Date:

Experiment - 1

**Aim:**

Demonstration of const, let, string templates, callbacks, arrow functions, class, class-properties, methods using Java Script.

**Description:**

**const and let**

* const is used for declaring constants that cannot be reassigned.
* let allows variable reassignment and is block-scoped.

**String Templates**

* Use backticks (``) instead of quotes.
* Embed expressions using ${}.
* Supports multi-line strings without escape characters.

**Callbacks**

* Functions passed as arguments to other functions.
* Useful for handling asynchronous operations.
* Helps in implementing event-driven programming.

**Arrow Functions**

* Uses => syntax for concise function definitions.
* Lexically binds this, preventing scope-related issues.
* Commonly used in callbacks and functional programming.

**Class**

* Defines object blueprints using class syntax.
* Supports constructors for initializing properties.
* Improves code organization and reusability.

**Class Properties**

* Properties can be declared inside a class directly.
* No need to define properties inside the constructor.
* Enhances readability and simplifies object structure.

**Methods**

* Functions inside a class that define behavior.
* Can be accessed through class instances.
* Supports encapsulation and modular programming.

**Program-1:**

//using const

const pi = 3.14159;

//using let

let radius = 5;

//string template

let area = pi \* radius \* radius;

console.log(`The area of a circle with radius ${radius} is ${area}`);

//callback function

function calculate(callback){

    let result = callback(5, 10);

    console.log('The result of calculation is: ' + result);

}

//callback: arrow function

calculate((a, b) => a + b);

//Arrow functions

const greet = (name) => {

    return 'Hello, ' + name;

}

console.log(greet('GVSR'));

//Class

class Circle{

    //Class Property

    constructor(radius){

        this.radius = radius;

    }

    //Method to calculate area

    getArea(){

        return pi \* this.radius \* this.radius;

    }

    //Method to calculate circumference

    getCircumference(){

        return 2 \* pi \* this.radius;

    }

}

//creating an object

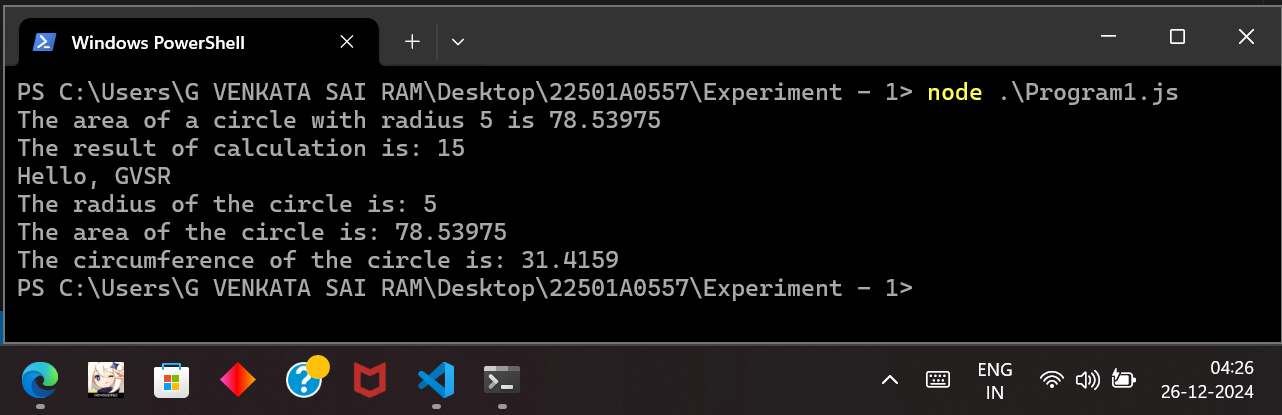
let myCircle = new Circle(5);

console.log('The radius of the circle is: ' + myCircle.radius);

console.log('The area of the circle is: ' + myCircle.getArea());

console.log('The circumference of the circle is: ' + myCircle.getCircumference());

**Output:**



Executing the Program1 Javascript file

**Program-2:**

let str = "Hello, My name is Ram"; // Defining a string variable

// Finding and printing the length of the string

console.log("str length: " + str.length);

// Getting the character at index 4 (0-based index)

console.log("char at inx 4: " + str.charAt(4));

// Extracting a portion of the string from index 7 to 15 (excluding 15)

console.log("Slice from index 7 to 15: " + str.slice(7, 15));

// Getting a substring from index 7 with length 8 (but substring's second argument is exclusive, so it's 7 to 8)

console.log("Substring from index 7 with length 8: " + str.substring(7, 8));

// Converting the entire string to uppercase

console.log("String in Upper case: " + str.toUpperCase());

// Converting the entire string to lowercase

console.log("String in Lower case: " + str.toLowerCase());

let newText = "Let's explore more!"; // Another string to demonstrate concatenation

// Concatenating the two strings

console.log("Concatenated string: " + str.concat(newText));

// Splitting the string into an array of words using space (" ") as a separator

let words = str.split(" ");

console.log("Words in the string: " + words);

// Replacing "Hello" with "Hi" in the string

let newStr = str.replace("Hello", "Hi");

console.log("String after replace: " + newStr);

// Finding the index of the word "Ram" in the string

let index = str.indexOf("Ram");

console.log("Index of Ram: " + index);

// Checking if the string contains the word "JavaScript"

let containsJavaScript = str.includes("JavaScript");

console.log("Have JavaScript: " + containsJavaScript);

// Getting the last occurrence of the character 'a' in the string

let li = str.lastIndexOf("a");

console.log("Last index of a: " + li);

// Searching for the word "Welcome" in the string (returns -1 if not found)

let regSearch = str.search("Welcome");

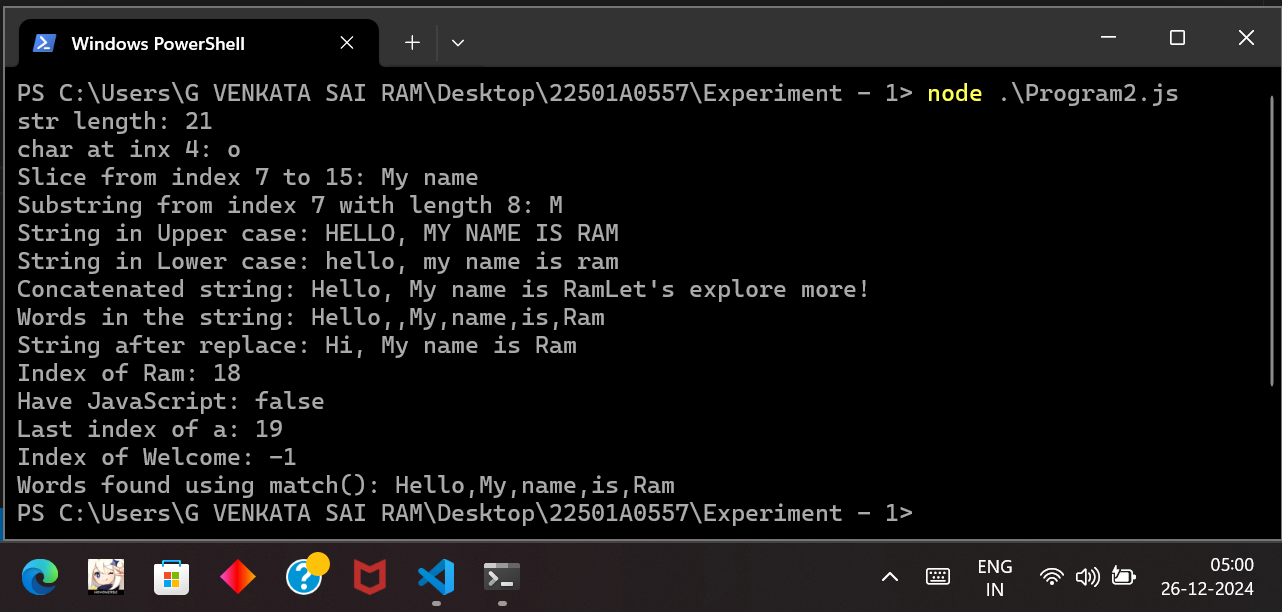
console.log("Index of Welcome: " + regSearch);

// Using regex to match all words (sequences of letters)

let matchResult = str.match(/[A-za-z]+/g);

console.log("Words found using match(): " + matchResult);

**Output:**



Executing Program 2 Javascript File

Date:

Experiment - 2

**Aim:**

Use of global object, Variables, Standard input and standard output in Node JS to build application.

**Description:**

**Global Object**

* The global object provides global variables and functions accessible throughout the application.
* Common properties include console, setTimeout(), setInterval(), and process.

**Variables**

* Variables can be defined using var, let, or const to store and manage data.
* const is preferred for constants, let for block-scoped variables, and var for function-scoped variables.
* Global variables in Node.js should be avoided unless necessary to prevent unintended side effects.

**Standard Input (stdin) and Standard Output (stdout)**

* process.stdin is used to read input from the user.
* process.stdout is used to print output to the console.
* readline module is commonly used for handling input and output efficiently.

These features enable the creation of interactive command-line applications, data processing tools, and automation scripts in Node.js.

**Program-1:**

// Importing required modules

const readline = require('readline'); // 'readline' module for user input

// Creating an interface for standard input and output

const r1 = readline.createInterface({

    input: process.stdin,

    output: process.stdout

});

// Displaying a welcome message

console.log("Welcome to PVPSIT");

// Asking user for their name

r1.question("What is your name? ", function(name) {

    // Asking user for their age

    r1.question("What is your age? ", function(age) {

        age = parseInt(age); // Converting age input to an integer

        const currYr = new Date().getFullYear(); // Getting the current year

        const YeartoTurn100 = currYr + (100 - age); // Calculating the year they turn 100

        // Displaying the calculated result

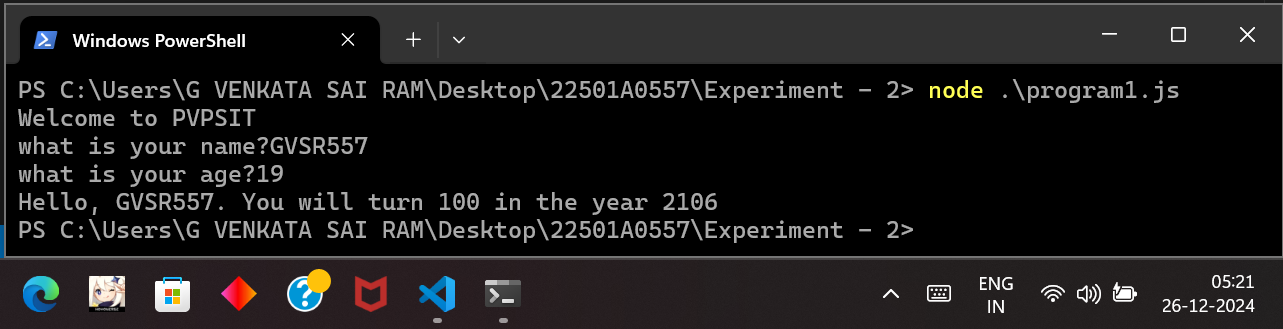
        console.log('Hello, ' + name + '. You will turn 100 in the year ' + YeartoTurn100);

        r1.close(); // Closing the input stream

    });

});

**Output:**



Executing Program 1 JavaScript file

**Program-2:**

const name = "Ram"; // Defining a constant variable 'name' with value "Ram"

var trade = "CSE"; // Declaring a global variable 'trade' with value "CSE"

function print() {

    console.log(name); // Printing the value of 'name' (Ram)

    var trade; // Redeclaring 'trade' inside the function

    if (trade == undefined) { // Checking if 'trade' is undefined

        trade = "AIML"; // Assigning "AIML" to 'trade' if the condition is true

        console.log(trade);

    }

}

print(); // Calling the function 'print'

// Listening for user input from the terminal

process.stdin.on('data', data => {

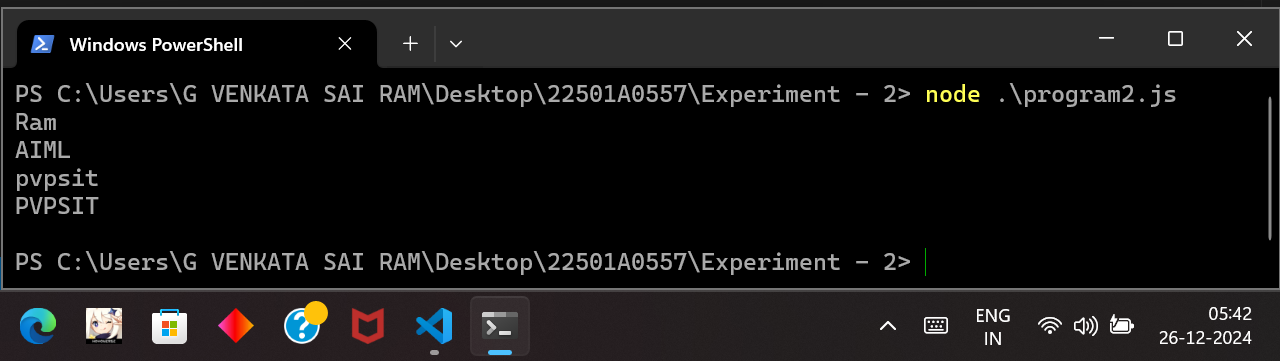
    data = data.toString().toUpperCase(); // Converting user input to uppercase

    process.stdout.write(data + '\n'); // Printing the modified input to the terminal

    process.exit(); // Exiting the process after one input

});

**Output:**



Executing Program 2 Javascript file

Date:

Experiment - 3

**Aim:**

Write a program to demonstrate the publishing and installation of npm package into npm registry

**Description:**

**Installing Node.js**

* Download Node.js from the official website ([nodejs.org](https://nodejs.org/)) and install it.
* Choose the LTS (Long-Term Support) version for stability.

