# Proof of Concept (POC)

## Federation Support through Java

*Version 1.0.0*
16.03.2024

---

### Objectives

Determine whether it's technically feasible to implement GraphQL federation using Java technologies by Identifying the suitable tools and libraries in the Java ecosystem that support GraphQL federation and evaluate their capabilities for subgraph creation and federation

### Summary

Develop the necessary components and services using Java to create a federated GraphQL system. Here the subgraph creation using Java and federation - combining the independent services through java was explored. Further the querying capabilities were analyzed for a simple query , and complex query (including nested queries and multiple queries in the same level). The ability of the federated service to generate a query plan and resolve the independent services one by one to respond accurately was also explored.

---

### Scenarios

| | Scenario | Requirement |
|---|---|---|
| 1 | Creation of federated graph by federating Subgraph endpoints | When Subgraph endpoint URL and schema is defined able to create federated graph with unified schema |
| 2 | Resolving a Simple Query | When simple query is executed respond with accurately by identifying relevant subgraph service |
| 3 | Resolving a Complex Query - Nested | When nested query is executed resolve which subgraphs to call , order of subgraphs to call and the sub queries to be called |
| 4 | Resolving a Complex Query - multiple queries in the same level | When multiple queries in same level is executed resolve which subgraphs to call , order of subgraphs to call and the sub queries to be called |

---

## Table of Content

# 1. Creation of federated graph by federating Subgraph endpoints

## 1.1. Requirement

- When the list of Subgraph endpoint URLs and schemas are provided the ability to create a federated graph with a unified schema.
- **Input** : Subgraph endpoints and schema with unique namespaces for each individual service
- **Output** : Federated schema unifying the schemas of individual schemas

## 1.2. Implementation

**Library Used**

- Since Apollo Only provides node.js support for supergraph creation and does not support supergraph creation using Java (more here), another java library was found to support the federation.
- This library is graphql-orchestrator-java library which aggregates and combines the schemas from multiple data providers and orchestrates the incoming graphql queries to the appropriate services.
    - Github : https://github.com/graph-quilt/graphql-orchestrator-java
    - Documentation : https://graph-quilt.github.io/graphql-orchestrator-java/

**Steps**

### 1.1. Subgraph Implementation

- Multiple independent graphql services (subgraphs) were created and run locally
- These services had schemas with federation directives and necessary resolvers
- The subgraphs were created using the open-source GraphQL server libraries which were recommended in the Apollo federation documentation as Federation-compatible subgraph implementations
- The subgraphs used for the POC were created using DGS framework-  GraphQL for Java with Spring Boot made easy.
    - Github : https://github.com/netflix/dgs-framework/
    - Documentation : https://netflix.github.io/dgs/federation/

- The subgraphs that were implemented for the POC are the following
    - Github : https://github.com/shamin2021/netflix-dgs-federation-examples

1. **Subgraph Review :**

**Schema** :

```
type Review @key(fields: "id") {
    id: ID!
    body: String
    author: User @provides(fields: "username")
    product: Product
}

type User @key(fields: "id") @extends {
    id: ID! @external
    username: String @external
    reviews: [Review]
}

type Product @key(fields: "upc") @extends {
    upc: String! @external
    reviews: [Review]
}
```

Note : Federation Directives have been used (more)

**Resolvers** :

```
@DgsComponent
public class Query {
    private final ReviewRepository reviewRepository;

    public Query(ReviewRepository reviewRepository) {
        this.reviewRepository = reviewRepository;
    }

    @DgsEntityFetcher(name = "Product")
    public Product product(Map<String, Object> values) {
        return new Product((String) values.get("upc"));
    }

    @DgsEntityFetcher(name = "User")
    public User author(Map<String, Object> values) {
        return new User((String) values.get("id"), null);
    }

    @DgsData(parentType = "Product", field = "reviews")
    public List<Review> getReviewsForProduct(DgsDataFetchingEnvironment dataFetchingEnvironment) {
        Product product = dataFetchingEnvironment.getSource();
        return reviewRepository.findByProductUpc(product.getUpc());
    }

    @DgsData(parentType = "Review", field = "product")
    public Product getReviewProduct(DgsDataFetchingEnvironment dataFetchingEnvironment) {
        Review review = dataFetchingEnvironment.getSource();
        return new Product(review.getProductUpc());
    }

    @DgsData(parentType = "Review", field = "author")
    public User getReviewAuthor(DgsDataFetchingEnvironment dataFetchingEnvironment) {
        Review review = dataFetchingEnvironment.getSource();
        return new User(review.getAuthorId(), review.getAuthorUsername());
    }
}
```

Note : DGS framework has been used to create resolvers

- Entity Fetcher has been implemented (here) : responsible for creating an instance of a object based on the representation in the _entities query
- DgsData : hydrate data for a field (here ) (here)

Medium Article for more information
https://medium.com/volvo-car-mobility-tech/working-with-entities-in-graphql-federation-a88e9f9865b2

2. **Subgraph Inventory**

Schema :

```
type Product @key(fields: "upc") @extends {
    upc: String! @external
    weight: Int @external
    price: Int @external
    inStock: Boolean
    shippingEstimate: Int @requires(fields: "price weight")
}
```

