# RESTful API with Nodejs

Ramesh Pandey

# Project Overview

- User CRUD Operations with Node.js and SQLite
- Backend service for building a web service with CRUD operations for managing user data.
- Objective: To create a RESTful API using Node.js and SQLite for performing CRUD operations on user data.
- Importance: This project serves as a foundation for understanding how to implement basic CRUD functionalities in a web application, using widely-used technologies.
- Target Audience: developers, learners, or anyone interested in understanding backend development with Node.js.
- Key Features: Supports creating, reading, updating, and deleting user data through a RESTful API.

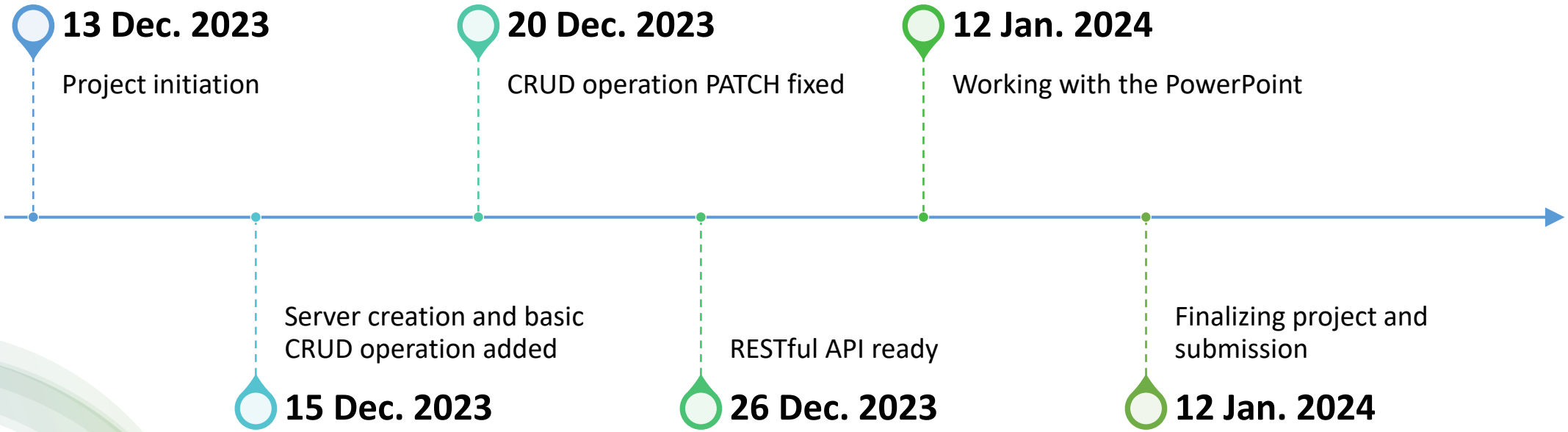
# Used technologies

- Node.js  : to create an HTTP server (express handles the HTTP part).
- Express.js : to simplify routing, middleware usage, and overall server setup.
  - ❖ Routing: Express.js is used to define and handle different routes for API (e.g., GET, POST, PATCH, PUT, DELETE).
  - ❖ Middleware: Express.js middleware functions to parse JSON in the request body (express.json()), handle errors globally, and more.
  - ❖ Serving Static Files: Express.js used to serve static files, although it's not explicitly used in the provided code.
- SQLite3
  - ❖ Database Operations: SQLite3 is used to perform various database operations, including creating tables, inserting data, querying data, updating records, and deleting records.
  - ❖ Database Connection: SQLite3 is used to connect to the SQLite database file (database.db) and perform operations on it.
- Postman: for CRUD operations handling.

# Challenges & Solutions

| Challenges | Solutions |
| --- | --- |
| 1. Configuring postman particularly for the PATCH function to work.<br>2. To check the ‘database.db’ created, in particular some tables or users.<br>3. Error handling. | 1. By fixing the headers, after good amount of debugging and info searching.<br>2. By copying the ‘database.db’ to the sqlite3 backed environment Cygwin folder, since I initially created the server using Ubuntu and it was not possible to download sqlite3 on Ubuntu for some reason.<br>3. Error handling was done at the last after the project was ready so need to go through all the code again. |

# Timetable


13 Dec. 2023
Project initiation
15 Dec. 2023
Server creation and basic CRUD operation added
20 Dec. 2023
CRUD operation PATCH fixed
26 Dec. 2023
RESTful API ready
12 Jan. 2024
Working with the PowerPoint
12 Jan. 2024
Finalizing project and submission

# Summary

- RESTful API project powered by Nodejs for CRUD operations.
- Technologies used: Nodejs, Expressjs, Sqlite3, Postman etc.
- Main challenges: configuration of Postman, Error handling etc.
- Timeline: completed in almost 1 month with 4-5 hrs work on 5-6 days.
- Achievements: API is able to do all the required operations.
- Future Use: can be used as the backend database for some fullstack project.
- Improvements: codes can be managed systematically in different files and possible the volume of code could be reduced too.

# Code:

**Setting Up Dependencies and Server:**

```
JS server.js > ...
1   const express = require('express');
2   const sqlite3 = require('sqlite3');
3   const path = require('path');
4
5   const app = express();
6   const port = 8000;
```

**Middleware Configuration:**

```
8   // Middleware to parse JSON in the request body
9   app.use(express.json());
```

The above code on the left, imports necessary modules: express, sqlite3, and path. It also creates an express application and sets the server port to 8000. While the code on the right, configures middleware to parse JSON in the request body.

```
10
11  // Use a persistent SQLite database file
12  const dbPath = path.join(__dirname, 'database.db');
13  const db = new sqlite3.Database(dbPath, (err) => {
14    if (err) {
15      console.error('Error opening database:', err.message);
16    } else {
17      console.log(`Connected to SQLite database at ${dbPath}`);
18
19      // Create users table if not exists
20      db.run(`
21        CREATE TABLE IF NOT EXISTS users (
22          id INTEGER PRIMARY KEY,
23          first_name TEXT,
24          last_name TEXT,
25          city TEXT,
26          department INTEGER
27        )
28      `, (createTableError) => {
29        if (createTableError) {
30          console.error('Error creating table:', createTableError.message);
31        } else {
32          console.log('Users table created or already exists');
33
34          // Insert sample data
35          db.run(`
36            INSERT INTO users (first_name, last_name, city, department)
37            VALUES ('Rohan', 'Dutta', 'Kolkata', 20)
38          `);
39        }
40      });
41    }
42  });
43
44  // CRUD Operations ----------------------------------------
```

# Code:

## Database Initialization:

The code on the picture on the left, connects to an SQLite database (database.db) and creates a users table if it doesn't exist. It also inserts sample data into the users table. The error handling is also done in this code.

