Jafar Beldar

Day 4 Assignment

1. What is the http module in Node.js used for?

The `http` module in Node.js is used to create web servers and handle HTTP requests and responses. It allows developers to build server-side applications without third-party packages. With the `http` module, we can create an HTTP server that listens to requests and responds accordingly. It is useful for creating APIs, web applications, and other network-based services.

2. How does the Node.js event-driven architecture benefit HTTP handling?

Node.js follows an event-driven, non-blocking architecture, which makes it highly efficient for handling multiple HTTP requests simultaneously. Instead of waiting for one request to complete before moving to the next, Node.js uses an asynchronous model, allowing the system to continue processing other requests. This results in better performance, lower resource usage, and faster response times, making it ideal for real-time applications.

3. Differentiate between http.createServer() and http.request().

- `http.createServer()`: Used to create an HTTP server that listens for incoming client requests. It allows handling routes, responses, and headers.
- `http.request()`: Used to make an outbound HTTP request to another server, usually to fetch data from an external API or service. It is useful when a Node.js application needs to communicate with another web service.

4. What is the purpose of the request and response objects in an HTTP server?

- The `request` (`req`) object contains details about the client's request, such as URL, method (GET, POST, etc.), headers, and body data.
- The `response` (`res`) object is used to send data back to the client, including setting headers, status codes, and response content.

5. Explain the role of the statusCode and setHeader() methods in the response object.

- `statusCode`: Defines the HTTP status code sent in the response (e.g., 200 for success, 404 for not found, 500 for server error). It informs the client about the result of their request.
- `setHeader()`: Used to set custom headers in the response, such as content type (`Content-Type: application/json`), caching policies, and security headers. This helps control how the client interprets the response.

◇ API Architecture

6. What is the difference between a monolithic and microservices API architecture?

Monolithic Architecture:
- In a monolithic API architecture, all components (UI, business logic, and database operations) are combined into a single system.
- The entire application is developed, deployed, and scaled as one unit.
- This approach is simple to develop but can become difficult to manage as the system grows.
- Scaling is challenging because even if only one component needs more resources, the entire application has to be scaled.
- A failure in one part of the system can cause the entire application to crash.

Microservices Architecture:
- In microservices architecture, the API is broken into multiple small, independent services, each responsible for specific functionality (e.g., authentication, payments, order management).
- These services communicate with each other via APIs, usually REST or gRPC.
- Each service can be developed, deployed, and scaled independently, improving flexibility and efficiency.
- Failures in one microservice do not affect the entire system, making it more reliable.
- However, managing microservices requires a proper communication mechanism and service discovery.

7. How does an API gateway work in a distributed API architecture?

An API gateway is an intermediary that manages and routes client requests to the appropriate microservice. It acts as a single entry point for all requests.
Key responsibilities of an API gateway include:
- Authentication & Authorization: Ensures only valid requests reach the backend.
- Rate Limiting: Prevents excessive requests from overloading the system.
- Load Balancing: Distributes requests efficiently among multiple instances.
- Caching: Improves performance by storing frequently requested data.
API gateways help simplify API management and improve security.

8. What are the common layers of an API system design?

An API system is usually divided into the following layers:
- Presentation Layer: Manages client interactions and request handling.
- Business Logic Layer: Contains the core logic of the application, such as processing data and applying rules.
- Data Layer: Handles data storage and retrieval from databases or external services.
This layered approach improves maintainability and scalability.

9. What is CORS and why is it important in API development?

CORS (Cross-Origin Resource Sharing) is a security feature in web browsers that controls whether a web page can make API requests to a different domain.
By default, browsers block cross-origin requests for security reasons. CORS allows API owners to specify which domains are permitted to access their resources by setting appropriate headers.

10. Explain how error handling should be managed in an API architecture.

Error handling in APIs should be structured and meaningful:
- Use proper HTTP status codes (e.g., 400 for bad requests, 401 for unauthorized, 500 for server errors).
- Provide clear error messages with details about what went wrong.
- Log errors for debugging purposes.
- Implement error handling middleware in frameworks like Express.js to catch and process errors efficiently.

