GraphQL workshop for NodeJS developers

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

v0.0.1

Contents

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 structure

Project follows the guidelines for structuring a NodeJS project

You will find the following directories:

- **src**: It contains the source code for the application
- **resources/graphql**: It contains the GraphQL schema implemented by our server.
- **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 verify that application was launched correctly, just open The GraphQL Playground

To clean the containers you just need to run

```
make docker-stop
```

or

```
docker-compose -f resources/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 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 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 Workshop Application

The application is a movie cataloging tool.

The purpose of this workshop is to enrich the application with new functionality that will be required in upcoming chapters.

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

id	full_name	country	male
5	Al Pacino	USA	true
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

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.
- **addDirector(request:DirectorRequest):Director!**: It adds a new director.
- **deleteDirector("Identifier of the director" direction:ID!):[Director!]**: It deletes the director with the given id.

2.4.2.3 Subscriptions

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

2.4.3 GraphQL schema

The graphql schema for our application looks like this:

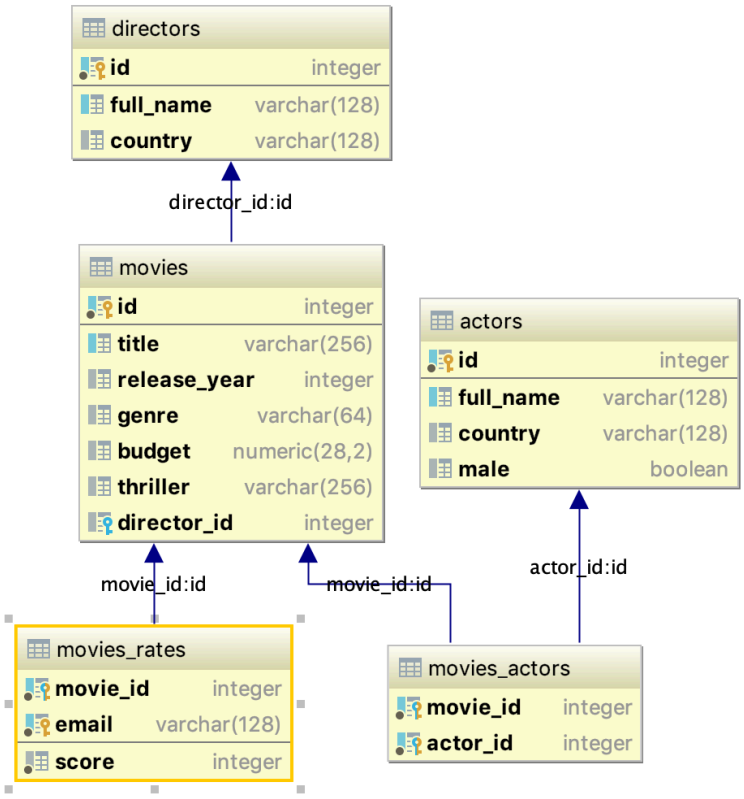

Figure 2.1: Workshop database model

```
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 fiven 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"""
    addDirector(request:DirectorRequest):Director!
    """I deletes the director with the fiven identifier"""
    deleteDirector("Identifier of the director" directorId:ID!):[Director!]
}

"""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
}

"""Director creation request"""
input DirectorRequest{
    "Full name of the director"
    fullName: String!
    "Country in which the director was born"
    country: 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**.

Just open ohttp://localhost:9001/graphql in your browser.

Queries are written on the left side and the response of these are displayed on the right one.

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

GraphQL operations are: queries, mutations and subscriptions. We can run only one operation per request.

On the other hand we can send more than a query or more than a mutation at time.

3.3 Challenges

1. Write a query that returns the below details (**getMovie**) for movie with **id 1**.

- How many actors are returned from the server?
- Are the returned fields the same ones that appear in the picture?

