VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES

Ngoc Tran

GRAPHQL

From Basic Concepts to a Federated Supergraph

Technology
2024

VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES
Information Technology

ABSTRACT

Author	Ngoc Tran
Title	GraphQL From Basic Concepts to a Federated Supergraph
Year	2024
Language	English
Pages	42
Name of Supervisor	Kenneth Norrgård

GraphQL has emerged as a powerful alternative to traditional RESTful APIs, offering a more flexible and efficient approach to data fetching and manipulation. This thesis delves into the foundational concepts of GraphQL, elucidating its core principles, syntax, and operations. Through a comprehensive examination, it elucidates how GraphQL enables clients to query precisely the data they need, fostering a more streamlined and tailored interaction between client and server.

Furthermore, the thesis investigates the evolution of GraphQL into a federated architecture, wherein multiple GraphQL services collaborate to form a unified graph, known as a supergraph. This federated approach allows organizations to scale their GraphQL implementations efficiently, enabling the composition of complex schemas from disparate sources. By implementing a sample project which demonstrates GraphQL Federation technologies in both Node.js and PHP environments, the thesis analyzes the principles and mechanisms underpinning GraphQL federation, and provides insights into the design, configuration and implementation of federated supergraphs.

Through a combination of theoretical analysis and practical examples, this thesis endeavors to equip readers with a comprehensive understanding of GraphQL and its federated extensions. By exploring the transition from basic GraphQL concepts to the construction of federated supergraphs, it seeks to contribute to the growing body of knowledge surrounding GraphQL and its applications in modern software development paradigms.

The thesis is beneficial for software developers who are looking for an advanced solution to replace RESTful API for scalable systems. System architects can also examine the GraphQL Federation in this thesis to consider this architecture for their system communication.

Keywords GraphQL, supergraph, federation, HTML, RESTful API

CONTENTS

LIST OF FIGURES AND TABLES

LIST OF ABBREVIATIONS

API	Application programming interface
CDMI	Cloud Data Management Interface
CSS	Cascading Style Sheets
DOM	Document Object Model
HTML	Hypertext Markup Language
MVC	Model-View-Controller
ID	Identity Document
IDEs	Integrated Development Environments
I/O	Input/Output
JSX	JavaScript XML
JWT	JSON Web Token
NPM	Node Package Manager
PHP	Hypertext Preprocessor
REST/ RESTful	Representation State Transfer
SOA	Service-oriented architecture
UIs	User interfaces
XML	Extensible Markup Language

1 INTRODUCTION

In the realm of web development, the evolution of headless architectures has ushered in a paradigm shift, where frontend and backend components are developed independently and interconnected via RESTful APIs. While RESTful APIs have been widely adopted for communication between frontend and server, as well as among various microservices within a system, they present inherent limitations that can hinder the performance of web and mobile applications.

One of the primary challenges associated with RESTful APIs is the issue of over-fetching, wherein a client retrieves more data than necessary for its operations. This inefficiency arises from the rigid structure of REST requests, making it difficult to retrieve only the specific fields required by the client. GraphQL is a query language for APIs that offers a solution to this problem by enabling clients to precisely request the data they need, eliminating the burden of over-fetching. GraphQL allows developers to get exactly the necessary data without any redundant parts.

Beyond optimizing data retrieval for frontend clients, GraphQL also facilitates communication between multiple microservices within a microservices architecture. However, the adoption of GraphQL introduces a new challenge: orchestrating communication between frontend clients and multiple microservices simultaneously. Sending individual queries from the frontend to disparate microservices can lead to suboptimal performance and increased complexity.

To address this challenge, the concept of federation emerges as a pivotal intermediary solution. Federation acts as a unifying layer between the clients and GraphQL servers, serving as the authoritative source of truth and the sole point of entry for requests originating from frontend clients. By federating GraphQL servers, organizations can streamline communication between frontend clients and distributed microservices, enhancing performance, scalability, and maintainability.

The aim of this thesis is the exploration of GraphQL, tracing its evolution from basic concepts to the implementation of federated supergraphs. The thesis delves into the foundational principles of GraphQL, examining how it revolutionizes data fetching and manipulation for frontend clients. Furthermore, the thesis investigates the emergence of GraphQL federation as a mechanism for orchestrating communication between frontend clients and distributed microservices. Through theoretical analysis, practical examples, and case studies, the thesis aims to provide insights into the design, implementation, and deployment of federated supergraphs, contributing to the broader discourse on GraphQL and its applications in modern web development architectures.

2 BACKGROUND

2.1 HTML, CSS and JavaScript

HTML, or Hyper Text Markup Language, is standard markup language used to create structure and content of Web pages on the internet. It consists of a system of tags enclosed in angle brackets, which define various elements like headings, paragraphs, links, images, and more. HTML documents are interpreted by web browsers to render visually appealing web pages for users to view and interact with. (W3Schools 2022.)

CSS standing for Cascading style Sheets describes how HTML elements are to be displayed on screen, paper, or in other media. (W3schools 2019.) By control the layout of multiple web pages all at once it can save a lot of work. External stylesheets are stored in CSS files.

JavaScript is a programming language which adds interactivity to a web page by dynamically changing the HTML content or CSS styles. Traditionally, JavaScript is understood as client-side script, which is executed by a web browser to interact with the opening web page. This limitation is extended nowadays, as JavaScript can run on a background process to support features like push notification or background sync. (Archibald 2015.)

Using Node.js as a JavaScript runtime environment, JavaScript source code can be executed as a standalone application and interact with the operating system, allowing JavaScript to be a server-side programming language like PHP, Java or Python. Node.js uses event-driven architecture with non-blocking operations that makes it becomes a high-performance server-side language. (Patel 2018.)

There are hundreds of thousand Node.js libraries written by the developer community. NPM is a dependency manager tool for Node.js applications. It arranges the libraries in place and manages version conflicts so that libraries can be integrated into Node.js applications. (npm Docs 2019.)

2.2 React

React is a popular JavaScript library for building user interfaces (UIs) in web applications. Developed by Facebook, React allows developers to create reusable UI components and efficiently manage the state of their applications. (React n.d.)

One of React's key features is its use of a virtual DOM (Document Object Model), which enables efficient updates to the user interface by only re-rendering components that have changed. This approach results in better performance and smoother user experiences, particularly in complex and dynamic web applications.

React employs a component-based architecture, where UIs are broken down into smaller, self-contained components that can be easily composed together to build complex interfaces. Each component manages its own state and can communicate with other components through props (properties) and callbacks.

React also promotes the use of JSX (JavaScript XML), a syntax extension that allows developers to write HTML-like code directly within JavaScript files. JSX makes it easier to create and maintain UI components by embedding HTML-like syntax within JavaScript, facilitating the development process, and improving code readability.

2.3 Node.JS and Express.JS framework

Node.js is a server-side JavaScript runtime environment that allows developers to build scalable and high-performance web applications (NodeJS n.d.). Because of being built on Chrome’s V8 JavaScript engine and using an event-driven, non-blocking I/O model, it is efficient and lightweight for handing concurrent connections and asynchronous operations. Using asynchronous and non-blocking I/O operations allows Node.JS to handle multiple concurrent connection efficiently. This makes it well-suited for building real-time application like chat applications, online gaming platforms and streaming services.

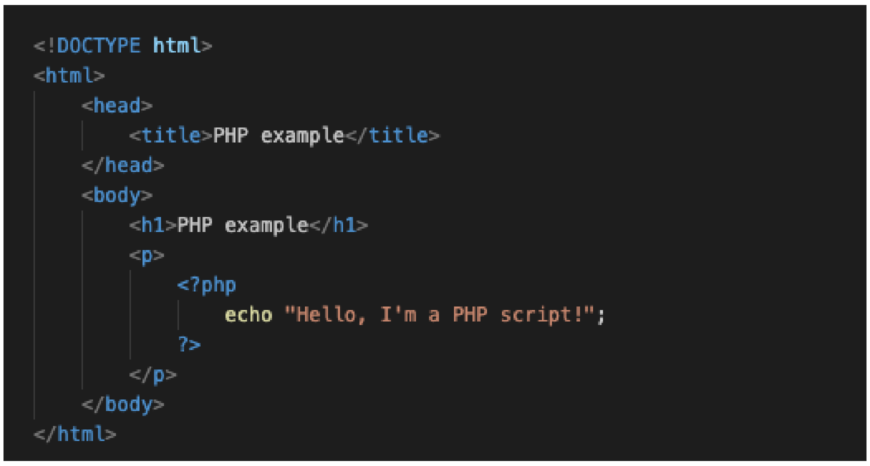
Besides, Node.js operates on a single-threaded event loop, but it can handle many concurrent connections without the need for multithreading. Instead, it employs asynchronous callbacks and event-driven programming to manage I/O operations without blocking the execution of other code. Furthermore, HTTP is a first-class citizen in Node.js, designed with streaming and low latency in mind. This makes Node.js well suited for foundation of a web library or framework.

Express.js is a minimal and flexible Node.js web application framework that provides a robust set of features for web and mobile applications. With a myriad of HTTP utility methods and middleware at your disposal, creating a robust API is quick and easy. Express provides a thin layer of fundamental web application features, without obscuring Node.js features that you know and love. Many popular frameworks are based on Express. (OpenJS Foundation 2017.)

2.4 PHP and Laravel Framework

PHP, standing for Hypertext Preprocessor is a widely used open-source general-purpose scripting language that is especially suited for web development and can be embedded into HTML. (PHP 2019.)

Instead of lots of commands to output HTML (as seen in C or Perl), PHP pages contain HTML with embedded code that does “something” (In this following example, output “Hello, I’m a PHP script!”).



```
<!DOCTYPE html>
<html>
    <head>
        <title>PHP example</title>
    </head>
    <body>
        <h1>PHP example</h1>
        <p>
            <?php
                echo "Hello, I'm a PHP script!";
            ?>
        </p>
    </body>
</html>
```

Figure 1. Example PHP Code snippet.

