

GraphQL workshop for NodeJS developers

Iván Corrales Solera ivan.corrales.solera@gmail.com

v0.0.1

Contents

1	Pre-requisites	5
2	Set up your environment	7
2.1	Download the project	7
2.2	Project organization	7
2.3	Running the server	7
2.4	The application	8
3	The GraphQL Playground	13
3.1	Introduction	13
3.2	GraphQL syntax	13
3.3	Challenges	13
4	GraphQL: Objects	17
4.1	Introduction	17
4.2	Code	17
4.3	Challenges	19
5	GraphQL: Operations	21
5.1	Introduction	21
5.2	Code	21
5.3	Challenges	22
6	GraphQL: Interfaces and Unions	23
6.1	Introduction	23
6.2	Fragments	24
6.3	Code	24
6.4	Challenges	25
7	GraphQL: Directives	27
7.1	Introduction	27
7.2	Code	27
7.3	Challenges	28
8	Challenges Solution	29

Chapter 1

Pre-requisites

- Git: Installation guide can be found [here](#).
- Access to Github: Resources for the course are hosted on Github.
- Docker & docker-compose. (docker-compose must support version:“2”).
- Npm installed in your computer.

Chapter 2

Set up your environment

2.1 Download the project

Clone the repository

```
git clone https://github.com/wesovilabs-workshops/workshop-graphql-nodejs.git
cd workshop-graphql-nodejs
```

2.2 Project organization

Project follows the guidelines for structuring a NodeJS project

In this project you will find two separated directories:

2.2.1 Javascript code

- **src**: It contains the source code for the application

2.2.2 Docker and configuration files

- **resources/graphql**: It contains the GraphQL schema implemented by the API.
- **resources/docker-compose**: It contains the docker-compose descriptor and the containers configuration files.
- **resources/local**: It contains default configuration file that will be used when running the server locally..

2.3 Running the server

From the root directory you just need to execute

```
make deploy
```

or in case of you don't have make command installed

```
docker build -t=wesovilabs/workshop-graphql-nodejs:local .;
docker-compose -f resources/docker-compose/docker-compose.yml run --rm -p9001:9001 api
```

To clean the launched containers you just need to perform

```
make docker-stop
```

or

```
docker-compose -f docker-compose/docker-compose.yml down -v
```

2.3.1 While you're coding

The above commands launch the full environment: database and API. On the other hand, when we're coding we could rather launch the database from the docker-compose and the API from our local machine.

1. Launching database container from docker-compose

```
make database
```

or

```
docker-compose -f src/main/resources/docker-compose/docker-compose.yml run --rm -p5456:5432 database
```

2. Run the application from your IDE or by command line

```
npm install;
APP_CONFIG_PATH=./resources/local/config.json NODE_ENV=local npm start;
```

2.4 The application

2.4.1 Database

Databases will be populated with below data when postgres container is launched.

Table 2.1: directors

id	full_name	country
1	Tim Burton	USA
2	James Cameron	Canada
3	Steven Spielberg	USA
4	Martin Scorsese	UK
5	Alfred Hitchcock	USA
6	Clint Eastwood	UK

Table 2.2: actors

id	full_name	country	male
1	Johnny Depp	USA	true
2	Winona Ryder	USA	false
3	Russell Crowe	Australia	true
4	Joaquin Phoenix	USA	true
5	Al Pacino	USA	true

id	full_name	country	male
6	Robert de Niro	USA	true

Table 2.3: movies

id	title	release_year	genre	budget	thriller	director_id
1	Edward Scissorhands	1990	SciFi	20	https://www.yout...	1
2	Gladiator	2000	Drama	103	https://www.yout...	7

Table 2.4: movies_actors

movie_id	actor_id
1	1
1	2
2	3
2	4

2.4.2 API

By default the below operations are already implemented in our project.

2.4.2.1 Queries

- **listDirectors:[Director!]**: It returns the list of directors.
- **listActors:[Actor!]**: It returns the list of actors.
- **listMovies:[Movie!]**: It returns the list of movies.
- **getMovie(movieId:ID!):Movie**: It returns the movie with given id.

2.4.2.2 Mutations

- **addMovie(request:MovieRequest):Movie!**: It adds a new movie.
- **addActor(request:ActorRequest):Actor!**: It adds a new actor.
- **deleteActor("Identifier of the actor" actorId:ID!):[Actor!]**: It deletes the actor with the given id.