◇ REST API Concepts

11. What are REST principles? List the constraints that define a RESTful system.

REST principles include:
- Statelessness
- Client-Server Model
- Cacheability
- Layered System
- Uniform Interface

12. What is the difference between PUT and PATCH in REST APIs?

- PUT: Updates the entire resource.
- PATCH: Updates only specific fields.

13. Why should HTTP methods (GET, POST, PUT, DELETE) be idempotent? Which ones are?

Idempotency means that making the same request multiple times should have the same effect as making it once. In other words, no matter how many times a request is repeated, the outcome remains unchanged.
In APIs, idempotency is important to prevent unintended side effects when requests are retried due to network failures or other issues. It ensures that duplicate requests do not modify data unexpectedly.

Idempotency of HTTP Methods:

- **GET** – Idempotent (Fetching data does not change it)
- **PUT** – Idempotent (Updating a resource with the same data does not change its state)

- **DELETE** – Idempotent (Deleting a resource multiple times has the same effect: the resource remains deleted)
- **POST** – **Not idempotent** (Each request creates a new resource, so multiple requests result in multiple entries)

14. How do you implement authentication and authorization in REST APIs?

Authentication (Verifying User Identity)

Authentication ensures that a user or system making a request is legitimate. One common way to implement authentication is by using JWT (JSON Web Token).

Authorization (Controlling User Access)

Authorization determines **what a user can do** after being authenticated. It is usually based on Role-Based Access Control (RBAC).

15. What are the common response codes used in REST APIs, and what do they indicate?

- 200: Success
- 201: Created
- 400: Bad Request
- 401: Unauthorized
- 404: Not Found
- 500: Server Error

