FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	GraphQL Architecture – a case study
Student:	Oleksandr Yakunin
Supervisor:	Ing. Michal Valenta, Ph.D.
Study program:	Informatics
Branch / specialization:	Software Engineering
Department:	Department of Software Engineering
Validity:	until the end of summer semester 2023/2024

Instructions

1. Describe GraphQL technology and its variant GraphQL Federation also in comparison to older approaches like REST API.
2. Together with the supervisor formulate a suitable domain and application for demonstration.
3. Analyze, design, implement and test a functional prototype of the application specified in step 2.
4. According to the theory and also to your own experience, try to evaluate the pros and cons of GraphQL Federation technology in software projects.

Electronically approved by Ing. Michal Valenta, Ph.D. on 20 October 2022 in Prague.

Bachelor's thesis

GRAPHQL ARCHITECTURE – A CASE STUDY

Oleksandr Yakunin

Faculty of Information Technology
Department of Software Engineering
Supervisor: Ing. Michal Valenta, Ph.D.
May 11, 2023

Contents

List of Figures

List of code listings

I would like to take this opportunity to express my sincere gratitude to all the teachers of the university for their valuable work and the knowledge they provided, which has helped me on my path towards becoming a software engineer. I would also like to add special thanks to Ing. Michal Valenta, Ph.D. for his guidance and support during the writing of this thesis.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis. I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46 (6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work for non-profit purposes only, in any way that does not detract from its value. This authorization is not limited in terms of time, location and quantity.

In Prague on May 11, 2023 .....

Abstract

This work has several objectives: to describe the fundamental concepts of GraphQL Federation and demonstrate them in practice, to analyze its production readiness, and to explore its potential contributions to the university. To address these objectives, the first part of the work provides the necessary theoretical background while the second part focuses on the actual implementation of a prototype in Java using Netflix DGS framework, showcasing a wide range of functionalities required for creating a project based on the GraphQL Federation concept.

The results of this work indicate that while GraphQL Federation offers significant benefits, there are several aspects that should be thoroughly considered before deciding to use the GraphQL Federation technology. For many projects, it may be too complex of a task. Instead, a smart combination of GraphQL and endpoint-based technologies may also bring significant improvements while requiring less effort.

Keywords GraphQL, GraphQL Federation, Apollo Federation, Netflix DGS, Java

Abstrakt

Tato práce má několik cílů: popsat základní koncepty GraphQL Federace a demonstrovat je v praxi, analyzovat její připravenost k produkci a zkoumat její potenciální přínosy pro univerzitu. Pro dosažení těchto cílů poskytuje první část práce nezbytné teoretické pozadí, zatímco druhá část se zaměřuje na skutečnou implementaci prototypu v jazyce Java s použitím frameworku Netflix DGS, který ukazuje širokou škálu funkcionalit potřebných pro vytvoření projektu založeného na konceptu GraphQL Federace.

Výsledky této práce naznačují, že i když GraphQL Federace nabízí významné výhody, existují určité aspekty, které by měly být pečlivě zvažovány před rozhodnutím použít technologii GraphQL Federace. Pro mnoho projektů může být příliš složité. Namísto toho může chytrá kombinace GraphQL a technologií založených na koncových bodech také přinést významné vylepšení při menším úsilí.0

Klíčová slova GraphQL, GraphQL Federation, Apollo Federation, Netflix DGS, Java

List of abbreviations

API Application Programming Interface

BOM Bill Of Materials

REST Representational State Transfer

RPC Remote Procedure Call

JPA Java Persistence API

PC Personal Computer

HTTP Hyper Text Transfer Protocol

DTO Data Transfer Object

POJO Plain Old Java Object

WEB World Wide Web

SDL Schema Definition Language

URL Uniform Resource Locator

CLI Command Line Interface

Chapter 1
Introduction

This chapter presents the motivation behind the thesis, along with its objectives and structure.

1.1 Motivation

"Creating an API is like creating a work of art. It requires creativity, vision, and attention to detail. But unlike a traditional work of art, an API has the power to make people's lives easier and more efficient."[1]

I have always been interested in studying new technologies and ideas. Moreover, I have always appreciated the beauty in creating useful and user-friendly APIs. Therefore, when I came across an article written by Netflix developers about their new architecture based on the GraphQL Federation concept, I was eager to explore it further. In addition, I consider it essential to keep up with the latest trends and technologies in the field of software development. This approach allows me to have more options to choose from when faced with a development problem and avoid using a particular technology just because I am familiar with it, even though it may not be the best fit for the problem at hand.

1.2 Thesis goals

The goal of this thesis is to introduce GraphQL Federation, explain its fundamental concepts, benefits, potential problems, and provide a functional prototype that contains numerous examples for the wide spectrum of problems that may be encountered during the development of a project based on the GraphQL Federation concept.

Using this information, a conclusion will be drawn about the production readiness of GraphQL Federation and its possible contribution.

1.3 Thesis structure

Thesis is divided into the following main sections:

1. **"Basic concepts and technologies"** - briefly describes the main concepts and technologies used in the thesis.
2. **"GraphQL"** - introduces GraphQL and provides the necessary theoretical background for subsequent chapters.

3. **"GraphQL Federation - theory"** - presents the concept of GraphQL Federation and offers essential theoretical information required for the implementation part.
4. **"GraphQL Federation - implementation"** - describes the functional prototype implementation, showcases Netflix DGS, and discusses common challenges encountered when applying GraphQL Federation in practice.
5. **"Conclusion"** - summarizes the results of the thesis and presents the final conclusions.

Chapter 2

Basic concepts and technologies

This chapter aims to list and provide a brief description of all the essential concepts and technologies used in this thesis. The chapter begins with a discussion of more theoretical concepts such as GraphQL Federation and concludes with specific technologies used in the implementation of the prototype.

2.1 GraphQL

"GraphQL is more than just a tool: it changes how teams communicate. It empowers developers to request for the data they want, reducing their dependency and communication overhead with API teams. It thrives on community and collaboration."[2]

GraphQL is a modern query language and runtime for APIs, developed by Facebook in 2012 and later released as an open-source project. It allows clients to request specific data from a server, enabling more efficient and flexible data retrieval compared to traditional REST APIs. With GraphQL, clients can define the shape of the response, reducing the amount of over- or under-fetched data. The language supports queries, mutations, and subscriptions, enabling read, write, and real-time updates for data. GraphQL's type system ensures that data is strongly typed, providing better error handling and improved development experience.

2.2 Microservice architecture

Microservice architecture is a modern software design approach that structures an application as a collection of small, loosely coupled services. Each microservice is responsible for a specific functionality or domain and can be developed, deployed, and scaled independently. This modularity promotes separation of concerns, enabling teams to work concurrently on different parts of the system while minimizing the impact of changes on other components.

2.3 GraphQL Federation

GraphQL Federation is an architectural pattern that enables the creation of a unified API by composing multiple, separately deployed GraphQL services. It allows developers to break down a monolithic GraphQL schema into smaller, more manageable microservices. Each microservice can define its own schema, which is then combined into a single, federated schema by a gateway. This approach promotes separation of concerns, easier scaling, and better maintainability while still providing a seamless experience for clients.

2.4 API Gateway

"The API gateway pattern is recommended if you want to design and build complex large microservices-based applications with multiple client applications. The pattern is similar to the facade pattern from object-oriented design, but it is part of a distributed system reverse proxy or gateway routing for using as a synchronous communication model. We said that It is similar to the facade pattern of Object-Oriented Design, so it provides a single entry point to the APIs with encapsulating the underlying system architecture. The pattern provides a reverse proxy to redirect or route requests to your internal microservices endpoints. An API gateway provides a single endpoint for the client applications, and it internally maps the requests to internal microservices. In summary, the API gateway locate between the client apps and the internal microservices. It is working as a reverse proxy and routing requests from clients to backend services. It is also provide cross-cutting concerns like authentication, SSL termination, and cache."[3]

2.5 Apollo Federation

Apollo Federation is an implementation of GraphQL Federation, developed by the Apollo team. It provides a set of tools and guidelines for building scalable and modularized GraphQL APIs using Federation principles. It also provides features such as distributed tracing, error propagation, and cross-service caching, which further enhance the performance and resilience of the federated API.

2.6 Netflix DGS

Netflix DGS is a GraphQL framework developed by Netflix to simplify the process of building and maintaining GraphQL APIs in Java and Kotlin applications. The framework leverages Spring Boot and offers a set of tools, conventions, and features that streamline the development experience.

2.7 Instrumentation

In the context of GraphQL, instrumentation refers to the ability to hook custom code before or after the execution of a query, enabling fine-grained control and observability of the GraphQL engine.

By using instrumentation, developers can gather metrics, implement tracing, apply custom logic, or modify query responses. The instrumentation mechanism allows for the creation of reusable components that can enhance the functionality and maintainability of the GraphQL API. This feature helps improve the overall development experience, allowing developers to monitor, diagnose, and optimize their GraphQL applications more effectively.

2.8 Introspection

"As a developer, we are always consuming new APIs. We often spend a lot of time reading the documentation (frequently out of date) and finding which resources are available. GraphQL provides a tool that tells you which resources you can get/modify and how to do it. GraphQL supports it with the concept known as Introspection. Introspection is the ability to query which resources are available in the current API schema. Given the API, via introspection, we can see the queries, types, fields, and directives it supports."[4]

2.9 Gradle

Gradle is a powerful and versatile build automation tool used primarily for Java, Kotlin, and other JVM-based projects, but it also supports a variety of other programming languages. It simplifies and streamlines the process of building, testing, and deploying software applications by automating various tasks, such as compilation, packaging, dependency management, and versioning.

Gradle leverages a domain-specific language based on Groovy or Kotlin to define and configure build scripts, providing developers with a readable, maintainable, and expressive way to specify build processes.

2.10 Spring

At its core, Spring provides a lightweight container for managing and configuring application components, known as beans. The framework emphasizes inversion of control and dependency injection patterns, which help promote modularity, testability, and maintainability in applications. Spring covers a wide range of functionality, including data access, web development, security, and messaging.

2.11 Spring Boot

Spring Boot is a widely-used framework that simplifies the process of creating, configuring, and deploying Java-based applications built with the Spring framework. By providing a set of auto-configuration options, Spring Boot enables developers to rapidly build stand-alone, production-ready applications with minimal effort and boilerplate code.

2.12 JPA

JPA is a standardized specification that provides a set of guidelines for object-relational mapping in Java applications. The primary goal of JPA is to simplify the process of persisting Java objects to relational databases by bridging the gap between object-oriented programming and relational database concepts.

2.13 Lombok

Lombok is a library that helps developers reduce boilerplate code and improve code readability. It provides a set of annotations that can be used to generate Java code during compilation time, such as getter and setter methods, constructors, and logging code.

2.14 Mapstruct

MapStruct is a code generation tool that simplifies the mapping between Java classes. It generates code that maps data from one Java class to another, reducing the amount of boilerplate code that developers need to write. MapStruct uses Java annotations to generate code that performs mapping operations, eliminating the need for developers to write mapping code manually.

2.15 Docker

Docker is a platform that simplifies the process of creating, deploying, and running applications in containers. Containers are lightweight, portable, and self-contained packages that include all the necessary software, libraries, and dependencies required to run an application. Docker allows developers to create and manage containers, which can be used to build, test, and deploy applications across different environments.

Chapter 3

GraphQL

This chapter explains the fundamental concepts of GraphQL and provides the theoretical background required for the following chapters. The chapter covers all the necessary concepts, starting from the schema definition to security and caching.

3.1 History of GraphQL

Currently, endpoint-based APIs such as REST and RPC dominate the field of Web APIs. These technologies have multiple advantages: they are easy to implement, cache, use and are really suitable for single use cases. However, with the constantly growing number of clients, the requirements for APIs have started to change. Different clients, such as PCs, phones, consoles, and watches, require distinct data, leading to the development of One-Size-Fits-All APIs that attempted to cover too many use cases. There have been numerous attempts to solve this problem, but in most cases, APIs became challenging to manage and evolve due to their tight coupling with various clients. For example, there were ideas to create new endpoint for each type of the client (see Listing 3.1) or use query parameter to distinguish between them (see Listing 3.2).

All these attempts failed to solve the problem, and as a result, around the beginning of the 21st century, major tech companies began investing in the development of a brand new solution. This is how GraphQL was developed in 2012 and then made available to the public in 2015.

```
GET api/mobile/products
GET api/watch/products
```

Code listing 3.1 Example of using new endpoint for each type of client

```
GET api/products?version=mobile
GET api/products?version=watch
```

Code listing 3.2 Example of using query parameter to distinguish between clients

3.2 What is GraphQL

It is very common to think that GraphQL is devloped using graph theory or to confuse it with different technologies such as graph database. Actually, "GraphQL is a query language for your API, and a server-side runtime for executing queries using a type system you define for your data. GraphQL isn't tied to any specific database or storage engine and is instead backed by your existing code and data."[5]. GraphQL follows three main concepts as listed below:

- **Declarative** - let the client define what it wants to get.
- **Compositional** - allow composing query out of multiple resources.
- **Strictly typed** - allow client developers to know exactly what kind of requests are supported, what they return and what parameters they accept.

