Nathan Dinh
04/14/2024

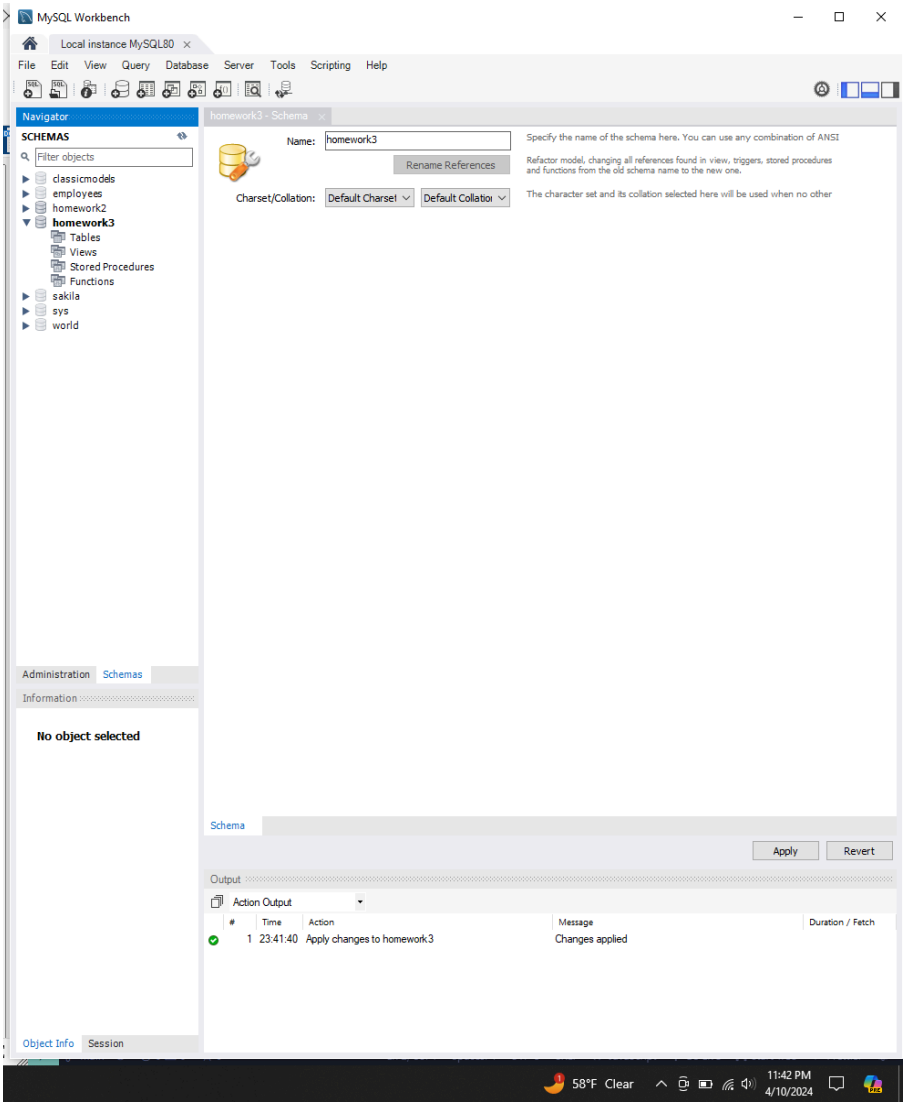
How To Deploy MySQL on AWS RDS and Express API on Render

<u>The code is available on GitHub:</u> https://github.com/nathan-dinh-dev/mysql-api
Please review the git commits to follow the coding steps

**** Skip Part 1 & 2 if you already had database schema & express API code already**
**** Part 5 is for the Deployment Strategy**

<u>Part 1:</u> Database Schema Creation

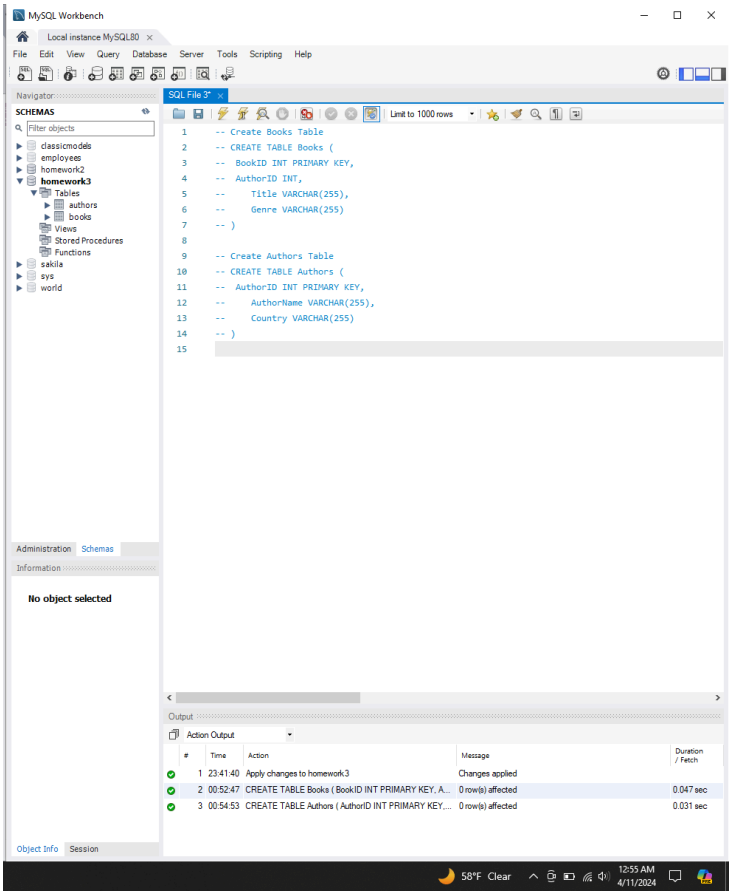
Step 1: Create a “homework3” schema



Schema Creation

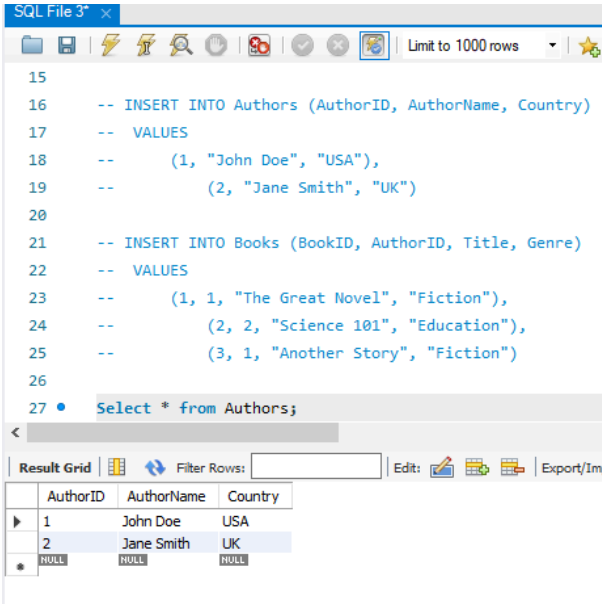
Step 2: Create a small library database application (Books and Authors Tables)

Books table has 4 columns which are BookID (primary key), AuthorID (foreign key), Title, and Genre
Authors table has 3 columns which are AuthorID (primary key), AuthorName, and Country

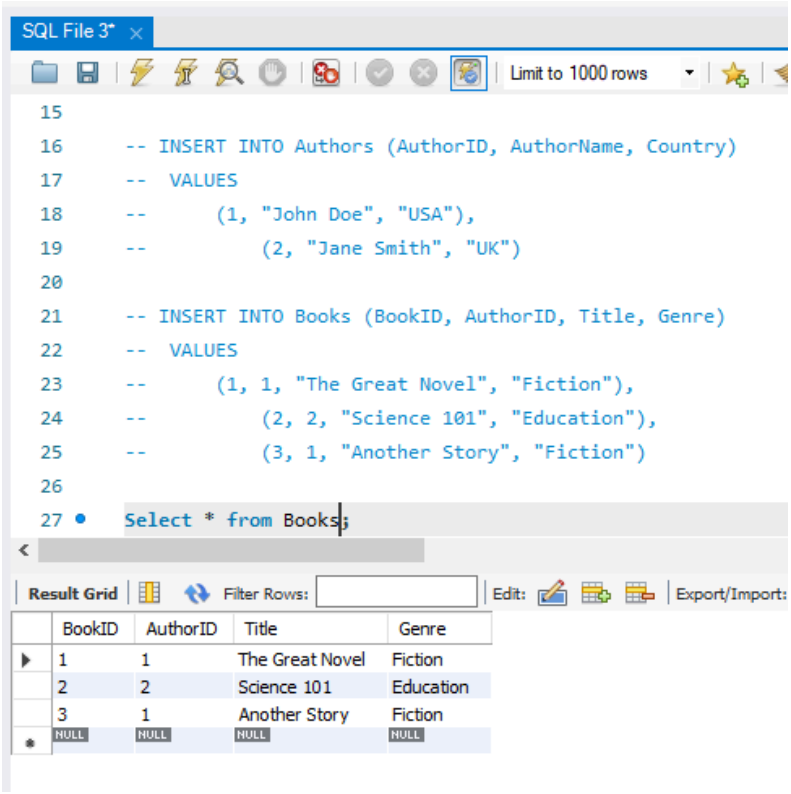


Create Books and Authors' tables

Step 3: Insert Sample Data into Authors & Books tables



Authors Table



Books Table

Part 2: Database Connection

Step 1: Create a JavaScript file responsible for database information to make connections, such as the host, user, password,...

