**Department of Computational Sciences**

**Laboratory Manual**

(BCA2023, SEMESTER 4TH,

FULL STACK DEVELOPMENT-1 and BCA47113(P))

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Table of Contents**

| Experiment/Lab Exercise/Activity No. | Name of the Experiment/Activity/Exercise | Page Number(s) |
| --- | --- | --- |
| 1 | Create a Node.js script demonstrating variables, functions, conditionals, loops, objects, arrays, and asynchronous code. |  |
| 2 | 1. Write a Node.js program to take user input, add two numbers, and display the result. 2. b. Create a Node.js script to display system environment variables. |  |
| 3 | 1. Write a Node.js program to demonstrate command-line arguments and parse their values. 2. b. Explain REPL in Node.js with examples of real-time JavaScript testing. |  |
| 4 | Write a Node.js script for the REPL environment, including functions, array methods, and asynchronous operations. |  |
| 5 | Write a Node.js program to calculate the execution time of multiple functions. |  |
| 6 | Write a Node.js program to explain asynchronous and synchronous functions with examples. |  |
| 7 | 1. Write a Node.js program to describe the core path module. 2. Write a Node.js program to describe the core modules path and fs. |  |
| 8 | 1. Create a Node.js program demonstrating the use of Buffers, including:  Creating buffers with Buffer.from() and Buffer.alloc(). Writing data using write(). Slicing buffers using slice(). |  |
| 9 | 1. Write a Node.js program to describe local modules with two functions. 2. Write a Node.js program to describe local modules with an object. |  |
| 10 | Create a package.json file for a Node.js project and describe its setup with an external package like dotenv. |  |
| 11 | 1. Write a Node.js program to create a server using the http core module. 2. Extend the server to handle routes and process different HTTP methods (GET, POST, PUT, DELETE). |  |
| 12 | Write a Node.js program to perform CRUD operations (Create, Read, Update, Delete) on a file using the fs core module. |  |
| 13 | 1. Create a Node.js project named "Calculator" with a server handling routes for: 2. A home page (/) displaying a welcome message and a link to the calculator page. 3. A calculator page (/calculator) with a form to input two numbers and calculate their sum. 4. A result page (/calculate-result) parsing user input, calculating the sum using a separate addition module, and displaying the result. |  |
| 14 | 1. Create a Node.js project using Express that: 2. Responds with both HTML and JSON. 3. Serves different HTML files at different routes. |  |
| 15 | 1. Extend the Express project to: 2. Serve different HTML files at different routes and perform a calculation (e.g., add two numbers from a user form). 3. Implement routing with dynamic parameters in the URL. 4. Use application-level middleware for tasks like logging or handling errors. |  |

**1. Assignment 1: Create a Node.js script demonstrating variables, functions, conditionals, loops, objects, arrays, and asynchronous code.**

**A. Aim/Purpose of the Experiment:**  
i. To develop a Node.js script that demonstrates the use of:  
o Variables  
o Functions  
o Conditionals  
o Loops  
o Objects  
o Arrays  
o Asynchronous programming

**B. Learning Outcomes:**  
i. Understand the fundamentals of Node.js.  
ii. Implement core JavaScript concepts in a server-side environment.  
iii. Learn how asynchronous operations work in Node.js.  
iv. Gain hands-on experience with functions, conditionals, loops, objects, and arrays.

**C. Prerequisites:**  
i. Basic understanding of JavaScript.  
ii. Familiarity with the Node.js runtime environment.  
iii. Knowledge of asynchronous programming concepts (callbacks, promises, async/await).

**D. Materials/Equipment/Apparatus/Devices/Software Required:**  
i. Computer with Node.js installed (latest LTS version recommended).  
ii. Code editor (e.g., VS Code, Sublime Text, or any preferred IDE).  
iii. Terminal/Command prompt for executing Node.js scripts.

**E. Introduction and Theory:**  
i. Node.js is a runtime environment that allows JavaScript to run on the server side.  
ii. It is non-blocking and event-driven, making it efficient for handling asynchronous operations.  
iii. The core concepts in this experiment include:  
o Variables: Used to store data values.  
o Functions: Blocks of reusable code.  
o Conditionals: Decision-making structures like if-else.  
o Loops: Iterative execution (for, while, etc.).  
o Objects & Arrays: Data structures for storing multiple values.  
o Asynchronous Code: Handling operations without blocking execution.

**F. Operating Procedure:**  
i. Setup Node.js Environment:  
o Install Node.js.  
o Initialize a new project (optional: npm init).  
ii. Create a Script (script.js) that includes:  
o Declaring variables.  
o Writing functions.  
o Using conditionals and loops.  
o Creating and manipulating objects and arrays.  
o Implementing asynchronous operations using callbacks, promises, and async/await.  
iii. Run the script using node script.js and analyze the output.

**G. Precautions and/or Troubleshooting:**  
i. Ensure Node.js is installed correctly (node -v to check version).  
ii. Use console.log() for debugging.  
iii. Handle errors in asynchronous functions properly with try-catch or .catch().

**H. Observations:**  
i. Note the behavior of synchronous vs. asynchronous code.  
ii. Observe how loops, conditionals, and functions work in Node.js.  
iii. Compare execution time for synchronous and asynchronous tasks.

**I. Calculations & Analysis:**  
i. Analyze execution order in asynchronous code.  
ii. Evaluate memory and performance usage for loops and data structures.

**J. Result & Interpretation:**  
i. Successfully demonstrated fundamental Node.js concepts.  
ii. Understood the working of asynchronous programming in Node.js.

**K. Follow-up Questions:**  
i. What are the advantages of using asynchronous programming in Node.js?  
ii. How do promises differ from callbacks?  
iii. Why is Node.js single-threaded, and how does it handle concurrency?

**L. Extension and Follow-up Activities (if applicable):**  
i. Extend the script to include file system operations using the fs module.  
ii. Implement an HTTP server using the http module.

**M. Assessments:**  
i. Write a script that reads a file asynchronously and processes the data.  
ii. Modify the script to include exception handling and logging.

**N. Suggested Reading:**  
i. Official Node.js Documentation  
ii. *Node.js Design Patterns* by Mario Casciaro

### **Assignment 2: Node.js Programs for User Input and System Variables**

**A. Aim/Purpose of the Experiment:**  
i. To develop Node.js programs that:

* Take user input, perform addition, and display the result.
* Retrieve and display system environment variables.

**B. Learning Outcomes:**

1. Understand how to take user input in a Node.js script.
2. Learn how to perform basic arithmetic operations.
3. Explore system environment variables in Node.js.
4. Gain hands-on experience with readline and process.env.

**C. Prerequisites:**

* Basic knowledge of JavaScript.
* Familiarity with the Node.js runtime environment.
* Understanding of standard input/output operations.

