University of Carthage
National Institute of Applied Sciences and Technology

GraphQL and Hasura

Author
Mohamed Helmi Boudhraa

December 8, 2020

Contents

1 GraphQL's Scheme Definition Language (SDL) **6**
1.1 Definition . . . 6
1.2 Types . . . 6
1.2.1 Object Types . . . 6
1.2.2 Built-in Scalar Types . . . 7
1.3 Enum . . . 7
1.4 Interface . . . 7
1.5 Scheme Directive . . . 8

2 Queries, Mutations and Subscriptions **9**
2.1 Introduction . . . 9
2.2 Queries . . . 9
2.2.1 Get all users . . . 9
2.2.2 Get user by id . . . 10
2.2.3 Get user by field name . . . 10
2.2.4 Get users with fragments . . . 10
2.2.5 Use variables . . . 10
2.3 Mutations . . . 11
2.4 Subscriptions . . . 11

3 GraphQL VS. REST **12**
3.1 Introduction . . . 12
3.2 Comparison Table . . . 12
3.2.1 Overfetching . . . 12
3.2.2 Underfetching . . . 12
3.2.3 Single Endpoint VS. Multiple Endpoints . . . 13

4 Practical case: Hasura **14**
4.1 Introduction . . . 14
4.2 Steps . . . 14
4.2.1 Create a new docker-compose file locally . . . 14
4.2.2 Copy the docker-compose content from GraphQL Github Repository . . . 14
4.2.3 Check the local file variables . . . 15
4.2.4 Run the docker-compose up command . . . 15
4.2.5 Open the console on port 8080 . . . 16
4.2.6 Create a new table: Users . . . 16
4.2.7 Create a new table: Courses . . . 17
4.2.8 Configure Primary and Foreign Keys in Courses . . . 17
4.2.9 Define the relationship between tables . . . 18

4.2.10 Mutation: Adding new users 18
4.2.11 Mutation: Adding new courses 19
4.2.12 Subscription: Subscribing to users 19

List of Figures

1.1 User Type 6
1.2 Presentation of object types 7
1.3 Gender enum 7
1.4 Interface Definition 8
1.5 Applying directive to name 8

2.1 Get the names of all the users 9
2.2 Get name of the user of id 1 10
2.3 Get user ages with name "iyadh" 10
2.4 Get name and age of all users 10
2.5 Set id as a variable that can be an input by the client 10
2.6 Creating a new user 11

3.1 REST Example Endpoint 13
3.2 REST Example Endpoint 2 13
3.3 GraphQL Example Endpoint 13

4.1 File creating command 14
4.2 Github File 14
4.3 Docker-compose local file 15
4.4 Command Running 15
4.5 GraphQL Console 16
4.6 Users Table Creation 16
4.7 Courses Table Creation 17
4.8 Key configuration 17
4.9 Relationship Definition 18
4.10 Adding users 18
4.11 Adding courses 19
4.12 User subscription 19

Introduction

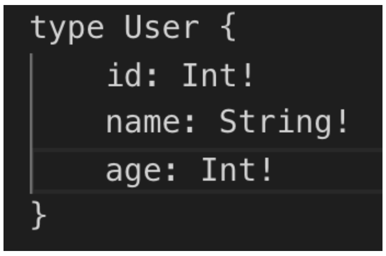
In 2012, Facebook internally developed a new Query Language for APIs and a server-side runtime for executing queries called GraphQL, which was publicly released in 2015 to join the open-source community. Later in 2018, a new foundation, GraphQL Foundation, was established to adopt the project from Facebook, aiming to make it omnipresent across web platforms.
GraphQL's approach to developing APIs is comparable and different in a certain way from other web service architectures such as REST. It enables declarative data fetching, where the client can specify what data he needs from an API, without going through multiple endpoints. And all that by defining a certain structure and a single endpoint of communication.
In this report, we will mainly cover different aspects of GraphQL as follows:
First, we will talk about Schema Definition Language of GraphQL as well as its different components.
Second of all, we will go through CRUD (Create Read Update Delete) and subscription using GraphQL's queries, mutations and subscriptions.
After that, we will illlustrate a high-level comparison between GraphQL and REST to better understand the pros of migrating.
Finally, we will go through a practical example using Hasura GraphQL Engine.

Chapter 1

GraphQL's Scheme Definition Language (SDL)

1.1 Definition

A GraphQL Schema Definition is the most concise way to specify a GraphQL schema. The syntax takes part in the official GraphQL specification as a well-defined structure. [Bur17] The GraphQL schema for a blogging app could be specified like this:



```
type User {
    id: Int!
    name: String!
    age: Int!
}
```

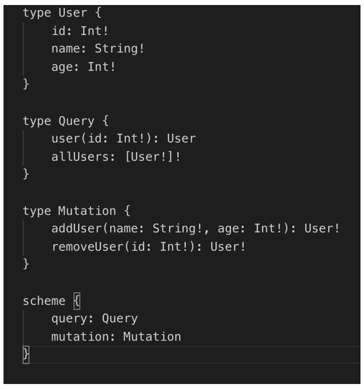
Figure 1.1: User Type

1.2 Types

1.2.1 Object Types

A GraphQL service has specific object types, which are defined with the **type** keyword and start with a capital letter by convention. [All18]
Object types can be the following:

- Query
- Mutation
- Subscription



```
type User {
    id: Int!
    name: String!
    age: Int!
}

type Query {
    user(id: Int!): User
    allUsers: [User!]!
}

type Mutation {
    addUser(name: String!, age: Int!): User!
    removeUser(id: Int!): User!
}

scheme {
    query: Query
    mutation: Mutation
}
```

Figure 1.2: Presentation of object types

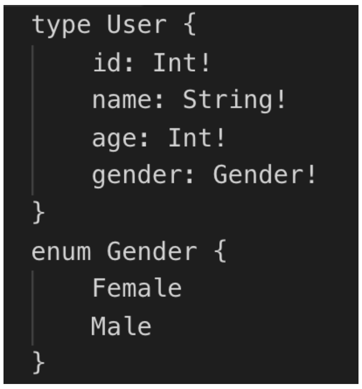
1.2.2 Built-in Scalar Types

GraphQL has 5 built-in scalar types, which are the following:

- Int
- Float
- String
- Boolean
- ID (resolves to a string, but expects a unique value)

1.3 Enum

An enum is a scalar value that has a specified set of possible values. In the example down below, we illustrate a Gender enum to limit the specifications for the user.

