**GraphQL Overview – 2025**

**Types**

Types define structure of the data that can be queried from GraphQL API. Each type represents a different object in your application, such as Product, User, or Order.

**type User {**

**id: ID!**

**name: String!**

**email: String!**

**orders: [Order!]!**

**}**

**type Product {**

**id: ID!**

**name: String!**

**description: String**

**price: Float!**

**inStock: Boolean!**

**}**

**Queries**

Queries fetch data from your GraphQL API. They are analogous to GET requests in REST.

**GraphQL**

**query {**

**products {**

**name**

**price**

**}**

**}**

**Mutations**

Mutations modify data on the server. They are analogous to POST, PUT, PATCH, or DELETE requests in REST. A mutation specifies the changes to make, for example:

**GraphQL**

**mutation {**

**addProduct(name: "New Product", price: 19.99) {**

**id**

**name**

**price**

**}**

**}**

**Subscriptions**

Subscriptions are used to listen for real-time updates from your GraphQL API. They enable real-time communication by allowing clients to receive data updates when specific events or changes occur. This example subscription listens for updates about newly added products:

**GraphQL**

**subscription {**

**productAdded {**

**id**

**name**

**price**

**}**

**}**

**GraphQL VS REST API**

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**/**graphql**/**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.

**Examples on GraphQL given below**

An example on schema.graphqls is given below.

**type Query {**

**getAppUserById(id: ID): AppUser**

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

**getAllTypesOfUser: [User]**

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

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

**}**

**type Mutation {**

**createNewAppUser(firstName: String, lastName: String): AppUser**

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

**updateAppUserByAddress(inUser: AppUserInput): String**

**deleteAppUser(userId: String): AppResult # How to get Request Header before removing**

**}**

**"""**

**How to Define Input Object Type**

**"""**

**input AppUserInput {**

**firstName: String**

**lastName: String**

**mStatus: MaritalStatus**

**status: Boolean**

**salary: Int**

**cityName: String**

**pinCode: String**

**phoneNos: [String]**

**bankName: String**

**accountNo: String**

**}**

**type AppUser {**

**id: ID**

**firstName: String**

**lastName: String**

**mStatus: MaritalStatus**

**status: Boolean**

**salary: Int**

**adrs: Address**

**phoneNos: [String]**

**bankAccounts: [BankAccount]**

**}**

**type BankAccount {**

**bankName: String**

**accountNo: String**

**}**

**"""**

**How to Define Enum in java**

**"""**

**enum MaritalStatus {**

**MARRIED**

**UNMARRIED**

**SINGLE**

**}**

**type Address {**

**cityName: String**

**pinCode: String**

**}**

**type AppResult {**

**result: String**

**}**

**"""**

**Below is the case for @SchemaMapping, but in higher version, it is not required.**

**The below type works fine.**

**"""**

**type Author {**

**id: ID**

**name: String**

**books: [Book]**

**}**

**type Book {**

**isbnNo: ID**

**name: String**

**authors: [Author]**

**}**

**Example on Mutation and @MutationMapping in GraphQL**

Example-1: How to provide input parameters to create data in server

**schema.graphqls**

type **Mutation** {

**createNewAppUser**(firstName: String, lastName: String): AppUser

}

**Controller class**

@Controller

**public** **class** AppUserController {

@MutationMapping

**public** AppUser createNewAppUser(@Argument String firstName, @Argument String lastName) {

AppUser newUser = **new** AppUser();

newUser.setFirstName(firstName);

newUser.setLastName(lastName);

**return** newUser;

}

}

GraphQL Query to create new user is given below.

**mutation** CreateNewAppUser {

    createNewAppUser(firstName: "John", lastName: "Abraham") {

        id

        firstName

        lastName

    }

}

**Response**

**{**

**"data": {**

**"createNewAppUser": {**

**"id": "0",**

**"firstName": "John",**

**"lastName": "Abraham"**

**}**

**}**

**}**

**Example-2**: How to input large data with **input object type** to create data in server

**schema.graphqls**

type Mutation {

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

}

type AppUser {

id: ID

firstName: String

lastName: String

mStatus: MaritalStatus

status: Boolean

salary: Int

adrs: Address

phoneNos: [String]

bankAccounts: [BankAccount]

}

input **AppUserInput** {

firstName: String

lastName: String

mStatus: MaritalStatus

status: Boolean

salary: Int

cityName: String

pinCode: String

phoneNos: [String]

bankName: String

accountNo: String

}

**Controller class**

@MutationMapping

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

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

AppUser appUser = AppUtil.*getDefaultUser*(Long.*valueOf*(123));

appUser.setPhoneNos(inUser.getPhoneNos());

