# GraphQL with Spring Boot for Resource Aggregation

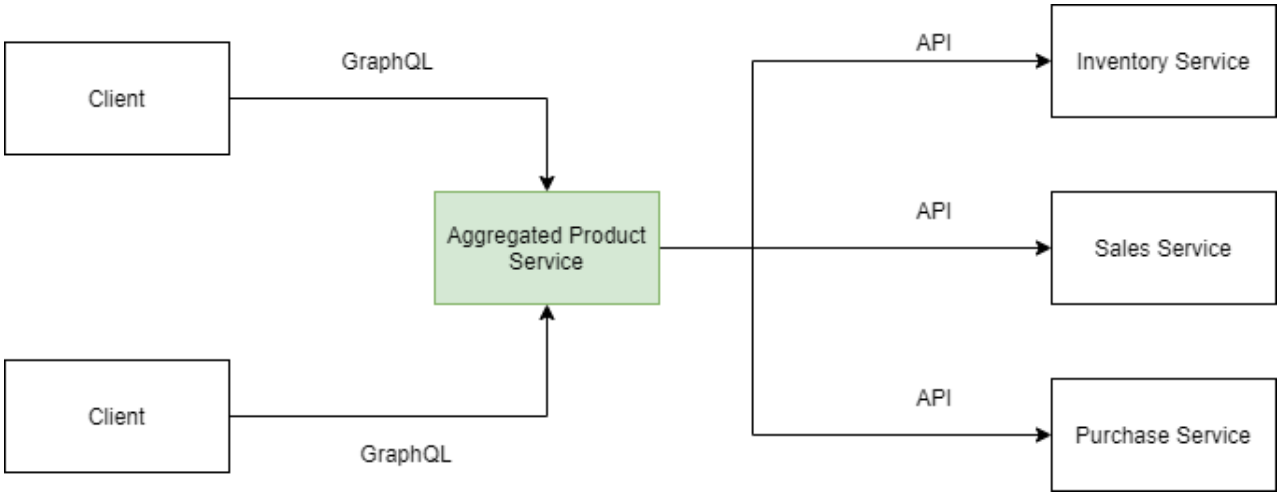
July 1, 2020July 1, 2020 Raymond Lee

GraphQL (https://graphql.org/) is a API query language which allows clients to define exactly what they want in the response by invoking an API. On the server side, a runtime engine is used to execute queries to return the data requested by the clients.

This article will demonstrate how GraphQL can be used to implement a composite service API that aggregates data from multiple services. Composite service is a common pattern in service based architectures such as Microservices and Service Oriented Architecture (SOA). For example, a Ecommerce company may want to aggregate its various information about their products such as pricing, stock levels, sales and purchase order data into a single API for consumption by its own internal and 3rd party systems.

## What we are building

In this blog, we are going to implement a product API that aggregates various product related information from individual systems and return the relevant data to the client based on the GraphQL query in the request. The diagram below shows the conceptual system architecture




Client applications request via GraphQL product information using the Aggregated Product Service. Based on the GraphQL query, the Aggregated Product Service gathers the required information from the

relevant service(s) and returns the result to the client.

# Project Setup

To implement the Aggregated Product Service, create a Spring Boot app with the following dependencies. GraphQL support is enabled via Spring Boot starters provided here (https://github.com/graphql-java-kickstart/graphql-spring-boot).

```
<properties>
     <java.version>1.8</java.version>
     <graphql-spring-boot.version>5.11.1</graphql-spring-boot.version>
     <graphql-java.version>5.7.1</graphql-java.version>
</properties>

<dependencies>
     <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
     </dependency>
     <dependency>
        <groupId>com.graphql-java-kickstart</groupId>
        <artifactId>graphql-spring-boot-starter</artifactId>
        <version>${graphql-spring-boot.version}</version>
     </dependency>
     <dependency>
        <groupId>com.graphql-java-kickstart</groupId>
        <artifactId>graphiql-spring-boot-starter</artifactId>
        <version>${graphql-spring-boot.version}</version>
     </dependency>
     <dependency>
        <groupId>com.graphql-java-kickstart</groupId>
        <artifactId>graphql-java-tools</artifactId>
        <version>${graphql-java.version}</version>
     </dependency>
</dependencies>
```

The graphql-java-tool library allows use of the GraphQL schema language to build the graphql-java schema. It parses the given GraphQL schema and allows you to BYOO (bring your own object) to fill in the implementations.

