GraphQL-SpringBoot-MySql-CRUD Api

1) What is GraphQL?

GraphQL is a query language for APIs and a runtime for fulfilling those queries with your existing data. GraphQL provides a complete and understandable description of the data in your API, gives clients the power to ask for exactly what they need and nothing more, makes it easier to evolve APIs over time, and enables powerful developer tools.
>> Official Documentation

Traditional REST APIs work with the concept of Resources that the server manages. These resources can be manipulated in some standard ways, following the various HTTP verbs. This works very well as long as our API fits the resource concept, but quickly falls apart when we need to deviate from it.

This also suffers when the client needs data from multiple resources at the same time. For example, requesting a blog post and the comments. Typically this is solved by either having the client make multiple requests or by having the server supply extra data that might not always be required, leading to larger response sizes.

GraphQL offers a solution to both of these problems. It allows for the client to specify exactly what data is desired, including from navigating child resources in a single request, and allows for multiple queries in a single request.

It also works in a much more RPC manner, using named queries and mutations instead of a standard mandatory set of actions. **This works to put the control where it belongs, with the API developer specifying what is possible, and the API consumer what is desired.**

For example, a blog might allow the following query:

```
query {
    recentPosts(count: 10, offset: 0) {
        id
        title
        category
        author {
            id
            name
            thumbnail
        }
    }
}
```

This query will:

- request the ten most recent posts
- for each post, request the ID, title, and category
- for each post request the author, returning the ID, name, and thumbnail

In a traditional REST API, this either needs 11 requests – 1 for the posts and 10 for the authors – or needs to include the author details in the post details.

Now We will Create a CRUD Api using GraphQL-SpringBoot-MySQL.

First of all we need to declare GraphQL Schema.

- **GraphQL Schema**

  The GraphQL server exposes a schema describing the API. This scheme is made up of type definitions. Each type has one or more fields, which each take zero or more arguments and return a specific type.

  The graph is made up from the way these fields are nested with each other. Note that there is no need for the graph to be acyclic – cycles are perfectly acceptable – but it is directed. That is, the client can get from one field to its children, but it can't automatically get back to the parent unless the schema defines this explicitly.

  Ex:

```
type Country {
id: ID!
countryCode: String!
countryName: String!
region: String!
}

input CreateCountryInput {
        countryCode: String!
        countryName: String!
        region: String!
```

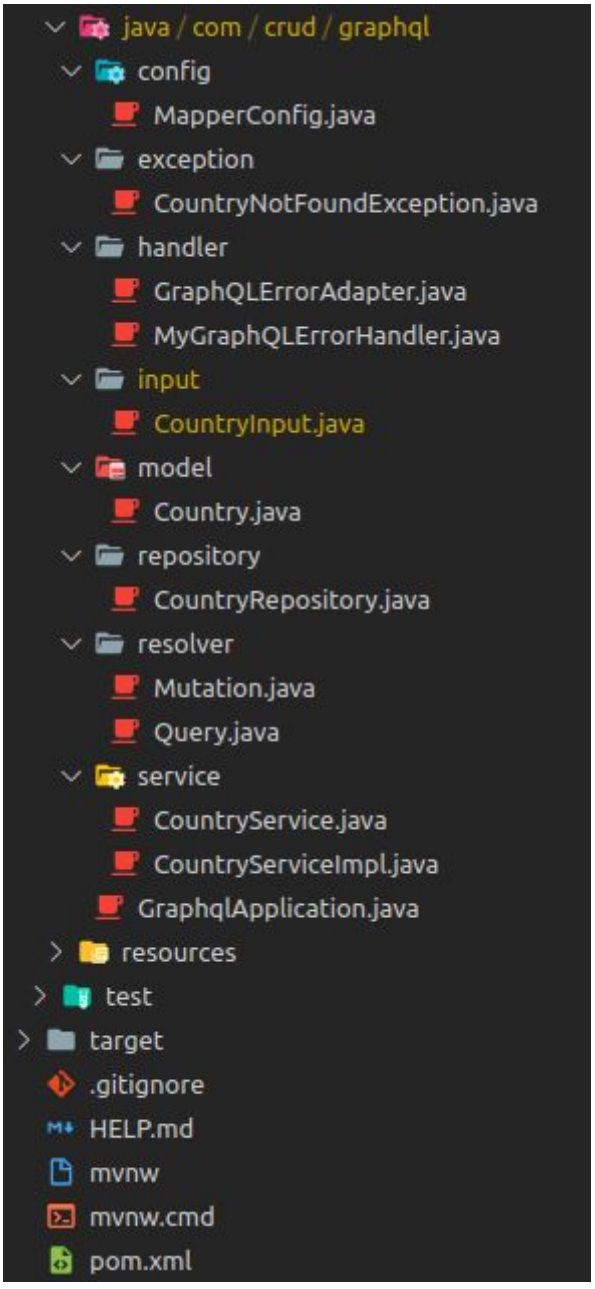
```
}

input UpdateCountryInput {
        countryCode: String!
        countryName: String!
        region: String!
}

# The Root Query for the Application.
type Query {
        getAllCountries: [Country!]!
        getCountryByCountryCode(countryCode: String!): Country
}

# The Root Mutation for the Application.
type Mutation {
        createCountry(countryInput:CreateCountryInput!): Country!
        updateCountry(id: ID!, countryInput:UpdateCountryInput!): Country!
        deleteCountry(id:ID!): Country!
}
```

