



Escola d'Enginyeria de Telecomunicació i
Aeroespacial de Castelldefels

UNIVERSITAT POLITÈCNICA DE CATALUNYA

TREBALL FINAL DE GRAU

TÍTOL DEL TFG: Implementación de GraphQL en la REST API existente

TITULACIÓ: Grau en Enginyeria de Sistemes de Telecomunicació

AUTOR: Binod Lama

DIRECTOR: Miguel Valero García

DATA: 24 de octubre 2019

Títol: Implementing GraphQL in Existing REST API

TITULACIÓ: Grau en Enginyeria de Sistemes de Telecomunicació

Autor: Binod Lama

Director: Miguel Valero Gracia

Data: 24 de Octubre 2019

Resum

Classpip es la aplicación desarrollada por el alumno y profesor de EETAC. El concepto de classpip se basa en el principio de la gamificación, que utiliza el principio del juego en el contexto no relacionado con el juego. Classpip aplica el concepto de gamificación en los sistemas educativos.

Todo el proyecto de class pip se divide en 3 partes principales: tablero de classpip que es la aplicación web para escritorio, classpip mobile es para dispositivos Android / IOS y el servicio classpip que sirve como back-end para estos dos clientes.

El objetivo principal de este proyecto es mejorar el proceso de obtención de datos mediante la implementación de GraphQL. Todo el trabajo se realiza para implementar graphql además de la API de REST existente que usa un loopback de llamada de marco. Hemos probado varias arquitecturas de graphql que pueden ayudar a mejorar el proceso general de recuperación de datos desde la base de datos.

GraphQL fue desarrollado por Facebook, cuyo objetivo principal es aumentar el rendimiento de los usuarios de dispositivos móviles porque en GraphQL solo obtenemos lo que necesitamos en una sola solicitud que ahorra el ancho de banda y el tiempo para cargar esos datos.

Las compañías de software populares como GitHub, Instagram, Twitter, Stack Share y más ya han implementado graphql.

Title: Implementing GraphQL in Existing REST API

Degree: Telecommunications systems Engineering

Author: Binod Lama

Director: Miguel Valero Gracia

Date: October 24 2019

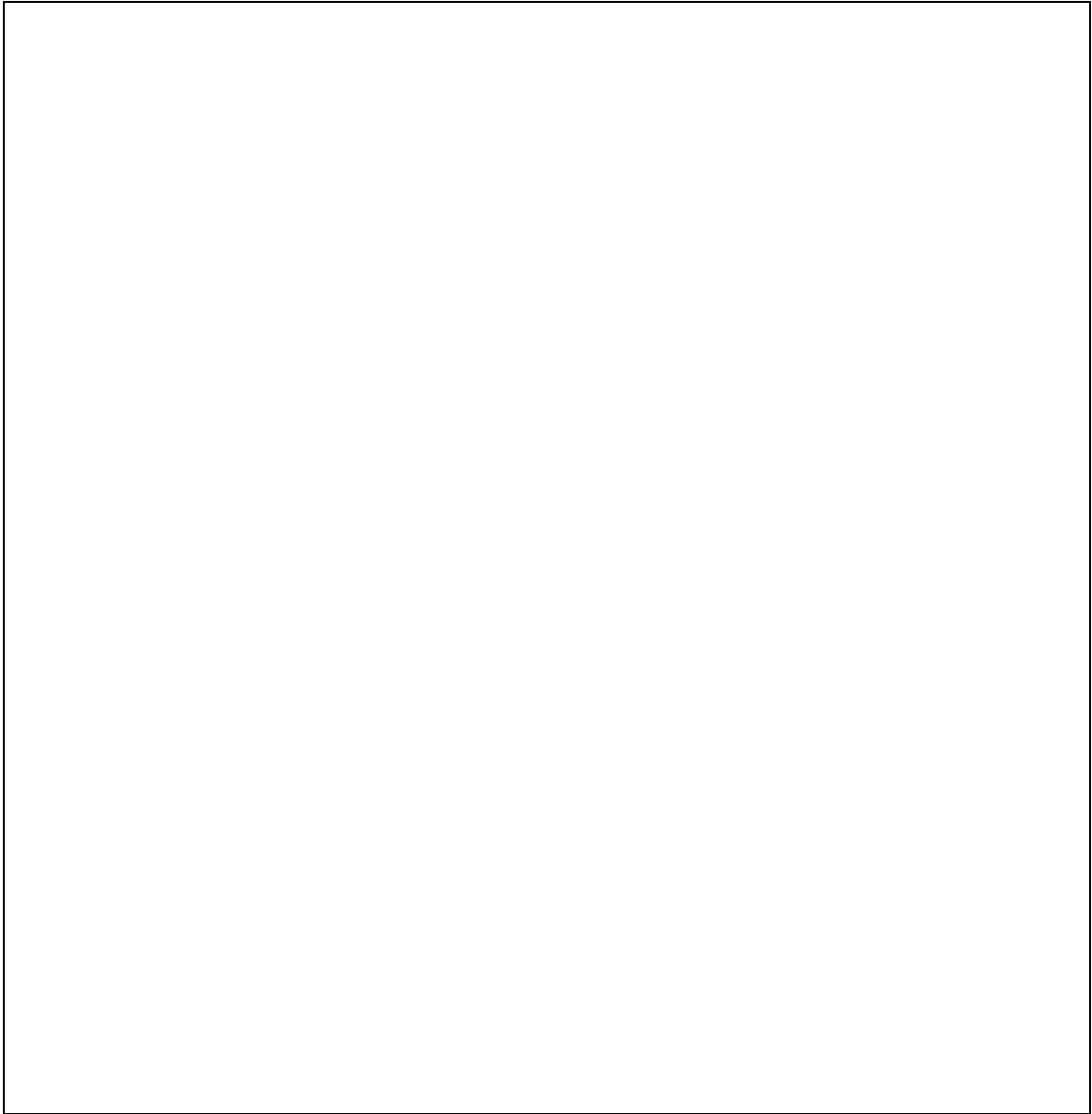
Overview

Classpip is the application developed by the student and professor of EETAC . The concept of classpip is based the principle of gamification which uses the gaming principle to the non-gaming context. Classpip applies the concept of gamification into education systems.

The whole project of class pip is divided into 3 main parts: classpip dashboard which is the web application for desktop, classpip mobile is for the android /IOS devices and the classpip service which serves as the backend for these two clients.

The main objective of this project is to improve the data fetching process by implementing **GraphQL**. The whole work is done to implement graphql on top of existing rest api which uses a framework call loopback .We have tried various graphql architecture which can help improve the overall process of data fetching from the database.

GraphQL was developed by Facebook whose main goal is to increase the performance of the mobile users because in GraphQL we only get what we need in just one request which saves the bandwidth and the time to load that data. Popular software companies like GitHub, Instagram, Twitter, Stack Share and more has already implemented graphql .



INDEX

INTRODUCTION	9
CHAPTER 1. APPLICATION PROGRAMMING INTERFACE	10
1. Application Programming Interface	10
1.1.1. Rest API.....	10
1.1.2. Limitation of REST API.....	11
1.2. Graphql API.....	12
1.2.1. Schema.....	13
1.2.2. Resolver.....	13
1.2.3. Create Graphql server	14
1.2.4. Limitations of Graphql.....	17
CHAPTER 2. ADVANTAGE OF USING GRAPHQL OVER REST.....	18
2.1. Eliminate under and over data fetching.....	18
2.2. Has strong schema type.....	20
2.3. Schema stitching.....	20
2.4. Rapid Product Iterations on the Frontend	20
2.5. Graphql has Rich open-source ecosystem	20
CHAPTER 3. POPULAR GRAPHQL ARCHITECTURE	21
CHAPTER 4. GRAPHQL ARCHITECTURE IMPLEMENTED TO REST	23
4.1. Replacing the existing REST with Graphql server	23
4.2. Graphql as a gateway between REST and the client.....	24
CHAPTER 5. IMPLEMENT GRAPHQL IN EXISTING REST API	25
5.1. Analyze the data model of the REST API.....	25
5.2. Generate Graphql schema based on data model.	25
5.3. Implement resolver functions for the schema.	25
CHAPTER 6. IMPLEMENT GRAPHQL IN CLASSPIP SERVICE.....	26
6.1. Graphql as the gateway architecture:	26
CHAPATER 7. PREFORMANCE TEST	29

7.1. Scenario 1: Get the student name and school information	29
7.2. Scenario 2: Get the student, school, rewards, point relation:	31
CHAPTER 8. CONSUME GRAPHQL IN CLASSPIP DASHBOARD	33
8.1. To display /groups pages:	33
8.2. To display /groupStudents/ page:	36
8.1.2. Create new collectionCards	40
CHAPTER 9. CONCLUSION	42
REFERENCES	43

INTRODUCTION

As the world of application Program interface (API) is evolving every day with the growing internet technology we need to get updated our applications with it. As the new technology comes that doesn't mean they are right to use in every scenario. We need to analyze their capabilities and use them as our needs.

Graphql is also a new evolving technology that lets rethink the developers how to build the client and API web applications. Graphql is a query language for APIS which lets client to fetch data from database in an efficient way without wasting any bandwidth. It lets the client specify exactly what data it needs.

We need to think Graphql as a tool to build our application. Before we implement Graphql in our project we need to rethink the cost of implementation and how it will help to grow our applications.

Classpip is a web application developed by the professors and the students of EETAC. It was started back in 2016. It is a server-based application that needs to interact with server to get its data. The main project of Classpip is divided into three parts: ClassPip Dashboard which is for the desktop web apps, Classpip Mobile and the Classpip Service which serves as a REST API for our desktop and Mobile clients. The main objective of this document is to explain the process of implementation of Graphql which can be useful for fetching the data. As the service of Classpip uses a framework called strong loopback which is based on REST architectural style, Web services.