Enclosed within special start and end processing instructions `<?php and ?>`, the PHP code enables developers to seamlessly transition in and out of "PHP mode". This key distinction sets PHP apart from client-side JavaScript as the code is executed on the server-side, resulting in the generation of HTML that is subsequently transmitted to the client. Consequently, while the client receives the output of the executed script, they are unaware of the underlying PHP code. Moreover, developers have the option to configure their web server to process all HTML files with PHP, ensuring that users are unable to distinguish the specific operations being carried out by the developers.

Laravel is an open-source PHP web framework known for its elegant syntax, robust features, and developer-friendly environment. It follows the Model-View-Controller (MVC) architectural pattern, which separates the application logic, presentation, and data layers, making it easier to develop and maintain complex web applications. (Laravel n.d.)

Laravel offers an expressive and intuitive syntax that allows developers to write clean and concise code, making the development process more efficient and enjoyable. It comes with a modular structure and built-in support for Composer, a PHP dependency manager, allowing developers to easily add or remove components and third-party packages to extend the framework functionality. Besides, Laravel provides a migration system that allows developers to define and manage database schemas using PHP code, making it easy to version control database changes and collaborate with other developers. Additionally, Laravel's database seeding feature enables developers to populate databases with sample data for testing and development purposes.

2.5 RESTful API and Microservice Architecture

A RESTful API (Representational State Transfer Application Programming Interface) is an architectural style for designing networked application. In that, API (an application programming interface) plays function role to define the rules in communication with two software systems. Then, other application can communication with APIs being created programmatically. The rest, REST (Representation State Transfer) is a software architecture that imposes condition on how an API should work. REST was initially created as a guideline to manage communication on a complex network like the Internet. APIs that following the REST architectural style are called REST API or RESTful API. (Amazon Web Services n.d.)

A RESTful API deconstructs transactions into smaller, modular components, each addressing a specific aspect of the transaction, offering developers flexibility but also posing challenges in designing from scratch. To mitigate this, various models provided by companies like Amazon S3, Cloud Data Management Interface (CDMI), and OpenStack Swift are available, among others. These APIs operate by utilizing commands to access resources, where the state of a resource at any given time is termed a resource representation. Leveraging HTTP methodologies outlined in RFC 2616, including GET for retrieval, PUT for state alteration, POST for creation, and DELETE for removal, a RESTful API must adhere to six architectural constraints for true RESTfulness. (S 2020.) These include uniform interface usage for resource identification, clear client-server delineation, stateless operations, resource caching, a layered system architecture, and the provision of executable code when required, ensuring a robust and efficient communication protocol for distributed systems.

Microservices represent a modern architectural paradigm where complex software applications are decomposed into smaller, independently deployable services, each responsible for a distinct business function. These services are designed to be modular and autonomous, enabling development teams to work on different services concurrently and deploy updates to individual services without affecting the entire application. (Richardson 2017.)

Each microservice typically encapsulates its own data store and business logic, communicating with other services via well-defined APIs over a network. This decentralized approach to data management and communication promotes loose coupling between services, allowing them to evolve independently and scale horizontally as needed. Additionally, microservices are often implemented using lightweight protocols like HTTP or messaging queues, facilitating efficient communication between services.

While microservices offer numerous benefits, including improved agility, scalability, and resilience, they also introduce challenges such as service discovery, orchestration, and monitoring. Organizations adopting a microservices architecture must carefully consider factors such as service boundaries, data consistency, and deployment strategies to effectively harness the advantages of this approach. Despite these challenges, microservices enable organizations to build complex, scalable applications that can adapt to changing business requirements and technological advancements.

One example of a microservices architecture is found in the e-commerce industry. On an online retail platform where various functionalities, such as user authentication, product catalog management, order processing, and payment processing, are each handled by separate microservices.

3 GRAPHQL

3.1 Concept

GraphQL is query language and runtime for APIs developed by Facebook. It provides are more efficient and flexible alternative to traditional RESTful APIs by allowing clients to request only the specific data they need. With GraphQL, clients specify the structure of the response, enabling them to fetch multiple resources in a single request. This helps to reduce over-fetching (retrieving more data than necessary) and under-fetching (not getting enough data) issues commonly seen in RESTful APIs.

3.1.1 Graph in GraphQL

Graph is known as a data structure to be built based on the natural way in building mental models and relate concepts. Graphs are abstracted by adding nodes or vertices and connecting them together with edge. Among different types of graphs, the acyclic directed graph is used in GraphQL. In an acyclic directed graph, a start node and an end node are connected by a directed edge and only be traversed following that direction. The meaning of the relationship between nodes is changed and a hierarchy is introduced by adding direction to the edges.

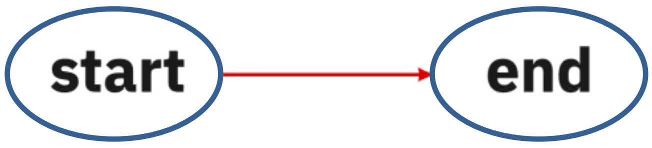


Figure 2. Node and edge in directional graph

Depending on the constraints between nodes and edges in an acyclic directed graph, the graph can be transformed into many different types of data structures. The tree structure is a directional graph that is also acyclic.

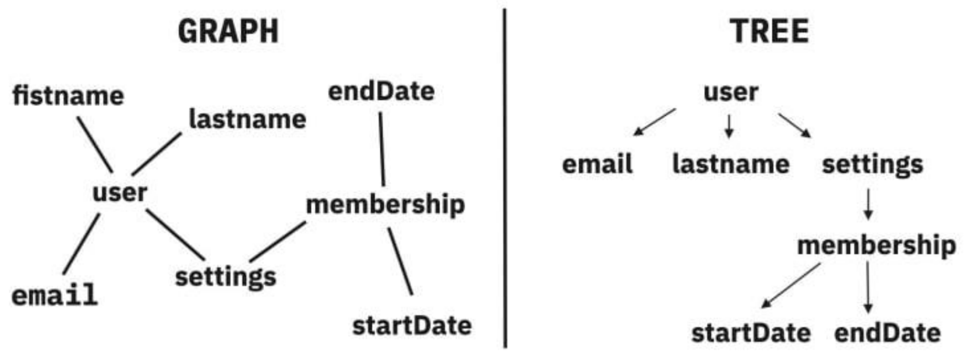
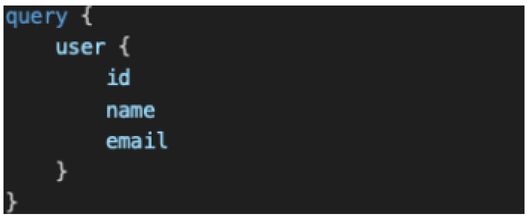


Figure 3. From graph to tree (DEV Community 2020.).

The advantage of the tree structure is its recursive nature which uses processing data by imposing the necessary constraints on it.

3.1.2 Query

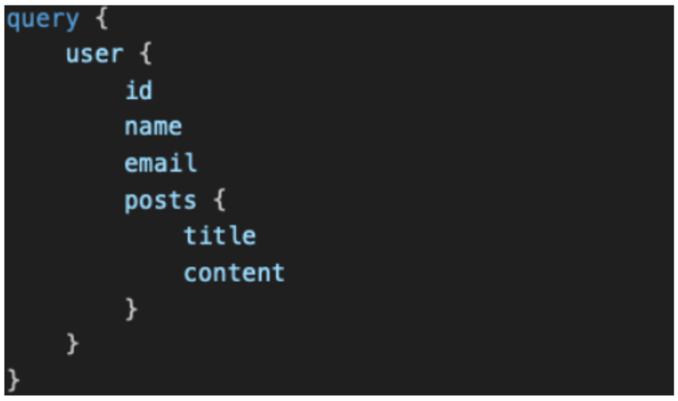
In GraphQL, a query is a fundamental concept that represents a request for specific data from GraphQL API. It serves as a structured way for clients to define precisely what information they need. Allowing them to retrieve only the required fields and nothing more. At the core of a GraphQL query are fields, which correspond to the properties or attributes of the data being requested. Clients specify the fields they want to retrieve, and the server responds with exactly those fields.



```
query {
    user {
        id
        name
        email
    }
}
```

Figure 4. Example GraphQL query.

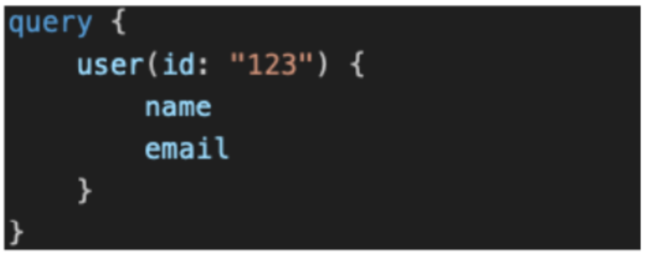
With GraphQL, nesting fields within other fields enables clients to fetch related data in single query.



```
query {
    user {
        id
        name
        email
        posts {
            title
            content
        }
    }
}
```

Figure 5. Example nested GraphQL query.

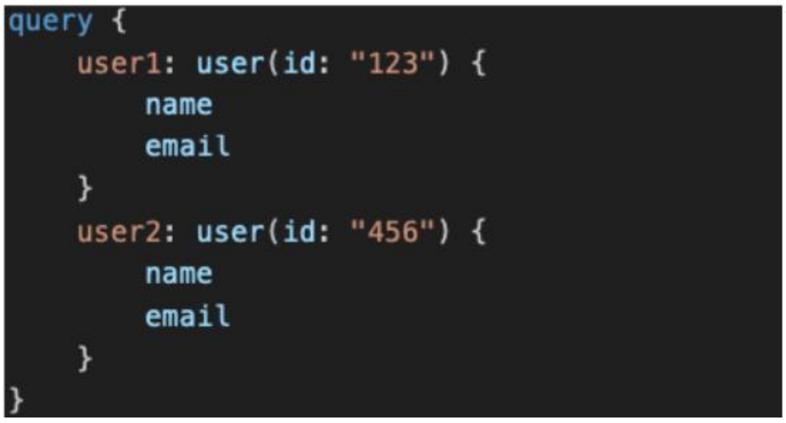
Queries can include arguments to filter, paginate or other specific the data to be retrieved. These arguments are passed to fields as paraments.



```
query {
    user(id: "123") {
        name
        email
    }
}
```

Figure 6. Example GraphQL query with parameters.

