Published in The Startup

Teja Swaroop Mylavarapu Follow

Jan 17, 2021 · 8 min read · Listen

Getting Started in GraphQL With Spring Boot in Java

GraphQL — **GraphQL** is a query language for APIs and a runtime for fulfilling those queries with your existing data.

This is what the standard definition of GraphQL is. I am going to break down the concept of GraphQL with a more hands on approach as described below.

> *In this blog, I am going to talk about:*

1. What exactly is GraphQL?
2. Why do we need GraphQL and what are the advantages of using GraphQL.
3. Setup the Java Spring Boot Micro Service on your local using Spring Initializr.
4. Write a simple GraphQL Spring Boot application in Java.
5. Build a GraphQL request via Postman and query our Application.

1. What exactly is GraphQL?

GraphQL is a query language and a standard for APIs that describes how to ask for data and load data from a server to a client.

It lets the client ask what data they want dynamically, not the usual standard 'REST' response.It is a more expressive standard that lets the Producer properly aggregate data on demand and serve that data on the fly.

The way Cloud Computing has revolutionized the mindset of 'Pay as you use' model, GraphQL lets users request response as they go.

2. Why do we need GraphQL and what are the advantages of using GraphQL.

> *Problem:*

Imagine a scenario, where you have a Mobile and Desktop User Interface. In today's world, you build an endpoint for both of them and retrieve the data from a standard REST endpoint.

For example, let us assume that our API returns back with a JSON comprising of 50 fields.

For Desktops, this is agreeable as the real estate on the desktop is much bigger than Mobile and all the variables and data can be accommodated in a succinct manner.

However, for Mobile, with lack of real estate, we cannot accommodate all the variables and we do not require all the 50 variables returned by our Micro Service.

> *Solution:*

The only solution we have today is to build out another endpoint for Mobile and serve the data. This is an overhead as both the endpoints deal with the similar Repository.

Enter GraphQL, which lets the user request the data as they want in the form of Queries (in Payload). The user can request what they want, GraphQL fetches only that data what the user wants and eliminates the unnecessary data transmission over the network.

This is just one simple problem which I have addressed and there are many more problems whichGraphQL can address.

There are many advantages of using GraphQL. Some of them are:

a. No Over-fetching and Under-fetching of Data — Subscribe for what you want

b. Reduces Network Latency

c. Supports Dynamic API Responses

d. Single Endpoint

e. Reducing the concept of Versioning the APIs

f. No more Strict Handshake between developers — De Coupling

and etc. The list does not stop here.

So, let’s quickly jump into the implementation part of how to implement a GraphQL layer using Spring Boot in Java.

3. Setup the Java Spring Boot Micro Service on your local using Spring Initializr.

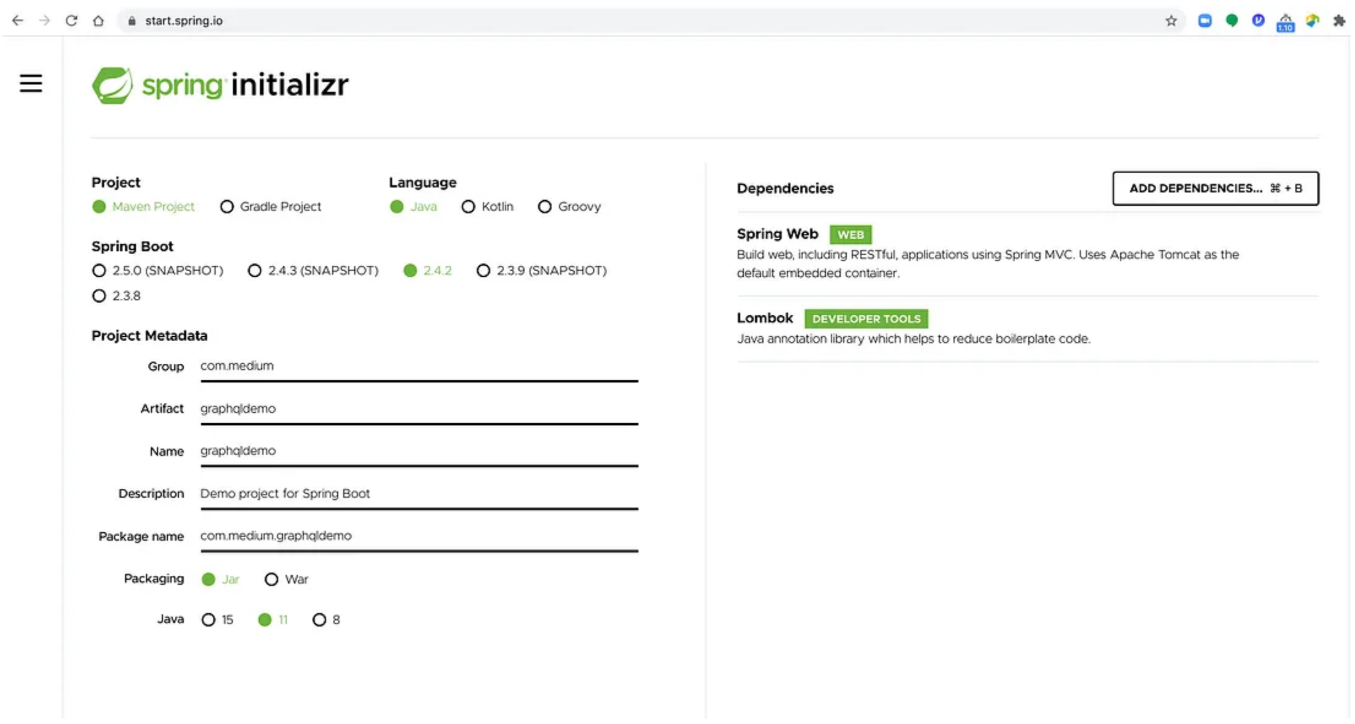
Go to the Spring Initializr URL, where you can create the basic Spring Boot Project located at:

Spring Initializr

Initializr generates spring boot project with just what you need to start quickly!

start.spring.io

Just give the appropriate GroupId and the Artifact Id and add some basic dependencies to kick start your Spring Boot Application.