# Code:

```
// CRUD Operations ----------------------------------------

// Create - Add a new user
app.post('/users', (req, res) => {
  const { first_name, last_name, city, department } = req.body;

  if (!first_name || !last_name || !city || !department) {
    return res.status(400).json({ error: 'All fields are required' });
  }

  const insertQuery = `
    INSERT INTO users (first_name, last_name, city, department)
    VALUES (?, ?, ?, ?)
  `;

  db.run(insertQuery, [first_name, last_name, city, department], function (err) {
    if (err) {
      console.error(err.message);
      return res.status(500).json({ error: 'Internal Server Error' });
    }

    res.status(201).json({ message: 'User added successfully', userId: this.lastID });
  });
});
```

## Create Operation (HTTP POST):

- The code on the picture on the left, Implements an endpoint for adding a new user.
- Validates the presence of required fields and responds with appropriate status.

```
69
70
71  // GET route for both all users and a single user
72  app.get('/users', (req, res) => {
73    const queryParams = req.query;
74
75    // If there are query parameters, construct the WHERE clause
76    let whereClause = '';
77    const values = [];
78
79    Object.keys(queryParams).forEach((key, index) => {
80      if (index > 0) {
81        whereClause += ' AND ';
82      }
83
84      const valuesArray = queryParams[key].split(',');
85
86      if (valuesArray.length > 1) {
87        // Handle OR condition
88        whereClause += `${key} IN (${valuesArray.map(() => '?').join(', ')})`;
89        values.push(...valuesArray);
90      } else {
91        // Handle AND condition
92        whereClause += `${key} = ?`;
93        values.push(valuesArray[0]);
94      }
95    });
96
97    // Query to retrieve user data
98    let selectQuery = 'SELECT * FROM users';
99
100   // If there is a WHERE clause, append it to the query
101   if (whereClause !== '') {
102     selectQuery += ` WHERE ${whereClause}`;
103   }
104
105   db.all(selectQuery, values, (err, rows) => {
106     if (err) {
107       console.error(err.message);
108       return res.status(500).json({ error: 'Internal Server Error' });
109     }
110
111     console.log('Retrieved user data:', rows); //  line for logging
112
113     res.json({ users: rows });
114   });
115 });
116
```

# Code:

## Read Operations (HTTP GET):

The code on the picture on the left, Fetches all users based on query parameters (AND and OR conditions). Similarly, there is also code to use the GET method to retrieve a single user by ID which is not visible on this picture due to space issues. The code looks like this: *app.get('/users/:id', (req, res) => { /* ... */ });*

```
142
143   // Update - Modify user information
144   app.patch('/users/:id', (req, res) => {
145     console.log('PATCH Request Received'); // log for testing purpose
146     console.log('User ID from URL:', req.params.id); // log for testing purpose
147     console.log('Request Body:', req.body); // log for testing purpose
148
149     const { first_name, last_name, city, department } = req.body;
150
151     const updateFields = [];
152     const updateValues = [];
153
154     // Check if each field is provided and add to the update query
155     if (first_name !== undefined) {
156       updateFields.push('first_name = ?');
157       updateValues.push(first_name);
158     }
159     if (last_name !== undefined) {
160       updateFields.push('last_name = ?');
161       updateValues.push(last_name);
162     }
163     if (city !== undefined) {
164       updateFields.push('city = ?');
165       updateValues.push(city);
166     }
167     if (department !== undefined) {
168       updateFields.push('department = ?');
169       updateValues.push(department);
170     }
171
172     const updateQuery = `
173       UPDATE users
174       SET ${updateFields.join(', ')}
175       WHERE id = ?
176     `;
177
178     console.log('Generated SQL Query:', updateQuery);
179
180     db.run(
181       updateQuery,
182       [...updateValues, req.params.id],
183       function (err) {
184         if (err) {
185           console.error('Error during update', err.message);
186           return res.status(500).json({ error: 'Internal Server Error' });
187         }
188
189         res.json({ message: `User with ID ${req.params.id} updated successfully` });
190       }
191     );
192   });
```

# Code:

## Update Operations (HTTP PATCH and PUT):

- The code on the picture on the left, Implements PATCH for partial updates based on provided fields.
- There is also kind of similar code which implements PUT for replacing the entire user information, but it is not visible here.
- Includes improved error handling middleware.

**Code example:**

*app.patch('/users/:id', (req, res) => { /* ... */ });*

*app.put('/users/:id', (req, res) => { /* ... */ });*

```
227
228  // Delete - Remove a user
229  app.delete('/users/:id', (req, res) => {
230    const deleteQuery = 'DELETE FROM users WHERE id = ?';
231
232    db.run(deleteQuery, [req.params.id], function (err) {
233      if (err) {
234        console.error(err.message);
235        return res.status(500).json({ error: 'Internal Server Error' });
236      }
237
238      res.json({ message: `User with ID ${req.params.id} deleted successfully` });
239    });
240  });
241
242  // Improved Error Handling
243  app.use((err, req, res, next) => {
244    console.error(err.stack);
245    res.status(500).json({ error: 'Internal Server Error' });
246  });
247
248  // Listen for requests
249  app.listen(port, () => {
250    console.log(`Server is running on http://localhost:${port}`);
251  });
252
253
254
```

# Code:

## Delete Operation (HTTP DELETE) + Error Handling and Port Listen:

- The code on the picture on the left, implements an endpoint to remove a user based on their ID.

- There is code which Provides improved error handling for the entire application.

- There is also code which starts the server and listens on port 8000.