**D. Materials/Equipment/Apparatus/Devices/Software Required:**

* Computer with Node.js installed.
* Code editor (e.g., VS Code, Sublime Text, or any preferred IDE).
* Terminal/Command prompt for executing Node.js scripts.

**E. Introduction and Theory:**

* **User Input Handling:** In Node.js, the readline module allows capturing user input from the console.
* **Arithmetic Operations:** JavaScript allows basic mathematical calculations.
* **System Environment Variables:** The process.env object contains environment variables accessible in Node.js.

**F. Operating Procedure:**  
**Part (a): Taking User Input and Adding Two Numbers**

1. Create a file add\_numbers.js.
2. Import the readline module.
3. Prompt the user for two numbers.
4. Convert the input strings to numbers and calculate the sum.
5. Display the result in the console.
6. Run the script using node add\_numbers.js.

**Part (b): Displaying System Environment Variables**

1. Create a file env\_variables.js.
2. Access process.env to retrieve environment variables.
3. Display the variables in the console.
4. Run the script using node env\_variables.js.

**G. Precautions and/or Troubleshooting:**

* Ensure Node.js is installed correctly (node -v to check version).
* Handle invalid inputs by validating user input.
* Use console.log() for debugging.

**H. Observations:**

* Note how user input is received and processed.
* Observe the behavior of process.env when displaying environment variables.

**I. Calculations & Analysis:**

* Analyze how input values are converted and summed.
* Evaluate system environment variables and their usage in applications.

**J. Result & Interpretation:**

* Successfully implemented a user input-based addition program.
* Retrieved and displayed system environment variables using Node.js.

**K. Follow-up Questions:**

1. What is the difference between synchronous and asynchronous input handling in Node.js?
2. How can we modify the script to handle floating-point numbers?
3. Why are environment variables important in software applications?

**L. Extension and Follow-up Activities (if applicable):**

* Modify the script to take multiple numbers as input and calculate their sum.
* Extend the environment variable script to display selected variables instead of all.

**M. Assessments:**

* Write a program that accepts a username as input and greets the user.
* Create a script to filter and display specific environment variables (e.g., PATH, USER).

**N. Suggested Reading:**

1. Official Node.js Documentation
2. *Node.js 8 the Right Way* by Jim R. Wilson
3. *JavaScript: The Good Parts* by Douglas Crockford

### **Assignment 3: Command-Line Arguments & REPL in Node.js**

#### A. Aim/Purpose of the Experiment:

i. To write a Node.js program that demonstrates:

* Command-line arguments and parsing their values
* The use of REPL (Read-Eval-Print Loop) for real-time JavaScript testing in Node.js

#### B. Learning Outcomes:

After completing this experiment, students will be able to:

1. Understand how to handle command-line arguments in Node.js.
2. Learn how to parse and manipulate arguments in scripts.
3. Gain hands-on experience with the REPL environment in Node.js.
4. Understand the benefits of using REPL for debugging and testing JavaScript code.

#### C. Prerequisites:

i. Basic knowledge of JavaScript.  
ii. Familiarity with the Node.js runtime environment.  
iii. Understanding of process arguments (process.argv).  
iv. Basic command-line usage.

#### D. Materials/Equipment/Apparatus/Devices/Software Required:

i. Computer with Node.js installed (latest LTS version recommended).  
ii. Code editor (e.g., VS Code, Sublime Text, or any preferred IDE).  
iii. Terminal/Command prompt for executing Node.js scripts.

#### E. Introduction and Theory:

##### (a) Command-Line Arguments in Node.js:

Node.js allows passing arguments from the command line using process.argv, where:

* process.argv[0] contains the Node.js executable path.
* process.argv[1] contains the script filename.
* process.argv[2] onwards contain user-supplied arguments.

##### (b) REPL (Read-Eval-Print Loop) in Node.js:

REPL is an interactive shell that allows users to:

* **Read** input from the user.
* **Evaluate** JavaScript expressions.
* **Print** the output.
* **Loop** for continuous execution until exited.

#### F. Operating Procedure:

##### (a) Writing a Node.js Program for Command-Line Arguments:

1. Create a new Node.js file (e.g., args.js).
2. Write a script that takes two numbers from the command line, adds them, and displays the result.
3. Run the script in the terminal using:
4. node args.js 5 10
5. Observe the output.

**Code Example (args.js):**

// Get command-line arguments

const args = process.argv.slice(2);

// Convert arguments to numbers

const num1 = parseFloat(args[0]);

const num2 = parseFloat(args[1]);

// Perform addition

if (!isNaN(num1) && !isNaN(num2)) {

console.log(`Sum: ${num1 + num2}`);

} else {

console.log("Please provide valid numbers as arguments.");

}

##### (b) Using REPL in Node.js:

1. Open the terminal and type:
2. node

This starts the REPL environment.

1. Execute JavaScript expressions interactively:
2. > let x = 10;
3. > let y = 20;
4. > x + y

Output:

30

1. Use .help to view REPL commands.
2. Use .exit or Ctrl + C to exit the REPL.

#### G. Precautions and/or Troubleshooting:

i. Ensure Node.js is installed correctly (node -v to check version).  
ii. Provide valid numbers as arguments to avoid NaN errors.  
iii. Use console.log() for debugging outputs in scripts.  
iv. In REPL, use .clear to reset the session.

#### H. Observations:

i. The script correctly parses and processes command-line arguments.  
ii. The REPL environment allows interactive execution and debugging.

#### I. Calculations & Analysis:

i. Analyze how command-line arguments are handled and converted into usable data.  
ii. Observe the difference between synchronous execution in scripts and interactive REPL execution.

#### J. Result & Interpretation:

i. Successfully implemented a script to handle command-line arguments in Node.js.  
ii. Understood the working of REPL for real-time JavaScript testing.

#### K. Follow-up Questions:

1. What is the purpose of process.argv in Node.js?
2. How can you parse named arguments from the command line?
3. What are some real-world applications of using REPL in development?

#### L. Extension and Follow-up Activities (if applicable):

i. Modify the script to support named arguments using the yargs module.  
ii. Implement a command-line calculator with multiple operations.

#### M. Assessments:

1. Write a script that takes multiple arguments and performs different mathematical operations based on user input.
2. Use REPL to test different JavaScript functions dynamically.

#### N. Suggested Reading:

1. [Official Node.js Documentation](https://nodejs.org/)
2. *Node.js Design Patterns* by Mario Casciaro

### **Assignment 4: Write a Node.js Script for the REPL Environment, Including Functions, Array Methods, and Asynchronous Operations**

#### **1. Aim/Purpose of the Experiment:**

* Learn to use functions, array methods, and asynchronous operations in Node.js REPL.

#### **2. Learning Outcomes:**

* Understand how to define and use functions, manipulate arrays, and handle asynchronous code.

#### **3. Prerequisites:**

* Basic JavaScript and Node.js knowledge.

#### **4. Materials Required:**

* Node.js installed, terminal/command prompt, code editor.