3.3 GraphQL vs REST

GraphQL and REST are two widely used approaches for building WEB APIs but as we already discovered they have fundamental differences in their philosophy. Therefore, it is useful to compare them and gain an overview of potential indicators that suggest the use of one technology over the other. This section will be divided into two parts: the first discussing the benefits of GraphQL and the second outlining its drawbacks.

3.3.1 GraphQL benefits

3.3.1.1 Under and over fetching

Unlike REST, GraphQL provides a concept of partial payload which eliminates the common problems of under and over fetching which often arise due to the constantly changing data. GraphQL allows clients to request exactly the data they need by specifying fields that should be present in the response. Let's imagine there exists GraphQL endpoint which allows to query for all teachers and retrieve their username, first name and last name but we are only interested in their usernames. You can find an example of how to use GraphQL query language to retrieve only the necessary fields in Figure 3.1.

Figure 3.1 Example of simple GraphQL query

3.3.1.2 Waterfall requests

The concept of enabling clients to request precisely the data they require, as mentioned in the previous subsection, is also beneficial in dealing with the waterfall requests problem. Waterfall

requests are a common issue in traditional REST APIs, where the client needs to make multiple requests to various endpoints to retrieve all the necessary data for a single view. This leads to slower performance and increased network traffic.

Imagine we have a Teacher resource that can have multiple assigned subjects. In REST, we would need to create two separate endpoints: one for fetching the teachers and one for fetching the subjects assigned to teacher. However, in GraphQL, we can easily combine both endpoints into a single query and get no drawbacks. See Figure 3.2 for an example.

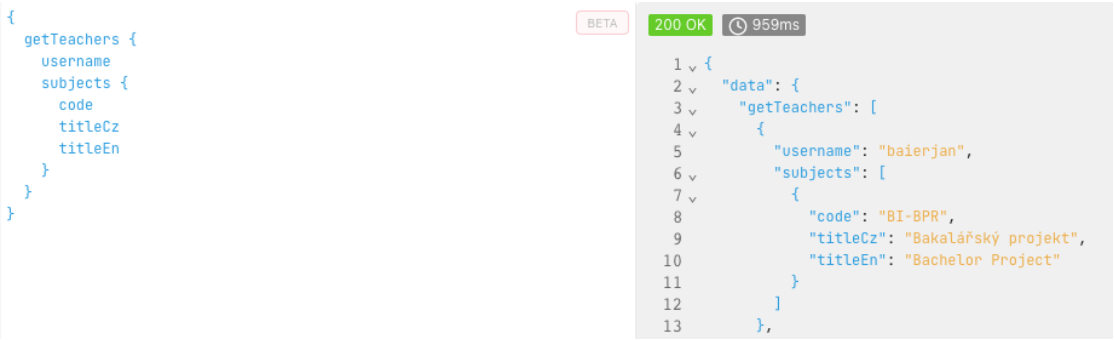

Figure 3.2 Example of using GraphQL query language to get nested object

3.3.1.3 Strong typing

GraphQL simplifies common problems such as determining which endpoint to use and which query parameters are valid by using a single endpoint in conjunction with a strongly-typed schema. The schema defines all available resources and input parameters and then server side runtime ensures that clients receive the expected data type as the response. Also, you can document the schema using code style comments which not only make development easier and faster but also eliminate the need for additional tools such as Swagger. See an example of documented schema in Figure 3.3

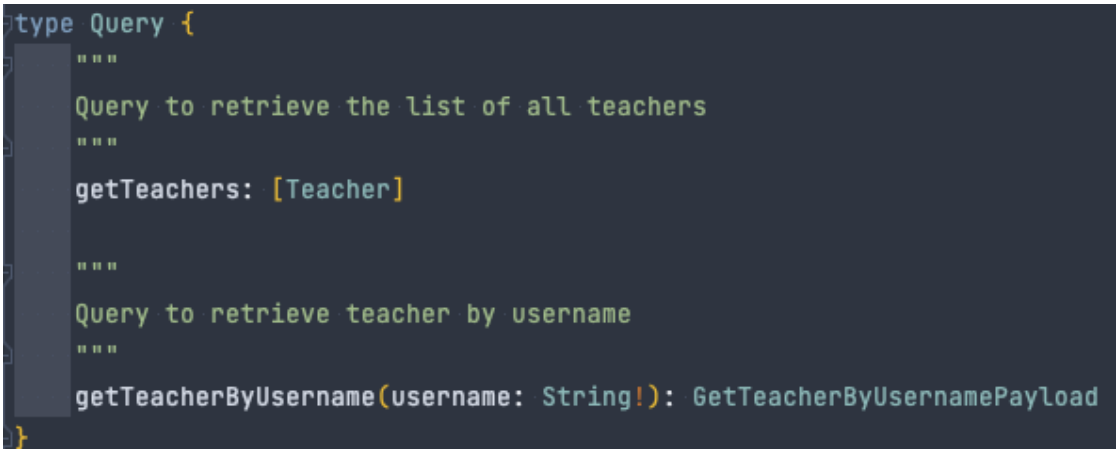

Figure 3.3 Example of documented GraphQL schema

In combination with introspection it becomes very powerful feature as introspection allows clients to query an API an obtain the description of schema, allowing them to understand the API's capabilities and structure without having to rely on external documentation or knowledge.

3.3.1.4 Versioning

In the REST architecture, it is typical to create new endpoints for each version of the API. However, this approach can create confusion as it becomes challenging to determine the latest version of the API capable of returning the correct data and track all the changes. On the other hand, in GraphQL, the result set of a query is tailored to the specific needs of the client. Therefore, the addition of new features or fields does not affect the existing clients. For instance, Facebook uses a single version of a graph in combination with Schema registry to track the evolution of the graph.

3.3.2 GraphQL drawbacks

3.3.2.1 Abusive queries

GraphQL provides clients with a significant amount of control and flexibility, which can lead to the execution of queries that may put excessive load on the server. Consequently, developers must take extra precautions to prevent undesirable queries from being executed, requiring additional effort.

3.3.2.2 Caching

In REST we can cache on resource level as URL may serve as an identifier. In GraphQL, there is no possibility to implement caching at HTTP level as there is only one POST endpoint and all queries even the ones operating on the same resource may be different. Thus, other techniques such as Persisted Queries should be used.

3.3.2.3 Rate limiting

On contrary to REST, rate limiting in GraphQL is more challenging as each query may have a different complexity, and thus limiting by the number of requests per unit of time is ineffective. Instead, a different approach based on query complexity or the time required for query execution should be considered.

3.3.2.4 Tooling

While the GraphQL ecosystem is growing rapidly, it may not have the same level of tooling and support as more established technologies like REST. This can make it more challenging to find libraries, frameworks, and other resources for development and debugging.

3.4 Schema definition language

“Because the shape of a GraphQL query closely matches the result, you can predict what the query will return without knowing that much about the server. But it’s useful to have an exact description of the data we can ask for - what fields can we select? What kinds of objects might they return? What fields are available on those sub-objects? That’s where the schema comes in.

Every GraphQL service defines a set of types which completely describe the set of possible data you can query on that service.”[6] This set of defined types is referred to as the **Schema** and is used for validation and execution of a query by the server-side runtime. Since GraphQL is a specification and can be implemented in any language, it cannot rely on a specific syntax. That is why GraphQL defines its own known schema language referred as the **Schema Definition Language**.

3.4.1 Object type and fields

Object type is a core building block of GraphQL schema which represents a set of related fields and is mainly used to represent a resource. See Listing 3.3 for an example of simple object type definition. You can see that each field in GraphQL is associated with specific data type. In our case, all fields are of a `String` type. Moreover, you may notice two special literals: `!` and `[]`. The `!` indicates that a field is non-nullable, guaranteeing that the value will always be present in the response. On the other hand, `[]` is used to signify that a field will return a list in the response..

```
type Teacher {
    username: String!
    firstName: String
    roles: [String]!
}
```

Code listing 3.3 Example of object type definition

3.4.2 Query, Mutation, Subscription

There are three special predefined object types: `Query`, `Mutatation` and `Subscription` that serve as an entry points to an application. Fields defined under these types correspond to the operations that can be performed on the server in the same way as REST endpoints do. See Listing 3.4 for an example of the definition.

The `Query` type defines the read operations that can be performed on the data. A query is a read-only operation and is used to retrieve data from the server.

On the other hand, the `Mutation` type defines the write operations that can be performed on the data, which usually includes modifying or creating data on the server.

`Subscription` type is used for real-time operation that enables clients to receive an update whenever specific event occurs on the server.

```
type Query {
    getTeachers: [Teacher]
    getTeacherByUsername(username: String!): Teacher
}

type Mutation {
    createTeacher(teacher: TeacherInput!): Teacher
    deleteTeacherByUsername(username: String!): Boolean
}
```

Code listing 3.4 Example of query and mutation definition

3.4.3 Arguments

In GraphQL, each field can have any number of associated named arguments. These arguments serve the same purpose as parameters in Java functions, allowing for more dynamic and cus-

tomized queries. For instance, Listing 3.5 showcases an example of `teacherByUsername` query returning corresponding teacher to the `username` argument.

```
type Query {
    teacherByUsername(username: String!): Teacher
}
```

Code listing 3.5 Example of using field arguments

As you can see, the `!` literal can be applied on field arguments too. This makes them non-nullable and thus mandatory for the query execution.

3.4.4 Scalars

Scalars are used to represent the leaf nodes of a GraphQL query, as they correspond to primitive data types that cannot have child fields. Their primary use case is to represent the types of arguments or return values of the fields. You can see the comprehensive list of the built-in scalar type below:

- **Int** - signed 32-bit integers.
- **Float** - signed double-precision fractional values.
- **String** - UTF-8 character sequences.
- **Boolean** - true or false value.
- **ID** - it is serialized the same way as String and is used to mark unique identifiers.

It's worth noting that GraphQL provides developers with the ability to define custom scalars in addition to the built-in ones. This means that developers can create their own scalar types to represent specialized data types, such as dates or geographical coordinates, that are not covered by the built-in scalar types. You can see Custom scalar section for an example of custom scalar implementation.

3.4.5 Enumeration

An enumeration type is a special scalar type that is constrained to a finite set of values. It functions similarly to traditional enums in Java and can be defined using the syntax shown in Listing 3.6. In the aforementioned listing, an enumeration type `User` is defined, with three possible values: `TEACHER`, `STUDENT`, and `ADMIN`. Enumeration types are useful for specifying fields that can only take on a limited number of valid values, making them an important tool for ensuring the integrity of data in a GraphQL schema.

```
enum User {
    TEACHER
    STUDENT
    ADMIN
}
```

Code listing 3.6 Example of an enumeration type

3.4.6 Interface

An interface is a type that defines a set of fields that any implementing type must include in its own definition. On contrary to Java interfaces which are used to define the behaviour, GraphQL interfaces are used to define common fields. Listing 3.7 provides an example of an interface `User` with a single field, `username` which `Student` type must include in its definition to become an implementing type.

```
interface User {
    username: String
}

type Student implements User {
    username: String
    department: String
    studyYear: Int
}
```

Code listing 3.7 Example of an interface definition

Interfaces are useful for creating reusable components in a GraphQL schema, as well as for enforcing consistency and reducing duplication in type definitions.

3.4.7 Union

Similar to an interface, a union type is used for achieving polymorphism. It also allows a field to return one of multiple object types, enabling more flexible and dynamic queries. The difference is that with a union type, the types do not need to share any common fields.

Listing 3.8 provides an example of a union type in GraphQL. The `GetUserByUsernamePayload` union type is defined as a combination of two types: `GetUserByUsernameSucess` and `UserNotFound`. It is used as a return type of `getUserByUsername` query to indicate that any union type may be returned in the response based on some condition. Note that querying of the union is different from regular GraphQL queries, for this fragments concept which is described in the Query Language should be used.

```
union GetUserByUsernamePayload = GetUserByUsernameSucess | UserNotFound

type Query {
    getUserByUsername(username: String!): GetUserByUsernamePayload
}
```

Code listing 3.8 Example of a union type definition

3.4.8 Input type

To avoid confusion and potential errors, GraphQL prohibits the use of object types as field arguments. Instead, a special **Input type** should be used. Consider Listing 3.9, which demonstrates the use of an input type.

```
type Mutation {
    creteTeacher(teacherInput: TeacherInput!): Teacher
}

type Teacher {
    username: String
    firstName: String
}

input TeacherInput{
    username: String
    firstName: String
}
```

Code listing 3.9 Example of input type

In this example, the `createTeacher` mutation takes an argument of type `TeacherInput` contains the necessary fields for creating a teacher.