Project
[✓ Maven Project] Gradle Project

Language
[✓ Java] Kotlin Groovy

Spring Boot
2.5.0 (SNAPSHOT) 2.4.3 (SNAPSHOT) [✓ 2.4.2] 2.3.9 (SNAPSHOT) 2.3.8

Project Metadata

Group	com.medium
Artifact	graphqldemo
Name	graphqldemo
Description	Demo project for Spring Boot
Package name	com.medium.graphqldemo
Packaging	[✓ Jar] War
Java	15 [✓ 11] 8

Dependencies

Spring Web WEB
Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

Lombok DEVELOPER TOOLS
Java annotation library which helps to reduce boilerplate code.

Once you have the Project downloaded, open it in your favorite editor and just modify your application.properties file to run this project on a local port and the context path. I am using IntelliJ for this demo.

> server.servlet.context-path=/graphql-learnings

> server.port = 6060

Your Application now runs on Port 6060. I used port 6060 to keep it unique and avoid port clashes. You are free to use any port you please.

You can further add security to your Application by creating and adding a self signed certificate and add the ‘https’ layer by following my other blog here:

Create a Self Signed Certificate (Keystore) using keytool and host your Spring Boot Application on...

https — Well, we all know that https stands for Hyper Text Transfer Protocol Secure.It is the de-facto standard...

medium.com

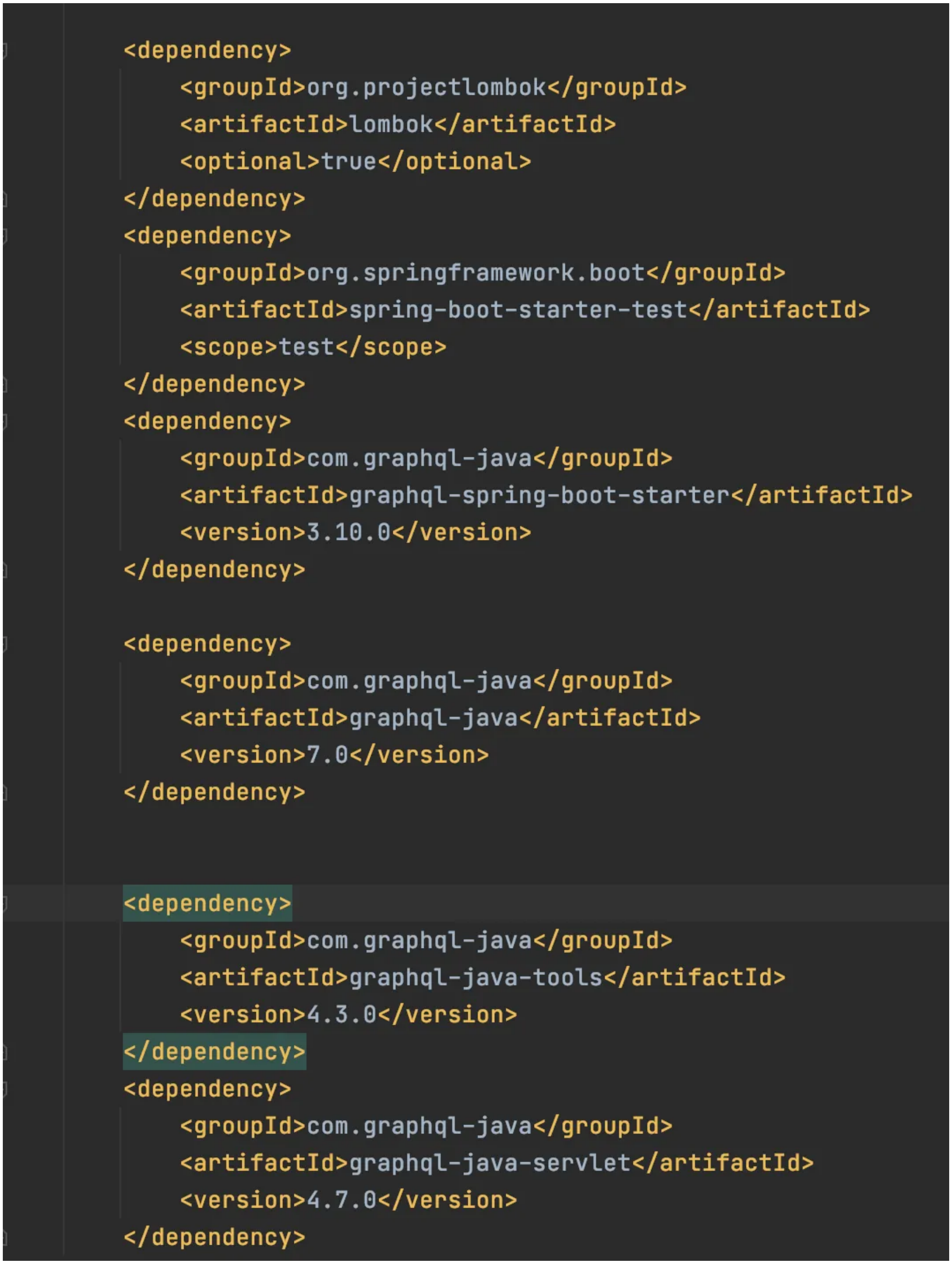
I am not going to to that for this below GraphQL Application.