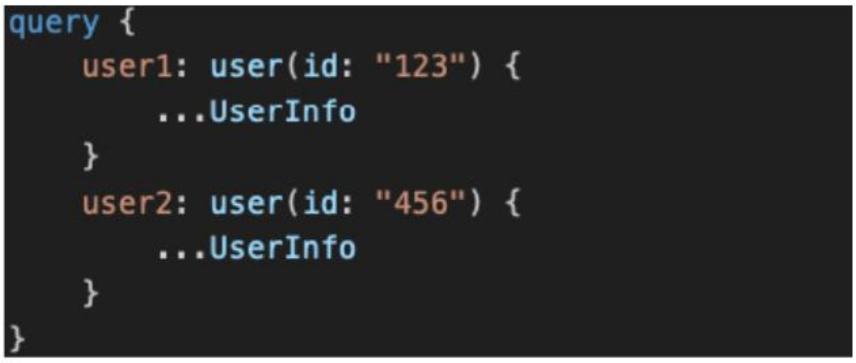
Clients can use aliases to request the same filed with different names in the response.



```
query {
    user1: user(id: "123") {
        name
        email
    }
    user2: user(id: "456") {
        name
        email
    }
}
```

Figure 7. Example GraphQL query with aliases.

Fragments are reusable units of fields that can be included in multiple queries.



```
query {
    user1: user(id: "123") {
        ...UserInfo
    }
    user2: user(id: "456") {
        ...UserInfo
    }
}
```

Figure 8. Example GraphQL query with reusable fragments.

3.1.3 Comparison between GraphQL with REST

GraphQL's flexibility stands out as a significant benefit. It allows clients to specify the precise data they require by nesting fields or fragments and the server delivers exactly that information in response. This streamlined approach minimizes both over-fetching and under-fetching of data, resulting in more efficient network usage with GraphQL. In contrast, REST APIs frequently encounter issues of over-fetching, where the server send unnecessary data, and under-fetching, which requires clients to make multiple requests to various endpoints to gather all necessary information.

REST, although simple in concept, often necessitates supplementary documentation to assist developers in understanding available endpoints and their associated data structures. Other side, GraphQL's robust typing system and introspection features contribute to a more seamless development process. Integrated Development Environments (IDEs) can leverage the GraphQL schema to offer auto-completion and documentation, assisting developers in crafting accurate queries and mutations.

Version control is a critical aspect of API management for any project. In the case of GraphQL, its single endpoint and robust typing system often allow for updates without disrupting existing clients. Clients that do not request newly added fields or types remain unaffected by these changes. In the opposite side, REST commonly employs versioning through URL paths (e.g., /v1/users) or headers to handle modifications. This approach can result in compatibility challenges, as clients must be informed of and adjust to API changes.

Efficient caching plays a vital role in enhancing API performance. In the context of GraphQL, clients can precisely define the data they require, which reduces the chances of excessive caching. However, incorporating caching into GraphQL systems requires additional considerations due to the dynamic nature of queries. On the other hand, REST, with its resource-oriented architecture, has established caching methods. Clients can cache responses based on resource identifiers, and servers can control caching behavior through cache headers.

Both GraphQL and REST offer effective scalability solutions, albeit with differing approaches. GraphQL's capability to request specific data proves advantageous in scenarios where bandwidth constraints are a concern. However, the versatility of GraphQL queries can present challenges when optimizing database queries and caching strategies. In contrast, REST's simplicity and stateless nature facilitate horizontal scaling by adding more servers. Additionally, caching implementation in REST is more straightforward due to the predictability of resource-based endpoints. (joan 2024.)

3.2 Apollo GraphQL Platform

Apollo is a platform for constructing data graphs, created by Meteor Development Group Inc. The data graph serves as a bridge between the client-side of applications and the internal services, allowing for seamless communication.

The Apollo platform aids in the creation, retrieval, and administration of a data graph. This unified data layer empowers applications to interact with data sourced from connected data repositories and external APIs. Positioned between application clients and backend services, the data graph streamlines the flow of data between them, as shown in the following figure.

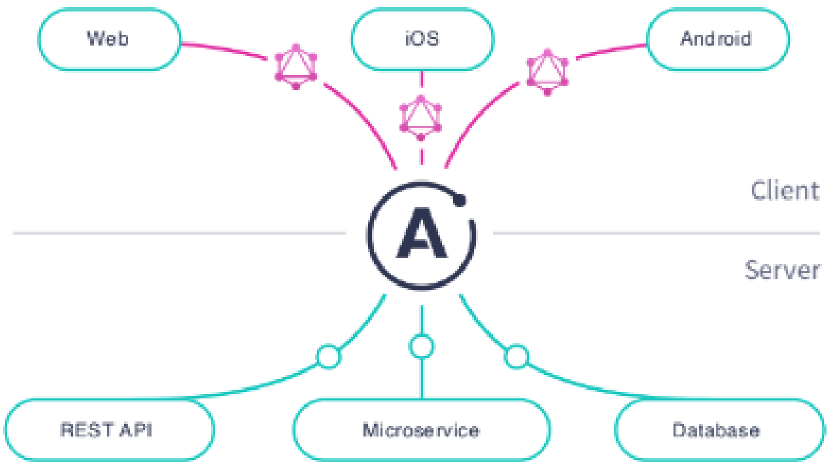


Figure 9. App data flow with Apollo and GraphQL (Apollo GraphQL Docs 2019.)

3.2.1 Apollo Client

Apollo Client is a robust state management library in JavaScript, designed for handing both local and remote data seamlessly with GraphQL. It simplifies tasks such as fetching, catching, and modifying application data while ensuring automatic UI updates. With Apollo Client, developers can structure their code in an efficient, predictable, and declarative manner that aligns with modern development practices. The core `@appllo/client` library comes with built-in integration for React, providing a solid foundation for managing data in React applications. Moreover, the boarder Apollo community maintains integration for various other popular view layers, extending its usability across different frameworks. (Apollo GraphQL Docs 2019.)

Developers can write queries and receive data without the need to manually track loading states. Apollo Client offers valuable tooling support for TypeScript, Chrome/ Firefox devtools, and VS Code, aiding in smoother development workflows. Besides, it takes full advantage of the latest React features, including hooks, to streamline development processes. Developers can seamlessly integrate Apollo Client into any JavaScript application, incorporating its features progressively as needed. With a vibrant and active community, developers can benefit from sharing knowledge, insight, and best practices within the GraphQL ecosystem, ensuring ongoing support and innovation.

The figure below shows the general Apollo Client architecture.

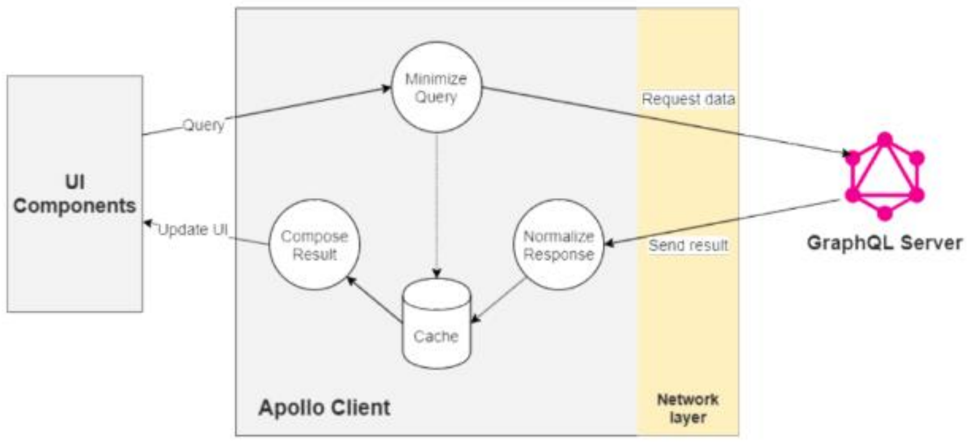


Figure 10. Apollo Client architecture (Huder 2019)

The core elements of the Apollo Client consist of the cache and network layers. The cache functionality within the Apollo Client is designed to store query results directly in the browser. This approach helps in minimizing unnecessary network requests, thereby enhancing the speed of the application. With different fetch policy settings, a query can either retrieve fresh data from the server or access it directly from the cache.

3.2.2 Apollo Server

Apollo Server is an open-source, spec-compliant GraphQL server that is compatible with any GraphQL client, including Apollo Client. It is the best way to build a production-ready, self-documenting GraphQL API that can use data from any source. Apollo Server can serve as the GraphQL server for a subgraph within a federated supergraph or an extension to any new/existing Node.js applications, which includes applications running on Express (including MERN stack apps), AWS Lambda, Azure Functions, Cloudflare, Fastify, and other platforms.

Apollo Server offers simple setup, allowing client developers to swiftly fetch data. It allows enabling the addition of features as required. Apollo Server provides compatibility with any data source, build tool, and GraphQL client and confidence in running the graph in a production environment.

3.3 GraphQL Federation

Microservices, the modern evolution of service-oriented architecture (SOA), represent a significant trend in the software development field, and GraphQL is emerging as the favored query language due to its versatility. However, managing microservices can meet challenges. For instance, to handle multiple endpoints for users, the implementation of federation is a potential solution. (Bhattacharya 2021.)

Federated architecture consolidates various services into a single API endpoint. By using GraphQL federation developers can set up a single GraphQL API, or a gateway which fetches from all other APIs. Then, each service is a subgraph now.