**Code example:**

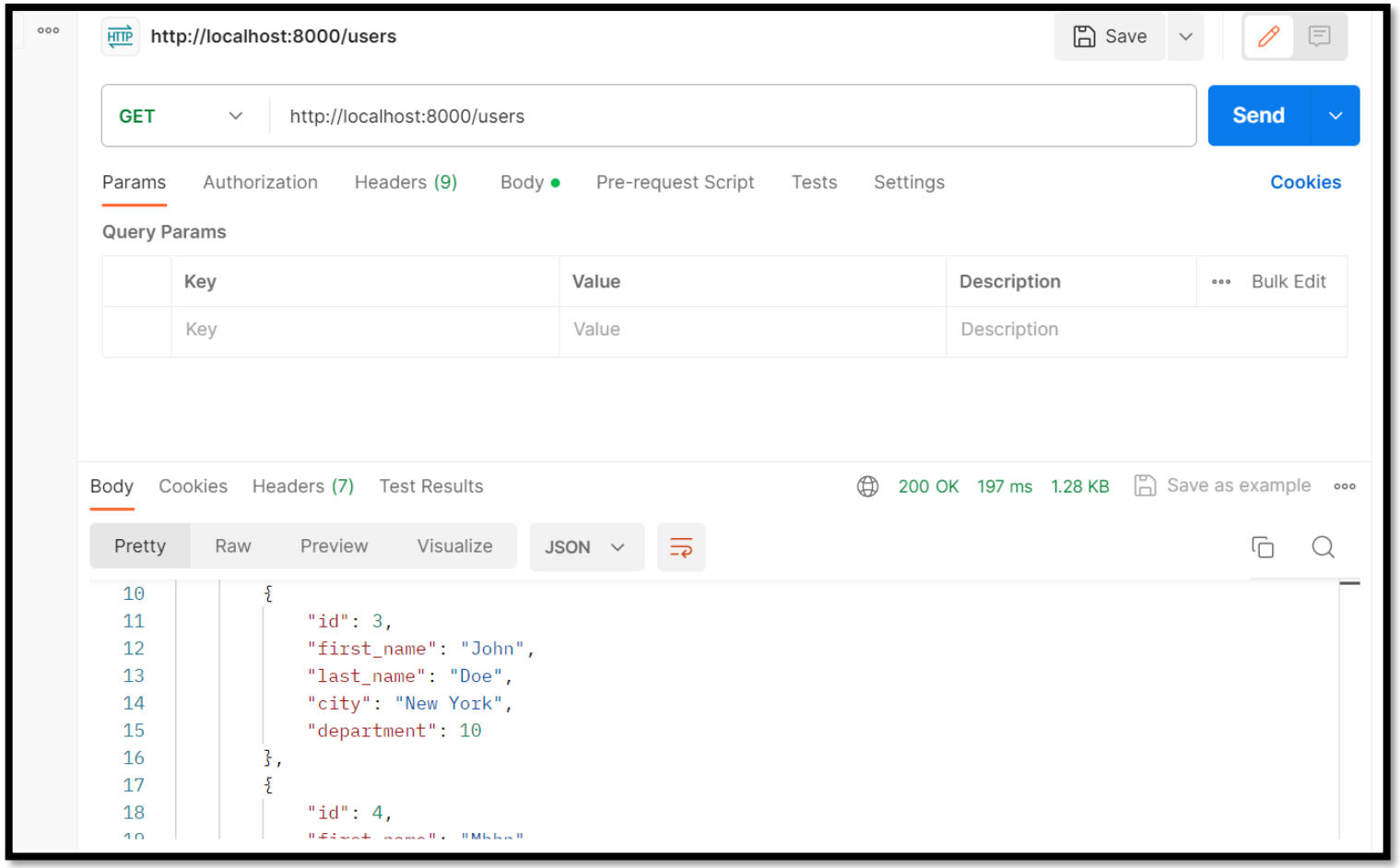
*For delete:*

*app.delete('/users/:id', (req, res) => { /* ... */ });*

*For error handling:*

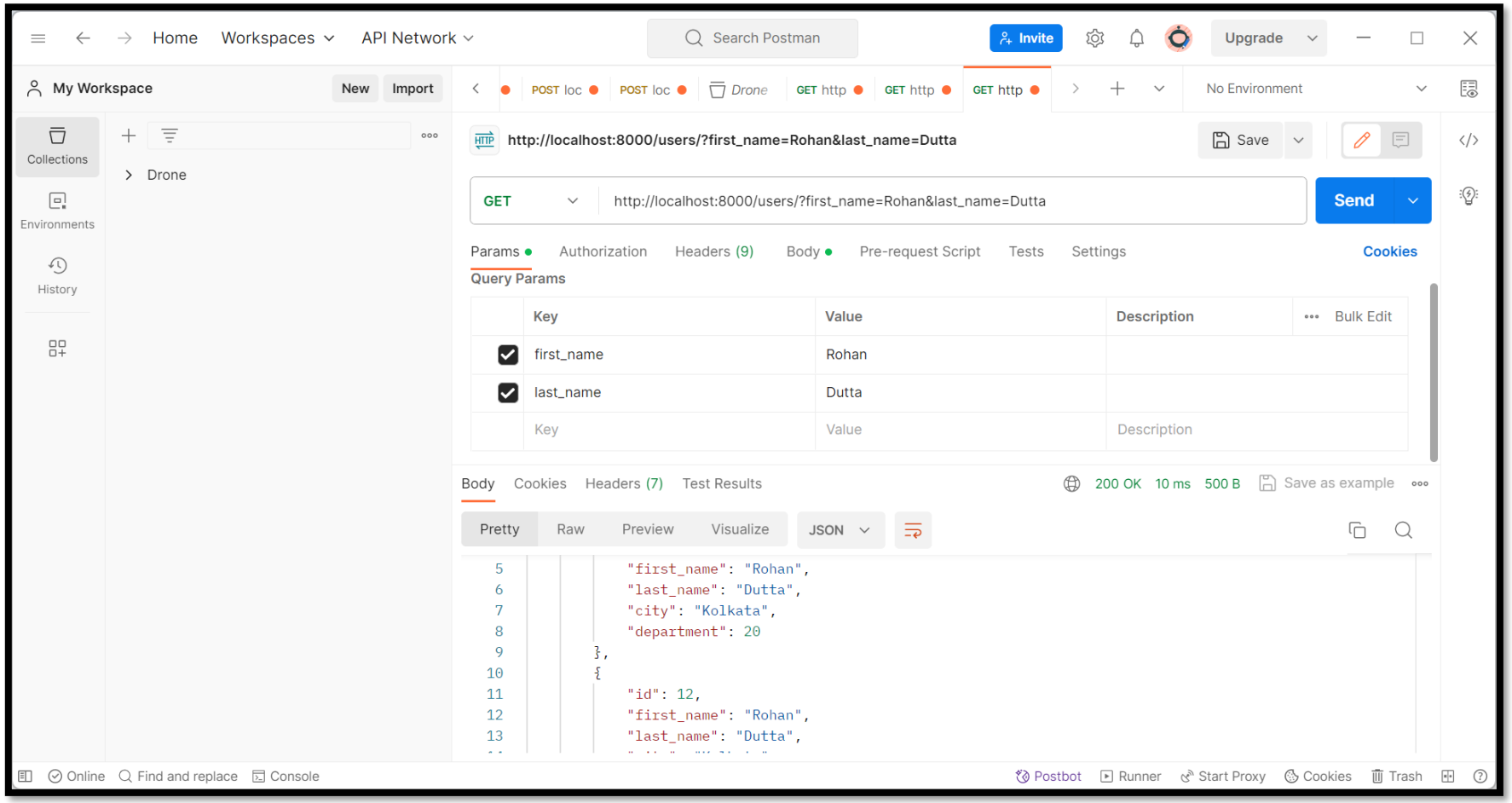
*app.use((err, req, res, next) => { /* ... */ });*

