**GraphQL Overview – 2025**

**Types**

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

**type Product {**

**id: ID!**

**name: String!**

**description: String**

**price: Float!**

**inStock: Boolean!**

**}**

**type User {**

**id: ID!**

**name: String!**

**email: String!**

**orders: [Order!]!**

**}**

**Queries**

Queries fetch data from your GraphQL API. They are analogous to GET requests in REST. A query specifies the shape and structure of the data to retrieve. For example:

**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 schema** defines which types and fields your clients can query. Your GraphQL server's resolvers define how those types and fields are populated from your data sources.

**Advantages**

GraphQL is way faster than other communication APIs because it facilitates you to cut down your request query by choosing only the specific fields you want to query.

The main advantage of GraphQl over REST is that REST responses contain too much data or sometimes not enough data, which creates the need for another request. GraphQL solves this problem by fetching only the exact and specific data in a single request.

GraphQL follows a hierarchical structure where relationships between objects are defined in a graphical structure.

**Examples on GraphQL given below**

**@MutationMapping**

public String createAppUser(

@Argument String firstName,

@Argument String lastName,

@Argument String nickName,

@Argument String shortDesc) {}

**Body**

mutation CreateAppUser {

createAppUser(

firstName: "Hati"

lastName: "Ghoda"

nickName: "HatiGhoda"

shortDesc: "Jungle Book"

)

}

The URI is always the same : <http://localhost:8080/graphql>

It means that there is always only one end point.

**Create New User**

mutation CreateNewAppUser {

createNewAppUser(

appUserDto: {

firstName: "Thekua"

lastName: "Signh"

nickName: "Miki"

shortDesc: "A very good Rabbit"

}

) {

id

firstName

lastName

nickName

shortDesc

}

}

**Update User**

@MutationMapping

public AppUser updateUser(

@Argument long id,

@Argument String nickName)

{

return serviceImpl.updateAppUser(id, nickName);

}

mutation UpdateUser {

updateUser(id: "2", nickName: "Some New Name") {

id

firstName

lastName

nickName

shortDesc

}

}

**Remove/Delete User**

@MutationMapping

public String removeUser(@Argument long id) {

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

return serviceImpl.deleteById(id);

}

mutation RemoveUser {

removeUser(id: "2")

}

**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

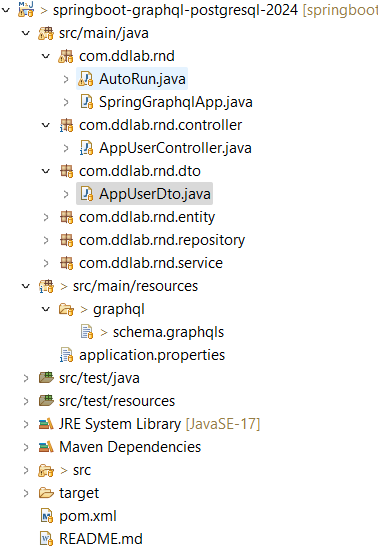
#spring.jpa.properties.hibernate.generate\_statistics=true

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.

