INFO3139 Lab 8

Rev 1.0

General JavaScript Topic – var, const and let

When we declare a new variable in JavaScript, we can either use **const, let, or var**. Each of these is used for something different and some might be needed in specific situations depending on what you are working with.

var

If your variable is declared outside of a function, it is **globally scoped**, meaning it is accessible anywhere in your code. Typically, var is function scoped and if declared in a function, it can only be accessed inside of the function. var can be reassigned AND re-declared:

```
// var
var x = "a";
var x = 1; // it's ok to redeclare with var
x = 5; // and it's ok to reassign with var
```

you might be thinking that this leads to more flexibility within your codebase, but the truth is, it can lead to issues. Since var can be redeclared and reassigned, there is a possibility you will completely forget you declared a certain variable, and accidentally override it later.

let

With let, variables can be accessed only within the scope of the block that they live in. Where a block is anything surrounded by a pair of braces (e.g. For loops, if statements, etc). let is like var in that it can be redefined, **but not redeclared**. This helps with the overwriting issue I mentioned with the var keyword. You can re-assign a let variable as many times as you would like, but the original declaration will always remain the same.

```
// let
let x = "a";
let x = 1; // not ok because you can't redeclare with let
x = 5; // but it's ok to reassign a variable declared with let
```

const

const variables are not able to be re-assigned or re-declared. When choosing a keyword for a variable that will NEVER change throughout your code, go with const. This will help increase readability

```
// const
const x = 1;
const x = "a"; // it's not ok to redeclare with const
x = "a"; // it's not ok to reassign with const
```

Conclusion: Avoid using var. If you have a variable that will never change, use const. Otherwise, use let

Now onto today's lab and case material….

A Partial REST API

Earlier we set up some database access routines, and we have seen how to set up a route with the express web framework. The last lab we combined both techniques and set up a single database enabled route. Today we will set up a small CRUD based REST API.

1. Create a new folder called **week5\class1** and copy the .env, app.js and config.js files from week4\class2 into it

2. Ensure the new .env has a COLLECTION variable set to **users**

3. Before we get to the API code, we need to ensure we're able to **parse JSON** when we **POST/PUT** data to the server. To ensure this, we need to update **app.js**, place this code prior to any routing middleware code

```
app.use(express.urlencoded({extended: true}));
app.use(express.json()) // To parse the incoming requests with JSON payloads
```

4. Copy the **db_routines.js** from week4\class1 into the week5**class1** folder

5. Add 2 new methods to the new **db_routines.js** with the following code and don't forget to update the export statement:

```
const updateOne = (db, coll, criteria, projection) =>
  db.collection(coll).findOneAndUpdate(criteria, { $set: projection}, {rawResult: true} );

const deleteOne = (db, coll, criteria) => db.collection(coll).deleteOne(criteria);
```

6. Add a new JavaScript file called **user_routes.js** with the following starting code:

```
import * as dbRtns from "./db_routines.js";
import * as cfg from "./config.js";
import { Router } from "express";

const user_router = Router();

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

export default user_router;
```

7. Return to **app.js** and update the import for routing with this:

```
import user_router from "./user_routes.js";
```