#### **5. Introduction and Theory:**

* **REPL** allows interactive JavaScript execution.
* Functions, array methods (map(), filter(), reduce()), and asynchronous operations are core concepts in Node.js.

#### **6. Operating Procedure:**

1. **Start REPL**: Type node in terminal.
2. **Functions**:
3. function add(a, b) { return a + b; }
4. add(5, 10); // 15
5. **Array Methods**:
6. let arr = [1, 2, 3, 4];
7. arr.map(x => x \* 2); // [2, 4, 6, 8]
8. arr.filter(x => x > 2); // [3, 4]
9. arr.reduce((sum, x) => sum + x, 0); // 10
10. **Asynchronous Operations**:
11. setTimeout(() => console.log('Done!'), 1000); // Prints after 1 second

#### **7. Precautions:**

* Ensure Node.js is installed and REPL is correctly initialized.

#### **8. Observations:**

* Functions worked as expected.
* Array methods transformed arrays.
* Asynchronous code executed without blocking.

#### **9. Result & Interpretation:**

* Node.js REPL can handle functions, array operations, and async code efficiently.

#### **10. Follow-up Questions:**

1. What are Promises in Node.js?
2. How does async/await work?

#### **11. Suggested Reading:**

* [Node.js Docs](https://nodejs.org/en/docs/)

### **Assignment 5: Write a Node.js Program to Calculate the Execution Time of Multiple Functions**

#### **1. Aim/Purpose of the Experiment:**

* Learn how to calculate the execution time of multiple functions in a Node.js program.

#### **2. Learning Outcomes:**

* Understand how to measure the time taken by different functions.
* Learn to use console.time() and console.timeEnd() to track execution time.

#### **3. Prerequisites:**

* Basic knowledge of Node.js and JavaScript functions.
* Familiarity with timing functions in Node.js (console.time() and console.timeEnd()).

#### **4. Materials/Equipment/Apparatus / Devices/Software Required:**

* Node.js installed on your system.
* Code editor (e.g., VS Code, Sublime Text).
* Terminal or command prompt.

#### **5. Introduction and Theory:**

* **Execution Time Measurement**: In Node.js, the console.time() and console.timeEnd() functions are used to measure the execution time of a block of code. These functions help to benchmark different functions and determine which ones perform better in terms of time complexity.

#### **6. Operating Procedure:**

1. **Write Functions to Measure Execution Time**:
   * Define functions that simulate work and measure their execution time.
2. function functionOne() {
3. for (let i = 0; i < 1000000; i++) {}
4. }
5. function functionTwo() {
6. for (let i = 0; i < 5000000; i++) {}
7. }
8. **Measure Execution Time**:
   * Use console.time() before the function and console.timeEnd() after to log the execution time.
9. console.time('functionOne');
10. functionOne();
11. console.timeEnd('functionOne'); // Outputs the time taken by functionOne
12. console.time('functionTwo');
13. functionTwo();
14. console.timeEnd('functionTwo'); // Outputs the time taken by functionTwo
15. **Run the Program**:
    * Save the file and run it using Node.js in the terminal:
16. node execution-time.js

#### **7. Precautions and/or Troubleshooting:**

* Ensure that the function names are correctly matched with console.time() and console.timeEnd().
* Functions should not run indefinitely, as that would cause the measurement to be inaccurate.

#### **8. Observations:**

* The execution times for functionOne and functionTwo will be displayed in the console, showing the time taken by each function in milliseconds.

#### **9. Calculations & Analysis:**

* The console.time() and console.timeEnd() functions will output the time taken by each function.
* The difference in execution times will help you understand the performance of each function.

#### **10. Result & Interpretation:**

* The time output will indicate the performance of each function. You can compare which function executes faster based on the number of iterations or operations.

#### **11. Follow-up Questions:**

1. What other methods can you use to measure execution time in Node.js?
2. How can you optimize functions to reduce execution time?

#### **12. Extension and Follow-up Activities (if applicable):**

* Experiment with larger datasets or more complex functions to see how the execution time increases.
* Implement benchmarking for asynchronous functions using Promises or async/await.

#### **13. Assessments:**

* Submit the program with accurate execution time logs.
* Answer the follow-up questions about optimizing function execution times.

#### **14. Suggested Reading:**

* [Node.js Documentation](https://nodejs.org/en/docs/)
* *JavaScript: The Good Parts* by Douglas Crockford

### **Assignment 6: Write a Node.js Program to Explain Asynchronous and Synchronous Functions with Examples**

#### **1. Aim/Purpose of the Experiment:**

* To understand and demonstrate the difference between synchronous and asynchronous functions in Node.js.

#### **2. Learning Outcomes:**

* Learn how synchronous functions block the execution and return results sequentially.
* Understand how asynchronous functions allow non-blocking execution and handle tasks concurrently.

#### **3. Prerequisites:**

* Basic knowledge of JavaScript and Node.js.
* Familiarity with callback functions, promises, and async/await in Node.js.

#### **4. Materials/Equipment/Apparatus / Devices/Software Required:**

* Node.js installed on your system.
* Code editor (e.g., VS Code, Sublime Text).
* Terminal/command prompt for running the Node.js program.

#### **5. Introduction and Theory:**

* **Synchronous Functions**: In synchronous programming, each task runs one after the other, blocking the execution of the next task until the current one completes. It can cause delays, especially with time-consuming operations.
* **Asynchronous Functions**: Asynchronous programming allows tasks to be executed independently without blocking the execution of other tasks. Functions in Node.js often use callbacks, promises, or async/await to handle operations asynchronously, like I/O tasks, file reading, or API calls.

#### **6. Operating Procedure:**

1. **Synchronous Example**:
   * A synchronous function waits for a task to complete before moving on to the next one.
2. console.log("Start");
3. function syncFunction() {
4. let i = 0;
5. while (i < 1000000000) {
6. i++; // Simulate a time-consuming task
7. }
8. console.log("Sync Function Completed");
9. }
10. syncFunction();
11. console.log("End");

**Explanation**:

* + The program first prints "Start".
  + Then, it runs the synchronous function syncFunction(), which takes time to complete and blocks further execution until it finishes.
  + After the function completes, it prints "Sync Function Completed" and finally "End".

1. **Asynchronous Example**:
   * Asynchronous functions allow other tasks to execute while waiting for a task to complete.
2. console.log("Start");
3. function asyncFunction(callback) {
4. setTimeout(() => {
5. console.log("Async Function Completed");
6. callback();
7. }, 2000); // Simulate an async task (2 seconds delay)
8. }
9. asyncFunction(() => {
10. console.log("End");
11. });
12. console.log("Async function started...");

**Explanation**:

* + The program first prints "Start".
  + The asyncFunction() is executed, but it doesn't block the subsequent code execution. The message "Async function started..." is printed immediately after, even before the asynchronous function completes.
  + After a 2-second delay (simulated by setTimeout()), it prints "Async Function Completed" and finally prints "End" once the callback is called.

