INFO3139 Lab 9

Rev 1.1

General JavaScript Topic – Why use === instead of ==

Although similar, they do different things:

- == is called the regular equality operator
- === is called the strict equality operator

The regular equality operator (==) only checks if operands **are similar**, which can cause some unpleasant surprises.

The strict equality operator (===) always checks that the operands are of different types and values and that they are exactly the same.

Here's an example to show the difference between the two operators:

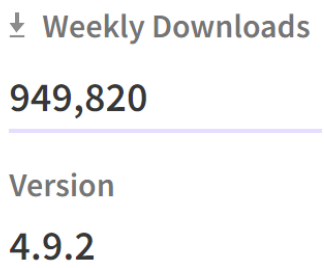
```
let someNumber = 10;
let someStringNumber = "10";
console.log(
  `T/F - using the regular equality operator the variables are considered equal - ${
    someNumber == someStringNumber
  }`
);
console.log(
  `T/F - using the strict equality operator the variables are considered equal - ${
    someNumber === someStringNumber
  }`
);
```

```
PS C:\Evan\Winter2023\info3139\programming\nodeexercises\week3\class2> node equality
T/F - using the regular equality operator the variables are considered equal - true
T/F - using the strict equality operator the variables are considered equal - false
```

Rule of thumb – always use the strict operator!

Case Study# 1 – cont'd

From the homework article we saw there is a newer framework that has some good reviews called **Fastify.** To start out today we're going to convert some of the work we did last class from using Express **to Fastify**. Keep in mind that Express is the most used, but it isn't subject to regular updates and doesn't offer the best performance, so for the remainder of the first case study we'll use **Fastify** (as a side note we'll revert back to Express for the 2nd case study).



1. From the nodeexercises folder **uninstall Express**:

```
PS E:\winter2023\info3139\programming\nodeexercises> npm uninstall express

removed 56 packages, and audited 138 packages in 3s

16 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
```

2. And in the same folder **install Fastify**:

```
PS E:\winter2023\info3139\programming\nodeexercises> npm i fastify

added 56 packages, and audited 194 packages in 49s

18 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
```

```
"dependencies": {
  "consola": "^2.15.3",
  "cors": "^2.8.5",
  "dotenv": "^16.0.3",
  "fastify": "^4.9.2",
  "got": "^12.5.2",
  "mongodb": "^4.11.0",
  "yargs": "^17.6.0"
}
```

3. Copy the .env, config.js, and db_routines.js files from week5\class1 to a new week5\class2 folder.

4. Add a new file called **fastify_routes.js** file with the following:

```
import * as dbRtns from "./db_routines.js";
import * as cfg from "./config.js";

// define a default route to retrieve all users
async function routes(fastify, options) {
  fastify.get("/api/users", async (request, reply) => {
    try {
      let db = await dbRtns.getDBInstance();
      let users = await dbRtns.findAll(db, cfg.collection);
      reply.status(200).send({ users: users });
    } catch (err) {
      console.log(err.stack);
      reply.status(500).send("get all users failed - internal server error");
    }
  });
}

export { routes };
```

5. Add a new **app.js** with the following:

```
import { port } from "./config.js";
import { routes } from "./fastify_routes.js";
import Fastify from "fastify";

const fastify = Fastify({
  logger: true,
});

// register the route module
fastify.register(routes);

// start the fastify server
fastify.listen(port, (err, address) => {
  if (err) {
    fastify.log.error(err);
    process.exit(1);
  }
});
```