This is the GraphQL Schema which I have used for My project.

The “!” at the end of some names indicates that this is a non-nullable type. Any type that does not have this can be null in the response from the server. The GraphQL service handles these correctly, allowing us to request child fields of nullable types safely.

The GraphQL Service also exposes the schema itself using a standard set of fields, allowing any client to query for the schema definition ahead of time.

Folder Structure of My Project:



Introduction to GraphQL SpringBoot Starter:

The Spring Boot GraphQL Starter offers a fantastic way to get a GraphQL server running in a very short time. Combined with the GraphQL Java Tools library, we need only write the code necessary for our service.

We need to add maven repositories for GraphQL in our **pom.xml**

```
<dependency>
	<groupId>com.graphql-java</groupId>
	<artifactId>graphql-spring-boot-starter</artifactId>
	<version>5.0.2</version>
</dependency>
<dependency>
	<groupId>com.graphql-java</groupId>
	<artifactId>graphql-java-tools</artifactId>
	<version>5.2.4</version>
</dependency>
<dependency>
	<groupId>com.graphql-java</groupId>
	<artifactId>graphiql-spring-boot-starter</artifactId>
	<version>5.0.2</version>
</dependency>
```

After adding these dependencies, our **pom.xml** file will look as given below.

Pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
	xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
	<modelVersion>4.0.0</modelVersion>
	<parent>
		<groupId>org.springframework.boot</groupId>
		<artifactId>spring-boot-starter-parent</artifactId>
		<version>2.2.4.RELEASE</version>
		<relativePath /> <!-- lookup parent from repository -->
	</parent>
	<groupId>com.crud</groupId>
	<artifactId>graphql</artifactId>
	<version>0.0.1-SNAPSHOT</version>
	<name>graphql</name>
	<description>Demo project for Spring Boot</description>

	<properties>
```

```
        <java.version>1.8</java.version>
        <orika.version>1.5.4</orika.version>
        <graphql.version>5.0.2</graphql.version>
        <graphql-java-tools.version>5.5.2</graphql-java-tools.version>
</properties>

<dependencies>
        <dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter</artifactId>
        </dependency>

        <dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-devtools</artifactId>
                <scope>runtime</scope>
                <optional>true</optional>
        </dependency>

        <dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-test</artifactId>
                <scope>test</scope>
        </dependency>

        <dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-data-jpa</artifactId>
        </dependency>

        <dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-web</artifactId>
        </dependency>

        <dependency>
                <groupId>com.graphql-java</groupId>
                <artifactId>graphql-spring-boot-starter</artifactId>
                <version>5.0.2</version>
        </dependency>
        <dependency>
                <groupId>com.graphql-java</groupId>
                <artifactId>graphql-java-tools</artifactId>
                <version>5.2.4</version>
        </dependency>
        <dependency>
```

```
                <groupId>com.graphql-java</groupId>
                <artifactId>graphiql-spring-boot-starter</artifactId>
                <version>5.0.2</version>
            </dependency>

            <!-- https://mvnrepository.com/artifact/mysql/mysql-connector-java -->
            <dependency>
                <groupId>mysql</groupId>
                <artifactId>mysql-connector-java</artifactId>
            </dependency>