#### **7. Precautions and/or Troubleshooting:**

* Ensure that the asynchronous function's callback is correctly defined to avoid undefined behavior.
* Be cautious about the execution order in asynchronous functions, especially when multiple async functions are involved.

#### **8. Observations:**

* **Synchronous** functions block the execution of subsequent code until they complete.
* **Asynchronous** functions allow the program to continue executing other code while waiting for the task to finish.

#### **9. Calculations & Analysis:**

* **Synchronous Execution**: Takes time to complete a task, affecting overall performance in tasks like I/O operations.
* **Asynchronous Execution**: Non-blocking, allows better performance, especially when dealing with time-consuming operations.

#### **10. Result & Interpretation:**

* The synchronous function blocks subsequent code execution, leading to delays.
* The asynchronous function demonstrates non-blocking behavior, allowing other tasks to execute while waiting for a task to finish.

#### **11. Follow-up Questions:**

1. How does Node.js handle I/O operations asynchronously?
2. What is the difference between callbacks, promises, and async/await in Node.js?
3. How can we handle multiple asynchronous operations in sequence?

#### **12. Extension and Follow-up Activities (if applicable):**

* Experiment with using promises and async/await for handling asynchronous code.
* Compare the performance of synchronous vs asynchronous functions in real-world applications (e.g., file I/O, database queries).

#### **13. Assessments:**

* Write a Node.js program that uses both synchronous and asynchronous functions to perform file operations and compare their execution times.
* Answer follow-up questions related to asynchronous patterns and handling errors.

#### **14. Suggested Reading:**

* [Node.js Documentation](https://nodejs.org/en/docs/)
* *JavaScript: The Good Parts* by Douglas Crockford
* *Node.js Design Patterns* by Mario Casciaro

### **Assignment 7: Core Path and FS Modules in Node.js**

#### **Aim/Purpose:**

* To understand and use the path module for manipulating file paths.
* To explore the fs module for basic file system operations like reading, writing, and deleting files.

#### **Learning Outcomes:**

* Learn to handle file paths and file operations using Node.js.

#### **Prerequisites:**

* Basic Node.js and JavaScript knowledge.

#### **Materials Required:**

* Node.js installed.
* Text editor (VS Code).
* Command Line/Terminal.

#### **Introduction and Theory:**

1. **Path Module**:
   * Helps work with file and directory paths across platforms.
   * Common methods: path.join(), path.basename(), path.dirname(), path.resolve().
2. **FS Module**:
   * Used for file system operations like reading, writing, and deleting files.

#### **Operating Procedure:**

**a) Path Module Example:**

const path = require('path');

const filePath = path.join('user', 'documents', 'file.txt');

console.log(path.basename(filePath)); // 'file.txt'

**b) Path & FS Modules Example:**

const path = require('path');

const fs = require('fs');

const filePath = path.join(\_\_dirname, 'sample.txt');

fs.writeFile(filePath, 'Hello, Node.js!', () => {

fs.readFile(filePath, 'utf8', (err, data) => {

console.log(data); // 'Hello, Node.js!'

fs.unlink(filePath, () => console.log('File deleted'));

});

});

#### **Observations:**

* The path module helps create and manipulate file paths.
* The fs module allows reading and modifying files.

#### **Result & Interpretation:**

* Successfully handled file paths and file operations using path and fs.

#### **Follow-up Questions:**

1. How does path.join() help in cross-platform development?
2. What is the difference between asynchronous and synchronous file operations in fs?

#### **Suggested Reading:**

* [Node.js Path Module Documentation](https://nodejs.org/api/path.html)
* [Node.js FS Module Documentation](https://nodejs.org/api/fs.html)

### **Assignment 8: Demonstrating the Use of Buffers in Node.js**

#### **Aim/Purpose:**

* To understand and demonstrate the use of Buffers in Node.js.
* Learn how to create buffers, write data into them, and slice them using different methods.

#### **Learning Outcomes:**

* Understand how to work with Buffers in Node.js.
* Learn to use Buffer.from(), Buffer.alloc(), write(), and slice() methods.

#### **Prerequisites:**

* Basic knowledge of Node.js and JavaScript.

#### **Materials Required:**

* Node.js installed.
* Text editor (VS Code).
* Command Line/Terminal.

#### **Introduction and Theory:**

* **Buffer in Node.js**: A Buffer is a raw memory allocation outside of the V8 JavaScript engine. It is used to handle binary data. Buffers are especially useful when dealing with I/O operations (e.g., reading files, network operations).
* **Methods**: Buffer.from(): Creates a buffer from a string, array, or array-like object. Buffer.alloc(): Creates a buffer of a specified size and initializes it with zeroes. write(): Writes data to a buffer at a specified offset. slice(): Extracts a portion of a buffer.

#### **Operating Procedure:**

// Creating a buffer with Buffer.from()

const bufferFrom = Buffer.from('Hello, Node.js!');

console.log(bufferFrom); // Displays the buffer representation

// Creating a buffer with Buffer.alloc()

const bufferAlloc = Buffer.alloc(10);

console.log(bufferAlloc); // Creates an empty buffer of size 10

// Writing data to a buffer using write()

const bufferWrite = Buffer.alloc(20);

bufferWrite.write('Node.js Buffers!');

console.log(bufferWrite.toString()); // 'Node.js Buffers!'

// Slicing a buffer using slice()

const slicedBuffer = bufferFrom.slice(0, 5);

console.log(slicedBuffer.toString()); // 'Hello'

#### **Observations:**

* The Buffer.from() creates a buffer from a string.
* The Buffer.alloc() creates a buffer of a specified size.
* The write() method writes data into the buffer, and slice() extracts a portion of the buffer.

#### **Result & Interpretation:**

* Successfully demonstrated how to create, write to, and slice Buffers in Node.js.

#### **Follow-up Questions:**

1. What is the difference between Buffer.from() and Buffer.alloc()?
2. How does slicing a buffer affect the original buffer?

#### **Suggested Reading:**

* [Node.js Buffer Documentation](https://nodejs.org/api/buffer.html)

### **Assignment 9: Describing Local Modules in Node.js**

#### **Aim/Purpose of the Experiment:**

* To learn how to create and use local modules in Node.js.
* To understand how to export and import functions and objects between modules.

#### **Learning Outcomes:**

* Understand how to create local modules in Node.js.
* Learn how to export functions and objects from one module and use them in another.

#### **Prerequisites:**

* Basic knowledge of Node.js and JavaScript.
* Understanding of how to use require() to import modules.

#### **Materials/Equipment/Apparatus / Devices/Software required:**

* Node.js installed.
* Code editor (e.g., VS Code).
* Terminal/Command Line.

#### **Introduction and Theory:**

* **Local Modules** in Node.js are custom modules that you create to organize your code.
* You can export functions, objects, or other data from a module using module.exports.
* Local modules are imported using require().