6. Start the new server:

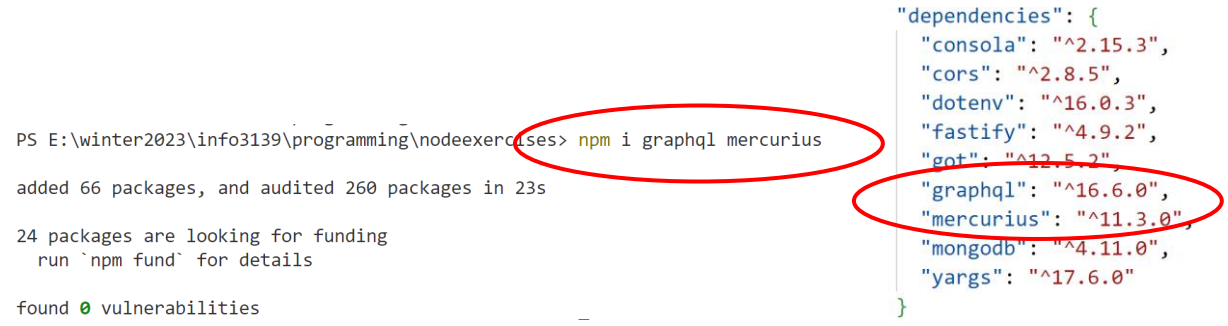
```
PS E:\winter2023\info3139\programming\nodeexercises\week5\class2> node app
(node:10544) [FSTDEP011] FastifyDeprecation: Variadic listen method is deprecated. Please use ".listen(optionsObject)" instead. The varia
dic signature will be removed in `fastify@5`.
(Use `node --trace-warnings ...` to show where the warning was created)
{"level":30,"time":1668460843790,"pid":10544,"hostname":"DESKTOP-5KL5GLM","msg":"Server listening at http://[::1]:5000"}
{"level":30,"time":1668460843804,"pid":10544,"hostname":"DESKTOP-5KL5GLM","msg":"Server listening at http://127.0.0.1:5000"}
```

7. Using a browser set the URL to **localhost:5000/api/users** and you should see the same JSON returned as the original express example:

With this simple Fastify server setup and tested we can now proceed to look at **GraphQL** and replace the REST calls with some GraphQL ones. Fastify doesn't understand GraphQL (fyi neither does express) by default so we need to install **graphql** and then a fastify graphql adapter called **Mercurius.**

Installing GraphQL and Mercurius

8. Shutdown the running app.js and issue the following command from the **nodeexercises** root to install **graphql and mercurius**:



9. Add the following entry to the **.env** file

```
GRAPHQLURL=/graphql
```

10. Replace the code in the **config.js** with this:

```
import { config } from "dotenv";
config();
export const atlas = process.env.DBURL;
export const db = process.env.DB;
export const collection = process.env.COLLECTION;
export const port = process.env.PORT;
export const graphql = process.env.GRAPHQLURL;
```

11. Create a new file called week5\class2**schema.js** with the following code (a schema describes the functionality available to the client applications that connect to it):

```
const schema = `
type Query {
        users: [User],
        userbyname(name: String): User
    },
type User {
        name: String
        age: Int
        email: String
    }
`;
export { schema };
```

12. Create another file called week5\class2**resolvers.js** with the following code (a resolver is a collection of functions that generate response for a GraphQL query, think of it as a GraphQL query handler):

```
import * as dbRtns from "./db_routines.js";
import * as cfg from "./config.js";

const resolvers = {
  users: async () => {
    let db = await dbRtns.getDBInstance();
    return await dbRtns.findAll(db, cfg.collection, {}, {});
  },
};

export { resolvers };
```

