Nodejs Assignments

1. Introduction to Node.js

Theory Assignment:

- Write an essay on the history and evolution of Node.js, discussing its architecture and key features.
- Compare Node.js with traditional server-side technologies like PHP and Java.

Practical Assignment:

- Install Node.js on your local machine and create a simple "Hello World" application. Include instructions for installation and running the application.

2. Node.js Environment Setup

Theory Assignment:

- Describe the role of npm (Node Package Manager) in Node.js development. Discuss common commands used in npm.

Practical Assignment:

- Create a project directory and initialize a new Node.js project using npm. Install at least two packages (e.g., Express, Nodemon) and demonstrate how to use them in your application.

3. Basics of JavaScript

1. Introduction to JavaScript

Theory Assignment:

- Write a brief essay on the history of JavaScript, its evolution, and its role in modern web development.

Practical Assignment:

- Set up a simple HTML file and include a JavaScript script. Display an alert box with a welcome message when the page loads.

2. JavaScript Syntax and Basics

Theory Assignment:

- Explain the basic syntax of JavaScript, including variables, data types, and operators. Discuss the differences between `let`, `const`, and `var`.

Practical Assignment:

- Create a JavaScript program that declares variables of different data types, performs basic arithmetic operations, and outputs the results to the console.

3. Control Flow Statements

Theory Assignment:

- Describe how control flow statements work in JavaScript, focusing on `if`, `else if`, `else`, and switch statements.

Assignment: Write a program that takes a student's score as input and outputs the corresponding grade based on the following criteria:

- A: 90-100
- B: 80-89
- C: 70-79
- D: 60-69
- F: Below 60

Instructions:

1. Prompt the user for their score.
2. Use `if`, `else if`, and `else` statements to determine the grade.
3. Display the grade to the user.

2. Age Classification

Assignment: Create a program that classifies a person based on their age group:

- Child: 0-12
- Teen: 13-19
- Adult: 20-64
- Senior: 65 and above

Instructions:

1. Prompt the user for their age.
2. Use `if`, `else if`, and `else` statements to determine the age group.
3. Output the corresponding age classification.

3. Even or Odd Number Checker

Assignment: Write a program that checks whether a given number is even or odd.

Instructions:

1. Prompt the user for a number.
2. Use the modulus operator (`%`) and an `if` statement to check if the number is even or odd.

3. Display the result to the user.

4. Days of the Week

Assignment: Create a program that prompts the user for a number between 1 and 7 and outputs the corresponding day of the week.

Instructions:

1. Use a `switch` statement to map numbers to days:
   - 1: Sunday
   - 2: Monday
   - 3: Tuesday
   - 4: Wednesday
   - 5: Thursday
   - 6: Friday
   - 7: Saturday
2. If the number is outside the range, display "Invalid input."

5. Temperature Converter

Assignment: Develop a program that converts a temperature from Celsius to Fahrenheit or vice versa based on user input.

Instructions:

1. Prompt the user to enter the temperature and the conversion type (C to F or F to C).
2. Use `if` statements to perform the appropriate conversion:
   - Celsius to Fahrenheit: `F = (C * 9/5) + 32`
   - Fahrenheit to Celsius: `C = (F - 32) * 5/9`
3. Display the converted temperature.

6. Simple Calculator

Assignment: Create a simple calculator that performs basic arithmetic operations: addition, subtraction, multiplication, and division.

Instructions:

1. Prompt the user to enter two numbers and an operator (+, -, *, /).
2. Use a `switch` statement to perform the appropriate operation based on the operator.
3. Display the result of the operation. Handle division by zero appropriately.

7. Largest of Three Numbers

Assignment: Write a program that takes three numbers as input and determines the largest among them.

Instructions:

1. Prompt the user to enter three numbers.
2. Use `if` and `else if` statements to compare the numbers and find the largest.
3. Display the largest number to the user.

8. Leap Year Checker

Assignment: Create a program that checks if a given year is a leap year.

Instructions:

1. Prompt the user for a year.
2. Use `if` statements to determine if the year is a leap year (divisible by 4 and not divisible by 100, or divisible by 400).
3. Display whether the year is a leap year or not.

9. Traffic Light Simulation

Assignment: Simulate a traffic light system where the user inputs the color of the traffic light and the program outputs the corresponding action.