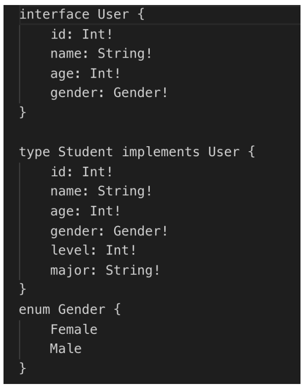


```
type User {
    id: Int!
    name: String!
    age: Int!
    gender: Gender!
}
enum Gender {
    Female
    Male
}
```

Figure 1.3: Gender enum

1.4 Interface

An interface is defined as a list of fields. A GraphQL type must have the same fields as all the interfaces it implements and all interface fields must be of the same type. In the example down below, we present a User interface to be inherited by multiple types of users.



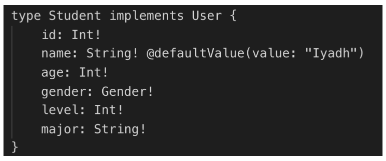
```
interface User {
    id: Int!
    name: String!
    age: Int!
    gender: Gender!
}

type Student implements User {
    id: Int!
    name: String!
    age: Int!
    gender: Gender!
    level: Int!
    major: String!
}
enum Gender {
    Female
    Male
}
```

Figure 1.4: Interface Definition

1.5 Scheme Directive

A directive enables to attach arbitrary information to any element. Directives are always placed behind the element they describe:



```
type Student implements User {
    id: Int!
    name: String! @defaultValue(value: "Iyadh")
    age: Int!
    gender: Gender!
    level: Int!
    major: String!
}
```

Figure 1.5: Applying directive to name

Chapter 2

Queries, Mutations and Subscriptions

2.1 Introduction

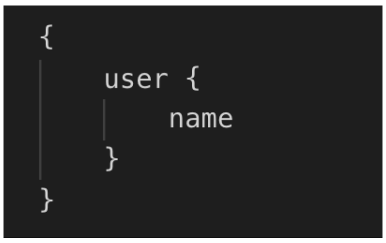
To manage the CRUD in different applications, GraphQL [Tea] offers different functions to manipulate these parts:

- query to fetch data
- mutation to add/update or delete data
- subscription to subscribe (listen) to a certain query

2.2 Queries

Queries in GraphQL are mainly about asking for specific fields on objects. It is completely flexible and lets the client decide what data is actually needed.
Queries enable different arguments to maintain their flexibility. Let's take an example of a user like we saw in the previous section (id, name and gender). We can do the following operations to get data:

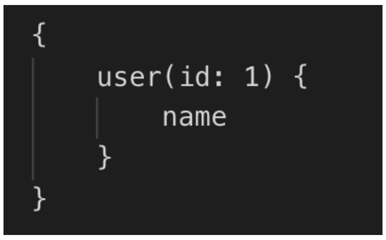
2.2.1 Get all users



```
{
    user {
        name
    }
}
```

Figure 2.1: Get the names of all the users

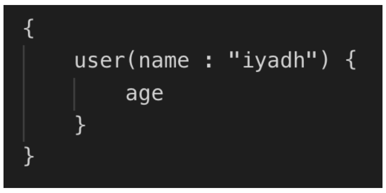
2.2.2 Get user by id



```
{
    user(id: 1) {
        name
    }
}
```

Figure 2.2: Get name of the user of id 1

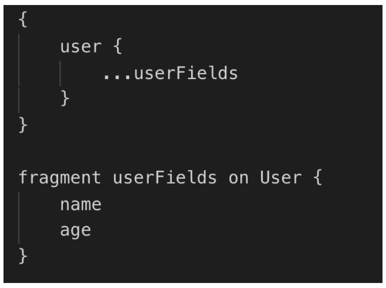
2.2.3 Get user by field name



```
{
    user(name : "iyadh") {
        age
    }
}
```

Figure 2.3: Get user ages with name "iyadh"

2.2.4 Get users with fragments

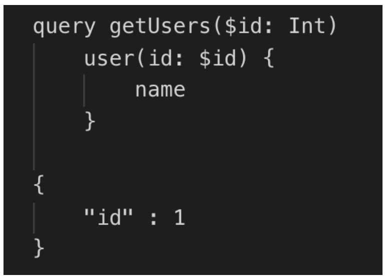


```
{
    user {
        ...userFields
    }
}

fragment userFields on User {
    name
    age
}
```

Figure 2.4: Get name and age of all users

2.2.5 Use variables



```
query getUsers($id: Int)
    user(id: $id) {
        name
    }

{
    "id" : 1
}
```

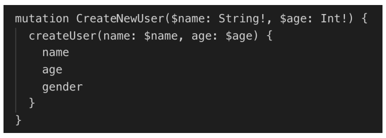
Figure 2.5: Set id as a variable that can be an input by the client