Resolvers :

```
@DgsComponent
public class Query {
    private final InventoryRepository inventoryRepository;

    public Query(InventoryRepository inventoryRepository) {
        this.inventoryRepository = inventoryRepository;
    }

    @DgsEntityFetcher(name = "Product")
    public Product product(Map<String, Object> values) {
        return new Product(
                (String) values.get("upc"),
                (int) values.getOrDefault("price", 0),
                (int) values.getOrDefault("weight", 0));
    }

    @DgsData(parentType = "Product", field = "inStock")
    public Boolean checkProductInStock(DgsDataFetchingEnvironment dataFetchingEnvironment) {
        Product product = dataFetchingEnvironment.getSource();
        ProductInventory inventory = inventoryRepository.findByUpc(product.getUpc());
        return inventory != null && inventory.isInStock();
    }

    @DgsData(parentType = "Product", field = "shippingEstimate")
    public Integer estimateShipment(DgsDataFetchingEnvironment dataFetchingEnvironment) {
        Product product = dataFetchingEnvironment.getSource();
        if (product.getPrice() > 100) return 0;
        return product.getWeight() * 2;
    }
}
```

3. **Subgraph Accounts :**

Schema :

```
type Query {
    me: User
}

type User @key(fields: "id") {
    id: ID!
    name: String
    username: String
}
```

Resolvers :

```
@DgsComponent
public class Query {
    private final UserRepository userRepository;

    public Query(UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    @DgsEntityFetcher(name = "User")
    public User resolveReference(Map<String, Object> values) {
        return userRepository.findById((String) values.get("id"));
    }

    @DgsQuery(field = "me")
    public User getCurrentUser() {
        return userRepository.findByUsername("john");
    }
}
```

4. **Subgraph Product :**

Schema :

```
type Query {
    topProducts(first: Int = 5): [Product]
}

type Product @key(fields: "upc") {
    upc: String!
    name: String
    price: Int
    weight: Int
}
```

Resolvers :

```
@DgsComponent
public class Query {
    private final ProductRepository productRepository;

    public Query(ProductRepository productRepository) {
        this.productRepository = productRepository;
    }

    @DgsEntityFetcher(name = "Product")
    public Product resolveReference(Map<String, Object> values) {
        return productRepository.findByUpc((String) values.get("upc"));
    }

    @DgsQuery
    public List<Product> topProducts(Integer first) {
        return productRepository.findAll()
                .stream()
                .limit(first)
                .collect(Collectors.toList());
    }
}
```

### 1.2. Federation of Subgraphs

This library graphql-orchestrator-java was used for the federation

1. Before Federating the services each service was required to implement a **service provider interface** of the library

```
    private String endpoint;
    private QueryExecutor queryExecutor;
    private String schema;
    private String nameSpace;

    public GenericProvider(String endpoint, WebClient webClient, String schemaFilePath, String nameSpace) {
        this.endpoint = endpoint;
        this.nameSpace = nameSpace;
        this.queryExecutor = (QueryExecutor) new WebClientQueryExecutor(webClient, endpoint);

        try {
            this.schema = readFileAsString(schemaFilePath);
        } catch (Exception e) {
            throw new RuntimeException("Error reading schema file", e);
        }
    }

    @Override
    public String getNameSpace() {
        // Return namespace based on the endpoint or any other logic you have
        return this.nameSpace;
    }

    @Override
    public ServiceType getSeviceType() { return ServiceType.FEDERATION_SUBGRAPH; }

    @Override
    public Map<String, String> sdlFiles() { return ImmutableMap.of( k1: "schema.graphqls", schema); }

    @Override
    public CompletableFuture<Map<String, Object>> query(final ExecutionInput executionInput,
                                                        final GraphQLContext context) {
        return queryExecutor.query(executionInput, context);
    }

    public static String readFileAsString(String filePath) throws Exception {
        Path path = Paths.get(filePath);
        byte[] bytes = Files.readAllBytes(path);
        return new String(bytes, StandardCharsets.UTF_8);
    }
}
```

- Here it was necessary for each service to have a unique namespace
- The subgraph services were to return FEDERATION_SUBGRAPH for getServiceType() function, This ensured that federation directives were recognized in the SDLs (here)
- Each service returned its SDLfiles in the sdlFiles
- Function query returned a completable future map that executed the query and returned the response

```
GenericProvider inventoryService = new GenericProvider( endpoint: "http://localhost:8084/graphql", webClient,
        schemaFilePath: "src/main/resources/Inventory.graphqls",  nameSpace: "inevntory");
GenericProvider productService = new GenericProvider( endpoint: "http://localhost:8081/graphql", webClient,
        schemaFilePath: "src/main/resources/Product.graphqls", nameSpace: "product");
GenericProvider reviewService = new GenericProvider( endpoint: "http://localhost:8083/graphql", webClient,
        schemaFilePath: "src/main/resources/ReviewN.graphqls", nameSpace: "review");
GenericProvider accountService  = new GenericProvider( endpoint: "http://localhost:8082/graphql", webClient,
        schemaFilePath: "src/main/resources/User.graphqls", nameSpace: "user");
```

