EXPERIMENT 6

AIM:

Building a RESTful API with Express

Experiment: Create routes for CRUD operations (Create, Read, Update, Delete) on a mock dataset. Use Express middleware for request handling and validation.

THEORY:

A RESTful API (Representational State Transfer Application Programming Interface) is a web service architectural style that uses standard HTTP methods to perform operations on resources. REST is based on the principles of stateless communication, resource-based URIs, and the use of standard HTTP verbs such as GET, POST, PUT, and DELETE. These verbs correspond to the basic CRUD (Create, Read, Update, Delete) operations that are used to manage data.

In a RESTful API, each resource (such as a user, product, or article) is identified by a unique URI, and each operation on the resource is performed using a specific HTTP method. For example, retrieving a list of users would involve a GET request to /users, while creating a new user would involve a POST request to the same endpoint. Similarly, updating a specific user would involve a PUT request to /users/:id, and deleting a user would use a DELETE request to the same URI.

Express.js is a web application framework for Node.js that simplifies the development of web servers and APIs. It provides a robust set of features for building single-page, multi-page, and hybrid web applications. Express allows developers to define routes for handling different HTTP methods and to use middleware functions for processing requests before they reach the route handler.

Middleware in Express plays a critical role in handling requests and responses. Middleware functions can be used for a variety of purposes such as parsing JSON data, logging request information, handling authentication, and validating user input. Middleware functions have access to the request and response objects and can either end the request-response cycle or pass control to the next middleware in the stack using the next() function.

Validation is an essential part of API development as it ensures that the data being received from clients is complete and follows the expected format. In this experiment, a custom middleware function is used to validate that required fields like "name" and

"email" are present in the request body before processing the data. If validation fails, the server returns an appropriate error response with a suitable status code.

The mock dataset used in this experiment acts as an in-memory database that stores user information temporarily for the duration of the application's runtime. Although not persistent, it is sufficient for demonstrating the structure and functionality of a RESTful API.

This experiment also involves using Postman, a tool for testing APIs by sending HTTP requests and viewing responses. Postman allows for easy testing of each route by enabling the user to send different types of requests (GET, POST, PUT, DELETE) with customized parameters and payloads.

By combining Express routing, middleware, and validation techniques, this experiment demonstrates how to build a clean, organized, and functional RESTful API that adheres to modern web development standards.

INPUT:
Filename: server.js

```
const express = require('express');
const app = express();
const PORT = 3000;

// Middleware to parse JSON
app.use(express.json());

// Logger middleware
app.use((req, res, next) => {
  console.log(`${req.method} ${req.url}`);
  next();
});

// Mock data
let users = [
  { id: 1, name: 'Alice', email: 'alice@example.com' },
  { id: 2, name: 'Bob', email: 'bob@example.com' },
];

// Validation middleware
```

```
function validateUser(req, res, next) {
  const { name, email } = req.body;
  if (!name || !email) {
    return res.status(400).json({ error: 'Name and email are required' });
  }
  next();
}

// Routes
app.get('/users', (req, res) => res.json(users));

app.get('/users/:id', (req, res) => {
  const user = users.find(u => u.id === parseInt(req.params.id));
  if (!user) return res.status(404).json({ error: 'User not found' });
  res.json(user);
});

app.post('/users', validateUser, (req, res) => {
  const newUser = {
    id: users.length + 1,
    name: req.body.name,
    email: req.body.email
  };
  users.push(newUser);
  res.status(201).json(newUser);
});

app.put('/users/:id', validateUser, (req, res) => {
  const user = users.find(u => u.id === parseInt(req.params.id));
  if (!user) return res.status(404).json({ error: 'User not found' });

  user.name = req.body.name;
  user.email = req.body.email;
  res.json(user);
});

app.delete('/users/:id', (req, res) => {
  const index = users.findIndex(u => u.id === parseInt(req.params.id));
  if (index === -1) return res.status(404).json({ error: 'User not found' });
```

```
  users.splice(index, 1);
  res.json({ message: 'User deleted' });
});

// Start server
app.listen(PORT, () => {
  console.log(`Server running at http://localhost:${PORT}`);
});
```

OUTPUT:

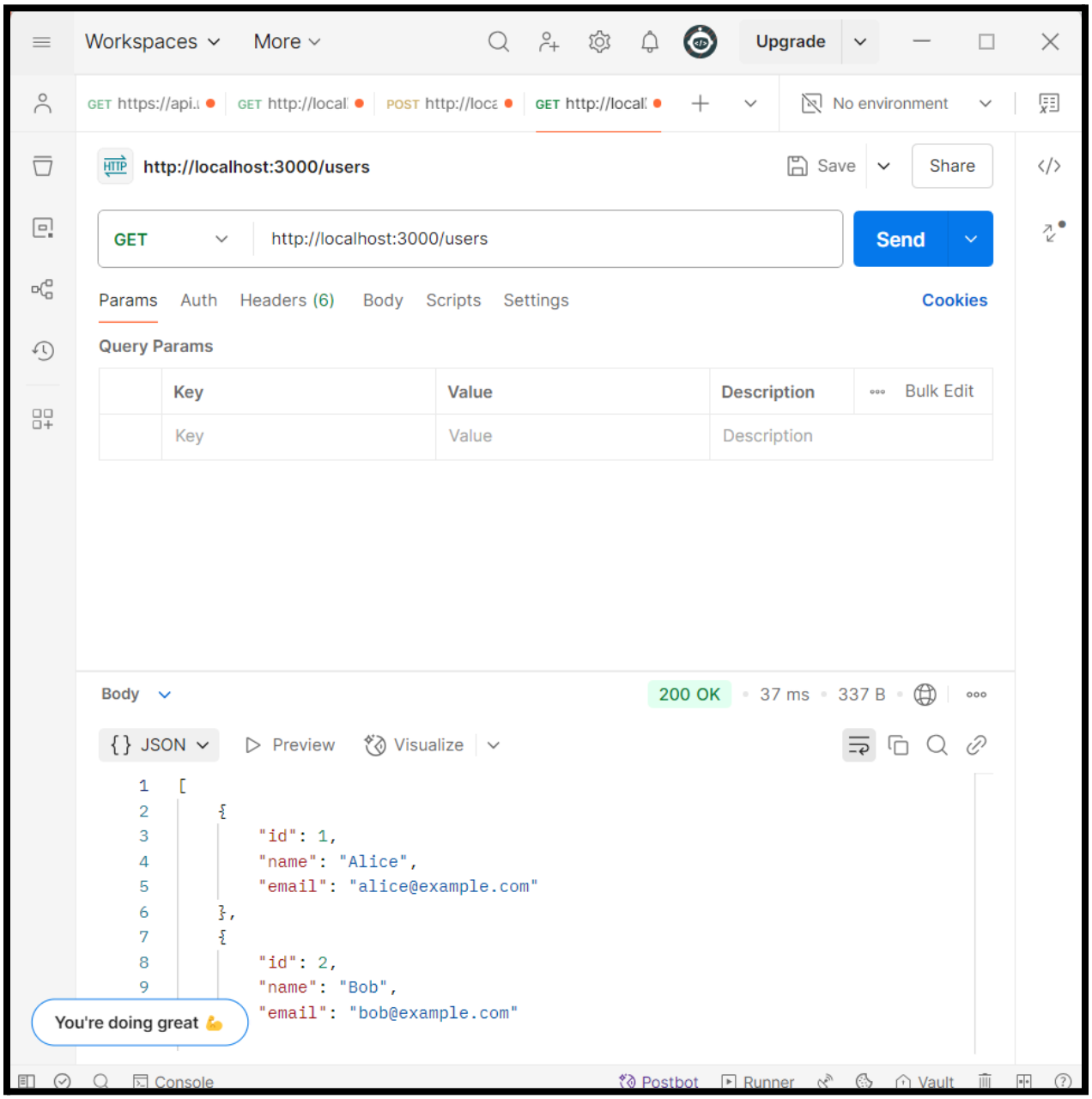
In this experiment, we successfully developed a RESTful API using Node.js and Express.js that performs CRUD operations on a mock dataset. We implemented Express middleware for JSON body parsing, request logging, and request validation to ensure data integrity and maintainability.