Instructions:

1. Use a `switch` statement to handle the following inputs:
    - Red: Stop
    - Yellow: Caution
    - Green: Go
2. Handle invalid input by displaying "Invalid color."

10. Coin Flip Simulation

Assignment: Write a program that simulates flipping a coin. The program should output either "Heads" or "Tails."

Instructions:

1. Generate a random number (0 or 1) using `Math.random()`.
2. Use an `if` statement to determine if the result is "Heads" (0) or "Tails" (1).
3. Display the result to the user.

11. BMI Calculator

Assignment: Create a program that calculates the Body Mass Index (BMI) and categorizes the result.

Instructions:

1. Prompt the user for their weight (in kilograms) and height (in meters).
2. Calculate the BMI using the formula: `BMI = weight / (height * height)`.
3. Use `if`, `else if`, and `else` statements to categorize the BMI:

    - Underweight: BMI < 18.5
    - Normal weight: 18.5 ≤ BMI < 24.9
    - Overweight: 25 ≤ BMI < 29.9
    - Obesity: BMI ≥ 30
4. Display the BMI and category to the user.

12. Simple Quiz Application

Assignment: Build a simple quiz application where the user answers a question, and the program checks if the answer is correct.

Instructions:

1. Ask the user a question (e.g., "What is the capital of France?").
2. Use an `if` statement to check if the user's answer matches the correct answer ("Paris").
3. Display whether the user's answer is correct or incorrect.

4. Loops in JavaScript

Theory Assignment:

- Explain the different types of loops in JavaScript (`for`, `while`, and `do...while`). Discuss their use cases.

Practical Assignment:

- Create a program that uses a `for` loop to print the numbers from 1 to 100. Additionally, implement a `while` loop that does the same.

1. Sum of Numbers

Assignment:
Write a program that calculates the sum of all integers from 1 to a user-specified number.

Instructions:

1. Prompt the user for a positive integer.
2. Use a `for` loop to iterate through numbers from 1 to the specified number.
3. Calculate the sum and display the result.

2. Factorial Calculation

Assignment:
Create a program that calculates the factorial of a given number.

Instructions:

1. Prompt the user for a positive integer.
2. Use a `for` loop to calculate the factorial ($n! = n \times (n - 1) \times (n - 2) \times \ldots \times 1$).

3. Display the factorial to the user.

3. Fibonacci Sequence

Assignment:
Generate the Fibonacci sequence up to a specified number.

Instructions:

1. Prompt the user for the number of terms in the Fibonacci sequence.
2. Use a `for` loop to calculate and display the Fibonacci numbers.
3. Display the sequence in the console.

4. Multiplication Table

Assignment:
Write a program that displays the multiplication table for a given number.

Instructions:

1. Prompt the user for a number.
2. Use a `for` loop to display the multiplication table (1 to 10) for that number.
3. Format the output nicely in the console.

5. Count Down

Assignment:
Create a program that counts down from a specified number to zero.

Instructions:

1. Prompt the user for a starting number.
2. Use a `while` loop to count down to zero, displaying each number.
3. Display "Blast off!" when reaching zero.

6. Reverse a String

Assignment:
Write a program that reverses a given string using a loop.

Instructions:

1. Prompt the user for a string.
2. Use a `for` loop to build a new string in reverse order.
3. Display the reversed string.

7. Prime Number Checker

Assignment:
Create a program that checks and displays all prime numbers up to a user-specified limit.

Instructions:

1. Prompt the user for a positive integer.
2. Use a nested `for` loop to check for prime numbers.
3. Display all prime numbers found.

8. Number Pyramid

Assignment:
Generate a pyramid of numbers.

Instructions:

1. Prompt the user for the height of the pyramid.
2. Use nested `for` loops to create a pyramid shape with numbers.
3. Display the pyramid in the console.

9. Sum of Even and Odd Numbers

Assignment:
Write a program that calculates the sum of even and odd numbers from 1 to a user-defined limit.

Instructions:

1. Prompt the user for a positive integer.
2. Use a `for` loop to iterate through the numbers and separate the sums of even and odd numbers.
3. Display both sums.

10. Print Array Elements

Assignment:
Create a program that prints each element of an array using a loop.

Instructions:

1. Define an array with at least five different elements (numbers, strings, etc.).
2. Use a `for` loop to iterate through the array and print each element.
3. Format the output nicely.