The project documentation is divided in 9 different chapters. In Chapter 1 we will briefly introduce the Api. This chapter will explain the core concept of API and how api world works.

In the second chapter we will briefly compare the concept of REST and Graphql. In this chapter we will explain the advantages of graphql.

In the third and the fourth chapter we will explain about the different architecture that can be used to implement Graphql with some examples. In chapter 5 and 6 we will finally explain the process of creating a new Graphql server and implement in Classpip dashboard. In these chapter we will explain how and why Graphql should be implemented in the coming future to increase the efficiency of our application classPip.

In chapter 7 you can find about the performance test between rest and graphql. We have created some scenario to test their performance.

In chapter 8 we have explained about the process of implementation of graphql in Classpip dashboard which is based on angular. Finally, in the last chapter we will provide the conclusion over this project.

.

CHAPTER 1. APPLICATION PROGRAMMING INTERFACE

1. Application Programming Interface

Application Programming interface (API) which acts as a middle man between client and the server. It is a way for software to communicate with each other.

When the data is requested from the frontend (Mobile, web apps) to the database then API is responsible to make it happen. It is a set of rules that can be used to share information between client and the server.

As the internet is expanding and more mobile application are developing which needs a complex and efficient way of data fetching from the server. Some of the web service api are described below:

1.1.1. Rest API

Rest (Representational state transfer) is a popular API design architecture used to implement web services. It is a simple and flexible way of accessing web services. The server will send the representation of state of the requested resource to the client. Most of the representation of the state can be in json or XML format.

Basically, Rest api are stateless which means clients are responsible for storing all the application state related to client-side information. Clients' needs to send state information to the server whenever it is needed. The client and the server can operate independently in REST api which means are replaceable and all the data fetched from the server can be cached in client side which will improve the overall performance.

REST api are based on 5 different http methods which are GET (to fetch data), POST (to modify the data in server), PUT, PATCH and DELETE.

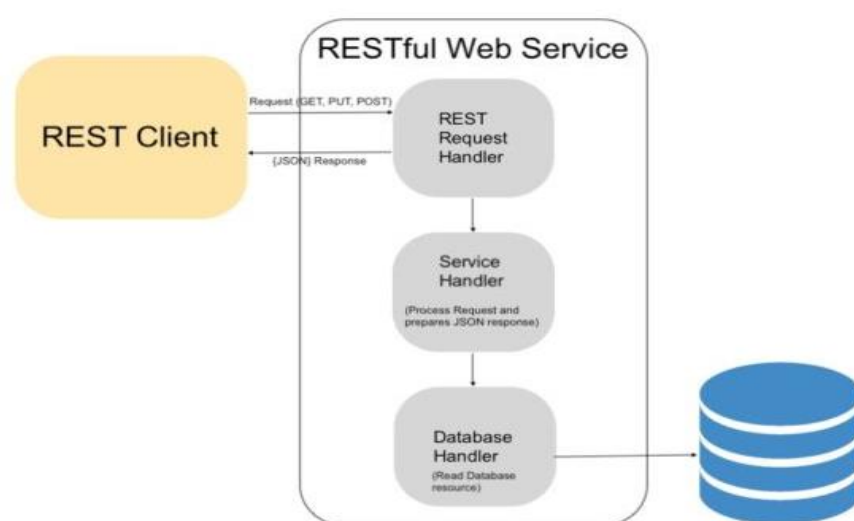


Fig. 1 Rest api Architecture source (<https://phpspot.com/php/php-restful-web-service/>)

Figure 1 shows the general architecture of Rest where it lies in between client and the database.

Our classpip service is also designed in REST architecture which uses an open source framework known as loopback. Loopback framework is built on top of express js with various set of node.js modules which can be used independently to build REST api with little or no coding. It also generates api code automatically which makes design and testing easier. It can also be connected to multiple data source and glued on top of existing services.

As our application grows we need to have many relational databases which will need many endpoints to get information. To get the complex information from our database we need to generate highly customized endpoints which may be used for only that kind of data request.

1.1.2. Limitation of REST API

Some of drawbacks of REST api which make it less effective are described below:

a) Multiple Round-trip Time

To get the information from different tables or databases we need to fetch data using different endpoints which leads to multiple round trip to get what we want.

b) Over and under data fetching

As our application grows, there will be new requirements of data from the server. The client often needs only some parts of the data stored in the server but api designed with rest do not have the function of providing which led to over fetching of the data. Over fetching led to degrade the performance of the product.

As we only want the name of students and their school's name but from the first request we will receive all the information about the student that are not useful which led to over data fetching and from the first request we cannot get the school name as we only get School's id, so we do not have enough information which led under data fetching

1.2. GraphQL API

Moto of GraphQL: **What you want is what you get**

GRAPHQL is basically a query language for an API. It is not tied to any specific database or storage engine, but it is instead backed by your existing code and data.

Basically, GraphQL has only one endpoint which is http post request. The client sends a customized post request get the data only what he wants. In other words, clients have the power to define what type of data to fetch from the database. It helps where our client needs a flexible response format to avoid extra massive data transfer. It is also supported in various popular programming languages like python, c#, JavaScript etc. In Figure 2 shows the general architecture of GraphQL Where all the client sends graphql query to get their data from the database or microservice or other rest api.

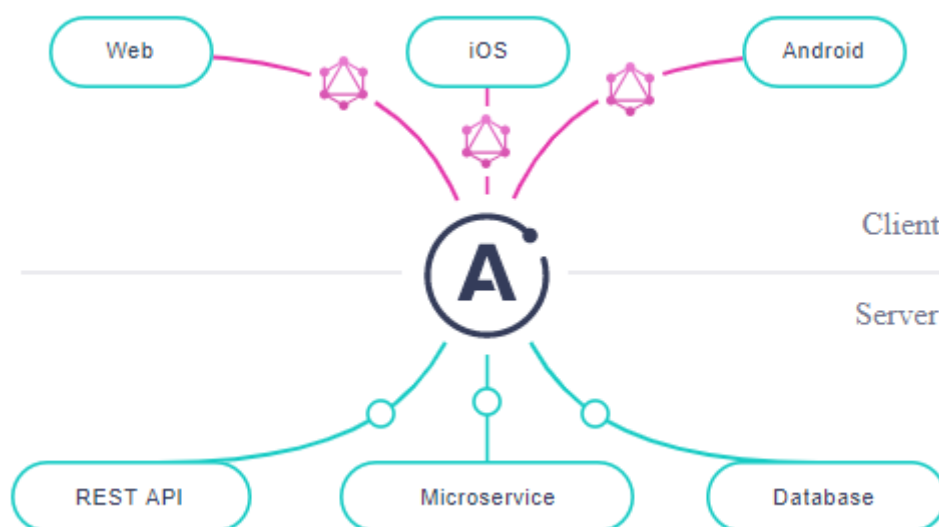


Fig. 2 GraphQL Architecture (source: <https://www.apollographql.com/docs/apollo-server/>)

Two things are the most important part of GraphQL: Schema and Resolver function.

1.2.1. Schema

Schema are the brain of a GraphQL api. It is where the data models are defined that can be fetched through GraphQL server. It defines what a client can get and can fetch from a server, and the relationship between the different types.

As schema are the contract between client and the server about how to fetch data.

Some important special types of schema:

Query: Query are used to read or fetch the data from the server. They can be comparable with REST GET requests which are used to fetch the data from the server. Every GraphQL server must include query type.

Mutation: To post or modify the data in the server mutations are used and also returns a value. They are used to insert, update or delete the data in the server. GraphQL server may or may not include mutation type.

Subscription: The main goal of the subscription is to send real time updates to subscribed client when new or existing link element is created or upvoted.

1.2.2. Resolver

Resolvers are function that resolves a value for a type or field in a schema. It is where our logics are defined. It generates the response for the GraphQL query that were defined in the schema. It acts as a query handler for our GraphQL.

Resolver function accepts four arguments which is described table below:

Fieldname(obj, args, contexts, info) => {results}

Arguments	Descriptions
obj	It contains the results from the resolver on the parent fields
args	It contains an object with arguments passed into the fields into query.
Context	Shared object by all the resolvers including authentication information(tokens), base endpoints
info	It contains the information about the execution state of the query, may include fields name, path to the field from the root

Table1: About resolver function

Table 1 describe the arguments that can be passed to a resolver function.

A resolver function can return different types of values:

Arguments	Descriptions
Null	Return null or undefined which indicates object could not be found.
array	if the schema indicates that the result of a field should be a list
Promise	return a promise that resolves an array or an array of promise
Scalar object	return value without special meanings but it is simply passed down into any nested resolvers.

Table 2 Return values of a resolver

In the above table 2 , it describes all the return types of values.

1.2.3. Create Graphql server

For the understanding of how resolver work we have created a simple Graphql server which stores information about students and their task.

By following the simple steps we can create our Graphql server.

Step1: Define schema

```

1  type User {
2    id: ID! @id
3    username: String! @unique
4    password: String!
5    rol: String!
6    puntos: Int!
7    tasks:[Task!]!
8  }
9
10 type Task{
11   id: ID! @id
12   title: String!
13   description:String!
14   status:String!
15   user:User!
16 }
17
18 type query{
19   tasks: [Task!]!
20   users:[User!]!
21   task(where: TaskWhereUniqueInput!): Task
22   user(where: UserWhereUniqueInput!): User
23 }
24
25 input TaskWhereUniqueInput {
26   id: ID
27 }
28
29 input UserWhereUniqueInput {
30   id: ID
31   username: String
32 }

```

Fig.3 Simple graphql schema

In figure 3 we have defined how schema are defined in graphql where we have 3 types User ,Task and query. we can query tasks which returns an array of tasks and users which returns an array of users. To retrieve specific task or user, we can use data type ID. (Note: “!” means that field cannot be null)

Step 2: Implement in Resolver function:

From the figure 3, we have demonstrated how to implement our graphql. Tasks, users, task and user are the resolver that handle the query. As discussed above, the taskId can be retrieved from args argument, obj will contain the query object itself. To return the task with specific taskId we can use **find** method to get that task from the database. To run the GraphQL server we have used graphql yoga framework which is based on express.js and Apollo.

