Todos App's GraphQL Backend

App Overview

To demonstrate the use of GraphQL I have created an application to manager a Todos list. It is capable to create a new ToDo item in the list, remove it, or list all items. Additionally, each ToDo can be in a completed or uncompleted state, so the application allows the user to change its state accordingly.

Using the GraphQL API I will create, retrieve (an item or more), update, delete Todos from the list. Using the GraphQL introspection system I will demonstrate how we can find more information about what types and mutation are available to be applied over the ToDo items and list.

On the design aspect of this application, it was implemented using the Java programming language together with Spring Boot framework. The design pattern utilized to achieve the *Object-Relational Mapping* (ORM) in this project was the *Repository Pattern* (Fowler, 2002) using Spring *Boot Java Persistence API* (JPA) Data.

Although this is a very simple and small application it was been made always keeping in mind the design principles stated by (Gamma, Helm, Johnson, & Vlissides, 1995).

Queries

Queries are the main concept behind GraphQL which means *Graph Query Language*. Everything in GraphQL is related to queries. So, in the specific case of the Todos App, how we can query Todos? First, differently of REST APIs, GraphQL has just a single point to ask for a Representational State. Every call to GraphQL is just a POST, containing in its body the query you want, to your GraphQL single end point URI address.

For example, let's say a client (or a frontend app) wants to list all Todos in the list. The only thing required is creating a query with the fields of the Todo you want to receive and send a POST to GraphQL end point. See below an example of a query of Todos:

```
1.  // POST to http://localhost:8080/graphql
2.  query {
3.      todos {
4.          text
5.          completed
6.      }
7.  }
```

Which will receive as response:

```
1.  // Response from the POST to http://localhost:8080/graphql
2.  {
3.      "data": {
4.          "todos": [
```

```
            {
                "text": "Hey",
                "completed": false
            },
            {
                "text": "Hello!",
                "completed": false
            },
            {
                "text": "Hi",
                "completed": false
            }
        ]
    }
}
```

An interesting part of this is the concept of Schema. When you going to create a query, you must first define the schema of you GraphQL types that will exist in your system. Here the schema for the Todos App:

```
// GraphQL Schema
type Todo {
    id: ID!
    text: String!
    completed: Boolean
}

type Query {
    todos(count: Int): [Todo]
    todo(id: ID!): Todo
}

type Mutation {
    createTodo(text: String!): Todo
    toggleTodoCompleted(id: ID!): Todo
    deleteTodo(id: ID!): Todo
    deleteAllTodos: [Todo]
}
```

Mutations

Up to now we have talked only about Queries which is a key concept in GraphQL. As I have said in GraphQL everything is query, but how I can change or create information if the only thing I have is queries? Simple! Using queries, as well, but if a subtle difference. We are going to use a special type of queries called mutation queries. Below is an example how you can create a new ToDo using this type of queries:

```
// POST to http://localhost:8080/graphql
mutation {
    createTodo(text: "Yep!") {
        text
        completed
    }
}
```

Note you have mainly the same structure as before with the difference that now you inform the type `mutation`. Another thing to pay attention is that you not only call the mutation itself, you also inform what you what to receive after the mutation is done. See what you get from the above call:

```
// Response from the POST to http://localhost:8080/graphql
{
    "data": {
        "createTodo": {
            "text": "Yep!",
            "completed": false
        }
    }
}
```

As you can see, after the creation you receive only the fields you asked so of the created ToDo.

Introspection

Finally, another powerful capability of the GraphQL is what they call introspection system. Let's suppose some another developer decide to create a frontend application to my ToDo API. He knows nothing about which types or mutations I have, the only thing he knows is that my API is for manager a list of Todos. What should he do?

For that, GraphQL has a special type of query field called `__schema`. So, if a developer need to know all type of queries He can use from my API just need to make a POST call with the following body:

```
// POST to http://localhost:8080/graphql
{
  __type(name: "Query") {
      name
      fields {
          name
          type {
              name
              kind
          }
      }
  }
}
```

Which my GraphQL API will respond:

```
// Response from the POST to http://localhost:8080/graphql
{
    "data": {
        "__type": {
            "name": "Query",
            "fields": [
                {
                    "name": "todos",
                    "type": {
                        "name": null,
```

```
                        "kind": "LIST"
                    }
                },
                {
                    "name": "todo",
                    "type": {
                        "name": "Todo",
                        "kind": "OBJECT"
                    }
                }
            ]
        }
    }
}
```

Now the developer knows what to implement to his frontend talk to the ToDo API without even to go to the API documentation.

REST vs GraphQL Comparation

For a better understand of GraphQL in terms of REST API I present below a chart matching the REST operation against GraphQL's:

REST			GraphQL		
HTTP Method	**Request Has Body**	**Response Has Body**	**HTTP Method**	**Request Body**	**Response Body**
GET	No	Yes	POST	query {...}	{ data: {...} }
POST	Yes	Yes	POST	mutation {...}	{ data: {...} }
PUT/PATCH	Yes	Yes	POST	mutation {...}	{ data: {...} }
DELETE	No	Yes	POST	mutation{...}	{ data: {...} }
OPTIONS	Optional	Yes	POST	__Schema{...}, __Type{...}, ...	{ data: {...} }

As you can see GraphQL uses only POST and is not the HTTP method which defines the creation or the modification in the data, but the mutation you define in the GraphQL's Schema.

Conclusion

As you can see from the examples above, GraphQL is a very power alternative for solving some of the REST API drawback, since it can reduce the traffic and the bandwidth usage (since it allow retrieve only the piece of information needed) and number of call (which reduces consequently latency) in the network specially in times when everything is been sending to the cloud and in times of IoT computing.

GraphQL's use of a declarative language eliminates over-fetching and avoids the multiple roundtrip problem, thereby increasing network efficiency, because only the data that is actually requested by the client is transferred. Over-fetching and the problem of multiple round-trips often occur in the context of a resource-based approach such as REST because the server

defines the resource layout. Contrary to REST, GraphQL does not account for caching in the specification, which has to be handled solely on the client (Eizinger, 2017).

For GraphQL the Achilles' heel bear on the processing time of the query in the server side. There are already studies related to this matter, which stats that issue can be overcome using some computational technics and algorithms (Hartig, 2018).

References

Eizinger, T. (2017). API Design in Distributed Systems: A Comparison between GraphQL and REST. Wien.

Fowler, M. (2002). *Patterns of Enterprise Application Architecture.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Gamma, D. E., Helm, R., Johnson, R., & Vlissides, a. J. (1995). *Design Patterns: Elements of Reusable Object-oriented Software.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Hartig, O. a. (2018). Semantics and complexity of GraphQL. *Proceedings of the 2018 World Wide Web Conference* (pp. 1155--1164). International World Wide Web Conferences Steering Committee.