**Censoring Words in Node.js**

* Implement middleware to scan requests for restricted words.
* Replace or block unwanted words before responding to the client.
* Useful for chat applications and content moderation.

**Verifying npm Installation**

npm -v

* If installed correctly, this command will return the npm version.

**What is npm?**

* Node Package Manager (npm) is a tool for managing JavaScript packages.
* It allows installation, updating, and removal of dependencies for a Node.js project.

**Initializing a Node.js Project (npm init)**

npm init

* This command creates a package.json file that stores project metadata and dependencies.
* Use npm init -y to generate the file with default values.

**package.json**

* It contains information about the project, including dependencies, scripts, and metadata.
* Essential fields include name, version, dependencies, and scripts.

**Node Package Registry**

* A repository of open-source Node.js packages.
* Developers can publish and share reusable modules.

**Node Package Manager Commands**

* **npm pack**: Creates a .tgz file for distribution.
* **npm publish**: Publishes the package to the npm registry.
* **npm unpublish**: Removes a published package (only within 72 hours).
* **npm login**: Authenticates a user for publishing packages.

**Program-1:** Working with modules (local)

**censorify.js**

// Predefined list of words to be censored

var censoredWords = ["sad", "bad", "mad"];

// Custom censored words added dynamically

var customCensoredWords = [];

// Function to censor words in a given input string

function censor(inStr) {

// Replace words from the predefined censored list

for (idx in censoredWords) {

inStr = inStr.replace(censoredWords[idx], "\*\*\*\*\*");

}

// Replace words from the user-added censored list

for (idx in customCensoredWords) {

inStr = inStr.replace(customCensoredWords[idx], "\*\*\*\*\*");

}

return inStr; // Return the modified string

}

// Function to add a new word to the custom censored list

function addCensoredWord(word) {

customCensoredWords.push(word); // Store the new word in the list

}

// Function to get the complete list of censored words

function getCensoredWords() {

return censoredWords.concat(customCensoredWords); // Merge predefined & custom words

}

// Exporting the functions for use in other files

exports.censor = censor;

exports.addCensoredWord = addCensoredWord;

exports.getCensoredWords = getCensoredWords;

**index.js**

// Import the 'censorify' module

var censor = require("censorify");

// Retrieve the list of currently censored words and print them

console.log(censor.getCensoredWords());

// Censor certain words in the given text and print the modified version

console.log(censor.censor("Some very sad, bad and mad text."));

// Add a new word ("gloomy") to the list of censored words

censor.addCensoredWord("gloomy");

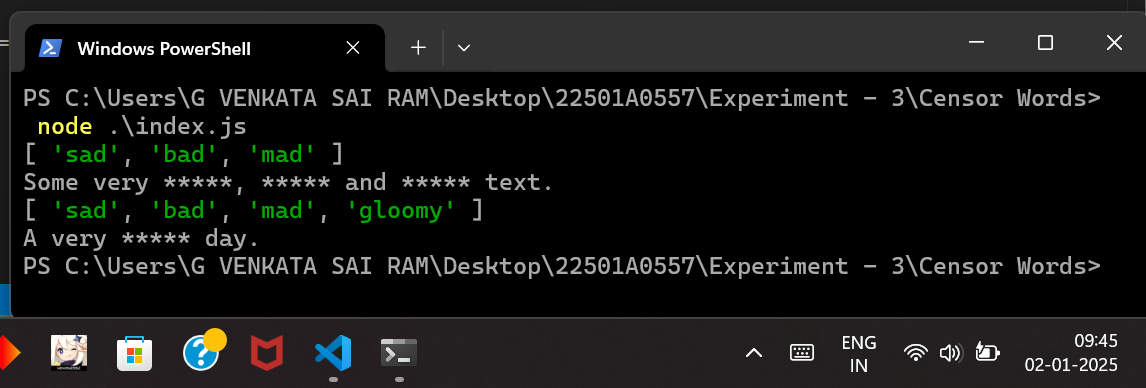
// Retrieve and print the updated list of censored words

console.log(censor.getCensoredWords());

// Censor the newly added word in the given text and print the modified version

console.log(censor.censor("A very gloomy day."));

**Output:**



Executing the Censor Words Program (index.js)

**Program-2:** Working with NPM Registry

**gv557\_exp3**

**package.json**

{

"name": "gv557\_exp3",

"version": "1.0.0",

"main": "rotateArr.js",

"scripts": {

"test": "echo \"Error: no test specified\" && exit 1"

},

"author": "gvsr557",

"license": "ISC",

"description": ""

}

**rotateArr.js**

// This function rotates the array elements in anti-clockwise direction

function leftRotateArr(arr) {

var temp = arr[0];

for (var i = 0; i < arr.length - 1; i++)

arr[i] = arr[i + 1];

arr[arr.length - 1] = temp;

return arr;

}

// This function rotates the array elements in clockwise direction

function rightRotateArr(arr) {

var temp = arr[arr.length - 1];

for (var i = arr.length - 1; i > 0; i--)

arr[i] = arr[i - 1];

arr[0] = temp;

return arr;

}

// Export the functions as an object

module.exports = {

rotateArr: {

leftRotateArr: leftRotateArr,

rightRotateArr: rightRotateArr

}

};

**gv\_rotating\_array**

**package.json**

{

"name": "gv\_rotating\_array",

"version": "1.0.4",

"main": "index.js",

"scripts": {

"test": "echo \"Error: no test specified\" && exit 1"

},

"author": "gv557",

"license": "ISC",

"description": "Adding the right folders",

"dependencies": {

"gv\_rotating\_array": "file:",

"gv557\_exp3": "^1.0.0",

"prompt-sync": "^4.2.0"

},

"keywords": [

"rotate array",

"rotating array",

"array rotation",

"array manipulation",

"clockwise rotation",

"anti-clockwise rotation",

"javascript array",

"node.js array tools",

"array utilities",

"rotate array left",

"rotate array right",

"data structures",

"array operations",

"array rotation algorithm",

"array processing",

"coding utilities",

"npm array tool",

"interactive array tool",

"array logic",

"array transformation",

"array shift",

"array rotate npm",

"gvsr array module",

"npm rotating array",

"node rotating array",

"programming tools",

"menu-driven array rotation"

],

"devDependencies": {}

}

**index.js**

//This module is to test my npm registry

const rtr = require('gv557\_exp3').rotateArr;

//This module is to take input easily

const prompt = require('prompt-sync')();

//Taking input from the user

let n = parseInt(prompt("Enter the size of the array: "));

console.log("Enter the elements of the array:");

let arr = new Array(n);

for (let i = 0; i < n; i++)

arr[i] = parseInt(prompt(`Element ${i+1}: `));

//Menu driven program

while(true){

console.log("1. Anti-Clockwise Rotate array\n2. Clockwise Rotate\n3. Exit\n");

let choice = prompt("Choose an operation: ");

let rotatedArr = [];

switch(choice){

case '1':

rotatedArr = rtr.leftRotateArr(arr);

console.log(rotatedArr);

break;

case '2':

rotatedArr = rtr.rightRotateArr(arr);

console.log(rotatedArr);

break;

case '3':

console.log("Exiting...");

process.exit(0);

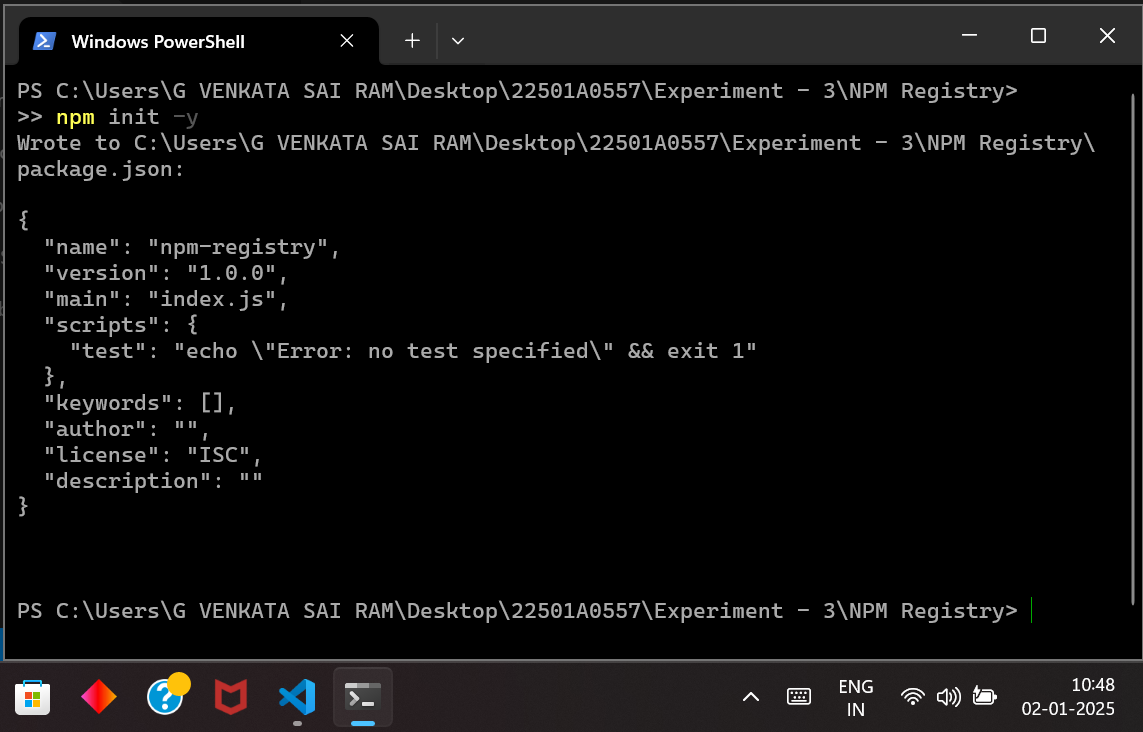
default:

console.log("Invalid choice. Please try again.");

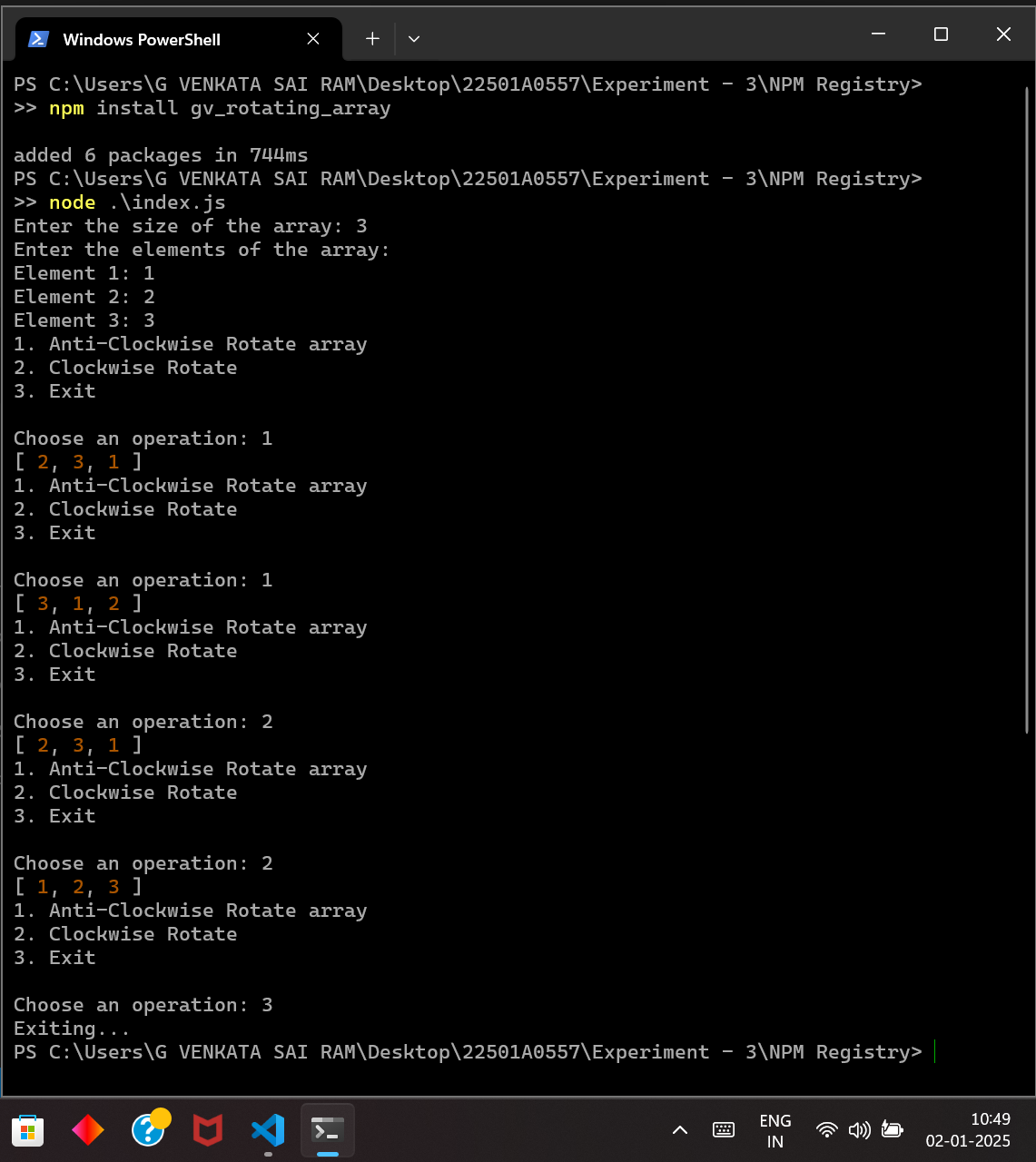
}

}

**Output:**



Initializing the npm environment



Executing the module downloaded from registry with index.js

Date:

Experiment - 4

**Aim:**

Develop node JS Application and Implement HTTP Services in Node JS (Request and Response)

**Description:**

This experiment focuses on developing a **Node.js application** and implementing **HTTP services** using the built-in http module. It demonstrates how to create an HTTP server that can handle **requests** and send appropriate **responses**. The experiment covers key concepts such as **handling different HTTP methods (GET, POST, etc.), processing client requests, sending JSON responses, and managing errors**. Additionally, it includes an example of an HTTP client that makes a request to the server, receives a response, and processes the data. This experiment helps in understanding the basics of **server-client communication** in Node.js and how to build lightweight web services.

