JAMSTACK: No need of server, just use of API

API is the main leading

Any kind of server can build a REST API.

For mobile app, we need to APIs. One is for the desktop and one is for the mobile. It is very difficult to maintain. REST returns all

**GraphQL provide Query language to contact to the server with in the specified terms (relevant data and save time, bandwidth etc)**

**We can deal with multiple data or api or type in single query**

**Means we can fetch multiple apis in the single call, unlike in REST API, we have to do one call for one api. So for fetching two apis, we need to do two calls to server, but in graphQL, single call is only needed to fetch multiple apis**

GRAPHQL give us support of what we want. It has a single point.

**In graphql, types define the structure of the data to the client**

GRAPHQL documents API very nicely

It has a tool that will generate code automatically.

For professional API, GRAPHQL is used.

Or GraphQl is alternative to REST

We can call multiple request in a single queries.

It is independent of programming language. We need to define standard query code only

We don’t need to customize API for mobile separately.

We write same query language at client and server side

**The main backbone is Schema Definition Language (SDL).**

GraphQl works as an API from client and server view

In REST, client get data in JSON format, irrespective of data type.

But in GraphQl, we need to specify the type of the data from client and server

**GraphQL is a query language for APIs, not databases you can use any database which you want.**

Example 1:

type Person {

name: String!

age: Int!

}

**The above is from the client side**

! shows this property is compulsory or required

FETCH DATA (Client Side):

Basic Query:

{

allPersons {

name

}

}

The allPersons (I need all the persons, it is a kind of property) field in this query is called the root field of the query. Everything that follows the root field, is called the payload of the query. The only field that’s specified in this query’s payload is name.

This query would return a list of all persons currently stored in the database. Here’s an example response:

{

"allPersons": [

{ "name": "Johnny" },

{ "name": "Sarah" },

{ "name": "Alice" }

]

}

Example 2:

type Post {

title: String!

author: Person!

}

type Person {

name: String!

age: Int!

posts: [Post!]!

}

Note that we just created a one-to-many-relationship between Person and Post since the posts field on Person is actually an array of posts.

Fetching data for this case:

{

allPersons {

name

age

posts {

title

}

}

}

We need name age of all the persons and we need ‘title’ only from post type. Here we don’t need all of the things from post, we only need title. This is the beauty of GraphQL

Entry points:

The Query, Mutation, and Subscription types are the entry points for the requests sent by the client

To retrieve the data -> type Query { ... }

Add or update the data -> type Mutation { ... }

For live update or live data or live response -> type Subscription { ... }

The root of all queries(what client ask from server) must be define in the type Query

At server side:

type Query{

person:Person

post : Post

}

type Person{

name: String

age: Int

}

type Post{

title: String

}

Fetching (client side):

{

person{

name

},

Post{

title

},

}

OR (specifying query, both are same)

query {

person{

name

},

Post{

title

},

}

So client asking only for the name of the person and title of the Post

Arguments:

It is kind of function that works on id.

In GraphQL, each field can have zero or more arguments if that’s specified in the schema. For example, the allPersons field could have a last parameter to only return up to a specific number of persons. Here’s what a corresponding query would look like:

{

allPersons(last: 2) {

name

}

}

This will return name of the last 2 persons only

Mutation:

Server Side:

type Mutation{

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

}

Mutation is used to add update the data.

We created a root field createPerson that requires two arguments, name and age of the new person and it will return object Person

Client Side:

Mutation {

createPerson(name:”Alice”, age:26){

id

}

}

It returns id

Subscription:

It will create live connection to the server

Server side:

type Subscription {

newPerson: Person!

}

Client Side

subscription {

newPerson {

name

age

}

}

Whenever a new mutation(new person) is added, the server due to the live connection sent name and age of the new person immediately to the client

Putting it all together, this is the full schema for all the queries and mutation that you have seen in this chapter:

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!

}

We cannot fetch post directly as it is not mentioned in Query end point

BIG ARCHITECTURE:

Create graphQl server that will directly taik to the database.

GraphQL can taik to the database or api.

MAIN WORKING:

Apollo is the library used in GraphQL by which, we can develope client and server side

For server side:

Installation:

npm install apollo-server

npm install graphql

Link if needed -> [Build a schema - Apollo Basics - Apollo GraphQL Docs](https://www.apollographql.com/docs/tutorial/schema/)

first I create folder start, in that I create folder server and in that folder, I typed above two command s in cmd.

After that in server folder, I create App.js to link with server and then define types in schema.js

gql is a function, we use template string with it

This all is for server side

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

const typeDefs = gql`

  type User{

      name: String

      age: Int

  }

  type Course{

      name: String

      courseId: Int

  }

  type Query{

      user: User

      course: Course

  }

`;

module.exports = typeDefs;

We defined above types in schema.js and use query to fetch user and course data. Template string is user.