Each service with input **(URL endpoint of service, webclient , schema path, namespace)** were used to create service providers

```
40      // create a runtimeGraph by stitching service providers
41      RuntimeGraph runtimeGraph = SchemaStitcher.newBuilder() Builder
42              .service(accountService)
43              .service(productService)
44              .service(inventoryService)
45              .service(reviewService)
46              .build() SchemaStitcher
47              .stitchGraph();
48
49      GraphQLOrchestrator.Builder builder = GraphQLOrchestrator.newOrchestrator();
50      builder.runtimeGraph(runtimeGraph);
51      builder.queryExecutionStrategy(queryExecutionStrategy);
52      GraphQLOrchestrator graphQLOrchestrator = builder.build();
53
54      String printSchema = new SchemaPrinter().print(runtimeGraph.getExecutableSchema());
55      System.out.println(printSchema);
56
```

Then the services were used to stitch the runtime graph
Stitching Services:

- **SchemaStitcher.newBuilder() i**nitializes a builder for stitching together GraphQL schemas.
- .service(accountService), .service(productService), etc., adds individual GraphQL services to be stitched together. Each service represents a separate GraphQL schema.
- .build().stitchGraph() builds the runtime graph by stitching the provided services together.

Building Orchestrator:

- **GraphQLOrchestrator.**Builder builder = GraphQLOrchestrator.newOrchestrator(); initializes a builder for creating a GraphQL orchestrator.
- builder.runtimeGraph(runtimeGraph); sets the runtime graph created by stitching services.
- builder.**queryExecutionStrategy**(queryExecutionStrategy); sets the query execution strategy for the orchestrator. This strategy determines how queries are executed across the federated graph.
- GraphQLOrchestrator graphQLOrchestrator = builder.build(); builds the GraphQL orchestrator using the configured settings.

Important Java classes in the library

- Stitcher : https://github.com/graph-quilt/graphql-orchestrator-java/blob/master/src/main/java/com/intuit/graphql/orchestrator/stitching/XtextStitcher.java
- https://github.com/graph-quilt/graphql-orchestrator-java/blob/master/src/main/java/com/intuit/graphql/orchestrator/stitching/SchemaStitcher.java

**Output**

```
"Directs the executor to include this field or fragment only when the `if` argument is true"
directive @include(
    "Included when true."
    if: Boolean!
  ) on FIELD | FRAGMENT_SPREAD | INLINE_FRAGMENT

"Directs the executor to skip this field or fragment when the `if`'argument is true."
directive @skip(
    "Skipped when true."
    if: Boolean!
  ) on FIELD | FRAGMENT_SPREAD | INLINE_FRAGMENT

directive @provides(fields: String!) on FIELD_DEFINITION

directive @external on FIELD_DEFINITION

directive @extends on OBJECT | INTERFACE

directive @tag(name: String!) repeatable on SCALAR | OBJECT | FIELD_DEFINITION | ARGUMENT_DEFINITION | INTERFACE | UNION | ENUM | ENUM_VALUE | INPUT_OBJECT | INPUT_FIELD_DEFINITION

directive @include(if: Boolean!) on FIELD | FRAGMENT_SPREAD | INLINE_FRAGMENT

directive @requires(fields: _FieldSet!) on FIELD_DEFINITION

directive @key(fields: _FieldSet) repeatable on OBJECT | INTERFACE

directive @skip(if: Boolean!) on FIELD | FRAGMENT_SPREAD | INLINE_FRAGMENT

"Exposes a URL that specifies the behaviour of this scalar."
directive @specifiedBy(
    "The URL that specifies the behaviour of this scalar."
    url: String!
  ) on SCALAR
```

```
directive @deprecated(reason: String = "No longer supported") on FIELD_DEFINITION | ENUM_VALUE

"[product]"
type Product {
  inStock: Boolean
  name: String
  price: Int
  reviews: [Review]
  shippingEstimate: Int
  upc: String!
  weight: Int
}

"[]"
type Query {
  _namespace: String
  me: User
  productB(upc: String!): Product
  topProducts(first: Int = 5): [Product]
}

"[review]"
type Review @key(fields : "id") {
  body: String
  id: ID!
  product: Product @resolver(arguments : [{name : "upc", value : "UPC001"}], field : "productB")
}
```

```
"[user]"
type User {
  id: ID!
  name: String
  reviews: [Review]
  username: String
}

"A selection set"
scalar _FieldSet

"[review]"
input ResolverArgument {
  name: String!
  value: String!
}
```

As required the graphql schema is federated with the addition of unique namespaces. The Scenario one is satisfied as it meets the success criteria

