Exercise Q

Create a simple GraphQL server in Node.js using Mongo

Prior Knowledge
Unix Command Line Shell
Some simple JavaScript (node.js)

Learning Objectives
Understand GraphQL

Software Requirements
Node.js
Npm
Mongo
Visual Studio Code

Thanks to this guide which this is heavily based on:

https://freo.me/do-node-graphql

1. First let's install MongoDB

   ```
   sudo apt install mongodb -y
   ```

2. Check it works:

   ```
   mongo

   MongoDB shell version v3.6.3
   connecting to: mongodb://127.0.0.1:27017
   MongoDB server version: 3.6.3
   Server has startup warnings:
   2019-11-27T08:42:30.127+0000 I STORAGE  [initandlisten]
   2019-11-27T08:42:30.127+0000 I STORAGE  [initandlisten] **
   WARNING: Using the XFS filesystem is strongly recommended
   with the WiredTiger storage engine
   2019-11-27T08:42:30.127+0000 I STORAGE  [initandlisten] **
   See http://dochub.mongodb.org/core/prodnotes-filesystem
   2019-11-27T08:42:30.707+0000 I CONTROL  [initandlisten]
   2019-11-27T08:42:30.707+0000 I CONTROL  [initandlisten] **
   WARNING: Access control is not enabled for the database.
   2019-11-27T08:42:30.707+0000 I CONTROL  [initandlisten] **
   Read and write access to data and configuration is
   unrestricted.
   ```

```
2019-11-27T08:42:30.707+0000 I CONTROL  [initandlisten]
>
```

3. Type
   `exit`
   to leave the mongo client command prompt.

4. Clone my simple sample repository:

   ```
   cd ~
   git clone https://github.com/pzfreo/graphql-example.git
   ```

5. Import some data into Mongo:

   ```
   mongoimport -d test -c bios bios.json
   ```

   This is this data:
   https://docs.mongodb.com/manual/reference/bios-example-collection/

6. Have a look using the mongo client

   ```
   mongo

   > use test
   switched to db test

   > db.bios.find({})
   ```
   You should see something like:

   ```
   { "_id" : 4, "name" : { "first" : "Kristen", "last" : "Nygaard" },
   "birth" : ISODate("1926-08-27T04:00:00Z"), "death" : ISODate("2002-08-
   10T04:00:00Z"), "contribs" : [ "OOP", "Simula" ], "awards" : [ { "award"
   : "Rosing Prize", "year" : 1999, "by" : "Norwegian Data Association" }, {
   "award" : "Turing Award", "year" : 2001, "by" : "ACM" }, { "award" :
   "IEEE John von Neumann Medal", "year" : 2001, "by" : "IEEE" } ] }
   ```

7. Please note that we haven't set up any security for the database. This is not a good thing. Don't do this for real.

8. Install the required npm dependencies:

   ```
   npm install
   ```

9. Take a look at our app:

   ```
   code index.js
   ```

The first interesting thing is:

```
const context = () => MongoClient.connect('mongodb://localhost:27017/test',
{ useNewUrlParser: true })
  .then(client => client.db('test'));
```

This connects us to the mongo test database where we imported the bios collection.
Now we define the GraphQL schema.

```
const schema = buildSchema(`
  type Query {
    bios: [Bio]
    bio(id: Int): Bio
  }
  type Mutation {
    addBio(input: BioInput) : Bio
  }
  input BioInput {
    name: NameInput
    title: String
    birth: String
    death: String
  }
  input NameInput {
    first: String
    last: String
  }
  type Bio {
    name: Name,
    title: String,
    birth: String,
    death: String,
    awards: [Award]
  }
  type Name {
    first: String,
    last: String
  },
  type Award {
    award: String,
    year: Float,
    by: String
  }
`);
```

The next interesting part is:

```
const resolvers = {

  bios: (args, context) =>context().then(db => db.collection('bios').find().toArray()),

  bio: (args, context) =>context().then(db => db.collection('bios').findOne({ _id: args.id })),

  addBio: (args, context) => context().then(db => db.collection('bios').insertOne({ name:
args.input.name, title: args.input.title, death: args.input.death, birth:
args.input.birth})).then(response => response.ops[0])
};
```

This defines what queries do when called. For example, when you do a GraphQL query “bios” this will do a mongodb `db.collection(‘bios’).find().toArray()`.

The rest of the file is basically “boilerplate”, and would be almost the same in any other example.

One interesting thing to note is the enabling of GraphiQL:

```
graphiql: true
```

This is super cool and we’ll see it in a minute.

