Bachelor Project

Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Computer Science

GraphQL layer for RESTful API in practice

Arina Iamshchikova

Supervisor: Ing. Martin Komárek
Field of study: Software engineering and technologies
May 2021

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení:	**Iamshchikova**	Jméno: **Arina**	Osobní číslo: **487215**
Fakulta/ústav:	**Fakulta elektrotechnická**		
Zadávající katedra/ústav:	**Katedra počítačů**		
Studijní program:	**Softwarové inženýrství a technologie**		

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

GraphQL vrstva pro RESTful API v praxi

Název bakalářské práce anglicky:

GraphQL Layer for RESTful API in practice

Pokyny pro vypracování:

For the web application developers, who intend to aggregate multiple RESTful APIs
into a single API Gateway, create an application that will automatically generate a
single API Gateway based on GraphQL.
1) Create a command line interface that takes a swagger specification files from
the existing service as arguments.
2) Handle a security http basic auth issues, if the service related to given
swagger file is secured.
3) Parse the swagger specification file into the GraphQL schema,
4) to parse the swagger specification file into Java POJO, resolvers and
mutations.
5) Create a single endpoint that will handle GET,POST,PUT and DELETE requests.
Generate a fully functional project representing the GraphQL API gateway.

Seznam doporučené literatury:

1. Marc-Andre Giroux. &quot;Production Ready GraphQL&quot;, 2020
2. Eve Porcello, Alex Banks. &quot;Learning GraphQL: Declarative Data Fetching for Modern
Web Apps&quot;, 2018
3. Roy Thomas Fielding. &quot;Architectural Styles and
the Design of Network-based Software Architectures&quot;, 2000
4. Eric Evans. &quot;Domain-Driven Design: Tackling Complexity in the Heart of Software&quot;,
2003

Jméno a pracoviště vedoucí(ho) bakalářské práce:

Ing. Martin Komárek,   katedra informační bezpečnosti   FIT

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **12.02.2021**   Termín odevzdání bakalářské práce: **21.05.2021**

Platnost zadání bakalářské práce: **30.09.2022**

Ing. Martin Komárek podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Studentka bere na vědomí, že je povinna vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací.
Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

.	
Datum převzetí zadání	Podpis studentky

Acknowledgements

I would like to express my appreciation for Ondřej Michalčík for providing technical consultations through this thesis. I would like to extend my thanks to my boyfriend that accompanied and supported me during my study.

Declaration

I hereby declare I have written this work independently and quoted all the sources of information used in accordance with methodological instructions on ethical principles for writing an academic thesis.

In Prague, 21. May 2021

Abstract

GraphQL establishes a new architecture style for web applications. Using it, this work introduces a new way of creating an API client for web applications. This work aims to design and implement the application that will simplify the transition between using the multiple life services and creating a single unified service for retrieving data. The resulting application is a CLI utility that takes URLs referencing Swagger specification files as input and producing a GraphQL application based on Spring Boot that is aggregating APIs described by those specifications. In case of APIs that are protected by HTTP Basic Authentication, it is possible to specify credentials for each API separately so that the resulting application would use the credentials for communication with APIs.

Keywords: web API, GraphQL, Swagger, RESTful architecture, HTTP Basic Auth

Supervisor: Ing. Martin Komárek

Abstrakt

GraphQL přináší nový architektonický styl pro webové aplikace.Pomocí něj, tato práce představuje nový způsob vytváření API klienta pro webové aplikace. Účelem této práci navrhnout a implementovat aplikaci, která zjednoduší přechod mezi používáním četných life-služeb a vytvořením jediné sjednocené služby pro načítání dat. Výslednou aplikací je CLI utilita, která přijímá URL adresy odkazující na soubory specifikací Swagger a produkuje GraphQL aplikaci za použitím Spring Boot frameworku, která agreguje API popsaná těmito specifikacemi. V případě, že API jsou chráněna pomocí HTTP Basic Authentication, je možné zadat pověření pro každé API samostatně, aby výsledná aplikace používala toto pověření pro komunikaci s API.