2.4.2.3 Subscriptions

- **listenDirectorMovies(directorId:ID!):Movie!**: It open a communication with the server and It is notified when a new movie is created for the directorId in the request.

2.4.3 GraphQL schema

The full GraphQL schema looks

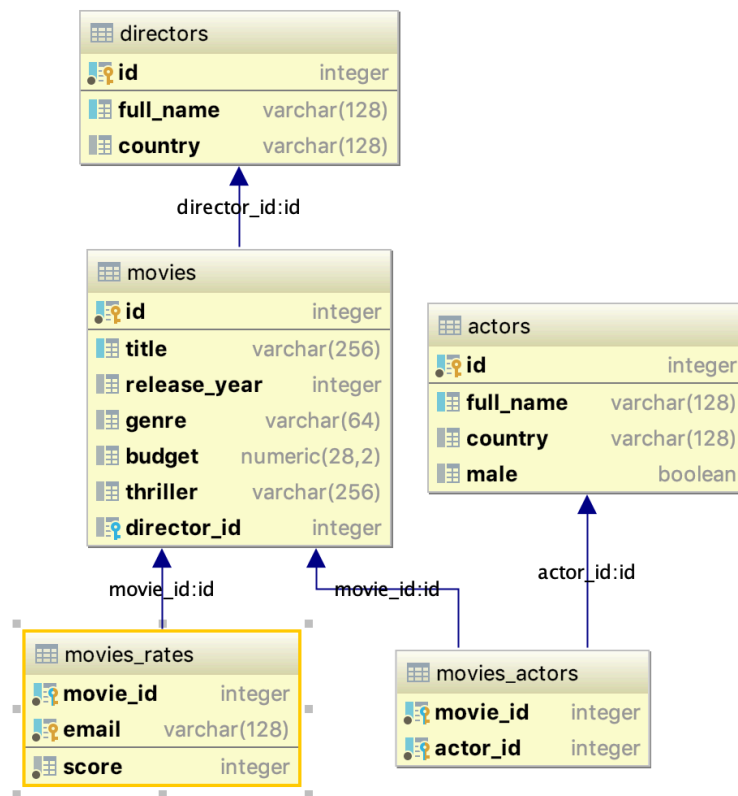


Figure 2.1: Workshop database model

```

schema {
  # The query root of Workshop GraphQL interface.
  query: Query
  # The root query for implementing GraphQL mutations.
  mutation: Mutation
  # The root query for implementing GraphQL subscriptions.
  subscription: Subscription
}

"""Available queries for Workshop API"""
type Query {
  """It returns the list of directors."""
  listDirectors:[Director!]
  """It returns the list of actors."""
  listActors:[Actor!]
  """It returns the list of movies."""
  listMovies:[Movie!]
  """It returns the movie with the given id"""
  getMovie("Movie identifier" movieId:ID!):Movie
}

"""Available mutations for Workshop API"""
type Mutation {
  """I adds a new movie"""
  addMovie(request:MovieRequest):Movie!
  """I adds a new actor"""
  addActor(request:ActorRequest):Actor!
  """I deletes an actor with the given identifier"""
  deleteActor("Identifier of the actor" actorId:ID!):[Actor!]
}

"""Available subscriptions for Workshop API"""
type Subscription {
  """It returns the movies for a given director"""
  listenDirectorMovies(directorId:ID!):Movie!
}

"""Request info for creating a movie"""
input MovieRequest {
  "Name of the movie"
  title: String!
  "Year when the movie was released"
  year: Int
  "Genre for the movie, supported values should be: SciFi, Drama, Comedy or Action"
  genre: String
  "Budget for the movie, the value is provided in Euro"
  budget: Float!
  "URL in which we can watch the thriller of this movie"
  thriller: String
  "Identifier of director"
  directorId: ID!
}