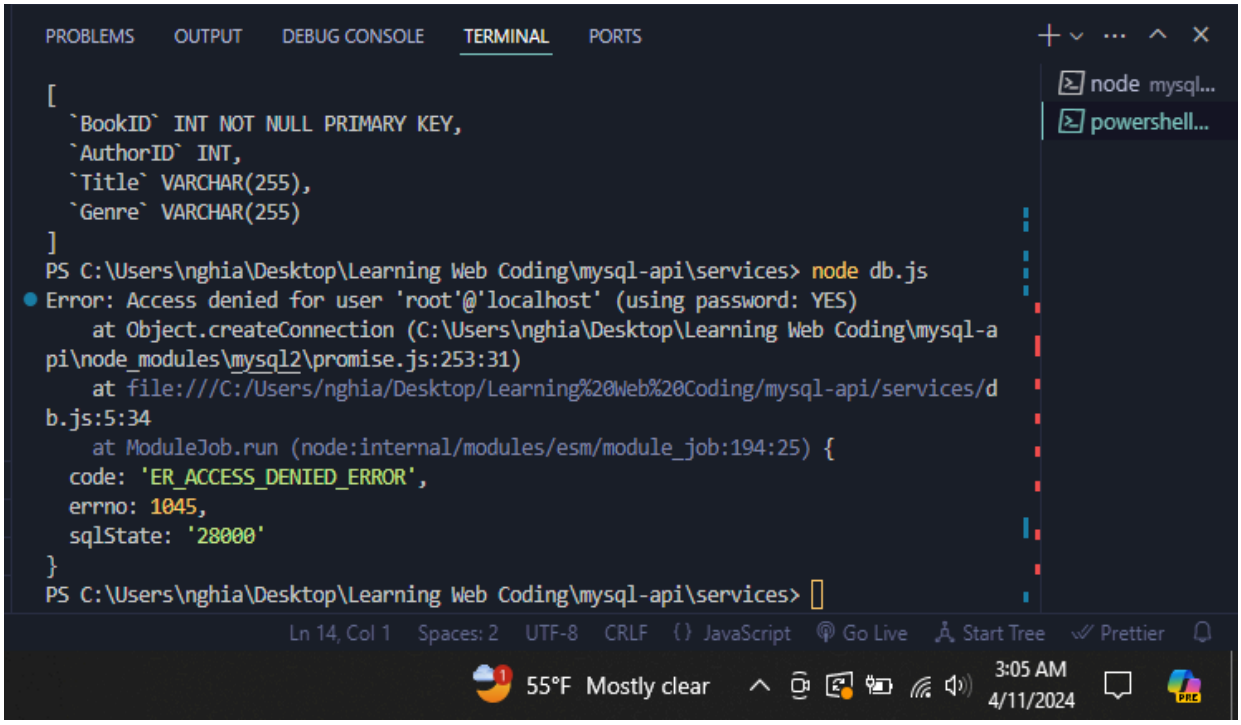
```
import "dotenv/config";
```

```
const config = {
  db: {
    host: process.env.MYSQL_HOST,
    user: process.env.MYSQL_USER,
    password: process.env.MYSQL_PASSWORD,
    database: process.env.MYSQL_DB,
    port: process.env.MYSQL_PORT,
    connectTimeout: 60000,
  },
  listPerPage: 10,
};

export default config;
```

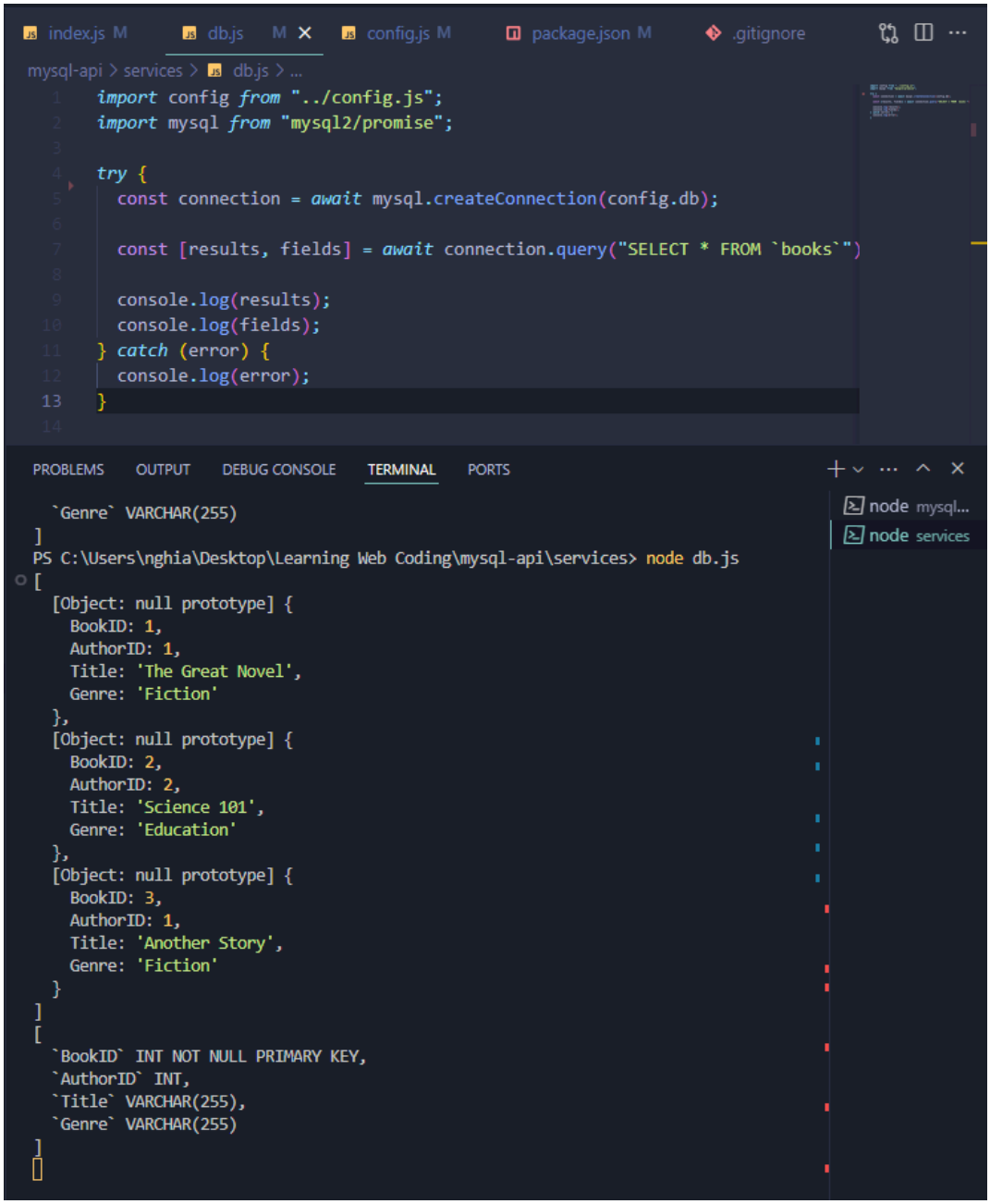
The .env file will store sensitive information for the database connection. To ensure a secure connection, this file will be utilized to conceal all sensitive information.

Step 2: Make a connection to the database with an exception handler and print out the retrieved data to the console to confirm a successful database connection.

```
import config from "../config.js";
import mysql from "mysql2/promise";

try {
  console.log(config.db);

  const connection = await mysql.createConnection(config.db);

  const [results, fields] = await connection.query("SELECT * FROM
`books`");

  console.log(results);
  console.log(fields);
} catch (error) {
  console.log(error);
}
```



Testing Error Handler When Wrong Password Input



The Sucess Result From the Terminal

Part 3: Building APIs

During this part, I used JavaScript with Express as a node.js framework to develop APIs that perform CRUD operations on the MySQL database.

Step 1: Install npm packages such as axios, cors, dotenv, express, mysql2, and nodemon for the application
Run this in vscode terminal: "*npm install axios cors dotenv express mysql2 nodemon*"

Axios: A JavaScript library used to make HTTP requests from node.js. It simplifies the process of sending asynchronous HTTP requests to REST endpoints and performing CRUD operations.

CORS: CORS stands for Cross-Origin Resource Sharing. It's a security feature implemented by web browsers to control requests made to a different domain than the one that served the web page. When developing web applications, CORS issues often arise when making requests from one domain to another. It is used in this app to make requests to localhost:5000.

dotenv: dotenv is a popular npm package that loads environment variables from a .env file into process.env. It is used in this app to hide all sensitive information that is needed for the connection such as password,...

Express: Express.js is a minimal and flexible Node.js web application framework that provides a robust set of features for building web and mobile applications. It is used in this app to provide a set of middleware functions and routes to handle HTTP requests and responses.

mysql2: mysql2 is a fast MySQL driver for Node.js that provides support for promises, async/await, and other modern JavaScript features. It allows developers to interact with MySQL databases using non-blocking, asynchronous methods, making it efficient for handling database operations in Node.js applications.

Nodemon: Nodemon is a utility tool that monitors changes in your Node.js application and automatically restarts the server whenever changes are detected. It's commonly used during the development phase to streamline the development process by eliminating the need to manually restart the server after making changes to the code.

Step 2: Create Express API which is listening on port 5000

```
import express from "express";
import cors from "cors";
import "dotenv/config";
import router from "./apiRouter.js";

const app = express();

const PORT = process.env.PORT || 5000;

const corsOptions = {
  origin: "http://localhost:5000",
  optionsSuccessStatus: 200,
};

// Middleware
```

```
app.use(cors(corsOptions));

// parse requests of content-type - application/json
app.use(express.json());

// parse requests of content-type - application/x-www-form-urlencoded
app.use(
  express.urlencoded({
    extended: true,
  })
);

// API routes
app.use("/", router);

app.get("/testAPI", (req, res) => {
  res.json({ message: "ok" });
});

app.listen(PORT, () => {
  console.log("Listening on Port", PORT);
});
```

Step 3: Create class DbService which holds CRUD methods

1. **Async function for when receiving GET request**

```
// GET method
  async getAllData(tableName) {
    try {
      if (tableName === "books") {
        const [results, fields] = await connection.query(
          "SELECT * FROM `books`"
        );

        return results;
      } else if (tableName === "authors") {
        const [results, fields] = await connection.query(
          "SELECT * FROM `authors`"
        );
```

```
        return results;
      }
    } catch (err) {
      console.log(err.message);
    }
  }
```

2. **Async functions for POST requests**

```
// POST method
  async insertNewBook(bookID, authorID, title, genre) {
    try {
      const query =
        "INSERT INTO `books` (`BookID`, `AuthorID`, `Title`,
`Genre`) VALUES (?, ?, ?, ?)";
      const [results, fields] = await connection.query(query, [
        bookID,
        authorID,
        title,
        genre,
      ]);
      return results;
    } catch (error) {
      console.log(error);
    }
  }
```

```
async insertNewAuthor(authorID, authorName, country) {
    try {
      const query =
        "INSERT INTO `authors` (`AuthorID`, `AuthorName`, `Country`)
VALUES (?, ?, ?)";
      const [results, fields] = await connection.query(query, [
        authorID,
        authorName,
        country,
      ]);
      return results;
    } catch (error) {
      console.log(error);
    }
```

```
  }
```

3. **Async functions for PUT method**

```
  // PUT method
  async updateAuthor({ authorID, authorName, country }) {
    try {
      let query = "";
      if (authorName && country)
        query = `UPDATE authors SET AuthorName="${authorName}",
Country="${country}" WHERE authorID=${authorID} LIMIT 1`;
      else if (authorName)
        query = `UPDATE authors SET AuthorName="${authorName}" WHERE
authorID=${authorID} LIMIT 1`;
      else if (country)
        query = `UPDATE authors SET Country="${country}" WHERE
authorID=${authorID} LIMIT 1`;

      const [results, fields] = await connection.query(query);
      return results;
    } catch (error) {
      console.log(error);
    }
  }

  async updateBook({ bookID, authorID, title, genre }) {
    try {
      let query = "";
      if (bookID && authorID && title && genre)
        query = `UPDATE books SET AuthorID=${authorID}, Title="${title}",
Genre="${genre}" WHERE bookID=${bookID} LIMIT 1`;
      else if (authorID && title)
        query = `UPDATE books SET AuthorID=${authorID}, Title="${title}"
WHERE bookID=${bookID} LIMIT 1`;
      else if (authorID && genre)
        query = `UPDATE authors SET AuthorID=${authorID}, Genre="${genre}"
WHERE bookID=${bookID} LIMIT 1`;
      else if (genre && title)
        query = `UPDATE books SET Genre="${genre}", Title="${title}" WHERE
bookID=${bookID} LIMIT 1`;
```

```
      else if (authorID)
        query = `UPDATE books SET AuthorID=${authorID} WHERE
bookID=${bookID} LIMIT 1`;
      else if (genre)
        query = `UPDATE books SET Genre="${genre}" WHERE bookID=${bookID}
LIMIT 1`;
      else if (title)
        query = `UPDATE books SET  Title="${title}" WHERE bookID=${bookID}
LIMIT 1`;

      const [results, fields] = await connection.query(query);
      return results;
    } catch (error) {
      console.log(error);
    }
  }
```