Klíčová slova: webové API, GraphQL, Swagger, architektura REST, HTTP Basic Auth

Překlad názvu: GraphQL vrstva pro RESTful API v praxi

Contents

Figures

Chapter 1

Introduction

This chapter defines the goals of the thesis as well as possible applications for the end-user.

1.1 Goals

The main goal of this work is to introduce a new, modern way of creating an API for web applications using GraphQL[11] and design an application for generating a GraphQL API gateway from life services.

For the coding part of this project will be used Java language v11. The goals of this part of the work can be divided into the following smaller subtasks:

- The application will have the command-line interface taken one or multiple Swagger[9] configuration files as a parameter. The path to the Swagger configuration file can be lead to a Swagger JSON[6] file on the PC or the Internet.
- The application will also handle the HTTP Basic Auth[4] security if the Swagger configuration file is secured.
- The received Swagger configuration file will then be parsed to create a GraphQL schema, mutations, and resolvers.
- Created components will then be placed into the newly generated Spring Boot[8] project using Maven[1].

The main output of the application will be a single HTTP endpoint that manages all the data and methods from every swagger file it receives as an input. The client request will then propagate through this API gateway to the separate microservices.

1.2 Possible applications

This project is a valuable tool for developers of web applications who intend to create an additional layer above their existing infrastructure to unify their APIs. By creating an API gateway, users will be able to access any interface of the system using a single endpoint, so not only will it simplify the transition to the GraphQL from the RESTful architectures[14], but it will also simplify the access to the APIs that are composing the system.

The generated project will simplify the data integration for the front-end developer as the developer doesn't need to call various endpoints to collect the required data. Instead, the developer can communicate using a single endpoint for fetching multiple data.

The generated application draws specific data like the URL of the web API or credentials for authorization from the configuration file. So change in resource location or credentials of one of the covered APIs can be met with the corresponding change in the configuration file.

Chapter 2
Analysis

This chapter introduces definitions that are essential for understanding the goals of the thesis and describes technologies used for implementing the project.

2.1 Important definitions

Definitions in this section are essential for understanding the discussion of the concept and implementation part of the thesis.

2.1.1 GraphQL

This section gives a brief introduction to the GraphQL.

Description

GraphQL is a query language designed to build client applications by providing an intuitive and flexible syntax and system for describing their data requirements and interactions.

Design principles

GraphQL was built based on the design principles that make it a powerful and productive environment for building client applications:

- **Hierarchical**
  Most product development today involves the creation and manipulation of view hierarchies. To achieve congruence with the structure of these applications, a GraphQL query itself is structured hierarchically. The query is shaped just like the data it returns.

- **Product-centric**
  GraphQL is unapologetically driven by the requirements of views and the front-end engineers that write them. GraphQL starts with their way of thinking and requirements and builds the language and runtime necessary to enable that.[11]

- **Strong-typing**
  Every GraphQL server defines an application-specific type system. Queries are executed within the context of that type system. Given a query, tools can ensure that the query is both syntactically correct and valid within the GraphQL type system before execution, i.e. at development time, and the server can make certain guarantees about the shape and nature of the response.[11]

- **Client-specified queries**
  Through its type system, a GraphQL server publishes the capabilities that its clients are allowed to consume. It is the client that is responsible for specifying exactly how it will consume those published capabilities. These queries are specified at field-level granularity. In the majority of client-server applications written without GraphQL, the server determines the data returned in its various scripted endpoints. A GraphQL query, on the other hand, returns exactly what a client asks for and no more.[11]

- **Introspective**
  A GraphQL server's type system must be queryable by the GraphQL language itself, as will be described in this specification.

GraphQL schema

A GraphQL service's collective type system capabilities are referred to as that service's "schema".[11] The GraphQL schema tells which queries, mutations, types, and directives exist on the server. To say it simple, GraphQL schema is a description of the server's data graph.