**Steps to Implement HTTP Server in Node.js**

* Create a server using the http module.
* Handle incoming requests and send responses.
* Serve different responses based on request type or URL.

**Client-Server using Express.js**

* Express.js is a lightweight framework for handling HTTP requests.
* Provides middleware, routing, and request handling capabilities.
* Simplifies the creation of RESTful APIs.

**Program-1:**

**server.js**

// Import required modules

const http = require('http');

// Create the server

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

    // Set the response header to specify content type as JSON

    res.setHeader('Content-Type', 'application/json');

    // Handle different request methods and URLs

    if (req.method === 'GET' && req.url === '/') {

        // Simple response to GET requests on the root URL

        const responseMessage = { message: 'Welcome to the Node.js server!' };

        res.statusCode = 200;

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

    } else if (req.method === 'GET' && req.url === '/data') {

        // Respond to GET requests on the "/data" URL

        const data = { name: 'Node.js', type: 'Server' };

        res.statusCode = 200;

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

    } else {

        // Handle other requests (404 Not Found)

        res.statusCode = 404;

        res.end(JSON.stringify({ error: 'Not Found' }));

    }

});

// Define the server port

const port = 3000;

// Start the server and listen on the specified port

server.listen(port, () => {

    console.log(`Server is running at http://localhost:${port}`);

});

**client.js**

// Import required modules

const http = require('http');

// Define the server options

const options = {

    hostname: 'localhost', // The server's hostname

    port: 3000,            // The server's port

    path: '/',             // Path to request

    method: 'GET',         // HTTP method (GET in this case)

};

// Send an HTTP request to the server

const req = http.request(options, (res) => {

    let data = '';

    // Set the encoding of the response

    res.setEncoding('utf8');

    // Collect the data chunks as they are received

    res.on('data', (chunk) => {

        data += chunk;

    });

    // When the response has ended, print the response

    res.on('end', () => {

        console.log('Response from server:', data);

    });

});

// Handle errors in the request

req.on('error', (err) => {

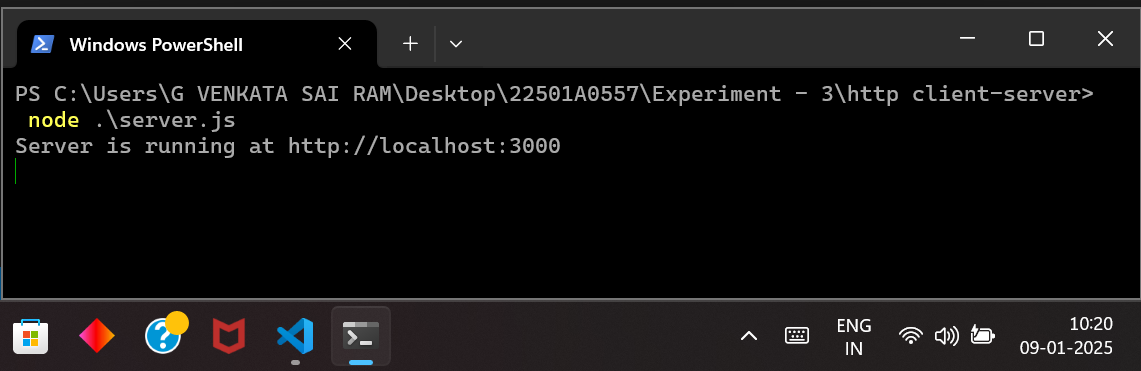
    console.error('Error during request:', err.message);

});

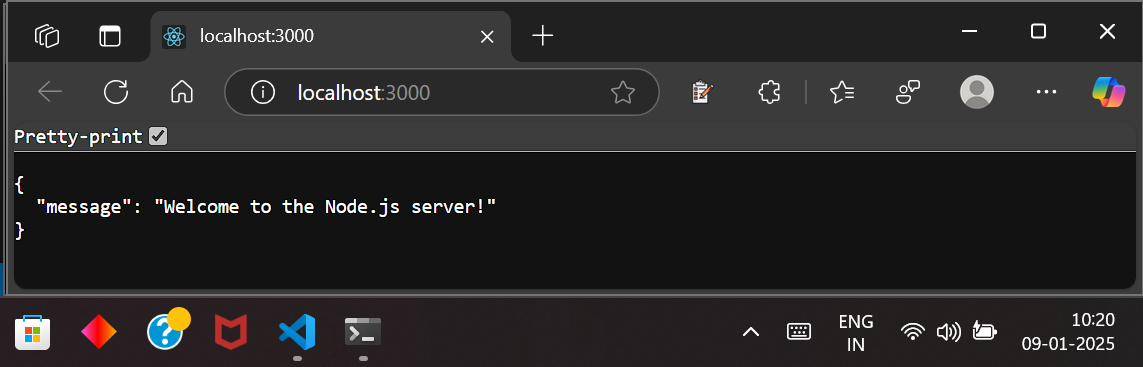
// End the request (send it to the server)

req.end();

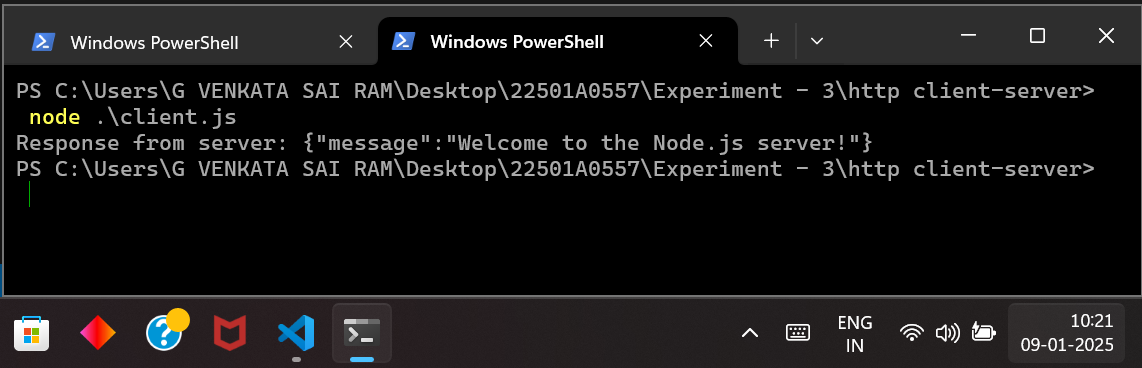
**Output:**



Running the server at port 3000



Verifying the output at port 3000



Executing the client.js file and receiving response from server

**Program-2:** Node.js express client-server

**server.js**

// Importing the Express framework

const express = require('express');

const app = express(); // Creating an instance of an Express application

const port = 3000; // Defining the port number for the server

// Handling GET request at the root URL ('/')

app.get('/', (req, res) => {

res.send("used GET method to access the server"); // Sending response for GET request

});

// Handling PUT request at the root URL ('/')

app.put('/', (req, res) => {

res.send("used PUT method to access the server"); // Sending response for PUT request

});

// Handling POST request at the root URL ('/')

app.post('/', (req, res) => {

res.send("used POST method to access the server"); // Sending response for POST request

});

// Starting the server and listening on the defined port

app.listen(port, () => {

console.log("Listening to the server " + port); // Logging the server start message

});

**client.js**

// Importing the built-in HTTP module to make HTTP requests

const http = require('http');

// Function to send a GET request

function getRequest() {

const options = {

hostname: 'localhost', // Server hostname (running locally)

port: 3000, // Server port

path: '/', // API endpoint

method: 'GET', // HTTP method

};

const req = http.request(options, (res) => {

let data = ''; // Variable to store response data

res.setEncoding('utf8'); // Setting response encoding to UTF-8

res.on('data', (chunk) => {

data += chunk; // Appending received data chunks

});

res.on('end', () => {

console.log('GET Response from server: used GET method to access the server'); // Logging response

});

});

req.on('error', (err) => {

console.error('Error during GET request:', err.message); // Handling errors

});

req.end(); // Ending the request

}

// Function to send a POST request

function postRequest() {

const options = {

hostname: 'localhost', // Server hostname

port: 3000, // Server port

path: '/', // API endpoint

method: 'POST', // HTTP method

};

const req = http.request(options, (res) => {

res.on('data', (chunk) => {}); // Handling response data (if any)

res.on('end', () => {

console.log('POST Response from server: used POST method to access the server'); // Logging response

});

});

req.on('error', (err) => {

console.error('Error during POST request:', err.message); // Handling errors

});

req.end(); // Ending the request

}

// Function to send a PUT request

function putRequest() {

const options = {

hostname: 'localhost', // Server hostname

port: 3000, // Server port

path: '/', // API endpoint

method: 'PUT', // HTTP method

};

const req = http.request(options, (res) => {

res.on('data', (chunk) => {}); // Handling response data (if any)

res.on('end', () => {

console.log('PUT Response from server: used PUT method to access the server'); // Logging response

});

});

req.on('error', (err) => {

console.error('Error during PUT request:', err.message); // Handling errors

});

req.end(); // Ending the request

}

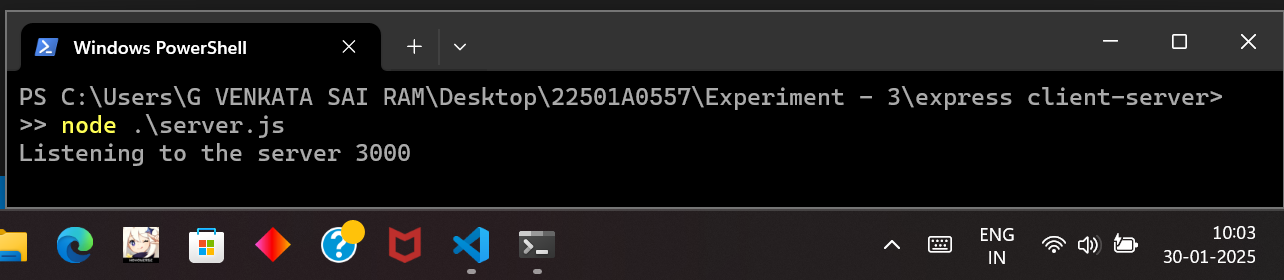
// Calling the functions to make HTTP requests

getRequest();

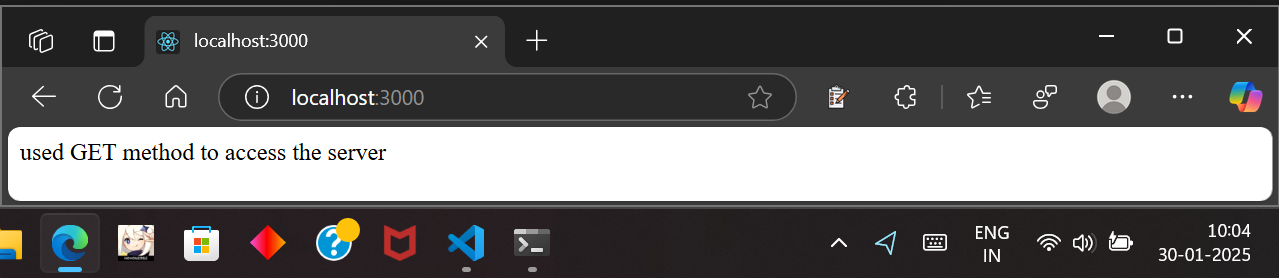
postRequest();

putRequest();

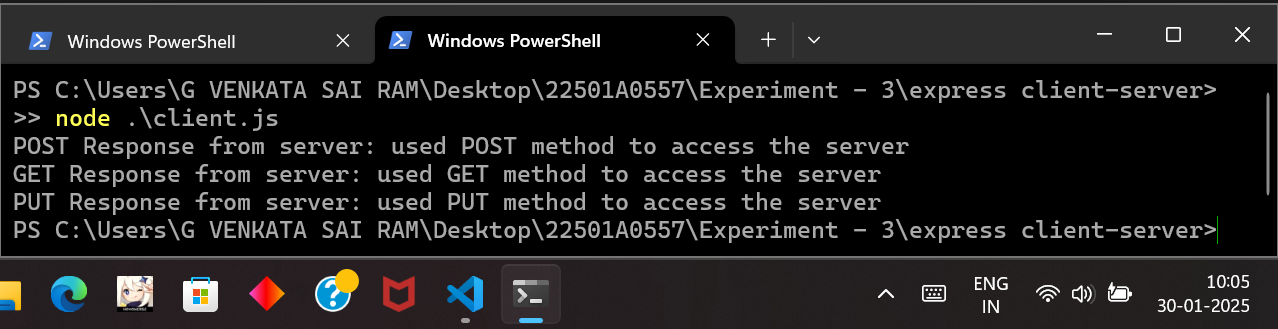
**Output:**



Running the server at port 3000



Verifying the server at port 3000



Accessing the Server with Various Methods (POST, GET, PUT)

Date:

Experiment - 5

**Aim:**

Write a program to demonstrate Routing in Express

**Description:**

This experiment demonstrates **Routing in Express.js**, which is a fundamental concept for handling different client requests based on the URL path. Express provides an efficient way to define routes for various endpoints and respond with the appropriate content. The program sets up an Express server that listens on a specified port and defines multiple routes such as **root (/), login (/login), home (/home), contact (/contact), and about (/about)**. Each route sends an HTML response corresponding to the request. This experiment helps in understanding how Express handles **incoming HTTP requests, URL path matching, and response generation**, making it a crucial aspect of building web applications and APIs.

