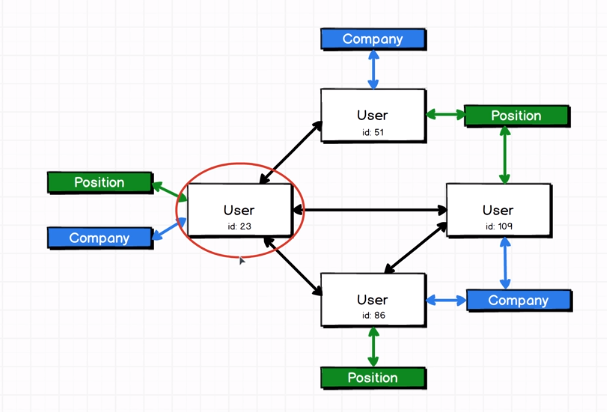
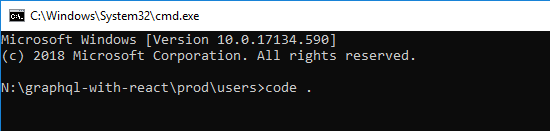
**LECTURE 5 – What is GraphQL**



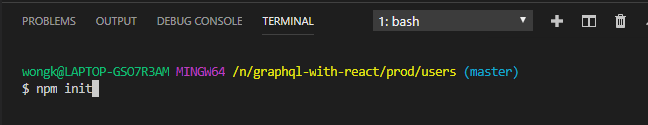
* This is the graph that graphql is referring to.

**LECTURE 6 – Working with GraphQL**

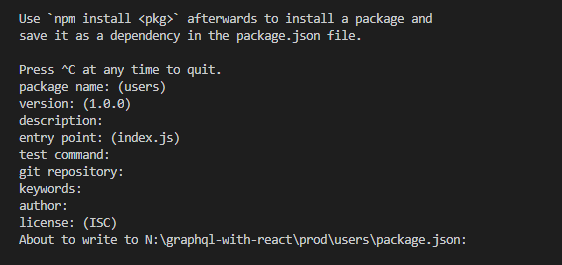
* Create a folder called **prod**.
* Inside that folder create another folder called **users**.
* Inside the users folder open up **vs code**.



* Open up a terminal and create a **package.json** file by using the command **npm init**.



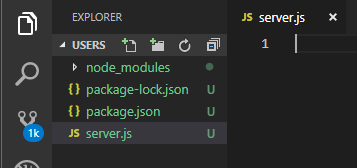
* Answer yes to all the questions by simply pressing **ENTER** until the package.json file has been created.



* Install the following NPM packages by typing **npm install –-save express express-graphql graphql lodash**.



* There are 4 packages being installed: **express**, **express-graphql**, **graphql** and **lodash**.
* At the root of our directory create a new file called **server.js**.



* In **server.js** write the following code to listen on port 4000 and console log the word ‘Listening’.

const express = require('express');

const app = express();

app.listen(4000, () => {

console.log('Listening');

});

**LECTURE 7 – Registering GraphQL with Express**

* In **server.js** import the GraphQL translation layer package.

const express = require('express');

const expressGraphQL = require('express-graphql');

const app = express();

* Add the following GraphQL middleware.

const express = require('express');

const expressGraphQL = require('express-graphql');

const app = express();

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

graphiql: true

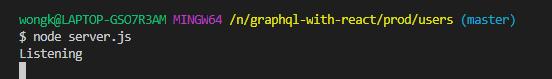
}));

app.listen(4000, () => {

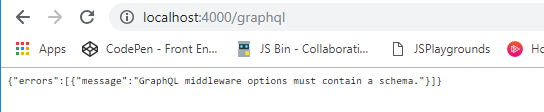
console.log('Listening');

});

* Run the server by typing in the terminal **node server.js**.

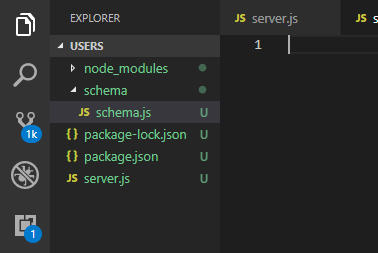


* In a web browser navigate to the url of our application, which is **localhost:4000/graphql**.



**LECTURE 8 – GraphQL Schemas**

* **app.use** is how we wire up middleware to an express application.
* Middlewares are tiny functions used to intercept or modify requests as they come through an express server.
* When we registered **expressGraphQL** it was registered as a middleware.
* A **Schema** tells GraphQL how all the data in our application is arranged and how it can be accessed. The Schema file is the absolute lynch pin of every GraphQL application.
* The Schema file tells GraphQL what type of data we are working with and how those different relations play out between the pieces of data.
* A Schema is what tells GraphQL what our data looks like.
* We will now create a Schema file.
* In the Root directory create a folder called **schema** and inside that folder create a file called **schema.js**.



**LECTURE 9 – Writing a GraphQL Schema**

* In the **schema.js** file, import the **graphql** library.

const graphql = require('graphql');

* We will now do some destructuring from graphql. Write the following code.

const graphql = require('graphql');

const {

GraphQLObjectType

} = graphql;

* Create the following object with the following properties.

const graphql = require('graphql');

const {

GraphQLObjectType

} = graphql;

const UserType = new GraphQLObjectType({

name: 'User',

fields: {

id: { type: GraphQLString },

firstName: { type: GraphQLString },

age: { type: GraphQLInt }

}

});

* Since we used the GraphQLString and GraphQLInt properties from graphql we need to import through destructuring. Make the following modifications.

const graphql = require('graphql');

const {

GraphQLObjectType,

GraphQLString,

GraphQLInt

} = graphql;

const UserType = new GraphQLObjectType({

name: 'User',

fields: {

id: { type: GraphQLString },

firstName: { type: GraphQLString },

age: { type: GraphQLInt }

}

});

**LECTURE 10 – Root Queries**

* GraphQL has difficulty finding a specific item within a set of data. To assist GraphQL we need to use a **Root Query**.
* A **Root Query** is something that allows us to jump into our graph of data.
* The Root Query is the entry point into our data.
* Write the following Root Query.

const graphql = require('graphql');

const {

GraphQLObjectType,

GraphQLString,

GraphQLInt

} = graphql;

const UserType = new GraphQLObjectType({

name: 'User',

fields: {

id: { type: GraphQLString },

firstName: { type: GraphQLString },

age: { type: GraphQLInt }

}

});

const RootQuery = new GraphQLObjectType({

name: 'RootQueryType',

fields: {

user: {

type: UserType,

args: { id: { type: GraphQLString } }

}

}

});

* The args parameter specifies the arguments that are necessary for the Root Query of this specific user.
* This Root Query is saying that if you’re looking for a User, and you give me an id, I’ll give you back a User.
* Within the Root Query we also need to add the resolve function.
* The **resolve** function is where we actually go into our database and we find the actual data that we are looking for. It executes that instructions that we have written before for the Root Query.
* Make the following amendments to the Root Query.

const UserType = new GraphQLObjectType({

name: 'User',

fields: {

id: { type: GraphQLString },

firstName: { type: GraphQLString },

age: { type: GraphQLInt }

}

});

const RootQuery = new GraphQLObjectType({

name: 'RootQueryType',

fields: {

user: {

type: UserType,

args: { id: { type: GraphQLString } },

resolve(parentValue, args) {

}

}

}

});

**LECTURE 11 – Resolving with Data**

* Rather than use a database for our data we are just going to create a hard-coded list of data.
* In **schema.js** create the following array of data.

const graphql = require('graphql');

const {

GraphQLObjectType,

GraphQLString,

GraphQLInt

} = graphql;

const users = [

{ id: '23', firstName: 'Bill', age: 20 },

{ id: '47', firstName: 'Samantha', age: 21 }

]

const UserType = new GraphQLObjectType({

name: 'User',

fields: {

id: { type: GraphQLString },

firstName: { type: GraphQLString },

age: { type: GraphQLInt }

}

});

* At the top import **lodash**.

const graphql = require('graphql');

const \_ = require('lodash');

const {

GraphQLObjectType,

GraphQLString,

GraphQLInt

} = graphql;

const users = [

{ id: '23', firstName: 'Bill', age: 20 },

{ id: '47', firstName: 'Samantha', age: 21 }

]

* Make the following changes to the Root Query.

const UserType = new GraphQLObjectType({

name: 'User',

fields: {

id: { type: GraphQLString },

firstName: { type: GraphQLString },

age: { type: GraphQLInt }

}

});

const RootQuery = new GraphQLObjectType({

name: 'RootQueryType',

fields: {

user: {

type: UserType,

args: { id: { type: GraphQLString } },

resolve(parentValue, args) {

return \_.find(users, { id: args.id });

}

}

}

});

* Import the following **helper** from the graphql library.

const graphql = require('graphql');

const \_ = require('lodash');

const {

GraphQLObjectType,

GraphQLString,

GraphQLInt,

GraphQLSchema

} = graphql;

const users = [

{ id: '23', firstName: 'Bill', age: 20 },

{ id: '47', firstName: 'Samantha', age: 21 }

]

* GraphQLSchema takes in a Root Query and returns a GraphQL instance.
* Create the following GraphQLSchema object.

const RootQuery = new GraphQLObjectType({

name: 'RootQueryType',

fields: {

user: {

type: UserType,

args: { id: { type: GraphQLString } },

resolve(parentValue, args) {

return \_.find(users, { id: args.id });

}

}

}

});

module.exports = new GraphQLSchema({

query: RootQuery

});

* Now we will need to import all this into our server.js file.
* Go back to **server.js** and make the following changes.

const express = require('express');

const expressGraphQL = require('express-graphql');

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

const app = express();

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

schema,

graphiql: true

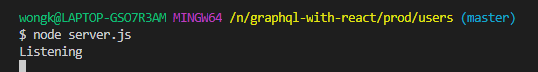
}));

app.listen(4000, () => {

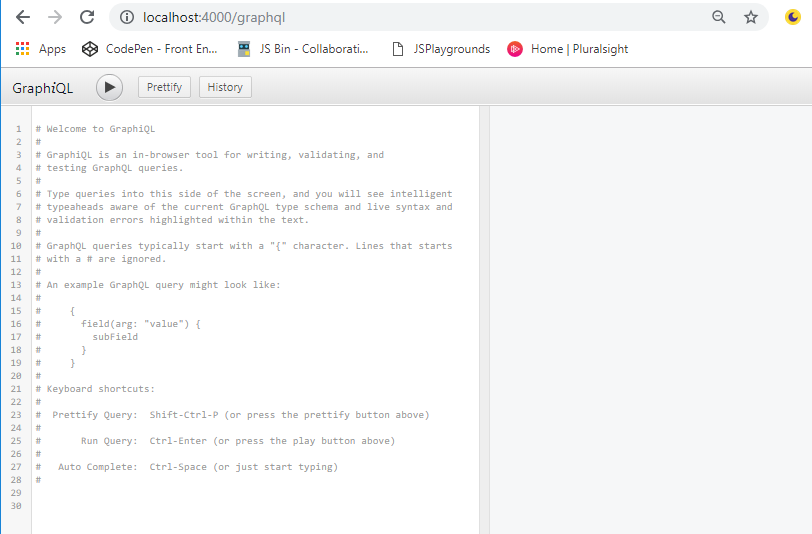
console.log('Listening');

});

* Now we will use this in the browser.
* First, restart the server by cancelling any existing server with CTRL + C and then typing **node server.js**.



* Now go to a web browser and navigate to **localhost:4000/graphql**
* You should get the following pop-up in the browser.



**LECTURE 12 – The GraphiQL tool**

* In the left-pane of the GraphiQL tool, type the following query.

{

user(id: "23") {

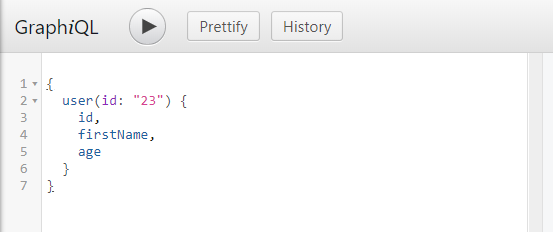
id,

firstName,

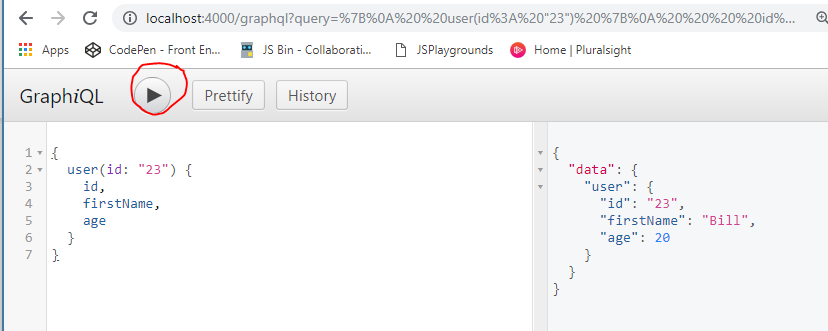
age

}

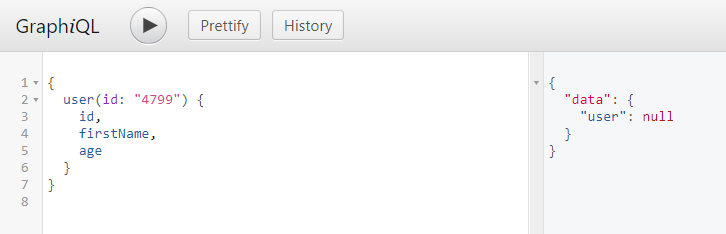
}



* Hit on the **Play** icon to execute the query, and the results should show up on the right-hand panel.



