**Assignments Number 2**

**COURSE OUTCOMES:Co1 and Co2**

Totally 30 marks ,each question 10 marks

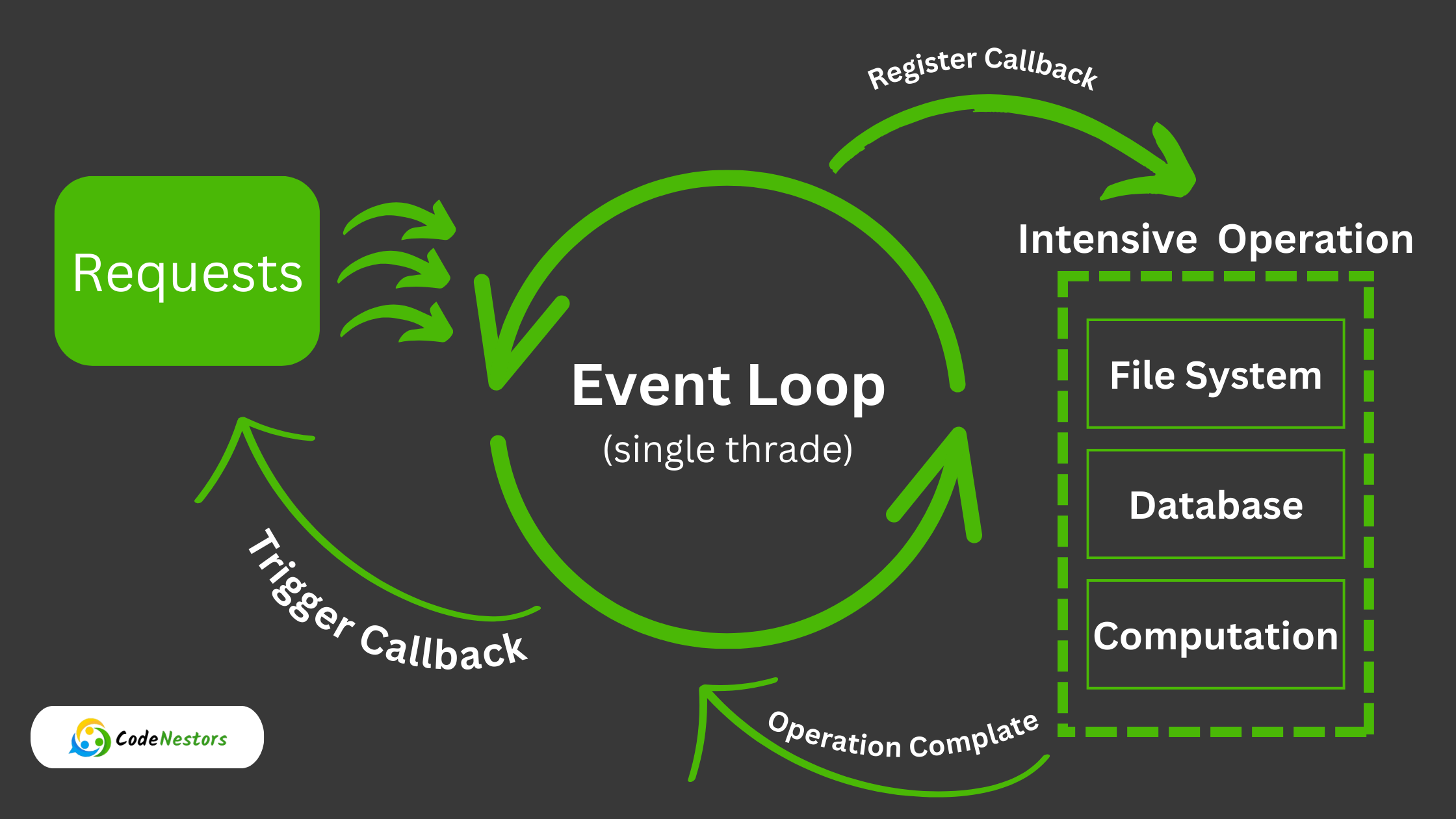
**Name: Ayyanraj C**

**Roll No: 21BCS013**

**Rubric for Question 1**: Design a Node.js Web Server with Asynchronous I/O and Callback Handling

Node.js is an runtime environment that allows running JavaScript on the server side. The core feature of Node.js includes an event-driven, non-blocking I/O model, which makes it very appropriate to build scalable network applications.