**Express.js Routing**

* Defines how an application responds to HTTP requests.
* Routes handle different request methods (GET, POST, etc.) at specific URLs.
* Middleware can be applied globally or per route to process requests before sending a response.

**Program:** Express routing

// Importing the Express framework

const express = require('express');

// Creating an Express application

const app = express();

// Defining the port number on which the server will listen

const port = 5757;

// Defining a route for the root URL ('/')

app.get('/', (req, res) => {

res.send('<h1>Welcome to the Express server!</h1>'); // Sending an HTML response

});

// Route for the login page

app.get('/login', (req, res) => {

res.send('<h2>Welcome to the login page</h2>'); // Sending an HTML response

});

// Route for the home page

app.get('/home', (req, res) => {

res.send('<h2>Welcome to the home page</h2>'); // Sending an HTML response

});

// Route for the contact page

app.get('/contact', (req, res) => {

res.send('<h2>Welcome to the contact page</h2>'); // Sending an HTML response

});

// Route for the about page (showing an ID)

app.get('/about', (req, res) => {

res.send('<h2>22501A0557</h2>'); // Sending an HTML response

});

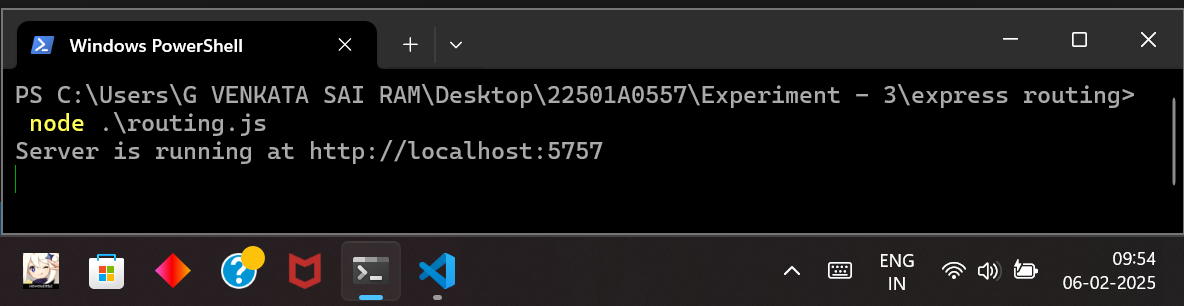
// Starting the server and listening on the specified port

app.listen(port, () => {

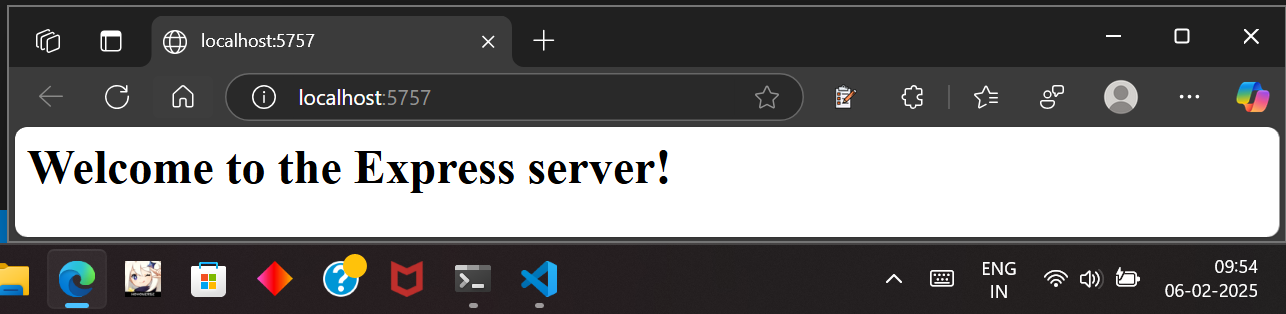
console.log(`Server is running at http://localhost:${port}`); // Logging the server URL

});

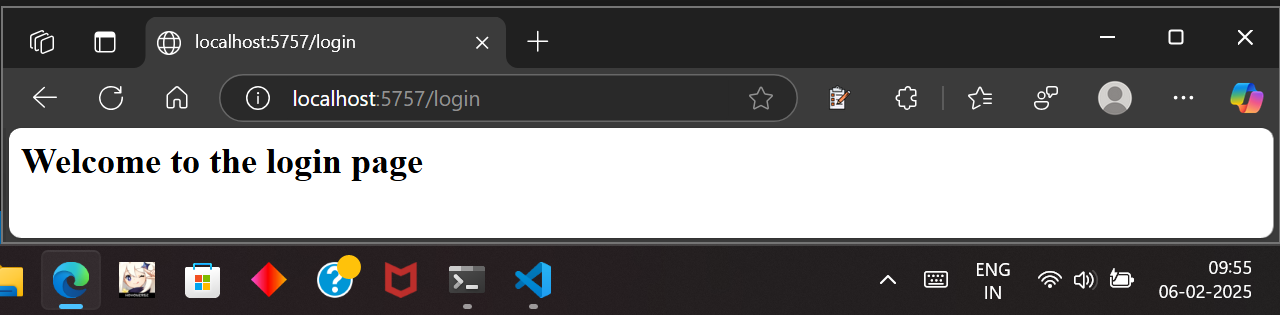
**Output:**



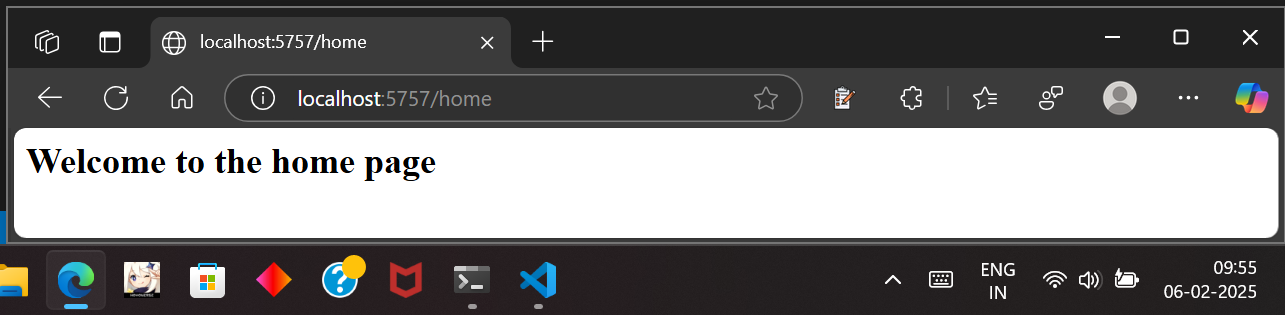
Running the server at port 5757



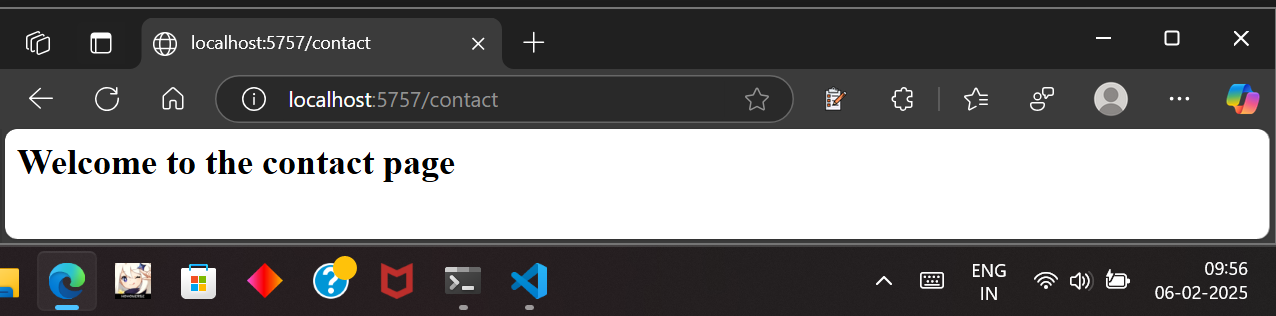
Accessing the Get Method ‘/’



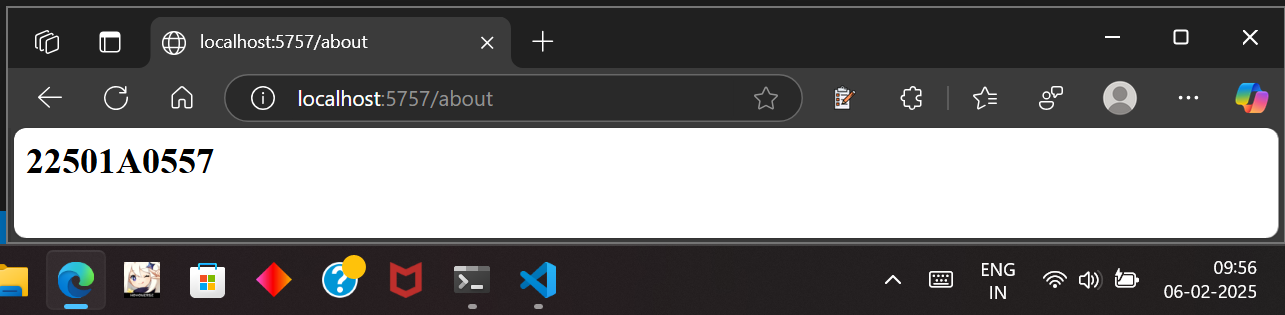
Accessing the Get Method ‘/login’



Accessing the Get Method ‘/home’



Accessing the Get Method ‘/contact’



Accessing the Get Method ‘/about’

Date:

Experiment - 6

**Aim:**

Reading and writing to MongoDb database using APIs

**Description:**

**1. Setup MongoDB and Install Dependencies**

* Install and start **MongoDB**.
* Install required **Node.js** packages:
* npm install express mongoose body-parser cors

**2. Connect to MongoDB**

* Use **Mongoose** to connect to the database (parkingdb).
* Define a **schema** for storing parking spot details.

**3. Implement API for Reading and Writing Data**

* **POST API**: Adds a new parking spot based on query parameters.
* **GET API**: Retrieves all parking spots from the database.

**4. Express API Implementation**

* Create an **Express server** with endpoints for CRUD operations.
* Use middleware (body-parser, cors) to handle JSON requests.

**5. Testing the API**

* Use **Postman, curl , or browser** to test API endpoints.
* Verify **data storage and retrieval** from MongoDB.

**Program:**

// Import required modules

const express = require("express"); // Web framework for Node.js

const mongoose = require("mongoose"); // ODM for MongoDB

const app = express(); // Initialize Express app

const port = 3000; // Define the port number

// Connect to MongoDB database (parkingdb)

mongoose.connect("mongodb://localhost:27017/parkingdb", {

useNewUrlParser: true, // Use new URL parser

useUnifiedTopology: true, // Use modern connection handling

});

// Define ParkingSpot Schema (structure of a parking spot document)

const ParkingSpotSchema = new mongoose.Schema({

spotNumber: Number, // Unique identifier for the parking spot

level: Number, // Parking level (e.g., 1, 2, 3)

section: String, // Section name (e.g., A, B, C)

isOccupied: Boolean, // Status if the spot is occupied

isReserved: Boolean, // Status if the spot is reserved

occupiedBy: String, // Vehicle or user occupying the spot

timestamp: { type: Date, default: Date.now }, // Auto-generated timestamp

});

// Create ParkingSpot Model

const ParkingSpot = mongoose.model("ParkingSpot", ParkingSpotSchema);

// POST API: Add a new parking spot using query parameters

app.post("/addParkingSpot", async (req, res) => {

const { spotNumber, level, section, isOccupied, isReserved, occupiedBy } = req.query;

// Validate required parameters

if (!spotNumber || !level || !section) {

return res.status(400).json({ error: "Missing required parameters" });

}

// Create a new parking spot object

const parkingSpot = new ParkingSpot({

spotNumber: Number(spotNumber), // Convert to number

level: Number(level), // Convert to number

section, // Section name

isOccupied: isOccupied === "true", // Convert string to boolean

isReserved: isReserved === "true", // Convert string to boolean

occupiedBy: occupiedBy || "", // Default empty if not provided

});

try {

await parkingSpot.save(); // Save the new parking spot to MongoDB

res.status(201).json({ message: "Parking spot added successfully!" }); // Success response

} catch (error) {

res.status(400).json({ error: error.message }); // Handle error response

}

});

// GET API: Retrieve all parking spots

app.get("/getParkingSpots", async (req, res) => {

try {

// Fetch all parking spots from the database

const parkingSpots = await ParkingSpot.find(

{},

"spotNumber level section isOccupied isReserved occupiedBy timestamp"

