**GraphQL VS REST API – 2025**

**Introduction**

Let us consider a sample example that we have large object model like Employee. The code is given below.

@Data @NoArgsConstructor @AllArgsConstructor

public class Employee {

private int id;

private String firstName;

private String lastName;

private String emailId;

private Address address;

private Department dept;

private MeritalStatus mStatus;

}

Now we write a REST controller to get the employee details.

@GetMapping(path="/emp/{id}")

public ResponseEntity<Employee> getEmployeeById(@PathVariable("id") String id) {

Employee emp = EmpUtil.*getDefaultEmployee*();

return ResponseEntity.*ok*(emp);

}

Below response is obtained.

{

"id": 123,

"firstName": "John",

"lastName": "Abraham",

"emailId": "john.abraham@ddlabinc.com",

"address": {

"cityName": "Bangalore",

"streetName": "Mallahalli",

"pinCode": "516638"

},

"dept": {

"name": "Finance Corp",

"type": "Finance",

"project": {

"id": 1785,

"name": "Zumbalica Modelling",

"desc": "An Inhouse project for CarterPillar",

"ptype": {

"projectType": "Financial",

"clientName": "CarterPillar"

}

}

},

"mstatus": {

"type": "Single"

}

}

In the above case, the user may not require all the data, user may be interested for few data.

To solve the above problem, GraphQL provides the efficient solution.

In case of GraphQL, we define a schema named “schema.graphqls” inside a folder called graphql inside src/main/resources.

The content is given below.

**src/main/resources**/**schema.graphqls**

type Query {

employeeById(id: ID): Employee

}

type Employee {

id: ID

firstName: String

lastName: String

address: Address

emailId: String

dept: Department

mStatus: MeritalStatus

}

type Department {

name: String

type: String

project: Project

}

type Address {

cityName: String

streetName: String

pinCode: String

}

type MeritalStatus {

type: String

}

type Project {

id: ID

name: String

desc: String

pType: ProjectType

}

type ProjectType {

projectType: String

clientName: String

}

Now we write a controller for this.

@Controller

public class EmployeeGraphQLController {

@QueryMapping

public Employee employeeById(@Argument String id) {

return EmpUtil.*getDefaultEmployee*();

}

}

To test, go to Postman client and hit the URL: <http://localhost:8080/graphql>, it will load the following.

A screenshot of a computer

Description automatically generated

Now, user is interested to the few details like employee firstName, address: cityName etc as shown below.

A screenshot of a computer

Description automatically generated

GraphQL query is given below.

query EmployeeById {

    employeeById(id: "111") {

        firstName

        address {

            cityName

        }

        emailId

        dept {

            name

            project {

                name

            }

        }

        mStatus {

            type

        }

    }

}

As it is clear that instead of getting all the data, user can see the few details which helps in reducing the data required for the functionality. We can design the data in any manner like user may not be interested in marital status or department etc.

The response is given below.

{

    "data": {

        "employeeById": {

            "firstName": "John",

            "address": {

                "cityName": "Bangalore"

            },

            "emailId": "john.abraham@ddlabinc.com",

            "dept": {

                "name": "Finance Corp",

                "project": {

                    "name": "Zumbalica Modelling"

                }

            },

            "mStatus": {

                "type": "Single"

            }

        }

    }

}

**Real Advantages of GraphQL**

GraphQL is an excellent solution particularly when dealing with large datasets. If you want the client to control the type and amount of data it needs, GraphQL is ideal for your project. The main benefit of using GraphQL is the ability to send a query that specifies only the information you need and receive exactly that.

**Data fetching control:** GraphQL was designed to allow the client to ask for only the data it needs. While the server might be able to deliver more data to the client for a single request, it would only send the data that the client requests.

**Alleviating bandwidth concerns:** Bandwidth is a problem for small devices like mobile phones, smartwatches, and IoT devices that can’t handle large amounts of data. Using GraphQL helps minimize this issue. Because GraphQL allows the client to specify what data it needs.

**Rapid prototyping:** GraphQL exposes a single endpoint that allows you to access multiple resources. In addition, resources are not exposed according to the views that you have inside your app. For example, if your UI changes, and requires either more or less data, it won’t have an impact or require changes from the server.

**Execution of multiple Queries:** Multiple queries can be executed in one call.

**Disadvantages**

With GraphQL, users can’t simply run any query they want. A GraphQL API must be carefully designed; it’s not just about putting it on top of a REST API or a database.

For complex queries, a REST API might be easier to design because you can establish multiple endpoints for specific needs, and you can fine-tune specific queries to efficiently retrieve the data.

It’s important to keep in mind that GraphQL is an alternative to REST for developing APIs, not a replacement.

**How to change the GraphQL endpoint in SpringBoot ?**

We normally access like this <http://localhost:8080/graphql>.