4. Write a simple GraphQL Spring Boot application in Java.

GraphQL as discussed above comprises of Schemas, queries, mutations and etc. Let’s go ahead and write a schema file with a simple query returning an object.

> *Step i: Add the GraphQL dependencies*

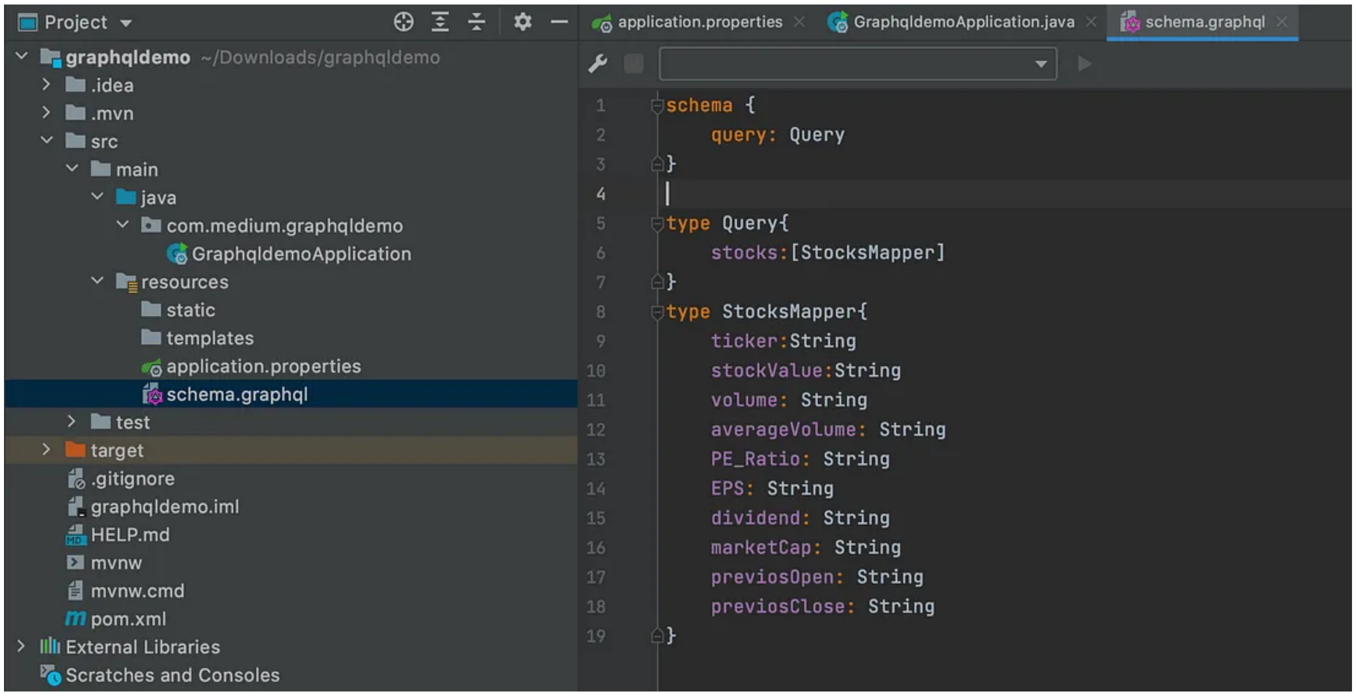
Go to pom.xml file and add the appropriate GraphQL dependencies as show below:



```xml
<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>com.graphql-java</groupId>
    <artifactId>graphql-spring-boot-starter</artifactId>
    <version>3.10.0</version>
</dependency>

<dependency>
    <groupId>com.graphql-java</groupId>
    <artifactId>graphql-java</artifactId>
    <version>7.0</version>
</dependency>


<dependency>
    <groupId>com.graphql-java</groupId>
    <artifactId>graphql-java-tools</artifactId>
    <version>4.3.0</version>
</dependency>
<dependency>
    <groupId>com.graphql-java</groupId>
    <artifactId>graphql-java-servlet</artifactId>
    <version>4.7.0</version>
</dependency>
```

