BACKEND WEB DEVELOPMENT (PERN STACK)

Table of Contents

Tuesday, January 27, 2026

1. Project setup (backend part): Run npm init -y (npm: node package manager)(-y: just answers few questions with yes instead of you doing it manually) to create package json file that acts like the identity for the project, it has the project name and dependencies.
2. We will work with express js framework, so run npm install express(or npm i express), and notice after that it got added to the dependency part in the package.json file.
3. Create a file 'server.js' in the backend folder.
4. To allow imports you must add *"type"*: "module", to the package.json file right after the main variable(remove whatever type you had before)
5. You must choose a port for your application(http for example uses 80 or 8080)
6. Don't forget to add node-modules to .gitignore when you make the repository cause it contains third-part packages(dependencies) and they can be autogenerated when someone clones the repository.
7. Final version of server.js in which you create an object from express(call it app)and make the first route then run the project

server.js | package.json

server.js > ...

```
// Import Express, you could've used keyword "require" if you were using CommonJS
import express from "express"

// Create an Express application
const app = express();

// Define a route for the root URL
// get function takes two arguments: route path and a callback function
app.get("/", (req, res) => {
  res.send("Hello, World!");
})

// Start the server
// listen function takes two arguments: port number and a callback function
app.listen(3000, () => {
  console.log("Server is running on http://localhost:3000");
})
```

8. In the package.json in the script part, replace test by *"dev"*: "node server.js" (so we can run npm run dev)
9. To run the setup, npm run dev in the terminal.
10. Without nodemon, if you edit the route send parameter, the browser won't load the new version, it will rather require you to stop the server then turn it on again to take the update. Which is bad and not responsive enough, thus we download a tool "npm install nodemon". Then in package.json edit the "dev" line again and write nodemon(node monitor) instead of node. Run the server again and :3

server.js package.json

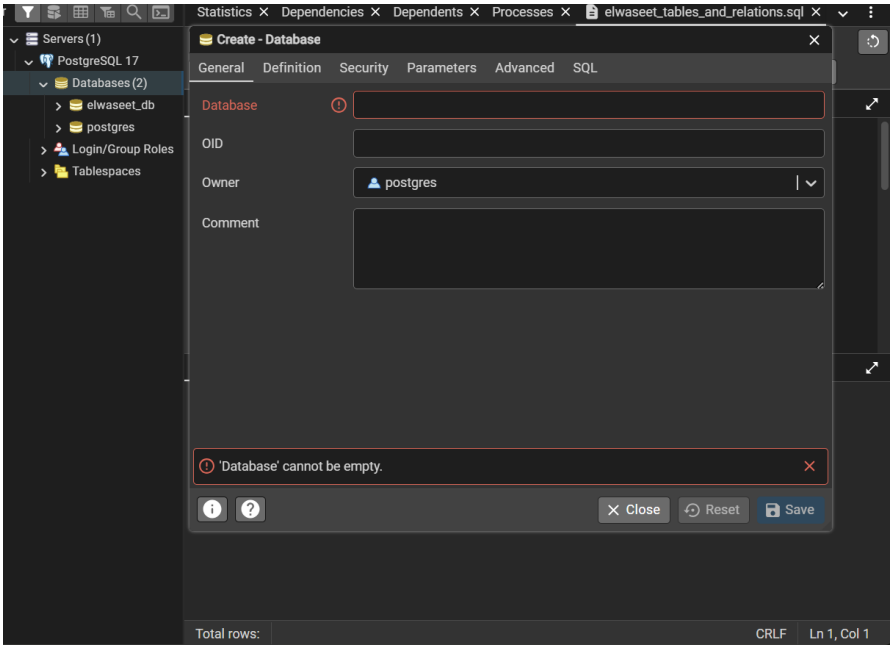
package.json > ...

```
{
  "name": "backend",
  "version": "1.0.0",
  "main": "index.js",
  "type": "module",
  Debug
  "scripts": {
    "dev": "nodemon server.js"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "description": "",
  "dependencies": {
    "express": "^5.2.1",
    "nodemon": "^3.1.11"
  }
}
```

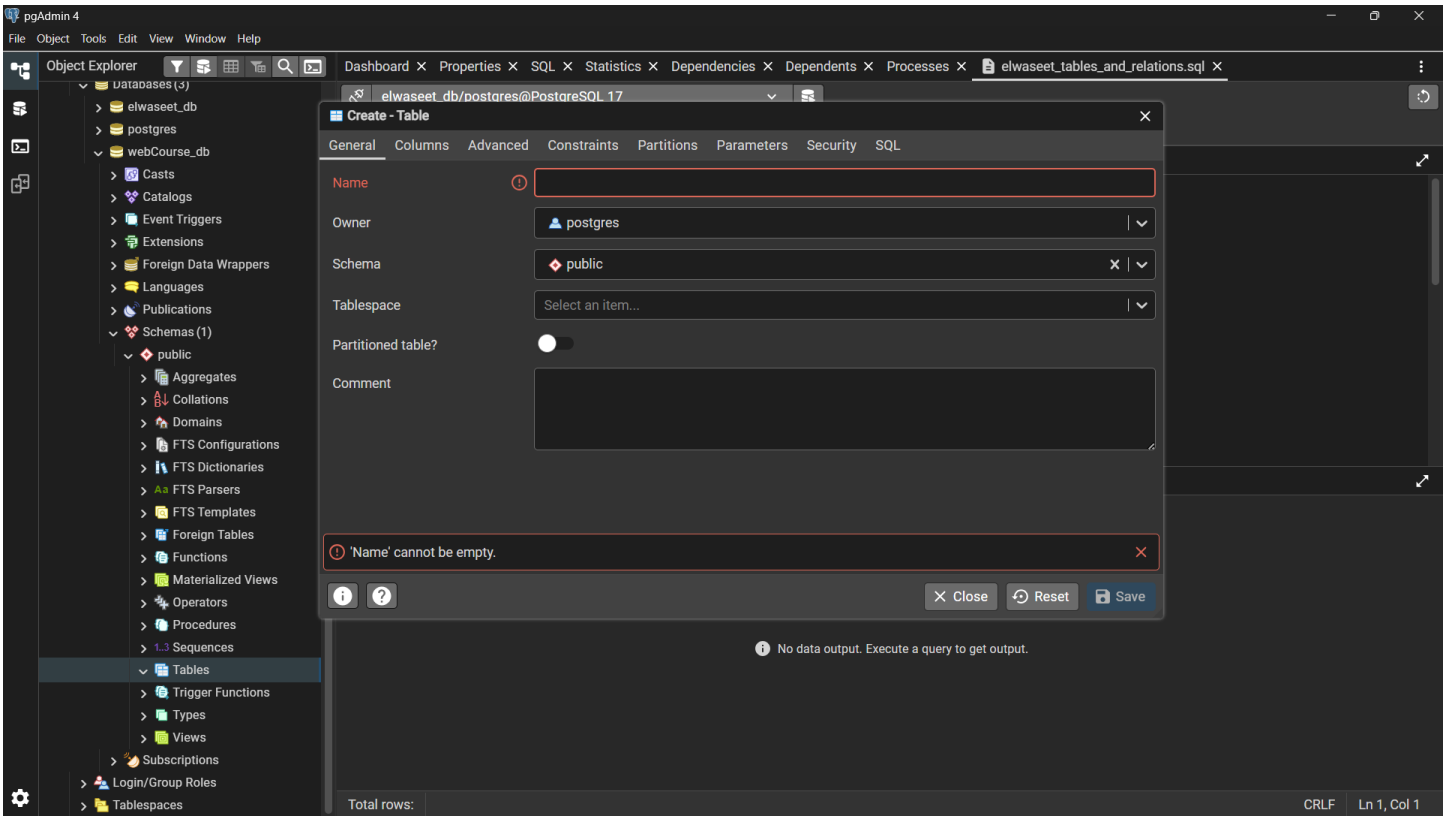
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS C:\FakeDesktop\Web Development  Course\backend> npm i nodemon
  run `npm fund` for details

found 0 vulnerabilities
PS C:\FakeDesktop\Web Development  Course\backend> npm run dev

> backend@1.0.0 dev
> nodemon server.js

[nodemon] 3.1.11
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,cjs,json
[nodemon] starting `node server.js`
Server is running on http://localhost:3000
```

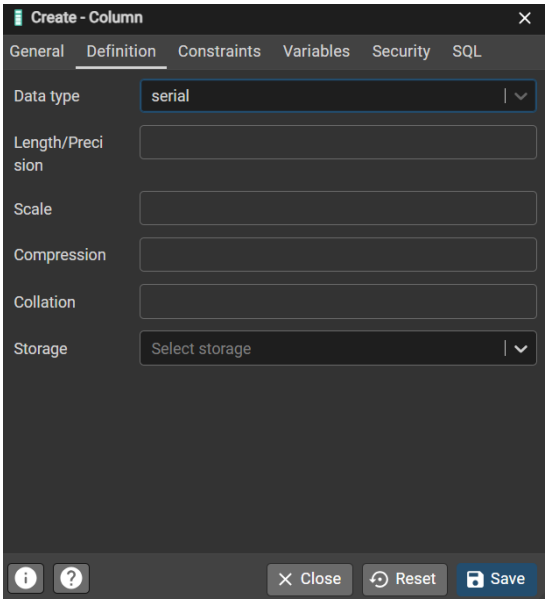
11. Last dependency to download is related to postgresql, using “npm install pg”
12. To create the db, enter postgresql, then right click on databases and create your db, in our case we called it webCourse_db.
13. >schemas>public>tables



14. Option 1: creating manually



15. To make a column autoincrement, you have to choose **serial** column definition. (right click on the table and create column then add info)



16. Option 2: by code
17. Right click on table and choose query tool, then start writing your schema.

```
create table vendors(
-- serial is like auto increment
vendorId serial primary key,
vendorName varchar(100),
vendorIsLocal boolean,
-- you could've used float. numeric(10,2) means there will be 8 numbers then comma then 2 numbers in decimal part
vendorCost numeric(10,2),
-- DOB: date of birth
vendorDOB date
)
```

18. CRUD operations: Create, Read, Update, Delete
19. Insert a user to the table(boolean is true and false not 0,1)

```
-- Create
insert into vendors (vendorname, vendorislocal, vendorcost, vendordob)
values ('lili', true, 12.12, '2004-2-1');

-- Read
select * from vendors;

-- Update
update vendors set vendorname='meow' where vendorname='lili';

-- Delete
delete from vendors where vendorid=1;
```

20. You can see all the table content manually, by right click on the table, view all rows.
21. Remember to always refresh tables to see the change.
22. Above you can view all 4 crud operation on the db.

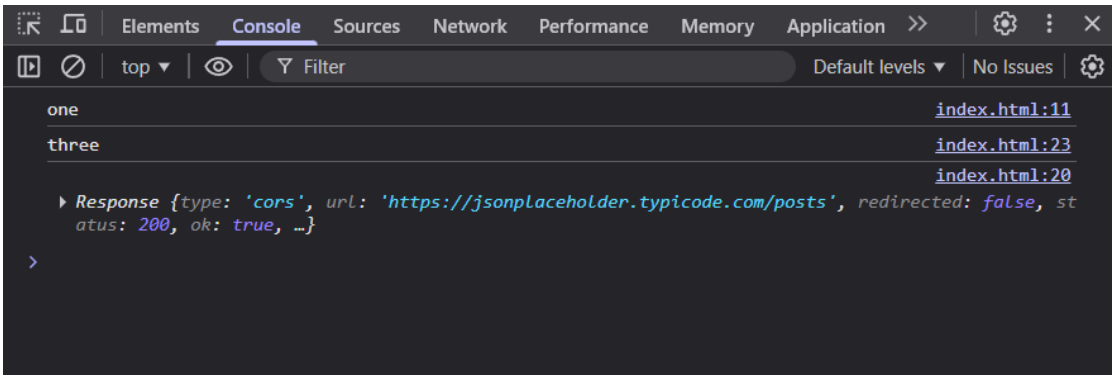
Thursday, January 29, 2026

1. Placeholder JSON is a website that offers free fake data that you can use.

https://jsonplaceholder.typicode.com/posts

```
<script>
    // log to print
    console.log("one")

    // function to print two
    // function printTwo() {
    //     console.log("two")
    // }

    // async function is used to handle asynchronous operations
    // await makes JavaScript wait until that promise settles and returns its result
    async function load() {
        const a = await fetch("https://jsonplaceholder.typicode.com/posts");
        console.log(a)
    }
    load()
    console.log("three")

</script>
```

2. To see the JS code you have to open the index.html page in the browser >> inspect >> choose concole.



3.

```
// normal functions are sync and block the code execution until they are complete
// async function is used to handle asynchronous operations, you can't use await and call api without it
// await makes JavaScript wait until that promise settles and returns its result
async function load() {
    const a = await fetch("https://jsonplaceholder.typicode.com/posts");
    // to see the data fetched from the API
    const b = await a.json();
    console.log(b);
    // the server response is stored in 'a'
    console.log(a);
}
load()
console.log("three")
```

4. Key points:
    a. Async functions **do not run "on their own"** ; they return a Promise and yield control at await.
    b. Async functions **do not block the main thread**; only synchronous code blocks.
    c. await pauses **that function**, not the entire script.
    d. You *can* call APIs without async/await; async/await is just cleaner syntax over .then().
5. Scenario:
    a. console.log("one") runs immediately → prints **one**
    b. function load() is called
    c. fetch(...) starts **asynchronously**
    d. execution **pauses only inside load**, not the whole script
    e. console.log("three") runs immediately → prints **three**
    f. When the fetch promise resolves:
    g. await a.json() resolves
    h. console.log(b) prints the fetched data
    i. console.log(a) prints the response object
6. Back to backend and db, create db.js in the backend folder.

```
import {Pool} from 'pg';

// Create a new connection pool to the PostgreSQL database
// POOL takes 4 parameters: host, user, password, database, port
const pool = new Pool({
    host:"localhost",
    user:"postgres",
    password:"admin",
    database:"webCourse_db",
    port:5432
});

export default pool;
```

7. Go to server.js to check the db connection, first import import pool from "./db.js";
8. Then write a function to test it and make sure to call it in the app.listen

```
// Start the server
// listen function takes two arguments: port number and a callback function
app.listen(3000, () => {
  console.log("Server is running on http://localhost:3000");
  // Call the function to check database connection when server starts
  checkDatabaseConnection();
})

// Function to check database connection
Qodo: Test this function
async function checkDatabaseConnection() {
  try{
    const a = await pool.query("select 1")
  console.log("connection is established.")
  }catch(err){
    console.error("Database connection failed:", err);
  }
}
```

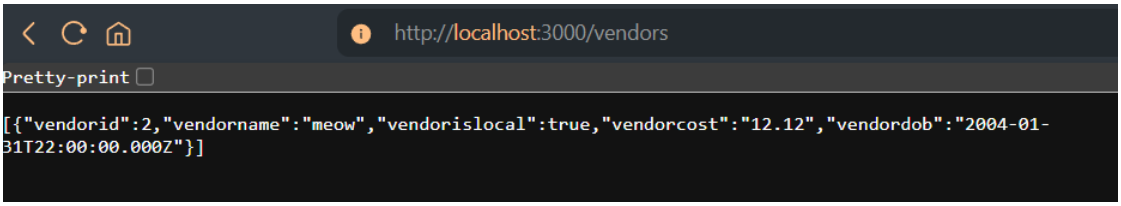
9. Expected result in the cmd after you run npm run dev:

```
[nodemon] restarting due to changes...
[nodemon] starting `node server.js`
Server is running on http://localhost:3000
connection is established.
```

10. Moving on to getting data from the database
    a. Write this function in server.js

```
// to call the db use async on the function
app.get("/vendors", async(req, res) => {
  const vendors = await pool.query("SELECT * FROM vendors");
  res.send(vendors.rows);
})
```

    b. Expected result in the browser



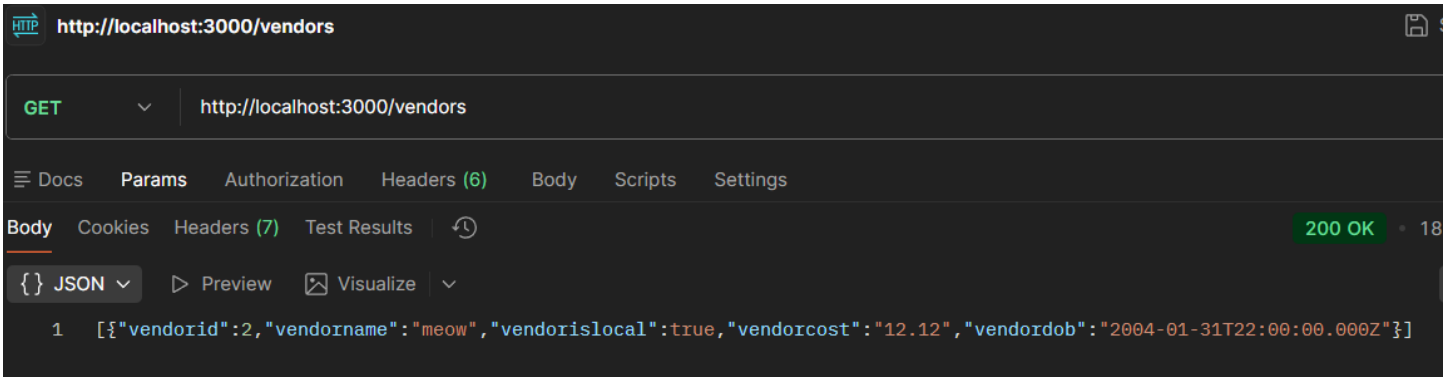
11. To test on postman do the usual.
12. Gui: graphic user interface

Tuesday, February 3, 2026

1. Revision on how to make routes and query the db

```
// to call the db use async on the function
// These are simple CRUD routes for the vendors table,
// the pattern is clear, define a route, async function, query the db, send response
app.get("/vendors", async(req, res) => {
  const vendors = await pool.query("SELECT * FROM vendors");
  res.send(vendors.rows);
})

app.post("/addVendor", async(req, res) => {
  const addVendor = await pool.query("insert into vendors (vendorname, vendorislocal, vendorcost, vendordob) values ('lili', true, 12.12, '2004-2-1');");
  res.send("Vendor added successfully");
})

app.put("/updateVendor", async(req, res) => {
  const updateVendor = await pool.query("update vendors set vendorcost = 15.15 where vendorid = 2;");
  res.send("Vendor updated successfully");
})

app.delete("/deleteVendor", async(req, res) => {
  const deleteVendor = await pool.query("delete from vendors where vendorid = 3;");
  res.send("Vendor deleted successfully");
})
```

2. Trying to test it in postman(notice how the request is empty for now, no json data was sent and processed accordingly)



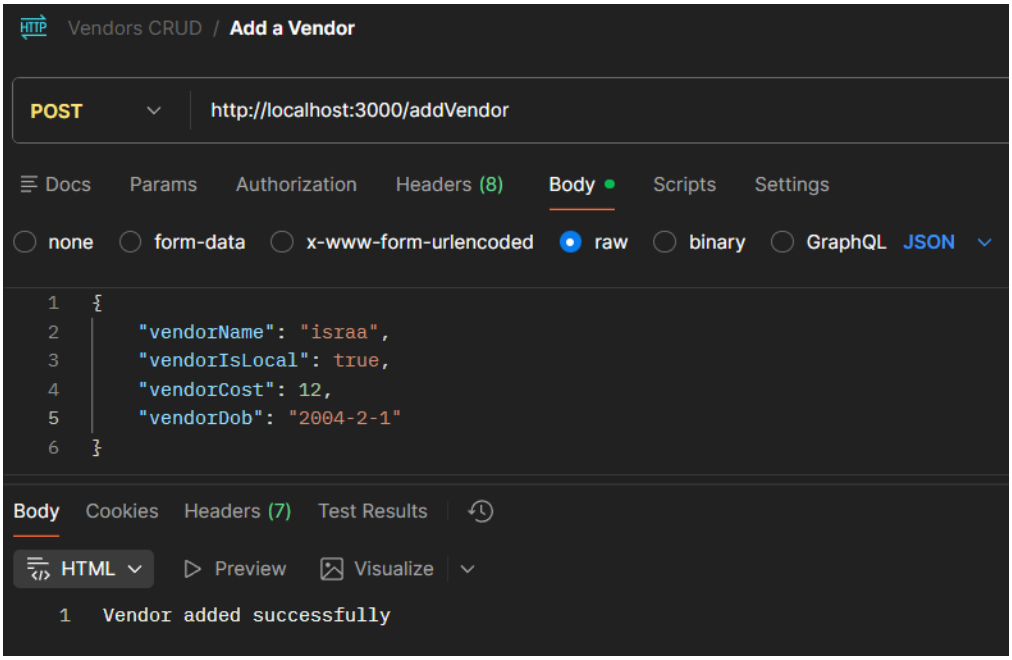
3. To solve the sql injection query issue you must do this, let the values hold indexes $1, ... and then fill them with an array.

```
app.post("/addVendor", async(req, res) => {
  const addVendor = await pool.query("insert into vendors (vendorname, vendorislocal, vendorcost, vendordob) values ($1, $2, $3, $4);", ['lili', true, 12.12, '2004-2-1']);
  res.send("Vendor added successfully");
})
```

4. Now we are gonna prepare the route to accept a JSON request

```
app.post("/addVendor", async(req, res) => {
  const addVendor = await pool.query("insert into vendors (vendorname, vendorislocal, vendorcost, vendordob) values ($1, $2, $3, $4);",
    [req.body.vendorName, req.body.vendorIsLocal, req.body.vendorCost, req.body.vendorDob]); // parameterized query to prevent SQL injection req.body.variableName
    // now we can send a JSON request with the vendor details in the body
  res.send("Vendor added successfully");
})
```

5. Now in postman you have to name the keys same as the ones the request in vs code is expecting
6. Add in server.js app.use(express.json()); // Middleware to parse JSON request bodies
7. View in postman:



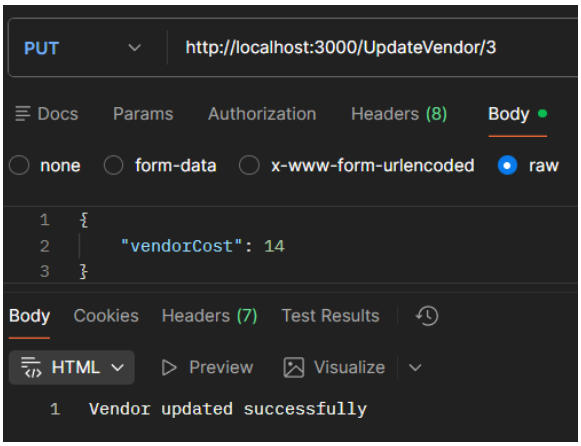
8. We can also simplify the code to achieve a readable code.

```
app.post("/addVendor", async(req, res) => {
  const {vendorName, vendorIsLocal, vendorCost, vendorDob}=req.body; // Destructure the request body to get vendor details
  const addVendor = await pool.query("insert into vendors (vendorname, vendorislocal, vendorcost, vendordob) values ($1, $2, $3, $4);",
    [vendorName, vendorIsLocal, vendorCost, vendorDob]);
  res.send("Vendor added successfully");
})
```

9. Now let's try updating a vendor but sending a specific id in the url, noice how you add it in the url part, and how you no longer type **req.body.id** but rather **req.params.id,** while also sending the vendor cost normally in body

```
app.put("/updateVendor/:id", async(req, res) => {
  const updateVendor = await pool.query("update vendors set vendorcost = $1 where vendorid = $2;", [req.body.vendorCost, req.params.id]);
  res.send("Vendor updated successfully");
})
```

And in postman

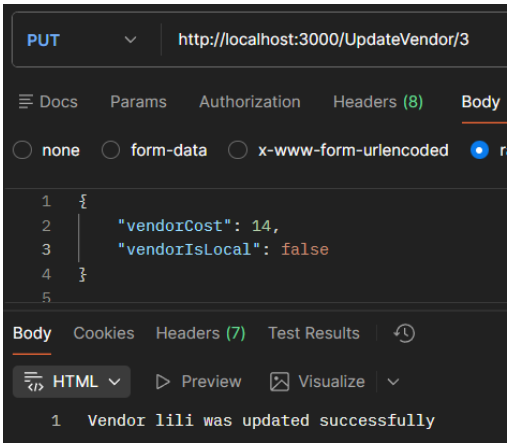


10. The variables in the query part should be same as those in the db however the variable names that are in the array should be same as those in the postman request.