11. Guess the Number Game

Assignment:
Develop a simple guessing game where the user has to guess a randomly generated number.

Instructions:

1. Generate a random number between 1 and 100.
2. Use a `do...while` loop to prompt the user for their guess until they guess correctly.
3. Provide hints (too high or too low) after each guess.

12. Print Odd Numbers in a Range

Assignment:
Write a program that prints all odd numbers in a given range.

Instructions:

1. Prompt the user for a starting and ending number.
2. Use a `for` loop to iterate through the range and print only the odd numbers.
3. Display the results in the console.

13. Vowel Counter

Assignment:
Create a program that counts the number of vowels in a given string.

Instructions:

1. Prompt the user for a string.
2. Use a `for` loop to iterate through each character in the string and count the vowels (a, e, i, o, u).
3. Display the total number of vowels found.

15. Character Frequency Counter

Assignment:
Write a program that counts the frequency of each character in a string.

Instructions:

1. Prompt the user for a string.
2. Use a loop to iterate through the string and count the occurrences of each character.
3. Display the character counts in a formatted output

5. Functions and Scope

Theory Assignment:

- Define what functions are in JavaScript. Discuss the concepts of function scope, block scope, and hoisting.

Practical Assignment:

- Write a JavaScript function that takes two numbers as parameters and returns their sum. Test the function with different values.

1. Basic Calculator

Assignment:
Create a basic calculator using functions for addition, subtraction, multiplication, and division.

Instructions:

1. Define four separate functions: `add`, `subtract`, `multiply`, and `divide`.
2. Each function should take two parameters and return the result.
3. Prompt the user to enter two numbers and an operator, then call the appropriate function based on the operator.

2. Temperature Converter

Assignment:
Write a function that converts temperatures between Celsius and Fahrenheit.

Instructions:

1. Create a function `convertTemperature` that takes two parameters: temperature and the scale to convert to (either 'C' or 'F').
2. Implement the conversion formulas:
   - Celsius to Fahrenheit: `F = (C * 9/5) + 32`
   - Fahrenheit to Celsius: `C = (F - 32) * 5/9`
3. Prompt the user for the temperature and scale, then call the function and display the result.

3. Palindrome Checker

Assignment:
Create a function that checks if a given string is a palindrome.

Instructions:

1. Define a function `isPalindrome` that takes a string as a parameter.
2. Inside the function, reverse the string and compare it to the original.
3. Return `true` if it is a palindrome and `false` otherwise.
4. Prompt the user for a string and display whether it is a palindrome.

4. Factorial Function

Assignment:
Implement a function that calculates the factorial of a given number.

Instructions:

1. Create a function `factorial` that takes a positive integer as a parameter.
2. Use a loop to calculate the factorial.
3. Prompt the user for a number and display the factorial result.

5. Array Sum

Assignment:
Write a function that calculates the sum of all elements in an array.

Instructions:

1. Define a function `sumArray` that takes an array as a parameter.
2. Use a loop to iterate through the array and calculate the sum.
3. Call the function with a sample array and display the result.

6. Fibonacci Sequence Function

Assignment:
Create a function that returns the Fibonacci sequence up to a specified number of terms.

Instructions:

1. Define a function `fibonacci` that takes a number `n` as a parameter.
2. Use a loop to generate the Fibonacci numbers and store them in an array.
3. Return the array and display it when the function is called.

7. Find Maximum in an Array

Assignment:
Write a function that finds the maximum number in an array.

Instructions:

1. Create a function `findMax` that takes an array as a parameter.
2. Use a loop to iterate through the array and keep track of the maximum value.
3. Call the function with a sample array and display the maximum number.

8. Reverse a String Function

Assignment:
Implement a function that reverses a given string.

Instructions:

1. Define a function `reverseString` that takes a string as a parameter.
2. Use a loop to build the reversed string.
3. Call the function with a sample string and display the reversed result.

9. Count Vowels in a String

Assignment:
Create a function that counts the number of vowels in a string.

Instructions:

1. Define a function `countVowels` that takes a string as a parameter.
2. Loop through the string and count occurrences of vowels (a, e, i, o, u).
3. Return the count and display it when the function is called.

10. Get Random Number in Range

Assignment:
Write a function that generates a random number within a specified range.