The one important rule of the GraphQL schema is that all types within a GraphQL schema must have unique names.

A common way of representing a schema is through the GraphQL Schema Definition Language (SDL). The great thing about the SDL is that it is language agnostic. No matter what language you're running a GraphQL API with, the SDL describes the final schema.[12]

GraphQL operations

GraphQL supports all the HTTP operations such as GET, POST, PUT and DELETE.

The GET operation in a GraphQL schema is wrapped into a Query data type. The query root operation type must be provided and must be an Object type.

All operations that mutate the application state or data are wrapped into one data type: the Mutation. The mutation root operation type is optional. If this type is not provided, it means the service does not support the data changes. If it is provided, it must be an Object type.

2.1.2 RESTful architecture

This section gives a brief introduction to the RESTful architecture style and describes concerns it introduces.

Description

REST is a resource-oriented architecture in which users would progress through web resources by performing operations such as GET, PUT, POST, and DELETE. The network of resources can be thought of as a virtual state machine, and the actions (GET, PUT, POST, DELETE) are state changes within the machine. In a RESTful architecture, routes represent information. For example, requesting information from each of these routes will yield a specific response:

- `/api/food/hot-dog`
- `/api/sport/skiing`
- `/api/city/Lisbon`

Concerns it introduces

There are several concerns that make REST API uncomfortable and not a good choice for every solution:

- **Overfetching**
  With REST, we're getting a lot of data back that we don't need. The client requires three data points, but we're getting back an object with 16 keys and sending information over the network that is useless.
- **Underfetching**
  If we wanted to list the characters that are part of this movie, we'd need to make a lot more requests. In this case, we'd need to hit 16 more routes and make 16 more roundtrips to the client. Each HTTP request uses client resources and overfetches data. The result is a slower user experience, and users with slower network speeds or slower devices might not be able to view the content at all.
- **Managing REST Endpoint**
  Another common complaint about REST APIs is the lack of flexibility. As the needs on the client change, you usually have to create new endpoints and those endpoints can begin to multiply quickly.

2.2 Technologies

The solution is written using the Java programming language v11, Maven build tool, and Spring Boot Framework. Other technologies will be represented as Maven dependencies:

- **PicoCLI**[7]
  A framework for exposing the CLI interface.
- **Swagger Parser**[10]
  For parsing the Swagger specification file and handling the access to the Swagger specification file secured using the HTTP Basic Auth.
- **GraphQL Java**[3]
  For building a GraphQL schema.
- **JavaPoet API**[5]
  Simple API for generating Java classes.
- **Apache Maven**[1]
  Maven is a build automation tool used primarily for Java projects.
- **Apache Maven Invoker API**[2]
  For generating a new Java project with Maven.
- **Java Spring Boot Framework**
  The output project will be running using the Spring Boot Framework.

Chapter 3
Design

This chapter is focused on the design of the final solution. This chapter describes the chosen architecture, project structure using package diagrams, and internal processes using the sequence diagram.

3.1 Architecture

This section is describing the architecture style that suggestively is the most suitable for the final implementation part of the thesis.

3.1.1 Monolithic architecture

Description

The software architecture that was built using monolithic architecture is designed to work as a single, self-contained unit. The components within a monolithic architecture are interconnected and interdependent, resulting in tightly coupled code.[13]

Advantages

There are several benefits of the monolithic architecture style. The most noticeable is its simplicity in every approach. A solution that is written as a monolith is simple to develop and test for small groups of developers due to its compactness. It is also simple to scale horizontally by running multiple copies behind a load balancer.

Disadvantages

Monolithic architecture has its downsides. The most significant one is that this system is hardly maintainable by multiple engineers simultaneously as the code is coupled too tight. Another significant disadvantage is that the applications are consuming too many resources when scaled horizontally.

Application to the project

The solution will be written as a monolith project that is not divided between different services(projects). The division between the specific tasks will be achieved using the package hierarchy and Object-Oriented Programming approach[15].