# Test Calls to API:



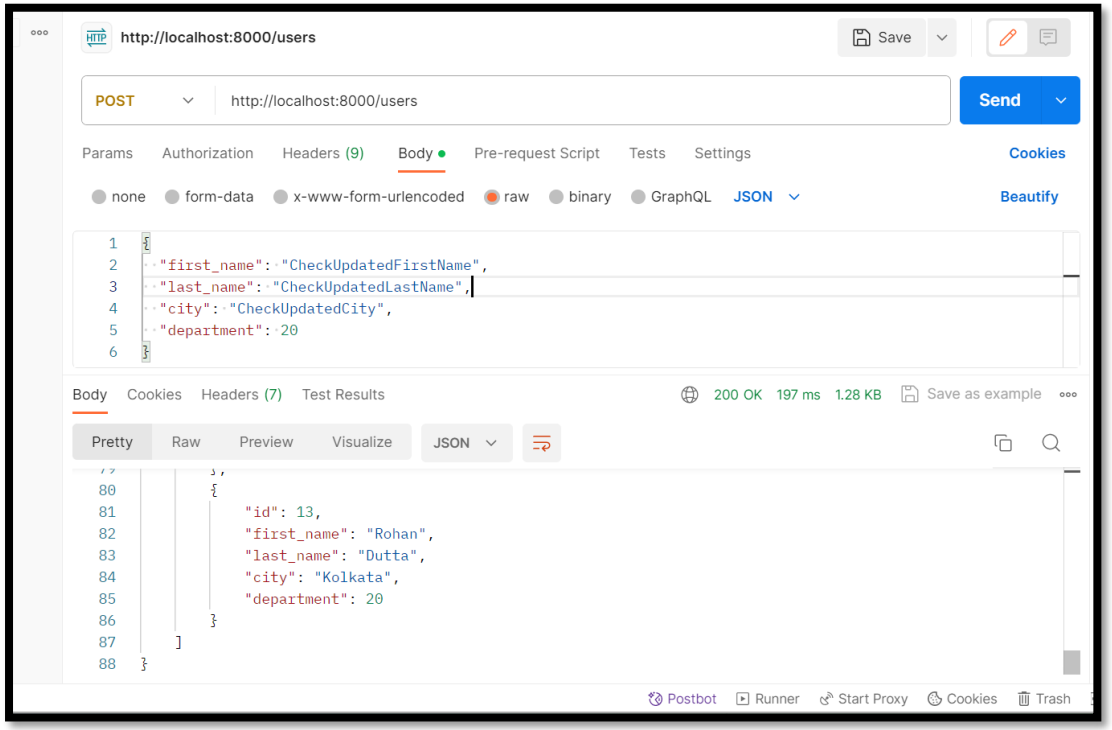

GET method to list all users

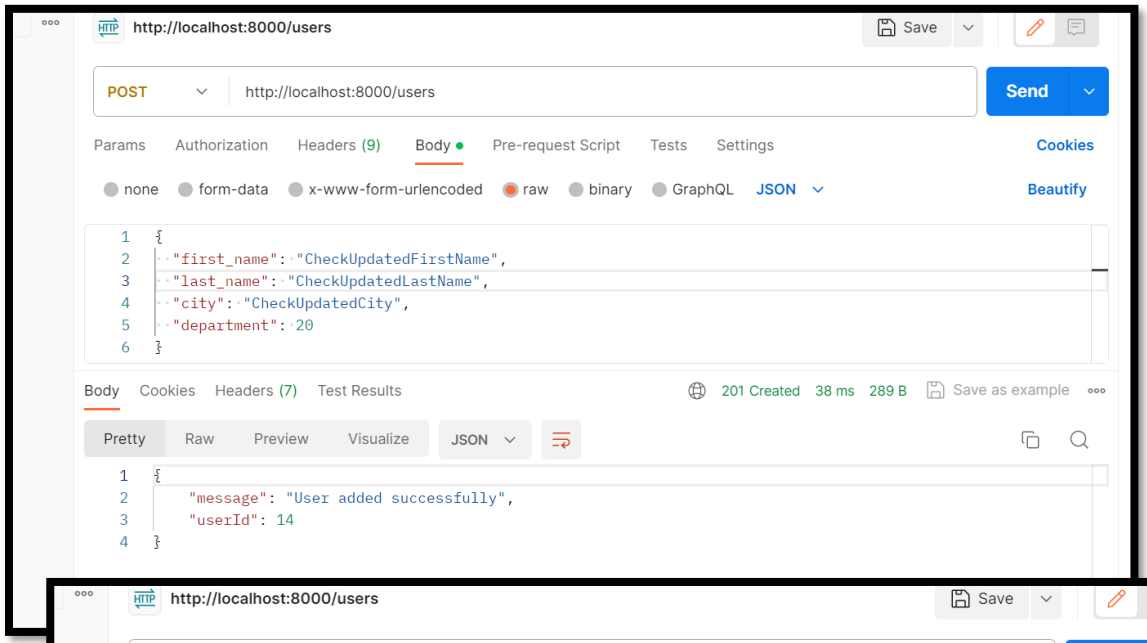
# Test Calls to API ‘GET’:

