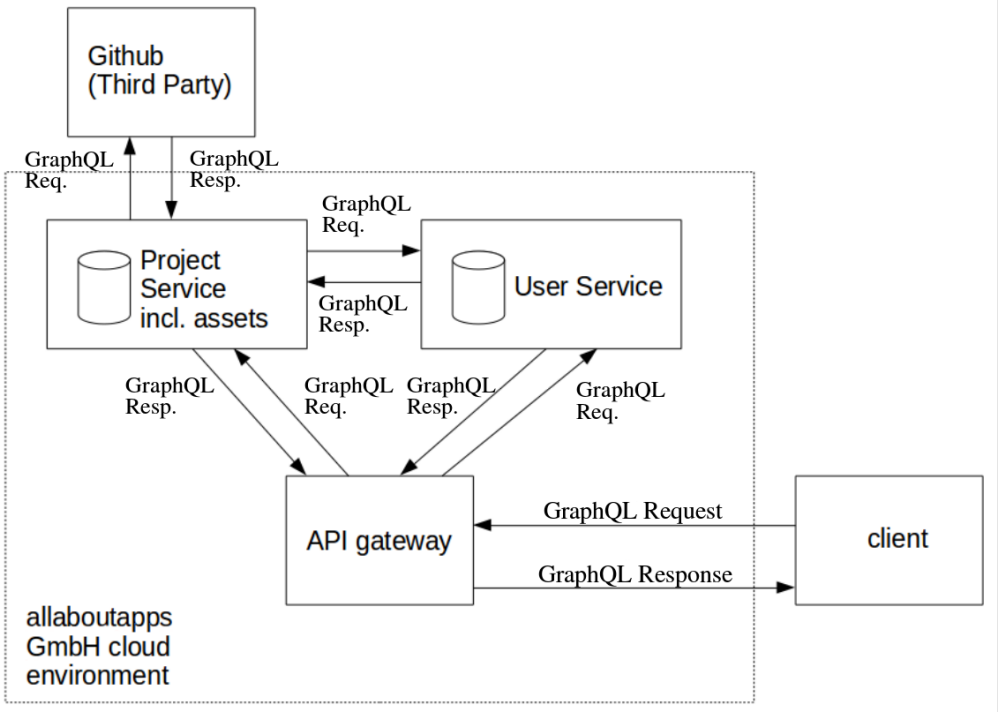

Figure 2: Microservices architecture with GraphQL as IPC mechanism

2 Basics

2.1 Monolithic Applications

Most common programming languages, such as Java, C++ or Python allow to break down the complexity of programs by organizing the code into components, such as classes and methods. During execution time the classes are instantiated and method calls are performed. Despite the separation into classes and methods the program still shares the resources of the underlying machine, such as the memory, databases and files [9] and is packaged and deployed as one self-contained application. Monolithic application have many advantages. For instance, they are simple to develop - the complexity of their lightweight and fast communication methods are hidden behind method calls and variables - and can easily be deployed by copying the packaged application to a server [2]. By running multiple instances of the application behind a load balancer scaling can be achieved as well - at least for the early stage of the project. The issue with scaling a monolith is that only the whole application can be instantiated, while individual requirements of components - such as CPU power, memory consumption or high I/O rates - cannot be addressed [10]. Other disadvantages are the deployment - changing one line of code results in a deployment of the whole application - or the dependency of previously chosen technologies [4].

2.2 Microservices

Microservices are independent services that work together and form a distributed system. Each of this microservice is a small, autonomous entity which fulfils one particular task [11]. One way of breaking down a system into tasks is by following the Domain-Driven-Design (DDD) pattern, where each microservice models a particular business domain. When modelling microservices according to the DDD pattern the bounded context should be respected too. An analogy often used when describing the bounded context are biological membranes which are built out of individual cells. While each cell defines a specific task the membrane defines external and internal boundaries [4] [11]. As an example, a given company A could have a finance department and a logistic department - two bounded contexts which communicate through external interfaces. However, each of this department exists of several domains (for the logistic department e.g. delivery, orders, inventory ...) which are internal to that bounded context. Specifying external and internal interfaces is critical to create a maintainable microservices architecture.

Microservices have several key benefits. Technological heterogeneity allows to use different

technologies in each microservice. While one microservice may be programmed in Java and may use a MySQL database another microservice may be programmed in Python and may use a document store such as MongoDB [2] [4] [11]. Microservices also allow individual scaling, according to their requirements some could be placed on powerful hardware while others could be placed on cheaper hardware. Disadvantages of microservices may include performance - calls are executed over the network - and the complexity of distributed systems [3].

2.3 IPC Mechanism

Monolithic applications use language-level methods for communication whereas microservices form a distributed system with each microservice running in its own process. In order to be able to communicate with each other an IPC mechanism is needed. Several different interaction styles exist, such as one-to-one or one-to-many communication methods and synchron or asynchron message handling [2]. This thesis focuses on REST and GraphQL which belong to the one-to-one category and use a synchronous interaction style. More specifically, REST and GraphQL follow the request-response model, where a client makes a request and waits for a response.

2.4 HTTP

This thesis researches REST and GraphQL as IPC mechanisms and uses the HyperText Transfer Protocol (HTTP) as underlying protocol for both technologies. As an application-level protocol for distributed and collaborative systems HTTP has not only be used by the World Wide Web but also for many other tasks such as microservices and fulfils therefore the technological purpose of this thesis [51] [52]. HTTP makes use of a Uniform Resource Identifier (URI) to exchange data between a client and a server. Such an URI encodes the protocol, the name of the server and the endpoint of the server within a string and is used by the client to identify the server. HTTP by default specifies several error codes but can be extended by custom error codes as well. However, HTTP does not specify the format of the data exchanged between client and server. For this work it was decided to make use of the JavaScript Object Notation (JSON) format due to its popularity and its lightweight semantic. Furthermore HTTP specifies multiple methods in order to indicate the serverside action which should be performed [52]. This methods are standardized by the RFC 2616, and are listed below [52]:

- OPTIONS: Used in a request to **retrieve** a list of **HTTP methods** available for a specific resource.
- GET: Used in a request to **retrieve the entity** identified by the Request-URI.
- HEAD: Identical to GET except that no message body is attached to the response.

- POST: Used in a request with an **entity** attached to the payload which the **server accepts as new subordinate**.
- PUT: Used in a request with an **entity** attached to the payload which the **server stores** under the Request-URI. In case the entity is new a 201 (Created) response code should be returned. In case the entity already exists and an update was performed a 200 (OK) or 204 (No Content) response code should be returned. While the Request-URI in a POST request identifies a resource to handle the entity attached to the request, the Request-URI in a PUT requests identifies the entity attached to the request itself.
- DELETE: Used in a request to **delete the entity** identified by the Request-URI.
- TRACE: Used in a request to invoke a remote loopback of the request message. This method is commonly used for debugging.
- CONNECT: Reserved for use with a proxy which can dynamically be switched to operate as tunnel.

In addition to these HTTP methods RFC 2068 - a previous version to RFC 2616 - also defined PATCH, LINK and UNLINK as HTTP methods [53]. HTTP is a stateless request/response protocol. In order to identify a user throughout multiple requests a JSON Web Token (JWT) described in Section 2.5 will be used throughout this work.

2.5 Bearer Tokens and OAuth 2.0

The microservices implemented for this thesis use so called bearer tokens in HTTP requests in order to allow a client to access OAuth 2.0 protected resources. OAuth 2.0 specifies a framework through which an application is enabled to obtain access to HTTP resources [55]. In this work the necessary authorization server and resource server to implement OAuth 2.0 are both located on the microservice responsible for the user management. Through this microservice a user can retrieved a token for example by making a login or registration request. Once the user retrieved a token he or she can further use this token to request protected resources [57]. Figure 3 illustrates the authentication and request flow using a sequence diagram. In order to access a protected resource the token is set in the *Authorization* header field of the HTTP request [54]. Because the token can be used by anyone, it is important to prevent its misuse by protecting the token from disclosure in storage - on the server as well as on the client - and in transport. Listing 2.1 shows an example of the *Authorization* header field.

```
"Authorization": "Bearer 569c8eff-2bda-4c4c-a1f6-67095938be9d"
```

Listing 2.1: GraphQL query example

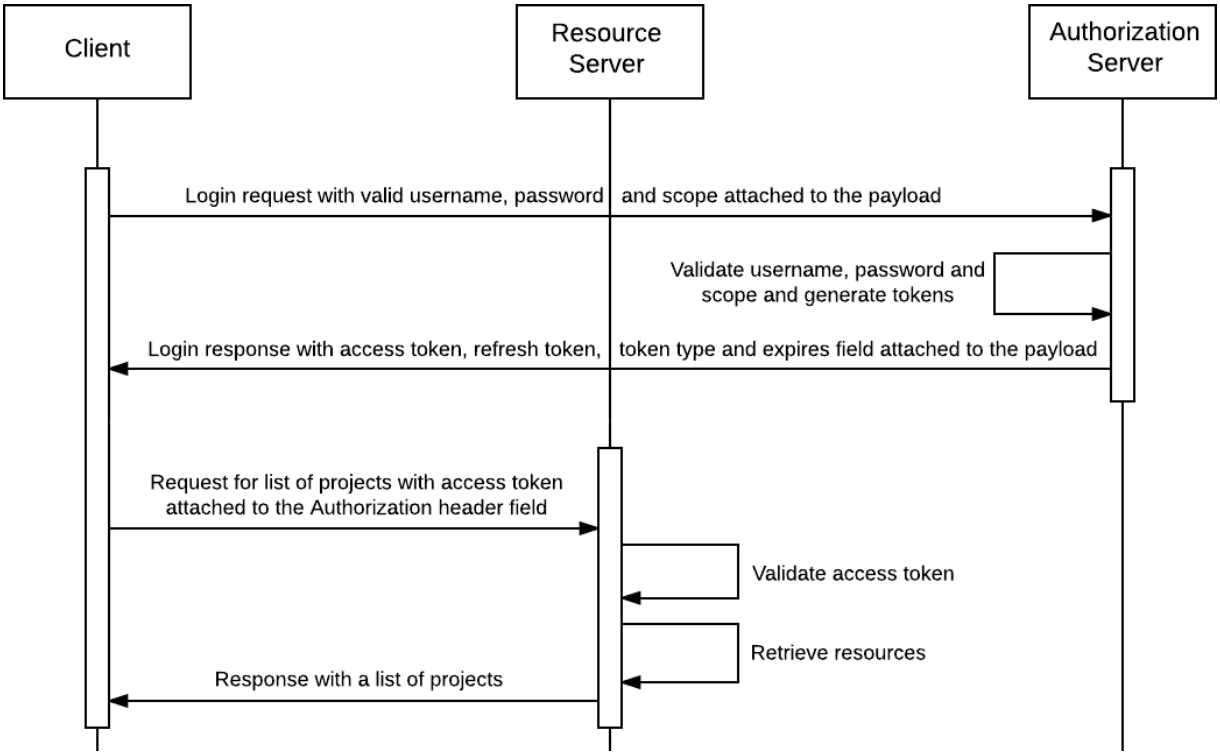

Figure 3: Authentication flow with a valid login credentials

The token used in Listing 2.1 was generated on the server due to a *login* request from a client. This *login* request was sent with a *username* and a *password* attached to the payload in order to authenticate the user. The response of this *login* request has been in the format of a "OAuth 2.0 access token response" [54] shown in Listing 2.2 and contains the properties *accessToken*, *refreshToken*, *tokenType* and *expiresIn*.

```
"tokenType":"Bearer",
"accessToken":"569c8eff-2bda-4c4c-a1f6-67095938be9d",
"refreshToken":"f2d927ff-fddc-4408-9b94-ac116f7bf1f8",
"expiresIn":86400000
```

Listing 2.2: Bearer Token

The *accessToken* is used to access OAuth 2.0 protected resources on the server. This is the same token which was set in the header field demonstrated in Listing 2.1. The *refreshToken* is used to obtain a new access token once the currently active *accessToken* expires. The *expiresIn* field contains the time in seconds for how long the *accessToken* will be valid. The *tokenType* further specifies the kind of token [55]. Tokens used throughout this thesis are always of type *Bearer* and are of the Universally Unique Identifier (UUID) v4 format specified in [56].

2.6 REST

Represential State Transfer (REST) defines an architectural style for distributed hypermedia systems with resources being the key abstraction of information. Such a resource can technically be any information that can be named [12]. REST is introduced by Roy Thomas Fielding in his work with the title "Architectural Styles and the Design of Network-based Software Architectures" which describes REST as a set of constraints applied to components within an architecture. This set of constraints and their order in which they are applied to the components of an architecture is derived in [12] and is summarized in the following listing:

1. Null Style: This style represents an **empty set of constraints** and is the starting point for deriving REST. Components within this style do not have any distinguishable boundaries.
2. Client Server: Through this constraint separation of concern is achieved by **splitting the components** of an architecture into **clients** and **servers**. While the server is responsible for the data the client provides the user interface to interact with the server and thus with the data.
3. Stateless: This constraint states that **communication** between the components **must be stateless**. Therefore each request performed must contain all information necessary so that the server can process the request.
4. Cache: By adding caching functionality the network efficiency is improved.
5. Uniform Interface: The uniform interface constraint to **emphasis on a uniform interface between components** is described as the central feature which differentiates REST from other network-based styles. This constraint is itself based on four interface constraint:
   - identification of resources: each resource can be **identified** through a **unique identifier**, such as a Uniform Resource Identifier (URI).
   - manipulation of resources through representations: **clients only interact with a representation** of the resource, but not with the resource itself. The resource could for example be represented in the XML or JSON format.
   - self-descriptive messages: such message may include information about the state of the resource, the format and the resource itself.
   - hypermedia as the engine of application state (HATEOS): a resource is represented with **links to related resources**.
6. Layered System: This constraint allows an architecture to exist of multiple layer, wherein **each layer can only interact with the immediate layer**. A common example is the three-tiered architecture composed of a presentation tier, an application tier and a data tier. Another example would be a microservices architecture in which servers are organized in layers offering distinct services.

7. Code-On-Demand: Through this constraint the client can download and execute applets or scripts. While this constraint improves extensibility it also reduces visibility and is therefore an optional constraint.

While REST as an architectural style is not coupled to any underlying communication protocol in specific it is often used in combination with the Hypertext Transfer Protocol (HTTP) in order to retrieve and manipulate resources through its uniform interface constraint [13]. REST in combination of HTTP reuses the HTTP methods defined in Section 2.4 to retrieve and manipulate resources. While their initial intention is kept the definitions vary slightly [58]:

- GET: Used in a request to **retrieve** a representation of a resource identified by the Request-URI
- HEAD: Identical to GET except that no message body is attached to the response
- PUT: Used in a request to **create** or **update** a resource identified by the Request-URI. A representation of this resource is attached to the payload
- POST: Used in a request to **insert** a resource **within a collection** identified by the Request-URI. A representation of this resource is attached to the payload
- DELETE: Used in a request to **delete** the resource identified by the Request-URI

Section 2.7 introduces a framework named *Hapi* which is used for the microservices developed for this thesis to offer an API based on the REST architectural style.

2.7 Hapi

Hapi (pronounced "Happy") is a web framework based on NodeJS which enables the developer to build web application and RESTful APIs [21]. While REST as an architectural style was introduced in Section 2.6 the term "RESTful" simply refers to an API which is implemented according to the REST architectural style. Hapi itself is based on NodeJS, a JavaScript runtime which is built on top of Chrome's V8 engine in order to run JavaScript on a server [20]. The RESTful API designed within the Hapi framework represents the resources in JSON format. In order to create a RESTful API the first thing to do is to create a Hapi server object and to specify a port the server is listening on. Listing 2.3 demonstrates the instantiation of a Hapi server listening on port 8081.

```
const server = new Hapi.Server();
this.server.connection({
   port: 8081
});
```

Listing 2.3: Creating a Hapi server object

This server object can then be used to configure the server for authentication. In order to configure the server to use Bearer tokens introduces in Section 2.5 a plugin called "hapi-auth-bearer-token" is used [22]. Through this plugin a authentication strategy can be setup which uses a *validateFunc* to validate a token. Listing 2.4 shows the configuration, in which a function named *generateBearerValidateFunction* defined on the *storage.models.User* is executed in order to validate the token.

```
this.server.auth.strategy("default-authentication-strategy",
    "bearer-access-token", true, {
   validateFunc: function(token, callback) {
      return storage.models.User.
         generateBearerValidateFunction(storage.models)(token, callback);
   }
});
```

Listing 2.4: Configure a Hapi server authentication strategy

The Hapi server object can further be used to define APIs - which are also referred to as routes. Listing 2.5 outlines the definition for a route which retrieves a *project* object by using a *projectId* as identifier.

```
{
   method: "GET",
   path: "/app/v1/project/{projectId}",
   handler: getProject,
},
```

Listing 2.5: Configure a route within Hapi

The route object comprises at least a *method*, a *path* and a *handler*. In Listing 2.5 "GET" is used as method in order to retrieve a representation of a resource, "/app/v1/project/projectId" is set as path and the function *getProject* is used as handler. Listing 2.6 demonstrates a query to the route definition of Listing 2.5 implemented in Python. This request performs a "GET" request by executing the *get* function on the *self.client* object and passes the path "/app/v1/project/4f1ff2cf-a159-4ddf-bbb8-7e38fb12f431" to the function.

```
   def get_project(self):
      res = self.client.get(
         "/app/v1/project/4f1ff2cf-a159-4ddf-bbb8-7e38fb12f431"
      )
```

Listing 2.6: Performing a request to a route defined within Hapi

The *getProject* function which is registered as handler in Listing 2.5 can then access the *projectId* by extracting it from the *path*. The function further executes the business logic and returns a response with the payload shown in Listing 2.7. This response which is in format

"JSON" contains the requested *project* composed of the fields *uid*, *userUid*, *name*, *createdAt*, *updatedAt* and the field *ProjectLanguages* which further contains an array of object expressing the *Languages* used for this particular project.

```
{
  "uid": "4f1ff2cf-a159-4ddf-bbb8-7e38fb12f431",
  "userUid": "74d7245d-50c9-45a1-835f-91417336d197",
  "name": "masterthesis",
  "createdAt": "2017-04-15T09:26:26.000Z",
  "updatedAt": "2017-04-15T09:26:26.000Z",
  "ProjectLanguages": [
    {
      "uid": "e9190242-480d-45d1-8d16-efeedcedbc28",
      "name": "JavaScript",
      "createdAt": "2017-04-15T09:26:42.004Z",
      "updatedAt": "2017-04-15T09:26:42.004Z",
      "ProjectUid": "4f1ff2cf-a159-4ddf-bbb8-7e38fb12f431"
    },
    {
      "uid": "1ed20b15-12b2-4061-a5e8-043c03082674",
      "name": "Python",
      "createdAt": "2017-04-15T09:26:46.916Z",
      "updatedAt": "2017-04-15T09:26:46.916Z",
      "ProjectUid": "4f1ff2cf-a159-4ddf-bbb8-7e38fb12f431"
    }
  ]
}
```

Listing 2.7: Hapi route response

2.8 GraphQL

GraphQL is a specification introduced by Facebook [31] in July 2015 and describes itself as "A query language for your API" [18]. It gives the client the opportunity to execute a query on an application server which has capabilities defined according to the specification [19]. At the time writing this thesis GraphQL is already implemented in all major programming languages, with a fast-growing ecosystem of libraries, tools, services and databases developed or adapted for GraphQL. [33] provides an overview of the ecosystem. For this thesis Facebooks open sourced reference implementation written in JavaScript [17] is used. GraphQL is based on several design principles [32]:

- Hierarchical: Queries in GraphQL are **structured hierarchically** in order to achieve congruence with modern web application which are often based on view hierarchies. Such a

view hierarchy can be seen as an inverted tree structure, with one view containing multiple subviews [34].