```
app.delete("/deleteVendor/:id", async(req, res) => { // :id is a route parameter
  const deleteVendor = await pool.query("delete from vendors where vendorid = $1;", [req.params.id]);
  res.send("Vendor deleted successfully");
})
```

11. This is how you add the name of the vendor although you didn't get it in body but you are fetching it from the db. We first fetch the name from the db in a query, and then we access the vendor name using ${vendorName.rows[0].vendorname}

```
app.put("/updateVendor/:id", async(req, res) => {
  const updateVendor = await pool.query("update vendors set vendorcost = $1, vendorislocal = $2 where vendorid = $3;",
    [req.body.vendorCost, req.body.vendorIsLocal, req.params.id]);
    const vendorName = await pool.query("select vendorname from vendors where vendorid = $1;", [req.params.id]);
  res.send(`Vendor ${vendorName.rows[0].vendorname} was updated successfully`); // Access vendor name from the query result
})
```



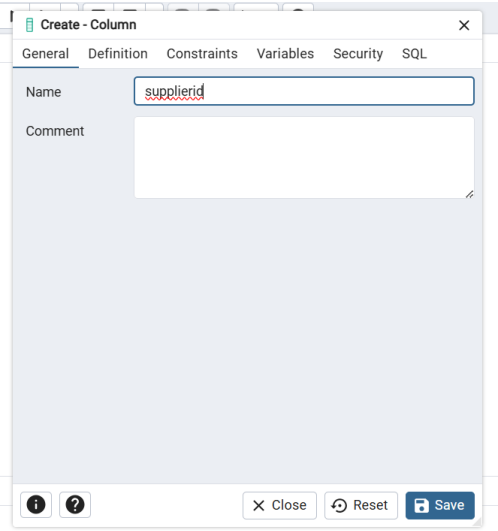
Thursday, February 5, 2026

1. To create a relationship between 2 entities, we need to use the foreign key. (1:1, 1:m, m:m)
2. When you reference a foreign key to another primary key, what matters is that both variables have to have the same type and size (e.g. varchar (20)) but they could have diff name.
3. Example: first create both these tables

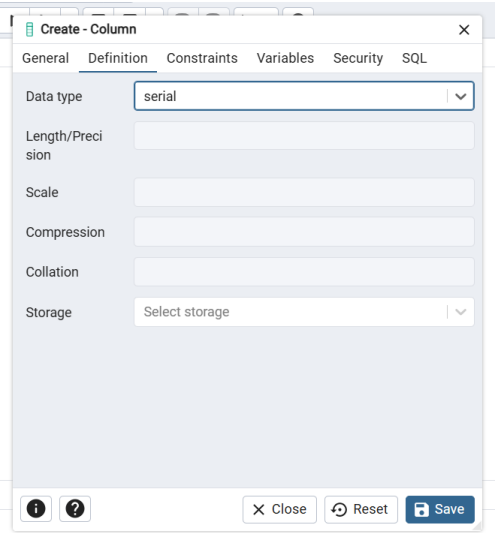
```
create table suppliers(
supplierid serial primary key,
suppliername varchar(20)
)

create table products(
productid serial primary key,
productname varchar(30),
productprice numeric(10,2),
productexist boolean,
productexp date
)
```

4. To connect these tables we have 2 options: (the relationship is 1 to many relationship, a supplier could have 1 or more products, so the fk will be in the products table)
    a. Option 1:

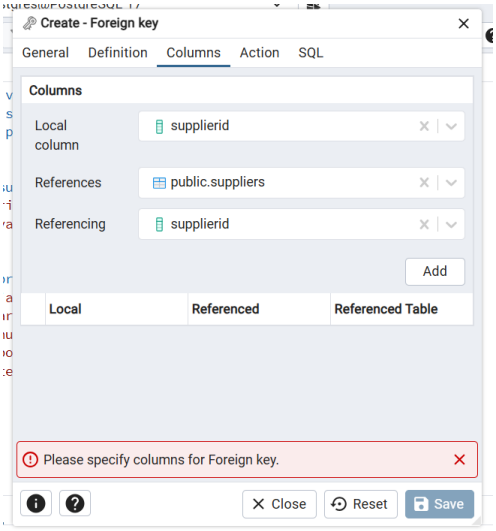
Right click on he products table, create a column, and call it supplier id, we will let this column become a foreign key in the next step.



Then in the products table go to constraints, right click, create a foreign key, name it.



After it navigate to columns fill in the local column the supplierid column that we just created in the products table, references will hold the supplier table, and the referencing will be the primary key of the suppliers table.



Press add then save.

b. Option 2:

Create the foreign key while creating the table or add a column to an existing table (using alter)

```
drop table products

create table products(
productid serial primary key,
productname varchar(20),
productprice numeric(10,2),
productexist boolean,
productexp date,
supplierid int references suppliers(supplierid) on delete cascade
)
```

5. What is delete on cascade? It means when a certain supplier is deleted, delete it's respective products. For example here we have lilis food supplied by supplier 2, below is the table before deleting the supplier

```
21  insert into suppliers(suppliername) values('lili')
22  insert into products(productname, supplierid) values('lilis food', 2)
```

Data Output | Messages | Notifications

	productid [PK] integer	productname character varying (20)	productprice numeric (10,2)	productexist boolean	productexp date	supplierid integer
1	6	lilis food	[null]	[null]	[null]	2
2	7	lilis food	[null]	[null]	[null]	2
3	8	lilis food	[null]	[null]	[null]	2
4	9	lilis food	[null]	[null]	[null]	2

6. When we delete supplier 2, all his products will be deleted as well, below is the table after deleting the supplier

```
23  delete from suppliers where supplierid=2
```

Data Output | Messages | Notifications

	productid [PK] integer	productname character varying (20)	productprice numeric (10,2)	productexist boolean	productexp date	supplierid integer

7. Now let's make a challenge, try to insert a product using a route on postman by sending a supplier name not id, you have to make a query to find the supplier name and add it accordingly.
    a. First let's try adding a product normally without supplier id

```
app.post("/addProduct", async(req, res) => {
  const {productName, productPrice, productExist, productEXP}=req.body;
  const addProduct = await pool.query("insert into products (productname, productprice, productexist, productexp) values ($1, $2, $3, $4);",
    [productName, productPrice, productExist, productEXP]);
  res.send(`Product added successfully`);
})
```

Vendors CRUD / Add a Product

POST http://localhost:3000/addProduct

Docs Params Auth Headers (8) Body Scripts Settings

raw JSON

```
{
    "productName":"meow",
    "productPrice":12.2,
    "productExist":true,
    "productEXP":"2004-2-1"
}
```

Body 200 OK 178

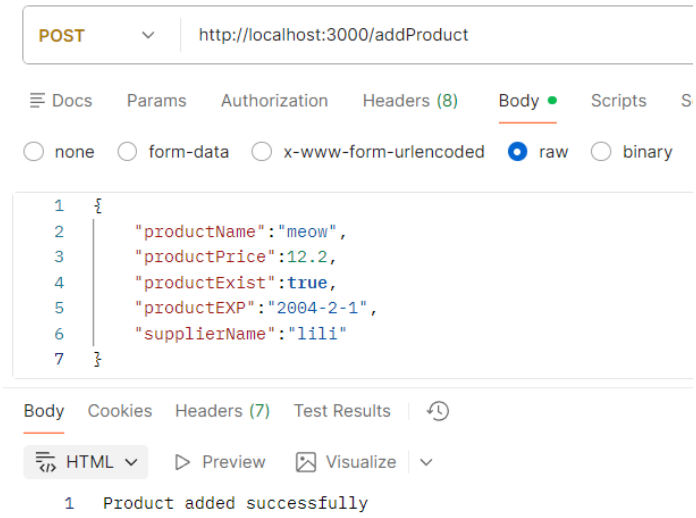
HTML Preview Visualize

```
Product added successfully
```

    b. First I added the supplierName to the array considered in req.body cause I want to get the name in the body request that postman is sending.
        i. Then we create a variable to get the supplier id using a simple sql query, in the query we compare the sapplier name given with what we have in the db and if the name matches, we send the id (this is not optimal for big programs cause what if there's 2 suppliers with the same name, id is the only unique thing)
        ii. Then after we got it, we can add normally to the product table but instead of taking the supplier id from the body, we will take it from supplieridResult variable.

```
app.post("/addProduct", async(req, res) => {
  const {productName, productPrice, productExist, productEXP, supplierName}=req.body;
  const supplieridResult = await pool.query("select supplierid from suppliers where suppliername = $1;", [supplierName]);
  const addProduct = await pool.query("insert into products (productname, productprice, productexist, productexp, supplierid) values ($1, $2, $3, $4, $5);",
    [productName, productPrice, productExist, productEXP, supplieridResult.rows[0].supplierid]);
  res.send(`Product added successfully`);
})
```

Postman response

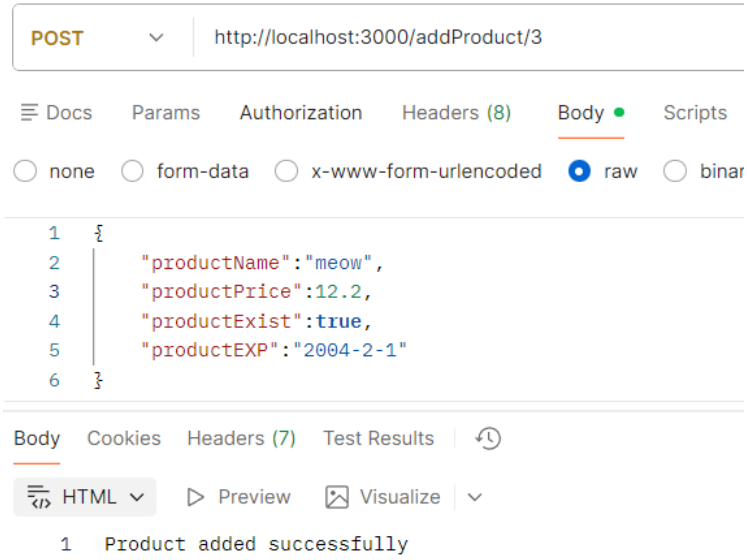


And if you check the db, the product was added successfully

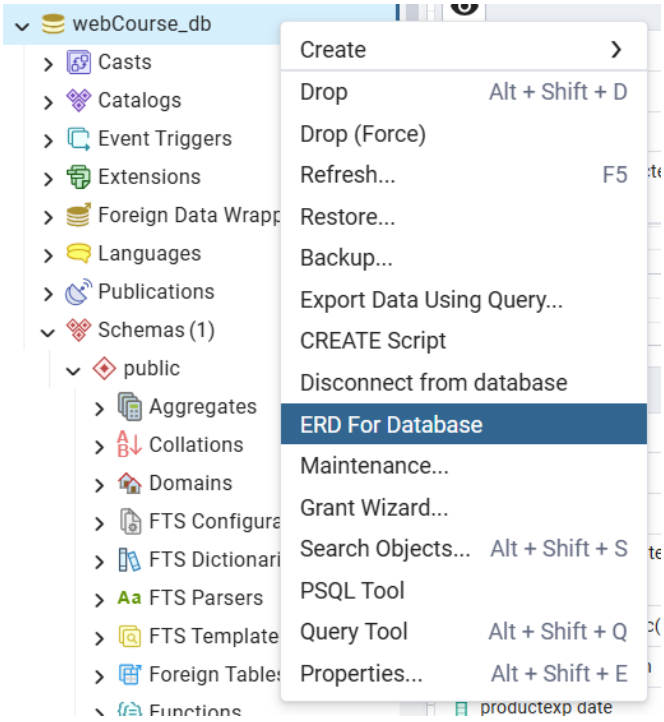
	productid [PK] integer	productname character varying (20)	productprice numeric (10,2)	productexist boolean	productexp date	supplierid integer
1	10	meow	12.20	true	2004-02-01	[null]
2	11	meow	12.20	true	2004-02-01	3

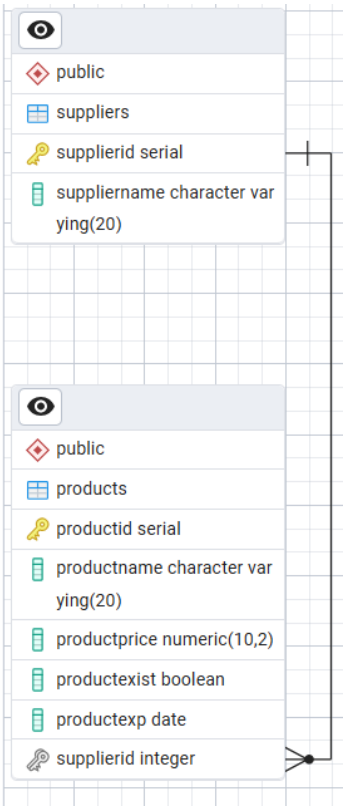
c. Okay what if I want to send the id and do it the easy way? Then send the supplier id in the url and call it using req.params

```
app.post("/addProduct/:id", async(req, res) => {
  const {productName, productPrice, productExist, productEXP}=req.body;
  const addProduct = await pool.query("insert into products (productname, productprice, productexist, productexp, supplierid) values ($1, $2, $3, $4, $5);",
    [productName, productPrice, productExist, productEXP, req.params.id]);
  res.send(`Product added successfully`);
})
```



8. To view to er diagram, right click on your db name, choose erd for databases





9. Then you'll be able to view the relationship between tables
10. Joining both tables on supplier id, and then putting it in a get route, this get route will represent only the products that have a supplier

```
select * from products
join suppliers on
products.supplierid=suppliers.supplierid
```

utput | Messages | Notifications

Showing rows: 1 to 2 | Page

productid integer	productname character varying (20)	productprice numeric (10,2)	productexist boolean	productexp date	supplierid integer	supplierid integer	suppliername character varying (20)
11	meow	12.20	true	2004-02-01	3	3	lili
12	meow	12.20	true	2004-02-01	3	3	lili

11. Now let's put it in a route

Choose only the fields you want to see

```
24  select suppliername, productname, productprice, productexist from products
25  join suppliers on
26  products.supplierid=suppliers.supplierid
```

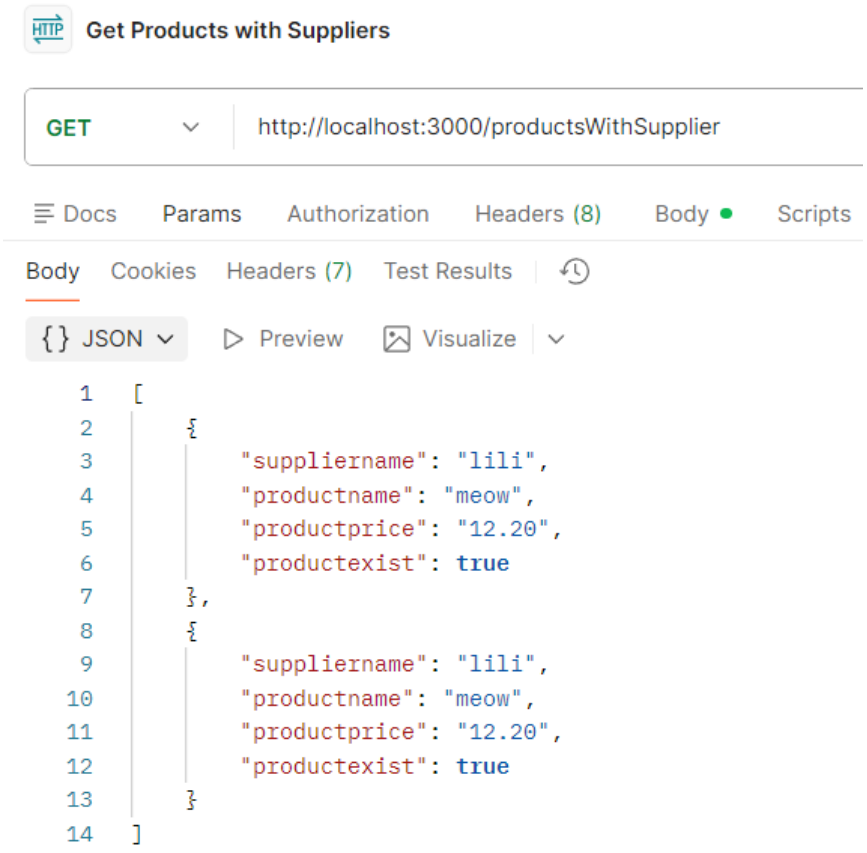
Data Output | Messages | Notifications

	suppliername character varying (20)	productname character varying (20)	productprice numeric (10,2)	productexist boolean
1	lili	meow	12.20	true
2	lili	meow	12.20	true

Copy paste the whole query into pool.query

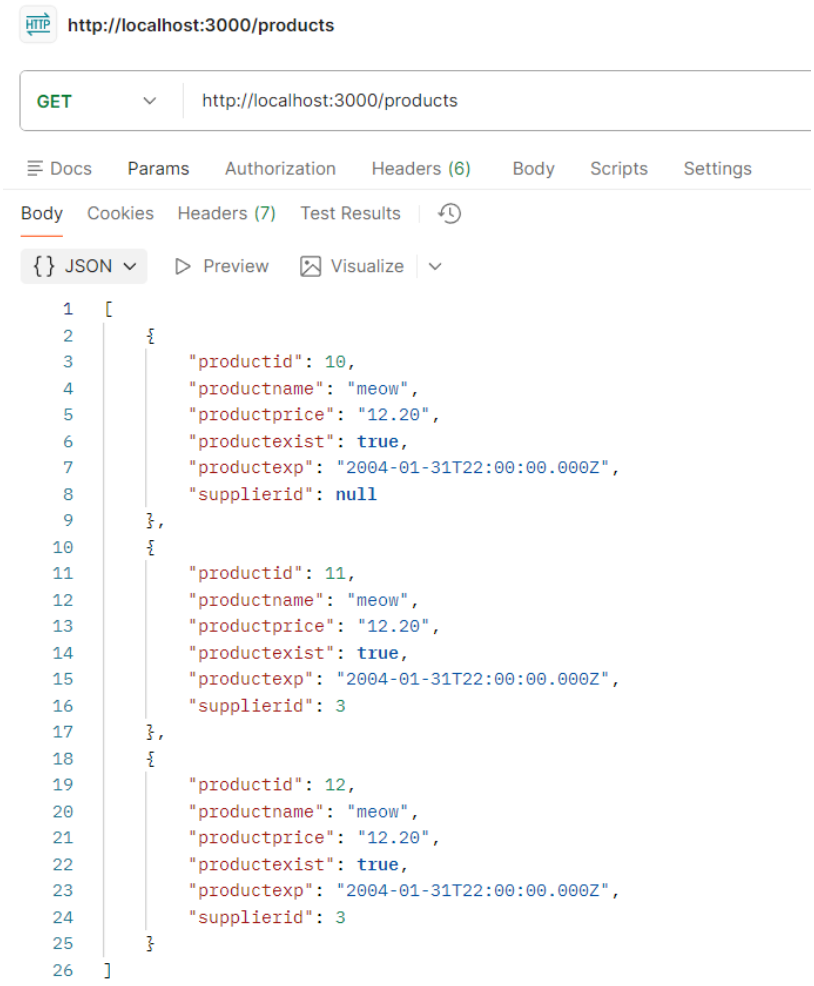
```
app.get("/productsWithSupplier", async(req,res) => {
  const result = await pool.query("select suppliername, productname, productprice, productexist from produc
  res.send(result.rows);
})
```

Result in postman



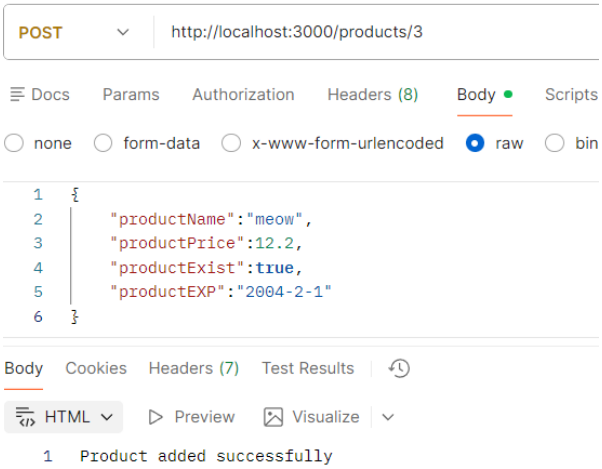
12. It is messy to write everything in server.js, thus we create a folder call it routes, and then create 2 files in it productsRoute, and supplierRoute, to tell the server to use them, type this in server.js, Import the folder : import productRoute from "./routes/productroutes.js"; and then use it app.**use**("/products", productRoute); *// Use the product routes* this means that whenever the server see /products, it will go to productRoute
13. Don't forget to add export default productRoute; at the end of the route file
14. Now that you told the server.js to go to productRoute use "/products", you no longer have to write it in the route in productroutes file => example on a get route (else the url expected will become /products/products)

```js
import express, { Router } from "express"
import pool from "../db.js";

const productRoute = Router(); // router instance
productRoute.use(express.json());


productRoute.get("/", async(req,res)=>{
    const products = await pool.query("SELECT * FROM products");
    res.send(products.rows);
})

export default productRoute;
```



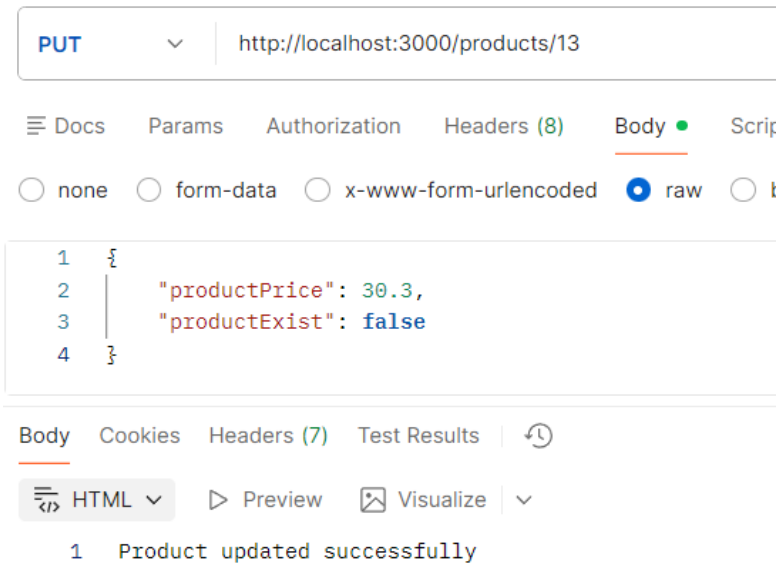
15. The post route of products: inserting a product and giving the supplier id as parameter

```js
productRoute.post("/:id", async(req,res)=>{
    const {productName, productPrice, productExist, productEXP}=req.body;
    const addProduct = await pool.query("insert into products (productname, productprice, productexist, productexp, supplierid) values ($1, $2, $3, $4, $5);",
    [productName, productPrice, productExist, productEXP, req.params.id]);
    res.send(`Product added successfully`);
})
```



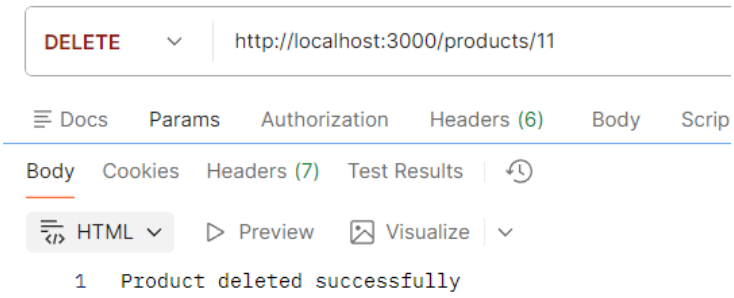
16. Updating a product, by sending the product id in the route as a parameter, and both the price and the existence in json body

```
productRoute.put("/:id", async(req,res)=>{
    const {productPrice, productExist}=req.body;
    const updateProduct = await pool.query("update products set productprice = $1, productexist = $2 where productid = $3;",
        [productPrice, productExist, req.params.id]);
    res.send(`Product updated successfully`);
})
```



17. Deleting a product by sending its id in the route as a param

```
productRoute.delete("/:id", async(req,res)=>{
    const deleteProduct = await pool.query("delete from products where productid = $1;", [req.params.id]);
    res.send(`Product deleted successfully`);
})
```



18. We have seen how we can send one sentence using res.send, you can also send an object res.**send**({products: products.rows, count: products.rows.length, message: "Products retrieved successfully"}); , but it's preferred to change send to json (just change the word send to json) this is how you can send multiple responses

```
productRoute.get("/", async(req,res)=>{
    const products = await pool.query("SELECT * FROM products");
    res.json({products: products.rows, count: products.rows.length, message: "Products retrieved successfully"});
})
```

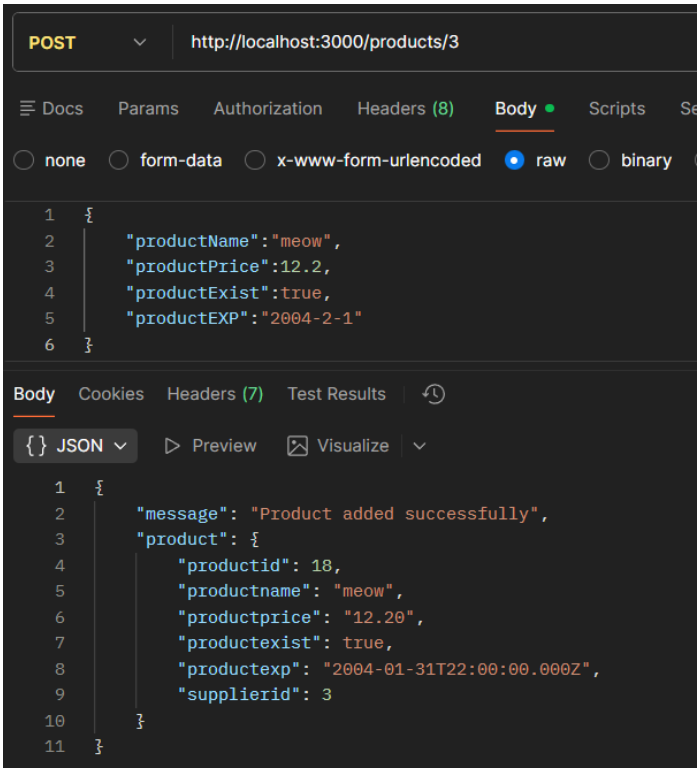
Response:


```
{
    "products": [
        {
            "productid": 10,
            "productname": "meow",
            "productprice": "12.20",
            "productexist": true,
            "productexp": "2004-01-31T22:00:00.000Z",
            "supplierid": null
        },
        {
            "productid": 12,
            "productname": "meow",
            "productprice": "12.20",
            "productexist": true,
            "productexp": "2004-01-31T22:00:00.000Z",
            "supplierid": 3
        },
        {
            "productid": 13,
            "productname": "meow",
            "productprice": "30.30",
            "productexist": false,
            "productexp": "2004-01-31T22:00:00.000Z",
            "supplierid": 3
        }
    ],
    "count": 3,
    "message": "Products retrieved successfully"
}
```


19. It's better when inserting data into the database to return the information that was just inserted, because the database may generate values we didn't explicitly send (like an id), and returning it lets the server immediately use the real saved data. For example, during registration, instead of sending the user back to the login page, the inserted user data can be returned in the response

and used to log them in by default. To do this, we add `returning *` to the INSERT query, and when sending the response we use `addProduct.rows[0]`, which represents the row that was just inserted. This is why returning the inserted information in the response is useful and considered good practice.