#### **Operating Procedure:**

##### a) **Local Modules with Two Functions:**

1. Create a file mathFunctions.js that contains two functions:

// mathFunctions.js

function add(a, b) {

return a + b;

}

function multiply(a, b) {

return a \* b;

}

module.exports = { add, multiply };

1. Create a file main.js to import and use the functions from mathFunctions.js:

// main.js

const mathFunctions = require('./mathFunctions');

console.log('Addition:', mathFunctions.add(5, 3)); // 8

console.log('Multiplication:', mathFunctions.multiply(5, 3)); // 15

##### b) **Local Modules with an Object:**

1. Create a file person.js that exports an object:

// person.js

const person = {

name: 'John Doe',

age: 30,

greet: function() {

return `Hello, my name is ${this.name} and I am ${this.age} years old.`;

}

};

module.exports = person;

1. Create a file mainPerson.js to import and use the object:

// mainPerson.js

const person = require('./person');

console.log(person.greet()); // "Hello, my name is John Doe and I am 30 years old."

#### **Observations:**

* Successfully created local modules with functions and objects.
* Used module.exports to export and require() to import the modules.

#### **Result & Interpretation:**

* Demonstrated how to create, export, and import local modules in Node.js with functions and objects.

#### **Follow-up Questions:**

1. What is the difference between exporting a function and an object in a local module?
2. Can you have multiple functions and variables in one module?

#### **Suggested Reading:**

* [Node.js Modules Documentation](https://nodejs.org/api/modules.html)

### **Assignment 10: Create a package.json file for a Node.js project and describe its setup with an external package like dotenv**

#### **Aim/Purpose of the Experiment:**

* To create a package.json file for a Node.js project.
* To demonstrate the setup and usage of the dotenv package for managing environment variables in Node.js.

#### **Learning Outcomes:**

* Understand the structure and purpose of the package.json file.
* Learn to install and configure external packages (like dotenv) in Node.js.
* Understand how to manage environment variables securely with .env files.

#### **Prerequisites:**

* Basic knowledge of Node.js and JavaScript.
* Familiarity with npm (Node Package Manager) commands and file structure.

#### **Materials/Equipment/Apparatus / Devices/Software required:**

* Node.js installed on the machine.
* A code editor (e.g., Visual Studio Code).
* Command Line or Terminal.
* Internet connection for package installation.

#### **Introduction and Theory:**

* **package.json**: The package.json file is the heart of a Node.js project. It includes metadata about the project, such as its name, version, description, dependencies, and scripts. It helps manage the project's dependencies and other configurations.
* **dotenv**: The dotenv package loads environment variables from a .env file into process.env. It is used to store sensitive information (like API keys, passwords) that should not be hard-coded into the application code.

#### **Operating Procedure:**

1. **Initialize a Node.js project**:
   * Open the terminal and navigate to your project folder.
   * Run npm init or npm init -y to generate a package.json file.
2. **Install the dotenv package**:
   * Run the following command to install dotenv:
3. npm install dotenv
4. **Create a .env file**:
   * In your project root directory, create a .env file to store environment variables:
5. DB\_HOST=localhost
6. DB\_USER=root
7. DB\_PASS=mysecretpassword
8. **Use dotenv in your project**:
   * Create an index.js file in your project directory and add the following code:
9. require('dotenv').config();
10. console.log('Database Host:', process.env.DB\_HOST);
11. console.log('Database User:', process.env.DB\_USER);
12. console.log('Database Password:', process.env.DB\_PASS);
13. **Run the application**:
    * In the terminal, run the following command:
14. node index.js
    * This will output the environment variables from the .env file.

#### **Precautions and/or Troubleshooting:**

* Ensure that the .env file is in the root directory of the project.
* Double-check that the environment variables in the .env file are correctly formatted (e.g., no spaces around =).
* If the dotenv package is not installed, run npm install dotenv again.

#### **Observations:**

* The package.json file was successfully created.
* The dotenv package loaded the environment variables from the .env file.
* The application correctly printed the environment variables.

#### **Result & Interpretation:**

* The experiment successfully demonstrated how to manage project metadata with package.json and how to use the dotenv package for managing sensitive configuration data in a Node.js application.

#### **Follow-up Questions:**

1. What is the role of the package.json file in a Node.js project?
2. Why is it important to keep sensitive data like API keys in environment variables rather than in the code?
3. What does the .env file typically contain?

#### **Extension and Follow-up Activities:**

* Explore other packages that are commonly used in Node.js projects, such as express for web development.
* Implement error handling for missing environment variables and improve security by using production-ready environment management.

#### **Assessments:**

* Ability to create a package.json file and add dependencies.
* Demonstrating understanding of the dotenv package and its usage for managing environment variables.

#### **Suggested Reading:**

* [Node.js Documentation](https://nodejs.org/en/docs/)
* [dotenv Documentation](https://www.npmjs.com/package/dotenv)

### **Assignment 11: Create a Server Using the HTTP Core Module and Handle Routes with Different HTTP Methods**

#### **Learning Outcomes:**

* Understand how to create a basic HTTP server in Node.js using the http core module.
* Learn to handle different HTTP methods (GET, POST, PUT, DELETE) and route requests based on the URL.

#### **Prerequisites:**

* Basic knowledge of Node.js and JavaScript.
* Familiarity with HTTP request methods (GET, POST, PUT, DELETE).

#### **Materials/Equipment/Apparatus / Devices/Software required:**

* Node.js installed on the machine.
* Code editor (e.g., Visual Studio Code).
* Command Line or Terminal.

#### **Introduction and Theory:**

* **http Module**: The http module in Node.js is used to create a web server and handle requests and responses. It provides a simple API for handling HTTP requests and sending responses.
* **HTTP Methods**:
  + **GET**: Used to retrieve data from the server.
  + **POST**: Used to send data to the server, typically for creating new resources.
  + **PUT**: Used to update existing data on the server.
  + **DELETE**: Used to remove data from the server.
* **Routing**: Routing is the mechanism that allows a server to respond differently based on the URL and the HTTP method of the request.