After schema and resolver are implemented we can run our first graphql application by hitting npm start command.

```
const resolvers = {
  Query: {
    tasks(parent, args, {db}, info) { return db.tasks; },
    users(parent, args, {db}, info) { return db.users; },
    task(parent, args, {db}, info) {
      {
        const task= db.tasks.find((task)=>task.id===args.where.id)
        if (task==null) throw new Error ("Task id not found Id:"+args.where.id)

        return task;
      },
    },
    user(parent, args, {db}, info) {
      {
        const user= db.users.find((user)=>user.id===args.where.id)
        if (user==null) throw new Error ("User id not found Id:"+args.where.id)

        return user;
      }
    },
  },
};
```

Fig .4 GraphQL Resolver example code

Figure 4 is the actual code of implementation of resolver in graphql server.

GraphQL comes with his own web-based client known as GraphQL playground to execute query to which makes it easy to test.

Once the server is up open the browser and enter <http://localhost:4000>.

To fetch the task information following query is needed.

```
1 query
2 {
3   task(where:{id:1}) {
4     id
5     title
6     description
7   }
8 }
9
```

```
{
  "data": {
    "task": {
      "id": "1",
      "title": "Clean BathRoom",
      "description": "Clean the whole
BathRoom"
    }
  }
}
```

Fig. 5 Executing query in graphql playground to get the task information

In Figure 5 we have defined the query in the left side and in the right is the response of that query which the task information.

```
1 query
2 {
3   tasks {
4     id
5     title
6     description
7   }
8 }
```

```
{
  "data": {
    "tasks": [
      {
        "id": "1",
        "title": "Clean BathRoom",
        "description": "Clean the whole BathRoom"
      },
      {
        "id": "2",
        "title": "Clean kitchen",
        "description": "Clean the whole Kitchen"
      },
      {
        "id": "3",
        "title": "Clean salon",
        "description": "Clean the whole Salon"
      }
    ]
  }
}
```

Fig. 6 Executing query in graphql playground to get list of tasks

In figure 6 we can see how we can execute tasks query; the response is the list tasks in Json format.

Similarly, we can execute the query for users to get an array of users and user by user id.

1.2.4. Limitations of GraphQL

Some of drawbacks of GraphQL are explain below:

1.Caching

Overall the idea of cache is to lower down the server work by saving the data in client side for repetitive same request. As we only have post request in GraphQL, which means we cannot cache those post request. We need to use different libraries to overcome this problem or set the global unique Ids.

2.Error handling

As queries always returns http status code 200 despite that query was unsuccessful which makes it more complex while handling errors.

3.Performance

As we can send the nested query which means it needs time to process all the data. So, we should keep in mind the limitation of nesting in particular query to improve our application performance.

Chapter 2. ADVANTAGE OF USING GRAPHQL OVER REST

Some of the advantages of using Graphql over Rest are briefly described below

2.1. Eliminate under and over data fetching.

As in the above example of user and task Graphql project we can send customized query to get the task information along with the username and id just in one request.

```
query {  
  tasks {  
    id  
    title  
    description  
    user {  
      id  
      username  
    }  
  }  
}
```

Fig. 7 Example of nested query

In figure 7 we have defined a query where we will get the list of tasks along with user's id and username. The response of this query is shown in figure 8 .

```
"data": {
  "tasks": [
    {
      "id": "1",
      "title": "Clean BathRoom",
      "description": "Clean the whole BathRoom",
      "user": {
        "id": "1",
        "username": "Binod"
      }
    },
    {
      "id": "2",
      "title": "Clean kitchen",
      "description": "Clean the whole Kitchen",
      "user": {
        "id": "2",
        "username": "Yelan"
      }
    },
    {
      "id": "3",
      "title": "Clean salon",
      "description": "Clean the whole Salon",
      "user": {
        "id": "3",
        "username": "Miguel"
      }
    }
  ]
}
```

Fig. 8 response of the nested query

We can include only the needed fields to get the response from the server and it's just needing to request one we can get what we want.

But in Rest we need to send multiple request because one request is not enough to get all the required information, and we also get unwanted data which will increase the payloads and decreases the overall performance of the application.

2.2. Has strong schema type

Graphql schemas are the backbone of every Graphql API. Documentation can be generated automatically which will help the developer's life to be simpler otherwise they need to be created manually.

2.3. Schema stitching

We can connect and merge multiple Graphql api which means clients can connect to multiple api just by using single api. [Graphql bindings](#) take the idea of schema stitching to the next level by enabling a simple approach to reuse and sharing Graphql APIs.

2.4. Rapid Product Iterations on the Frontend

As we add new function in frontend we need different types of data. In Graphql they can send custom query to get those data which means they don't need to inform backend every time that they need certain types of data which can significantly increase the efficiency of overall work.

2.5. Graphql has Rich open-source ecosystem

As Graphql is used by most of the software companies like GitHub, Facebook (which created the Graphql) many tools are developed day by day which will certainly help to make Graphql a better api language.

Chapter 3. POPULAR GRAPHQL ARCHITECTURE

Some of the popular graphql Architecture in real world application are described below:

1: GraphQL directly connected to database. In this architecture graphql is responsible to fetch data directly from the database which is shown in figure 9 .

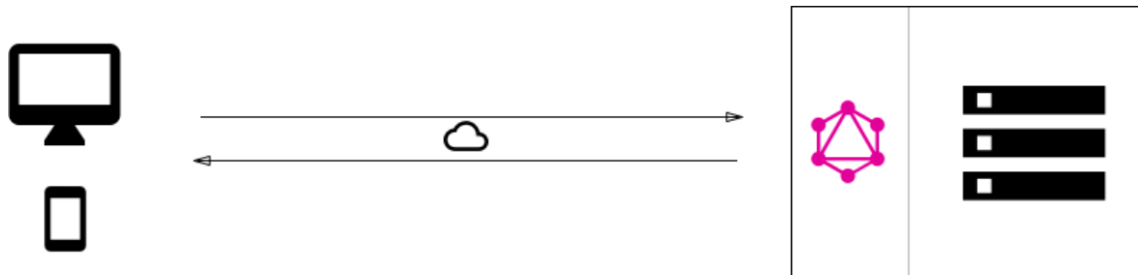


Fig. 9 Architecture (source: <https://www.howtographql.com/basics/3-big-picture/>)

2: GraphQL server as a layer between third party. GraphQL act as a gateway to different services or to the database or may be to the different REST api which is shown in the figure10.

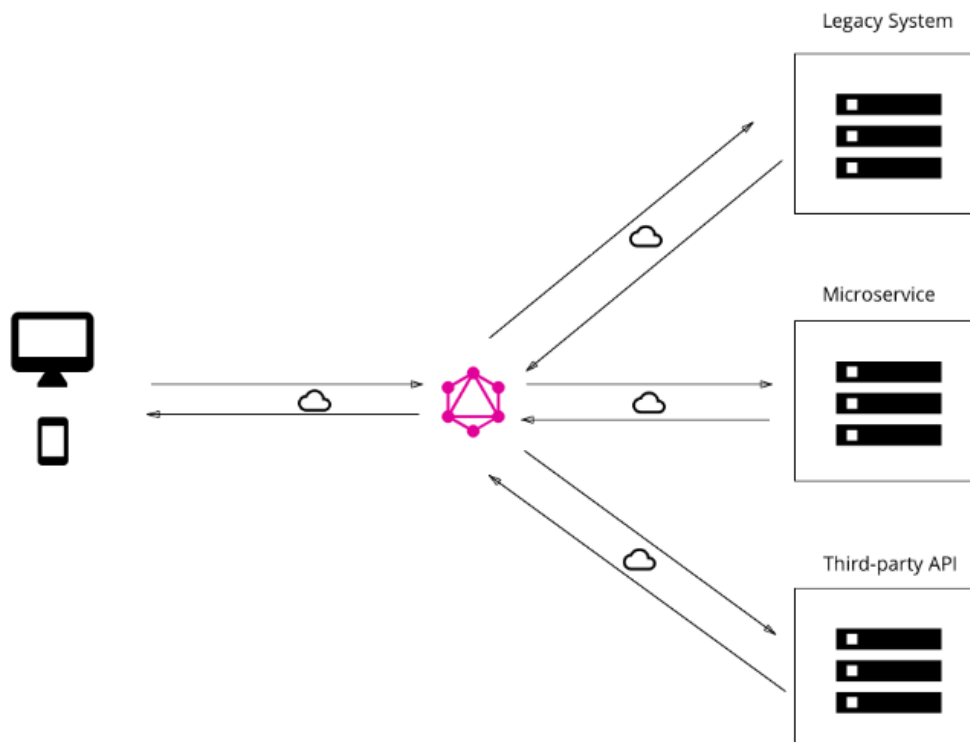


Fig.10 Architecture (source: <https://www.howtographql.com/basics/3-big-picture/>)

3: Hybrid approach which means it is connected to a database and different services and different rest api which is shown in figure 11.