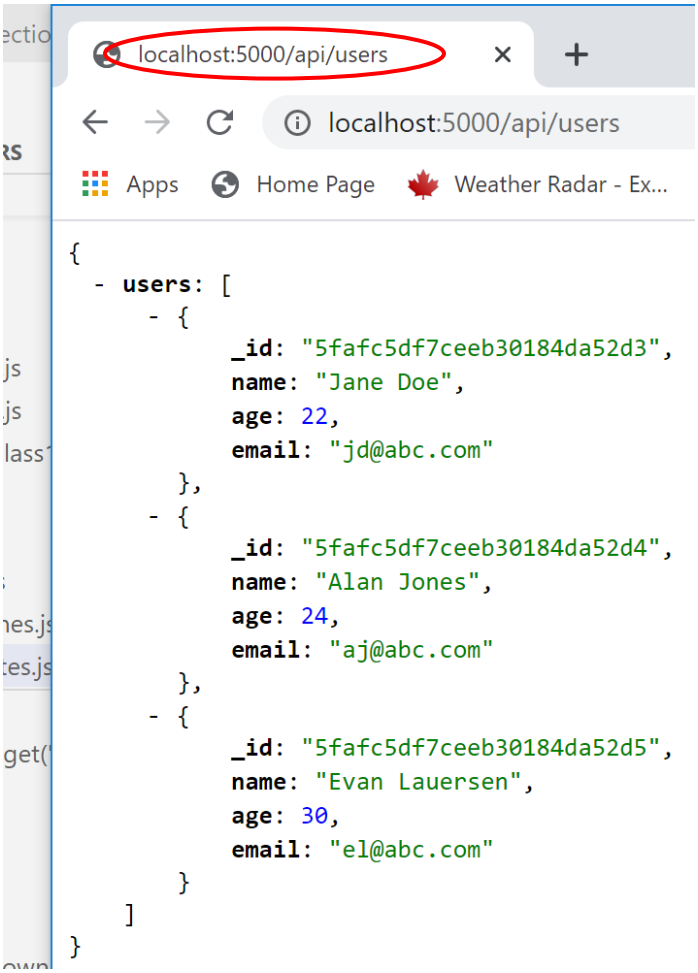
8. Then setup the base route URL mapping with this line of code, again after the parser code we looked at earlier:

```
app.use("/api/users", user_router);
```

9. Start the app (I used the optional **--watch** switch):

```
PS E:\winter2023\info3139\programming\nodeexercises\week5\class1> node --watch app
(node:13780) ExperimentalWarning: Watch mode is an experimental feature. This feature could change at any time
(Use `node --trace-warnings ...` to show where the warning was created)
listening on port 5000
```

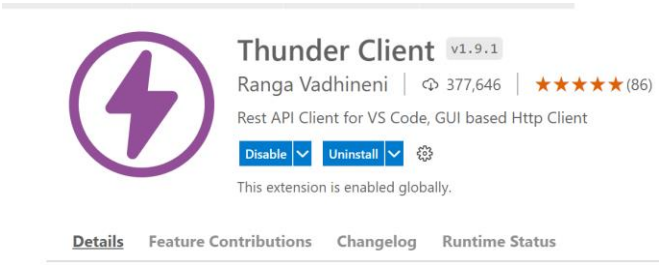
10. Test the get all functionality out by just pointing a browser to **localhost:5000/api/users** (note your data may look different as I'm using a chrome extension called JSONView to format the JSON):



Lab 8 – 2.5%

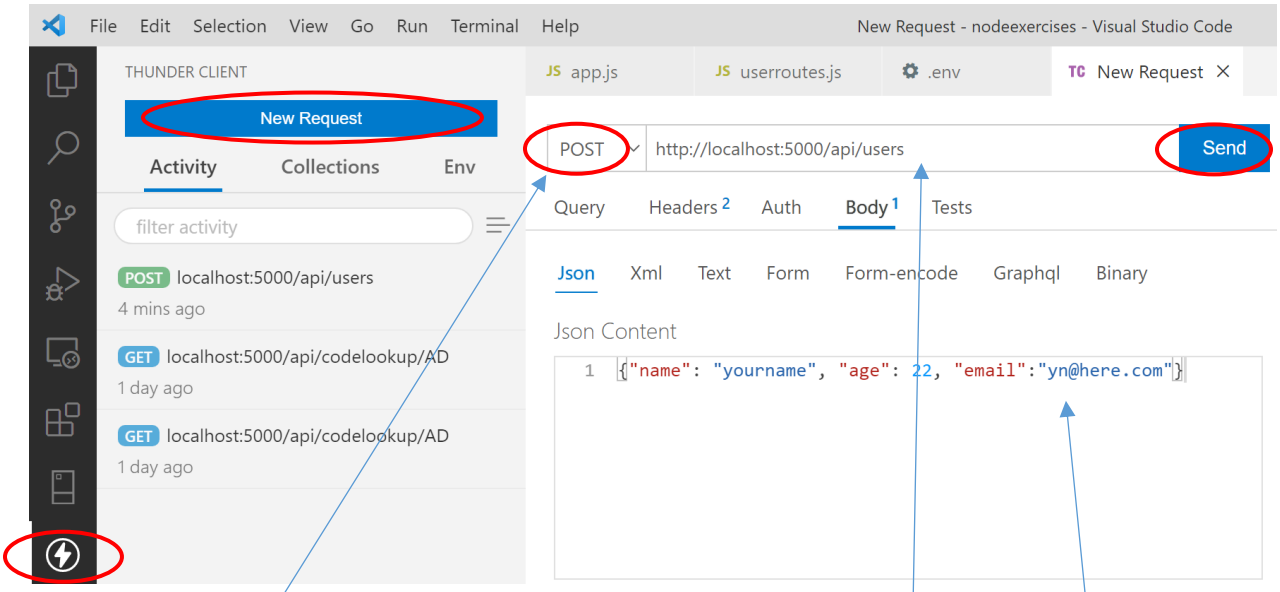
Part A - (2%)