```
// returning * to get the inserted row back
productRoute.post("/:id", async(req,res)=>{
    const {productName, productPrice, productExist, productEXP}=req.body;
    const addProduct = await pool.query
    ("insert into products (productname, productprice, productexist, productexp, supplierid) values ($1, $2, $3, $4, $5) returning *;",
    [productName, productPrice, productExist, productEXP, req.params.id]);
    res.json({message: "Product added successfully", product: addProduct.rows[0]});
})
```



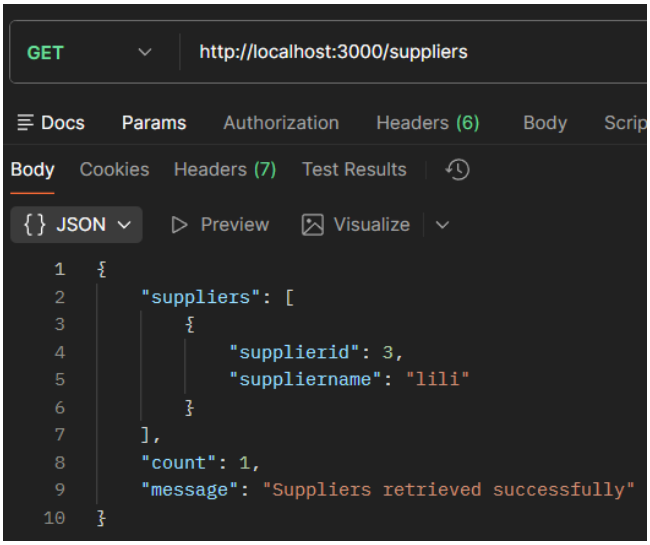
20. Next, we'll create all CRUD supplier routes:
    a. Setup:
        i. server.js:
            1. import supplierRoute from "./routes/supplierroutes.js";
            2. app.use("/suppliers", supplierRoute); // Use the supplier routes
        ii. supplierroutes.js:
            1. import express, { Router } from "express" : why not import express from "express" like in server.js? bc in server.js we need the main express function, but in supplier route or product route files, we need express and Router class, router class connects your route file to the main app. It lets server.js use all the routes defined in your route file, bridging them together so requests are handled properly.
            2. import pool from "../db.js";

3. const supplierRoute = Router(); // router instance
4. supplierRoute.use(express.json());
5. and at the end: export default supplierRoute; (the name supplierRoute is exactly the instance name not the file name)

b. Get all suppliers:

```
supplierRoute.get("/", async(req,res)=>{
    const suppliers=await pool.query("SELECT * FROM suppliers");
    res.json({suppliers: suppliers.rows, count: suppliers.rows.length, message: "Suppliers retrieved successfully"});
})
```

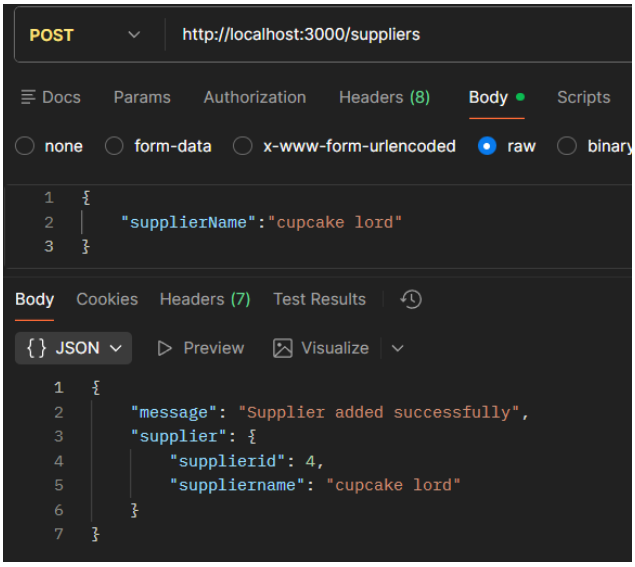
Expected Result



c. Add a supplier, body only has supplier name

```
supplierRoute.post("/", async(req,res)=>{
    const {supplierName}=req.body;
    const addSupplier = await pool.query("insert into suppliers (suppliername) values ($1) returning *;", [supplierName]);
    res.json({message: "Supplier added successfully", supplier: addSupplier.rows[0]});
})
```

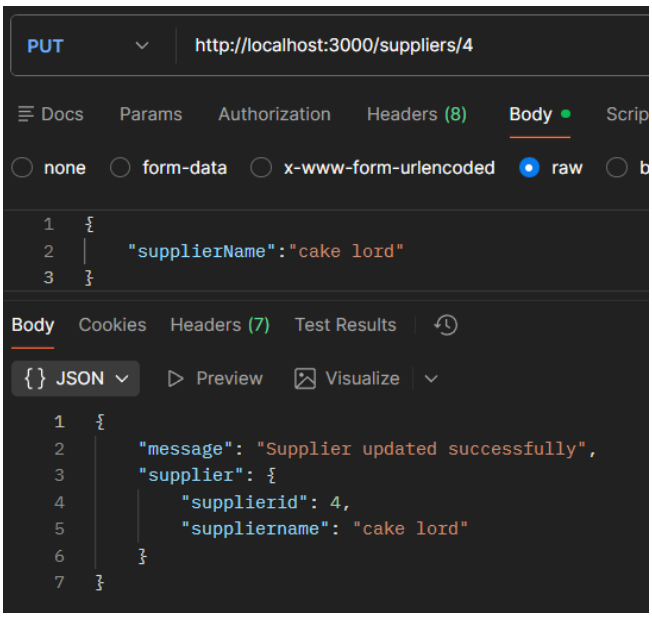
Expected Result:



d. Update a supplier by sending the suppliers id in the route params

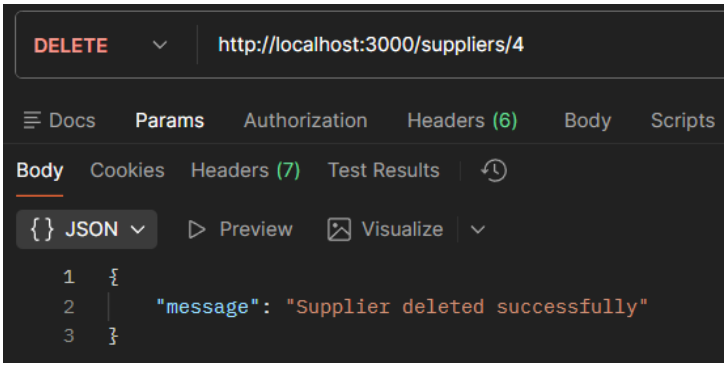
```
supplierRoute.put("/:id", async(req,res)=>{
    const {supplierName}=req.body;
    const updateSupplier = await pool.query("update suppliers set suppliername = $1 where supplierid = $2 returning *;",
        [supplierName, req.params.id]);
    res.json({message: "Supplier updated successfully", supplier: updateSupplier.rows[0]});
})
```

Expected Result:



e. Delete a supplier by sending the suppliers id in the route params

```
supplierRoute.delete("/:id", async(req,res)=>{
    const deleteSupplier = await pool.query("delete from suppliers where supplierid = $1;", [req.params.id]);
    res.json({message: "Supplier deleted successfully"});
})
```

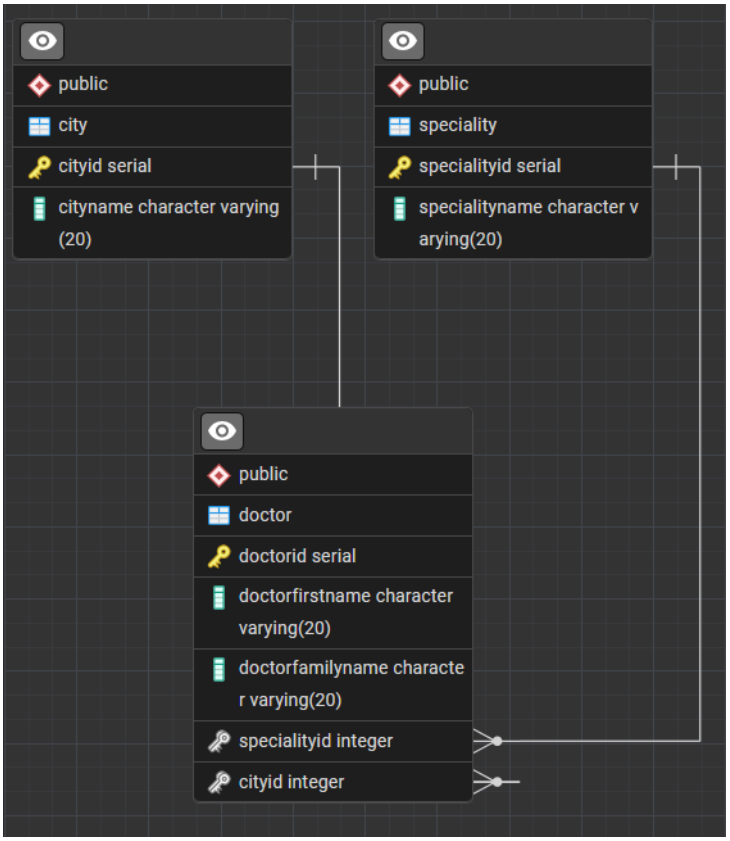


Friday, February 6, 2026

1. Exercise: Doctor lives in one city and only has one speciality.
2. So the relationships are 1 to many, a doctor can have only one specialty, but specialty can have more than one dr, a city can have many drs, but a dr can only be in one city. So both foreign keys are in dr entity.
3. First create the tables

```
create table city(
cityid serial primary key,
cityname varchar(20)
);

create table speciality(
specialityid serial primary key,
specialityname varchar(20)
);

create table doctor(
doctorid serial primary key,
doctorfirstname varchar(20),
doctorfamilyname varchar(20),
specialityid int references speciality(specialityid) on delete cascade,
cityid int references city(cityid) on delete cascade
);
```

View as ER



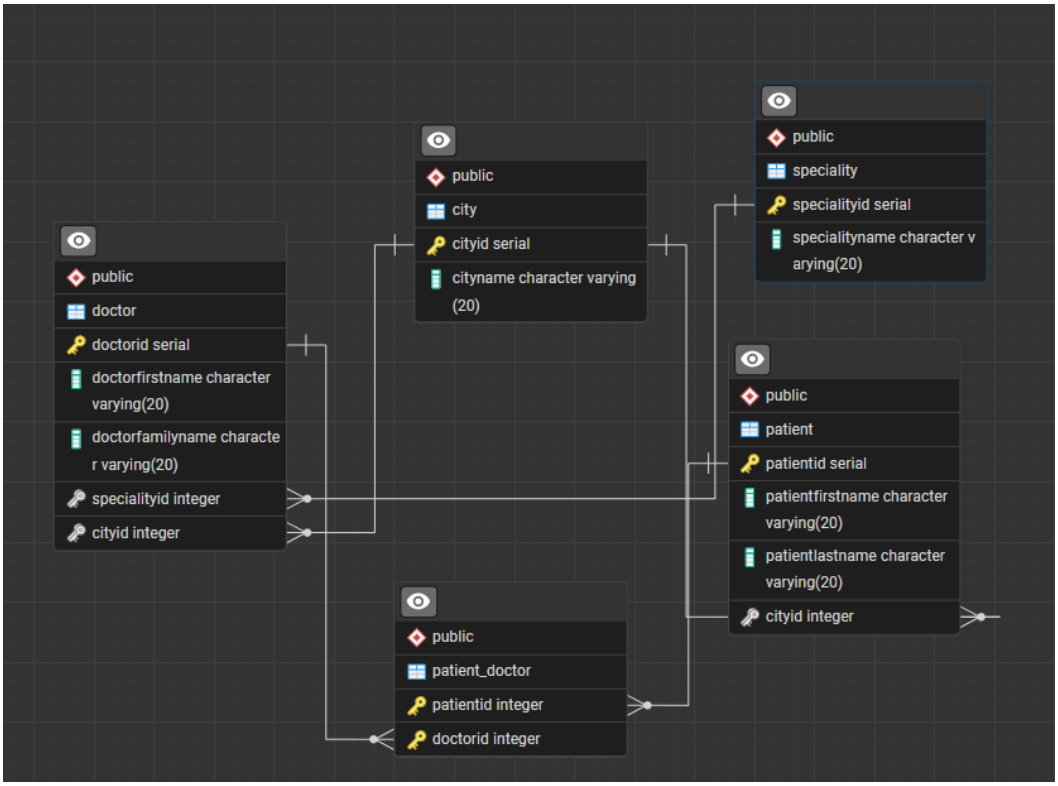
4. Then create the 3 route files doctorroute.js, specialityroute.js, cityroute.js in the route folder
5. After that we have import them in server.js and use them

```
import doctorRoute from "./routes/doctorroutes.js";
import specialityRoute from "./routes/specialityroute.js";
import cityRoute from "./routes/cityroute.js";
```

```
app.use("/doctors", doctorRoute); // Use the doctor routes
app.use("/specialities", specialityRoute); // Use the speciality routes
app.use("/city", cityRoute); // Use the city routes
```

6. To increase the complexity and learn about many-to-many relationship I added the patient entity, a dr can have multiple patients, and a patient can have multiple drs. A patient lives in a city, and a city can have multiple patients, so the city fk will be in the patient, a one-to-many relationship.

```sql
create table patient(
patientid serial primary key,
patientfirstname varchar(20),
patientlastname varchar(20),
cityid int references city(cityid) on delete cascade
);

create table doctor(
doctorid serial primary key,
doctorfirstname varchar(20),
doctorfamilyname varchar(20),
specialityid int references speciality(specialityid) on delete cascade,
cityid int references city(cityid) on delete cascade
);

create table patient_doctor(
patientid int references patient(patientid) on delete cascade,
doctorid int references doctor(doctorid) on delete cascade,
primary key (patientid, doctorid)
);
```



7. Insert data into the tables (5 records in each)

```
insert into city(cityname) values('Sour');
insert into speciality(specialityname) values('Cardiology');
insert into patient(patientfirstname, patientlastname,cityid) values('Marilyn','Hig',5);
insert into doctor(doctorfirstname,doctorfamilyname, specialityid, cityid) values('Lyla', 'Neal', 5, 5);
insert into patient_doctor(patientid, doctorid) values(5,5);

select * from speciality;
select * from city;
select * from doctor;
select * from patient;
select * from patient_doctor;
```

8. Add needed route files for patient and the patient_doctor, patientroute.js, and assign_doctor, below is the full imports and more in server.js

```
import doctorRoute from "./routes/doctorroute.js";
import specialityRoute from "./routes/specialityroute.js";
import cityRoute from "./routes/cityroute.js";
import patientRoute from "./routes/patientroute.js";
import assignRoute from "./routes/assigndoctortopatient.js";
```

```
app.use("/doctor", doctorRoute);
app.use("/speciality", specialityRoute);
app.use("/city", cityRoute);
app.use("/patient", patientRoute);
app.use("/assign", assignRoute);
```

9. CityRoute: below is the full view of cityroute.js
    a. Get: returns all existing cities from the db
    b. Post: inserts a new city into the db by only taking cityName in the json body of the post request.
    c. Put: updates an existing city by taking the city id as a param in the route url, comparing it to the cities in the db, then updating the city with the new cityName sent in the json body of the request.
    d. Both Put and Post will return the request data just entered as a response as a best practice.
    e. Delete: deletes a city according to a given cityid given as a param.

```
import express, { Router } from "express" ;
import pool from "../db.js";

const cityRoute = Router();
cityRoute.use(express.json());

cityRoute.get("/", async(req,res)=>{
    const city=await pool.query("SELECT * FROM city");
    res.json({city: city.rows, count: city.rows.length, message: "city retrieved successfully"});
})

cityRoute.post("/", async(req,res)=>{
    const {cityName}=req.body;
    const addcity = await pool.query("insert into city (cityname) values ($1) returning *;", [cityName]);
    res.json({message: "city added successfully", city: addcity.rows[0]});
})

cityRoute.put("/:id", async(req,res)=>{
    const {cityName}=req.body;
    const updatecity = await pool.query("update city set cityname = $1 where cityid = $2 returning *;",
        [cityName, req.params.id]);
    res.json({message: "city updated successfully", city: updatecity.rows[0]});
})

cityRoute.delete("/:id", async(req,res)=>{
    const deletecity = await pool.query("delete from city where cityid = $1;", [req.params.id]);
    res.json({message: "city deleted successfully"});
})

export default cityRoute;
```

f. Below is the postman requests and responses for city.

   i. Get:

Doctor/Patient System / **Get a City**

GET http://localhost:3000/city

Docs Params Authorization Headers (6) Body Scr

Body Cookies Headers (7) Test Results

{} JSON Preview Visualize


```
{
    "city": [
        {
            "cityid": 1,
            "cityname": "Beirut"
        },
        {
            "cityid": 2,
            "cityname": "Saida"
        },
        {
            "cityid": 3,
            "cityname": "Nabatiye"
        },
        {
            "cityid": 4,
            "cityname": "Baalbek"
        },
        {
            "cityid": 5,
            "cityname": "Sour"
        }
    ],
    "count": 5,
    "message": "city retrieved successfully"
}
```


ii. Post:

Doctor/Patient System / **Add a City**

POST http://localhost:3000/city

Docs Params Authorization Headers (8) Body

none form-data x-www-form-urlencoded raw

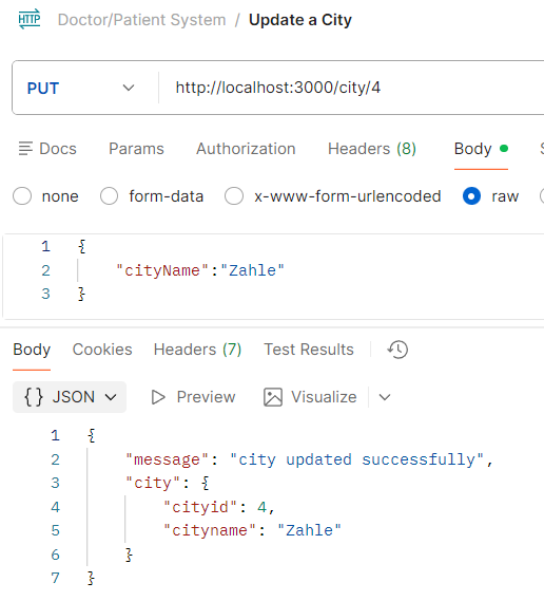
```
{
    "cityName":"Jounieh"
}
```

Body Cookies Headers (7) Test Results

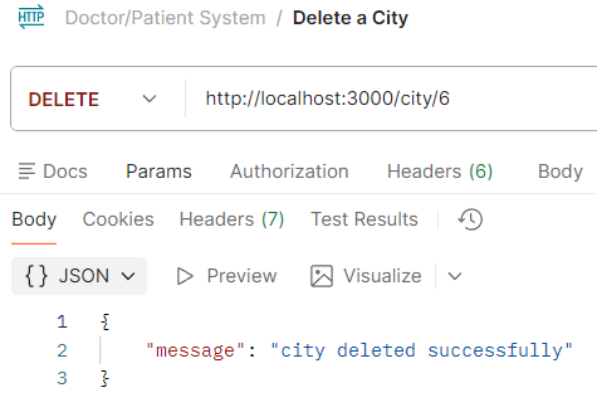
{} JSON Preview Visualize