> *Step ii: Create the Schema file:*

Go to src/man/resources and create a schema.graphql file



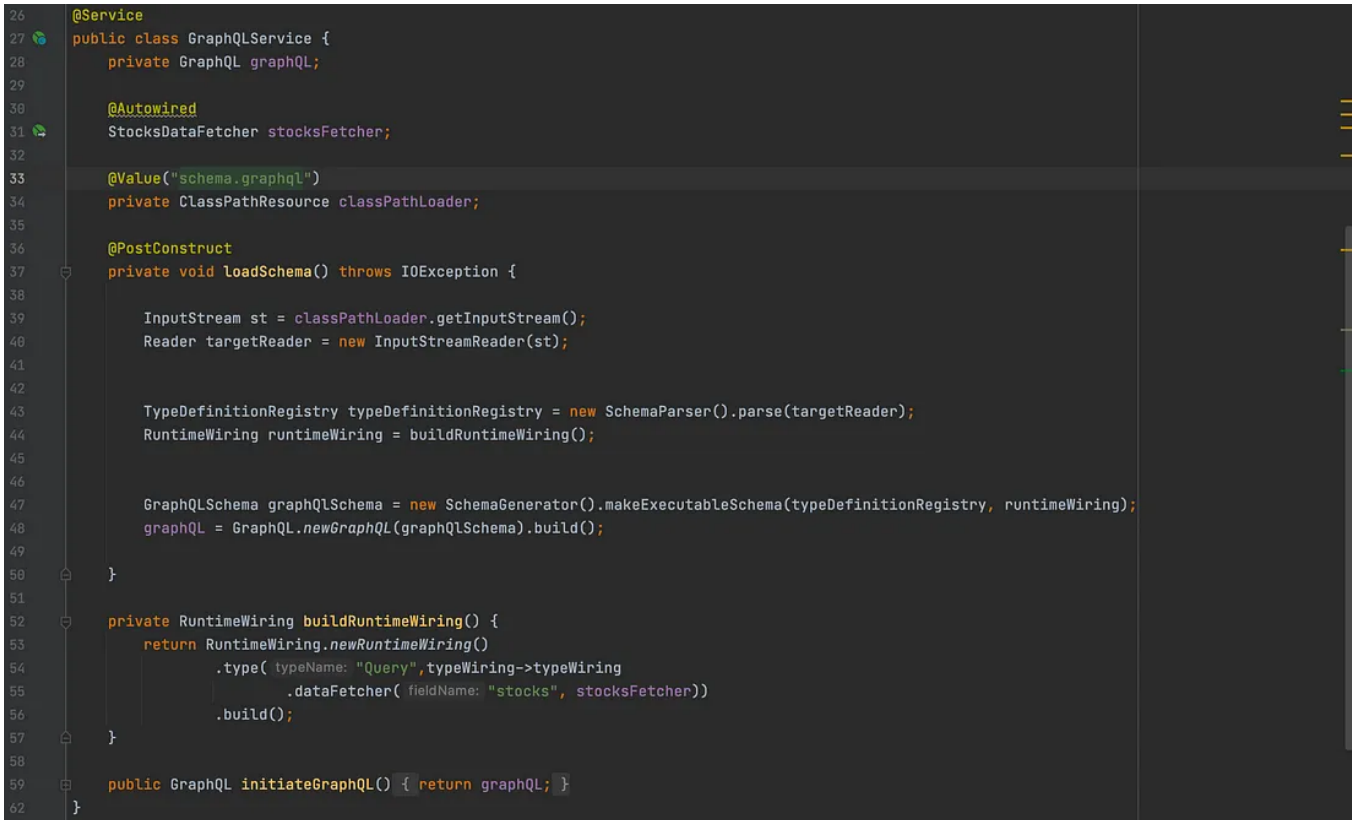
```graphql
schema {
    query: Query
}

type Query{
    stocks:[StocksMapper]
}
type StocksMapper{
    ticker:String
    stockValue:String
    volume: String
    averageVolume: String
    PE_Ratio: String
    EPS: String
    dividend: String
    marketCap: String
    previosOpen: String
    previosClose: String
}
```

We have written a simple query for stocks where we return a Mapper Object with all the above detailed field variables. The type StockMapper is the superset object that our Application returns.

The User, can cherry pick the fields he/she wants and get back the desired fields as requested upon. They get what they want, but not what we return and thus more power to them.

> *Step iii: Add the standard GraphQL Layer*

Now, let's go ahead and introduce the GraphQL Layer into our code by creating a package and add a Service layer as below:



```
26  @Service
27  public class GraphQLService {
28      private GraphQL graphQL;
29
30      @Autowired
31      StocksDataFetcher stocksFetcher;
32
33      @Value("schema.graphql")
34      private ClassPathResource classPathLoader;
35
36      @PostConstruct
37      private void loadSchema() throws IOException {
38
39          InputStream st = classPathLoader.getInputStream();
40          Reader targetReader = new InputStreamReader(st);
41
42
43          TypeDefinitionRegistry typeDefinitionRegistry = new SchemaParser().parse(targetReader);
44          RuntimeWiring runtimeWiring = buildRuntimeWiring();
45
46
47          GraphQLSchema graphQlSchema = new SchemaGenerator().makeExecutableSchema(typeDefinitionRegistry, runtimeWiring);
48          graphQL = GraphQL.newGraphQL(graphQlSchema).build();
49
50      }
51
52      private RuntimeWiring buildRuntimeWiring() {
53          return RuntimeWiring.newRuntimeWiring()
54                  .type( typeName: "Query",typeWiring->typeWiring
55                          .dataFetcher( fieldName: "stocks", stocksFetcher))
56                  .build();
57      }
58
59      public GraphQL initiateGraphQL() { return graphQL; }
62  }
```

Let us talk about the above code line by line for a good understanding as this is wiring is very important.

Line 27: The main GraphQL Object which holds all the data of the schema and the Runtime wiring for our Application. This object is returned on Line 58 to the controller layer where we expose the GraphQL endpoint in the coming sections.

Line 30: The repository layer which we will be building next which fetches the data that we want/ the user wants.

Line 33: This is required to load the schema file present under the classpath inside the resources folder to serve GraphQL the schema we desire.