```

```

"""Movie details"""
type Movie {
    "Unique identifier for each movie"
    id: ID!
    "Name of the movie"
    title: String!
    "Year when the movie was released"
    year: Int
    "Genre for the movie, supported values should be: SciFi, Drama, Comedy or Action"
    genre: String
    "Budget for the movie, the value is provided in Euro"
    budget: Float!
    "URL in which we can watch the thriller of this movie"
    thriller: String
    "The director details of the movie"
    director: Director!
    "List of actors for the movie"
    actors("Total of returned actors" total:Int=1): [Actor!]
}

"""Director details"""
type Director{
    "Unique identifier for each director"
    id: ID!
    "Full name of the director"
    fullName: String!
    "Country in which the director was born"
    country: String
}

"""Actor creation info"""
input ActorRequest {
    "Full name of the director"
    fullName: String!
    "Country in which the actor was born"
    country: String
    "Gender of actor: Supported values are male or female"
    gender: String
}

"""Actor details"""
type Actor {
    "Unique identifier for each actor"
    id: ID!
    "Full name of the actor"
    fullName: String!
    "Country in which the actor was born"
    country: String
    "Gender of actor: Supported values are male or female"
    gender: String
}

```

Chapter 3

The GraphQL Playground

3.1 Introduction

Once the server is up and ready we can interact with our API by making use of the **GraphQL Playground**. There are several desktop applications that allow us to run our queries against GraphQL API's. On the other hand, to follow the workshop we will make use of an embedded web client which is deployed within our api.

Just open `http://localhost:9001/graphql` in your browser.

Learn to use it is not rocket science at all. Queries are written on the left side and the result of these is displayed on the right one.

Headers can be entered by clicking the **HTTP Headers** button on the bottom.

To check the **API documentation** we just need to click on the green button on the right (SCHEMA) and a handy menu will be shown.

3.2 GraphQL syntax

On the below picture we can observe how queries, mutations and subscriptions are made from the GraphQL. Bear in mind that you can only run one of these three operations at time. On the other hand you can perform several queries or several mutations at time. Have a look at the examples to understand how queries can be executed.

3.3 Challenges

1. Write a query that returns the below details (**getMovie**)
 - How many actors are shown?
 - Are the returned fields the same that appear in the picture?
2. Create a new director (**addDirector**)
3. Subscribe to the movies for the created director in the previous step. (**listenDirectorMovies**)
4. Open another tab in your GraphQL Playground and add a new movie in which the director is the one that you just created. (**addMovie**).
5. Verify that new movie has been notified to the subscription that we launched in step 3.