- Product-centric: The GraphQL language and runtime is based on the **thinking and requirements of views**.
- Strong-typing: Each GraphQL application server defines a **type system**. This enables **syntactically correctness** of a given query at development time and allows to gather information about available types.
- Client-specified queries: The GraphQL application server publishes the defined capabilities and gives the **client** the **responsibility** to ask exactly for only those capabilities necessary to fulfil the request.
- Introspective: **Introspection allows to query the capabilities** defined on the GraphQL application server.

In contrast to a RESTful API which makes use of several HTTP methods in order to define the action to be performed, GraphQL only makes use of the "POST" method. In addition GraphQL only uses one path mostly named "/graphql" and encodes all further information in the query itself. Figure 2.8 shows an example of a GraphQL query. Such a query is also referred to as document and can contain operations and fragments. The two defined operations in GraphQL are query and mutation. The query operation is used to retrieve data from the GraphQL application server whereas the mutation operation is used to manipulate data on the GraphQL application server for example by creating, updating or deleting an entity or some fields on this entity. A fragment is used as a common unit of composition which can be reused in several queries. Figure 2.8 defines a query to retrieve a *project* object composed by two fields, namely *uid* and *name*. These two fields form a so called selection set, are nested into the root field *project* and could have their own selection sets, allowing for a deeply nested query. In Figure 2.8 however *uid* and *name* are so called scalars, which form the leaves of the hierarchical query. Scalars can either be of type *GraphQLInt*, *GraphQLFloat*, *GraphQLString*, *GraphQLBoolean* or *GraphQLID*. Fields in GraphQL can further be wrapped by a so called *wrapper* type. The two wrapper types GraphQL defines are *GraphQLList* which represents a list of the wrapped type and *GraphQLNonNull* which represents a non-null version of the wrapped type. Fields can also accept arguments. As an example, the field *project* takes an argument *uid*. Depending on the server side implementation such arguments may or may not be optional. [32]

```
query{
   project(uid: "9d48edbe-eec0-414e-bf57-c703612662ae") {
      uid,
      name
   }
}
```

Listing 2.8: GraphQL query example

Figure 2.9 and Figure 2.10 outline the implementation of the GraphQL endpoint. While Figure 2.9 defines the *ProjectType*, Figure 2.10 implements an endpoint which references the *ProjectType*. It would also be possible to directly specify the *ProjectType* within the endpoint implementation without referencing it. However, defining the *ProjectType* separately and referencing it makes it possible to use it in multiple endpoint definitions.

```
const ProjectType = new GraphQLObjectType({
    name: "Project",
    fields: () => ({
        name: {
            type: GraphQLString
        },
        userUid: {
            type: GraphQLString
        },
        uid: {
            type: GraphQLString
        },
        createdAt: {
            type: GraphQLString
        },
        updatedAt: {
            type: GraphQLString
        },
        ProjectLanguages: {
            type: new GraphQLList(LanguageTyp)
        }
    })
});
```

Listing 2.9: GraphQL type definition

The *ProjectType* in Listing 2.9 has six fields named *name*, *userUid*, *uid*, *createdAt*, *updatedAt* and *ProjectLanguages*. While the fields *name*, *userUid*, *uid*, *createdAt* and *updatedAt* are of type *GraphQLString*, the field *ProjectLanguages* is of another *GraphQLObjectType* named *LanguageType*. Furthermore this field is wrapped within the wrapper type *GraphQLList* to express that a *Project* has null, one or more *ProjectLanguages* associated with it.

The *ProjectType* is then referenced in the endpoint definition outlined in Listing 2.10. Doing so exposes the *ProjectType* with its six fields *name*, *userUid*, *uid*, *createdAt*, *updatedAt* and *LanguageType* to the client, who can decide which fields should be included in the response. In addition the endpoint definition in Listing 2.10 specifies an argument named *uid* of type *GraphQLString* which is not allowed to be null. This argument is then used by the *resolver* function, which also registers a function named *getProject* as handler and passes the argument to that function in order to retrieve the project from the database. The handler function is

furthermore exactly the same as in the RESTful example illustrated in Listing 2.5.

```
            project: {
                type: ProjectType,
                args: {
                    uid: { type: new GraphQLNonNull(GraphQLString) }
                },
                resolve: (value, { uid }) => {
                    return getProject(uid);
                }
            },
```

Listing 2.10: GraphQL endpoint definition

Listing 2.11 illustrates how to implementing the query specified in Listing 2.8 using Python in order to fetch data from the endpoint implemented in Figure 2.9 and 2.10. This Python snippet defines a function named *get_project* in which an HTTP request to the GraphQL server is performed using the HTTP method "POST" and querying the path "/graphql", attaching the query as payload to the request.

```
def get_project(self):
   res = self.client.post("/graphql",{
      "query": "query{\
         project(uid:\"9d48edbe-eec0-414e-bf57-c703612662ae\"){\
            uid,\
            name\
         }\
      }"
   })
```

Listing 2.11: GraphQL Query Python implementation

Executing the Python snippet defined in Listing 2.11 on a client against the serverside implementation specified in Listing 2.9 and Listing 2.10 results in the output of Listing 2.12. The output is of format "JSON" and includes the *project* object with the two fields *uid* and *name*, which are exactly the fields which have been requested in the query.

```
{
   "data": {
      "project": {
         "uid": "9d48edbe-eec0-414e-bf57-c703612662ae",
         "name": "masterthesis"
      }
   }
}
```

Listing 2.12: GraphQL Response

2.9 Robustness Principle

The term robustness principle, which is also known as Postel's law, was formulated by Jon Postel in the year 1981 [35] and was initially stated for the Transmission Control Protocol (TCP). RFC793 - the Request for Comments publication for TCP - formulates that the TCP stack implements this principle, which basically states to "be conservative in what you do, be liberal in what you accept from others." [36]. The general idea of the principle - according to the TCP stack - is to accept any incoming datagram which can be interpreted, even if it is not consistent with the standard. However, if the incoming datagram can be interpreted in multiple ways because of ambiguity about the meaning or the datagram cannot be interpreted at all an error must be thrown [37]. Moreover, an outgoing datagram must be conform to the standard [35].

While the principle was initially formulated for TCP, it is also used in microservices. In [38] Martin Fowler references the robustness principle and states that systems which communicate with each other through interfaces should follow this principle in order to gain a higher level of decoupling. [38] also points out that one of the most difficult parts of collaborating services is evolution and that the robustness principle makes it simpler to make changes to those systems in order to support new demands while minimizing breaking changes to existing clients. In addition to the robustness principle the term "Tolerant Reader" is often used when stating that an API should be as tolerant as possible when processing incoming data. The term is introduced in [39] in order to design microservices with high availability while allowing changes with minimal breakage.

2.10 Schema Validation

The APIs of the microservices programmed for this thesis use schema validation in order to validate the payload of requests and responses. This ensures that the data format of the payload is structured in the expected manner and can be interpreted. In case of a request which does not fulfil the validation pattern defined on the server a HTTP error with status code 400 is returned to the client indicating that the request was malformed. In case of a response which does not fulfil the validation pattern a HTTP error with status code 500 is returned to the client indicating that the response was malformed. In both cases the validation error is attached as payload. The activity diagram in Figure 4 illustrates the validation process.

However, different ways to validate a payload exists. One way would be to validate a payload using a conservative validation strategy, thus returning an error in case the payload does not match the validation pattern. Another way is to be more liberal when retrieving data. For example several violations could be ignored as long as its possible to parse the payload. While this could be done in several ways one approach is to respect the robustness principle which was discussed in Section 2.9. This thesis therefore compares a conservative validation strategy with a more liberal validation strategy which follows the guidelines of the robustness principle.

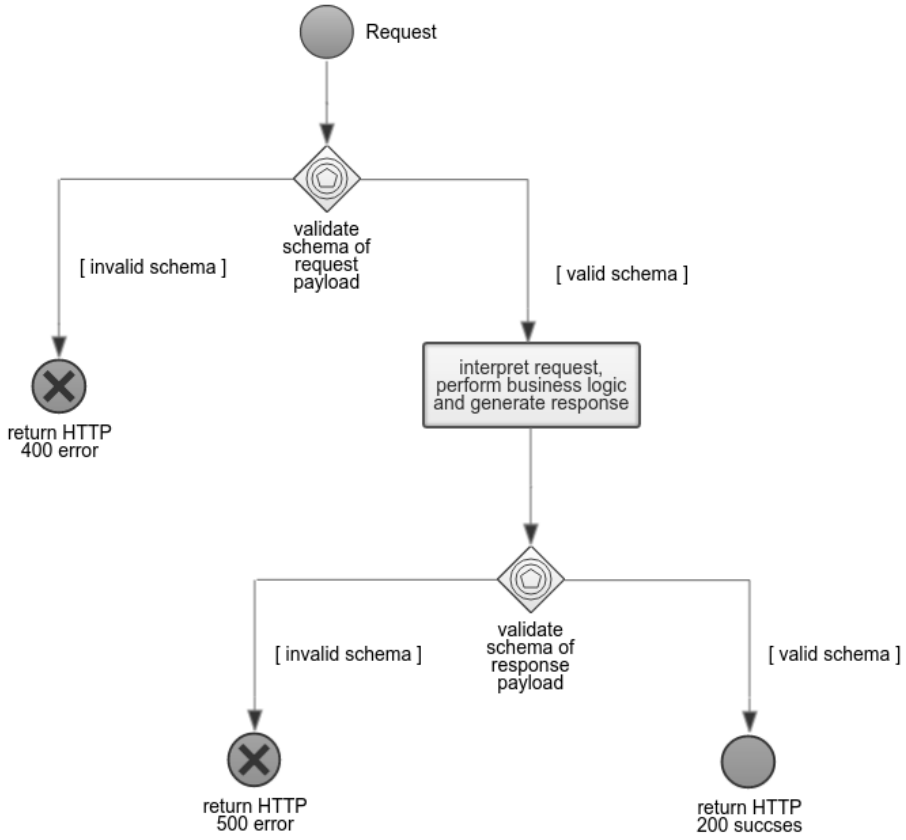

Figure 4: Activity Diagram for Schema Validation

Section 5.3.1 and Section 5.3.2 illustrate the necessary steps to add schema validation to the RESTful API as well as to the GraphQL endpoints.

3 Methods

This thesis has the purpose to transform a monolithic backend into a microservices architecture and to analyze the impact of REST and GraphQL as IPC mechanism. This impact is analyzed by evaluating the performance of both IPC mechanism, by examining the process of schema validation and by investigating the most popular developer tools for both mechanism.

3.1 Transforming a Backend into a Microservices Architecture

This thesis will first describe the process of transforming an existing backend into a microservices architecture. An already existing backend which provides a RESTful API will therefore be cloned three times to form a microservices architecture existing of three services and each of this clones will be refactored to provide a specific service. Each of this services will define its own RESTful API and will make use of the RESTful APIs offered by the other services in order to form one coherent distributed system. Because each of this services fulfils only one particular task, which itself does not exceed a certain size, they represent microservices and form the microservices architecture used within this thesis.

To implement GraphQL as IPC mechanism, the developed microservices are cloned and their RESTful API is substituted with GraphQL endpoints. Because both API layers use HTTP as transport protocol and have good support within the Hapi framework the integration this substitution can be made without affecting any of the existing business logic nor the authentication strategy.

3.2 Evaluating the Performance of the IPC mechanism

In order to evaluate the performance of both IPC mechanism a Python script based on Locust [24] is developed. This script acts as a client and executes requests in order to test the microservices. The script is executed on the same machine the API gateway is running in order to keep the latency between the client and the microservices low and to avoid additional response time. The API gateway acts as an intermediary and forwards the incoming request to the responsible microservice and vice versa. Using this API gateway makes it possible to abstract the complexity and offer clients a single system to query. To measure the performance Locust keeps track of several metrics, namely the request count, the median-, average-, min-

and max-values of the response time in milliseconds and the content size in bytes of the performed request. This metrics are then analyzed and discussed.

3.3 Evaluating the Process of Schema Validation

This thesis further examines the steps to add schema validation to both the RESTful API and the GraphQL endpoints. Schema validation itself is a process to validate the payload of requests and responses against a predefined schema. This process is by default not supported for RESTful APIs developed within Hapi, but can be added using a library named Joi [59]. In contrast, GraphQL endpoints are based on a type definition which is used to predetermine if a query is valid or invalid.

Because several different strategies on how to implement schema validation exists this thesis researches how one can apply necessary modification in order to support multiple strategies. In more detail a conservative validation strategy and a more liberal validation strategy which considers the robustness principle introduced in Section 2.9 are analyzed. For the endpoints configured with REST the necessary changes are made by refactoring the schema provided by Joi. For the endpoints implemented using GraphQL the GraphQL library itself is modified. This is because GraphQL by default only supports a validation strategy which can be seen as conservative and does not allow to change this strategy.

To validate the implemented schemas three different calls are executed using the command line tool *curl*. The first call is executed with a payload which matches exactly the defined schema. The second call is executed with a payload which contains a value of a type not defined on the schema and the third call is executed with a payload which contains a field not defined on the schema. In all cases the responses from the server is outlined and analyzed.

3.4 Analyzing Developer Tools

Developer tools are powerful tools to document, visualize and to interact with the defined API. This thesis analyzes the most popular developer tools for both REST and GraphQL, which are Swagger and GraphiQL respectively. The thesis outlines how one can auto-generate an API documentation, how this documentation can be manually extended and furthermore shows the steps to interact with the API. Because the microservices offer capabilities to perform authenticated calls the thesis especially examines how such authenticated calls can be performed within those tools. While Swagger offers the user to enter an authentication token which is then attached in the header field of the request in order to perform an authenticated call GraphiQL by default is lacking this feature. Therefore the integration of the GraphiQL plugin is refactored on the server in order to attach an Authorization header field. This change is then reflected within GraphiQL by adding an input field and a button which can be used by the user to enter an authentication token and to perform authenticated requests.

4 Environment

The microservices developed and investigated throughout this thesis are based on the template project used within aaa - all about apps GmbH to start new projects. Because of that a technology stack already exists. This stack is based on *CentOS* as operating system, *NodeJS* as runtime environment, *PostgreSQL* as database and makes use of several node packages. In addition *Locust* and GraphQL are added as dependencies. The following list outlines the technology stack:

- Operating system: CentOS Linux 7.3
- NodeJS environment: v6.9.4
- npm package manager: v3.10.3
- PostgreSQL: v9.4.10
- Typescript: v2.0.3
- npm packages: The following list outlines the most important npm packages in respect to this work.
  - bluebird: v3.3.3
  - boom: v4.2.0
  - cls-bluebird: v2.0.1
  - graphiql: v0.8.0
  - graphql: v0.7.2
  - graphql-custom-types: v0.7.1
  - graphql-server-hapi: v0.4.3
  - hapi: v14.2.0
  - hapi-auth-basic: v4.1.0
  - hapi-auth-bearer-token: v4.3.1
  - hapi-swagger: v7.4.0
  - joi: v9.2.0
  - lodash: v4.16.4
  - moment: v2.15.2
  - node-uuid: v1.4.3

  - pg: v6.1.0
  - pg-native: v1.10.0
  - prettyjson: v1.1.3
  - sequelize: v3.24.6
  - unirest: v0.5.1
- Python: v2.7.12
- Locust: v0.7.5

The microservices are deployed within the aaa - all about apps GmbHs infrastructure. This infrastructure is hosted by a web hosting provider named Hetzner based in Germany and is used as Infrastructure as a Service (IaaS). For the deployment of the microservices two virtual machines are used. Each of this machines consumes 2 CPU Cores and 2 GB memory and are located on two different servers. These servers have the following specification:

- CPU: Intel Xeon E5-1650 v3 Hexa-Core Haswell inkl. Hyper-Threading-Technologie
- memory: 256 GB DDR4 ECC RAM
- harddisks: x 480 GB SATA 6 Gb/s Data Center Series SSD (Software-RAID 1)
- network: 1 GBit/s-Port

Section 5.1 will outline the microservices architecture and Section 5.2 will conduct the performance measurement. This measurements test the performance of the microservices architecture by performing requests on the *Project Service*. However, the requests are first handled by the *API Gateway* who then forwards the requests to the *Project Service* which in addition conducts the *User Service* to validate if the user is authorized to perform the request. Because of this interactions the *API Gateway* and the *User Service* are located within one virtual machine and the *Project Service* is located within the second virtual machine in order ensures that all requests performed during the performance tests are executed over the network. The below Listing shows the configuration with domain names and ports:

- Microservices hosted by server mhoesel-master-dev.allaboutapps.at:
  - API Gateway with RESTful API listening on port 8080
  - User Service with RESTful API listening on port 8081
  - API Gateway with GraphQL API listening on port 8090
  - User Service with GraphQL API listening on port 8091
- Microservices hosted by server mhoesel-master2-dev.allaboutapps.at:
  - Project Gateway with RESTful API listening on 8080
  - Project Service with GraphQL API listening on 8090

5 Implementation

5.1 Microservices Architecture

For the purpose of this thesis aaa - all about apps GmbHs template project - which is used as baseline for new projects - is transformed from the monolithic pattern into a microservices architecture. This template project is implemented in JavaScript using NodeJS [20] and Hapi [21] as web framework and makes use of REST for its endpoints. It furthermore implements user management with registration, login and logout capabilities, allows the user to get or update his or her profile and ensures authentication by validating a token [22]. By default the template also allows to retrieve assets stored on the backend without being required to authenticate. Figure 5 shows the architecture of the template project which is based on the monolithic pattern.

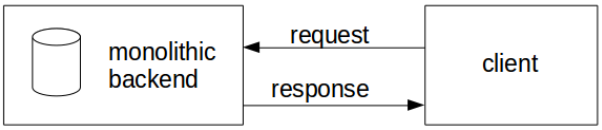

Figure 5: Monolithic architecture

For the objective of this thesis a microservices architecture based on the template project is developed. This microservices architecture forms an application which allows to manage users as well as projects. To put this into practice a microservices architecture consisting of the following services is implemented:

- User service: This microservice manages the user by offering endpoints for registration, login, logout and profile update.
- Project service: This microservice manages projects, which in this context represent programs consisting of a name and multiple programming languages. The endpoints implemented within this microservice allow to create new projects, add programming languages to a project, retrieve a list of projects, retrieve a specific project identified by an id, retrieve a specific language identified by an id and to retrieve statistics from *Github* as a third party application. In addition project related assets can be stored and retrieved from this microservice.
- API gateway: This microservice is the single entry point for clients. Its purpose is to act as a proxy and to forward incoming requests either to the *User service* or to the *Project service* and to forward the responses from the *User service* or *Project service* back to

the client. Through this *API gateway* the client has a single point of entrance and the complexity of the architecture behind is hidden.

The ER diagram in Figure 6 and in Figure 7 shows the structure of a user and a project respectively.