**Event-Driven Architecture**: Node.js uses an event loop as a design architecture to handle all asynchronous operations. The event loop lets Node.js achieve non-blocking I/O operations, such as reading files, without blocking the main execution thread.



**Asynchronous I/O**: In Node.js, all I/O operations are done in the background asynchronously, and the server does not wait for I/O operations-for instance, reading to a file or querying a database-to complete the operation before moving on to the next task. Instead, Node.js continues processing other tasks and utilizes callbacks or Promises when an asynchronous operation is ready with results.

**Callback Handling:** A callback is a function that is passed as an argument to other functions and is executed when an asynchronous operation is over. It is basically the essence of Node.js, handling every kind of asynchronous task in it.

**Example:**

(The code uses Node.js's built-in http and fs modules. http instead of express so need for npm initialization and express installation)

1. Create a folder for application
2. Create a js file for server code and html file.

Server.js

const http = require("http");

const fs = require("fs");

const server = http.createServer((req, res) => {

  fs.readFile("index.html", (err, data) => {

    if (err) {

      res.writeHead(500, { "Content-Type": "text/plain" });

      res.end("500 Internal Server Error");

    } else {

      res.writeHead(200, { "Content-Type": "text/html" });

      res.end(data);

    }

  });

});

server.listen(3000, () => {

  console.log("Server running at http://localhost:3000/");

});