Instructions:

1. Define a function `getRandomNumber` that takes two parameters: `min` and `max`.
2. Use the formula `Math.floor(Math.random() * (max - min + 1)) + min` to generate a random number.
3. Call the function and display a random number between two user-specified limits.

11. Array Filter Function

Assignment:
Create a function that filters an array based on a specified condition.

Instructions:

1. Define a function `filterArray` that takes an array and a condition function as parameters.
2. Use a loop to return a new array containing only elements that meet the condition.
3. Test the function with different conditions (e.g., even numbers, odd numbers).

12. Concatenate Arrays

Assignment:
Write a function that concatenates two arrays into one.

Instructions:

1. Define a function `concatArrays` that takes two arrays as parameters.
2. Use the `concat` method to combine the arrays.

3. Call the function with two sample arrays and display the combined array.

13. Remove Duplicates from an Array

Assignment:
Create a function that removes duplicate values from an array.

Instructions:

1. Define a function `removeDuplicates` that takes an array as a parameter.
2. Use a loop to create a new array without duplicates.
3. Return the new array and display it when the function is called.

14. Check Even or Odd

Assignment:
Implement a function that checks if a number is even or odd.

Instructions:

1. Define a function `isEven` that takes a number as a parameter.
2. Use the modulus operator to determine if the number is even or odd.
3. Return `true` if even and `false` if odd. Prompt the user for a number and display the result.

15. Greet User Function

Assignment:
Create a function that greets the user with their name.

Instructions:

1. Define a function `greetUser` that takes a name as a parameter.
2. Display a greeting message using the name provided.
3. Prompt the user for their name and call the function to display the greeting.

6. Arrays and Objects

Theory Assignment:

- Compare and contrast arrays and objects in JavaScript. Explain how to manipulate them using built-in methods.

Practical Assignment:

- Create an array of objects representing students (name, age, grade). Write a program to loop through the array and display each student's details in the console.

1. Array Creation and Access

Assignment:
Create an array of five different fruits and display each fruit in the console.

Instructions:

1. Define an array named `fruits` containing five fruit names.
2. Use a loop to iterate through the array and log each fruit to the console.

2. Finding the Length of an Array

Assignment:
Write a program that calculates and displays the length of an array.

Instructions:

1. Define an array with at least six elements.
2. Use the `length` property to find and display the length of the array.

3. Adding and Removing Elements

Assignment:
Create a program that adds a new fruit to an array and removes the last fruit.

Instructions:

1. Define an array with three fruits.
2. Use the `push` method to add a new fruit to the end of the array.
3. Use the `pop` method to remove the last fruit and display the updated array.

4. Looping Through an Array

Assignment:
Write a program that loops through an array of numbers and displays each number squared.

Instructions:

1. Define an array with at least five numbers.
2. Use a `for` loop to iterate through the array, square each number, and log the result.

5. Filtering Even Numbers

Assignment:
Create a function that filters and returns only even numbers from an array.

Instructions:

1. Define an array with mixed numbers.
2. Create a function `filterEvenNumbers` that takes the array as a parameter and returns a new array with only even numbers.
3. Log the filtered array to the console.

6. Array Sorting

Assignment:
Write a program that sorts an array of strings alphabetically.

Instructions:

1. Define an array of string names.
2. Use the `sort` method to sort the array alphabetically.
3. Display the sorted array.

7. Merging Arrays

Assignment:
Create a program that merges two arrays into one.

Instructions:

1. Define two separate arrays.
2. Use the `concat` method to merge them into a new array.
3. Log the merged array to the console.

8. Summing Array Elements

Assignment:
Write a function that calculates the sum of all elements in an array of numbers.

Instructions:

1. Define an array of numbers.
2. Create a function `sumArray` that iterates through the array and returns the total sum.
3. Call the function and display the sum.

9. Reversing an Array

Assignment:
Implement a function that reverses the order of elements in an array.

Instructions:

1. Define an array with at least five elements.
2. Create a function `reverseArray` that uses a loop to reverse the array.
3. Log the reversed array to the console.

10. Finding the Maximum Value

Assignment:
Create a program that finds and displays the maximum value in an array of numbers.

Instructions:

1. Define an array of numbers.
2. Use a loop to iterate through the array and find the maximum value.
3. Display the maximum value in the console.

Practical Tasks for Objects

11. Creating an Object

Assignment:
Define an object that represents a book with properties like title, author, and year.