* If you provide an id that doesn’t exist then you will get a **null** response.



* If you don’t provide an id at all then you will get an error response.

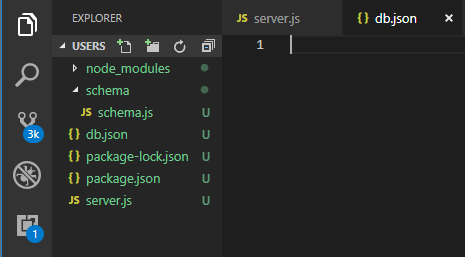


**LECTURE 13 – A Realistic Data Source**

* We will be using an npm package called **JSON Server** to act as our dynamic datastore.
* In the terminal, stop your server.
* In the terminal type **npm install –-save json-server**.



* At the root of the directory, create a new file called **db.json**.



* In the **db.json** file create the following data objects.

{

"users": [

{ "id": "23", "firstName": "Bill", "age": 20 },

{ "id": "40", "firstName": "Alex", "age": 40 }

]

}

* We are going to write a little helper code to start the **db.json**.
* Go to **package.json** and write the following code.

{

"name": "users",

"version": "1.0.0",

"description": "",

"main": "index.js",

"scripts": {

"test": "echo \"Error: no test specified\" && exit 1",

"json:server": "json-server --watch db.json"

},

"author": "",

"license": "ISC",

"dependencies": {

"express": "^4.16.4",

"express-graphql": "^0.7.1",

"graphql": "^14.1.1",

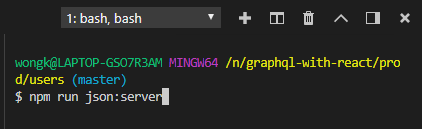
"json-server": "^0.14.2",

"lodash": "^4.17.11"

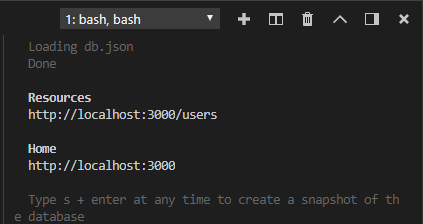
}

}

* We now need to run both servers so you can split the terminal or open up a new one.
* In the new terminal window, run the db.json server by typing **npm run json:server**.



* The server is telling us that the data is being served up at [**http://localhost:3000/users**](http://localhost:3000/users).

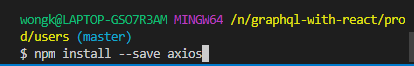


* Navigate to that address in the web browser.



**LECTURE 14 – Async Resolve Functions**

* We will need to install **axios** from npm.
* In the terminal type **npm install –-save axios**.



* Go to **shema.js** and import **axios**. Also, delete **lodash** as we no longer need it.

const graphql = require('graphql');

const axios = require('axios');

const \_ = require('lodash');

const {

GraphQLObjectType,

GraphQLString,

GraphQLInt,

GraphQLSchema

} = graphql;

const users = [

{ id: '23', firstName: 'Bill', age: 20 },

{ id: '47', firstName: 'Samantha', age: 21 }

]

* Delete the static list of users.

const graphql = require('graphql');

const axios = require('axios');

const {

GraphQLObjectType,

GraphQLString,

GraphQLInt,

GraphQLSchema

} = graphql;

const users = [

{ id: '23', firstName: 'Bill', age: 20 },

{ id: '47', firstName: 'Samantha', age: 21 }

]

const UserType = new GraphQLObjectType({

name: 'User',

fields: {

id: { type: GraphQLString },

firstName: { type: GraphQLString },

age: { type: GraphQLInt }

}

});

* Make the following changes to the Root Query.

const UserType = new GraphQLObjectType({

name: 'User',

fields: {

id: { type: GraphQLString },

firstName: { type: GraphQLString },

age: { type: GraphQLInt }

}

});

const RootQuery = new GraphQLObjectType({

name: 'RootQueryType',

fields: {

user: {

type: UserType,

args: { id: { type: GraphQLString } },

resolve(parentValue, args) {

return axios.get(`http://localhost:3000/users/${args.id}`)

.then(resp => resp.data);

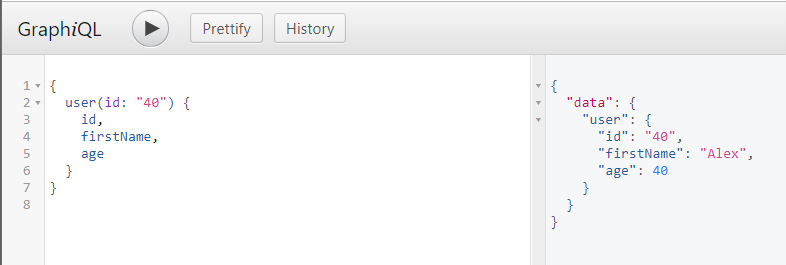
}

}

}

});

* Now we will test this.
* Go back to the terminal and restart the server by typing **node server.js**.
* Go back to GraphiQL and send a query.



**LECTURE 15 – Nodemon Hookup**

* Install **nodemon** by going to the terminal and typing **npm install –-save nodemon**.



* Now we will make changes to the **package.json** file to add another script for the dev environment.
* Go to **package.json** and make the following changes.

{

"name": "users",

"version": "1.0.0",

"description": "",

"main": "index.js",

"scripts": {

"test": "echo \"Error: no test specified\" && exit 1",

"json:server": "json-server --watch db.json",

"dev": "nodemon server.js"

},

"author": "",

"license": "ISC",

"dependencies": {

"axios": "^0.18.0",

"express": "^4.16.4",

"express-graphql": "^0.7.1",

"graphql": "^14.1.1",

"json-server": "^0.14.2",

"lodash": "^4.17.11",

"nodemon": "^1.18.10"

}

}

* Go back to the terminal and restart the server by typing **npm run dev**.

**LECTURE 16 – Company Definitions**

* We are now going to hookup relating a company to a user.
* Go to **db.json** and add the following data.

{

"users": [

{ "id": "23", "firstName": "Bill", "age": 20 },

{ "id": "40", "firstName": "Alex", "age": 40 }

],

"companies": [

{ "id": "1", "name": "Apple", "description": "iphone" },

{ "id": "2", "name": "Google", "description": "search" }

]

}

* Add an extra user.

{

"users": [

{ "id": "23", "firstName": "Bill", "age": 20 },

{ "id": "40", "firstName": "Alex", "age": 40 },

{ "id": "41", "firstName": "Nick", "age": 40 }

],

"companies": [

{ "id": "1", "name": "Apple", "description": "iphone" },

{ "id": "2", "name": "Google", "description": "search" }

]

}

* Add in a **companyId** property for each user.

{

"users": [

{ "id": "23", "firstName": "Bill", "age": 20, "companyId": "1" },

{ "id": "40", "firstName": "Alex", "age": 40, "companyId": "2" },

{ "id": "41", "firstName": "Nick", "age": 40, "companyId": "2" }

],

"companies": [

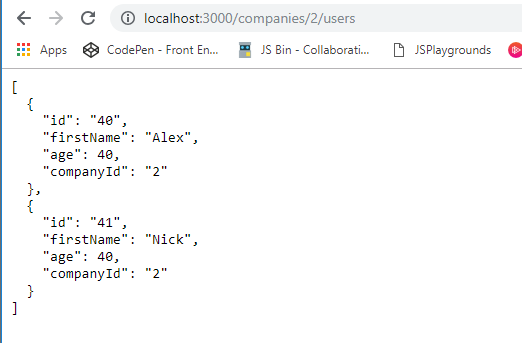
{ "id": "1", "name": "Apple", "description": "iphone" },

{ "id": "2", "name": "Google", "description": "search" }

]

}

* Now if you go to a web browser and type in the url: [**http://localhost:3000/users**](http://localhost:3000/users) you should get the list of users, and if you type [**http://localhost:3000/companies**](http://localhost:3000/companies) you should get the list of companies.
* Now if you want to get the list of users that work for Google (companyId of 2), in the web browser just type in the Url [**http://localhost:3000/companies/2/users**](http://localhost:3000/companies/2/users).



**LECTURE 17 – Nested Queries**

* We are now going to add the idea of a **CompanyType** into our Schema.
* Go to **schema.js** and add the following code, making sure that that CompanyType is **ABOVE** the UserType.

const graphql = require('graphql');

const axios = require('axios');

const {

GraphQLObjectType,

GraphQLString,

GraphQLInt,

GraphQLSchema

} = graphql;

const CompanyType = new GraphQLObjectType({

name: "Company",

fields: {

id: { type: GraphQLString },

name: { type: GraphQLString },

description: { type: GraphQLString }

}

});

const UserType = new GraphQLObjectType({

name: 'User',

fields: {

id: { type: GraphQLString },

firstName: { type: GraphQLString },

age: { type: GraphQLInt }

}

});

* Add the following CompanyType field to the UserType object.

const CompanyType = new GraphQLObjectType({

name: "Company",

fields: {

id: { type: GraphQLString },

name: { type: GraphQLString },

description: { type: GraphQLString }

}

});

const UserType = new GraphQLObjectType({

name: 'User',

fields: {

id: { type: GraphQLString },

firstName: { type: GraphQLString },

age: { type: GraphQLInt },

company: {

type: CompanyType

}

}

});

**LECTURE 18 – More on Nested Queries**

* You will notice in the UserType object, instead of creating a field called **companyId** we called it just **company**.
* Make the following changes to the UserType object.
* Go to **schema.js** and make the following changes.

const CompanyType = new GraphQLObjectType({

name: "Company",

fields: {

id: { type: GraphQLString },

name: { type: GraphQLString },

description: { type: GraphQLString }

}

});

const UserType = new GraphQLObjectType({

name: 'User',

fields: {

id: { type: GraphQLString },

firstName: { type: GraphQLString },

age: { type: GraphQLInt },

company: {

type: CompanyType,

resolve(parentValue, args) {

return axios.get(`http://localhost:3000/companies/${parentValue.companyId}`)

.then(res => res.data)

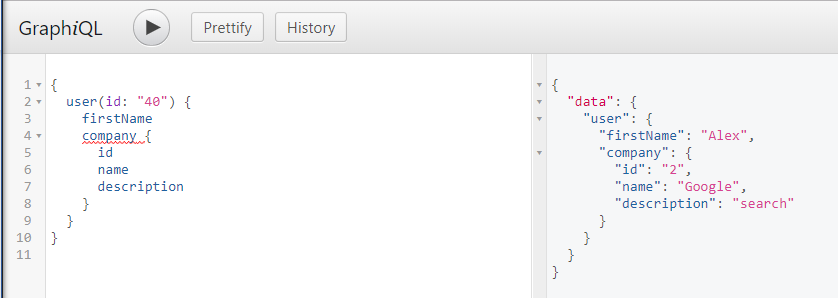
}

}

}

});

* Now go back to GraphiQL and write the following query to get the Company Id, name and description of the company that User of Id 40 works for.



**LECTURE 19 – A Quick Breather**

**LECTURE 20 – Multiple RootQuery Entry Points**

* First off, go to **server.js** and make the following changes.

const express = require('express');

const expressGraphQL = require('express-graphql');

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

const app = express();

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

schema,

graphiql: true

}));

const port = 4000;

app.listen(port, () => {

console.log(`Listening on port ${port}`);

});

* Write the following query that will ask for a specific user, ask for their company and then get the name of that company.
* In GraphiQL write the following query.