3.5 Errors

In GraphQL, errors are indicated using top level field `errors` (see Listing 3.10) instead of common response status approach. `errors` field consists of an array of error objects, each with a `message`, `locations`, `path`, and `extensions` fields. Note that `extensions` field may be used to add custom information.

```
{
  "errors": [
    {
      "message": "message",
      "locations": [],
      "path": [],
      "extensions": { }
    }
  ]
}
```

Code listing 3.10 Example of errors field

This approach definitely reduces the declarative power of the schema, so it is wise to combine it with the approach known as errors as data. The idea is to use the `errors` field only for developer errors, such as timeouts and internal server errors while to return errors as data in case of user errors such as duplicate password or username taken.

The errors as data approach involves defining errors as types in the schema and using them as operations return types. There is no established convention on how to implement this approach so I will show the one I like the most. See Listing 3.11 for an example.

Benefits of the described approach are following:

- **Expressive and discoverable schema** - by defining errors as types in the schema, the

```
type Mutation {
    createTeacher(teacher: TeacherInput!): CreateTeacherPayload!
}

type CreateTeacherPayload {
    teacher: Teacher
    errors: [CreateTeacherError!]!
}

union CreateTeacherError = UserNameTaken | FirstNameNotSet

interface UserError {
    message: String!
}

type UserNameTaken implements UserError {
    message: String!
    suggestion: String
}
```

Code listing 3.11 Example of errors as data

schema becomes more expressive and easier to understand. Developers can quickly see what errors can be returned by each field and handle them accordingly.

- **Easier evolution** - as the schema evolves, it is easier to add or modify error types without affecting existing clients. Error types can be deprecated or renamed, and clients can be updated accordingly.
- **Structured error handling** - by defining error types, developers can implement structured error handling in their code. They can catch specific errors and handle them appropriately, rather than relying on generic catch-all error handling.

The drawback of the approach is the amount of effort. It is quite verbose and requires significant amount of developer attention.

3.6 Abusive Queries

The prevention of abusive queries is a hot topic in GraphQL, as it gives clients a lot of power which may result in executing queries that overload the server. For example, defining a two-way relationship in the schema can lead to unlimited nesting, causing server overload. Initially, validating the depth of the incoming query may seem sufficient, but it is far from it. GraphQL queries can also be abused in breadth, requiring different handling logic. The recommended approach is to compute the complexity of the incoming query based on the number of required operations.

Additionally, setting a timeout for the server can be useful, as clients may try to overload the server by sending an enormous list of input parameters which needs to be parsed and validated by the server. As you can see, there are many potential issues with abusive queries that require a lot of attention from developers. See Listing 3.12 for an example of abusive recursive query.

```
{
  getTeachers {
    username
    subjects {
      code
      teacher {
        username
        subjects {
          code
          teacher {
            // ...
          }
        }
      }
    }
  }
}
```

Code listing 3.12 Example of abusive recursive query

3.7 Rate limiting

In REST, rate limiting is very easy to do just by limiting the number of incoming requests per some amount of time. But what to do if every query may have different complexity. See Figure 3.4 and Figure 3.5 for an example of same query having different complexity.

```
{
  getTeachers {
    username
    firstName
    lastName
    titleBefore
    titleAfter
    department
    position
    employmentStatus
  }
}
```

Figure 3.4 Example of getTeachers query

```
{
  getTeachers {
    username
    subjects {
      lectures {
        code
      }
      literature {
        isbn
      }
    }
  }
}
```

Figure 3.5 Example of getTeachers query

This means that we have to implement rate limiting based on different factors like query complexity or the time required for query execution, which adds additional complexity to the implementation.

3.8 Persisted Queries

For each GraphQL query, server lexes, parses, validates and executes the query and then returns the result. This is costly operation and there is no point in doing it over and over again if the query doesn't change. There are multiple options to consider but one of the most popular techniuqes is called **Persisted Queries**. It is used to optimize performance and reduce network overhead. The difference with regulat approach is that instead of sending the whole query every time, client firstly registers queries with the server and obtains some unique identifiers such as query hashes or URLs as a response.

"With persisted queries, instead of sending the full query document on every request, the client starts by registering queries with the server, before any query is even sent. Sometimes these queries are registered before or during the deploy process. In other cases, the first query from a client is used as the "registration". In exchange, a GraphQL server capable of supporting persisted queries will provide the client with an identifier for that query. Examples of good identifiers are query hashes, URLs at which the queries can be accessed, or simple IDs. Once the client has the identifier for a particular query, it can send the identifier along with any needed variables to execute the query, this time without passing the full query document. For example, if the server returned an URL after the registration of a certain query, the client could use this URL instead of sending the query document every single time.

Besides the performance and bandwidth improvements, this also helps API providers secure GraphQL APIs. Earlier we covered whitelisted queries, which meant allowing only certain queries to be ran against our GraphQL server. With persisted queries, this is even more straightforward, since an API provider could allow only pre-registered queries to be run, essentially blocking access to all other queries against the API. The funny thing with persisted queries implemented that way is that it starts to look a lot like what we wanted to escape in the first place, endpoint based and fixed queries! However, there's a small detail that makes this so powerful. Even though we're dealing with static queries/resources, these resources are generated by the clients rather than the server. In fact I love thinking of persisted queries as client dynamically generated resources, using the dynamic GraphQL engine to support as many different resources as needed by clients."[7]

3.9 Query Language

GraphQL provides a simple and intuitive way to query data from a server. When making a request, you simply specify the fields you want to retrieve. For example, let's say we have a schema like the one shown in Listing 3.13. To retrieve a list of teachers, their usernames and first names, we can use the query as demonstrated in Listing 3.14. In this query, we specify that we want to retrieve data from the `getTeachers` field of `Query` type which returns a list of `Teacher` objects. We then specify the fields we are interested in for each `Teacher` object, which in this case are `username` and `firstName`. As you can see, it is all about selecting desired fields from the list of all available ones.

Now, let's consider different schema as shown in the Listing 3.15. As you can see, query `getTeachers` might return a `Success` or an `Error` type, depending on whether the operation was successful. To handle these scenarios, you can use concept knows as fragments which is shown in the Listing 3.16. In the mentioned listing, we defined two fragments, one for each response type using `... on` syntax. Then, we can simply select desired return fields.

3.10 OpenAPI to GraphQL

Sometimes there is a need to convert REST service in the GraphQL one in rapid way. Usually, this is accomplished by creating a GraphQL wrapper that calls REST endpoints internally.

```
type Query {
    getTeachers: [Teacher]
}

type Teacher {
    username
    firstName
    lastName
}
```

Code listing 3.13 Example schema

```
query {
    getTeachers {
        username
        firstName
    }
}
```

Code listing 3.14 Example of a query

```
type Query {
    getTeachers: Response
}

union Response = Success  | Error
```

Code listing 3.15 Example schema

```
query {
    getTeachers {
        ... on Success {
            // fields
        }
        ... on Error {
            // fields
        }
    }
}
```

Code listing 3.16 Example of fragments usage

It is worth noting that there are libraries available that allow automatic wrapping of existing Swagger-documented REST services in GraphQL. You can find complete example of using IBM framework for wrapping of REST server in the practical part of the thesis. In this section, I want to highlight two most important and not obvious possibilities.

First one, is the possibility to add security to the GraphQL wrapper. It can be done by applying the `@SecurityScheme` annotation to the controller or configuration class, and then using the `@SecurityRequirement` annotation for each endpoint that requires protection. An example of implementing basic authorization can be seen in Listing 3.17.

```
@Tag(name = "Teacher API", description = "API for accessing teacher
↪  information. Protected by oAuth/fake tokens.")
@SecurityScheme(name = "basic", type = SecuritySchemeType.HTTP, scheme =
↪  "basic")
@RestController
@RequiredArgsConstructor
public class TeacherController {

        private final TeacherService teacherService;

        @ResponseStatus(HttpStatus.OK)
        @GetMapping(value = "/teachers", produces =
        ↪  MediaType.APPLICATION_JSON_VALUE)
        @SecurityRequirement(name = "basic")
        @Operation(summary = "Retrieve a list of all teachers.")
        @ApiResponse(responseCode = "200")
        public List<Teacher> findAllTeachers() {
                return teacherService.findAll();
        }
```

Code listing 3.17 Example of defining basic authorization in Swagger

The second thing is that in GraphQL it is very common to fetch multiple objects via single query which is not the case in REST. We can use `@Link` annotation to instruct IBM framework to query multiple REST endpoints in single GraphQL query. See Listing 3.18 for the example.

```
@GetMapping(value = "/people/{username}", produces =
↪  MediaType.APPLICATION_JSON_VALUE)
@Operation(summary = "Retrieve a person with provided username.",
↪  responses = {
        @ApiResponse(responseCode = "200", links = {
                @Link(name = "Find Car", description = "Find connected
                ↪  car", operationId = "findCar", parameters = {
                    @LinkParameter(name = "username", expression =
                    ↪  "$request.path.id")
                })
        })
})
public Person findPerson(@PathVariable String username) {
    return personService.findPersonByUsername(username)
            .orElseGet(Person::new);
}

@GetMapping(value = "/people/{username}/car", produces =
↪  MediaType.APPLICATION_JSON_VALUE)
@Operation(summary = "Retrieve a car.")
public Car findCar(@PathVariable String username) {
    return new Car("Porsche", "Cayenne");
}
```

Code listing 3.18 Example of defining link in Swagger

Chapter 4

GraphQL Federation - theory

This chapter presents GraphQL Federation, discussing its origin, explaining key concepts, and providing the necessary theoretical foundation for subsequent chapters.

4.1 History of GraphQL Federation

The origin of GraphQL comes from the monolithic API layer but with the recent trend towards distributed architecture, it has found its way into various contexts and proved itself with the API gateway pattern (see Figure 4.1). This is true due to the fact that it is beneficial to use GraphQL as an abstraction over existing APIs as it provides a stable and easily discoverable schema along with the query language that enables clients to consume only the necessary data. One of the main challenges of using GraphQL with a gateway lies in the fact how to compose the schema in a way that preserves the key benefits of GraphQL. As a result, there are several different approaches to schema composition that have emerged over time which will be described in following subsections.

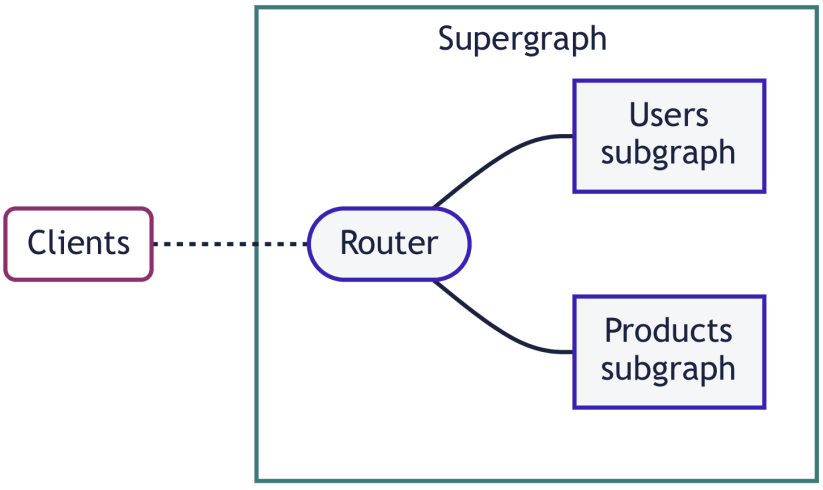

Figure 4.1 Example of gateway pattern [8]

4.1.1 Simple Merging

One of the initial ideas for composing GraphQL schemas was to merge all the fields together in the simplest possible manner and halt the composition process if any conflicts arose. In the scenario where we have two schemas, as depicted in Listing 4.1 and Listing 4.2, the resulting schema would look as presented in Listing 4.3.

```
type Query {
    getLecturesByTeacherUsername(username: String!): [Lecture]
}
```

Code listing 4.1 Lecture service schema

```
type Query {
    getCoursesByTeacherUsername(username: String!): [Course]
}
```

Code listing 4.2 Course service schema

```
type Query {
    getLecturesByTeacherUsername(username: String!): [Lecture]
    getCoursesByTeacherUsername(username: String!): [Course]
}
```

Code listing 4.3 Resulting merged schema

This approach doesn't have any special handling logic thus it is easy to implement. The drawback is that it sacrifices declarative power of GraphQL. It becomes impossible to express relation between two types defined in different services as shown in Listing 4.4. Instead, as demonstrated in Listing 4.3, we revert to a REST-style approach where the client must query both `getLecturesByTeacherUsername` and `getCoursesByTeacherUsername` to obtain the data required for a single view.

```
type Course {
    code: ID
    lectures: [Lecture]
}
```

Code listing 4.4 Example of declarative solution

4.1.2 Schema Stitching

In an effort to restore the declarative power of GraphQL, a technique known as Schema Stitching was devised. The idea is to delegate all the relationship logic to the gateway, as it possesses

enough information to generate a valid schema. For instance, taking the same example as described in the previous section, we can use Apollo Stitching to instruct our gateway on how to compose the schema, as illustrated in Listing 4.5.

```
const typeExtensionDefinition = `
    extend type Course {
        lectures: [Lecture]
    }
`;

mergeSchemas({
    schemas: [
        courseService,
        lectureService,
        typeExtensionDefinition
    ]
});
```

Code listing 4.5 Example of type extension in the gateway

In the aforementioned listing, we specified that the teacher and course service schemas should be composed, and a `typeExtensionDefinition` should be added to the resulting schema. As a result, the schema now has the same structure as presented in Listing 4.4. The declarative power of GraphQL is back as you can clearly see the connection between `Course` and `Lecture` from the schema. However, if we query the gateway in the manner depicted in Listing 4.6, `null` will be returned for the `lectures` field.

```
{
    getCourses {
        code
        lectures {
            // ...
        }
    }
}
```

Code listing 4.6 Example query

The issue is in the fact that we did not define a resolver for the `lectures` field. We need to instruct GraphQL what to do when `lectures` field is queried. As demonstrated in Listing 4.7 we can delegate resolutiom of field to `lecturesByCourseCode` query.