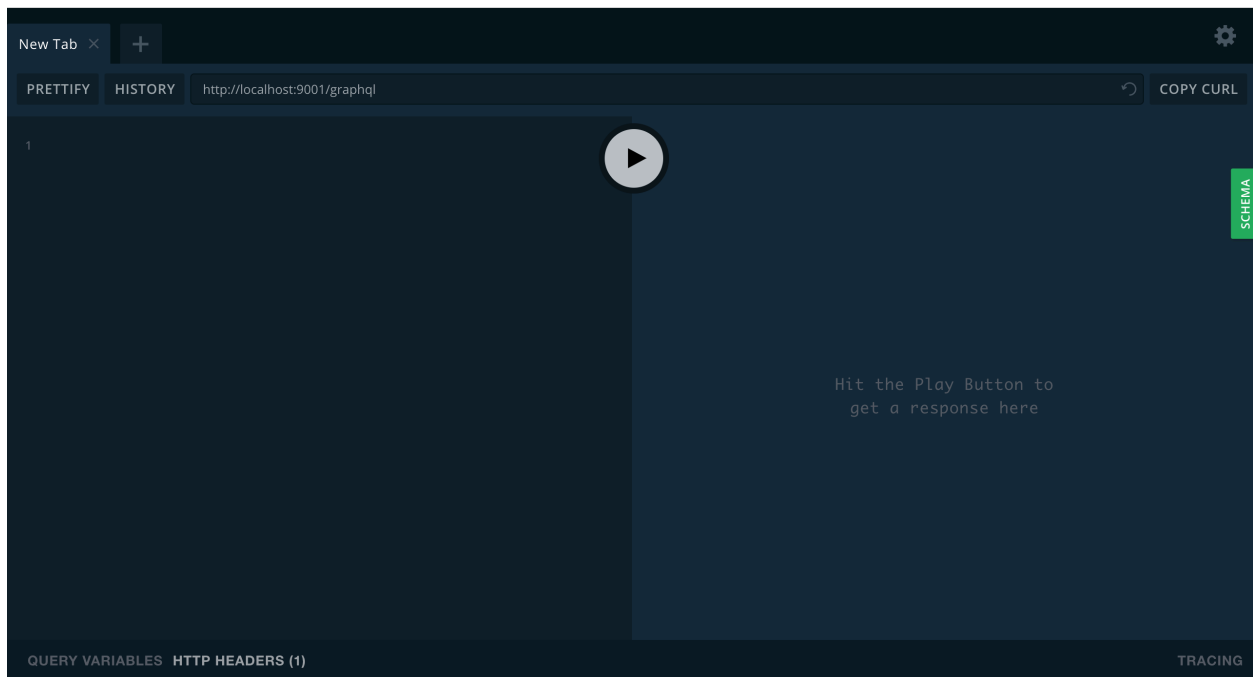


Figure 3.1: GraphQL playground

```
1 query {  
2   actors: listActors {  
3     fullName  
4   }  
5   myMovies: listMovies {  
6     title  
7     director {  
8       fullName  
9     }  
10  }  
11 }
```

Figure 3.2: Queries with alias

```
1 mutation {  
2   addActor(request:{  
3     fullName:"John Travolta"  
4     country:"USA"  
5     gender:"male"  
6   }) {  
7     id  
8     fullName  
9   }  
10 }
```

Figure 3.3: Mutations

```
1 subscription {  
2   listenDirectorMovies(directorId:1){  
3     actors{  
4       fullName  
5     }  
6   }  
7 }
```

Figure 3.4: Subscriptions

```
{
  "data": {
    "getMovie": {
      "id": "1",
      "title": "Edward Scissorhands",
      "year": 1990,
      "budget": 20,
      "thrillerUrl": "https://www.youtube.com/watch?v=M94yyfWy-KI",
      "director": {
        "fullName": "Tim Burton"
      },
      "actors": [
        {
          "fullName": "Johnny Depp",
          "country": "USA"
        },
        {
          "fullName": "Winona Ryder",
          "country": "USA"
        }
      ]
    }
  }
}
```

Figure 3.5: Edward Scissorhands

Chapter 4

GraphQL: Objects

4.1 Introduction

The GraphQL specification includes the following default scalar types: Int, Float, String, Boolean and ID. While this covers most of the use cases, often you need to support custom atomic data types (e.g. Date), or you want a version of an existing type that does some validation. To enable this, GraphQL allows you to define custom scalar types. Enumerations are similar to custom scalars, but their values can only be one of a pre-defined list of strings.

The way to define new scalars or enums in the schema is shown below:

```
scalar MyCustomScalar

enum Direction {
  NORTH
  EAST
  SOUTH
  WEST
}

type MyType {
  myAttribute: MyCustomScalar
  direction: Direction
  ...
}
```

Fields can take arguments as input. These can be used to determine the return value (eg, filtering search results) or to modify the application state. These are known as **field arguments**.

If you have a look at our schema.graphql you can find an example of usage of a field argument for attribute actors in type Movie.

4.2 Code

4.2.1 Resolvers

Let's imagine that we have an operation that returns a Employee type and this type contains an attribute details of type SocialDetails whose information needs to be taken from an external API. And this attribute

won't be always required by the API consumers. Server should not waste time on obtaining something that clients do not need.

resolvers.js

```
export default {
  Person: {
    details: async ({personId}) => {
      return await getDetailsFromLinkedin(personId)
    },
  }
}
```

Now, image that SocialDetails can be taken from more than one social network and we want to permit the consumers to decide which social network must be used. (It's known as field arguments)

schema.graphql.

```
enum Source{
  Linkedin
  Facebook
}

type Person {

  details(source:Source=Linkedin):SocialDetails
}
```

Our resolver could look like this

resolvers.js

```
export default {
  Person: {
    details: async ({personId},{source}) => {
      if (source === 'Linkedin'){
        return await getDetailsFromLinkedin(personId)
      }
      return await getDetailsFromFacebook(personId)
    },
  }
}
```

4.2.2 Scalars

The library **graphql** provides us with class **GraphQLScalarType**. We just need to create a new instance and add it as a new resolver. An example of scalar type is shown below:

scalars.js

```
import {GraphQLScalarType} from 'graphql';

var OddType = new GraphQLScalarType({
  name: 'Odd',
  serialize: oddValue,
  parseValue: oddValue,
  parseLiteral(ast) {
```

```
    if (ast.kind === Kind.INT) {  
      return oddValue(parseInt(ast.value, 10));  
    }  
    return null;  
  }  
});  
  
function oddValue(value) {  
  return value % 2 === 1 ? value : null;  
}
```

schema.js

```
export default makeExecutableSchema({  
  resolvers: {  
    Odd: OddType  
  }  
})
```