**return** appUser;

}

**Graphql query**

**Response**

{

    "data": {

        "createAppUser": {

            "id": "123",

            "firstName": "John",

            "lastName": "Abraham",

            "mStatus": "SINGLE",

            "status": **true**,

            "salary": 2345,

            "phoneNos": [

                "111",

                "222"

            ]

        }

    }

}

**mutation** CreateAppUser {

    createAppUser(

        inUser: {

            firstName: "John"

            lastName: "Abraham"

            mStatus: MARRIED

            status: **true**

            phoneNos: ["111", "222"]

        }

    ) {

        id

        firstName

        lastName

        mStatus

        status

        salary

        phoneNos

    }

}

**Example on Query and @QueryMapping**

**schema.graphqls**

type Query {

getAppUserById(id: ID): AppUser

}

**Controller class**

@QueryMapping

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

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

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

}

**Response**

**{**

**"data": {**

**"getAppUserById": {**

**"id": "123",**

**"firstName": "John"**

**}**

**}**

**}**

**graphql query**

**query** GetAppUserById {

    getAppUserById(id: "123") {

        id

        firstName

    }

}

**Example-2:** To get list of appusers

**schema.graphqls**

type Query {

getAllAppUsers: [AppUser] # To get list of AppUser

}

**Controller class**

@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;

}

**Graphql query**

**query** GetAllAppUsers {

    getAllAppUsers {

        id

        firstName

        lastName

        mStatus

        status

        salary

        phoneNos

    }

}

Response

{

    "data": {

        "getAllAppUsers": [

            {

                "id": "123",

                "firstName": "John",

                "lastName": "Abraham",

                "mStatus": "SINGLE",

                "status": **true**,

                "salary": 2345,

                "phoneNos": [

                    "PhoneNo-1",

                    "Phone-No-2"

                ]

            },

            {

                "id": "4238206656399",

                "firstName": "zpTONjN3ac",

                "lastName": "83Fqb9z2hF",

                "mStatus": "SINGLE",

                "status": **true**,

                "salary": 2139615936,

                "phoneNos": [

                    "FUKPr7z6QO",

                    "fhVR30DzYS",

                    "Aj\_1mBe9ci",

                    "SzPo4eGQN8",

                    "iSVm2CCMOM"

                ]

            }

        ]

    }

}

**Example-3**: Send the list of phone and receive

**schema.graphqls**

type Query {

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

}

**Controller Class**

@QueryMapping

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

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

}

**Graphql query**

**query** GetAllPhoneNosByNames {

    getAllPhoneNosByNames(names: ["111","222"])

}

**Response**

{

    "data": {

        "getAllPhoneNosByNames": [

            "111",

            "222",

            "333"

        ]

    }

}

**Complete Example on GraphQL with SpringBoot Version 3.3.4**

**Relevant pom.xml ()**

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-graphql</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-jpa</artifactId>

</dependency>

<dependency>

<groupId>org.postgresql</groupId>

<artifactId>postgresql</artifactId>

<scope>runtime</scope>

</dependency>

<dependency>

<groupId>org.projectlombok</groupId>

<artifactId>lombok</artifactId>

<optional>true</optional>

</dependency>

</dependencies>

**application.properties**

spring.datasource.url=jdbc:postgresql://localhost:5432/c3\_test\_db\_admin

spring.datasource.username=postgres

spring.datasource.password=Abcd@1234

spring.jpa.properties.hibernate.default\_schema=public

spring.jpa.database=POSTGRESQL

spring.datasource.driverClassName=org.postgresql.Driver

spring.jpa.database-platform=org.hibernate.dialect.PostgreSQLDialect

# Do not use below in higher version of Spring Boot, hibernate will not be able to create the sequence

#spring.jpa.properties.hibernate.id.db\_structure\_naming\_strategy=legacy

# Hibernate ddl auto (create, create-drop, validate, update)

spring.jpa.hibernate.ddl-auto = update

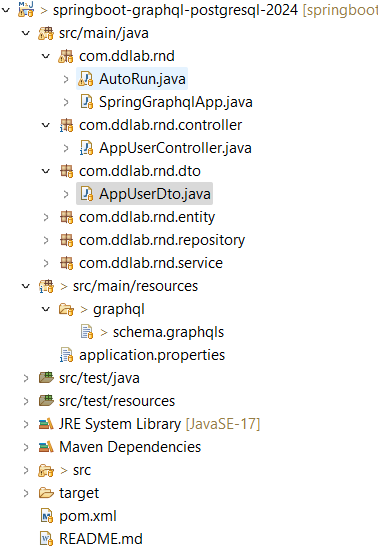
#spring.jpa.properties.hibernate.generate\_statistics=true