            <!-- https://mvnrepository.com/artifact/org.projectlombok/lombok -->
            <dependency>
                <groupId>org.projectlombok</groupId>
                <artifactId>lombok</artifactId>
                <scope>provided</scope>
            </dependency>

            <!-- https://mvnrepository.com/artifact/ma.glasnost.orika/orika-core -->
            <dependency>
                <groupId>ma.glasnost.orika</groupId>
                <artifactId>orika-core</artifactId>
                <version>1.4.1</version>
            </dependency>

        </dependencies>

        <build>
            <plugins>
                <plugin>
                    <groupId>org.springframework.boot</groupId>
                    <artifactId>spring-boot-maven-plugin</artifactId>
                </plugin>
            </plugins>
        </build>

</project>
```

Now we will create our POJO class. Create package under the folder src/main/java named as " **com.crud.graphql.model** " and create a **Country.java** class beneath that package.

Country.java [class]

```
package com.crud.graphql.model;

import java.time.LocalDateTime;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;


import org.springframework.data.annotation.CreatedDate;
import org.springframework.data.annotation.LastModifiedDate;

@Entity
@Table(name = "country")
public class Country {

	@Id
	@Column(name = "id")
	@GeneratedValue(strategy = GenerationType.AUTO)
	private int id;

	@Column(unique = true, name = "countryCode")
	private String countryCode;

	@Column(name = "countryName")
	private String countryName;

	@Column(name = "region")
	private String region;

	@CreatedDate
	@Column
	private LocalDateTime createdAt;


	@Column
	private String createdBy = "Manual";
```

```
@LastModifiedDate
@Column
private LocalDateTime updatedAt;


@Column
private String updatedBy = "Manual";


public int getId() {
	return id;
}

public void setId(int id) {
	this.id = id;
}

public String getCountryCode() {
	return countryCode;
}

public void setCountryCode(String countryCode) {
	this.countryCode = countryCode;
}

public String getCountryName() {
	return countryName;
}

public void setCountryName(String countryName) {
	this.countryName = countryName;
}

public String getRegion() {
	return region;
}

public void setRegion(String region) {
	this.region = region;
}

public LocalDateTime getCreatedAt() {
	return createdAt;
}
```

```
	public void setCreatedAt(LocalDateTime createdAt) {
		this.createdAt = createdAt;
	}

	public String getCreatedBy() {
		return createdBy;
	}

	public void setCreatedBy(String createdBy) {
		this.createdBy = createdBy;
	}

	public LocalDateTime getUpdatedAt() {
		return updatedAt;
	}

	public void setUpdatedAt(LocalDateTime updatedAt) {
		this.updatedAt = updatedAt;
	}

	public String getUpdatedBy() {
		return updatedBy;
	}

	public void setUpdatedBy(String updatedBy) {
		this.updatedBy = updatedBy;
	}
}
```

This is our pojo class with getter and setter. After building successfully, it will generate a table named as "**country**" in database.

Now we will create Repository and Service for this POJO. Repository and Service will be the Interface and will create class for Service Implementation. Create "**com.crud.graphql.repository**" and "**com.crud.graphql.service**" package.

CountryRepository.java [Interface]

```
package com.crud.graphql.repository;


import java.util.Optional;

import org.springframework.data.jpa.repository.JpaRepository;
```

```
import com.crud.graphql.model.Country;

public interface CountryRepository extends JpaRepository<Country, String> {

	Optional<Country> findByCountryCode(String countryCode);

	Country findById(int id);

}
```

CountryService.java [Interface]

```
package com.crud.graphql.service;

import java.util.List;

import com.crud.graphql.model.Country;


public interface CountryService {

	List<Country> getAllCountries();

	Country validateAndGetCountryById(int id);

	Country saveCountry(Country country);

	boolean deleteCountry(Country country);

	Country validateAndGetCountryByCountryCode(String countryCode);

}
```

CountryServiceImpl.java [Class]

```
package com.crud.graphql.service;

import java.util.List;


import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import com.crud.graphql.exception.CountryNotFoundException;
```

```
import com.crud.graphql.model.Country;
import com.crud.graphql.repository.CountryRepository;

@Service
public class CountryServiceImpl implements CountryService{

	@Autowired
	private CountryRepository countryRepository;

	public CountryServiceImpl(CountryRepository countryRepository) {
		this.countryRepository = countryRepository;
	}

