Contents

[Url’s 2](#_Toc7973032)

[Good css library to use 2](#_Toc7973033)

[Basics 2](#_Toc7973034)

[Mechanism 2](#_Toc7973035)

[Process for building a GQL server 3](#_Toc7973036)

[Step 1: Environment setup 3](#_Toc7973037)

[Setp 2: index.js 4](#_Toc7973038)

[Step 3: create schema in schema.graphql 4](#_Toc7973039)

[Step 4: Create resolvers in resolvers.js 5](#_Toc7973040)

[Step 5: Create data 6](#_Toc7973041)

[Process for building a GQL client 8](#_Toc7973042)

[Step 1: Environment setup 8](#_Toc7973043)

[Step 2: Setup of Apollo client and graphQl hook 8](#_Toc7973044)

[Step 3: Call the hook 9](#_Toc7973045)

# Revisit on 11-10-2019

Very good short implementation

<https://www.robinwieruch.de/graphql-apollo-server-tutorial>

* adam2-test-graphql-server is a short implementation of Apollo GraphQL server having just three files index.js, schema.js and resolvers.js.
* In GraphQL you need to define the schema. The schema has Query and types. In Query you mention all the fields which will be available for query. There are other ‘type’ in the schema definition.
* Resolvers implement the methods which return the fields which are defined in Query of schema. Resolvers can return promise which is automatically computed by query.

# Python + GraphQL + Flask

<https://www.agiliq.com/blog/2019/08/getting-started-with-python-graphene/>

<https://codeburst.io/how-to-build-a-graphql-wrapper-for-a-restful-api-in-python-b49767676630>

Slightly off the track but useful

<https://stackabuse.com/building-a-graphql-api-with-django/>

//This is alternative to Graphene

<https://ariadnegraphql.org/>

//With async graphql

<https://cito.github.io/blog/shakespeare-with-graphql/>

See documentation in learning /python folder

# Url’s

Core concepts

<https://www.howtographql.com/basics/2-core-concepts/>

Basic tutorial for GraphQL with React (Best one)

<https://www.tutorialspoint.com/graphql/graphql_react_integration.htm>

# Good css library to use

<http://tachyons.io/>

completed study of GraphQl on 05-05-2019

# Basics

GraphQL has query, mutations and subscriptions. You need to define type and schema. An example of type and schema is as below

type Query {

allPersons(last: Int): [Person!]!

}

type Mutation {

createPerson(name: String!, age: Int!): Person!

}

type Subscription {

newPerson: Person!

}

type Person {

name: String!

age: Int!

posts: [Post!]!

}

type Post {

title: String!

author: Person!

}

# Mechanism

You create a GraphQL server. You can query a successful server from browser by <http://localhost:xxx/graphql>. It should fire an interface where you can run GQL queries and get results.

You need to also create a GQL client which programmatically connects to GQL server. Basically, you can directly connect to GQL server from client by sending GQL queries in body of post requests but a typical GQL client provides some other advanced concepts.

 The [graphql](https://github.com/graphql/graphql-js) library package allows GraphQL queries, mutations, and subscriptions on both the client and server side.

I found following approach good for GraphQL

* Create GraphQL server using Apollo. Here you define schema and resolvers. Resolvers can point to local files, any local database, any cloud database, any api. Resolvers return promise. As a result you get all sort of data from a single server. A successful GQL server exposes GQL playground at <http://localhost:3000/graphql>. Here localhost:3000 is typical url of GQL server. In GQL playground you can test your query and mutations. It is very easily possible that the resolvers point to various complex SQL’s for database
* I successfully created mutations resolvers in GQL server which did update and inserts in database.
* I investigate Hasura which directly exposes GQL playground from database, there is no need of any GQL server. But I realized that you need to convert your complex SQL to Views. Similarly complex SQl queries are cumbersome with Prisma based approach which uses ORM. I found above approach of creating GQL server from scratch providing all individual SQL’s to resolvers. This approach is highly flexible.
* It is also possible to use a generic way of calling SQl from GQL using Json.stringify() the JSON object and returning it as string.
* The two projects of GraphQl server and client are in github with names adam2-graphql-server and adam2-graphql-client

# Process for building a GQL server

It uses Apollo framework.

## Step 1: Environment setup

Create a new node.js application by npm init adam2-graphql-server. Include following dependencies in package.json and do npm install

"dependencies": {

"apollo-server-express": "^2.4.8",

"axios": "^0.18.0",

"body-parser": "^1.19.0",

"cors": "^2.8.5",

"express": "^4.16.4",

"graphql": "^14.2.1",

"graphql-tools": "^4.0.4",

"notarealdb": "^0.2.2",

"pg-promise": "^8.6.5"

}

## Setp 2: index.js

const cors = require('cors');

const express = require('express');

const { ApolloServer } = require('apollo-server-express');

const port = process.env.PORT || 9000;

const app = express();

const fs = require('fs')

const typeDefs = fs.readFileSync('./schema.graphql',{encoding:'utf-8'})

const resolvers = require('./resolvers')

app.use(cors());

const server = new ApolloServer({ typeDefs, resolvers });

server.applyMiddleware({ app });

app.listen(

port, () => console.info(

`Server started on port ${port}`

)

);

## Step 3: create schema in schema.graphql

type Query {

    greeting: String

    test: String

    students: [Student]

    colleges: [College]

    contacts: [Contact]

    brands: [Brand]