#scalar Upload

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

}

#type UploadedFile {

# filename: String

# type: String

# content: 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);

}

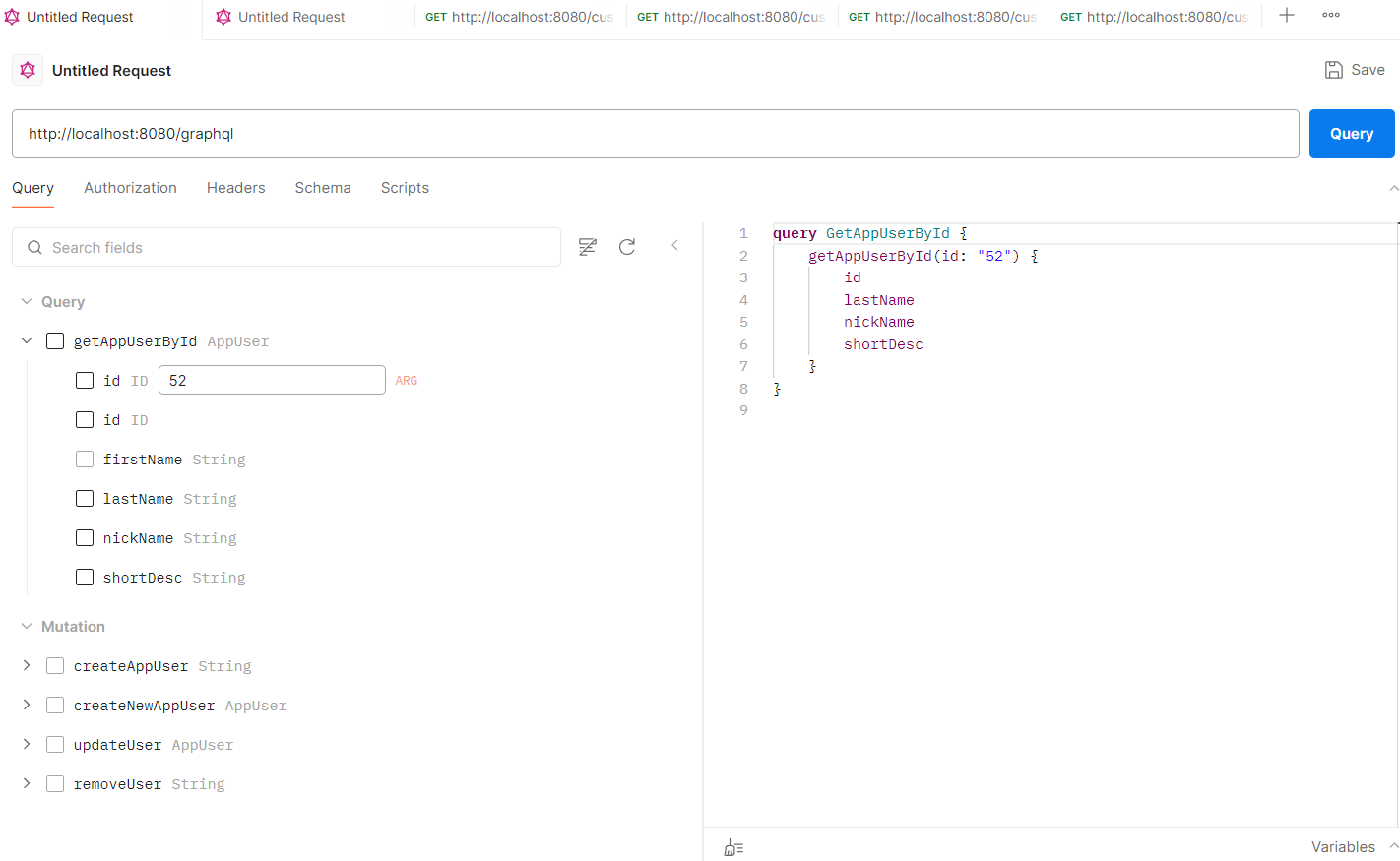
}

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"