2.3 Mutations

Next to requesting information from a server, the majority of applications also need a way to make changes to the data that's currently stored in the backend. With GraphQL, these changes are made using so-called mutations. There generally are three kinds of mutations:

- creating new data
- updating existing data
- deleting existing data

Mutations follow the same syntactical structure as queries, but they always need to start with the mutation keyword. Here's an example for how we might create a new User:



```
mutation CreateNewUser($name: String!, $age: Int!) {
  createUser(name: $name, age: $age) {
    name
    age
    gender
  }
}
```

Figure 2.6: Creating a new user

2.4 Subscriptions

Another important requirement for many applications is to have a real-time connection to catch important events. For this use case, GraphQL offers the concept of subscriptions.
When a client subscribes to an event, it will initiate and hold a steady connection to the server. Whenever that particular event then actually happens, the server pushes the corresponding data to the client. Unlike queries and mutations that follow a typical "request-response-cycle", subscriptions represent a stream of data sent over to the client.
Subscriptions are written using the same syntax as queries and mutations.

Chapter 3

GraphQL VS. REST

3.1 Introduction

GraphQL and REST [web] diverge in many ways. Thus, in this chapter, we will present the core difference between the two while explaining the concepts of each point.

3.2 Comparison Table

GraphQL	REST
Single Endpoint	Multiple Endpoints
GraphQL Scheme	URL Routes
No Overfetching or Underfetching	Overfetching and Underfetching
Fast Iterations on Front-End	Slow Iterations on Front-End
Fetched via HTTP GET Request via URL	Fetched via HTTP GET Request via URL
Returns JSON Data	Returns JSON Data

Table 3.1: GraphQL VS. REST

3.2.1 Overfetching

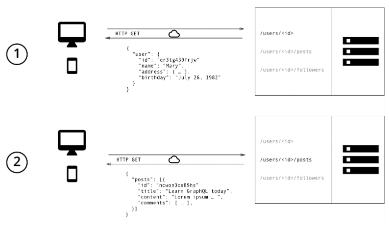
Overfetching implies that more information is accessed by a client than is currently required in the app. For instance, imagine a screen that needs to show a list of users with only their names. This app will usually reach the endpoint of /users in a REST API and receive a JSON array of user data. However this response can provide more information about the users that are returned, such as their birthdays or addresses - information that is irrelevant to the consumer because it only needs to show the names of the users.

3.2.2 Underfetching

Underfetching typically implies that not enough of the information needed is supplied by a particular endpoint. To fetch all it wants, the customer will have to make more requests. This can lead to a situation where a client wants to download a list of components first but then needs to make one additional request per component to retrieve the necessary information. As an example consider that the last three followers per user will also need to be displayed in the same app. The additional endpoint /users/followers is provided by the API. The app would have to make one request to the /users endpoint to be able to view the appropriate data and then reach the /users/followers endpoint for each user.

3.2.3 Single Endpoint VS. Multiple Endpoints

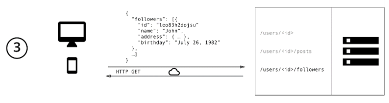
You can usually collect the data with a REST API by accessing several endpoints. In this case, the endpoint could be /users/<id> to fetch the initial user data. Second, there is likely to be an endpoint of /users/<id>/posts that returns all posts to a user. The third endpoint will then be the /users/<id>/followers that returns a list of followers per user.



```
{
  "user": {
    "id": "er3tg439frjw"
    "name": "Mary",
    "address": { … },
    "birthday": "July 26, 1982"
  }
}
```

```
{
  "posts": [{
    "id": "ncwon3ce89hs"
    "title": "Learn GraphQL today",
    "content": "Lorem ipsum … ",
    "comments": [ … ],
  }]
}
```

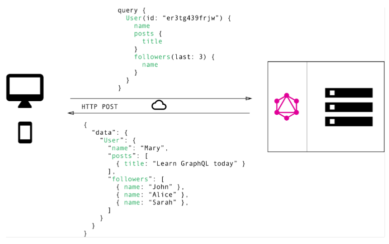
Figure 3.1: REST Example Endpoint



```
{
  "followers": [{
    "id": "leo83h2dojsu"
    "name": "John",
    "address": { … },
    "birthday": "July 26, 1982"
  },
  …]
}
```

Figure 3.2: REST Example Endpoint 2

On the other hand, in GraphQL, you simply send a single query that contains the basic data specifications to the GraphQL server. The server then responds, when these requirements are met, with a JSON object.



```
query {
  User(id: "er3tg439frjw") {
    name
    posts {
      title
    }
    followers(last: 3) {
      name
    }
  }
}
```

```
{
  "data": {
    "User": {
      "name": "Mary",
      "posts": [
        { title: "Learn GraphQL today" }
      ],
      "followers": [
        { name: "John" },
        { name: "Alice" },
        { name: "Sarah" },
      ]
    }
  }
}
```

Figure 3.3: GraphQL Example Endpoint

Chapter 4

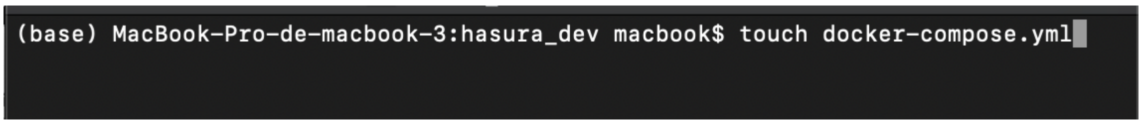
Practical case: Hasura

4.1 Introduction

In this section, we are going to present multiple steps to configure Hasura. Although there are different ways to do so, we chose to use Docker in order to make the installation smoother and easier. [?]