}

type Generic {

    gen:String

}

type Brand {

    id: ID!

    name: String

}

type Student {

    id: ID!

    firstName: String

    lastName: String

    password: String

    collegeId: String

}

type College {

    id: ID!

    name: String

    location: String

    rating: Float

}

type Contact {

    isActive: Boolean

    eyeColor: String

    name: String

    gender: String

    company: String

    email: String

    phone: String

    address: String

    age: Int

}

type Mutation {

    insertTest(name: String, address: String): Test

}

type Test {

    name: String

    address: String

}

## Step 4: Create resolvers in resolvers.js

const db = require('./db');

const axios = require('axios');

const pgp = require('pg-promise')();

const cn = {

    host: 'localhost',

    port: 5432,

    database: 'ecomm',

    user: 'postgres',

    password: 'su$hant123'

};

const pdb = pgp(cn);

const sqls = {

    'sql-category': 'select id, label from category',

    brands: `select \* from brand`

};

const qr = async (args) => {

    const data = await pdb.any(sqls[args]);

    return data;

};

const Query = {

    test: () => 'Test Success, GraphQL server is up & running !!',

    greeting: () => 'Hello how are you',

    colleges: () => db.colleges.list(),

    students: () => db.students.list(),

    contacts: () =>

        axios.get('http://chisel.cloudjiffy.net/contacts/short').then(function(response) {

            return response.data;

        }),

    brands: () => qr('brands')

};

const Mutation = {

    insertTest: async (parent, args) => {

        const res = await pdb.none('insert into test(name,address) values(${name}, ${address})', { ...args });

        return { ...args };

    }

};

module.exports = { Query, Mutation };

## Step 5: Create data

Some local data is converted to database from notarealdb library. for that create data folder and add two json files students .json and colleges.json.

Colleges.json

[

{

"id": "col-101",

"name": "AMU",

"location": "Uttar Pradesh",

"rating":5.0

},

{

"id": "col-102",

"name": "CUSAT",

"location": "Kerala",

"rating":4.5

}

]

Students.json

[

{

"id": "S1001",

"firstName":"Mohtashim",

"lastName":"Mohammad",

"email": "mohtashim.mohammad@tutorialpoint.org",

"password": "pass123",

"collegeId": "col-102"

},

{

"id": "S1002",

"email": "kannan.sudhakaran@tutorialpoint.org",

"firstName":"Kannan",

"lastName":"Sudhakaran",

"password": "pass123",

"collegeId": "col-101"

},

{

"id": "S1003",

"email": "kiran.panigrahi@tutorialpoint.org",

"firstName":"Kiran",

"lastName":"Panigrahi",

"password": "pass123",

"collegeId": "col-101"

}

]

In db.js

const { DataStore } = require('notarealdb');

const store = new DataStore('./data');

module.exports = {

students:store.collection('students'),

colleges:store.collection('colleges')

};

# Process for building a GQL client

## Step 1: Environment setup

Create-react-app adam2-graphql-client –typescript

Cd adam2-graphql-client

Npm install Apollo-boost graphql --save

The Apollo-boost is a set of libraries. If you check the package.json of Apollo-boost you will find following dependencies, that means these libraries are installed.

"apollo-cache": "^1.2.1",

"apollo-cache-inmemory": "^1.5.1",

"apollo-client": "^2.5.1",

"apollo-link": "^1.0.6",

"apollo-link-error": "^1.0.3",

"apollo-link-http": "^1.3.1",

"graphql-tag": "^2.4.2",

"ts-invariant": "^0.2.1",

"tslib": "^1.9.3"

The graphql library provides a function gql. This parses the graphql query in back tick notation and it works just as GraphQL playground tool as if in code. You can write the same query in gql function. Gql function can be used with back tick or as a normal function.

From the project src I removed all files except App.tsx and index.tsx.

## Step 2: Setup of Apollo client and graphQl hook

Create a hook file named as gql-hook.tsx

import { ApolloClient, HttpLink, InMemoryCache } from 'apollo-boost';

import gql from 'graphql-tag';

const endPointUrl = 'http://localhost:9000/graphql';

const client = new ApolloClient({

    link: new HttpLink({ uri: endPointUrl }),

    cache: new InMemoryCache()

});

const queries: any = {

    a: `query {

brands {

id

name

}}`

};

function useGql() {

    async function doQuery(qr: string) {

        const query = gql(queries[qr]);

const { data } = await client.query({ query });

        return { data };

    }

    return { doQuery };

}

export { useGql };

This hook file has everything related to graphQl. All the queries can be stored in a different file with key index object format. While calling the hook you only pass the query key as argument. The useGql function is exported from the hook.

It is assumed that a graphQl server is running at 'http://localhost:9000/graphql' and a query as query{ brands{id name}} returns brands with name property from the database.

## Step 3: Call the hook

In App.tsx file put following code

import React, { useState } from 'react';

import './App.css';

import { useGql } from './gql-hook';

function App() {

  const { doQuery } = useGql();

  const [ brands, setBrands ] = useState([]);

  return (

    <div>

      <button

        onClick={async () => {

          const d = await doQuery('a');

          setBrands(d.data.brands);

        }}

      >

        Do query

      </button>

      {brands.map((x: any, index: number) => {

        return <div key={index}>{x.name}</div>;

      })}

    </div>

  );

}

export default App;

This displays the brands in browser after execution of graphQl query