In conclusion, while the distributed architecture was created to decentralize development, Schema Stitching centralizes the logic within the gateway layer and adds too much logic in it. Therefore, Schema Stitching approach was recently deprecated in favor of GraphQL Federation.

4.1.3 Composition in GraphQL Federation

“Netflix is known for its loosely coupled and highly scalable microservice architecture. Independent services allow for evolving at different paces and scaling independently. Yet they add complexity for use cases that span multiple services. Rather than exposing 100s of microservices

```
mergeSchemas({
    schemas: [
        courseService,
        lectureService
        typeExtensionDefinition
    ],
    resolvers: {
        Teacher: {
            lectures: {
                fragment: `... on Course { lectures }`,
                resolve(course, args, context, info) {
                    returns info.mergeInfo.delegateToSchema({
                        schema: courseService,
                        operation: 'query',
                        fieldName: 'lecturesByCourseCode',
                        args: {
                            code: course.code
                        },
                        context,
                        info
                    });
                },
            },
        },
    },
});
```

Code listing 4.7 Example of adding resolver in the gateway

to UI developers, Netflix offers a unified API aggregation layer at the edge. UI developers love the simplicity of working with one conceptual API for a large domain. Back-end developers love the decoupling and resilience offered by the API layer. But as our business has scaled, our ability to innovate rapidly has approached an invisible asymptote. As we've grown the number of developers and increased our domain complexity, developing the API aggregation layer has become increasingly harder. In order to address this rising problem, we've developed a federated GraphQL platform to power the API layer. This solves many of the consistency and development velocity challenges with minimal tradeoffs on dimensions like scalability and operability. We've successfully deployed this approach for Netflix's studio ecosystem and are exploring patterns and adaptations that could work in other domains."[9]

GraphQL Federation was developed to enhance Schema Stitching approach by relocating all relationship code from the gateway to the schemas themselves. This is accomplished with the help of special directives which are used to express relationships with other services, without the need for a centralized gateway to manage those relationships. The gateway layer can then use this information to construct a federated schema that can be queried as a single entity, while still maintaining the autonomy and independence of each service.

Both Schema Stitching and GraphQL Federation offer a significant advantage - they allow for a separation of concerns, as illustrated in Figure 4.2. This means that each service can define only the types and fields it is capable of resolving, resulting in more streamlined code. GraphQL Federation solves important problem, it enables developers to keep the code for a specific feature within a single subgraph isolated from unrelated concerns while creating a product-centric schema

with rich types that reflect the natural way developers would consume the graph.

Figure 4.2 Example of separation of concers [8]

4.2 Directives

The most important thing to know in GraphQL Federation is directives. Thus, we will go through all of them in this chapter and discuss their primary use cases.

4.2.1 @key

The `@key` directive allows developers to define unique identifiers for types. This is a mandatory requirement if a type needs to be referenced, extended, or used in another service within a federated graph. See Listing 4.8 for an example of `@key` directive definition.

```
type Teacher @key(fields: "username") {
    username: String
    firstName: String
    lastName: String
}
```

Code listing 4.8 Example of simple @key definition

In addition to defining a single key for a type, it is also possible to define compound keys (see Listing 4.9) or multiple keys (see Listing 4.10) . This can be particularly useful when a type is used in multiple services that require different fields for an identification.

```
type Teacher @key(fields: "username firstName") {
    username: String
    firstName: String
    lastName: String
}
```

Code listing 4.9 Example of compound @key definition

```
type Teacher @key(fields: "username") @key(fields: "firstName") {
    username: String
    firstName: String
    lastName: String
}
```

Code listing 4.10 Example of multiple @key definitions

4.2.2 @extends

The `@extends` directive allows a service to extend the type defined in another service. This can be done without modifying the original definition or requesting changes from the service that owns the type. The ability to extend types in this way helps with the separation of concerns and allows different service teams to collaborate more easily.

When using the `@extends` directive, it is important to note that it is mandatory that `@key` directive is defined for the type being extended. As an example, consider the original type definition as shown in Listing 4.11 and its extension as shown in Listing 4.12.

```
type Teacher @key(fields: "username") {
    username: String
    firstName: String
    lastName: String
}
```

Code listing 4.11 Original definition

```
type Teacher @key(fields: "username") @extends {
    username: String
    lectures: [Lecture]
}
```

Code listing 4.12 Extending definition

4.2.3 @external

The `@external` directive is used to mark a field as being owned by another service. It means that even though this field is defined in the schema, it is not resolved by the service itself but

instead by the owning service. See Listing 4.13 for the example.

```
type Teacher @key(fields: "username") {
    username: String
    firstName: String @external
    lastName: String @external
}
```

Code listing 4.13 Example of @external directive

4.2.4 @requires

The `@requires` directive is used to mark the dependencies of the field. In the Listing 4.14 `numOfLectures` and `lectureLength` fields are marked as `@external`, indicating that they are resolved by an external service. The `@requires` directive specifies that the `courseLength` field requires these fields to be resolved first before it can be computed correctly.

When a query is executed that includes the `courseLength` field, the gateway will first fetch the `numOfLectures` and `lectureLength` fields from the external service before resolving the `courseLength` field. This ensures that the required data is available before attempting to compute the final result.

```
type Course @key(fields: "code") {
    code: ID
    numOfLectures: Int @external
    lectureLength: Int @external
    courseLength: Int @requires(fields: "numOfLectures lectureLength")
}
```

Code listing 4.14 Example of @requires directive

4.2.5 @provides

The `@provides` directive allows a service to specify that field can be resolved by the service but only at a specific schema path. In Listing 4.15, the `firstName` field is marked as external and generally cannot be resolved by the current service. However, the `@provides` directive indicates that `teachers` query is capable of resolving `firstName` field.

4.3 Entity

In GraphQL Federation, an entity is an object type that has a unique identifier defined using the `@key` directive. Entities represent the data shared across multiple services. When a service defines the `@key` directive on one of its types, it indicates that the service can resolve an instance of the entity based on the identifier. Then, service must define an entity fetcher that can process the gateway request and resolve associated entities from it.

An entity fetcher is a function that takes in a list of entity references (see Listing 4.16) and returns the corresponding entity objects. The entity references are objects that have the same shape as the entity's unique identifier, which includes the entity's typename and any fields that

```
type Teacher {
    username: String
    firstName: String @external
}

type Query {
    teachers: [Teacher] @provides(fields: "firstName")
}
```

Code listing 4.15 Example of @provides directive

make up the identifier. The entity fetcher is responsible for mapping these references to actual entity objects, which can be fetched from a database, an API, or any other data source.

```
[
  {
    "__typename": "Teacher",
    "username": "teacher1"
  },
  {
    "__typename": "Teacher",
    "username": "teacher2"
  },
  //...
]
```

Code listing 4.16 Example of gateway entity request

4.4 Value types

On contrary, to entities value types are not derived from a specific service but rather belong to each individual service that defines them. For instance, if you need to share an enum across all services, you would need to define it in each of them. It is essential to note that the definition of the value type must be identical in each service to ensure successful schema composition.

4.5 Schema registry

"At its core, the schema registry is a version control system for our schema. It stores our schema's change history, tracking the types and fields that were added, modified, and removed. Similar to how we commit and push changes to our codebase to a Git repository, we should push every new version of our schema to the registry. Thanks to the schema registry, we can track variants of the same graph that are deployed in different environments, such as staging and production. We can run schema checks to detect when a potential change might break one of our clients."[10]

Schema Registrty is a crucial concept in the GraphQL Federation concept. See Federation with Schema Registry and Uplink section for usage example.

4.6 Federation with Schema Registry and Uplink

There are two ways to utilize the gateway. The first way involves providing the gateway with service URLs, which it then scrapes and uses to create a federated schema. The second approach is to provide the gateway with an already composed schema. While the first approach may be convenient for local development, it is strongly discouraged for production environments due to its possible failure, the need for introspection, and the inability to reload the schema without redeploying.

Instead, the approach depicted in Figure 4.3 is recommended. This approach involves utilizing a Schema Registry that not only tracks changes but also validates and composes the schemas. If the composition is successful, the registry updates the uplink, which is periodically queried by the gateway for the updates. This approach has numerous benefits:

- **Router stability** - you can modify subgraph schemas (and even add or remove entire subgraphs) without needing to modify or redeploy your router. Your router is the point of entry to your entire graph, and it should maximize its uptime.
- **Composition stability** - whenever your router obtains an updated configuration from Apollo, it knows that the updated set of subgraph schemas will compose, because the configuration includes the composed supergraph schema.

  The router also knows that your subgraphs are prepared to handle operations against the updated set of schemas. This is because your subgraphs should publish their updated schemas as part of their deployment, meaning they're definitely running by the time the router is aware of the configuration change.

  And whenever a subgraph accidentally pushes a schema change that doesn't compose, Uplink continues to provide the most recent valid configuration to your router.
- **Schema flexibility** - by using a configuration manager that's external to your router, you help ensure the safety of certain schema changes. For example, if you want to migrate a type or field from one subgraph's schema to another, you can perform this migration safely only if you externalize your configuration.

[11]

Though this architecture proves to be highly efficient, it can pose considerable challenges for smaller projects. One of the main obstacles is the lack of reliable open-source solutions for the Schema Registry and Uplink. Many large corporations develop these tools in-house, with no public access available. However, there is some hope as companies like Netflix have indicated their intention to release such tools in the near future.

Thus, there are two approaches left: either develop your own solution or go for Managed Federation from Apollo. While the latter is an efficient choice, it is hosted on their cloud which may be not suitable for some projects.

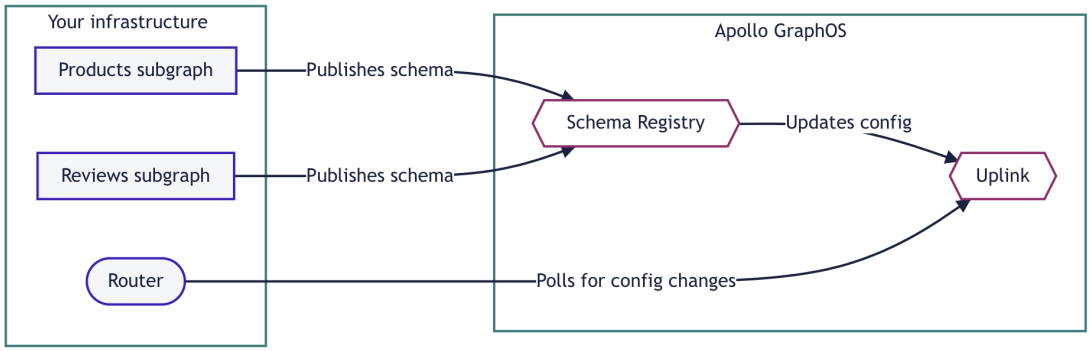

Figure 4.3 Example of managed federation [11]

Chapter 5

GraphQL Federation - implementation

This section is dedicated to discussing the practical aspects of the thesis, focusing on the most essential concepts, issues, and their solutions required to develop a GraphQL Federation-based project in Java. It is divided into two major subsections. The first one is presented in the form of a tutorial, which introduces Netflix DGS and showcases the critical problems and their solutions. It is worth noting that all examples provided in this chapter are taken from the practical part of the thesis. To keep things concise, unrelated code, such as JPA repositories and DTO converters, is omitted from the examples. However, for a comprehensive understanding, please refer to the practical part of the thesis. The second part of chapter will concentrate on the practical aspects of the thesis, outlining the work completed, its intended usage, and other relevant details.

