**CRUD with graphql-server**

In this section, we’ll go over some GraphQL CRUD examples to help you understand how CRUD operations work in a React and GraphQL app.

**Setting up the server**

We are going to spin off a simple GraphQL server using [express-graphql](https://www.npmjs.com/package/express-graphql) and get it connected to a MySQL database. The source code and the MySQL files are in this [repository](https://github.com/AvanthikaMeenakshi/graphqlReactBoilerplate).

A GraphQL server is built on top of schema and resolvers. As a first step, we build a schema (defining types, queries, mutations, and subscriptions). This schema describes the whole app structure.

Secondly, for the stuff defined in the schema, we’re building respective resolvers to compute and dispatch data. A resolver maps actions with functions; for each query declared in typedef, we create a resolver to return data.

Finally, we complete server settings by defining an endpoint and passing configurations. We initialize /graphql as the endpoint for our app. To the graphqlHTTP middleware, we pass the built schema and root resolver.

Along with the schema and root resolver, we’re enabling the [GraphiQL](https://www.npmjs.com/package/graphiql) playground. GraphiQL is an interactive in-browser GraphQL IDE that helps us play around with the GraphQL queries we build.

var express = require('express');

var graphqlHTTP = require('express-graphql');

var { buildSchema } = require('graphql');

var schema = buildSchema(`

type Query {

hello: String

}

`);

var root = {

hello: () => "World"

};

var app = express();

app.use('/graphql', graphqlHTTP({

schema: schema,

rootValue: root,

graphiql: true,

}));

app.listen(4000);

console.log('Running a GraphQL API server at localhost:4000/graphql');

Once the server is good to go, running the app with node index.js will start the server on <http://localhost:4000/graphql>. We can query for hello and get the string “World” as a response.

**Connecting the database**

I’m going to establish the connection with the MySQL database as shown below:

var mysql = require('mysql');

app.use((req, res, next) => {

req.mysqlDb = mysql.createConnection({

host : 'localhost',

user : 'root',

password : '',

database : 'userapp'

});

req.mysqlDb.connect();

next();

});

We can connect multiple databases/sources and get them consolidated in the resolvers. I’m connecting to a single MySQL database here. The database dump I’ve used for this article is in the GitHub repository.

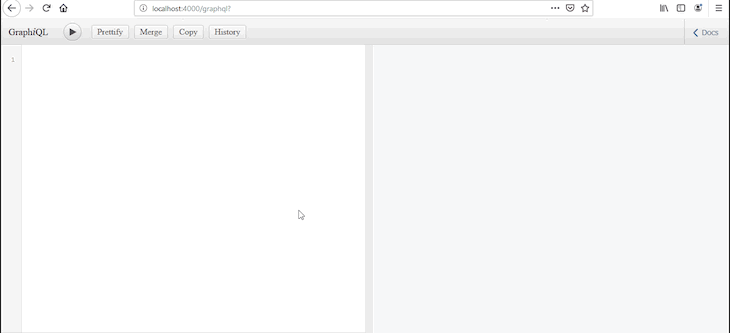
**Reading and writing data with GraphQL**

We use queries and mutations to read and modify data in data-sources. In this example, I’ve defined a generic queryDB function to help query the database.

**Queries**

All the SELECT statements (or read operations) to list and view data goes into the type Query typedef. We have two queries defined here: one to list all the users in the database, and another to view a single user by id.

1. **Listing data:**To list users, we’re defining a GraphQL schema object type called User, which represents what we can fetch or expect from the getUsers query. We then define the getUsers query to return an array of users
2. **Viewing a single record:**To view a single record, we’re taking id as an argument with the getUserInfo query we have defined. It queries for that particular id in the database and returns the data to the front end



Now that we have put together the queries to fetch all records and to view record by ID, when we try to query for users from GraphiQL, it will list an array of users on the screen!

var schema = buildSchema(`

type User {

id: String

name: String

job\_title: String

email: String

}

type Query {

getUsers: [User],

getUserInfo(id: Int) : User

}

`);

const queryDB = (req, sql, args) => new Promise((resolve, reject) => {

req.mysqlDb.query(sql, args, (err, rows) => {

if (err)

return reject(err);

rows.changedRows || rows.affectedRows || rows.insertId ? resolve(true) : resolve(rows);

});

});

var root = {

getUsers: (args, req) => queryDB(req, "select \* from users").then(data => data),

getUserInfo: (args, req) => queryDB(req, "select \* from users where id = ?", [args.id]).then(data => data[0])

};

**Mutations**

The write operations for the database — CREATE, UPDATE, DELETE — are generally defined under mutations. The mutations are executed in a sequential manner by the GraphQL engine. Queries are executed parallelly.

1. Creating data: We have defined a mutation, createUser, that takes the specified arguments to create data in the MySQL database.
2. Updating or deleting data: Similar to viewing a record, update (updateUserInfo) and delete (deleteUser) take id as a param and modify the database.