{

user(id: "23") {

firstName

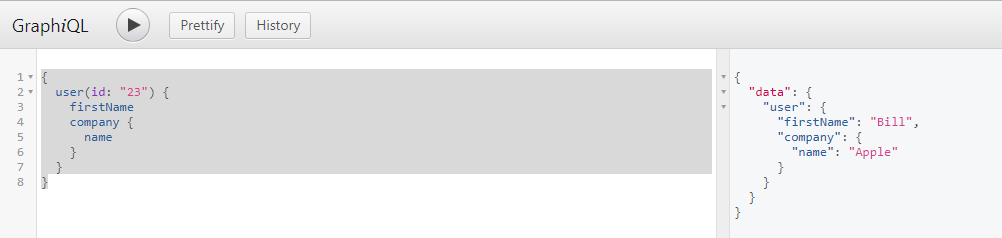
company {

name

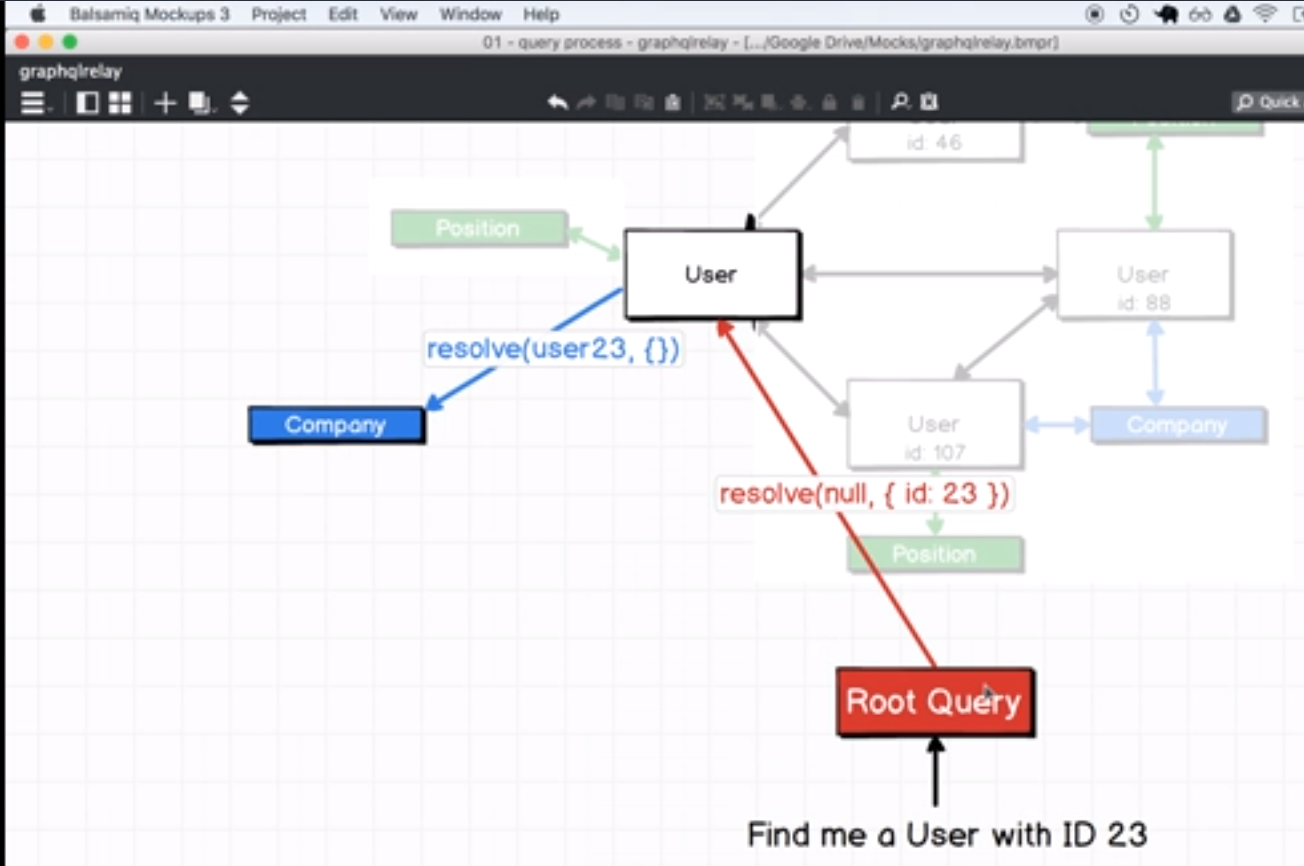
}

}

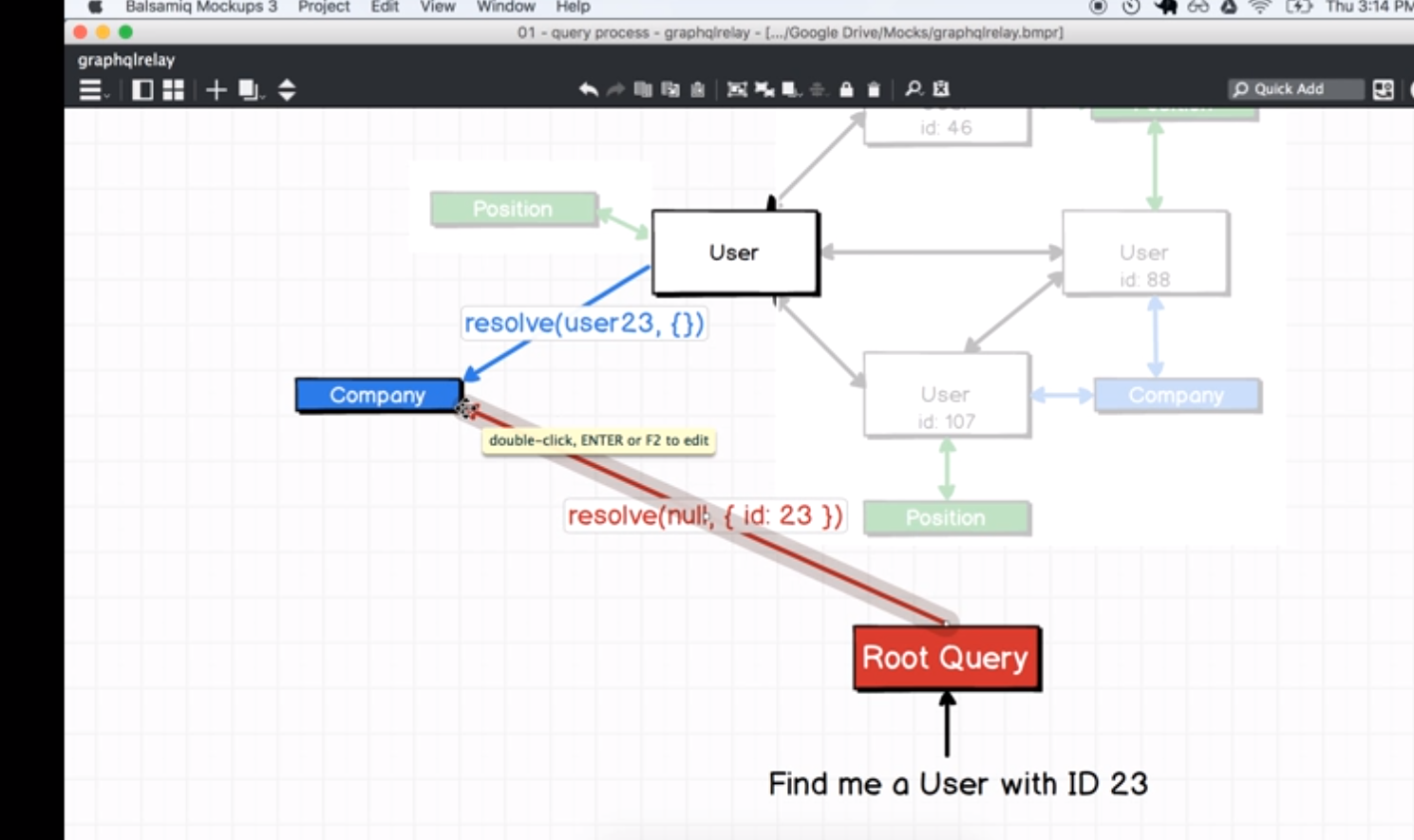
}



* Currently we don’t have the ability to go directly to a company. This is because in our Root Query we only have the field that connects directly to the User.



* In this section we will work on getting our Root Query to connect directly to our company.



* Go to **schema.js** and add the following field.

const RootQuery = new GraphQLObjectType({

name: 'RootQueryType',

fields: {

user: {

type: UserType,

args: { id: { type: GraphQLString } },

resolve(parentValue, args) {

return axios.get(`http://localhost:3000/users/${args.id}`)

.then(resp => resp.data);

}

},

company: {

type: CompanyType,

args: { id: { type: GraphQLString} },

resolve(parentValue, args) {

return axios.get(`http://localhost:3000/companies/${args.id}`)

.then(resp => resp.data);

}

}

}

});

module.exports = new GraphQLSchema({

query: RootQuery

});

* In GraphiQL write the following queries.

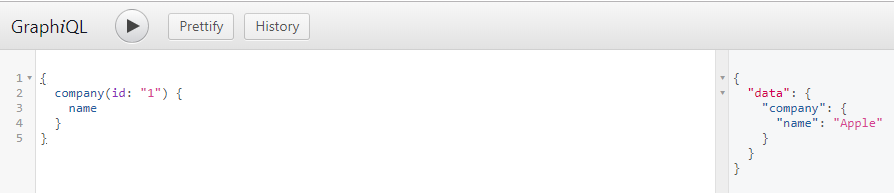
{

company(id: "1") {

name

}

}



* Write this other query.

{

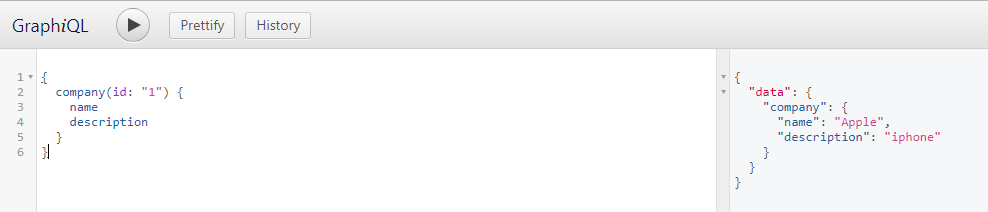
company(id: "1") {

name

description

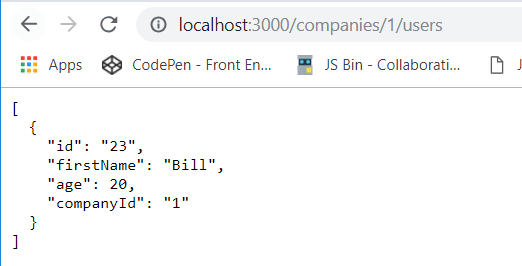
}

}



**LECTURE 21 – Bidirectional Relations**

* We are now going to make amendments so that we can get all the users for a particular company.
* First we will try to find out who all the users are for company 1. This can be done by querying the db server rather than the server.js.
* In a web browser navigate to [**http://localhost:3000/companies/1/users**](http://localhost:3000/companies/1/users)



**LECTURE 22 – More on Birectional Relations**

* Go to **schema.js** and within the **CompanyType**, add a new property to the fields as follows.

const CompanyType = new GraphQLObjectType({

name: "Company",

fields: {

id: { type: GraphQLString },

name: { type: GraphQLString },

description: { type: GraphQLString },

users: {

type: new GraphQLList(UserType),

resolve(parentValue, args) {

return axios.get(`http://localhost:3000/companies/${parentValue.id}/users`)

.then(res => res.data)

}

}

}

});

* Destructure the GraphQLList property from graphql.

const graphql = require('graphql');

const axios = require('axios');

const {

GraphQLObjectType,

GraphQLString,

GraphQLInt,

GraphQLSchema,

GraphQLList

} = graphql;

**LECTURE 23 – Resolving Circular References**

* Since we are using the UserType before it is defined, we will inevitably get an error.
* In order to fix this we will need to wrap the fields of the CompanyType inside an arrow function.

const CompanyType = new GraphQLObjectType({

name: "Company",

fields: () => ({

id: { type: GraphQLString },

name: { type: GraphQLString },

description: { type: GraphQLString },

users: {

type: new GraphQLList(UserType),

resolve(parentValue, args) {

return axios.get(`http://localhost:3000/companies/${parentValue.id}/users`)

.then(res => res.data)

}

}

})

});

* This means that the fields property will not get used until everything else on the page gets defined first.
* Now we will do the same thing for the **UserType** fields property.

const UserType = new GraphQLObjectType({

name: 'User',

fields: () => ({

id: { type: GraphQLString },

firstName: { type: GraphQLString },

age: { type: GraphQLInt },

company: {

type: CompanyType,

resolve(parentValue, args) {

return axios.get(`http://localhost:3000/companies/${parentValue.companyId}`)

.then(res => res.data)

}

}

})

});

* In GraphiQL, do the following query.

