GraphQL workshop for Java 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”).
- Gradle correctly installed in your computer. Installation guide

Chapter 2

Set up your environment

2.1 Download the project

The workshop can be followed in: NodeJS. (Java, Python, Go are coming soon).

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

2.2 Project organization

Project follows the guidelines for structuring a Java project with Gradle.

2.2.1 ./src/main/java

- **com.wesovilabs.workshops.graphql.domain**: It contains the classes that represent the GraphQL domain model.
- **com.wesovilabs.workshops.graphql.database**: This package contains two sub-packages model and repository. The first onw contains the database model and the second one the DAO layer.
- **com.wesovilabs.workshops.graphql.converter**: This package contains classes that will help in order to make transformation between objects. Basically for transforming the GraphQL domain objects into entities that will be persisted into the database.
- **com.wesovilabs.workshops.graphql.publisher**: Pub-sub implementations that will be used by subscription operations.
- **com.wesovilabs.workshops.graphql.resolver**: It contains the resolver for our application: queries, mutations and subscriptions but also the resolver of the output types when it is required.
- **com.wesovilabs.workshops.graphql.service**: This is the business layer which is call from the resolver package and It delegates the request to the repository.
- **com.wesovilabs.workshops.graphql.directive**: Empty package that will be used during the workshop.
- **com.wesovilabs.workshops.graphql.scalae**: Empty package that will be used during the workshop.

2.2.2 ./src/main/resources

- **workshop.graphqls**: GraphQL schema that is implemented by our application.
- **application.yaml**: Default configuration file used by spring-boot when running the application locally.
- **docker/Dockerfile**: Docker descriptor for out application.
- **docker-compose/docker-compose.yml**: Descriptor for launching bot database and our application.
- **docker-compose/api**: Configuration files used by api container.
- **docker-compose/postgres**: Configuration files used by postgres container.

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

```
gradle build;
docker build -f src/main/resources/docker/Dockerfile -t=wesovilabs/workshop-graphql-java:local .;
docker-compose -f docker-compose/docker-compose.yml -run -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
```

2.4 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

id	full_name	country	male
2	Winona Ryder	USA	false
3	Russell Crowe	Australia	true
4	Joaquin Phoenix	USA	true
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

By defualt 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.