4. **Async functions for DELETE requests**

```
// Delete Method
  async deleteBook({ bookID, authorID, title, genre }) {
    try {
      let query = "";
      if (bookID) query = `DELETE FROM books WHERE BookID=${bookID}
LIMIT 1`;
      else if (authorID)
        query = `DELETE FROM books WHERE AuthorID=${authorID} LIMIT
1`;
      else if (genre)
        query = `DELETE FROM books WHERE Genre="${genre}" LIMIT 1`;
      else if (title)
        query = `DELETE FROM books WHERE Title="${title}" LIMIT 1`;

      const [results, fields] = await connection.query(query);
      return results;
    } catch (error) {
      console.log(error);
    }
  }
```

```
  async deleteAuthor({ authorID, authorName, country }) {
    try {
      let query = "";
      if (authorID)
        query = `DELETE FROM authors WHERE AuthorID=${authorID}
LIMIT 1`;
      else if (authorName)
        query = `DELETE FROM authors WHERE
AuthorName="${authorName}" LIMIT 1`;
      else if (country)
        query = `DELETE FROM authors WHERE Country="${country}"
LIMIT 1`;

      const [results, fields] = await connection.query(query);
      return results;
    } catch (error) {
      console.log(error);
    }
  }
}
```

Step 4: Create apiRouter.js which handles POST, GET, PUT, and DELETE requests

1. Create a new row in the database using **POST** requests

```
// Create a new row in the database
```

Add a new book

```
router.post("/insert/book", (req, res) => {
  const { bookID, authorID, title, genre } = req.body;
  const db = new DbService();
  const result = db.insertNewBook(bookID, authorID, title, genre);

  result
    .then((data) => res.json({ message: "ok", data: data }))
    .catch((err) => console.log(err));
});
```

Add a new author:

```
router.post("/insert/author", (req, res) => {
  const { authorID, authorName, country } = req.body;
```

```
  const db = new DbService();
  const result = db.insertNewAuthor(authorID, authorName, country);

  result
    .then((data) => res.json({ message: "ok", data: data }))
    .catch((err) => console.log(err));
});
```

2. Retrieve data from database using **GET** requests

Get all books in the books table:

```
// Retrieve data from database
router.get("/get", (req, res) => {});

router.get("/get/books/all", (req, res) => {
  const db = new DbService();

  const result = db.getAllData("books");

  result
    .then((data) => res.json({ data: data }))
    .catch((err) => console.log(err));
});
```

Get all authors in authors table:

```
router.get("/get/authors/all", (req, res) => {
  const db = new DbService();

  const result = db.getAllData("authors");

  result
    .then((data) => res.json({ data: data }))
    .catch((err) => console.log(err));
});
```

3. Update tables in the database using **PUT** requests

Update Author's table

```
router.put("/update/author", (req, res) => {
  const db = new DbService();

  const result = db.updateAuthor(req.body);

  result
    .then((data) => res.json({ data: data }))
    .catch((err) => console.log(err.message));
});
```

Update Books table

```
router.put("/update/book", (req, res) => {
  const db = new DbService();

  const result = db.updateBook(req.body);

  result
    .then((data) => res.json({ data: data }))
    .catch((err) => console.log(err.message));
});
```

4. Delete row in the database using DELETE requests

Delete a book:

```
// Delete database
router.delete("/delete/book", (req, res) => {
  const db = new DbService();

  const result = db.deleteBook(req.body);

  result
    .then((data) => res.json({ data: data }))
    .catch((err) => console.log(err.message));
});
```

Delete an author:

```
router.delete("/delete/author", (req, res) => {
  const db = new DbService();

  const result = db.deleteAuthor(req.body);
```

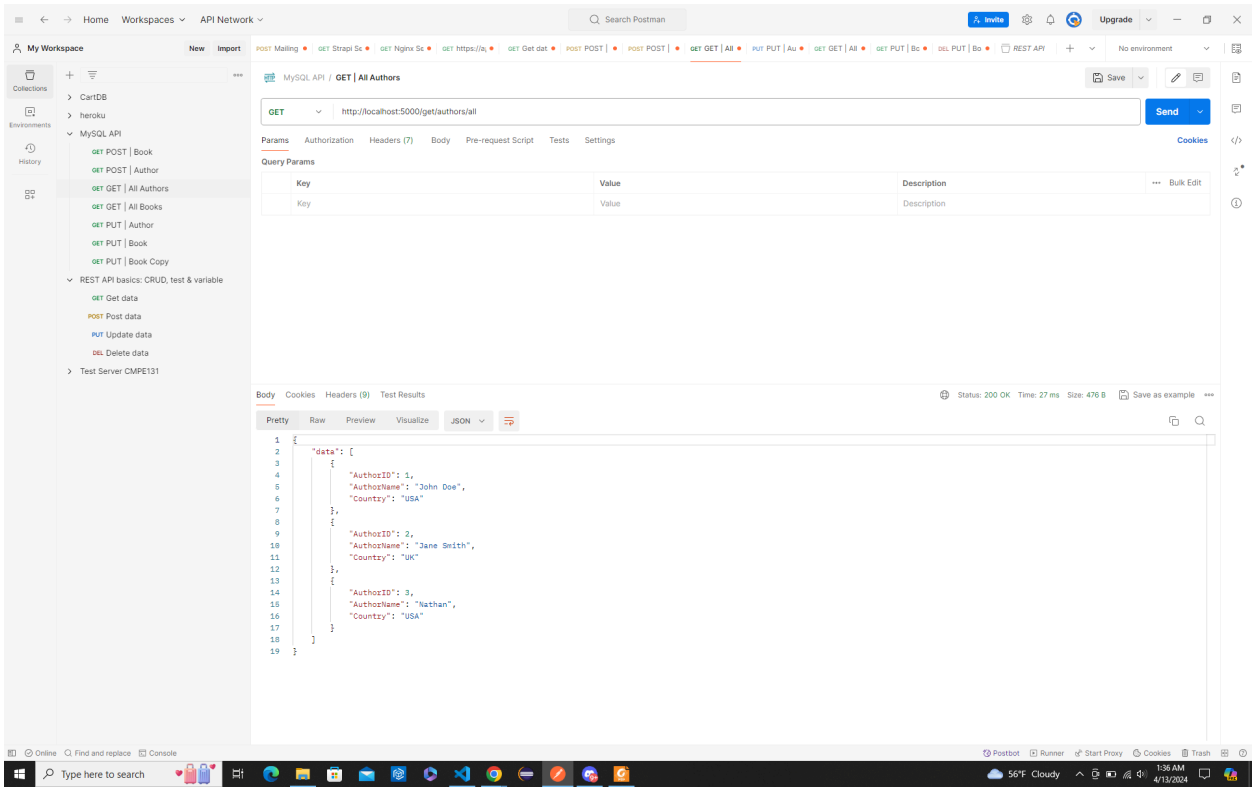
```
  result
    .then((data) => res.json({ data: data }))
    .catch((err) => console.log(err.message));
});
```

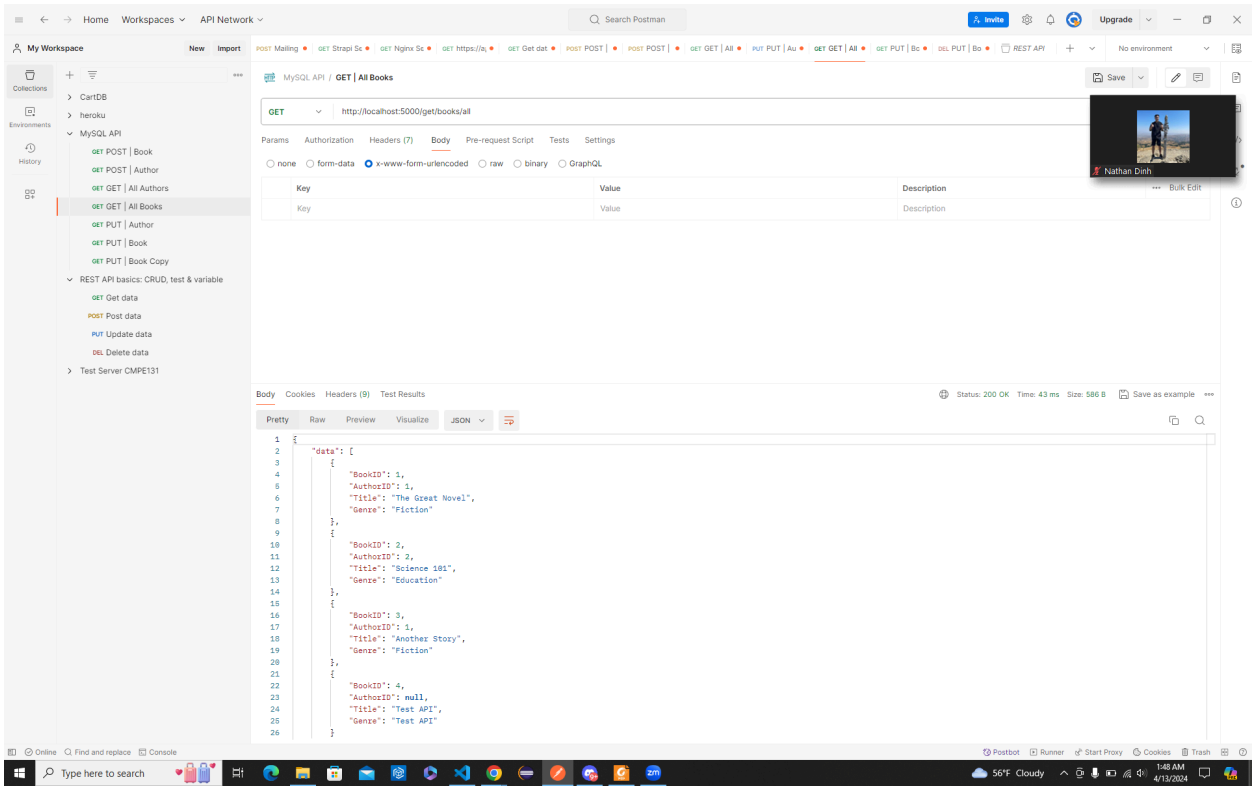
Part 4: API Testing

I used Postman for testing API in this assignment.