#### **Operating Procedure:**

1. **Create a basic HTTP server**:
   * In a file called server.js, create the following code to start an HTTP server:
2. const http = require('http');
3. const server = http.createServer((req, res) => {
4. res.setHeader('Content-Type', 'text/plain');
5. res.statusCode = 200;
6. res.end('Hello, World!');
7. });
8. server.listen(3000, () => {
9. console.log('Server running at http://localhost:3000/');
10. });
11. **Handle Routes and Different HTTP Methods**:
    * Modify the server to handle specific routes and different HTTP methods.
12. const http = require('http');
13. const server = http.createServer((req, res) => {
14. const url = req.url;
15. const method = req.method;
16. if (url === '/users' && method === 'GET') {
17. res.setHeader('Content-Type', 'application/json');
18. res.statusCode = 200;
19. res.end(JSON.stringify([{ name: 'Alice' }, { name: 'Bob' }]));
20. } else if (url === '/users' && method === 'POST') {
21. let body = '';
22. req.on('data', chunk => {
23. body += chunk;
24. });
25. req.on('end', () => {
26. const user = JSON.parse(body);
27. res.statusCode = 201;
28. res.end(`User ${user.name} added.`);
29. });
30. } else if (url === '/users' && method === 'PUT') {
31. res.statusCode = 200;
32. res.end('User updated');
33. } else if (url === '/users' && method === 'DELETE') {
34. res.statusCode = 200;
35. res.end('User deleted');
36. } else {
37. res.statusCode = 404;
38. res.end('Not Found');
39. }
40. });
41. server.listen(3000, () => {
42. console.log('Server running at http://localhost:3000/');
43. });
44. **Test the Server**:
    * Use tools like Postman or cURL to send different HTTP requests to the server:
      + **GET** request: http://localhost:3000/users
      + **POST** request: Send JSON body (e.g., { "name": "Charlie" })
      + **PUT** request: Send a request to /users for updating.
      + **DELETE** request: Send a request to /users to delete.

#### **Precautions and/or Troubleshooting:**

* Ensure the server is running on the correct port (3000 in this case).
* Double-check the request method and URL while testing using cURL or Postman.
* Handle JSON data properly for POST and PUT requests (parse and stringify).

#### **Observations:**

* The server correctly handles GET, POST, PUT, and DELETE requests.
* Responses are returned according to the requested method and route.

#### **Calculations & Analysis:**

* Analysis is not applicable as this is a simple server setup.

#### **Result & Interpretation:**

* The program successfully creates an HTTP server and handles different HTTP methods based on routes. The server responds differently based on the request method (GET, POST, PUT, DELETE) and the URL (/users).

#### **Follow-up Questions:**

1. How would you handle a non-existent route in Node.js?
2. What is the purpose of the req.on('data') event in the POST method?
3. Why is it important to set the correct Content-Type header in HTTP responses?

#### **Extension and Follow-up Activities:**

* Implement more complex routing by using external frameworks like Express.
* Add support for handling different content types (e.g., XML, form data).
* Explore adding middleware for logging and error handling in the server.

#### **Assessments:**

* Ability to create and configure an HTTP server in Node.js.
* Understanding of handling different HTTP methods and routing.

#### **Suggested Reading:**

* [Node.js HTTP Module Documentation](https://nodejs.org/dist/latest-v14.x/docs/api/http.html)
* [MDN Web Docs: HTTP Methods](https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods)

### **Assignment 12: Perform CRUD Operations on a File Using the FS Core Module in Node.js**

#### **Learning Outcomes:**

* Understand how to perform basic file operations (Create, Read, Update, Delete) in Node.js using the fs (File System) core module.
* Learn how to manipulate files by reading from, writing to, updating, and deleting files.

#### **Prerequisites:**

* Basic knowledge of Node.js and JavaScript.
* Familiarity with the fs (File System) module in Node.js.

#### **Materials/Equipment/Apparatus / Devices/Software required:**

* Node.js installed on the machine.
* Code editor (e.g., Visual Studio Code).
* Command Line or Terminal.

#### **Introduction and Theory:**

* The **fs module** in Node.js provides an API for interacting with the file system, allowing you to create, read, update, and delete files.
* **CRUD Operations**:
  + **Create**: Writing data to a file (using fs.writeFile() or fs.appendFile()).
  + **Read**: Reading data from a file (using fs.readFile()).
  + **Update**: Modifying existing data in a file (using fs.appendFile() or fs.writeFile()).
  + **Delete**: Deleting a file (using fs.unlink()).

#### **Operating Procedure:**

1. **Create a New File**:
   * Use fs.writeFile() to create a new file and write data to it.
2. const fs = require('fs');
3. fs.writeFile('example.txt', 'Hello, Node.js!', (err) => {
4. if (err) {
5. console.log('Error writing file:', err);
6. } else {
7. console.log('File created and data written.');
8. }
9. });
10. **Read the File**:
    * Use fs.readFile() to read the contents of a file.
11. fs.readFile('example.txt', 'utf8', (err, data) => {
12. if (err) {
13. console.log('Error reading file:', err);
14. } else {
15. console.log('File contents:', data);
16. }
17. });
18. **Update the File**:
    * Use fs.appendFile() to append data to the file.
19. fs.appendFile('example.txt', '\nAppended text in Node.js.', (err) => {
20. if (err) {
21. console.log('Error updating file:', err);
22. } else {
23. console.log('Data appended successfully.');
24. }
25. });
26. **Delete the File**:
    * Use fs.unlink() to delete the file.
27. fs.unlink('example.txt', (err) => {
28. if (err) {
29. console.log('Error deleting file:', err);
30. } else {
31. console.log('File deleted.');
32. }
33. });

#### **Precautions and/or Troubleshooting:**

* Ensure that the file paths are correct and the necessary permissions are granted.
* Handle errors gracefully, especially for operations like reading or deleting files that may not exist.

#### **Observations:**

* The file is successfully created, data is written to it, updated, and deleted without issues.
* The program outputs the expected messages on success or error.

#### **Calculations & Analysis:**

* N/A. This assignment does not require calculations.

#### **Result & Interpretation:**

* The program successfully performs CRUD operations on a file using the fs module in Node.js. It demonstrates how to create, read, update, and delete files, which are basic file operations in a server environment.

#### **Follow-up Questions:**

1. How would you handle cases where the file does not exist when trying to read or delete it?
2. What is the difference between fs.writeFile() and fs.appendFile() in terms of file handling?
3. How can you handle large files in Node.js?

#### **Extension and Follow-up Activities:**

* Explore synchronous versions of the file operations (e.g., fs.readFileSync(), fs.writeFileSync()).
* Implement error handling for missing files and other edge cases.
* Create a more advanced program that allows user input for CRUD operations (e.g., using command-line arguments).

#### **Assessments:**

* Ability to perform basic file system operations using the fs module.
* Understanding of asynchronous file handling in Node.js.

#### **Suggested Reading:**

* [Node.js File System Documentation](https://nodejs.org/dist/latest-v14.x/docs/api/fs.html)
* [MDN Web Docs: File Handling](https://developer.mozilla.org/en-US/docs/Web/API/File)

### **Assignment 13: Create a Node.js Project for a Calculator with Multiple Routes**

#### **Learning Outcomes:**

* Understand how to create a simple web application with multiple routes using Node.js.
* Learn how to handle user input through forms, perform operations (e.g., sum calculation), and display results dynamically.

#### **Prerequisites:**

* Basic knowledge of Node.js and Express framework.
* Familiarity with HTML forms and HTTP methods (GET and POST).