10. Start the server

```
$ node index.js
🚀 Server ready at http://localhost:4000/graphql
```

11. You may also see a warning. You can ignore this.

```
(node:16940) DeprecationWarning: current Server Discovery and Monitoring engine
is deprecated, and will be removed in a future version. To use the new Server
Discover and Monitoring engine, pass option { useUnifiedTopology: true } to the
MongoClient constructor.
```

12. In a new window try:

```
http localhost:4000/graphql query='{ bios { name { first }}}’
```

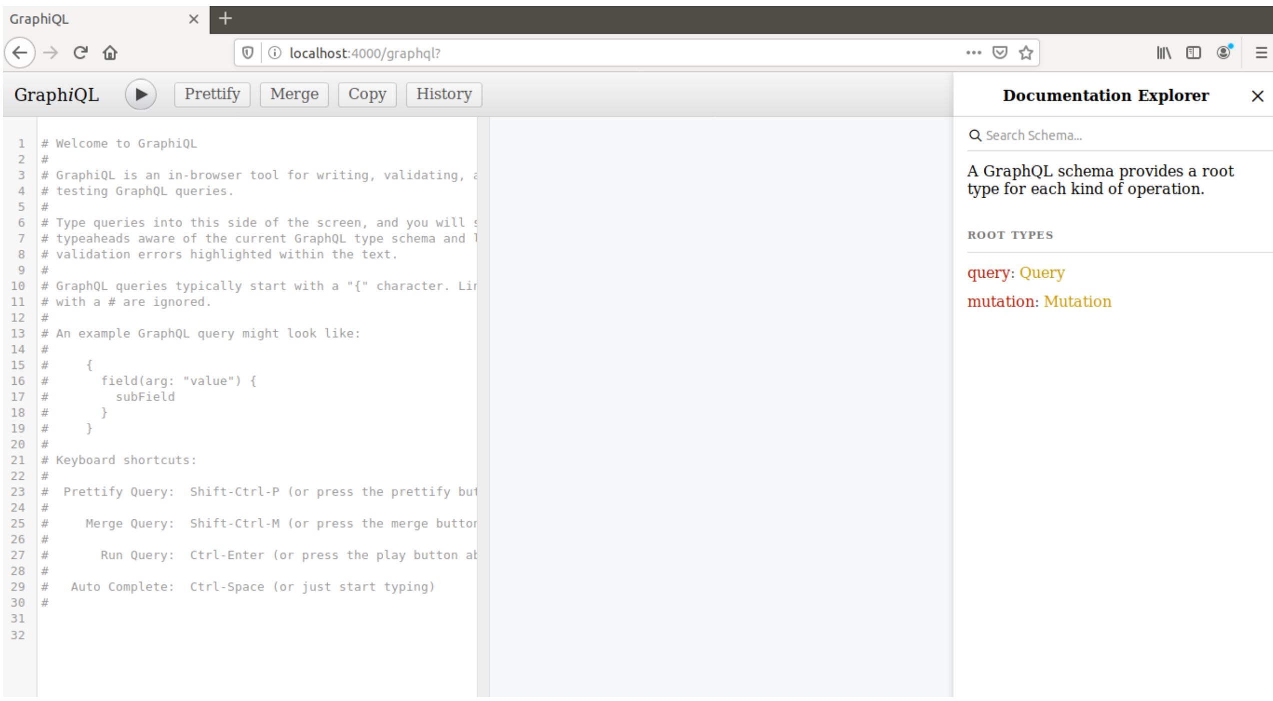
You should see something like:

```
HTTP/1.1 200 OK
Connection: keep-alive
Content-Length: 298
Content-Type: application/json; charset=utf-8
Date: Wed, 27 Nov 2019 08:56:09 GMT
ETag: W/"12a-aMvPeBKQdQnnT/UJvxWxZ4tD9Pc"
X-Powered-By: Express

{
    "data": {
        "bios": [
            {
                "name": {
                    "first": "Kristen"
                }
            },
            {
                "name": {
                    "first": "Ole-Johan"
                }
            },
```

13. Now browse to http://localhost:4000/graphql
    This is the GraphiQL interface (pronounced "graphical").

    You should see something like:



```
# Welcome to GraphiQL
#
# GraphiQL is an in-browser tool for writing, validating, a
# testing GraphQL queries.
#
# Type queries into this side of the screen, and you will s
# typeaheads aware of the current GraphQL type schema and l
# validation errors highlighted within the text.
#
# GraphQL queries typically start with a "{" character. Lin
# with a # are ignored.
#
# An example GraphQL query might look like:
#
#     {
#       field(arg: "value") {
#         subField
#       }
#     }
#
# Keyboard shortcuts:
#
#  Prettify Query:  Shift-Ctrl-P (or press the prettify but
#
#     Merge Query:  Shift-Ctrl-M (or press the merge buttor
#
#       Run Query:  Ctrl-Enter (or press the play button ab
#
#   Auto Complete:  Ctrl-Space (or just start typing)
#
```

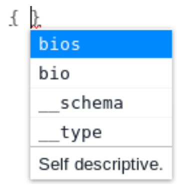
A GraphQL schema provides a root type for each kind of operation.