You can find more details [here](#)

4.3 Challenges

1. Define an enum type Genre whose values are Drama and SciFi (add as many other as you want) and use it for attribute genre in type Movie and MovieRequest.
2. Define an enum Gender and use it for attribute gender in type Actor.
3. Define a scalar type Url and use it in attribute thriller of types Movie and MovieRequest.
4. Define an enum type Currency whose possible values are Euro and Dollar. Our API must permit the API consumers to decide in which currency they want to obtain attribute budget in type Movie.

Chapter 5

GraphQL: Operations

5.1 Introduction

GraphQL provides us 3 different operations:

- **Queries:** Operation to retrieve data from the server.
- **Mutations:** CUD operations: Create, Update and Delete.
- **Subscriptions:** Create and maintain real time connection to the server. This enables the client to get immediate information about related events. Basically, a client subscribes to an event in the server, and whenever that event occurs, the server sends data to the client.

In our workshop.graphql we will find already implemented operations.

5.2 Code

Actually there's not difference between implement a query or a mutation. We will just implement a function that can retrieve 4 arguments:

- **parent:** The result of the previous resolver call.
- **args:** The arguments of the resolver's field.
- **context:** A custom object each resolver can read from/write to.
- **info:** It contains the query AST and more execution information.

queries.js

```
export const myQuery = (parentValue, args, ctx, info) => {  
  return {  
  
  }  
};
```

mutations.js

```
export const myMutation = (parentValue, args, ctx, info) => {  
  return {  
  
  }  
};
```

Subscriptions looks a little bit different because we need to register to an event.

subscriptions.js

```
export const listenChangesInTeam = {
  subscribe: (
    (_, {teamId}) => {
      return pubsub.asyncIterator(`teams.${teamId}`);
    }
  ),
  resolve: (payload, args, context, info) => {
    return payload;
  }
}
```

5.3 Challenges

1. Implement operations **addActor** and **deleteActor**.
2. Implement operation **rateMovie** that retrieves a new Input type **MovieRateRequest**. **MovieRateRequest** contains the **movieID**, the user email and the score. The operation will persist data into table **movies_rates** and will return the **Movie**.
3. Modify type **Movie** and add a new attribute **rate** whose value is the average score for all the given rates.
4. Modify operation **addMovie**. Add a new attribute **actorsId** (array with the id's of the actors).
5. Define a new query **getMovieRate** that retrieves an argument **movieId** and the output type is **MovieRate**. The output must look like this:

```
{
  "rate": "7",
  "rates": [
    {
      "email": "john.doe@mail.com",
      "score": 8
    },
    {
      "email": "john.doe@mail.com",
      "score": 6
    }
  ]
}
```

5. Create a new subscription **listenRates**. This operation retrieves an argument **movieId** and It displays the new rates for the given **movieId**.

Chapter 6

GraphQL: Interfaces and Unions

6.1 Introduction

An interface exposes a certain set of fields that a type must include to implement the interface.

`schema.graphql`

```
interface Restaurant {
  id:ID!
  name: String!
}

type Indian implements Restaurant{
  id:ID!
  name: String!
  brewedBeer:Boolean!
}

type Burger implements Restaurant{
  id:ID!
  name: String!
  vegetarianOptions: Boolean!
}

type Query{
  listRestaurants: [Restaurant!]
}
```

Unions are identical to interfaces, except that they don't define a common set of fields. Unions are generally preferred over interfaces when the possible types do not share a logical hierarchy.

```
union Item = Food | Electronic | Customer
```

```
type Electronic {
  size: Float
  weight: Float
}

type Food {
  family: String
}
```

```

}

type Customer {
  fullName: String
  zip: String
}

type Query{
  listItems: [Item!]
}

```

6.2 Fragments

Fragments are powerful technique when we are consuming a query that returns an Interface or an Union. They are used to define what attributes we want to obtain from the server depending on the type of the concrete element.

```

query {
  listRestaurants:{
    id
    name
    ... on Indian {
      brewedBeer
    }
    ... on Burger {
      vegetarianOptions
    }
    __typename
  }
}