If you want to change the path, you have to use the below property.

**spring.graphql.path=/service/api/query**

**@QueryMapping vs @SchemaMapping**

**@QueryMapping is a composed annotation that acts as a shortcut for @SchemaMapping with typeName="Query".**

@QueryMapping //Alias for SchemaMapping.field().

**@SchemaMapping**

**The @SchemaMapping annotation maps a handler method to a field in the GraphQL schema and declares it to be the DataFetcher for that field.**

**Nullability and lists**

type Author {

books: [Book!]! # This list can't be null AND its list \*items\* can't be null

}

**If ! appears inside the square brackets, the returned list can't include items that are null.**

**If ! appears outside the square brackets, the list itself can't be null.**

**In any case, it's valid for a list field to return an empty list.**

**What is an exclamation point in GraphQL**

In GraphQL, an exclamation point (!) indicates that a field in a query or a field argument is non-nullable. This means that the field must contain a value and cannot be empty.

**Fragments in GraphQL Quer – 2025**

Fragments are defined using the fragment keyword followed by a name and a set of fields. To use a fragment in a query, simply include the fragment name preceded by **... (three dots)** and followed by any additional fields you need. Fragments help reduce duplication in the query documents and make the queries more maintainable

Example:

fragment empSalary on Employee {

name

}

Implemented and Working Sample Queries are given below.

**Use Case-1**: We are only interested to know firstName and lastName of person along with other details.

**query** GetPersonInfo {

    getPersonInfo {

**...personFragment**

        dept {

            id

            name

**type**

            project {

                id

                name

                clientName

                cost {

                    infraCost

                    hardwareCost

                    securityCost

                }

            }

        }

    }

}

**fragment** **personFragment** **on** Person {

    firstName

    lastName

}

**personFragment** is just a name on the type Person defined in **schema.graphqls**.

Response is given below

{

    "data": {

        "getPersonInfo": {

            "firstName": "QU\_U8\_qLLb",

            "lastName": "FaGWb2BarO",

            "dept": {

                "id": "1518290995200",

                "name": "9JjxdLBJaQ",

                "type": "MORHT4OtxC",

                "project": {

                    "id": "1670865484",

                    "name": "1l5G3rfjFO",

                    "clientName": "RjJKQsxvi6",

                    "cost": {

                        "infraCost": 0.2531779,

                        "hardwareCost": 0.34294894,

                        "securityCost": 0.7137665

                    }

                }

            }

        }

    }

}

**Use Case-2**: Use two fragments, one for Person and Another for Department.

Graphql query is given below.

**query** GetPersonInfo {

    getPersonInfo {

**...personFragment**

        dept {

**... deptFragment**

            project {

                id

                name

                clientName

                cost {

                    infraCost

                    hardwareCost

                    securityCost

                }

            }

        }

    }

}

**fragment** personFragment **on** Person {

    firstName

    lastName

}

**fragment** deptFragment **on** Department {

    name

**type**

}

**personFragment** and **deptFragment** are the two names of the fragments.

Note: There is no change in the server side code. These are operational ways to retrieve the information.

**Use Case: - 3**: Multiple fragments with nested object details.

**query** GetPersonInfo {

    getPersonInfo {

**...personFragment**

        adrs {

**... adrsFragment**

        }

        dept {

**... deptFragment**

            project {

                id

                name

                clientName

                cost {

                    infraCost

                    hardwareCost

                    securityCost

                }

            }

        }

    }

}

**fragment** personFragment **on** Person {

    firstName

    lastName

}

**fragment** adrsFragment **on** Address {

    cityName

    pinCode

}

**fragment** deptFragment **on** Department {

    name

**type**

}

**Implementation Details**

In **schema.graphqls**, brief and relevant portion is given below.

type Query {

getPersonInfo: Person

}

type Employee {

id: ID

firstName: String

lastName: String

fullName: String

name: String @deprecated(reason: "Use `fullName`.")

empSalary: EmpSalary

}

**Java code:** Podam library has been used to auto-populate the object**.**

@Controller

**public** **class** PersonController {

@QueryMapping

**public** Person getPersonInfo() {

PodamFactory factory = **new** PodamFactoryImpl();

Person person1 = factory.manufacturePojo(Person.**class**);

**return** person1;

}

}

How to run multiple GraphQL queries and combine multiple sources

**Advantages of running multiple queries**

* It optimizes the number of network requests, instead of 5 API calls to the server we need to make only one.
* It helps to minimize effort required in writing the client side code
* It is easier to build complex UI components that need data from different data points.

**Technical Implementation**

Following is the schema.graphqls.