	@Override
	public List<Country> getAllCountries() {
		return countryRepository.findAll();
	}


	@Override
	public Country saveCountry(Country country) {
		return countryRepository.save(country);
	}

	@Override
	public boolean deleteCountry(Country country) {
		countryRepository.delete(country);
		return true;
	}

	@Override
	public Country validateAndGetCountryByCountryCode(String countryCode) {
		return countryRepository.findByCountryCode(countryCode).orElseThrow(() -> new
CountryNotFoundException("Country Not Found!!", "countryCode", countryCode));
	}

	@Override
	public Country validateAndGetCountryById(int id) {
		return countryRepository.findById(id);
	}

}
```

Now we have an operation to work with database, so now we will create API, but before that we will create an Input that we will use in the Api as a parameter.

Create "**com.crud.graphql.input**" package and inside that create **Countryinput.java**.

CountryInput.java [Class]

```
package com.crud.graphql.input;

import lombok.Data;

@Data
public class CountryInput {

	private String countryCode;
	private String countryName;
	private String region;
}
```

Now we will create GraphQL Api for CRUD operation. So, for that create "**com.crud.graphql.resolver**" package and create **Mutation.java** and **Query.java** classes beneath that package.

Mutation.java [Class]

```
package com.crud.graphql.resolver;



import java.time.LocalDateTime;

import org.springframework.stereotype.Component;

import com.coxautodev.graphql.tools.GraphQLMutationResolver;
import com.crud.graphql.input.CountryInput;
import com.crud.graphql.model.Country;
import com.crud.graphql.service.CountryService;


import ma.glasnost.orika.MapperFacade;

@Component
public class Mutation implements GraphQLMutationResolver {

	public CountryService countryService;

	public MapperFacade mapperFacade;

	public Mutation(CountryService countryService, MapperFacade mapperFacade) {
```

```
		this.countryService = countryService;
		this.mapperFacade = mapperFacade;
	}


	public Country createCountry(CountryInput countryInput) {
		Country country = mapperFacade.map(countryInput, Country.class);
		country.setCreatedAt(LocalDateTime.now());
		return countryService.saveCountry(country);
	}

	public Country updateCountry(int id, CountryInput countryInput) {
		Country country = countryService.validateAndGetCountryById(id);
		mapperFacade.map(countryInput, country);
		country.setUpdatedAt(LocalDateTime.now());
		return countryService.saveCountry(country);
	}

	public Country deleteCountry(int id) {
		Country country = countryService.validateAndGetCountryById(id);
		countryService.deleteCountry(country);
		return country;
	}

}
```

Basically **Mutation.java class** contains methods which will be used to create, update and delete information in database. It will Mutate or Alter the database. It is DDL queries. On other hand **Query.java** only contains those methods which will be used to get data from the database. It is DML queries.

Query.java [Class]

```
package com.crud.graphql.resolver;

import java.util.List;

import org.springframework.stereotype.Component;

import com.coxautodev.graphql.tools.GraphQLQueryResolver;
import com.crud.graphql.model.Country;
import com.crud.graphql.service.CountryService;

@Component
public class Query implements GraphQLQueryResolver{
```

```
	private CountryService countryService;

	public Query(CountryService countryService) {
		this.countryService = countryService;
	}

	public List<Country> getAllCountries() {
		return countryService.getAllCountries();
	}

	public Country getCountryByCountryCode(String countryCode) {
		return countryService.validateAndGetCountryByCountryCode(countryCode);
	}

}
```

We have our API and Database, now we will create graphql schema file. Create "**graphql**" folder under the **src/main/resources** path and create **Country.graphqls** file inside that folder. This file will work as query input for our API, we do not need to give path of this file into resolver as it takes automatically.

Country.graphqls [Schema File]

```
type Country {
	id: ID!
	countryCode: String!
	countryName: String!
	region: String!
}

input CreateCountryInput {
	countryCode: String!
	countryName: String!
	region: String!
}

input UpdateCountryInput {
	countryCode: String!
	countryName: String!
	region: String!
}

