((CENTRIC))

GraphQL

Shashank Saurabh

August, 2022

Why GraphQL ??

Lets understand with an example

Limitations of REST APIs

- Resource centric: many endpoints
- The endpoints are often organized the way they are stored in DB, not the way the clients need it.
- Large and fixed responses.
- Getting complex response requires multiple request.
- We would like to get on a single request, ALL the data that we need in single request and ONLY data the data we need.

What is GraphQL? 

- GraphQL stands for Graph Query Language.
- GraphQL was developed by Facebook.
- It is an open-source data query and manipulation language for API.
- It allows clients, to write queries that have exact data shape they want.
- Like SQL here we query to fetch the data, You have flexibility which data to fetch.
- Like REST APIs you can use GraphQL with different programming languages.
- GraphQL was internally developed in 2012 and its open source version was released in 2015.
- GraphQL official website is https://graphql.org/ .

As per official website :

A query language for your API

GraphQL is a query language for APIs and a runtime for fulfilling those queries with your existing data. GraphQL provides a complete and understandable description of the data in your API, gives clients the power to ask for exactly what they need and nothing more, makes it easier to evolve APIs over time, and enables powerful developer tools.

Language supported by GraphQL

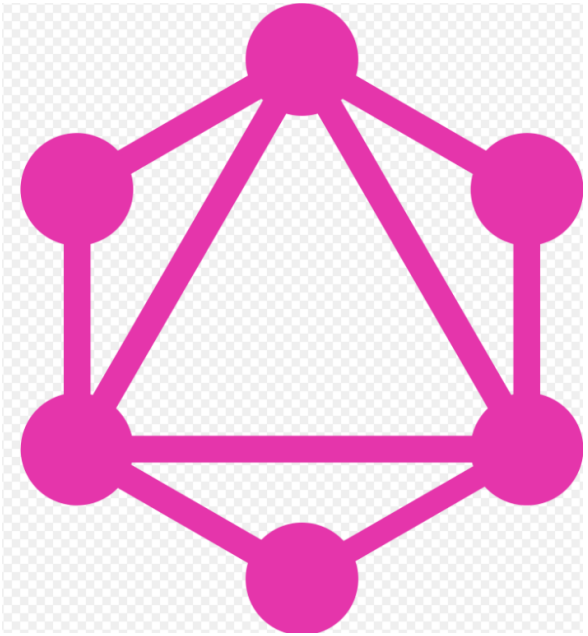
- Java
- C# /.Net
- Clojure
- Erlang
- Go
- Groovy
- JavaScript
- PHP
- Python
- Scala
- Ruby etc..

Rest API vs GraphQL

- A REST API is an "architectural concept" for network-based software. GraphQL, on the other hand, is a query language.
- REST API has fixed response where as GraphQL provides you flexibility to tailor the response.
- REST has been used to make new APIs, while the focus of GraphQL has been to optimize for performance and flexibility.
- REST API has different HTTP methods and different endpoints for each resource whereas GraphQL works on single endpoint.

GraphQL

- Query
- Mutation
- Schema
- Types
- Resolvers



Query

- GraphQL query is used to fetch the data
- You can compare it with GET REST APIs.
- Unlike a Restful API, GraphQL allows a user to restrict fields that should be fetched from the server.
- This means smaller queries and lesser traffic over the network; which in turn reduces the response time.

Mutation

- GraphQL Mutation is used to **Create, Update or Delete** the data.
- You can compare it with POST, PUT or DELETE REST APIs.
- Mutation also return data just like query does.

Schema

- Schema provides flexibility to consumer that which attributes they want in the response.
- It has .graphqls file extension.
- Contract between consumer and the provider on to get and alter the data for the application.
- Schema is collection of GraphQL types. Query and Mutation are the root types in the schema i.e entry point of the application.

Type

- GraphQL is a strongly types language.
- Type system defines various data types that can be used in a GraphQL application.
- The type system define the schema.

List of Types :

- Scalar Type
- Object Type
- Query Type
- Mutation Type
- Enum Type
- Non-Nullable Type
- List Type
- Input type

Type

Scalar Type

Scalar Types represent the primitive leaf values in the GraphQL type system.

GraphQL responses take the form of a hierarchical tree, the leaves of these tree are GraphQL Scalar.

Built-in Scalar

- Int : A signed 32-bit integer.
- Float : A signed double-precision floating-point value.
- String : A UTF-8 character sequence.
- Boolean : true or false
- ID: The ID scalar type represents a unique identifier, often used to refetch an object or as the key for a cache.

Apart from these Built-in scalar types we also have some custom Scalar like Date.

Syntax *- field : data_type*

Type

Query Type

Entry point to other specific types.

A GraphQL query is used to read or fetch values.

Syntax

```
type Query {
  field1: data_type
  field2:data_type
  field2(param1:data_type,param2:data_type,...paramN:data_type):data_type
  }
```

Type

Object Type:

The Object type is the most common type used in a schema and represent a group of fields.

Each field inside an Object type maps to another type, thereby allowing nested types.

Syntax

```
type object_type_name
 {
   feild1: data_type
   feild2: data_type
   feild3: data_type
   …..
    feildn: data_type
  }
```

Type

Mutation Type

Mutations are operations sent to the server to create, update or delete data.