Fill the page. If the figure does not fit here, then moce text from under the figure here.

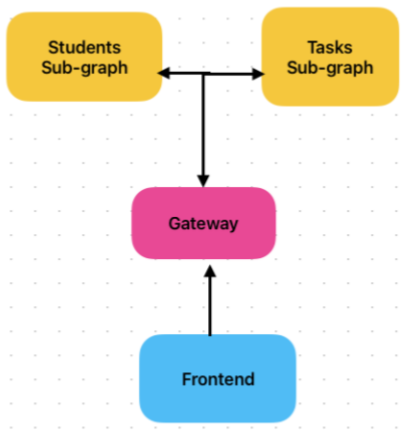


Figure 11. Federation architecture

The gateway becomes the source of truth where the frontend sends the query to retrieve needed data.

3.3.1 Apollo Federation

Apollo Federation allows for the declarative merging of multiple GraphQL APIs into a unified, federated graph. This federated graph empowers clients to communicate with multiple APIs through a solitary request. When a client initiates a request, it is directed to the single-entry point of the federated graph, known as the router. The router then efficiently coordinates and disperses the request across the connected APIs, providing a consolidated response. From the client's perspective, the process of querying the router appears identical to querying a standard GraphQL server.

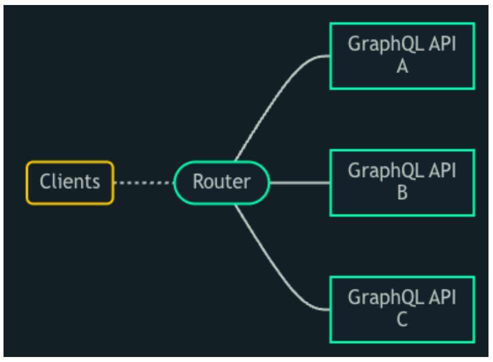


Figure 12. Apollo federation architecture (Apollo GraphQL Docs 2019.)

4 IMPLEMENTATION

4.1 Project Overview

4.1.1 Use Case

This thesis project encompasses three primary use cases aimed at demonstrating various functionalities of the system. These use cases include retrieving product information based on an ID, obtaining a list of all messages within the system, and facilitating user login to generate JWT tokens for authentication purposes.

The first use case involves retrieving product information by ID as a demonstration. When a user sends a query to the federation service, the federation service analyzes and forwards the query to "server-PHP," the service responsible for processing and returning the requested data. Subsequently, the federation service responds to the client with the relevant product information.

The second use case demonstrates retrieving all system messages, which is processed by the Node.js GraphQL server. This query requires authentication, thereby showcasing the authentication mechanism using JWT tokens with the GraphQL server.

The third log in the use case illustrates how users can log into the system using a GraphQL query to obtain a JWT token for authentication purposes. This use case showcases the login process and the subsequent reception of a JWT token to enable authenticated access to system resources.

4.1.2 Project System Architecture

The thesis implementation project is a monorepo project following the architecture below:

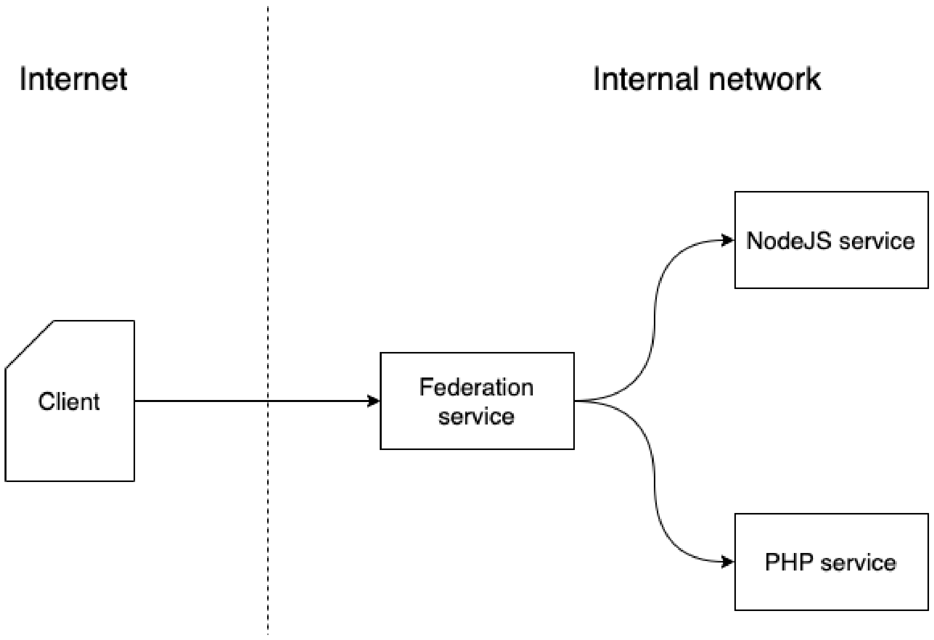
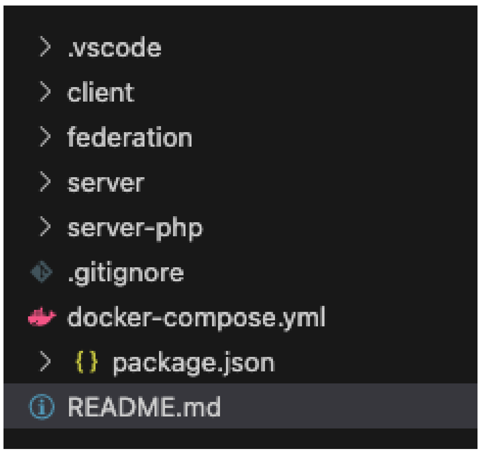


Figure 13. Project system architecture.

4.1.3 Project Source Code Structure

The project implementation for this thesis involves a monorepo structure comprising several child projects, focusing on three main parts: the server included Nodejs service and PHP service, federation service, and client.



```
> .vscode
> client
> federation
> server
> server-php
  .gitignore
  docker-compose.yml
> {} package.json
  README.md
```

Figure 14. Project source structure.

The server project is constructed with a microservice architecture, featuring two distinct services. The first service, named `server`, utilizes the Node.js platform to implement GraphQL functionalities. Additionally, there is a service developed on a PHP environment, known as `server-php` facilitating GraphQL operations within the PHP framework.

The federation service follows GraphQL principles, serving as the central synthesis point for the GraphQL schema supported by the two aforementioned services: Node.js and PHP. Through federation architecture, these services transform into subgraphs, each fulfilling specific roles within the federated system.

Lastly, the client directory functions as a user-facing application, responsible for real-time data presentation. When the client requires data, it sends requests directly to the federation. The federation service then evaluates the query and forwards it to the respective server responsible for resolution, ensuring seamless data retrieval for the client interface.

Within the scope of the thesis, this project can be run locally using Docker. There is `docker-compose.yml` file configuration that helps to start all these child projects in the local environment.

4.2 Development Process

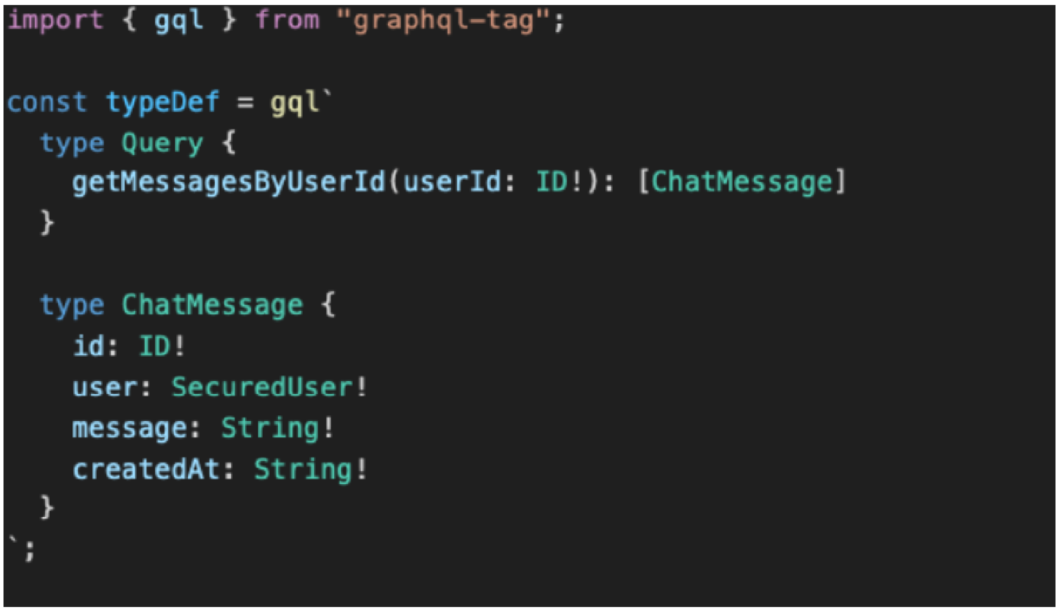
The overall process includes developing backend services with additional support for GraphQL query. After that, the federation service is implemented with configurations to connect to the backend services GraphQL endpoints via internal network. Finally, the client application can send GraphQL queries to the federation service without communicating to the backend services.