Practical Questions

1. Write a simple Node.js server using the http module that responds with 'Hello, World!'.

```
22  const http = require('http');
23
24  const server = http.createServer((req, res) => {
25      res.end('Hello, World!');
26  });
27
28  server.listen(3000, () => {
29      console.log('Server running on port 3000');
30  });
31
32
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS COMMENTS

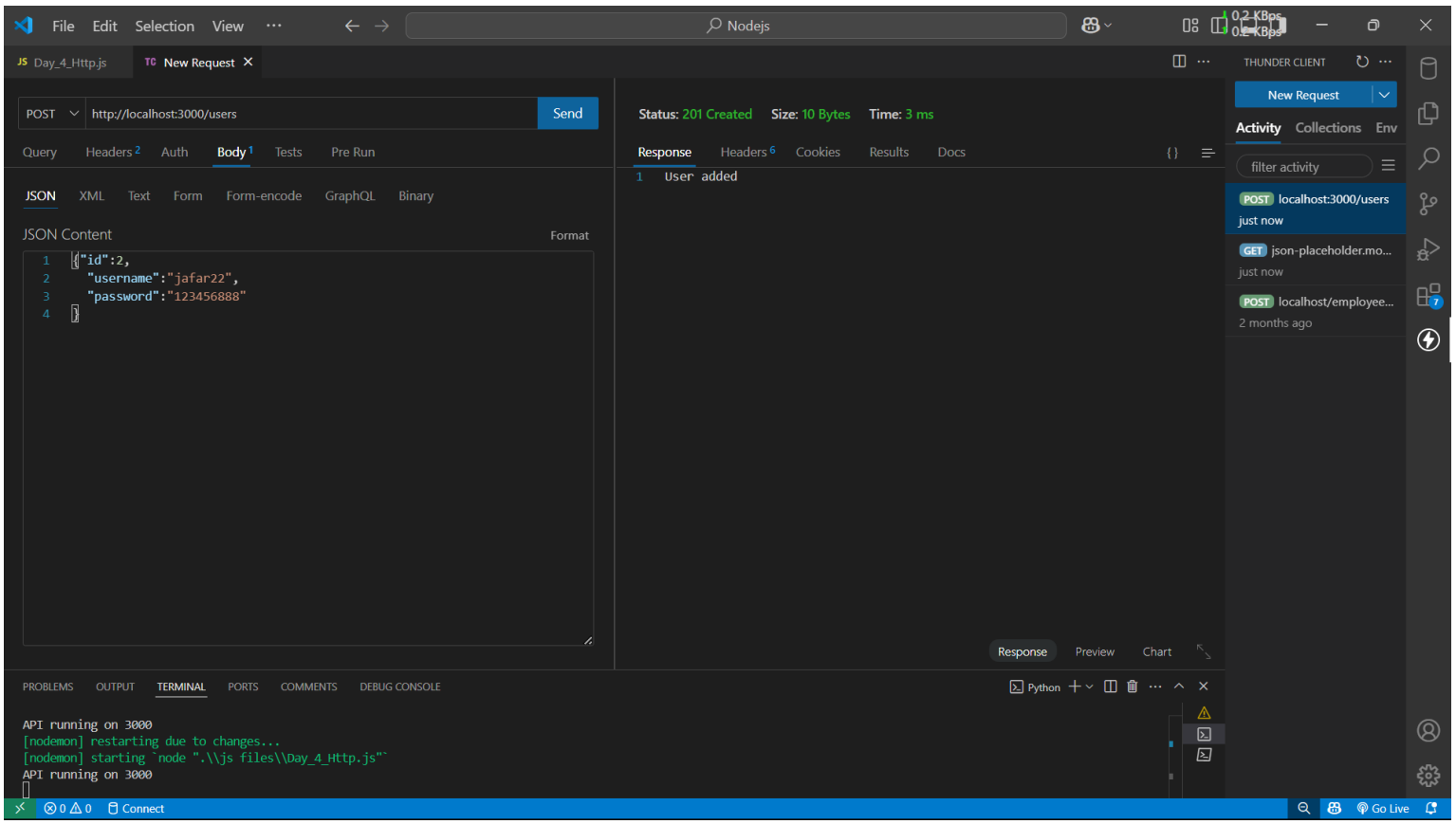
```
Server running on port 3000
[nodemon] restarting due to changes...
```

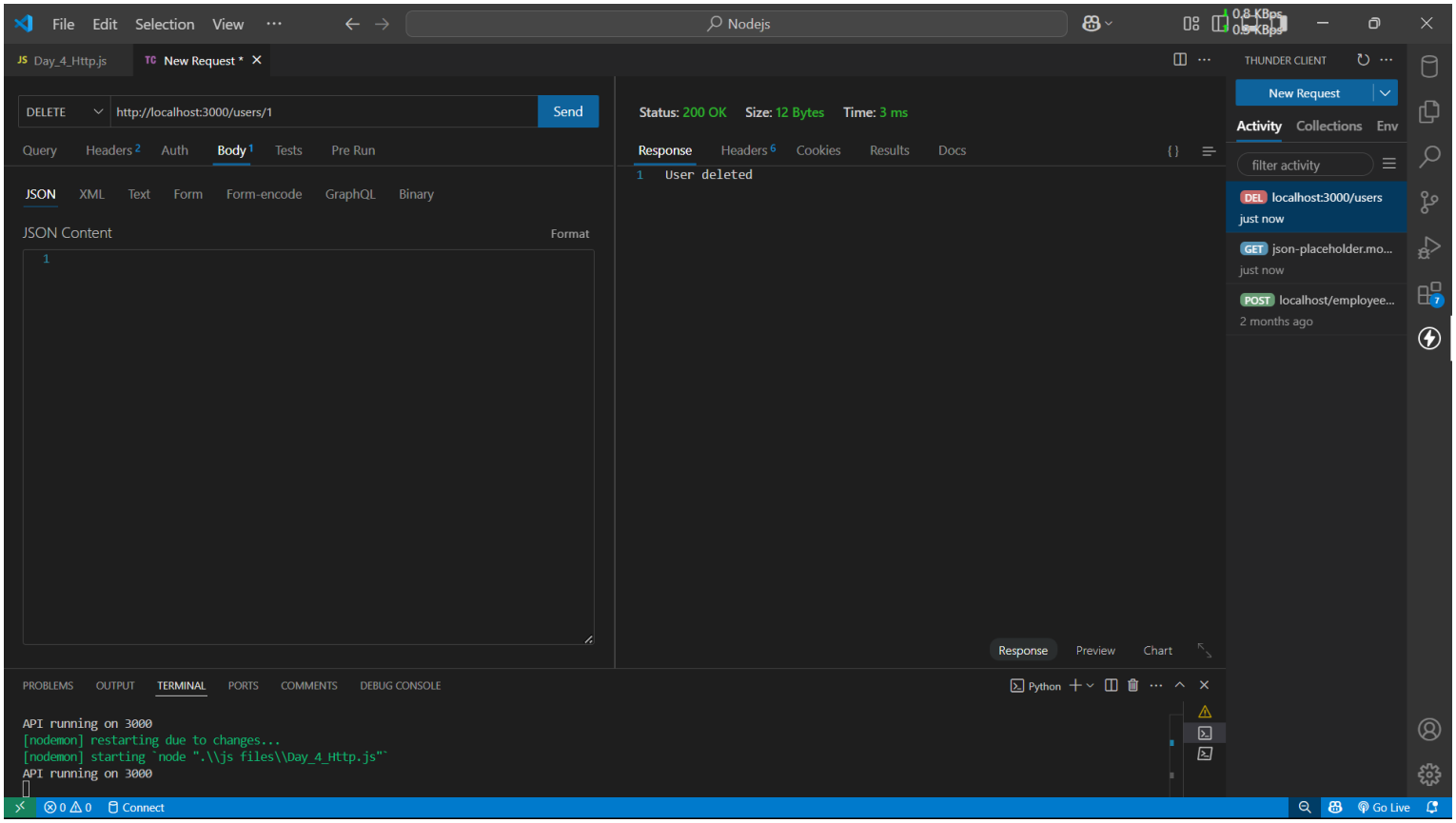
2. Create a RESTful API with endpoints to GET all users, POST a new user, and DELETE a user.

