GRAPH QL

- → developed by facebook
- → allows for requesting and getting data exactly you need (no overfetching or underfetching)
- → alternative to REST
- → Has single endpoint instead of multiple routes.

Ex – For getting data – QUERY

```
{ film (filmID : 1) {
          title
          director
          producers
      }
  }
```

collection from which data required

fields required

Result –

```
{   "data" : {
        "film" : {
            "title" : "A New Hope",
            "director" : "George Lucas",
            "producers" : ["Gary Kurtz",
                           "Rick McCallum"
                          ]
        }
    }
}
```

★ Got exactly what asked for in query.

→ Moree querying

Suppose theres a Person collection with feilads like 'name', 'eyeColor', 'homeWorld' where homeWorld is a 'Planet' collection with a 'name', 'population', 'orbit' etc—

or

```
Person — name
         eyeColor
            :
            :
         homeWorld — name
          : (Planet)  population
          :            orbit Period
```

To fetch such nested data query is like —
(Also suppose we want a film data also with this)

— Query

```
{ film(filmID:1){
     title
     director
     producer
  }
  person(personID:5){
        name
        homeworld {
                   name
                   climates
                  }
        }
}
```

Data

```
{ "data": {
     "film" {
       "title": "A New Hope",
       "director": "George Lucas",
       "producers": ["Gary Kurtz",
                     "Rick McCallum"
                    ]
     },
     "person": {
        "name": "Leia Organ
        "homeworld": {
             "name": "Tatooin
           "population": 200000
           }
        }
     }
}
```

GraphQL setup with graphql packge & express -

npm install express graphql express-graphql

Important Operations & Key Concepts in GraphQL

① Schema ② Type System ③ Resolver

> These three are important to setup a GraphQL API

④ Query ⑤ Mutation

> These two are important for CRUD operation on a GraphQL API

Setting up & Defining the GraphQL schema

1- Schema - strongly typed structure that defines the shape of data.

- defines how the data will be stored, like its bluprint. Specifies types, their field and entrypoints & (Mutation, Query, Subscription)

2- Type System → ① Query > entry point for all read operation

- Ex - (Schema)

```
type Query {
    products : [Product]
    orders : [Order]
}
```

list of Product & Order!

• explanation - defines what we can query (operation) and what will we get.

- products and orders are fields that return a list of 'Product' and 'Order' objects respectively

→ ② Mutation > another type linked to same named operation (ie Mutation) for creating, updating or deleting data in the existing API

> Explained later —

→ Custom Types

API developer decides on what all similarly & closely related data can be made to fall under same umbrella for which he can create a Custom Type.

Ex -
```
type Query {
        products : [Product]
        orders : [Orders]
      }
type Product {
            id : ID!
            description : String!
            reviews : [Review]
            price : Float!
         }
```

represent a 'product' with an id (mandatory, description (mandatory), reviews (optional) and price (mandatory) field.

```
type Order {
          data : String!
          subtotal : Float!
            items : [OrderItem]
        }
```

represents a customer order, including date, subtotal cost and a list of order items.

```
type Review {
          rating : Int!
          Comment : String
        }
type OrderItem {
          product : Product!
          quantity : Int!
        }
```

3- **Resolvers** - a function that's responsible for returning the data for a field in GraphQL query.

- Its most important part of schema setup after you've defined entrypoints (Query, Mutation, Subscription) and their fields.
- with resolver you decide how the queries and mutations will be handled

- Resolver Signature →

```
(parent, args, context, info)=>{
    // resolving logic
}
```

where other parameters can simply be looked up online but most important 'args' is a object containing arguments passed to the field while querying or mutationg.

```
Ex - Query{
        getProducts(id: 2){
            name
            color
            description
        }
```

* explanation: when this query is made then getProduct resolver will be populated with args-id=2 which then it can use to get ~~data~~ product with id=2 from db.

example of advanced query with filtering

```
- Ex: Query : {
    products: ()=> { return getAllProducts()
},
```

query object for Query Resolver

```
, productsByPrice : ( _ , args) => {
                    return getProductsByPrice (args.min, args.max)
          },

  productsById: ( _ ,args ) => {
                    return getProductById (args.id )
             }
  }
```

~~these~~ products , productsByPrice , productsById are handled upon querying by resolver function against them.

Following is how products.schema.js could look like—

```
type Query {
        products : [Product]
        productsByPrice(min: Float!, max: Float) : [Product]
        productById (id: String ! ) : [Product]
     }

  type Product {
         :
         :
     }

  ~~type~~ a
```

return type for Query is list of Product

Implementing Mutations.

S1 - In schema define a Mutation type

```
Ex - type Mutation {
           newProduct (id: ID! , description: String!,
                       price : Floate ! ) : Product
     }
```

return type upon mutation

S2 - Define the Resolver for the mutation ; use 'args' to pass value to a model function (this makes query to database)

```
Ex - Mutation : {
            newProduct : (_, args) => {
                    return addNewProduct (args. id,
                                          args. description,
                                          args. price)
            }
```

S3 - In Model file create and export the addNewProduct() function required in Resolver and which returns a 'Product' as defined per Mutation Schema. This function handles logic for interacting with database or mock data.

```
Ex - function addNewProduct ( id, description, price)
     {
          // adding product to db
          :
          return newProduct;
     }
```

Now mutation can be made on client side that looks like —

```
mutation {
      addNewProduct ( id : "purpleshirt", description: "A
                      Purple Shirt", price: 10.22) {
            id
            description
            price
      }
```

what you need from returned Product (new Product)

Setting up a graphQL server — (modular)

1) Install required npm packages
2) Install graphql-tools (for supporting modular (much cleaner) approach)

3) In server.js —

```
const path = require('path')

const express = require('express')
const app = express()

const {graphqlHTTP} = require('express-graphql')
 const { makeExecutableSchema } = require ('@graphql-tools/schema'
 const { loadFilesSync } = require ('@graphql-tools/ loadfiles')
```

recursively loads all files and stores them in array

```
const typesArray = loadFilesSync(path.join(_dirname, '**/*.graphql'
const resolversArray = loadFilesSync (path.join(_dirname, '**/*.resolver.js')
 const schema = makeExecutableSchema ({
            typeDefs: typesArray,
            resolvers: resolversArray
         })
```

Combine schema + resolvers clearly

GraphQL Middleware

mounting GraphQL server at route '/graphql'

```
app.use ('/graphql', graphqlHTTP({
            schema : schema,
            graphiql: true   // enable 'graphiql' playground
      )});
```

listen to server at Port 3000

```
app.listen(3000, () => {
        console.log ("Running a graphql se
      }
   )
```

- Further more practical Approach —

Use Apollo ?

→ Apollo is a full package of tools to build, consume and manage GraphQL APIs.

→ built on top of graphQL

→ Includes (mainly used) — Apollo Server - build GrapQL server

Apollo Client - a state management library for fetching & managing GraphQL in frontend

→ Previous code can be updated with —

① @apollo/server and expressMiddleware instead of using express-graphql

② Apollo ~~does~~ is compatible with most of stuff including 'graphql-tools'.

③ Few changes & it will work like MAGIC!

Ex- Apollo do not use graphqlHTTP insteadd it uses expressMiddleware ( )

- Extra

→ Aliasing queries & mutation to run same query/mutation multiple times in a single request.

This avoids conflicts from repeating same mutation/query

Ex - Add Multiple Reviews

```
mutation {
    pant: newReview (id: "beigePant", rating: 5, Comment: "~~"
      { id }
    jacket: newReview (id: "bigbluejacket", rating: 5, comment: "~~"
      { id }
}
```