Line 35: The below method is loaded during runtime as soon as the Application starts to serve the data.

Line 42: Used for compiling a graphQL schema definition file with SchemaParsers.

Line 51: Builds the runtime wiring object for GraphQL. Here is where we map the schema in the schema file to the repository layer. In the schema.graphql file on Line 6, we had stocks as the query object coming in from the user and the user would send

out the desired query. We need to map this query to our Repository layer and serve this data back to the user. Line 54 does exactly this. It maps the incoming request from the client (user query) to our backend Repo layer and does the magic.

> *Step iv: Build the Repository Layer:*

After the wiring of the Service, we need to build a Repo Layer to get the data. This part is going to be very simple as I leave it to the developer to build this layer as per their own use case.

Data can be fetched from many sources not limiting to:

a. Database

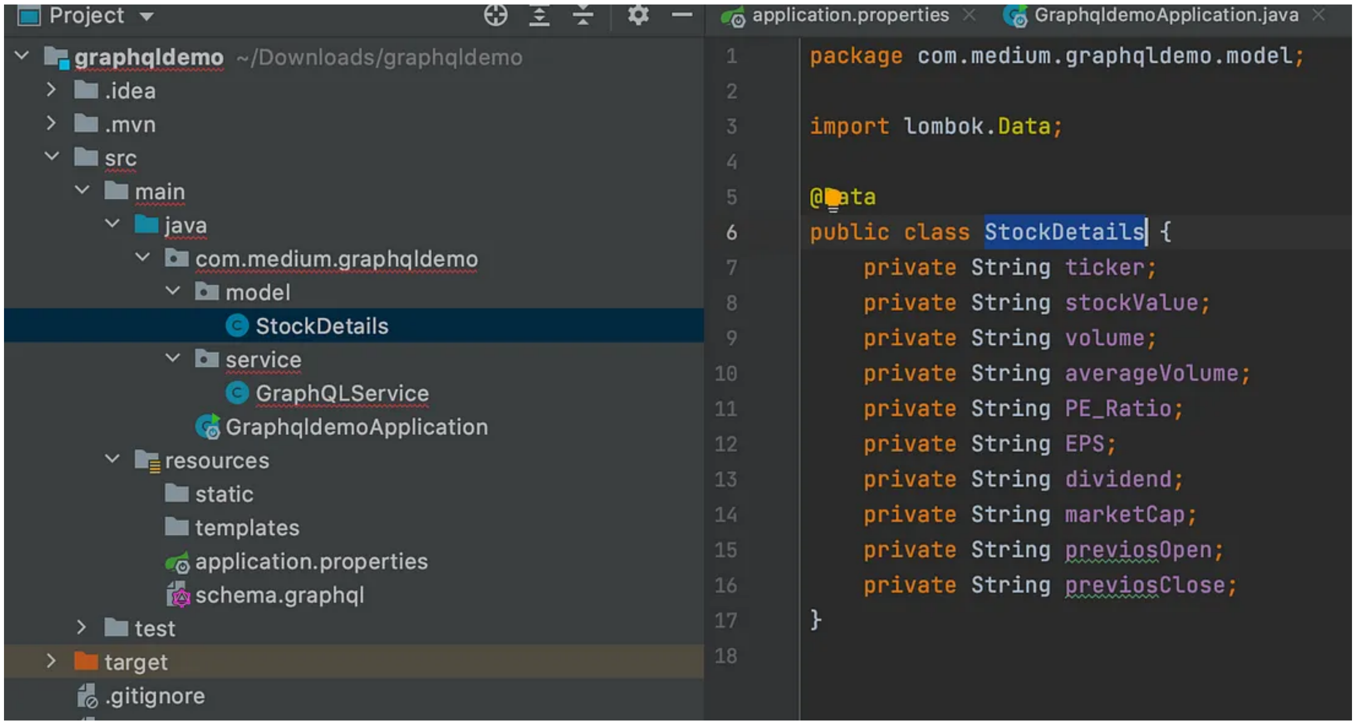
b. Cache Services

c. Another downstream API call

d. Kafka Consumer Services

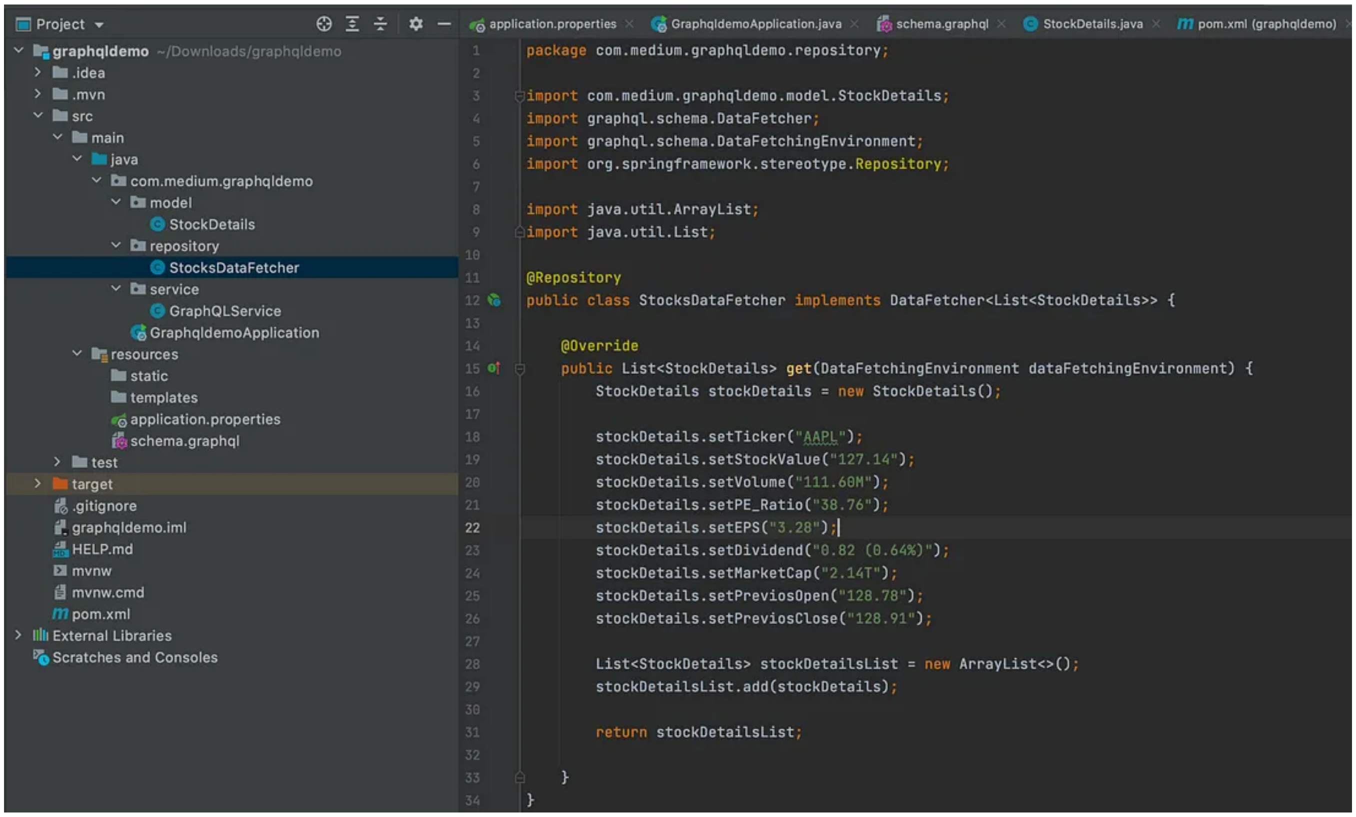
and etc. In this blog, I shall be constructing a simple Object for Demo purposes.

> *Create a model package and create a class called StockDetails.java which has all the required fields.*



```java
package com.medium.graphqldemo.model;

import lombok.Data;

@Data
public class StockDetails {
    private String ticker;