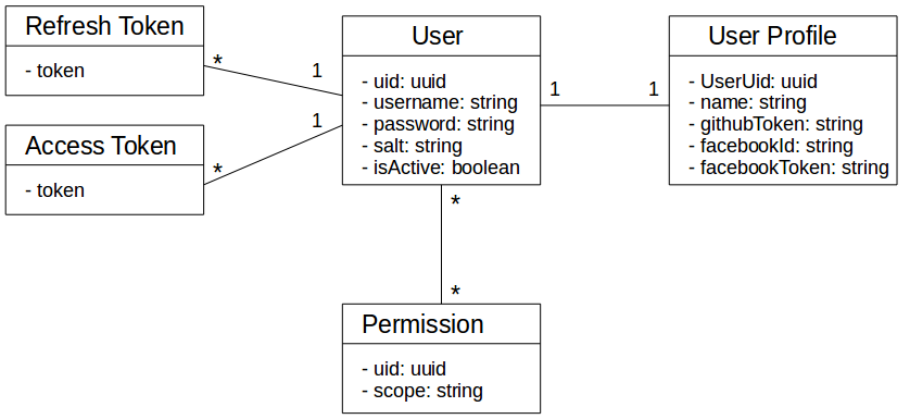

Figure 6: User ER-diagram

A *User* has exactly one *User Profile* and can have several *Permissions*, *Refresh Tokens* and *Access Tokens*. The permissions specify the scope of a user - for example a user can have standard permissions, CMS permissions and/or administrator permissions. A user can also have several access token and refresh token. This tokens are assigned to a user after a successful registration or login request. While the access token within this microservices architecture is only valid for a 24 hour period the refresh token is forever valid and can be used to obtain a new access token without performing a registration or login request. In addition a new access token and refresh token is generated for each device a user logs in.

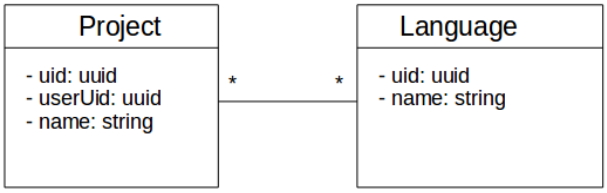

Figure 7: Project ER-diagram

A *Project* has a name, belongs to a *User* which is identified through a *userUid* and may be written in several languages (e.g. JavaScript, Python, HTML ...). Through the use of microservices and the separation of the database it is no longer possible to model the relation between a *Project* and a *User* through an association. Instead the *uid* of a *User* is saved in the *Project* table in order to keep track to which *User* a *Project* belongs. It should be pointed out that another option would have been the usage of a shared database. However, such a database conflicts with the microservice pattern as it does not allow a microservice to be self-contained, modular and loosely coupled [4] [11] [23].

5.1.1 Implementing Microservices based on REST

The process for implementing the microservices architecture based on REST as IPC mechanism is obtained in two steps. As a first step the template project is cloned three times and code refactoring is performed. Figure 8 shows the resulting architecture after applying the first step, for which the guidelines of domain driven design (DDD) introduced in Section 2.2 have been used. The changes made to the individual service are:

- User service: All user management related REST endpoints from the template project are kept while the asset endpoint is deleted from this service.
- Project service: All user management related REST endpoints are deleted from this service while the asset endpoint from the template project is kept.
- API gateway: All endpoints from the template project are kept but instead of performing any business logic the endpoints are configured to forward incoming requests to the accountable microservice.

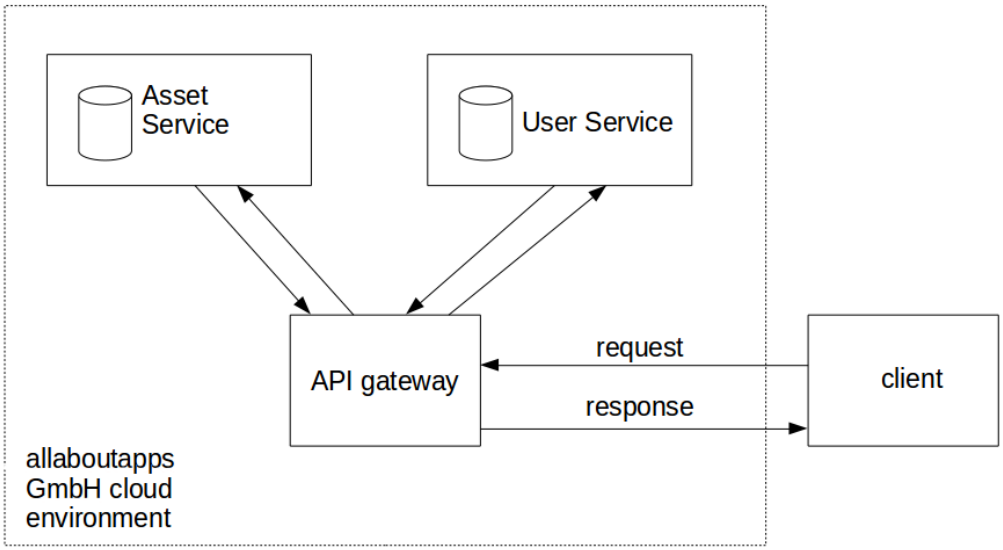

Figure 8: microservices architecture after refactoring

As a second step the *Project service* is extended to support project related tasks. This includes the implementation for maintaining projects as well as interacting with a third party API to request further information. Figure 9 outlines the final microservices architecture used within this thesis. In order to migrate to this architecture from the one described in Figure 8 the following changes need to be applied to the *Project Service* and to the *API Gateway*:

- Project service:
  - Implementation of endpoints and associated business logic to create and retrieve projects and programming languages used within those projects.
  - Implementation of endpoints and associated business logic to communicate with the REST API offered by *Github*.
  - Implementation of functionality to communicate with the *User service* in order to validate if a user has necessary rights to create or retrieve projects or programming languages.
- API gateway:
  - Implementation of new endpoints to reflect the changes made to the *Project service*.

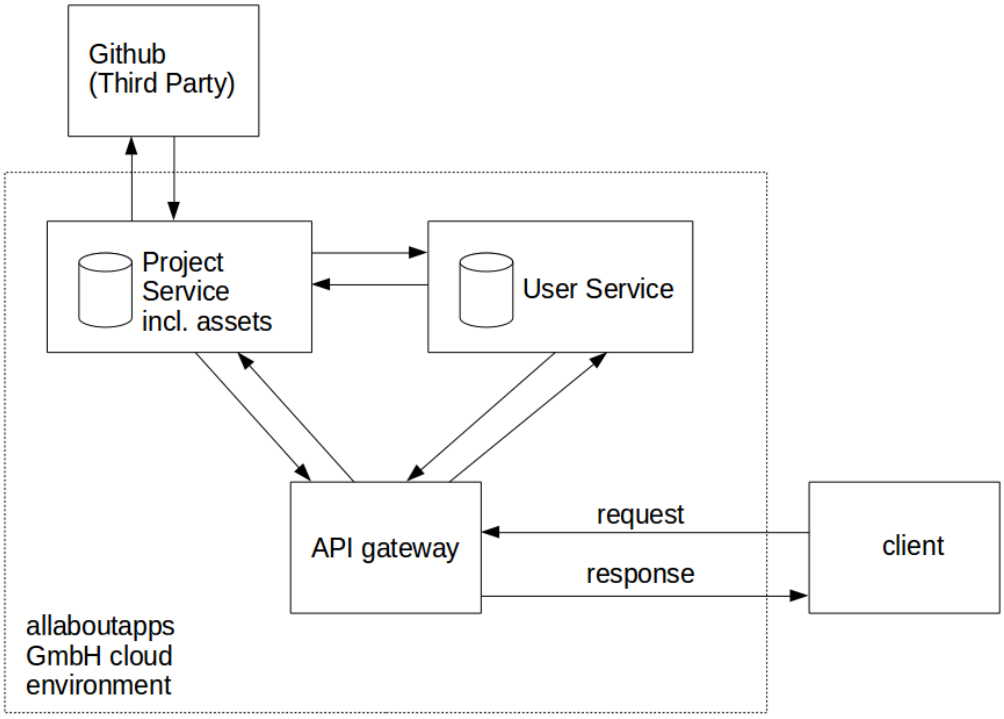

Figure 9: Final microservices architecture

5.1.2 Implementing Microservices based on GraphQL

The implementation for a microservices architecture based on GraphQL as IPC mechanism is done by cloning the developed microservice architecture from Section 5.1.1 composed of the three microservices *User service*, *Project service* and *API gateway* and substituting all REST routes with GraphQL endpoints while keeping the functionality exactly the same. In addition to this change Githubs GraphQL API is used instead of their REST API.

5.2 Performance Measurements

In order to evaluate the performance of the microservices architecture several measurements are conducted using a script based on Locust. Locust defines itself as "An open source load testing tool" [24], offers a web based user interface and keeps track of the following metrics for each API endpoint included in the measurements:

- Request count: Records how many requests have been performed
- Median value: Average response time in milliseconds based on the median
- Average value: Average response time in milliseconds based on the arithmetic mean
- Min value: Value of the response with the lowest latency
- Max value: Value of the response with the highest latency
- Content size: Size of the payloads in bytes

Table 1 gives an overview of the endpoints for which measurements are conducted.

ID	Method	Description
1	create project	Creates a new Project Entity
2	create language	Adds a new Language Entity to an existing Project
3	get project	Retrieves a Project with its attributes including an array of Language ids as Foreign Key
4	get language	Retrieves a Language with its attributes
5	get project ids	Returns a list of Project ids
6	get project ids - only GraphQL	Returns a list of Project ids with an optimized GraphQL query
7	get project with languages	Retrieves a Project with its attributes and includes an array of Languages with its attributes
8	get project with languages - only REST	Retrieves a Project with its attributes and includes an array of Languages with its attributes within a single request
9	get github stats	Returns a statistic about languages used in repositories of followers

Table 1: Endpoint definitions

These endpoints are located on the microservice responsible for hosting the *Project service*. However, the requests are not directly executed on this microservice. Instead the requests are sent to the *API gateway*, which acts as a single entrance point and forwards the request to the *Project service*. Because all requests can only be performed successfully by an authenticated user an authentication token is attached to the header of the requests. The *Project service*

extracts this authentication token from the header and contacts the *User service* in order to validate that the user is authorized to retrieve or manipulate the requested resource. In case the *User service* responses that the validation was successful the *Project service* performs its business logic and responses with the requested resource. In case the validation is not successful an error is returned indicating that the user is not authorized to perform the request. For the conducted tests however, it is ensured that the user is authorized. This is done by performing a login call as first request and using the authentication token from the response for all further requests during the measurement. Figure 10 shows the process of retrieving or manipulating a resource which is further described by the listing below.

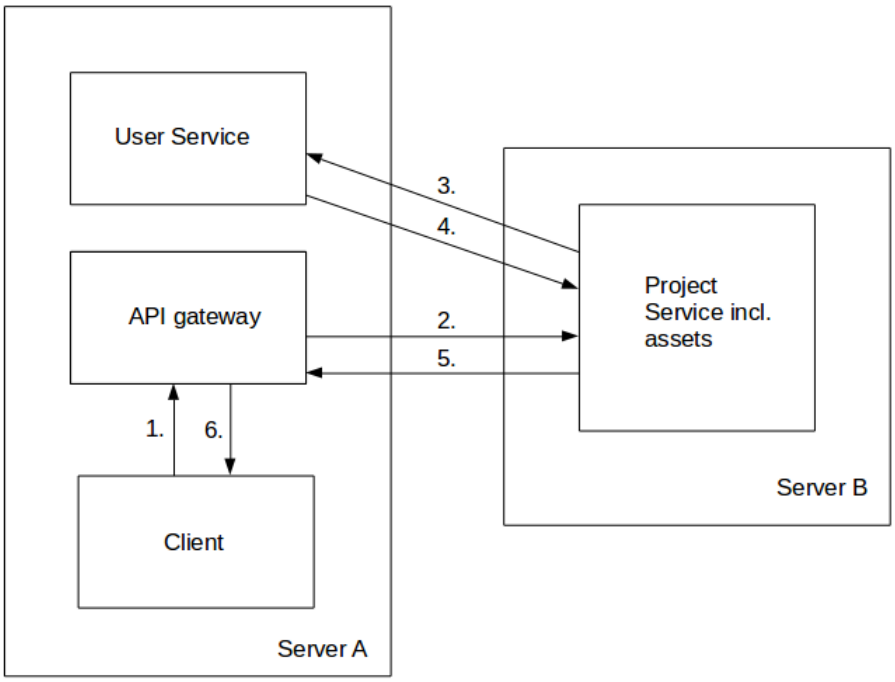

Figure 10: Performance measurement request/response flow

1. The request created by the client is sent to the *API gateway*.
2. The *API gateway* forwards the request to the *Project Service*.
3. The *Project Service* extracts the token from the request and sends this token within another request to the *User Service*.
4. The *User Service* validates the token and returns the result to the *Project Service*.
5. The *Project Service* performs the business logic in case the validation of the token was successful and returns the result back to the *API gateway*.
6. The *API gateway* returns the result back to the client.

All endpoints defined in Table 1 except the endpoints with *ID* 6 and *ID* 8 are implemented in both microservices architectures. The endpoint with *ID* 6 however is only implemented in the

microservices using GraphQL as IPC mechanism while the endpoint with *ID* 8 is only implemented in the microservices using REST as IPC mechanism. This decision was made to reflect the characteristics unique to the IPC mechanism. The below Listing highlights the reason for this decision:

- While a RESTful endpoint always returns the same response the response of a GraphQL endpoint depends on the fields requested within the query sent to the server. This must also be considered when developing the business logic. If for example a user specifies in a GraphQL query that he or she is only interested on a particular field it is not efficient to query the requested entity with all its fields from the database and then returning only the requested field. Instead only those fields which are requested by the user should be queried from the database. This aspect is considered in the endpoint with the *ID* 6, while the endpoint with the *ID* 5 queries all fields from the database. The performance results between those two endpoint will therefore show the impact of an optimized business logic.
- There are many cases in which a user is interested on an information which is nested within another one. For example, a user could be interested in the programming languages used for a specific project. However, such a project could theoretically have a lot more information. For example, it could offer the information which developers have participated on the project or which changes have been made to the project. Returning all this information within a single response would not be efficient. Instead of attaching all this information, a common way is to only attach identifies which can be used by the client to send an additional query. The endpoint with the *ID* 3 for example allows to query a project, which beside the basic information about the project itself attaches also a list of ids to further request the details for the programming languages used for that particular project. The endpoint with the *ID* 4 can then be used to retrieve the individual programming languages. While this is common for RESTful APIs it can lead to huge amounts of requests. The endpoint with the *ID* 8 was therefore implemented to return a project with not just a list of ids for the programming languages but also includes all other details for these programming languages used within this project. The measurements will show the contrast of latency and payload size between these two strategies.

5.3 Schema Validation

Schema validation - introduced in Section 2.10 - enables the serverside application to validate the incoming request or the outgoing response against a predefined schema. This Section explains the process of schema validation for both RESTful routes and GraphQL endpoints using the registration process as an example. It describes the serverside configuration as well as the queries which are executed against this configuration. In consideration of the robustness principle and the tolerant reader principle described in Section 2.9 the following scenarios are explored:

1. endpoint is configured without any schema validation
2. endpoint is configured with conservative schema validation
3. endpoint is configured with liberal schema validation

While the first case - no schema validation - is only valid for a RESTful endpoint, the second and third case - conservative and liberal schema validation - are examined on a RESTful API server as well as on a GraphQL server. The different schema validation configurations are tested with slightly different queries, which are listed below:

1. query with the expected payload is sent to the server
2. query with a payload containing additional fields to the ones required is sent to the server
3. query with contains a field of type *Number* instead of type *String* is sent to the server

Those three queries are used to test the different schema implementations specified in the beginning of this Chapter in order to validate or invalidate the expected results.

5.3.1 Schema validation in REST

This Section shows the process of implementing REST endpoints with and without schema validation, the definition and execution of queries against the defined endpoints and shows the result of the executed queries. The structure of the Section consists of:

1. endpoint implementation without any schema validation
2. testing of the endpoint specified in 1. with the queries defined in Section 5.3
3. refactoring of the endpoint specified in 1. to support conservative schema validation
4. testing of the endpoint specified in 3. with the queries defined in Section 5.3
5. refactoring of the endpoint specified in 3. to support liberal schema validation
6. testing of the endpoint specified in 5. with the queries defined in Section 5.3

Implementing a REST Endpoint without Schema Validation

The Hapi framework enables the developer to easily build REST routes by specifying a route definition which comprises at minimum a *method*, a *path*, a *handler* and may comprises an optional *config* object to further extend the route definition. Figure 5.1 shows the initial route definition for the *registration* route. This route definition specifies the HTTP Verb "POST" as *method*, sets "/api/v1/auth/register" as *path* and registers a function called *register* as *handler*. Each time a "POST" request to the *path* "/api/v1/auth/register" is performed the *register* function is executed. This function contains the business logic to register a new user in the database as well as to generate a valid authentication token and response. The example also sets the *auth* flag on the optional config object to false, indicating that no authentication process needs to be performed for this route definition.

```
{
   method: "POST",
   path: "/api/v1/auth/register",
   handler: ctrl.register,
   config: { auth: false }
},
```

Listing 5.1: Hapi Route Definition

Querying a REST Endpoint without Schema Validation

In order to successfully register a new user, a request with a payload consisting of a *username* and *password* needs to be generated and sent to the route definition specified in Figure 5.1. In case the server can successfully process the request a response with a payload consisting of a *tokenType*, *accessToken*, *refreshToken* and *expiresIn* field is generated. Those fields can further be used by the client in order to perform queries as authenticated user. In case the server is not able to successfully process the request an error containing an error code and an error message is returned. The error code further indicates if either the client or the server itself is responsible for the error while the error message gives a description of the error itself. Figure 5.2 shows a successful registration request using the command line tool *curl*. The command creates a "POST" request and attaches a payload consisting of the two fields *username* and *password*. It further sets two header fields indicating that the payload is of type "JSON" and that the response has to be in type "JSON" as well. The generated request is then sent to the server with the domain name *mhoesel-master-dev.allaboutapps.at* and the port *8081* using HTTP as underlying protocol and specifying *"/api/v1/auth/register"* as path. Because the request matches the route definition in Figure 5.1, the *registration* handler is executed. This handler registers the user, generates an *accessToken* as well as a *refreshToken* and attaches those information together with the *tokenType* and the *expiresIn* field to the response payload. The *accessToken* can then be used by the client in order to make authenticated requests to the server. This can be done until the token gets invalidated due to its limited lifetime, which is expressed by the *expiresIn* field. The *refreshToken* can then be used to generate a new *accessToken* on the server.

```
curl -X POST
   --header 'Content-Type: application/json'
   --header 'Accept: application/json'
   -d '{"username":"user1","password":"password1"}'
      'http://mhoesel-master-dev.allaboutapps.at:8081/api/v1/auth/register'
{
   "tokenType":"Bearer",
   "accessToken":"b25b8caa-362f-4de1-a14b-ad89b15153fd",
   "refreshToken":"5a9ff925-2b76-4430-884e-b756580a70f5",
```

```
   "expiresIn":86400000
}
```

Listing 5.2: Successful registration request to REST route

Anyhow, by default no schema validation takes place. Therefore a client can sent a malformed request which is missing one or both of the two necessary fields *username* and *password* and can also attach fields which are not expected. Such a malformed request is then passed directly to the handler function of the route definition without any notice that the payload is malformed. Figure 5.3 illustrates such a case. The payload does not include a valid value for the *username* field as the handler function of the *registration* route expect a value of type *String* but a value of type *Number* is given. Also an extra field named *additionalField* is attached to the payload. However, the implementation assumes that there is a field *username* and that this field contains a value of type *String*. As a result the application crashes and generates an *"Internal Server Error"* error when it tries to process the query. The reason for the crash in this specific case is that the insert statement of the user into the relation *User* of the database fails due to the wront type provided for the *username*.

```
curl -X POST
   --header 'Content-Type: application/json'
   --header 'Accept: application/json'
   -d '{"username":1245, "password":"password1",
      "additionalField":"additionalValue"}'
      'http://mhoesel-master-dev.allaboutapps.at:8081/api/v1/auth/register'
{
   "statusCode":500,
   "error":"Internal Server Error","message":"An internal server error
      occurred"
}
```

Listing 5.3: Error prone registration request to REST route

The error message outlined in Figure 5.3 is furthermore not very meaningful. Moreover, the crash itself could have left serious damage - for example by creating an inconsistent state in the database. In order to prevent such an error it would have been possible to first check if the expected fields are attached to the payload and that this fields contain a value of the expected type. Doing so makes it also possible to throw a custom error with a meaningful error message attached to it. However, further checks on the value may be necessary. For the *password* field we would need to ensure that the value is not only of type *String*, but also that it only contains alphanumeric characters and has a minimum as well as a maximum character length. Performing such checks can be vast, especially if the payload is of high complexity. This task can be implemented more efficiently using libraries. One such library, which is very popular within the Hapi framework, is Joi. This library is also used in this thesis in order to perform validation of payloads.