3.2 The project structure

This section will describe the structure of the implemented project.

3.2.1 Package diagram

The subsection describes the project structure using package diagrams and how it evolved during the implementation process.

First concept

The first concept of the package hierarchy with related java classes that were created before the implementation process has started looks like the Fig 3.1.

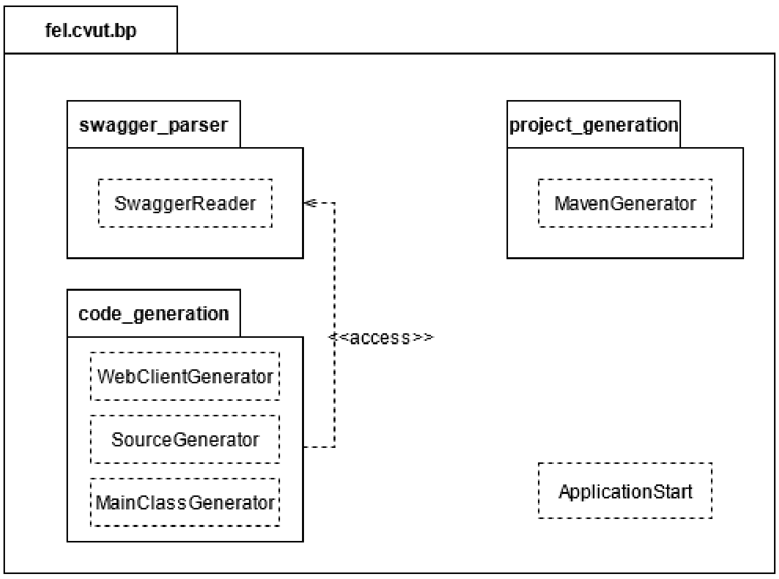

Figure 3.1: Package diagram - first iteration concept

Final structure

The final concept of the package hierarchy with related java classes that were created after the implementation process has begun looks like the Fig 3.2. This concept is an extension of the first concept.
The main differences are:

- *SwaggerReader* class was renamed to *GraphQLSchemaGenerator* due to the intuitivity of the project structure.

- The package *project_generator* was extended for *model* and *service* packages with related java classes.

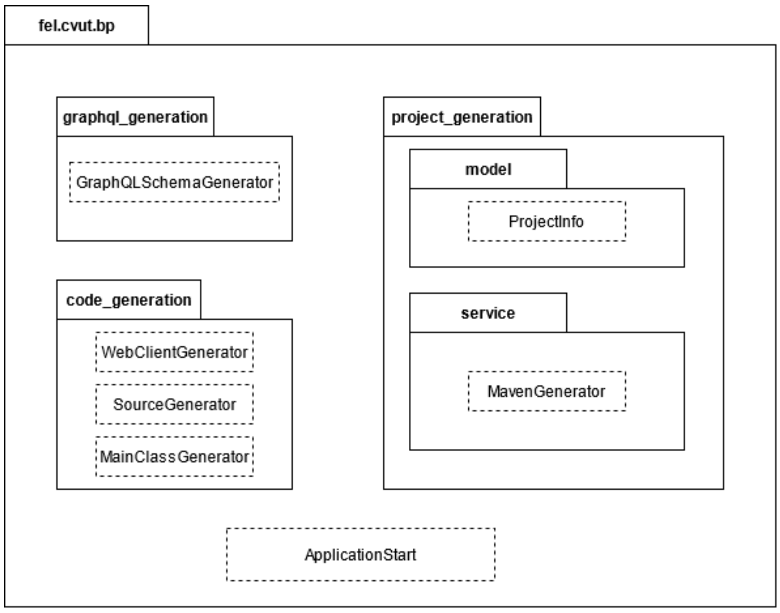

Figure 3.2: Package diagram - final structure

3.3 The processes of the project

This section describes processes that are running inside the implemented application during the project generation process.