```
{
    "message": "city added successfully",
    "city": {
        "cityid": 6,
        "cityname": "Jounieh"
    }
}
```


iii. Put:



iv. Delete:

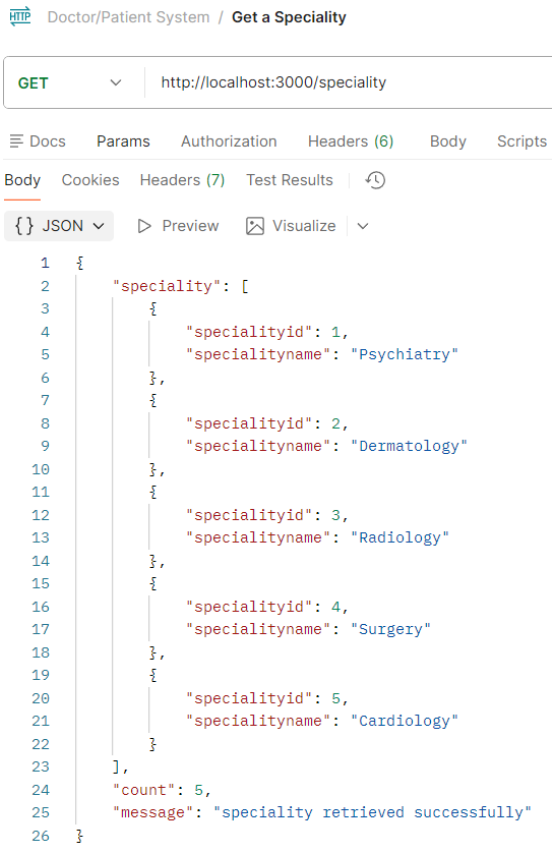


10. SpecialityRoute: below is the full view of specialityroute.js
    a. Get: returns all existing specialties from the db
    b. Post: inserts into the db a new speciality by taking the specialityName in the json body of the request.
    c. Put: updates an existing speciality's name by identifying it from the id passed in the route param, and the specialityName in the json body of the request.
    d. Delete: deletes an existing speciality by identifying it from the id passed in the route param.

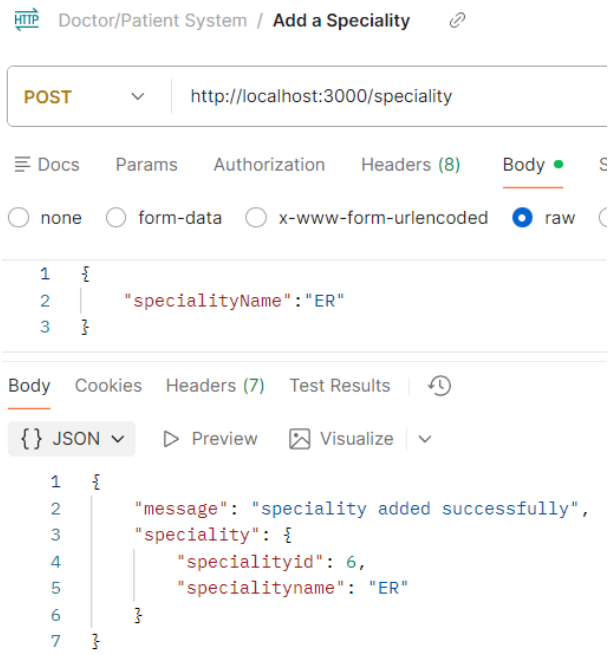
```
import express, { Router } from "express" ;
import pool from "../db.js";

const specialityRoute = Router();
specialityRoute.use(express.json());

specialityRoute.get("/", async(req,res)=>{
    const speciality=await pool.query("SELECT * FROM speciality");
    res.json({speciality: speciality.rows, count: speciality.rows.length, message: "speciality retrieved successfully"});
})

specialityRoute.post("/", async(req,res)=>{
    const {specialityName}=req.body;
    const addspeciality = await pool.query("insert into speciality (specialityname) values ($1) returning *;", [specialityName]);
    res.json({message: "speciality added successfully", speciality: addspeciality.rows[0]});
})

specialityRoute.put("/:id", async(req,res)=>{
    const {specialityName}=req.body;
    const updatespeciality = await pool.query("update speciality set specialityname = $1 where specialityid = $2 returning *;",
        [specialityName, req.params.id]);
    res.json({message: "speciality updated successfully", speciality: updatespeciality.rows[0]});
})

specialityRoute.delete("/:id", async(req,res)=>{
    const deletespeciality = await pool.query("delete from speciality where specialityid = $1;", [req.params.id]);
    res.json({message: "speciality deleted successfully"});
})

export default specialityRoute;
```

e. Below is the postman requests and responses for speciality.

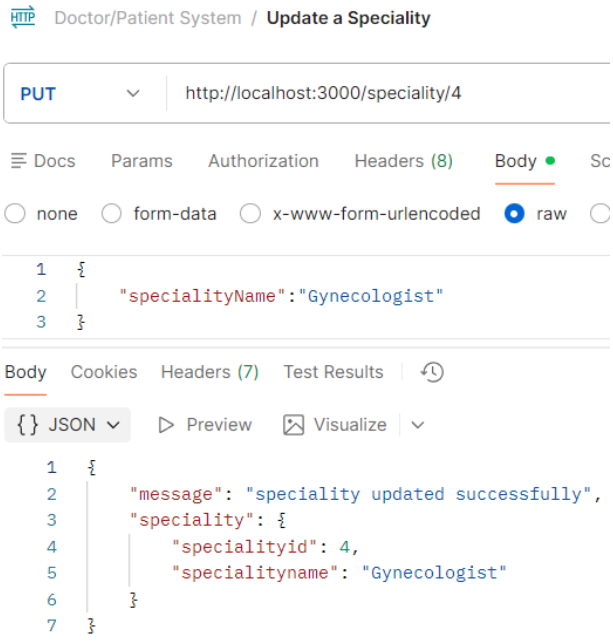
i. Get:



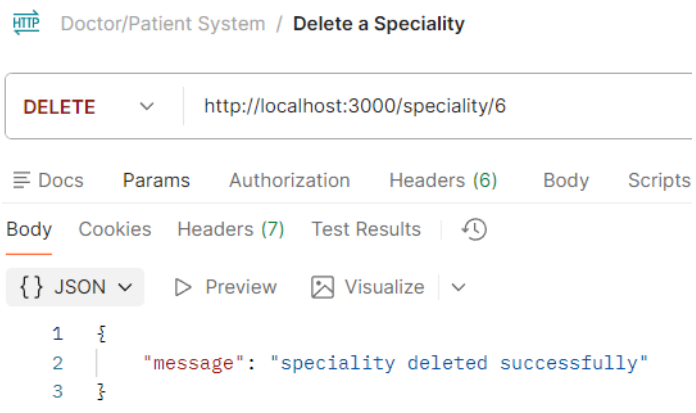
ii. Post:



iii. Put:



iv. Delete:



11. PatientRoute: below is the full view of patientroute.js
    a. Get: retrieves all the existimg patients from the db
    b. Post: inserts into the db a new patient by taking their first name, last name, and city id in the json body of the request.
    c. Put: updates a patient record in the db by identifying them using the id ssent in the route param and replacing their first and last name with the json body of the request.
    d. Delete: deletes a patient record from the db by identifying them with the id sent in the route param.

```
import express, { Router } from "express" ;
import pool from "../db.js";

const patientRoute = Router();
patientRoute.use(express.json());

patientRoute.get("/", async(req,res)=>{
    const patient=await pool.query("SELECT * FROM patient");
    res.json({patient: patient.rows, count: patient.rows.length, message: "patient retrieved successfully"});
})

patientRoute.post("/", async(req,res)=>{
    const {patientfirstname, patientlastname, cityid}=req.body;
    const addpatient = await pool.query("insert into patient (patientfirstname, patientlastname, cityid) values ($1, $2, $3) returning *;",
        [patientfirstname, patientlastname, cityid]);
    res.json({message: "patient added successfully", patient: addpatient.rows[0]});
})

patientRoute.put("/:id", async(req,res)=>{
    const {patientfirstname, patientlastname}=req.body;
    const updatepatient = await pool.query("update patient set patientfirstname = $1, patientlastname = $2 where patientid = $3 returning *;",
        [patientfirstname, patientlastname, req.params.id]);
    res.json({message: "patient updated successfully", patient: updatepatient.rows[0]});
})

patientRoute.delete("/:id", async(req,res)=>{
    const deletepatient = await pool.query("delete from patient where patientid = $1;", [req.params.id]);
    res.json({message: "patient deleted successfully"});
})

export default patientRoute;
```

    e. Below is the postman requests and responses for the patient
        i. Get:


Doctor/Patient System / Get a Patient
GET
http://localhost:3000/patient
Docs
Params
Authorization
Headers (6)
Body
Scripts
Body
Cookies
Headers (7)
Test Results
{} JSON
Preview
Visualize
{
"patient": [
{
"patientid": 1,
"patientfirstname": "Fox",
"patientlastname": "Stark",
"cityid": 1
},
{
"patientid": 2,
"patientfirstname": "Angel",
"patientlastname": "Stark",
"cityid": 2
},
{
"patientid": 3,
"patientfirstname": "Gab",
"patientlastname": "Conner",
"cityid": 3
},
{
"patientid": 4,
"patientfirstname": "Cruz",
"patientlastname": "Bruce",
"cityid": 4
},
{
"patientid": 5,
"patientfirstname": "Marilyn",
"patientlastname": "Hig",
"cityid": 5
}
],
"count": 5,
"message": "patient retrieved successfully"
}

ii. Post:

Doctor/Patient System / **Add a Patient**

POST http://localhost:3000/patient

Docs Params Authorization Headers (8) Body

none form-data x-www-form-urlencoded raw

```
{
    "patientfirstname":"May",
    "patientlastname":"Robon",
    "cityid":"3"
}
```

Body Cookies Headers (7) Test Results

{ } JSON Preview Visualize

```
{
    "message": "patient added successfully",
    "patient": {
        "patientid": 6,
        "patientfirstname": "May",
        "patientlastname": "Robon",
        "cityid": 3
    }
}
```

iii. Put:

Doctor/Patient System / **Update a Patient**

PUT http://localhost:3000/patient/4

Docs Params Authorization Headers (8) Body S

none form-data x-www-form-urlencoded raw

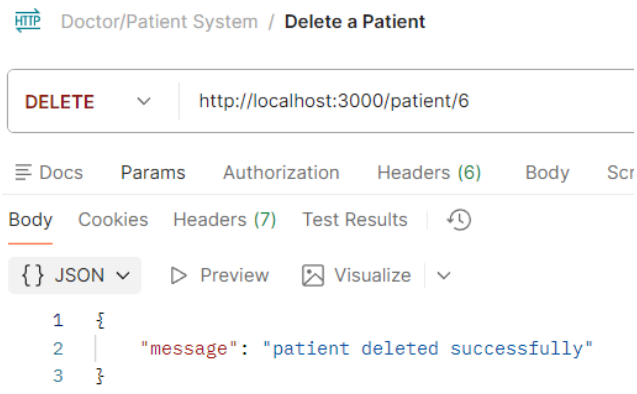
```
{
    "patientfirstname":"Ray",
    "patientlastname":"Brown"
}
```

Body Cookies Headers (7) Test Results

{ } JSON Preview Visualize

```
{
    "message": "patient updated successfully",
    "patient": {
        "patientid": 4,
        "patientfirstname": "Ray",
        "patientlastname": "Brown",
        "cityid": 4
    }
}
```

iv. Delete:



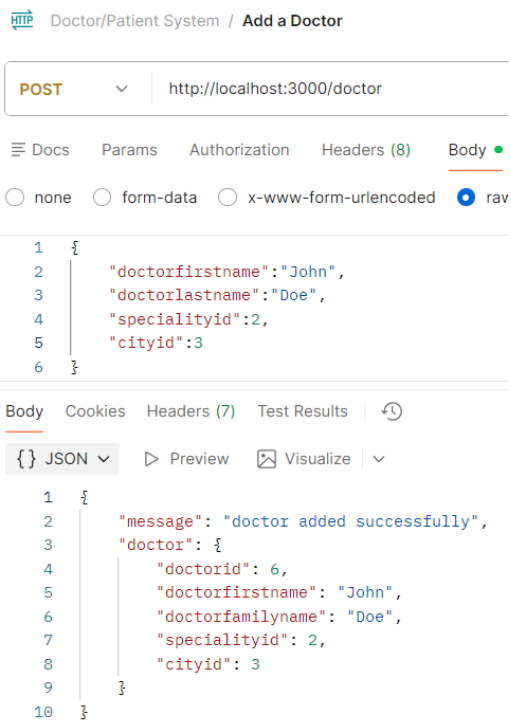
12. DoctorRoute: below is the full view of doctorroute.js
    a. Get: retrieves all the doctors from the db.
    b. Post: inserts a new doctor record into the db by taking the first name, last name, speciality id, city id, in the json body of the request.(in the db the doctor last name is called doctor family name so you will notice using doctorfamilyname in query and doctorlastname while calling the instance out of the query)
    c. Put: updates a doctor record by first identifying the dr using the id from the route param, and then replacing existing first name and last name with what was sent in the json body of the request.
    d. Delete: deletes a doctor record by identifying the dr using the id from the route param.

```
import express, { Router } from "express" ;
import pool from "../db.js";

const doctorRoute = Router();
doctorRoute.use(express.json());

doctorRoute.get("/", async(req,res)=>{
    const doctor=await pool.query("SELECT * FROM doctor");
    res.json({doctor: doctor.rows, count: doctor.rows.length, message: "doctor retrieved successfully"});
})

doctorRoute.post("/", async(req,res)=>{
    const {doctorfirstname, doctorlastname, specialityid, cityid}=req.body;
    const adddoctor = await pool.query("insert into doctor (doctorfirstname, doctorfamilyname, specialityid, cityid) values ($1, $2, $3, $4) returning *;",
        [doctorfirstname, doctorlastname, specialityid, cityid]);
    res.json({message: "doctor added successfully", doctor: adddoctor.rows[0]});
})

doctorRoute.put("/:id", async(req,res)=>{
    const {doctorfirstname, doctorlastname}=req.body;
    const updatedoctor = await pool.query("update doctor set doctorfirstname = $1, doctorfamilyname = $2 where doctorid = $3 returning *;",
        [doctorfirstname, doctorlastname, req.params.id]);
    res.json({message: "doctor updated successfully", doctor: updatedoctor.rows[0]});
})

doctorRoute.delete("/:id", async(req,res)=>{
    const deletedoctor = await pool.query("delete from doctor where doctorid = $1;", [req.params.id]);
    res.json({message: "doctor deleted successfully"});
})

export default doctorRoute;
```

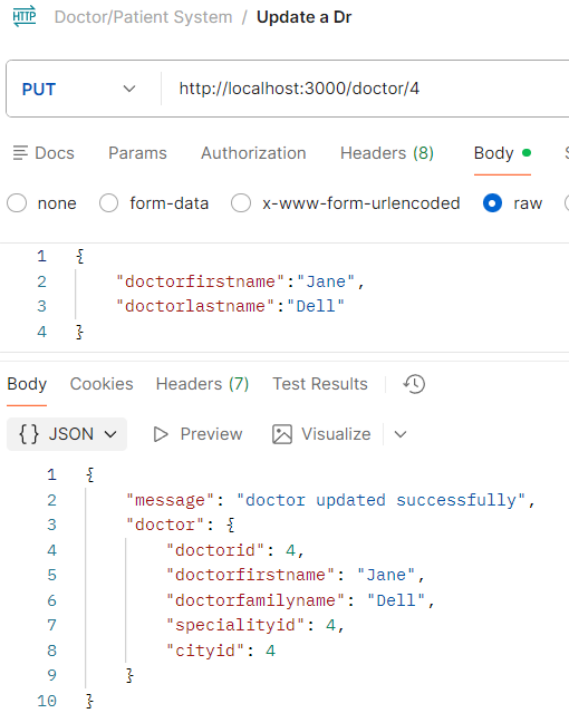
    e. Below is the postman requests and responses for the doctor
        i. Get:


```
{
    "doctor": [
        {
            "doctorid": 1,
            "doctorfirstname": "Philip",
            "doctorfamilyname": "Bruce",
            "specialityid": 1,
            "cityid": 1
        },
        {
            "doctorid": 2,
            "doctorfirstname": "John",
            "doctorfamilyname": "Lawso",
            "specialityid": 2,
            "cityid": 2
        },
        {
            "doctorid": 3,
            "doctorfirstname": "Cameron",
            "doctorfamilyname": "Flynn",
            "specialityid": 3,
            "cityid": 3
        },
        {
            "doctorid": 4,
            "doctorfirstname": "Talia",
            "doctorfamilyname": "Kelly",
            "specialityid": 4,
            "cityid": 4
        },
        {
            "doctorid": 5,
            "doctorfirstname": "Lyla",
            "doctorfamilyname": "Neal",
            "specialityid": 5,
            "cityid": 5
        }
    ],
    "count": 5,
    "message": "doctor retrieved successfully"
}
```

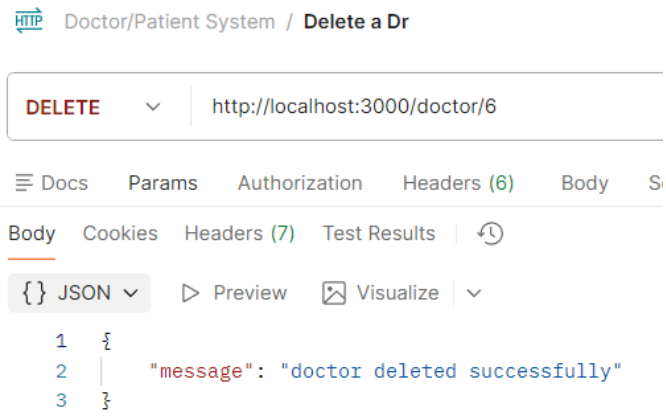

ii. Post:



iii. Put:



iv. Delete:



13. AssignRoute: below is the full view of assignroute.js, this route is a bit tricky, it represents the m:m relationship btw patient and dr, the entity(patient_doctor) itself has dr and patient ids as fk and pk.
    a. Get: retrieves all the existing records from the db
    b. Post: inserts a new relationship into the table, if a patient got assigned to a dr newly, use this route, send the doctor id and patient id in the json body of the request
    c. Put: updating an existing assignment(e.g. patient 1 assigned to dr 2, but I want to assign patient 1 to dr 1) in the route param we'll take doctor_id and patient_id which represent the existing patient and dr assignment, in this table you have to search(WHERE) using both fk cause their relation together is unique, they are the pk. While doctorid and patientid are given in the json request body cause they are the new values to be inserted. In short, take the ids to identify what assignment you want to edit in route param, and take the new data to be set in the json request.
    d. Delete: identify the relation using doctor_id and patient_id in route param and delete.

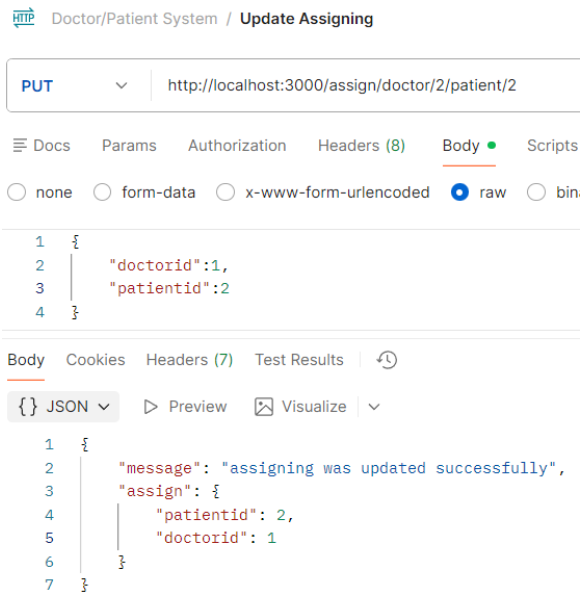
```
import express, { Router } from "express" ;
import pool from "../db.js";

const assignRoute = Router();
assignRoute.use(express.json());

assignRoute.get("/", async(req,res)=>{
    const assign=await pool.query("SELECT * FROM patient_doctor");
    res.json({assign: assign.rows, count: assign.rows.length, message: "assign retrieved successfully"});
})

assignRoute.post("/", async(req,res)=>{
    const {doctorid, patientid}=req.body;
    const addassign = await pool.query("insert into patient_doctor (patientid, doctorid) values ($1, $2) returning *;", [patientid, doctorid]);
    res.json({message: "a patient was assigned for a doctor", assign: addassign.rows[0]});
})

assignRoute.put("/doctor/:doctor_id/patient/:patient_id", async(req,res)=>{
    const {doctorid, patientid}=req.body;
    const updateassign = await pool.query("update patient_doctor set doctorid = $1, patientid = $2 where doctorid = $3 and patientid = $4 returning *;",
        [doctorid, patientid, req.params.doctor_id, req.params.patient_id]);
    res.json({message: "assigning was updated successfully", assign: updateassign.rows[0]});
})

assignRoute.delete("/doctor/:doctor_id/patient/:patient_id", async(req,res)=>{
    const deleteassign = await pool.query("delete from patient_doctor where doctorid = $1 and patientid = $2;",
        [req.params.doctor_id, req.params.patient_id]);
    res.json({message: "assigning is deleted successfully"});
})

export default assignRoute;
```

    e. Below is the postman requests and responses for assign

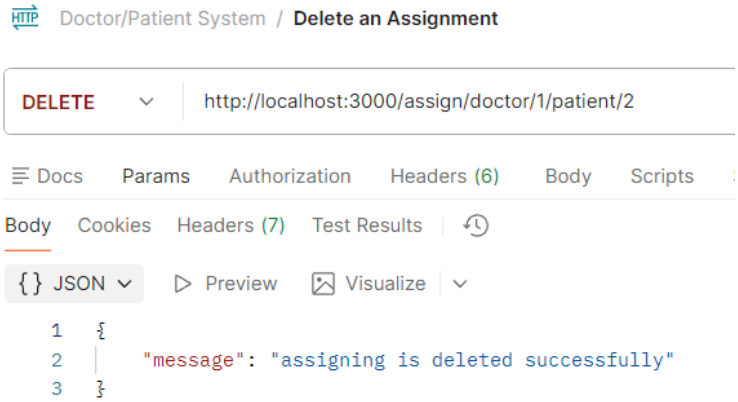
i. Get:

ii. Post:

iii. Put:



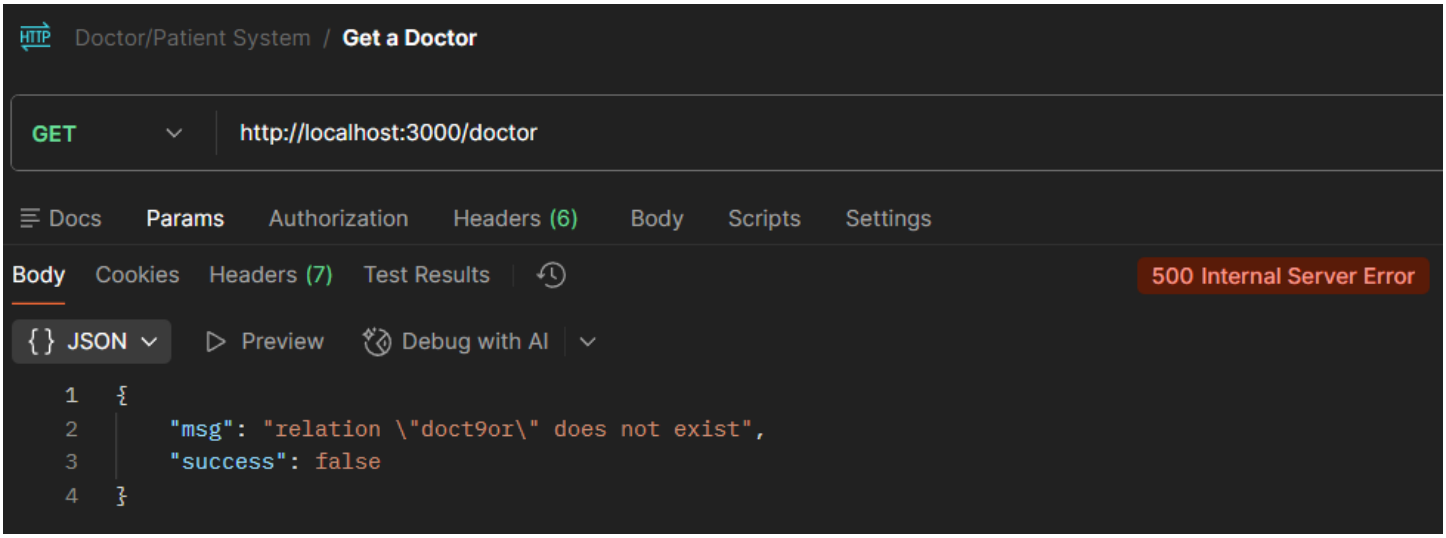
iv. Delete:



14. The user doesn't always enter the exact information that the system wants, so as a best practice and to protect the db as well as save our program from crashing, we use a try catch, surround your code with a try catch, if the user input or the process you are running doesn't trigger an error then the code will run successfully, but if an error was triggered, the catch will help you save the program from crashing and send you a clear error message instead of long complicated logs.
15. In the doctorroute.js, we will try to add try catch and specific error msgs to enhance pur routes and functions.
16. Get: we surround the function logic with a try catch, if there's an error, the catch will run and it will post the error reason only along with success variable clarifying the failure. We also added status(200) to req.json => req.status(200).json, it's usually the default success code anw, but specifying it explicitly is a better practice.

```
doctorRoute.get("/", async(req,res)=>{
    try{
    const doctor=await pool.query("SELECT * FROM doctor");
    res.status(200).json({doctor: doctor.rows, count: doctor.rows.length, message: "doctor retrieved successfully"});
    }
    catch(err){
        res.status(500).json({msg:err.message,success:false});
    }
})
```

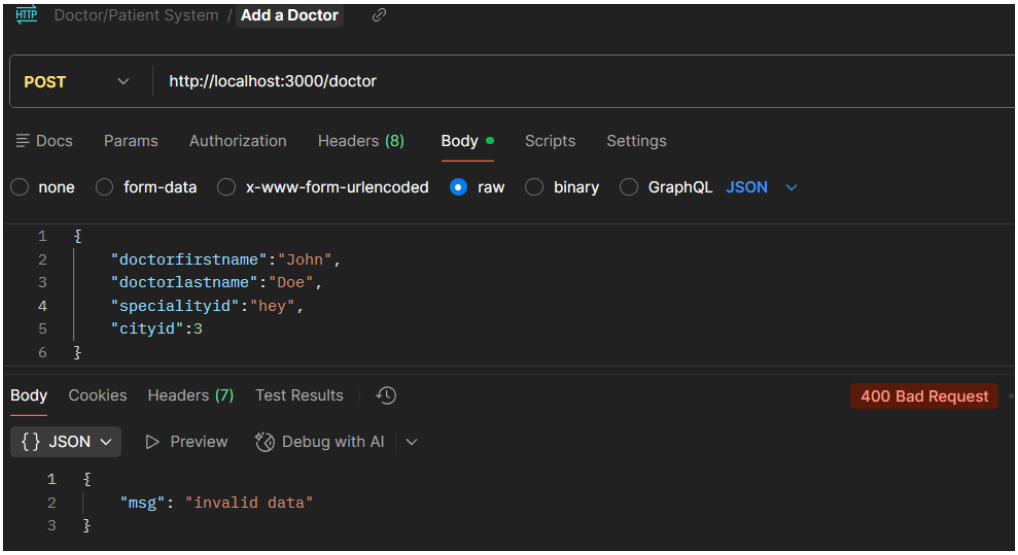
Now if I mess up the query and send the request through postman, the error msg will be clear



17. Post: here as well, first step surround your code logic with a try catch, add the error message and the status. But in the post we are inserting data, which means we expect string in **first name and last name** and we expect numerical number in **speciality id and city id.**
18. To check if a variable is a string simply use typeof which gets the type of the variable (the type will be assigned as soon as the input is in it) then if the type of this variable !== (isn't) string OR the type of the ids isn't numerical (checked in a function isNaN()) , if any of these 4 conditions is true, it means the type of data entered is bad data, then enter the if statement and throw an error.
19. 201 is the code of creation, returned after a new record gets created in the db.