4.2.1 NodeJS Backend Microservice Components

For developing the backend service with Node.js, the `@apollo/server` library is utilized, which means the required npm dependencies are `@apollo/server` and `graphql` can be installed by the following command:

```
npm install @apollo/server graphql
```

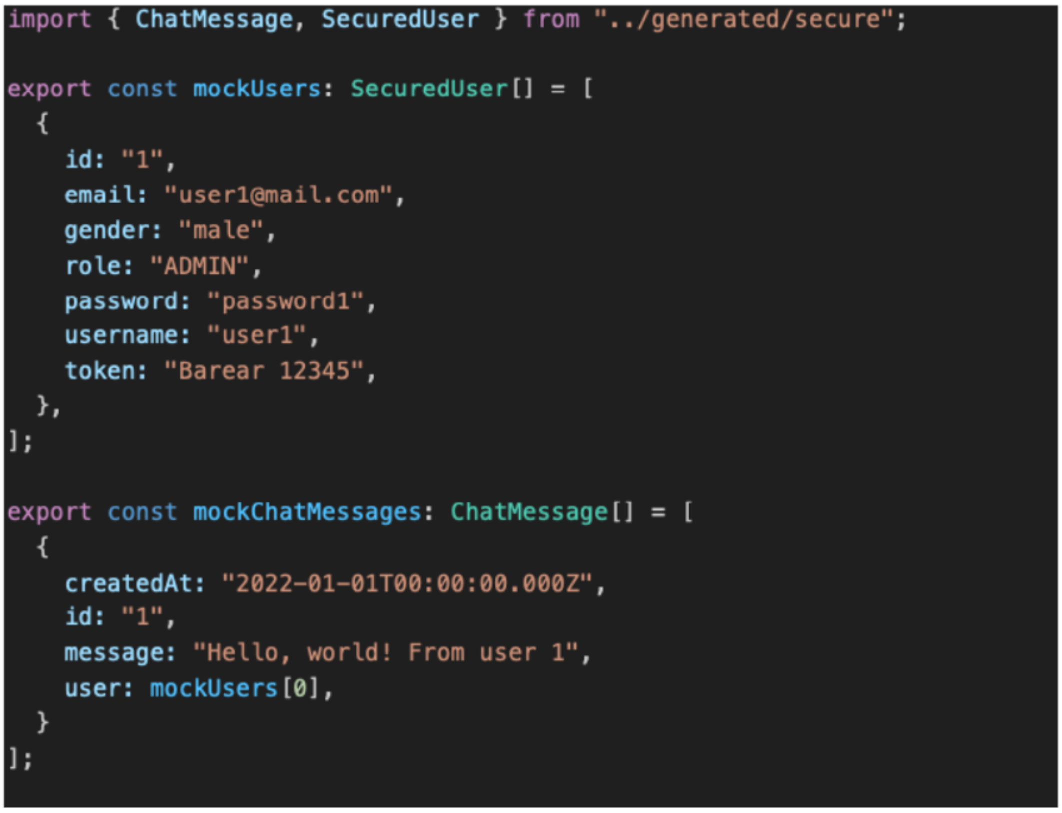
In a GraphQL server, the first step is defining the GraphQL schema. Every GraphQL server, including Apollo Server, relies on a schema to outline the structure of data that clients can query. This schema comprises type definitions, often referred to as `typeDefs` which collectively define the framework for executing queries against the data of the project. In this example, a server is created for querying a collection of messages by user.



```
import { gql } from "graphql-tag";

const typeDef = gql`
  type Query {
    getMessagesByUserId(userId: ID!): [ChatMessage]
  }

  type ChatMessage {
    id: ID!
    user: SecuredUser!
    message: String!
    createdAt: String!
  }
`;
```

Figure 15. GraphQL query definition for NodeJS server.

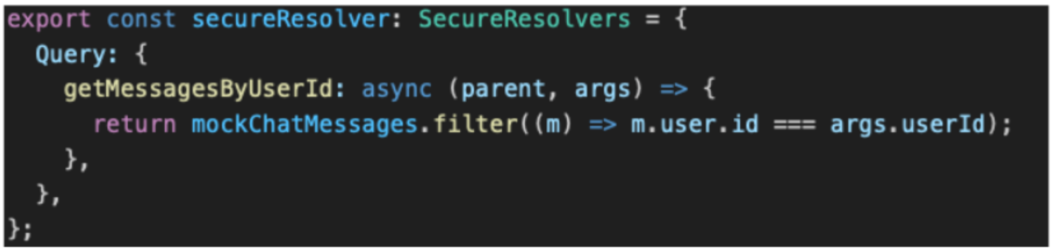
Once the structure of the project's data is defined, the next step is to define the data itself. Apollo Server has the capability to fetch data from various sources that developers connect to, such as databases, REST APIs, static object storage services, or even other GraphQL servers. In the scope of this thesis project, the illustrative data is defined as below, although in a real-world application, this data would typically be stored in a database.



```
import { ChatMessage, SecuredUser } from "../generated/secure";

export const mockUsers: SecuredUser[] = [
  {
    id: "1",
    email: "user1@mail.com",
    gender: "male",
    role: "ADMIN",
    password: "password1",
    username: "user1",
    token: "Barear 12345",
  },
];

export const mockChatMessages: ChatMessage[] = [
  {
    createdAt: "2022-01-01T00:00:00.000Z",
    id: "1",
    message: "Hello, world! From user 1",
    user: mockUsers[0],
  }
];
```

Figure 16. Mock dataset for the NodeJS server.

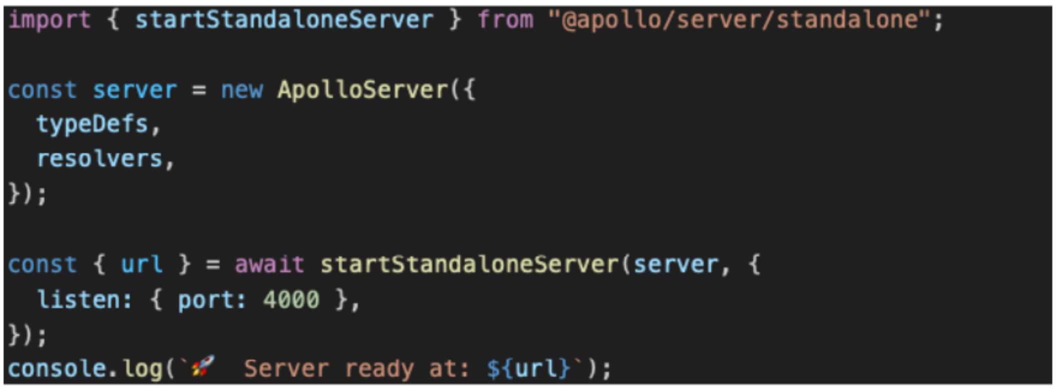
Having established our dataset, the Apollo Server needs to be instructed to utilize this data during query execution. This is achieved by creating resolvers, which define how the server retrieves the associated data for a specific type. Since mocked chat messages collection is hardcoded for demonstration purposes, the corresponding resolver is straightforward.



```
export const secureResolver: SecureResolvers = {
  Query: {
    getMessagesByUserId: async (parent, args) => {
      return mockChatMessages.filter((m) => m.user.id === args.userId);
    },
  },
};
```

Figure 17. GraphQL resolver definition for NodeJS server.

To initialize Apollo Server, the resolver and query definitions must be provided during the ApolloServer constructor invocation. Upon passing an ApolloServer instance to the startStandaloneServer function, the process involves the following steps: creation of an Express app, installation of the ApolloServer instance as middleware and preparation of the app to handle incoming requests.



```
import { startStandaloneServer } from "@apollo/server/standalone";

const server = new ApolloServer({
  typeDefs,
  resolvers,
});

const { url } = await startStandaloneServer(server, {
  listen: { port: 4000 },
});
console.log(`🚀  Server ready at: ${url}`);
```

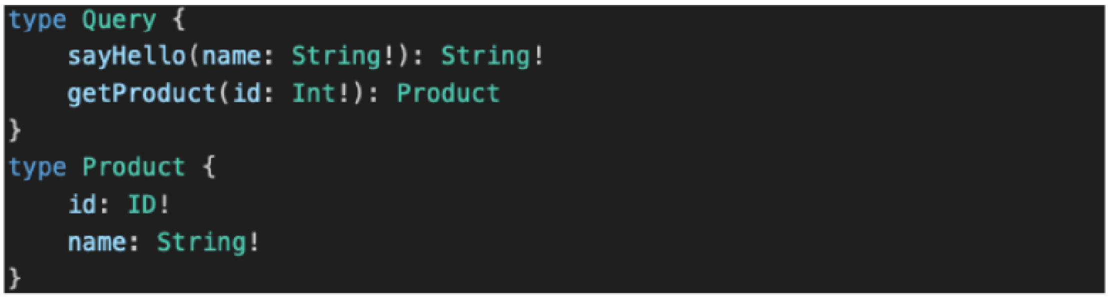
Figure 18. Apollo Server definition.

4.2.2 PHP Backend Microservice Components

The PHP backend microservice was developed using the Laravel framework with LightHouse PHP to facilitate GraphQL functionality. LightHouse can be easily set up via Composer using the following command:

```
composer require nuwave/lighthouse
```

A schema defines the capabilities of a GraphQL server. The syntax for defining a schema in PHP with LightHouse server is similar to the Node.js with Apollo Server. In this project, the schema is defined in `schema.graphql` file with the code below.



```
type Query {
    sayHello(name: String!): String!
    getProduct(id: Int!): Product
}
type Product {
    id: ID!
    name: String!
}
```

Figure 19. GraphQL query definition for PHP server.

