GraphQL

Mimanshu Maheshwari

December 4, 2024

Contents

List of Figures

Listings

Chapter 1

Why and What is GraphQL?

1.1 Why GraphQL

We all know that REST contributes more towards API development. In recent years, REST has become the dominant style for creating web services that are reusable and manageable. Everything in REST is represented as a resource which in turn has a unique URI to get identified. Also, REST works with a set of HTTP methods/verbs to define the operations being performed on the REST resources. Each one of us, the developers, will accept that REST is a great approach to proceed with API development as

- It leverages the usage of the standard HTTP protocol
- It is stateless
- It supports caching
- It is light-weight
- It supports multiple representations for resources and so on

But, we should agree that REST has its own limitations as well. And, the limitations are

- **Over-fetching**: The client might be interested to get the id and name of a resource, but the REST call will fetch the whole set of fields corresponding to that specific resource.
- **Multiple requests**: in case the client needs multiple data - REST API routes get longer when the data goes increasingly complex.

Because of these complexities, the client devices will slow down when there is limited network bandwidth. One more limitation of REST that needs to be put forth here is, it's important for the client to know the exact location of every single resource - If the API server holds multiple resources named customer, plan and so on, the client is expected to know the absolute URI of all the resources that are available with the server. Do we have anything in hand to overcome these issues?

Yes, we have something called GraphQL to help us with these issues.

1.2 What is GraphQL?

GraphQL is a server-side runtime and a query language for the APIs that we write It was developed by Facebook to optimize the REST calls to meet the growing requirements Later, Facebook open-sourced GraphQL as a specification for building APIs

GraphQL makes it possible to organize our data in the form of a graph and uses a powerful query syntax to traverse over the graph to retrieve and modify the data being stored in the graph.

From the definition above, it is evident that GraphQL is not only a specification for how to build APIs but also a specification for how to access the APIs from the client end. GraphQL queries are declarative by nature.

Now let's take a look at the advantages being offered by GraphQL that this course covers with detailed dem

- No over and under fetching – Ask for what is required and get that alone.
- A single request can help to retrieve multiple resources
- Describes well what's possible with a well-structured type system to help the clients understand the API better
- Move faster - in the sense of rapid API development with the help of powerful developer tools.

Resource utilization is much optimized because of the power of asking for only the things that are required and the ability to get multiple resources in a single hit/request. Hence, the client devices can perform better even on very less network bandwidth.

1.3 Ask for what do you want

We can send GraphQL queries to the APIs and get exactly what we need - nothing more, nothing less. A GraphQL query will always return a predictable result. Applications that use GraphQL are both fast and stable. Unlike RESTful calls, a GraphQL client call can actually restrict data that needs to be retrieved from the server. The following example can help us understand better. Let us consider an object Employee with the attributes - id, firstName, lastName, and technologyName. And, assume that we have a mobile application that needs to fetch the employees' firstName and id alone. If we design a RESTful endpoint for this requirement- `/api/v1/employees`, it just will end up fetching/retrieving data for every single field of the employee objects. This literally means, data is being over-fetched by the RESTful service. This problem can efficiently be addressed with the power of GraphQL. Using GraphQL, we can create a query similar to the one being shown below.

```
{
  employees {
    id
    firstName
  }
}
```

Listing 1.1: Employee fetch id and name only using GraphQL query

Now, this query shall return the values of the `id` and `firstName` fields alone and will ignore other attributes like `lastName` and `technologyName` of the employee object. Observe the response for the above query:

```
{
 "data": {
   "employees": [
     {
       "id": "E1001",
       "firstName": "John"
     },
     {
       "id": "E1002",
       "firstName": "Jane"
     }
   ]
 }
```

```
}
```