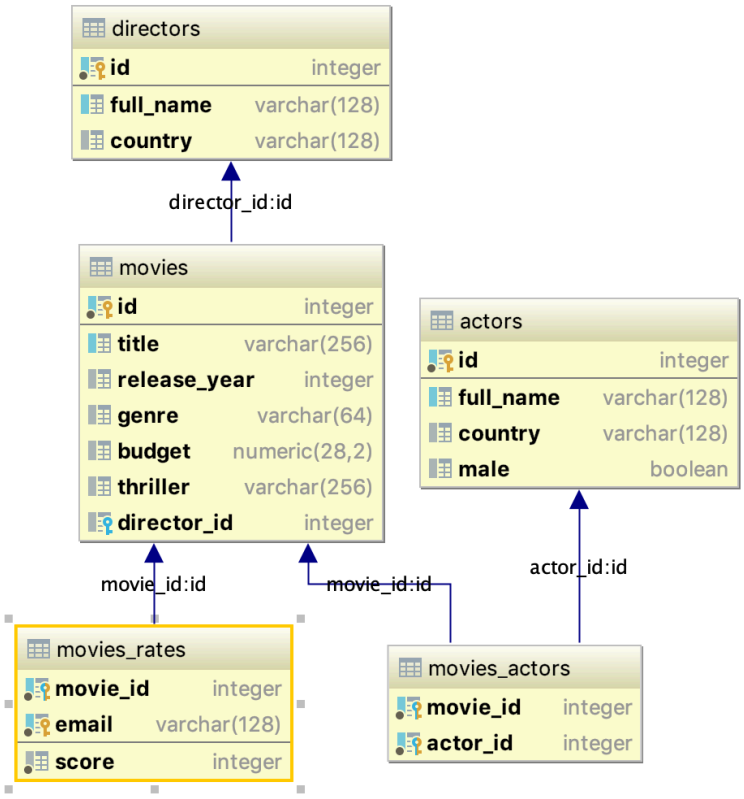

Figure 2.1: Workshop database model

2.4.3 GraphQL schema

The full GraphQL schema looks

```
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"""
    addActor(request:ActorRequest):Actor!
    """I deletes an actor with the fiven 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 out api.

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

Learn to use Graphiql is not rock science. We will write our queries on the left panel and the result will be displayed on the right panel.

To check the **API documentation** we just need to click on button on the right (< Docs) 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 Graphiql. Beer in mind that you can only run one ot 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 cna eb executed.

3.3 Challenges

1. Write a query that returns the below details (**getMovie**)

- How many actors are shown?
- The returned fields in the picture are the same that the returned by your query?

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.

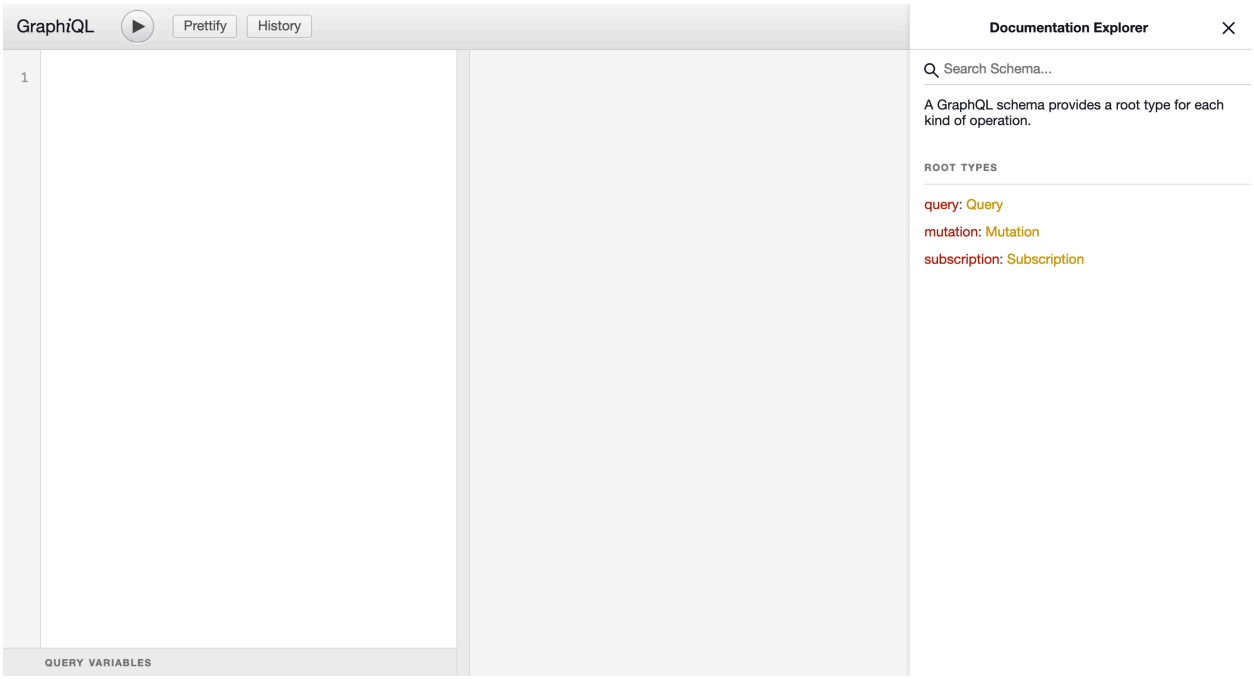

Figure 3.1: GraphQL playground

```
query {
  actors: listActors {
    fullName
  }
  myMovies:listMovies {
    title
    director{
      fullName
    }
  }
}
```

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

```
{
  "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 Enum

In the introduction we see how to define a enum type in the schema. In the code we just need to create an enum type with the same name.

Direction.java (This file should be in package domain)

```
public enum Direction {
    NORTH,
    EAST,
    SOUTH,
    WEST
}
```

4.2.2 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.

PersonResolver.js (This file should be in package resolver)

```
@Component
public class PersonResolver implements GraphQLResolver<Person> {

    public SocialDetails details(Person person) {
        /**
        *  We will invoke Linkedin API to obtain the info.
        **/
        SocialDetails details = linkedinAPI.getDetals(person.getId());
        return details;
    }

}
```

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.graphqls.

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

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

Our resolver could look like this

Source.java (This file should be in package domain)

```
public enum Source {
    Facebook,
    Linkedin
}
```

SouPersonResolverrce.java (This file should be in package resolver)

```
@Component
public class PersonResolver implements GraphQLResolver<Person> {
```

```
    public SocialDetails details(Person person, Source source) {
        if (source==Source.Linkedin){
            SocialDetails details = linkedinAPI.getDetals(person.getId());
            return details;
        }
        if (source==Source.Facebook){
            SocialDetails details = facebookAPI.getDetals(person.getId());
            return details;
        }
    }

}
```

4.2.3 Scalars

The java-graphql library that we use provides with class `GraphQLScalarType` that we need to extend to define our own scalars.

Scalar documentation can be found here

Keep in mind that our application works with Spring, so the code could be a little bit different than the examples in the above documentation.

Below we can see the implementation for a scalar type that only permit odd numbers.