The following commands is to utilize the included LightHouse configuration:

```
mkdir --parents config
```

```
cp vendor/nuwave/lighthouse/src/lighthouse.php config/
```

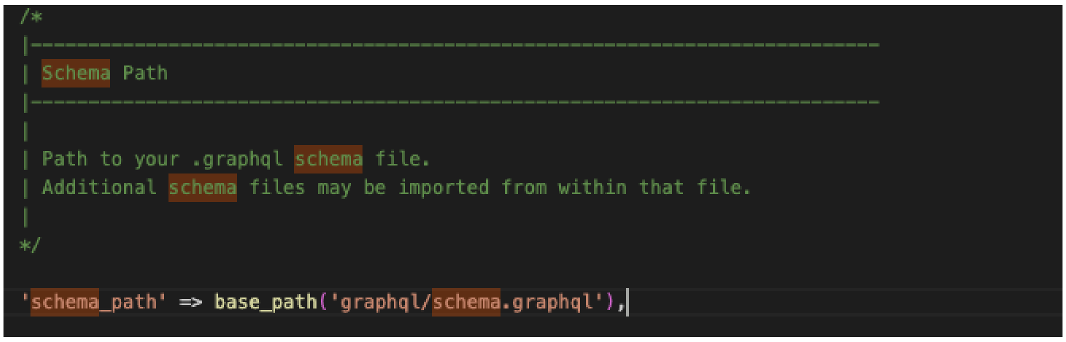
Next, the configuration file is registered in the `bootstrap/app.php` file:

```
$app->configure('lighthouse');
```

The service provider is registered in `bootstrap/app.php` file:

```
$app-
>register(\Nuwave\Lighthouse\LighthouseServiceProvider:
:class);
```

After registering the configuration file, the schema path which indicate the location of the defined schemas is configured in the `lighthouse.php` file as shown below:



```
/*
|--------------------------------------------------------------------------
| Schema Path
|--------------------------------------------------------------------------
|
| Path to your .graphql schema file.
| Additional schema files may be imported from within that file.
|
*/

'schema_path' => base_path('graphql/schema.graphql'),
```

Figure 20. Schema path configuration.

Similar to the Node.js Apollo service, the PHP server needs query resolvers. These resolvers are implemented as classes within the `server-php/app/GraphQL` directory. Each resolver class corresponds to a query declared in the `schema.graphql` file.

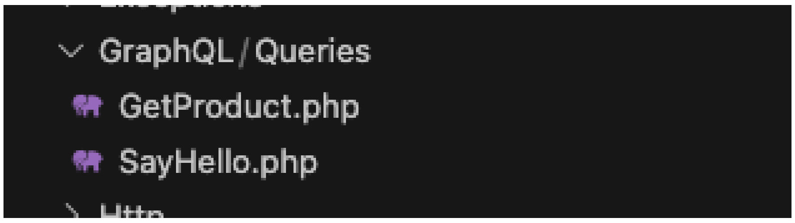
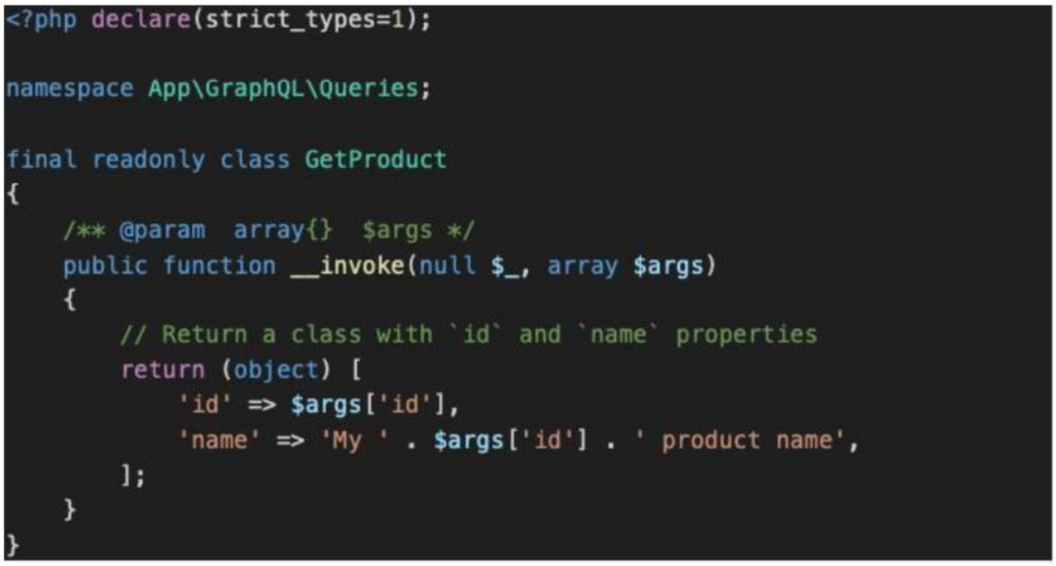


Figure 21. Resolver classes of the project.

The implementation of a resolver class might look like this:



```php
<?php declare(strict_types=1);

namespace App\GraphQL\Queries;

final readonly class GetProduct
{
    /** @param  array{}  $args */
    public function __invoke(null $_, array $args)
    {
        // Return a class with `id` and `name` properties
        return (object) [
            'id' => $args['id'],
            'name' => 'My ' . $args['id'] . ' product name',
        ];
    }
}
```

Figure 22. GraphQL resolver implementation for PHP server.

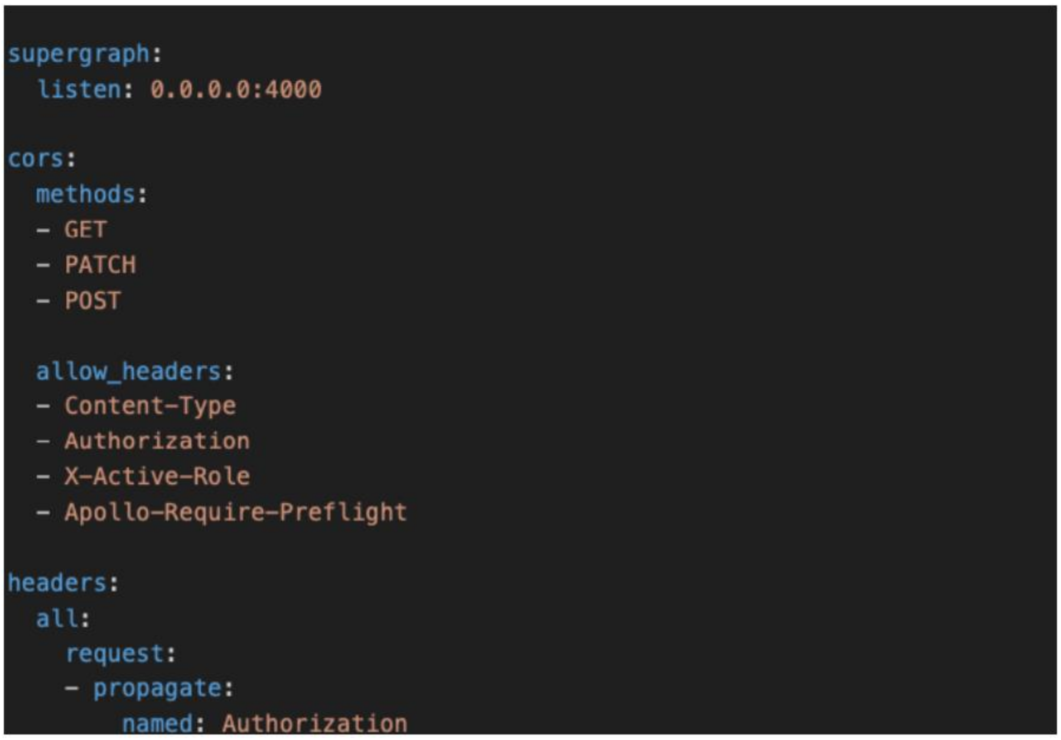
The `__invoke` function within the resolver class is responsible for returning the desired data. In this project, the resolvers work with hardcoded data for demonstration purposes.

Furthermore, all GraphQL components such as mutations, types, and directives must be declared within the namespaces of the Laravel application in the lighthouse.php file. This ensures that all GraphQL components are properly registered and accessible within the GraphQL schema..

4.2.3 GraphQL Federation Service

The GraphQL federation service in this thesis project serves as a central synthesis point for GraphQL schemas, harmonizing the Node.js and PHP services into specialized subgraphs within a federated system. Through this federation architecture, our service offers a unified approach to GraphQL, facilitating efficient data aggregation and streamlined communication between services. The federation service can be implemented by the following steps.

The first step is to configure the router location, including the supported methods, the allowed headers for the HTTP requests, and the endpoint to access this federation supergraph.

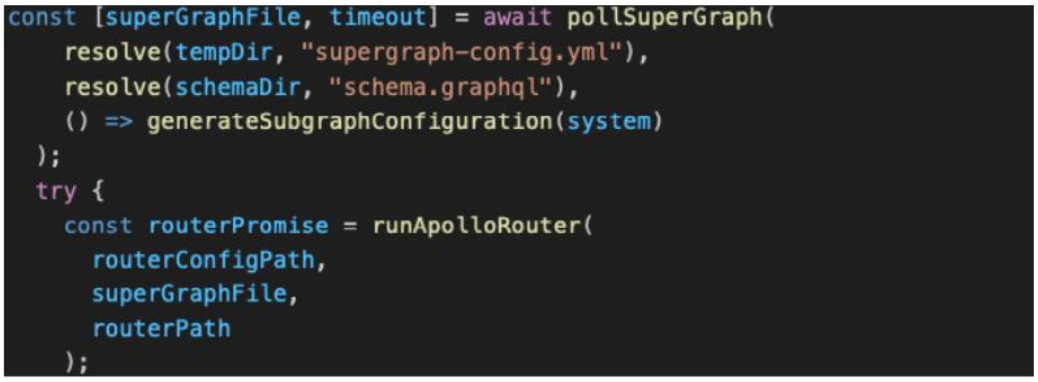


```
supergraph:
  listen: 0.0.0.0:4000

cors:
  methods:
  - GET
  - PATCH
  - POST

  allow_headers:
  - Content-Type
  - Authorization
  - X-Active-Role
  - Apollo-Require-Preflight

headers:
  all:
    request:
    - propagate:
        named: Authorization
```

Figure 23. GraphQL Federation configuration.

Because the federation service does not define its own GraphQL schemas, it retrieves GraphQL schemas from the backend services periodically and stores in a temporary location.