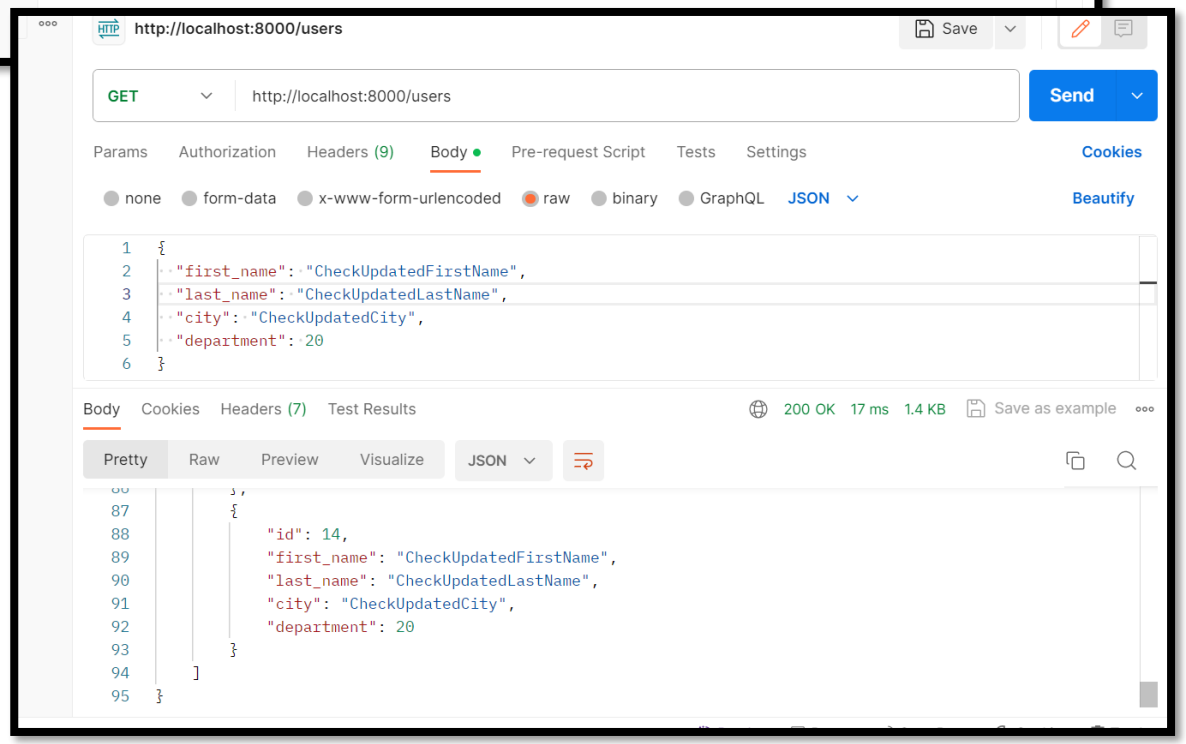



GET method to list users by first_name & last_name. Similarly other attributes could be used for searching too, using AND, OR method or even by user id too the end point should be users/id. Id is replaced by the desired search id. For example, for id 2 users/2.