        }

    }

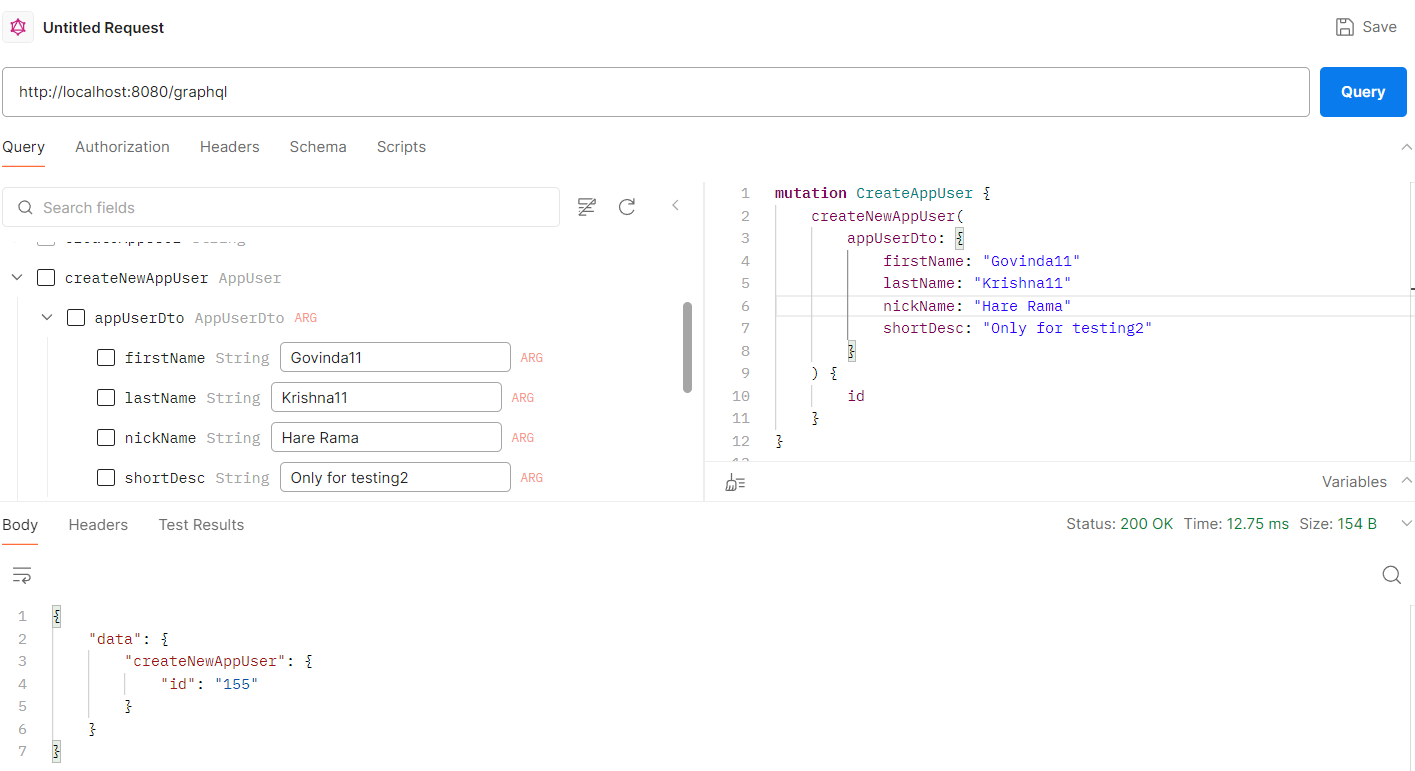
}

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": {

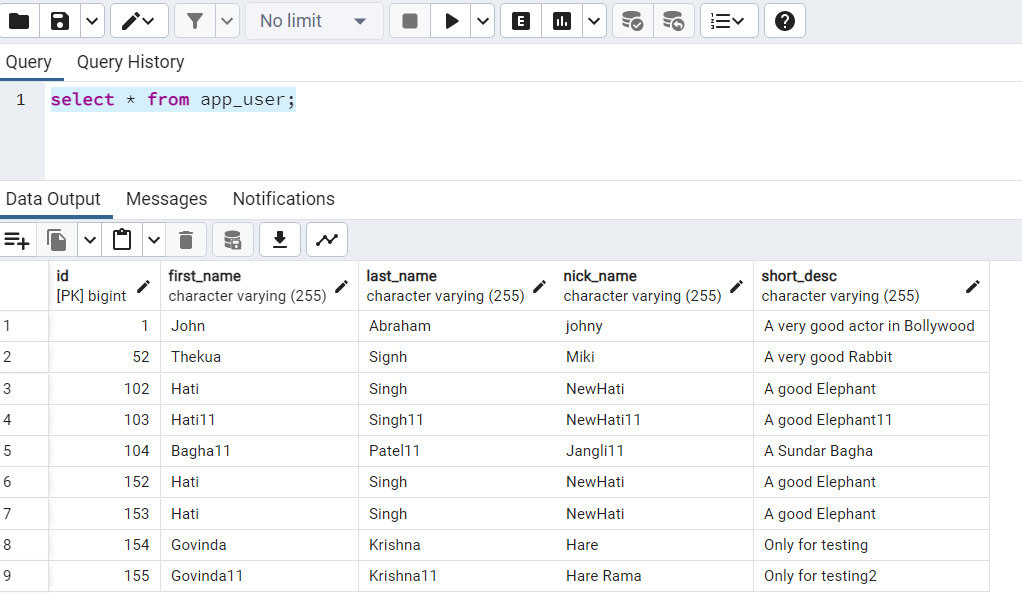
            "id": "155"

        }

    }

}

Structure of Database table is given below.



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

**SpringBoot GraphQL Client – 2025**

This is about how to access SpringBoot GraphQL based application using SpringBoot.

**Project Structure**

A screenshot of a computer

Description automatically generated

**application.properties**

server.port=9090

**Relevant Portion of 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-webflux</artifactId>** </dependency>  
 <dependency>  
 <groupId>org.projectlombok</groupId>  
 <artifactId>lombok</artifactId>  
 <optional>true</optional>  
 </dependency>  
</dependencies>

Create a directory named **graphql-documents** inside **src/main/resources** and create a file called **getAppUserById.graphql**. Just check whether it is required or not. The contents are given below.

query GetAppUserById($slug: ID!) {  
 getAppUserById(id: $slug) {  
 id  
 firstName  
 lastName  
 nickName  
 shortDesc  
 }  
}

**AutoRun class is given below for more details**

@Component

**public** **class** AutoRun {

**public** **void** makeMutationQuery() {

WebClient webClient = WebClient.*create*("http://localhost:8080/graphql");

HttpGraphQlClient graphQlClient11 = HttpGraphQlClient.*builder*(webClient).build();