4.2 Steps

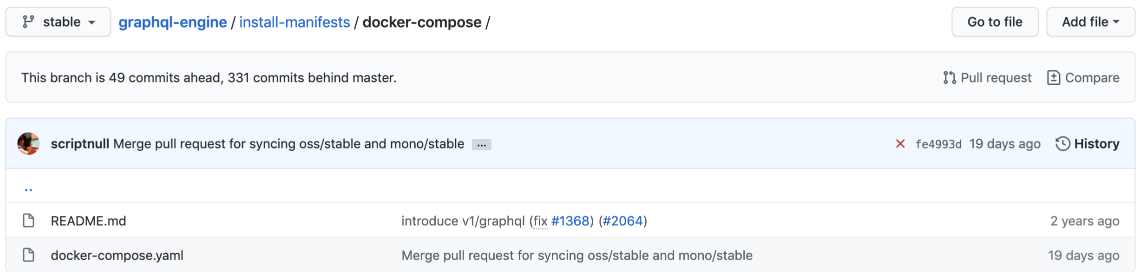
4.2.1 Create a new docker-compose file locally



```
(base) MacBook-Pro-de-macbook-3:hasura_dev macbook$ touch docker-compose.yml
```

Figure 4.1: File creating command

4.2.2 Copy the docker-compose content from GraphQL Github Repository



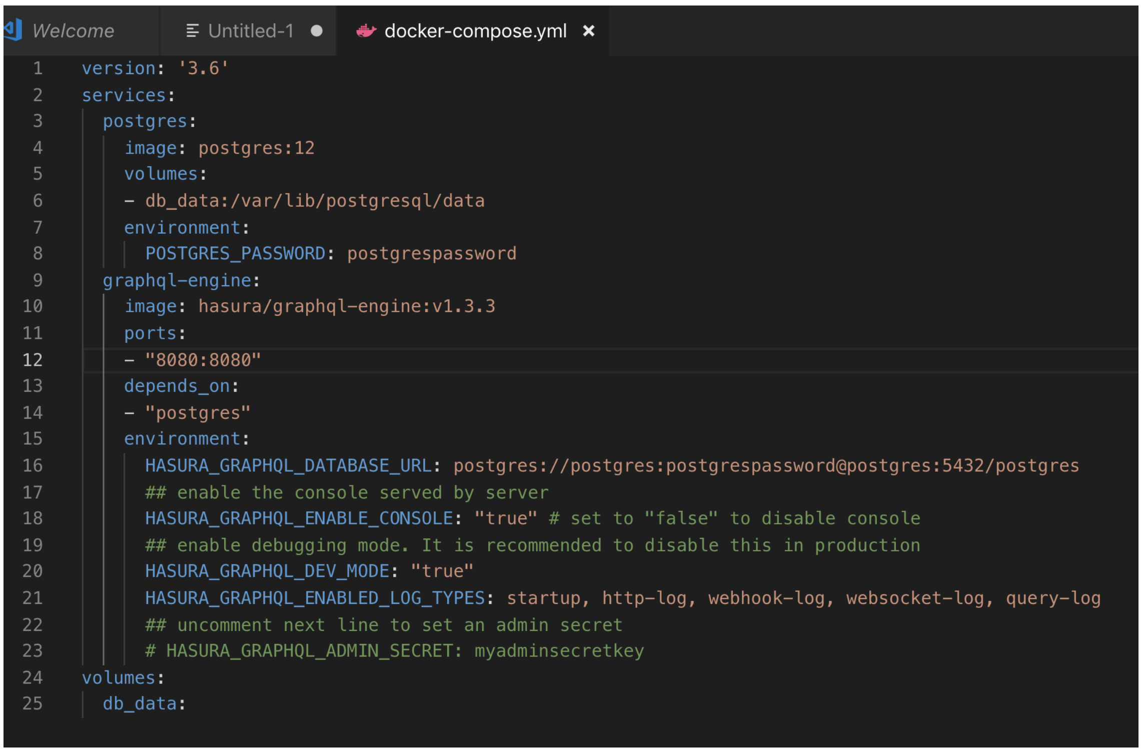
```
stable   graphql-engine / install-manifests / docker-compose /          Go to file   Add file

This branch is 49 commits ahead, 331 commits behind master.            Pull request   Compose

scriptnull Merge pull request for syncing oss/stable and mono/stable ...   × fe4993d 19 days ago   History
..
README.md             introduce v1/graphql (fix #1368) (#2064)                    2 years ago
docker-compose.yaml   Merge pull request for syncing oss/stable and mono/stable   19 days ago
```

Figure 4.2: Github File

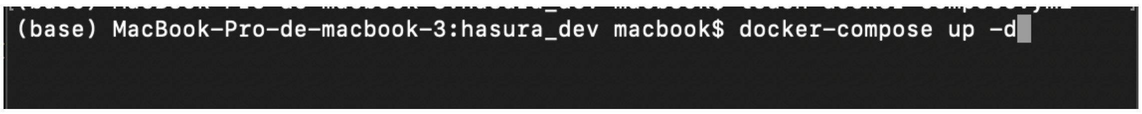
4.2.3 Check the local file variables



```yaml
version: '3.6'
services:
  postgres:
    image: postgres:12
    volumes:
    - db_data:/var/lib/postgresql/data
    environment:
      POSTGRES_PASSWORD: postgrespassword
  graphql-engine:
    image: hasura/graphql-engine:v1.3.3
    ports:
    - "8080:8080"
    depends_on:
    - "postgres"
    environment:
      HASURA_GRAPHQL_DATABASE_URL: postgres://postgres:postgrespassword@postgres:5432/postgres
      ## enable the console served by server
      HASURA_GRAPHQL_ENABLE_CONSOLE: "true" # set to "false" to disable console
      ## enable debugging mode. It is recommended to disable this in production
      HASURA_GRAPHQL_DEV_MODE: "true"
      HASURA_GRAPHQL_ENABLED_LOG_TYPES: startup, http-log, webhook-log, websocket-log, query-log
      ## uncomment next line to set an admin secret
      # HASURA_GRAPHQL_ADMIN_SECRET: myadminsecretkey
volumes:
  db_data:
```