# Test Calls to API ‘POST’:

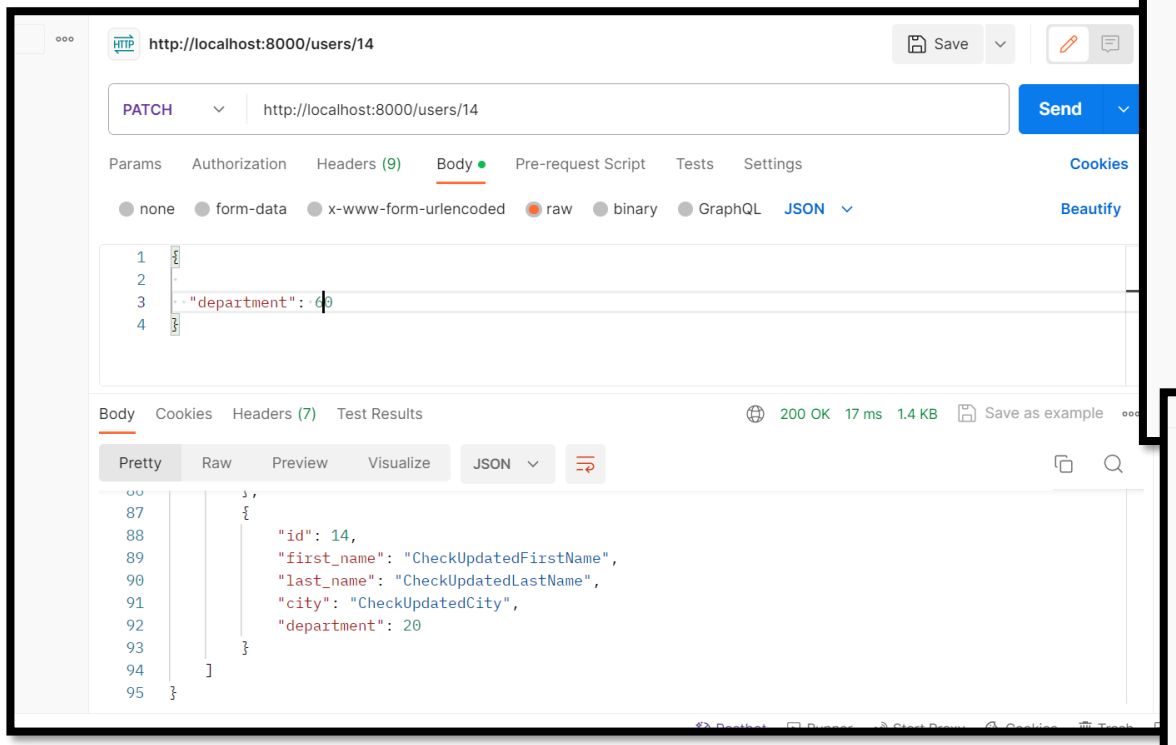


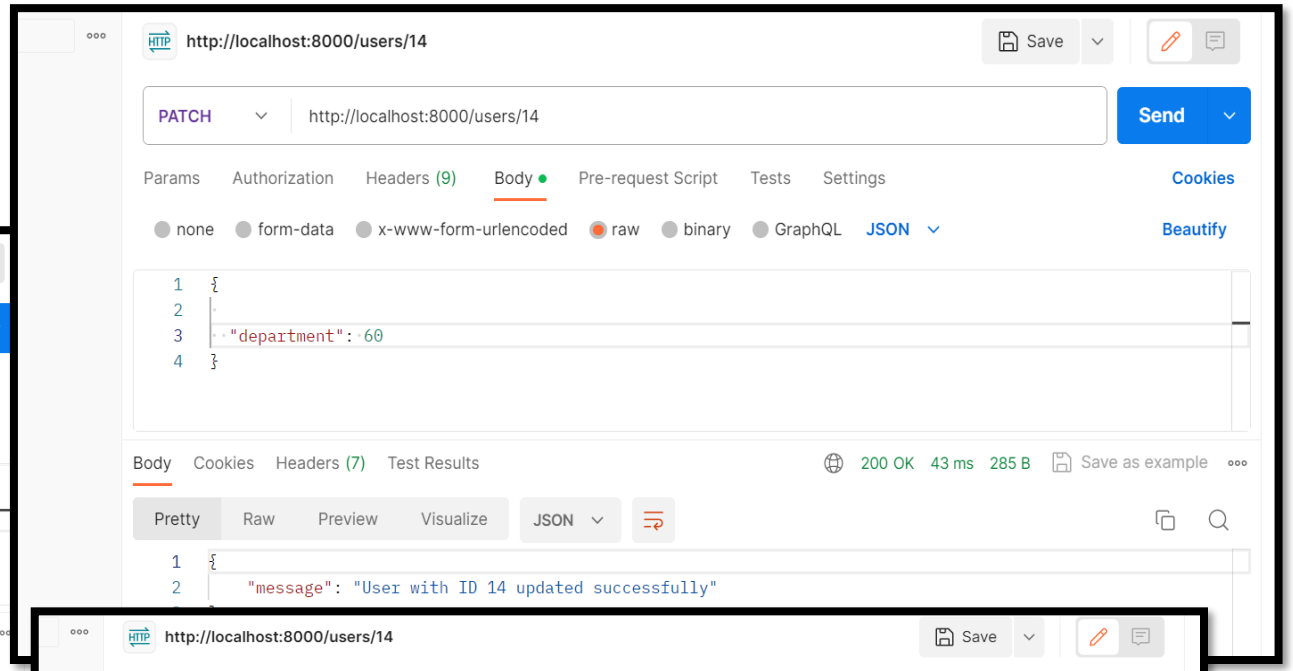



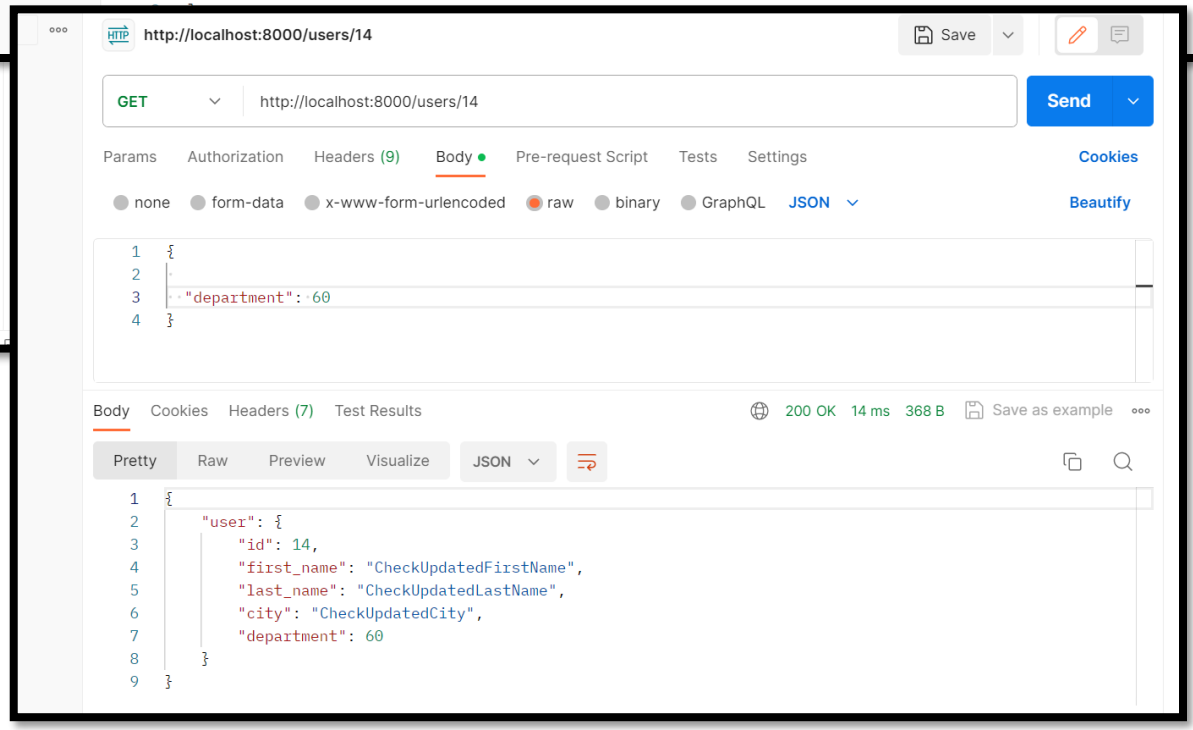




As we can see on the pictures above on the first picture left, POST method is used to post new user. On the right side top the we can see the message user added successfully and UserId 14 allotted. At the right down picture, we can confirm the results.