## 2. Resolving a Simple Query

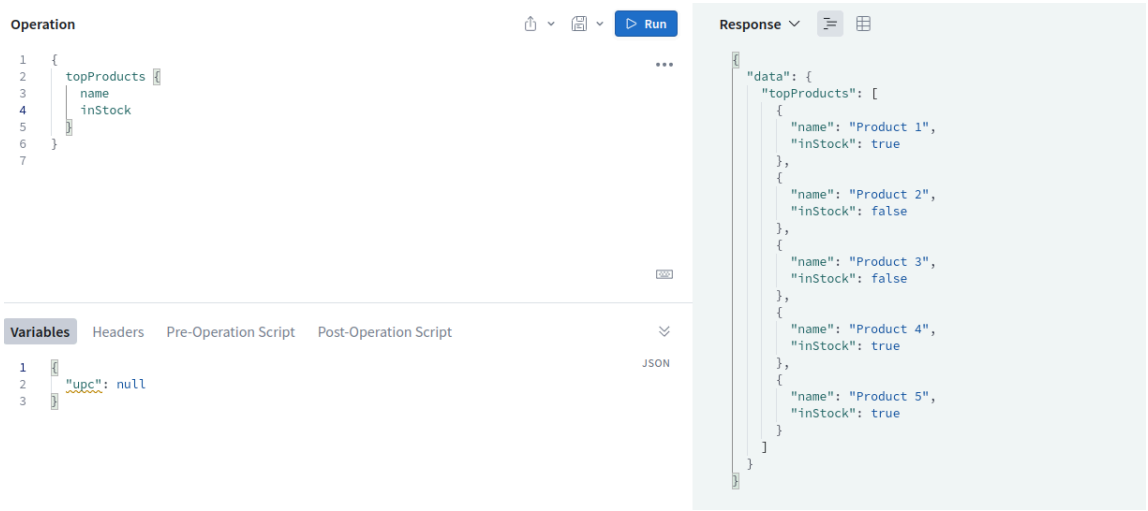
With the setup mentioned above, The POC for this scenario was carried out,

### 2.1. Requirement

- When a simple query is carried out resolve and fetch data from the associated service
- **Input** : Simple query,

```
{
 topProducts {
   name
   inStock
  }
}
```

- **Output** : Output with accurate data fetched from relevant service




### 2.2. Implementation

The following simple query was carried out that involved requesting data from two services , namely product and inventory,

#### 2.2.1. Query and Execution

```
ExecutionInput.Builder eiBuilder = ExecutionInput.newExecutionInput();
  eiBuilder.query("{ me { name reviews { body product {inStock} }}}");
eiBuilder.query("{\n" +
        "  topProducts {\n" +
        "    name\n" +
        "    inStock\n" +
        "  }\n" +
        "} ");
ExecutionInput executionInput = eiBuilder.build();

executionResult = graphQLOrchestrator.execute(executionInput).get().toSpecification();
```

**ExecutionInput** was to be initialized with the required query passed as a string and this was then executed via **graphQLOrchestrator.execute** which orchestrated the subqueries to the required services and returned the required response.

### 2.2.2. Subqueries and Service Invocation

- The first subquery was carried out on product service (port:8081) to resolve *'name'* Additionally *'upc',* which is the **entity** field of the product type was also fetched for the consequent subquery resolution automatically.

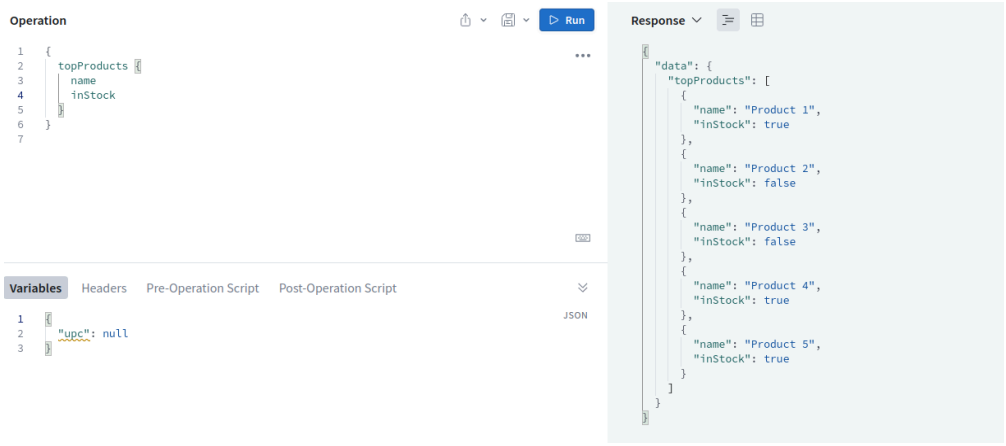
```
query QUERY {topProducts {name upc}}
{}
{}
-----------------------------------------------
query QUERY {topProducts {name upc}}
http://localhost:8081/graphql
```

- After the first subquery returns the data along with entity *'upc'*. The second subquery was called on the service inventory (port:8084). Here in order to fetch the relevant data for a product , the entity field data of each product was passed along as **representations** in the subquery.

```
query ($REPRESENTATIONS:[_Any!]!) {_entities(representations:$REPRESENTATIONS) {... on Product {inStock}}}
{REPRESENTATIONS=[{__typename=Product, upc=UPC001}, {__typename=Product, upc=UPC002}, {__typename=Product, upc=UPC003}, {__typename=Product, upc=UPC004},
{REPRESENTATIONS=[{__typename=Product, upc=UPC001}, {__typename=Product, upc=UPC002}, {__typename=Product, upc=UPC003}, {__typename=Product, upc=UPC004},
-----------------------------------------------
query ($REPRESENTATIONS:[_Any!]!) {_entities(representations:$REPRESENTATIONS) {... on Product {inStock}}}
http://localhost:8084/graphql
```

## 2.3. Output

The expected output was :




Received Output:

```
{data={topProducts=[{name=Product 1, inStock=true}, {name=Product 2, inStock=false},
{name=Product 3, inStock=false}, {name=Product 4, inStock=true}, {name=Product 5, inStock=true}]}}
```

## 2.4. Success Criteria and Additional Notes

As required the query is resolved .The Scenario two is satisfied as it meets the success criteria.