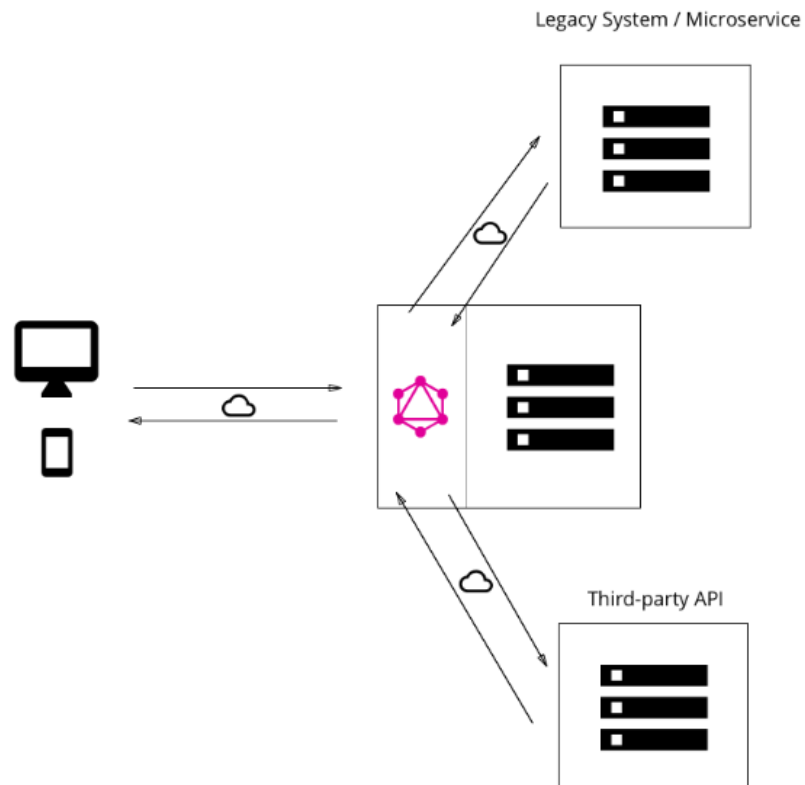


Fig. 11 Architecture (source: <https://www.howtographql.com/basics/3-big-picture/>)

CHAPTER 4. GRAPHQL ARCHITECTURE IMPLEMENTED TO REST

To implement graphql in our service class pip we have used the following two architecture which are replacing existing REST with graphql completely and GraphQL as thin layer in between REST and the client.

4.1. Replacing the existing REST with GraphQL server

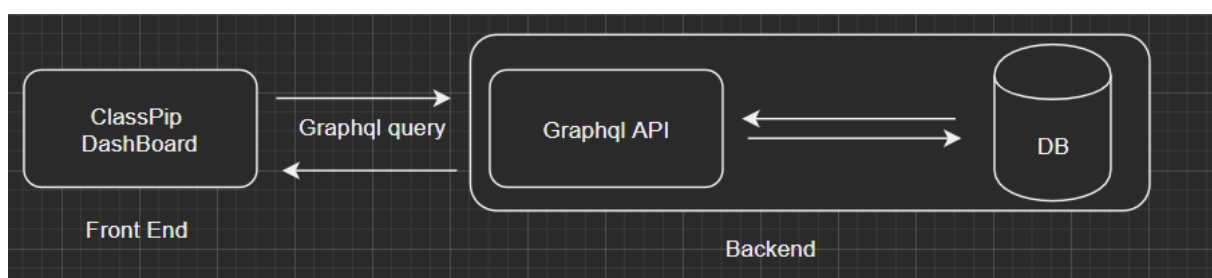


Fig. 12 Replacing REST with GraphQL

To implement the above architecture shown in fig 12 we need to change the following things:

In Server side: Create graphql server and connect to database.

In Client side: Change all the data fetching methods by using graphql query.

Some of the Advantages of this architecture are listed below:

- 1.No need to map every Rest api endpoint
2. Can configure graphql to obtain the nested and complex data type which needs multiple data fetching if we use Rest.

Some of the disadvantages of this architecture are listed below:

- 1.Difficult to connect with existing databases (Database fields mismatch)
2. Needs to reconfigure all the data fetching process in Client side which means a lot works in frontend side.

4.2. Graphql as a gateway between REST and the client

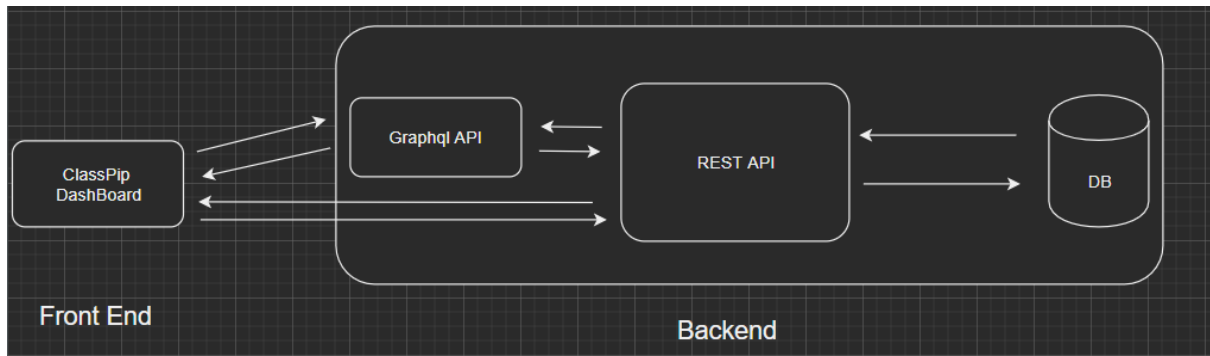


Fig.13 Graphql as gateway

To implement the above architecture shown in figure 13 we need to change the following things:
configuration:

In server: Create graphql server and map the rest endpoint

In client: Fetch the required complex data using graphql.

Some of the Advantages of this architecture are listed below:

- No need to configure database
- No need to change the current architecture and least work we can use the power of graphql.
- Can be extended in future for new functionality
- GraphQL and rest can work side by side
- Can configure graphql to obtain the nested and complex data type which needs multiple data fetching if we use Rest.

Some of the disadvantages of this architecture are listed below:

- Manually map the rest endpoints which can be complex.

CHAPTER 5. IMPLEMENT GRAPHQL IN EXISTING REST API

In the following simple steps described below we can implement GraphQL to our classPip easily:

5.1. Analyze the data model of the REST API

At first, we need to understand the data models of REST API. Basically, we need to know what the fields and their relationships are.

5.2. Generate GraphQL schema based on data model.

Once we understood the data model we can start creating GraphQL schema. AS there are different approaches to create our schema, some of them are listed below:

1.Manully

We can create the schema of our graphql server manually. This approach may be simple for the small project but for the large and complex data model it's difficult to create all the schema manually.

2.Using Prisma to create schema automatically.

We can use Prisma to create our graphql schema automatically. Prisma is a database attraction layer that turn database into graphql api. It is glue in between our database and the graphql server. With Prisma we can create automatically the CURD based Schema for our graphql.

5.3. Implement resolver functions for the schema.

Once the schemas are defined we need to write the resolver functions for it. How to write the resolver function which can be found in chapter 7.

CHAPTER 6. IMPLEMENT GRAPHQL IN CLASSPIP SERVICE

We have used 2 different architecture to implement graphql on classpip rest service where graphql as the thin layer in between rest and the client is the best architecture because we don't need to modify all the data fetching codes in the client sides and in the server, we can avoid the database mis match process, we do not need to connect directly to the database.

6.1. Graphql as the gateway architecture:

At first, we need to create a graphql server. Things that we need to create a graphql Server.

A. Graphql Schema

We have used Prisma to create our graphql schema. For this approach we need to have docker and Prisma installed. In the following steps we can create our graphql schema:

Step 1: create a folder called Prisma

Step 2: Inside Prisma folder hit **prisma init** command. We can select existing database or just create new database. Once done 3 files will be created in Prisma folder.

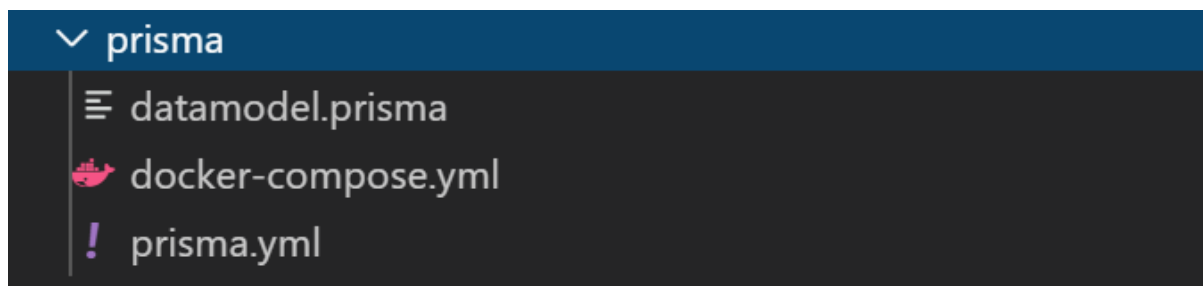


Fig.14 Prisma folder contains

In fig 14 we can see the folder structure of the folder Prisma.

In the first file datamodel.prisma we can define our data models for our database. Docker-compose.yml file contains all the required information about the docker Prisma images.

Prisma.yml is the root configuration file for Prisma service where we define the service endpoint, database type, Prisma client

Step 3: Modify the data model. Prisma file with classpip service model.

Step 4: run the **docker-compose up -d** command which will create the Prisma image file in the docker.

Step 5: Run Prisma deploy. Once it done we can run localhost:4466 in our browsers to check it's working.

Step 6: we have created a script that will automatically generate the prisma.graphql schema from our Prisma server .

Step7: To run the server up execute the **npm run get-schema** command.

B. Resolver:

As we are going to use graphql just as gateway, so we need to map the required REST api endpoints in our resolver function. In the example below, we can see how we can map our rest endpoints to graphql resolvers.

For example: Get the list of questions:

```
questions(parent, args, {
  db,
  RESTURL
}, info) {
  return fetch(`${RESTURL}/questions`).then(res => res.json())
},
```

Fig.15: Mapping Rest URL in resolver function

In fig15 we have a resolver name question. It takes argument RESTURL which is the URL for the Rest api. The return value will the list of questions in json format.

```
question(parent, args, {
  db,
  RESTURL
}, info) {
  const id = args.where.id
  return fetch(`${RESTURL}/questions/${id}`).then(res => res.json())
}
```

Fig.16: mapping Rest URL in resolver function

In the fig 16 we have a resolver name question. It takes argument RESTURL which is the URL for the Rest api and the id of question. The return value will be a question in json format.