**type Query {**

**getBookByName(name: String): Book**

**getAuthorByName(authorName: String): Author**

**}**

**type Book {**

**id: ID**

**isbnNo: String**

**bookName: String**

**}**

**type Author {**

**authorName: String**

**bookName: String**

**}**

Two separate queries are given below to get the Book and Author Details.

Query-1: **query GetBookByName {**

**getBookByName(name: "someBookName") {**

**id**

**isbnNo**

**bookName**

**}**

**}**

Query-2: **query GetAuthorByName {**

**getAuthorByName(authorName: "John Michael") {**

**authorName**

**bookName**

**}**

**}**

**Java implementation**

@Controller

**public** **class** AuthorBookController {

@QueryMapping

**public** Book getBookByName(@Argument String name) {

System.***out***.println("------ getBookByName called ----------");

PodamFactory factory = **new** PodamFactoryImpl();

Book book = factory.manufacturePojo(Book.**class**);

book.setBookName(name);

**return** book;

}

@QueryMapping

**public** Author getAuthorByName(@Argument String authorName) {

System.***out***.println("------ getAuthorByName called ----------");

PodamFactory factory = **new** PodamFactoryImpl();

Author author = factory.manufacturePojo(Author.**class**);

author.setAuthorName(authorName);

**return** author;

}

}

**To combine both the queries**, you can write like this.

**query getAllBookAndAuthor {**

**getBookByName(name: "someBookName") {**

**id**

**isbnNo**

**bookName**

**}**

**getAuthorByName(authorName: "John Michael") {**

**authorName**

**bookName**

**}**

**}**

**Note: Here, the below query name is optional, you can write without name also.**

**query {**

**getBookByName(name: "someBookName") {**

**id**

**isbnNo**

**bookName**

**}**

**getAuthorByName(authorName: "John Michael") {**

**authorName**

**bookName**

**}**

**}**

**Response is given below**

**{**

**"data": {**

**"getBookByName": {**

**"id": "5187400853600",**

**"isbnNo": "7UXZcXDQWj",**

**"bookName": "someBookName"**

**},**

**"getAuthorByName": {**

**"authorName": "John Michael",**

**"bookName": "DnNyxcM63l"**

**}**

**}**

**}**

**GraphQL Directives**

A directive decorates part of a GraphQL schema or operation with additional configuration.

They are used to add conditional logic, apply transformations, or control the execution flow of a query or mutation.

There are two types of directives, **Schema directives** , **Operational directives.** An **operational directive** is applied to the operations (query and mutation), affecting how the GraphQL server processes an operation. **@skip and @include directives can be applied to query fields**. They allow you to skip or include a field based on the value of the if argument that is passed to the directive.

**@include directive** can be used to include a field based on the value of the if argument.

**@deprecated(reason: String) - Schema Directive**

**@skip(if: Boolean!) and @include(if: Boolean!) are operational directives**

Example on @deprecated Schema Directive

type Employee {

id: ID

firstName: String

lastName: String

fullName: String

**name: String @deprecated(reason: "Use `fullName`.")**

empSalary: EmpSalary

}

A GraphQL directive is a way to add an extra behavior by annotating parts of the GraphQL schema. We can define a directive starting with @ character followed by the name, argument (optional), and execution location.

**How to use @skip directive.**

User Case-1: Do not show full name while making query.

In **schema.graphqls**, the query is given below.

type Query {

getEmpInfo(skipFullName: Boolean!): Employee

}

The actual graphql query is given below.

query GetEmpInfo($skipFullName: Boolean!) {

getEmpInfo(skipFullName: true) {

firstName

fullName **@skip(if: $skipFullName)**

name

}

}

**and the variable has been defined below.**

{

"skipFullName":true

}

The screenshot is given below.

A screenshot of a computer

AI-generated content may be incorrect.

**Response is given below**

{

"data": {

"getEmpInfo": {

"id": "0",

"firstName": "Ram",

"lastName": "Dasa",

"name": "Some name"

}

}

}

**To show full name**

**query GetEmpInfo($skipFullName: Boolean!) {**

**getEmpInfo(skipFullName: false) {**

**id**

**firstName**

**lastName**

**fullName @skip(if: $skipFullName)**

**name**

**}**

**}**

**Variables to be used as**

{

"skipFullName": **false**

}

**Response is given below**

{

"data": {

"getEmpInfo": {

"id": "0",

"firstName": "Ram",

"lastName": "Dasa",

**"fullName": "Ram Dasa"**,

"name": "Some name"

}

}

}

**Use of @include directive in graphql**

Use Case: whether to include a particular information or not.

In **schema.graphqls**, it has been define as below.

**type Query {**

**getEmpInfo(skipFullName: Boolean!, includeSal: Boolean!): Employee**

**}**

Actual running graphql query is given below to include the salary information

**query GetEmpInfo($skipFullName: Boolean!, $includeSal: Boolean!) {**

**getEmpInfo(includeSal: true, skipFullName: true) {**

**empSalary @include(if: $includeSal) {**

**name**

**basicPay**

**}**

**fullName @skip(if: $skipFullName)**

**lastName**

**firstName**

**}**

**}**

Variables used are given below.