{

company(id: "2") {

id

name

description

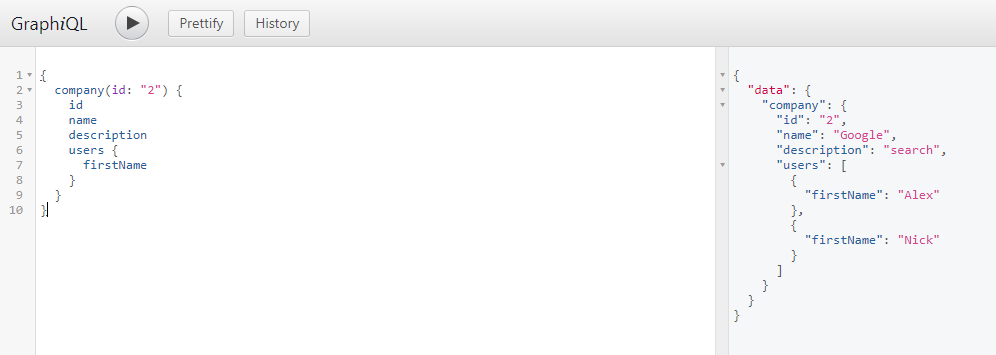
users {

firstName

}

}

}



* You can also perform strange queries with nested data as follows.

{

company(id: "2") {

id

name

description

users {

id

firstName

age

company {

name

users {

firstName

company {

name

}

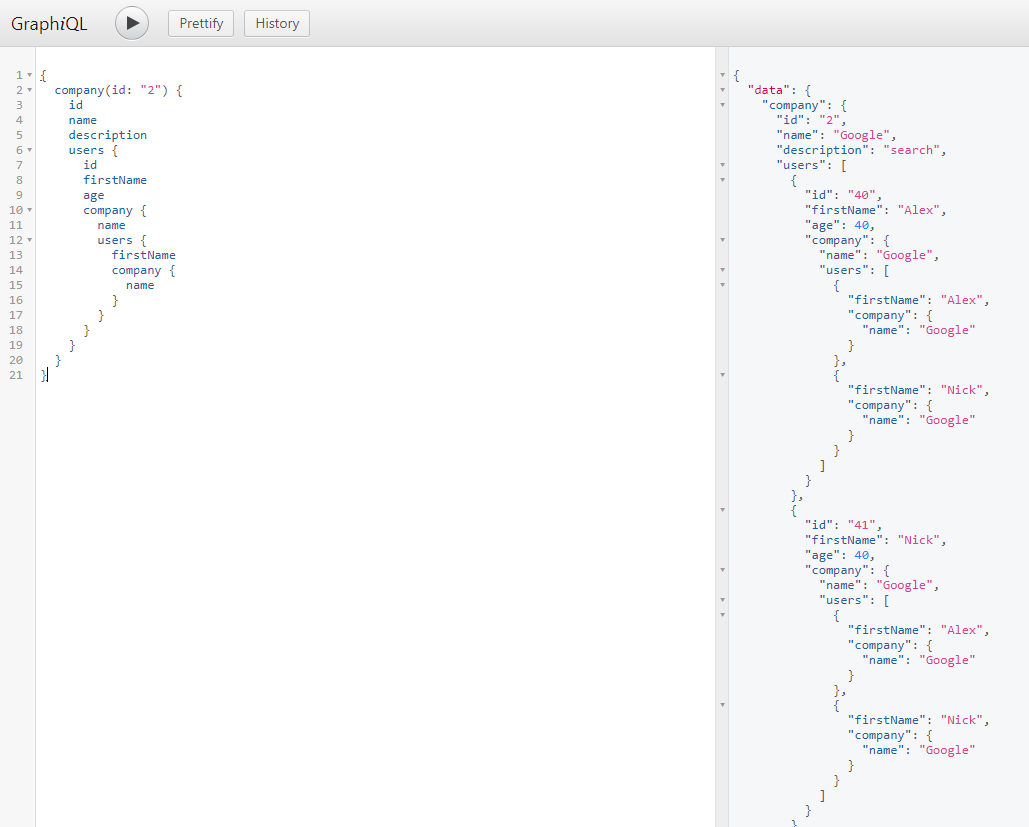
}

}

}

}

}



**LECTURE 24 – Query Fragments**

* Sometimes the queries are written as follows.

query{

company(id:"1") {

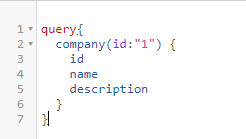
id

name

description

}

}



* You can actually add a name to the query so that it becomes easier to re-use the query.
* This is particularly useful in the front-end.
* There is an example of this as follows.

query findCompany{

company(id:"1") {

id

name

description

}

}

* You can ask for as many companies you like within a query.
* However you need to assign a key to each query if the queries are for the same field.
* An example is below.

{

apple:company(id:"1") {

id

name

description

}

google:company(id:"2") {

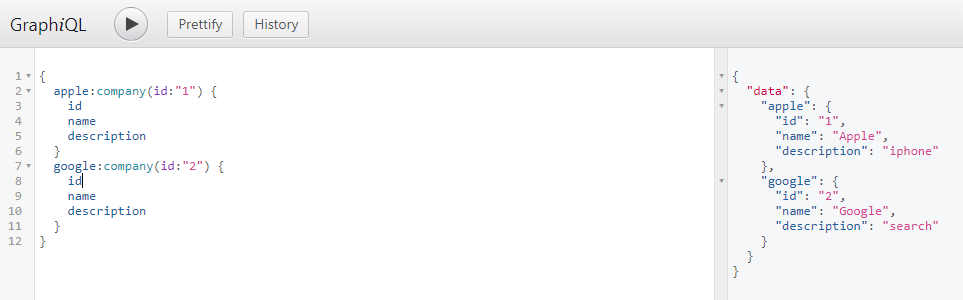
id

name

description

}

}



* We will now look at **Query Fragments**.
* A Query Fragment is a list of properties that we want to get access to for the sake of reusability.
* An example of Query Fragments is below.

{

apple:company(id:"1") {

...companyDetails

}

google:company(id:"2") {

...companyDetails

}

}

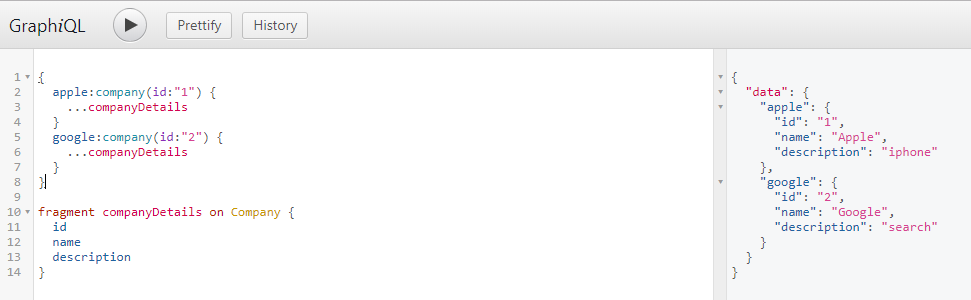
fragment companyDetails on Company {

id

name

description

}



**LECTURE 25 – Introduction to Mutations**

* Now we are going to look at how to modify our data from the data store in some fashion by using a system called **mutations**.
* Mutations are used to change our data in some fashion.
* We are now going to start by defining our **Root Mutation**.
* The fields in a mutation describe the operation that mutation is going to undertake.
* In a Mutation, the type refers to the type of data that will eventually be returned from the resolve function.
* In **schema.js** write the following mutation object.

const mutation = new GraphQLObjectType({

name: 'Mutation',

fields: {

addUser: {

type: UserType,

args: {

firstName: { type: GraphQLString },

age: { type: GraphQLInt },

companyId: { type: GraphQLString }

},

resolve() {

}

}

}

})

module.exports = new GraphQLSchema({

query: RootQuery

});

**LECTURE 26 – NonNull Fields and Mutations**

* We are now going to create validations for user input on some of the field arguments.
* Make the following changes to the mutation object. These changes wrap 2 of the arguments in help or validation wrapper.

const mutation = new GraphQLObjectType({

name: 'Mutation',

fields: {

addUser: {

type: UserType,

args: {

firstName: { type: new GraphQLNonNull(GraphQLString) },

age: { type: new GraphQLNonNull(GraphQLInt) },

companyId: { type: GraphQLString }

},

resolve() {

}

}

}

})

module.exports = new GraphQLSchema({

query: RootQuery

});

* Add **GraphQLNonNull** to the destructuring from the graphql library.

const graphql = require('graphql');

const axios = require('axios');

const {

GraphQLObjectType,

GraphQLString,

GraphQLInt,

GraphQLSchema,

GraphQLList,

GraphQLNonNull

} = graphql;

* In **schema.js**, make the following changes to the mutation object on the resolve function.

const mutation = new GraphQLObjectType({

name: 'Mutation',

fields: {

addUser: {

type: UserType,

args: {

firstName: { type: new GraphQLNonNull(GraphQLString) },

age: { type: new GraphQLNonNull(GraphQLInt) },

companyId: { type: GraphQLString }

},

resolve(parentValue, { firstName, age }) {

return axios.post(`http://localhost:3000/users/`, { firstName, age })

.then(res => res.data);

}

}

}

})

module.exports = new GraphQLSchema({

query: RootQuery

});

* We now need to add the mutation to the GraphQLSchema export as follows.

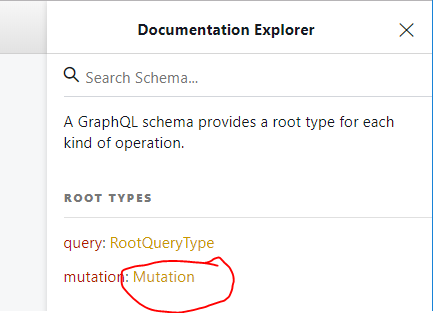
module.exports = new GraphQLSchema({

query: RootQuery,

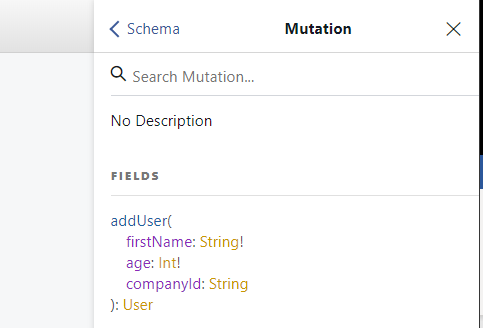
mutation: mutation

});

* Now go back to GraphiQL and you’ll notice in the docs that there is now a mutation field. Click on the mutation.



* You’ll notice that **firstName** and **age** have an exclamation mark next to them. This indicates that this is a required field.



* We will now call them mutation.
* In GraphiQL, write the following mutation query so that the firstName is Stephen and the age is 26. You also need to add the fields that you will get back from the query. These fields should be resolved when you send the query and the resolved data gets sent back to you.

mutation {

addUser(firstName:"Stephen", age:26) {

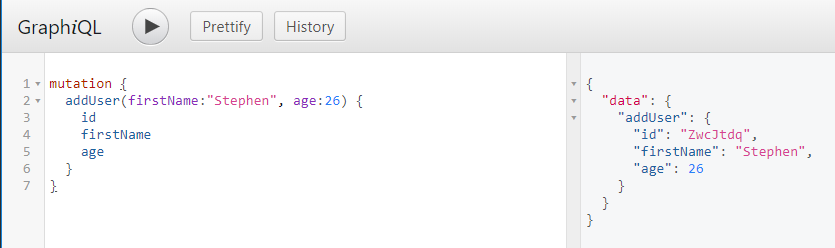
id

firstName

age

}

}



**LECTURE 27 – Do It Yourself. Delete Mutation!**

* We will now create another mutation field that will delete a user.
* In the mutation, add the following **deleteUser** mutation.

const mutation = new GraphQLObjectType({

name: 'Mutation',

fields: {

addUser: {

type: UserType,

args: {

firstName: { type: new GraphQLNonNull(GraphQLString) },

age: { type: new GraphQLNonNull(GraphQLInt) },

companyId: { type: GraphQLString }

},

resolve(parentValue, { firstName, age }) {

return axios.post(`http://localhost:3000/users/`, { firstName, age })

.then(res => res.data);

}

},

deleteUser: {

type: UserType,

args: {

id: { type: new GraphQLNonNull(GraphQLString) }

},

resolve(parentValue, { id }) {

return axios.delete(`http://localhost:3000/users/${id}`)

.then(res => res.data);

}

}

}

})

module.exports = new GraphQLSchema({

query: RootQuery,

mutation: mutation

});

* In GraphiQL send the following query.