spring.jpa.show-sql=true

spring.jpa.properties.hibernate.format\_sql=true

spring.jpa.properties.hibernate.temp.use\_jdbc\_metadata\_defaults=false

**Project Structure**



Create a directory named **graphql** inside **src/main/resources.** Create a file called **schema.graphqls.**

The contents are given below.

type Query {

getAppUserById(id: ID): AppUser

}

# Input types

input AppUserDto {

firstName: String

lastName: String

nickName: String

shortDesc: String

}

type Mutation {

createAppUser(firstName: String, lastName: String, nickName: String, shortDesc: String): String

createNewAppUser(appUserDto: AppUserDto): AppUser

updateUser(id: ID, nickName: String): AppUser

removeUser(id: ID): String

# upload(file: Upload): UploadedFile

}

type AppUser {

id: ID

firstName: String

lastName: String

nickName: String

shortDesc: String

}

Controller class (**AppUserController**) is given below.

@RestController

**public** **class** AppUserController {

@Autowired

**private** AppUserServiceImpl serviceImpl;

**@QueryMapping**

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

**return** serviceImpl.getAppUserById(id).get();

}

**@MutationMapping**

**public** AppUser updateUser(@Argument **long** id, @Argument String nickName) {

**return** serviceImpl.updateAppUser(id, nickName);

}

**@MutationMapping**

**public** String removeUser(@Argument **long** id) {

System.***out***.println("Trying to delete the user ...");

**return** serviceImpl.deleteById(id);

}

**@MutationMapping**

**public** AppUser createNewAppUser(@Argument(name = "appUserDto") AppUserDto appUserDto) {

System.***out***.println("appUserDto: "+appUserDto);

AppUser user = **new** AppUser();

user.setFirstName(appUserDto.getFirstName());

user.setLastName(appUserDto.getLastName());

user.setNickName(appUserDto.getNickName());

user.setShortDesc(appUserDto.getShortDesc());

**return** serviceImpl.createNewAppUser(user);

}

**@MutationMapping**

**public** String createAppUser(

**@Argument String firstName,**

**@Argument String lastName,**

**@Argument String nickName,**

**@Argument String shortDesc**) {

AppUser appUser = **new** AppUser();

appUser.setFirstName(firstName);

appUser.setLastName(lastName);

appUser.setNickName(nickName);

appUser.setShortDesc(shortDesc);