We can use graphql playground to test our graphql query:

A screenshot of the GraphQL Playground interface. On the left, a query is defined:

```
1 query {  
2   questions {  
3     id  
4     name  
5   }  
6 }  
7
```

 A play button is visible. On the right, the JSON response is shown:

```
{  
  "data": {  
    "questions": [  
      {  
        "id": "1",  
        "name": "Respecto al ensanchamiento de UMTS, es cierto  
que:"  
      },  
      {  
        "id": "2",  
        "name": "Sea un sistema CDMA a 3.84 Mcps que ofrece un  
servicio de voz de 12.2 Kbps que requiere una Eb/No de 6 dB.  
Considerando una relación inter/intracell (f=0.6) y factor de  
actividad unitario. Podemos afirmar que:"  
      },  
      {  
        "id": "3",  
        "name": "La respiración celular o cell breathing es:"  
      }  
    ]  
  }  
}
```

Fig. 17 Testing in GraphQL play ground

The cool thing about graphql is that we can request only the fields that we need to fetch like in the above picture we can only fetch the list of questions but only its id and name which will help to reduce the payload.

In fig 17 we see how we have executed questions query using graphql playground whose response is the list of questions in json format.

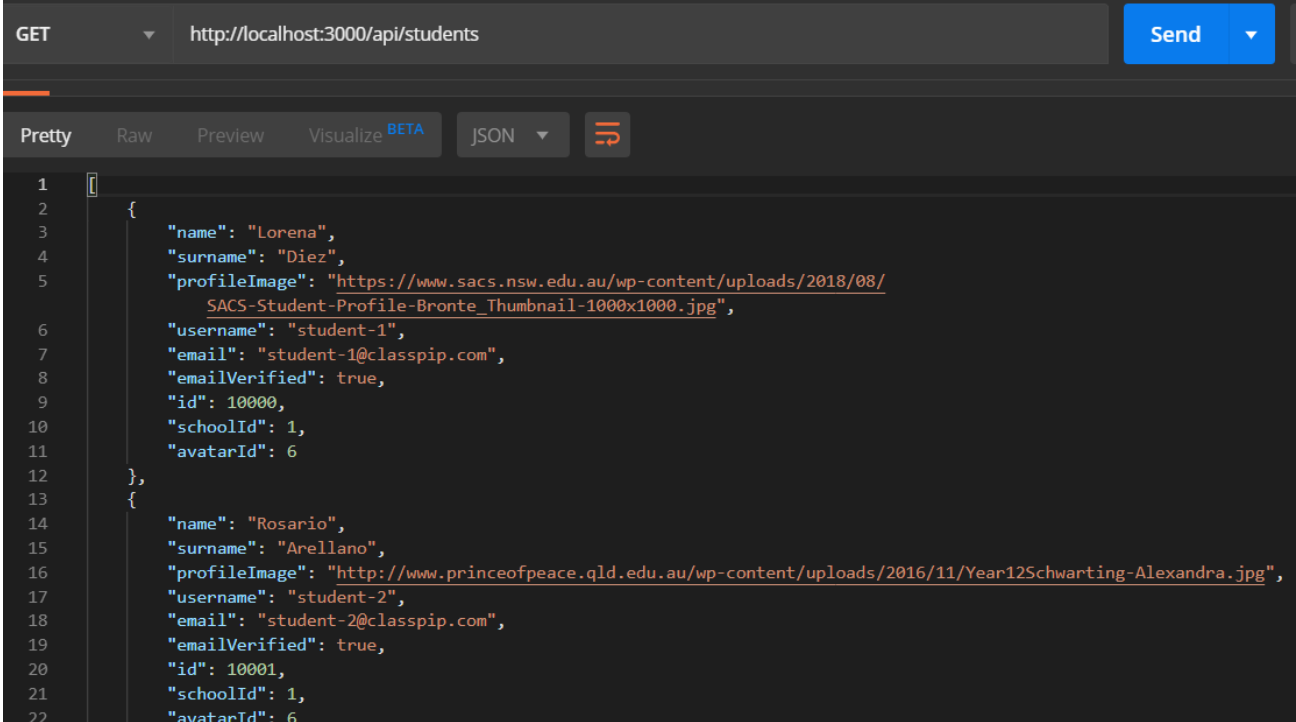
CHAPATER 7. PREFORMANCE TEST

To test performance of initial Rest implementation and after implementation we have created different scenario list below. For the purpose of the performance test we have used Postman.

7.1. Scenario 1: Get the student name and school information

To get the information about student and school information using REST we need to send the following GET request.

1.Get students list <http://localhost:3000/api/students>



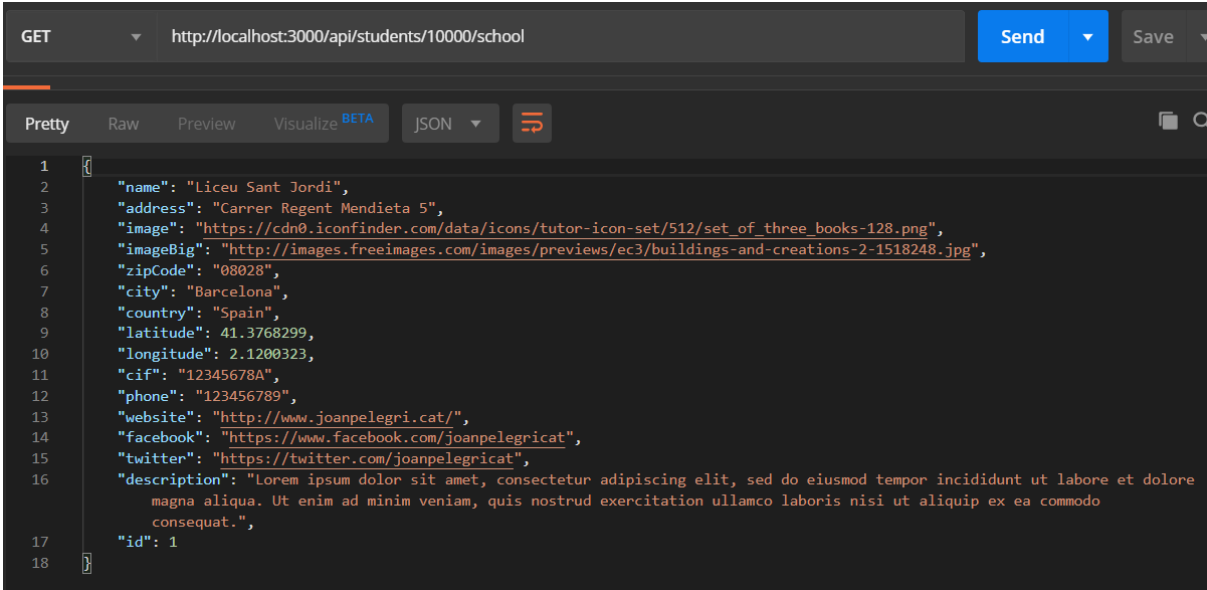
The screenshot shows a Postman interface with a GET request to `http://localhost:3000/api/students`. The response is displayed in JSON format, showing a list of two students. The first student is Lorena Diez, and the second is Rosario Arellano. Both students have their email verified and are associated with school ID 1.

```
1  {
2    {
3      "name": "Lorena",
4      "surname": "Diez",
5      "profileImage": "https://www.sacs.nsw.edu.au/wp-content/uploads/2018/08/
6      SACS-Student-Profile-Bronte_Thumbnail-1000x1000.jpg",
7      "username": "student-1",
8      "email": "student-1@classpip.com",
9      "emailVerified": true,
10     "id": 10000,
11     "schoolId": 1,
12     "avatarId": 6
13   },
14   {
15     "name": "Rosario",
16     "surname": "Arellano",
17     "profileImage": "http://www.princeofpeace.qld.edu.au/wp-content/uploads/2016/11/Year12Schwaring-Alexandra.jpg",
18     "username": "student-2",
19     "email": "student-2@classpip.com",
20     "emailVerified": true,
21     "id": 10001,
22     "schoolId": 1,
23     "avatarId": 6
```

Fig.18 List of students

In fig 18 we can see the list of students after we send the above get request to the Rest api using Postman.

2. Get `http://localhost:3000/api/students/{id}/school`



```
GET http://localhost:3000/api/students/10000/school Send Save

Pretty Raw Preview Visualize BETA JSON ↻

1 {
2   "name": "Liceu Sant Jordi",
3   "address": "Carrer Regent Mendieta 5",
4   "image": "https://cdn0.iconfinder.com/data/icons/tutor-icon-set/512/set_of_three_books-128.png",
5   "imageBig": "http://images.freeimages.com/images/previews/ec3/buildings-and-creations-2-1518248.jpg",
6   "zipCode": "08028",
7   "city": "Barcelona",
8   "country": "Spain",
9   "latitude": 41.3768299,
10  "longitude": 2.1200323,
11  "cif": "12345678A",
12  "phone": "123456789",
13  "website": "http://www.joanpelegri.cat/",
14  "facebook": "https://www.facebook.com/joanpelegricat",
15  "twitter": "https://twitter.com/joanpelegricat",
16  "description": "Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.",
17  "id": 1
18 }
```

Fig.19 response of the get request

In fig 19 we get school information for student whose id is 10000. So, we need to send N number of get request to get the school information for N number of students in the list.

Time consumed: 533 ms to get the student information + $9 \times 542\text{ms} = 0.542$ sec to get the information of 9 students and their school information

Total size of the payload: $2,68\text{KB} + 9 \times 1,14\text{KB} = 12.94\text{KB}$

To get the same information using graphql:

We just need to send a query with required field and call the graphql endpoint only once.