Implementing a REST Endpoint with Conservative Schema Validation

The microservices in this thesis which communicate via REST use a library named Joi for payload validation. The library allows a developer to define a schema which is used to validate the payload of requests and responses, creating meaningful error message in case the payload is not conform with the defined schema. Figure 5.4 extends Figure 5.1 by implementing such a schema definition. Two objects are added to the *config* object, namely *validate* and *response*. Both objects contain a schema defined with Joi.

```
// route definition for registering new user
{
   method: "POST",
   path: "/api/v1/auth/register",
   handler: ctrl.register,
   config: {
      auth: false,
      validate: {
         payload: Joi.object().keys({
            username: Joi.string().required(),
            password: types.password.required()
         }).required()
      },
      response: {
         schema: Joi.object().keys({
            refreshToken: types.uidv4.allow(null).required(),
            accessToken: types.uidv4.required(),
            expiresIn: Joi.number().integer().required(),
            tokenType: Joi.string().required()
         }).required()
      },
   }
},
```

Listing 5.4: Conservative Hapi Route Definition with Joi Validation

The object *validate* is used to validate the payload of a request. In this example, it expects the payload to consist of an object which contains the following two fields:

- *username* of type *String*
- *password* of custom type *password*

The custom type *password* further restricts the validation to only allow values which have at least six characters and no whitespace characters. If an incoming request matches this schema the *register* function - which is defined as a handler - is called. This function executes the business logic and creates a response. This response is then validated against the response

schema in the route definition and must therefore contain the following fields with corresponding types:

- *refreshToken* of type *uidv4* or *null*
- *accessToken* of type *uidv4*
- *expiresIn* of type *Integer*
- *tokenType* of type *String*

In case the payload of the request does not match the defined schema an HTTP client error is thrown. Likewise if the payload of the response does not match the defined schema an HTTP server error is thrown. In both cases a message with the reason why the validation did not succeed is attached to the error.

Querying a REST Endpoint with Conservative Schema Validation

Listing 5.5 and Listing 5.6 perform the same queries which have been used in Listing 5.2 and Listing 5.3. But this time the requests are validated at the backend using the schemas implemented in Listing 5.4. Listing 5.5 shows the result of a query with the correct payload attached to it. Similar to Listing 5.2 this query succeeds and a response containing a payload with an object consisting of the fields *accessToken*, *refreshToken*, *tokenType* and *expiresIn* is returned. However, in contrast to Listing 5.2, the payload defined in Listing 5.5 was first validated against the schema defined at the backend, as well as the response which was generated by the *handler* function of the route definition.

```
curl -X POST
   --header 'Content-Type: application/json'
   --header 'Accept: application/json'
   -d '{"username":"user10","password":"password10"}'
      'http://mhoesel-master-dev.allaboutapps.at:8081/api/v1/auth/register'
{
   "tokenType":"Bearer",
   "accessToken":"b25b8caa-362f-4de1-a14b-ad89b15153fd",
   "refreshToken":"5a9ff925-2b76-4430-884e-b756580a70f5",
   "expiresIn":86400000
}
```

Listing 5.5: Successful registration request to REST route with conservative validation

In addition to the similarity of the successful queries in Listing 5.2 and in Listing 5.5, Listing 5.6 also throws an error like the query in Listing 5.3 did. However, the major difference is that this time the schema validation failed and therefore the error is already thrown before the *handler* function is executed. The error thrown has furthermore the status code 400, indicating a *"Bad Request"* error. This is in contrast to the Listing in 5.3 which generated an error with status code

500, indicating an *"Internal Server Error"*. Moreover the message part of the error contains a descriptive text indicating that the error was thrown because the value in field *username* was not of type *String* and because the field *additionalField* is attached while it is not specified in the schema definition.

```
curl -X POST
   --header 'Content-Type: application/json'
   --header 'Accept: application/json'
   -d '{"username":12345, "password":"password10",
       "additionalField":"additionalValue"}'
       'http://mhoesel-master-dev.allaboutapps.at:8081/api/v1/auth/register'
{
   "statusCode":400,
   "error":"Bad Request","message":"
      child \"username\" fails because [\"username\" must be a string].
      \"additionalField\" is not allowed
   ",
       "validation":{"source":"payload","keys":["username","additionalField"]}
}
```

Listing 5.6: Error prone registration request to REST route with conservative validation

Implementing a REST Endpoint with Liberal Schema Validation

The schema validation implemented in Listing 5.4 follows a conservative style. As we can see in Listing 5.6 a request or a response has to match exactly the schema in order to be valid, otherwise an error is thrown. However, in Section 2.9 the robustness principle was introduced. This principle states that one should be conservative in what one does but be liberal in what one accepts from others [36]. Strictly said, the principle states that as long as a request can be interpreted it should be interpreted while a response still has to exactly match the defined schema. Listing 5.7 shows one possible way to be more liberal when interpreting a request.

```
// route definition for registering new user
{
   method: "POST",
   path: "/api/v1/auth/register",
   handler: ctrl.register,
   config: {
      auth: false,
      validate: {
         options: { allowUnknown: true },
         payload: Joi.object().keys({
            username: Joi.any().allow(Joi.string(),
               Joi.number()).required(),
            password: types.password.required()
```

```
         }).required()
      },
      response: {
         schema: Joi.object().keys({
            refreshToken: types.uidv4.allow(null).required(),
            accessToken: types.uidv4.required(),
            expiresIn: Joi.number().integer().required(),
            tokenType: Joi.string().required()
         }).required()
      },
   }
},
```

Listing 5.7: Liberal Hapi Route Definition with Joi Validation

In comparison to the conservative route definition in Listing 5.4 this route definition adds an *options* object to the *validation* object and sets a field named *allowUnknown* with value true on the *options* object in order to allow a payload to contain unknown fields. Therefore additional fields beside *username* and *password* can be attached to the payload. In addition the field *username* is changed in order to allow a value of type *String* and of type *Number*. However, allowing a value of type *Number* in the payload enforces the developer to refactor the *handler* function. Otherwise an exception would be generated similar to Listing 5.3 in which we tried to register a user with a *username* of type *Number* and got an *"Internal Server Error"* as the *username* column of the *Users* relation is designed to only accept values of type *String*. One solution to prevent this error is to convert the username inside the *handler* function to the type expected by the database. Listing 5.8 outlines the code snippet for this solution which not only parses the *username* from the *request.payload* object but also converts the value to type *String*.

```
export function register(request, reply) {
   const username = request.payload.username.toString();
    ...
```

Listing 5.8: Type transformation in controller code

Querying a REST Endpoint with Liberal Schema Validation

Querying the backend with the two queries introduced in Listing 5.2 and Listing 5.3 results in two error free and valid responses. Listing 5.9 and Listing 5.10 outline the query and their results. The query in Listing 5.9 succeeds as the payload of the request contains exactly the fields used in the *handler* function implemented on the server. This query already returned a successful response in the example performed earlier, at first with no validation in place and afterwards with conservative validation in place.

```
curl -X POST
   --header 'Content-Type: application/json'
   --header 'Accept: application/json'
   -d '{"username":"user11","password":"password11"}'
      'http://mhoesel-master-dev.allaboutapps.at:8081/api/v1/auth/register'
{
   "tokenType":"Bearer",
   "accessToken":"39390657-02c8-46b5-96a7-00b6f227371c",
   "refreshToken":"028997af-42b4-4338-881d-e88f63e00abf",
   "expiresIn":86400000
}
```

Listing 5.9: Successful registration request to REST route with liberal validation

With the liberal schema validation in place the second query succeeds as well. This is in contrast to the example performed earlier, as the first example with no validation in place returned an *"Internal server error"* and the second example with conservative validation in place returned an *"Bad request"* error. The request now succeeds because the schema allows to also sent a value of type *Number* as username and allows additional fields as well. While the value of type *Number* is transformation to the type *String* in the controller code all additional fields are ignored.

```
curl -X POST
   --header 'Content-Type: application/json'
   --header 'Accept: application/json'
   -d '{"username":1245 ,"password":"password10",
      "additionalField":"additionalValue"}'
      'http://mhoesel-master-dev.allaboutapps.at:8081/api/v1/auth/register'
{
   "tokenType":"Bearer",
   "accessToken":"14d3478f-e0d6-4c4b-8e63-accc74be3f72",
   "refreshToken":"087577f4-49c4-488d-afc6-381c84070dee",
   "expiresIn":86400000
}
```

Listing 5.10: Successful registration request to REST route with liberal validation

5.3.2 Schema Validation in GraphQL

This Section describes the implementation of GraphQL endpoints, the definition and execution of queries against the defined endpoints and outlines the result of the executed queries. This Section - in contrast to Section 5.3.1 - does not implement endpoints without any schema validation. This is because GraphQL is designed to enforce a schema. The default validation process of GraphQL can be further seen as conservative. In order to support a more liberal

schema validation changes to the library itself are made. This changes proof that GraphQL can be adapted to allow a more liberal schema validation and that GraphQL therefore allows to support the robustness principle described in Section 2.9. The structure of the Section consists of:

1. endpoint implementation with default schema validation
2. testing of the endpoint specified in 1. with the queries defined in Section 5.3
3. adding support for liberal schema validation to the GraphQL library
4. endpoint implementation with liberal schema validation
5. testing of the endpoint specified in 4. with the queries defined in Section 5.3

Implementing a GraphQL Endpoint

GraphQL is a strongly-typed query language. Because of this principle, which is described in detail in Section 2.8, every endpoint must define a type definition which specifies the fields exposed to the client. Each time a client queries a GraphQL application server a validation of that query against the type definition is performed. In case the query is not conform with the defined type definition a HTTP error with status code 400 and an error message further describing the error is thrown. Otherwise the resolver function is executed and - assuming that the resolver function does not throw an error itself - the requested fields as well as the HTTP status code 200 are returned to the client. GraphQLs design is in contrast to REST endpoints, where - by default - no schema validation is performed at all. Because of that nature REST endpoints are also defined as weakly-typed.

Listing 5.11 outlines the type definition for the *RegistrationType* which defines four fields, namely *accessToken*,*refreshToken*, *tokenType* and *expiresIn*, and associates this fields with a GraphQL scalar type. In addition the GraphQL wrapper type *GraphQLNonNull* is used in order to prevent *null* as value. Furthermore a custom type named *GraphQLUUID* is used which enforces a value to not only be of type *GraphQLString* but also to match a specific pattern in order to ensure the value is of a valid uuid v4 format.

```
const RegistrationType = new GraphQLObjectType({
    name: "Registration",
    fields: () => ({
        tokenType: {
            type: new GraphQLNonNull(GraphQLString)
        },
        accessToken: {
            type: new GraphQLNonNull(GraphQLUUID)
        },
        refreshToken: {
            type: GraphQLUUID
        },
```

```
        expiresIn: {
            type: GraphQLInt
        }
    })
});
```

Listing 5.11: GraphQL registration type definition

Listing 5.12 shows the implementation for the *register* endpoint. This endpoint can be called by a client in order to signup a new user. The definition consists of three fields. The *type* field references the *Registration* type defined in Listing 5.11. The *arg* fields defines two arguments, namely *username* and *password* with each of those fields specifying a type of *GraphQLString* which is not allowed to be null. The *resolve* field takes those arguments and executes a function named *postUser* in order to resolve the query. This function contains the business logic to register a new user in the database as well as to generate a valid authentication token and a response. The response has to be conform to the type definition made for the *RegistrationType*.

```
register: {
   type: RegistrationType,
   args: {
      username: { type: new GraphQLNonNull(GraphQLString) },
      password: { type: new GraphQLNonNull(GraphQLString) }
   },
   resolve: (value, { username, password }) => {
      return postUser(username.toString(), password);
   }
},
```

Listing 5.12: GraphQL registration endpoint definition

Querying a GraphQL endpoint

Similar to the registration example in Section 5.3.1, a request with a *username* and *password* attached to the payload is executed against the endpoint definition specified in Listing 5.12. Due to the type definition made for the *RegistrationType* in Listing 5.11 the payload of the response can contain the fields *tokenType*, *accessToken*, *refreshToken* and *expiresIn*. However, in contrast to REST where the payload cannot be influenced from the client, GraphQL allows to only attach those fields to the payload the client is interested in. Listing 5.13 shows a successful registration using the command line tool *curl*. The command creates a "POST" request and attaches a payload of format "JSON". This payload contains the query which further specifies that a mutation should be performed on the *register* endpoint, that the two arguments *username* and *password* should be passed to the *register* function and that the response payload should only include the two fields *tokenType* and *accessToken*. This request is then sent to the server with the domain name address *mhoesel-master-dev.allaboutapps.at* and the port *8091* using

the HTTP protocol and specifying *"/graphql"* as path. Because this query is conform to the endpoint definition in Listing 5.11 the *resolver* function is executed. This function registers the user in the database, generates the tokens and returns the *tokenType* and *accessToken* to the caller. However, the two fields *refreshToken* and *expiresIn* are not returned in this example as those fields have not been specified in the query.

```
curl -X POST
   --header 'Content-Type: application/json'
   --header 'Accept: application/json'
   -d '{
      "query":"mutation{
         register(username:\"user22\",password:\"password22\")
         {tokenType, accessToken}
      }"}}'
   'http://mhoesel-master-dev.allaboutapps.at:8091/graphql'
{
  "data": {
    "register": {
      "tokenType": "Bearer",
      "accessToken": "9d2e07d3-f60b-4b3c-a446-4f1580696c5c"
    }
  }
}
```

Listing 5.13: Successful registration request to a GraphQL endpoint

GraphQL by default only allows to query capabilities which have been defined on the server. Queries which include additional arguments or ask for additional fields not defined on the server are invalid. GraphQL can therefore be seen as a library which is conservative in what it receives and in what it response with. Listing 5.14, Listing 5.15 and Listing 5.16 outline queries which are invalid due to the conservative validation process.

The request performed in Listing 5.14 attaches a value of type *Number* instead of type *String* to the *username*. Because the type definition of the *Registration* type defines that *username* has to be of type *String* an error is thrown indicating that *username* has an invalid value. This behavior is comparable to the request performed against the REST route in Listing 5.6.

```
curl -X POST
   --header 'Content-Type: application/json'
   --header 'Accept: application/json'
   -d '{
      "query":"mutation {
         register(username:12345,password:\"password22\")
         {tokenType, accessToken}
      }"}}'
   'http://mhoesel-master-dev.allaboutapps.at:8091/graphql'
```

```
{"errors": [{
      "message": "Argument \"username\" has invalid value 12345.\nExpected
         type \"String\", found 12345.",
      "locations": [{
          "line": 2,
          "column": 22
        }]
}]}
```

Listing 5.14: Error prone registration request containing a wrong type

Listing 5.15 furthermore illustrates a request in which an additional argument - one which is not specified in the endpoint definition in Listing 5.12 - is attached. The query is therefore not conform to the definition and an error is thrown, containing the information that an "unknown argument was provided" in the error message. This behavior can be compared to the request performed against the REST route in Listing 5.6 as both example provide an additional field to be processed by the server.

```
curl -X POST
   --header 'Content-Type: application/json'
   --header 'Accept: application/json'
   -d '{
      "query":"mutation {register(
            username:\"user23\",
            password:\"password23\",
            extraField:\"extraValue\"
         )
         {tokenType, accessToken}
      }"}'  'http://mhoesel-master-dev.allaboutapps.at:8091/graphql'
{"errors": [{
      "message": "Unknown argument \"extraField\" on field \"register\" of
         type \"RootMutationType\".",
      "locations": [{
          "line": 2,
          "column": 56
        }]
}]}
```

Listing 5.15: Error prone registration request containing an unknown argument

Listing 5.16 demonstrates a request with a field which is not specified in the type definition of the *Registration* type is requested to be returned by the server. Similar to Listing 5.14 and Listing 5.15 this request throws an error as well due to the fact that the validation fails. The error message returned by the server further states that the extra field requested by the query could not be queried. Because one cannot ask for specific fields to be returned from a REST endpoint no comparable example exists.

```
curl -X POST
   --header 'Content-Type: application/json'
   --header 'Accept: application/json'
   -d '{
      "query":"mutation {
         register(username:\"user23\", password:\"password23\")
         {tokenType, accessToken, myField}
      }"}}'
   'http://mhoesel-master-dev.allaboutapps.at:8091/graphql'
{"errors": [{
      "message": "Cannot query field \"myField\" on type \"Registration\".",
      "locations": [{
          "line": 5,
          "column": 5
        }]
}]}
```

Listing 5.16: Error prone registration request containing an unknown field

Adding Support for Liberal Schema Validation to the GraphQL Library

Because of its strongly typed architecture GraphQL does not allow to attach fields not specified on the type definition, nor is it possible to modify the type definition to allow two different scalars for one input parameter. This is in contrast to the REST endpoints defined in Hapi using Joi for schema validation. Listing 5.7 demonstrated how a REST endpoint can be configured with a more liberal schema validation which on the one hand allowed a value to be either of type *String* or of type *Number* and on the other hand allowed to attach fields not specified in the schema itself. Multiple discussions are currently taking place weather or not to allow more liberal schema validation in GraphQL as well. [40] and [41] reference *Github issues* which have been in state "open" at the time writing this thesis. In order to proof that the necessary changes can be integrated into the library, the Graphql library itself as well as corresponding libraries have been forked from the official Github repository into the personal Github profile of the author [1] and the necessary changes to the code have been made. The list below summarizes this changes:

- Introducing a new scalar type *GraphQLStringOrInt* which allows a value to be of either type *String* or of type *Number*
- Removal of the *KnownArgumentNames* validation rule in order to allow arguments which are not specified in the type definition
- Removal of the *FieldsOnCorrectType* validation rule in order to query fields not specified in the type definition

[1] https://github.com/hoeselm

Listing 5.17 outlines the new scalar type *GraphQLStringOrInt* which is an object of type *GraphQLScalarType* consisting of the fields *name*, *description*, *serialize*, *parseValue* and *parseLiteral*.. The function assigned to the *parseValue* field validates the given value while the function assigned to the *parseLiteral* field validates the type of the value. In addition the function specified for the *serialization* field returns the actual value. In case the value is of type *Number* or of type *String* but can be converted to type *Number* the serialized value will be of type *Number*, otherwise - if the value is of type *String* - the value is returned as type *String*. The way this is done is shown in Listing 5.18.

```
export const GraphQLStringOrInt = new GraphQLScalarType({
  name: 'StringOrInt',
  description:
    'The `IntOrString` scalar type represents non-fractional signed whole '
       +
    'numeric values or textual data, represented as UTF-8. Int can ' +
    'represent values between -(2^31) and 2^31 - 1 and string can ' +
    'represent free-form human-readable text ',
  serialize: coerceStringOrInt,
  parseValue: coerceStringOrInt,
  parseLiteral(ast) {
    if (ast.kind === Kind.INT || ast.kind === Kind.STRING) {
      const num = parseInt(ast.value, 10);
      if (num <= MAX_INT && num >= MIN_INT) {
        return num;
      }
     return ast.kind === Kind.STRING ? ast.value : null;
    }
    return null;
  }
});
```

Listing 5.17: New scalar type GraphQLStringOrInt

The function *coerceStringOrInt* tries to convert a given value to type *Number* or to type *String*. This function is used by Listing 5.17 to serialize the value and to validate the value. The way this is done is by first trying to convert the given value to type *Number*. If the value is valid and within the range of *MIN_INT* and *MAX_INT* the value is rounded and returned. Otherwise, the value is converted to type *String* and if that value is valid it is returned. Finally, if the value is neither a valid *Number* nor a valid *String* an error is generated. In addition the function assigned to *parseLiteral* in Listing 5.17 is used to validate the type of the value by first checking if the type is a *Number* or a *String* and if its value can be converted to number and if not checking if the type is a *String*. If none of this rules succeeds the value *null* is returned to *parseLiteral* signaling that the given type does not match.

```
function coerceStringOrInt(value: mixed): ?mixed {
  const num = Number(value);
  if (num === num && num <= MAX_INT && num >= MIN_INT) {
    return (num < 0 ? Math.ceil : Math.floor)(num);
  }
  const string = String(value);
  if (string === string) {
    return string;
  }
  throw new TypeError(
    'StringOrInt cannot represent non 32-bit signed integer value or
       string:' +
    String(value)
  );
}
```

Listing 5.18: Function coerceStringOrInt

In order to use the newly created *GraphQLStringOrInt* type within the Typescript environment configured for this thesis it was added to the typed-graphql library. While [42] references the Github page which hosts the forked GraphQL project, [43] references the Github page which hosts the forked typed-graphql library.