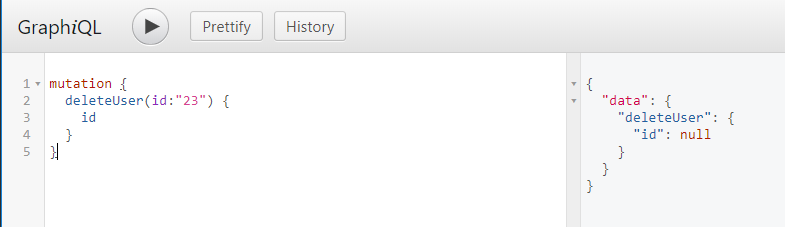
mutation {

deleteUser(id:"23") {

id

}

}



* Now when you go to the **db.json** file you will notice that the user of id 23 would have been deleted while a new user has been added.
* The response back from a delete request is a null object. This is because **Json server** does not return any data for a delete request and graphql **expects** some data to be returned. This is one of the quirks of GraphQL.

**LECTURE 28 – Do it Yourself. Edit Mutation!**

* A **PUT Request** is used when we want to completely replace an existing record saved on some other remote database, with the details inside of our Request Body.
* A **PATCH Request** only overwrites the properties that are contained within the Request Body.
* First we are going to make changes for a **Patch Request**.
* Make the following amendments to the mutation object so that a new function called **editUser** is inserted.

const mutation = new GraphQLObjectType({

name: 'Mutation',

fields: {

addUser: {

type: UserType,

args: {

firstName: { type: new GraphQLNonNull(GraphQLString) },

age: { type: new GraphQLNonNull(GraphQLInt) },

companyId: { type: GraphQLString }

},

resolve(parentValue, { firstName, age }) {

return axios.post(`http://localhost:3000/users/`, { firstName, age })

.then(res => res.data);

}

},

deleteUser: {

type: UserType,

args: {

id: { type: new GraphQLNonNull(GraphQLString) }

},

resolve(parentValue, { id }) {

return axios.delete(`http://localhost:3000/users/${id}`)

.then(res => res.data);

}

},

editUser: {

type: UserType,

args: {

id: { type: new GraphQLNonNull(GraphQLString) },

firstName: { type: GraphQLString },

age: { type: GraphQLInt },

companyId: { type: GraphQLString }

},

resolve(parentValue, args) {

return axios.patch(`http://localhost:3000/users/${args.id}`, args)

.then(res => res.data);

}

}

}

})

module.exports = new GraphQLSchema({

query: RootQuery,

mutation: mutation

});

* We will now test this.
* Go back to GraphiQL and write the following mutation.

mutation {

editUser(id:"40", age:10) {

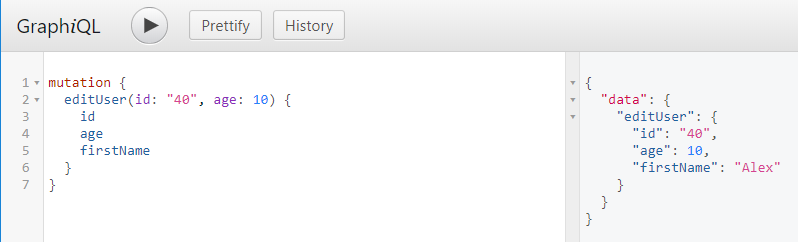
id

firstName

age

}

}



* Now when you go to **db.json** you should see the updated values for the User Alex.



* Now change Alex’s name to **Samantha2**.

mutation {

editUser(id: "40", firstName: "Samantha2") {

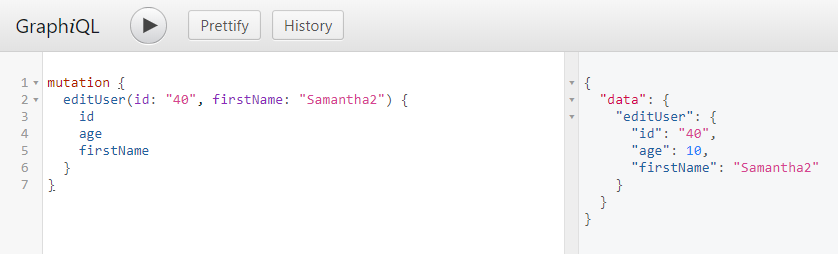
id

age

firstName

}

}



* Now change Samantha2’s age to 50.

mutation {

editUser(id: "40", firstName: "Samantha2", age:50) {

id

age

firstName

}

}



**LECTURE 29 – GraphQL Clients – Apollo vs Relay**

* We will now set up enzyme so that a React App can grab the data from GraphQL and have it show up on the screen.
* The 3 big **GraphQL Clients** (Clients that act as a translation layer between React and Express/GraphQL and are tightly coupled in the React side), are:
* **Lokka**
* **Apollo Client**
* **Relay**
* The purpose of the GraphQL Clients is to make a request to our GraphQL Server, get the response back and forward it on to our React Application.
* We will be going forward with the **Apollo Client**.

**LECTURE 30 – Sidenote – Apollo Server vs GraphQL Server**

**LECTURE 31 – The Next App**

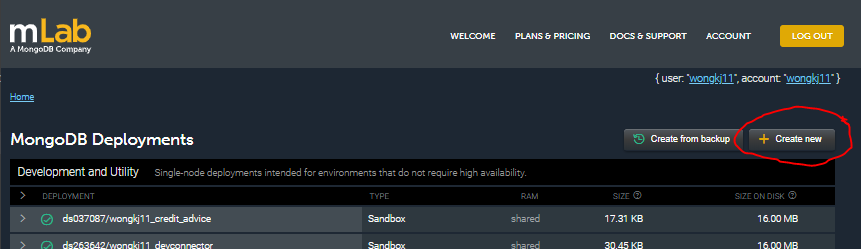
* Now in this section we are going to primarily focus on the front-end.
* this means that we won’t bother with the back-end and will just clone a github repo that the lecturer prepared earlier.
* Go to the following github repo and clone it locally: **https://github.com/stephengrider/lyrical-graphql**
* Open VS Code with the **Lyrical-GraphQL** folder as the root.
* In the VS Code terminal, use **npm install** to download all the dependencies from the package.json file.



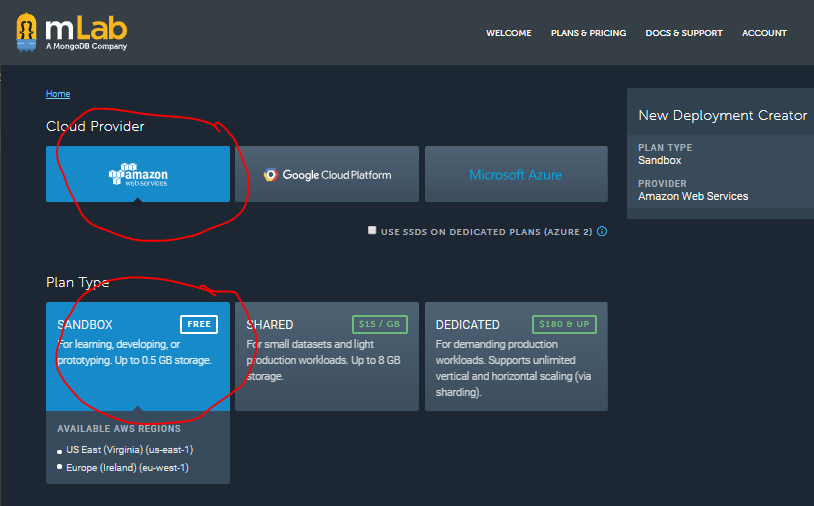
**LECTURE 32 – Starter Pack Walkthrough**

**LECTURE 33 – MongoLab Setup**

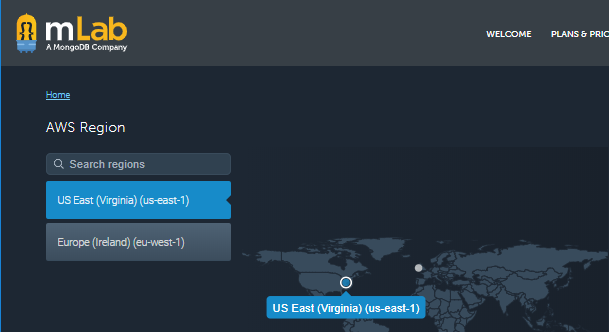
* We need to set up an account with **mLab**.
* I already have an account. To find the instructions on how to set up account go to the Instruction Documents **MERN Stack Front to Back**.
* Log in to your mLab Account.
* We will need to create a new Database.
* On the top right of the page, click on **Create new**.



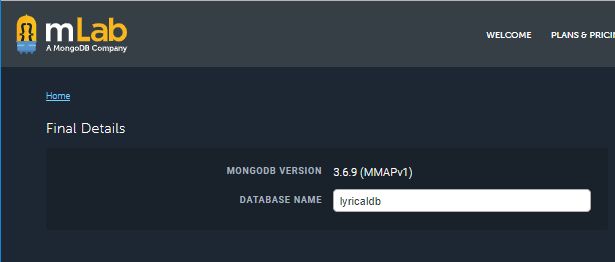
* Select **Amazon** and **Sandbox** then click on **Continue**.



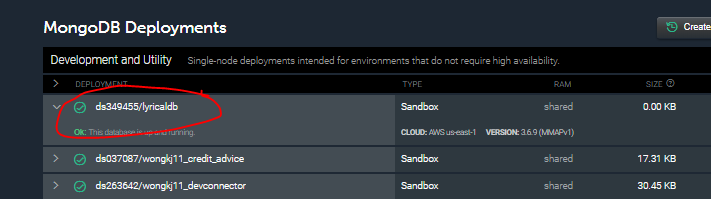
* Choose **US East** and click on **Continue**.



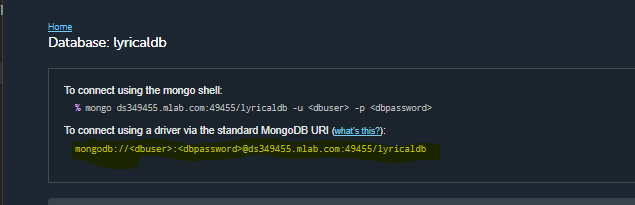
* Type in **lyricaldb** as the database name and click on **Continue**.



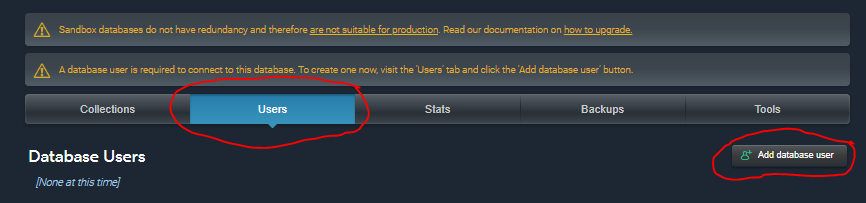
* Then click on **Submit Order**.
* Click on the **lyricaldb** database.



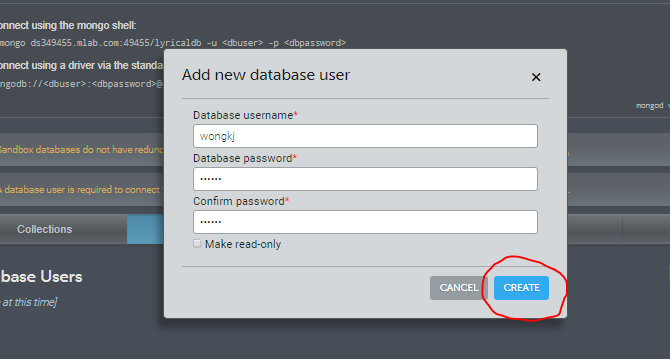
* The following string is the string we need to connect to mLab database through a driver.



* First we need to create a new User in order to provide the **dbuser** and **dbpassword**.
* We are now going to add a Database User.
* Click on the **Users** tab and then click on **Add database user**.



* The Username and Password chosen are as follows:
* **Username:** wongkj
* **Password:** abc123
* Click on **CREATE**.



* Copy that **MongoDB URI** from before.
* In VS Code, in the Lyrical-GraphQL folder, go to **server > server.js**.
* Add the MongoDB URI with the username and password into the following.

const express = require('express');

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

const expressGraphQL = require('express-graphql');

const mongoose = require('mongoose');

const bodyParser = require('body-parser');

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

const app = express();

// Replace with your mongoLab URI

const MONGO\_URI = 'mongodb://wongkj:abc123@ds349455.mlab.com:49455/lyricaldb';

if (!MONGO\_URI) {

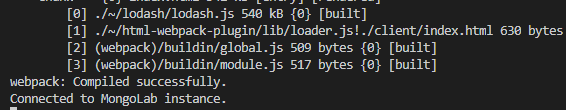
throw new Error('You must provide a MongoLab URI');

}

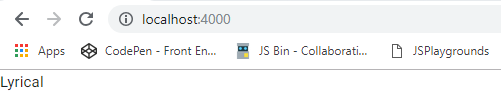
* Now go to the terminal and run the server by typing **npm run dev**.