Listing 1.2: Employee fetch id and name only response

1.4 Get many resources in a single request

GraphQL queries can help us smoothly retrieve multiple resources in a single request. In contrast to this, a typical REST API might require loading data from multiple URLs. Since the API calls are optimized to a great extent, applications that use GraphQL can perform faster even on a slow mobile network connection! The following example can make us understand this concept even better. Let us consider an object employee that has fields like id, firstName, lastName, jobLevel and companyId. Let us consider another object, Company which has attributes – name, and description. The employee resource can be provided by the endpoint, `/api/v1/employees` and the company resource can be provided by the endpoint, `/api/v1/companies` ↪ . If the client wants both employee and company details, the client should two calls, on to fetch the employee details (`/api/v1/employees`) and the other to fetch the company details (`/api/v1/companies`). This is nothing but under-fetching of data where multiple calls are required to fetch the needed information. Hence, applications are forced to give multiple calls to the server to get the data that is required. However, GraphQL can help an application to fetch details for both Employee and Company in a single request. Observe the below GraphQL query to fetch the employee and the company together.

```
query
{
   employees{firstName,lastName,jobLevel},

   companies{id,name,description}
}
```

Listing 1.3: fetch multiple resource in single query

```
{
  "data": {
    "employees": [
    {
      "firstName": "Steffy",
      "lastName": "Mayor",
      "jobLevel": 5
    }
    {
      "firstName": "Steffy2",
      "lastName": "Mayor2",
      "jobLevel": 6
    }
    ],
    "companies": [
    {
      "id": "C232",
      "name": "Product Company",
      "description": "Tech Ex"
    }
    {
      "id": "C233",
      "name": "Tech Company",
      "description": "Infy"
    }
```

```
        ]
      }
    }
```

Listing 1.4: fetch multiple resource in single query response

1.5 Describe what's possible with a type system

GraphQL is strongly typed. GraphQL queries are based on the fields and their associated data types. If there is a type mismatch in a GraphQL query, the server applications will return error messages that are very clear and much helpful to the clients to introspect further in way smoother. Along with this, GraphQL provides various client-side libraries that actually can help to reduce explicit way of data conversion and parsing.

```
type Query {
  employees: [Employee]
}
type Employee {

  id: ID!
  firstName: String
  lastName: String
  fullName: String
  company: Company
}
type Company {
  id: ID!
  name: String
  description: String
  rating: Float
  employees: [Employee]
}
```

Listing 1.5: Example of GraphQL typing

1.6 Hit the ground running with powerful developer tools

GraphQL offers a decent number of developer tools that can help to document the APIs and testing the queries.

- **GraphiQL** is a wonderful tool that can help us generate documentation for the query and the schema to which the query belongs. It also provides a query editor to test our GraphQL APIs. Apart from these, GraphiQL has an intelligent code completion capability that helps us quickly build our queries.
- **GraphQL Voyager** is yet another tool that can help us represent/depict a GraphQL API as an interactive graph! GraphQL Voyager's motto is – "It's time to finally see the graph behind GraphQL"! It also lets us explore several public GraphQL APIs from its list. It also includes a wide range of examples of GraphQL schemas and allows us to connect them to our own GraphQL endpoints.
- **Apollo SandBox**: A unique Apollo Studio mode which is a web IDE for creating, running and managing graphQL operations called Apollo Sandbox aids with local development.

GraphQL is a server-side runtime and a query language for the APIs.

- No over and under fetching – Ask for what is required and get that alone.

- A single request can help to retrieve multiple resources.
- Describes well what's possible with a well-structured type system to help the clients understand the API better.
- Move faster - in the sense of rapid API development with the help of powerful developer tools.

Chapter 2

GraphQL Application Architecture: An Overview

2.1 Various GraphQL architecture approaches

2.1.1 GraphQL application architecture

GraphQL is just a specification that set out how a GraphQL server should behave. This essentially means, a set of guidelines will be defined by any GraphQL based application - on how requests and responses to be handled by the server.

The guidelines generally include:

- What are the supported protocols?
- Which format of data can be acceptable for the server?
- Which format of response is possible for the server to return? etc.

The GraphQL mindset

The request from a GraphQL client to the GraphQL server is known as Query. GraphQL is transport-layer agnostic. That is, GraphQL actually is agnostic with respect to how data flows over the network. GraphQL server can work with protocols other than HTTP as well. To quote a few, WebSockets and TCP. But, the most widely used one is HTTP indeed. GraphQL is neutral to databases. That is, we are free to use GraphQL with SQL or NoSQL databases or even the combination of both!