## 3. Resolving a Complex Query - Nested

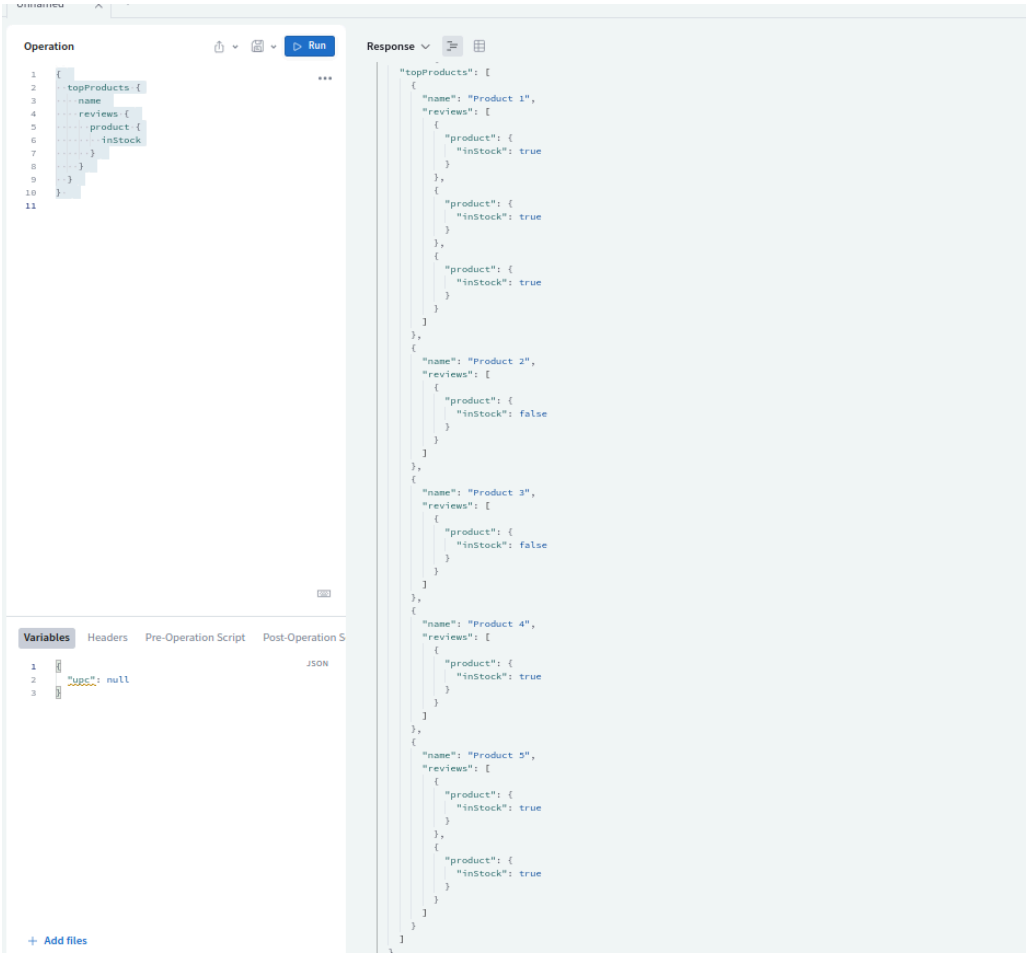
With the setup mentioned above, The POC for this scenario was carried out,

### 3.1. Requirement

- When a complex nested query is carried out resolve and fetch data from the associated service
- **Input** : Complex query,

```
{
  topProducts {
        name
        reviews {
        product {
        inStock
        }
        }
  }
}
```

- **Output** : Output with accurate data fetched from relevant service



### 3.2. Implementation

The following complex query that is nested was carried out that involving requesting data from multiple services , namely product and inventory and , reviews

#### 3.2.1. Query and Execution

```
ExecutionInput.Builder eiBuilder = ExecutionInput.newExecutionInput();
  eiBuilder.query("{ me { name reviews { body product {inStock} }}}");
eiBuilder.query("{\n" +
        "  topProducts {\n" +
        "    name\n" +
        "    reviews {\n" +
        "      product {\n" +
        "        inStock\n" +
        "      }\n" +
        "    }\n" +
        "  }\n" +
        "} \n ");
ExecutionInput executionInput = eiBuilder.build();
```

#### 3.2.2. Subqueries and Service Invocation

- The **first** subquery was carried out on product service (port:8081) to resolve *'name'* Additionally *'upc',* which is the **entity** field of the product type was also fetched for the consequent subquery resolution automatically.

```
query QUERY {topProducts {name upc}}
{}
{}
------------------------------------------------
query QUERY {topProducts {name upc}}
http://localhost:8081/graphql
Content-Type
application/json
```