); // Select only required fields

res.status(200).json(parkingSpots); // Return data as JSON response

} catch (error) {

res.status(500).json({ error: error.message }); // Handle error response

}

});

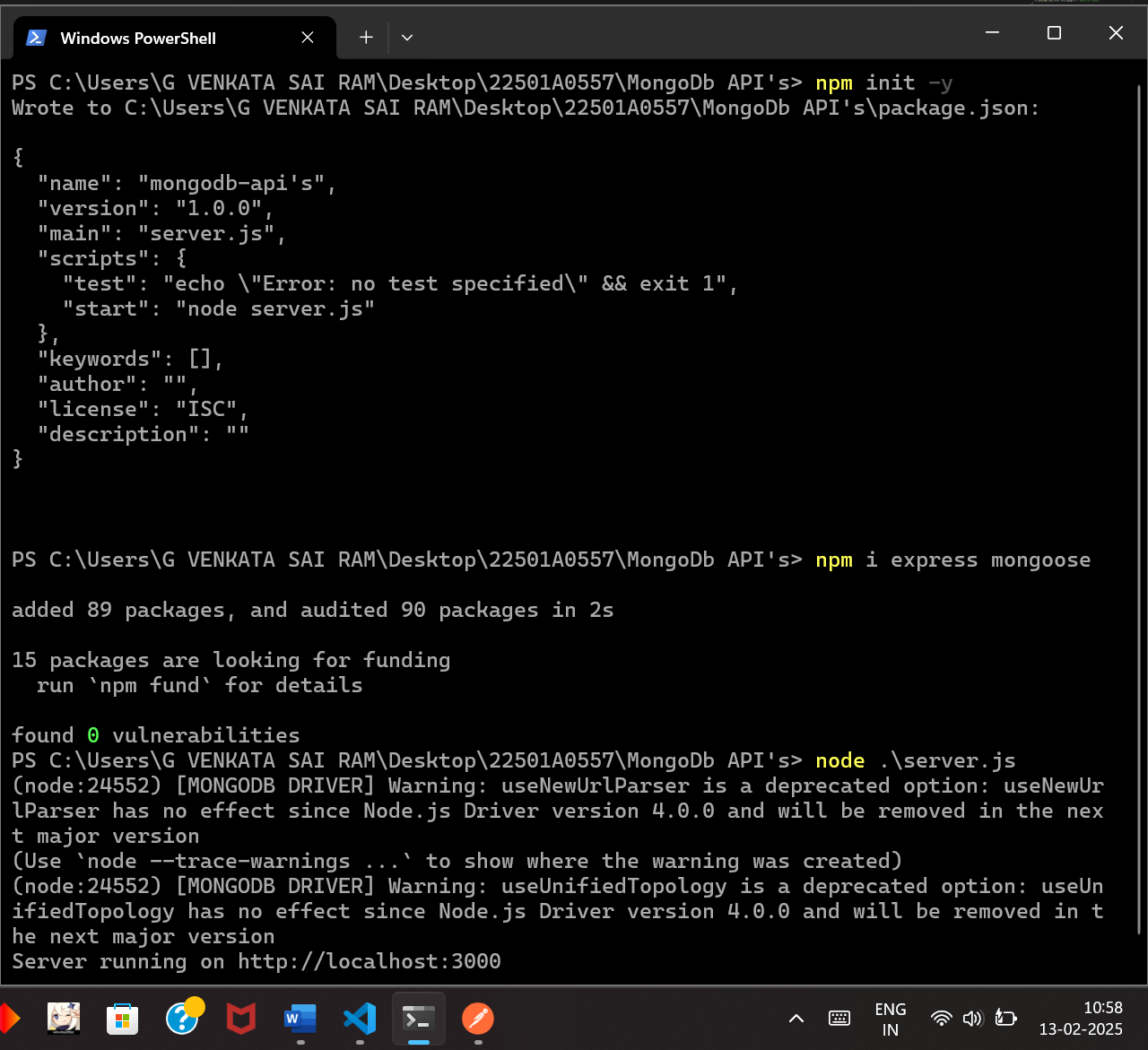
// Start the Express server

app.listen(port, () => {

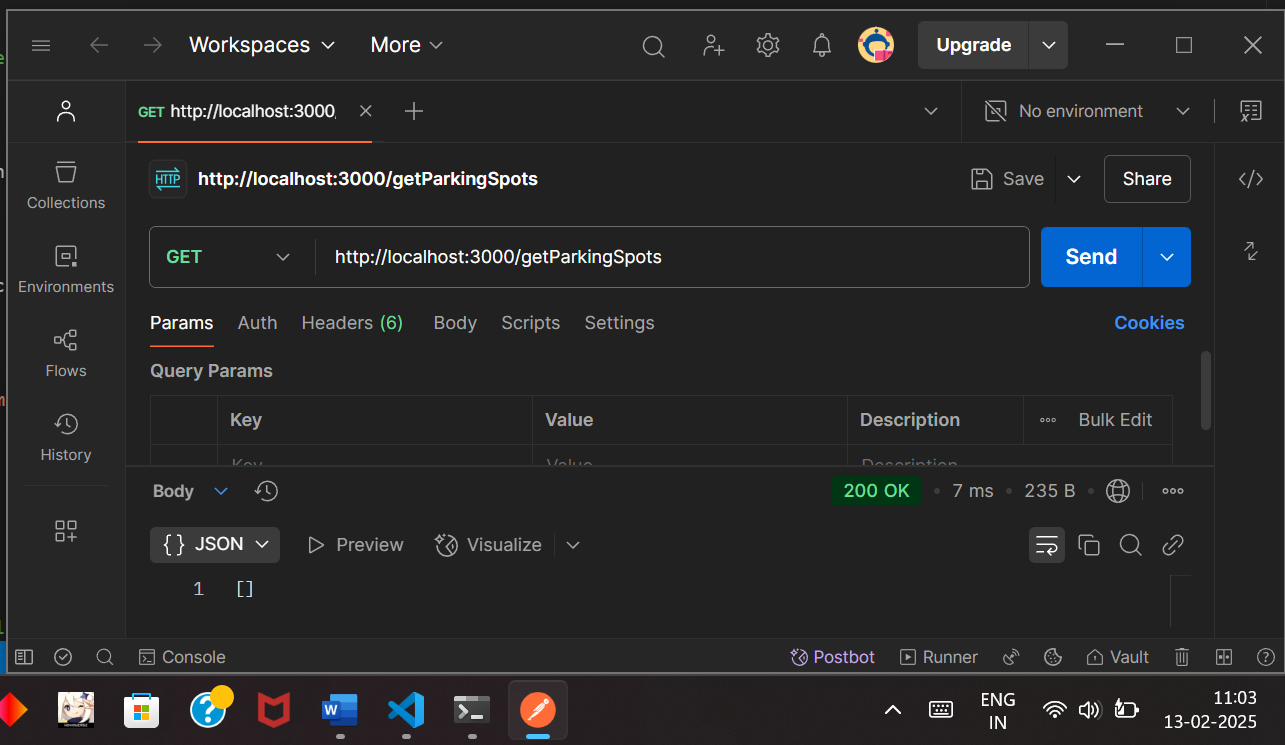
console.log(`Server running on http://localhost:${port}`); // Log server start

});

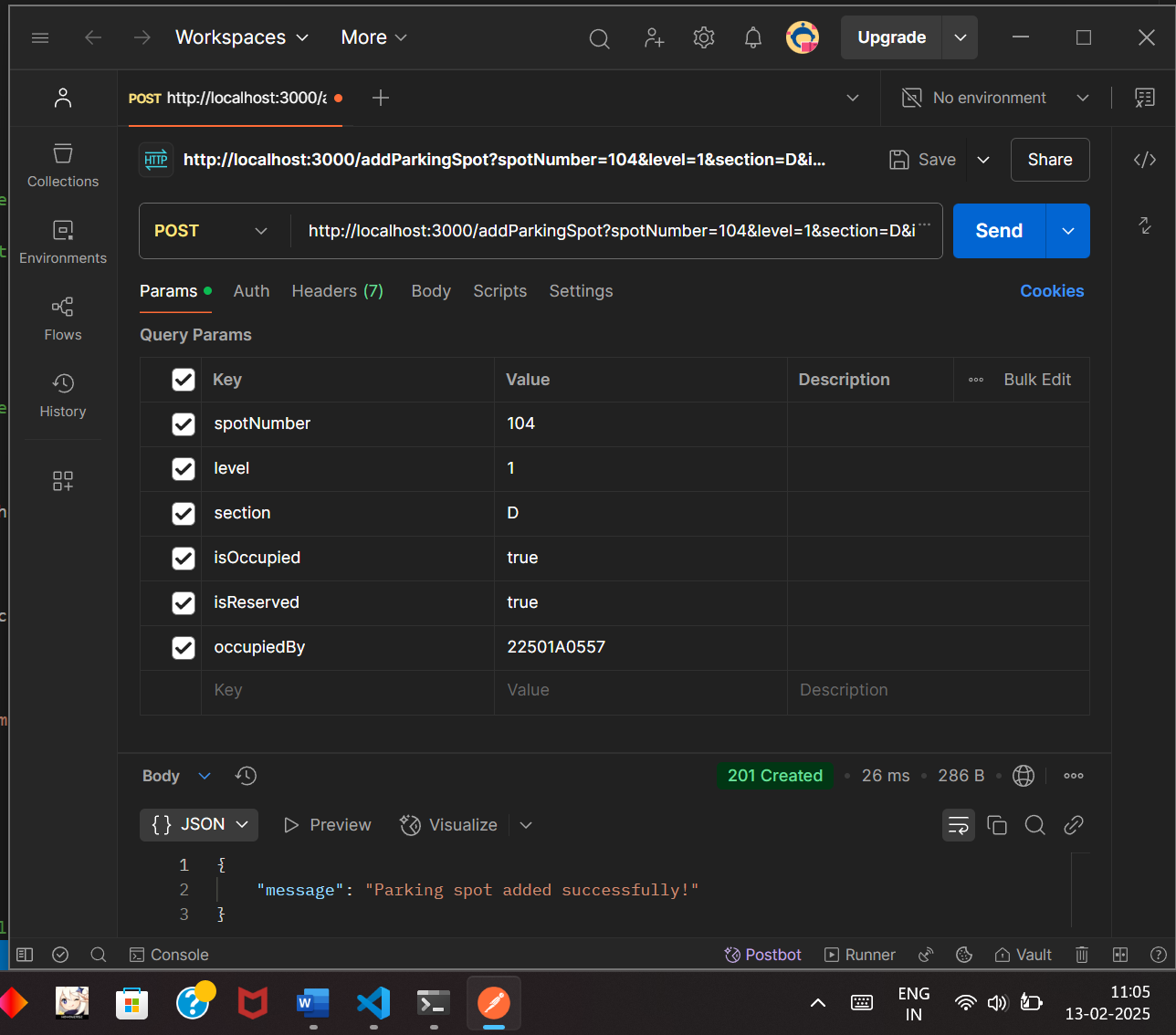
**Output:**



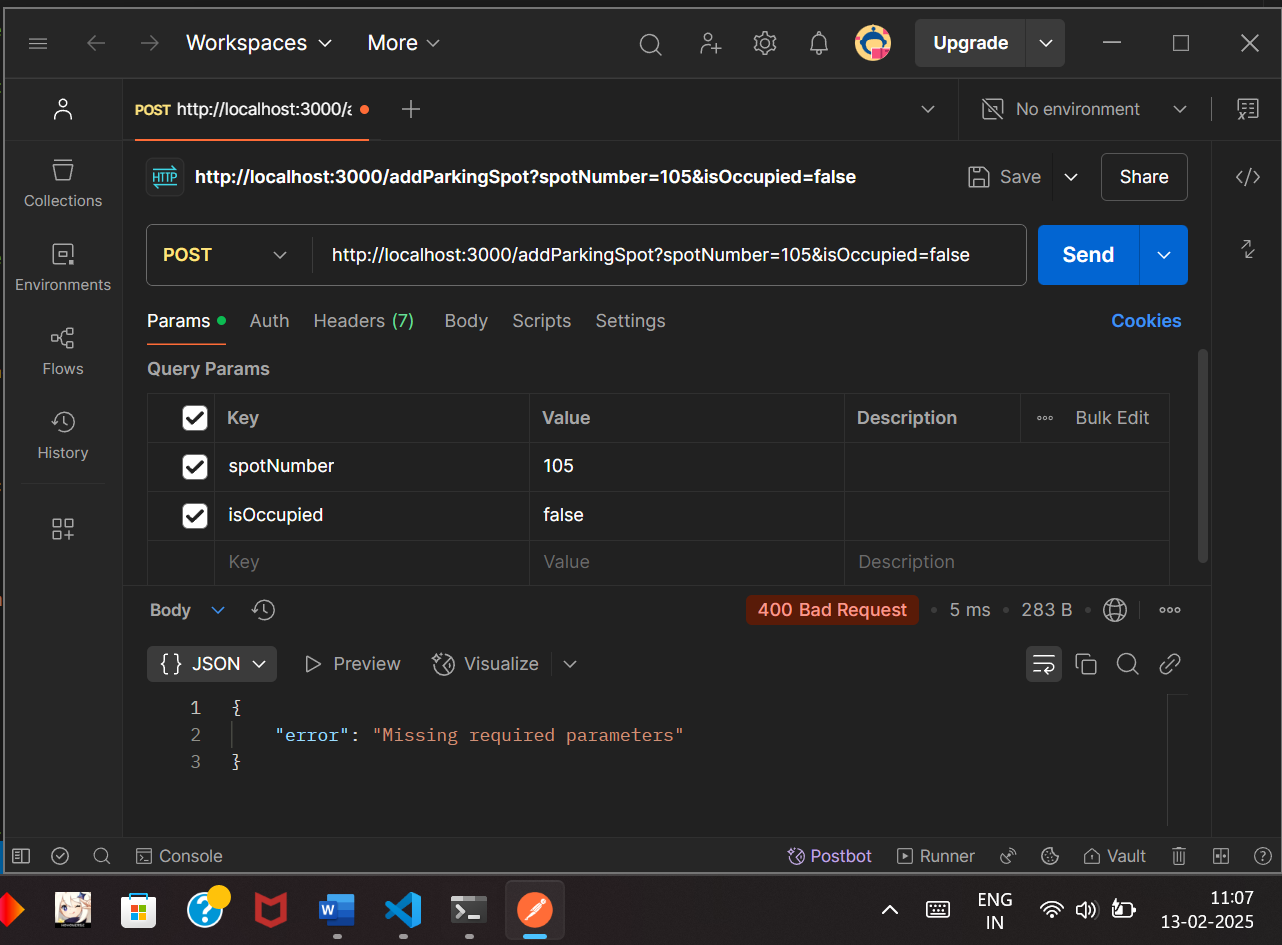
Initializing, Installing and Running the Server at port 3000