5.1 Netflix DGS

5.1.1 Dependencies

When starting your project, the first task is to make sure all necessary dependencies are included. You can find a detailed list of required dependencies in the `build.gradle` file, located in the practical part of the thesis. Here, I want to encourage the usage of BOM which helps to align dependencies and avoid version conflicts.

The DGS team provides two variations of BOM: `graphql-dgs-platform`, which manages only the DGS framework dependencies, and `graphql-dgs-platform-dependencies`, which additionally handles all dependencies used by the DGS framework itself, such as Spring, Jackson and others. Listing 5.1 demonstrates an example of BOM usage.

5.1.2 Schema

There are two approaches in the GraphQL world: client-first and schema-first. I will be focusing on the latter one in this thesis. The DGS framework has been set up to look for schema files in the `resources/schema` directory. Therefore, you must create a `schema.graphqls` file within this directory. Then, you can proceed with definition of your schema.

```
dependencies {

    ↪  implementation(platform("com.netflix.graphql.dgs:graphql-dgs-platform-dependencies:3.

    // BOM will take care of dependencies below so there is no need to
    ↪  manually define versions
    implementation "com.netflix.graphql.dgs:graphql-dgs-spring-boot-starter"
    implementation
    ↪  'com.netflix.graphql.dgs:graphql-dgs-subscriptions-websockets-autoconfigure'
}
```

Code listing 5.1 Example of using BOM

5.1.3 Codegen plugin

Once you've defined the schema, it's common to use the DTO pattern and create POJO classes for the schema types. Creating these classes manually can be tedious and time-consuming, so it's better to use the Codegen plugin, which automates this process. To include the plugin in your `build.gradle` file, follow the example shown in Listing 5.2.

```
plugins {
    id "com.netflix.dgs.codegen" version "5.6.9"
}
```

Code listing 5.2 Example of adding codegen plugin

After adding the Codegen plugin to your project, it automatically creates a Gradle task `generateJava` under the build group. This task has many configuration options that can be set using standard Gradle syntax, as shown in Listing 5.3.

```
generateJava {
    schemaPaths = ["${projectDir}/src/main/resources/schema"]
    packageName = 'com.example.graphqlthesis.dto'
}
```

Code listing 5.3 Example of configuring codegen plugin

By running `gradle generateJava`, the Codegen plugin produces the required Java classes similar to the types defined in the schema and stores them in the `/build/generated/dgs-codegen` directory.

Additionally, the Codegen plugin provides extra features like generating a `DgsConstants` class with constants for all field and type names in the schema, or even a type-safe query API.

5.1.4 Type safe query API

One of the valuable features offered by the Codegen plugin is the ability to generate a type-safe query API, which is beneficial for testing and server-to-server communication. At its core, this API promotes compile-time safety and enhanced readability. This is achieved by generating Java

classes that are used to construct queries. You can enable query API generation the same way as demonstrated in Listing 5.4. Also, see Listing 5.5 for example of using query API.

```
generateJava {
    generateClient = true
}
```

Code listing 5.4 Example of configuring codegen plugin to generate query API

```
private final GraphQLQueryRequest getLiteratureQuery = new
↪  GraphQLQueryRequest(
    new GetProseminarsGraphQLQuery(),
        new GetProseminarsByCodeProjectionRoot()
                .code()
                .parallel()
        .studentCount());
```

Code listing 5.5 Example of using query API

It's important to note that only one `generateJava` gradle task can exist per one java module. This essentially means that you cannot have multiple schemas within a single module. Instead, you must create separate modules for each new schema. So, you will need to create a new module where to locate other server schema and then use this module as a dependency.

5.1.5 Data Fetcher

Data Fetchers (also known as data resolvers) are a key concept in the DGS framework. They are responsible for retrieving data from a data source in response to a GraphQL query. Listing 5.6 demonstrates an example of creating a data fetcher for the `getSubjects` query.

```
@DgsComponent
@RequiredArgsConstructor
public class TeacherDataFetcher {

        private final ISubjectService subjectService;
        private final SubjectMapper subjectMapper;

        @DgsData(parentType = DgsConstants.QUERY.TypeName, field =
        ↪  DgsConstants.QUERY.GetSubjects)
        public List<Subject> getSubjects() {
                return subjectService.findAll().stream()
                        .map(subjectMapper::toDto)
                        .toList();
        }
}
```

Code listing 5.6 Example of data fetcher

First, note the `@DgsComponent` annotation. This annotation informs the DGS framework to examine the bean for instructions and thus all DGS related code should be placed under this annotation. Second thing to note, is the `@DgsData` annotation which allows to designate a function as responsible for a specific part of the schema. This annotation has two parameters:

- **parentType** - a mandatory parameter representing the name of the type containing the required field.
- **field** - an optional parameter representing the name of the field for which the function is resolving the data.

If the field parameter is omitted, it will be inferred from the name of the function. It is worth mentioning that there are three shortcut annotations that can be used instead of `@DgsData`:

- **DgsQuery** - the parent type is always equal to Query.
- **DgsMutation** - the parent type is always equal to Mutation.
- **DgsSubscription** - the parent type is always equal to Subscription.

Thus, all definitions shown in the Listing 5.7 are equivalent.

```
@DgsData(parentType = "Query", field = "getSubjects")
public List<Subject> function() {}

@DgsQuery(field = "getSubjects")
public List<Subject> function() {}

@DgsQuery
public List<Subject> getSubjects() {}
```

Code listing 5.7 Example of equivalent definitions

To conclude, all you need to create data fetcher is to use a combination of `@DgsComponent` and `@DgsData` annotation with regular java function.

5.1.6 Nested data fetcher

In GraphQL, we have the option to create data fetchers for any field defined in a schema. Thus, we can create multiple data fetchers for a single type, with a maximum of one per field. There is no need to create nested data fetchers for types whose fields can be loaded from the database via a single query. However, let us consider the schema defined in Listing 5.8, and imagine that the `additionalField` requires an additional database operation to be resolved.

It would be inefficient to execute an additional query for the field even if it is not requested by the client. To avoid this, we can create a nested data fetcher that resolves `additionalField` only when requested by the client. See an example of nested data fetcher in Listing 5.9.

Defining a nested data fetcher is similar to creating a regular one, with the main distinction being that the `parentType` of `@DgsData` is now set to `Subject` not `Query`.

As a result, the nested data fetcher only fetches the data when necessary. It's also worth noting the `DgsDataFetchingEnvironment` class, which provides access to the data fetching context, including the query, data loaders, source object, and more. One of its powerful features is the local context. You can populate it yourself, and it will be passed to the nested data fetcher. This enables query optimization, as demonstrated in Listing 5.10, where reviews are loaded only once

```
type Subject {
    code: ID
    titleCz: String
    titleEn: String
    garant: String
    addditionalField: [AdditionalField]
}
```

Code listing 5.8 Example schema

```
@DgsComponent
@RequiredArgsConstructor
public class TeacherDataFetcher {

        private final ISubjectService subjectService;
        private final SubjectMapper subjectMapper;

        @DgsQuery(field = DgsConstants.QUERY.GetSubjects)
        public List<Subject> getSubjects() {
                return subjectService.findAll().stream()
                        .map(subjectMapper::toDto)
                        .toList();
        }

    @DgsData(parentType = DgsConstants.SUBJECTS.TYPE_NAME, field =
    ↪ DgsContants.SUBJECTS.AdditionalInfo)
    public AdditionalInfo additionalInfo(DgsDataFetchingEnvironment dfe) {
        Subject subject = dfe.getSource();
        // ...
    }
}
```

Code listing 5.9 Example of using nested data fetcher

and then passed to the nested data fetcher through the local context. This approach allows for fetching `AdditionalInfo` only once using a batch request, instead of loading them one by one which requires higher number of database operations.

5.1.7 Extending type

Let's go through an example of type extension in Netflix DGS, by examining the schema defined in Listing 5.11.

As described in Entity section, it is essential to be prepared for entity queries from the gateway when extending the type. To handle entity query, it is needed to create an entity fetcher by defining a function that accepts a `Map<String, Object>` as a parameter and constructs corresponding `Subject` objects based on the entity reference. Last step, is to annotate the function with `@DgsEntityFetcher` and specify the name of type we are resolving. See an example illustrated in the Listing 5.12.

Next, we need to create regular data fetcher where instruct GraphQL on how `literature`

```
@DgsQuery(field = DgsConstants.QUERY.GetSubjects)
public DataFetcherResult<List<Subject>> getSubjects(DataFetchingEnvironment
↪  dfe) {
    List<Subject> subjects = subjectService.findAll().stream()
        .map(subjectMapper::toDto)
                .toList();

    if (dfe.getSelectionSet().contains("additionalInfo") {
        Map<Integer, List<AdditionalInfo>> additionalInfo = // load
        ↪  additionalInfo

        return DataFetcherResult.<List<Subject>>result()
            .data(subjects)
            .localContext(additionalInfo)
            .build()
    } else {
        return DataFetcherResult.<List<Subject>>result()
            .data(subjects)
            .build()
    }
}

@DgsData(parentType = DgsConstants.SUBJECTS.TYPE_NAME, field =
↪  DgsContants.SUBJECTS.AdditionalInfo)
public AdditionalInfo additionalInfo(DgsDataFetchingEnvironment dfe) {
    Subject subject = dfe.getSource();

    Map<Integer, List<AdditionalInfo>> additionalInfo = dfe.getLocalContext();
    return additionalInfo.get(subject.getCode());
}
```

Code listing 5.10 Example of using local context of DataFetchingEnvironment

```
type Literature {
    isbn: ID
    code: String
    title: String
}

type Subject @key(fields: "code") @extends {
    code: ID
    literature: [Literature]
}
```

Code listing 5.11 Example schema

field should be fetched. You can see example in the Listing 5.13.

In described example, literature is loaded individually for each subject, resulting in the well-known **N+1** problem, which negatively impacts the performance. To address this issue, consider

```
@DgsEntityFetcher(name = DgsConstants.SUBJECT.TYPE_NAME)
public Subject subject(Map<String, Object> values) {
        return Subject.newBuilder()
                .code((String) values.get("code"))
                .build();
}
```

Code listing 5.12 Example of entity fetcher

```
@DgsData(parentType = DgsConstants.SUBJECT.TYPE_NAME, field =
↪  DgsConstants.SUBJECT.Literature)
public List<Literature> literature(DgsDataFetchingEnvironment
↪  dataFetchingEnvironment) {
        Subject subject = dataFetchingEnvironment.getSource();
        return literatureService.findByCode(subject.getCode()).stream()
                .map(literatureMapper::toDto)
                .toList();
        }
```

Code listing 5.13 Example of fetcher

using the Data Loader approach, which is discussed in detail in the following chapter.

5.1.8 Data Loader

Data Loaders provide a mechanism to efficiently fetch data by batching multiple requests, thereby mitigating the N+1 problem commonly encountered in GraphQL applications. By aggregating multiple keys and retrieving the corresponding data in a single batch request, Data Loaders significantly improve performance and reduce the load on underlying data sources.

First thing to do, is to make sure that you are supporting batch requests in underlying layers such as repositories. Next, create a class, annotate it with the `@DgsDataLoader` annotation, and implement the `BatchLoader` interface by overriding the `load` method. An example of a data loader can be found in Listing 5.14.

A frequent pitfall when crating data loader is overlooking scenarios where no associated literature exists for a given entity key. In these instances, the entire query fails, and no partial data is returned. To address this issue, consider implementing the `MappedBatchLoader` interface which maps each key to the response using map. For this scenario, you would need to rewrite data loader as shown in the Listing 5.15.

When data loader is ready it is time to use it in the data fetcher. As mentioned before `DataFetchingEnvironment` contains all context information and can be used to get required data loader. See Listing 5.16 for the example.

There are few features of data loader that are worth noting. First one is the possibility to optimize thread pool by creating new thread if all others are already utilized. For this purpose we need to define bean in configuration class as shown in the Listing 5.17.

After defining the bean in the configuration class, the next step is to autowire the `Executor` bean in the data loader. From there, DGS will take care of the rest of the process automatically.