```
doctorRoute.post("/", async(req,res)=>{
    try{
    const {doctorfirstname, doctorlastname, specialityid, cityid}=req.body;
    if(typeof doctorfirstname !=="string" || typeof doctorlastname !=="string" || isNaN(specialityid) || isNaN(cityid))
        return res.status(400).json({msg:"invalid data"});
    const adddoctor = await pool.query("insert into doctor (doctorfirstname, doctorfamilyname, specialityid, cityid) values ($1, $2, $3, $4) returning *;",
        [doctorfirstname, doctorlastname, specialityid, cityid]);
    res.status(201).json({message: "doctor added successfully", doctor: adddoctor.rows[0]});
    }
    catch(err){
        res.status(500).json({msg:err.message,success:false});
    }
})
```

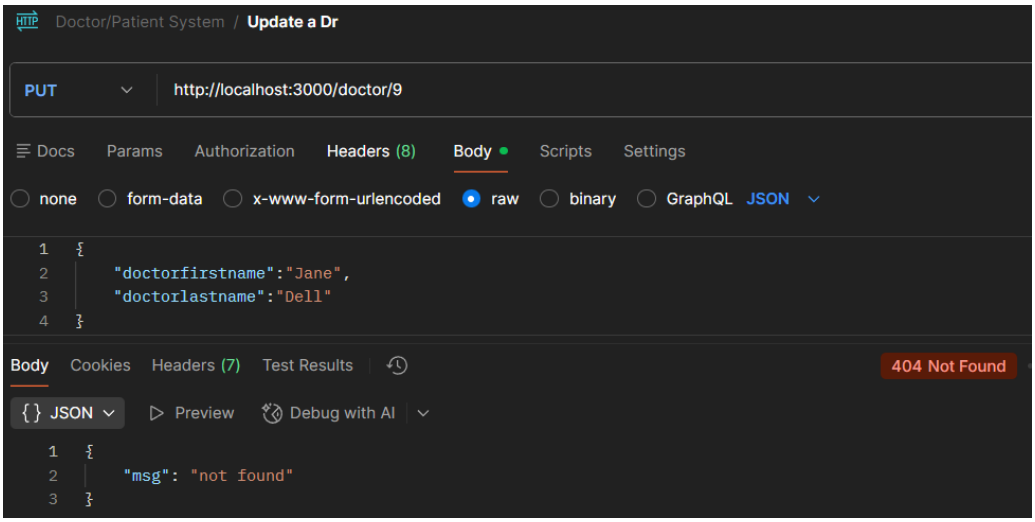
Now if you try sending string in speciality id for example



20. Put: surround by try catch, add err msg. Add first if statement that checks the input like we did before, 400 for bad request, bad input data. The new is the second if statement, it checks if the row count returned is 0 or not, if it's zero then there was no doctor to edit so throw 404:not found

```
doctorRoute.put("/:id", async(req,res)=>{
    try{
        const {doctorfirstname, doctorlastname}=req.body;
        if(typeof doctorfirstname !=="string" || typeof doctorlastname !=="string" || isNaN(req.params.id))
            return res.status(400).json({msg:"invalid data"});
        const updatedoctor = await pool.query("update doctor set doctorfirstname = $1, doctorfamilyname = $2 where doctorid = $3 returning *;",
            [doctorfirstname, doctorlastname, req.params.id]);
        if(updatedoctor.rowCount===0)
            return res.status(404).json({msg:"not found"})
        res.json({message: "doctor updated successfully", doctor: updatedoctor.rows[0]});
    }
    catch(err){
        res.status(500).json({msg:err.message,success:false});
    }
})
```

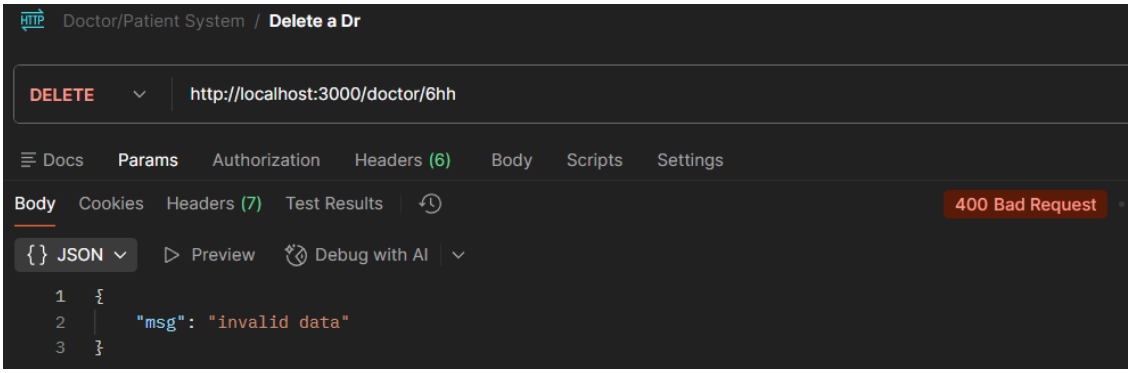
So if I try to update a non existing dr with sending a non existing id it the route param:



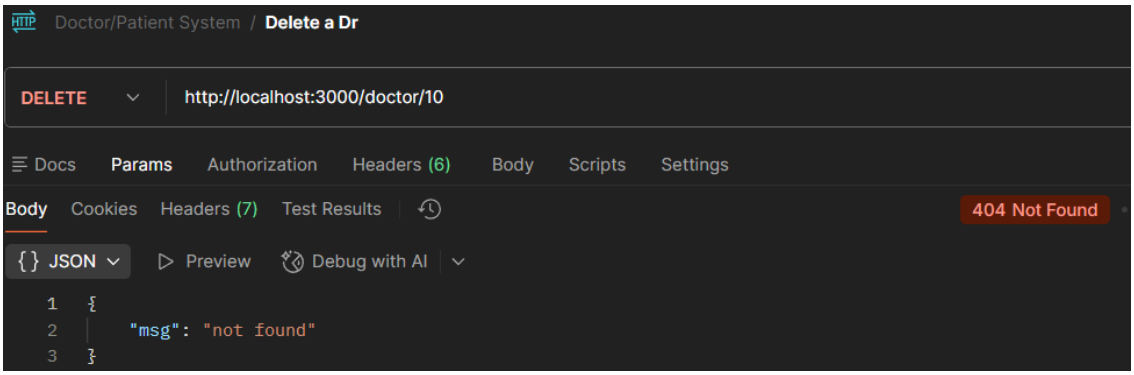
21. Delete: for the delete we can add the same errors as before, check if the id is numerical and exist.

```
doctorRoute.delete("/:id", async(req,res)=>{
    try{
        if(isNaN(req.params.id))
            return res.status(400).json({msg:"invalid data"});
        const deletedoctor = await pool.query("delete from doctor where doctorid = $1;", [req.params.id]);
        if(deletedoctor.rowCount===0)
            return res.status(404).json({msg:"not found"});
        res.json({message: "doctor deleted successfully"});
    }catch(err){
        req.status(500).json({msg: err.message, success:false});
    }
})
```

When sending improper data, string instead of int id



When sending a non existing id



Tuesday, February 10, 2026

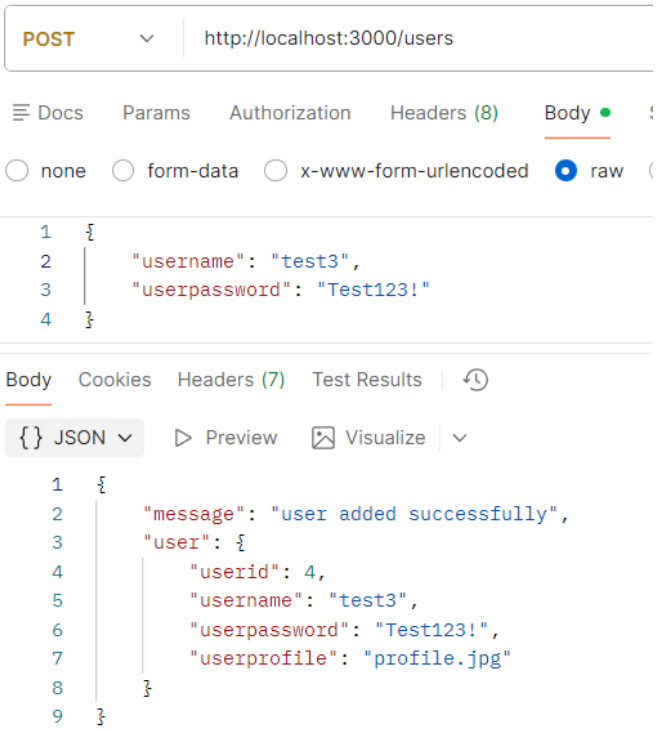
1. For this practice, let's do the routine things first, create the users table in the db

```
create table users(
    userid serial primary key,
    username varchar(50),
    userpassword varchar(100),
    userprofile varchar(1000) default 'profile.jpg'
);
```

2. Then create the usersRoute.js and import it in server.js, below is userRoute with the most important route for our exercise, the post route.

```
import express, { Router } from "express" ;
import pool from "../db.js";

const usersRoute = Router();
usersRoute.use(express.json());

usersRoute.get("/", async(req, res)=>{
    const users = await pool.query("select * from users");
    res.json(users.rows);
})

usersRoute.post("/", async(req, res)=>{
    const {username, userpassword}=req.body;
    const adduser = await pool.query("insert into users (username, userpassword) values ($1, $2) returning *;",
        [username, userpassword]);
    res.json({message: "user added successfully", user: adduser.rows[0]});
})

export default usersRoute;
```

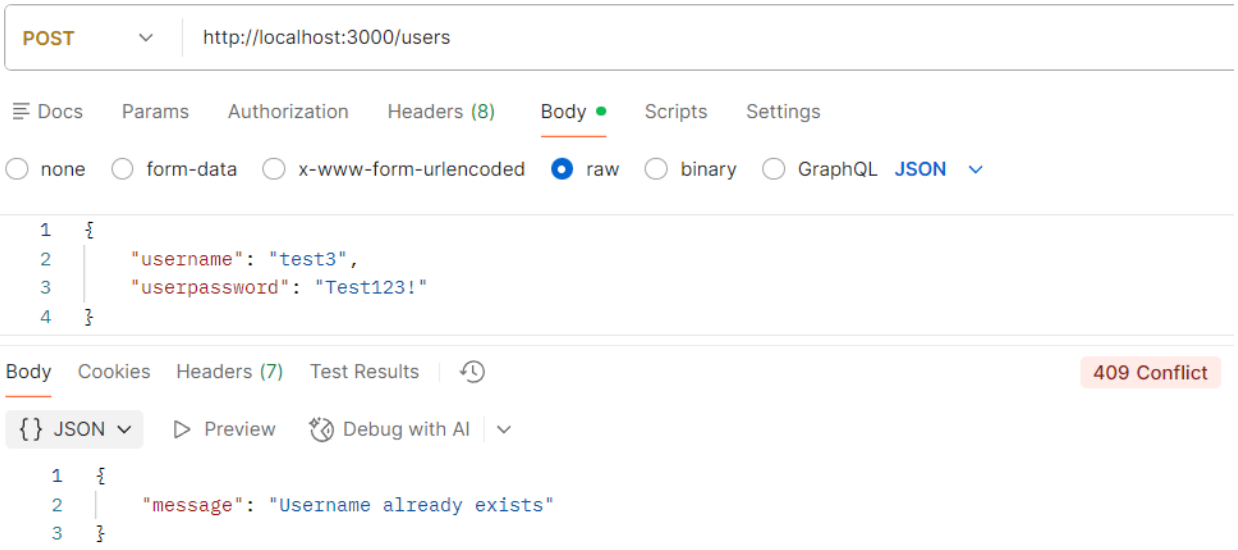
3. Test in postman



4. The latter request will give you response status 200, which is mostly for the get function status, but we entered data, we created a record, so we want status 201, thus it's preferred to specify it like this:
   `res.status(201).json({message: "user added successfully", user: adduser.rows[0]});`
5. Now let's add try catch for the route to avoid crashing as well as an if statement to avoid having duplicate usernames

```
usersRoute.post("/", async(req, res)=>{
    try{
        const check=await pool.query("select * from users where username = $1", [req.body.username]);
        if(check.rows.length > 0){
            return res.status(409).json({message: "Username already exists"}); //409: conflict
        }
        const {username, userpassword}=req.body;
        const adduser = await pool.query("insert into users (username, userpassword) values ($1, $2) returning *;",
            [username, userpassword]);
        res.status(201).json({message: "user added successfully", user: adduser.rows[0]});
    }catch(err){
        res.status(500).json({message: "Error adding user", error: err.message});
    }
})
```

6. Test in postman:



7. Last addition would be hashing the password cause it doesn't make sense to store it raw for that download in the terminal this package npm i bcrypt
8. Import it in the usersRoute import bcrypt from "bcrypt";
9. Then hash the password before inserting it into the db

```
usersRoute.post("/", async(req, res)=>{
    try{
        const check=await pool.query("select * from users where username = $1", [req.body.username]);
        if(check.rows.length > 0){
            return res.status(409).json({message: "Username already exists"}); //409: conflict
        }
        // Hash the password before storing it in the database, this function takes 2 parameters,
        // the password to hash and the number of salt rounds (10 is a common choice)
        const hashpassword= await bcrypt.hash(req.body.userpassword, 10);
        const {username}=req.body;
        const adduser = await pool.query("insert into users (username, userpassword) values ($1, $2) returning *;",
            [username, hashpassword]);
        res.status(201).json({message: "user added successfully", user: adduser.rows[0]});
    }catch(err){
        res.status(500).json({message: "Error adding user", error: err.message});
    }
})
```

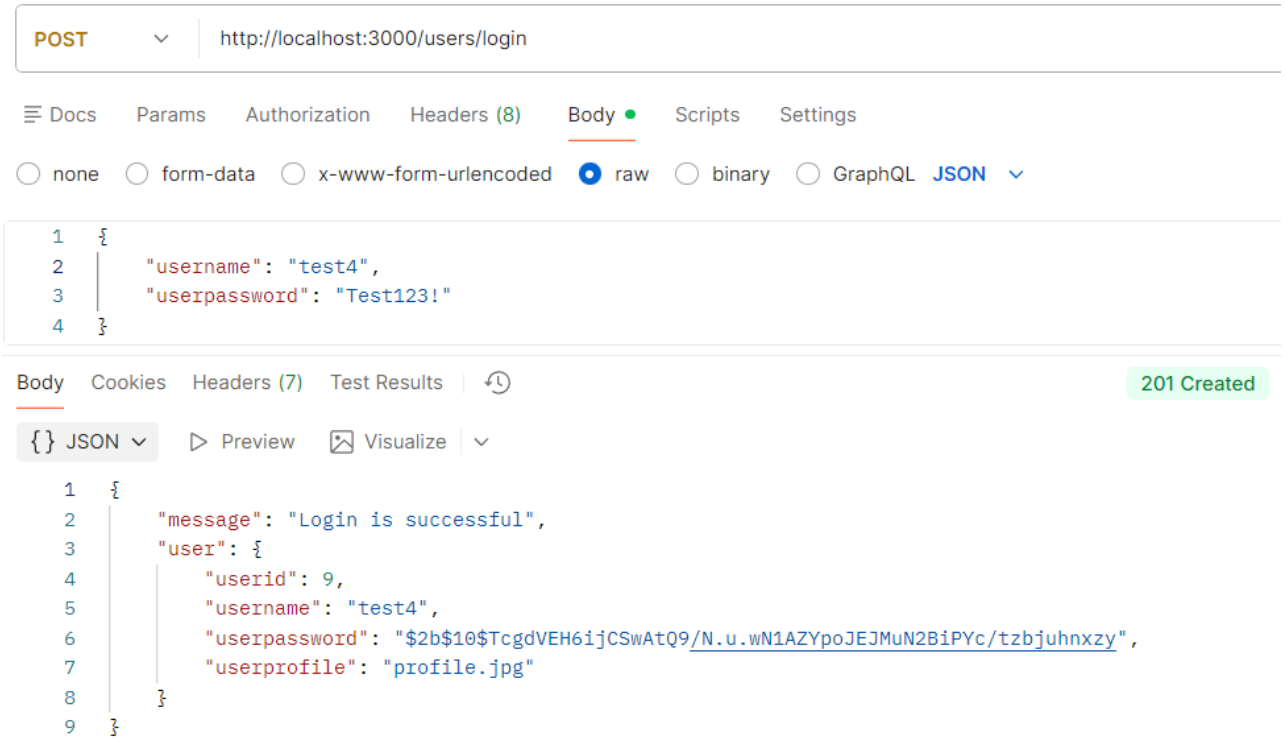
10. The higher the salt, the heavier it is on the cpu but the higher the security is. (10 is the default), below is the postman request of hashing the password


```
{
    "message": "user added successfully",
    "user": {
        "userid": 9,
        "username": "test4",
        "userpassword": "$2b$10$TcgdVEH6ijCSwAtQ9/N.u.wN1AZYpoJEJMuN2BiPYc/tzbjuhnxzy",
        "userprofile": "profile.jpg"
    }
}
```


11. Encryption is different than hashing, encryption can be decrypted but hashing is only one way, that's why it's strong and then the only way to be hacked is for the user to guess your password.
12. Now let's do login, post cause we need the user credentials, then we compare them with what we already have in the db.

```
usersRoute.post("/login", async(req, res)=>{
    try{
        // check if user exists
        const check=await pool.query("select * from users where username = $1", [req.body.username]);
        // if the query returns 0 rows, it means user doesn'y exist
        if(check.rows.length === 0){
            return res.status(404).json({message: "User does not exist"});
        }
        const {username, userpassword}=req.body;
        // compare the provided password with the hashed password stored in the database using bcrypt's compare function
        // userpassword: provided password
        // check.rows[0].userpassword: hashed password from the database (check has the query result from before)
        const checkpassword = await bcrypt.compare(userpassword, check.rows[0].userpassword);
        // if statement to determine the outcome of the comparison
        if(!checkpassword){
            return res.status(409).json({message: "Invalid credentials"});
        }
        // considering the user credntials are correct, return the user credentials and indeicate he's logged in
        const loginuser = await pool.query("select * from users where username = $1",
            [username]);
        if(loginuser.rows.length === 0){
            return res.status(409).json({message: "Invalid credentials"});
        }
        res.status(201).json({message: "Login is successful", user: loginuser.rows[0]});
    }catch(err){
        res.status(500).json({message: "Error adding user", error: err.message});
    }
})
```

13. Postman test



14. Next session we will be uploading photos too, for that download npm i multer

Thursday, February 12, 2026

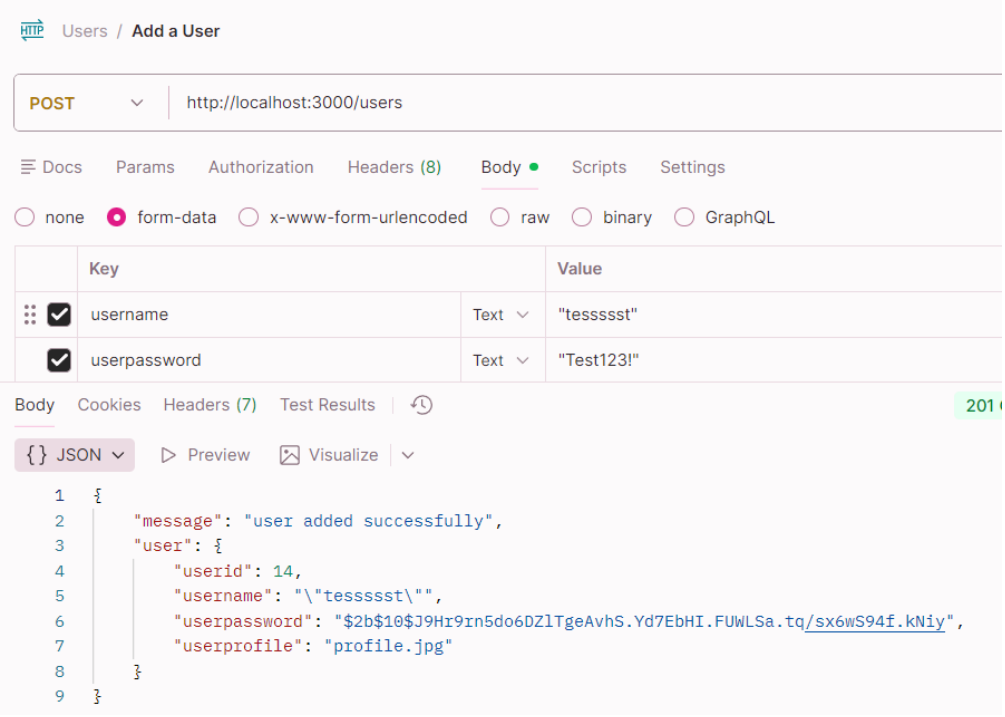
1. In the backend folder create a new folder call it upload.js

```
backend > upload.js > default
1  import multer from "multer";
2
3  const upload=multer();
4
5  export default upload;
```

2. Then go to userRoute.js and import this file import upload from "../upload.js";
3. After you import it you can now use, it for example in the post route, before the async function put upload.none() ,

```
// upload is middleware that processes the incoming request and makes the form data available in req.body.
// The none() method is used to indicate that we are not expecting any files to be uploaded, only form data.
usersRoute.post("/", upload.none() , async(req, res)=>{
```

4. Now you can send form data



5. You can see in the response the password is returned but we don't need that so add these 2 lines after the add user section before the response, and in the response send new user not add user.