2. Create a new director (**addDirector**)
3. Subscript 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.

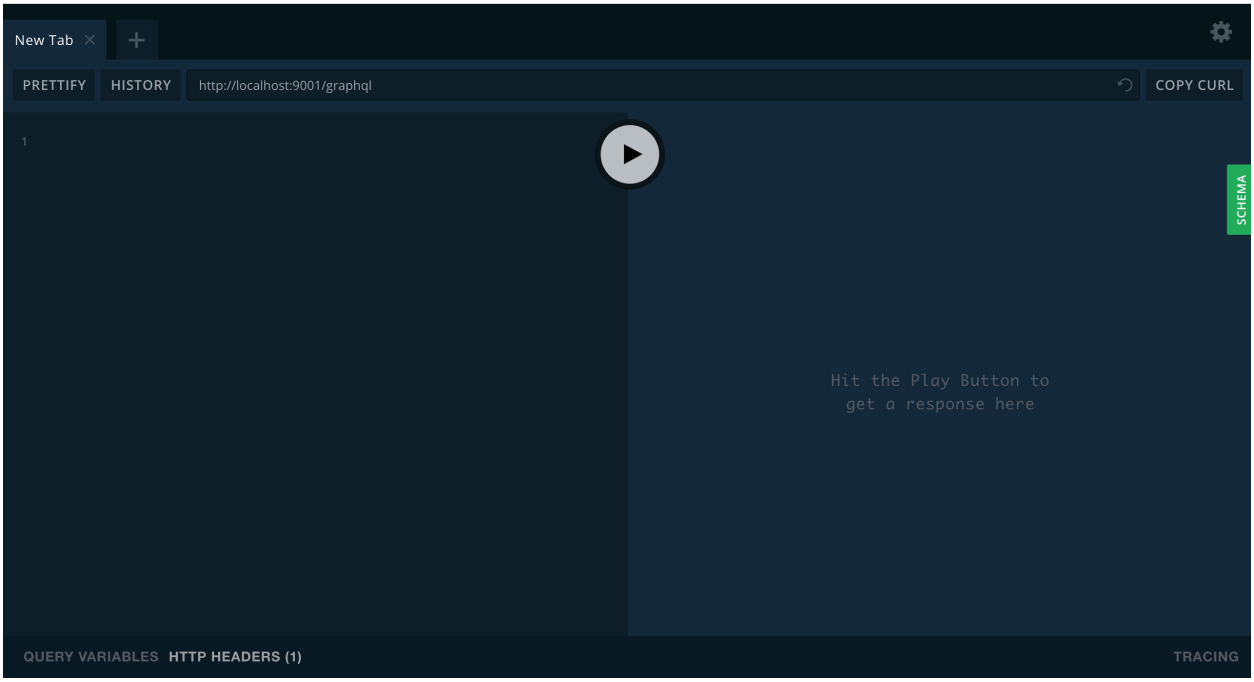

Figure 3.1: GraphQL playground

```
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

```
mutation {
  addActor(request:{
    fullName:"John Travolta"
    country:"USA"
    gender:"male"
  }){
    id
    fullName
  }
}
```

Figure 3.3: Mutations

```
subscription {
  listenDirectorMovies(directorId:1){
    actors{
      fullName
    }
  }
}
```

Figure 3.4: Subscriptions

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 customize the return value (eg, filtering search results). This is known as **field argument**.

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. The total argument is used to define the max number of actors returned from the server.

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.

src/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.

resources/graphql/schema.graphql.

```
enum Source{
    Linkedin
    Facebook
}
type Person {

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

Our resolver would look like this example:

src/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 an add it as a new resolver. An example of scalar type is shown below:

src/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;
}
```

src/resolvers.js

```
export default {
    Odd: OddType,
}
```

Have a look here to learn more about how to implement custom scalars with gaphql-js.

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.

Run this query to verify your implementation works as expected

```
mutation {
  addMovie (request:{
    title: "Corpse Bride"
    year: 2005
    budget: 35000000
    directorId: 1
    genre: SciFi
    thriller: "https://www.youtube.com/watch?v=o5qOjhD8j08"
  }){
    id
    director{
      fullName
      country
    }
    genre
  }
}
```

2. Define an enum Gender and use it for attribute gender in type Actor.

Run this query to verify your implementation works as expected

```
query {
```

```
  listActors{
    fullName
    gender
  }
}
```

3. Define a scalar type Url and use it in attribute thriller of Movie and MovieRequest. Only valid url's should be permitted.

This code could be useful for validate the url's

```
export const validateUrl = (str) => {
    let pattern = new RegExp('^(https?:\\/\\/)' + // protocol
        '((([a-z\\d]([a-z\\d-]*[a-z\\d])*)\\.?)+[a-z]{2,}|' + // domain name
        '((\\d{1,3}\\.){3}\\d{1,3}))' + // OR ip (v4) address
        '(\\:\\d+)?(\\/[-a-z\\d%_.~+]*)*' + // port and path
        '(\\?[;&a-z\\d%_.~+=-]*)?' + // query string
        '(\\#[-a-z\\d_]*)?$', 'i'); // fragment locator
    return pattern.test(str);
}
```

Run this query to verify that only valid url's are permitted. Actually, the movie should not be saved into the database.

```
mutation {
  addMovie (request:{
    title: "Gran Torino"
    year: 2009
    budget: 28000000
    directorId: 6
    genre: Drama
    thriller: ".http"
  }){
    id
    director{
      fullName
      country
    }
    genre
    thriller
  }
}
```

Run this query to verify that your scalar Url works as expected.