- After the first subquery returns the data along with entity 'upc'. The **second** subquery was called on the service Reviews (port:8083). Here in order to fetch the relevant data for a review , the entity field data of each product was passed along as representations in the subquery.

```
query ($REPRESENTATIONS:[_Any!]!) {_entities(representations:$REPRESENTATIONS) {... on Product {reviews {id __typename}}}}
{REPRESENTATIONS=[{__typename=Product, upc=UPC001}, {__typename=Product, upc=UPC002}, {__typename=Product, upc=UPC003}, {__typename=Product, upc=UPC004},
{REPRESENTATIONS=[{__typename=Product, upc=UPC001}, {__typename=Product, upc=UPC002}, {__typename=Product, upc=UPC003}, {__typename=Product, upc=UPC004},
------------------------------------------------
query ($REPRESENTATIONS:[_Any!]!) {_entities(representations:$REPRESENTATIONS) {... on Product {reviews {id __typename}}}}
http://localhost:8083/graphql
Content-Type
application/json
```

- After the Second subquery returns the data along with entity 'id' of the review service that is needed for resolution of data called from another service. The **third** subquery was carried out on product service (port:8081) to resolve the entity associated with each product in order to formulate the final query

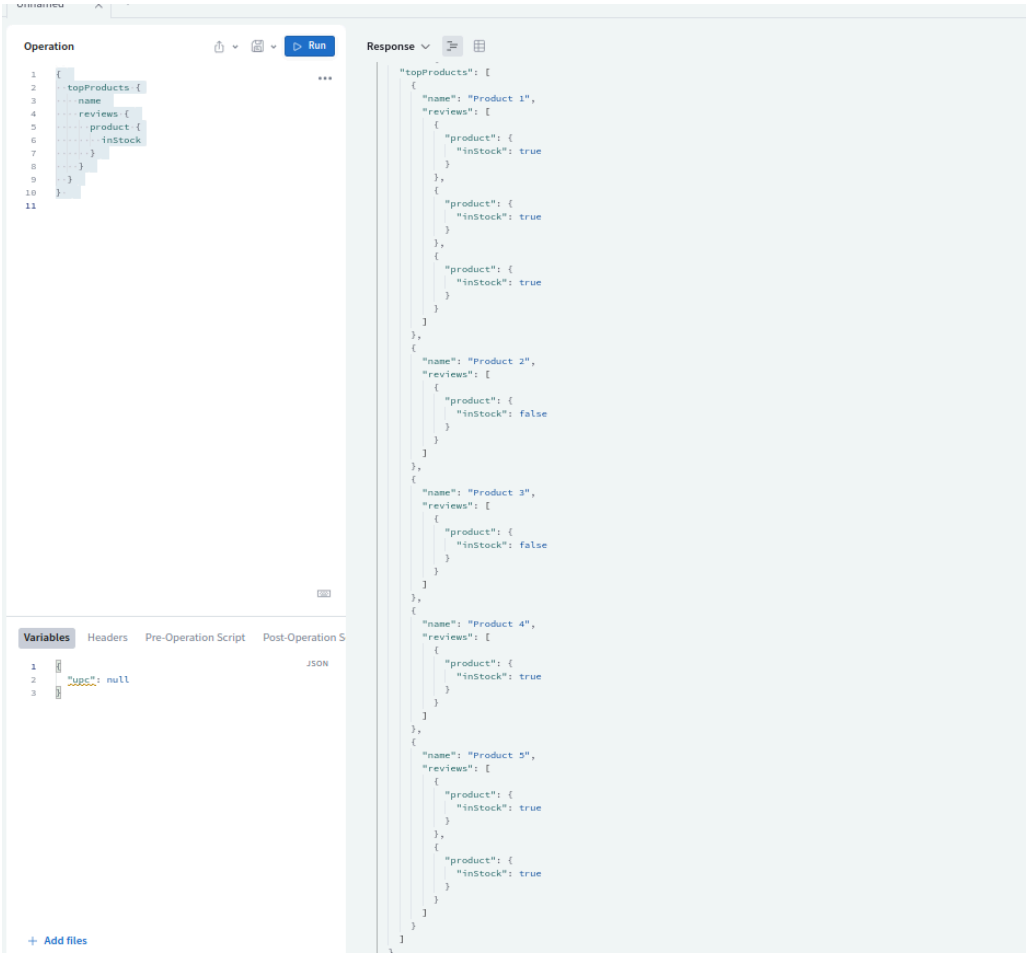
```
query Resolver_Directive_Query {productB_0:productB(upc:"UPC001") {upc} productB_1:productB(upc:"UPC001") {upc} productB_2:productB(upc:"UPC001") {upc} productB_3:productB(upc:"UPC001") {upc} productB_4:productB
 (upc:"UPC001") {upc} productB_5:productB(upc:"UPC001") {upc} productB_6:productB(upc:"UPC001") {upc} productB_7:productB(upc:"UPC001") {upc}}
{}
{}
------------------------------------------------
query Resolver_Directive_Query {productB_0:productB(upc:"UPC001") {upc} productB_1:productB(upc:"UPC001") {upc} productB_2:productB(upc:"UPC001") {upc} productB_3:productB(upc:"UPC001") {upc} productB_4:productB
 (upc:"UPC001") {upc} productB_5:productB(upc:"UPC001") {upc} productB_6:productB(upc:"UPC001") {upc} productB_7:productB(upc:"UPC001") {upc}}
http://localhost:8081/graphql
```