Various GraphQL app architecture possibilities

A GraphQL server can be created using either of the methods being listed below –

1. GraphQL server with a connected database
2. GraphQL server that integrates existing (possibly legacy) systems
3. Hybrid approach – which is a mix of the above two approaches

2.1.2 GraphQL Server with Connected Database

In this architecture, we can find GraphQL Servers with a tightly coupled/integrated database. Upon receiving a query from the client, the GraphQL server will first make sure reading the request payload and then fetching the required data from the database. This is known as resolving the query which in turn results in a response from the GraphQL server. The response is returned to the client and it adheres to the format being specified

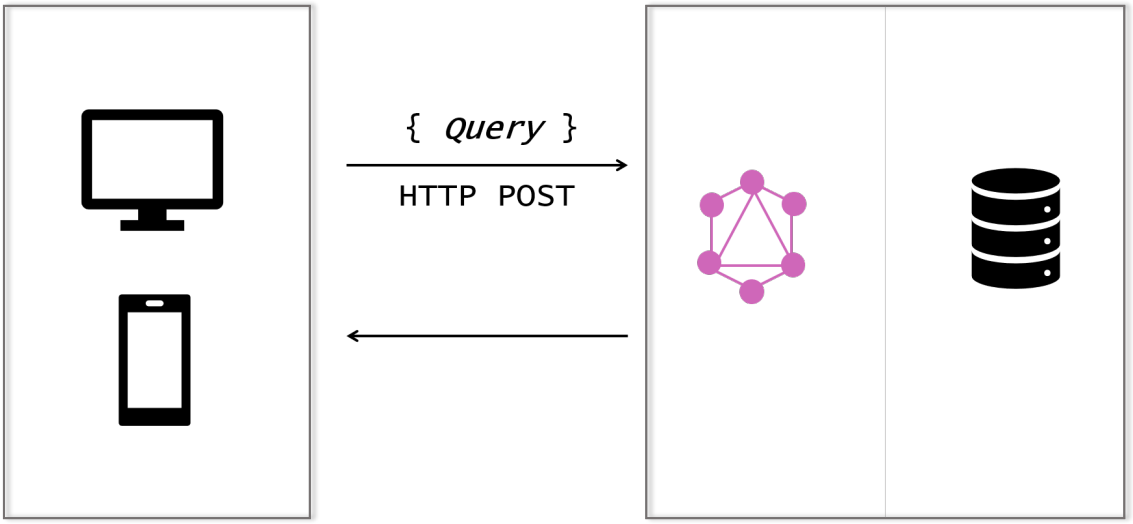

Figure 2.1: GraphQL Server with a Connected DataBase

in the GraphQL specification that is provided by the Schema and the Type system. This kind of architecture can often be used in projects that start from scratch.

In the diagram being shown above, the GraphQL server and the database (SQL or NoSQL) are made to be available on a single node. The Client (desktop or mobile) communicates with the GraphQL server over HTTP. The server processes the request fetches/retrieve data from the DB and, returns it finally to the client.

2.1.3 GraphQL server integrating the existing Systems

This approach would be much helpful for the companies that have legacy infrastructure and possess a collection of different APIs. GraphQL can be employed in this kind of architecture to unify microservices, legacy infrastructure, and even third-party APIs. Observe the below block diagram:

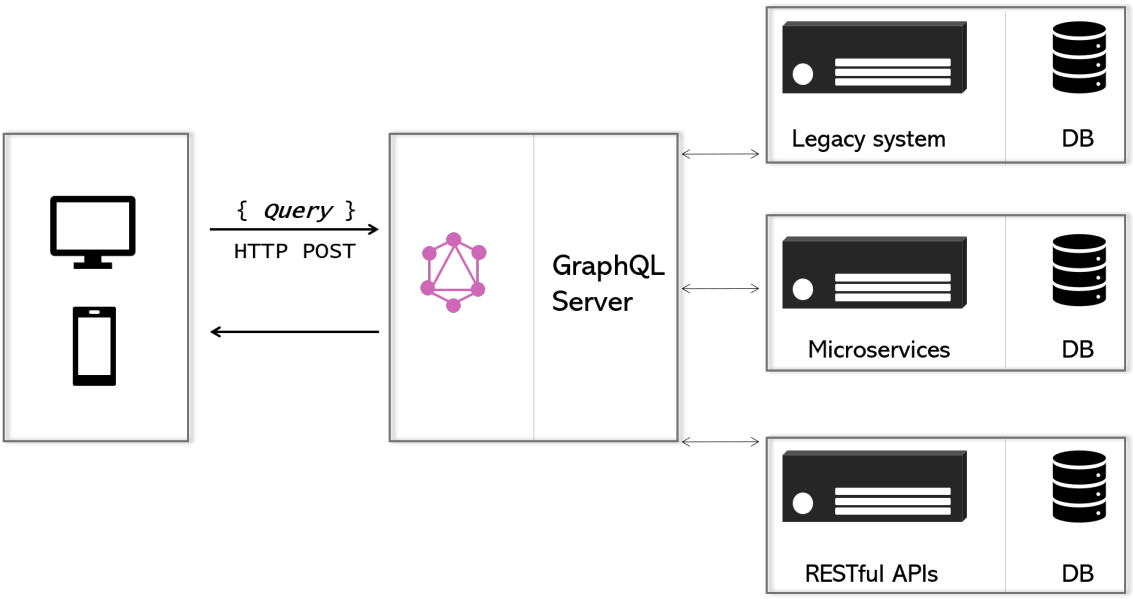

Figure 2.2: GraphQL server integrating the existing Systems

In the given diagram, GraphQL Server acts as an interface between the client and the existing systems. Client applications reach out to the GraphQL server which in turn, resolves the query and gets data from the existing APIs, and renders the response back to the client applications.

2.1.4 Hybrid Approach

Finally, we can put the first two approaches together that is, the dedicated DB approach and the integrated legacy systems approach, to build a GraphQL server. In this architecture, the GraphQL server shall resolve the requests being received. It will then, retrieve data either from the connected database or from the integrated APIs.

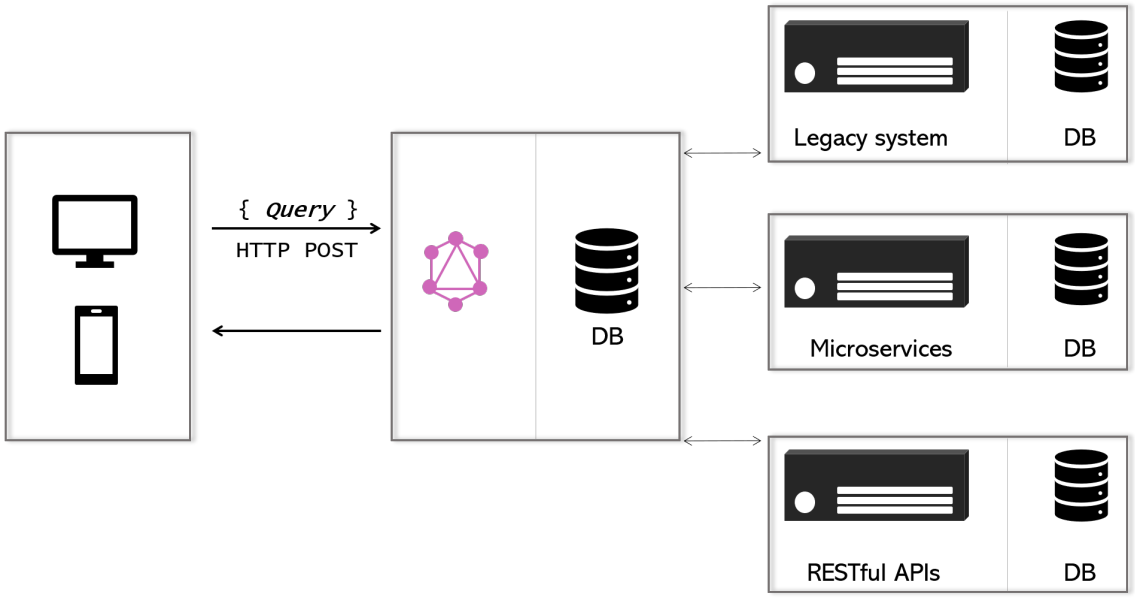

Figure 2.3: Hybrid Approach

2.2 GraphQL Application Components

Just like any other full-stack application, a GraphQL based application will also have the following types of components:

- Server-side components
- Client-side components

2.2.1 GraphQL Application Components - Server Side

Any GraphQL server will contain the following components.