Figure 4.3: Docker-compose local file

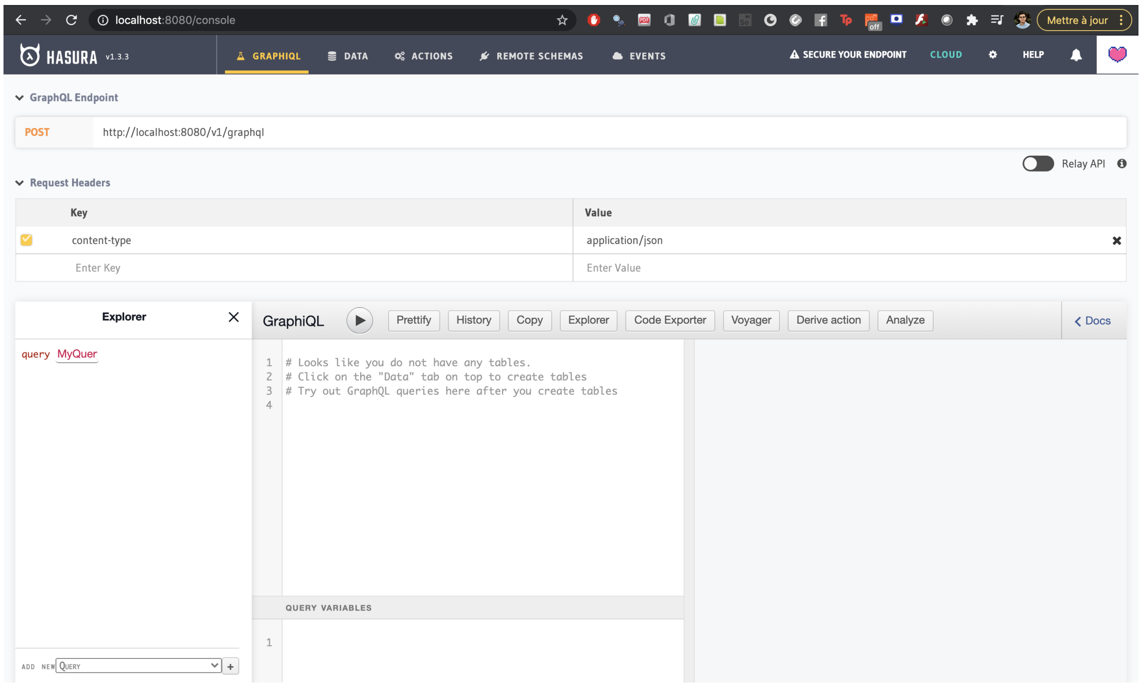
4.2.4 Run the docker-compose up command



```
(base) MacBook-Pro-de-macbook-3:hasura_dev macbook$ docker-compose up -d
```

Figure 4.4: Command Running

4.2.5 Open the console on port 8080



GraphQL Endpoint

POST http://localhost:8080/v1/graphql

Request Headers

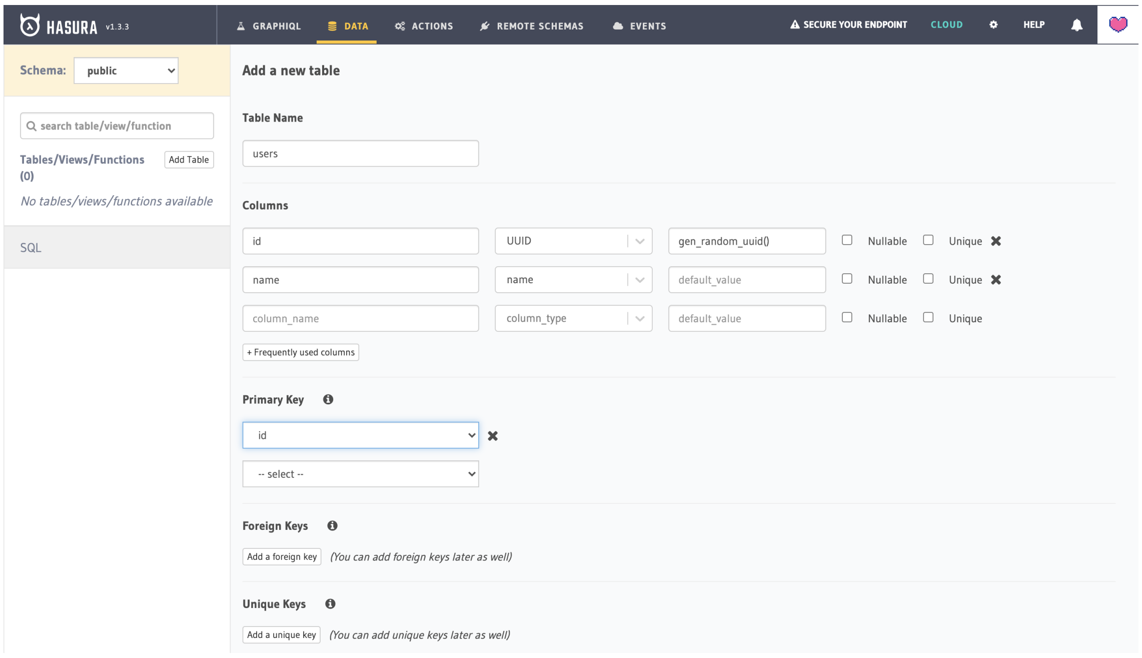
Key	Value
content-type	application/json
Enter Key	Enter Value

query MyQuer

```
# Looks like you do not have any tables.
# Click on the "Data" tab on top to create tables
# Try out GraphQL queries here after you create tables
```