- The **fourth** final subquery was generated and called on service inventory (port 8084) to fetch if a product was inStock by passing the entity ‘upc’ as representations once again

```
User:reviews, review, review:Review:product, Product:reviews, user, Product:shippingEstimate])
////////////////////////////////////////////////////////
query ($REPRESENTATIONS:[_Any!]!) {_entities(representations:$REPRESENTATIONS) {... on Product {inStock}}}
{REPRESENTATIONS=[{__typename=Product, upc=UPC001}, {__typename=Product, upc=UPC001}, {__typename=Product, upc=UPC001}, {__typename=Product, upc=UPC001}, {__typename=Product,
 upc=UPC001}, {__typename=Product, upc=UPC001}, {__typename=Product, upc=UPC001}]}
{REPRESENTATIONS=[{__typename=Product, upc=UPC001}, {__typename=Product, upc=UPC001}, {__typename=Product, upc=UPC001}, {__typename=Product, upc=UPC001}, {__typename=Product,
 upc=UPC001}, {__typename=Product, upc=UPC001}, {__typename=Product, upc=UPC001}]}
------------------------------------------------
query ($REPRESENTATIONS:[_Any!]!) {_entities(representations:$REPRESENTATIONS) {... on Product {inStock}}}
http://localhost:8084/graphql
```

### 3.3. Output

The expected output was :




Received Output:

```
{data={topProducts=[{name=Product 1, reviews=[{product={inStock=true}}, {product={inStock=true}}, {product={inStock=true}}]}, {name=Product 2, reviews=[{product={inStock=true}}]}, {name=Product 3,
 reviews=[{product={inStock=true}}]}, {name=Product 4, reviews=[{product={inStock=true}}]}, {name=Product 5, reviews=[{product={inStock=true}}, {product={inStock=true}}]}]}}
```

### 3.4. Success Criteria and Additional Notes

As required the query is resolved .The Scenario two is satisfied as it meets the success criteria.

# 4. Resolving a Complex Query - multiple queries in the same level

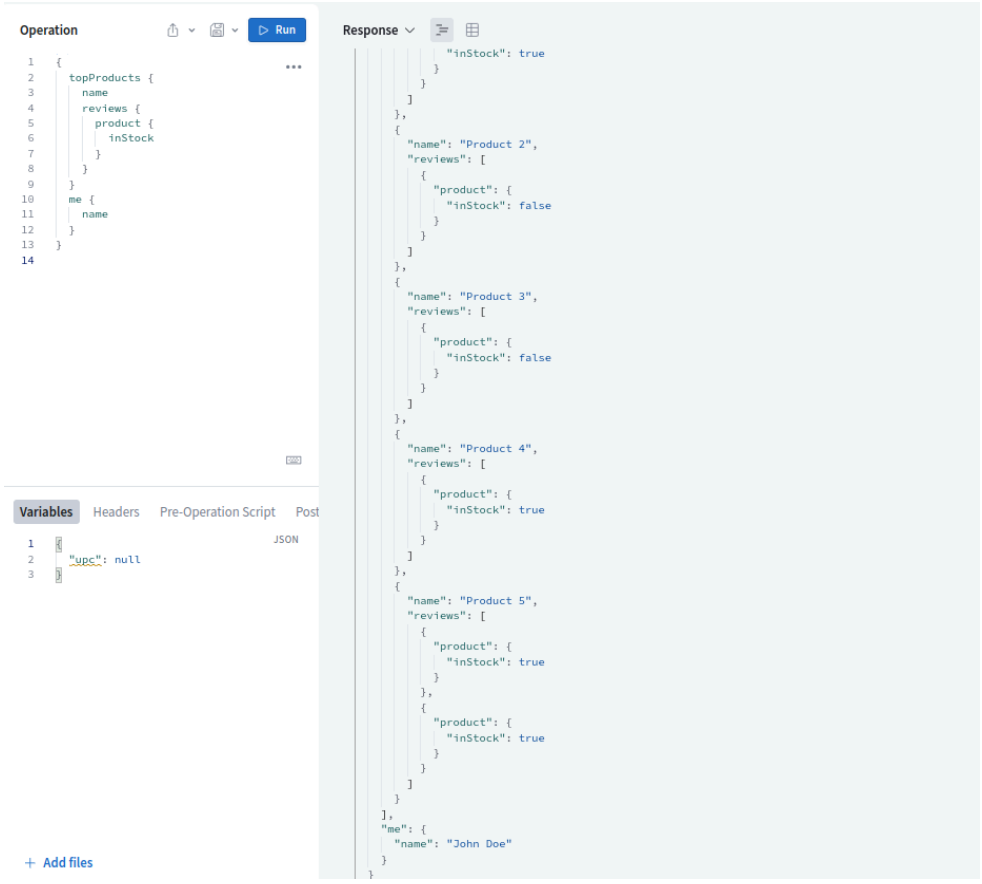
With the setup mentioned above, The POC for this scenario was carried out,

## 4.1. Requirement

- When a complex nested multilevel query is carried out resolve and fetch data from the associated service
- **Input** : Complex query where toProducts query fetched from product service and me query fetched from user service is in the same level

```
{
 topProducts {
   name
   reviews {
     product {
       inStock
     }
   }
 }
 me {
   name
 }
}
```