The functions resolve with a boolean to indicate whether the change happened or not.

var schema = buildSchema(`

type Mutation {

updateUserInfo(id: Int, name: String, email: String, job\_title: String): Boolean

createUser(name: String, email: String, job\_title: String): Boolean

deleteUser(id: Int): Boolean

}

`);

var root = {

updateUserInfo: (args, req) => queryDB(req, "update users SET ? where id = ?", [args, args.id]).then(data => data),

createUser: (args, req) => queryDB(req, "insert into users SET ?", args).then(data => data),

deleteUser: (args, req) => queryDB(req, "delete from users where id = ?", [args.id]).then(data => data)

};

Now that we have set and sorted the server side of things, let’s try and connect the back end to our React app.

**CRUD in React using graphql-client**

Once we have the server in place, creating client logic to display and mutate data is easy. Apollo Client helps in state management and caching. It is also highly abstracted and quick: all of the logic for retrieving your data, tracking loading and error states, and updating UI is encapsulated by the useQuery Hook.

**Connecting to graphql-server**

I have created a CRA boilerplate and have installed [GraphQL](https://www.npmjs.com/package/graphql), [apollo-boost](https://www.npmjs.com/package/apollo-boost), and [@apollo/react-hooks](https://www.npmjs.com/package/@apollo/react-hooks). We initialize Apollo Client and get it hooked to React.

import React from 'react';

import ReactDOM from 'react-dom';

import App from './App';

import ApolloClient from 'apollo-boost';

import { ApolloProvider } from '@apollo/react-hooks';

const client = new ApolloClient({

uri: 'http://localhost:4000/graphql'

});

ReactDOM.render(

<ApolloProvider client={client}>

<App />

</ApolloProvider>,

document.getElementById('root')

);

**Reading and mutating data**

I have managed all the GraphQL queries in the Queries folder of my source code. I’m going to request data from the server with the useQuery Hook, which is built on top of the React Hooks API. It helps in bringing in data into the UI.

GraphQL queries are generally wrapped in the gql function. gql helps convert query string into a query document. Here’s how we define queries in our app.

import { gql } from 'apollo-boost';

export const GET\_USERS = gql`

{

getUsers {

id,

name,

job\_title,

email

}

}

`;

export const VIEW\_USERS = gql`

query ($id: Int){

getUserInfo(id: $id) {

id,

name,

job\_title,

email

}

}

`;

export const ADD\_USER = gql`

mutation($name: String, $email: String, $job\_title: String) {

createUser (name: $name, email: $email, job\_title: $job\_title)

}

`;

export const EDIT\_USER = gql`

mutation($id: Int, $name: String, $email: String, $job\_title: String) {

updateUserInfo (id: $id, name: $name, email: $email, job\_title: $job\_title)

}

`;

export const DELETE\_USER = gql`

mutation($id: Int) {

deleteUser(id: $id)

}

`

Once ApolloProvider is set, we can request data from our GraphQL server. We pass the query we are trying to make to the useQuery Hook, and it will provide the result for us.

I’ve made two queries, with and without arguments, to show how we should be handling queries and mutations in the front end. useQuery tracks error and loading states for us and will be reflected in the associated object. Once the server sends the result, it will be reflected by the data property.

import React from 'react';

import { useQuery } from '@apollo/react-hooks';

import { GET\_USERS, VIEW\_USERS } from "./Queries";

import { Card, CardBody, CardHeader, CardSubtitle, Spinner } from 'reactstrap';

function App() {

const getAllUsers = useQuery(GET\_USERS);

const userInfo = useQuery(VIEW\_USERS, { variables: { id: 1 }});

if (getAllUsers.loading || userInfo.loading) return <Spinner color="dark" />;

if (getAllUsers.error || userInfo.error) return <React.Fragment>Error :(</React.Fragment>;

return (

<div className="container">

<Card>

<CardHeader>Query - Displaying all data</CardHeader>

<CardBody>

<pre>

{JSON.stringify(getAllUsers.data, null, 2)}

</pre>

</CardBody>

</Card>

<Card>

<CardHeader>Query - Displaying data with args</CardHeader>

<CardBody>

<CardSubtitle>Viewing a user by id</CardSubtitle>

<pre>

{JSON.stringify(userInfo.data, null, 2)}

</pre>

</CardBody>

</Card>

</div>

)

}

export default App;

Similar to querying, mutations will use the same useQuery Hook and will pass data as variables into the query.

const deleteMutation = useQuery(DELETE\_USER, { variables: { id: 8 }});

const editMutation = useQuery(EDIT\_USER, { variables: { id: 9, name: "Username", email: "email", job\_title: "job" }});

const createMutation = useQuery(ADD\_USER, { variables: { name: "Username", email: "email", job\_title: "job" }});