Figure 4.5: GraphQL Console

4.2.6 Create a new table: Users



Schema: public

Add a new table

Table Name

users

Columns

id	UUID	gen_random_uuid()	☐ Nullable	☐ Unique
name	name	default_value	☐ Nullable	☐ Unique
column_name	column_type	default_value	☐ Nullable	☐ Unique

Primary Key

id

-- select --

Foreign Keys

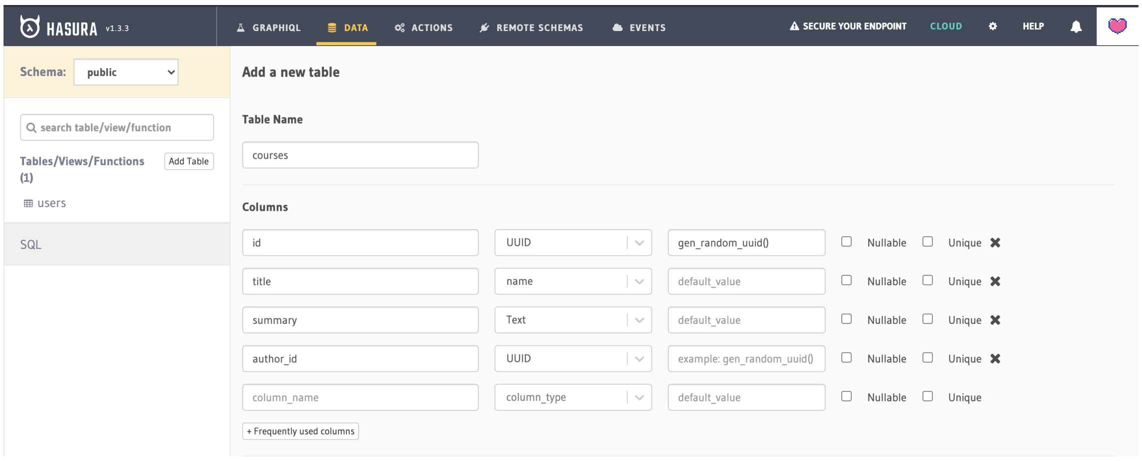
(You can add foreign keys later as well)

Unique Keys

(You can add unique keys later as well)

Figure 4.6: Users Table Creation

4.2.7 Create a new table: Courses



Schema: public

Add a new table

Table Name

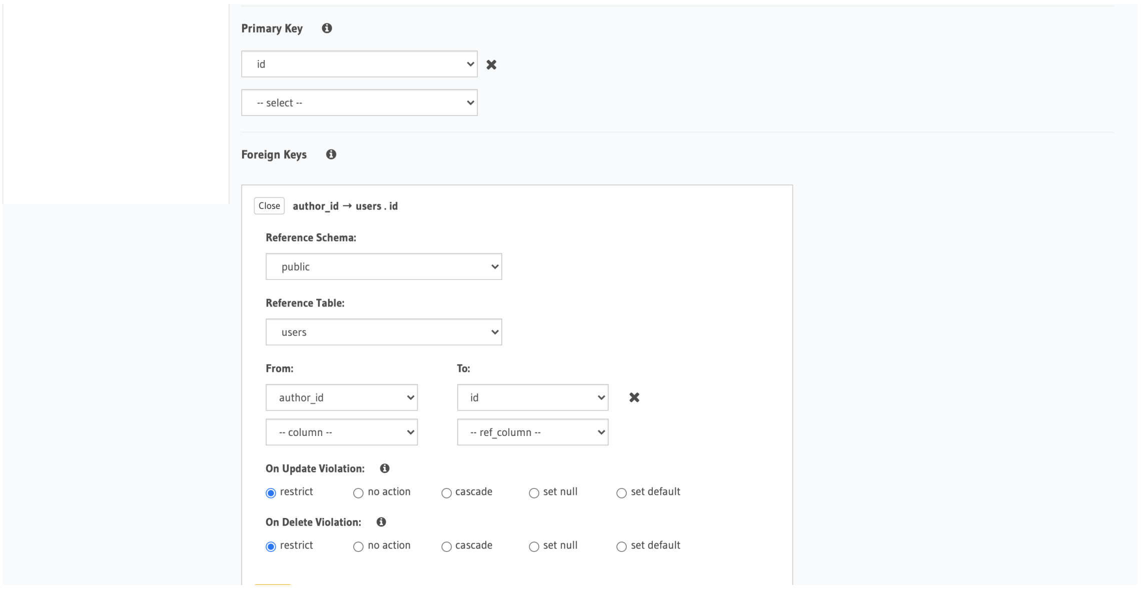
courses

Columns

id	UUID	gen_random_uuid()	☐ Nullable	☐ Unique ✖
title	name	default_value	☐ Nullable	☐ Unique ✖
summary	Text	default_value	☐ Nullable	☐ Unique ✖
author_id	UUID	example: gen_random_uuid()	☐ Nullable	☐ Unique ✖
column_name	column_type	default_value	☐ Nullable	☐ Unique