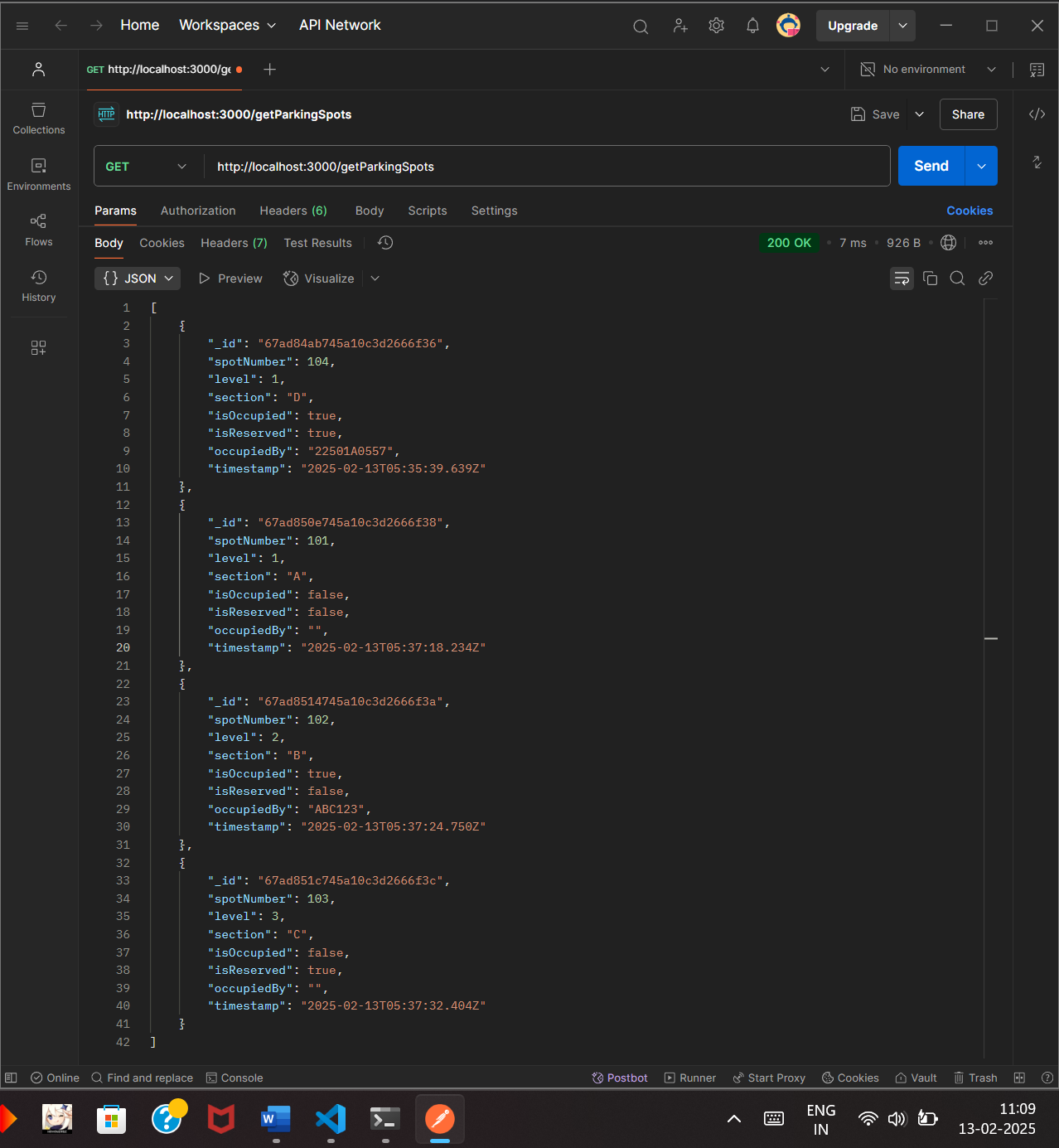
Checking the data from the database



Adding some data to the database



Trying to add Null values to required parameters



Retrieving the parking spots from the database

Date:

Experiment - 7

**Aim:**

Developing a simple CRUD application using the MERN stack

**Description:**

The **MERN Stack** is a JavaScript-based technology stack used for building full-stack web applications. It consists of:

* **MongoDB**: A NoSQL database used to store application data in JSON-like documents.
* **Express.js**: A backend framework that simplifies handling API requests and managing routes.
* **React.js**: A frontend framework for building dynamic user interfaces.
* **Node.js**: A runtime environment that allows JavaScript code to be executed on the server side.

**Steps to Develop a CRUD Application with MERN**

1. **Setup the Backend with Express and MongoDB**
   * Install **Node.js** and **MongoDB**
   * Create an **Express server** using Node.js
   * Connect the server to **MongoDB** using Mongoose
   * Define a **Schema** for the data model
   * Implement **API endpoints** to handle CRUD operations (Create, Read, Update, Delete)
2. **Develop the Frontend with React.js**
   * Create a **React application**
   * Build **components** for displaying and interacting with data
   * Use **fetch() or Axios** to communicate with the backend
   * Implement **state management** for dynamic updates
3. **Connecting Frontend and Backend**
   * Use **CORS middleware** to enable cross-origin requests
   * Fetch data from the backend and display it on the frontend
   * Handle **user interactions** like adding, updating, and deleting records
4. **Testing and Deployment**
   * Test all API routes using **Postman or a browser**
   * Ensure frontend components update correctly after backend changes

**Program:**

**server.js**

// Import required modules

const express = require("express"); // Web framework for Node.js

const mongoose = require("mongoose"); // MongoDB ODM

const cors = require("cors"); // Enables Cross-Origin Resource Sharing

const app = express(); // Initialize Express app

const port = 5000; // Define the port number

// Middleware

app.use(cors()); // Enable CORS for cross-origin requests

app.use(express.json()); // Parse incoming JSON data

// Connect to MongoDB

mongoose.connect("mongodb://localhost:27017/gamingDB", {

useNewUrlParser: true,

useUnifiedTopology: true,

});

// Define the Tournament Schema (structure of the tournament document)

const tournamentSchema = new mongoose.Schema({

name: String, // Tournament name

game: String, // Game name (e.g., FIFA, Dota, CSGO)

date: String, // Tournament date

prize: String, // Prize money

});

// Create a Model based on the Schema

const Tournament = mongoose.model("Tournament", tournamentSchema);

// Create a new tournament (POST)

app.post("/addTournament", async (req, res) => {

try {

const newTournament = new Tournament(req.body); // Create a new tournament instance

await newTournament.save(); // Save to MongoDB

res.status(201).json({ message: "Tournament Added Successfully!" });

} catch (error) {

res.status(400).json({ error: error.message }); // Return error if request fails

}

});

// Get all tournaments (GET)

app.get("/getTournaments", async (req, res) => {

try {

const tournaments = await Tournament.find(); // Retrieve all tournaments from DB

res.status(200).json(tournaments); // Send as JSON response

} catch (error) {

res.status(500).json({ error: error.message }); // Return error if request fails

}

});

// Update an existing tournament (PUT)

app.put("/updateTournament/:id", async (req, res) => {

try {

await Tournament.findByIdAndUpdate(req.params.id, req.body); // Find by ID & update

res.json({ message: "Tournament Updated Successfully!" }); // Success response

} catch (error) {

res.status(400).json({ error: error.message }); // Return error if request fails

}

});

// Delete a tournament (DELETE)

app.delete("/deleteTournament/:id", async (req, res) => {

try {

await Tournament.findByIdAndDelete(req.params.id); // Find by ID & delete

res.json({ message: "Tournament Deleted Successfully!" }); // Success response

} catch (error) {

res.status(400).json({ error: error.message }); // Return error if request fails

}

});

// Start the Express server

app.listen(port, () => {

console.log(`Server running on http://localhost:${port}`);

});

**index.html**

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

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

<title>Gaming Tournament CRUD</title>

<style>

/\* Basic styling \*/

body { font-family: Arial, sans-serif; text-align: center; }

table { width: 80%; margin: auto; border-collapse: collapse; }

th, td { border: 1px solid black; padding: 10px; }

input, button { margin: 5px; padding: 8px; }

</style>

</head>

<body>

<h2>Gaming Tournament CRUD</h2>

<!-- Input fields for adding a tournament -->

<input type="text" id="name" placeholder="Tournament Name">

<input type="text" id="game" placeholder="Game Name">

<input type="date" id="date">

<input type="text" id="prize" placeholder="Prize Money">

<button onclick="addTournament()">Add Tournament</button>

<h3>List of Tournaments</h3>

<!-- Table to display tournament data -->

<table>

<thead>

<tr>

<th>Name</th>

<th>Game</th>

<th>Date</th>

<th>Prize</th>

<th>Actions</th>

</tr>

</thead>

<tbody id="tournamentTable"></tbody> <!-- Table body for tournament data -->

</table>

<script>

const apiUrl = "http://localhost:5000"; // Backend API URL

// Fetch all tournaments from the server and display in the table

async function fetchTournaments() {

const res = await fetch(apiUrl + "/getTournaments"); // Send GET request

const data = await res.json(); // Parse response as JSON

const table = document.getElementById("tournamentTable"); // Get table body

table.innerHTML = ""; // Clear previous data

// Loop through each tournament and add to table

data.forEach(tournament => {

table.innerHTML += `

<tr>

<td>${tournament.name}</td>

<td>${tournament.game}</td>

<td>${tournament.date}</td>

<td>${tournament.prize}</td>

<td>

<button onclick="editTournament('${tournament.\_id}')">Edit</button>

<button onclick="deleteTournament('${tournament.\_id}')">Delete</button>

</td>

</tr>

`;

});

}

// Function to add a new tournament

async function addTournament() {

const name = document.getElementById("name").value;

const game = document.getElementById("game").value;

const date = document.getElementById("date").value;

const prize = document.getElementById("prize").value;

// Send POST request to add tournament

await fetch(apiUrl + "/addTournament", {

method: "POST",

headers: { "Content-Type": "application/json" },

body: JSON.stringify({ name, game, date, prize }) // Convert data to JSON format

});

fetchTournaments(); // Refresh tournament list

}

// Function to delete a tournament

async function deleteTournament(id) {

await fetch(apiUrl + "/deleteTournament/" + id, { method: "DELETE" }); // Send DELETE request

fetchTournaments(); // Refresh tournament list

}

// Function to edit a tournament

async function editTournament(id) {

const newName = prompt("Enter new name:"); // Prompt user for new values

const newGame = prompt("Enter new game:");

const newDate = prompt("Enter new date:");

const newPrize = prompt("Enter new prize:");

// Send PUT request to update tournament

await fetch(apiUrl + "/updateTournament/" + id, {

method: "PUT",

headers: { "Content-Type": "application/json" },

body: JSON.stringify({ name: newName, game: newGame, date: newDate, prize: newPrize })

});

fetchTournaments(); // Refresh tournament list

}

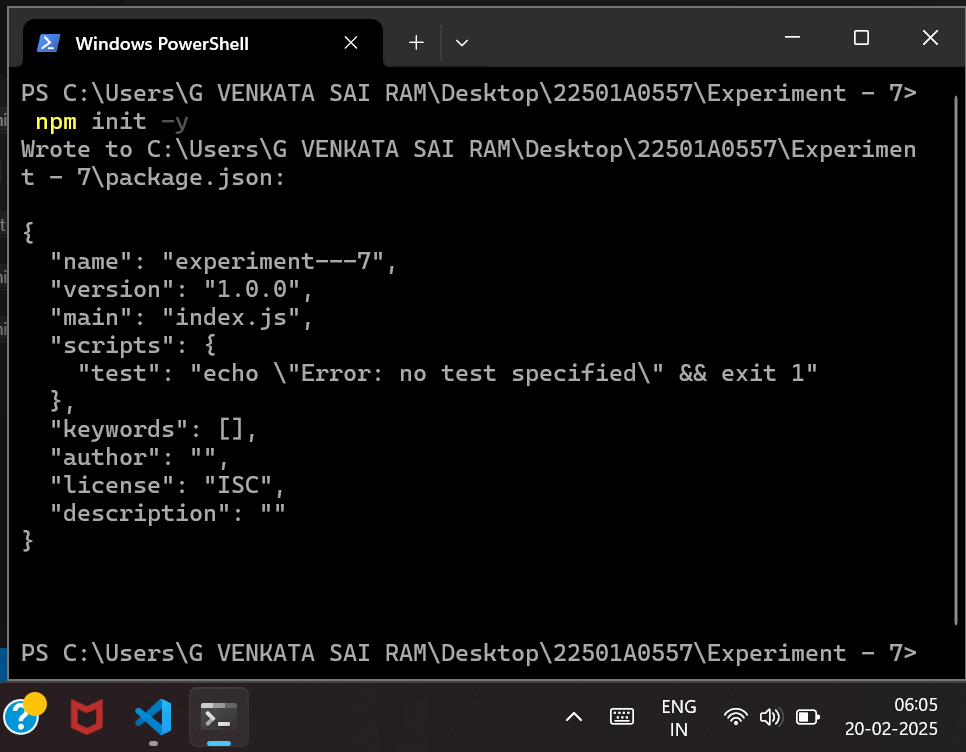
fetchTournaments(); // Load tournaments on page load