- Download and install a VSCode extension called **Thunder Client**:



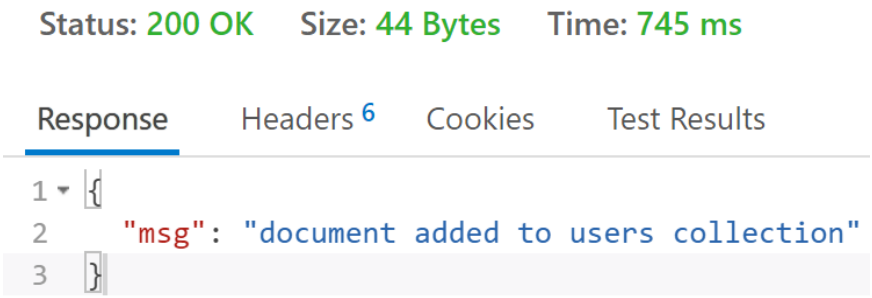
Using the **/api/users** route functionality as a template, add the following 4 routes to create a more robust REST framework for the database's user documents:

- Add a POST method to **user_routes.js** to accept raw JSON from a client (via a HTTP POST) and subsequently add the user JSON to the users collection using the db_routines.js addOne method from last week
- To test the POST operation, use the new Thunder Client extension with the following settings:



  - Set the HTTP Method to **POST**
  - Set the request URL to **http://localhost:5000/api/users**
  - Add the following JSON data in the **body tab** (yourname = your actual name): `{"name": "yourname", "age": 22, "email":"yn@here.com"}`

Press the **SEND** button and if all is well you should see the following:

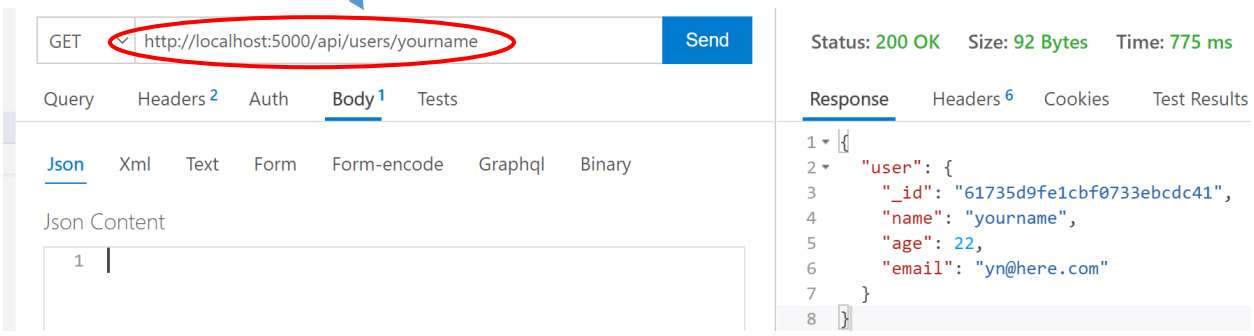


- Confirm the new entry in the database by viewing it on Atlas (it should be the last document in the collection):

- Next, add a **GET** method to user_routes.js so you can retrieve an individual user by name using the db_routines.findOne method.