Sequence diagram

The internal processes of the implemented application described using the sequence diagram look like the Fig 3.3.

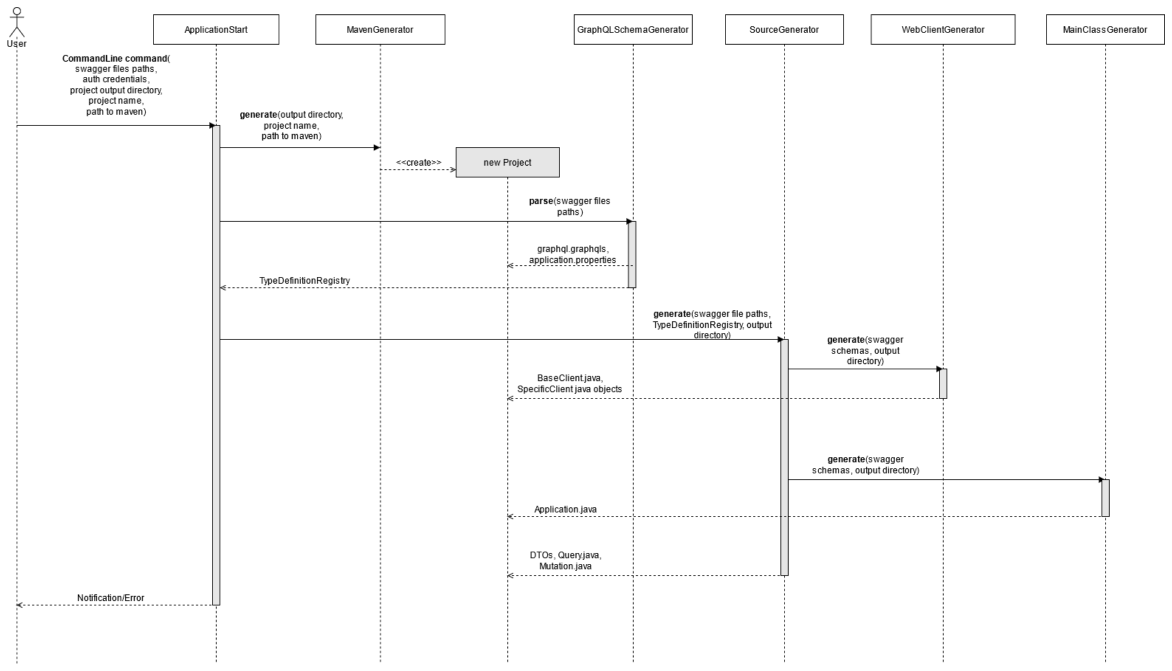
User
ApplicationStart
MavenGenerator
GraphQLSchemaGenerator
SourceGenerator
WebClientGenerator
MainClassGenerator
CommandLine command(swagger files paths, auth credentials, project output directory, project name, path to maven)
generate(output directory, project name, path to maven)
<<create>>
new Project
parse(swagger files paths)
graphql.graphqls, application.properties
TypeDefinitionRegistry
generate(swagger file paths, TypeDefinitionRegistry, output directory)
generate(swagger schemas, output directory)
BaseClient.java, SpecificClient java objects
generate(swagger schemas, output directory)
Application.java
DTOs, Query.java, Mutation.java
Notification/Error

Figure 3.3: Sequence diagram of the application

3.4 The structure of the generated project

This section describes the structure of the project generated as an output of the implemented application.

3.4.1 Package diagram

The structure of the generated project will look like the Fig 3.4.