Implementing a GraphQL Endpoint with Liberal Schema Validation

In order to create a GraphQL endpoint which follows a more liberal schema validation approach three changes to the microservices are made. First, the modified GraphQL library is used. Second, the implementation of the *register* endpoint specified in Listing 5.12 is refactored using the new type *GraphQLStringOrInt* instead of the type *GraphQLString* for the *username* argument. Listing 5.19 shows the refactored endpoint.

```
register: {
   type: RegistrationType,
   args: {
      username: { type: new GraphQLNonNull(GraphQLStringOrInt) },
      password: { type: new GraphQLNonNull(GraphQLString) }
   },
   resolve: (value, { username, password }) => {
      return postUser(username.toString(), password);
   }
},
```

Listing 5.19: GraphQL refactored registration endpoint definition

Third, the *username* argument needs to be converted to type *String* in order to successfully

insert it into the *username* column of the *User* relation, otherwise an *"Internal Server Error"* exception will be thrown by the database in case the *username* is of type *Number*. This exception has been described in detail in Section 5.3.1. The resolver function in Listing 5.19 shows how the username is converted to type *String* before it is passed as argument to the *postUser* function.

Querying a GraphQL Endpoint with Liberal Schema Validation

Performing the queries specified in Listing 5.14, Listing 5.15 and Listing 5.16 using the adapted library results in error-free responses. The queries as well as their results are listed in Listing 5.20, Listing 5.21 and Listing 5.22

Listing 5.20 executes the same query as in Listing 5.14. Due to the updated *registration* endpoint, which is now configured to support both a value of type *Number* and a value of type *String* this call is performed successfully and the two requested fields *accessToken* and *tokenType* are returned.

```
curl -X POST
   --header 'Content-Type: application/json'
   --header 'Accept: application/json'
   -d '{
      "query":"mutation {
         register(username:12345,password:\"password22\")
         {tokenType, accessToken}
      }"}}'
   'http://mhoesel-master-dev.allaboutapps.at:8091/graphql'
{
  "data": {
    "register": {
      "tokenType": "Bearer",
      "accessToken": "75b7d01d-6671-4b12-8f30-5a08f94f5cb3"
    }
  }
}
```

Listing 5.20: Successful registration request containing a value of type Number

Listing 5.21 executes a query with an additional argument. In contrast to Listing 5.15 where an "Unknown argument" exception was thrown the request was processed successfully. While the two arguments *username* and *password* have been passed to the *resolver* function the argument *extraField* was ignored.

```
curl -X POST
   --header 'Content-Type: application/json'
   --header 'Accept: application/json'
   -d '{
```

```
      "query":"mutation {
         register(
            username:\"user23\",
            password:\"password23\",
            extraField:\"extraValue\"
         )
         {tokenType, accessToken}
      }"}}'
   'http://mhoesel-master-dev.allaboutapps.at:8091/graphql'
{
  "data": {
    "register": {
      "tokenType": "Bearer",
      "accessToken": "a6db8bac-4660-4ffb-be2f-bf03807b04f8"
    }
  }
}
```

Listing 5.21: Successful registration request containing an unknown argument

Listing 5.22 requests an additional field *myfield* to be returned by the server. This query generated an exception in Listing 5.16 as the field is not defined on the type definition. In Listing 5.22 however the call succeeds due to the changes made to the library. While the two fields *tokenType* and *accessToken* are returned in the payload the field *myfield* is ignored.

```
curl -X POST
   --header 'Content-Type: application/json'
   --header 'Accept: application/json'
   -d '{
      "query":"mutation{
         register(username:\"user22\",password:\"password22\")
         {tokenType, accessToken, myfield}
      }"}}'
   'http://mhoesel-master-dev.allaboutapps.at:8091/graphql'
{
  "data": {
    "register": {
      "tokenType": "Bearer",
      "accessToken": "c16869a1-2ec0-4d4c-a477-1b6da5410268"
    }
  }
}
```

Listing 5.22: Successful registration request containing an unknown field

5.4 API Documentation with Swagger and GraphiQL

This Chapter describes the implementation and usage of Swagger and GraphiQL, which - at the time writing this thesis - are the most popular tools to document and explore a RESTful API and a GraphQL API respectively. The primary goal of these tools is to allow the developer of the APIs to easily document those APIs and to give a client developer an easy way to explore and test those APIs. This Chapter therefore outlines how unauthenticated and authenticated requests to the backend can be made using these tools. Because GraphiQL does not support authenticated requests by default the chapter also outlines the necessary changes made to the GraphiQL library and to the GraphQL server in order to support those authenticated requests.

5.4.1 Swagger

Swagger describes itself as "The world's most popular API framework" [28] and gives the developer the options to design, build, document and consume RESTful APIs [28]. It is built on top of the OpenAPI specification which was originally known as Swagger specification and which defines a standard for describing a language-agnostic interface to RESTful APIs [44]. In order to enable Swagger to self-document the API defined in Hapi a plugin called hapi-swagger is used. This plugin - similar to other Hapi plugins - needs to be registered within the Hapi framework in order to use it. In addition each API which should be documented via Swagger must be tagged individually. According to the hapi-swagger plugin documentation this design decision was enforced in order to give the user the freedom to only tag certain endpoints, mostly because a project may consist of endpoints to serve web pages as well as APIs designed for data exchange only [29].

In order to demonstrate the process to add Swagger to an endpoint, the registration example from Listing 5.1 is extended. As first step - to enable self-documentation of the registration route - the hapi-swagger plugin is added to the configuration of this route object. Listing 5.23 shows the extended configuration for the registration route including the hapi-swagger plugin configuration. This plugin allows further configuration. The example adds a custom error code which is thrown when a user tries to register with a username which already exists. Adding this error code makes it possible to Swagger to include it to the auto generated documentation. The example furthermore is configured with schema validation using Joi. The schema used in the example is the same as in Listing 5.4 but a description was added to each filled. This description gets parsed by Swagger and is included in the Swagger documentation.

```
// route definition for registering new user
{
   method: "POST",
   path: "/api/v1/auth/register",
   handler: ctrl.register,
   config: {
      auth: false,
```

```
      validate: {
         payload: Joi.object().keys({
            username: Joi.string().required().description("Username of the
               new user. This value will be used for password
               authentication"),
            password: types.password.required().description("clear password
               of the user")
         }).required().label("RegisterPostRequest")
      },
      response: {
         schema: Joi.object().keys({
            refreshToken: types.uidv4.allow(null).required().description("A
               token for this user that can be used to get a new
               accessToken using the /auth/token endpoint. This token
               might change over time, so for a new login, or when
               exchanging a refreshToken for a new accessToken a new
               refreshToken might be generated. Guest users don't receive
               a refreshToken (will be null)"),
            accessToken: types.uidv4.required().description("The main
               authorization token for API usage. This token has to be
               provided with the Authorization HTTP header as Bearer for
               all authenticated calls. Only valid for a limited time -
               see validUntil"),
            expiresIn: Joi.number().integer().required().description("Time
               in seconds how long the Access Token will be valid on the
               server. Guest users will have an accessToken that never
               expires (-1)"),
            tokenType: Joi.string().required().description("a string that
               tells you what type of token it is. You should use this
               type in your Authorization requests.")
         }).required().label("Token").description("The standard response
             for all authentication related call");
      },
      plugins: {
         "hapi-swagger": {
            responses: statusCodes.getSwaggerStatusResponses({
               code: 409,
               message: "This username is no longer available"
            })
         }
      },
   }
},
```

Listing 5.23: Hapi Route Definition with Swagger enabled

The self-documented API can be accessed using a web browser, by default the site is served via the */documentation* route. Figure 11, Figure 12 and Figure 13 show three parts of the self-documented API for the registration route implemented in Listing 5.23.

The first Figure shows the documentation part, in which the response model for a successful request is described. The name *Token* is parsed from the label attribute defined in Listing 5.23 and the four fields *refreshToken*, *accessToken*, *expiresIn*, *tokenType* with their types and descriptions are parsed as well. The documentation part gives a user a good overview how the response for an API will look like.

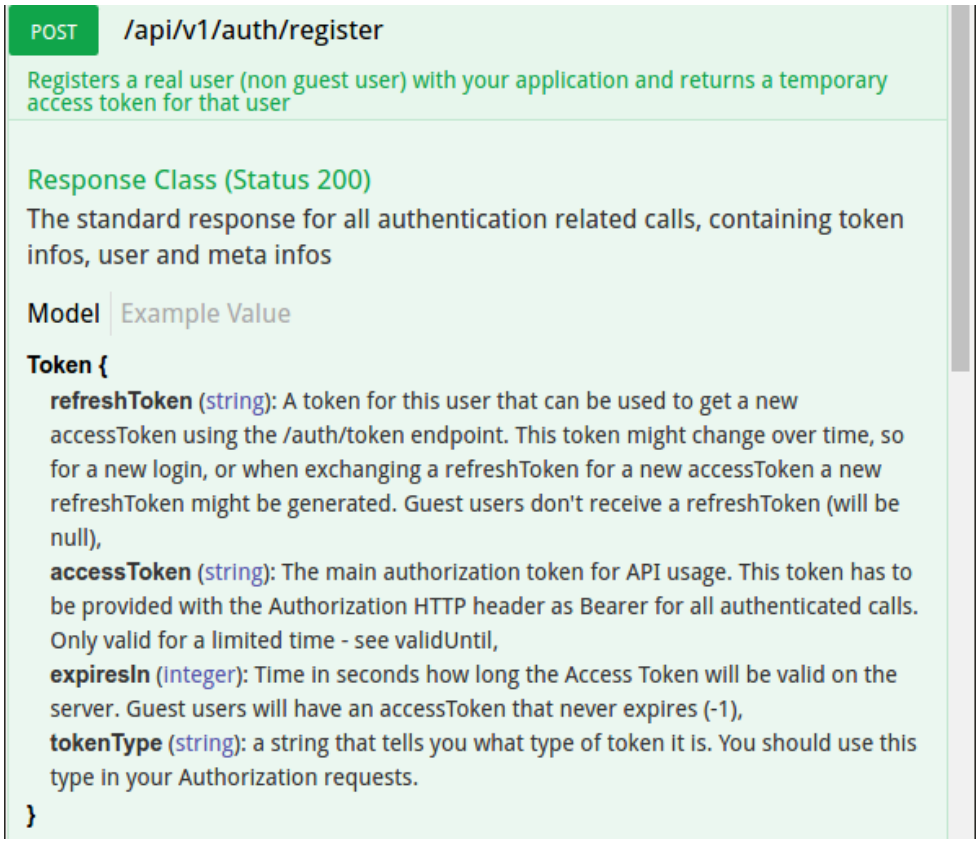

Figure 11: Swagger documentation.

Figure 12 shows the request part, in which one can perform a request to the server by simply filling out the form and clicking on the button with the label "Try it out". The form in the example consists of the two fields *username* and *password* and have been extracted from the payload definition of Listing 5.23. This part also shows the possible response messages apart from a HTTP response with status 200 and includes the generic HTTP response message with status code 400 and the custom defined HTTP error code 409.

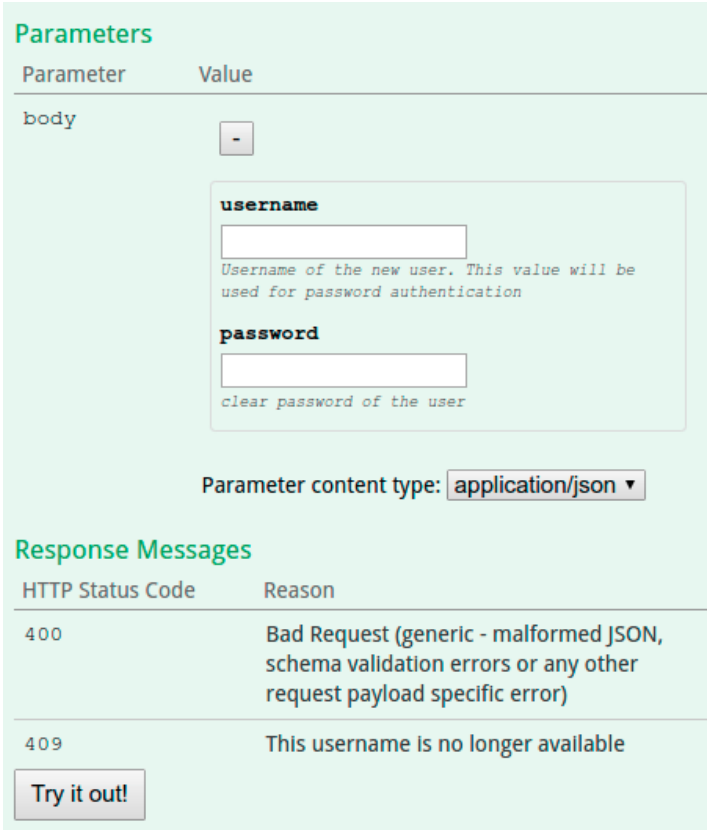

Figure 12: Swagger request

Performing a request with valid values resolves into the response shown in Figure 13 which contains the response body and the the response code.

Figure 13: Swagger response

Performing Requests as authenticated User

Performing a request as authenticated user enforces the user to pass a token along the request. In Swagger this can be done by copying a token and pasting this token in the auth field on the top right corner of the Swagger website. Such a token can be copied for example from the response of a login or registration request, such as in Figure 13 which shows the response of a registration request.

5.4.2 GraphiQL

GraphiQL quickly gained popularity with the rise of GraphQL. It is defined as "A graphical interactive in-browser GraphQL IDE" [31] and provides similar interaction functionalities to GraphQL endpoints as Swagger provides for RESTful endpoints. In order to use GraphiQL within the Hapi framework a plugin named hapi-plugin-graphiql is used. Similar to the hapi-swagger plugin used in Section 5.4.1, the hapi-plugin-graphiql is as well registered in the postinit step on the server startup. However, no further configuration on the endpoints is necessary. Instead GraphiQL leverages the introspection capability to fetch the type definitions written on the server and gives the user the possibility to explore and navigate through those type definitions using the documentation panel of the GraphiQL tool. However, in order to extend those documentation one can add a description field to the type definition on the GraphQL server. As an example, Listing 5.24 extends the type definition made for the *RegistrationType* in Listing 5.11 by adding a description to each of the fields.

```
// route definition for registering new user
const RegistrationType = new GraphQLObjectType({
    name: "Registration",
    fields: () => ({
        tokenType: {
            description: "a string that tells you what type of token it is.
               You should use this type in your Authorization requests.",
            type: new GraphQLNonNull(GraphQLString)
        },
        accessToken: {
            description: "The main authorization token for API usage. This
               token has to be provided with the Authorization HTTP header
               as Bearer for all authenticated calls. Only valid for a
               limited time - see validUntil",
            type: new GraphQLNonNull(GraphQLUUID)
        },
        refreshToken: {
            description: "A token for this user that can be used to get a
               new accessToken using the /auth/token endpoint. This token
               might change over time, so for a new login, or when
               exchanging a refreshToken for a new accessToken a new
```

```
                refreshToken might be generated. Guest users don't receive
                a refreshToken (will be null)",
            type: GraphQLUUID
        },
        expiresIn: {
            description: "Time in seconds how long the Access Token will be
                valid on the server. Guest users will have an accessToken
                that never expires (-1)",
            type: GraphQLInt
        }
    })
});
```

Listing 5.24: GraphQL type definition with description

Figure 14 shows the GraphiQL interface. It consists of a header on the top, an inspection panel on the left site, a results panel in the middle of the screen and a documentation panel on the right site. These three panels can be compared with the parts used to describe a Swagger documentation in Section 5.4.1. The documentation panel can further be toggled by clicking on the arrow button on the upper right corner of the interface.