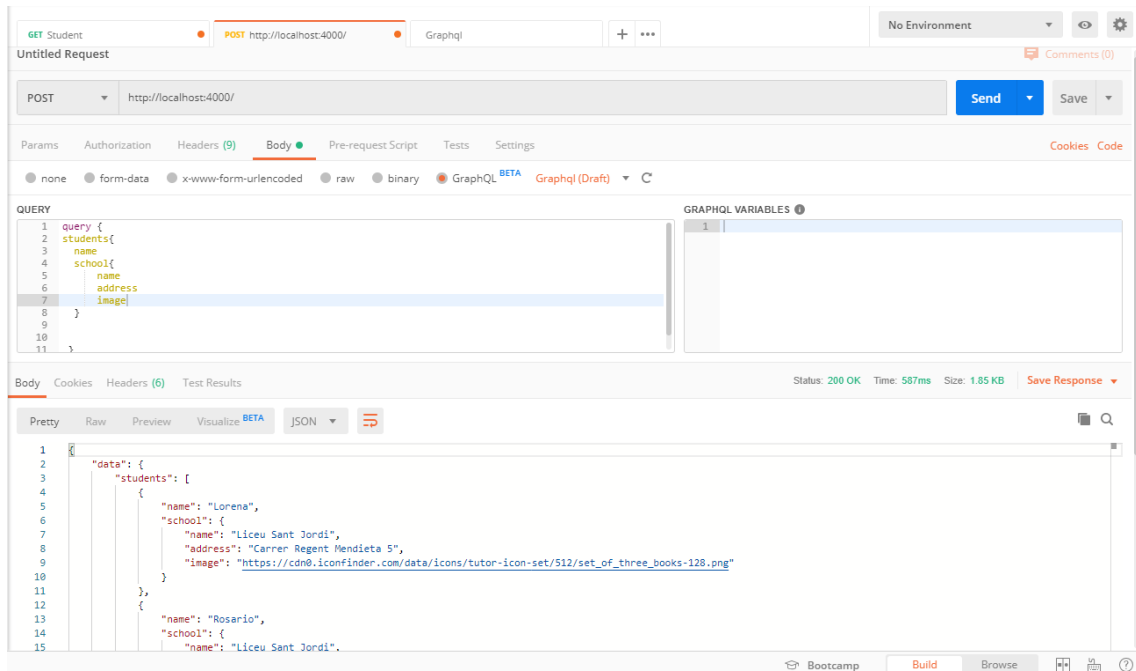


Fig.20 Graphql query to get the required information

In figure 20 shows the query to fetch data using graphql query.

Overall comparison of REST and graphql to get the information is shown in the table 3

	Time consumed	Total payload
REST	0.542 s	12.94 KB
Graphql	587 ms	1.85 KB

Table: 3 General comparision

7.2. Scenario 2: Get the student, school, rewards, point relation:

To get the same information about student name, school name, rewards and point relation using graphql

We just need to send a query with required field and call the graphql endpoint only once

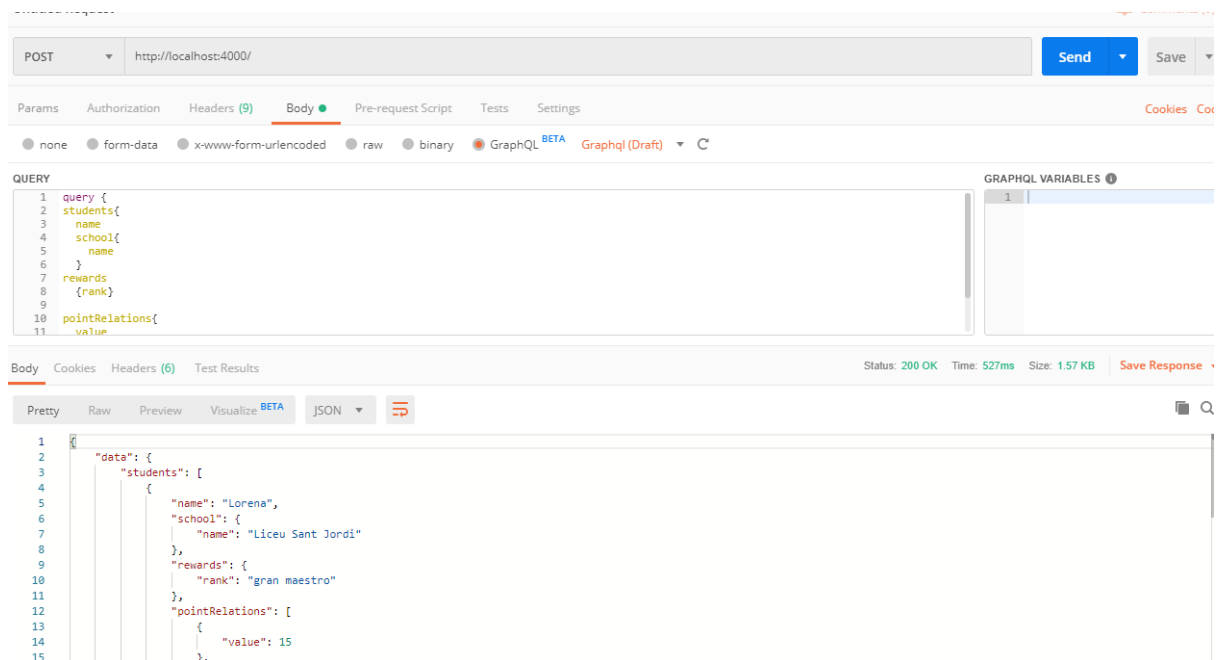


Fig. 21 Graphql query and response

Total time consumed: 527ms,
Total size:1.57 KB

CHAPTER 8. CONSUME GRAPHQL IN CLASSPIP DASHBOARD

To consume the graphql implemented api in classpip dashboard we just need to send post request to a graphql api to get what we need.

For demonstration how to consume a graphql api for our Classpip dashboard we have implemented two pages in which we need to fetch the information about groups and the students

8.1. To display /groups pages:

Data needed to display the /groups pages are list of Group names with group grade names and group matter names.

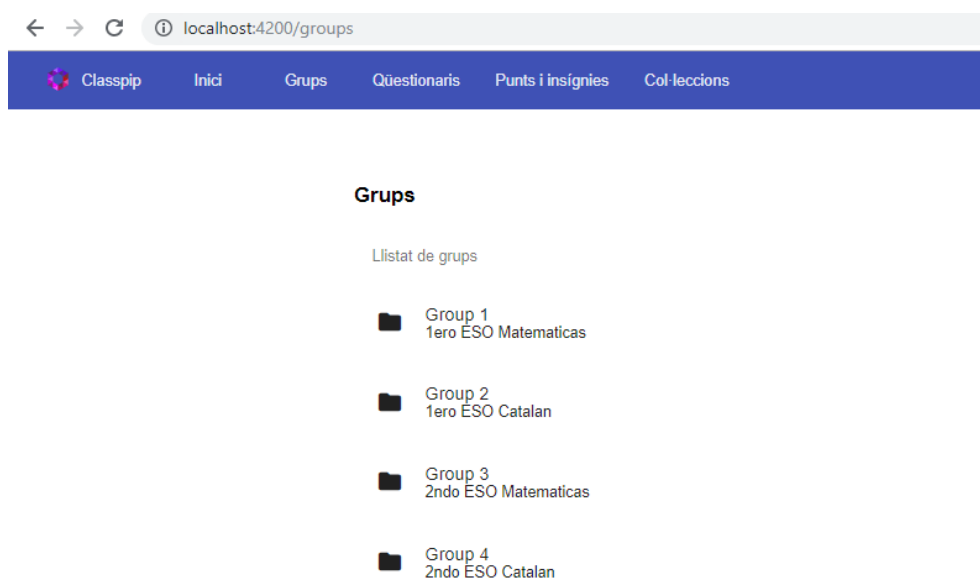


Fig. 22 Groups page in classpip dashboard

Rest implementation:

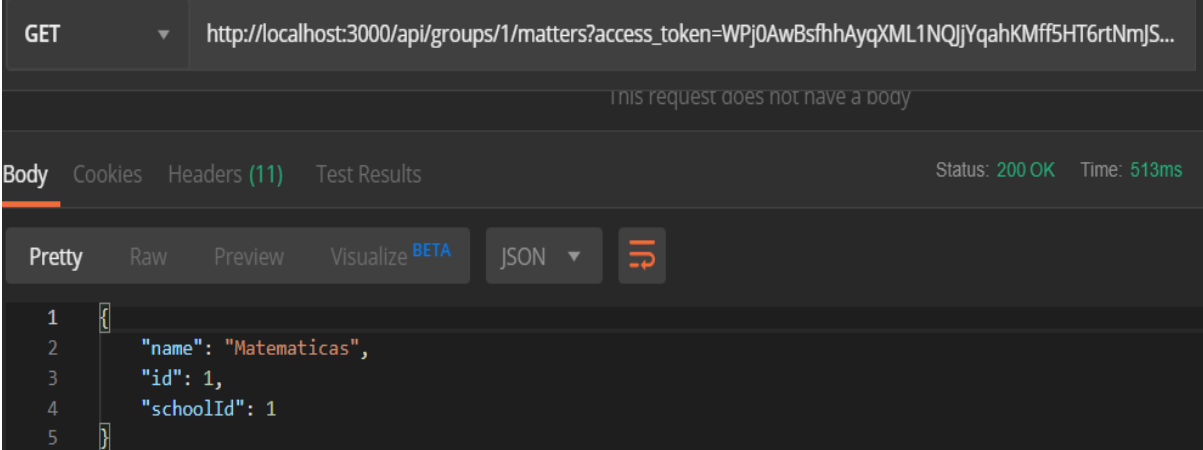
When a groups page is clicked then it needs to call 3 different endpoints to get all the required information:

- Get localhost:3000/api/groups where we will get the list of group id, group name, schoolId, teacherId, gradeId and matterId.

```
{
  "name": "Group 1",
  "id": 1,
  "schoolId": 1,
  "teacherId": 1000,
  "gradeId": 1,
  "matterId": 1
},
{
  "name": "Group 2",
  "id": 2,
  "schoolId": 1,
  "teacherId": 1000,
  "gradeId": 1,
  "matterId": 2
},
{
  "name": "Group 3",
  "id": 3,
  "schoolId": 1,
  "teacherId": 1000,
  "gradeId": 2,
  "matterId": 1
},
}
```

Fig. 23 Response list of groups

b. Get `localhost:3000/api/groups/{id}/matters` where we get the group matters name, id and schoolId. To get the information of whole groups matters information we need to call N times the number of Group.



```
GET http://localhost:3000/api/groups/1/matters?access_token=WPj0AwBsfhhAygXML1NQJjYqahKMff5HT6rtNmJS...
This request does not have a body
Body Cookies Headers (11) Test Results Status: 200 OK Time: 513ms
Pretty Raw Preview Visualize BETA JSON
1 {
2   "name": "Matematicas",
3   "id": 1,
4   "schoolId": 1
5 }
```

Fig. 24 Rest response of groups matter.

c. Get `localhost:3000/api/groups/{id}/grades` where we will get the groups grades name, id and schoolId. To get the information of whole groups grades information we need to call N times the number of Group.