The API follows RESTful design principles by using appropriate HTTP methods and structured URIs to interact with the users resource. The modular use of middleware enhances code readability and reusability, making the API scalable and efficient.

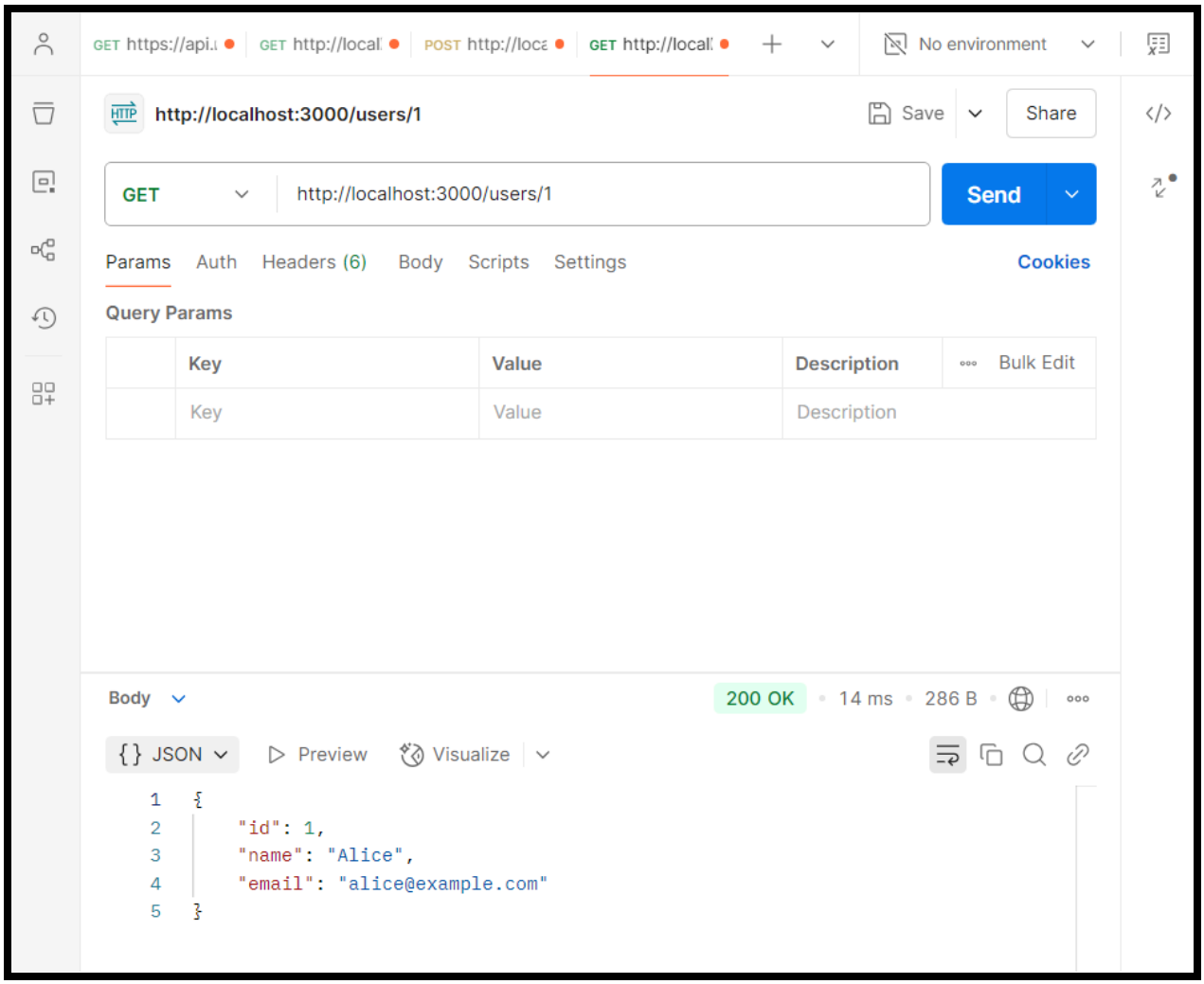
This experiment demonstrated how REST APIs are constructed in real-world web applications, laying the foundation for connecting client-side applications (like Postman or frontend frameworks) with backend services.

```
E:\FSWD_LAB>cd EXP_6

E:\FSWD_LAB\EXP_6>mkdir express-crud-api

E:\FSWD_LAB\EXP_6>cd express-crud-api

E:\FSWD_LAB\EXP_6\express-crud-api>npm init -y

Wrote to E:\FSWD_LAB\EXP_6\express-crud-api\pa
ckage.json:

{
  "name": "express-crud-api",
  "version": "1.0.0",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\"
 && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "description": ""
}



E:\FSWD_LAB\EXP_6\express-crud-api>
E:\FSWD_LAB\EXP_6\express-crud-api>npm install
 express

added 66 packages, and audited 67 packages in
12s

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

found 0 vulnerabilities

E:\FSWD_LAB\EXP_6\express-crud-api>
```

