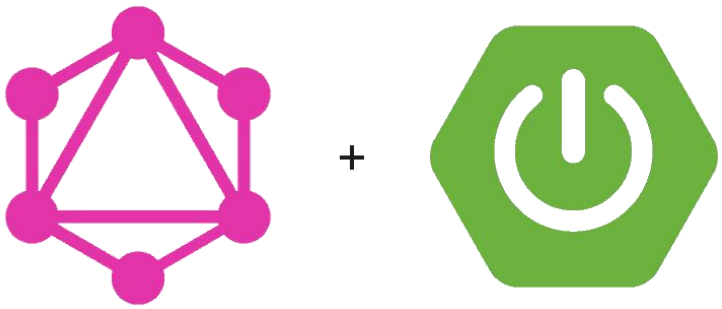
How to implement GraphQL with spring boot ?



Hetasvi Ahir

Rysun

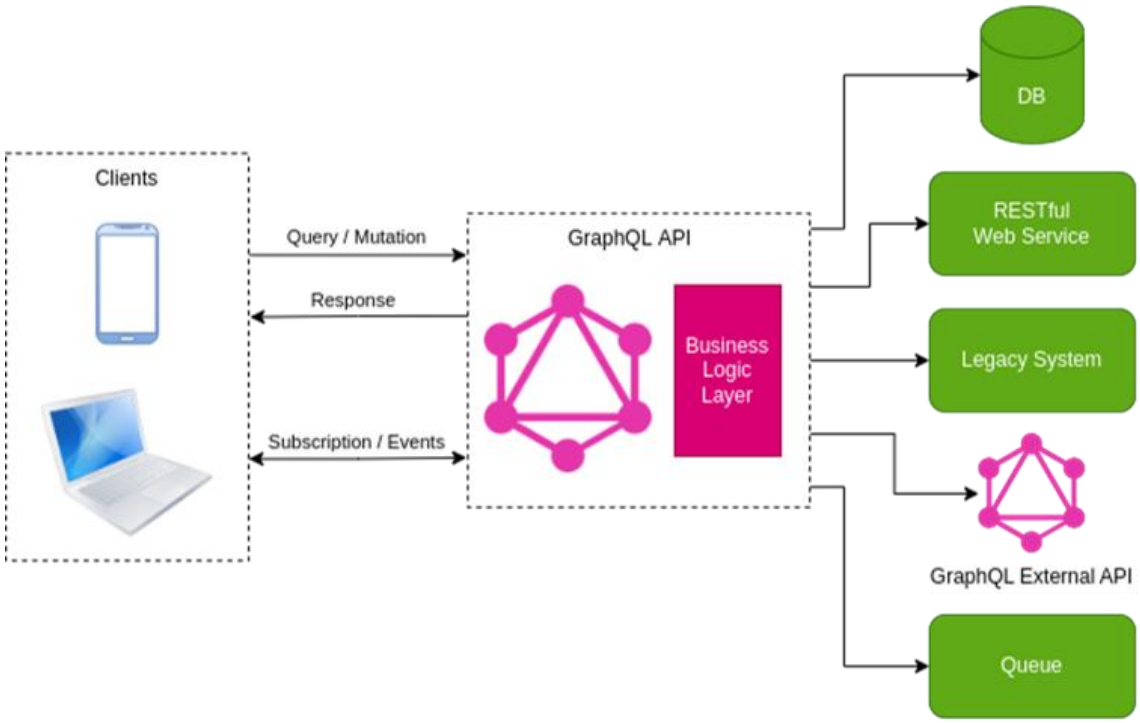
Introduction to GraphQL

- GraphQL is a query language to retrieve data from a server.
- It is an alternative to REST, SOAP or GRPC.
- GraphQL isn't tied to any specific database or storage engine and is instead backed by your existing code and data.
- Written down in Schema Definition Language (SDL)
- GraphQL is designed to make APIs better, faster, and more developer-friendly.
- It allows clients to define the structure of the data they need, and the server will return only the requested data.
- GraphQL allows for more flexible and efficient communication between the client and the server.
- It also allows for real-time updates and can be integrated with existing systems.
- 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.
- GraphQL queries access not just the properties of one resource but also smoothly follow references between them. While typical REST APIs require loading from multiple URLs, GraphQL APIs get all the data your app needs in a single request.
- Apps using GraphQL can be quick even on slow mobile network connections.
- GraphQL APIs are organized in terms of types and fields, not endpoints.
- Access the full capabilities of your data from a single endpoint.
- GraphQL uses types to ensure Apps only ask for what’s possible and provide clear and helpful errors.
- Apps can use types to avoid writing manual parsing code.