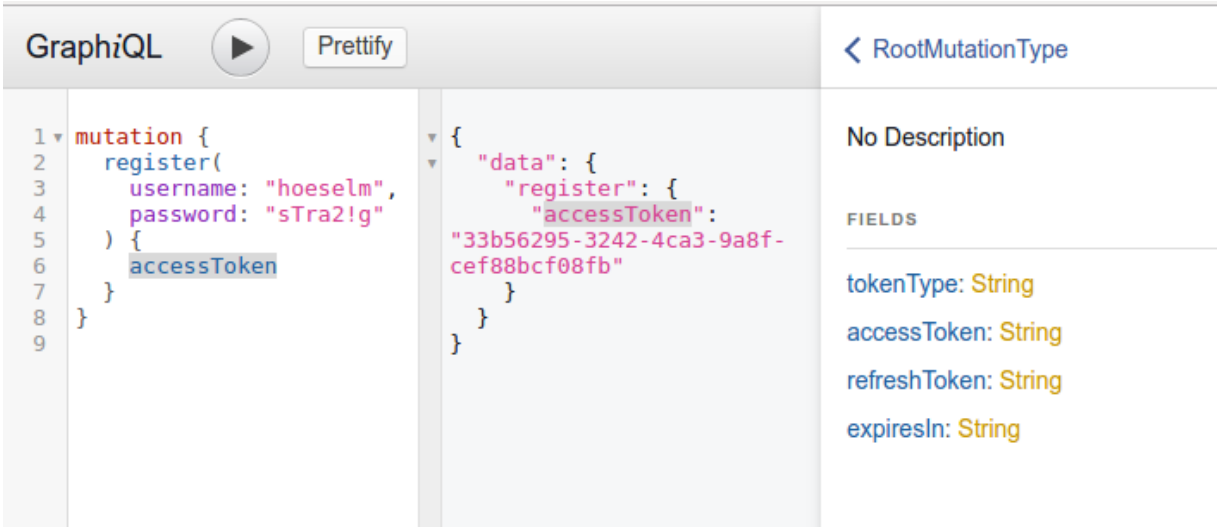

Figure 14: GraphiQL

The inspection panel can be used to inspect the capabilities defined on the GraphQL server and to form a query. In Figure 14 this was done for the registration process. The panel supports auto completion of the keywords as one types and marks unknown fields and errors by underlining them in red. Using the *Prettify* button on the top one can format the query and fix the line intending in order to make the query more readable. This is especially useful in case one typed the query in one line or did not pay attention to the intending while writing the query. The result panel shows the response of a query. Figure 14 for example shows the response of the registration query defined in the query panel. This query was executed by clicking on the *Play* button

on the top of the interface. Because the query specifies to only include the *accessToken* field in the response, the response is limited to this field. In addition the *refreshToken*, *tokenType* and *expiresIn* field could have been requested as well. This can be seen on the documentation panel which allows the user to browse through the documentation.

In order to see the complete description of the *tokenType* one could perform a mouse click on that field. This click will open the description as shown in Figure 15. This way one can click through the complete graph defined on the server till one reaches the leafs of the graph, which are represented by the GraphQL Scalar type.

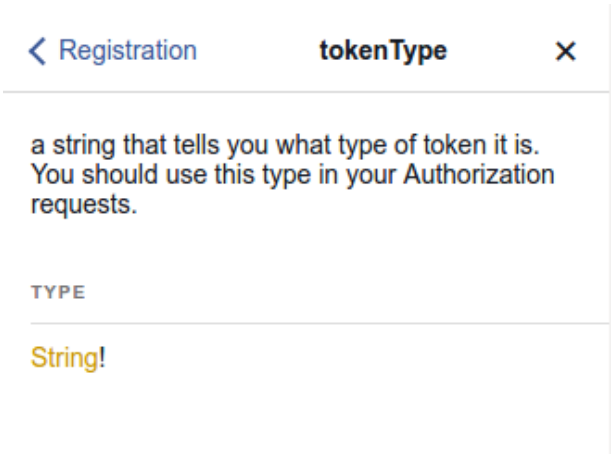

Figure 15: GraphiQL documentation

Performing Requests as authenticated User

GraphiQL, by default, does not support to set an authentication token in the header part of the HTTP request. This means that by default no queries to GraphQL endpoints which enforce authentication can be executed successfully. This is in contrast to Swagger, which allows a user to set an authentication token. While there have been discussions to add a field to the GraphiQL interface in order to set a token or to add a header editor to GraphiQL no agreement and no work was done yet and the discussions are still ongoing [45] [46] [47]. The main reason to not include such functionality into GraphiQL has been to keep GraphiQL generic and to not make assumptions about how the network and server works [45]. However, the GraphiQL plugin on the server allows to add headers to the header object. This technique has been used in [46] to enable authentication in GraphiQL and was added to the configuration of the GraphiQL plugin in this thesis. Listing 5.25 shows the configuration on the server.

```
graphiqlOptions: {
     endpointURL: "/graphql",
     passHeader: "'Authorization': window.__TOKEN ? 'Bearer ' +
        window.__TOKEN : ''" // custom header can be set on the browser
        console
},
```

Listing 5.25: GraphiQL's passHeader field

Through the *passHeader* field the *Authorization* header will be attached to each request from the GraphiQL interface. However, by default the value of the *Authorization* header is empty. This is because the token is parsed from the *__TOKEN* field of the *window* object from the browser which by default does not exist. In order to perform a request with a valid token one has to open the developer tools of the browser and manually set a token before executing the query via the GraphiQL interface. Figure 16 shows how one can set a token using the developer tools. Once a token is set on the *window* object of the browser this token will be parsed, thus making it possible to send authenticated requests.

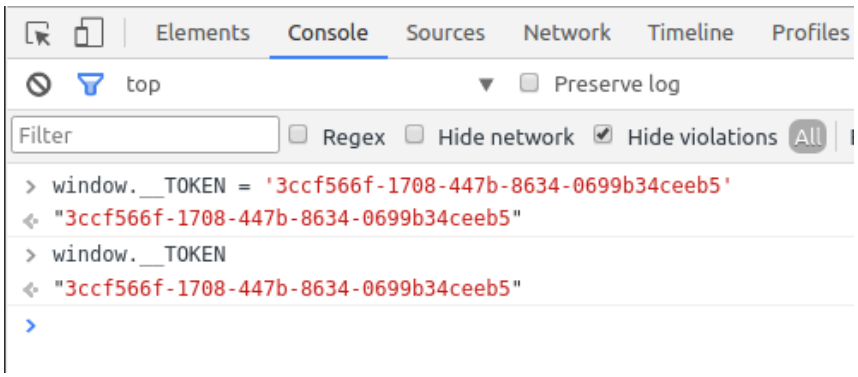

Figure 16: GraphiQL with auth token set via developer tools

Adding Authentication to the GraphiQL Interface

While the process shown in Figure 16 allows to send authenticated requests from GraphiQL to a GraphQL server the process can be tiring and time consuming. In addition this process must be documented and explained to people new to this implementation. An obvious better way is to let the user set the token directly within the GraphiQL interface. This thesis therefore investigates on this problem by forking GraphiQL, which is hosted on Github and allows modification and redistribution of the source code by granting a "non-exclusive, worldwide, royalty-free copyright license to (1) use and copy the GraphiQL software; and (2) reproduce and distribute the GraphiQL software as part of your own software ('Your Software')" [25].

GraphiQL is implemented using React - a library for building user interfaces [50]. React itself is declarative and component based. User interfaces which are implemented using React exist of either one component or are composed of multiple components, depending on the complexity of the interface. Every component however implements a so called *render* function which returns what to display. In addition each component can take input data and maintain its own internal state. Each time this state is changed by either external or internal events the *render* function is triggered in order to return the updated markup [50].

In order to let the user enter an authentication token on the GraphiQL user interface a new component named *AuthBox* is created. This component exists of an input field and a button with the label *Set Token*. Figure 17 shows the component which was added to the header bar of the GraphiQL interface.

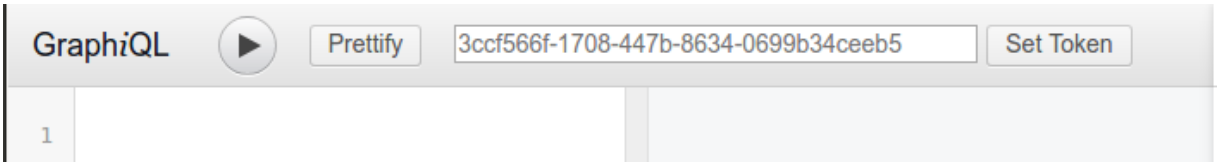

Figure 17: GraphiQL with auth token set via modified interface

The authentication box lets the user insert a token. This token is assigned to the *__Token* field of the *window* object after the user clicks the *Add Token* button or after the user presses the enter key on the keyboard. Listing 5.26 outlines the *render()* function and Listing 5.27 the event handler of the component.

```
render() {
  return (
    <div className="authBox">
      <input
        value={this.state.value}
        onChange={this.handleChange}
        onKeyDown={this.handleKeyDown}
        type="text"
        placeholder={window.__TOKEN || this.props.inputFieldPlaceholder}
      />
      <a
        className={'toolbar-button'}
        onMouseDown={preventDefault}
        onClick={this.handleClick}
        title={this.props.buttonTitle}>
        {this.props.buttonLabel}
      </a>
    </div>
  );
}
```

Listing 5.26: GraphiQL AuthBox render function

```
handleChange = event => {
  const value = event.target.value;
  this.setState({ value });
}

handleKeyDown = event => {
```

```
    if (event.keyCode == 13) {
      this.handleToken();
    }
  }

  handleClick = () => {
    this.handleToken();
  }

  handleToken = () => {
      window.__TOKEN = this.state.value;
      this.setState({
        value: ''
      })
  }

}
```

Listing 5.27: GraphiQL AuthBox event handler

Each time a user types something into the input box the *onChange* handler is called. This handler updates the state of the component which in turn triggers the *render()* function which returns the rendered markup including the characters typed by the user. In addition to the *onChange* handler a *onKeyDown* handler is registered, which checks if the key typed by the user has the code *13*, which represents the enter key. In case the enter key was pressed the check evaluates to true and the *handleToken* function is called which sets the *__TOKEN* field of the *window* object to the value currently stored on the state. The value on the state is then set to an empty string in order to allow the user to insert a new token. Beside the *input* box with the two handlers *onChange* and *onKeyDown* an HTML *<a>* element which defines an hyperlink gets rendered as well. This *<a>* elements represents the button with the label *Set Token*. Each time a click on this button is performed the *handleClick* handler is called, which - similiar to the *handleKeyDown* handler - calls the *handleToken* function in order to set the *__TOKEN* field of the *window* object. Once this token is set it is used within each request perfomed within the GraphiQL interface, thus enabling the user to sent authenticated requests to the GraphQL server.

In order to show the *AuthBox* component on the GraphiQL interface the base component called *GraphiQL* creates an instance of it. A CSS class is furthermore added to configure the component with proper styling. [48] references the Github page which hosts the forked GraphiQL project and all changes made to the project.

Adding the updated GraphiQL Interface to the Microservices

The microservices in this thesis which communicate via GraphQL are based on the graphql-

server library. Using this library, it is very simple to register a graphql server within the Hapi framework. This library also provides a module for the GraphiQL tool. However, instead of directly using the source code of GraphiQL this module exists of a file which loads GraphiQL via Content Delivery Networks (CDNs). In order to let this module use the GraphiQL implementation with authentication token support, this version was uploaded to the authors server with the domain name *www.hoeselm.at*. The graphql-server library hosted on Github was forked and the entries which pointed to Facebooks GraphiQL implementation have been changed to point to the authors server. Listing 5.28 outline the updated CDN links for the GraphiQL JavaScript and CSS file along the CDNS links for the *fetch*, *react* and *react-dom* links which are kept.

```
<link href="//www.hoeselm.at/graphiql-hoeselm/graphiql-hoeselm.css"
    rel="stylesheet" />
<script src="//cdn.jsdelivr.net/fetch/0.9.0/fetch.min.js"></script>
<script src="//cdn.jsdelivr.net/react/15.0.0/react.min.js"></script>
<script src="//cdn.jsdelivr.net/react/15.0.0/react-dom.min.js"></script>
<script
    src="//www.hoeselm.at/graphiql-hoeselm/graphiql-hoeselm.js"></script>
```

Listing 5.28: graphql-server CDN update

Using the modified GraphiQL library in addition to the modified graphql-server library allows the user to perform authenticated requests to the GraphQL server used in this setup by setting an authentication token via the GraphiQL interface. [49] references the Github page which hosts the forked graphql-server project and all changes made to the project.

6 Results

6.1 Performance of the IPC Mechanism

The performance of the microservices architectures implemented in this thesis was measured by querying several endpoints located on the *Project service*. The requests however have not been directly sent to the *Project service* but instead have been sent to the *API gateway* which redirected the requests to the *Project service*. In addition the *Project service* forwarded the authentication token attached to each request to the *User service*. This service then validated the token and returned the result of the validation back to the *Project service*. In case the validation was successful the *Project service* executed its business logic and returned a response to the API gateway which redirected the response back to the client. These steps have been illustrated in Figure 10 and are summarizes in the listing below:

1. Measurement starts.
2. Client sends a request to the *API gateway*.
3. The *API gateway* forwards the request to the *Project service*.
4. The *Project service* parses the payload from the request and extracts the authentication token from the header of the request.
5. The *Project service* then sends a new request containing the authentication token to the *User service*.
6. The *User service* validates the authentication token and sends the result back to the *Project service*.
7. The *Project service* parses the result.
   - In case the validation is successful the *Project service* executes its business logic and generates a response containing the requested data.
   - In case the validation is not successful a response containing the error is generated.
8. The *Project service* sends the response to the *API gateway*.
9. The *API gateway* sends the response back to the client.
10. The client receives the response.
11. Measurement stops.

This procedure was executed with a script written in Python which used a library named Locust in order to record several metrics. The script was introduced in Section 5.2 and executed one request every 100 milliseconds. In total there have been between 200 and 300 requests for each endpoint which are listed in Table 1 of Section 5.2. For each of those requests the count, the median and average response times in milliseconds as well as the average content size in bytes have been recorded. Table 2 lists the result of the measurements performed within the RESTful microservices while Table 3 lists the result of the measurements performed for the microservices based on GraphQL.

ID	Method	Requests	Median (ms)	Avg (ms)	Payload (Bytes)
1	create project	237	69	72	213
2	create language	263	51	53	216
3	get project	278	48	48	6453
4	get language	262	43	64	216
5	get project ids	226	44	46	5504
7	get project with languages	263	5800	6099	28375
8	get project with languages - only REST	248	53	74	31189
9	get github stats	10	1600	1598	753

Table 2: REST Performance Measurement Results in Hapi with Joi validation

6.2 Schema Validation

This thesis researched the process of schema validation within a RESTful microservices architecture and a GraphQL based microservices architecture using the methods described in Section 3.3. The research revealed several differences between those two API layers. REST as an architectural style does not specify if and how schema validation should be performed. However, as an important part to build stable microservices schema validation plays an important role. In order to setup schema validation for a RESTful API an external library named Joi has been used. This library is very popular within the Hapi framework and can be integrated very easily. However, using such a library makes it necessary to first type the schemas which should be used for the payload validation. In contrast to REST GraphQL enforces schema validation. Therefore no external libraries are needed. In terms of the validation process itself GraphQL uses the types defined within GraphQLs type system. Because of that no additional schemas need to be written.

ID	Method	Requests	Median (ms)	Avg (ms)	Payload (Bytes)
1	create project	243	50	53	253
2	create language	240	34	36	257
3	get project	263	39	40	5819
4	get language	208	28	30	253
5	get project ids	222	49	50	5901
6	get project ids - only GraphQL	250	40	41	6257
7	get project with languages	258	41	43	28850
9	get github stats	12	690	706	784

Table 3: GraphQL Performance Measurement Results

Schema validation can be done in many different ways. This thesis researched schema validation with a conservative validation strategy and a more liberal validation strategy which followed the guidelines of the robustness principle. In order to test those schemas multiple requests with varying payloads have been sent to the *registration* endpoint of the microservice architectures. Those payloads have been:

- *Payload 1*: Payload conform to the schema defined on the server
- *Payload 2*: Payload with a field attached to it which was not defined in the schema
- *Payload 3*: Payload with a value of type *Number* for the *username* field while the schema expected a value of type *String*

For the RESTful microservices architecture three different scenarios have been tested. The Listing below lists those scenarios and the response from the server.

- RESTful API without schema validation in place:
  - Result for *Payload 1*: Request was successfully processed as all fields required by the backend have been attached to the payload.
  - Result for *Payload 2*: Request was successfully processed as all fields required by the backend have been attached to the payload. Additional fields attached to the payload have not been processed.
  - Result for *Payload 3*: Request generated a *"Internal Server Error"* exception on the server as the backend was unable to process the given type.

- RESTful API with a conservative schema validation in place:
  - Result for *Payload 1*: Schema validation was successful because the payload was composed of exactly those fields defined in the schema.
  - Result for *Payload 2*: Schema validation threw an error because the payload contained a field not defined in the schema.
  - Result for *Payload 3*: Schema validation threw an error because the payload contained a value of type *Number* while a value of type *String* was expressed in the schema definition.
- RESTful API with a liberal schema validation in place:
  - Result for *Payload 1*: Schema validation was successful because the payload was composed of exactly those fields defined in the schema.
  - Result for *Payload 2*: Schema validation was successful because all fields defined in the schema have been attached to the payload and all additional fields attached to the payload have been ignored.
  - Result for *Payload 3*: Schema validation was successful because the schema was changed to support a range of types for the field *username* and the value of the field *username* was within that range.

For the GraphQL based microservices architecture a conservative and a liberal schema validation was tested as well. However, GraphQL by default validates a payload using the type system and enforces each field within the payload to match the type defined within the type system. Such approach can be seen as conservative. In order to support a more liberal schema validation process within GraphQL the GraphQL library itself was changed. Those changes have been:

- A new scalar named *GraphQLStringOrInt* which allowed a value to be either of type *String* or of type *Number* was implemented.
- The *KnownArgumentNames* validation rule was removed.
- The *FieldsOnCorrectType* validation rule was removed.

Testing a conservative and a more liberal validation strategy within GraphQL led to the following results:

- GraphQL endpoints with a conservative schema validation in place:
  - Result for *Payload 1*: Schema validation was successful because the payload was composed of fields defined on the GraphQL type.
  - Result for *Payload 2*: Schema validation threw an error because the payload contained a field not defined on the GraphQL type.

  - Result for *Payload 3*: Schema validation threw an error because the payload contained a value of type *Number* while the GraphQL type expected a value of type *String*.

- GraphQL endpoints with a liberal schema validation in place:
  - Result for *Payload 1*: Schema validation was successful because the payload was composed of fields defined on the GraphQL type.
  - Result for *Payload 2*: Schema validation was successful because all fields defined on the GraphQL type have been attached to the payload and all additional fields attached to the payload have been ignored.
  - Result for *Payload 3*: Schema validation was successful because the GraphQL type was changed to support a range of types by using the *GraphQLStringOrInt* scalar and the type of the value was within that range.

Table 4 summarizes the results by outlining the HTTP codes returned from the server.

API layer	Validation Pattern	Payload conform to Schema	Payload with an additional field	Payload with a value of wrong type
REST	without validation	200 OK	200 OK	500 Internal Server Error
REST	conservative validation	200 OK	400 Bad Request	400 Bad Request
REST	liberal validation	200 OK	200 OK	200 OK
GraphQL	conservative validation	200 OK	400 Bad Request	400 Bad Request
GraphQL	liberal validation	200 OK	200 OK	200 OK

Table 4: Schema Validation Result

6.3 API Documentation with Swagger and GraphiQL

Analyzing the integration and usage of popular developer tools to document and test the API of the microservices was one of the key tasks of this thesis. The results show that both tools can be used to auto generate an API documentation which can be extended with additional information and that both tools can be used to send unauthorized as well as authorized requests to the API.