#### **Materials/Equipment/Apparatus / Devices/Software required:**

* Node.js installed on the system.
* Express.js library.
* Code editor (e.g., Visual Studio Code).
* Command Line or Terminal.

#### **Introduction and Theory:**

* **Node.js** allows the creation of web servers that can handle multiple routes for different pages of a web application.
* **Express.js** is a framework built on top of Node.js, making it easier to handle routes, middleware, and server-side logic.
* This assignment focuses on creating a basic calculator that allows the user to input two numbers, calculates their sum, and displays the result.

#### **Operating Procedure:**

1. **Initialize the Project:**
   * Create a new directory for the project and initialize a Node.js project.
2. mkdir calculator
3. cd calculator
4. npm init -y
5. npm install express
6. **Create Server with Routes:**
   * Create a server.js file for the Node.js application.
   * Define the routes for the home page, calculator page, and result page.

**server.js:**

const express = require('express');

const app = express();

const port = 3000;

// Middleware to parse form data

app.use(express.urlencoded({ extended: true }));

// Home page

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

res.send('<h1>Welcome to the Calculator App</h1><a href="/calculator">Go to Calculator</a>');

});

// Calculator page with form

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

res.send(`

<h1>Calculator</h1>

<form action="/calculate-result" method="POST">

<label for="num1">Number 1:</label><br>

<input type="text" id="num1" name="num1"><br><br>

<label for="num2">Number 2:</label><br>

<input type="text" id="num2" name="num2"><br><br>

<input type="submit" value="Calculate">

</form>

`);

});

// Calculation result

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

const { num1, num2 } = req.body;

const sum = addNumbers(Number(num1), Number(num2));

res.send(`<h1>Result</h1><p>The sum is: ${sum}</p><a href="/calculator">Back to Calculator</a>`);

});

// Addition module function

function addNumbers(a, b) {

return a + b;

}

// Start server

app.listen(port, () => {

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

});

1. **Running the Server:**
   * Run the server using the following command:
2. node server.js
   * Open a web browser and go to http://localhost:3000/ to see the home page. Click the link to navigate to the calculator page.
3. **Calculator Functionality:**
   * On the /calculator page, enter two numbers and click the "Calculate" button. It will redirect to the /calculate-result page, where the sum of the two numbers is displayed.

#### **Precautions and/or Troubleshooting:**

* Ensure that all routes are defined correctly in the server.js file.
* Check that the form method is POST when submitting data.
* Ensure Express is correctly installed using npm install express.

#### **Observations:**

* The application successfully creates routes and serves HTML pages.
* The form on the /calculator page collects user input and performs calculations when submitted.

#### **Calculations & Analysis:**

* The addition operation is performed by a simple JavaScript function addNumbers() that adds the two input values.

#### **Result & Interpretation:**

* The project demonstrates how to handle multiple routes, accept form input, and process data asynchronously in Node.js.
* It shows how Node.js and Express can be used to build simple web applications.

#### **Follow-up Questions:**

1. How can you extend this project to support other operations (like subtraction, multiplication, and division)?
2. How can you validate the input to ensure that users enter only numbers?
3. How would you handle cases where the user submits an empty form?

#### **Extension and Follow-up Activities:**

* Add functionality for additional arithmetic operations like multiplication and division.
* Implement input validation and error handling (e.g., check if the inputs are valid numbers).
* Use templating engines like **EJS** or **Pug** to render the pages instead of sending raw HTML.

#### **Assessments:**

* Ability to create a Node.js server with multiple routes.
* Understanding of handling form data using Express.
* Knowledge of modularizing functions and processing user input.

#### **Suggested Reading:**

* [Node.js Documentation](https://nodejs.org/en/docs/)
* [Express.js Documentation](https://expressjs.com/)
* [MDN Web Docs on Forms](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/form)

### **Node.js Assignment 14: Create a Node.js Project Using Express to Respond with HTML and JSON**

#### **Learning Outcomes:**

* Understand how to use Express to create a Node.js web server.
* Learn how to serve different content types, including HTML and JSON, based on the route.
* Gain experience in serving different HTML files at different routes in a Node.js project.

#### **Prerequisites:**

* Basic knowledge of Node.js and the Express framework.
* Familiarity with routing in Express.
* Basic understanding of how to handle HTML and JSON responses in Express.

#### **Materials/Equipment/Apparatus / Devices/Software required:**

* Node.js installed on the system.
* Express.js library.
* Code editor (e.g., Visual Studio Code).
* Command Line or Terminal.

#### **Introduction and Theory:**

* **Node.js** with **Express** allows the creation of web servers that can handle HTTP requests and responses.
* **HTML** is served when the client requests a webpage, while **JSON** is typically used to send data from the server to the client in a machine-readable format.
* The goal of this assignment is to create a Node.js project using Express that serves both HTML files and JSON data at different routes.

#### **Operating Procedure:**

1. **Initialize the Project:**
   * Create a new directory and initialize a Node.js project.
2. mkdir express-html-json
3. cd express-html-json
4. npm init -y
5. npm install express
6. **Create the Server with Routes:**
   * Create a server.js file.
   * Set up routes to serve HTML files and JSON responses.

**server.js:**

const express = require('express');

const app = express();

const path = require('path');

const port = 3000;

// Middleware to serve static files (if needed)

app.use(express.static('public'));

// Route to serve HTML file

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

res.sendFile(path.join(\_\_dirname, 'views', 'home.html'));

});

// Route to serve another HTML file

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

res.sendFile(path.join(\_\_dirname, 'views', 'about.html'));

});

// Route to serve JSON response

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

const data = {

name: 'Node.js Example',

type: 'Web Server',

features: ['HTML Serving', 'JSON Responses']

};

res.json(data);

});

// Start the server

app.listen(port, () => {

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

});

1. **Create HTML Files:**
   * Create a views folder and add two HTML files: home.html and about.html.

**views/home.html:**

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

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

<title>Home Page</title>

</head>

<body>

<h1>Welcome to the Home Page</h1>

<a href="/about">Go to About Page</a>

</body>

</html>

**views/about.html:**

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

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

<title>About Page</title>

</head>

<body>

<h1>About Us</h1>

<p>This is the About Page.</p>

<a href="/">Go to Home Page</a>

</body>

</html>

1. **Running the Server:**
   * Run the server using the command:
2. node server.js
3. **Testing:**
   * Open a browser and go to:
     + http://localhost:3000/ to see the home page.
     + http://localhost:3000/about to see the about page.
     + http://localhost:3000/data to see the JSON data.

#### **Precautions and/or Troubleshooting:**

* Ensure the views folder is created in the project directory and contains the HTML files.
* Check if the paths in res.sendFile() are correct.
* Ensure that the server is running and accessible at the specified port (3000).

#### **Observations:**

* The application successfully serves HTML files at specific routes (/ and /about).
* The /data route correctly returns JSON data.