Why do we use GraphQL ?

- GraphQL simplifies the workflow to build client applications such as iOS, Android, React-Native.
- It helps clients fetch the right amount of data needed to render the view.
- GraphQL lets clients define the shape of the response for each request.
- It removes the complexity of API Endpoint management on these clients, as it exposes a single HTTP Endpoint (and is usually /graphql) to fetch the required data.

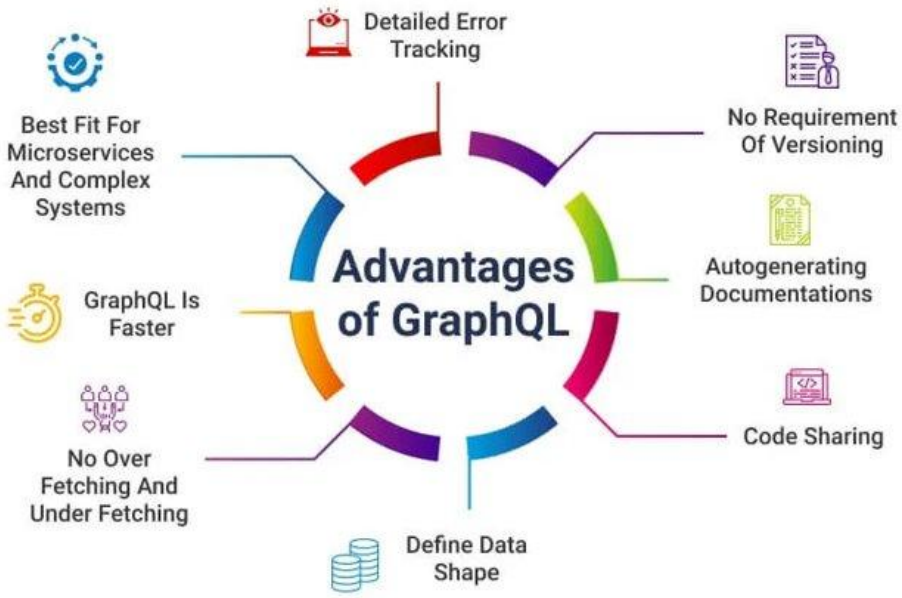
Workflow of GraphQL :



REST VS GRAPHQL

GraphQL	REST
Client-driven	Server-Driven
Fast performance	Multiple network calls take up more time
Available Operations : Query , Mutation and Subscription	Operations : Create , Read , Update , Delete
Query can become very complex	Queries are Simple
It is still growing in popularity	Very popular among people
It doesn't support file uploading	Supports file uploading
Uses Web library in web caching	Inbuilt web caching available.
Used in : Multiple microservices and mobile applications	Used in : Simple apps

Advantages :



Limitations :

- Query Complexity
- Queries always return HTTP status code 200.
- GraphQL doesn't understand Files.
- Understanding and Implementing queries and mutation is not a easy task.

Based on the graph data modeling with the schema at its core, GraphQL has three primary operations:

1. Query for reading data
2. Mutation for writing data
3. Subscription for automatically receiving real-time data over time.

Things to remember :

1. GraphQL service is created by defining types and fields on those types, then providing functions for each field on each type.
2. Most types in your schema will just be normal object types, but there are two types that are special within a schema:

```
schema {
  query: Query
  mutation: Mutation
}
```

3. Every GraphQL service has a query type and may or may not have a mutation type. These types are the same as a regular object type, but they are special because they define the entry point of every GraphQL query.
4. The GraphQL schema language supports the scalar types of : String , Int, Float ,Boolean,ID
5. Use an exclamation point to indicate a type can not be nullable , So String! Is a non-nullable String.
6. To use a list type , surround the type in square brackets , so [Int] is a list of integers.