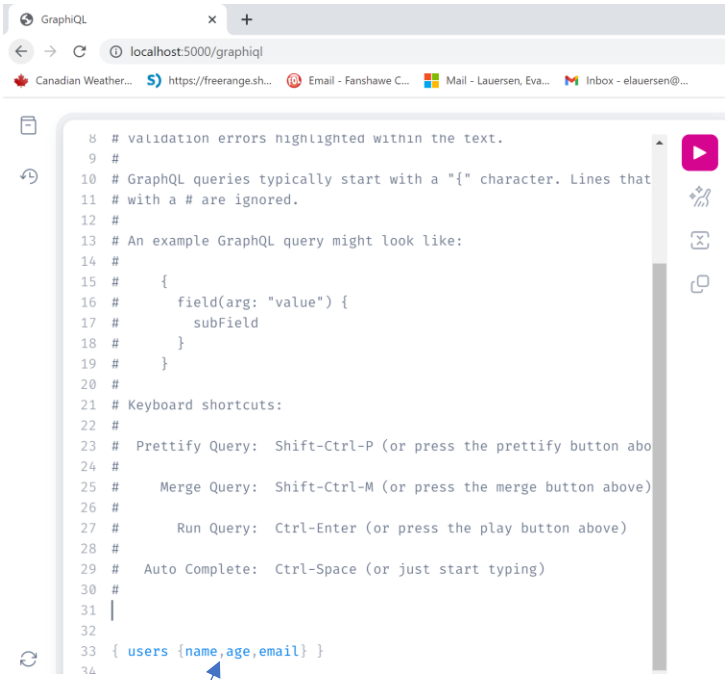
13. Add a new file called **gqlapp.js** with the following contents:

```
"use strict";
import * as cfg from "./config.js";
import Fastify from "fastify";
import mercurius from "mercurius";
import { schema } from "./schema.js";
import { resolvers } from "./resolvers.js";

const app = Fastify();

app.register(mercurius, {
  schema,
  resolvers,
  graphiql: true, // web page for to test queries
});

app.listen({ port: cfg.port });
```

14. Execute the **glqapp.js** in the terminal window:

```
PS E:\winter2023\info3139\programming\nodeexercises\week5\class2> node gqlapp
```

15. Open a browser and point the url to **localhost:5000/graphiql**

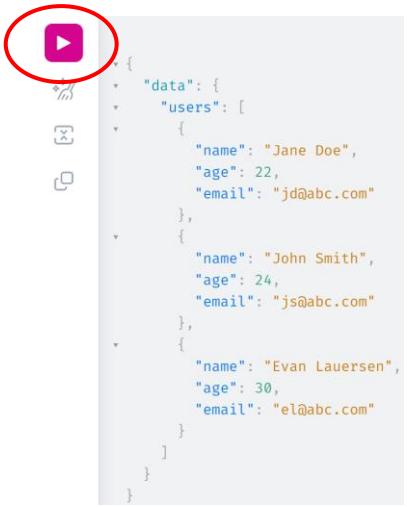


Note the spelling with an 'i'

We can now start testing some GraphQL queries using this utility called **GraphiQL** (pronounced graphical).

1. Enter the following query in the query pane and then press the play button:

```
{ users {name,age,email} }
```



Ok, so let's figure out what's going on here. First off, we have our new http server (fastify) that we added the graphql engine to it . To get the fastify server to communicate with the graphql we also installed an adapter called mercurius. Graphql includes a tool called GraphiQL where we can enter GraphQL queries in the left pane with the results and errors showing up in the right pane. The **queries** we can execute can be found in the **schema**.js file. In our case we currently have a couple querys; one query called **users** that returns an array of User types and another that returns a single user (userbyname). Let's focus on the first query. The resolver file we entered in step 7 included a **users** property that gives the users query the database access it needs to provide data for the query.

From last class' homework reading we know that one of the advantages GraphQL has over REST is the ability to return just the fields we need. To demonstrate this, change the query to return **just the name field**, and re-run the query:

```
{ users {name} }
```


```
{
  "data": {
    "users": [
      {
        "name": "Jane Doe"
      },
      {
        "name": "John Smith"
      },
      {
        "name": "Evan Lauersen"
      }
    ]
  }
}
```


We can also pass arguments to query to establish the **criteria**. Currently in the **schema.js** we have the following Query that passes a name argument:

```
type Query {
  users: [User],
  userbyname(name: String): User
}
```

16. To utilize the query, modify the **resolvers.js** and add a **userbyname** resolver:

```
  userbyname: async (args) => {
    let db = await dbRtns.getDBInstance();
    return await dbRtns.findOne(db, cfg.collection, { name: args.name });
  },
```

17. Save everything, restart the server and enter the following query in GraphiQL and you should see:

```
{ userbyname(name: "Jane Doe") {name, age, email} }
```

So, again we see how the GraphQL pattern works, this time we setup a query called **userbyname** that accepts a parameter(name) and returns a single User instance (name, age,email). This query gets its state from a resolver of the same name. This resolver accepts an args parameter and we strip the name property from it and pass it to the dbroutines.findOne method. The findOne method in turn uses the name to populate its criteria property.