    private String stockValue;
    private String volume;
    private String averageVolume;
    private String PE_Ratio;
    private String EPS;
    private String dividend;
    private String marketCap;
    private String previosOpen;
    private String previosClose;
}
```

> *Create a Repository Layer as below and return the details as an ArrayList*

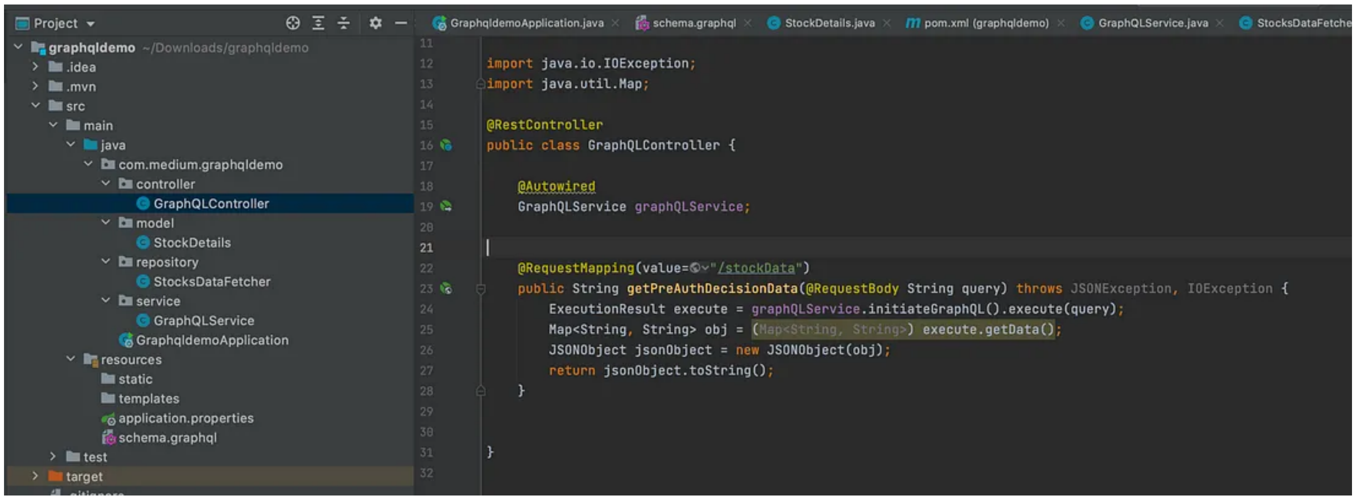


```java
package com.medium.graphqldemo.repository;

import com.medium.graphqldemo.model.StockDetails;
import graphql.schema.DataFetcher;
import graphql.schema.DataFetchingEnvironment;
import org.springframework.stereotype.Repository;

import java.util.ArrayList;
import java.util.List;

@Repository
public class StocksDataFetcher implements DataFetcher<List<StockDetails>> {

    @Override
    public List<StockDetails> get(DataFetchingEnvironment dataFetchingEnvironment) {
        StockDetails stockDetails = new StockDetails();

        stockDetails.setTicker("AAPL");
        stockDetails.setStockValue("127.14");
        stockDetails.setVolume("111.60M");
        stockDetails.setPE_Ratio("38.76");
        stockDetails.setEPS("3.28");
        stockDetails.setDividend("0.82 (0.64%)");
        stockDetails.setMarketCap("2.14T");
        stockDetails.setPreviosOpen("128.78");
        stockDetails.setPreviosClose("128.91");

        List<StockDetails> stockDetailsList = new ArrayList<>();
        stockDetailsList.add(stockDetails);

        return stockDetailsList;

    }
}
```