1. Testing **GET** requests

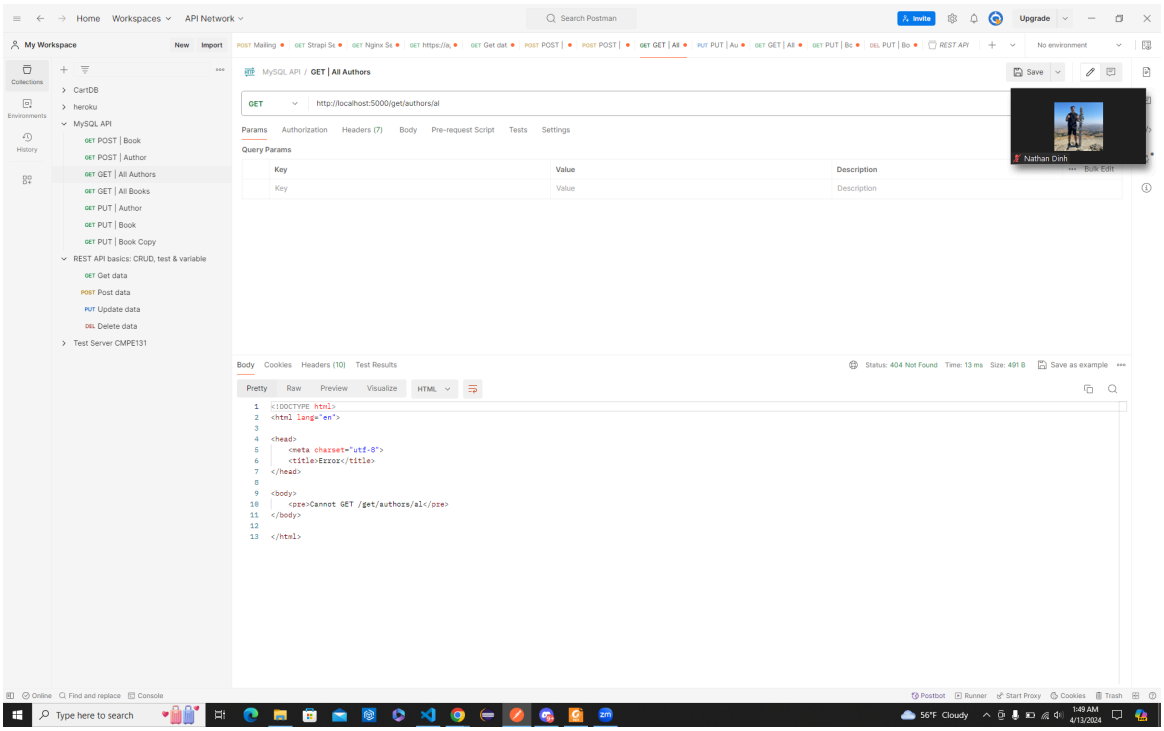
Test **Success GET** request to retrieve all books and all authors
Expected to receive all data in JSON file





Success GET requests

Test **Error GET** requests
Expected to receive ERROR message “Cannot GET request”

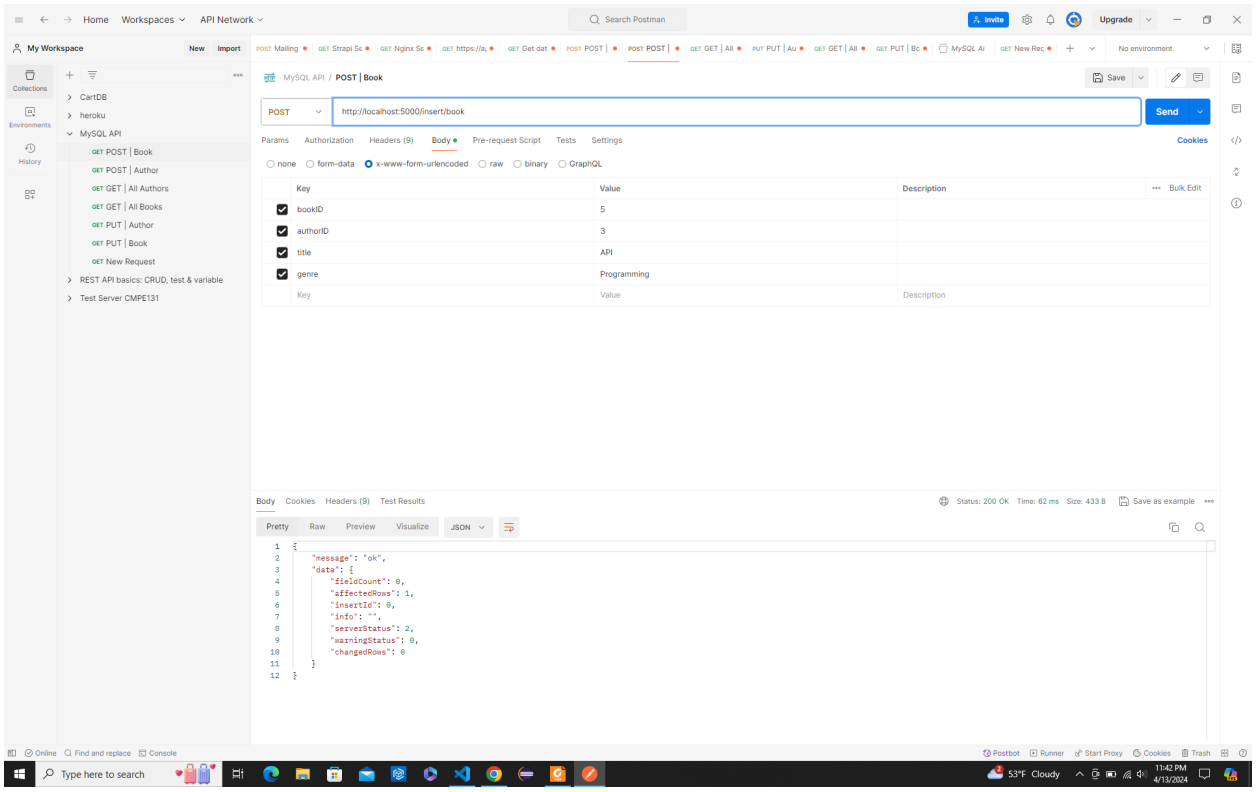


Error GET request

2. Testing **POST** requests

Test **Success POST** request
Expected to receive all data in JSON:


```
{
    "message": "ok",
    "data": {
        "fieldCount": 0,
        "affectedRows": 1,
        "insertId": 0,
        "info": "",
        "serverStatus": 2,
        "warningStatus": 0,
        "changedRows": 0
    }
}
```

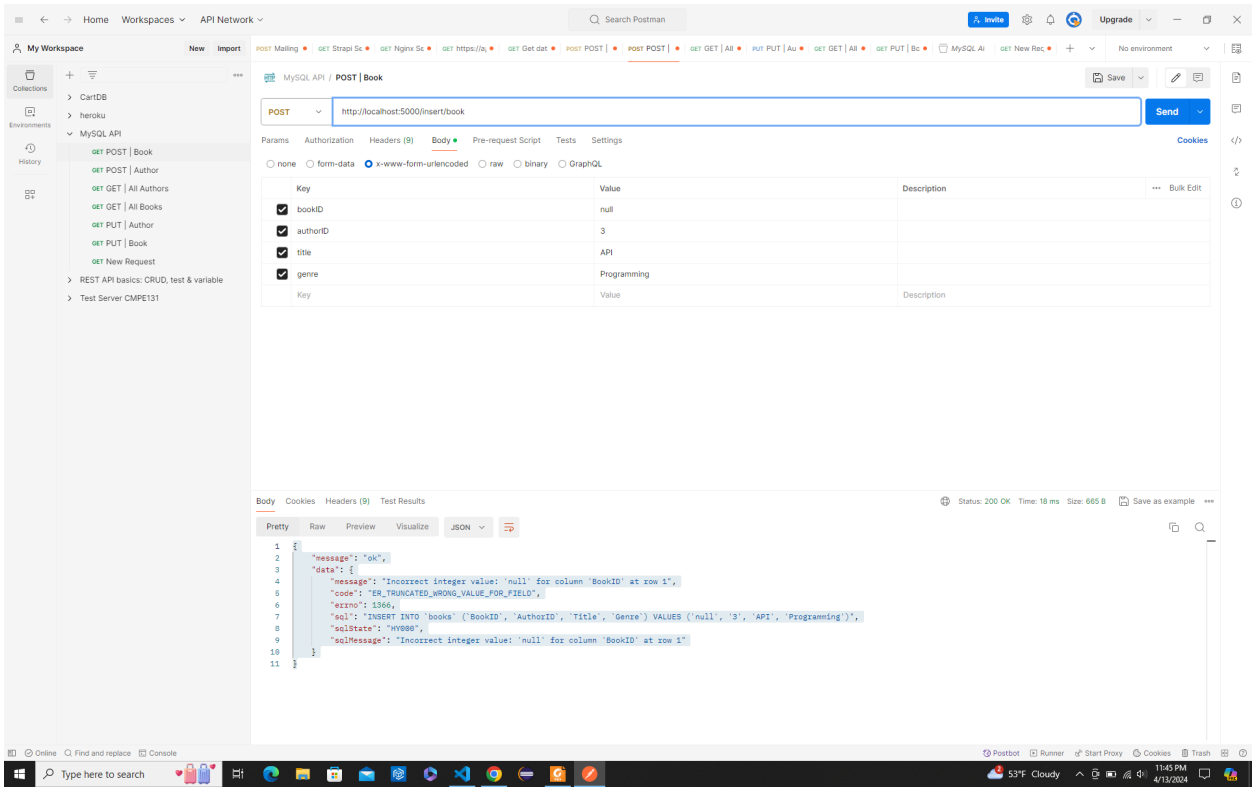



Success POST requests

Test **Error POST** request with **NULL primary key**
Expected to receive error JSON message


```
{
    "message": "ok",
    "data": {
```

```
        "message": "Incorrect integer value: 'null' for column 'BookID' at row 1",
        "code": "ER_TRUNCATED_WRONG_VALUE_FOR_FIELD",
        "errno": 1366,
        "sql": "INSERT INTO `books` (`BookID`, `AuthorID`, `Title`, `Genre`) VALUES ('null', '3', 'API', 'Programming')",
        "sqlState": "HY000",
        "sqlMessage": "Incorrect integer value: 'null' for column 'BookID' at row 1"
    }
}
```