{

"skipFullName":true,

"includeSal": true

}

**Response**

{

"data": {

"getEmpInfo": {

"empSalary": {

"name": "Ram",

"basicPay": 1234

},

"lastName": "Dasa",

"firstName": "Ram"

}

}

}

If you do not want to include salary info, Actual GraphQL query is given below.

**query GetEmpInfo($skipFullName: Boolean!, $includeSal: Boolean!) {**

**getEmpInfo(includeSal: false, skipFullName: true) {**

**empSalary @include(if: $includeSal) {**

**name**

**basicPay**

**}**

**fullName @skip(if: $skipFullName)**

**lastName**

**firstName**

**}**

**}**

**Variables are given below**

{

"skipFullName":true,

"includeSal": false

}

**Response**

{

"data": {

"getEmpInfo": {

"lastName": "Dasa",

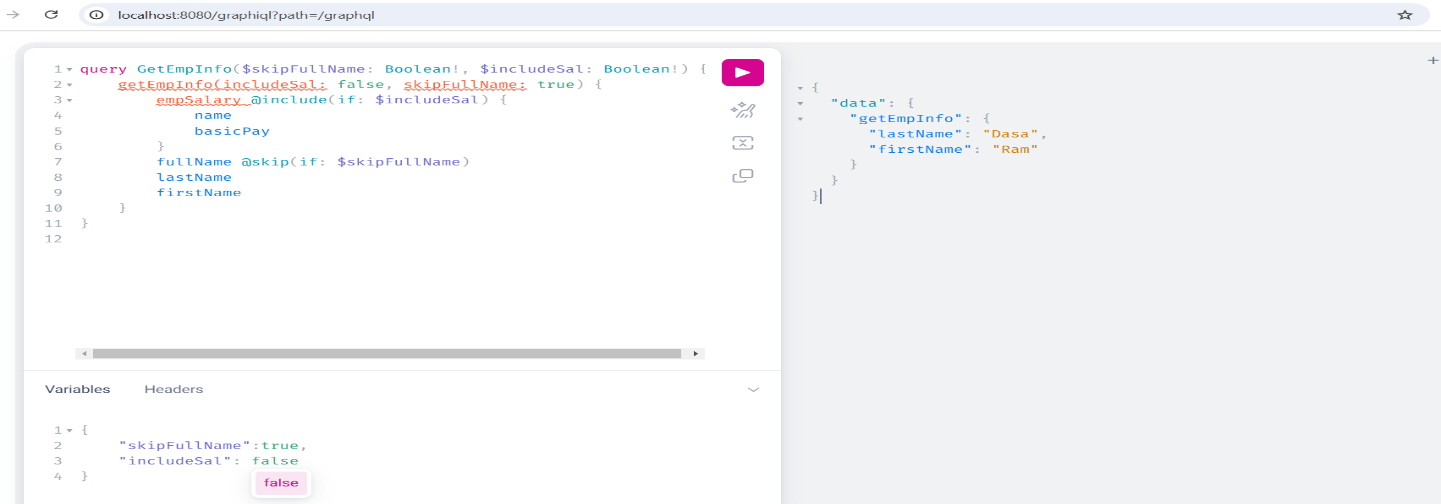
"firstName": "Ram"

}

}

}

Screenshot is given below.



**How to get Request Header in GraphQL**

Use Case:- We need to pass certain headers to delete user information.

In **schema.graphqls**, the mutation query is given below.

type Mutation {

deleteAppUser(userId: String): AppResult

}

Java code is given below.

@MutationMapping

**public** AppResult deleteAppUser(

**@ContextValue(name = "myHeader1") String header1,**

**@ContextValue(name = "myHeader2") String header2,**

@Argument String userId) {

System.***out***.println("What is the userId to be deleted: "+userId);

System.***out***.println("What is the header1: ?"+header1);

System.***out***.println("What is the header2: ?"+header2);

**return** **new** AppResult("AppUser information deleted successfully ...");

}

**Create an Interceptor class to get the value and add to GraphQl Context**.

@Component

**public** **class** GraphQlRequestHeaderInterceptor **implements** WebGraphQlInterceptor {

@Override

**public** Mono<WebGraphQlResponse> intercept(**WebGraphQlRequest request**, Chain chain) {

String value1 = request.getHeaders().getFirst("myHeader1");

String value2 = request.getHeaders().getFirst("myHeader2");

**if** (value1 != **null** && value2 != **null**) {

request.configureExecutionInput(

(executionInput, builder) -> builder

.graphQLContext(Collections.*singletonMap*("myHeader1", value1))

