**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.**