**return** serviceImpl.createAppUser(appUser);

}

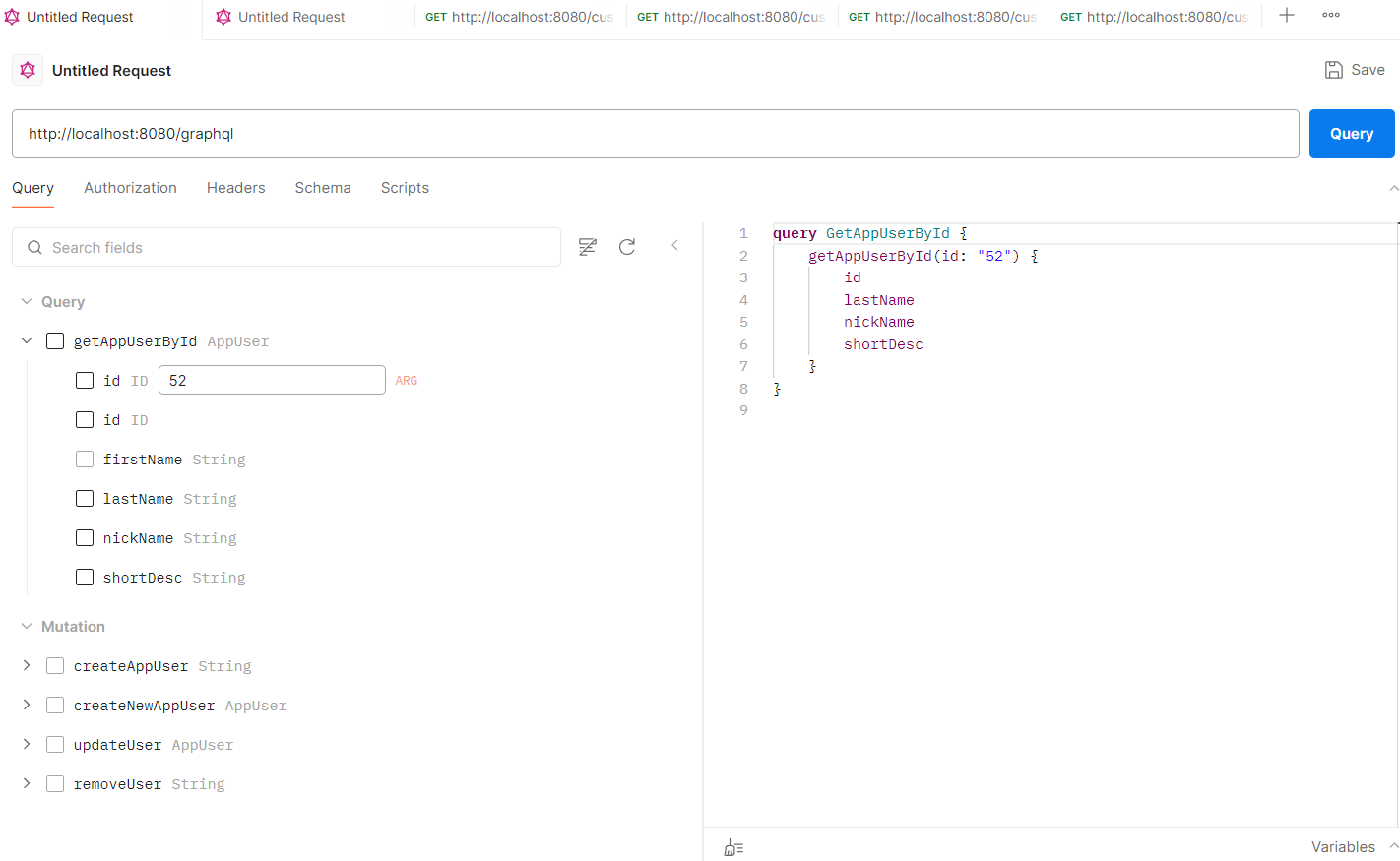
}

How to test GraphQL application using Postman client ?

You have to Postman client for GraphQL based application.

Open GraphQL in Postman client and hit the URL: <http://localhost:8080/graphql>

It will display all the Query and Mutation as shown below.



To test getAppUserById(@Argument **long** id), select the getAppUserById in postman client and enter the id as shown below.

A computer screen shot of a computer code

Description automatically generated

Once you hit the query button in postman client, you will get the following response.

{

    "data": {

        "getAppUserById": {

            "id": "52",

            "lastName": "Signh",

            "nickName": "Miki",

            "shortDesc": "A very good Rabbit"