18. Try going the other way now. Use GraphQL to add a user to the database, in GraphQL terms this is called a **mutation**. Update the **schema.js** file and add this to the existing code:

```
type Mutation {
  adduser(name: String, age: Int, email: String): User
}
```

19. Add this to the **resolvers.js** to process the new mutation:

```
    adduser: async args => {
        let db = await dbRtns.getDBInstance();
        let user = {name: args.name, age: args.age, email: args.email};
        let results = await dbRtns.addOne(db,coll,user);
        return results.insertedCount === 1 ? user : null;
    }
```

20. Save everything and restart the server with the new changes

21. Enter and execute the following command in the GraphiQL query pane:

```
mutation {
 adduser(name: "New User", age: 23, email: "nu@here.com") {name,age,email}
}
```

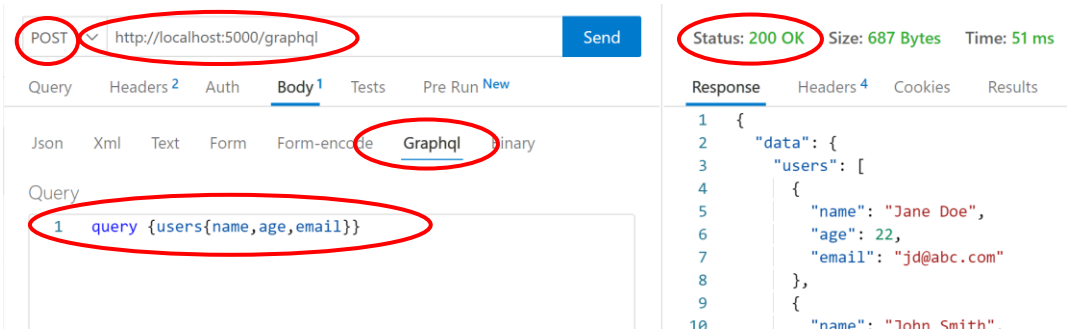
22. Confirm the data was added to the users collection on **Atlas**:

```
_id: ObjectId("5ddd3d26b9581a2e9478f5f2")
name: "New User"
age: 23
email: "nu@here.com"
```

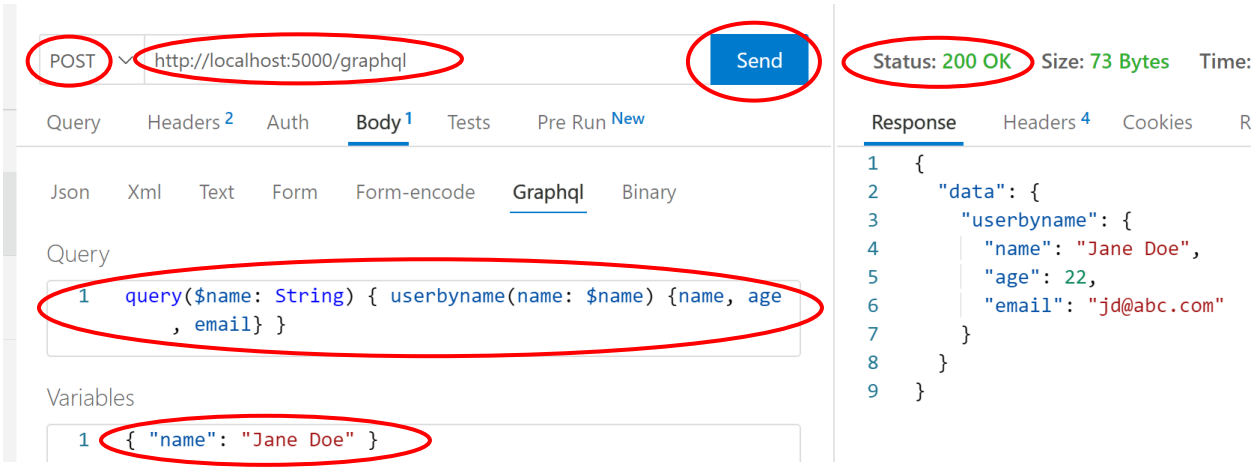
GraphiQL is a well and good, but we cannot expect an end user to know how to construct a query or mutation. We can however set up an API like what we did last class with our REST API. For a client to call our fastify server we simply need to send the query always using an HTTP **POST**.