```
18  import express from "express";
19  const app = express();
20  app.use(express.json());
21  let users = [];
22  app.get("/users", (req, res) => res.json(users));
23  app.post("/users", (req, res) => {
24    users.push(req.body);
25    res.status(201).send("User added");
26  });
27  app.delete("/users/:id", (req, res) => {
28    users = users.filter((u) => u.id !== req.params.id);
29    res.send("User deleted");
30  });
31
32  app.listen(3000, () => console.log("API running on 3000"));
```

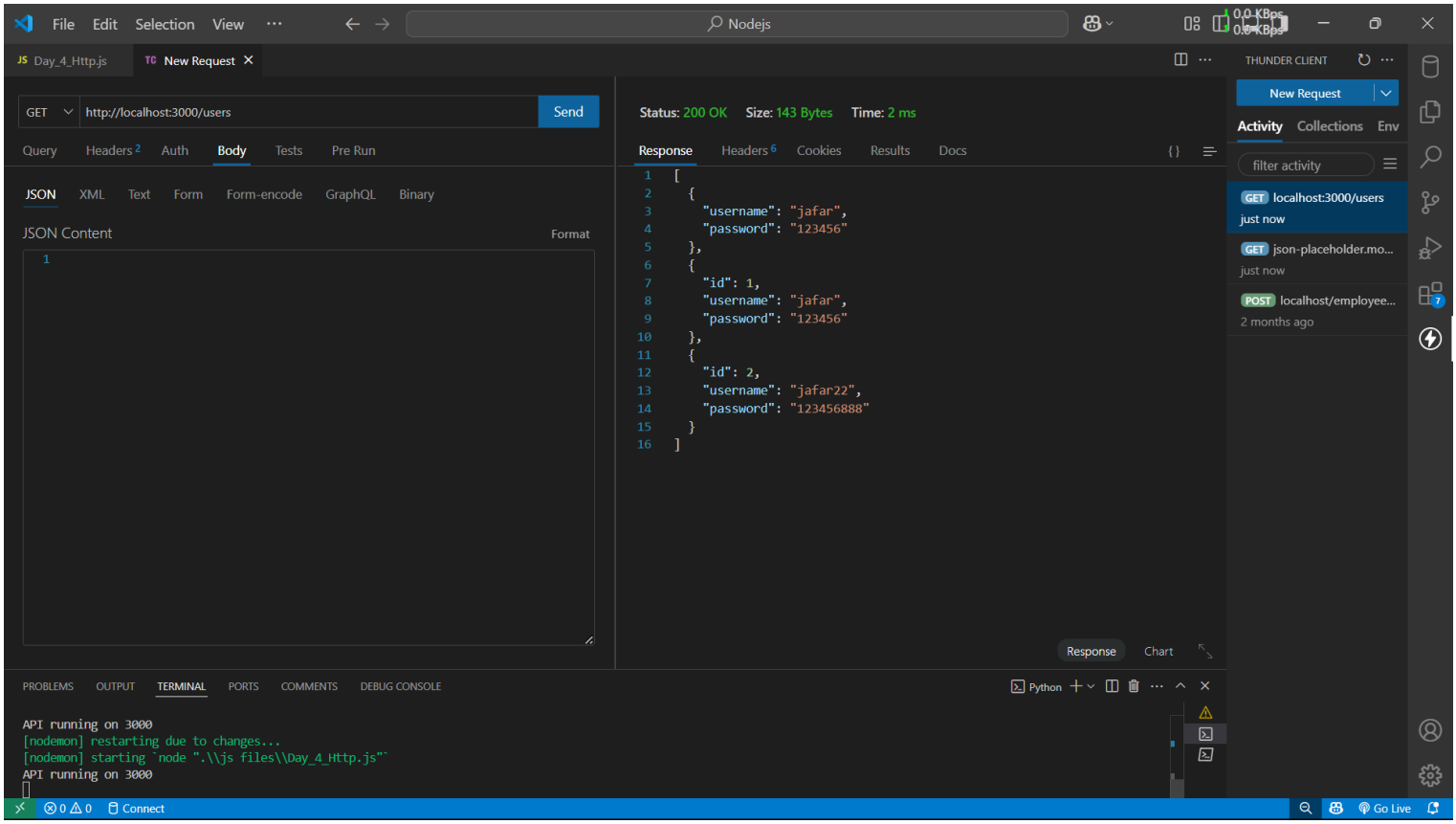
PROBLEMS OUTPUT TERMINAL PORTS COMMENTS DEBUG CONSOLE