# The Root Query for the Application.
type Query {
	getAllCountries: [Country!]!
```

```
	getCountryByCountryCode(countryCode: String!): Country
}

# The Root Mutation for the Application.
type Mutation {
	createCountry(countryInput:CreateCountryInput!): Country!
	updateCountry(id: ID!, countryInput:UpdateCountryInput!): Country!
	deleteCountry(id:ID!): Country!
}
```

Basically, it contains an operation that will work with API. First of all we have declared our "**type Country**." It contains information that we need to work with API. After that **"type Query"** contains methods as schema and it's parameters as we have declared in the **Query.java** and "**type Mutation**" Contains methods as schema and it's necessary parameters as a schema as we have declared in the **Mutation.java**.

Note: The name of types and it's parameters name should be the same as declared in the POJO, and Resolver. Small letters should be in small and capital should be in capital, otherwise it will show Error like…"Error creating bean name as SchemaParser".

Application.properties [Configuration File]

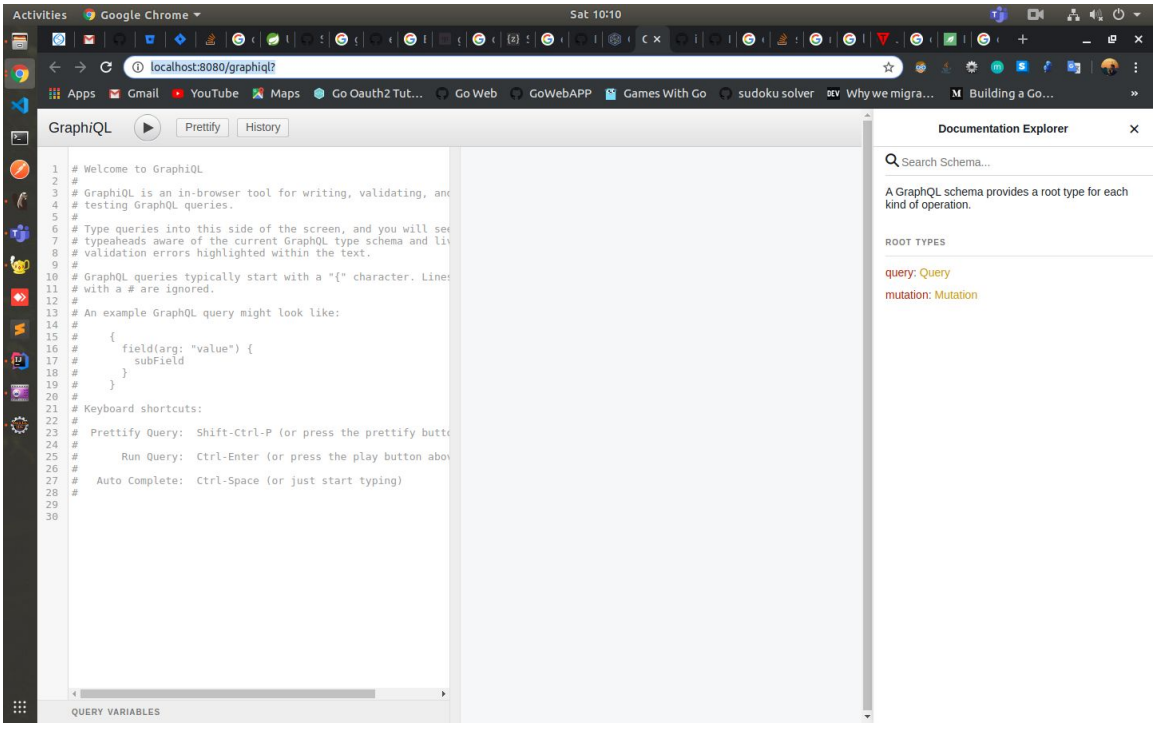
```
spring.datasource.url = jdbc:mysql://192.168.0.214:3306/dev46?useSSL=false
spring.datasource.username = root
spring.datasource.password = root

spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQL5InnoDBDialect
spring.jpa.hibernate.ddl-auto=update
```

Here we are done with our API. Our main class is given below.

GraphqlApplication.java [Main Class]----------

```
package com.crud.graphql;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class GraphqlApplication {

	public static void main(String[] args) {
		SpringApplication.run(GraphqlApplication.class, args);
	}

}
```

After this you can run our application and test the apis. After building successfully go to this http://localhost:8080/graphiql? Link. it will show you a page something like given below.



We can go further and can create an error handler for graphql errors.
Create "**com.crud.graphql.handler**" package and create **GraphQLErrorAdapter.java** and **MyGraphQLErrorHandler.java** classes under that package.
First class will work as a Error Adapter and Second class will process those errors.

GraphQLErrorAdapter.java