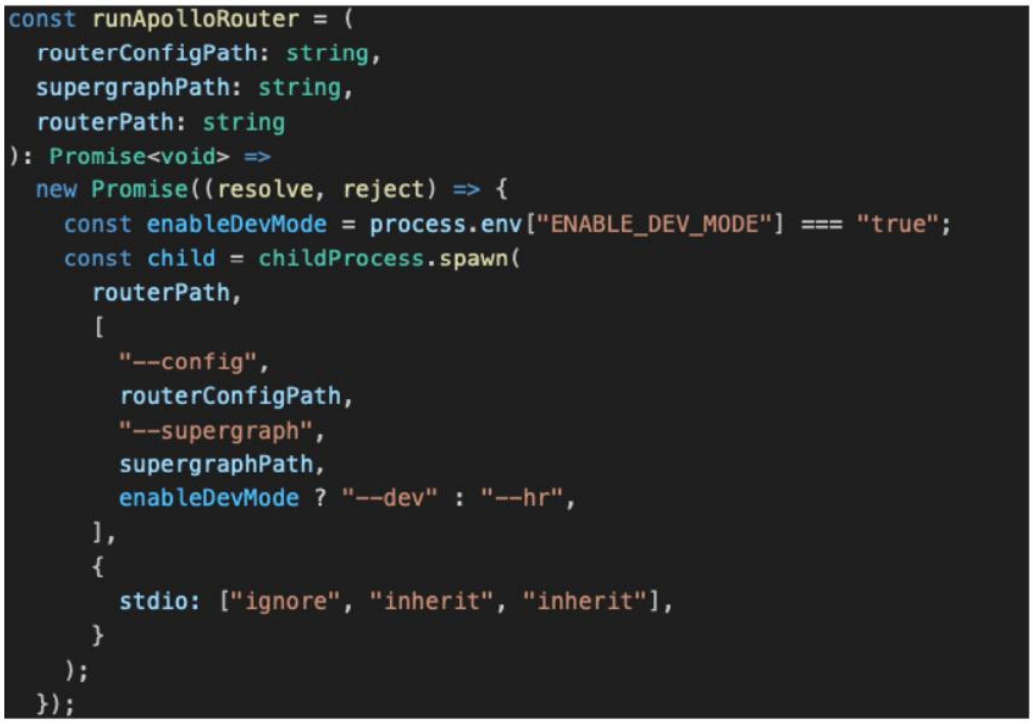
The script below is to build the retrieved schemas. The temporary data of the federation service including schema and its configuration are stored in `schemaDir` and `tempDir` locations respectively.



```
const [superGraphFile, timeout] = await pollSuperGraph(
    resolve(tempDir, "supergraph-config.yml"),
    resolve(schemaDir, "schema.graphql"),
    () => generateSubgraphConfiguration(system)
  );
  try {
    const routerPromise = runApolloRouter(
      routerConfigPath,
      superGraphFile,
      routerPath
    );
```

Figure 24. Define locations for storing temporary data.

The federation service is a supergraph, it uses Apollo GraphOS platform for building, managing, and scaling the graphs via the Rover CLI tool. Rover commands can be executed with TypeScript code as below.



```
const runApolloRouter = (
  routerConfigPath: string,
  supergraphPath: string,
  routerPath: string
): Promise<void> =>
  new Promise((resolve, reject) => {
    const enableDevMode = process.env["ENABLE_DEV_MODE"] === "true";
    const child = childProcess.spawn(
      routerPath,
      [
        "--config",
        routerConfigPath,
        "--supergraph",
        supergraphPath,
        enableDevMode ? "--dev" : "--hr",
      ],
      {
        stdio: ["ignore", "inherit", "inherit"],
      }
    );
  });
```

Figure 25. Executing Rover CLI commands.

4.2.4 React Frontend

The client is a React application that was setup Apollo Client dependencies to send request query to federation service. The following is the way client can send request query.

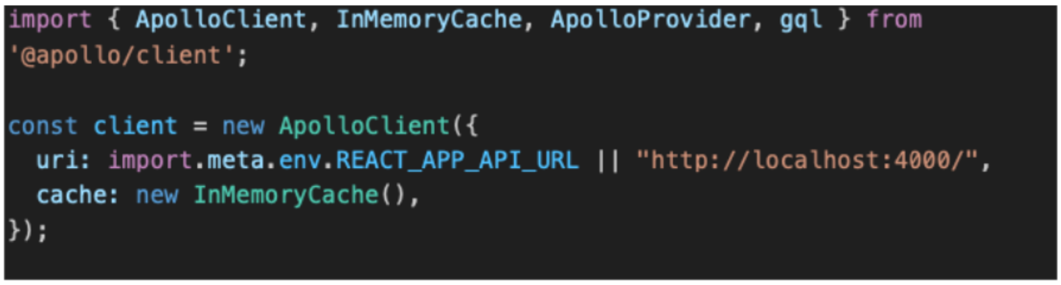
In order to incorporate Apollo Client into applications, two essential top-level dependencies are required:

- `@apollo/client`: This comprehensive package acts as the foundation for Apollo Client, encompassing vital functionalities such as the in-memory cache, local state management, error handling, and a React-based view layer.
- `graphql`: This package provides the necessary logic for parsing GraphQL queries within the Apollo Client framework.

To install these dependencies, the following command is executed:

```
npm install @apollo/client graphql
```

The ApolloClient initialization is done by passing its constructor a configuration object with the URI and cache fields:



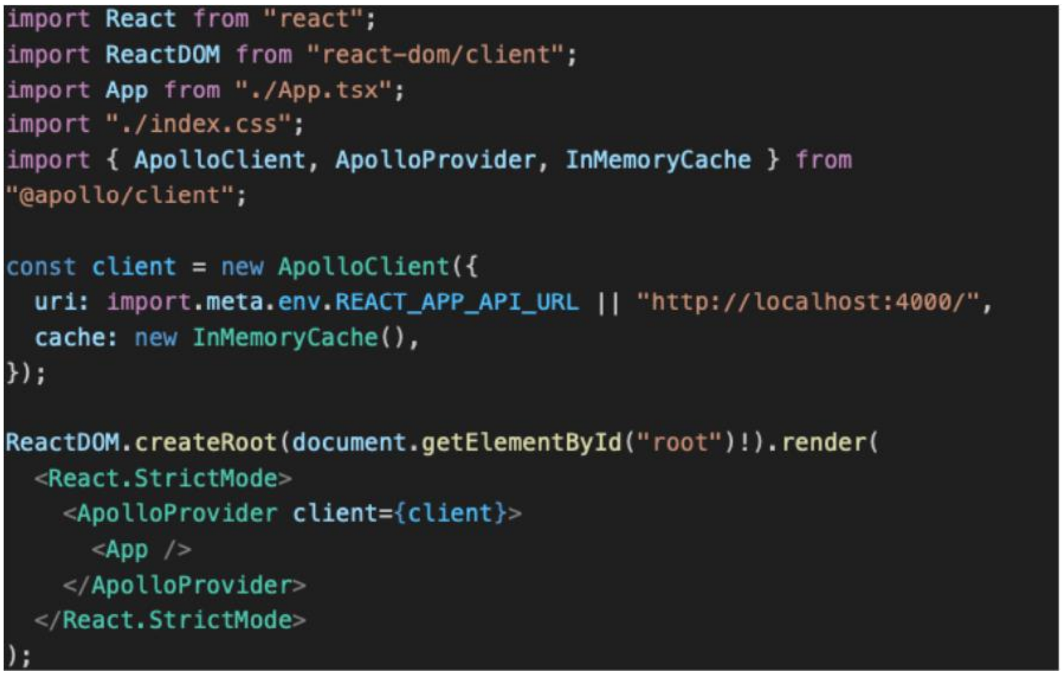
```
import { ApolloClient, InMemoryCache, ApolloProvider, gql } from
'@apollo/client';

const client = new ApolloClient({
  uri: import.meta.env.REACT_APP_API_URL || "http://localhost:4000/",
  cache: new InMemoryCache(),
});
```

Figure 26. Frontend ApolloClient initialization.

The `uri` parameter defines the URL of our GraphQL server. The `cache` variable represents an instance of InMemoryCache, which Apollo Client utilizes to store and cache query results obtained after fetching them.

The `ApolloProvider` component integrates Apollo Client with React. Similar to React's Context Provider, it encapsulates the React application, making Apollo Client accessible throughout the component hierarchy via the application context.

In the `main.tsx` file, the React app is wrapped by the ApolloProvider. It is recommended to position the ApolloProvider at the highest level, preferably above any components that require access to GraphQL data.