Figure 4.7: Courses Table Creation

4.2.8 Configure Primary and Foreign Keys in Courses



Primary Key

id ✖

-- select --

Foreign Keys

author_id → users . id

Reference Schema: public

Reference Table: users

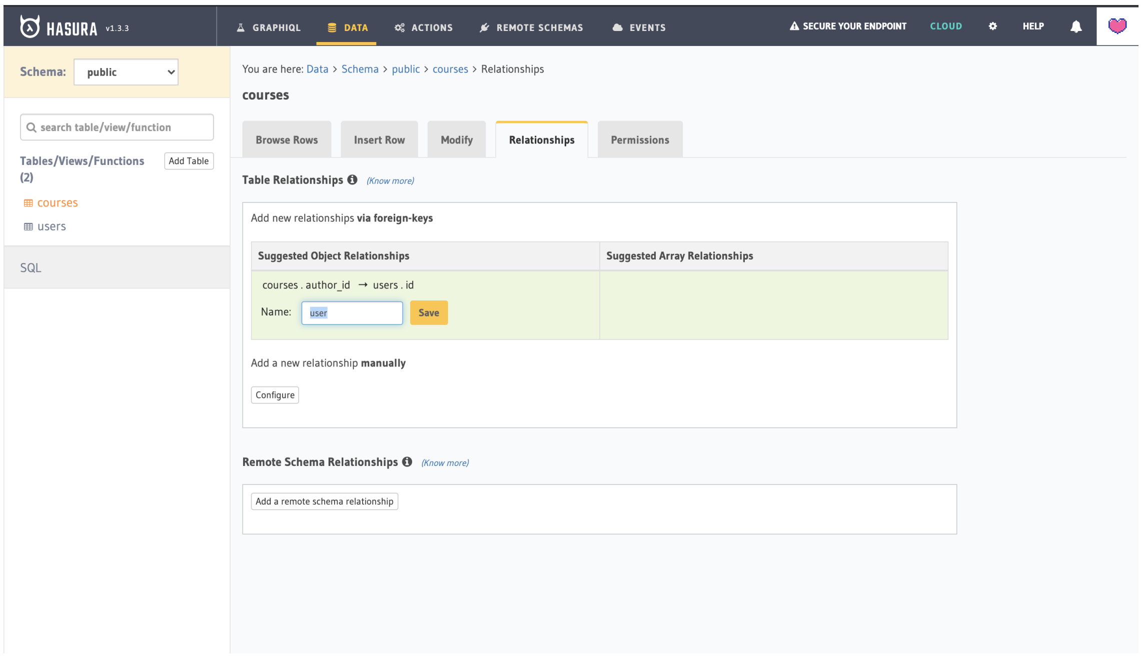
From:	To:
author_id	id ✖
-- column --	-- ref_column --

On Update Violation: [✓ restrict] no action cascade set null set default

On Delete Violation: [✓ restrict] no action cascade set null set default

Figure 4.8: Key configuration

4.2.9 Define the relationship between tables



You are here: Data > Schema > public > courses > Relationships

courses

Table Relationships

Add new relationships **via foreign-keys**

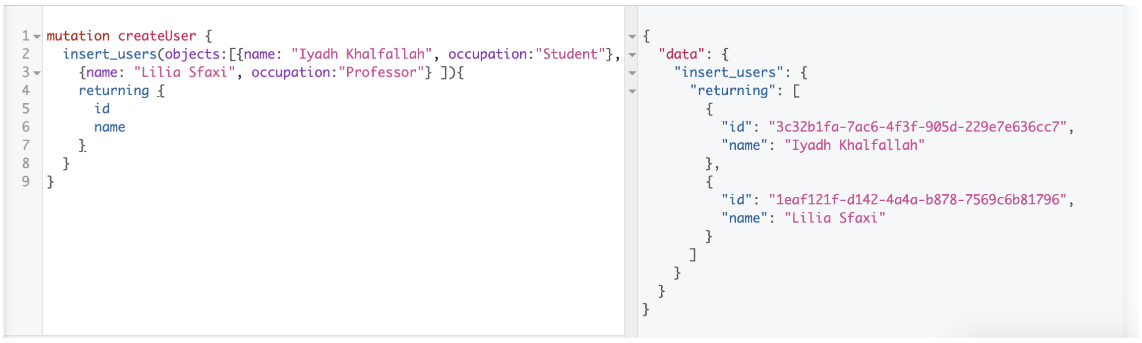
Suggested Object Relationships	Suggested Array Relationships
courses . author_id → users . id Name: user Save	

Add a new relationship **manually**

Remote Schema Relationships

Figure 4.9: Relationship Definition

4.2.10 Mutation: Adding new users

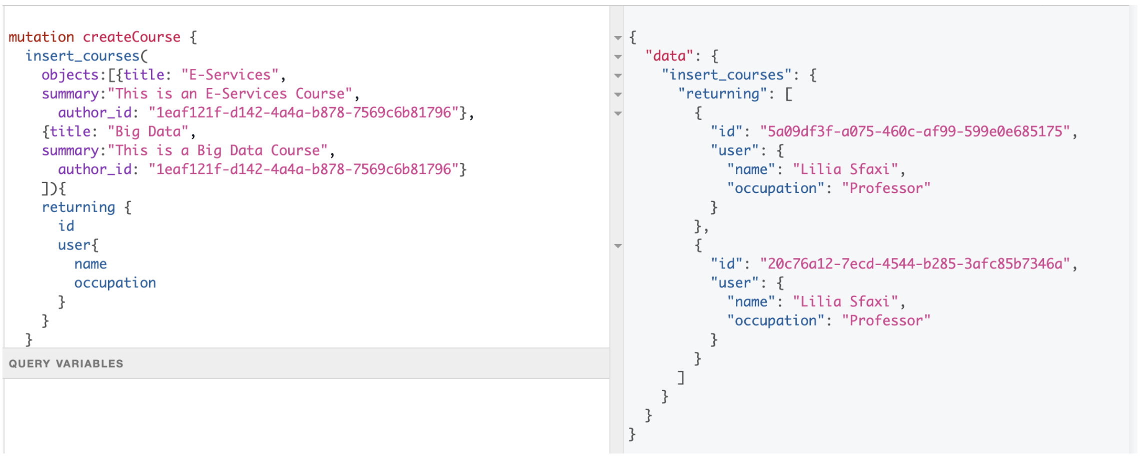


```
mutation createUser {
  insert_users(objects:[{name: "Iyadh Khalfallah", occupation:"Student"},
    {name: "Lilia Sfaxi", occupation:"Professor"} ]){
    returning {
      id
      name
    }
  }
}
```

```
{
  "data": {
    "insert_users": {
      "returning": [
        {
          "id": "3c32b1fa-7ac6-4f3f-905d-229e7e636cc7",
          "name": "Iyadh Khalfallah"
        },
        {
          "id": "1eaf121f-d142-4a4a-b878-7569c6b81796",
          "name": "Lilia Sfaxi"
        }
      ]
    }
  }
}
```