You can build this Object as desired by getting the data from any other source.

> *Step v: Build the Controller layer and expose a GraphQL endpoint to the user*

Let us build a controller layer and expose a POST endpoint to the end user and enable the feature to let the user query our Application.



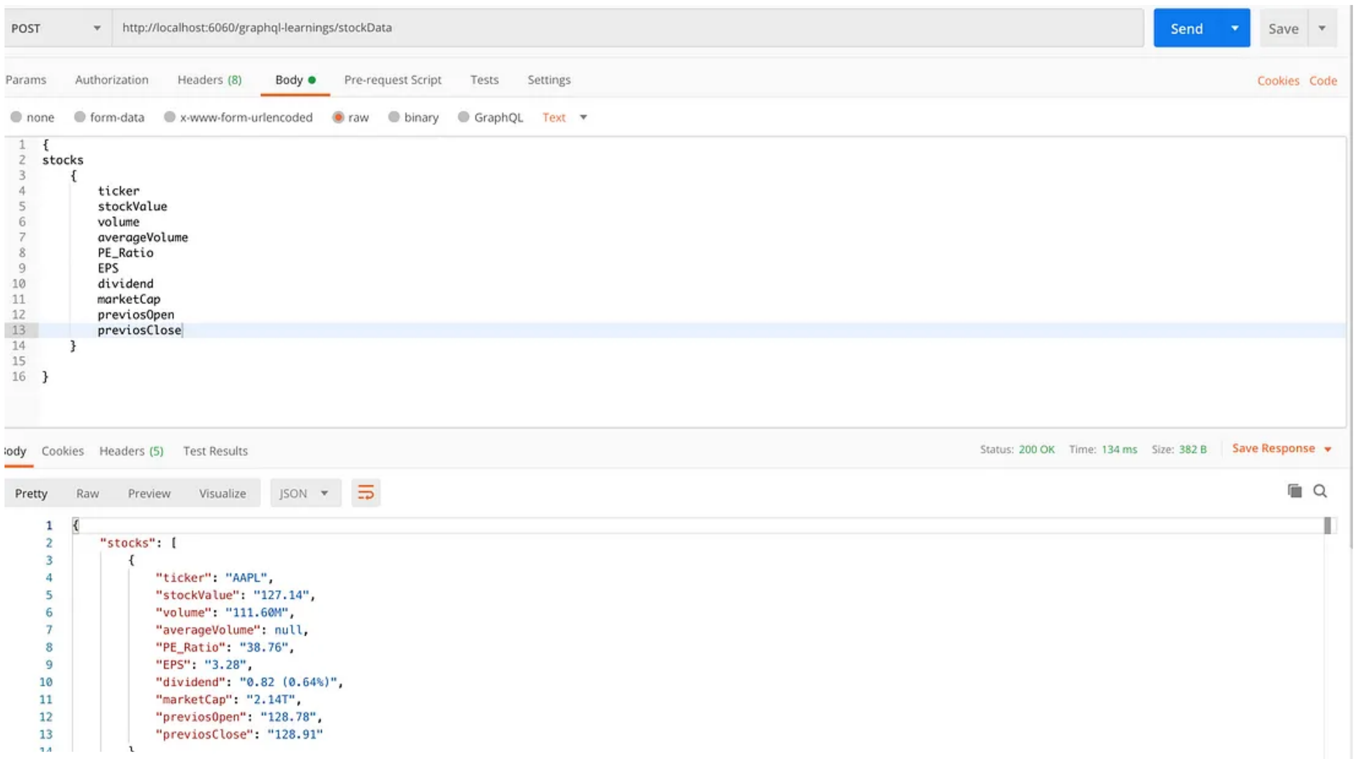
```java

import java.io.IOException;
import java.util.Map;

@RestController
public class GraphQLController {

    @Autowired
    GraphQLService graphQLService;


    @RequestMapping(value="/stockData")
    public String getPreAuthDecisionData(@RequestBody String query) throws JSONException, IOException {
        ExecutionResult execute = graphQLService.initiateGraphQL().execute(query);
        Map<String, String> obj = (Map<String, String>) execute.getData();
        JSONObject jsonObject = new JSONObject(obj);
        return jsonObject.toString();
    }


}

```

The above code pretty self explanatory as we have created an Autowired Object pointing to the Service layer which triggers the graphQL Layer. Next, there is a new POST endpoint which has been created and which accepts String as payload which the user sends as Query. The String is sent out to the GraphQL Service Layer and the data is obtained back and parsed back to the user in the form of String.

With this, we have completed the coding part of our Application and its time to start the Application which should start on port 6060 as mentioned in our previous steps on localhost.