1. **Schema**: GraphQL schema is the heart of any GraphQL server. The schema is a description file that contains the types that are available with the system and the various set of operations that can be performed on those types. In other words, GraphQL helps to expose the various "data resources" that are available to the clients connected to the GraphQL server.

2. **Query**: GraphQL query is nothing but the request being initiated by the GraphQL client to the GraphQL server. The response to the request can be from various data sources like databases or other legacy REST APIs. An important point to be noted here is, GraphQL query is **declarative** (easy for

us to predict the content that will be received by the client on seeing the query itself) by nature and helps to query the GraphQL server in a precise manner.

3. **Resolver**: Resolvers are nothing but the entities which provide instructions that are required to turn a GraphQL query/operation into data. They actually "resolve" the query and return data by defining various resolver functions that correspond to the schema.

These server-side components help the GraphQL server to parse the queries that are coming from the GraphQL client applications and to generate the relevant response. We are aware that GraphQL is a specification and has got multiple implementations. We have a variety of languages supporting the creation of the GraphQL server. In this course, we shall see the Spring Boot way of creating a GraphQL server.

2.2.2 GraphQL Application Components - Client Side

Like server-side components, there are various components on the client-side as well. Take a look at the below client-side components.

1. **GraphQL Playground**: A browser-based interface that helps to edit and testing GraphQL queries/-mutations/subscriptions.
2. **Apollo Client**: Tool/library that helps to build GraphQL client applications and can integrate well with any JavaScript-based front-end.
3. **Postman**: Tool to test REST API endpoints. The recent versions of Postman have support for sending GraphQL queries as well.

We have got an idea of GraphQL's client and server-side components. Before getting deeper, let's have a look at how happens the interaction between a GraphQL client and the GraphQL server (Spring Boot based).

- The web server can be built on Spring Boot.
- A request is made to the Spring Boot GraphQL Server by a GraphQL client. The client application can either be a ReactJS application (the one that is built with the help of Apollo Client library) or GraphQL playground or even GraphiQL browser application.
- The query will get parsed and validated against the schema that is defined in the server.
- If the request passes the validation, the associated resolver methods will be executed.
- The resolver-contained code then will fetch data from a database or an API.
- Finally, the result gets rendered back to the client that initiated the request.

Chapter 3

First GraphQL application using Spring Boot

3.1 Setting up a Spring boot GraphQL Server

Based on the concepts used in an application, other dependencies are also used. Some of them are:

- **spring-boot-starter-jpa** is used to connect our application with data bases in an efficient manner. This dependency internally uses spring-data-jpa dependency.
- **mysql-connector-j** provides a driver connectivity between our Java application and mysql data base. This dependency provides a Type-4 driver connectivity.
- **spring-boot-starter-websocket** establishes a connection between a Client and Server to communicate. This connection requires in Subscription like applications.
- **spring-webflux** publishes the data on User interface. To do this, GraphQL uses Mono, Publisher like classes.

3.2 Writing GraphQL schema

Once done with adding the GraphQL related libraries, we need to create GraphQL schema. Writing GraphQL schemas are quite easy, and they are portable across for promoting reusability. It becomes possible to write schemas with GraphQL's own language which is nothing but Schema Definition Language (SDL). The schema that we write for our application will contain the functionalities and the types that are being exposed by the application. In short, this schema can be considered as a contract between the client and the server that exposes the services. GraphQL Tools library processes the GraphQL Schema files in order to build the correct structure. Once after building, wires special beans to this structure. There is no special configuration needed to register this schema as graphql-spring-boot-starter automatically detects this schema file.

```
1  type Query
2  {
3    message: String
4  }
```

Listing 3.1: Sample SDL

As our project holds a very simple service for exposure, the schema that we have written looks simple too. Otherwise, the schema will consist of the user-defined types that can be exposed. For example, Employee.

```
type Employee
{
  id: ID! #non-nullable
  name: String
}
type Query
{
  employees:[Employee]
}
```

Listing 3.2: Employee SDL schema

Also, the schema can hold type, mutations, and subscriptions. The schema file that we write for a GraphQL application should be saved with graphqls extension (`.graphqls`) and can be available anywhere on the classpath. Let's create a folder named graphql under resources and place this schema file of extension, graphqls.