Index.html

<!DOCTYPE html>

<html lang="en">

  <head>

    <meta charset="UTF-8" />

    <meta name="viewport" content="width=device-width, initial-scale=1.0" />

    <title>Node.js Server</title>

  </head>

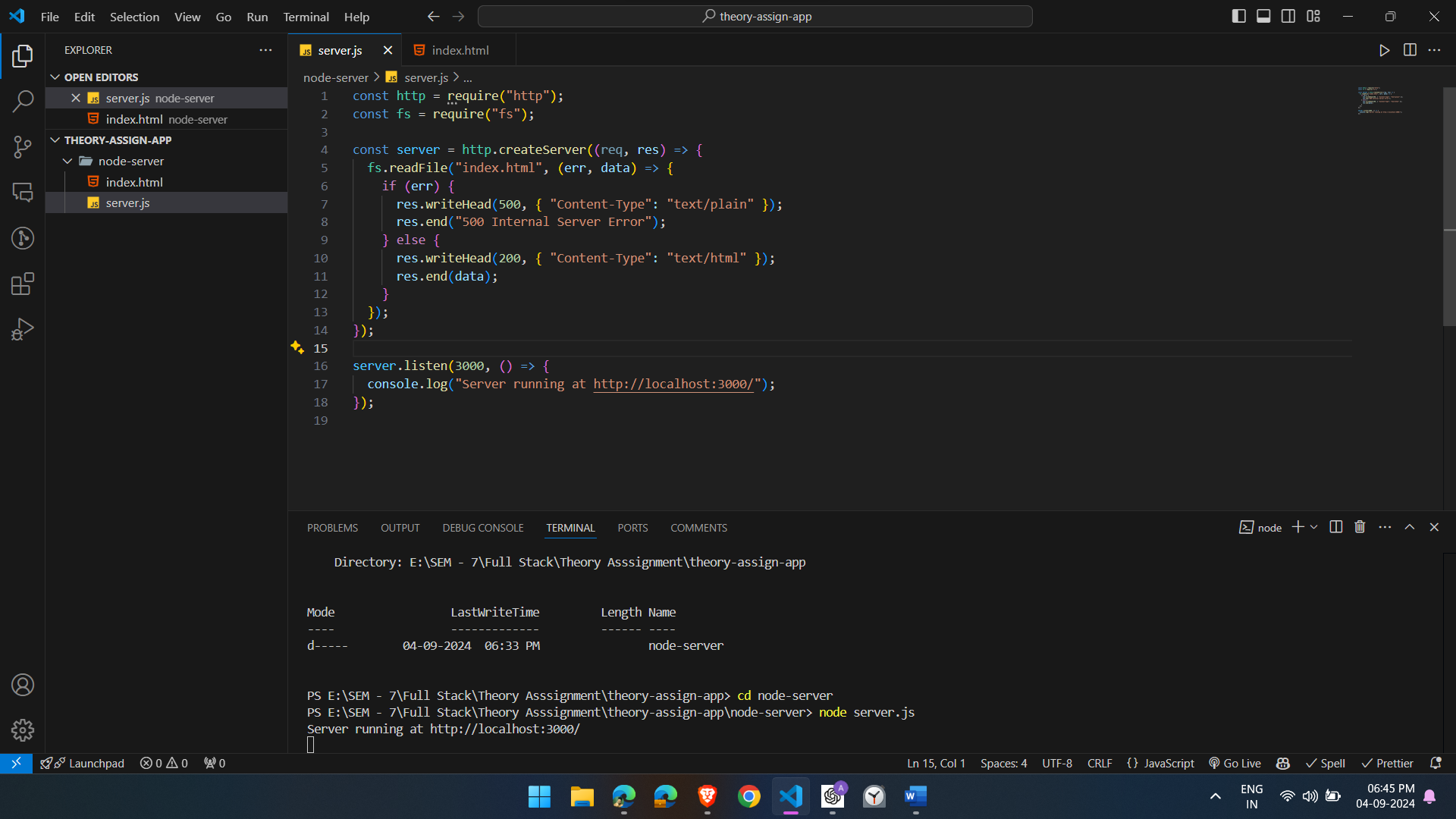
  <body>

    <h1>Welcome to My Node.js Server!</h1>

  </body>

</html>

1. Run server.js and open <http://localhost:3000/>



A computer screen with a white background

Description automatically generated

A screenshot of a computer

Description automatically generated

Changing file name to check the error

A screenshot of a computer

Description automatically generated

**Explanation**: The server is set up to listen on port 3000. When a request is made, the server asynchronously reads the index.html file. If an error occurs during the read operation, a 500 Internal Server Error response is sent. If the read operation is successful, the file content is sent back with a 200 OK status.

**Implication Asynchronous I/O:**

* **Scalability:** Since Node.js does not block the main thread on I/O, it can service many requests concurrently, making it super scalable.
* **Exception Handling:** Proper exception handling is key, more so with callbacks. In the example above, the “**fs.readFile”** callback contains exception handling to provide an appropriate response in case a file read error occurs, or otherwise.
* **Efficiency:** Asynchronous I/O makes the server more efficient to work with since it can handle any other requests at the same time as it has requested an I/O operation.
* **Solution to Possible Problems:** Good utilization of asynchronous operations lets the server process many requests without blocking its event loop. This makes the server more responsive, even at high loads.
* **Callback Hell:** This is a situation where callbacks are nested inside other callbacks, making the code unreadable and hard to maintain. Refactor your code with Promises or async/await that modernizes and simplifies asynchronous code handling in Node.js.

**Rubric for Question 2:** Managing Asynchronous Operations with Anonymous Callback Functions and JSON Data

Node.js is non-blocking and event-driven, which makes it suited well for handling operations asynchronously. **For example**, in Node.js, when the server has to read something from a file or access data over an HTTP request, it need not sit idle until it gets data back; instead, it sends the request and in that time can execute other pending requests. This would be made possible with the help of callback functions, functions passed as an argument to other functions and executed once the asynchronous operation has finished.

**Anonymous callback functions** are unnamed functions, mostly defined directly in the argument list of the asynchronous function. This is a horribly common pattern in Node.js, as it keeps things short and sweet, especially for simple operations.

**JSON**, short for JavaScript Object Notation, is a light-weight data-interchange format that is often used in asynchronous operations to transfer data from a server to a client. It is very easy to read and write and is, by default, available within JavaScript through the methods JSON.parse and JSON.stringify, allowing one to go between JSON strings and JavaScript objects.