Instructions:

1. Create an object named `book` with properties for title, author, and year.
2. Log the object to the console.

12. Accessing Object Properties

Assignment:
Write a program that accesses and displays the properties of an object.

Instructions:

1. Define an object representing a car with properties like make, model, and year.
2. Access and log each property of the object to the console.

13. Updating Object Properties

Assignment:
Create an object and update one of its properties.

Instructions:

1. Define an object representing a person with properties like name and age.
2. Update the age property and log the updated object.

14. Adding Methods to Objects

Assignment:
Define an object that has a method to display a greeting.

Instructions:

1. Create an object named `greeting` with a method `sayHello` that logs a greeting message.
2. Call the method to display the greeting.

15. Object with Nested Objects

Assignment:
Create an object that contains another object as a property.

Instructions:

1. Define an object named `school` that has properties for name and address, and a nested object for the principal with properties for name and years of service.
2. Log the nested object to the console.

16. Object from User Input

Assignment:
Create an object based on user input.

Instructions:

1. Prompt the user for their name and age.
2. Create an object named `user` that contains these properties.
3. Log the user object to the console.

17. Looping Through Object Properties

Assignment:
Write a program that loops through an object's properties and displays each property and its value.

Instructions:

1. Define an object with at least three properties.
2. Use a `for...in` loop to iterate through the object and log each property and value.

18. Array of Objects

Assignment:
Create an array of objects representing different books.

Instructions:

1. Define an array named `books`, where each element is an object with properties for title, author, and year.
2. Use a loop to display the title of each book in the array.

19. Object Destructuring

Assignment:
Implement object destructuring to extract properties from an object.

Instructions:

1. Define an object representing a laptop with properties for brand, model, and price.
2. Use destructuring to extract the properties and log them to the console.

20. Merging Objects

Assignment:
Create a program that merges two objects into one.

Instructions:

1. Define two separate objects with some properties.
2. Use the `Object.assign()` method or spread operator to merge them into a new object.
3. Log the merged object to the console.

7. Higher-Order Functions and Callbacks

Theory Assignment:

- Explain the concept of higher-order functions and how callbacks work in JavaScript.

Practical Assignment:

- Write a higher-order function that takes an array and a callback function as arguments. The function should apply the callback to each element in the array and return a new array with the results.

8. Promises and Asynchronous JavaScript

Theory Assignment:

- Describe what promises are and how they are used to handle asynchronous operations in JavaScript.

Practical Assignment:

- Write a program that simulates an asynchronous operation using `setTimeout`. Create a promise that resolves after a specified delay and logs a message to the console.

9. The DOM and Event Handling

Theory Assignment:

- Explain the Document Object Model (DOM) and how JavaScript interacts with it. Discuss event handling and the event loop.

Practical Assignment:

- Create a simple HTML page with buttons. Write JavaScript to change the background color of the page when each button is clicked.

10. ES6 Features

Theory Assignment:

- Discuss the key features introduced in ES6 (ECMAScript 2015), including arrow functions, template literals, and destructuring.

Practical Assignment:

- Rewrite a set of traditional JavaScript functions using ES6 arrow functions and use template literals for string concatenation.

1. Using `let` and `const`

Assignment:
Demonstrate the difference between `let` and `const` by creating variables.

Instructions:

1. Create a variable using `let` and change its value.
2. Create a variable using `const` and try to change its value. Log both attempts to the console to show the difference.

2. Arrow Functions

Assignment:
Convert a traditional function to an arrow function.

Instructions:

1. Create a regular function that takes two parameters (e.g., `a` and `b`) and returns their sum.
2. Convert this function to an arrow function and log the result of calling it.

3. Template Literals

Assignment:
Use template literals to create a formatted string.

Instructions:

1. Define variables for your first name, last name, and age.
2. Use a template literal to create a greeting string that includes these variables and log it to the console.

4. Default Parameters

Assignment:
Create a function that accepts parameters with default values.

Instructions:

1. Define a function `greet` that takes two parameters: `name` and `greeting` (default to "Hello").
2. Call the function with and without the second parameter and log the results.

5. Destructuring Assignment

Assignment:
Use destructuring to extract properties from an object.

Instructions:

1. Create an object representing a person with properties like `name`, `age`, and `city`.
2. Use destructuring to extract the `name` and `age` properties into separate variables and log them.