The second noteworthy feature of the data loader is its ability to dispatch queries. You can define rules that determine whether a query should be resolved with a data loader based on factors such as query depth or estimated execution time as shown in the Listing 5.18.

```
@DgsDataLoader
@RequiredArgsConstructor
public class LiteratureDataLoader implements BatchLoader<List<Literature>> {

        private final ILiteratureService literatureService;
        private final LiteratureMapper literatureMapper;

        @Override public CompletionStage<List<Literature>> load(Set<String>
        ↪  keys) {
                return CompletableFuture.supplyAsync(() ->
                        literatureService.findAllByIds(keys).stream()
                .map(literatureMapper::toDto)
                .toList());
        }
}
```

Code listing 5.14 Example of data loader

```
@Override public CompletionStage<Map<String, List<Literature>>>
↪  load(Set<String> keys) {
        return CompletableFuture.supplyAsync(() ->
                literatureService.findAllByIds(keys).entrySet().stream()
                        .map(entry -> Map.entry(entry.getKey(),
                        ↪  entry.getValue().stream()
                                .map(literatureMapper::toDto).toList()))
                        .collect(Collectors.toMap(Map.Entry::getKey,
                        ↪  Map.Entry::getValue)));
}
```

Code listing 5.15 Example of mapped data loader

```
@DgsData(parentType = DgsConstants.SUBJECT.TYPE_NAME, field =
↪  DgsConstants.SUBJECT.Literature)
public CompletableFuture<List<Literature>>
↪  literatures(DgsDataFetchingEnvironment dataFetchingEnvironment) {
        DataLoader<String, List<Literature>> dataLoader =
        ↪  dataFetchingEnvironment.getDataLoader(LiteratureDataLoader.class);
        Subject subject = dataFetchingEnvironment.getSource();
        return dataLoader.load(subject.getCode());
}
```

Code listing 5.16 Example of using data loader

Another useful feature of the data loader is the ability to limit the maximum batch size. To implement this feature, you can specify the maximum batch size in the `@DgsDataLoader` annotation, as shown in the Listing5.19

```
public class DataLoaderConfig {

        @Bean(name = "LiteratureDataLoaderExecutor") Executor
        ↪  literatureDataLoaderExecutor() {
                return Executors.newCachedThreadPool();
        }
}
```

Code listing 5.17 Example of cached thread pool

```
@DgsDispatchPredicate private final DispatchPredicate predicate =
↪  DispatchPredicate.dispatchIfLongerThan(Duration.ofSeconds(2));
@DgsDispatchPredicate private final DispatchPredicate predicate =
↪  DispatchPredicate.dispatchIfDepthGreaterThan(2);
```

Code listing 5.18 Example of data loader dispatching

```
@DgsDataLoader(maxBatchSize = 300)
```

Code listing 5.19 Example of limiting batch size

5.1.9 Gateway

We can use any compatible to GraphQL Federation specification implementation of the Gateway. There are two most popular ones: Apollo Router, which is written in Rust language and designed for high performance and Apollo Server used with Apollo Gateway extensions which is the predecessor of the Apollo Router. In this chapter, I will focus on the later one.

Installing the gateway is a straightforward process. First, you need to run the command `npm init` to initialize your project, and then run `npm install @apollo/gateway @apollo/server graphql` to install the necessary dependencies for the gateway. Once this is complete, you will have a fully functional gateway at your disposal.

However, in order to use the gateway to compose services, we need to inform it about the addresses of the services to be used. This can be accomplished by using the `IntrospectAndCompose` function, which takes a list of URLs as a parameter. Listing 5.20 provides an example of how this function can be used.

```
const gateway = new ApolloGateway({
  supergraphSdl: new IntrospectAndCompose({
    subgraphs: [
      { name: 'teacherService', url: 'http://localhost:8080' },
      { name: 'courseService', url: 'http://localhost:8081' },
    ],
  }),
});
```

Code listing 5.20 Example of using IntrospectAndCompose

Note that while `IntrospectAndCompose` is perfect fit for local development, it is strongly discouraged from the production use. See Federation with Schema Registry and Uplink for detailed information.

5.1.10 Authentication and Authorization

Suppose we wish to restrict access to one of our queries so that it is only available to users with administrative privileges. DGS framework support integration with Spring Security via `@Secured` annotation so lets setup in-memory user details service as shown in the Listing 5.21.

Next, we can just simply apply `@Secured` annotation on data fetcher or service API as shown in the Listing 5.22.

This approach functions well until we attempt to use the service with the gateway. If we use the `IntrospectAndCompose` function, it will fail with a 401 response status. This occurs because our service now requires authentication to access its endpoints and it makes introspection query fail. To address this issue, we can use the `introspectionHeaders` function in `IntrospectAndCompose` to add an authorization header for the schema composition process. Refer to Listing 5.23 for an example.

With the composition issue resolved, another problem arises when attempting to execute the secured query even with the correct authorization header, it fails with a 401 error. The issue here is that the gateway receives the authorization header but does not propagate it to the underlying services. We can address this by using the `buildService` function, as demonstrated in Listing 5.24. `buildService` allows to modify every request sent to the underlying services so we can make use of it to propagate authorization header.

At this point, we once again encounter a composition failure with a 401 error code. The `buildService` function is applied during each request, which causes it to overwrite the header set by `introspectionHeaders`. As a result, we need to differentiate between regular and introspection requests within the `buildService`. To achieve this, let's modify the `AuthenticatedDataSource` as shown in the Listing 5.25.

5.1.11 Subscription

Subscriptions enable clients to subscribe to real-time updates from a server by specifying a query that will be executed whenever new data is available to be pushed from the server. Defining a subscription is quite similar to creating a query or mutation, but it necessitates the use of reactive programming concepts. See Listing 5.26 for an example of subscription.

For a complete example, refer to the implementation of the `literatureAdded` subscription which can be found in the practical section of the thesis.

5.1.12 Instrumentation

Monitoring the state of an application and swiftly detecting issues is essential for production environments. To this end, DGS provides various functionalities to address these tasks. One key feature is the concept of instrumentation, which allows developers to hook code before or after query execution. To create custom instrumentation, developers can create a spring bean and implement the `SimpleInstrumentation` interface. An example of custom instrumentation that logs the data fetcher name and type upon the completion of any query can be found in Listing 5.27.

```
@Configuration
@EnableWebSecurity
@EnableMethodSecurity(securedEnabled = true)
public class SecurityConfig {

        @Bean public SecurityFilterChain securityFilterChain(HttpSecurity
        ↪  httpSecurity) throws Exception {
                httpSecurity
                        .csrf().disable()
                        .authorizeHttpRequests(auth ->
                        ↪  auth.anyRequest().authenticated())
                        .sessionManagement(session ->
                        ↪  session.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
                        .httpBasic(Customizer.withDefaults());

                return httpSecurity.build();
        }

        @Bean public UserDetailsService userDetailsService(PasswordEncoder
        ↪  passwordEncoder) {
                UserDetails admin = User.builder()
                        .username("admin")
                        .password(passwordEncoder.encode("admin"))
                        .roles("ADMIN")
                        .build();

                UserDetails user = User.builder()
                        .username("user")
                        .password(passwordEncoder.encode("user"))
                        .roles("USER")
                        .build();

                return new InMemoryUserDetailsManager(admin, user);
        }

        @Bean PasswordEncoder passwordEncoder() {
                return new BCryptPasswordEncoder();
        }
}
```

Code listing 5.21 In-memory user details service

```
@Secured("ROLE_ADMIN")
List<LectureEntity> findAll();
```

Code listing 5.22 Example of using @Secured

5.1.13 Tracing

Another crucial aspect is query tracing. Although it can be implemented manually through instrumentation, there is no need to do so, as Apollo already provides available solutions. All

```
const gateway = new ApolloGateway({
    supergraphSdl: new IntrospectAndCompose({
        subgraphs: [
            { name: 'teacher_service', url: 'http://localhost:8080/graphql' },
            { name: 'lecture_service', url: 'http://localhost:8081/graphql' },
        ],
        introspectionHeaders: {
            Authorization: 'admin admin',
        }
    })
});
```

Code listing 5.23 Example of using introspection headers

```
class AuthenticatedDataSource extends RemoteGraphQLDataSource {
    willSendRequest({ request, context }) {
        request.http.headers.set('Authorization', context.authorization);
    }
}

const gateway = new ApolloGateway({
    supergraphSdl: new IntrospectAndCompose({
        subgraphs: [
            { name: 'teacher_service', url: 'http://localhost:8080/graphql' },
            { name: 'lecture_service', url: 'http://localhost:8081/graphql' },
        ],
        introspectionHeaders: {
            Authorization: 'admin admin',
        }
    }),
    buildService({name, url}) {
        return new AuthenticatedDataSource({url})
    }
});
```

Code listing 5.24 Example of using buildService

you need to do is register an `Instrumentation` bean with the `TracingInstrumentation` implementation, as demonstrated in Listing 5.28.

While this approach works well for regular GraphQL queries, it doesn't support federated ones. To enable federated traces, you need to add dependency demonstrated in Listing 5.29:

After adding the dependency, register the instrumentation bean with `FederatedTracingInstrumentation` as shown in the Listing 5.30.

5.1.14 Metrics

Another essential aspect for every production ready application is metrics. Luckily metrics are implemented out of the box by DGS framework. Simply, add dependency shown in the Listing 5.31.

```
class AuthenticatedDataSource extends RemoteGraphQLDataSource {
    willSendRequest({ request, context }) {
        if (request.query === 'query __ApolloGetServiceDefinition__ { _service
        ↪  { sdl } }') {
            context.authorization = 'Basic dXNlcjp1c2Vy'
        }

        request.http.headers.set('Authorization', context.authorization);
    }
}
```

Code listing 5.25 Example of using AuthenticatedDataSource

```
        @DgsData(parentType = DgsConstants.SUBSCRIPTION_TYPE, field =
        ↪  DgsConstants.SUBSCRIPTION.LiteratureAdded)
        public Publisher<Literature> literatureAdded() {
                return literatureService.getLiteraturePublisher();
        }
```

Code listing 5.26 Example of subscription

As you can see, metrics are implemented using Micrometer, so you can utilize any compatible solution, such as Prometheus and Grafana, to display them. Of course, you can use Micrometer to add your custom metrics too.

5.1.15 Custom scalar

GraphQL comes with predefined scalars, but sometimes there is a need to add custom ones, such as `Long` or `Date` scalars. Let's look at an example of a custom scalar that will represent java `Long` class. The first step is to create a class that implements the `Coercing` interface and annotate it with `@DgsScalar` annotation. To implement the `Coercing` interface, you need to provide instructions on how to serialize and deserialize the value. You can see an implementation of custom Long scalar in the Listing 5.32.

Then, you can simply register and use scalar in the schema as shown in the Listing 5.33.

5.1.16 Dynamic schemas

Sometimes, it can be helpful to modify the schema during runtime. This can be achieved by creating a function with a return type of `TypeDefinitionRegistry` and annotating it with `@DgsTypeDefinitionRegistry`, as demonstrated in Listing 5.34.

Once the query is defined, it's time to add a data fetcher. This can be accomplished by creating a function with a return type of `GraphQLCodeRegistry` and annotating it with `@DgsCodeRegistry`, as shown in Listing 5.35.

5.1.17 Testing

DGS supports testing with the `spring-boot-test` so it is pretty straightforward task. Simply annotate your class with `@SpringBootTest` and autowire the `DgsQueryExecutor` class. To reduce

```
@Component
@Slf4j
public class LoggingTracingInstrumentation extends SimpleInstrumentation {

        @Override public DataFetcher<?> instrumentDataFetcher(DataFetcher<?>
        ↪  dataFetcher, InstrumentationFieldFetchParameters parameters) {
                if(parameters.isTrivialDataFetcher()) {
                        return dataFetcher;
                }

                return environment -> {
                        Object result = dataFetcher.get(environment);
                        if(result instanceof CompletableFuture) {
                                ((CompletableFuture<?>) result)
                                        .whenComplete((r, ex) ->
                                        ↪  log.info("Async datafetcher {}
                                        ↪  finished execution",
                                        ↪  findDatafetcherTag(parameters)));
                        } else {
                                log.info("Datafetcher {} finished execution",
                                ↪  findDatafetcherTag(parameters));
                        }

                        return result;
                };
        }

        private String findDatafetcherTag(InstrumentationFieldFetchParameters
        ↪  parameters) {
                GraphQLOutputType type =
                ↪  parameters.getExecutionStepInfo().getParent().getType();
                GraphQLObjectType parent;
                if (type instanceof GraphQLNonNull) {
                        parent = (GraphQLObjectType) ((GraphQLNonNull)
                        ↪  type).getWrappedType();
                } else {
                        parent = (GraphQLObjectType) type;
                }

                return  parent.getName() + "." +
                ↪  parameters.getExecutionStepInfo().getPath().getSegmentName();
        }
}
```

Code listing 5.27 Example of simple instrumentation

the number of components set up for the test, you can limit the number to `DgsAutoConfiguration` and the classes you're actually testing.

For building of queries, it's highly recommended to use the Type safe query API as demonstrated in Listing 5.36. This approach streamlines the process and makes it easier to manage

```
@Configuration
public class AppConfig {

    @Bean
    public Instrumentation tracingInstrumentation(){
        return new TracingInstrumentation();
    }
}
```

Code listing 5.28 Example of Tracing Instrumentation

```
implementation
↪  'com.apollographql.federation:federation-graphql-java-support:$latestVersion'
```

Code listing 5.29 Apollo dependency

```
@Configuration
public class AppConfig {

    @Bean
    public Instrumentation tracingInstrumentation(){
        return new FederatedTracingInstrumentation();
    }
}
```

Code listing 5.30 Example of Federated Tracing Instrumentation

```
implementation 'com.netflix.graphql.dgs:graphql-dgs-spring-boot-micrometer'
```

Code listing 5.31 DGS metrics dependency

complex queries.