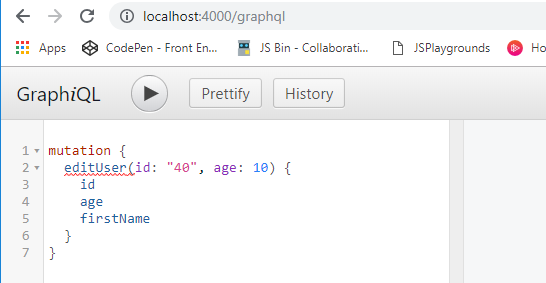
* You should see the following message in the terminal.



* Now we will visit the application in the web browser.
* In a web browser, navigate to [**http://localhost:4000**](http://localhost:4000)
* You should get the following.



* Since this is also a GraphQL Server we should be able to navigate to [**http://localhost:4000/graphql**](http://localhost:4000/graphql)



**LECTURE 34 – Working through the Schema**

* We are now going to use **GraphiQL** to add a song and add some lyrics to it.
* In GraphiQL, write the following query which will add a song to our database.

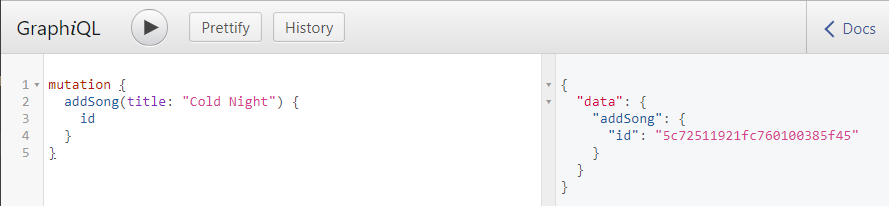
mutation {

addSong(title: "Cold Night") {

id

}

}



* Now we will add some lyrics to the song.
* First you will need to copy the id of the song that you just created as we need this to add the lyrics.
* Write the following lyric that takes in the **songId** and **content (lyrics to add to the song)** as the input and then have the **id** as the output.

mutation {

addLyricToSong(songId: "5c72511921fc760100385f45",

content: "Oh my oh my its a cold night") {

id

}

}



* Now we will write a query to extract that data.
* First we need to make a modification to the coding as there is an error.
* Go to **server > models > song.js**.
* In the **SongSchema** object, make the following changes.

const mongoose = require('mongoose');

const Schema = mongoose.Schema;

const SongSchema = new Schema({

title: { type: String },

user: {

type: Schema.Types.ObjectId,

ref: 'user'

},

lyrics: [{

type: Schema.Types.ObjectId,

ref: 'lyric'

}]

}, {

usePushEach: true

});

* In GraphiQL write the following query that gets back a list of songs, the id and title of the song, any lyrics in each song and the content of those lyrics.

{

songs {

id

title

lyrics {

content

}

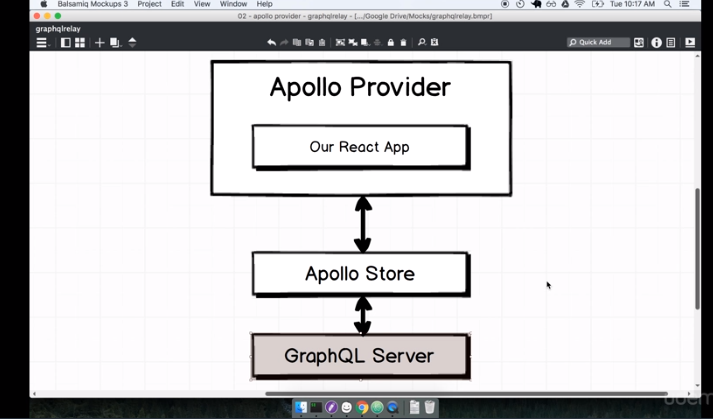
}

}



**LECTURE 35 – Apollo Client Setup**

* In VS Code, go to **client > index.js**.



* In between the React Application and the GraphQL Server are 2 key pieces of technology. The first is the Apollo Provider, the second is the Apollo Store.
* The **Apollo Store** is what is going to communicate directly with the GraphQL Server, and store data that comes back from it. You can think of the Apollo Store as a store of data that exists on the client side of our application.
* The Apollo Provider is the integration layer between our React Application and the Apollo Store. It is a Provider of data to our React Application. It provides the data from the Apollo Store to the React Application.
* The vast amount of the setup will be around the Apollo Provider.
* Go back to **client > index.js**.
* Import the **ApolloClient** from the **Apollo-client** library and import **ApolloProvider** from **react-apollo**.

import React from 'react';

import ReactDOM from 'react-dom';

import ApolloClient from 'apollo-client';

import { ApolloProvider } from 'react-apollo';

const Root = () => {

return <div>Lyrical</div>

};

* Make the following changes to the **Root** object.

import React from 'react';

import ReactDOM from 'react-dom';

import ApolloClient from 'apollo-client';

import { ApolloProvider } from 'react-apollo';

const Root = () => {

return (

<ApolloProvider>

<div>

Lyrical

</div>

</ApolloProvider>

);

};

* Create the following **client** object and add the client as a parameter to the **ApolloProvider** element.

import React from 'react';

import ReactDOM from 'react-dom';

import ApolloClient from 'apollo-client';

import { ApolloProvider } from 'react-apollo';

const client = new ApolloClient({});

const Root = () => {

return (

<ApolloProvider client={client}>

<div>

Lyrical

</div>

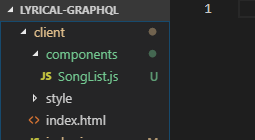
</ApolloProvider>

);

};

**LECTURE 36 – React Component Design**

* Remember, ApolloClient is front-end agnostic, meaning it doesn’t care what front-end framework you are using. It only wants to get data from our server and store it locally.
* We are now going to create a component that will list out all the songs in our application then write out a query that’s going to fetch that data from the server.
* We will start with the React side first.
* In VS Code, inside the **client** folder, create a new folder called **components** and then inside the components folder create a new file called **SongList.js**.



* Inside **SongList.js** write the following code.

import React, { Component } from 'react'

class SongList extends Component {

render() {

return (

<div>

SongList

</div>

)

}

}

export default SongList;

* Go back to **client > Index.js** and then import the **SongList** component and insert the **SongList** component inside the render.

import React from 'react';

import ReactDOM from 'react-dom';

import ApolloClient from 'apollo-client';

import { ApolloProvider } from 'react-apollo';

import SongList from './components/SongList';

const client = new ApolloClient({});

const Root = () => {

return (

<ApolloProvider client={client}>

<SongList />

</ApolloProvider>

);

};

**LECTURE 37 – GQL Queries in React**

* The first thing we are going to do is identify the data we need.
* The data we are looking for is the Song Titles.
* Now we write out the query in GraphiQL to extract the information that we are looking for.

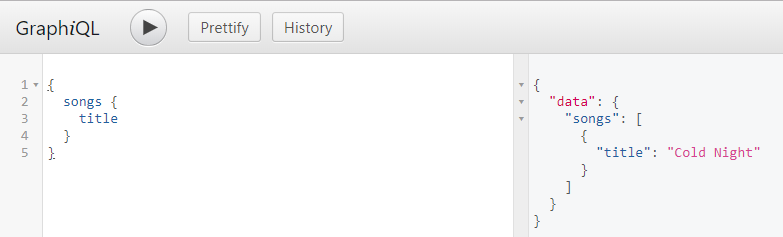
{

songs {

title

}

}



* Go back to **SongList.js** and import **graphql-tag**.

import React, { Component } from 'react';

import gql from 'graphql-tag';

class SongList extends Component {

render() {

return (

<div>

SongList

</div>

)

}

}

* In **SongList.js** write the following query object.

class SongList extends Component {

render() {

return (

<div>

SongList

</div>

)

}

}

const query = gql`

{

songs {

title

}

}

`;

export default SongList;

**LECTURE 38 – Bonding Queries with Components**

* We are now going to bond the query and component together.
* First, import **graphql** from **react-apollo** in the **SongList.js** file.

import React, { Component } from 'react';

import gql from 'graphql-tag';

import { graphql } from 'react-apollo';

class SongList extends Component {

render() {

return (

<div>

SongList

</div>

)

}

}

* In **SongList.js** at the bottom export make the following changes.

const query = gql`

{

songs {

title

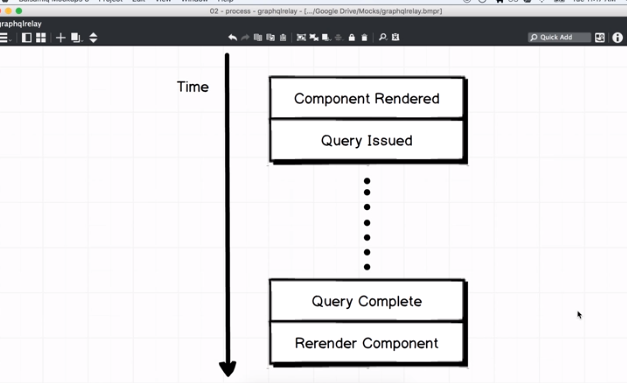
}

}

`;

export default graphql(query)(SongList);

* Below is a diagram on how the data is queried and then rendered.



* The data that is extracted and then stored inside the Components **props** object.
* The **props** object is used inside of React for communicating some outside data into the component itself.
* Go to the web browser, at [**http://localhost:4000**](http://localhost:4000) and open the Console.
* Refresh the browser and you should see in the Console, the data.



**LECTURE 39 – Handling Bonding Queries**

* Our data is located in the **this.props.data.songs** property.
* In **SongList.js** create the following Helper Method.

class SongList extends Component {

renderSongs() {

return this.props.data.songs.map(song => {

return (

<li>

{ song.title }

</li>

);

});

}

render() {

return (

<div>

{ this.renderSongs() }

</div>

)

}

}

* As per the diagram above, the query actually renders twice but the first time that it is rendered there is no object being returned. In order to get around this we need to use the following line of code that states that if the query is still loading then to simply return a div with the words **loading…**.

class SongList extends Component {

renderSongs() {

return this.props.data.songs.map(song => {

return (

<li>

{ song.title }

</li>

);

});

}

render() {

if (this.props.data.loading) { return <div>Loading...</div>; }

return (

<div>

{ this.renderSongs() }

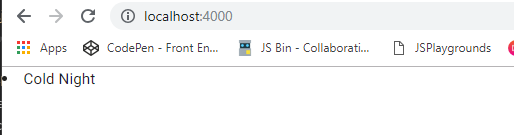
</div>

)

}

}

* When you reload the web browser you should get the list of songs.



**LECTURE 40 – Key Warnings**

* We need to add a key for each item of data returned.
* We will be using the id of each song as our key, which means that we need to fetch the data through our query.
* Modify the query in **SongList.js** to also include the **id** as returned item.

const query = gql`

{

songs {

id

title

}

}

`;

export default graphql(query)(SongList);

* Make the following changes to the **renderSongs** method so that it also includes the key for each List Item.

import React, { Component } from 'react';

import gql from 'graphql-tag';

import { graphql } from 'react-apollo';

class SongList extends Component {

renderSongs() {

return this.props.data.songs.map(song => {

return (

<li key={ song.id }>

{ song.title }

</li>

);

});

}

render() {

if (this.props.data.loading) { return <div>Loading...</div>; }

return (

<div>

{ this.renderSongs() }

</div>

)

}

}

* We are going to add some styling to the project by using the **materialize css** library which this project is already hooked up to.
* Make the following amendments to the **li** list item of the **renderSongs** method and also to the render return statement.

import React, { Component } from 'react';

import gql from 'graphql-tag';

import { graphql } from 'react-apollo';

class SongList extends Component {

renderSongs() {

return this.props.data.songs.map(song => {

return (

<li key={ song.id } className="collection-item">

{ song.title }

</li>

);

});

}

render() {

if (this.props.data.loading) { return <div>Loading...</div>; }

return (

<ul className="collection">

{ this.renderSongs() }

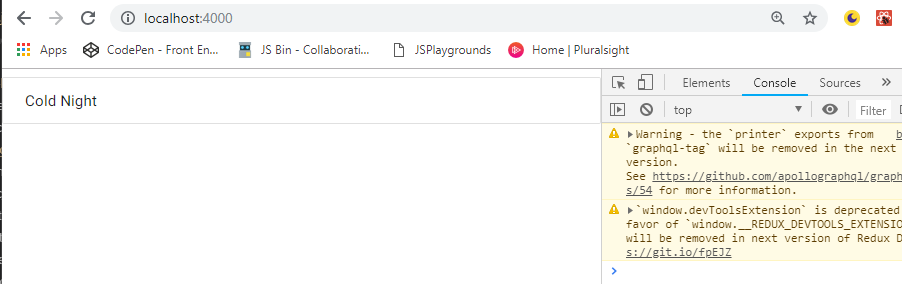
</ul>

)

}

}

* If you go back to the Web Browser and refresh the page you should get the following.