Figure 4.10: Adding users

4.2.11 Mutation: Adding new courses



```
mutation createCourse {
  insert_courses(
    objects:[{title: "E-Services",
    summary:"This is an E-Services Course",
      author_id: "1eaf121f-d142-4a4a-b878-7569c6b81796"},
    {title: "Big Data",
    summary:"This is a Big Data Course",
      author_id: "1eaf121f-d142-4a4a-b878-7569c6b81796"}
    ]){
    returning {
      id
      user{
        name
        occupation
      }
    }
  }
```

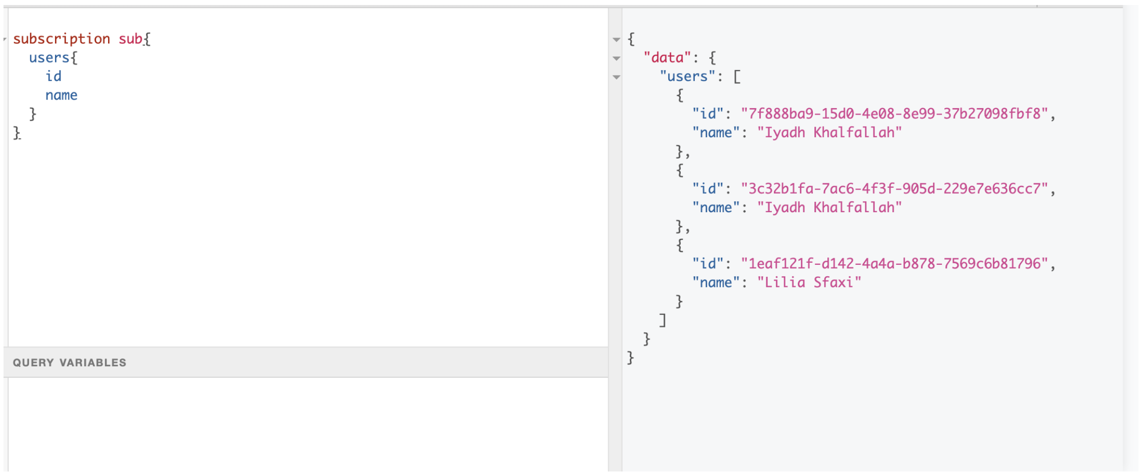
QUERY VARIABLES

```
{
  "data": {
    "insert_courses": {
      "returning": [
        {
          "id": "5a09df3f-a075-460c-af99-599e0e685175",
          "user": {
            "name": "Lilia Sfaxi",
            "occupation": "Professor"
          }
        },
        {
          "id": "20c76a12-7ecd-4544-b285-3afc85b7346a",
          "user": {
            "name": "Lilia Sfaxi",
            "occupation": "Professor"
          }
        }
      ]
    }
  }
}
```

Figure 4.11: Adding courses

4.2.12 Subscription: Subscribing to users

The api-explorer keeps on listening for new users and appends them to the response JSON in real-time.



```
subscription sub{
  users{
    id
    name
  }
}
```

QUERY VARIABLES

```
{
  "data": {
    "users": [
      {
        "id": "7f888ba9-15d0-4e08-8e99-37b27098fbf8",
        "name": "Iyadh Khalfallah"
      },
      {
        "id": "3c32b1fa-7ac6-4f3f-905d-229e7e636cc7",
        "name": "Iyadh Khalfallah"
      },
      {
        "id": "1eaf121f-d142-4a4a-b878-7569c6b81796",
        "name": "Lilia Sfaxi"
      }
    ]
  }
}
```

Figure 4.12: User subscription

Conclusion and perspectives

GraphQL is an increasingly important technology. We can see its magic through different pros compared to REST, from a single endpoint to communicate, to the avoidance on Overfetching and Underfetching. In some papers, it is shown that GraphQL can reduce the size of the JSON documents returned by REST APIs in 94% (in number of fields) and in 99% (in number of bytes) [GB19]. Nevertheless, GraphQL has numerous drawbacks that can harm the backend system in different ways. Thus, before choosing the backend technology, one might study different aspects of his system.

Bibliography

[All18] Alligator.io. A graphql sdl reference. digitalocean.com/community/tutorials/graphql-graphql-sdl, 2018.

[Bur17] Nikolas Burk. Graphql sdl — schema definition language. prisma.io/blog/graphql-sdl-schema-definition-language-6755bcb9ce51, 2017.

[GB19] Marco Tulio Valente ASERG Group Department of Computer Science Federal University of Minas Gerais Brazil Gleison Brito, Thais Mombach. Migrating to graphql: A practical assessment. 2019.

[Tea] GraphQL Team. Introduction to graphql. graphql.org/learn/.

[web]