```
package com.crud.graphql.handler;

import java.util.List;
import java.util.Map;

import graphql.ErrorType;
import graphql.ExceptionWhileDataFetching;
import graphql.GraphQLError;
import graphql.language.SourceLocation;

public class GraphQLErrorAdapter implements GraphQLError {


	private static final long serialVersionUID = 1L;
	private GraphQLError error;

	public GraphQLErrorAdapter(GraphQLError error) {
```

```
		this.error = error;
	}

	@Override
	public Map<String, Object> getExtensions() {
		return error.getExtensions();
	}

	@Override
	public List<SourceLocation> getLocations() {
		return error.getLocations();
	}

	@Override
	public ErrorType getErrorType() {
		return error.getErrorType();
	}

	@Override
	public List<Object> getPath() {
		return error.getPath();
	}

	@Override
	public Map<String, Object> toSpecification() {
		return error.toSpecification();
	}

	@Override
	public String getMessage() {
		return (error instanceof ExceptionWhileDataFetching)
				? ((ExceptionWhileDataFetching) error).getException().getMessage()
				: error.getMessage();
	}
}
```

MyGraphQLErrorHandler.java

```
package com.crud.graphql.handler;

import java.util.ArrayList;
import java.util.List;
import java.util.stream.Collectors;

import graphql.ExceptionWhileDataFetching;
import graphql.GraphQLError;
```

```
import graphql.servlet.GraphQLErrorHandler;

public class MyGraphQLErrorHandler implements GraphQLErrorHandler{


   @Override
   public List<GraphQLError> processErrors(List<GraphQLError> graphQLErrors) {
      List<GraphQLError> clientErrors = graphQLErrors.stream()
            .filter(this::isClientError)
            .collect(Collectors.toList());

      List<GraphQLError> serverErrors = graphQLErrors.stream()
            .filter(e -> !isClientError(e))
            .map(GraphQLErrorAdapter::new)
            .collect(Collectors.toList());

      List<GraphQLError> graphQLErrorList = new ArrayList<>();
      graphQLErrorList.addAll(clientErrors);
      graphQLErrorList.addAll(serverErrors);
      return graphQLErrorList;
   }

   private boolean isClientError(GraphQLError error) {
      return !(error instanceof ExceptionWhileDataFetching || error instanceof Throwable);
   }


}
```

One More package we will create for configuration name as "**com.crud.graphql.config**" and we will create **MapperConfig.java** class inside that package. It will work as a configuration.

MapperConfig.java

```
package com.crud.graphql.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import ma.glasnost.orika.MapperFacade;
import ma.glasnost.orika.MapperFactory;
import ma.glasnost.orika.impl.DefaultMapperFactory;

@Configuration
public class MapperConfig {
```

```
    @Bean
    MapperFactory mapperFactory() {
        return new DefaultMapperFactory.Builder().useAutoMapping(true).mapNulls(false).build();
    }

    @Bean
    MapperFacade mapperFacade() {
        return mapperFactory().getMapperFacade();
    }

}
```

Here our coding parts come to an End. Now i will show you the examples of all the mutation and query.

Mutation

1) **createCountry**

```
mutation {
  createCountry(
    countryInput:{
      countryCode: "+91",
      countryName: "India"
      region: "Asia"
    }){
    id
  }
}
```

2) **updateCountry**

```
mutation{
  updateCountry(id:37, countryInput:{
    countryCode: "+001",
    countryName: "canada",
    region: "North America"
  }){
    id
  }
}
```

3) **deleteCountry**

```
mutation{
  deleteCountry(id:38){
    id
  }
}
```

Query

1) **getAllCountries**

```
query{
  getAllCountries{
    countryCode,
    countryName
  }
}
```

```
{
  "data": {
    "getAllCountries": [
      {
        "countryCode": "+91",
        "countryName": "India"
      },
      {
        "countryCode": "+001",
        "countryName": "canada"
      }
    ]
  }
}
```

2) **getCountryByCountryCode**

```
query{
  getCountryByCountryCode(countryCode:"+91"){
    countryName
  }
}
```

```
{
  "data": {
    "getCountryByCountryCode": {
      "countryName": "India"
    }
  }
}
```

Video Link for GraphQL Mutation API and Query API Output

The End