Dependency :

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

Annotations :

I. @QueryMapping - For Get requests

II. @SchemaMapping - For Post requests

III. @MutationMapping - For Post requests

IV. @Argument - Parameter passing /Request Body

V. @MutationMapping – For Post/Delete/Put requests

VI. @SubscriptionMapping - For reactive method calls

Steps to create a GraphQL Project to use it with your existing spring boot application using OAuth 2 Authentication :

We will have 3 Projects :

1. Main application having CRUD operation of Book and Author .
2. Authorization Server
3. GraphQL server - From which we will call APIs of our main application

Authorization Server configuration :

Step 1 : Create new project .

Dependency :

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-security</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.security</groupId>
        <artifactId>spring-security-oauth2-authorization-server</artifactId>
        <version>1.0.0</version>
    </dependency>
    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <optional>true</optional>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.springframework.security</groupId>
        <artifactId>spring-security-test</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>
```

Step 2 : Create Authorization server Config .

```
@Configuration
public class AuthorizationServerConfig {
@Bean
@Order(Ordered.HIGHEST_PRECEDENCE)
public SecurityFilterChain authServerSecurityFilterChain(HttpSecurity http) throws
Exception {
	OAuth2AuthorizationServerConfiguration.applyDefaultSecurity(http);
	return http.build();
}
@Bean
public PasswordEncoder passwordEncoder() {
	return NoOpPasswordEncoder.getInstance();
}
@Bean
public RegisteredClientRepository registeredClientRepository() {
	RegisteredClient registeredClient
	= RegisteredClient.withId(UUID.randomUUID().toString()).clientId("client")
	.clientSecret("secret")
	.clientAuthenticationMethod(ClientAuthenticationMethod.CLI ENT_SECRET_BASIC)
	.authorizationGrantType(AuthorizationGrantType.CLIENT_CREDENTIAL  S)
	.scope(OidcScopes.OPENID).scope("admin.write")
	.tokenSettings(TokenSettings.builder()
	.accessTokenTimeToLive(Duration.ofMinutes(50l)).build()).build();
	return new InMemoryRegisteredClientRepository(registeredClient);
}
@Bean
public AuthorizationServerSettings authorizationServerSettings() {
	return AuthorizationServerSettings.builder().build();
}
@Bean
public JwtDecoder jwtDecoder(JWKSource<SecurityContext> jwkSource) {
	return OAuth2AuthorizationServerConfiguration.jwtDecoder(jwkSource);
}
@Bean
public JWKSource<SecurityContext> jwkSource() {
	RSAKey rsaKey = generateRsa();
	JWKSet jwkSet = new JWKSet(rsaKey);
	return (jwkSelector, securityContext) -> jwkSelector.select(jwkSet);
}
public static RSAKey generateRsa() {
	KeyPair keyPair = generateRsaKey();
	RSAPublicKey publicKey = (RSAPublicKey) keyPair.getPublic();
	RSAPrivateKey privateKey = (RSAPrivateKey) keyPair.getPrivate();
	return new RSAKey.Builder(publicKey).privateKey(privateKey)
	.keyID(UUID.randomUUID().toString()).build();
}
```

```
static KeyPair generateRsaKey() {
	KeyPair keyPair;
	try {
		KeyPairGenerator keyPairGenerator =
		KeyPairGenerator.getInstance("RSA");
		keyPairGenerator.initialize(2048);
		keyPair = keyPairGenerator.generateKeyPair();
	} catch (Exception ex) {
		throw new IllegalStateException(ex);
	}
		return keyPair;
	}
}
```

Now We have to create **GraphQL server** which will call our APIs from our main application .

Step 1 : Create new project .

Dependency :