# GraphQL Schema

GraphQL schema is defined using the Schema Definition Language (SDL) which specifies all the fields and operations exposed by the APIs. It is based around a type system (https://graphql.org/learn/schema/).

Include the following scheme as file product.graphqls under src/main/resources/graphql folder. By default, the GraphQL starter scans GraphQL schema files from classpath folders **/*.graphql

```
# File src/main/resources/graphql/product.graphqls
type Product {
   sku: String
   stock: [Inventory!]!
   purchaseOrders: [PurchaseOrder!]!
   salesOrders: [SalesOrder!]!
}

type Inventory {
   location: String
   qtyAvailable: Int
}

type PurchaseOrder {
   orderDate: String
   shipDate: String
   qty: Int
}

type SalesOrder {
   item: Product!
   qty: Int
   unitPrice: Float
   totalAmount: Float
}

type Query {
   findProduct(sku:String): Product
}
```

GraphQL by default supports the scalar types: **Int**, **Float**, **String**, **Boolean** and **ID**.

**[]** is used to indicate an array and **!** marks the attribute as Non-Null. For example

```
Product {
  ...
  stock: [Inventory!]!
```

means the stock attribute should be a Non-Null list of Inventory and each Inventory is also Non-Null.

The type **Query** is special in that it defines the entry point of every GraphQL queries. In our case, there is only 1 query *findProduct*

# Data Classes

The types defined in the schema can be implemented as POJOs. For example

```
// Product.java
public class Product {
     private String sku;
     private List<Inventory> stock;
     private List<SalesOrder> salesOrders;
     private List<PurchaseOrder> purchaseOrders;
     ...
```

# Resolvers

The GraphQL Java Tools provides the GraphQLQueryResolver interface for implementing the queries.

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;
import com.coxautodev.graphql.tools.GraphQLQueryResolver;
import au.com.arbco.graphqlspike.model.Product;
import au.com.arbco.graphqlspike.service.ProductService;

@Component
public class QueryResolver implements GraphQLQueryResolver {

     @Autowired
     private ProductService productService;

     public Product findProduct(String sku) {
         return productService.getProduct(sku);
     }
}
```

Since we define the query findProduct in the schema, we implement the method with the same name here. If we implements the aggregated API as REST, it would fetch the various data from the corresponding services (inventory, sales, purchase), aggregate them into the Product and return it in the response. With GraphQL, fetching complex (non scalar) attributes is *lazy* and is only invoked when needed, i.e. the incoming query requests the attribute to be returned.

To resolve complex attributes of a type, the Java Tools provides the GraphQLResolver interface. The codes below demonstrate how to resolve the inventory, sales and purchase order attributes of the product type

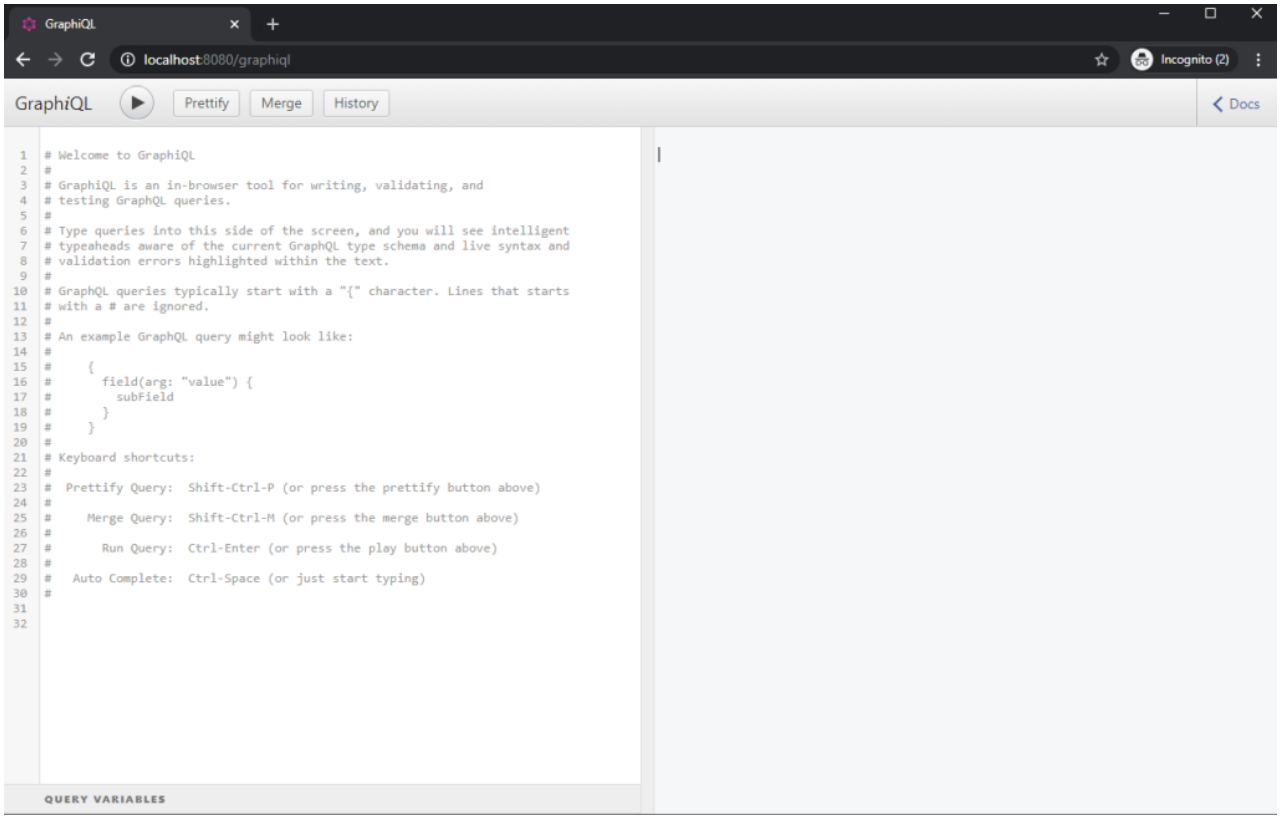
```
@Component
public class ProductResolver implements GraphQLResolver<Product> {
     private Logger logger = LoggerFactory.getLogger(getClass());
     @Autowired
     private ProductService productService;