- Test the route with Thunder Client by switching the method to GET and using a URL like **localhost:5000/api/users/yourname.** Remember, to accept a parameter on a route, you would code something like:

```
user_router.get("/:name", async (req, res) => {
  let name = { name: req.params.name };
```

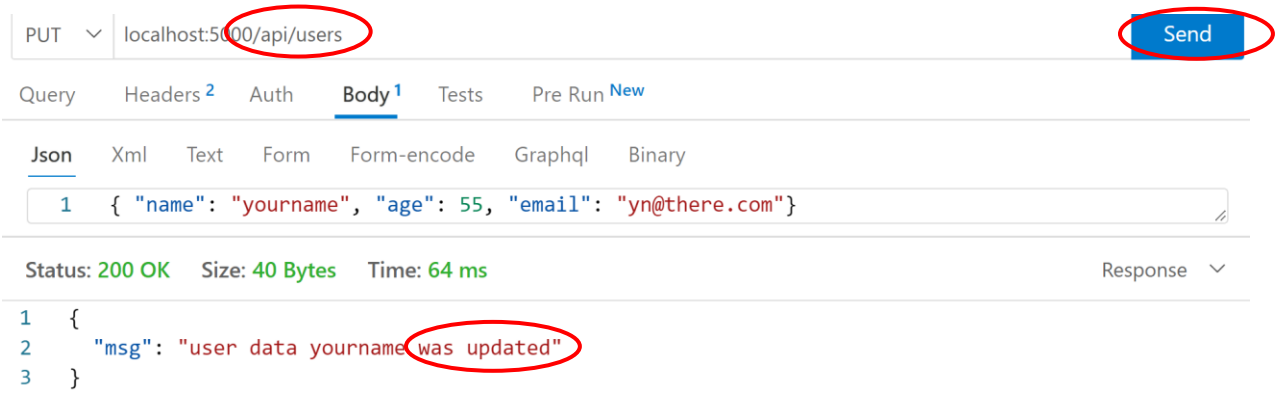


- Add a 3rd (update) route in the **user_routes.js** module to provide a way to update an existing user using HTTP PUT.
    - Works like the POST method, so you test the results like this to see if the update worked or not:

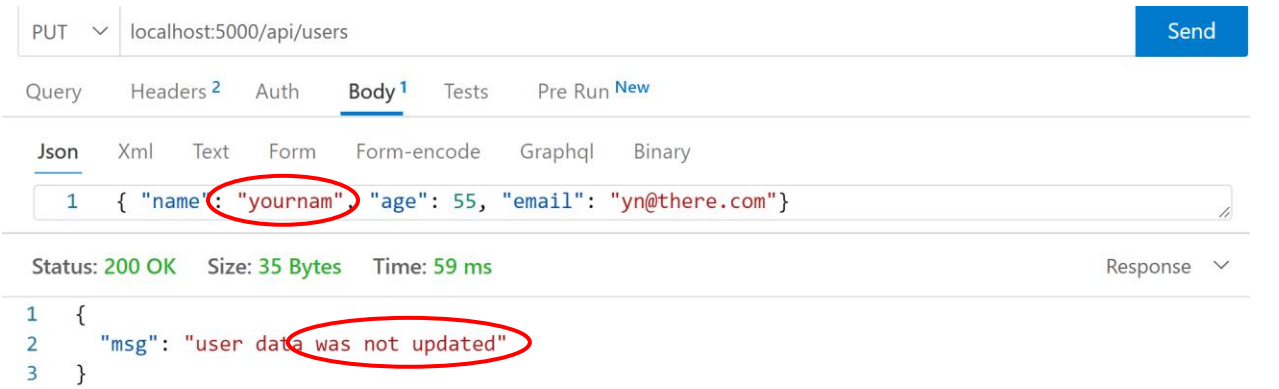
```
let updateResults = await dbRtns.updateOne(…
…
updateResults.lastErrorObject.updatedExisting
    ? (msg = `user data ${updateResults.value.name} was updated`)
    : (msg = `user data was not updated`);
res.status(200).send({ msg: msg });
..
```