# Test Calls to API ‘PATCH’:

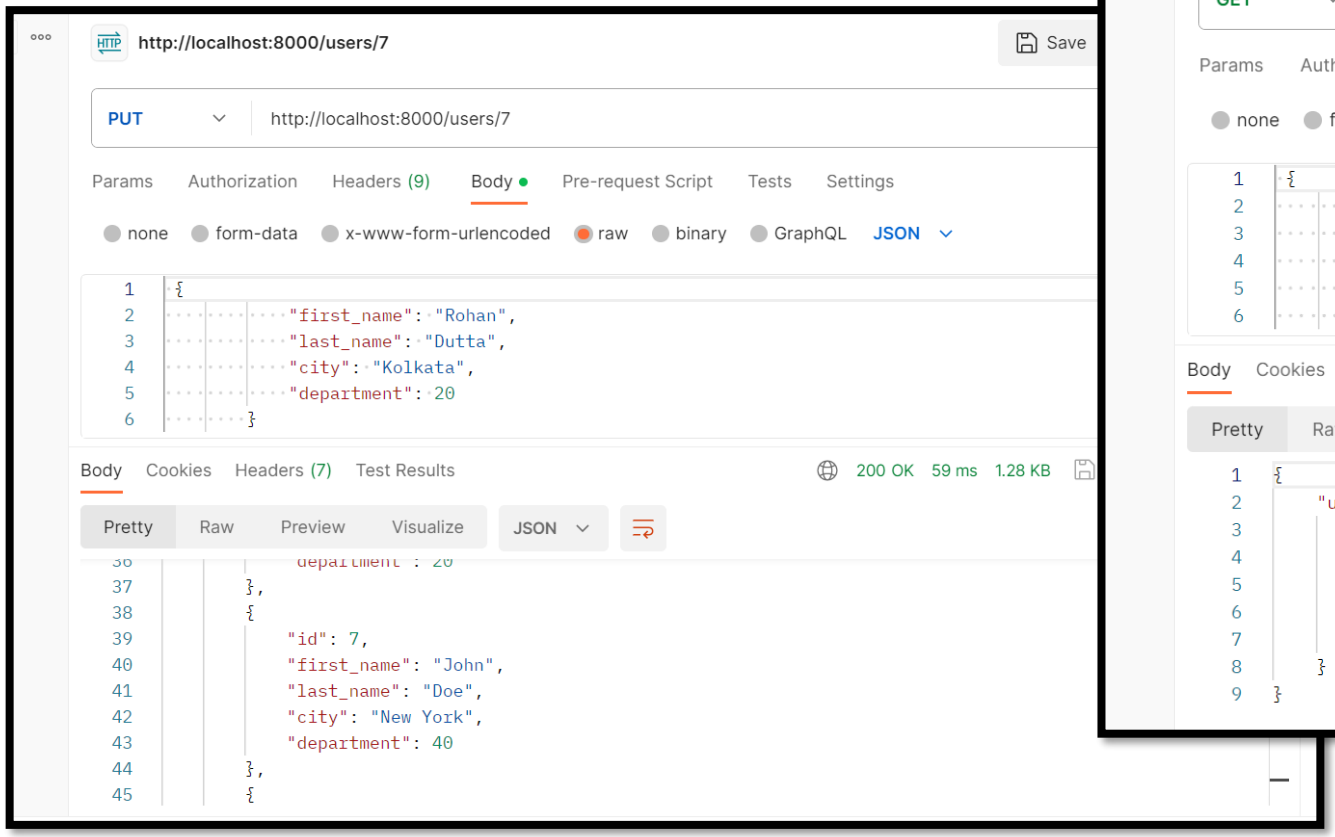


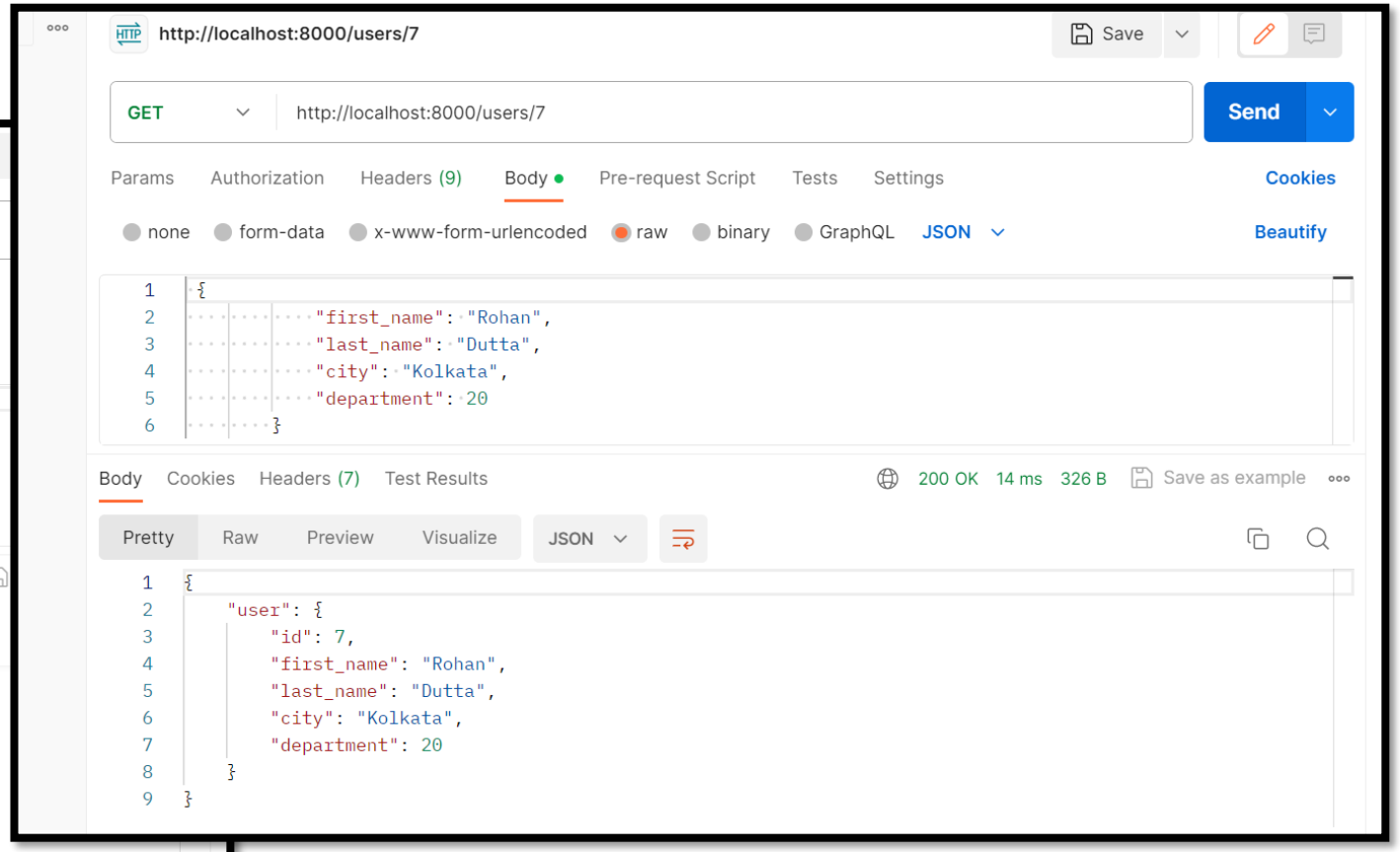




As we can see on the pictures above on the first picture left, PATCH method is used to update certain data of the particular id = 14 user. On the right side top the we can see the message user updated successfully with UserId 14. At the right down picture, we can confirm the results.

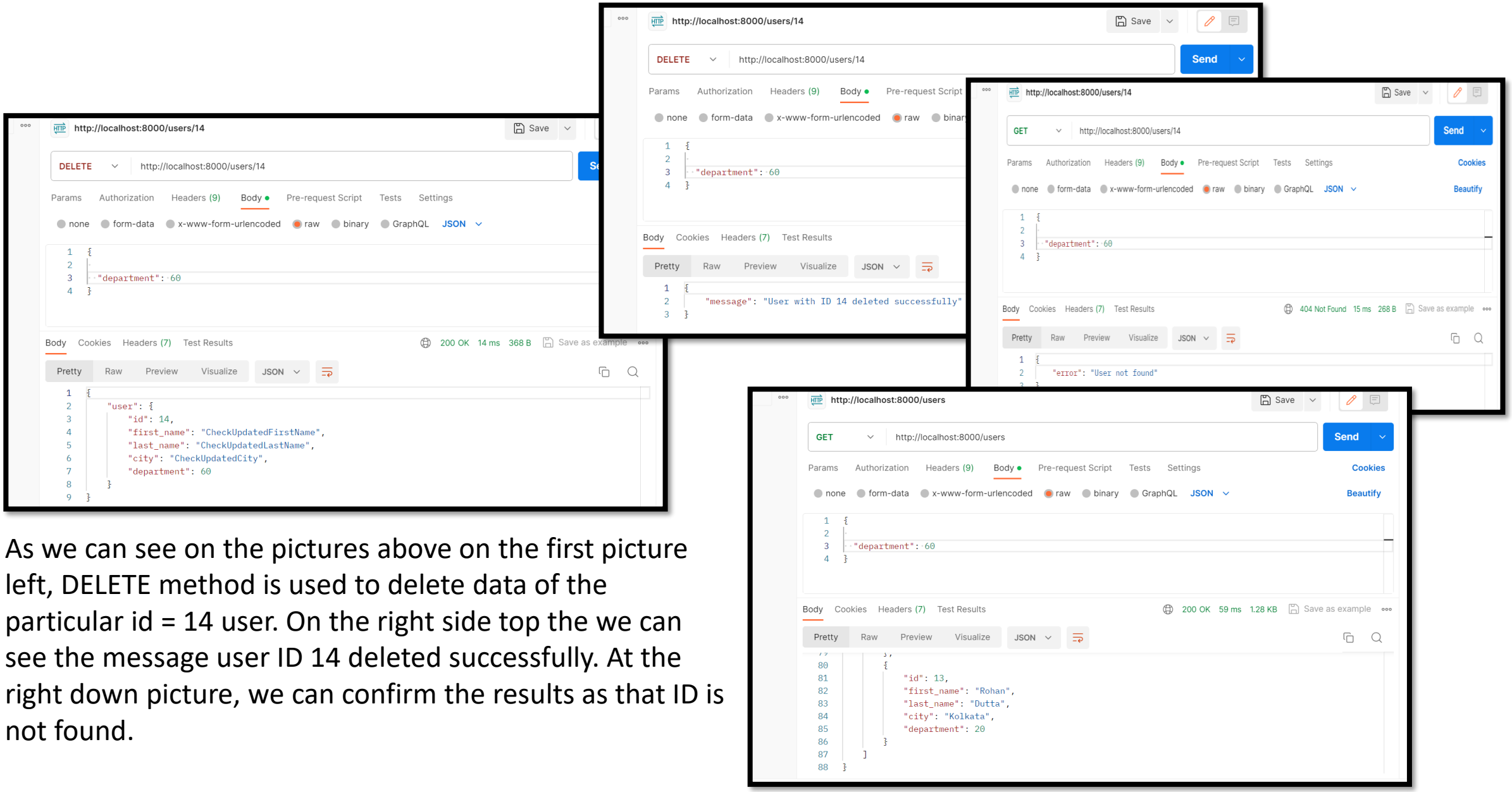
# Test Calls to API ‘PUT’:




As we can see on the pictures above on the first picture left, PUT method is used to update the whole data of the particular id = 7 user. On the right we can confirm the results as that ID 7 has the updated info.

# Test Calls to API ‘DELETE’:




As we can see on the pictures above on the first picture left, DELETE method is used to delete data of the particular id = 14 user. On the right side top the we can see the message user ID 14 deleted successfully. At the right down picture, we can confirm the results as that ID is not found.

# References


- https://datatracker.ietf.org/doc/html/rfc7231#section-4.3.3
- https://nodejs.org/api/errors.html
- https://www.postman.com/
- https://www.sqlite.org/index.html
- https://www.json.org/json-en.html
- https://expressjs.com/

# Thank you !