To run the server:

Update app.js

const { ApolloServer } = require('apollo-server');//fetch appollo server

const typeDefs = require('./schema.js');//it fetch all types from schema file

const server = new ApolloServer({ typeDefs });//and then we specify those types in appollo server

server.listen().then(() => {

    console.log(`

      Server is running!

      Listening on port 4000

      Explore at https://studio.apollographql.com/dev

    `);

  });

We run app using node.

node app.js

This server will run on localhost:4000

For client side:

After opening localhost:4000, it will provide the playground. To call data, we use this playground.

Code of Playground:

# Write your query or mutation here

{

user{

name

age

}

}

Resolvers run the logic on server to return the data

As we don’t have any database now, so we will create static data in app.js under user

const { ApolloServer } = require('apollo-server');//fetch appollo server

const typeDefs = require('./schema.js');//it fetch all types from schema file

const resolvers = {

    Query : {

       user : () => {

           //logic to call the database or another api

           return {

               name: "muneeb",

               age: 21

           }

       }

    }

}

const server = new ApolloServer({ typeDefs, resolvers });//and then we specify those types in appollo server

server.listen().then(() => {

    console.log(`

      Server is running!

      Listening on port 4000

      Explore at https://studio.apollographql.com/dev

    `);

  });

This time, it will return the data like this

{

"data": {

"user": {

"name": "muneeb",

"age": 21

}

}

}

Now our server is ready, after that, we need to create client part. For that we will create react app

For client side:

Create react app. In that install;

Installation:

npm install @apollo/client graphql

After that in app.js paste this code

import { ApolloClient, InMemoryCache } from '@apollo/client';

const client = new ApolloClient({

  uri: 'http://localhost:4000',

  cache: new InMemoryCache()

});

Under uri, we need to pass url of the Apollo server so that client can taik to server. In this case, our server link was 'http://localhost:4000', so we paste it here

It works like a context api, in that, we have a provider to which we pass the value, and that value can be accessed by all child components under provider tag, so we do same here

We pass above client to Apollo provider

function App() {

  return (

    <ApolloProvider client={client}>

    <div>

      Hello World from GraphQL

      <User/>

    </div>

    </ApolloProvider>

  );

Now requesting data from server:

Here child component user will have access of all information of the client due to provider. Under user.js, we need to specify these to things

1. Query to call the server
2. Hook UseQuery in which we specify query to be called and other logics to handle loading error or data part itself

Now we are calling the query to get user data only. We can also provide course here, incase of REST API, we need to call user and course in different calls, so it will be two calls to the server. But here if we pass the course also, it will only do single call to the server and get data of user and course in the same single call

1. import React from 'react';
2. import {gql, useQuery} from '@apollo/client';
3. const GET\_USER = gql`
4. query{
5. user {
6. name
7. age
8. }
9. }
10. `;
11. function User() {
12. const {loading, error, data} = useQuery(GET\_USER);
13. if(loading) return <div>Loading..</div>
14. if(error) return <div>Error</div>
15. return (
16. <div>
17. Hello World from User
18. <div>
19. data object = {JSON.stringify(data)}
20. <br/>
21. Name = {data.user.name}
22. <br />
23. age = {data.user.age}
24. </div>
25. </div>
26. );
27. }
28. export default User;

Remember, here we are working on the client side.

We create GET\_USER and use gql and template string, and in order to fetch the data from the server, we use query keyword or endpoint. As in server type Query, we passed two things. One is user and the other is course. So here we are fetching user only and in user we are fetching name and age property. Now to handle query loading, error and data part, we use useQuery hook and pass our query (GET\_USER) to it. Now we get data under word data. It will be in object format. As it can be verify using command

1. data = {JSON.stringify(data)}

Now to fetch name only, we use data.user.name and same for the age.

And this whole will return in App,js as we have called <User /> their.

CONCLUSION:

SERVER SIDE:

* First create server (start/server)
* Under server, create schema.js file and write all types and query their.
* Then create server.
* Then for each mutation or data client want, we create resolvers and pass all those data like user course etc which you want on server and which your client want and in that, we can connect to any database or api (in this case, we provide to static data as we don’t have database).
* Now here, server part ends

CLIENT SIDE:

* Create react file, connect/contact that to server using apollo client and pass server url to uri
* Then use Apollo provider so that all components or child under that will have access to all of the data coming from the server
* Then call query(in User.js) to fetch data from server using gql and template string and for fetching, use query and pass all properties client want, remember, that property must be mentioned in server resolvers. In this case, we are calling the query to get user data only. We can also provide course here, in case of REST API, we need to call user and course in different calls, so it will be two calls to the server. But here if we pass the course also, it will only do single call to the server and get data of user and course in the same single call(or fetch both the apis in the single call). So we can fetch multiple apis in the single call
* . Only needed information should be mentioned in the resolver(server)
* Now to handle query loading error data(and many other properties like network status etc.), use hook UseQuery and pass query to it. And it will return object and we can extract information from that object as per our need