Mutation is one of the root-level data-types in GraphQL. The Query type defines the entry-points for data-fetching operations whereas the Mutation type specifies the entry points for data-manipulation operations.

Syntax

```
type Mutation {
    field1: data_type
    field2(param1:data_type,param2:data_type,...paramN:data_type):data_type
    }
```

Type

Enum Type

An Enum is similar to a scalar type. Enums are useful in a situation where the value for a field must be from a prescribed list of options.

Syntax

```
type enum_name{
        value1
        value2
 }
```

List Type

Lists can be used to represent an array of values of specific type. Lists are defined with a type modifier [] that wraps object types, scalars, and enums

Syntax

```
field:[data_type]
```

Type

Non-Nullable Type

By default, each of the core scalar types can be set to null. In other words, these types can either return a value of the specified type or they can have no value. To override this default and specify that a field must be defined, an exclamation mark (!) can be appended to a type.

Syntax

```
field:data_type!
```

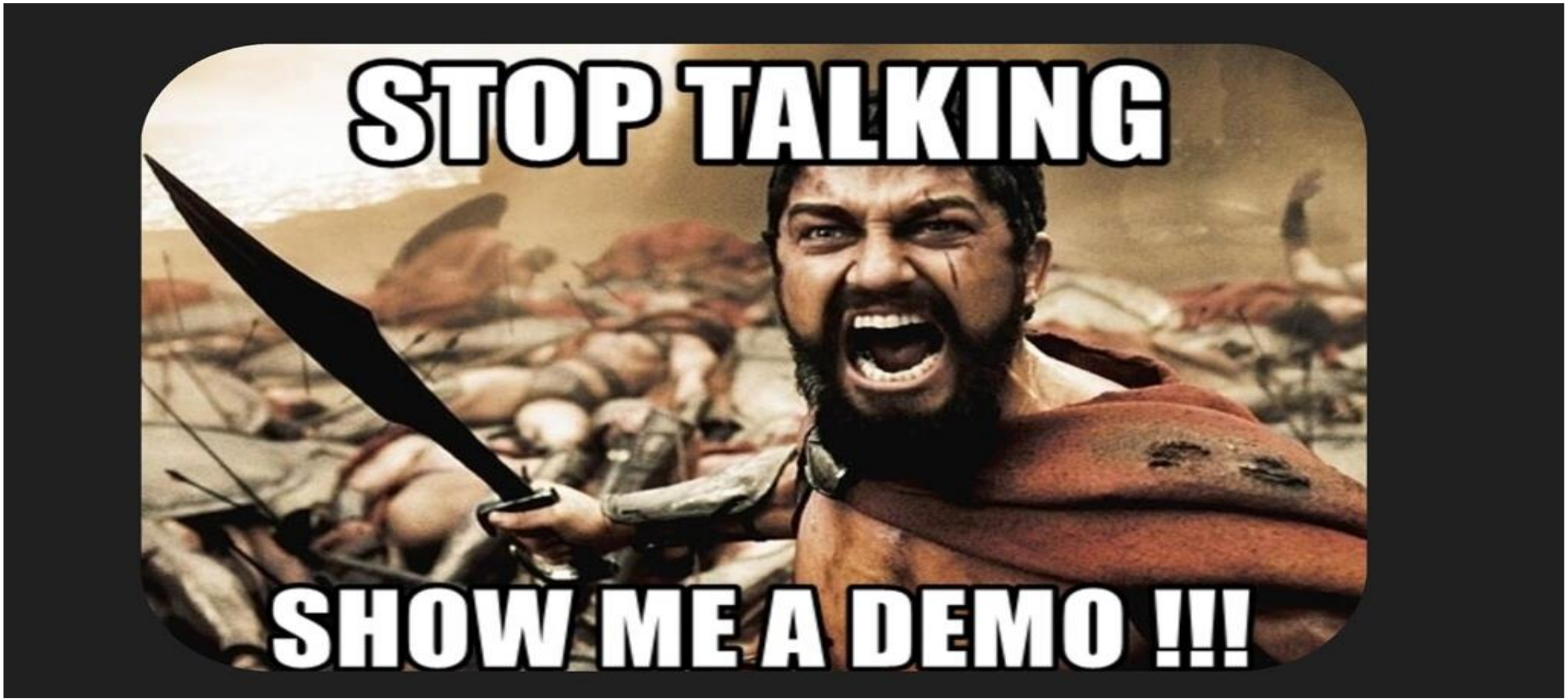
Input Type

Input type look exactly same as the Object type. Mostly it is used with Mutation type for data operation like add, edit and delete.

Syntax

```
input object_type_name
 {
   feild1: data_type
   feild2: data_type
   …..
    feildn: data_type
  }
```

Time for demo



Resolver

Every field on every type is backed by a function called a resolver.

A resolver is a function that resolves a value for a type or field in a schema.

Resolvers can return objects or scalars like Strings, Numbers, Booleans, etc.

If an Object is returned, execution continues to the next child field. If a scalar is returned (typically at a leaf node), execution completes.

Thank you