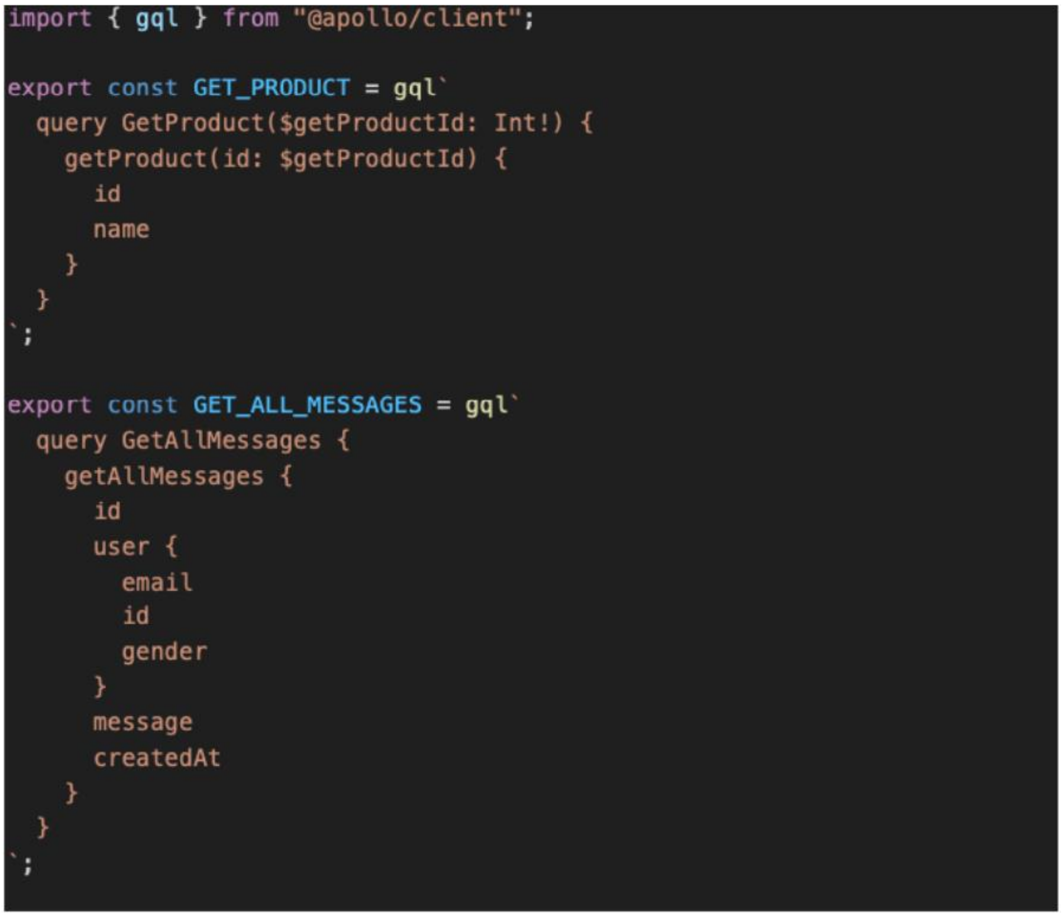


```
import React from "react";
import ReactDOM from "react-dom/client";
import App from "./App.tsx";
import "./index.css";
import { ApolloClient, ApolloProvider, InMemoryCache } from
"@apollo/client";

const client = new ApolloClient({
  uri: import.meta.env.REACT_APP_API_URL || "http://localhost:4000/",
  cache: new InMemoryCache(),
});

ReactDOM.createRoot(document.getElementById("root")!).render(
  <React.StrictMode>
    <ApolloProvider client={client}>
      <App />
    </ApolloProvider>
  </React.StrictMode>
);
```

Figure 27. ApolloClient and React app integration.

The executable queries can be defined by wrapping each of them in the `gql` template literal in file `graphql-queries.ts.`

For example, the GET_PRODUCT query can take a product id as an input, the backend uses the given id as an argument to filter the correct product and response to the client. On the other hand, the GET_ALL_MESSAGES query has no arguments because all messages will be returned.

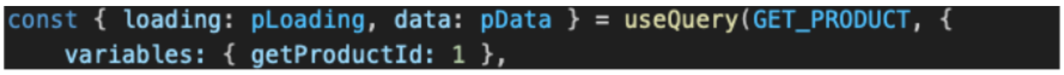


```
import { gql } from "@apollo/client";

export const GET_PRODUCT = gql`
  query GetProduct($getProductId: Int!) {
    getProduct(id: $getProductId) {
      id
      name
    }
  }
`;

export const GET_ALL_MESSAGES = gql`
  query GetAllMessages {
    getAllMessages {
      id
      user {
        email
        id
        gender
      }
      message
      createdAt
    }
  }
`;
```

Figure 28. Executable queries in client side.

In a React component, React Hook `useQuery` or `useLazyQuery` provided by `@apollo/clients` library is to call and use the query.

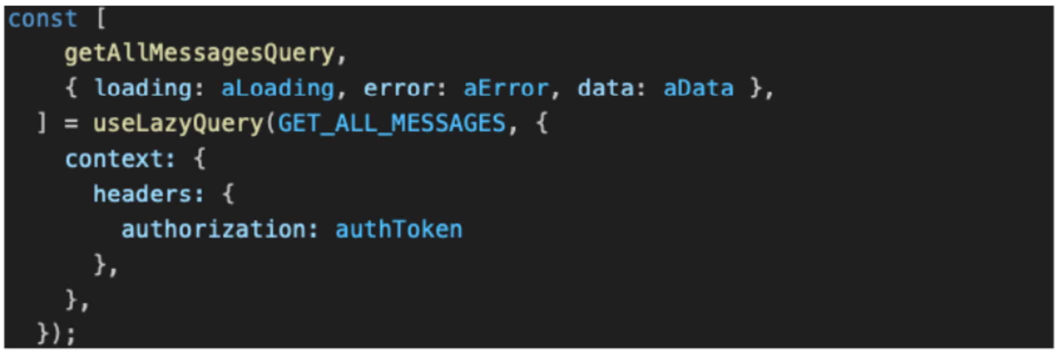
If the query request needs inputs such as id, the inputs can be defined with the `variables` field, similar to the code below.



```
const { loading: pLoading, data: pData } = useQuery(GET_PRODUCT, {
    variables: { getProductId: 1 },
```

Figure 29. Client query with inputs.

For queries that require authentication, the authorization header can be added to the request JWT token.



```
const [
    getAllMessagesQuery,
    { loading: aLoading, error: aError, data: aData },
  ] = useLazyQuery(GET_ALL_MESSAGES, {
    context: {
      headers: {
        authorization: authToken
      },
    },
  });
```

Figure 30. Client query with authentication.

Upon rendering, the `useQuery` hook initiates the execution of the query and yields a result object comprising loading, error, and data properties. Apollo Client seamlessly manages the loading state of the query, providing the loading property to reflect its progress. Likewise, the error property indicates any potential errors encountered during the query execution.

5 CONCLUSIONS AND ASSESSMENT

Throughout the development and testing phases of this thesis project, the reliability of the GraphQL Federation Service has been a paramount focus. Testing procedures and validation checks have been implemented to ensure the stability and robustness of the system. The integration of Node.js and PHP services, orchestrated through federation architecture, has demonstrated consistent and accurate responses to client requests. By adhering to best practices in software development and GraphQL standards, the service has proven its reliability in handling various use cases.

The system architecture, built upon GraphQL principles, provides a unified and intuitive interface for clients to interact with diverse APIs. The federation architecture seamlessly integrates multiple services into specialized subgraphs, simplifying data retrieval and enhancing efficiency. Developers and end-users alike benefit from rich documentation, standardized GraphQL schemas, and streamlined communication between services. The user-friendly design and flexibility of the service contribute to its high usability rating.

The transferability of the GraphQL Federation Service is a notable feature, allowing for easy integration into a variety of projects and environments. The modular design of the service, with separate Node.js and PHP services transformed into subgraphs, offers flexibility and adaptability. By utilizing standard GraphQL libraries such as Apollo Server and LightHouse PHP, the service ensures compatibility with a range of frameworks and platforms. The schema definitions and query resolvers are structured to facilitate easy transferability to different applications, making the service adaptable for future projects.

In conclusion, this thesis project has successfully implemented a GraphQL Federation Service, demonstrating reliability, usability, and transferability. The reliability of the system has been validated through testing, ensuring stable and accurate responses to client requests. Usability has been a central focus, with the service providing an intuitive interface for interacting with diverse APIs. Transferability is a key feature, allowing for easy integration into various projects and environments. Overall, the adoption of GraphQL Federation has proven beneficial, offering improved efficiency, flexibility, and scalability in handling data queries and interactions within the system.

REFERENCES

Amazon Web Services, Inc. (n.d.). What is RESTful API? - RESTful API Explained - AWS. Amazon Web Services, Inc. Retrieved March 2, 2024, from https://aws.amazon.com/what-is/restful-api

Apollo GraphQL Docs. (2019). Introduction. Apollo GraphQL Docs. https://www.apollographql.com/docs/react/

Archibald, J. (2015, December 8). Introducing Background Sync | Blog. Chrome for Developers. https://developer.chrome.com/blog/background-sync

Bhattacharya, B. (2021, October 28). An introduction to GraphQL federation. Tyk API Management. https://tyk.io/blog/an-introduction-to-graphql-federation/

DEV Community. (2020, April 14). The Graph in GraphQL. DEV Community. https://dev.to/bogdanned/the-graph-in-graphql-1l99

Huder, K. N. (2019). Modifikovanij sposib kešuvannya danih klients'koj biblioteki Apollo-Client dlya GraphQL. [Modified method of caching Apollo-Client client libraryfor GraphQL.] Original publication: Модифікований спосіб кешування даних
клієнтської бібліотеки Apollo-Client для GraphQL. Master's thesis, Igor Sikorskij KPI.

joan? (2024, February 13). GraphQL vs REST: Choosing the Right API for Your Project. DEV Community. https://dev.to/joanayebola/graphql-vs-rest-choosing-the-right-api-for-your-project-193m?fbclid=IwAR2wKbLk5sVDdewo9wxfkblGqmfH3PVlSRx1pHiBQW25GnZMTitOUDHsorQ

Laravel. (n.d.). Installation. Laravel. Retrieved March 2, 2024, from https://laravel.com/docs/10.x#meet-laravel

NodeJS. (n.d.). About Node.js. Node.js. Retrieved March 2, 2024, from https://nodejs.org/en/about

npm Docs. (2019). npm | npm Documentation. Npm Docs. https://docs.npmjs.com/cli/npm

OpenJS Foundation. (2017). Express - Node.js web application framework. Expressjs.com. https://expressjs.com/

Patel, P. (2018, April 18). What exactly is Node.js? FreeCodeCamp.org. https://www.freecodecamp.org/news/what-exactly-is-node-js-ae36e97449f5

PHP. (2019). PHP: What is PHP? - Manual. Php.net. https://www.php.net/manual/en/intro-whatis.php

React. (n.d.). Describing the UI. React. Retrieved March 1, 2024, from https://react.dev/learn/describing-the-ui

Richardson, C. (2017). Microservices.io. Microservices.io; Chris Richardson. https://microservices.io/

S. Gillis, A. S. G. (2020, September). What is REST API (RESTful API)? Techtarget.com. https://www.techtarget.com/searchapparchitecture/definition/RESTful-API

W3Schools. (2022). Introduction to HTML. W3schools.com. https://www.w3schools.com/html/html_intro.asp

W3schools. (2019). CSS Introduction. W3schools.com. https://www.w3schools.com/Css/css_intro.asp