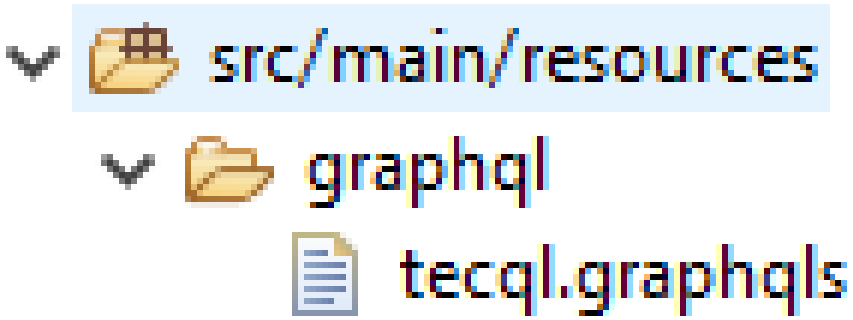

Figure 3.1: Spring boot structure to store graphql queries

3.2.1 Writing Resolvers

We have added the GraphQL related dependencies to our spring boot project and defined a query that can fetch a String literal. Is there anything else to be done? Yes, we need to create a resolver. The need for the resolver is to handle the various fields being mentioned in the type, query or mutation of the schema file (.graphqls file). Mutation is not covered yet and shall be covered later in this course. Resolvers are the ones that can resolve the things that are present in the GraphQL schema. So, the query that we have in the schema should have a resolver associated. We shall create a query resolver by using annotation `SchemaMapping`, which takes two parameters - `typeName` and `field`. Out of these two parameters, `typeName` is mandatory and field is optional. The typeName parameter takes Query/Mutation/Subscription/any user-defined type. Have a look at the resolver that we have created for our GraphQL project.

```
//package and import statements

@Controller
public class QueryResolver {

  @Autowired
  private Services service;

  @SchemaMapping(typeName="Query",field="message")
  public String getMessage() {
    return service.getMessage();
  }

}
```

Listing 3.3: Resolver example

It is always very important to name the methods of the resolver using the field names available in the query. In simple words, there should be a one-to-one correspondence between the fields of the query and the methods of the resolver. The naming convention for the methods of the resolver can be either of the following.

- field
- isField (if the return type is boolean)
- getField

Also, the method should return the type being available for the field in the query. Now, take a look at the signature of the resolver method

```
public String getMessage( )
```

that we have written for the type, query in the GraphQL schema.

```
type Query
{
  message: String
}
```

The resolver has got a method named `getMessage()` for the field message in the Query and the return type of the method is String as the field message is of type String.

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

```
server.port = 8080
spring.graphql.graphiql.enabled = true
```

```
# Comments in GraphQL strings start with the hash (#) symbol.
# The "Query" type is special: it lists all of the available queries that
# clients can execute, along with the return type for each.
type Query {
  message: String
}
```

```
//package and import statements

@Controller
public class QueryResolver {
```

```
5
6       @Autowired
7       private Services service;
8
9       @SchemaMapping(typeName="Query",field="message")
10      public String getMessage() {
11        return service.getMessage();
12      }
13
14    }
```

Create a service called `Services.java` inside the package, `com.infy.service` under `src/main/java`.

```
1   //package and import statements
2
3   @Service
4   public class Services {
5
6     static final String MESSAGE = "Welcome to GraphQL Annotations Practice Session";
7
8     public String getMessage() {
9       return MESSAGE;
10    }
11  }
```

Done with arriving at a Spring Boot GraphQL server with the necessary set of components.

We have finished creating a Spring Boot GraphQL server that can render a string saying Welcome to GraphQL Annotations Practice Session. Let's start the application for execution. Make a right click on the project and click on Run as Spring Boot application/Java Application. If everything goes fine, the project will get started successfully listening to port number, 8080. Now, let's open a browser window and reach out to the URL that can open graphiQL for testing the GraphQL AP

localhost:8080/graphiql