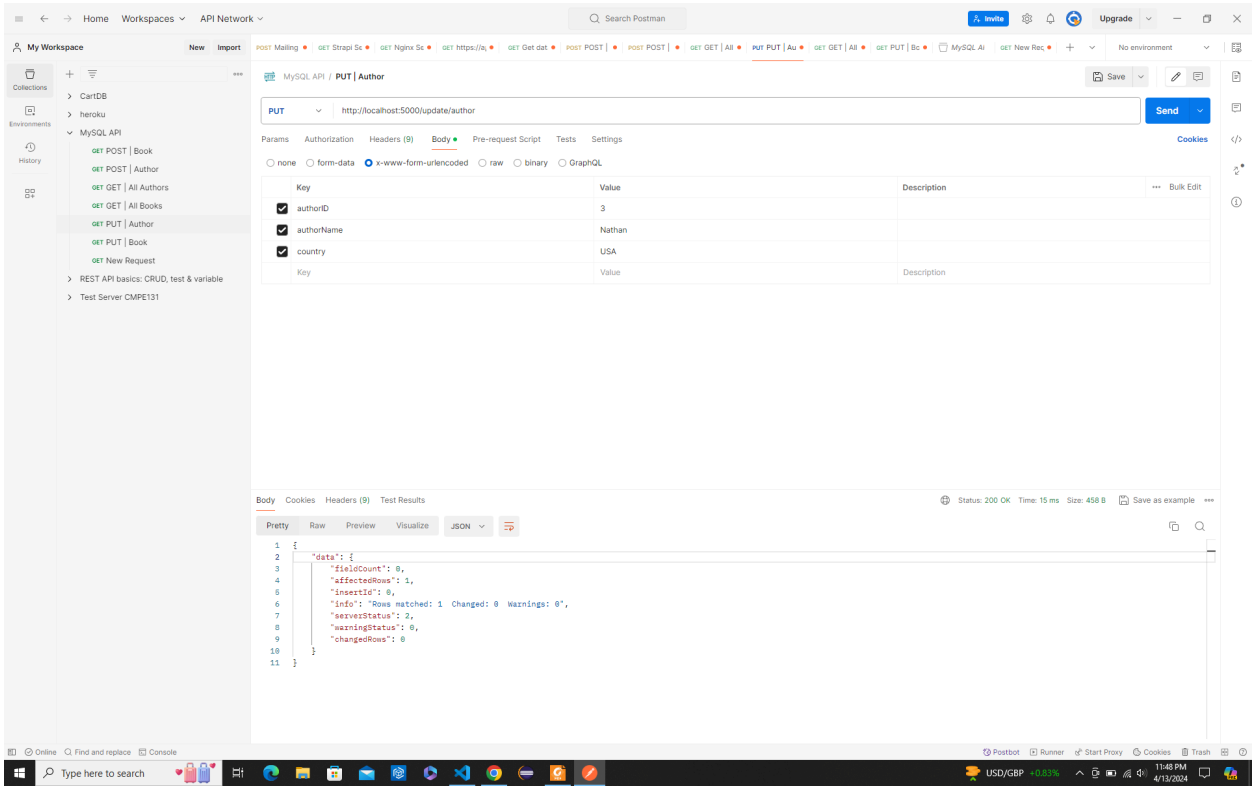
Error POST requests

3. Testing **PUT** requests

Test **Success POST** request
Expected to receive all data in JSON:


```
{
    "data": {
        "fieldCount": 0,
        "affectedRows": 1,
        "insertId": 0,
        "info": "Rows matched: 1  Changed: 1  Warnings: 0",
        "serverStatus": 2,
        "warningStatus": 0,
```

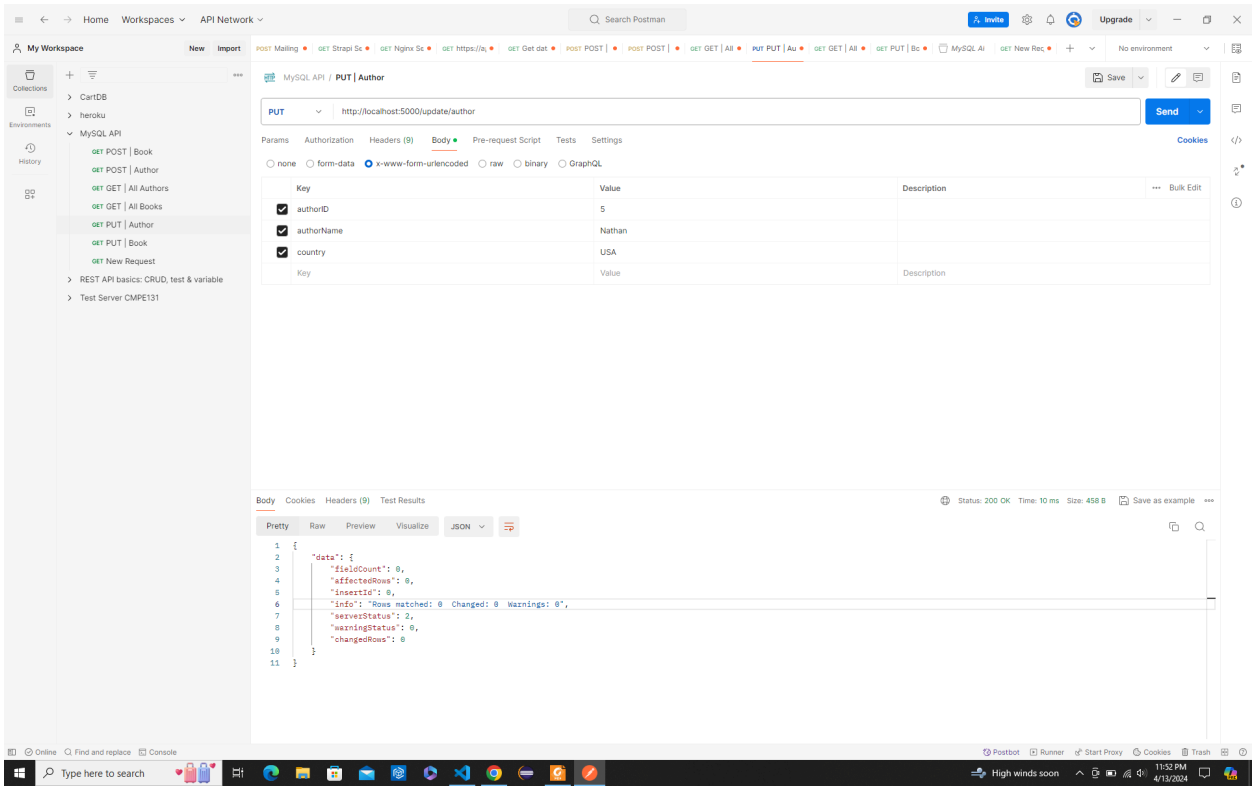
```
        "changedRows": 0
    }
}
```

Success **PUT** requests

Test **Error PUT** request with **not found primary key**
Expected to receive error JSON message


```
{
    "data": {
        "fieldCount": 0,
        "affectedRows": 0,
        "insertId": 0,
        "info": "Rows matched: 0  Changed: 0  Warnings: 0",
        "serverStatus": 2,
        "warningStatus": 0,
        "changedRows": 0
    }
}
```



Error **PUT** requests

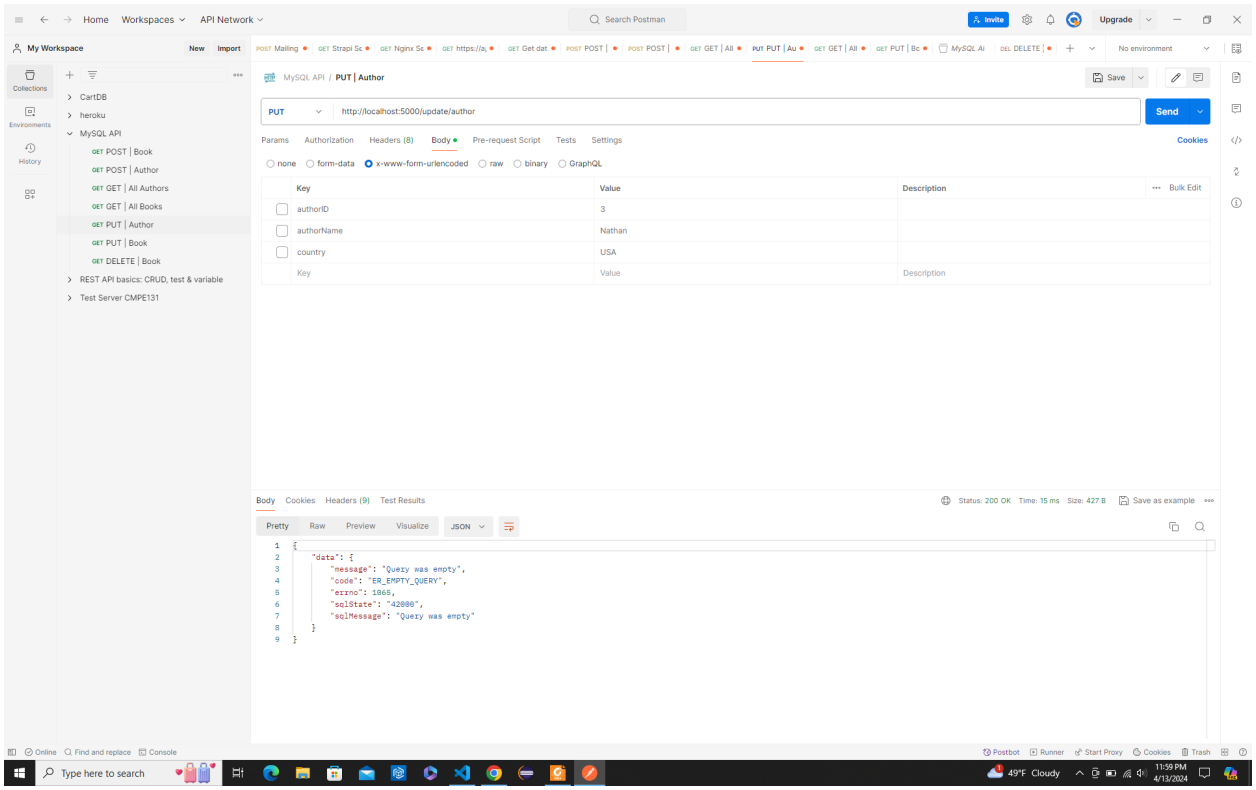
Explanation: There is no row found that match the record in the table to make changes

“"info": "Rows matched: 0 Changed: 0 Warnings: 0",”

Test **Error PUT** request with **empty query**

Expected to receive error JSON message

```
{
    "data": {
        "message": "Query was empty",
        "code": "ER_EMPTY_QUERY",
        "errno": 1065,
        "sqlState": "42000",
        "sqlMessage": "Query was empty"
    }
}
```

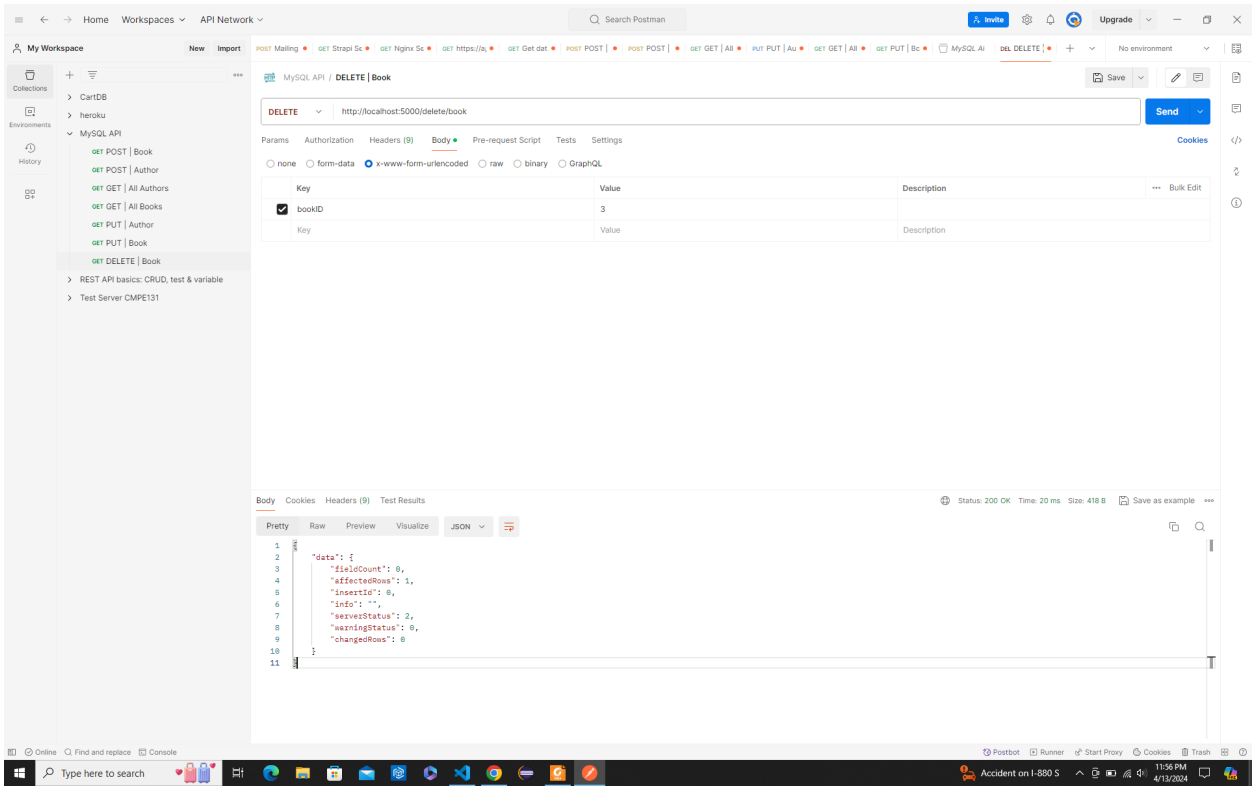


Error **PUT** requests

4. Testing **DELETE** requests

Test **Success DELETE** request with **BookID = 3**
Expected to receive all data in JSON:

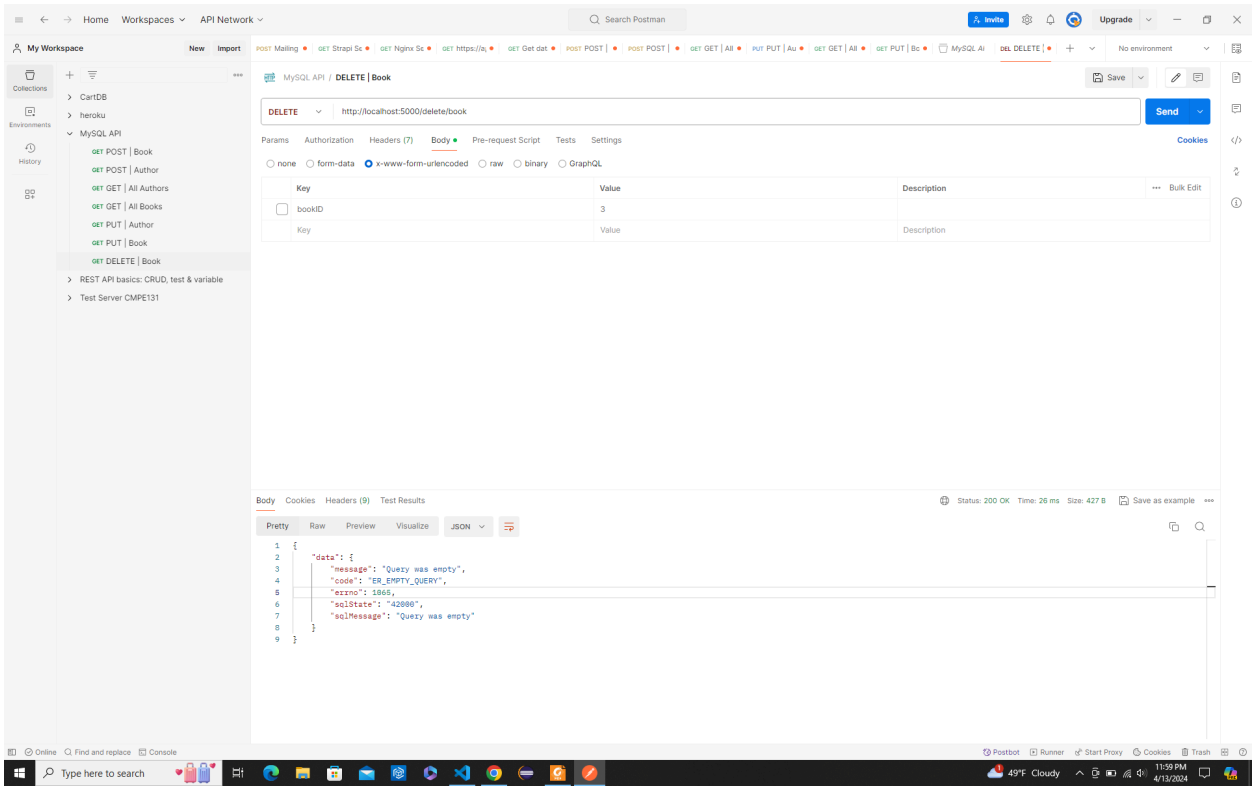

```
{
    "data": {
        "fieldCount": 0,
        "affectedRows": 1,
        "insertId": 0,
        "info": "",
        "serverStatus": 2,
        "warningStatus": 0,
        "changedRows": 0
    }
}
```



Success **DELETE** requests

Test **Error DELETE** request with **empty query**
Expected to receive error JSON message


```
{
    "data": {
        "message": "Query was empty",
        "code": "ER_EMPTY_QUERY",
        "errno": 1065,
        "sqlState": "42000",
        "sqlMessage": "Query was empty"
    }
}
```



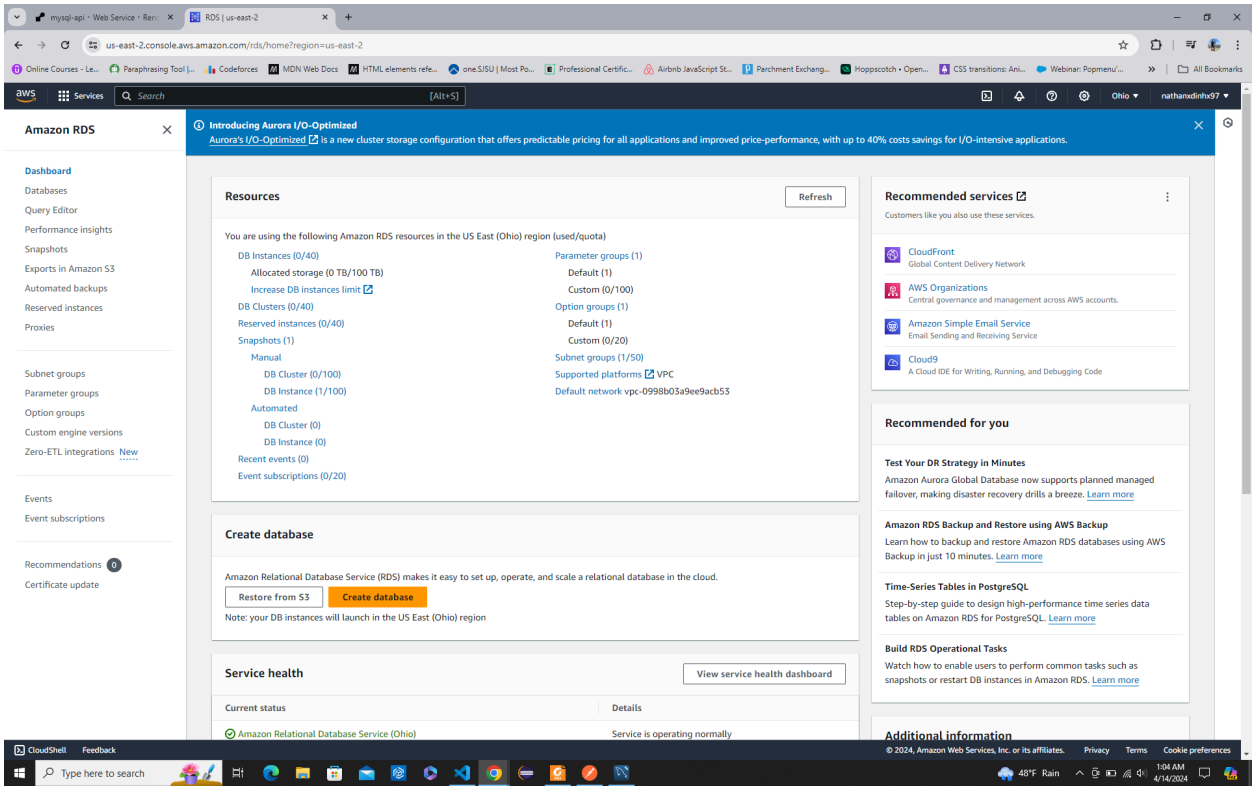
Error **PUT** requests

Part 5: Deployment Strategy

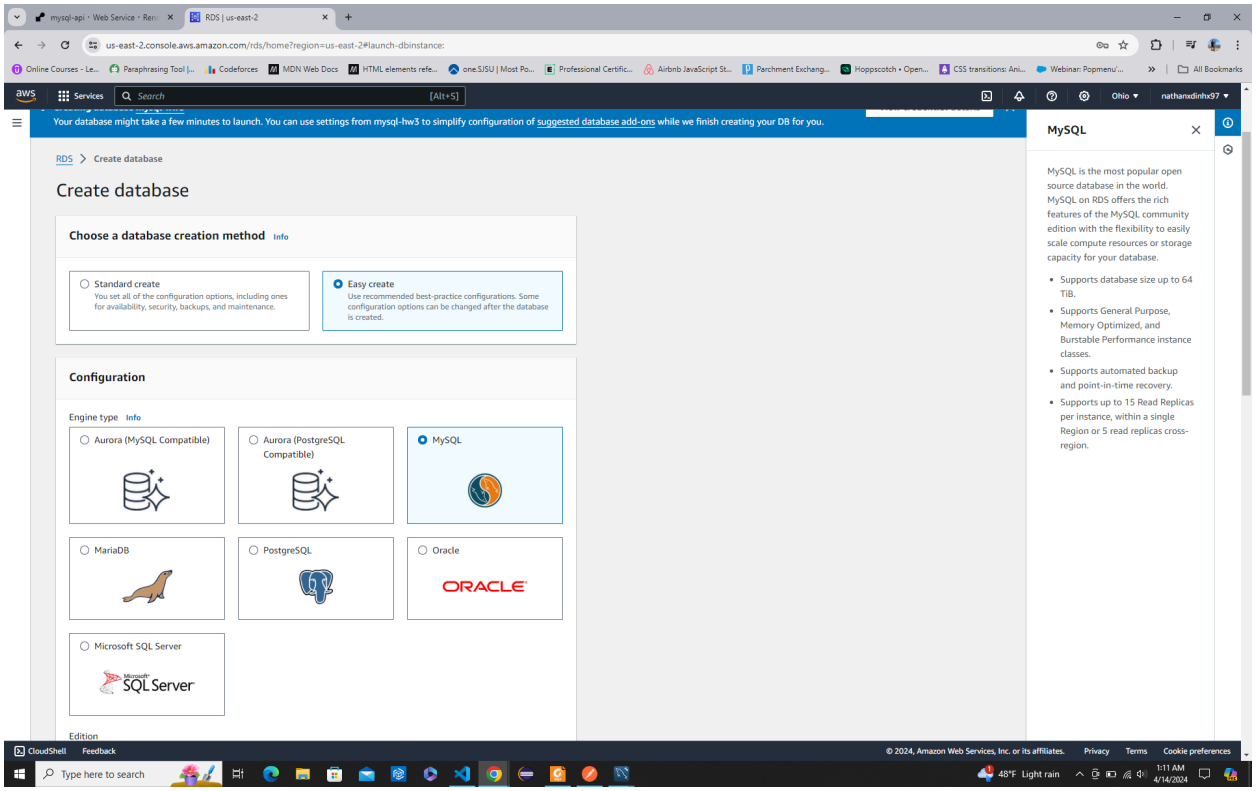
A. Deploy MySQL on AWS RDS DB Instance

Step 1: Create an AWS account with root user and login to aws console

Step 2: Go to RDS dashboard and click on the orange button “Create Database”