.graphQLContext(Collections.*singletonMap*("myHeader2", value2))

.build());

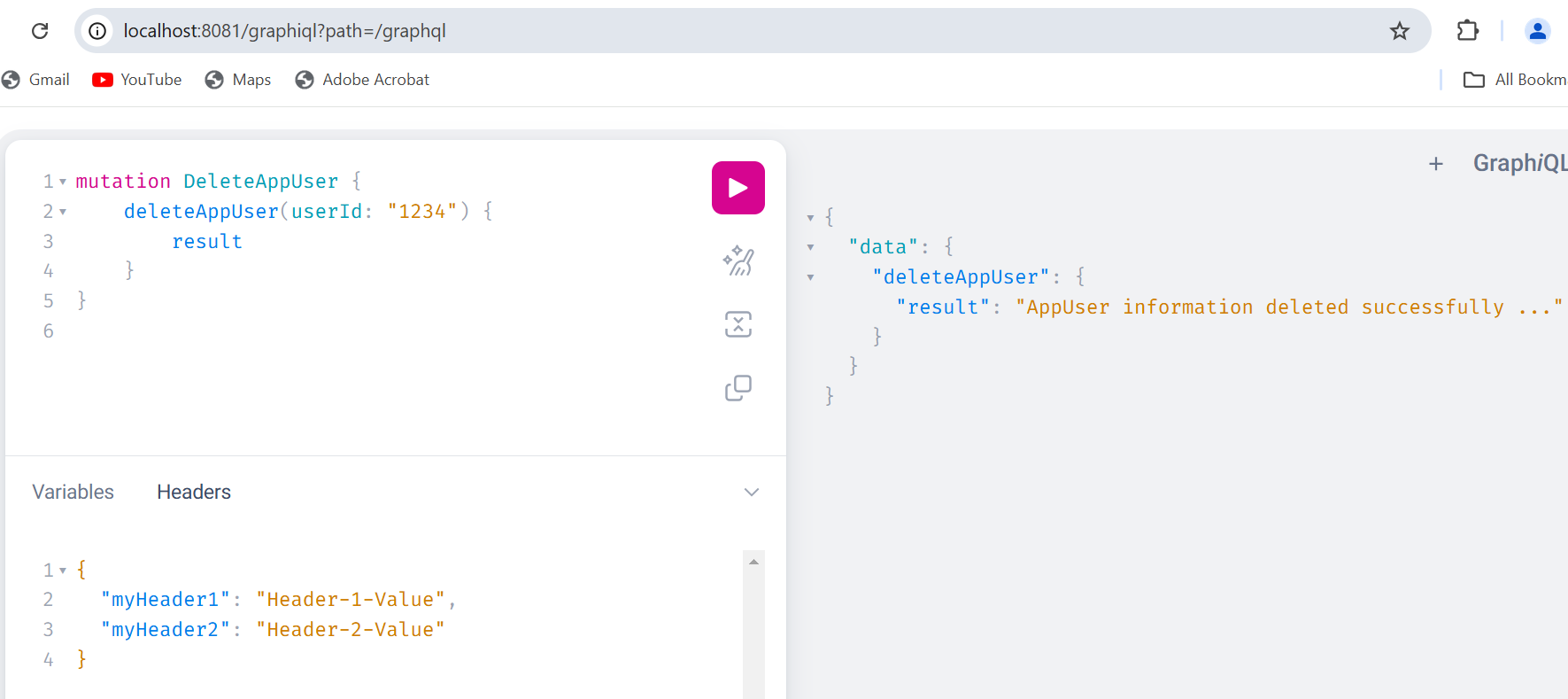
}

**return chain.next(request);**

}

}

The screen shot is given below.



**How to set Response Header in GraphQL**

Use Case:- We need to send certain headers as part of response.

In **schema.graphqls**, the query has been defined as below.

type Query {

getBookByName(name: String): Book # How to set Response Header

}

**Create an interceptor class as given below.**

@Component

**public** **class** GraphQlRequestHeaderInterceptor **implements** **WebGraphQlInterceptor** {

@Override

**public** Mono<WebGraphQlResponse> intercept(WebGraphQlRequest request, Chain chain) {

String value1 = request.getHeaders().getFirst("myHeader1");

String value2 = request.getHeaders().getFirst("myHeader2");

**if** (value1 != **null** && value2 != **null**) {

request.configureExecutionInput(

(executionInput, builder) -> builder.

graphQLContext(

Collections.*singletonMap*("myHeader1", value1))

.graphQLContext(Collections.*singletonMap*("myHeader2", value2)).build());

}

**return** chain.next(request).doOnNext((response) -> {

GraphQLContext graphQlContext = response.getExecutionInput().getGraphQLContext();