String mutationQuery = """

mutation CreateAppUser {

createAppUser(

firstName: "Narottam"

lastName: "Mishra"

nickName: "Naraha"

shortDesc: "A Sundar Manisha"

)

}

""";

String response = graphQlClient11.document(mutationQuery).retrieve("createAppUser")

.toEntity(String.**class**).block();

System.***out***.println("Response: " + response);

}

**public** **void** makeQuery1() {

WebClient webClient = WebClient.*create*("http://localhost:8080/graphql");

HttpGraphQlClient graphQlClient = HttpGraphQlClient.*builder*(webClient)

.build();

String query = """

query GetAppUserById {

getAppUserById(id: "52") {

id

firstName

lastName

nickName

shortDesc

}

}

""";

AppUser appUser = graphQlClient.document(query).retrieve("getAppUserById")

.toEntity(AppUser.**class**).block();

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

}

**public** **void** makeQuery2() {

WebClient webClient = WebClient.*create*("http://localhost:8080/graphql");

HttpGraphQlClient graphQlClient = HttpGraphQlClient.*builder*(webClient).build();

AppUser country = graphQlClient.documentName("getAppUserById")

.variable("slug", 102).retrieveSync("getAppUserById")

.toEntity(AppUser.class);

System.out.println(country);

}

@EventListener(ApplicationReadyEvent.**class**)

**public** **void** run() {

System.***out***.println("Application started running ...");

**// makeQuery1();**

**// makeQuery2();**

**makeMutationQuery();**

}

}