**Example:**

Server.js

const http = require("http");

const fs = require("fs");

const server = http.createServer((req, res) => {

  // Asynchronous file read operation with an anonymous callback function

  fs.readFile("data.json", "utf8", (err, data) => {

    if (err) {

      res.writeHead(500, { "Content-Type": "application/json" });

      res.end(JSON.stringify({ error: "Internal Server Error" }));

    } else {

      // Parse the JSON data

      const jsonData = JSON.parse(data);

      res.writeHead(200, { "Content-Type": "application/json" });

      res.end(JSON.stringify(jsonData));

    }

  });

});

server.listen(3000, () => {

  console.log("Server running at http://localhost:3000/");

});

Data.json

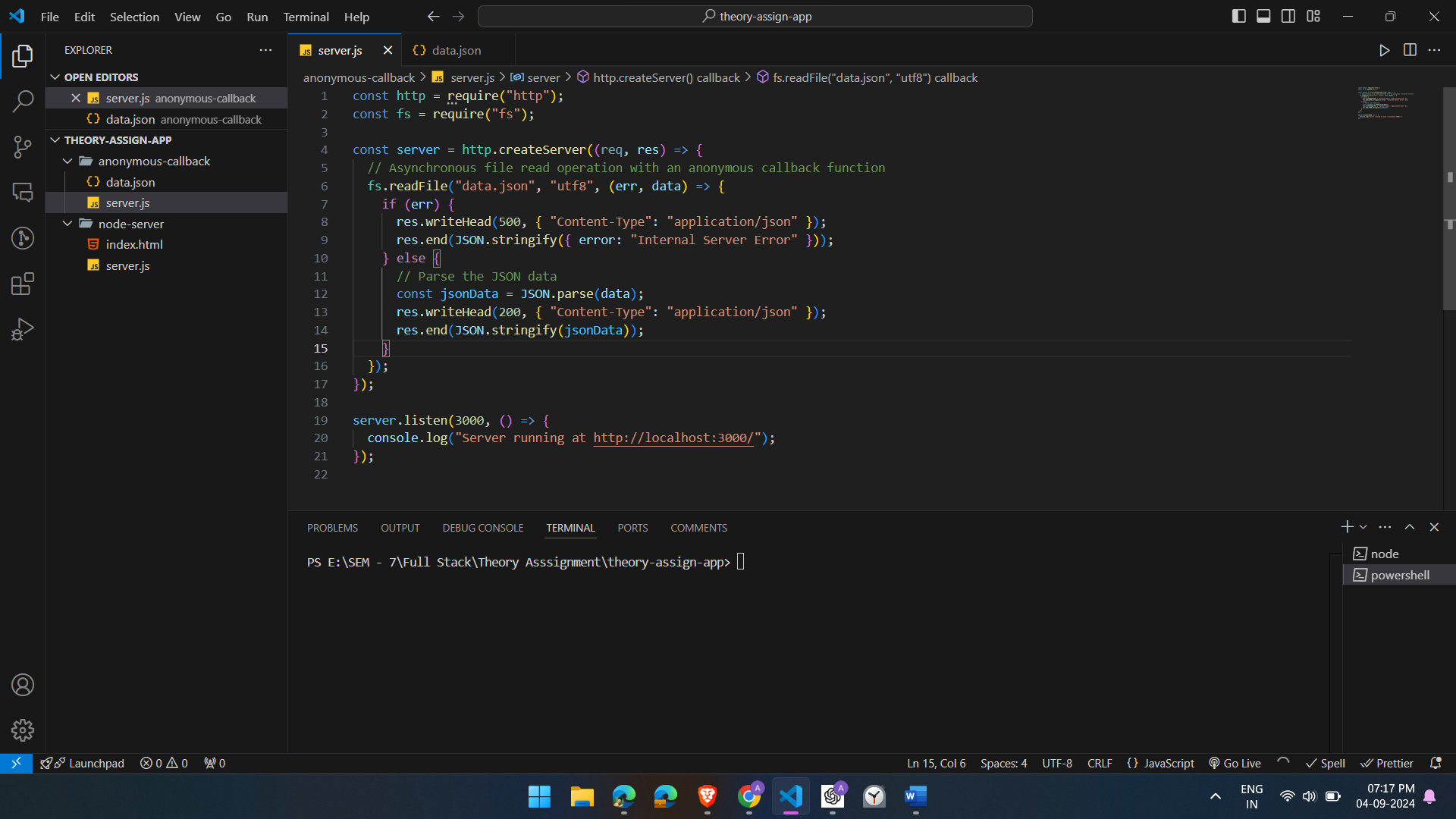
{

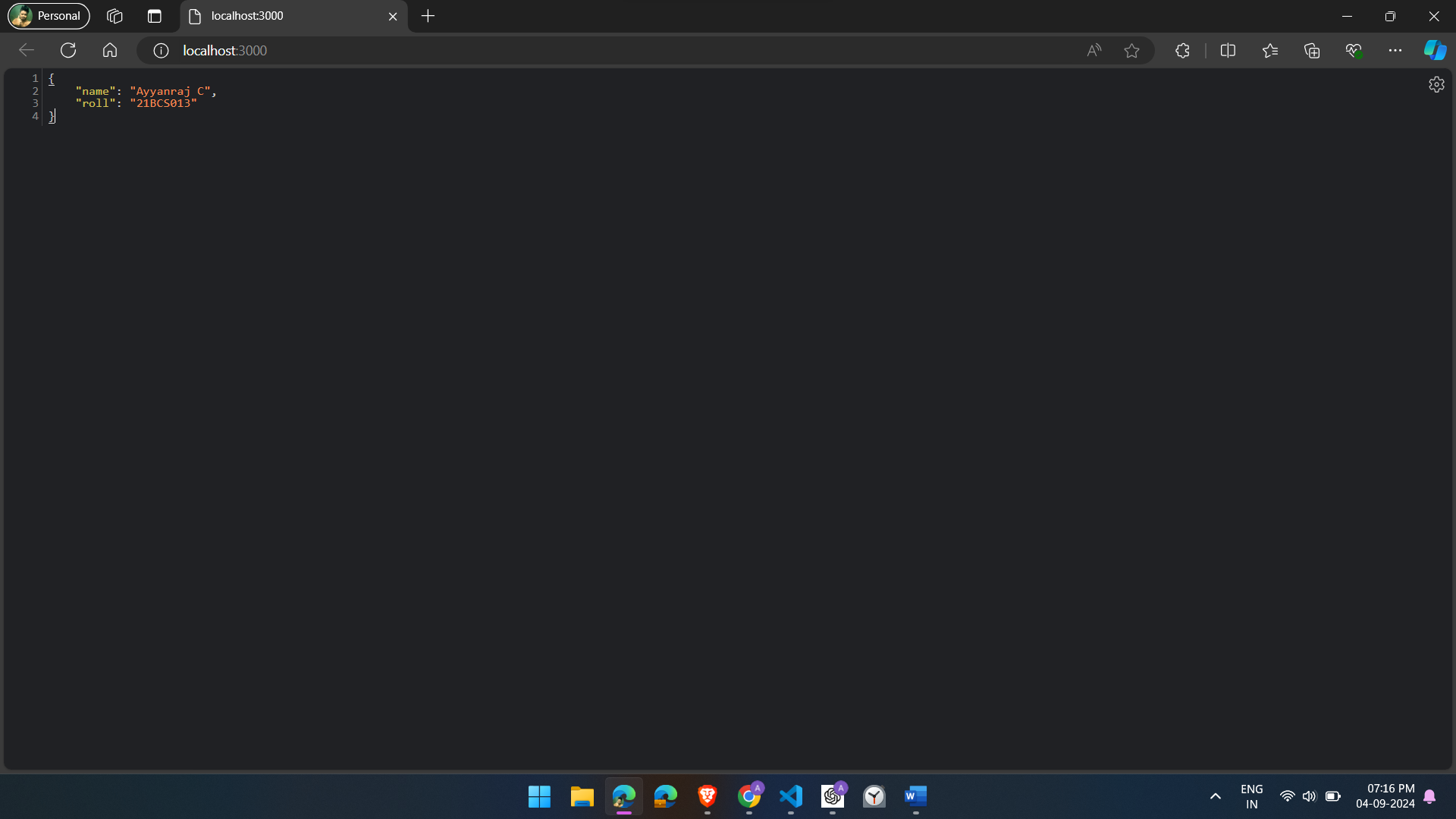
  "name": "Ayyanraj C",

  "roll": "21BCS013"

}

**Output:**

****



**Explanation**

**Asynchronous File Reading:**

* The fs.readFile function is utilized for the asynchronous reading of the data.json file.
* An unnamed callback function is passed to fs.readFile which is invoked when the file read operation has finished.