String responseValue1 = graphQlContext.get("ResponseHeader1");

String responseValue2 = graphQlContext.get("ResponseHeader1");

**if**(responseValue1 != **null** && responseValue2 != **null**)

response.getResponseHeaders().add("ResponseHeader1", responseValue1);

response.getResponseHeaders().add("ResponseHeader2", responseValue2);

});

}

}

Java code for Query Mapping is given below

@QueryMapping

**public** Book getBookByName(GraphQLContext context, @Argument String name) {

Book book = **new** Book();

book.setIsbnNo("Some ISBN No");

book.setName("A Good Book");

List<Author> authors = List.*of*(

**new** Author(111, "James Bond", List.*of*(book)),

**new** Author(333, "Charles Dickens", List.*of*(book))

);

book.setAuthors(authors);

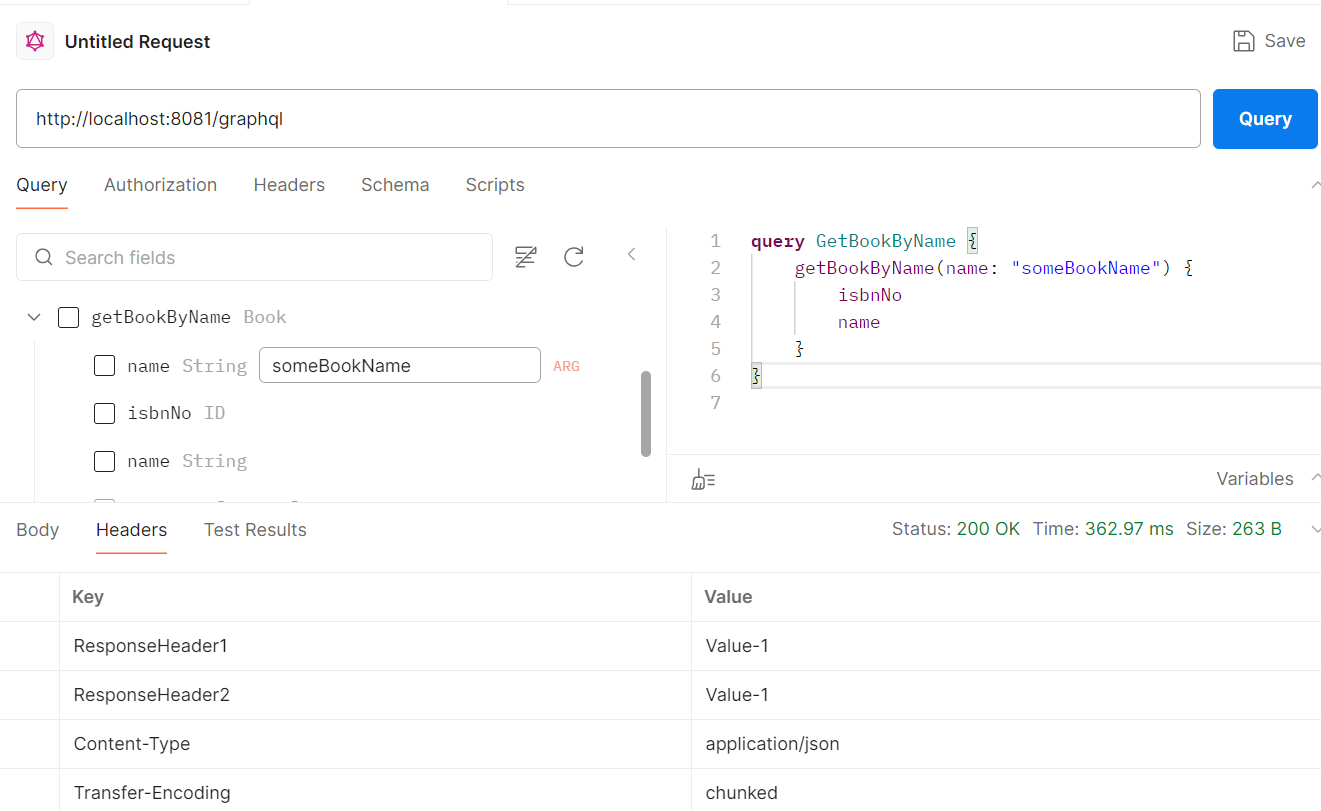
context.put("ResponseHeader1", "Value-1");

context.put("ResponseHeader2", "Value-2");

**return** book;

}

The screenshot is given below.



**How to handle Exception in GraphQL with SpringBoot**

In schema.graphqls, the query has been defined as below.

type Query {

getAppUserById(id: ID): AppUser

}

Java code for Query Mapping is given below.