5.Build a GraphQL request via Postman and query our Application.

Build a GraphQL Request via postman or via Curl command on to our application with all the Payload Parameters sent as query as below.



POST http://localhost:6060/graphql-learnings/stockData

```
{
stocks
    {
        ticker
        stockValue
        volume
        averageVolume
        PE_Ratio
        EPS
        dividend
        marketCap
        previosOpen
        previosClose
    }

}
```

Status: 200 OK Time: 134 ms Size: 382 B

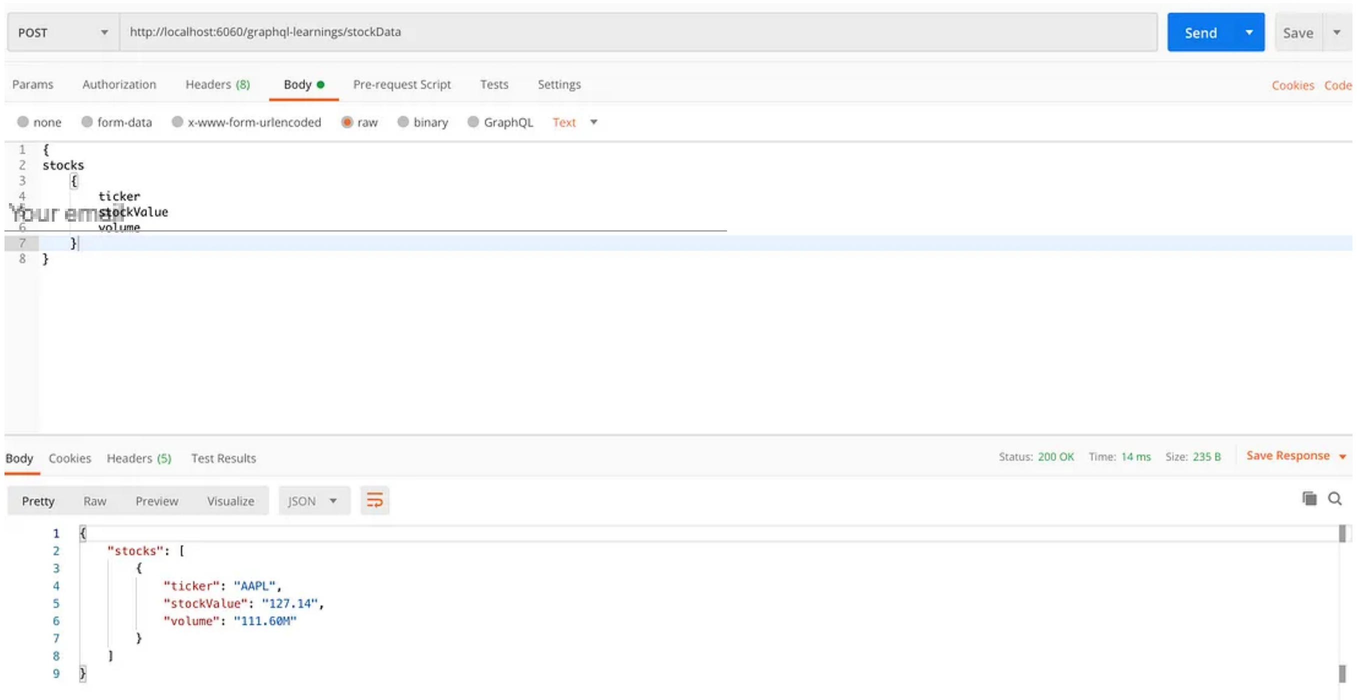
```
{
    "stocks": [
        {
            "ticker": "AAPL",
            "stockValue": "127.14",
            "volume": "111.60M",
            "averageVolume": null,
            "PE_Ratio": "38.76",
            "EPS": "3.28",
            "dividend": "0.82 (0.64%)",
            "marketCap": "2.14T",
            "previosOpen": "128.78",
            "previosClose": "128.91"
        }
```

GraphQL Graphql Schema Graphql Java Graphql Spring Boot Graphql Server

Voila..!!! There you go. Once the user requests what they want in the form of GraphQL query, they get back the data as desired.

Now, let's leverage the power of Gra 151 3 only a small subset of fields which we are interested.

Sign up for Top 5 Stories



POST http://localhost:6060/graphql-learnings/stockData

```
{
stocks
    {
        ticker
        stockValue
        volume
    }
}
```

Your email

Status: 200 OK Time: 14 ms Size: 235 B

```
{
    "stocks": [
        {
            "ticker": "AAPL",
            "stockValue": "127.14",
            "volume": "111.60M"
        }
    ]
}
```

About Help Terms Privacy

If we can observe the Request and Response in the above diagram, we can see that we have requested only 3 fields from our GraphQL Application and we got back only those 3 fields which we requested. Even though our Applications can respond back with a whole lot of otl

er l get back what WE want not what the Application can

This, in my opinion is a game changer. As we clearly observe on the 'Time' and the 'Size' fiel on Postman, we can clearly see the difference in those fields taken for heavy load vs the normal load which we requested.

Depending on the use case, users can query the data how much ever they want and reduce t latency and also the Network bandwidth over the network as less data will be traversed.

I hope this article of mine has shed some light on how to connect to create a simple GraphQ Application on Spring Boot using Java. Happy Coding…!!

If you have liked what you read, Please Clap hands below as this encourages me to write more.