     public Inventory[] stock(Product product) {
          logger.info("Get inventory data for product: {}",
product.getSku());
          return productService.getInventories(product);
     }

     public PurchaseOrder[] purchaseOrders(Product product) {
          logger.info("Get POs for product: {}", product.getSku());
          return productService.getPurchaseOrders(product);
     }

     public SalesOrder[] salesOrders(Product product) {
          logger.info("Get SOs for product: {}", product.getSku());
          return productService.getSalesOrders(product);
     }
}
```

In our scenario, the data is retrieved via REST APIs from the individual services. Below is a mock up implementation of the class ProductService

```
@Service
public class MockProductService implements ProductService {
     @Autowired
     private RestTemplate restTemplate;

     // ... URLs to services

     @Override
     public Inventory[] getInventories(Product product) {
          return restTemplate.getForObject(inventoryUrl,
Inventory[].class, product.getSku());
     }

     @Override
     public PurchaseOrder[] getPurchaseOrders(Product product) {
          return restTemplate.getForObject(poUrl,
PurchaseOrder[].class, product.getSku());
     }

     @Override
     public SalesOrder[] getSalesOrders(Product product) {
          return restTemplate.getForObject(soUrl, SalesOrder[].class,
product.getSku());
     }

     @Override
     public Product getProduct(String sku) {
          Product product = new Product();
          product.setSku(sku);
          return product;
     }
}
```

# Testing

The GraphiQL UI tool (included in the Maven dependencies) can be used for testing the server. Start up Spring Boot server as usual and go to link /graphiql (e.g. http://localhost:8080/graphiql (http://localhost:8080/graphiql)) in the browser.

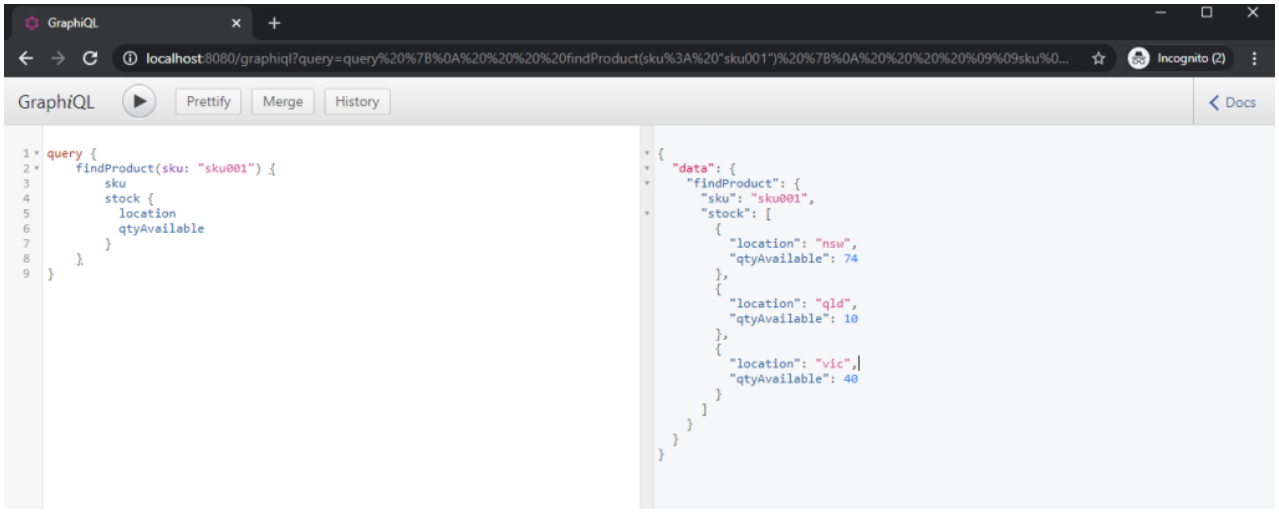



## To get product inventory only

Type the following query in the left panel:

```
query {
     findProduct(sku: "sku001") {
         sku
         stock {
            location
            qtyAvailable
         }
     }
}
```

will return the following results in GraphiQL




Note only the inventory API (method *stock* in the ProductResolver) is invoked. Below is the log message from the server

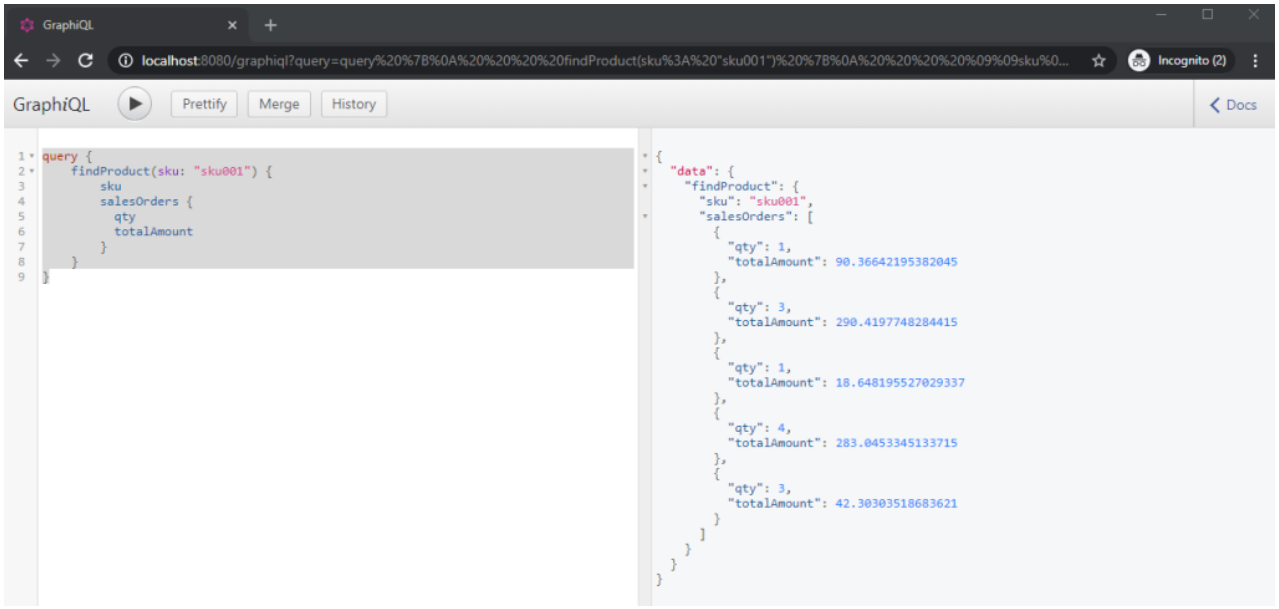
```
2020-07-01 14:04:15.506 INFO 13744 --- [io-8080-exec-10]
a.c.a.g.resolver.ProductResolver : Get inventory data for product:
sku001
```

## To get sales orders only

Type the following query in the left panel:

```
query {
     findProduct(sku: "sku001") {
          sku
          salesOrders {
              qty
              totalAmount
          }
     }
}
```

will return the following results in GraphiQL



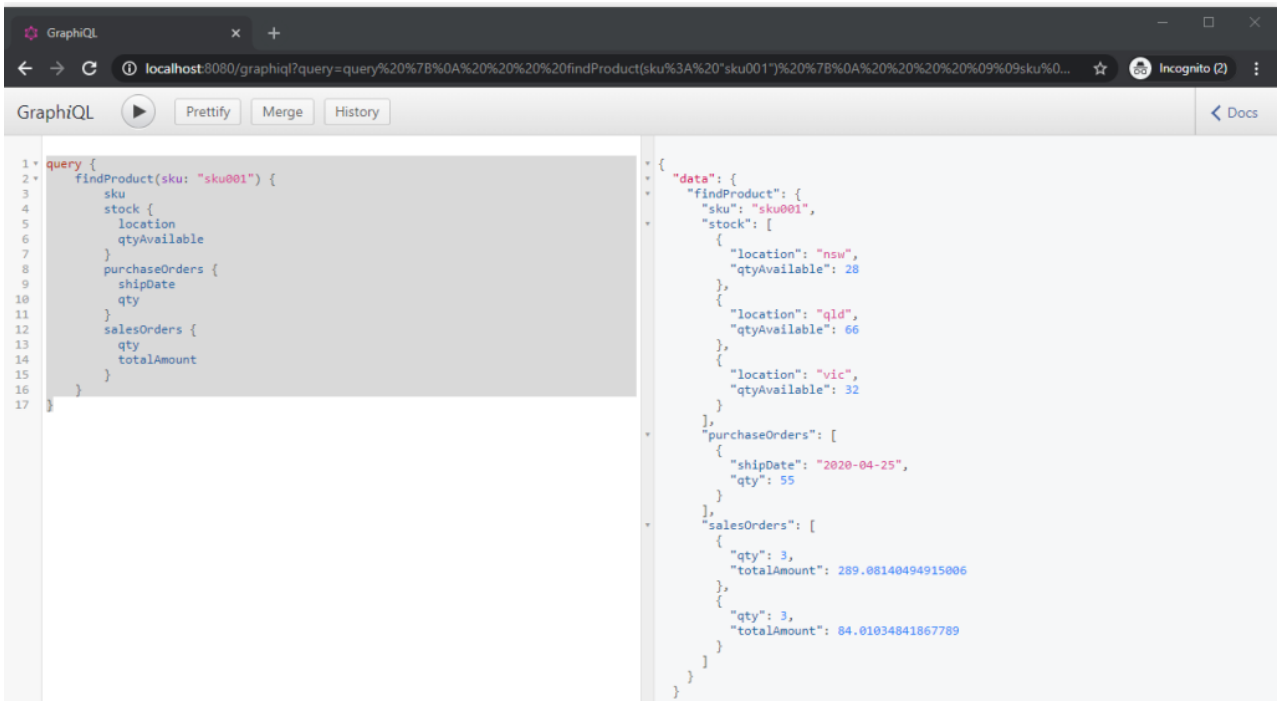

Note the query only requests some of the sales order attributes (qty and totalAmount).

## To get all product attributes

Type the following query in the left panel:

```
query {
     findProduct(sku: "sku001") {
          sku
          stock {
               location
               qtyAvailable
          }
          purchaseOrders {
               shipDate
               qty
          }
          salesOrders {
               qty
               totalAmount
          }
     }
}
```

will return the following results in GraphiQL




# Conclusions

GraphQL seems to be a good choice to implement resource aggregation to expose a higher level API using resources from individual APIs. If design properly, it could provide some performance improvement. Another advantage compared with REST is versioning. For example, adding new attributes to an API would not affect any existing clients as the responses to them would not be changed (as it is specified in the query). However, GraphQL also has some limitations (https://dzone.com/articles/graphql-with-springboot) so careful considerations should be made when applying it in different context.

Architecture & Design, Java, Spring  api, graphql, microservices, spring boot