**Error Handling:**

* If an error occurs on reading the file, for example, if the file doesn't exist, then it's handled by the callback function, which resonds to the client with a 500 Internal Server Error and a JSON-formatted error message:.

**JSON Parsing and Response:**

* Upon successful reading of the file, the JSON string contained is then converted into a JavaScript object through JSON.parse.
* Afterwards, the processed object is returned to the client in JSON string format, along with a status of 200 OK. Server Setup: The server runs on port 3000, and if started, should be able to handle incoming HTTP requests by providing responses based on the JSON file.

**Rubric for Question 3:** Debugging Node.js Application with Asynchronous JSON Processing Issues

**Understanding the Problem:**

In the context of Node.js, challenges associated with asynchronous JSON processing may originate from multiple factors, including erroneous JSON parsing, management of race conditions, or inadequate error handling. This discussion will focus on diagnosing a situation in which a Node.js application asynchronously reads a JSON file but faces complications arising from improper management of asynchronous processes or JSON data.

**Scenario:**

let's think about a Node.js application that reads data from a JSON file and sends this data as a response to an HTTP request. Here, this application is problematic in either correctly processing or returning the data. The complications within this have to be identified and resolved.

**Example:**

const http = require("http");

const fs = require("fs");

const server = http.createServer((req, res) => {

  fs.readFile("data.json", "utf8", (err, data) => {

    if (err) {

      res.writeHead(500, { "Content-Type": "application/json" });

      res.end(JSON.stringify({ error: "Internal Server Error" }));

    } else {

      // Parsing JSON data

      const jsonData = JSON.parse(data);

      res.writeHead(200, { "Content-Type": "application/json" });

      res.end(JSON.stringify(jsonData));

    }

  });

});