5.2 Proof of Concept

5.2.1 Domain

The proof of concept is based on the `uvazky` database which is illustrated in Figure 5.2. For demonstration purposes, only a portion of the database is used, specifically the `Teacher`, `Subject`, `Lecture`, `Literature` and `Seminar` entities.

To showcase the capabilities of GraphQL Federation, separate services have been created for each entity, resulting in Teacher, Subject, Lecture, Seminar, and Literature services. This approach allows us to envision each individual service being managed by a different team, highlighting the real power of GraphQL Federation.

Additionally, the proof of concept includes a gateway that composes schemas and acts as an entry point for the application. The overall architecture is depicted in Figure 5.1.

```
@DgsScalar(name="Long")
public class LongScalar implements Coercing<Long, String> {

        @Override public String serialize(@NotNull Object dataFetcherResult)
        ↪  throws CoercingSerializeException {
                if (dataFetcherResult instanceof Long result) {
                        return String.valueOf(result);
                } else {
                        throw new CoercingSerializeException("Invalid Long
                        ↪  type");
                }
        }

        @Override public @NotNull Long parseValue(@NotNull Object input)
        ↪  throws CoercingParseValueException {
                try {
                        if (input instanceof String val) {
                                return Long.valueOf(val);
                        } else {
                                throw new
                                ↪  CoercingParseValueException("Expected
                                ↪  input type is String");
                        }
                } catch (Exception e) {
                        throw new CoercingParseValueException(e);
                }
        }

        @Override public @NotNull Long parseLiteral(@NotNull Object input)
        ↪  throws CoercingParseLiteralException {
                if (input instanceof StringValue stringValue) {
                        try {
                                return Long.valueOf(stringValue.getValue());
                        } catch (Exception e) {
                                throw new CoercingParseLiteralException(e);
                        }
                } else {
                        throw new CoercingParseLiteralException("Value is not
                        ↪  a  valid long scalar");
                }
        }
}
```

Code listing 5.32 Custom Long scalar

Furthermore, there is additional standalone project called OpenAPIToGraphQL which serves as an illustration for the OpenAPI to GraphQL section included in the thesis.

```
type Lecture {
    parallel: Long
}

scalar Long
```

Code listing 5.33 Example of using custom scalar in schema

```
@DgsTypeDefinitionRegistry
public TypeDefinitionRegistry registry() {
    TypeDefinitionRegistry typeDefinitionRegistry = new
    ↪  TypeDefinitionRegistry();

        ObjectTypeExtensionDefinition query =
        ↪  ObjectTypeExtensionDefinition.newObjectTypeExtensionDefinition().
        ↪  name("Query").fieldDefinition(
                FieldDefinition.newFieldDefinition().name("getTeachers").
                ↪  type(new ListType(new TypeName("Teacher"))).build()
        ).build();

        typeDefinitionRegistry.add(query);
        return typeDefinitionRegistry;
}
```

Code listing 5.34 Example of using TypeDefinitionRegistry

```
@DgsCodeRegistry
public GraphQLCodeRegistry.Builder registry(GraphQLCodeRegistry.Builder
↪  codeRegistryBuilder, TypeDefinitionRegistry registry) {
        DataFetcher<List<Teacher>> df = dfe ->
        ↪  teacherService.findAll().stream()
           .map(teacherMapper::toDto)
                .toList();

        FieldCoordinates fieldCoordinates =
        ↪  FieldCoordinates.coordinates("Query", "getTeachers");
        return codeRegistryBuilder.dataFetcher(fieldCoordinates, df);
}
```

Code listing 5.35 Example of using GraphQLCodeRegistry

5.2.2 Service architecture

In the proof of concept, each service follows the three-tier architecture, dividing the application into three separate logical components to achieve modularity, maintainability, and scalability.

Each service has its own PostgreSQL database. Consequently, the data access tier is represented by JPA repositories, which query data from the associated database.

The business tier is represented by services. This layer is quite thin and primarily delegates

```
@Test
void getLiteratures() {
    // setup data
        var projectionRoot = new GetLiteraturesProjectionRoot().isbn().code();
        GraphQLQueryRequest graphQLQueryRequest = new
        ↪  GraphQLQueryRequest(GetLiteraturesGraphQLQuery.newRequest().build(),
        ↪  projectionRoot);

    // execute query
        List<Literature> literature =
        ↪  dgsQueryExecutor.executeAndExtractJsonPathAsObject(graphQLQueryRequest.serialize(
        ↪  "data.getLiteratures", new TypeRef<List<Literature>>() {});

    // assert result
        Assertions.assertEquals(4, literature.size());
}
```

Code listing 5.36 Example of testing data fetcher

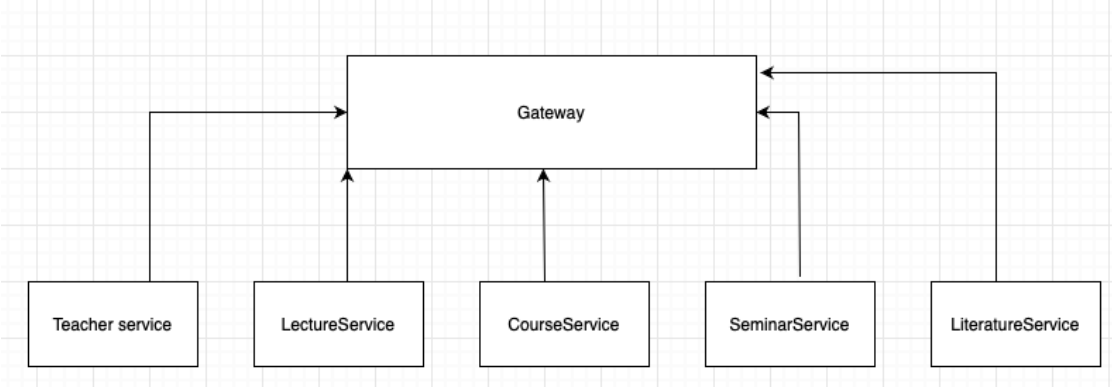

Figure 5.1 Overall proof of concept architecture

calls to the data access tier. The simplicity of this layer is due to the demonstration purposes of the proof of concept, which does not require complex business logic.

The presentation layer is represented by data fetchers, which are responsible for handling user input and presenting the response.

5.2.3 How to use

5.2.3.1 Services

All five services and their associated databases are designed to be launched as Docker containers. To facilitate this, a `/docker` folder is located in the root of each service, containing `db` and `server` sub-folders. The `server` folder holds a `Dockerfile` responsible for creating the server image, while the `db` folder contains a `Dockerfile` that creates the database image and populates it with data found in the `scripts` sub-folder of `db`.

The images are used in the `docker-compose.yml` file, which creates the containers. You can launch services one by one, going to their individual folder and executing `docker compose up` or

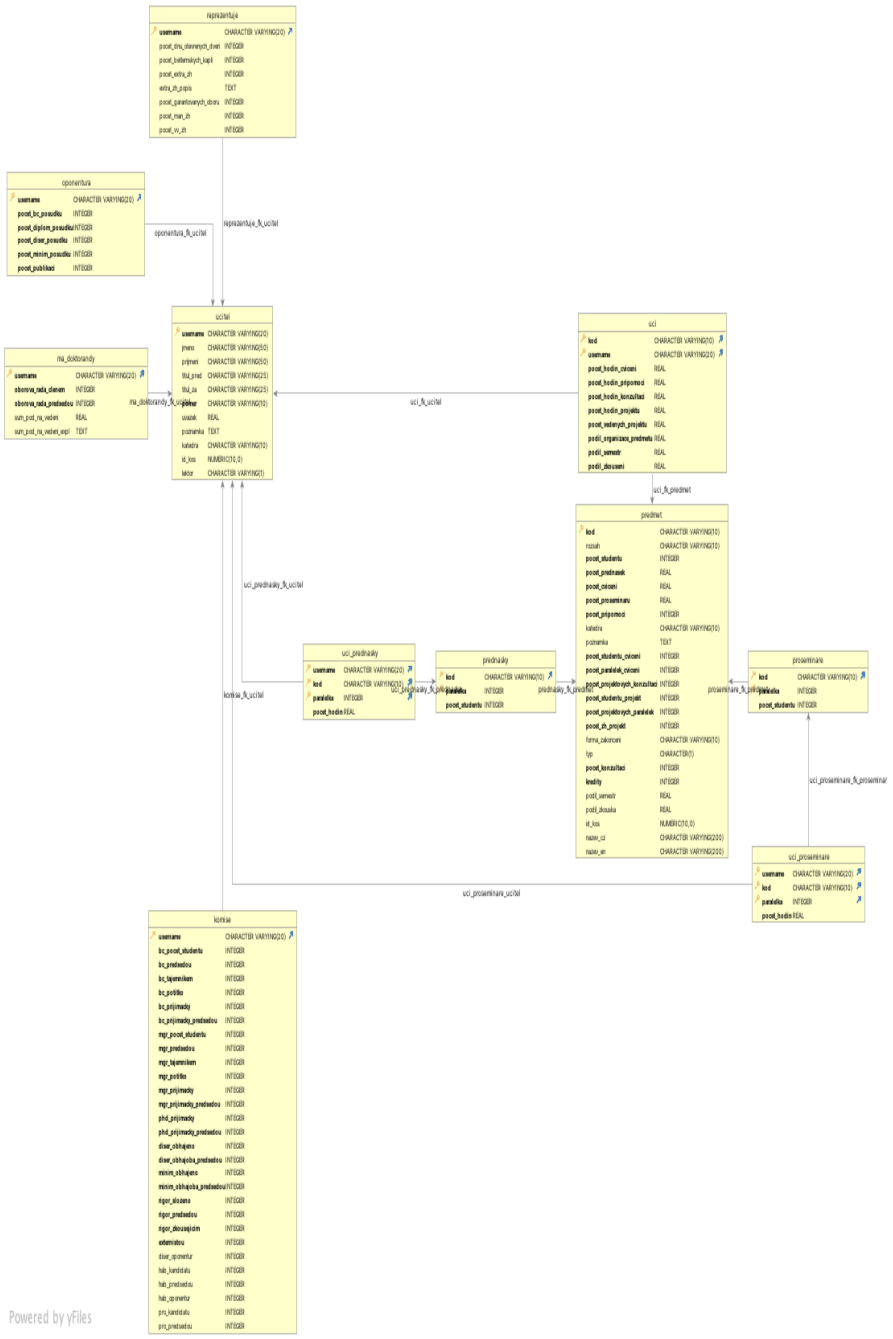

Figure 5.2 Uvazky database

you can execute `docker compose up` in the root of the main project to get all the services and databases running. Each service can be manually tested by going to the `/graphiql` endpoint where built-in GUI for query execution is located. **Note that LiteratureService requires authentication with username and password equal to admin. For this, you need to add authorization header as show in the Figure 5.3.**

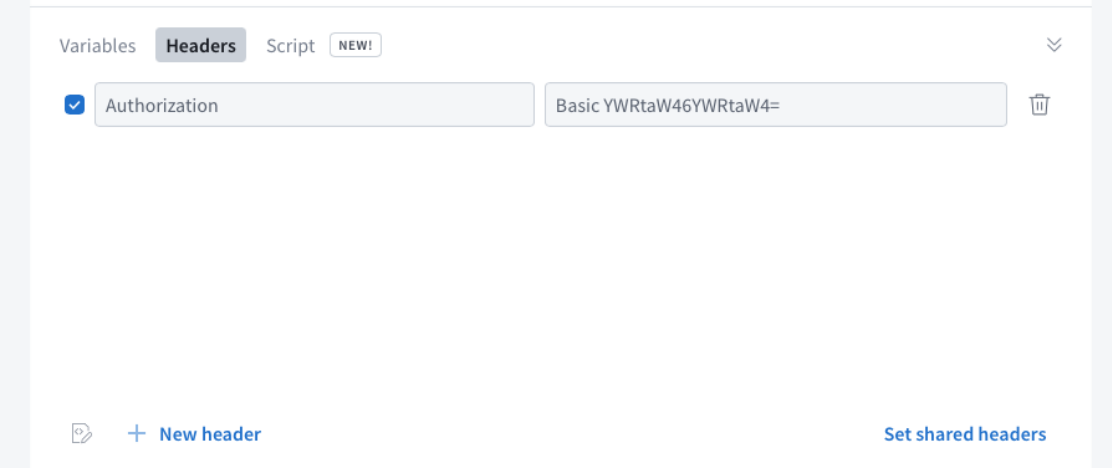

Figure 5.3 Example of adding authorization header

5.2.3.2 Gateway

Once all services and databases are up and running, you can start the gateway. For this, ensure that Node.js is installed on your system and then, run `npm install` and `npm start` in the root folder of the gateway. As a result, the gateway should print a log confirming that it has started successfully, along with the port on which it is running.