</script>

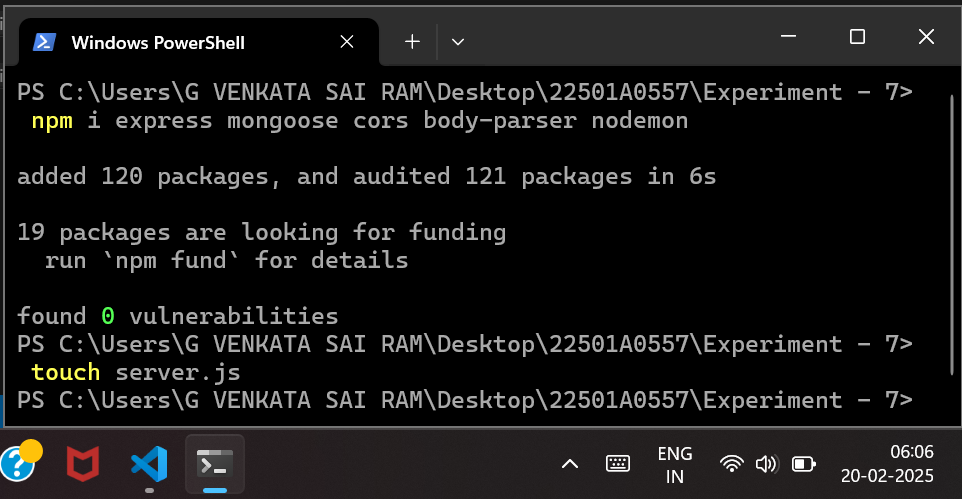
</body>

</html>

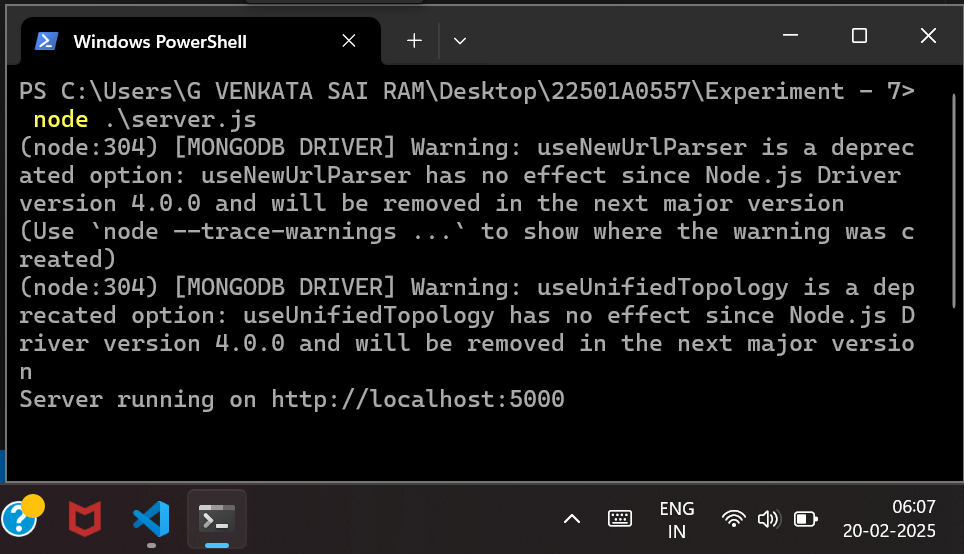
**Output:**



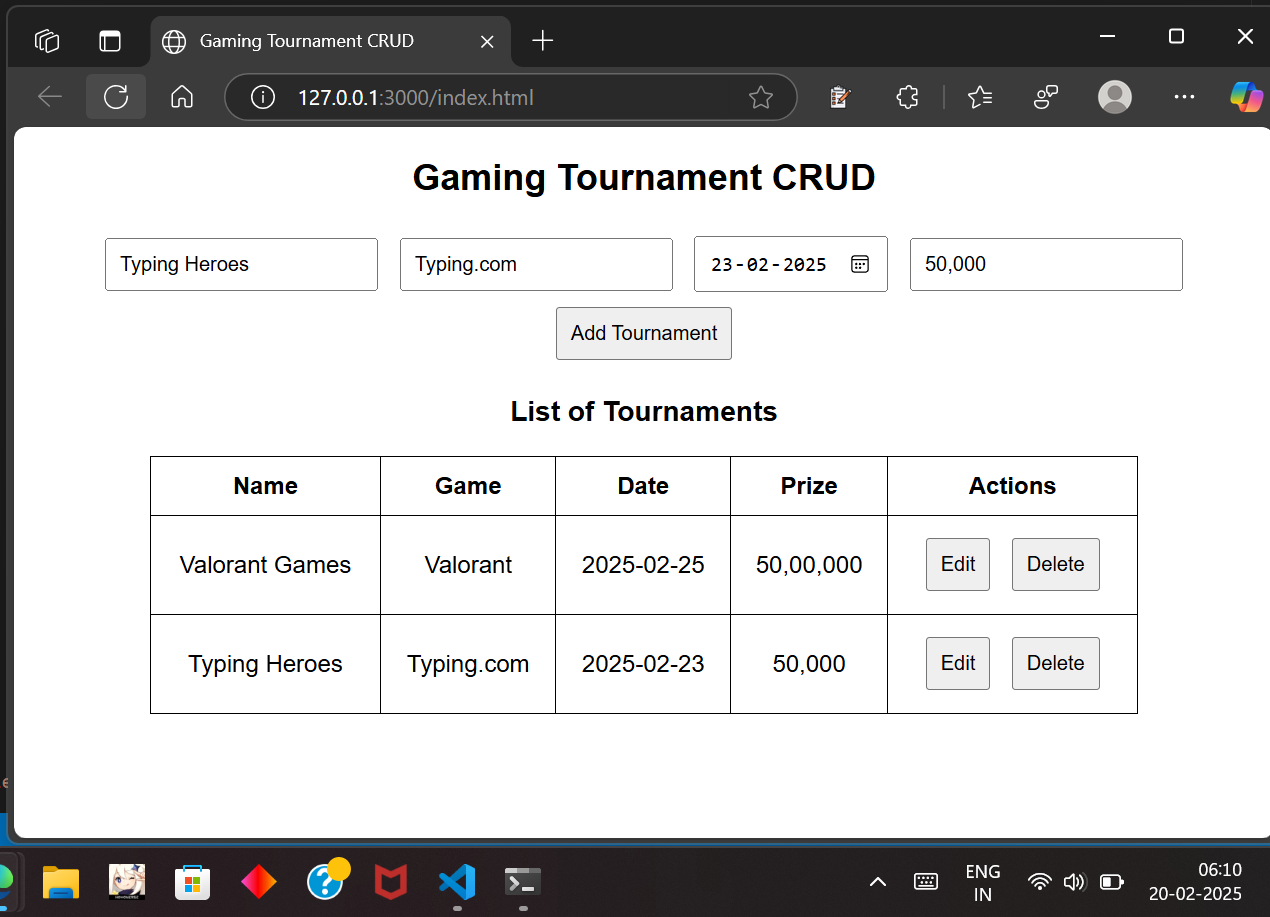
Initializing the npm environment



Installing the necessary modules



Running the server.js using Node



The Gaming Tournament website with CRUD Operations

Date:

Experiment - 8

**Aim:**

Implement React Elements and Components

**Description:**

**React Elements**

* The **smallest building blocks** of a React application.
* Represent **UI elements** such as buttons, headings, paragraphs, or divs.
* Can be created using **React.createElement()** or **JSX syntax** (e.g., <h1>Hello</h1>).
* **Immutable** once created, meaning they cannot be changed after rendering.
* React Elements are responsible for **describing what should appear on the screen**.

**React Components**

* **Reusable UI pieces** that return **React Elements**.
* Help in structuring **complex UIs** by breaking them into smaller parts.
* Two types of components:
  + **Functional Components**: Defined as functions that return JSX, recommended for most cases.
  + **Class Components**: Defined using ES6 classes, primarily used when lifecycle methods are needed.
* Components **can be nested inside other components** to create a **hierarchical UI structure**.
* React components allow **code reusability, better maintainability, and efficient rendering**.

**Program:**

**App.js**

// Importing React and child components

import React from "react";

import Header from "./Header";

import Main from "./Main";

import Footer from "./Footer";

// Root component that holds the structure of the application

function App() {

return (

<div style={{ textAlign: "center", fontFamily: "Arial, sans-serif" }}>

{/\* Header Component \*/}

<Header />

{/\* Main Content Component \*/}

<Main />

{/\* Footer Component \*/}

<Footer />

</div>

);

}

// Export App component for use in index.js

export default App;

**Header.js**

// Importing React

import React from "react";

// Header Component to display the title

function Header() {

return (

<header>

<h1>React Elements & Components</h1>

</header>

);

}

// Export Header component for use in App.js

export default Header;

**Main.js**

// Importing React and child components

import React from "react";

import WelcomeElement from "./WelcomeElement";

import FunctionalComponent from "./FunctionalComponent";

import ClassComponent from "./ClassComponent";

// Main Component that contains different types of elements and components

function Main() {

return (

<main>

{/\* Using a React Element \*/}

<WelcomeElement />

{/\* Using a Functional Component with props \*/}

<FunctionalComponent name="Alice" />

{/\* Using a Class Component with props \*/}

<ClassComponent name="Bob" />

</main>

);

}

// Export Main component for use in App.js

export default Main;

**WelcomeElement.js**

// Importing React

import React from "react";

// Creating a React element using React.createElement()

const WelcomeElement = () => {

return React.createElement("h2", {}, "Welcome to React Elements!");

};

// Export WelcomeElement for use in Main.js

export default WelcomeElement;

**FunctionalComponent.js**

// Importing React

import React from "react";

// Functional Component that receives props

function FunctionalComponent(props) {

return <h3>Hello, {props.name}! This is a Functional Component.</h3>;

}

// Export FunctionalComponent for use in Main.js

export default FunctionalComponent;

**ClassComponent.js**

// Importing React and Component class

import React, { Component } from "react";

// Class Component that receives props

class ClassComponent extends Component {

render() {

return <h3>Hello, {this.props.name}! This is a Class Component.</h3>;

}

}

// Export ClassComponent for use in Main.js

export default ClassComponent;

**Footer.js**

// Importing React

import React from "react";

// Footer Component to display the copyright notice

function Footer() {

return (

<footer>

<p>© 2024 React Elements & Components</p>

</footer>

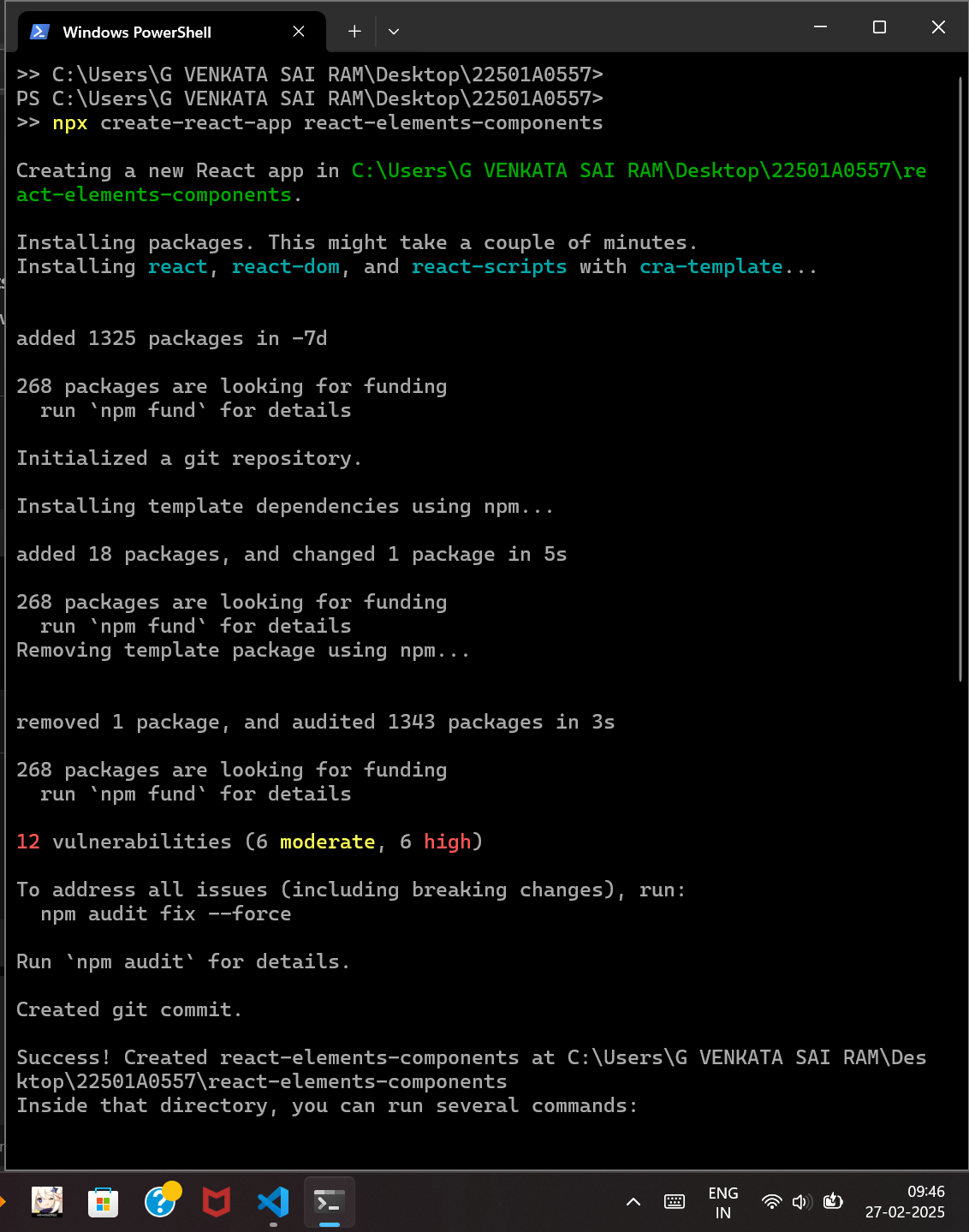
);