Writing an API documentation can be a time consuming task. Therefore both tools have been investigated to auto generate such an API documentation for which the process varies slightly. While GraphiQL introspects GraphQLs type system in order to generate a documentation about the available types Swagger was configured to use Joi in order to retrieve those information from the schemas. Additionally, both tools allow to extend the auto generated configuration by setting a description on the fields which are defined within the GraphQL type definition or within the Joi schema definition. Figure 11 and Figure 14 showed a screenshot of how such a generated API documentation is represented in Swagger and GraphiQL respectively.

Another advantage of Swagger and GraphiQL is to simply test the documented APIs. In case of Swagger this can be done by filling out and submitting a form in order to create a request whereas in GraphiQL one can interactively create queries which are then attached to the payload of the request. Figure 12 and Figure 13 demonstrated how such a request is created and how the response is represented in Swagger whereas Figure 14 demonstrated this for GraphiQL.

While both tools can be used to create requests GraphiQL by default has no option to attach an authentication token to the header field of the requests. Therefore GraphiQL is by default not capable to successfully execute requests for which a user needs to be authenticated. This is in contrast to Swagger, which allows to set an authentication token in the top right corner of the web interface. Therefore the GraphiQL library and its interaction with GraphQL was analyzed in order to proof that necessary changes can be made in order to support authenticated calls within GraphiQL. In order to realize this two changes have been made:

- Configuring the *passHeader* field to read the authentication token from the *window object* of the browser. Listing 5.25 outlined the step.
- Adding an input field and a button with corresponding logic in order to enable the user to set an authentication token on the *window object* of the browser. Figure 17 showed the updated GraphiQL interface.

Through this two changes it is possible to perform authenticated requests. This is done by filling out the input field and clicking the button or pressing the enter key in order to set the authentication token on the *window object*. This *window object* is then used to parse the authentication token and to add it as header field on the request before the request is sent to the server.

7 Discussion

7.1 Performance of the IPC Mechanism

This thesis researched the performance of REST and GraphQL as IPC mechanism by conducting measurements on several endpoints. Those endpoints have been defined in Table 1 of Section 5.2 and the results of the performance measurements have been presented in Table 2 and Table 3 of Section 6.1. By analyzing those results one can see that the overall response times are slightly lower when using GraphQL as IPC mechanism. Figure 18, Figure 19 and Figure 20 visualizes those results using the average response times from Table 2 and Table 3 of Section 6.1.

Response times of the endpoints with id 1, 2, 3 and 4

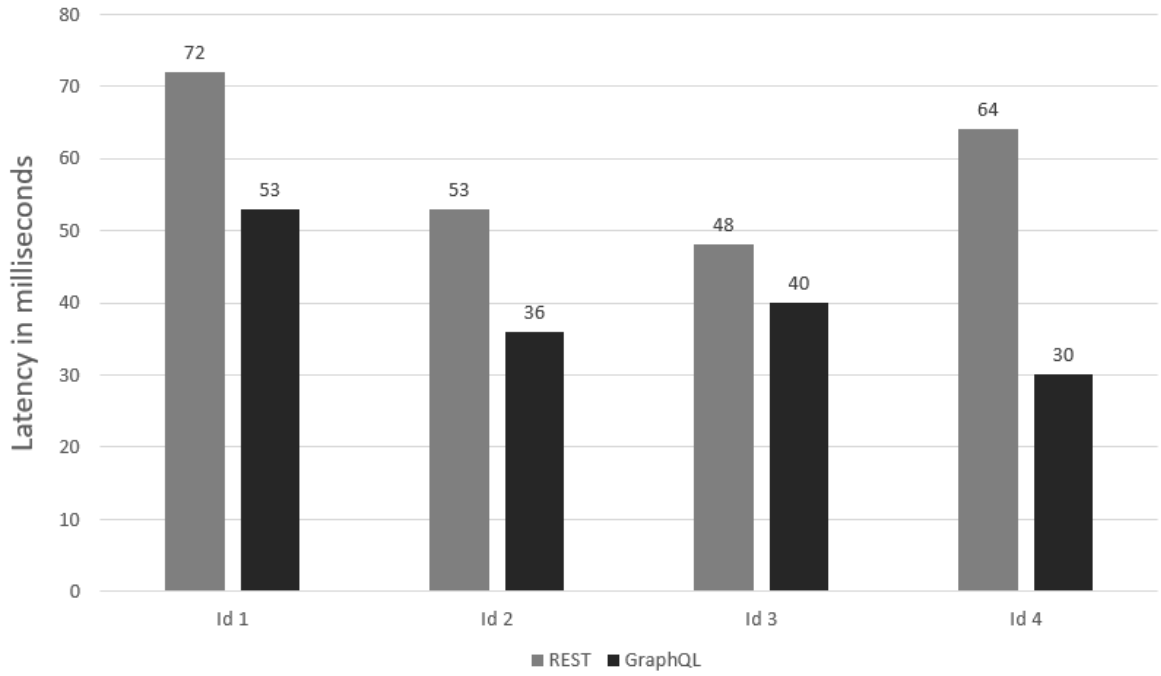

Figure 18: Performance comparison of endpoint 1, 2, 3 and 4

Figure 18 compares the endpoints with the ids 1, 2, 3 and 4. Those endpoints produce the same actions on both microservices architectures in terms of the number of requests performed within the microservice architecture in order to generate a response and in terms of the structure

of the payload returned within the response. By interpreting the diagram one can see that the response time of GraphQL is between 8 milliseconds and 34 milliseconds lower compared to REST. The reason for that is because the GraphQL library handles requests more efficient then the RESTful API used within this thesis. That API however was formed by Hapi as web framework and Joi as schema validator. The results therefore do not represent a RESTful API in general, but instead only the implementation of Hapi in combination with Joi.

Response times of the endpoints with id 5 and 6

Figure 19 shows the results for the endpoints with the ids 5 and 6. In order to highlight a potential issue within GraphQL two comparison have been pointed out in the diagram. The first one compares the REST endpoint with id 5 with the GraphQL endpoint with id 5. The second comparison in addition compares the REST endpoint with id 5 and the GraphQL endpoint with id 6.

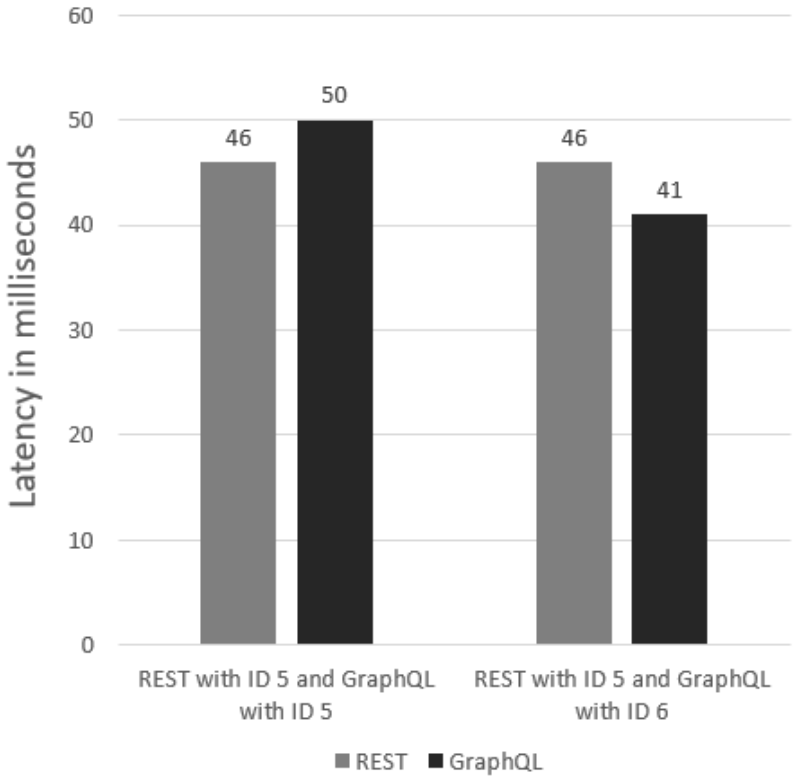

Figure 19: Performance comparison of endpoint 5 and 6

Both endpoints fetched the project ids from the backend. However, two ways of fetching those project ids have been implemented within GraphQL while only one endpoint was implemented for the RESTful API. The way the RESTful API handled this was by implementing a database query on the server which only queried for the *uid* attribute within the *Project* table. Listing 7.1 outlines the controller code with the database query implemented in the ORM Mapper

Sequelize. In case the query was successful the list of project ids was returned, otherwise an error was thrown.

```
export function getProjectIds(request, reply) {
    return storage.models.Project.findAll({
        attributes: ["uid"]
    }).then(res => {
        return reply(res);
    }).catch(err => {
        return reply(Boom.create(500, err));
    });
}
```

Listing 7.1: Query implemented in REST for retrieving a list of project ids

In GraphQL however the endpoint which was used to query a list of projects with its languages was also used to query the list of project ids. This seems reasonable as one key feature of GraphQL is to ask for only those fields one is interested in. Listing 7.2 and Listing 7.3 list the GraphQL queries for retrieving a list of projects with its languages and for retrieving a list of project ids only - which both make use of the *projects* endpoint.

```
{
  projects {
    name
    userUid
    uid
    ProjectLanguages {
      name
      uid
    }
  }
}
```

Listing 7.2: Query implemented in GraphQL for retrieving a list of projects

```
{
  projects {
    uid
  }
}
```

Listing 7.3: Query implemented in GraphQL for retrieving a list of project ids

The way this data is retrieved from the database is however not very efficient. Listing 7.4 lists the controller code for the *projects* endpoint.

```
export function getProjects() {
```

```
    return storage.models.Project.findAll({
        include: [{
            model: storage.models.ProjectLanguage
        }]
    }).then(res => {
        return res;
    }).catch(err => {
        return Boom.create(500, err);
    });
}
```

Listing 7.4: Serverside implementation to fetch projects

For each requests made to the projects endpoint all *Projects* with their attributes as well as the *Languages* with their attributes are queried from the database. GraphQL then ignores all fields which are returned by the controller code but have not been asked for within the GraphQL query, returning only those fields which have been asked for within the GraphQL query. While this database query is necessary to fulfil the request from Listing 7.2 it is inefficient to use the same query for fulfilling the request from Listing 7.3, resulting in higher response times compared to the implementation used within REST. Therefore an additional GraphQL endpoint named *projectids* was implemented using the same controller code which has been used within the REST implementation. This additional endpoint has the id 6. The results show that while the response time for the GraphQL endpoint with id 5 is 4 milliseconds higher compared to REST the response time for the GraphQL endpoint with the id 6 is 5 milliseconds lower compared to REST.

While this thesis implemented two separate endpoints in order to investigate this problem it would not be meaningful to do so in a real world implementation. Instead, the controller would be refactored in a way to dynamically create efficient database queries based on the fields which have been specified within the GraphQL query. However, implementing this can be hard and time consuming. Consequently new libraries have been created which handle this task. In case of the object relational mapper Sequelize one popular library would be *graphql-sequelize* [30].

Response times of the endpoints with id 7, 8 and 9

Figure 20 analyzes the response time when retrieving the projects with their associated languages and when retrieving project statistics from Github. The first comparison within Figure 20 shows a huge difference between the response times. The way that endpoint was implemented within GraphQL is shown in Listing 7.2 which specifies that the *Projects* with their *Languages* should be queried from the database using a single statement. In contrast the RESTful API handled this differently. The client first requested a list of projects. The response to this request then returned the requested list of projects containing the attributes *name*, *userUid*, *createdAt*, *updatedAt* and *uid* as well as a list of *LanguageIds*. For each of those *LanguagesIds* a separate request was performed in order to retrieve all attributes of the languages. In case a project had

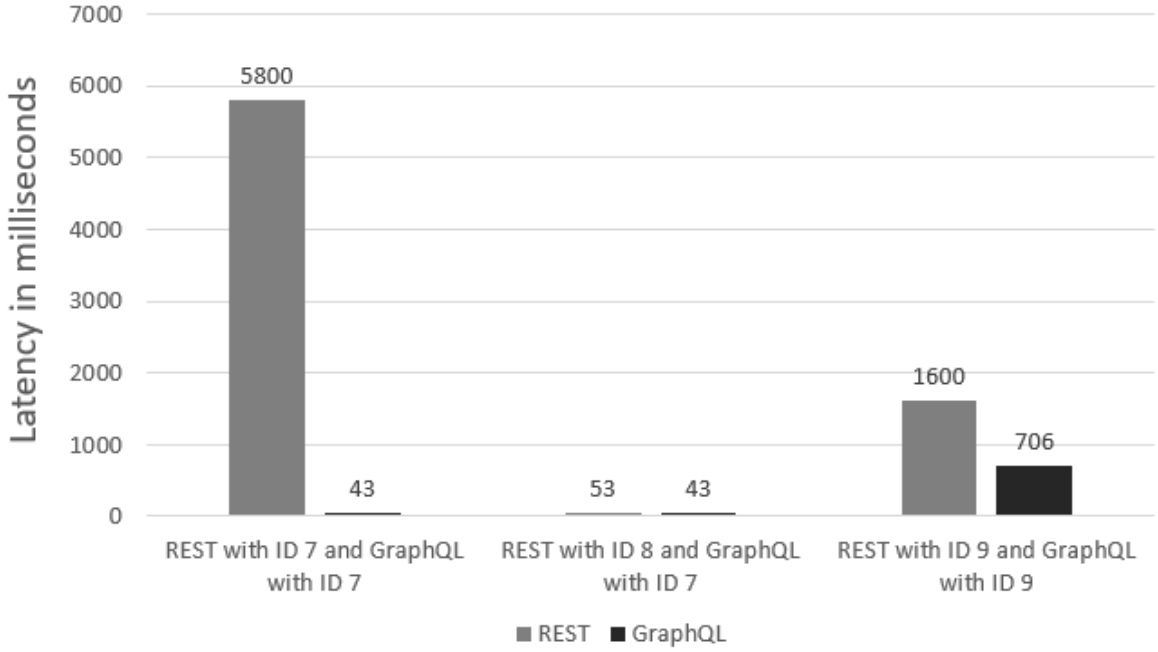

Figure 20: Performance comparison of endpoint 7, 8 and 9

ten languages eleven calls had been performed - one to retrieve the *Project* and ten to retrieve the *Languages*. Implementing endpoints in that way is common for RESTful APIs, because returning all data within one response would require more complex backend implementations and database queries and in case a client is only interesting in the attributes of the *Project* the request would lead to an unnecessary large payload. However, if there is the need to retrieve a large dataset within one request an additional endpoint can be implemented. This was done through the endpoint with the id 8. In contrast to the endpoint with the id 7, which on average took 5757 milliseconds longer than the GraphQL response due to the number of requests which had been performed, the REST endpoint with the id 8 took only 31 milliseconds longer.

However, for the endpoint with the id 9 the same issue arises. While the RESTful implementation had to perform multiple requests between the microservices architecture and Githubs REST API only one request was necessary between the microservices architecture based on GraphQL and Githubs GraphQL API. Because this time the endpoint was not located within the microservices architecture but was located on Github an optimization of the API endpoint could not be made.

Differences in payloads

Analyzing the payloads from Table 2 and Table 3 reveal that the payload returned within a GraphQL response is on average larger than the payload returned within a RESTful API response. This is because GraphQL returns the result to a query and encodes this information within the response. In contrast REST simply returns the requested payload and the informa-

tion about which data is sent is encoded within the URL. Listing 7.5 and Listing 7.6 demonstrate such a payload and the URL used to retrieve the payload. In those examples GraphQLs payload is 100 Bytes long whereas the payload in REST is only 66 Bytes long. In contrast the URL of REST discloses that the language stats of github have been retrieved whereas the URL of GraphQL only states that the *graphql* endpoint was queried.

```
URL: /github/language/stats
Payload:
[
   {"language":"C++","occurence":5},
   {"language":"C","occurence":5}
]
```

Listing 7.5: REST Payload

```
URL: /graphql
Payload:
{"data": {
   "githubLanguageStats":[
      {"language":"C++","occurence":5},
      {"language":"C","occurence":5}
   ]
}}
```

Listing 7.6: GraphQL Payload

7.2 Schema Validation

This thesis evaluated the process of schema validation by investigating two different schema validation strategies. The first one was a conservative validation strategy for which the payload had to be conform to the specified schema. The second one was a more liberal validation strategy which followed the guidelines of the robustness principle and thus also successfully validated certain payloads which have not been conform to the specified schema. In addition to this the RESTful API layer was also tested with no schema validation process configured. In order to get further insights on those strategies three requests with different payloads have been formulated and tested. The payload of the first request was conform with the defined schema, the second request had an additional field configured on its payload and the third request had a value of different type attached to its payload. While those requests with a payload conform to the schema had been processed successfully, those requests with a payload not conform to the schema partially failed. Section 6.2 outlined the results. This Chapter further discusses those results.

Microservices without Schema Validation

Section 5.3.1 implemented and tested a REST endpoint without any schema validation and outlined the responses from the server. This endpoint successfully processed the first two requests but threw an error on the first request as that request had a payload with a value of different type attached to it which the server was not able to process. While the sections below will also discuss other errors, this error was the only one thrown with an HTTP status code 500 - indicating an *"Internal Server Error"*. The reason for this was because no schema validation was in place to validate the payload of the query before processing it. Because this error was thrown while the backend was executing its business logic, the request could have left databases in an inconsistent state or could have left damage to the application in many different ways. While it may seem obvious that the server will throw an error when a client sends requests with malformed payloads this case highlights the importance of schema validation.

Conservative and Liberal Schema Validation

Both the RESTful API as well as the GraphQL API have been configured with conservative and liberal schema validation and both APIs returned the same results. While the liberal validation strategy accepted additional fields and supported a range on types for specific fields the conservative strategy rejected those results and returned an HTTP error with status code 400 to the client. It may seem reasonable to throw an error if the request is not conform to the defined schema, thus using a conservative validation strategy. However, using a more liberal validation strategy can lead to a more robust system for example by following the guidelines of the robustness principle. Section 2.9 introduced the term and explained that such systems gain a higher level of decoupling and make it easier to perform changes. The following listing highlights such a change and outlines the difference between a conservative and a more liberal validation strategy by first implementing an endpoint and then changing that endpoint:

1. A new endpoint is implemented:
    - A new endpoint named *registration* is added to the system in order to allow new users to signup. That endpoint expects that the request contains a payload with a *username*, a *password* and a *subscription plan*.
    - A call to that endpoint is implemented in clients in order to signup new users. Using such clients will therefore result in requests with a payload consisting of a *username*, a *password* and a *subscription plan*.
    - Performing schema validation at this point will succeed for both strategies as the requests contain the payload expected by the server.
2. An existing endpoint is changed:
    - Based on a new demand all users should be assigned with a default subscription plan, making it unnecessary to choose one at registration time. In order to reflect

this new demand the field *subscription plan* is removed from the *registration* endpoint and from the clients as well. Updated clients will therefore only send requests with a payload consisting of a *username* and a *password*.

- However, it is technically not possible to perform the API change and a rollout of the updated clients at the exact same time. Moreover, browsers may use a cached site instead of retrieving an updated one or smartphone apps and desktop applications may depend on user interaction in order to be updated. Therefore it is very likely that the backend receives requests from old clients which still have the *subscription plan* included in their payload as well as from new clients which do not have the *subscription plan* included in their payload anymore.
- A conservative schema validation which does not allow fields which have not been defined on the schema will now throw an error for requests performed from old clients as those requests still contain a payload consisting of the fields *username*, *password* and *subscription plan* while the schema only allows the two fields *username* and *password*. With a more liberal validation strategy additional fields can be ignored. In this particular case the validation would succeed as the two required fields *username* and *password* are attached and all additional fields are ignored.