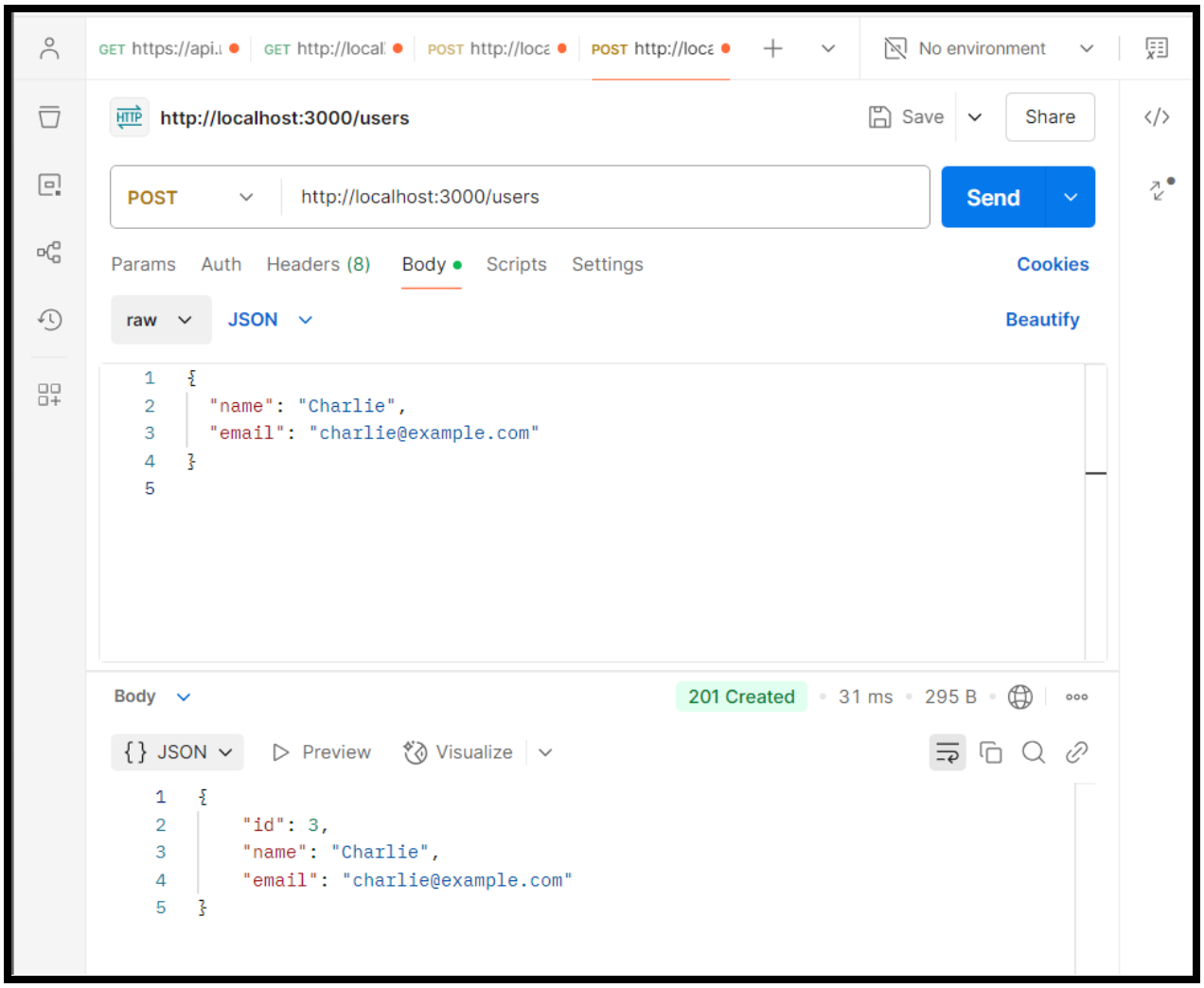
GET



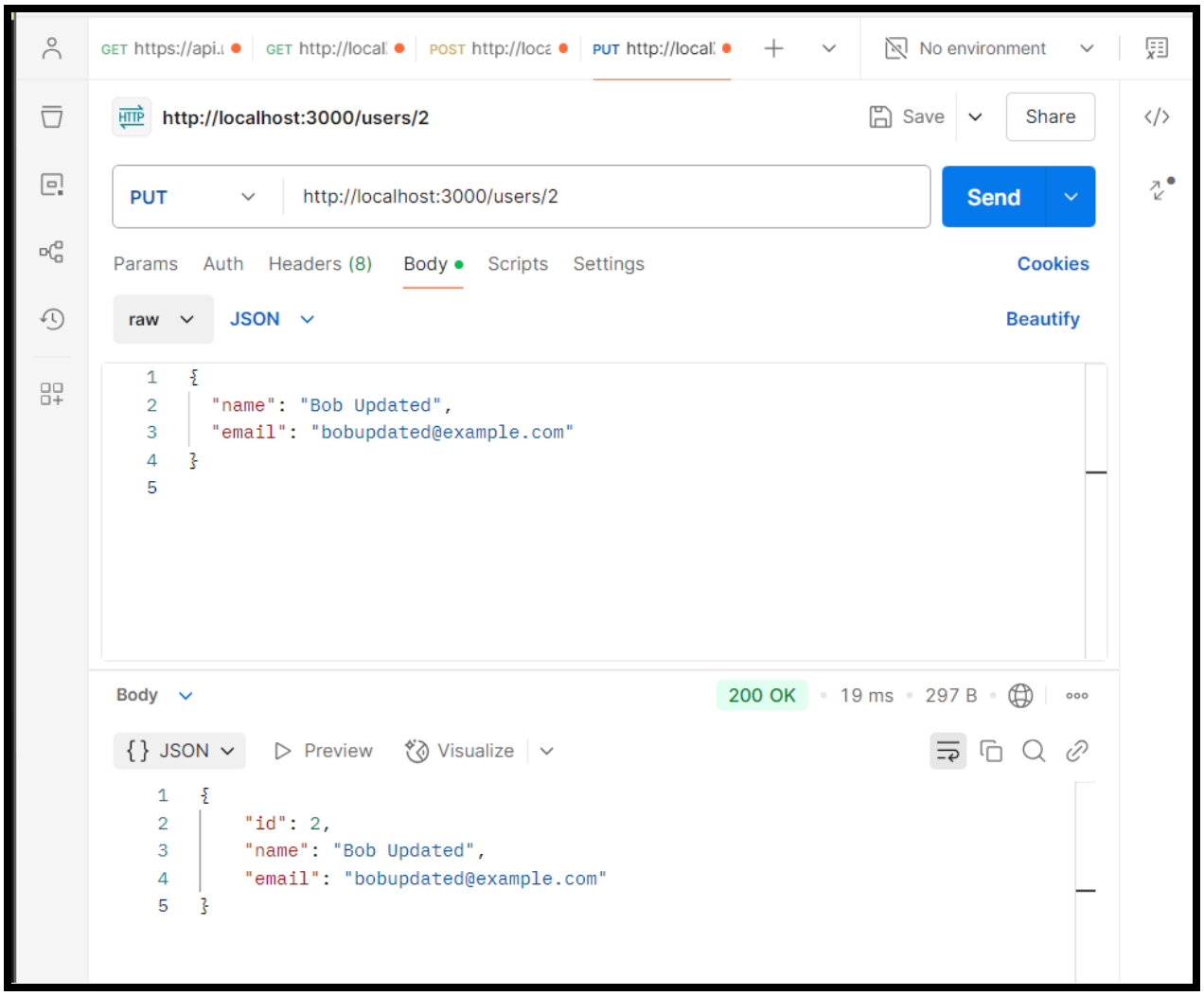
GET ONLY USER 1



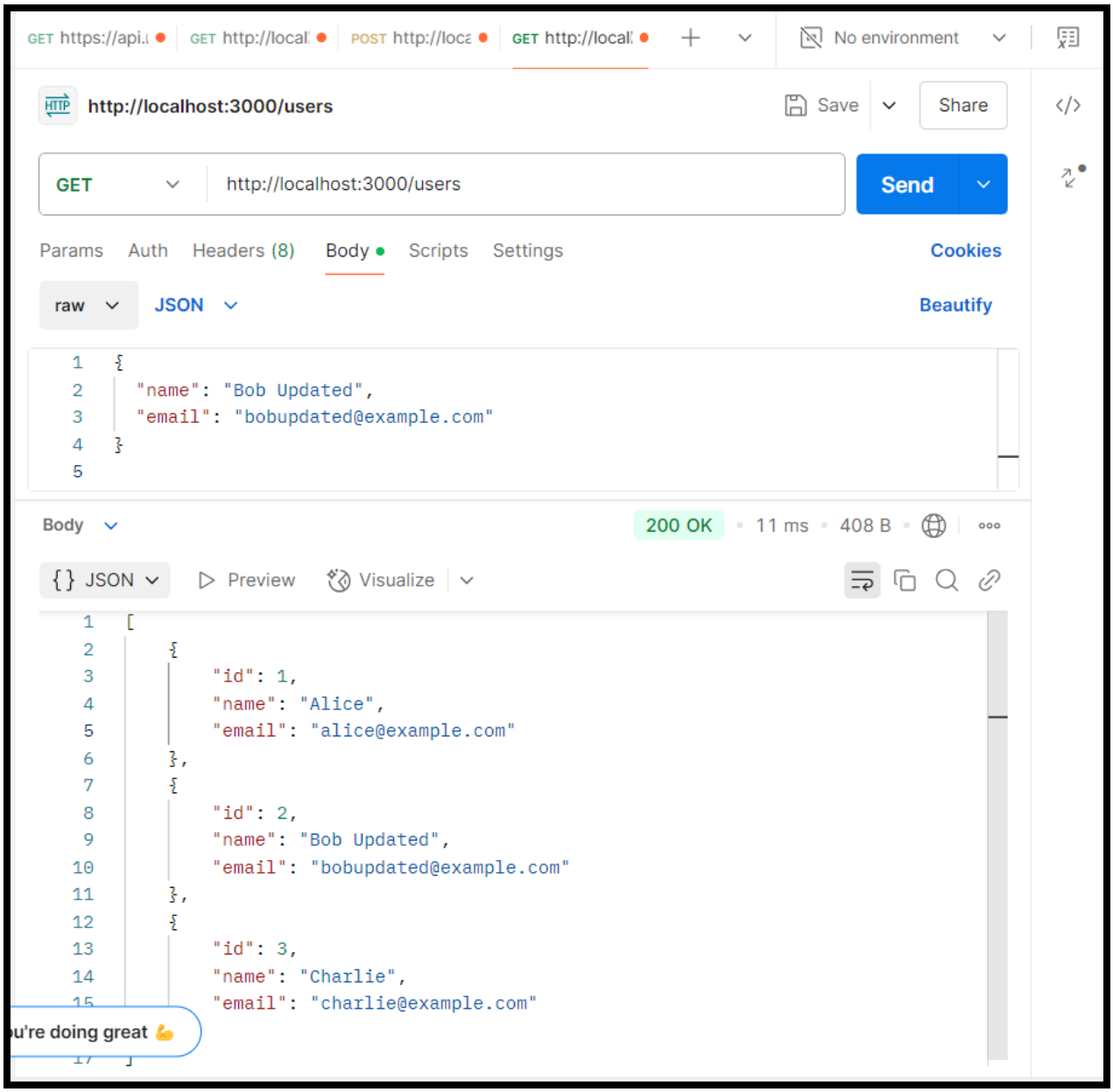
CREATE NEW USER


GET https://api.
GET http://local
POST http://loca
POST http://loca
No environment
http://localhost:3000/users
Save
Share
POST
http://localhost:3000/users
Send
Params
Auth
Headers (8)
Body
Scripts
Settings
Cookies
raw
JSON
Beautify
{
"name": "Charlie",
"email": "charlie@example.com"
}
Body
201 Created
31 ms
295 B
{} JSON
Preview
Visualize
{
"id": 3,
"name": "Charlie",
"email": "charlie@example.com"
}