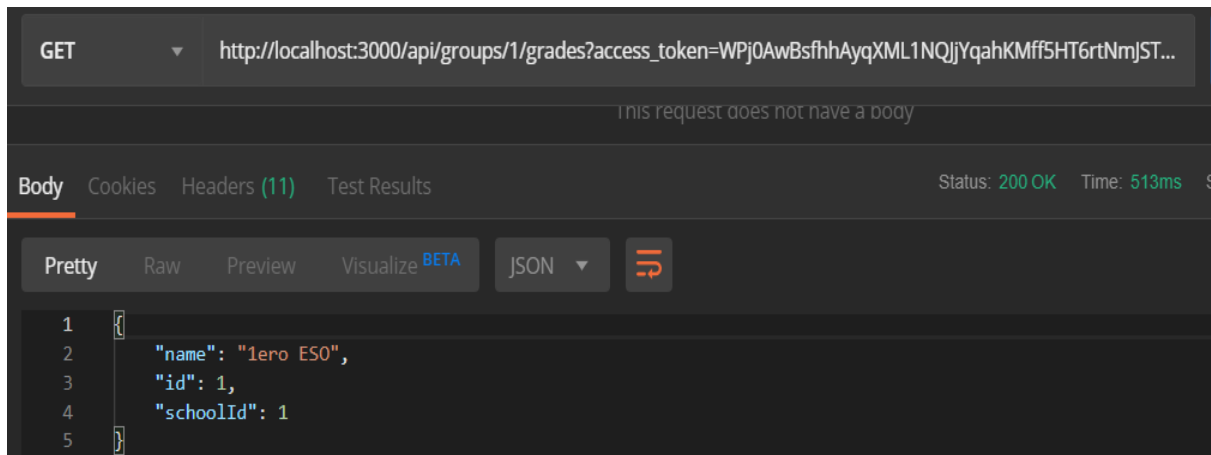


Fig. 25 Rest response of groups grade.

GraphQL Implementation:

We just send a post request where the body contains our query as a string. Even if the query includes child's object the total number of requests is only one time to get all the necessary information to display the /group page.

```
let options: RequestOptions = new RequestOptions({
  headers: this.utilsService.setAuthorizationHeader(new Headers(), this.utilsService.currentUser.id)
});

let url: string;
url = "http://localhost:4000/";
var body = {
  query: `
    groups {
      id
      name
      grade {
        id
        name
      }
      matter {
        name
        id
      }
    }
  `
}

var res = this.http.post(url, body, options);
res.map(r => r.json()).subscribe(res => {
  // // var e1 = JSON.stringify(res);
  console.log(res)
});
```

Fig. 26 Code to fetch data in classip dashboard from graphql server

```
"groups": [
  {
    "id": "1",
    "name": "Group 1",
    "grade": {
      "id": "1",
      "name": "1ero ESO"
    },
    "matter": {
      "name": "Matematicas",
      "id": "1"
    }
  },
  {
    "id": "2",
    "name": "Group 2",
    "grade": {
      "id": "1",
      "name": "1ero ESO"
    },
    "matter": {
      "name": "Catalan",
      "id": "2"
    }
  },
  {
    "id": "3",
    "name": "Group 3",
    "grade": {
      "id": "2",
      "name": "2ndo ESO"
    },
    "matter": {
      "name": "Matematicas",
      "id": "1"
    }
  },
  {
    "id": "4",
    "name": "Group 4",
    "grade": {
      "id": "2",
      "name": "2ndo ESO"
    },
    "matter": {
      "name": "Catalan",
      "id": "2"
    }
  }
]
```

Fig. 27 Query response with all the group information to display the group page

From the figure 26 and 27 we can see that we just send a query with required fields to display the group page and the response is only what we have defined in query filed in just one http request.

Whereas in figure 23 ,24, and 25 we can see that we need to send multiple request to REST api to display the same page and needs to do multiple trip cause just one request is not enough.

8.2. To display /groupStudents/ page:

Figure 28 shows the groupstudents pages of group 1

When a group folder is clicked then it shows the list of students of that group with its student name, last name, email and their avatar image:

Avatar	Id. Estudiant	Nom	Cognom	Email
	10000	Lorena	Diez	student-1@classpip.com
	10001	Rosario	Arellano	student-2@classpip.com
	10002	Gillermo	Macho	student-3@classpip.com
	10004	Mariano	Morales	student-4@classpip.com
	10005	Julia	Rojo	student-5@classpip.com
	10006	Juan	Alfonso	student-6@classpip.com
	10007	Eva	Marchena	student-7@classpip.com
	10008	Paco	Porras	student-8@classpip.com

Fig. 28 GroupStudents pages

Rest implementation:

To show this information we need to call these endpoints:

- a. Get localhost:3000/api/groups/{id}/students where we will have the list of student's information in that particular group.

```

1  {
2    {
3      "name": "Lorena",
4      "surname": "Diez",
5      "profileImage": "https://www.sacs.nsw.edu.au/wp-content/uploads/2018/08/SACS-Student-Profile-Bronte-Thumbnail-1000x1000.jpg",
6      "username": "student-1",
7      "email": "student-1@classpip.com",
8      "emailVerified": true,
9      "id": 10000,
10     "schoolId": 1,
11     "avatarId": 6
12   },
13   {
14     "name": "Rosario",
15     "surname": "Arellano",
16     "profileImage": "http://www.princeofpeace.qld.edu.au/wp-content/uploads/2016/11/Year12Schwartzing-Alexandra.jpg",
17     "username": "student-2",
18     "email": "student-2@classpip.com",
19     "emailVerified": true,
20     "id": 10001,
21     "schoolId": 1,
22     "avatarId": 6
23   }
24 }
    
```

Fig. 29 List of students from REST

Figure 29 is the response of the localhost:3000/api/groups/{id}/students.

b. GET request to localhost:3000/api/students/{studentId}/avatar where we will get the students avatar information like name, id and image URL. To get the full students avatar information we need to call N times the number of students in that group.

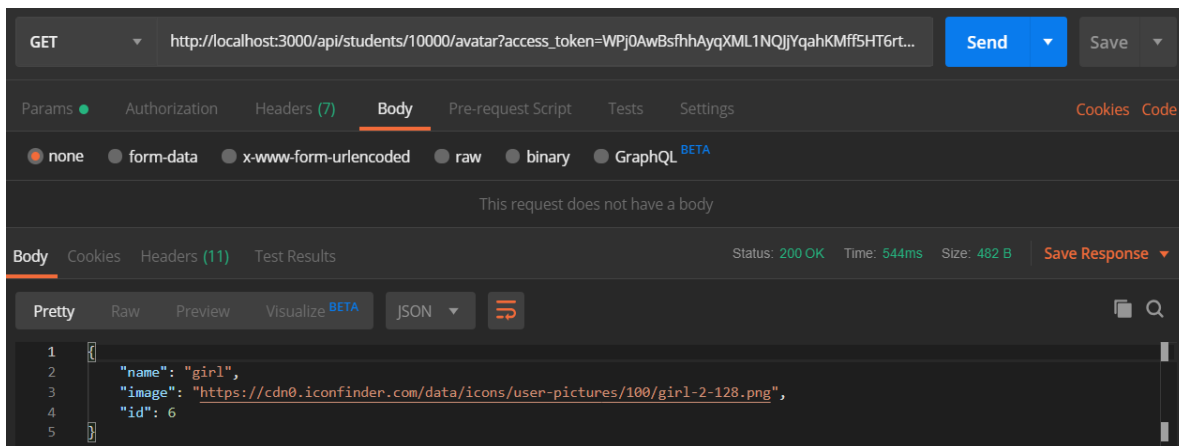


Fig. 30 Avatar information of a student whose id is 10000

Figure 30 show the response of localhost:3000/api/students/{studentId}/avatar

Graphql implementation:

To display the /groupstudents page, we include our query as string in our post request where we have defined required fields to display that page.

```
public getMyGroupStudentsGraphql(id: string): Observable<any> {
  let options: RequestOptions = new RequestOptions({
    headers: this.utilsService.setAuthorizationHeader(new Headers(), this.utilsService.currentUser.id)
  });

  let url: string;
  url = "http://localhost:4000/";
  var body = {
    query: `{ group(where: { id: ${id} }) {
      id
      students {
        id
        name
        surname
        email
        avatar {
          id
          image
        }
      }
    }`
  };

  return this.http.post(url, body, options)
    .catch((error: Response) => this.utilsService.handleAPIError(error));
}
```