- Change the user's email for the user you just added
- Then call the db_routines.js with criteria syntax similar to (assumes user = req.body contents): `{name: user.name}` and projection syntax like: `{age: user.age, email: user.email}`
- Test the **PUT** by using Thunder Client and just change the request data so the properties are different than they were for the POST, something like:

```
{"name": "yourname", "age": 24, "email":"yn@there.ca"}
```



- Then try and update a name that doesn't exist (the e was dropped in the example below:

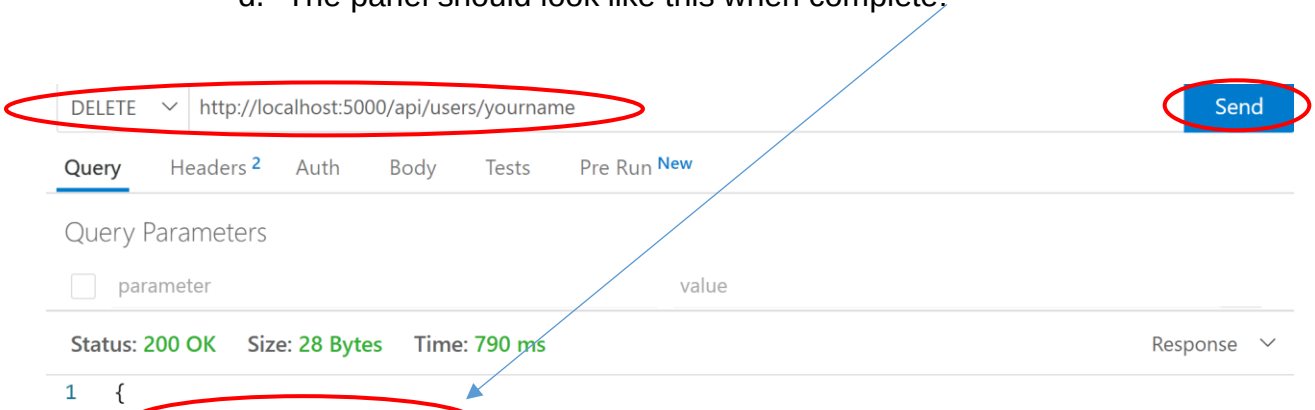


Finally, add a DELETE method to user_routes.js to test HTTP DELETE functionality (calls new deleteOne method) by using the Thunder Client..

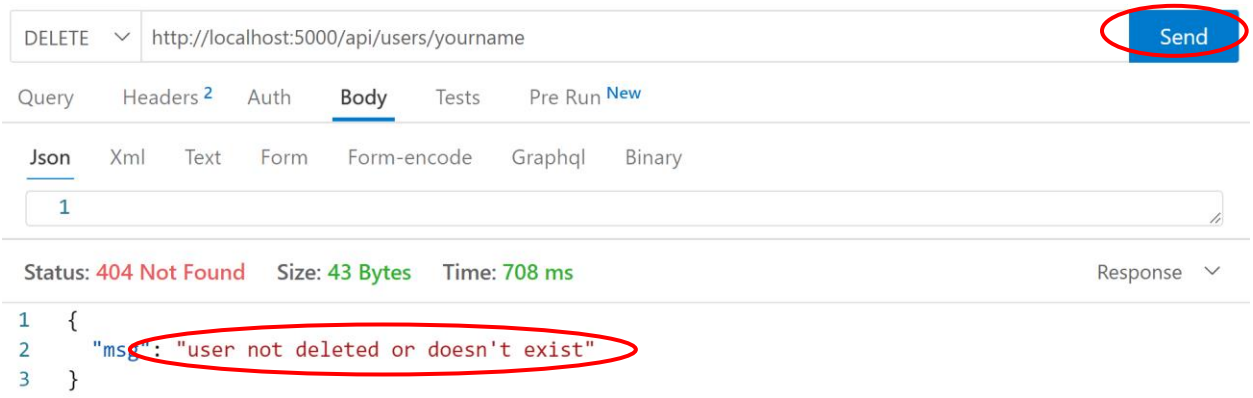
- Options this time are:
  a. Set the HTTP Method to **DELETE**
  b. Set the URL to **http://localhost:5000/api/users/yourname**
  c. You can test the results buy using the **deletedCount** property e.g.:

```
deleteResults.deletedCount === 1
    ? ...
```