It should be noted that removing or adding fields from an existing API endpoint is not the only way to change the system. In RESTful application APIs often encode the version of the endpoint within their URL. In order to signup a user a client could therefore sent a request to an endpoint named */api/v1/registration* in first place. Changes to that endpoint could be made public by exposing an additional endpoint named */api/v2/registration*. However, introducing a new endpoint for each change will most likely result in a system with a lot of endpoints which all need to be maintained. GraphQL on the other hand handles this differently. Its default validation strategy can be seen as conservative as all fields the client requests within a query need to be defined on the GraphQL server. However, because GraphQL is also query language a client has to specify the fields it wants the server to respond. This is in contrast to REST where those fields are hardcoded within the server. Adding new fields within GraphQL is therefore straightforward. This new fields can then be queried by the client at a later release. Removing fields however will result in errors as the thesis demonstrated in Section 5.3. The way GraphQL handles this by default is to mark fields which should be removed as deprecated. Those fields are then not exposed to the client when introspecting the types but can still be queried. In order to have a more liberal validation process the GraphQL library itself was forked on Github and necessary changes have been applied. Those changes have been outlined in Section 5.3.2. In addition to removing fields the thesis also analyzed whether the liberal validation strategy can be changed to support a range of types for a certain field. In particular it was researched if the validation strategy can be configured to allow a value of type *String* and of type *Number*. For the RESTful API this change was pretty simple as Joi provided the necessary operation. In GraphQL such a change is more complicated. The way this was done for this thesis was to

implement an additional scalar named *GraphQLStringOrInt*.

In the end both validation strategies have their advantages and disadvantages. Based on my background I recommend a validation strategy which allows to add new fields as this strategy allows to add new features without creating breaking changes and without introducing new endpoints. While I also recommend a validation strategy which allows to remove existing fields it is not very common in practice to do so. Based on that conclusion I prefer to implement a validation strategy which supports adding and removing fields for the RESTful implementation. However, because GraphQL by default already allows adding new fields and because removing fields is not a use case which is needed very often I recommend to keep the default GraphQL implementation as the advantage of having type safety has more value then being able to occasionally remove fields.

7.3 API Documentation with Swagger and GraphiQL

The results in Section 6.3 revealed that both Swagger and GraphiQL can be used to auto generate an API documentation and to send requests to that API. That said, GraphiQL seems to be tailored to this exact need while Swagger is a more generic framework which offers additional functionality. One such useful functionality is that Swagger views a request as a curl command. This is very handy as this command can be used for further testing on the command line or within other programs. Other functionality includes an editor to add new endpoints or to edit existing ones. In addition Swagger allows to store an authentication token which is used within the header field of a request while GraphiQL does not provide such functionality. The GraphiQL library was therefore forked on Github and changes to it have been made in order to support setting an authentication token in order to perform authenticated requests.

8 Management Resume

This thesis was written for aaa - all about apps GmbH, which evaluated the use of microservices for their infrastructure. As a company focused on developing mobile application aaa - all about apps GmbH already has a lot of experience with REST and GraphQL for the client server communication and had therefore a particular interest in using those API layers for the inter-process communication between microservices as well. For this task two microservices architectures have been created. Both architectures consisted of three microservices and differentiated only in their API layer. While the first microservices architecture was based on a RESTful API, the second microservices architecture implemented GraphQL as IPC mechanism. This resume recaps the findings and outlines the advantages and disadvantages of both API layers.

Both microservices architectures have been analyzed in respect to their performance and to their ability to validate a payload in order to prevent processing of malformed requests. In terms of performance GraphQL outperforms REST as it was optimized for performance and flexibility. While REST is designed to exchange a representation of a predefined resource GraphQL leverages a query language specifically designed for the exchange of data. With this approach a client can ask for exactly those fields he or she is interested in, thus reducing the payload and the number of requests required to satisfy the caller. But even in cases where the exact same amount of data is exchanged, GraphQLs JavaScript implementation responded on average between 10 milliseconds and 30 milliseconds faster then the REST implementation within the Hapi framework when used in combination with Joi. Recalling that a GraphQL request on average took between 30 milliseconds and 50 milliseconds this is an improvement of between 25 and 50 percent.

However, in terms of schema validation the microservices architecture based on REST allowed to easily modify the process while this was not possible in GraphQL. The reason therefore is that REST as an architectural style has no standard for schema validation while GraphQL leverages its type system to perform a validation of payloads. In case of REST a library named Joi was used. This library not only allowed to strictly validate a payload using a predefined schema but also allowed to ignore fields which have been attached to the payload but have not been defined on the schema. In addition it was also possible to allow multiple types for a specific field. In order to add such exceptions within GraphiQL the library itself needed to be changed. While those changes proofed that its possible to change the validation process it also showed that it is more complicated compared to Joi. Therefore, in case the default validation process of GraphQL is to strict, it may be a better option to use REST as IPC mechanism and Joi as library for the schema validation.

9 Summary

This thesis evaluated the inter-process communication between microservices using REST and GraphQL as API layers. For this evaluation two microservices architectures have been developed, with both microservices architectures differ only through the used API layer. Such a microservice architecture was based on three services, namely an *API gateway*, a *User service* and a *Project service*. In addition Github - which offers a RESTful API as well as a GraphQL API - was used as third party provider. Both microservices architectures have been investigated in terms of performance and in terms of their schema validation process. Furthermore the most popular developer tools - namely Swagger and GraphiQL - have been researched.

The performance results reveal that the response time of the microservices architecture based on GraphQL is on average between 10 milliseconds and 30 milliseconds lower than the response time of the microservices architecture which implements a RESTful API. In addition, as GraphQL does not only access the properties of resources but also resolves references between those resources it is capable of retrieving complex data structures within one request whereas multiple request need to be performed within a RESTful API. This results in overall response times which are multiple times lower compared to REST. However, because GraphQL lets the client define which properties should be returned the implementation on the server can be more complex. As a consequence the business logic of such an API needs to be implemented in a way to only retrieve those properties which have been requested within the GraphQL query in order to be performant. This is in contrast to REST which always returns the same properties for a specific resource, making it easier to implement the endpoint as those properties can be hardcoded within the source code.

It should be noted that the performance results presented within this thesis have been produced by using GraphQLs JavaScript implementation for the GraphQL API as well as Hapi and Joi for the RESTful implementation and may differ when implemented with a different technology stack. However, while it might be possible that another RESTful API implementation reveals lower response times then the RESTful API implementation used within this thesis, GraphQLs advantage of retrieving multiple resources within one request will almost always outperform REST, unless the REST endpoint itself is implemented in a way that it returns multiple resources. Such an implementation however can lead to a huge payload containing data the client is not interesting in and can eventually create an unnecessary high load on the backend due to multiple database or API queries for example.

The investigations on the schema validation process revealed that GraphQL by default is not as flexible as Joi, which was used as schema validation library within the RESTful microservices architecture. While GraphQLs default validation strategy can be seen as conservative, the validation strategy used with REST can be easily adapted to support a conservative as well as a more liberal validation strategy. In contrast to the conservative validation strategy the liberal validation strategy ignored fields which have been attached to the payloads but have not been specified within the schema. In addition the liberal validation strategy was also configured to

support a range of types for a single value. In order to proof that GraphQL can be designed to support such a liberal validation strategy too, the library itself as well as the typed-graphql project have been studied and necessary changes have been made. With this changes in place GraphQL showed the same behavior as REST for both the conservative and the liberal validation strategy.

In terms of developer tools Swagger and GraphiQL have been analyzed and it was shown how both tools can be used to auto generate and manually extend an API documentation and how these tools can be used to send requests to the APIs. While Swagger is a very popular tool used within many RESTful API services GraphiQL quickly became the de facto standard tool for the GraphQL counterpart. However, while Swagger offers additional functionalities, such as storing an authentication token, GraphiQL is lacking those features. This thesis therefore studied the graphiql and the graphql-server library in order to extend GraphiQL with a feature to let users store authentication tokens. With those changes in place the thesis proofed that both Swagger and GraphiQL can be used to test APIs as unauthorized and as authorized user.

While this thesis reveals several interesting insights on REST and GraphQL as IPC mechanism it also led to changes of four libraries within the GraphQL ecosystem. Those changes have been made by the author of this thesis after a comprehensive research was performed through which it was discovered that multiple discussions are already taking place but no work was done yet. The changed libraries are hosted on the authors Github account. The first request was to implement a liberal validation strategy within GraphQL. Related discussions to this can be viewed in [40] and [41]. The implementation affected the *graphql-js* and *typed-graphql* library. The second request was to enable GraphiQL to send authorized requests using a *Bearer* authentication token. Related discussions can be viewed in [46] and [47]. The request was realized by changing parts of the *graphiql* and *graphql-server* library.

9.1 Future Work

This thesis evaluated the performance of REST and GraphQL using NodeJS, Hapi, Joi and the JavaScript implementation of GraphQL. In addition, implementations which use different technology stacks - for example written in Java, C++ or Ruby - could be evaluated and compared to the results of this thesis. Moreover, the interaction of microservices within a microservices architecture where each microservice is based on a different technology could be analyzed.

In order to investigate a liberal validation strategy within GraphQL two libraries have been changed. Those libraries, namely *graphql-js* and *typed-graphql*, are currently hosted on the Github page of the author [42] [43]. In order to integrate those changes into the *upstream* repository a solution must be found to support the default validation strategy as well as the liberal validation strategy. One possible solution would be to introduce a flag which indicates the preferred validation strategy for a specific endpoint. Another possible solution would be to add a configuration object which can be used to specify in detail which validation rules should be applied and which validation rules should be ignored. In order to find the best solution this

decision should be discussed within the community.

The thesis also described the process on how to set an authentication token as header field within the GraphiQL interface, for which the two libraries *graphiql* and *graphql-server* have been changed. Similar to [42] [43] those libraries are hosted on the Github page of the author and can be revisited in [48] [49]. However, in order to integrate those changes into the *upstream* it would first be necessary that the community agrees on a specific strategy on how to handle authentication in general. Contributions to those discussions, which can be found in [45] [46] or [47], could therefore be made in order to participate on a possible solution.

Bibliography

[1] Kit Gustavsson, Erik Stenlund, Efficient data communication between a webclient and a cloud environment, http://lup.lub.lu.se/luur/download?func=downloadFile&recordOId=8885754&fileOId=8885760 last access on 2017-01-06) .

[2] Chris Richardson, Microservices - From Design to Deployment, http://feederio.com/files/book/14770835511053_0c1b8063c763bb871e5731095ac099bc.pdf (last access on 2017-01-06) .

[3] James Lewis, Martin Fowler, Microservices a definition of this new architectural term, https://www.martinfowler.com/articles/microservices.html (last access on 2017-01-06) .

[4] Eberhard Wolff, Microservices: Flexible Software Architecture, 1 st . Edition, ISBN: 978-0134602417, Addison-Wesley Professional, October 22, 2016 .

[5] Johannes Thönes, Microservices, IEEE Software Volume 32, Pages 113-116, 2015 .

[6] Arne N. Johanson, Sascha Flögel, Wolf-Christian Dullo, Wilhelm Hasselbring, OCEANTEA: EXPLORING OCEAN-DERIVED CLIMATE DATA USING MICROSERVICES, 6 th International Workshop on Climate Informatics, 2016 .

[7] Steven Ihde, Karan Parikh, From a Monolith to Microservices + REST: the Evolution of LinkedIn's Service Architecture, Qcon London, 2017 .

[8] Geoff Schmidt, http://www.graphql.com/articles/rise-of-graphql (last access on 2017-01-06) .

[9] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente1, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, Larisa Safina, Microservices: yesterday, today, and tomorrow, Cornell UniversityLibrary, https://arxiv.org/abs/1606.04036 (last access on 2017-01-06) .

[10] Chris Richardson, Pattern: Monolithic Architecture, http://microservices.io/patterns/monolithic.html (last access on 2017-01-06) .

[11] Sam Newman, Building Microservices, 1 st .Edition, ISBN: 978-1491950357, O'Reilly and Associates, 2015 .

[12] Architectural Styles and the Design of Network-based Software Architectures, http://www.ics.uci.edu/ fielding/pubs/dissertation/top.htm (last access on 2017-01-28) .

[13] Robert Battle, Edward Benson, Bridging the semantic Web and Web 2.0 with Representational State Transfer (REST), Web Semantics: Science, Services and Agents on the World Wide Web, Volume 6, Pages 61-69 .

[14] Polo M, Piattini M, Ruiz F. Reflective Persistence (Reflective CRUD: Reflective Create, Read, Update and Delete), Sixth European Conference on Pattern Languages of Programs (EuroPLOP), Universitätsverlag Konstanz GmBH: Irsee, Germany, 2001; 69-85. .

[15] D. Crockford, The application/json Media Type for JavaScript Object Notation (JSON), https://tools.ietf.org/html/rfc4627 (last access on 2017-01-07) .

[16] Github, https://github.com/ (last access on 2017-01-07) .

[17] Facebook GraphQL, https://github.com/graphql (last access on 2017-01-07) .

[18] GraphQL, http://graphql.org/ (last access on 2017-01-07) .

[19] Lee Byron, GraphQL: A data query language, https://code.facebook.com/posts/1691455094417024/graphql-a-data-query-language/ (last access on 2017-01-07) .

[20] NodeJs, https://nodejs.org/en/ (last access on 2017-01-07) .

[21] hapi, https://hapijs.com/ (last access on 2017-01-07) .

[22] hapi auth bearer token, https://github.com/johnbrett/hapi-auth-bearer-token (last access on 2017-01-07) .

[23] Chris Richardson, Pattern: Shared database, http://microservices.io/patterns/data/shared-database.html (last access on 2017-01-07) .

[24] Locust, http://locust.io/ (last access on 2017-01-07) .

[25] GraphiQL, https://github.com/graphql/graphiql (last access on 2017-01-07) .

[26] Postman, https://www.getpostman.com/ (last access on 2017-01-07) .

[27] Restlet Studio, https://studio.restlet.com/ (last access on 2017-01-07) .

[28] Swagger, http://swagger.io/ (last access on 2017-01-07) .

[29] hapi-swagger, https://github.com/glennjones/hapi-swagger (last access on 2017-01-07) .

[30] graphql-sequelize https://github.com/mickhansen/graphql-sequelize (last access on 2017-01-28) .

[31] GraphQL: A data query language, https://code.facebook.com/posts/1691455094417024/graphql-a-data-query-language/ (last access on 2017-02-23) .

[32] GraphQL Specification, http://facebook.github.io/graphql/ (last access on 2017-03-11) .

[33] awesome-graphql, https://github.com/chentsulin/awesome-graphql/ (last access on 2017-03-11) .

[34] Cocoa Application Competencies for iOS, https://developer.apple.com/library/content/documentation/General/Conceptual/Devpedia-CocoaApp/View%20Hierarchy.html/ (last access on 2017-03-11) .

[35] Eric Allman, The robustness principle reconsidered, Communications of the ACM Volume 54 Issue 8, Pages 40-45, 2011 .

[36] Information Sciences Institute, Marina del Rey, RFC793 TRANSMISSION CONTROL PROTOCOL, 1981 .

[37] Len Sassaman, Meredith L. Patterson, Sergey Bratus, A Patch for Postel's Robustness Principle, IEEE Security & Privacy Volume 10 Issue 2, Pages 87 - 91 2012 .

[38] Martin Fowler, Tolerant Reader, https://martinfowler.com/bliki/TolerantReader.html (last access on 2017-03-17) .

[39] Rebecca Parson, Evolutionary Architecture & Micro-Services, https://youtu.be/WhHtVUlJNA0?t=21m37s (last access on 2017-03-17) .

[40] Less strict validation of input objects, https://github.com/graphql/graphql-js/issues/303 (last access on 2017-03-24) .

[41] Ignore undefined input object fields, https://github.com/facebook/graphql/issues/235 (last access on 2017-04-02) .

[42] Forked graphql-js library, https://github.com/hoeselm/graphql-js (last access on 2017-04-02) .

[43] Forked typed-graphql library, https://github.com/hoeselm/typed-graphql (last access on 2017-04-02) .

[44] The OpenAPI Specification (fka The Swagger Specification), https://github.com/OAI/OpenAPI-Specification (last access on 2017-04-11) .

[45] How to use Graphiql when /graphql protected by JWT token (authorization header), https://github.com/graphql/graphiql/issues/59 (last access on 2017-04-11) .

[46] Discussion / Request for Proposal - Enabling Authentication Headers in GraphiQL, https://github.com/apollographql/graphql-server/issues/299 (last access on 2017-04-11) .

[47] Support of the Headers, https://github.com/graphql/graphiql/issues/96 (last access on 2017-04-11) .

[48] Forked graphiql library, https://github.com/hoeselm/graphiql (last access on 2017-04-11) .

[49] Forked graphql-server, https://github.com/hoeselm/graphql-server (last access on 2017-04-11) .

[50] React, https://facebook.github.io/react/ (last access on 2017-04-12) .

[51] Namiot Dmitry, Sneps-sneppe Manfred, ON MICRO-SERVICES ARCHITECTURE, International Journal of Open Information Technologies Vol 9, 2014 .

[52] RFC 2616: Hypertext Transfer Protocol – HTTP/1.1, https://tools.ietf.org/html/rfc2616 (last access on 2017-04-14) .

[53] RFC 2068: Hypertext Transfer Protocol – HTTP/1.1, https://tools.ietf.org/html/rfc2068 (last access on 2017-04-14) .

[54] RFC 6750: The OAuth 2.0 Authorization Framework: Bearer Token Usage, https://tools.ietf.org/html/rfc6750 (last access on 2017-04-14).

[55] RFC 6749: The OAuth 2.0 Authorization Framework, https://tools.ietf.org/html/rfc674 (last access on 2017-04-14).

[56] RFC 4122: A Universally Unique IDentifier (UUID) URN Namespace, https://tools.ietf.org/html/rfc4122 (last access on 2017-04-14).

[57] Suhas Pai, Yash Sharma, Sunil Kumar, Radhika M. Pai, Sanjay Singh, Formal Verification of OAuth 2.0 Using Alloy Framework, Communication Systems and Network Technologies (CSNT), 2011 .

[58] Mark Masse, REST API Design Rulebook, 1st. Edition, ISBN: 978-1449310509, O'Reilly Media, October 31, 2011 .

[59] Joi, https://github.com/hapijs/joi (last access on 2017-04-11) .

List of Figures

List of Tables

List of Listings

Acronyms

API	Application programming interface
CDN	Content Delivery Network
CMS	Content Management System
CPU	Central processing unit
CRUD	Create, read, update and delete
CSS	Cascading Style Sheets
DDD	Domain-driven design
ER	Entity Relationship
GB	Gigabyte
HATEOS	Hypermedia As The Engine Of Application State
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IDE	Integrated development environment
IaaS	Infrastructure as a service
IP	Internet Protocol
IPC	Inter-process Communication
JSON	JavaScript Object Notation
JWT	JSON Web Token
REST	Representational state transfer
RFC	Request for Comments
SQL	Structured Query Language
TCP	Transmission Control Protocol
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
UTF	Unicode Transformation Format
UUID	Universally unique identifier
WWW	World Wide Web
XML	eXtensible Markup Language