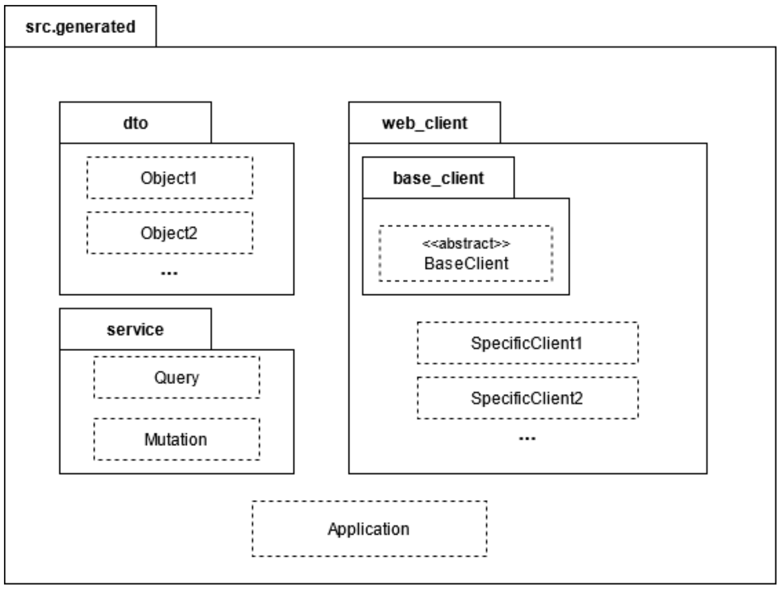

Figure 3.4: Package diagram for the generated project

3.4.2 Example of project invocation

The following command is an example of how to invoke the project generation:

```
java -jar "./target/swagger-to-graphql-0.0.1-SNAPSHOT.jar"
parse
-i "https://petstore.swagger.io/v2/swagger.json"
-i "https://swagger-example-app.azurewebsites.net/swagger/v1/swagger.json;A:B"
-o "C:/"
-m "C:/Program Files/maven"
-n "NewProject"
```

Where:

- **-i** receives the path to a Swagger specification file. Note, that if the file is secured using the HTTP Basic Auth, the path and semicolons should be followed by the credentials in a form of *"A:B"*.
- **-o** accepts the path that will be used for generating a new project.
- **-m** specifies the path to the local Maven executable file.
- **-n** specifies the project name.

Chapter 4

Implementation difficulties

This chapter discusses the difficulties that were met during the implementation process and the solution that was offered.

4.1 Parsing the Swagger

This section defines and provides a solution for difficulties that were met during the parsing process of Swagger specification files.

4.1.1 Parsing parameters

One of the difficulties that were met during the Swagger parsing process was the parsing of the Swagger parameters.

Swagger defines many different parameters, such as PathParameters for the argument that will be passed into the URL of the endpoint or BodyParameters representing either the object that is returned or operated in the given endpoint.

The problem was to identify each type of parameter and parse them separately. Each parameter type has its distinctive properties that should be handled in order to create a proper Java object.

The offered solution was to identify the parameter type before processing it using the if statement that defines the behavior for every type.

4.1.2 Swagger version

During the implementation process was established that every Swagger version requires a different parser. After inspecting the Swagger specification files of some specific companies, was chosen 2.0 Swagger version to be parsed. The choice is supported by the assumption that most of the Swagger specifications are written using the 2.0 version based on this analysis.

4.2 Generating a BaseClient.java

The section describes problems and implemented solutions for the BaseClient.java abstract class generating process.

4.2.1 Creating a URL with the parameters

Swagger offers two ways of passing the parameter into the endpoint path(URL): either by using the PathParameter or QueryParameter.

The methods of creating the path for the web client to be able to request the data are distinct, which leads to a necessity of different parameter handling.

The PathParameter, as well as QueryParameter is represented as a parameter inside the requested path. Distinctive is only the way of their representation. For example in the path *"petstore/pets/petId/status?="available"* the *petId* parameter is a PathParameter and *"available"* is a QueryParameter of type String.

The solution for the correct parameter handling was to identify the parameter type using the if statement and place the corresponding path inside the request method.

4.3 Generating the Maven project

The section describes difficulties that were met during the project generating process.

4.3.1 Default generated classes

During generating the Maven project using the Apache Maven Invoker API was established that for the Spring Boot project, the interface generates the App.java class with the method that writes "Hello World" in the output.