#### **Calculations & Analysis:**

* The JSON data is sent in response to the /data route, while HTML files are served in response to the other routes.
* No calculations are involved in this assignment, but understanding routing and handling different content types is key.

#### **Result & Interpretation:**

* The project demonstrates how to use Express to serve both HTML files and JSON data depending on the route requested.
* This basic server can be extended to serve more complex pages or handle more types of requests.

#### **Follow-up Questions:**

1. How can you modify this project to serve additional HTML files for new routes?
2. How would you handle errors or invalid routes in this project (e.g., 404 Not Found)?
3. How can you modify this server to respond with XML instead of JSON?

#### **Extension and Follow-up Activities:**

* Add more routes and corresponding HTML pages.
* Integrate a template engine like **EJS** or **Pug** to dynamically render HTML based on data from the server.
* Use middleware to handle errors and return proper status codes for different types of responses.

#### **Assessments:**

* Ability to create a Node.js server with Express.
* Understanding of how to serve HTML and JSON responses on different routes.
* Familiarity with how to organize project files and use Express routing.

#### **Suggested Reading:**

* [Node.js Documentation](https://nodejs.org/en/docs/)
* [Express.js Documentation](https://expressjs.com/)
* [MDN Web Docs on HTML](https://developer.mozilla.org/en-US/docs/Web/HTML)
* [MDN Web Docs on JSON](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON)

### **Node.js Assignment 15: Extend the Express Project**

#### **Learning Outcomes:**

* Understand how to extend Express to serve different HTML files at different routes.
* Learn how to process user input through forms and perform calculations (e.g., adding two numbers).
* Implement dynamic parameters in the URL for routing.
* Use application-level middleware for tasks like logging or error handling.

#### **Prerequisites:**

* Basic knowledge of Node.js and Express.
* Familiarity with HTML forms and basic JavaScript operations.
* Understanding of middleware in Express and how it is applied to requests.

#### **Materials/Equipment/Apparatus / Devices/Software required:**

* Node.js installed on the system.
* Express.js library.
* Code editor (e.g., Visual Studio Code).
* Command Line or Terminal.

#### **Introduction and Theory:**

* **Express** is a minimal and flexible Node.js web application framework that allows for easy routing, middleware support, and handling HTTP requests.
* **Middleware** in Express allows you to handle HTTP requests at a central place before they reach the route handlers.
* **Dynamic routing** allows you to capture values in the URL and use them within your application.
* We will build an Express app to serve HTML files, accept user input from a form, perform calculations (addition), and handle dynamic routing and middleware tasks.

#### **Operating Procedure:**

1. **Initialize the Project:**
   * Create a new directory and initialize a Node.js project if not already done.
2. mkdir express-calculator
3. cd express-calculator
4. npm init -y
5. npm install express
6. **Create the Server with Routes and Middleware:**
   * Create a server.js file to set up routes, dynamic parameters, form handling, and middleware.

**server.js:**

const express = require('express');

const app = express();

const path = require('path');

const port = 3000;

// Middleware for logging requests

app.use((req, res, next) => {

console.log(`${req.method} request for ${req.url}`);

next(); // Move to the next middleware or route handler

});

// Middleware for parsing form data

app.use(express.urlencoded({ extended: true }));

// Route to serve home page

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

res.sendFile(path.join(\_\_dirname, 'views', 'home.html'));

});

// Route to serve calculator form

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

res.sendFile(path.join(\_\_dirname, 'views', 'calculator.html'));

});

// Route to handle form submission and perform calculation

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

const num1 = parseInt(req.body.num1);

const num2 = parseInt(req.body.num2);

const result = num1 + num2;

res.send(`<h1>Result: ${result}</h1><a href="/calculator">Back to Calculator</a>`);

});

// Dynamic route with parameter

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

const name = req.params.name;

res.send(`<h1>Hello, ${name}!</h1>`);

});

// Error handling middleware

app.use((req, res) => {

res.status(404).send('<h1>404 - Page Not Found</h1>');

});

// Start the server

app.listen(port, () => {

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

});

1. **Create HTML Files for the Routes:**
   * Create a views folder with two HTML files: home.html and calculator.html.

**views/home.html:**

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

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

<title>Home Page</title>

</head>

<body>

<h1>Welcome to the Calculator App</h1>

<a href="/calculator">Go to Calculator</a>

</body>

</html>

**views/calculator.html:**

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

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

<title>Calculator</title>

</head>

<body>

<h1>Calculator</h1>

<form action="/calculate" method="POST">

<label for="num1">Number 1:</label>

<input type="number" name="num1" required>

<br>

<label for="num2">Number 2:</label>

<input type="number" name="num2" required>

<br>

<button type="submit">Calculate</button>

</form>

</body>

</html>

1. **Run the Server:**
   * Run the server using the command:
2. node server.js
3. **Test the Application:**
   * Open a browser and visit:
     + http://localhost:3000/ to see the home page.
     + http://localhost:3000/calculator to access the calculator form.
     + Submit the form to see the result of the addition.
     + http://localhost:3000/greet/John to see a dynamic greeting with the name in the URL.

#### **Precautions and/or Troubleshooting:**

* Ensure that the views folder and the HTML files (home.html, calculator.html) are in the correct path.
* Verify the proper use of middleware, such as URL encoding and logging.
* Make sure the server is running without errors, and routes are correctly configured.

#### **Observations:**

* The Express application serves HTML files at different routes.
* The calculator form processes user input, performs the addition, and displays the result.
* The dynamic route responds with a greeting using the name parameter from the URL.

#### **Calculations & Analysis:**

* The form submission sends two numbers to the server, which adds them and sends back the result to the user.
* The dynamic route captures the name parameter from the URL and sends a personalized response.

#### **Result & Interpretation:**

* The Express app successfully serves HTML files at different routes, processes user input, and performs a calculation.
* The dynamic routing feature works by taking parameters from the URL and responding accordingly.
* Middleware is used for logging and error handling in the application.

#### **Follow-up Questions:**

1. How can you modify the project to handle more complex calculations?
2. How would you add validation to ensure that only valid numbers are entered in the form?
3. How can you use middleware to log additional request information, such as request headers or user agents?

#### **Extension and Follow-up Activities:**

* Add additional calculations such as subtraction, multiplication, and division.
* Use **session** or **cookies** to store and track user input and results across different pages.
* Implement more dynamic routes that take other parameters (e.g., /calculator/operation/add).

#### **Assessments:**

* Ability to create dynamic routes with Express.
* Understanding how to handle form submissions and calculations.
* Familiarity with middleware and its application in logging and error handling.

#### **Suggested Reading:**

* [Express.js Documentation](https://expressjs.com/)
* [Node.js Documentation](https://nodejs.org/en/docs/)
* [MDN Web Docs on Forms](https://developer.mozilla.org/en-US/docs/Learn/Forms)
* [Middleware in Express](https://expressjs.com/en/guide/using-middleware.html)