```

6.3 Code

To implement a new operation with interfaces or unions is easy. We just need to do it as we did in the previous chapter GraphQL: Operations

On the other hand, we need to define new resolvers to make the server understand which kind of inherited type it must return. Below we can find a real example:

resolvers.js

```

export default {
  Url: Url,
  Query:{

  },
  Mutation: {

  },
  Restaurant: {
    __resolveType(restaurant, context, info){
      if(restaurant.brewedBeer){
        return 'Indian';
      }
    }
  }
}

```



```
        }
        return 'Burger'
    },
}
...
}
```

6.4 Challenges

- Define an interface `Person` with common attributes for `Actor` and `Director`. Add a new query `listPeople` that returns a list of people (`[Person!]`).
- Define an union named `Item` that could be a `Movie` or an `Actor`. Add an operation `listItems` that return the full list of `Items`. `[Item!]`

Chapter 7

GraphQL: Directives

7.1 Introduction

A GraphQL schema describes directives which are used to annotate various parts of a GraphQL document as an indicator that they should be evaluated differently by a validator, executor, or client tool such as a code generator. GraphQL implementations should provide the `@isAuthenticated` and `@hasRole` directives. GraphQL implementations that support the type system definition language must provide the `@deprecated` directive if representing deprecated portions of the schema. Directives must only be used in the locations they are declared to belong in. In this example, a directive is defined which can be used to annotate a field: facebook.github.io/graphql

Authorization is a good and common scenario in which we usually will make use of directives. We could control what users are allowed to fetch an object (or even an attribute) from the server.

```
directive @isAuthenticated on FIELD | FIELD_DEFINITION
directive @hasRole(role: String) on FIELD | FIELD_DEFINITION
```

or for clients tools as It was mentioned on the above paragraph.

```
directive @deprecated(
  reason: String = "No longer supported"
) on FIELD_DEFINITION | ENUM_VALUE
```

```
type ExampleType {
  newField: String
  oldField: String @deprecated(reason: "Use `newField`.")
}
```

7.2 Code

From my point of view, this could be the challenging part when implementing a GraphQL API. Have a look at apollographql.com to understand how we can implement our own directives.

Below a very basic example of directive.

directives.js

```
import { SchemaDirectiveVisitor } from "graphql-tools";

class DeprecatedDirective extends SchemaDirectiveVisitor {

  visitFieldDefinition(field) {
    field.isDeprecated = true;
    field.deprecationReason = this.args.reason;
  }

  visitEnumValue(value) {
    value.isDeprecated = true;
    value.deprecationReason = this.args.reason;
  }
}
```

Once we've implemented the directive, we just need to add the directives as shown below:

schema.js

```
export default makeExecutableSchema({
  typeDefs,
  resolvers,
  schemaDirectives: {
    deprecated: DeprecatedDirective,

  },
  ...
});
```

7.3 Challenges

1. Create a directive `@uppercase` that can be assigned to fields. This directive will transform the value of the attribute into uppercase. The directive declaration will look like this

```
directive @uppercase on FIELD_DEFINITION
```

2. Create a directive `@multiply` with an attribute factor. The directive declaration should look like this

```
directive @multiply (
  factor: Int!
) on FIELD_DEFINITION
```

And when the directive is assigned to a field its value will be multiplied by the given factor.

```
input CarRequest {
  km: Int! @multiply(factor:2)
}
```

Chapter 8

Challenges Solution

This workshop follows a story, and you should not start a new chapter if you did not complete the purposed challenges in the previous chapters.

The workshop is completely open source and elaborated with great dedication and effort. So if you are taken the workshop is due to you want to learn GraphQL. That's why I invite you to try to solve all the purposed challenges by yourself.

On the other hand, you could be stuck in one of the chapters. Just in that case, you could checkout the solutions for the challenges.

4. GraphQL: Objects - branch: feature/objects
5. GraphQL: Operations - branch: feature/operations
6. GraphQL: Interfaces and unions - branch: feature/interfaces-unions
7. GraphQL: Directives - branch: feature/directives

Please if you have any doubt contact me at ivan.corrales.solera@gmail.com

- GraphQL Java Quickstart
- GraphQL Specification
- Spring Boot