6. Spread Operator

Assignment:
Use the spread operator to combine arrays.

Instructions:

1. Create two arrays of numbers.
2. Use the spread operator to combine these arrays into a new array and log the result.

7. Rest Parameters

Assignment:
Implement a function that uses rest parameters to accept an indefinite number of arguments.

Instructions:

1. Define a function `sumAll` that takes any number of arguments and returns their sum.
2. Call the function with different sets of numbers and log the results.

8. Classes

Assignment:
Create a simple class to represent a Rectangle.

Instructions:

1. Define a class `Rectangle` with a constructor that takes `width` and `height`.
2. Add a method to calculate the area and log it when an instance of the class is created.

9. Promises

Assignment:
Create a promise that resolves after a specified time.

Instructions:

1. Create a function `wait` that returns a promise which resolves after 2 seconds.
2. Call the function and log a message when the promise resolves.

10. Async/Await

Assignment:
Use async/await syntax to work with a promise.

Instructions:

1. Create a function that fetches data from an API (you can use a mock API).
2. Use `async` and `await` to handle the promise and log the fetched data.

11. Modules

Assignment:
Create a simple module and import it into another file.

Instructions:

1. Define a module that exports a function to calculate the square of a number.
2. Import the module into another JavaScript file and call the function with a number, logging the result.

12. Set and Map

Assignment:
Demonstrate the usage of `Set` and `Map`.

Instructions:

1. Create a `Set` and add several elements to it. Show how duplicates are handled.
2. Create a `Map` and add key-value pairs. Log the keys and values to the console.

13. Object Literal Enhancements

Assignment:
Use enhanced object literals to create an object.

Instructions:

1. Create two variables for a person's name and age.
2. Use an enhanced object literal to create an object that includes these variables as properties, and log the object.

14. Array Methods (`map`, `filter`, `reduce`)

Assignment:
Use `map`, `filter`, and `reduce` to manipulate an array.

Instructions:

1. Define an array of numbers.
2. Use `map` to create a new array with each number squared.
3. Use `filter` to create an array of even numbers.
4. Use `reduce` to sum all numbers in the original array and log all results.

15. Enhanced `for...of` Loop

Assignment:
Use the `for...of` loop to iterate over an array.

Instructions:

1. Define an array of strings.
2. Use a `for...of` loop to iterate through the array and log each string to the console.

11. Error Handling

Theory Assignment:

- Explain the importance of error handling in JavaScript. Discuss the `try`, `catch`, and `finally` statements.

Practical Assignment:

- Write a function that attempts to parse a JSON string and uses `try...catch` to handle any errors that may occur.

12. Module System

Theory Assignment:

- Describe the module system in JavaScript, including CommonJS and ES Modules. Discuss their significance for code organization.

Practical Assignment:

- Create two separate JavaScript files: one for a module exporting functions and another for importing and using those functions.

13. JavaScript APIs

Theory Assignment:

- Discuss what APIs are and how JavaScript can interact with them, focusing on Fetch API for making network requests.

Practical Assignment:

- Write a JavaScript program that fetches data from a public API (like JSONPlaceholder) and displays the results on a web page.

1. Fetch Data from a Public API

Assignment:
Make a GET request to a public API and display the data.

Instructions:

1. Use the Fetch API to get data from the JSONPlaceholder API (e.g., `/posts`).
2. Parse the JSON response and log the title of each post to the console.

2. Display Data on a Web Page

Assignment:
Fetch data from an API and display it on a web page.

Instructions:

1. Create a simple HTML page with a `<div>` to display the data.
2. Use the Fetch API to get data from the JSONPlaceholder API.
3. Loop through the response and create HTML elements to display the titles and bodies of the posts in the `<div>`.

3. Search Functionality

Assignment:
Implement a search feature that fetches data based on user input.

Instructions:

1. Create a search input field and a button on your HTML page.
2. When the button is clicked, fetch data from the JSONPlaceholder API based on the input (e.g., search for posts by user ID).
3. Display the filtered results on the web page.

4. Error Handling

Assignment:
Implement error handling for your API requests.

Instructions:

1. Use the Fetch API to make a request to a public API (like JSONPlaceholder).
2. Implement `.catch()` to handle any errors during the fetch operation.
3. Display a user-friendly message if an error occurs.