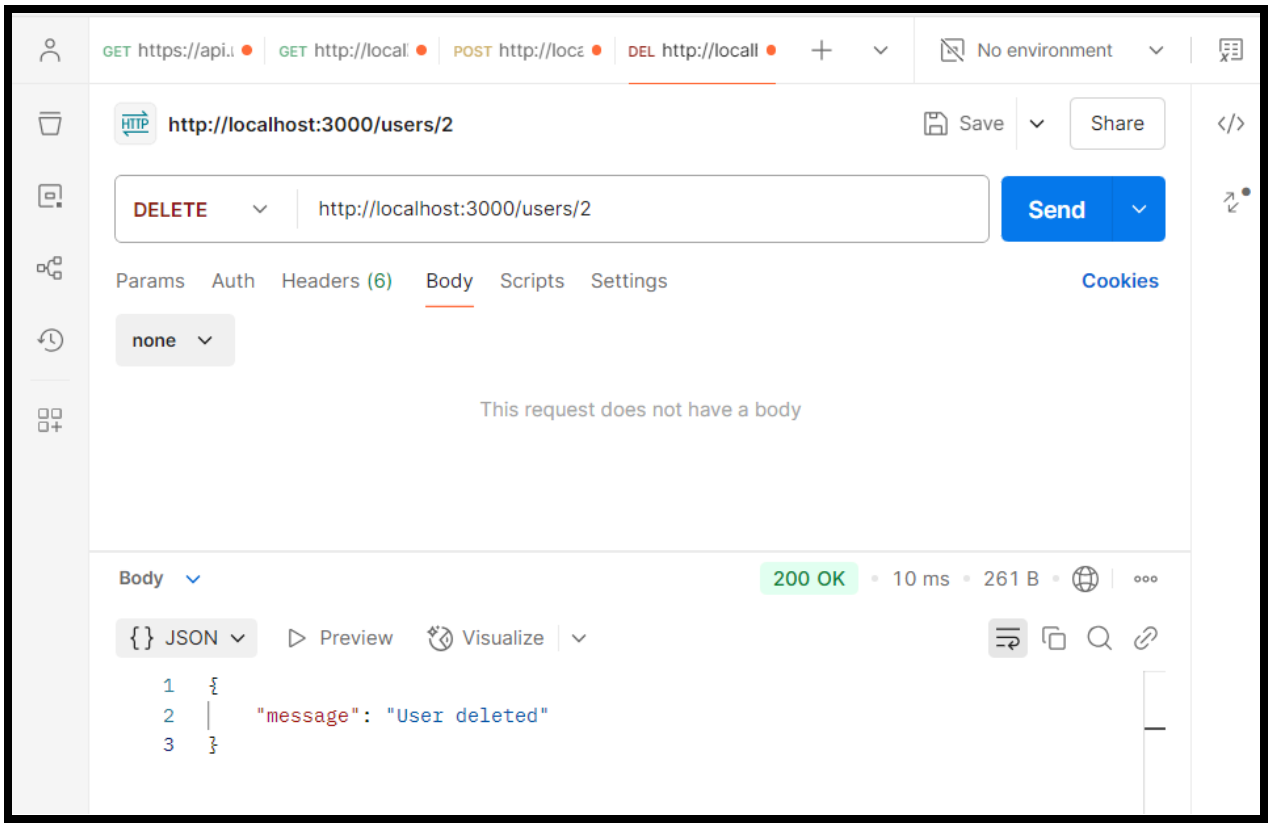
UPDATE USER

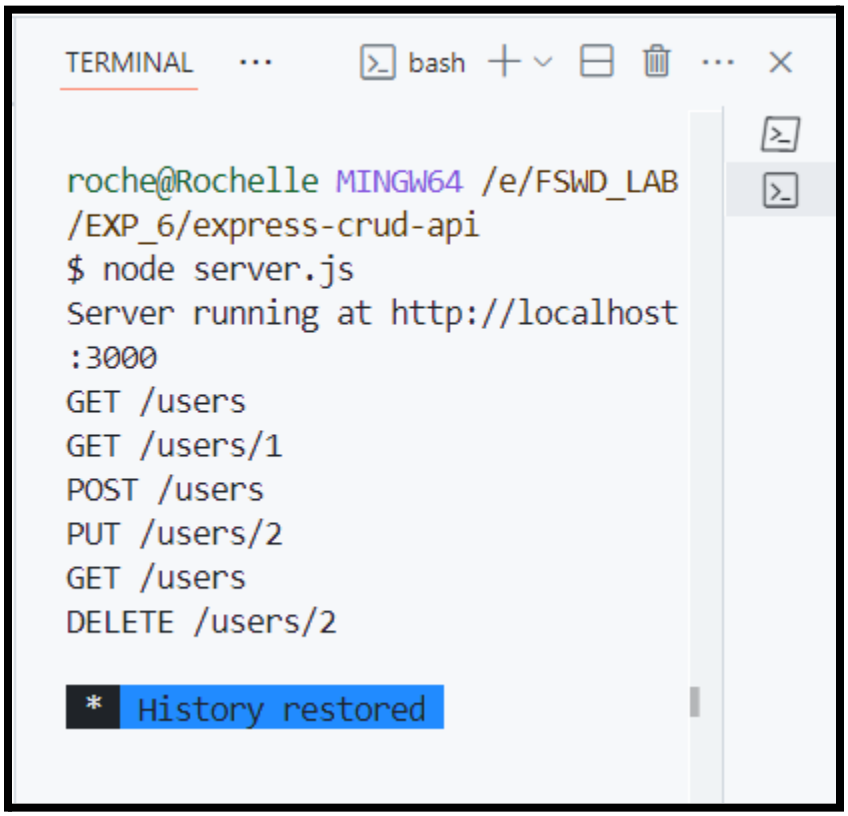


GET WITH UPDATED VALUES


GET https://api.
GET http://local
POST http://loca
GET http://local
No environment
http://localhost:3000/users
Save
Share
GET
http://localhost:3000/users
Send
Params
Auth
Headers (8)
Body
Scripts
Settings
Cookies
raw
JSON
Beautify
{
"name": "Bob Updated",
"email": "bobupdated@example.com"
}
Body
200 OK
11 ms
408 B
JSON
Preview
Visualize
[
{
"id": 1,
"name": "Alice",
"email": "alice@example.com"
},
{
"id": 2,
"name": "Bob Updated",
"email": "bobupdated@example.com"
},
{
"id": 3,
"name": "Charlie",
"email": "charlie@example.com"
u're doing great

DELETE USER


GET https://api.
GET http://local
POST http://loca
DEL http://locall
No environment
http://localhost:3000/users/2
Save
Share
DELETE
http://localhost:3000/users/2
Send
Params
Auth
Headers (6)
Body
Scripts
Settings
Cookies
none
This request does not have a body
Body
200 OK
10 ms
261 B
{} JSON
Preview
Visualize
1 {
2 "message": "User deleted"
3 }


TERMINAL
bash
roche@Rochelle MINGW64 /e/FSWD_LAB
/EXP_6/express-crud-api
$ node server.js
Server running at http://localhost
:3000
GET /users
GET /users/1
POST /users
PUT /users/2
GET /users
DELETE /users/2
* History restored

CONCLUSION:

In this experiment, we successfully developed a RESTful API using Node.js and Express.js that performs CRUD operations on a mock dataset. We implemented Express middleware for JSON body parsing, request logging, and request validation to ensure data integrity and maintainability.

The API follows RESTful design principles by using appropriate HTTP methods and structured URIs to interact with the users resource. The modular use of middleware enhances code readability and reusability, making the API scalable and efficient.

This experiment demonstrated how REST APIs are constructed in real-world web applications, laying the foundation for connecting client-side applications (like Postman or frontend frameworks) with backend services.