Fig. 31 Group query implementation in classPip Dashboard

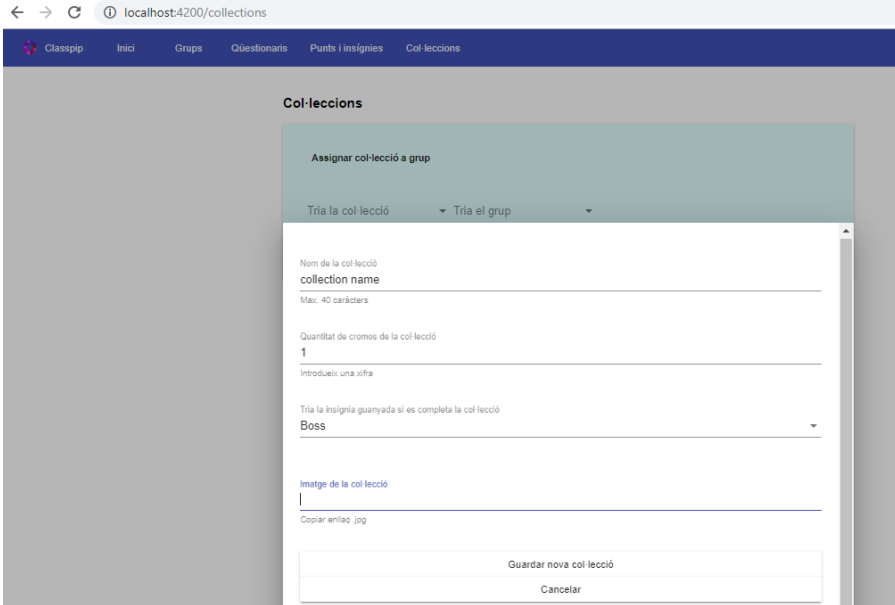
```
"group": {
  "id": "1",
  "students": [
    {
      "id": "10000",
      "name": "Lorena",
      "surname": "Diez",
      "email": "student-1@classpip.com",
      "avatar": {
        "id": "6",
        "image": "https://cdn0.iconfinder.com/data/icons/user-
pictures/100/girl-2-128.png"
      }
    },
    {
      "id": "10001",
      "name": "Rosario",
      "surname": "Arellano",
      "email": "student-2@classpip.com",
      "avatar": {
        "id": "6",
        "image": "https://cdn0.iconfinder.com/data/icons/user-
pictures/100/girl-2-128.png"
      }
    },
    {
      "id": "10002",
      "name": "Gillermo",
      "surname": "Macho",
      "email": "student-3@classpip.com",
      "avatar": {
        "id": "7",
        "image": "https://cdn0.iconfinder.com/data/icons/user-
pictures/100/boy-2-128.png"
      }
    }
  ]
}
```

Fig. 31 Response of the above query

In figure 28 and 29 we can see that we need to send multiple request to REST api to display the groupstudent page and needs to do multiple trip cause just one request is not enough.

From the figure 30 and 31 we can see that we just send a nested query with required fields to display the *groupStudents* page and the response is only what we have defined in query filed in just one http request.

8.1.2. Create new collectionCards



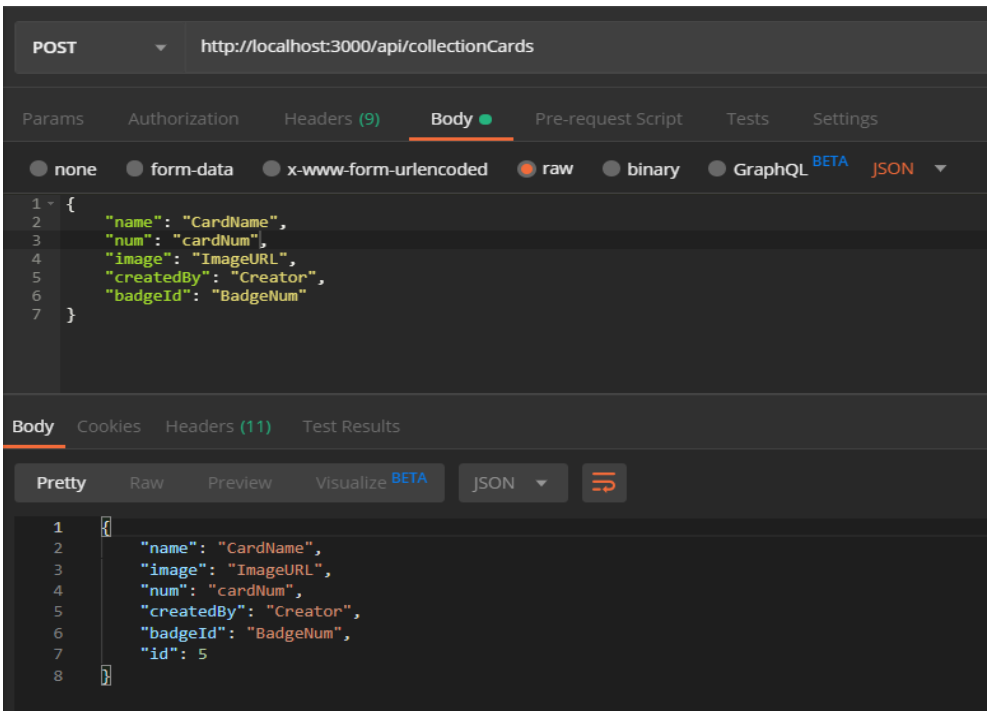
The screenshot shows a web browser at localhost:4200/collections. The page title is 'Col·leccions'. A modal form titled 'Assignar col·lecció a grup' is displayed. It contains two dropdown menus: 'Tria la col·lecció' and 'Tria el grup'. Below these are several input fields: 'Nom de la col·lecció' (collection name) with a maximum of 40 characters, 'Quantitat de cromos de la col·lecció' (quantity of cards) set to 1, 'Tria la insígnia guanyada si es completa la col·lecció' (select a badge) with 'Boss' selected, and 'Imatge de la col·lecció' (collection image) with a 'Copiar enllaç .jpg' link. At the bottom of the form are 'Guardar nova col·lecció' and 'Cancelar' buttons.

Fig.32 Form to create new collection card.

Figure 32 shows the form to create new collection card which was taken from classip dashboard. To create the new collection cards, we have tested one using rest and another using graphql which is shown below:

Rest implementation:

We sent a post request with the information of new collectionCards where in the post body we define the required information about the new cards which is shown in figure 33.



The screenshot shows a REST client interface. The method is 'POST' and the URL is 'http://localhost:3000/api/collectionCards'. The 'Body' tab is selected, showing a JSON request body:

```
1 {
2   "name": "CardName",
3   "num": "cardNum",
4   "image": "ImageURL",
5   "createdBy": "Creator",
6   "badgeId": "BadgeNum"
7 }
```

The 'Body' tab is also selected for the response, showing a JSON response body:

```
1 {
2   "name": "CardName",
3   "image": "ImageURL",
4   "num": "cardNum",
5   "createdBy": "Creator",
6   "badgeId": "BadgeNum",
7   "id": 5
8 }
```

Fig. 33 Post request to create a new collection card

Graphql implementation:

To create the new collection cards using Graphql we send our query in a post body which is shown in the figure 34 where we need to send the mutation with required information to create the new collection card.

```
public GraphQLpostCollection(collectionCard: CollectionCard) {
  let options: RequestOptions = new RequestOptions({
    headers: this.utilsService.setAuthorizationHeader(new Headers(), this.utilsService.currentUser.id)
  });

  let url: string;
  url = "http://localhost:4000/";

  var body = {
    query: `mutation {
      createCollectionCard(data: {
        name: "${collectionCard.name}",
        image: "${collectionCard.image}",
        num: "${collectionCard.num}",
        createdBy: "${collectionCard.createdBy}",
        badgeId: "${collectionCard.badgeId}"
      })
    }
  {
    id
    name
    image
    num
    badgeId
    createdBy
  }
  `
  }

  return this.http.post(url, body, options)
    .catch((error: Response) => this.utilsService.handleAPIError(error));
}
```

Fig. 34 Graphql query implement in classpip

CHAPTER 9. CONCLUSION

As we need different tools to complete certain types of work. GraphQL has its own cool features but it doesn't mean it can be used in every environment. The main idea of using GraphQL in Classip Service which act as a thin layer between REST and the database solves the various problems like over and under fetching, multiple round-trip time but it also comes with problems. It will be perfect for the application where data types are complex and needs to call multiple times the server to work that function.

As the mobile applications are increasing, graphql will be the perfect choice because it reduces the payload and data fetching round trip time.

At the end of this project, I myself learnt how the world of api is working and improved the understanding of different programming languages and their techniques. And time management was the most difficult part of the project . To conclude, I personally feel that this project has been a very good experience and developed my programming skills much better.

References

- [1] The modern graphql Bootcamp(with Node.js and Apollo)
(<https://www.udemy.com/course/graphql-bootcamp/>)
- [2] GraphQL Documentation (<https://graphql.org/learn>)
- [3] Loopback documentation(<https://loopback.io/doc/en/lb4>)
- [4] Angular documentation (<https://angular.io/docs>)
- [5] Migrating to GraphQL: Assessment(<https://arxiv.org/pdf/1906.07535.pdf>)
- [6] GraphQL or Bust(<https://19yw4b240vb03ws8qm25h366-wpengine.netdna-ssl.com/wp-content/uploads/GraphQL-or-Bust-2018.pdf>)
- [7] Prisma documentation (<https://www.prisma.io/docs>)
- [8] What ,why and how graphql
(<https://developer.akamai.com/blog/2019/04/08/graphql-101-what-why-and-how>)

ANEXIOS

Graphql code :

<https://bitbucket.org/chakulama/apigraphql>

Classpip dashboard using graphql

<https://bitbucket.org/chakulama/classpipv0.5.0graphql>

Classpip services

<https://github.com/alejandromartincruz/classpip-services.git>