        }

    }

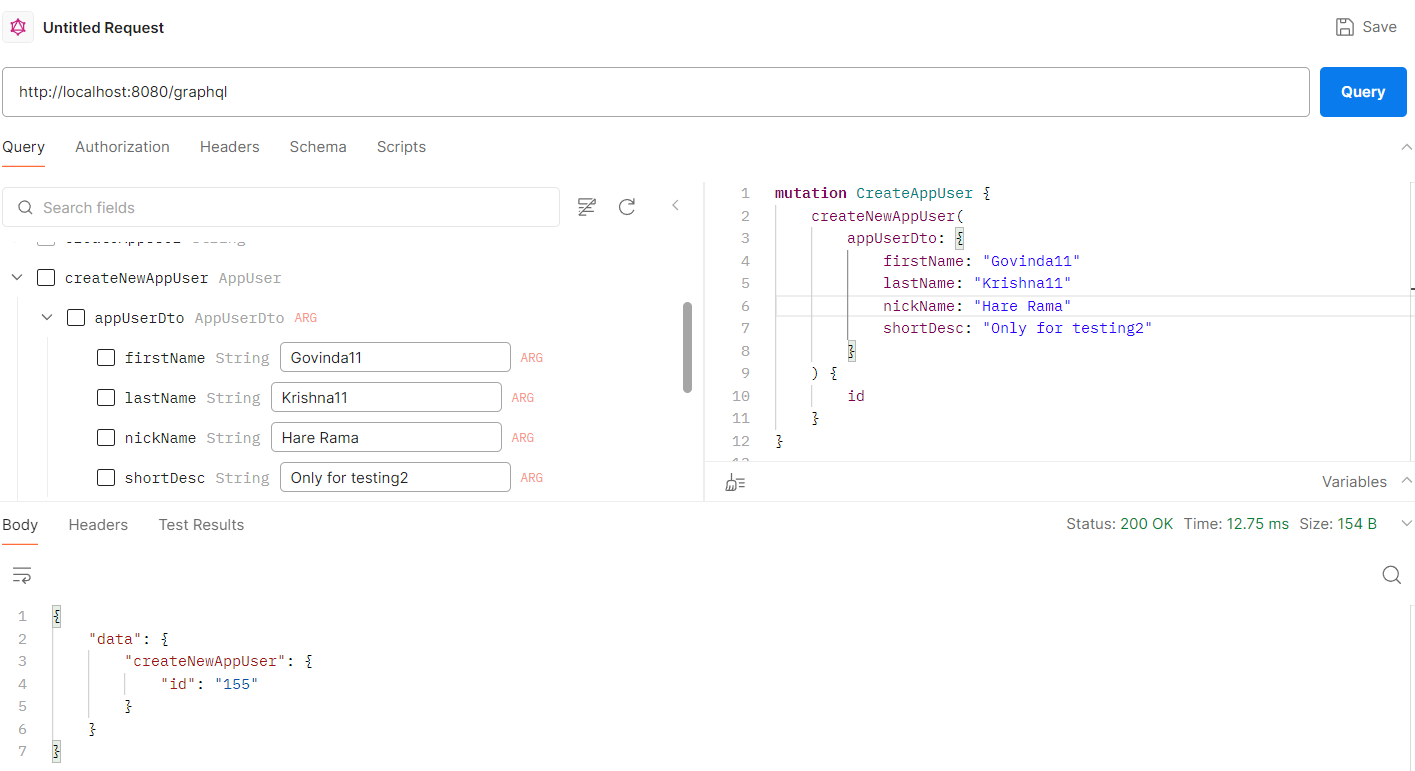
}

To Test the following scenario

@MutationMapping

**public** AppUser createNewAppUser(@Argument(name = "appUserDto") AppUserDto appUserDto) {}

Select createNewAppUser in PostMan client and fill up the fields as shown below and then hit the query button.



The response is given below.

{

    "data": {

        "createNewAppUser": {

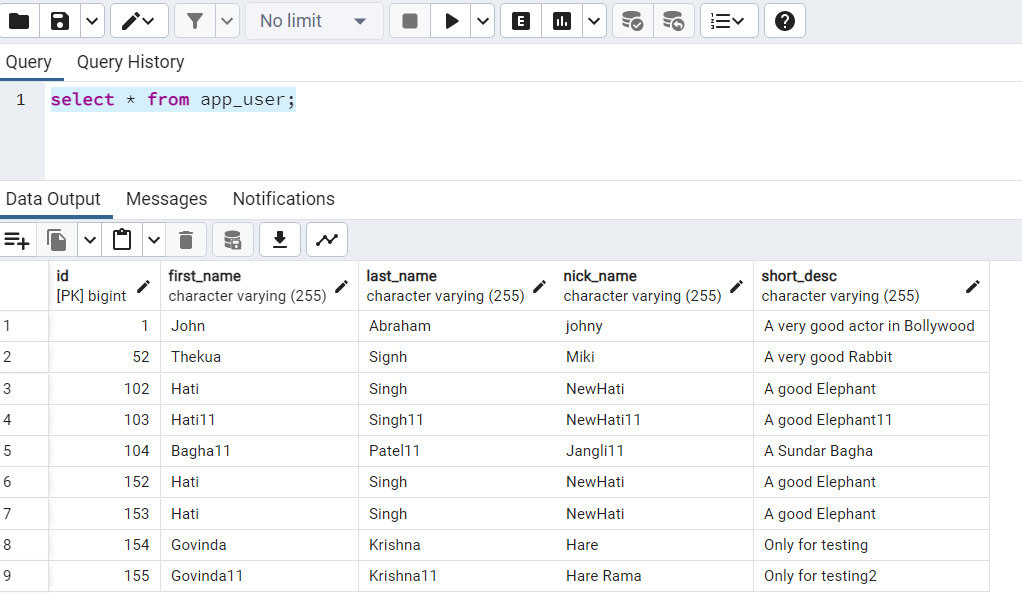
            "id": "155"

        }

    }

}

Structure of Database table is given below.



Similarly, you can verify other functionalities with the same end point.