```
API running on 3000
[nodemon] restarting due to changes...
```

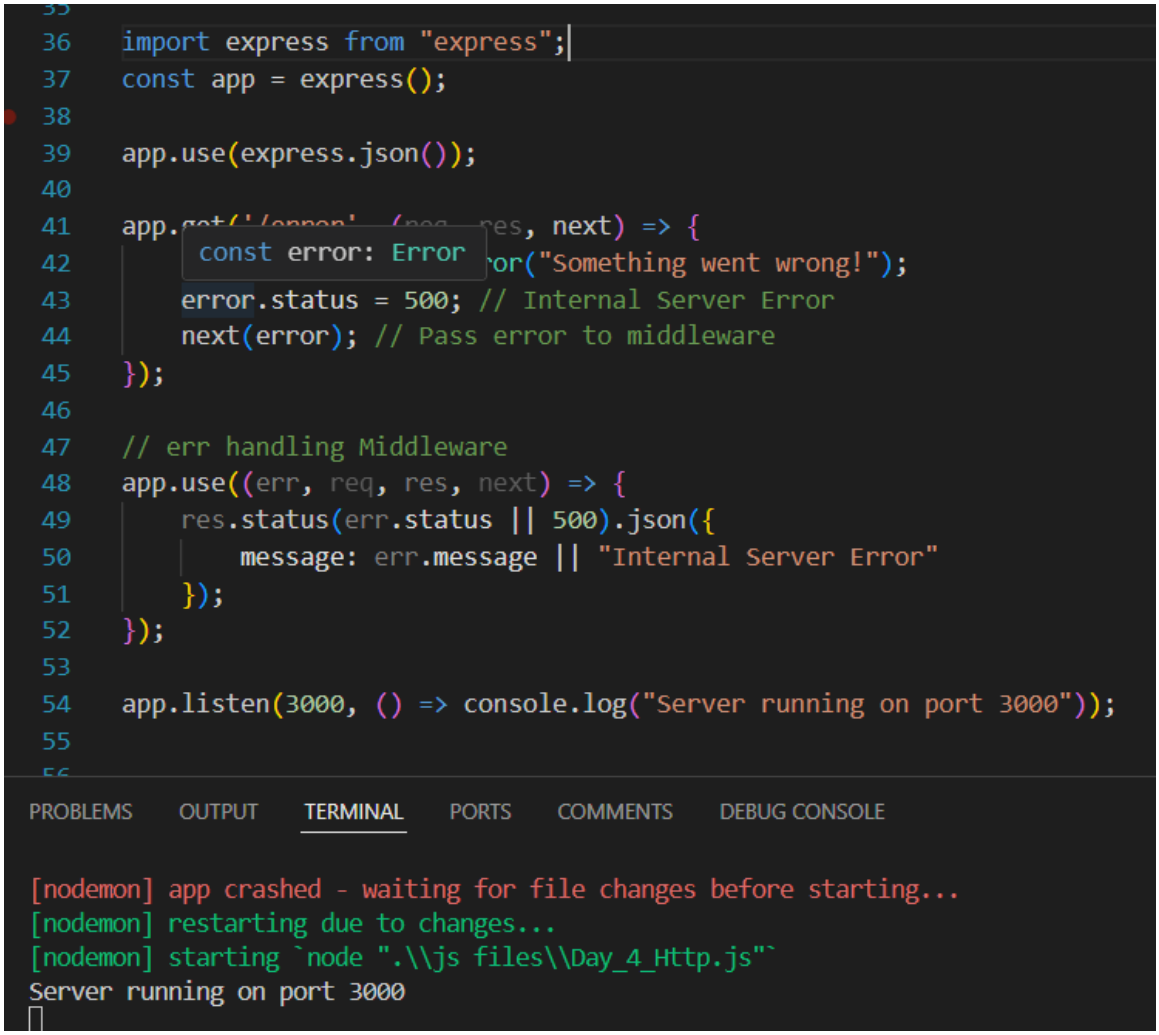

File Edit Selection View
Nodejs
Day_4_Http.js
New Request
POST
http://localhost:3000/users
Send
Query Headers 2 Auth Body 1 Tests Pre Run
JSON XML Text Form Form-encode GraphQL Binary
JSON Content
Format
{"id":2,
"username":"jafar22",
"password":"123456888"
}
Status: 201 Created Size: 10 Bytes Time: 3 ms
Response Headers 6 Cookies Results Docs
User added
Response Preview Chart
THUNDER CLIENT
New Request
Activity Collections Env
filter activity
POST localhost:3000/users
just now
GET json-placeholder.mo...
just now
POST localhost/employee...
2 months ago
PROBLEMS OUTPUT TERMINAL PORTS COMMENTS DEBUG CONSOLE
Python
API running on 3000
[nodemon] restarting due to changes...
[nodemon] starting `node ".\\js files\\Day_4_Http.js"`
API running on 3000
Connect
Go Live

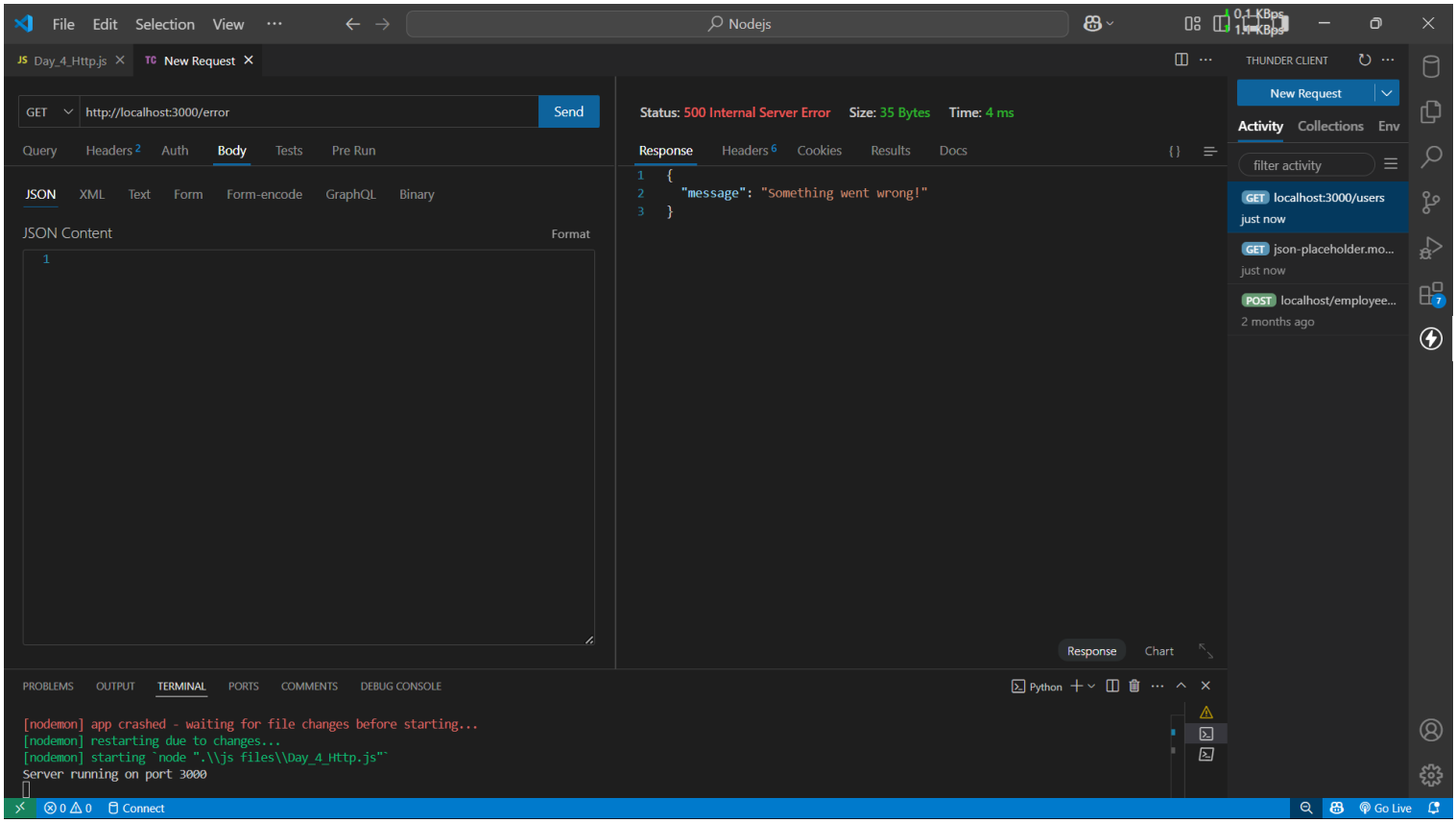

File Edit Selection View
Nodejs