Keep the gqlapp.js server running and then load **the Thunder Client** and enter the following settings and when complete press the SEND button:

```
query {users{name,age,email}}
```



And the other query **userbyname** in our schema can be tested as well, with the following in the Query Pane:
`query($name: String) { userbyname(name: $name) {name, age, email} }` and
`{ "name": "Jane Doe" }` for the GraphQL Variables Pane:

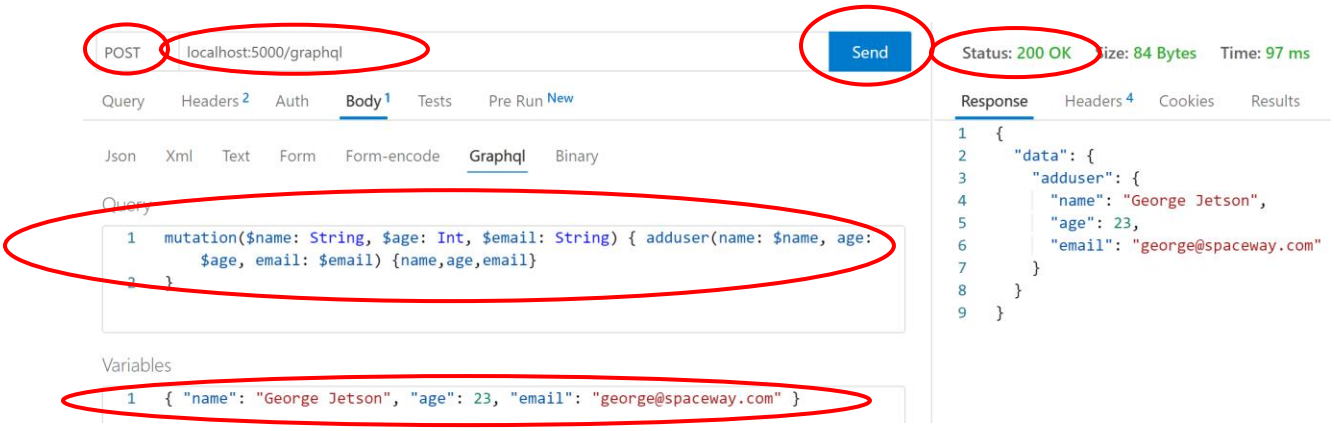


We can also do the **mutation** in the Thunder Client as well:

```
mutation($name: String, $age: Int, $email: String) { adduser(name: $name, age: $age, email: $email) {name,age,email} }
```

and

```
{ "name": "George Jetson", "age": 23, "email": "george@spaceway.com" }
```