OddScalar.js (This file should be in package scalar)

```
@Component
public class OddScalar extends GraphQLScalarType {


    public OddScalar() {
        /**
        * args[0] Scalar name: It must be the one defined in the GraphQL schema
        * args[1] Scalar description: A brief description for our scalar type
        * args[2] A Coercing instance that we define below
        */
        super("Odd", "Odd scalar", coercing);
    }

    private static final Coercing coercing = new Coercing<Object, Object>() {
        @Override
        public Object serialize(Object input) {
             if (input instanceof Integer) {
                if (result % 2 !=0){
                    throw new CoercingSerializeException(
                        "It's not a valid odd number."
                    );
                }
                return result;
            }
            throw new CoercingSerializeException(
                     "Expected type 'Int' but was other."
            );
```

```
        }

        @Override
        public Object parseValue(Object input) {
            return serialize(input);
        }

        @Override
        public Object parseLiteral(Object input) {
           if (!(input instanceof IntValue)) {
               throw new CoercingParseLiteralException(
                   "Expected AST type 'IntValue' but was other'."
               );
           }
           Integer value = ((IntValue) input).getValue();
           if (result % 2 !=0){
                throw new CoercingSerializeException(
                    "It's not a valid odd number."
                );
            }
            return value;
        }
    };

}
```

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 ocurrs, the server sends data to the client.

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

5.2 Code

The only difference between implementing a mutation and a query is that methods for mutations are in a class that implements GraphQLMutationResolver and for queries in a class that implements GraphQL-QueryResolver.

QueryResolver.java

```
@Component
public class QueryResolver implements GraphQLQueryResolver {

    public List<Monkeys> listMonkeys(DataFetchingEnvironment env) {
        return monkeyService
            .listMonkeys()
            .stream()
            .map(monkeyModelToMonkey::convert)
            .collect(Collectors.toList());
    }

    public Movie getMonkey(Integer monkeyId,DataFetchingEnvironment env) {
        MovieEntity entity = monkeyService.findMonkeyById(monkeyId);
        return monkeyModelToMonkey.convert(entity);
    }

}
```

MutationResolver.java

```
@Component
public class MutationResolver implements GraphQLMutationResolver {
     public Monkey addMonkey(MonkeyRequest request,DataFetchingEnvironment env) {
        try {
            MonkeyModel entity = monkeyRequestToMonkeyModel.convert(request);
            entity = monkeyService.addMonkey(entity);
            return monkeyModelToMonkey.convert(entity);
        } catch (Exception ex) {
            throw ex;
        }
    }
}
```

Subscriptions looks a little bit different because we need to subscribe to an event. The methods must be defined in a class that implements GraphQLSubscriptionResolver.

SubscriptionResolver.java

```
import org.reactivestreams.Publisher;

@Component
public class SubscriptionResolver implements GraphQLSubscriptionResolver {

    public Publisher<Monkey> checkNewMonekys() {
        return monkeysPublisher.getPublisher();
    }

}
```

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

Unions and interfaces can be easily represented in Java since the language provides us with abstract classes and interfaces. The below pieces of code show an example for the GraphQL definitions in the previous point.

Restaurant.java

```java
public class Restaurant {

    private String id;

    private Integer name;
}
```

Indian.java

```java
public class Indiand extends Restaurant {
    private Boolean brewedBeer;
}
```

Burger.java

```
public class Burger extends Restaurant {
    private Boolean vegetarianOptions;
}
```

QueryResolver.java

```
@Component
public class QueryResolver implements GraphQLQueryResolver {

    public List<Restaurant> listRestaurants() {
        List<Resturant> restaurants = new ArrayList<Restaurant>();
        restaurants.addAll(indianService.listAll());
        restaurants.addAll(burgerService.listAll());
        return restaurants;
    }
}
```

To implement an Union classes don't have common attributes so we could classes implements and Interface instead of extending a class.

Item.java

```
public interface Item{

}
```

Electronic.java

```
public class Electronic implements Item{
    private Float size;
    private Float weight;
}
```

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!]).
- 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!]

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

In the below example we will implement a directive that can be assigned to fields This directive transform to lowercase the string attribute.

LowercaseDirective.js (this file will be in package directive)

```
@Component
public class LowercaseDirective implements SchemaDirectiveWiring {
```

```
    @Override
    public GraphQLFieldDefinition onField(SchemaDirectiveWiringEnvironment<GraphQLFieldDefinition> env)
        GraphQLFieldDefinition field = env.getElement();
        DataFetcher dataFetcher = DataFetcherFactories.wrapDataFetcher(field.getDataFetcher(), (((dataF
            if (value instanceof String) {
                return ((String) value).toLowercase();
            }
            return value;
        })));
        return field.transform(builder -> builder.dataFetcher(dataFetcher));
    }
}
```

Once we've defined our directive we just need to register the bean.

Application.java

```
@SpringBootApplication
public class Application {

    @Bean
    @Autowired
    public SchemaDirective lowerCaseDirective(LowercaseDirective directive) {
        return new SchemaDirective("lower", directive);
    }
}
```

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
```

2. Create a directive **?** 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 an 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