@QueryMapping

**public** AppUser getAppUserById(@Argument Long id) {

**if**(id == **null**) **throw** **new** NullPointerException("Id can't be null");

**return** AppUtil.*getDefaultUser*(Long.*valueOf*(id));

}

Java code for Exception Handling like SpringBoot.

**@ControllerAdvice**

**public** **class** AppExceptionHandler {

**@GraphQlExceptionHandler**

**public** GraphQLError handleValidationException(RuntimeException ex,

DataFetchingEnvironment env) {

**return** GraphqlErrorBuilder

.*newError*()

.message("Error Message: " + ex.getMessage())

.errorType(ErrorType.***BAD\_REQUEST***)

.build();

}

}

If you execute the below, query

**query** GetAppUserById {

    getAppUserById(id: **null**) {

        firstName

        lastName

    }

}

**Response is given below**

{

    "errors": [

        {

**"message": "Error Message: Id can't be null",**

            "locations": [],

            "extensions": {

**"classification": "BAD\_REQUEST"**

            }

        }

    ],

    "data": {

**"getAppUserById": null**

    }

}

**How to use List, Set, Map, Enum in types in GraphQL**

There is **no map type in GraphQL**. Because maps are basically containing dynamic keys and, they do not play well into the static types that GraphQL is expecting. If you are really unsure of what type the data is going to be in that map, you can always store that dynamic information as a String.

Usage of List, Set and Enum in GraphQL

In **schema.graphqls**, types and queries have been defined below.

type Query {

**getAllAppUsers: [AppUser] # To get list of AppUser**

}

type Mutation {

**createAppUser(inUser: AppUserInput): AppUser # To send list of phone numbers**

}

"""

How to Define Enum in java

"""

enum MaritalStatus { 🡸 Enum type in Java

MARRIED

UNMARRIED

SINGLE

}

Java code for controllers given below.

@QueryMapping

**public** List<AppUser> getAllAppUsers() {

List<AppUser> users = **new** ArrayList<>();

PodamFactory factory = **new** PodamFactoryImpl();

AppUser appUser1 = factory.manufacturePojo(AppUser.**class**);

users.add(AppUtil.*getDefaultUser*(Long.*valueOf*(123)));

users.add(appUser1);

**return** users;

}

To get all the app user, graphql query is given below.

**query** GetAllAppUsers {

    getAllAppUsers {

        id

        firstName

        lastName

    }

}

**Response is given below**

{

    "data": {

        "getAllAppUsers": [

            {

                "id": "123",

                "firstName": "John",

                "lastName": "Abraham"

            },

            {

                "id": "19175601775000",

                "firstName": "0j7nEGE2HJ",

                "lastName": "PeTzuoddxc"

            }

        ]

    }

}

**Java code to create App User in the system**

@MutationMapping

**public** AppUser createAppUser(@Argument AppUserInput inUser) {

System.***out***.println("User Details: "+inUser);

**return** AppUtil.*getDefaultUser*(Long.*valueOf*(123));

}

To test the above scenario, the graphql query is given below.

**mutation** CreateAppUser {

    createAppUser(

        inUser: {

            firstName: "Ram"

            lastName: "Shyam"

            mStatus: MARRIED 🡸 INPUT Section

            cityName: "Bangalore"

            pinCode: "12345",

**phoneNos: ["111111","222222"]**

        }

    ) {

        firstName

        lastName 🡸 OUTPUT Section

    }

}

**How to send a list of String and to receive a list of Strings**

Define the below in **schema.graphqls**. This is bit tricky.

type Query {

getAllPhoneNosByNames(names:[String]): [String] # Send list of Strings to get list of Strings

}

Controller Method is given below.

@QueryMapping

**public** List<String> getAllPhoneNosByNames(@Argument List<String> names) {

**return** List.*of*("111", "222", "333");

}

Actual GraphQL query is given below.

**query** GetAllPhoneNosByNames {

**getAllPhoneNosByNames(names: ["Ram","Shyam"])**

}

Response is given below

{

    "data": {

        "getAllPhoneNosByNames": [

            "111",

            "222",

            "333"

        ]

    }

}

**Explain the difference between type and input type in GraphQL**

**Type**: **A type in GraphQL defines the shape and structure of an object**. It represents data that can be queried or returned by the API. Types can have fields and relationships, and they are used to define the structure of data in queries and responses.

**Input type**: **An input type in GraphQL is used to represent complex input arguments in mutations or query variables**. It allows clients to pass a structured set of data as an argument to a mutation or query. Input types cannot have fields that refer to other object types, as they are only used for input, not output.

The key difference is that **types are used for defining the structure of data in queries and responses**, **while input types are used specifically for passing structured input data in mutations or query variables**.

**What is the purpose of mutations in GraphQL**

Mutations in GraphQL are used to modify data on the server. They allow clients to perform operations that create, update, or delete data. Mutations are similar to queries but are executed with the intent of modifying data rather than fetching it.