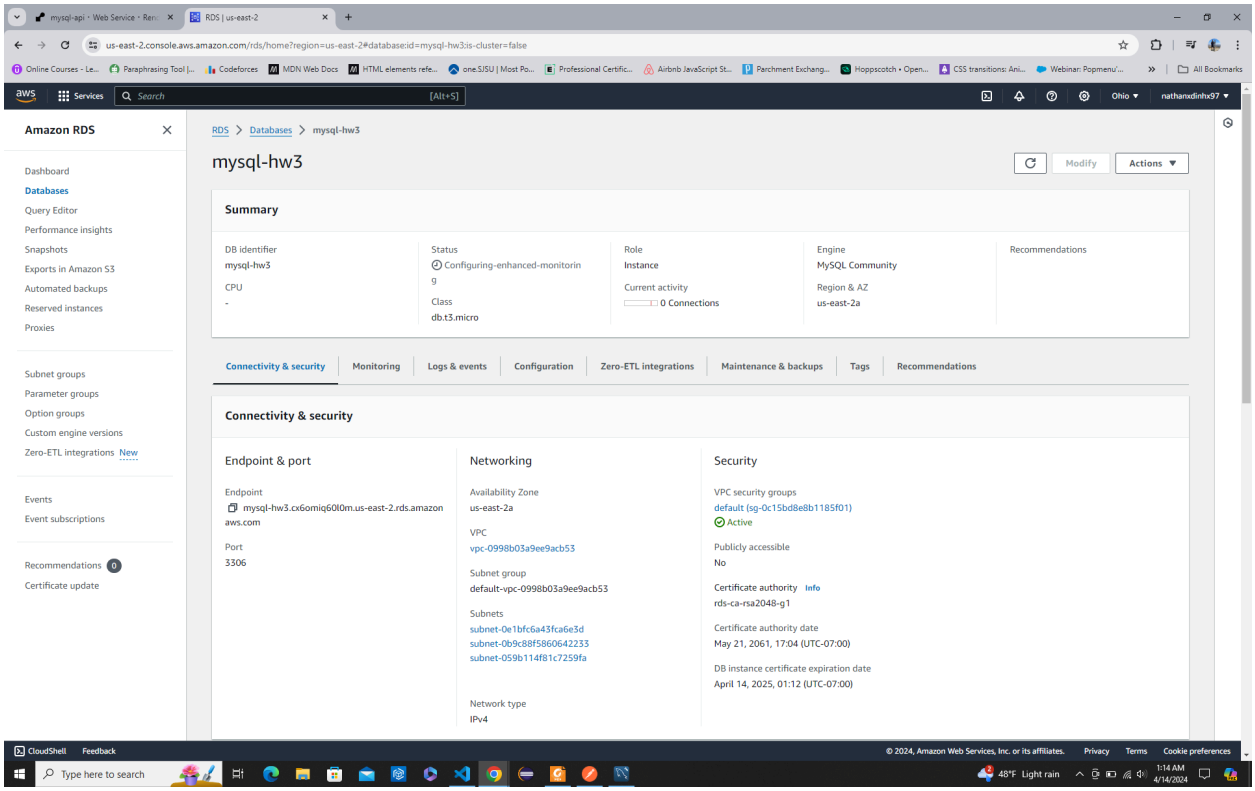


Step 4: For testing/learning/practicing, I chose Easy Create with MySQL engine



Step 5: Set the root name as admin and your password

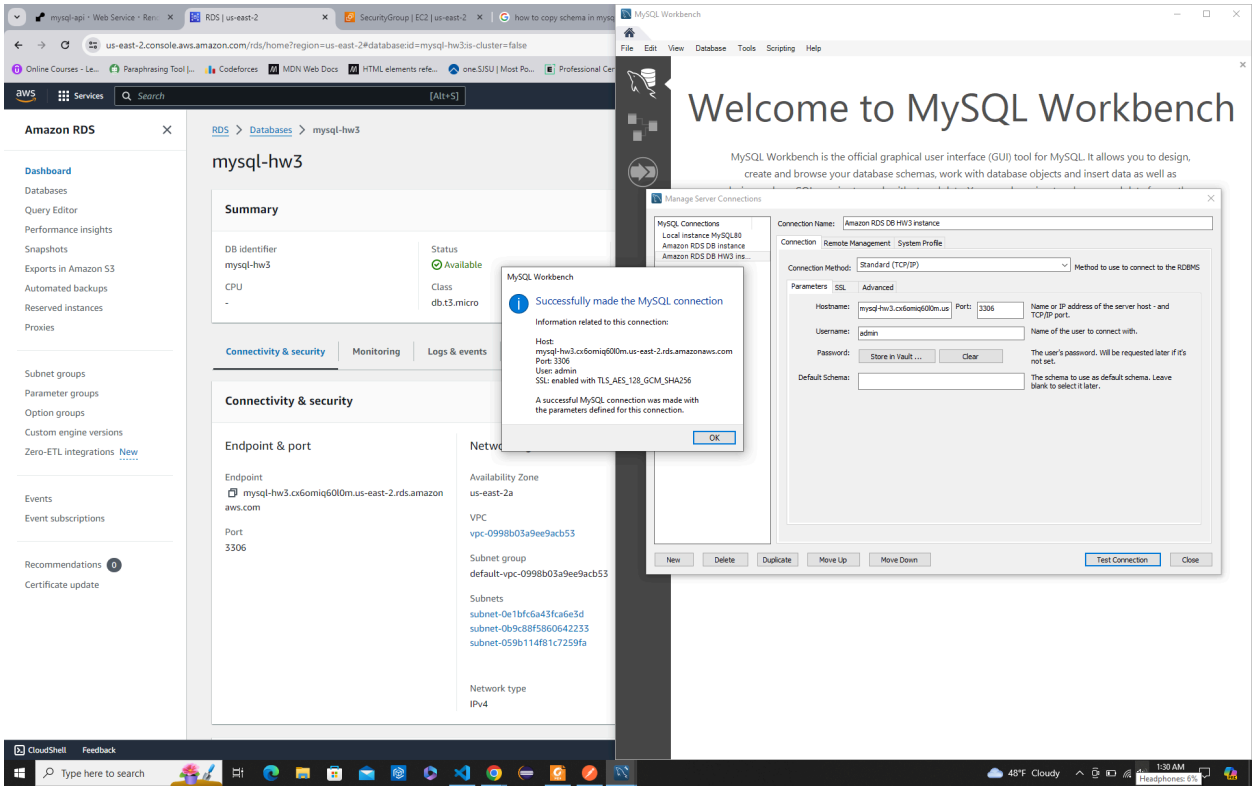
Step 7: After creating, save the port (default is 3306) and the endpoint for linking the MySQL later on



Step 8: Change the access rule to public and EC2 rule to access from any IP address for testing purposes:

a. Go to EC2 Dashboard
b. Go to the Security Groups tab
c. Select and only select the RDS database security group. You'll see the security group detail at the bottom
d. Click Inbound tab
e. Click Edit button
f. Add-Type:MYSQL/Aurora;Protocol:TCP;Range:3306;Source:0.0.0.0/0

Step 9: Test MySQL connection



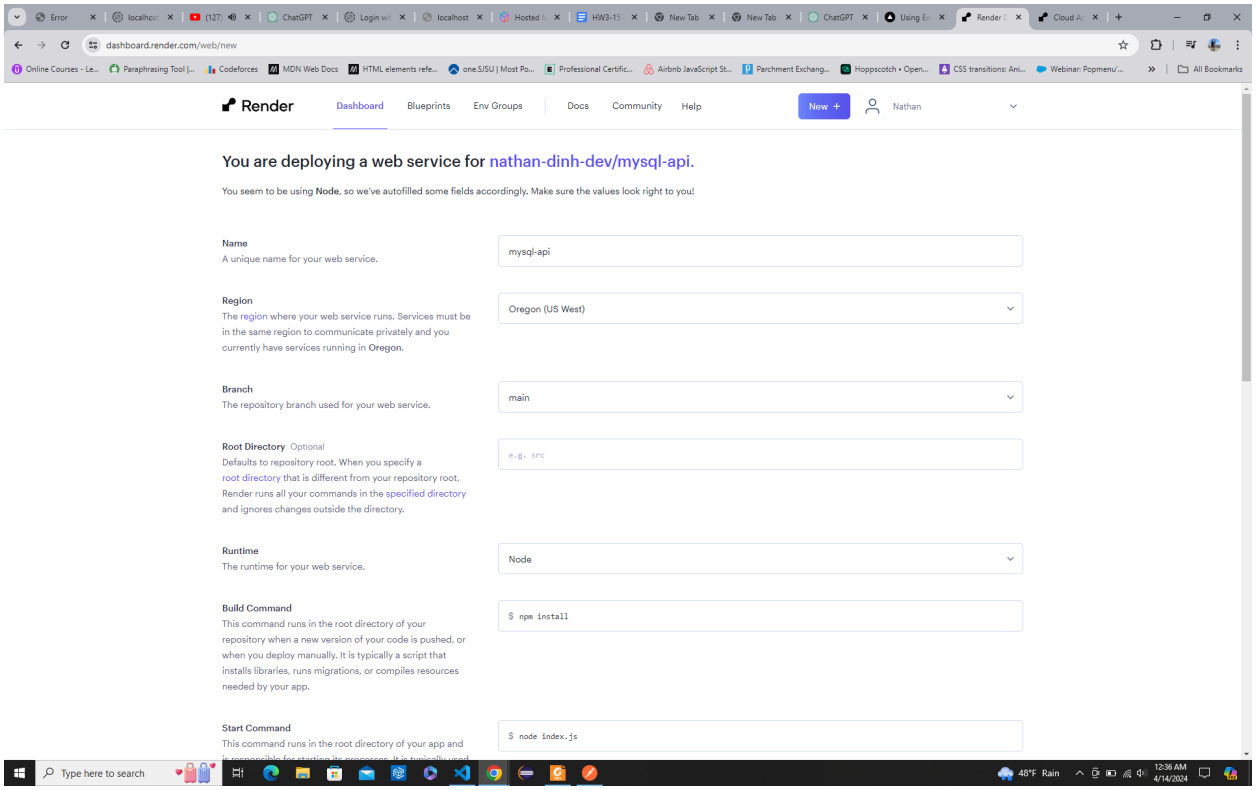
B. Deploy your NodeJs API on Render that links to your deployed MySQL

Step 1: Create an account on Render

Step 2: Create a new Render Web Service

Step 3: Authorize Render to access your GitHub repositories

Step 4: Setting the deployment configuration



Note: You need to override:

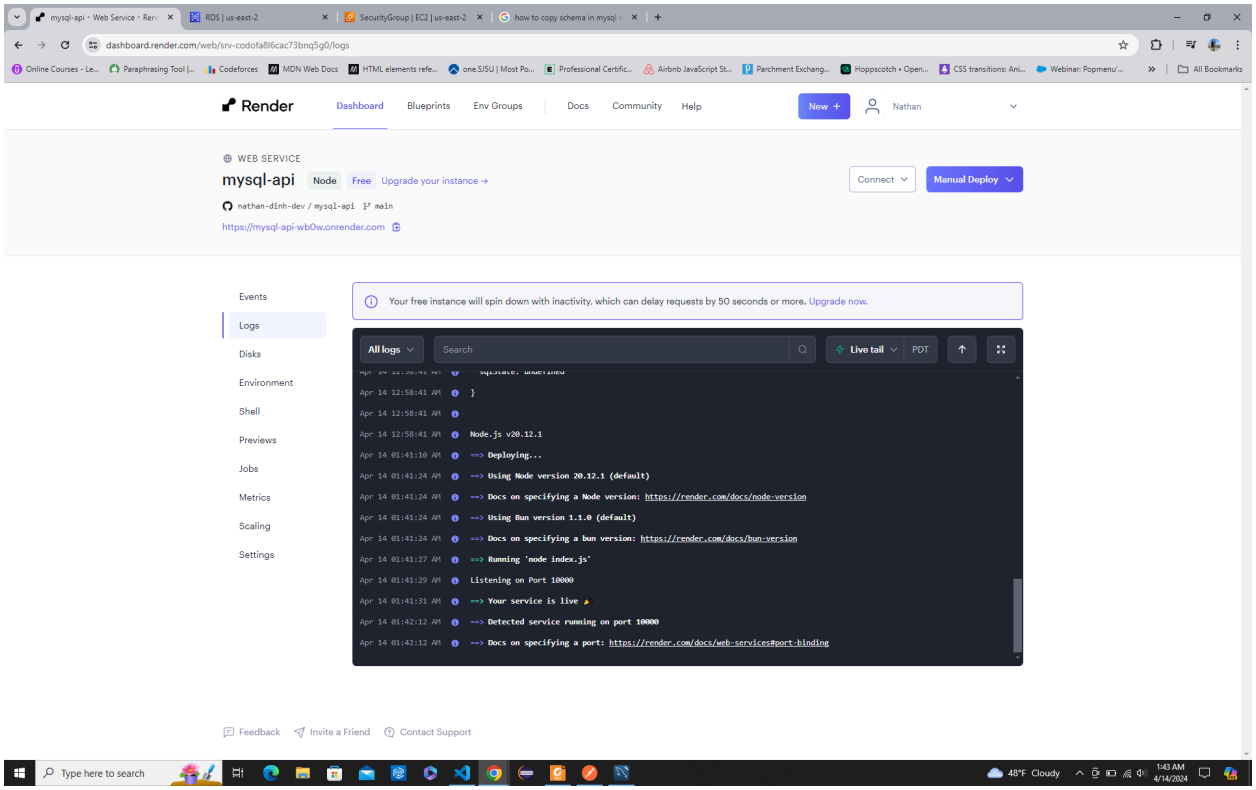
Build Command: npm install
Start Command: node index.js
Added Environment Keys which are from MySQL AWS that already deployed

Example in my application:
MYSQL_DB="homework3"
MYSQL_HOST="mysql-hw3.cx6omiq60l0m.us-east-2.rds.amazonaws.com" // This need to be same as the one you created on AWS RDS
MYSQL_PORT=3306
MYSQL_USER="admin"
MYSQL_PASSWORD="********"

Step 5: Setting deployment commands and choosing an instance type

Step 6: Overriding the default Node version

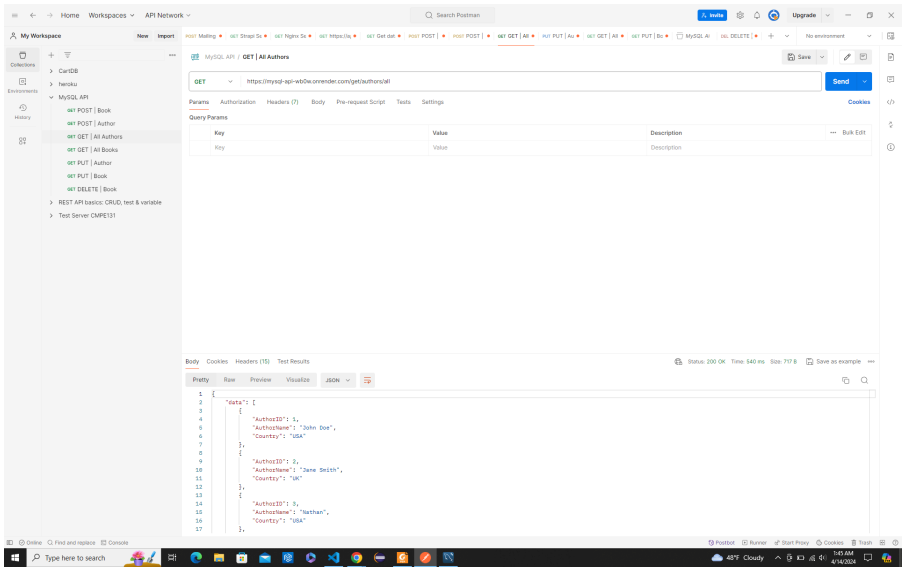
Step 7: Deployed successfully



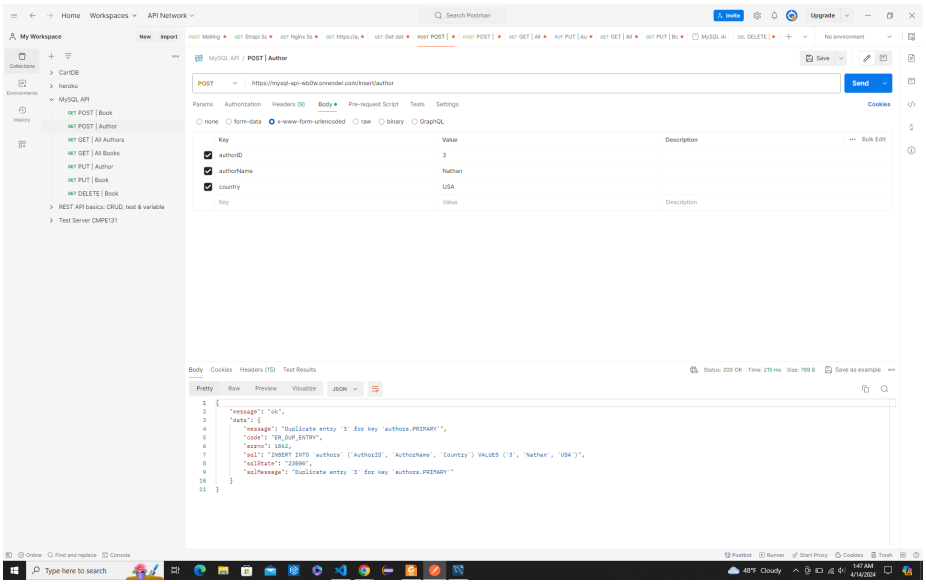
This is my API endpoint: https://mysql-api-wb0w.onrender.com

Final Step:

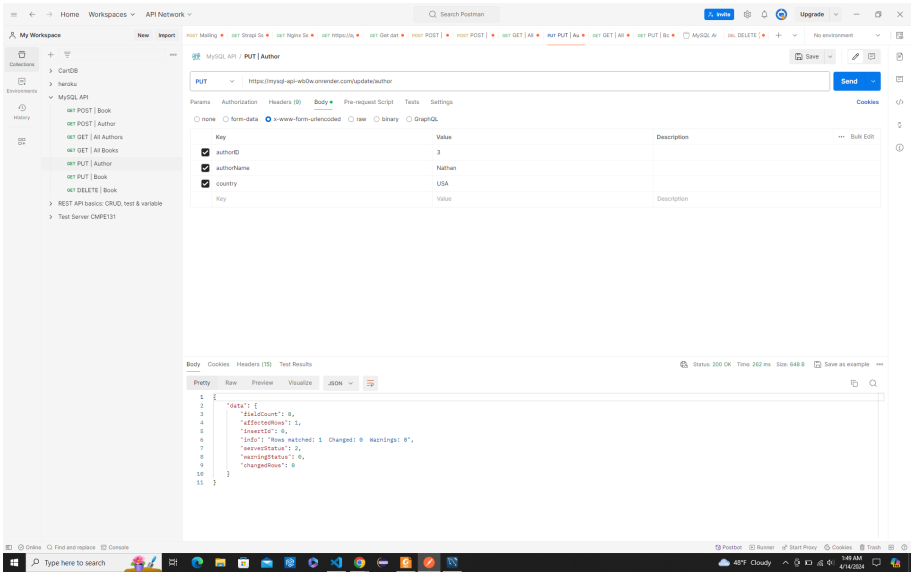
Test deployed API with **GET**, **POST**, **PUT**, and **DELETE** using Postman



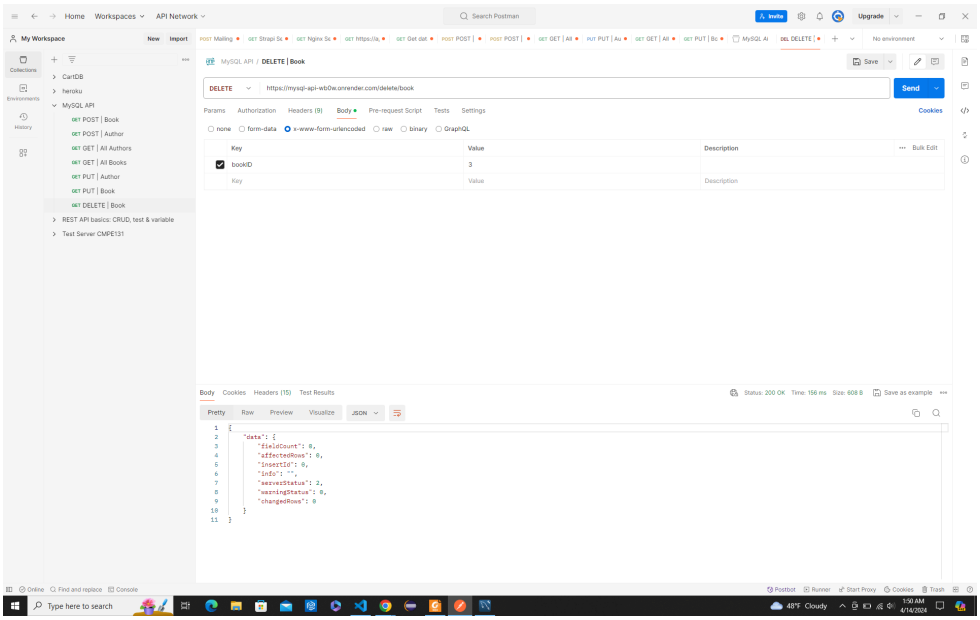
GET Request



POST request



PUT request



DELETE request

Conclusion: API - MySQL database deployed successfully.

References

MySQL2 npm package documentation: https://sidorares.github.io/node-mysql2/docs
AWS rds for deploying MySQL: https://aws.amazon.com/rds/mysql/