**LECTURE 41 – Architecture Review**

**LECTURE 42 – Adding React Router**

* We are going to start by importing a couple **helpers** from the **react-router** library.
* Go to **client > index.js** and import the following library.

import React from 'react';

import ReactDOM from 'react-dom';

import ApolloClient from 'apollo-client';

import { ApolloProvider } from 'react-apollo';

import { Router, Route, hashHistory, IndexRoute } from 'react-router';

import SongList from './components/SongList';

* In the **Root** object, in the return statement, make the following amendments.

const client = new ApolloClient({});

const Root = () => {

return (

<ApolloProvider client={client}>

<Router history={hashHistory}>

<Route path="/" component={App}>

</Route>

</Router>

</ApolloProvider>

);

};

ReactDOM.render(

<Root />,

document.querySelector('#root')

);

* Insert the **IndexRoute** so that when someone navigates to the index page (root page), the SongList component will be displayed.

const Root = () => {

return (

<ApolloProvider client={client}>

<Router history={hashHistory}>

<Route path="/" component={App}>

<IndexRoute component={SongList} />

</Route>

</Router>

</ApolloProvider>

);

};

* Import the **App** Component which we haven’t created yet.

import React from 'react';

import ReactDOM from 'react-dom';

import ApolloClient from 'apollo-client';

import { ApolloProvider } from 'react-apollo';

import { Router, Route, hashHistory, IndexRoute } from 'react-router';

import App from './components/App';

import SongList from './components/SongList';

* In the components folder, create a new file called **App.js**.
* Go to **App.js** and inside the file use the **rfc** short-cut keys to create a Functional Component.

import React from 'react';

export default function App() {

return (

<div>

</div>

)

}

* Use destructuring to modify the code so that any child components of the App Component are shown in the return statement.

import React from 'react';

export default function App({

children

}) {

return (

<div className="container">

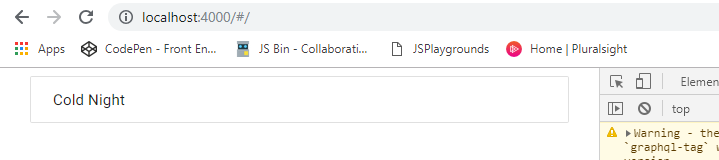
{children}

</div>

)

}

* What this code is saying is that if the application decides to show the IndexRoute of SongList, then that will be passed to the parent App Component and the App Component will then display the SongList Component.
* Now when you go to the browser and refresh the page you should get the url with a hash next to it indicating the root page and the song title should now have the container formatting applied to it.



**LECTURE 43 – Create a Song**

* In the components folder create a new file called **SongCreate.js**.
* In the **SongCreate.js** file use the **rcc** short-cut keys add the boiler plate code.

import React, { Component } from 'react';

class SongCreate extends Component {

render() {

return (

<div>

</div>

)

}

}

export default SongCreate;

* Add a **form** to the return statement.

import React, { Component } from 'react';

class SongCreate extends Component {

render() {

return (

<div>

<form>

</form>

</div>

)

}

}

export default SongCreate;

* Add the following Heading.

class SongCreate extends Component {

render() {

return (

<div>

<h3>Create a New Song</h3>

<form>

</form>

</div>

)

}

}

* Now we need to import **SongCreate.js** in **client > index.js** and then hook it up to ReactRouter.

import React from 'react';

import ReactDOM from 'react-dom';

import ApolloClient from 'apollo-client';

import { ApolloProvider } from 'react-apollo';

import { Router, Route, hashHistory, IndexRoute } from 'react-router';

import App from './components/App';

import SongList from './components/SongList';

import SongCreate from './components/SongCreate';

const client = new ApolloClient({});

const Root = () => {

return (

<ApolloProvider client={client}>

<Router history={hashHistory}>

<Route path="/" component={App}>

<IndexRoute component={SongList} />

<Route path="songs/new" component={SongCreate} />

</Route>

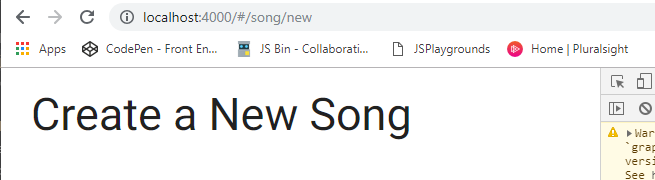
</Router>

</ApolloProvider>

);

};

* Now we will test if the Route works.
* In the browser navigate to [**http://localhost:4000/#/songs/new**](http://localhost:4000/#/songs/new)
* You should see the following.



* We will now add an input into the form.
* Go to **SongCreate.js** and add the following constructor method in order to initialize the state of the title parameter.
* Also, add the following label and input with the input getting an **onChange** parameter that will mutate the state based on the User input.

class SongCreate extends Component {

constructor(props) {

super(props);

this.state = { title: '' }

}

render() {

return (

<div>

<h3>Create a New Song</h3>

<form>

<label>Song Title:</label>

<input

onChange={ event => this.setState({ title: event.target.value }) }

value={ this.state.title }

/>

</form>

</div>

)

}

}

**LECTURE 44 – Mutations in React**

* In **SongCreate.js** make the following changes so that we handle the submitting functions ourselves.

class SongCreate extends Component {

constructor(props) {

super(props);

this.state = { title: '' }

}

onSubmit(event) {

event.preventDefault();

}

render() {

return (

<div>

<h3>Create a New Song</h3>

<form onSubmit={ this.onSubmit.bind(this) }>

<label>Song Title:</label>

<input

onChange={ event => this.setState({ title: event.target.value }) }

value={ this.state.title }

/>

</form>

</div>

)

}

}

* We are now going to test GraphiQL again to see if the addSong function still works.
* Go to GraphiQL and write the following query.

mutation {

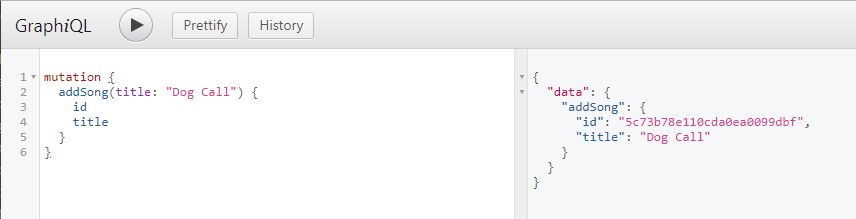
addSong(title: "Dog Call") {

id

title

}

}



* We are now going to take the mutation and move it on to the SongCreate Component file.
* Go to **SongCreate.js** and import **gql** from the **graphql-tag** library.

import React, { Component } from 'react';

import gql from 'graphql-tag';

class SongCreate extends Component {

constructor(props) {

super(props);

* Go to the bottom of the file and write the following mutation object.

const mutation = gql`

mutation {

addSong(title: "Dog Call") {

id

title

}

}

`;

export default SongCreate;

**LECTURE 45 – Query Params**

* We now need to somehow find a way to get the input value from the SongCreate Component into our mutation object.
* In order to rectify this problem we use a GraphQL Method called **Query Variables**.
* **Query Variables** are used to inject some variable from outside of the query into the query.
* Go to GraphiQl and write the following query that includes **Query Variables**.
* This query adds in the external argument of **$title** which we have initialized to a null object and then

mutation AddSong($title: String){

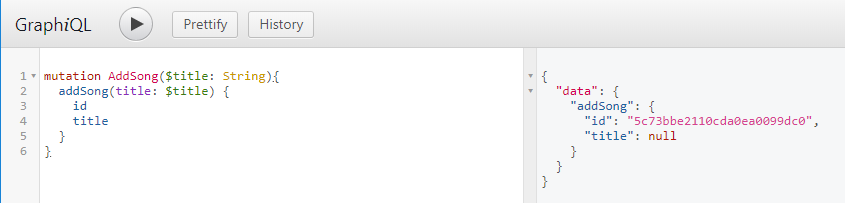
addSong(title: $title) {

id

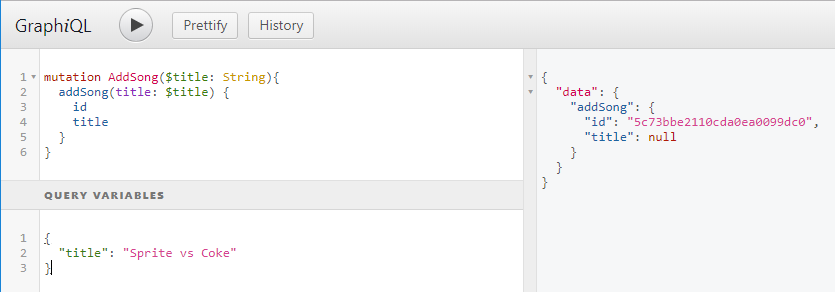
title

}

}



* Go to mLab and delete that null titled item.
* Now go back to GraphiQL and write the following **Query Variable** that has the title of **Sprite vs Coke**.
* Run this query and you should get a new item in the database of a song with a title of Sprite vs Coke.



**LECTURE 46 – Defining Query Variables in React**

* We now need to figure out how we get some information from our Component into the mutation by using the Query Variable system.
* Go do **SongCreate.js** and make the following changes to the mutation object.

const mutation = gql`

mutation AddSong($title: String) {

addSong(title: $title) {

title

}

}

`;

export default SongCreate;

* We are now going to use the **graphql helper** at the bottom that will enable us to combine the 2 sets of data together.
* At the top of **SongCreate.js** import the following from **react-apollo**.

import React, { Component } from 'react';

import gql from 'graphql-tag';

import { graphql } from 'react-apollo';

class SongCreate extends Component {

constructor(props) {

super(props);

* Next, make the following changes to the export function at the bottom of the file.

const mutation = gql`

mutation AddSong($title: String) {

addSong(title: $title) {

title

}

}

`;

export default graphql(mutation)(SongCreate);

* In the **onSubmit** function, add the following code to extract the data from the state.title and pass that into the mutation object query.

class SongCreate extends Component {

constructor(props) {

super(props);

this.state = { title: '' }

}

onSubmit(event) {

event.preventDefault();

this.props.mutate({

variables: {

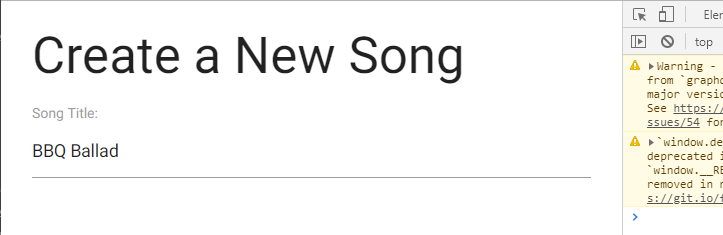
title: this.state.title

}

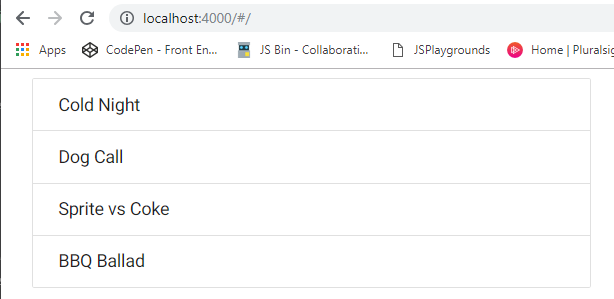
})

}

* Now we will test this.
* Go to the browser and navigate to [**http://localhost:4000/#/songs/new**](http://localhost:4000/#/songs/new)
* As the Song Title write **BBQ Ballad** and then hit **ENTER**.



* When you navigate to the root page and refresh it you should see the new song title in the list.