ROOT TYPES

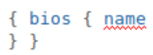
query: Query
mutation: Mutation

14. Have a read of the commented out help.

15. Below the comments start typing:
    `{ bi`
    You will see the auto-completion kick in:



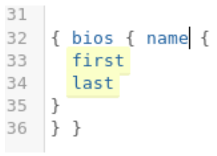
16. Add name to the query:



```
{ bios { name
} }
```

17. Hit the Play button or Ctrl-Enter

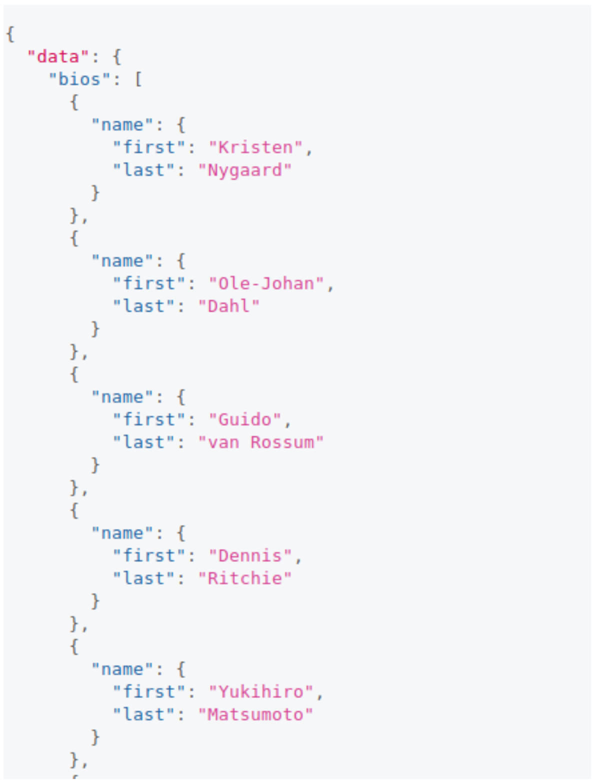
18. You will see GraphiQL will add first / last into the query to make it into a valid query:



```

{ bios { name {
  first
  last
}
} }
```

19. You should see the query response like this:



```
{
  "data": {
    "bios": [
      {
        "name": {
          "first": "Kristen",
          "last": "Nygaard"
        }
      },
      {
        "name": {
          "first": "Ole-Johan",
          "last": "Dahl"
        }
      },
      {
        "name": {
          "first": "Guido",
          "last": "van Rossum"
        }
      },
      {
        "name": {
          "first": "Dennis",
          "last": "Ritchie"
        }
      },
      {
        "name": {
          "first": "Yukihiro",
          "last": "Matsumoto"
        }
      },
```

20. If we look at the schema again, you should see this part:

```
type Query {
    bios: [Bio]
    bio(id: Int): Bio
}
```

And this is the corresponding code:

```
bios: (args, context) =>context().then(db =>
      db.collection('bios').find().toArray()),
bio: (args, context) =>context().then(db =>
      db.collection('bios').findOne({ _id: args.id })),
```

What this means, is that the “bios” query has no parameters and pulls back all the records from the collection (find()), while the “bio” query has a single parameter (id) and queries the collection to findOne with that id.

21. Try out the find one method:

```
{ bio(id:1) {
  name {
    first
    last
  }
}}
```

22. Updates in GraphQL are called mutations.

Here is the definition of the schema that lets us do an update:

```
type Mutation {
 addBio(input: BioInput) : Bio
}
input BioInput {
 name: NameInput
 title: String
 birth: String
 death: String
}
input NameInput {
 first: String
 last: String
}
```

And here is the code that is called when you do a mutation:

```
addBio: (args, context) =>
   context().then(db => db.collection('bios').insertOne(
      { name: args.input.name, title: args.input.title, death:
         args.input.death, birth: args.input.birth}
      )).then(response => response.ops[0])
```

23. Try adding some data into the database:

```
mutation {
        addBio(input: { name: { first: "John", last: "Smith" } })
        { name { first, last } }
}
```

24. Rerun the "bios" query and you will now see John Smith in the list

25. Re-run the update and new query from HTTPie (i.e. not using GraphiQL)

That's all!

Extension 1:

Add a query to search by first name and return all the records with that first name.

Extension 2 (hard):

Create an Order service that has a similar schema to our RESTful service but uses GraphQL instead.