- **Output** : Output with accurate data fetched from relevant service



### 4.2. Implementation

The following complex query that is nested was carried out that involving requesting data from multiple services , namely product and inventory and , reviews

#### 4.2.1. Query and Execution

```
ExecutionInput.Builder eiBuilder = ExecutionInput.newExecutionInput();
eiBuilder.query("{\n" +
        "  topProducts {\n" +
        "    name\n" +
        "    reviews {\n" +
        "      product {\n" +
        "        inStock\n" +
        "      }\n" +
        "    }\n" +
        "  }\n" +
        "  me {\n" +
        "    name\n" +
        "  }\n" +
        "} \n");
ExecutionInput executionInput = eiBuilder.build();
```

#### 4.2.2. Subqueries and Service Invocation

- The **first** subquery was carried out on product service (port:8081) to resolve *'name'* Additionally *'upc',* which is the **entity** field of the product type was also fetched for the consequent subquery resolution automatically.

```
query QUERY {topProducts {name upc}}
{}
{}
-----------------------------------------------
query QUERY {topProducts {name upc}}
http://localhost:8081/graphql
Content-Type
application/json
```

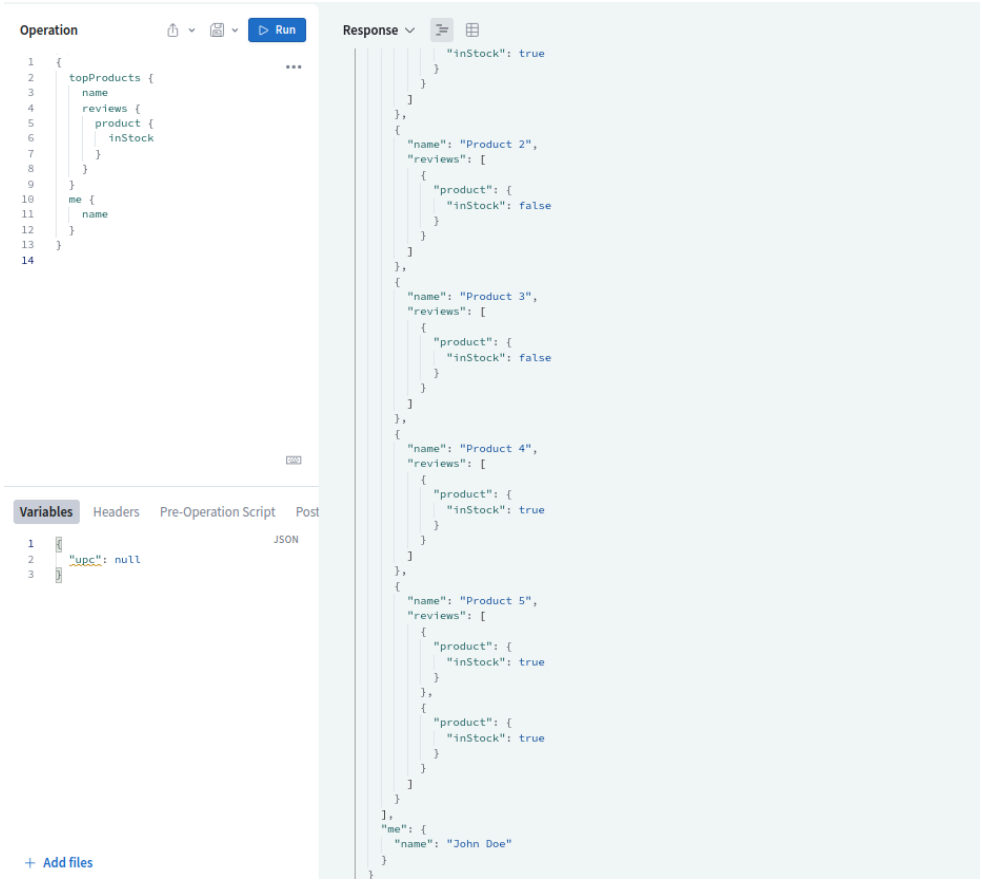
- Another subquery was carried out on service user to fetch data relevant to the query that was mentioned in the same level as the topProucts query.
  * It is important to note that the queries in the same level was resolved before moving on to the complex nested query that is present in the topProduct query

```
query QUERY {me {name id}}
{}
{}
-----------------------------------------------
query QUERY {me {name id}}
http://localhost:8082/graphql
```

- After the complex query was carried out as the section before

### 4.3. Output

The expected output was :




Received Output:

```
{data={topProducts=[{name=Product 1, reviews=[{product={inStock=true}}, {product={inStock=true}}, {product={inStock=true}}]}, {name=Product 2, reviews=[{product={inStock=true}}]}, {name=Product 3,
 reviews=[{product={inStock=true}}]}, {name=Product 4, reviews=[{product={inStock=true}}]}, {name=Product 5, reviews=[{product={inStock=true}}, {product={inStock=true}}]}], me={name=John Doe}}}
```

### 4.4. Success Criteria and Additional Notes

As required the query is resolved .The Scenario two is satisfied as it meets the success criteria.

# Appendix

1. Federated supergraph using apollo