With these examples covered we can now proceed to this week's lab

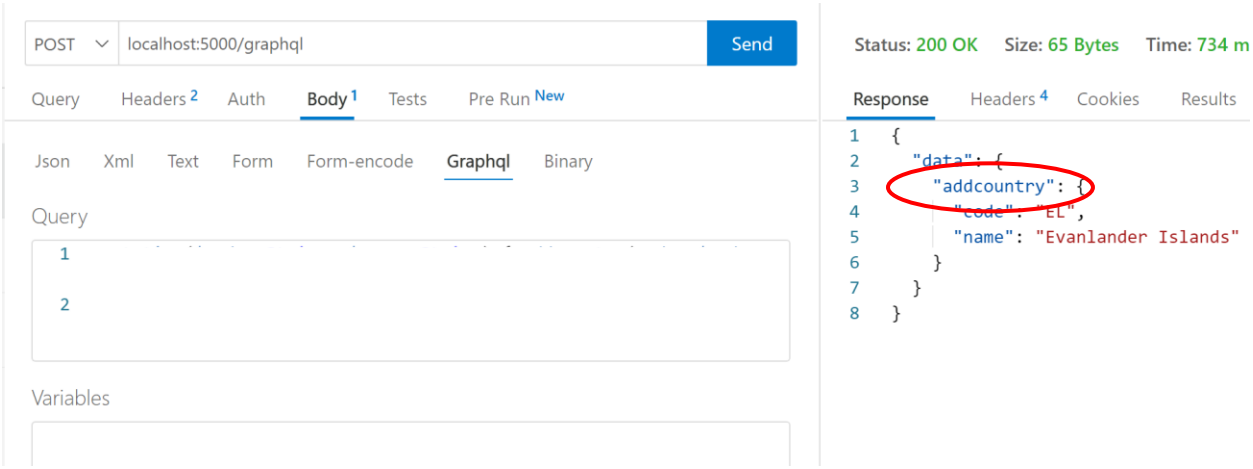
Lab 9

Part A - (1%)

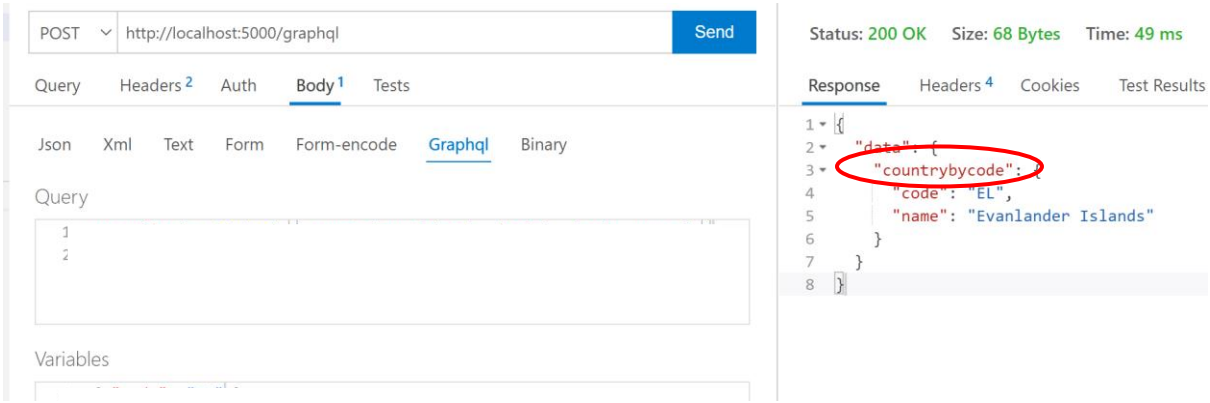
Using the Thunder Client, create 4 GraphQL queries that replicate user's example to go after the **country** collection that we set up in Week 4 on Atlas. Include the following 4 screenshots in a **single Word document**.:

You must use **GRAPHQL VARIABLES** in the next 3 screen shots (none are needed in #4)