To remove the unnecessary class was created a method that after the project is generated finds the App.java file and deletes it from the project folder.

4.3.2 Java version

The java version that is specified for the newly generated project is 11. If the version is lower, the project won't compile correctly.

Chapter 5

Testing

This section will focus on the testing part of the project implementation. The testing process will be performed for both the developing solution and the project that is generated as an output.

5.1 Testing of the implemented solution

For the testing of the solution are used Unit Tests that are validating the generation of the required files rather than the correctness of the content inside them. Files are generated into the temp folders of the OS, which are cleaned after the restart of the operational system. The paths to the Maven executable file and to the folder where the project will be generated are defined inside the *application.properties* in the test resource folder.

5.2 Testing of the generated project

To test the generated GraphQL server several, tests were created using Postman. By default, the generated project is running on the localhost on port 8080. Based on this information, the URL for test invocation is *"localhost:8080/graphql"*.
All test cases are collected in the following list:

- Create a single object via API without required authentication.
- Get a single object via API without required authentication.
- Create a single object secured via an API secured with HTTP Basic Authorization.
- Get a single object secured via an API secured with HTTP Basic Authorization.
- Create data via two APIs simultaneously where one API is secured and another is not.
- Get data using the objects created by two APIs where one API is secured and another is not.

All the tested features with the test example are represented in the following subsections.

5.2.1 Create a single object via API without required authentication.

QUERY:

```
  mutation Mutation($pet: PetInput!) {
    addPet(body: $pet)
}
```

GRAPHQL VARIABLES:

```
{
    "pet": {
        "id": "123",
        "name": "The big black dog",
        "status": "available",
        "photoUrls": ["firstUrl", "secondUrl"],
        "category": {
            "id" : 1,
            "name" : "Large"
        }
    }
}
```

5.2.2 Get a single object via API without required authentication.

QUERY:

```
{
    getPetById(petId: 123) {
        name
    }
}
```

5.2.3 Create a single object secured via an API secured with HTTP Basic Authorization.

QUERY:

```
  mutation Mutation($client: ClientDTOInput!) {
    client_createClient(dto: $client){
        clientname,
        id
    }
}
```

GRAPHQL VARIABLES:

```
{
    "client": {
        "email": "john.doe@mail.com",
        "firstName": "John",
        "lastName": "Doe",
        "id": "a",
        "password": "1337p455w012D",
        "phone": "555-555-555",
        "clientStatus": 1,
        "clientname": "jodo"
    }
}
```

5.2.4 Get a single object secured via an API secured with HTTP Basic Authorization.

QUERY:

```
{
    client_readClient(ClientId: "{{client_id}}") {
        firstName
    }
}
```

5.2.5 Create data via two APIs simultaneously where one API is secured and another is not.

QUERY:

```
mutation Mutation($client: ClientDTOInput!, $pet: PetInput!) {
    client_createClient(dto: $client){
        clientname,
        id
    }
    addPet(body: $pet)
}
```

GRAPHQL VARIABLES:

```
{
    "pet": {
        "id": "124",
        "name": "Pikachu",
        "status": "pending",
        "photoUrls": ["firstUrl"],
        "category": {
            "id" : 125,
            "name" : "Small"
        }
    },
    "client": {
        "email": "jonny.heresn@mail.com",
        "firstName": "Jonny",
        "lastName": "Heres",
        "id": "b",
        "password": "j02nyH4r4\$",
        "phone": "123-124-125",
        "clientStatus": 1,
        "clientname": "jo2ny"
    }
}
```

5.2.6 Get data using the objects created by two APIs where one API is secured and another is not.

QUERY:

```
{
    client_readClient(ClientId: "{{client_id}}") {
        firstName
    }
    getPetById(petId: 124) {
        name
    }
}
```

5.3 Testing conclusion

All test cases that were running against the newly generated project were using the example command from the subsection *"Example of project invocation"* of the *"Design"* chapter.