```
const adduser = await pool.query("insert into users (username, userpassword) values ($1, $2) returning *;",
    [username, hashpassword]);

const newUser={...adduser.rows[0]}; // Create a new user object based on the result of the insert query
// The adduser.rows[0] contains the newly added user data, including the hashed password.
// but doesnt point to the same memory location as the original object,
// so we can safely modify it without affecting the original data in the database.
delete newUser.userpassword; // Remove the password from the response for security

res.status(201).json({message: "user added successfully", user: newUser});
```

6. Our goal is to upload a photo, so first let's change the query, add the photo variable. add the variable into the query cause we expect to insert it, req.file.filename is not to be changed

```
const adduser = await pool.query("insert into users (username, userpassword, userprofile) values ($1, $2, $3) returning *;",
    [username, hashpassword, req.file.filename]);
```

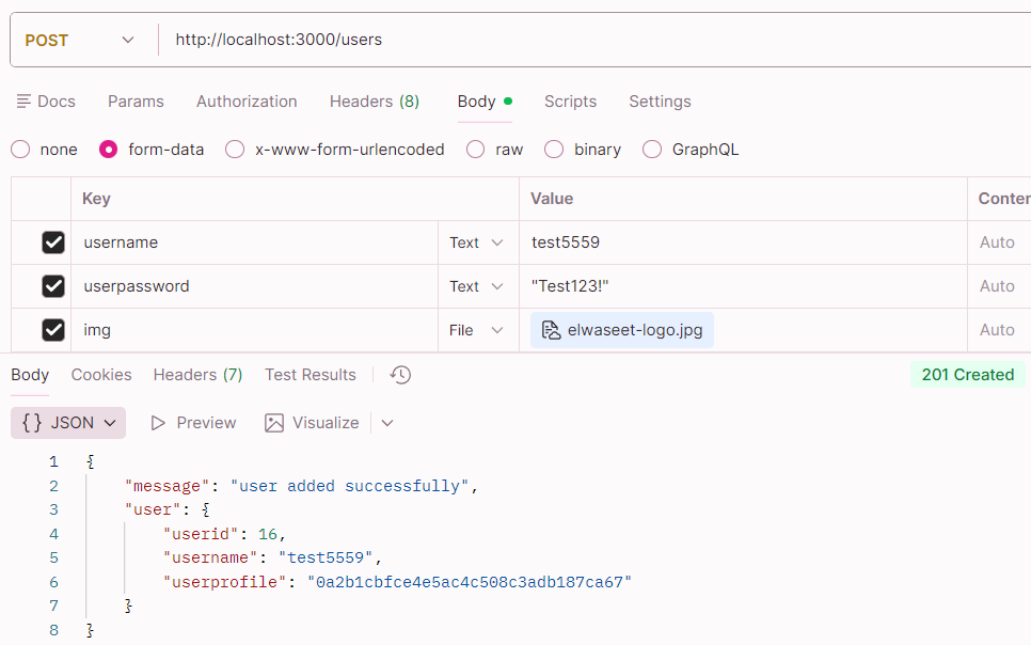
7. Then change the function we are using from upload.none() to single() to upload 1 file

```
// single() method is used to handle a single file upload, and the argument "img" specifies the name of the form field that contains the file.
usersRoute.post("/", upload.single("img") , async(req, res)=>{
```

8. In upload.js :

```
import multer from "multer";

const upload=multer({dest:"images"});

export default upload;
```

9. Postman response, plus you can find it in the db and in the images folder in the backend



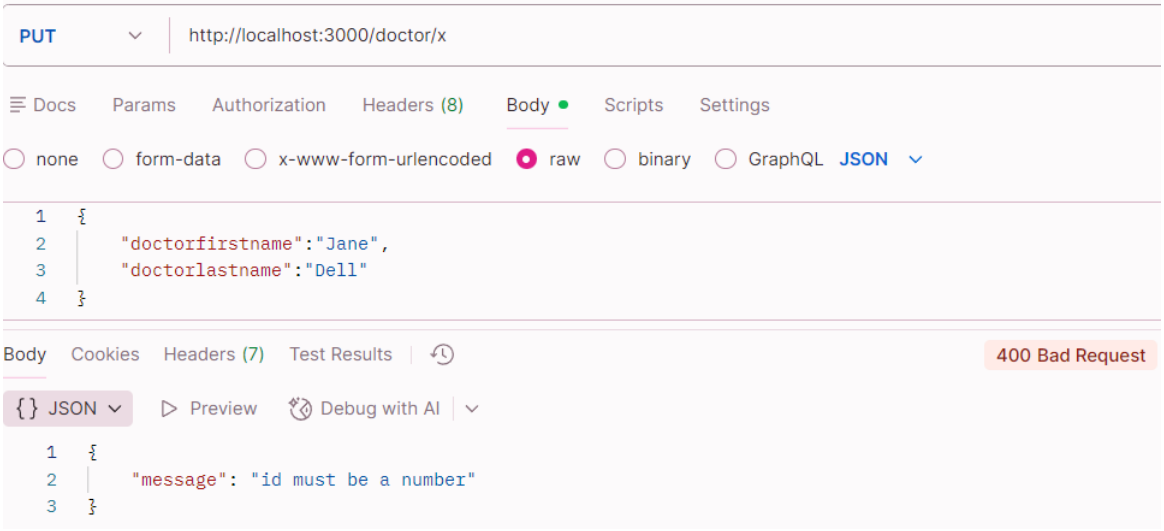
10. Multer is a type of middleware.
11. Let's try creating our own middleware, create folder middleware in backend, then create file test.js
12. Inside it we have function testMiddleware, that takes 3 parameters, req res like normal, and next.
13. In this middleware we will check if the id is a number, our goal is to simplify our routes and let the middleware handle edge cases like this.

```
function testMiddleware(req, res, next) {
    const id = req.params.id;
    if(isNaN(id)){
        return res.status(400).json({message: "id must be a number"});
    }
    console.log("This is a test middleware");
    next();
}

export default testMiddleware;
```

14. In the doctor route import this single function import testMiddleware from "../middleware/test.js";
15. Then call it in the put function, just like that, no need to fill the parameters manually.

```
doctorRoute.put("/:id", testMiddleware, async(req,res)=>{
    try{
```

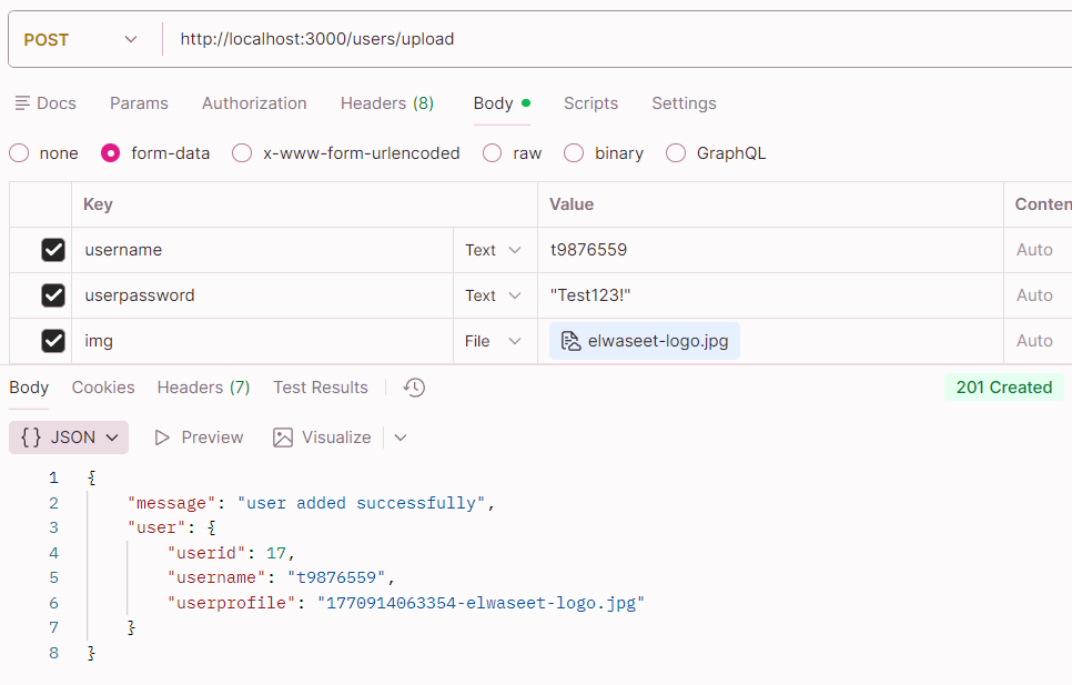
16. Postman response if you try to edit a dr and send a gibberish id



17. When the image was saved before, its name changed to random id that multer created, what if I want it to keep the og name? First go to upload.js

```
import multer from "multer";

// const upload=multer({dest:"images"});

const storage=multer.diskStorage({
    destination: "images/",
    filename: (req, file, cb) =>{ //cb: callback function
        // The filename is generated by concatenating the current timestamp (Date.now()) with the original
        // name of the uploaded file (file.originalname).
        cb(null, Date.now() + "-" + file.originalname);
    } });
// multer({storage}) creates an instance of multer with the specified storage configuration,
// which will be used to handle file uploads in the routes.
const upload=multer({storage});

export default upload;
```

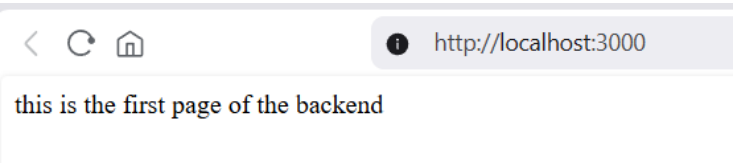
18. Postman test: now the name stays and it's no longer a random id



19. Moving on, create public folder in the backend folder, and create index.html file, write anything simple.

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    this is the first page of the backend
</body>
</html>
```

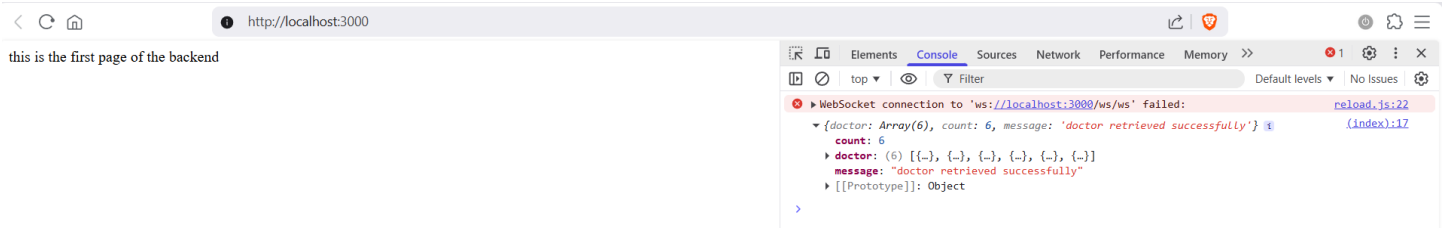
20. In server.js : app.**use**(express.**static**("public"));
21. Now by default, index.html view is displayed in the browser cause it's named index.html



22. .result and then press tab to create a div with class result
23. Let's try rendering the drs

```
<body>
    this is the first page of the backend
    <br>
    <div class="result">
        <script>
            let result  =   document.querySelector(".result");
            async function loadData(){
                const data=await fetch("http://localhost:3000/doctor/");
                const result=await data.json();
                console.log(result);
            }
            loadData();
        </script>
    </div>
</body>
```

24. Response in the browser



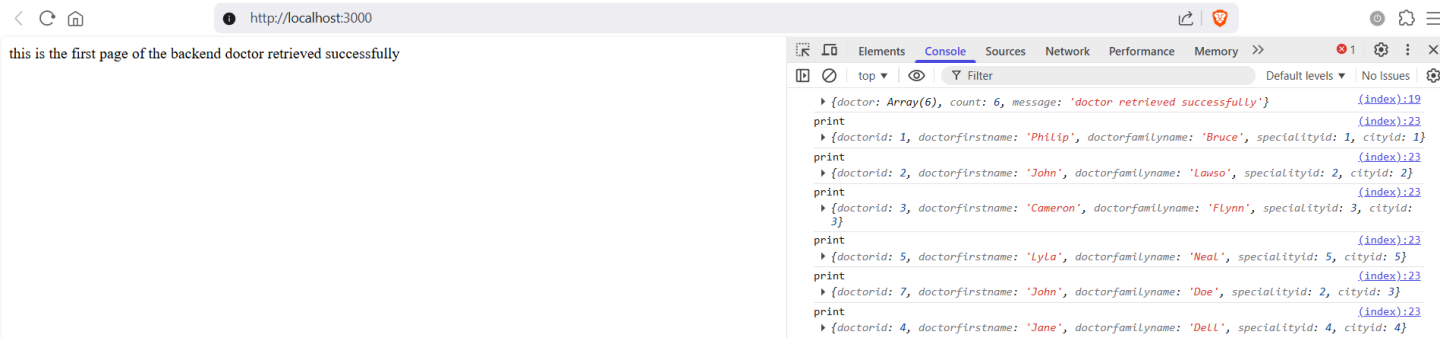
25. Now let's try displaying the message from this result in a span

```
<span id="answer"></span>
<br>
<div class="result">
    <script>
        let result  =   document.querySelector(".result");
        let answer  =   document.querySelector("#answer");
        async function loadData(){
            const data=await fetch("http://localhost:3000/doctor/");
            const result=await data.json();
            console.log(result);
            answer.textContent = result.message;
        }
        loadData();
    </script>
</div>
```

26. Now let's try catching the drs array of objects and displaying them, for that we need a loop

```
<script>
    let result  =   document.querySelector(".result");
    let answer  =   document.querySelector("#answer");
    async function loadData(){
        const data=await fetch("http://localhost:3000/doctor/");
        const result=await data.json();
        console.log(result);
        answer.textContent = result.message;
        // result is the JSON response from the server, which contains a property called doctor.
        // We are assigning that property to the doctors variable.
        const doctors = result.doctor;
        doctors.forEach((dr) => {
            console.log("print", dr);
        });
    }
    loadData();
</script>
```

27. Below is the drs in the console.



Friday, February 13, 2026 (Absent)

1. Error codes:
   - 200 OK: Request succeeded.
   - 201 Created: Resource created, typical for POST/PUT.
   - 204 No Content: Request processed, no content returned.
   - 301/302: Permanent or temporary redirection.
   - 304 Not Modified: Resource not modified, used for caching.
   - 400 Bad Request: Invalid syntax or data.
   - 401 Unauthorized: Authentication required.
   - 403 Forbidden: Authenticated, but lacking permission.
   - 404 Not Found: Resource not found.
   - 405 Method Not Allowed: Unsupported HTTP method.
   - 429 Too Many Requests: Rate limit exceeded.
   - 500 Internal Server Error: Unexpected server error.
   - 502 Bad Gateway: Invalid response from upstream server.
   - 503 Service Unavailable: Server overloaded or down for maintenance.
   - 504 Gateway Timeout: Upstream server timed out.

2. React Core Concepts Explained
    1. Component: A component is just a **function that returns UI**.

```
function Greeting() {
  return <h1>Hello!</h1>
}
```

That's it. React apps are just components inside components.

    2. useState: Stores a value that, **when it changes, re-renders the component**.

```
function Counter() {
  const [count, setCount] = useState(0)

  return <button onClick={() => setCount(count + 1)}>{count}</button>
}
```

Rule of thumb: if changing a value should update what the user sees → use state.

    3. useEffect: Runs code **after the component renders**. Used for side effects ; things outside the UI like fetching data, subscriptions, timers.

```
useEffect(() => {
  console.log("I run after every render")
})

useEffect(() => {
  console.log("I run once, on mount")
}, [])

useEffect(() => {
  console.log("I run when `count` changes")
}, [count])
```

The `[]` at the end controls *when* it runs. That's the dependency array.

    4. useRef: Stores a value that **persists across renders but doesn't cause a re-render when changed**. Also used to directly access a DOM element.

```
// Access a DOM element
const inputRef = useRef(null)
<input ref={inputRef} />
inputRef.current.focus() // direct DOM access

// Store a value without triggering re-render
const timerRef = useRef(null)
timerRef.current = setTimeout(...)
```

Think of it as a box you can put anything in and it stays there ; but React doesn't watch it.

---

    5. .map(): Not a hook, just JavaScript. Used to **turn an array of data into an array of components**.

```
const users = ["Alice", "Bob", "Carol"]

return (
  <ul>
    {users.map((user) => (
      <li key={user}>{user}</li>
    ))}
  </ul>
)
```

The `key` prop is required ; it helps React track which item is which.

6. Quick mental model

Hook/Concept	Think of it as…
Component	A lego brick
useState	Memory that updates the screen
useEffect	"Do this after rendering"
useRef	A sticky note React ignores
.map()	Stamp the same template per item

- **loadData()** ; fetches all doctors from the backend and displays them on the page as a list.
- **loadCity()** ; fetches all cities and fills the city dropdown in the form.
- **loadSpeciality()** ; fetches all specialities and fills the speciality dropdown in the form.
- **frm.addEventListener("submit",…)** ; listens for the form submission, sends the new doctor data to the backend, then reloads the list.
- **document.querySelector("#answer") ;** grabs the element with id `answer` from the HTML so you can change its content in JS.
- **document.createElement("p")** ; creates a new HTML element in memory, not on the page yet.
- **element.classList.add("spn")** ; adds the CSS class `spn` to the element.
- **element.textContent** ; sets the text inside an element.
- **parent.appendChild(child) ;** puts the child element inside the parent, this is when it actually appears on the page.
- **e.preventDefault()** ; stops the form from refreshing the page on submit, which is the default browser behavior.
- **opt.setAttribute("value", city.cityid)** ; sets the `value` attribute on the option element, so when you select it the form knows the id not just the name.

3. we created another middlware file called auth.js, this file will be responsible for checking if authenticated and if authorized. Below is the full code.
4. In auth() we check if the auth header exist, if no throw error, if yes attach the user role

5. In isAdmin() we check if the role is admin then we grant the privilege, if not, throw error.

```
// Middleware to check if user is authenticated
// Reads the 'auth' header and attaches user info to the request
Qodo: Test this function
function auth(req, res, next) {
    const role = req.headers.auth

    // If no auth header, reject the request
    if (!role)
        return res.status(401).json({msg: "you are not authenticated"})

    // Attach user role to request so next middleware can use it
    req.user = {role: role}

    // Pass control to next middleware
    next()
}

// Middleware to check if authenticated user is an admin
// Must be used AFTER auth middleware
Qodo: Test this function
function isAdmin(req, res, next) {
    // Check if user role is admin (req.user was set by auth middleware)
    if (req.user.role !== "admin")
        res.status(403).json({msg: "only admin"})
    else
        next() // User is admin, allow request to proceed
}

export {auth, isAdmin}
```

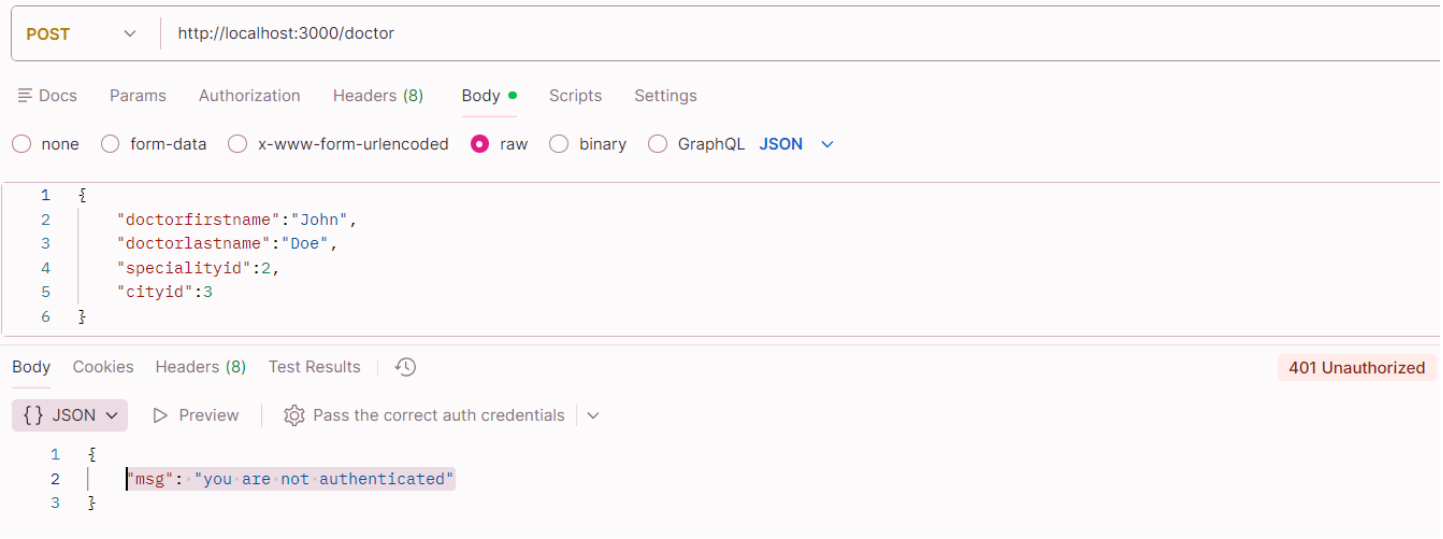
6. Working example of the later middleware, try adding it to the post doctor function in which we try to add a dr, first import it in doctor route file

```
import { auth, isAdmin } from "../middleware/auth.js"
```

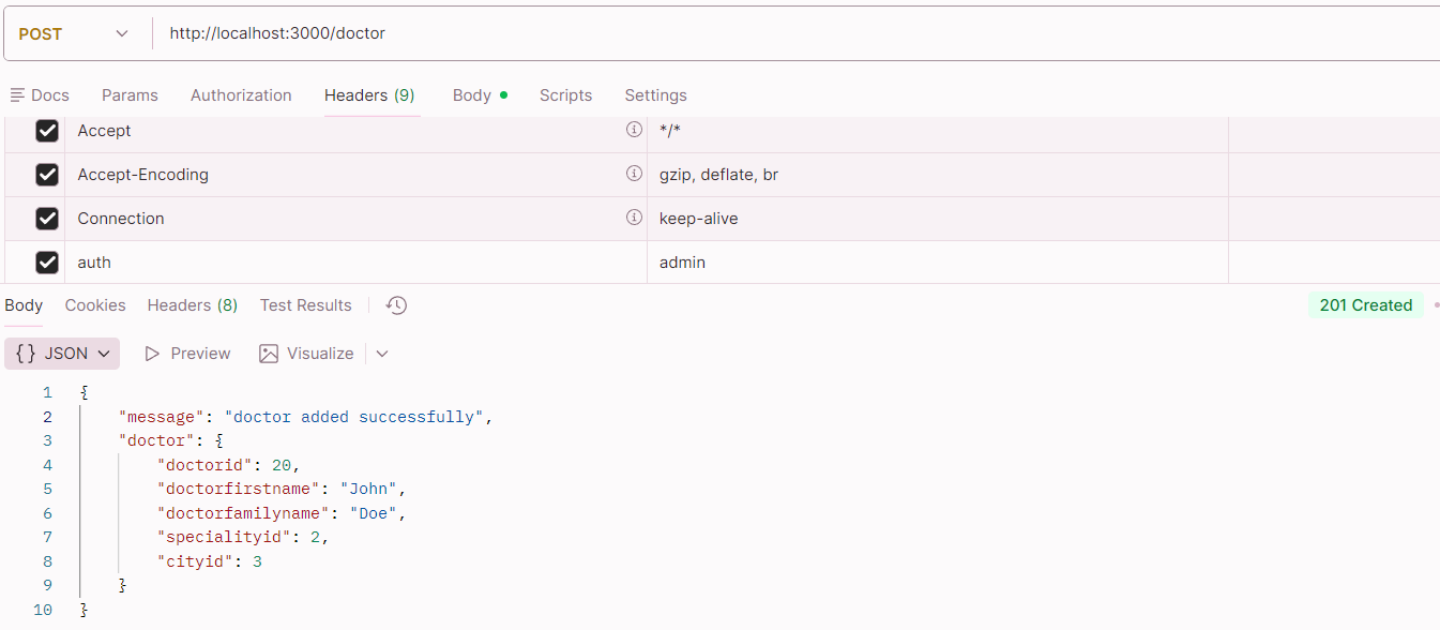
7. Then in the post method, call the middleware before the function that adds the dr

```
doctorRoute.post("/", auth, isAdmin, async(req,res)=>{
```

8. Now in postman if you try to add a dr without a role you will be denied access



The correct form, press on Header from the top bar in postman and add a header, auth and give it value admin



9. Now we are gonna try to connect these routes to index.html, for that am gonna edit the doctor route to return speciality and city names not ids cause editing the query here is easier

```
doctorRoute.get("/", async(req,res)=>{
    try{
    const doctor=await pool.query(`
        SELECT doctor.doctorid, doctor.doctorfirstname, doctor.doctorfamilyname, speciality.specialityname, city.cityname
        FROM doctor
        JOIN speciality ON doctor.specialityid = speciality.specialityid
        JOIN city ON doctor.cityid = city.cityid
    `);
    res.status(200).json({doctor: doctor.rows, count: doctor.rows.length, message: "doctor retrieved successfully"});
    }
    catch(err){
        res.status(500).json({msg:err.message,success:false});
    }
})
```

10. Before anything let's complete last session work, we are going to display the drs from the db, starting now these js functions should be all added in the script tag.
11. This function grabs all the doctors from the database using a fetch call to the backend, then it loops through each doctor and builds HTML elements (paragraphs and spans) to show their info like ID, name, speciality, and city. It clears out any old content first, converts the server's JSON response into something JavaScript can work with, then uses `createElement` and `appendChild` to dynamically add each doctor to the page so you see the full list without needing to refresh.

```
// Select elements to display messages and results
let a = document.querySelector("#answer")
let b = document.querySelector(".result")

// Fetches all doctors from backend and displays them on the page
async function loadData() {
    // Clear previous content
    b.innerHTML = ""

    // Fetch doctors from backend API
    const data = await fetch("http://localhost:3000/doctor/")
    const final = await data.json()
    console.log(final)

    // Display server message
    a.textContent = final.message
    console.log(final.doctor)

    // Get array of doctors from response
    let B = final.doctor

    // Loop through each doctor and create HTML elements
    B.forEach(element => {
        // Create a paragraph to hold all doctor info
        let par = document.createElement("p")

        // Create span for doctor ID
        let spanid = document.createElement("span")
        spanid.textContent = element.doctorid
        spanid.classList.add("spn") // Add CSS class for styling

        // Create span for first name
        let spanname = document.createElement("span")
        spanname.textContent = element.doctorfirstname
        spanname.classList.add("spn")
```

```
        // Create span for last name
        let spanfamily = document.createElement("span")
        spanfamily.textContent = element.doctorfamilyname
        spanfamily.classList.add("spn")

        // Create span for speciality
        let spanspec = document.createElement("span")
        spanspec.textContent = element.specialityname
        spanspec.classList.add("spn")

        // Create span for city
        let city = document.createElement("span")
        city.textContent = element.cityname
        city.classList.add("spn")

        // Add all spans to the paragraph
        par.appendChild(spanid)
        par.appendChild(spanname)
        par.appendChild(spanfamily)
        par.appendChild(spanspec)
        par.appendChild(city)

        // Add paragraph to the result div
        b.appendChild(par)
    });
}
// Load doctors when page loads
loadData()
```