```
<dependencies>
	<dependency>
		<groupId>org.springframework.boot</groupId>
		<artifactId>spring-boot-starter-graphql</artifactId>
	</dependency>
	<dependency>
		<groupId>com.graphql-java</groupId>
		<artifactId>graphql-java-spring-boot-starter-webmvc</artifactId>
		<version>2019-06-24T11-47-27-31ab4f9</version>
	</dependency>
	<dependency>
		<groupId>org.springframework.boot</groupId>
		<artifactId>spring-boot-starter-web</artifactId>
	</dependency>
	<dependency>
		<groupId>org.springframework.boot</groupId>
		<artifactId>spring-boot-starter-oauth2-resource-server</artifactId>
	</dependency>
	<dependency>
		<groupId>org.projectlombok</groupId>
		<artifactId>lombok</artifactId>
		<optional>true</optional>
	</dependency>
	<dependency>
		<groupId>org.springframework.graphql</groupId>
		<artifactId>spring-graphql-test</artifactId>
		<scope>test</scope>
	</dependency>
</dependencies>
```

Step 2 : Create Schema files .

- Schema.graphqls

```
schema {
query: Query
mutation: Mutation
}
type Query {
}
type Mutation {
}
```

- Book.graphqls

```
extend type Query {
	allBooks:[BookDto]
	getBook(bookId:Int):BookDto
}
extend type Mutation {
	createBook(book:BookInput):BookDto
	updateBook(input:UpdateInput):BookDto
	deleteBook(bookId:Int):String
}
type BookDto
{
	id:ID
	title:String
	description:String
	#author:String
	price:Int
	pages:Int
	author:AuthorDto
}
input BookInput
{
	title:String
	description:String
	author:Int
	price:Int
	pages:Int
}
input UpdateInput
{
	id:Int
	title:String
	price:Int
	pages:Int
}
```

- Author.graphqls

```
extend type Mutation{
	createAuthor(author:AuthorInput):AuthorDto
	updateAuthor(input:AuthorUpdateInput):AuthorDto
	deleteAuthor(id:Int):String
}
extend type Query{
	getAuthors:[AuthorDto]
	getAuthor(id:Int):AuthorDto
}
input AuthorInput
{
	name:String
}
input AuthorUpdateInput
{
	id:Int
	name:String
}
type AuthorDto
{
	id:ID!
	name:String
}
```

Step 3 : Create one controller in which we are using rest teamplate to call our API of main application for Book operations .

Step 4 : Create one controller in which we are using rest teamplate to call our API of main application for Author operations .

Step 5 : Create Resource Server Config file .

```
1  package com.query.config;
2
3  import org.springframework.context.annotation.Bean;
7
8  @EnableWebFluxSecurity
9  public class ResourceServerConfig {
10
11     @Bean
12     SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
13         http.authorizeRequests().requestMatchers("/graphql/**").access("hasAuthority('SCOPE_admin.write')").and()
14                 .oauth2ResourceServer().jwt().and();
15
16         return http.build();
17     }
18 }
19
```

Step 6 : Add properties in application.properties file.

Here issuer-uri is of Authorization Server

```
server.port= 9091
spring.security.oauth2.resourceserver.jwt.issuer-uri=http://localhost:9090
```

Postman Enpoint will be :

Generate token endpoint :

http://localhost:9090/oauth2/token

GraphQL endpoint :

localhost:8080/graphql

Query and Mutations for Postman request of Book CRUD

Save Book :

```
mutation{
    createBook(book:{
        title :"Java"
        description:"Hello World "
        author : 2
        price : 45211
        pages : 2021

    })
    {
        id
        title
        pages
        author{
            id
            name
        }
    }
}
```

Get All Books :

```
query{
    allBooks{
        id
        title
        description
        price
        author{
            id
            name
        }
    }
}
```

Get Book by Id :

```
query{
    getBook(bookId:2){
        id
        title
    }
}
```

Update Book :

```
mutation{
    updateBook(input:{
        id : 2
        title :"My new book 456"
        price : 4521
        pages : 10000

    })
```

	{
	id
	title
	pages
	price
	}
	}

Delete Book :
mutation{
deleteBook(bookId:2)
}	

Query and Mutation for Author CRUD Operation .

Create Author :
mutation{
createAuthor(author:{
name: "Kinjal"
})
{
id
name
}
}
	Get All Author :
	query{
	getAuthors{
	id
	name
	}
	}

Delete Author :
mutation{
deleteAuthor(id:4)
}	