d. The panel should look like this when complete:



e. Then press the Send button again (should result in an error):



In summary, your user_routes.js ends up having an API that should look like this:

```
 7  // define a default route to retrieve all users
 8  user_router.get("/", async (req, res) => {···
17  });
18
19  // define a get route with a name parameter to retrieve 1 user
20  user_router.get("/:name", async (req, res) => {···
32  });
33
34  // define a post route to add a user, ensure body-parser has been installed
35  user_router.post("/", async (req, res) => {···
57  });
58
59  // define a put route to update a user, ensure body-parser has been installed
60  user_router.put("/", async (req, res) => {···
94  });
95
96  // define a delete route to remove a user based on name
97  user_router.delete("/:name", async (req, res) => {···
119 });
```

Submit a single .zip file containing:

1. A short .mp4 video that does the following:
   - Perform the GET default route to return all users in your users collection
   - Perform a POST from Thunder Client, make sure the Http method and path **are visible**, and you are **using your actual name** in the text entry, also make sure the results panel is visible after the Send button is pressed.
   - GET results with api/users/**youractualname** as the path, also make sure the results panel is visible after the Send button is pressed.
   - GET results with api/users/**andanamethatdoesnotexist**, also make sure the results panel is visible after the Send button is pressed so the error is present
   - PUT results from Thunder Client, again make sure the Http Method and path **are visible**, make sure you change some data and also make sure the results panel is visible after the Send button is pressed.
   - GET results with api/users/**youractualname** as the path, also make sure the results panel is visible after the Send button is pressed so you can see the updated data in the results pane
   - Do another PUT with a user that does not exist, be sure to show the resulting error
   - DELETE results from Thunder Client, again make sure the Http Method and path and your name **are visible**, also make sure the results panel is visible after the Send button is pressed.
   - DELETE the same user one more time and be sure to capture the resulting error

2. The source code for **user_routes.js**

Part B - (.5%) - Lab 8 Theory Quiz on FOL - 10 questions

Review

1. What code do we place in our app.js to configure the middleware so we are able to POST/PUT JSON data?
2. What 2 methods were added to db_routines to facilitate the REST api
3. When sending the response back from the POST/PUT/DELETE routes what property(ies) can I use to confirm the add/update/delete worked?
4. What URL did we use to test the get route that returns all users?
5. What URL did we use to test the get route that returns a single user?
6. What extension did we install to test the POST, PUT, and DELETE routes?
7. Describe the 2 operations that occur during the db routines update method?
8. What is the argument called that we pass to the db routine delete method?

Homework Reading:

1. https://techformist.com/posts/2019/2019-12-25-fastify-great-plugins/
2. https://pawelgrzybek.com/from-express-to-fastify-in-node-js/
3. https://www.freecodecamp.org/news/a-beginners-guide-to-graphql-60e43b0a41f5/ just down to the section "*A Totally Original and Not Overused Code Example*" and note:
    - What technology it competes with
    - What advantages it offers
    - 5 terms you need to understand – Query, Mutation, Schema, Resolver, Type