12. Expected Result:

doctor retrieved successfully

Hello, this is a list of doctors

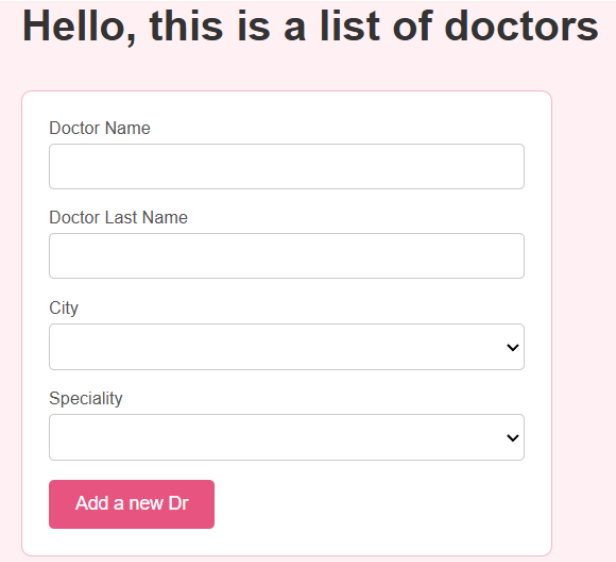
1	Philip	Bruce	Psychiatry	Beirut
2	John	Lawso	Dermatology	Saida
3	Cameron	Flynn	Radiology	Nabatiye
5	Lyla	Neal	Cardiology	Sour
7	John	Doe	Dermatology	Nabatiye
4	Jane	Dell	Gynecologist	Zahle
20	John	Doe	Dermatology	Nabatiye

13. In index.html we want to take data from the admin so he can add a dr, for that we want to create a form, below is simple html code for the fields that must be entered

```
<h1 class="maintitle">Hello, this is a list of doctors</h1>

<span id="answer"></span>
<div class="result"></div>

<!-- Form to add new doctor -->
<form id="frm">
    <div class="input-control">
        <label for="drname">Doctor Name</label>
        <input type="text" id="drname">
    </div>

    <div class="input-control">
        <label for="">Doctor Last Name</label>
        <input type="text" id="drfamily">
    </div>

    <div class="input-control">
        <label for="">City</label>
        <!-- Options will be added dynamically via JavaScript -->
        <select name="" id="city"></select>
    </div>

    <div class="input-control">
        <label for="">Speciality</label>
        <!-- Options will be added dynamically via JavaScript -->
        <select name="" id="spec"></select>
    </div>
    <button>Add a new Dr</button>
</form>
```

Expected frontend if you have styles.css linked (code is after this part)

<link *rel*="stylesheet" *href*="style.css">



14. The styles.css sheet used:

```css
* {
    box-sizing: border-box;
    margin: 0;
    padding: 0;
}

body {
    font-family: sans-serif;
    padding: 20px;
    background: #fff0f3;
    color: #333;
}

.maintitle {
    margin-bottom: 20px;
}

/* Each doctor row */
.result p {
    background: white;
    padding: 10px 14px;
    margin-bottom: 8px;
    border-radius: 6px;
    border: 1px solid #f7b8c8;
}

.spn {
    display: inline-block;
    margin-right: 16px;
}

/* Form */
form {
    background: white;
    padding: 20px;
    border-radius: 8px;
    border: 1px solid #f7b8c8;
    max-width: 400px;
    margin-top: 30px;
}

.input-control {
    display: flex;
    flex-direction: column;
    margin-bottom: 14px;
}

.input-control label {
    font-size: 13px;
    margin-bottom: 4px;
    color: #555;
}

.input-control input,
.input-control select {
    padding: 8px;
    border: 1px solid #ccc;
    border-radius: 4px;
    font-size: 14px;
}

button {
    padding: 10px 20px;
    background: #e75480;
    color: white;
    border: none;
    border-radius: 4px;
    cursor: pointer;
    font-size: 14px;
}

button:hover {
    background: #c9365f;
}

.wlc {
    margin-top: 12px;
    color: green;
}
```

15. Now let's fill the city dropdown

```
// Fetches cities from backend and populates the city dropdown
async function loadCity(){
    const city = await fetch("/city")
    const result = await city.json()
    console.log(result)

    // Select the city dropdown element
    let cityList = document.querySelector("#city")

    // Loop through cities and create option elements
    result.city.forEach(city => {
        let opt = document.createElement("option")
        // Set the value attribute to city ID
        opt.setAttribute("value", city.cityid)
        // Set the visible text to city name
        opt.textContent = city.cityname
        // Add option to the select element
        cityList.appendChild(opt)
    })
}
// Load cities when page loads
loadCity()
```

16. Now let's fill the speciality dropdown

```
// Fetches specialities from backend and populates the speciality dropdown
async function loadSpeciality(){
    const spec = await fetch("/speciality")
    const result = await spec.json()

    // Select the speciality dropdown element
    let specList = document.querySelector("#spec")

    // Loop through specialities and create option elements
    result.speciality.forEach(s => {
        let opt = document.createElement("option")
        opt.setAttribute("value", s.specialityid)
        opt.textContent = s.specialityname
        specList.appendChild(opt)
    })
}
// Load specialities when page loads
loadSpeciality()
```

17. Now the form is ready but it's not connected to the backend yet so we have to do that in order for us to send the information, pay attention to send auth : admin in the header of the request as to grant access.
18. The code below grabs all the form input fields (doctor name, family name, city, and speciality), then listens for when the user submits the form. When they click submit, it prevents the page from refreshing, collects all the input values, and sends a POST request to the backend with the doctor data in JSON format along with an "admin" authentication header. After the server adds the new doctor to the database and responds, it calls loadData() again to refresh the list on the page so the newly added doctor shows up immediately without needing a manual refresh.

```
// Select form input elements
let drname = document.querySelector("#drname")
let drfamily = document.querySelector("#drfamily")
let drcity = document.querySelector("#city")
let drspec = document.querySelector("#spec")

// Select the form element
let frm = document.querySelector("#frm")

// Add event listener for form submission
frm.addEventListener("submit", async (e) => {
    // Prevent page refresh on form submit
    e.preventDefault()
    alert(drname.value + " " + drfamily.value + " " + drcity.value)

    // Send POST request to add new doctor
    // fetch is used to send a POST request to the server with the data from the form.
    // The server will then add the new doctor to the database and return a response.
    // We are then displaying that response on the page and reloading the data to show the new doctor in the list.
    const addDoctor = await fetch("/doctor", {
        method: "post",
        headers: {
            "content-type": "application/json",
            "auth": "admin" // Authentication header
        },
        // Convert form data to JSON string
        body: JSON.stringify({
            doctorfirstname: drname.value,
            doctorlastname: drfamily.value,
            specialityid: drspec.value,
            cityid: drcity.value
        })
    })

    // Get response as text
    const answer = await addDoctor.text()
    let wlc = document.querySelector(".wlc")
    wlc.innerHTML = "<h3>answer</h3>"
    // alert(answer)

    // Reload doctor list to show newly added doctor
    loadData()
})
```

<p *class*="wlc"></p> existed after the <form> in the html code before the js.

Tuesday, February 17, 2026

1. In this session we will work with react, so first ofc we need a react project, cd the frontend folder in the terminal then write: npm create vite@latest
2. Then answer the few questions that come after like choosing React and javasacript and naming the project.
3. In the backend folder, download cors so the fronetend can talk wih the backend, npm install cors
4. Copy styles.css into index.css

5. Create form.jsx component and create list.jsx then Clear App.jsx and fill it with the following as we will be rendering the following components.

```
import Form from "./form";
import List from "./list";

Qodo: Test this function
export default function App() {
  // Form is a function that returns a form element. We are importing it from the form.jsx file and then rendering it in the App component.
  return <div><Form /><List /></div>;
}
```

6. In fact writing react is similar to html but with key changes like below\
    a. Class becomes className
        i. In HTML you write class="box", but class is a reserved keyword in JavaScript, so JSX uses className="box" instead.
    b. for becomes htmlFor
        i. so on labels you write htmlFor="inputId" instead of for="inputId".
    c. Self-closing tags are required
        i. In HTML you can write <input> and leave it open. In JSX every tag must be properly closed, so it's <input /> with the slash.
    d. Event names are camelCase
        i. HTML uses lowercase like onclick, onchange, onsubmit. JSX uses camelCase: onClick, onChange, onSubmit.
    e. JavaScript goes inside curly braces {}
        i. In HTML, everything between tags is treated as text. In JSX, if you want to run actual JavaScript ; a variable, a function, a condition ; you wrap it in {}. For example {city.cityname} or {cities.map(...)}.
    f. hidden attribute
        i. In HTML you might write selected disabled. In JSX boolean attributes work the same way, but JSX is stricter ; you can also write them as hidden={true} explicitly.
    g. Style is an object, not a string
        i. In HTML: style="color: red". In JSX: style={{ color: "red" }} ; an object with camelCase properties, wrapped in double curly braces (one for JS mode, one for the object itself).

7. **Let's start by form.jsx**
    a. **Setting up the data we need**

    When the form loads for the first time, it immediately goes and fetches two lists from the backend: a list of cities and a list of specialities. These get stored in memory so React can use them to populate the dropdowns. This happens automatically once, right when the page opens, thanks to useEffect.

    b. **Keeping track of what the user types**

    Every field in the form ; the doctor's first name, last name, selected city, and selected speciality ; is linked to a piece of state. State is just React's way of remembering values.

When the user types or selects something, the state updates instantly, and React keeps the form in sync. This is called a "controlled component" ; the form doesn't manage itself, React does.

c. Submitting the form

When the admin clicks the button, the form collects all those stored values and sends them to the backend as a POST request ; essentially saying "here's a new doctor, please save them." The backend responds with a message (success or error), which gets displayed on screen and also shown as an alert popup.

d. Resetting after submission

Once the doctor is added, the form cleans itself up ; all fields go back to empty, and the cursor jumps back to the first field automatically so the admin can immediately start entering the next doctor without touching the mouse.

```jsx
import { useState, useEffect, useRef} from "react";

Qodo: Test this function
export default function Form() {
    // useState creates variables that React watches - when they change, the UI updates automatically
    const [message, setMessage] = useState("");
    const [cities, setCities] = useState([]);
    const [name, setName] = useState("");
    const [familyName, setFamilyName] = useState("");
    const [specs, setSpecs] = useState([]);
    const [selectedCity, setSelectedCity] = useState(0);
    const [selectedSpec, setSelectedSpec] = useState(0);

    // useRef creates a reference to a DOM element (the input field in this case)
    const nameRef = useRef();

    // Fetch cities from backend and store in state
    async function loadCity() {
        const city = await fetch("http://localhost:3000/city");
        const result = await city.json();
        setCities(result.city); // Updates cities state, triggers re-render
    }

    // Fetch specialities from backend and store in state
    async function loadSpec(){
        const spec = await fetch("http://localhost:3000/speciality");
        const result = await spec.json();
        setSpecs(result.speciality);
    }

    // useEffect runs once when component first loads (empty [] means "run once")
    useEffect(() => {
        loadCity();
        loadSpec();
    }, []);

    // Handles form submission - adds new doctor to database
    async function addDoctor(e){
        e.preventDefault(); // Stops page from refreshing
```

```
// Handles form submission - adds new doctor to database
async function addDoctor(e){
    e.preventDefault(); // Stops page from refreshing

    // Send POST request to backend with form data
    const response = await fetch("http://localhost:3000/doctor", {
        method: "POST",
        headers: {
            "Content-Type": "application/json",
            "auth": "admin"
        },
        body: JSON.stringify({
            doctorfirstname: name,
            doctorlastname: familyName,
            specialityid: parseInt(selectedSpec),
            cityid: parseInt(selectedCity)
        })
    });

    const result = await response.json();
    setMessage(result.message);

    // Reset form fields back to empty/default
    setName("");
    setFamilyName("");
    setSelectedCity(0);
    setSelectedSpec(0);
    nameRef.current.focus(); // Put cursor back in name field
    alert(result.message);
}
```

```jsx
return (
    <div>
        <form action="">
            <div className="input-control">
                <label htmlFor="drname">Doctor Name</label>
                {/* value connects input to state, onChange updates state when user types */}
                <input type="text" id="drname" value={name} onChange={(e) => setName(e.target.value)} autoFocus ref={nameRef}/>
            </div>

            <div className="input-control">
                <label htmlFor="drfamily">Doctor Last Name</label>
                <input type="text" id="drfamily" value={familyName} onChange={(e) => setFamilyName(e.target.value)} />
            </div>

            <div className="input-control">
                <label htmlFor="city">City</label>
                <select id="city" value={selectedCity} onChange={(e) => setSelectedCity(e.target.value)}>
                    <option value="0" hidden>select a city</option>
                    {/* map loops through cities array and creates an option for each */}
                    {cities.map((city) => (
                        <option value={city.cityid} key={city.cityid}>
                            {city.cityname}
                        </option>
                    ))}
                </select>
            </div>

            <div className="input-control">
                <label htmlFor="spec">Speciality</label>
                <select id="spec" value={selectedSpec} onChange={(e) => setSelectedSpec(e.target.value)}>
                    <option value="0" hidden>select a speciality</option>
                    {specs.map((spec) => (
                        <option value={spec.specialityid} key={spec.specialityid}>
                            {spec.specialityname}
                        </option>
                    ))}
                </select>
            </div>
            <button onClick={addDoctor}>Add a new Dr</button>
            {/* Only show message if it exists (&&) */}
            {message && <p>{message}</p>}
        </form>
    </div>
);
```

8. Same for list.jsx but we will change it later and use props:

    a. When the page loads

    Just like the form, useEffect triggers once when the component first appears. It calls loadDoctors, which goes to the backend, grabs all the doctors, and stores them in state; one array for the doctors list, one string for the response message.

    b. Displaying the list

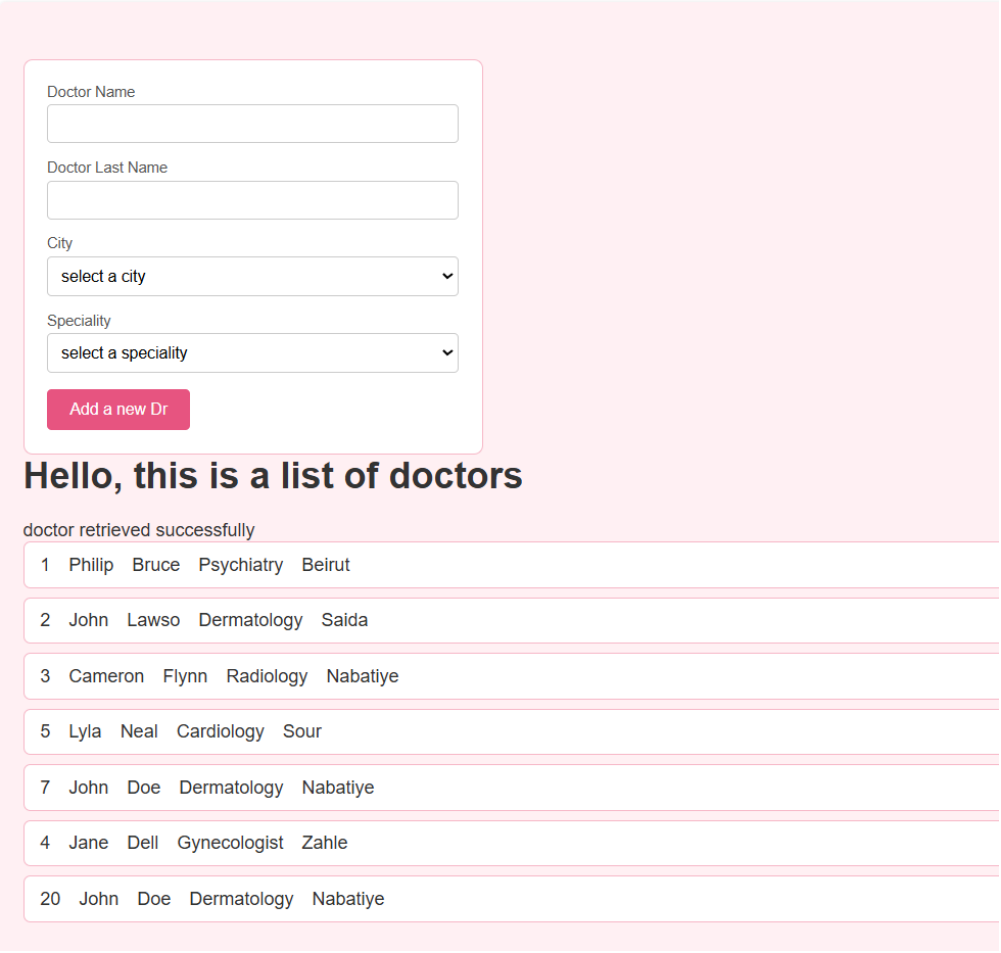
    Once the doctors are loaded into state, React re-renders and loops through the array using .map(). For each doctor it creates a <p> tag containing several elements; one for the ID, first name, family name, speciality, and city. Each <p> gets a unique key using the doctor's ID, which React needs internally to keep track of each item in the list efficiently.

    c. The message

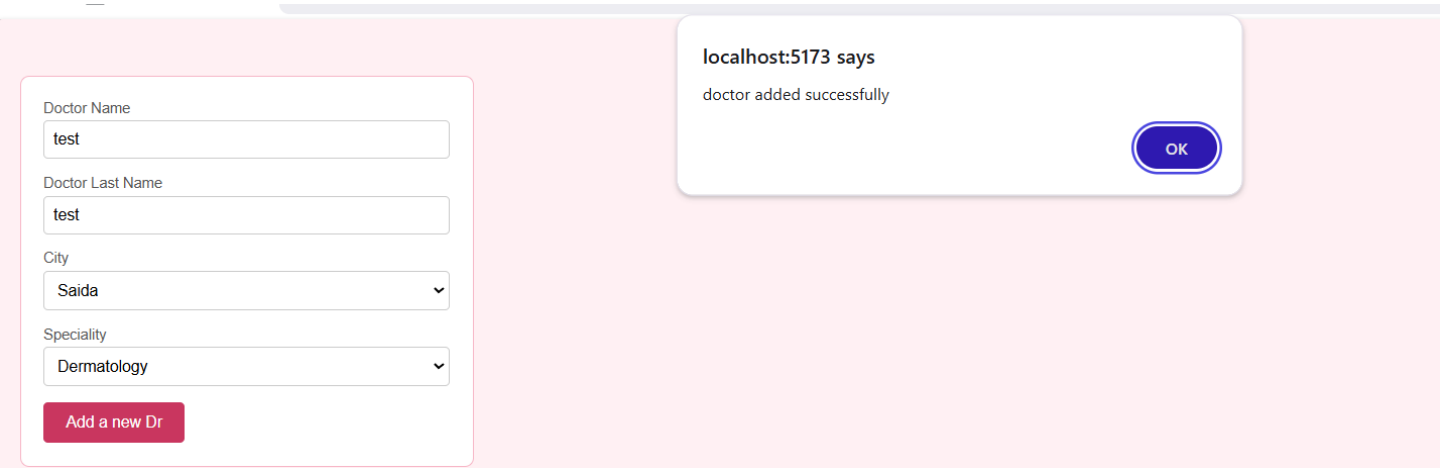
    The backend sends back a message alongside the data ("doctors fetched successfully"). It gets displayed in a above the list; useful for showing errors too if the fetch fails.

```jsx
import { useState, useEffect } from "react";

Qodo: Test this function
export default function List() {
    // doctors holds the array, message holds the backend response text
    const [doctors, setDoctors] = useState([]);
    const [message, setMessage] = useState("");

    // Fetches all doctors from the backend and stores them in state
    async function loadDoctors() {
        const data = await fetch("http://localhost:3000/doctor/");
        const result = await data.json();
        setMessage(result.message);
        setDoctors(result.doctor);
    }

    // Runs loadDoctors once when the component first loads
    useEffect(() => {
        loadDoctors();
    }, []);

    return (
        <div>
            <h1 className="maintitle">Hello, this is a list of doctors</h1>
            <span>{message}</span>
            <div className="result">
                {/* Loop through doctors array and render a row for each one */}
                {doctors.map((dr) => (
                    // key helps React track each item in the list
                    <p key={dr.doctorid}>
                        <span className="spn">{dr.doctorid}</span>
                        <span className="spn">{dr.doctorfirstname}</span>
                        <span className="spn">{dr.doctorfamilyname}</span>
                        <span className="spn">{dr.specialityname}</span>
                        <span className="spn">{dr.cityname}</span>
                    </p>
                ))}
            </div>
        </div>
    );
}
```

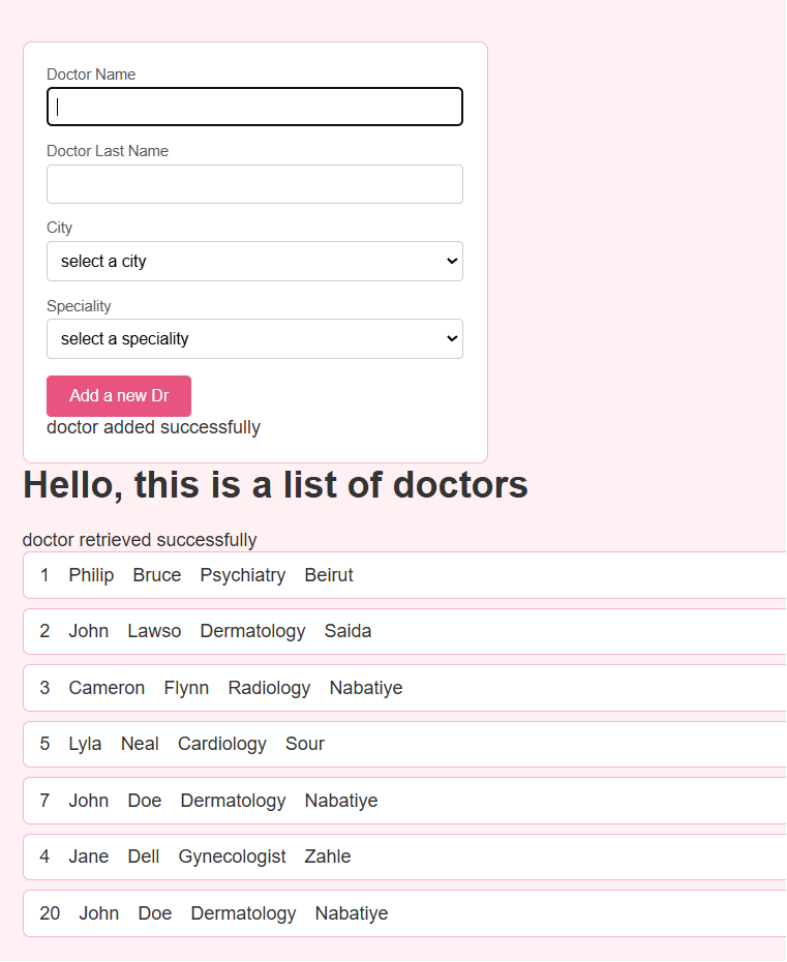
9. Final version:



Fill data and then press the add new dr button to get this alert.



After adding the dr, notice how the form is emptied again, the list didn't update and wouldn't unless you refresh the page, this is an issue for later fix when we learn props.