The gateway includes a built-in GUI that can be used for executing queries and testing the application. To access the GUI, navigate to the `http://localhost:9200/graphiql` endpoint and click Explore graph button. Please note that this feature is only compatible with Google Chrome and Mozilla Firefox browsers.

5.2.3.3 OpenAPIToGraphQL

The OpenAPIToGraphQL project can also be launched as a Docker container by running `docker compose up`. The next step is to install the IBM CLI tool, which is responsible for converting OpenAPI to GraphQL. To do this, ensure that Node.js is installed on your system and then run `npm i -g openapi-to-graphql-cli`. Next step, is to execute `openapi-to-graphql http://localhost:9040/v3/api-docs -p 9999` which should start graphql wrapper on port 9999. Now, you should be able to query graphql wrapper with any client tool like Postman. For example, you may try query as showed in the Listing 5.37.

5.2.4 Predefined ports

- **Teacher service server** - 8080
- **Teacher service database** - 5432
- **Subject service server** - 8081
- **Literature service server** - 8082
- **Subject service database** - 5632
- **Literature service database** - 5732
- **Lecture service server** - 8083

```
{
  viewerBasicAuth(username: "string", password: "string") {
    teacher(username: "yakunole") {
      findCar {
        name
      }
      firstName
      lastName
      username
    }
  }
}
```

Code listing 5.37 Example of query

- **Lecture service database** - 5832
- **Seminar service server** - 8084
- **Seminar service database** - 5932
- **Gateway** - 9200
- **OpenAPIToGraphQL** - 9040

5.2.5 Testing

Note that all queries, mutations and entity fetchers are covered by unit tests. You can find associated tests in the `/src/main/test` of each service. Testing is done as described in the Testing section.

5.2.6 Teacher Service

Teacher service is responsible for all operations related to `Teacher` entity. It contains only teacher information and doesn't know anything about the relation with other entities. Instead, `Teacher` is extended by other services. This is done to showcase the separation of concerns which is possible because of GraphQL Federation. You can see the schema of the service in the Listing A.1.

5.2.6.1 Dynamic schema

Although the schema only defines the `getTeacherByUsername` query, there is an additional `getTeachers` query that is dynamically generated. The code for this query can be found in the `DynamicTypeDefinitions` and `TeacherDataFetcher` classes. For more information on this topic, refer to the Dynamic schemas section.

5.2.6.2 Errors as data

This service also demonstrates the "errors as data" concept. It does so by specifying possible user errors for the `getTeacherByUsername` query within the schema, fully utilizing the declarative nature of GraphQL. With this approach, it is clear from the schema that the `getTeacherByUsername`

query has two possible outcomes: success and teacher not found. For more information on this topic, refer to the Errors section.

5.2.7 Subject service

The Subject Service manages all operations related to the `Subject` entity. Additionally it populates `Teacher` entity with the list of related subjects. The schema for this service can be found in Listing A.2.

5.2.7.1 Type extension

This service demonstrates the use of type extension in the DGS framework. To achieve this, both the entity fetcher and nested data fetcher are defined within the `TeacherDataFetcher` class as described in Extending type section.

5.2.8 Literature service

The Literature Service manages all operations related to the `Literature` entity and extends the `Subject` entity with a list of related literature. The schema for this service can be found in Listing A.3.

5.2.8.1 Data Loader

This service demonstrates the data loader concept. The `MappedBatchLoader` implementation can be found in the `LiteratureDataLoader` class, while the thread executor configuration is located in the `DataLoaderConfig` class. For more information on this topic, refer to the Data Loader section.

5.2.8.2 Subscription

Additionally, a real-time subscription called `literatureAdded` located in the `LiteratureDataFetcher` is also implemented. You can find the associated test in the `LiteratureDataFetcherTest`. See Subscription section for more information.

5.2.8.3 Authentication and Authorization

The service also showcases an example of authentication and authorization in the DGS framework. This is achieved by registering an admin user in the `SecurityConfig` class and restricting access to the `getLiteratureByCode` query for user with admin role only. Note that this has a significant impact on the gateway and for more information on this topic, see the Authentication and Authorization section.

5.2.8.4 Mutation

The implementation of mutations and the usage of input types are also demonstrated. The `addLiterature` mutation example can be found in the `LiteratureDataFetcher` class.

5.2.9 Lecture service

The Lecture Service is responsible for managing the `Lecture` entity and extending the `Subject` entity with a list of related lectures. See the service schema in the Listing A.4.

5.2.9.1 Instrumentation

This service demonstrates the basic usage of the Instrumentation concept by providing the `LoggingTracingInstrumentation` class, which logs the name of the data fetcher when its execution is complete. For more information on this topic, refer to the Instrumentation section.

5.2.9.2 Tracing

The service showcases tracing functionality by implementing `TracingInstrumentation` in the `TracingConfig` class. Consequently, when executing queries for this service, traces are present in the response. For more information on this topic, see the Tracing section.

5.2.9.3 Metrics

Metrics are also demonstrated in this service and can be accessed at the `/actuators` endpoint. A list of default actuator metrics is available from the start, and GraphQL metrics will appear after executing queries. For more information, refer to the Metrics section.

5.2.9.4 Custom scalar

Occasionally, it may be necessary to create a custom scalar, such as for handling date types. In this example, a custom Long scalar is implemented. You can refer to the `LongScalar` class for an example of implementation and to Custom scalar section for additional information.

5.2.10 Seminar service

Seminar service is responsible for the `Seminar` entity. It also extends `Subject` entity with the list of related seminars. See the schema in the Listing A.5.

5.2.11 Gateway

Gateway is the default ”apollo server” used with the ”apollo gateway” extension. Additionally, it is customized to allow local composition of schemas and passing of authorization header to the underlying services. You can find more information in the Gateway section.

5.2.12 OpenAPIToGraphQL

OpenAPIToGraphQL is implemented as regular web server with three swagger documented REST endpoints. You can see the swagger documentation by going to `/swagger-ui.html` endpoint of running server. It is created to showcase the OpenAPI to GraphQL section, so refer there for more details.

Chapter 6
Conclusion

The main goal of this thesis was to introduce GraphQL Federation, explain its fundamental concepts, benefits, potential problems, and provide a functional prototype. The first three chapters were dedicated to providing a comprehensive theoretical foundation, while the final chapter explored the implementation of Netflix DGS and a prototype. Throughout this study, we have thoroughly examined the key benefits and limitations of the GraphQL Federation technology, enabling us to now draw the conclusion.

Gateway Federation is particularly advantageous for organizations that are required to serve significant number of distinct clients requiring different data, thus necessitating the use of GraphQL to manage this demand. It also proves beneficial when operating in a distributed architecture where it significantly simplifies communication between teams and reduces the need for requesting changes from other teams. Therefore, Gateway Federation stands out as a great fit for complex, distributed systems, requiring a more efficient working environment.

However, smaller companies may encounter a number of challenges trying to adopt GraphQL Federation. For instance, a well-implemented GraphQL Federation architecture necessitates Schema Registry and Uplink which are not available as open-source solutions, but rather are built in-house by large tech companies.

Additionally, GraphQL Federation is a relatively new technology and the community knowledge and support is still growing. This means that best practices are not yet established, and there may be fewer experts or resources to turn to when encountering problems and issues.

To conclude, I would say that GraphQL Federation is a great choice in case you are prepared to invest the necessary resources and effort to overcome potential challenges. However, if your priority is a workable solution that is easy to implement, I would say that a combination of GraphQL with other technologies is beneficial. For example, I would suggest to consider a single GraphQL service serving as an abstraction layer communicating with underlying services to resolve incoming requests.

Appendix A

Appendix

```
type Query {

    """
    Query to retrieve teacher by username
    """
    getTeacherByUsername(username: String!): GetTeacherByUsernamePayload
}

union GetTeacherByUsernamePayload = GetTeacherByUsernameSuccess |
↪  TeacherNotFound

"""
Type representing teacher of the university
"""
type Teacher @key(fields: "username") {
    username: ID
    firstName: String
    lastName: String
    titleBefore: String
    titleAfter: String
    employmentStatus: String
    position: String
    department: String
}

type GetTeacherByUsernameSuccess {
    teacher: Teacher
}

interface UserError {
    message: String!
}

type TeacherNotFound implements UserError {
    message: String!
}
```

Code listing A.1 Teacher service schema

```
type Query {
    """
    Query to retrieve the list of all subject
    """
    getSubjects: [Subject]

    """
    Query to retrieve subject by code
    """
    getSubjectByCode(code: String!): Subject
}

"""
Type representing subject in the university
"""
type Subject @key(fields: "code") {
    code: ID
    titleCz: String
    titleEn: String
    garant: String
    kat: String
    psp: String
    psem: String
    ects: Int
    rozshah: String
    end: String
    role: String
    poznamka: String
    osnovacz: String
    anotacecz: String
    osnovaen: String
    anotaceen: String
}

"""
Extension of teacher type with the list of related subjects
"""
type Teacher @key(fields: "username") @extends {
    username: ID
    subjects: [Subject]
}
```

Code listing A.2 Subjects service schema

```
type Query {
    getLiteratures: [Literature]

    getLiteraturesByCode(code: String!): [Literature]
}

type Mutation {
    addLiterature(literature: AddLiteratureInput!): Literature
}

type Subscription {
    literatureAdded: Literature
}

type Literature {
    isbn: ID
    code: String
    title: String
    author: String
    publisher: String
    year: String
}

type Subject @key(fields: "code") @extends {
    code: ID
    literature: [Literature]
}

input AddLiteratureInput {
    isbn: String
    code: String
    title: String
    author: String
    publisher: String
    year: String
}
```

Code listing A.3 Literature service schema

```
type Query {
    getLectures: [Lecture]

    getLecturesByCode(code: String!): [Lecture]
}

type Lecture {
    code: String
    parallel: Int
    studentCount: Int
}

type Subject @key(fields: "code") @extends {
    code: ID
    lectures: [Lecture]
}
```

Code listing A.4 Lecture service schema

```
type Query {
    getLiteratures: [Literature]

    getLiteraturesByCode(code: String!): [Literature]
}

type Mutation {
    addLiterature(literature: AddLiteratureInput!): Literature
}

type Subscription {
    literatureAdded: Literature
}

type Literature {
    isbn: ID
    code: String
    title: String
    author: String
    publisher: String
    year: String
}

type Subject @key(fields: "code") @extends {
    code: ID
    literature: [Literature]
}

input AddLiteratureInput {
    isbn: String
    code: String
    title: String
    author: String
    publisher: String
    year: String
}
```

Code listing A.5 Seminar service schema

Bibliography

1. LANE, Kin. *The API-first transformation*. Postman, Inc, 2022. ISBN 9798986951805.
2. SHRADDHA. *5 years of graphql: Lee Byron and others look back at their graphql journey* [online]. Hasura GraphQL Engine Blog, 2021. Available also from: `https://hasura.io/blog/graphql-at-five-years`.
3. OZKAYA, Mehmet. *API gateway pattern* [online]. Design Microservices Architecture with Patterns and Principles, 2023. Available also from: `https://medium.com/design-microservices-architecture-with-patterns/api-gateway-pattern-8ed0ddfce9df`.
4. CHIAZZO, Ignacio. *Introspection in graphql* [online]. Medium, 2021. Available also from: `https://ignaciochiazzo.medium.com/introspection-in-graphql-a5a5bd744a66`.
5. *What is GraphQL* [online]. [N.d.]. Available also from: `https://graphql.org/learn/`.
6. *What is GraphQL Schema* [online]. [N.d.]. Available also from: `https://graphql.org/learn/schema`.
7. GIROUX, Marc-Andre. *Production ready GraphQL* [online]. 2020.
8. *Apollo Federation Introduction* [online]. [N.d.]. Available also from: `https://www.apollographql.com/docs/federation/`.
9. BLOG, Netflix Technology. *How netflix scales its API with Graphql Federation (part 1)* [online]. Netflix TechBlog, 2020. Available also from: `https://netflixtechblog.com/how-netflix-scales-its-api-with-graphql-federation-part-1-ae3557c187e2`.
10. *Schema registry in GraphQL Federation* [online]. [N.d.]. Available also from: `https://www.apollographql.com/tutorials/lift-off-part5/02-what-is-the-schema-registry`.
11. *Apollo managed federation overview* [online]. 2022. Available also from: `https://www.apollographql.com/docs/federation/managed-federation/overview`.

Enclosed media

- `Readme.txt` ..... Description of starting the application
- `src`
  - `impl` ..... Source code of a functional prototype
  - `thesis` ..... Source of thesis in the LaTeX
- `text` ..... Text of the thesis
  - `thesis.pdf` ..... Text of thesis in PDF format