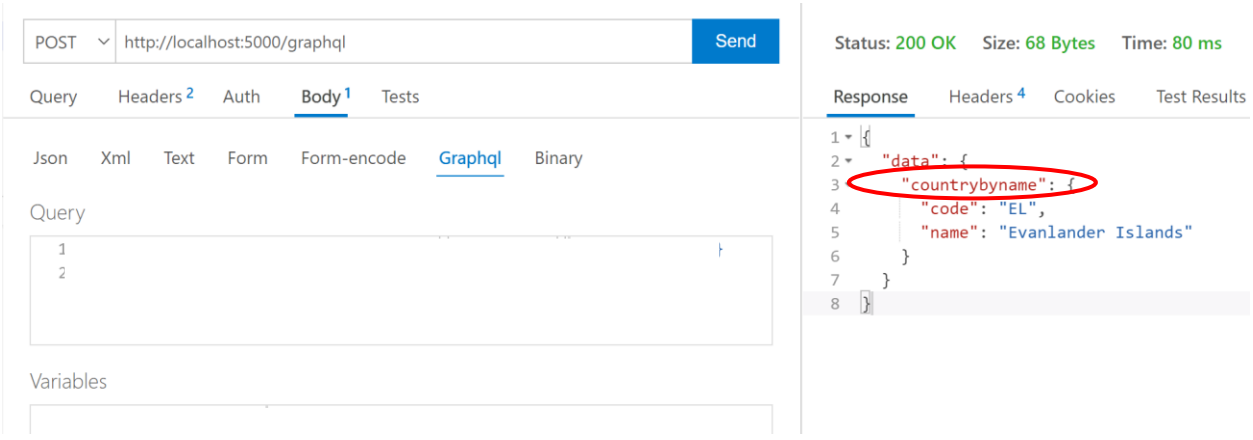
1. Display the results of a **mutation** posting a fictious country that includes **your name** in it and your initials as the country code, if there is already a country code with your initials use your middle initial, ensure to **use graphql variables** for this one. **Display the entire Thunder Client New Request Tab contents** (note I've blanked the query and the graphql variables in the screen shots below and just show the results):



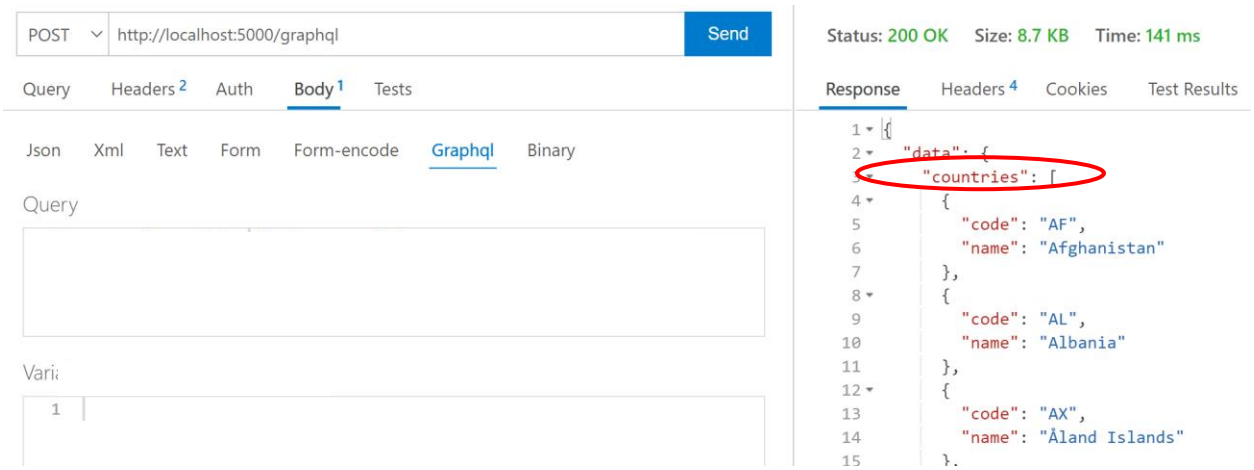
2. Display the results of querying a country **by code** with the code you used in the mutation, again my query is hidden but be **sure to include the query and graphql variables in the screen shot:**



3. Display the results of querying a country **by name** again with the name you used in part 2**,** again be **sure to include the query and graphql variables in the screen shot**



4. Display the start of the entire collection **(I have hidden the query**, **but ensure your screen shot displays the query** in addition to these results):