}

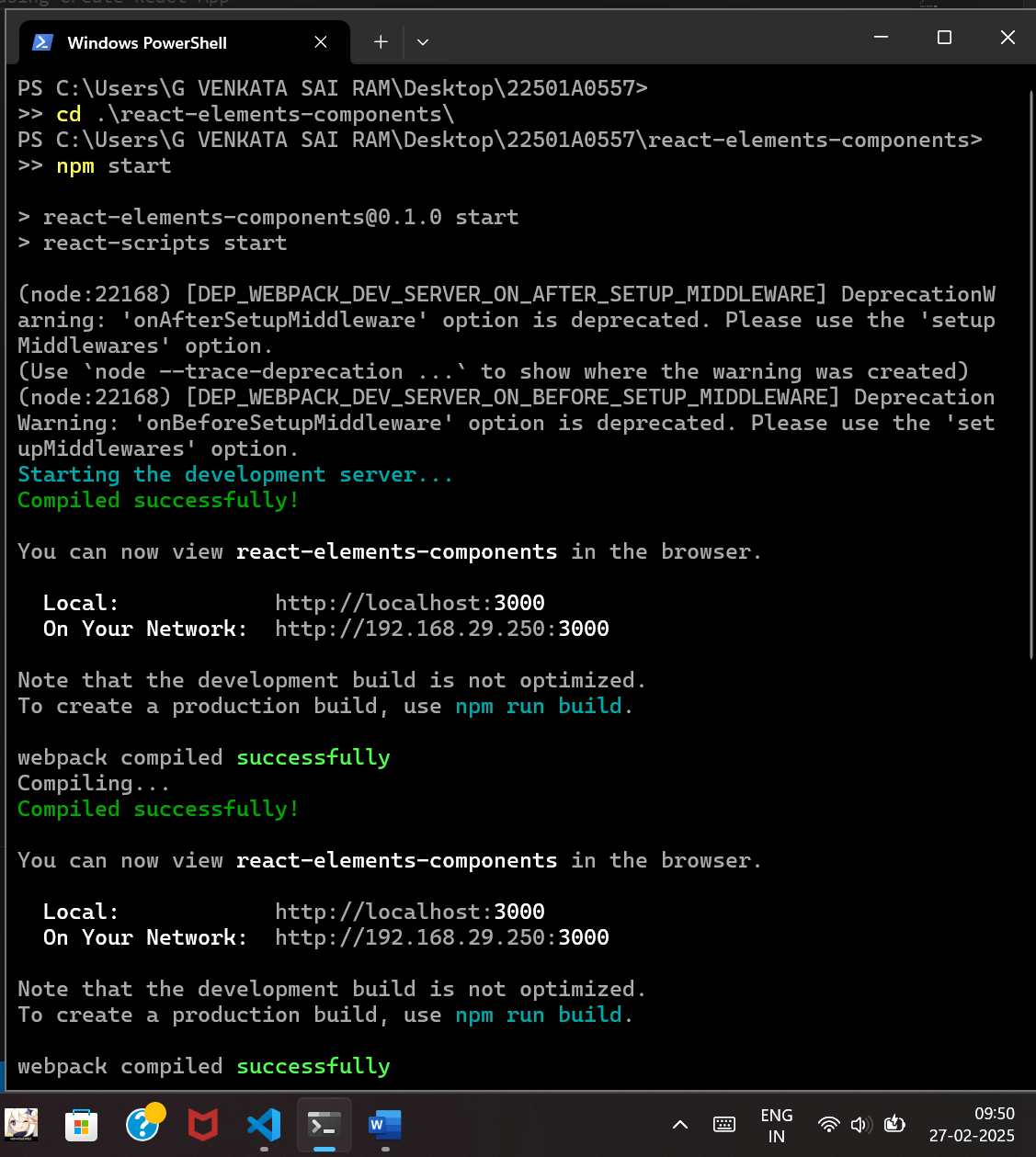
// Export Footer component for use in App.js

export default Footer;

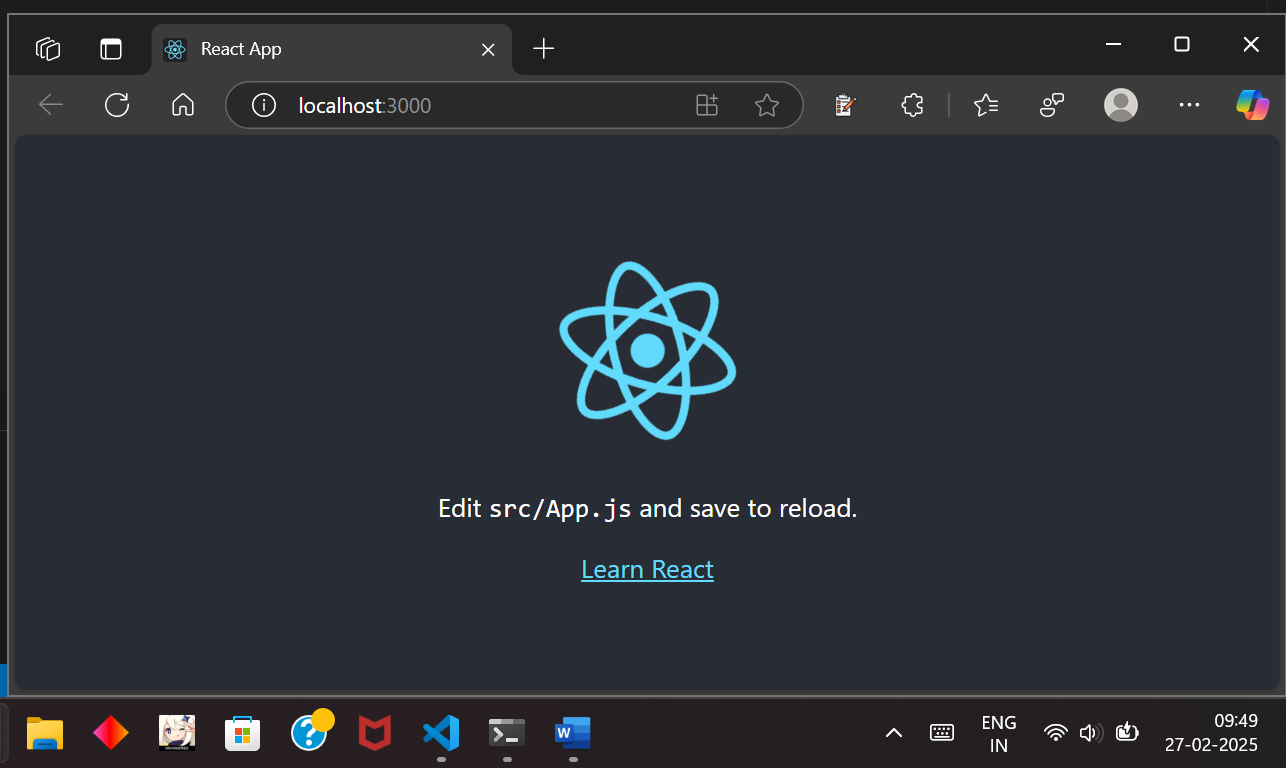
**Output:**



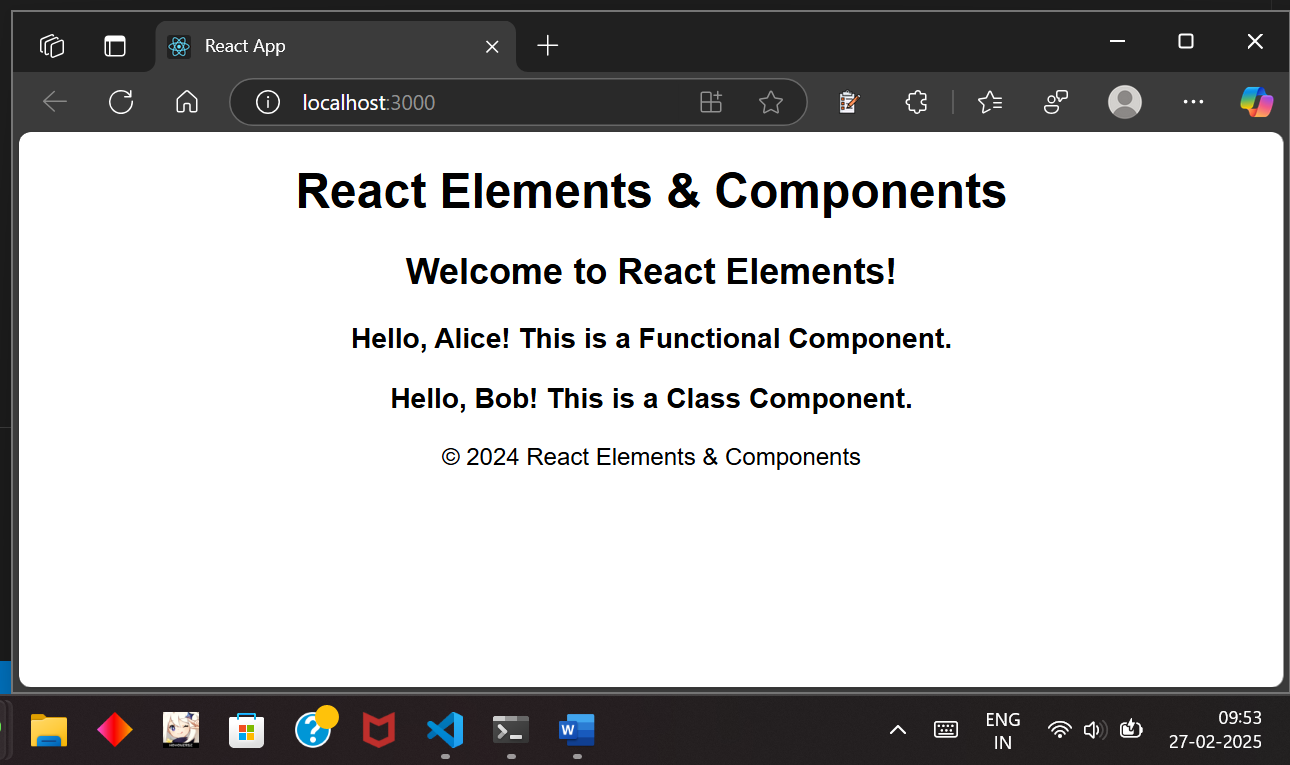
Creating a react app using npx



Starting the server using npm



Initially we can see this page by default



Example page for React Elements & Components

Date:

Experiment - 9

**Aim:**

Develop a Single Page Application (SPA)

**Description:**

**What is a Single Page Application (SPA)?**

* A web application that dynamically updates content without reloading the entire page.
* Uses JavaScript frameworks like React, Angular, or Vue.js.
* Improves performance and user experience by fetching only necessary data.

**Key Features of a SPA**

* Uses client-side routing (e.g., React Router).
* Loads only required components when navigating between views.
* Reduces server requests by managing state on the client side.

**Steps to Develop an SPA with React**

1. **Install Node.js and npm**
   * Ensure Node.js is installed (node -v and npm -v to verify).
2. **Create a React App**
3. npx create-react-app my-spa
4. cd my-spa
5. npm start
   * This sets up a new React project and runs the development server.
6. **Install React Router for Navigation**
7. npm install react-router-dom
   * Enables client-side routing without full-page reloads.
8. **Define Routes in the Application**
   * Use BrowserRouter, Routes, and Route components from react-router-dom.
   * Example routes: Home, About, Contact.
9. **Implement Components for Different Views**
   * Create separate functional components for each page.
   * Example: Home.js, About.js, Contact.js.
10. **Navigation with React Router**
    * Use Link components instead of <a> tags to prevent full-page reloads.
11. **Manage State Efficiently**
    * Use useState and useEffect for handling application state.
    * For complex state management, consider Redux or Context API.
12. **Deploy the Application**
    * Build the project using:
    * npm run build
    * Deploy to platforms like Vercel, Netlify, or Firebase Hosting.

**Program:**

**Output:**