**LECTURE 47 – Navigating on Successful Mutation**

* We are now going to create a button on the **SongList** page to enable the user to navigate to the SongCreate page.
* Go to **SongList.js** and import the **Link** helper from the **react-router** library.

import React, { Component } from 'react';

import gql from 'graphql-tag';

import { graphql } from 'react-apollo';

import { Link } from 'react-router';

* Wrap the cod in the render with a div element.

class SongList extends Component {

renderSongs() {

return this.props.data.songs.map(song => {

return (

<li key={ song.id } className="collection-item">

{ song.title }

</li>

);

});

}

render() {

if (this.props.data.loading) { return <div>Loading...</div>; }

return (

<div>

<ul className="collection">

{ this.renderSongs() }

</ul>

</div>

)

}

}

* In SongList.js make the following changes.

render() {

if (this.props.data.loading) { return <div>Loading...</div>; }

return (

<div>

<ul className="collection">

{ this.renderSongs() }

</ul>

<Link

to="/songs/new"

className="btn-floating btn-large green right"

>

<i className="material-icons">add</i>

</Link>

</div>

)

}

}

* Now we are going to add a button in the SongCreate page that will enable us to navigate back to the SongList page.
* Go to **SongCreate.js** and import the **Link** helper from react-router.

import React, { Component } from 'react';

import gql from 'graphql-tag';

import { graphql } from 'react-apollo';

import { Link } from 'react-router';

* Make the following changes.

render() {

return (

<div>

<Link

to="/"

>

Back

</Link>

<h3>Create a New Song</h3>

<form onSubmit={ this.onSubmit.bind(this) }>

<label>Song Title:</label>

<input

onChange={ event => this.setState({ title: event.target.value }) }

value={ this.state.title }

/>

</form>

</div>

)

}

}

* Now we need to modify the code so that whenever a User enters a new Song, the application automatically kicks them back to the SongList page upon a successful mutation.
* Go to **SongCreate.js** and make the following changes to the onSubmit mutation so that a promise will be executed upon the successful mutation and this promise will navigate the User back to the root page.

class SongCreate extends Component {

constructor(props) {

super(props);

this.state = { title: '' }

}

onSubmit(event) {

event.preventDefault();

this.props.mutate({

variables: {

title: this.state.title

}

}).then(() => hashHistory.push('/'))

}

* Make sure to import **hashHistory** at the top of the page.

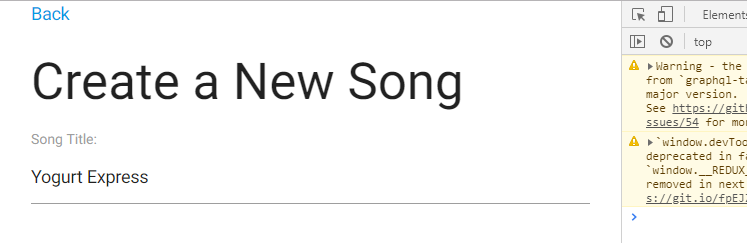
import React, { Component } from 'react';

import gql from 'graphql-tag';

import { graphql } from 'react-apollo';

import { Link, hashHistory } from 'react-router';

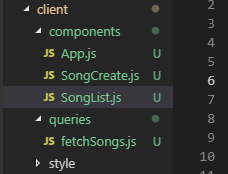
* We will now test this out.
* Go to the browser and navigate to [**http://localhost:4000/#/songs/new**](http://localhost:4000/#/songs/new)
* Type in the song name **Yogurth Express** and hit **ENTER**. As soon as you hit Enter you should be navigated back to the root page.



**LECTURE 48 – Troubleshooting List Fetching**

**LECTURE 49 – Refetching Queries**

* When you do more than one creation of songs and redirected to the SongList page, on the second redirection you will not see the new song that was created. A Requery needs to be done to extract this.
* We will be using the **refetchQueries** method to rectify this.
* First we will need to move the **query** object from the **SongList.js** page into a separate page and then import it into **SongList.js**.
* In the **client** folder create another folder called **queries**.
* Inside the queries folder create a new file called **fetchSongs.js**.



* In **fetchSongs.js** write the following code.

import gql from 'graphql-tag';

export default gql`

{

songs {

id

title

}

}

`;

* We now need to import this into **SongList.js**.
* Go to **SongList.js** and import the **fetchSongs.js** file.

import React, { Component } from 'react';

import gql from 'graphql-tag';

import { graphql } from 'react-apollo';

import { Link } from 'react-router';

import query from '../queries/fetchSongs';

* At the bottom of **SongList.js** delete the **query** object.

const query = gql`

{

songs {

id

title

}

}

`;

export default graphql(query)(SongList);

* Now we will need to import the query in to the **SongCreate.js** file as we will be using the query in there as well.

import React, { Component } from 'react';

import gql from 'graphql-tag';

import { graphql } from 'react-apollo';

import { Link, hashHistory } from 'react-router';

import query from '../queries/fetchSongs';

* In **SongCreate.js** and add the **refetchQueries** method and the array items that need to be added to this method is the array of queries you want to re-run after a successful mutation. Also add the **query** object as the query parameter.

class SongCreate extends Component {

constructor(props) {

super(props);

this.state = { title: '' }

}

onSubmit(event) {

event.preventDefault();

this.props.mutate({

variables: {

title: this.state.title

},

refetchQueries: [{ query: query }]

}).then(() => hashHistory.push('/'))

}

* We will now test this.
* Go to the web browser and navigate to [**http://localhost:4000/#/songs/new**](http://localhost:4000/#/songs/new)
* Create a new song called **Mlab Rock**.

**LECTURE 50 – Deletion by Mutation**

* We are going to test how to do a deletion mutation.
* First list all the songs using the following query.

{

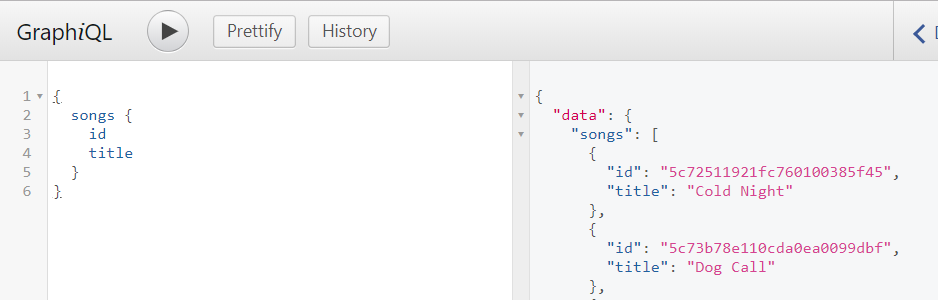
songs {

id

title

}

}



* Next, write the following delete mutation with associated query variable.

**IMPORTANT:** On the back-end the Lecturer changed the type of the id to **ID** instead of **String** as it was originally.

mutation DeleteSong($id: ID) {

deleteSong(id: $id) {

id

title

}

}



**LECTURE 51 – Associating Mutations with a Component**

* Since we are only going to be using the **deleteSong** mutation in the SongList file, we don’t need to create a separate file for this mutation and we will just host the mutation in the SongList file.
* Go to **SongList.js** and add the following mutation.

const mutation = gql`

mutation DeleteSong($id: ID) {

deleteSong(id: $id) {

id

}

}

`;

export default graphql(query)(SongList);

* We will now need to add the mutation to the query helper at the bottom but since we can’t use more that one query/mutation per graphql help we need to modify the code.
* The below says that you start by invoking the **query** graphql helper, and then the result of that will get passed to the **mutation** graphql helper which will then execute it’s mutation on that result.

const mutation = gql`

mutation DeleteSong($id: ID) {

deleteSong(id: $id) {

id

}

}

`;

export default graphql(mutation)(

graphql(query)(SongList)

);

**LECTURE 52 – Invoking Delete Mutations**

* In **SongList.js** add the following delete icon that has an onClick method assigned to it called **onSongDelete**.

class SongList extends Component {

renderSongs() {

return this.props.data.songs.map(song => {

return (

<li key={ song.id } className="collection-item">

{ song.title }

<i

className="material-icons"

onClick={ () => this.onSongDelete(song.id) }

>delete</i>

</li>

);

});

}

* Do some refactoring so that we destructure the **id** and **title** from the **song** object.

renderSongs() {

return this.props.data.songs.map(({ id, title }) => {

return (

<li key={ id } className="collection-item">

{ title }

<i

className="material-icons"

onClick={ () => this.onSongDelete(id) }

>delete</i>

</li>

);

});

}

* Write the following **onSongDelete** function.

class SongList extends Component {

onSongDelete(id) {

this.props.mutate({ variables: { id: id } })

}

renderSongs() {

return this.props.data.songs.map(({ id, title }) => {

return (

<li key={ id } className="collection-item">

{ title }

<i

className="material-icons"

onClick={ () => this.onSongDelete(id) }

>delete</i>

</li>

);

});

}

**LECTURE 53 – Refetching a Query**

* In this section, we are going to use Refetching to make sure that the item deleted automatically gets removed from the UI.
* We are now going to use a different approach to refetch a query as opposed to the other approach that we used before.
* The original approach for refetching is used when the query being fetched is located in another Component. If the query is located in the same Component then it is better to use the second approach.
* In **SongList.js** make the following changes to the **onSongDelete** function.

class SongList extends Component {

onSongDelete(id) {

this.props.mutate({ variables: { id: id } })

.then(() => this.props.data.refetch())

}

renderSongs() {

return this.props.data.songs.map(({ id, title }) => {

return (

<li key={ id } className="collection-item">

{ title }

<i

className="material-icons"

onClick={ () => this.onSongDelete(id) }

>delete</i>

</li>

);

});

}

**LECTURE 54 – A Quick CSS Breather**

* We are going to modify the delete icon so that the delete icon is located on the far-right-side of the row and so that a finger pointing icon appears when you hover over the icon.
* Go to **client > style > style.css** and add the following css.

.collection-item {

display: flex;

justify-content: space-between;

}

.material-icons {

cursor: pointer;

}

* We are now going to import the **style.css** file in to our project.
* Go to **client > index.js** and import the style.css file.

import './style/style.css';

import React from 'react';

import ReactDOM from 'react-dom';

import ApolloClient from 'apollo-client';

import { ApolloProvider } from 'react-apollo';

import { Router, Route, hashHistory, IndexRoute } from 'react-router';

import App from './components/App';

import SongList from './components/SongList';

import SongCreate from './components/SongCreate';

**LECTURE 55 – Showing a Particular Song**

* We are now going to start working on the **SongDetail** page which displays the information related to a particular song.
* Inside the **components** folder create another file called **SongDetail.js**.
* Inside **SongDetail.js** create a Class Component.

import React, { Component } from 'react';

class SongDetail extends Component {

render() {

return (

<div>

</div>

)

}

}

export default SongDetail;

* Add the following boiler-plate code.

class SongDetail extends Component {

render() {

return (

<div>

<h3>Song Detail</h3>

</div>

)

}

}

* Go to **client > index.js** and import the **SongDetail** Component.

import './style/style.css';

import React from 'react';

import ReactDOM from 'react-dom';

import ApolloClient from 'apollo-client';

import { ApolloProvider } from 'react-apollo';

import { Router, Route, hashHistory, IndexRoute } from 'react-router';

import App from './components/App';

import SongList from './components/SongList';

import SongCreate from './components/SongCreate';

import SongDetail from './components/SongDetail';

* Add the following Route for the **SongDetail** Component.

const Root = () => {

return (

<ApolloProvider client={client}>

<Router history={hashHistory}>

<Route path="/" component={App}>

<IndexRoute component={SongList} />

<Route path="songs/new" component={SongCreate} />

<Route path="songs/:id" component={SongDetail} />

</Route>

</Router>

</ApolloProvider>

);

};

**LECTURE 56 – Fetching Individual Records**

* First we need to test out extracting a specific song by first going through GraphiQL.
* Type in the following query with a query variable in GraphiQL.

query SongQuery($id: ID!){

song(id: $id) {

id

title

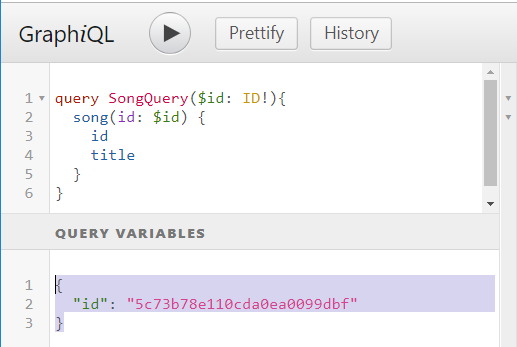
}

}

{

"id": "5c73b78e110cda0ea0099dbf"

}



**IMPORTANT:** The exclamation mark right next to **ID** means that this particular argument MUST be provided otherwise the query will throw an error.

* We will now need to bring in this query in to SongDetail.js.
* First, create a separate file for the query.
* Go to **client > queries** and create a new file called **fetchSong.js**.



* Inside **fetchSong.js** write the following code.

import gql from 'graphql-tag';

export default gql`

query SongQuery($id: ID!){

song(id: $id) {

id

title

}

}

`;

**LECTURE 57 – Interacting React Router with GraphQL**

* Sf
* Asf
* Asf
* Asf
* fasdf
* Sfsdf
* Sd
* Fsd
* Fsd
* Fsd
* Fdd
* Sdfs
* Fss
* Sf
* Sf
* Sff
* Fs
* Asd
* Fa
* Sf
* Asf
* Asf
* Asf
* fasdf
* Sfsdf
* Sd
* Fsd
* Fsd
* Fsd
* Fdd
* Sdfs
* Fss
* Sf
* Sf
* Sff
* Fs
* Asd
* Fa
* Sf
* Asf
* Asf
* Asf
* fasdf