Wednesday, February 18, 2026

1. We will now do routing in react.
2. **The entry point (main.jsx)**
    a. This is where your entire React app starts. createRoot finds the single in your HTML file and tells React to take over that div and render everything inside it. Your whole app lives inside that one div.
    b. BrowserRouter is wrapped around everything because it's what enables routing in your app — it watches the URL in the browser and lets React respond to changes in it.
3. **The App component**
    a. This is the shell of your app. It has two parts: a persistent header that's always visible, and a dynamic area that changes based on the URL.
    b. The <Link> tags work like normal <a> tags but without refreshing the page; clicking them just updates the URL silently, which is what makes it feel like a real app.
    c. <Routes> is the decision maker. It looks at the current URL and decides which component to show. If the URL is /add, it renders the Form. If it's /list, it renders the List. Only one renders at a time.
    d. The big picture

e. So the flow is: browser opens → main.jsx boots React into the page → App renders the header and sets up the routes → user clicks a link → URL changes → the matching component appears, no page reload. That's client-side routing in a nutshell.

4. App.jsx:

```jsx
import Form from "./form";
import List from "./list";
import {Link, Routes, Route} from "react-router-dom";

Qodo: Test this function
export default function App() {
  // Form is a function that returns a form element. We are importi
  // return <div><Form /><List /></div>;

  return(
    <>
    <header>
      <Link to="/add">add new doctor</Link>   
      <Link to="/list">list of doctors</Link>
    </header>
    <Routes>
      <Route path="/add" element={<Form />}/>
      <Route path="/list" element={<List />}/>
    </Routes>
    </>
  );
}
```

5. Main.jsx:

```jsx
import { createRoot } from 'react-dom/client'
import './index.css'
import App from './App.jsx'
import { BrowserRouter } from 'react-router-dom';

createRoot(document.getElementById('root')).render(
  <BrowserRouter>
    <App />
  </BrowserRouter>,
)
```

6. Now we will add the delete button and link it to the backend, for that we need helper function handledelete(x) that takes the dr id and send it to the delete route in the backend

```jsx
async function handleDelete(x){
    const response = await fetch(`http://localhost:3000/doctor/${x}`, {
        method: "Delete"
    });
    console.log(await response.json())
    alert(x)
};
```

7. And here in the list of drs we added the delete button and called the handle delete function

```
return (
    <div>
        <h1 className="maintitle">Hello, this is a list of doctors</h1>
        <span>{message}</span>
        <div className="result">
            {/* Loop through doctors array and render a row for each one */}
            {doctors.map((dr) => (
                // key helps React track each item in the list
                <p key={dr.doctorid}>
                    <span className="spn">{dr.doctorid}</span>
                    <span className="spn">{dr.doctorfirstname}</span>
                    <span className="spn">{dr.doctorfamilyname}</span>
                    <span className="spn">{dr.specialityname}</span>
                    <span className="spn">{dr.cityname}</span>
                    <button onClick={()=>handleDelete(dr.doctorid)}>delete</button>
                </p>
            ))}
        </div>
    </div>
);
```

8. Adding loadDoctors at the end of the handle delete function means that whenever you delete the page will bring the data from the db again and display

```
async function handleDelete(x){
    const response = await fetch(`http://localhost:3000/doctor/${x}`, {
        method: "Delete"
    });
    console.log(await response.json())
    alert(x)
    loadDoctors()
};
```

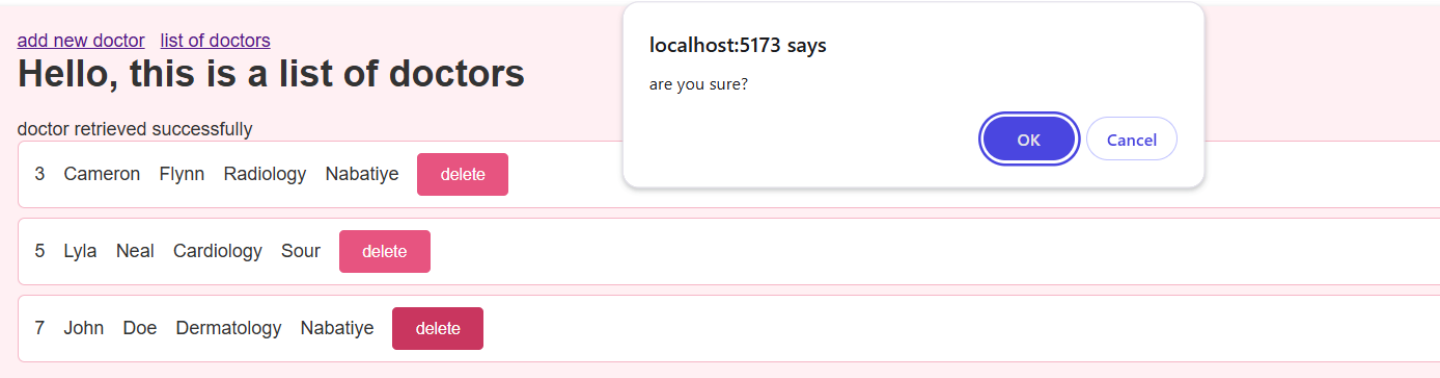
9. But there's a better way and that is to call the filter function. Here even though we deleted the dr from th edb, it still exist in the state, thus we can use the filter function on the state, we make a newList, and filter the old one from the existing state, then set the state. This way we update the state to match the db without actually having to go fetch everything again.

```
async function handleDelete(x){
    const response = await fetch(`http://localhost:3000/doctor/${x}`, {
        method: "Delete"
    });
    console.log(await response.json())
    alert(x)
    // loadDoctors()
    const newList = doctors.filter((d)=>{
        return d.doctorid!=x;
    })
    setDoctors(newList)
};
```

10. Now we will add an if statement so that the browser asks you if you want to delete before it actually deletes

```
async function handleDelete(x){
    if(window.confirm("are you sure?")){
        const response = await fetch(`http://localhost:3000/doctor/${x}`, {
            method: "Delete"
        });
        console.log(await response.json())
        alert(x)
        // loadDoctors()
        const newList = doctors.filter((d)=>{
            return d.doctorid!=x;
        })
        setDoctors(newList)
    }
};
```

11. Result in browser:



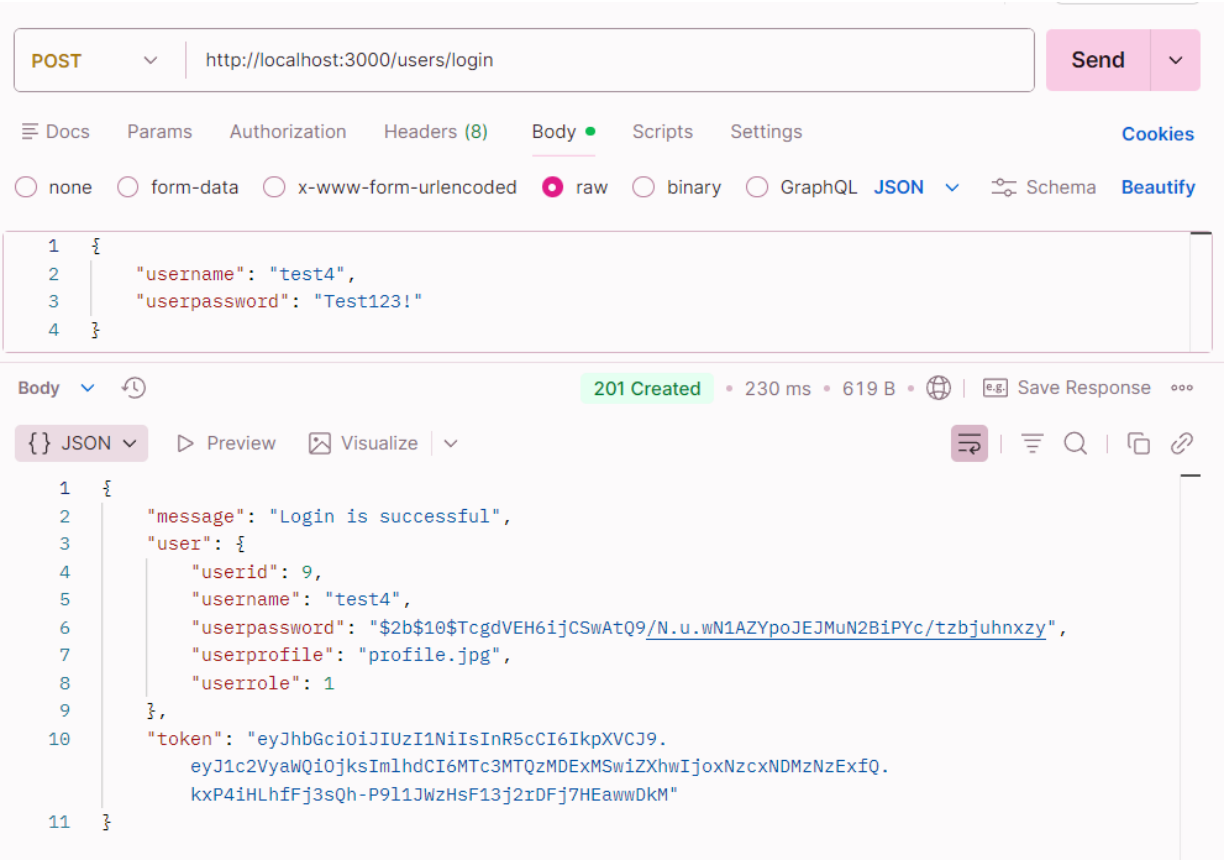
12. One way to secure your website is jwt tokens so first npm install jsonwebtokens in the backend folder
13. Then import jwt from "jsonwebtoken"; in the userRoute
14. Go to the login route add the part directly before the sending a response

```
const token = jwt.sign(
    {
        userid:check.rows[0].userid,
        userrole:check.rows[0].userrole
    },
    "edgvhgvbujjugvftgdxsesxxftyghb",
    {expiresIn: "1hr"}
)

// in the db alter table users add column userrole int default 1
```

15. Then show the token in the postman response by adding it to the json object

```
res.status(201).json({message: "Login is successful", user: loginuser.rows[0], token: token});
```

16. Postman Result: you can view now the token



17. Now let’s create the login form

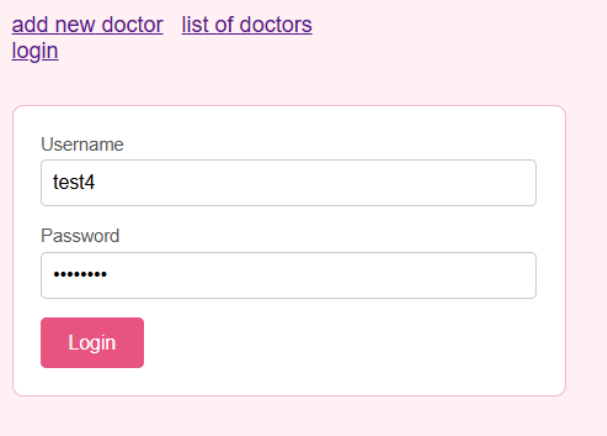
```jsx
import { useState } from "react";
import { useNavigate } from "react-router-dom";

Qodo: Test this function
export default function Login() {
    // usestates
    const [username, setUsername] = useState("");
    const [password, setPassword] = useState("");
    const [message, setMessage] = useState("");
    const navigate = useNavigate();

    async function handleLogin(e) {
        e.preventDefault(); //prevent page reload

        // send the result to the login route acording to what the backend expects
        const response = await fetch("http://localhost:3000/users/login", {
            method: "POST",
            headers: { "Content-Type": "application/json" },
            body: JSON.stringify({ username, userpassword: password })
        });

        const result = await response.json();
        //since the result is json object we have to nitpick from it
        if(result.token) {
            // Save token so we can use it in future requests
            localStorage.setItem("token", result.token);
        }

        if (response.status === 201) { //in my case the backend returns 201, so it depends
            navigate("/list");
        } else {
            console.log(result.message);
        }

        // Reset form fields back to empty/default
        setUsername("");
        setPassword("");
        setMessage("");
    }
```

```jsx
    return (
        <div>
            <form>
                <div className="input-control">
                    <label htmlFor="username">Username</label>
                    {/* value connects input to state, onChange updates state when user types */}
                    <input type="text" id="username" value={username} onChange={(e) => setUsername(e.target.value)} />
                </div>

                <div className="input-control">
                    <label htmlFor="password">Password</label>
                    <input type="password" id="password" value={password} onChange={(e) => setPassword(e.target.value)} />
                </div>

                <button onClick={handleLogin}>Login</button>
                {message && <p>{message}</p>}
            </form>
        </div>
    );
```

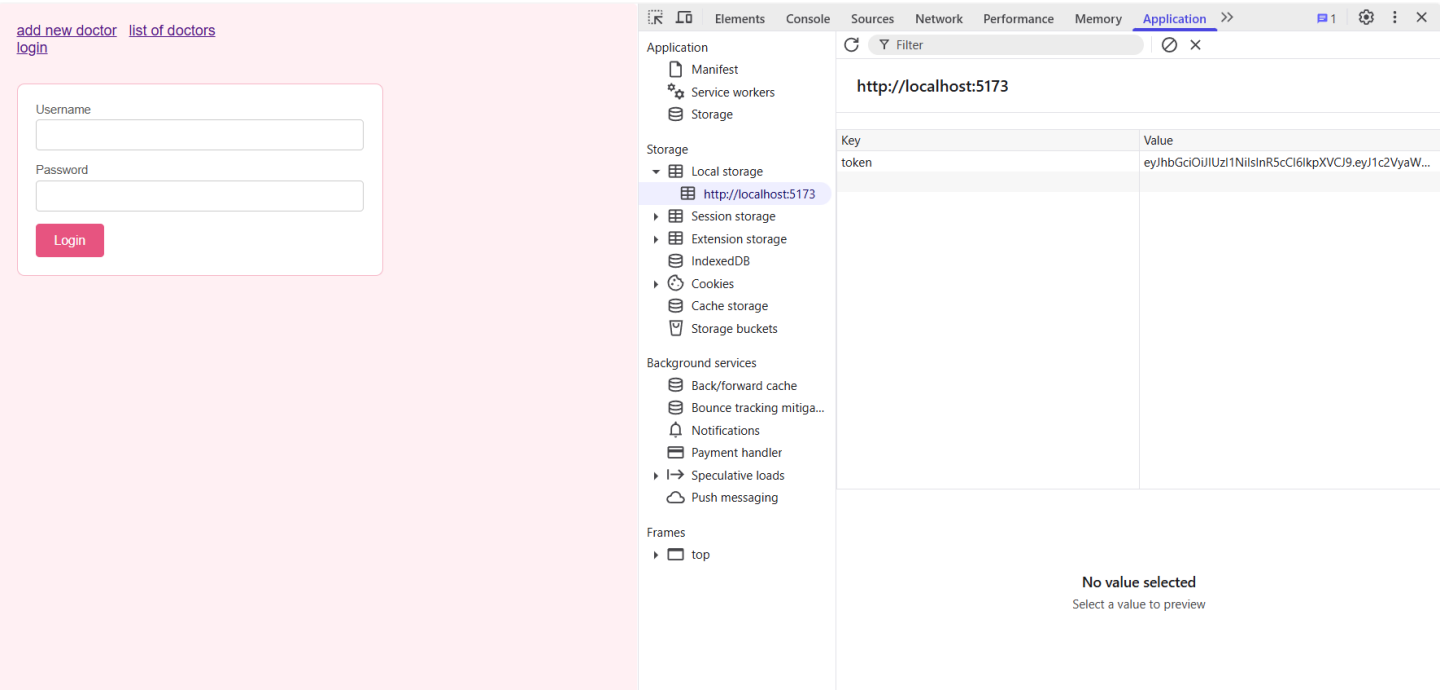
18. Below I added the login as a route, it doesn't make sense but for the sake of testing rn

```jsx
import Form from "./form";
import List from "./list";
import Login from "./login";
import {Link, Routes, Route} from "react-router-dom";

Qodo: Test this function
export default function App() {
  // Form is a function that returns a form element. We a
  // return <div><Form /><List /></div>;

  return(
    <>
    <header>
      <Link to="/add">add new doctor</Link>   
      <Link to="/list">list of doctors</Link>
      <br/>
      <Link to="/login">login</Link>
    </header>
    <Routes>
      <Route path="/add" element={<Form />}/>
      <Route path="/list" element={<List />}/>
      <Route path="/login" element={<Login />}/>
    </Routes>
    </>
  );
}
```

19. Fill the data and then submit the login form



20. Now if you press F12 and navigate to Application then find the local storage of this website, you'll find the token.



21. Now the login will be the first page the user views when they enter the website (using navigate)

```
import Form from "./form";
import List from "./list";
import Login from "./login";
import {Link, Routes, Route, Navigate} from "react-router-dom";

Qodo: Test this function
export default function App() {
  // Form is a function that returns a form element. We are impor
  // return <div><Form /><List /></div>;

  return(
    <>
    <header>
      <Link to="/add">add new doctor</Link>   
      <Link to="/list">list of doctors</Link>
      <br/>
      <Link to="/login">login</Link>
    </header>
    <Routes>
      <Route path="/" element={<Navigate to="/login" />}/>
      <Route path="/add" element={<Form />}/>
      <Route path="/list" element={<List />}/>
      <Route path="/login" element={<Login />}/>
    </Routes>
    </>
  );
}
```

22. Mission: add logout button that clears your token from local storage, and also modify auth.js to be capable of verifying using tokens and int roles(if you do not have that column in the users table make sure to alter the table in pg admin and default it to 1), then restrict dr deletions to only admins which role is 1. For that let's start by modifying auth.js:

```
import jwt from "jsonwebtoken";

// auth middleware: runs before protected routes to verify the user has a valid token
Qodo: Test this function
function auth(req, res, next) {
    // grab the token from the request header (sent by the frontend)
    const token = req.headers.token;

    // if no token was sent, reject the request
    if (!token)
        return res.status(401).json({msg: "you are not authenticated"});

    try {
        // jwt.verify does two things:
        // 1. checks the token hasn't been tampered using the secret key
        // 2. decodes it and returns the payload we originally signed (userid, userrole)
        const decoded = jwt.verify(token, "edgvhgvbujjugvftgdxsesxxftyghb");

        req.user = decoded;
        next();
    } catch(err) {
        return res.status(401).json({msg: "invalid token"});
    }
}

// isAdmin middleware: runs after auth to check if the user has admin privileges
// req.user is already set by auth middleware above
Qodo: Test this function
function isAdmin(req, res, next) {
    if (req.user.userrole !== 1)
        return res.status(403).json({msg: "only admin"});
    next(); // user is admin, allow the request to proceed
}

export {auth, isAdmin}
```

23. Edit the delete route in doctor route so it passes by authentication(middleware) before processing further.

```
doctorRoute.delete("/:id", auth, isAdmin, async(req,res)=>{
```

24. No need to save the token in the login page then send it to the list page, why? Cause, localStorage is browser-wide storage, any component can access it at any time so we save the token on login and retrieve it whenever we need it (e.g. delete). Here we edit the delete request to send a token to the backend.

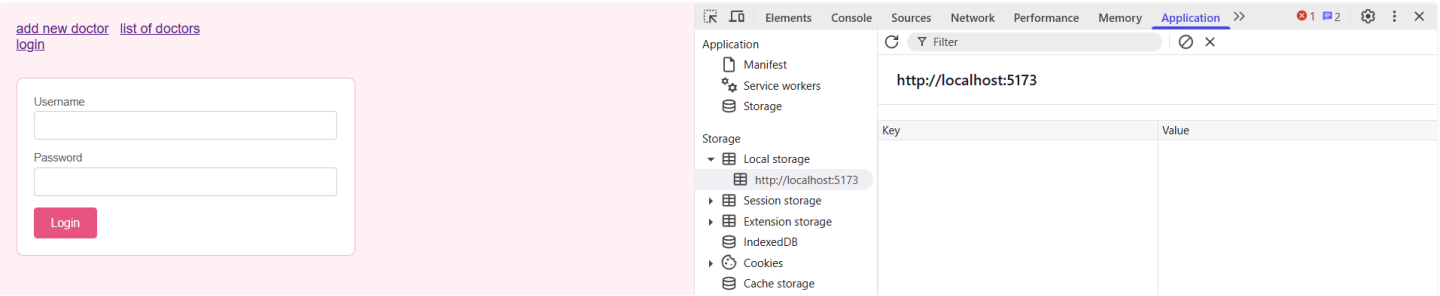
```
async function handleDelete(x){
    if(window.confirm("are you sure?")){
        const token = localStorage.getItem("token");
        const response = await fetch(`http://localhost:3000/doctor/${x}`, {
            method: "DELETE",
            headers: { "token": token }
        });
        const result = await response.json();
        if(response.status === 200){
            const newList = doctors.filter((d) => d.doctorid != x);
            setDoctors(newList);
        } else {
            alert(result.msg);
        }
    }
}
```

25. Add a simple logout button that clears the local storage. In list.jsx
    a. import { useNavigate } from "react-router-dom";
    b. const navigate = useNavigate();
    c. then add this function

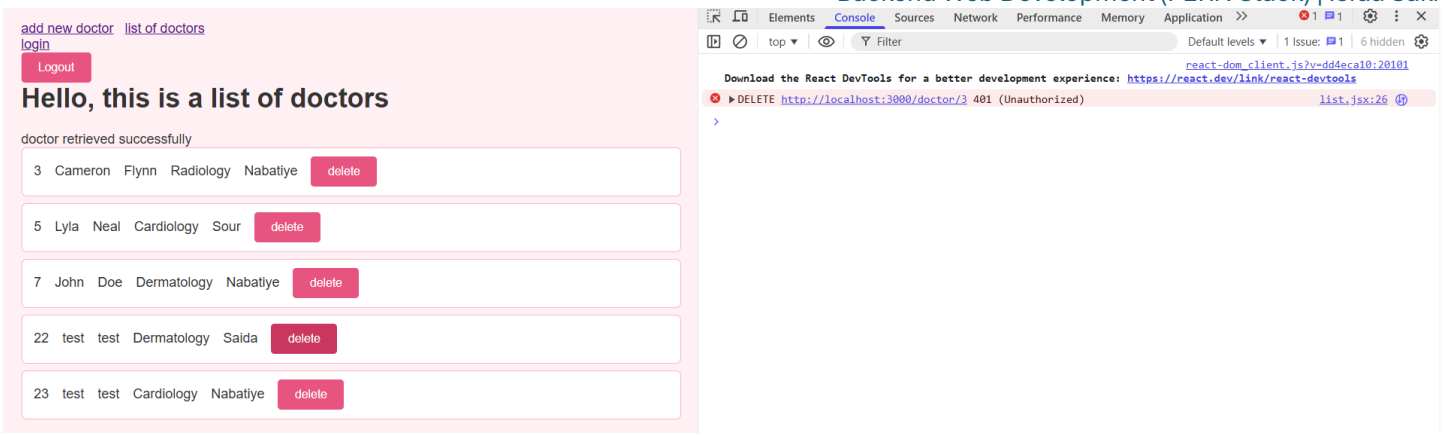
```
function handleLogout(){
    localStorage.removeItem("token");
    navigate("/login");
}
```

    d. then add this logout button anywhere in the return statement: <button *onClick*={handleLogout}>Logout</button>

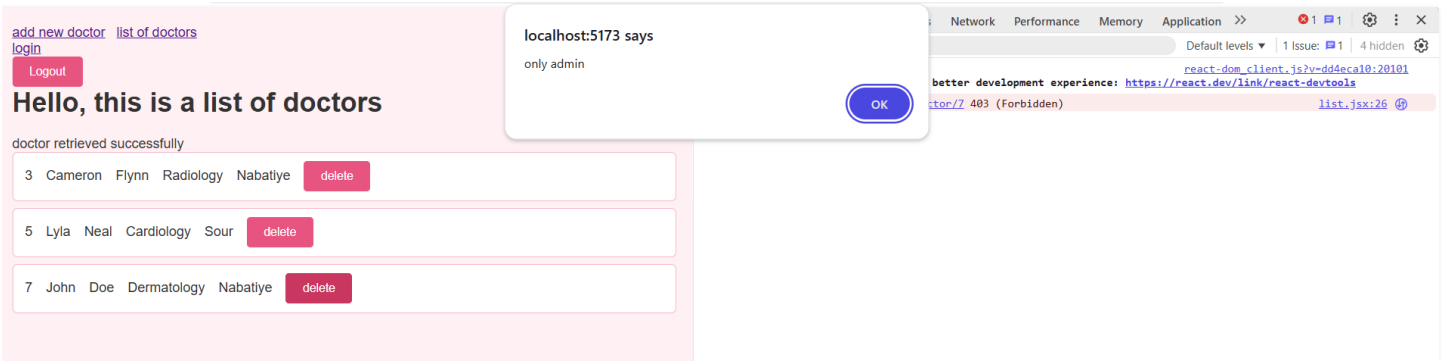
26. Result of local storage after pressing logout



27. Result if you try to delete without a token



Attempting to delete as role 2



Done :3