The tests against both projects were successfully executed and returned the expected results.

Chapter 6
Conclusion

This chapter is drawing the results of this project and concludes this work. It also suggests topics for further extension of this project.

6.1 Summary

All goals that were identified at the beginning of the work were reached at least on a fundamental level. A detailed description of how every goal was reached is discussed below.

6.1.1 Command-line interface

The application is accessible via the Command-line Interface. When the user wants to generate a project, he can do so by executing the .jar file via terminal and passing all the required arguments.

6.1.2 Generating the Spring Boot project with Maven

A new project is generated into the specified path, and its POM file is modified after the generation to contain all required dependencies.

6.1.3 Creating a GraphQL schema

All swagger schemes are read from specific files and then parsed into a single GraphQL schema file, which contains all methods and objects following the GraphQL syntax.

6.1.4 HTTP Basic Authorization

Authorization is handled via injection of the encoded credentials from the configuration file into the request header.

6.1.5 Generating the java source code

All the classes (such as DTOs, web clients, and services) are generated into the related packages inside of the newly generated project.

6.2 Further extensibility

Even though the output for this thesis is a project that generates the fully working GraphQL based server, there are some parts of it that can be extended and improved. This section is discussing the ways it can be achieved.

6.2.1 Error handling

Usually, in the Swagger specification, there are several return types that are defined by the status code they are assigned to. For the purpose of this thesis was decided to handle only the return types for the status code *200* and *201*. In the future, the project can be extended so that it handles all the status codes and its return types that are defined inside the specific Swagger specification file.

6.2.2 Parse different Swagger versions

The project is currently focusing on parsing the Swagger 2.0 version, and it doesn't support the latest 3.0 Swagger version. As one of the significant extensibility for the project could be the support of the latest Swagger version.

6.2.3 Generate unit tests

Currently, there are no unit or integration tests generated inside the output project. It would be a benefit if unit tests were generated with the source code. This way, a user doesn't need to create tests by himself before starting the server.

6.2.4 Injection of a logger into the generated classes

The other minor recommendation for further extensibility is the injection of a logger for the source code to log events inside the generated project.

6.2.5 OAuth authorization support

OAuth is an open-source authorization method that is gaining popularity, and support of this authorization method would certainly be critical for a fair share of the web developers considering the usage of the project of this work.

Bibliography

[1] Apache Maven. https://maven.apache.org/. [Online].

[2] Apache Maven Invoker. https://maven.apache.org/shared/maven-invoker/. [Online].

[3] GraphQL Java. https://www.graphql-java.com/documentation/v16/. [Online].

[4] HTTP Basic Auth. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Authorization. [Online].

[5] JavaPoet. https://github.com/square/javapoet. [Online].

[6] JSON. https://www.json.org/json-en.html. [Online].

[7] PicoCLI. https://picocli.info/. [Online].

[8] Spring Boot Framework. https://spring.io/projects/spring-boot. [Online].

[9] Swagger. https://swagger.io/. [Online].

[10] Swagger Parser. https://github.com/swagger-api/swagger-parser. [Online].

[11] Inc. Facebook. GraphQL. https://spec.graphql.org/June2018/, June 2018 Edition. [Online].

[12] Marc-Andre Giroux. *Production Ready GraphQL*, chapter 1.2. 2020.

[13] Joseph Ingeno. *Software architect's handbook: become a successful software architect by implementing effective architecture concepts.* Packt Publishing, 2018.

[14] Mark Masse. *REST API Design Rulebook: Designing Consistent Restful Web Service Interfaces.* O'Reilly Media, 31 Oct. 2011.

[15] Lewis J.Pinson Richard Wiener. *Fundamentals of OOP and Data Structures in Java.* Cambridge University Press, 10 Aug. 2000.

Appendix A
Attachments

GraphQLProjectGenerator.zip

```
├── graphql-project-generator..................... source code
└── README.md.............. instructions on project execution
```