```
mutation {
  addMovie (request:{
    title: "Gran Torino"
    year: 2009
    budget: 28000000
    directorId: 6
    genre: Drama
    thriller: "https://www.youtube.com/watch?v=9ecW-d-CBPc"
  }){
    id
    director{
      fullName
```

```
      country
    }
    genre
    thriller
  }
}
```

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. **1€ => 1.14$**

Run this query

```
query {
  getMovie(movieId:1){
    budgetInEuros: budget(currency:Euro)
    budgetInDollars: budget(currency:Dollar)
  }
}
```

and verify that the output should be this:

```
{
  "data": {
    "getMovie": {
      "budgetInEuros": 20,
      "budgetInDollars": 22.799999999999997
    }
  }
}
```

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 ocurrs, 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 implament 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.

src/queries.js

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

    }
};
```

src/mutations.js

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

    }
};
```

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

src/subscriptions.js

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

Have a look at already implemented queries,mutations and subscriptions in this project.

5.3 Challenges

1. Implement operations **addActor** and **deleteActor**.

Run these queries to verify they work as expected.

```
mutation {
  addActor(request:{
    fullName: "Penelope Cruz"
    gender:female
    country:"Spain"
  }){
    id
  }
}
```

```
mutation {
  deleteActor(actorId:7){
    id
    fullName
  }
}
```

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.

(Email must be a new scalar type)

```
input MovieRateRequest {
    movieId:ID!
    email:Email!
    score:Int!
}
type Mutation {
    ...
    rateMovie(request:MovieRateRequest!):Movie!
    ...
}
```

Run this query to verify it works as expected

```
mutation {
  rateMovie(request:{
    movieId:1
    email: "thisisme@mail.com"
    score: 8
  }){
    title
  }
}
```

3. Modify type Movie and add a new attribute rate whose value is the average score for all the given rates.

```
mutation {
  rateMovie(request:{
    movieId:1
    email: "thisisme2@mail.com"
    score: 6
  }){
    title
    rate
  }
}
```

The returned attribute **rate** must be the average.

4. Modify operation addMovie. Add a new attribute actorsId (array with the id's of the actors).

Run this query

```
mutation {
  addMovie(request:{
    title: "The Irishman"
    year: 2019
    budget:130
    genre: Action
    actorsId: [5,6]
    directorId: 4
  }){
    director{
      fullName
    }
    actors(total:5){
      id
      fullName
    }
  }
}
```

and the output should be

```
{
  "data": {
    "addMovie": {
      "title": "The Irishman",
      "director": {
```

```
        "fullName": "Martin Scorsese"
      },
      "actors": [
        {
          "id": "5",
          "fullName": "Al Pacino"
        },
        {
          "id": "6",
          "fullName": "Robert de Niro"
        }
      ]
    }
  }
}
```

5. Define a new query **getMovieRate** that retrieves an argument movieId and the output type is MovieRate.

Run this query

```
query{
  getMovieRate(movieId:1){
    rate
    rates{
      email
      score
    }
  }
}
```

and the output should look similar to 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.

From one tab we run the subscription

```
subscription{
  listenRates(movieId:1){
    title
    rate
  }
```

```
}
```

and then we add a new rate for movie with identifier 1

```
mutation{
  rateMovie(request:{
    movieId: 1
    email:"yo@mail2.com"
    score:3
  }){
    director{
      id
      fullName
    }
  }
}
```

The subscription should print the details for movie with id 1

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 commons attributes for Actor and Director. Add a new query listPeople that returns a list of people ([Person!]).

```
query{
    listPeople:[Person!]
}
```

Once you've implemented this query make use of fragments to return the below details

```
{
  "data": {
    "listPeople": [
      {
        "__typename": "Actor",
        "fullName": "Johnny Depp",
        "gender": "female"
      },
      ...
      {
          "__typename": "Director",
          "fullName": "Steven Spielberg",
          "country": "USA"
       }
       ...
    ]
  }
}
```

- Define an union named Item that could be a Movie or an Actor. Add an operations listItems that return the full list of Items. [Item!]

```
query{
    listItems:[Item!]
}
```

Once you've implemented this query make use of fragments to return the below details

```
{
  "data": {
    "listItems": [
      {
        "__typename": "Movie",
        "title": "Edward Scissorhands"
      },
       {
         "__typename": "Actor",
          "fullName": "Russell Crowe"
```

```
        }
        ...
    ]
  }
}
```

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 **?** and **?** directives. GraphQL implementations that support the type system definition language must provide the **?** 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 allowd 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

When defining a directive we need to define its scope. Have a look at apollographql.com to understand how to 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:

src/schema.js

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

    },
    ...
});
```

7.3 Challenges

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

```
directive @uppercase on FIELD_DEFINITION

"""Movie details"""
type Movie {
    ...
    title: String! @uppercase
    ...
}
```

To verify the directive was implemented correctly we can run

```
query {
  listMovies{
    title
  }
}
```

and check that the title for all the movies was returned in uppercase.

```
{
  "data": {
```

```
    "listMovies": [
      {
        "title": "EDWARD SCISSORHANDS"
      },
      {
        "title": "GLADIATOR"
      }
    ]
  }
}
```

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

- Apollo GraphQL
- GraphQL Specification