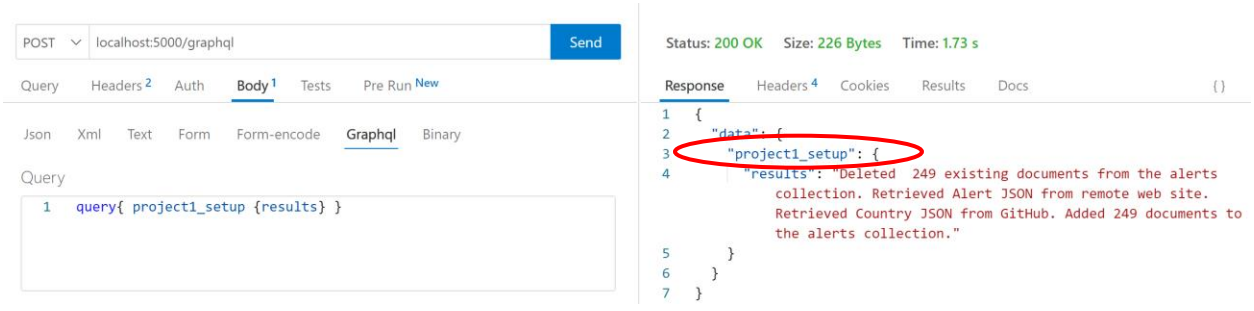


Part B – (1%)

We are now in shape to add the Fastify framework to the case study. For now, we will just focus on web enabling our current server functionality. When we last looked at the case study project we were at the point where we were dumping all the steps out to the browser like this:

```
PS E:\winter2023\info3139\programming\nodeexercises\project1> node project1_setup
establishing new connection to Atlas
Deleted  249 existing documents from the alerts collection.
Retrieved Alert JSON from remote web site.
Retrieved Country JSON from GitHub.
added 249 documents to the alerts collection
```

Now, instead of dumping out to the node console, we want to return a JSON object that can be returned from a graphql query (project1_setup below):



To achieve this, we need to do the following

1. Update the **.env** and **config.js** files in the project1 folder to add port 5000 like we did today.
2. Copy **gqlapp.js** from the week5\class2 to **app.js** in the project1 folder and use that code as your server
3. Update the existing **project1_setup.js** so it returns the concatenated results as JSON, the end of the method would be something like:

```
…
  } catch (err) {
    console.log(err);
  } finally {
    return { results: results };
  }
};

export { loadAlerts };
```

4. Add a new **schema.js** file to the project1 folder that contains a **Results** Type **with a single String property**, and a single query (**project1_setup** above) that returns the Results type
5. Add a new **resolvers.js** to project1 folder that contains a single query handler that returns the result JSON from the setup module
6. Add a single Thunder Client screen shot like the one above to the existing Word document **and also** include the same document the **source code for your schema and resolver files**.

Part C - (.5%) - Lab 9 Theory Quiz on FOL - 10 questions

Review

1. What is GraphQL
2. GraphQL offers some advantages over REST, what is the best advantage of using GraphQL according to the homework reading
3. What is a GraphQL Query, Mutation, Schema, Type and Resolver
4. What npm packages do we need to install to setup a GraphQL server in node?
5. Explain what this piece of schema code is trying to achieve:

```
type Query {
  users: [User],
  userbyname(name: String): User
}
```

6. T/F – our resolver file contained database code
7. T/F – our resolver file contained a resolver object whose property names match the names found in the schema
8. What 3 properties were passed when registering the mecurial middleware
9. What GUI is provided with the graphql server?

10. Give an example of what a query would look like when entered in GraphiQL
11. If I have a query with a format of: { x {y,x} }, what does the y,x represent
12. If I have a query with a format of:{ x(y:z) {a,b,c} }, what does the y:z represent
13. What would my query look like if I wanted to add a user to our users collection in Atlas with GraphiQL
14. GraphiQL isn't the only way to enter queries, what other software did we utilize today to query our GraphQL server?
15. What would be the format for entering the query from Q12 in the Thunder client?
16. What is the name of the property in the **results** that all the information is contained in?
17. What HTTP code is returned in Thunder Client extension if the GraphQL query was successful

Homework

To get an insight into the React framework we'll start looking at next class read the following links **:**

1. https://www.stackchief.com/tutorials/Learn%20ReactJS%20Introduction
2. https://www.stackchief.com/tutorials/Learn%20ReactJS%20JSX
3. https://www.stackchief.com/tutorials/Learn%20ReactJS%20Components