Day_4_Http.js
New Request *
DELETE
http://localhost:3000/users/1
Send
Query Headers 2 Auth Body 1 Tests Pre Run
JSON XML Text Form Form-encode GraphQL Binary
JSON Content
Format
Status: 200 OK Size: 12 Bytes Time: 3 ms
Response Headers 6 Cookies Results Docs
User deleted
Response Preview Chart
THUNDER CLIENT
New Request
Activity Collections Env
filter activity
DEL localhost:3000/users
just now
GET json-placeholder.mo...
just now
POST localhost/employee...
2 months ago
PROBLEMS OUTPUT TERMINAL PORTS COMMENTS DEBUG CONSOLE
Python
API running on 3000
[nodemon] restarting due to changes...
[nodemon] starting `node ".\\js files\\Day_4_Http.js"`
API running on 3000
Connect
Go Live

Jafar Beldar



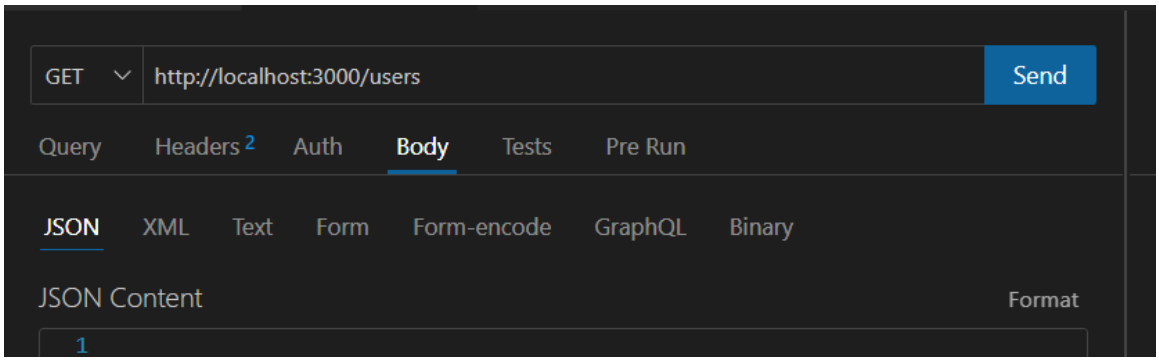
3. Simulate error handling by returning appropriate HTTP status codes and messages in your REST API.

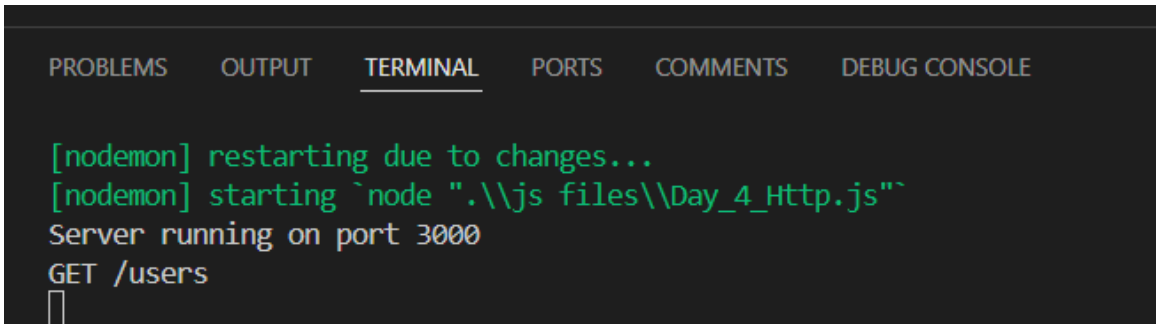




4. Implement a simple middleware to log request method and URL for each incoming request.

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





5. Use http.request() to make an outbound API call to a public REST API and log the response.

```
66  import http from 'http';
67  const options = { hostname: 'jsonplaceholder.typicode.com', path: '/posts/1', method: 'GET' };
68  const req = http.request(options, res => {
69      res.on('data', data => console.log(data.toString()));
70  });
71  req.end();
```

```
PROBLEMS  OUTPUT  TERMINAL  PORTS  COMMENTS  DEBUG CONSOLE
[nodemon] app crashed - waiting for file changes before starting...
[nodemon] restarting due to changes...
[nodemon] starting `node ".\\js files\\Day_4_Http.js"`
{
  "userId": 1,
  "id": 1,
  "title": "sunt aut facere repellat provident occaecati excepturi optio reprehenderit",
  "body": "quia et suscipit\nsuscipit recusandae consequuntur expedita et cum\nreprehenderit molestiae ut ut quas
ecto"
}
[nodemon] clean exit - waiting for changes before restart
```