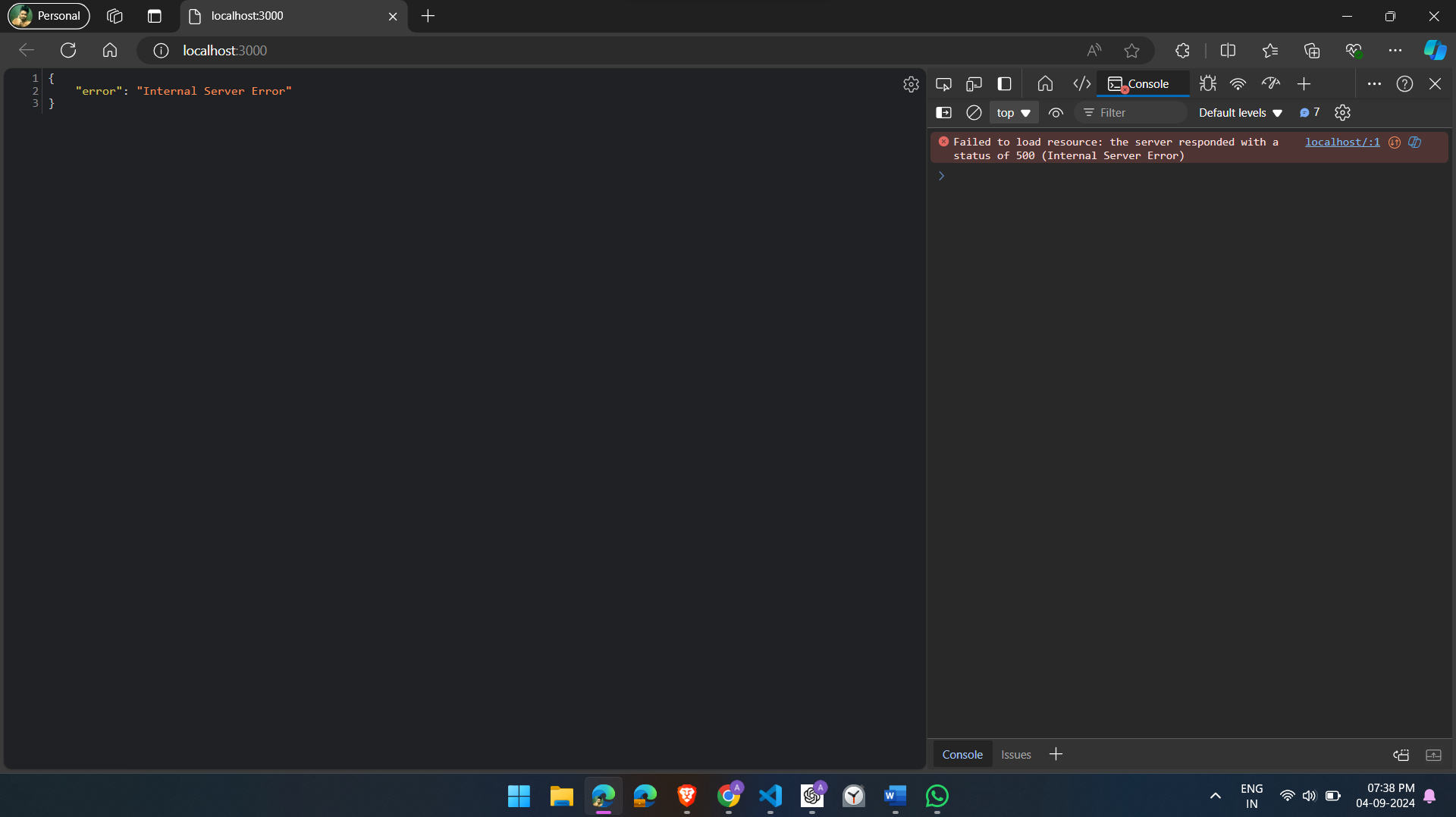
server.listen(3000, () => {

  console.log("Server running at http://localhost:3000/");

});

Top of Form

**Output:**

Bottom of Form

**Identifying the Problem**

1. **Error Handling**: The server should handle errors gracefully, but if the JSON file is missing or inaccessible, it should log specific errors to help diagnose the issue.
2. **JSON Parsing**: when there is JSON data that is malformed, or when the file contents are not valid JSON, JSON.parse(data) throws an error. That needs to be caught and handled.

**Debugging Steps**

1. **Add Logging:**

const http = require("http");

const fs = require("fs");

const server = http.createServer((req, res) => {

  fs.readFile("data.json", "utf8", (err, data) => {

    if (err) {

      console.error("File read error:", err);

      res.writeHead(500, { "Content-Type": "application/json" });

      res.end(JSON.stringify({ error: "Internal Server Error" }));

      return;

    }

    try {

      // Parsing JSON data

      const jsonData = JSON.parse(data);

      res.writeHead(200, { "Content-Type": "application/json" });

      res.end(JSON.stringify(jsonData));

    } catch (parseError) {

      console.error("JSON parse error:", parseError);

      res.writeHead(400, { "Content-Type": "application/json" });

      res.end(JSON.stringify({ error: "Bad Request: Invalid JSON" }));

    }

  });

});

server.listen(3000, () => {

  console.log("Server running at http://localhost:3000/");

});

Include logging in the callback function for observing the returned data and possible errors.

**2. Check File Path and Permissions:**

* Double check if data.json is in the right folder, and has the proper permissions.
* If the file path is incorrect, correct it.

**3. Verify JSON Data:**

* Make sure that data.json file contains proper JSON.
* For example, data.json should be something like this:

**4. Test and Verify:**

* Now start the server and send requests to see whether the bug has gone away. Check in the console logs if there are still any errors.

**5. Consider Race Conditions:**

* When multiple asynchronous operations are interdependent, it is crucial to ensure their proper sequencing or to utilize promises to manage these dependencies effectively.

Here I missed data.json file after that