**Explain the concept of interfaces and unions in GraphQL**

Interfaces and unions in GraphQL allow for defining more flexible and polymorphic schemas:

**Interfaces**: An interface in GraphQL defines a set of fields that must be implemented by any type that implements that interface. It allows for defining common fields or behaviors shared among multiple types. Interface fields can be queried directly, and queries can be performed on any type that implements the interface.

**Unions**: A union in GraphQL represents a type that can be one of several possible types. It allows for defining a field that can return different types based on runtime conditions. Unions are useful when a field can have multiple types of values, and you want to allow clients to query fields specific to those types.

Interfaces and unions provide mechanisms for polymorphism in GraphQL schemas, allowing for more flexibility and extensibility in representing complex data structures and relationships.

Example on Interface is given below

In schema.graphqls, it is defined as below.

type Query {

getAllTypesOfUser: [User]

}

"""

How to Define Interface Types

"""

interface User {

id : ID!

fullName : String!

cityName: String

phoneNos: [String]

}

type ContractUser implements User {

id : ID!

fullName : String!

cityName: String

phoneNos: [String]

}

type FulltimeUser implements User {

id : ID!

fullName : String!

cityName: String

phoneNos: [String]

}

Java code is given below.

**public** **interface** User {

}

@Data @NoArgsConstructor @AllArgsConstructor

**public** **class** FulltimeUser **implements** User {

**private** **long** id;

**private** String fullName;

**private** String cityName;

**private** List<String> phoneNos;

}

@Data @NoArgsConstructor @AllArgsConstructor

**public** **class** ContractUser **implements** User {

**private** **long** id;

**private** String fullName;

**private** String cityName;

**private** List<String> phoneNos;

}

**QueryMapping controller is given below**

@QueryMapping

**public** Collection<User> getAllTypesOfUser() {

**return** List.*of*(

**new** ContractUser(123L,"John Abraham","Bangalore", List.*of*("111","222")),

**new** FulltimeUser(456L,"Vidya Balan","Chennai", List.*of*("6666","999"))

);

}

Actual GraphQL Query is given below

**query** GetAllTypesOfUser {

    getAllTypesOfUser {

        fullName

        cityName

        phoneNos

    }

}

**Response**

{

    "data": {

        "getAllTypesOfUser": [

            {

                "fullName": "John Abraham",

                "cityName": "Bangalore",

                "phoneNos": [

                    "111",

                    "222"

                ]

            },

            {

                "fullName": "Vidya Balan",

                "cityName": "Chennai",

                "phoneNos": [

                    "6666",

                    "999"

                ]

            }

        ]

    }

}

**What is GraphQL scalars**

GraphQL scalars are atomic (indivisible) values, meaning that they can only be selected as a whole and cannot be divided into multiple fields.

In GraphQL, a scalar is a primitive data type that represents a leaf value in a query (i.e., a value that cannot contain sub-fields). While GraphQL provides default scalars like Int, String, Boolean, Float, and ID, you can define custom scalars to handle specialized data types (e.g., Date, JSON, or UUID). How to add scalars.

@Configuration

public class GraphQlConfiguration {

@Bean

public RuntimeWiringConfigurer runtimeWiringConfigurer() {

**return wiringBuilder -> wiringBuilder.scalar(ExtendedScalars.Date)**

**.scalar(ExtendedScalars.Url)**

**.scalar(DurationSecondsScalar.INSTANCE);**

}

}

**Explain the concept of resolvers in GraphQL**

Resolvers in GraphQL are functions responsible for fetching the data for each field in a GraphQL schema. They resolve the values for the fields by executing the appropriate logic, such as querying a database, calling an external API, or computing values on the fly.

Each field in a GraphQL schema can have a resolver associated with it. When a query is executed, the GraphQL engine invokes the relevant resolvers to resolve the data for the requested fields. Resolvers can be asynchronous and may return Promises or use other techniques to handle asynchronous operations.

Resolvers are an essential part of GraphQL server implementations and play a crucial role in retrieving and manipulating the data for a GraphQL API.

**What is introspection in GraphQL, and how is it useful**

Introspection in GraphQL refers to the ability of a GraphQL server to provide information about its schema and types at runtime. It allows clients to query the server's schema and discover the available fields, types, arguments, and directives. This feature is built into GraphQL and is used by various tools and libraries.

**How do handle Void Return Type**

You cannot return nothing. You can define a return type which is nullable e.g.

type Mutation {

addElement(element: ElementData): ID

removeElement(id: ID): Boolean

}

**How to access graphiql**

provide the following property in application.properties

**spring.graphql.graphiql.enabled=true**

Access the URL in Browser: [**http://localhost:8081/graphiql?path=/graphql**](http://localhost:8081/graphiql?path=/graphql)