5. POST Request to Create Data

Assignment:
Make a POST request to an API to create new data.

Instructions:

1. Create a form on your HTML page to collect user input (e.g., title and body for a new post).
2. When the form is submitted, use the Fetch API to send a POST request to the JSONPlaceholder API.
3. Display the response (the created post) on the web page.

6. Update Existing Data with PUT

Assignment:
Make a PUT request to update an existing resource.

Instructions:

1. Use the Fetch API to update a post in the JSONPlaceholder API (e.g., update a post with ID 1).
2. Log the updated post to the console and display it on the web page.

7. DELETE Request to Remove Data

Assignment:
Make a DELETE request to remove a resource.

Instructions:

1. Use the Fetch API to send a DELETE request to the JSONPlaceholder API to delete a post.
2. Log a confirmation message to the console and display the updated list of posts after deletion.

8. Handling Promises with Async/Await

Assignment:
Use async/await syntax to handle API requests.

Instructions:

1. Create a function that fetches data from the JSONPlaceholder API using async/await.
2. Log the user data to the console and handle any errors.

9. Using API Response Data

Assignment:
Process and use data received from an API.

Instructions:

1. Fetch data from the OpenWeatherMap API (you'll need an API key).
2. Extract the temperature and weather description from the response and log them to the console.

10. Create a Weather App

Assignment:
Build a simple weather application that fetches and displays weather data.

Instructions:

1. Create an input field to accept a city name and a button to fetch the weather data.
2. Use the OpenWeatherMap API to get the weather data for the entered city when the button is clicked.
3. Display the temperature, weather condition, and city name on the web page.

11. Pagination with API Data

Assignment:
Implement pagination for API data.

Instructions:

1. Fetch data from the JSONPlaceholder API.
2. Display only a limited number of posts (e.g., 5) at a time on the web page.
3. Create "Next" and "Previous" buttons to navigate between pages of results.

12. Caching API Responses

Assignment:
Implement basic caching for API responses.

Instructions:

1. Fetch data from the JSONPlaceholder API.
2. Store the response data in a variable or use local storage to cache the data.
3. Check if the data is cached before making a new request.

13. Creating a Simple API Client

Assignment:
Create a simple API client with functions to GET, POST, PUT, and DELETE.

Instructions:

1. Create a JavaScript object that contains methods for each of the HTTP methods (GET, POST, PUT, DELETE).
2. Use the Fetch API within these methods to interact with the JSONPlaceholder API.
3. Test each method by calling them and logging the results.

14. Form Validation before API Call

Assignment:
Implement form validation before making an API call.

Instructions:

1. Create a form to collect user data (e.g., name and email).
2. Validate the form input to ensure the fields are not empty before making a POST request to a mock API (use JSONPlaceholder).
3. Display appropriate error messages if validation fails.

14. Functional Programming Concepts

Theory Assignment:

- Explain the principles of functional programming as they apply to JavaScript. Discuss concepts like immutability and pure functions.

Practical Assignment:

- Write a program that demonstrates functional programming techniques, such as using `map`, `filter`, and `reduce` on an array of numbers.

4. Understanding Node.js Modules

Theory Assignment:

- Explain the CommonJS module system in Node.js. Discuss how it differs from ES modules.

Practical Assignment:

- Create a simple application that utilizes at least three custom modules. Each module should export a function that can be imported and used in the main application.

5. Working with File Systems

Theory Assignment:

- Discuss the importance of the File System module in Node.js. Explain the difference between synchronous and asynchronous file operations.

Practical Assignment:

- Write a program that reads the contents of a text file and writes the output to another file. Implement both synchronous and asynchronous methods to perform the same task.

1. Reading a File

Assignment:
Read the contents of a text file and log it to the console.

Instructions:

1. Create a text file named `sample.txt` with some sample text.
2. Use the `fs.readFile` method to read the file asynchronously.
3. Log the contents of the file to the console.

2. Writing to a File

Assignment:
Write data to a text file.

Instructions:

1. Create a new text file named `output.txt`.
2. Use the `fs.writeFile` method to write some sample data to the file.
3. Log a success message to the console once the write operation is complete.

3. Appending Data to a File

Assignment:
Append data to an existing file.

Instructions:

1. Create an existing file named `append.txt` and add some initial content.
2. Use the `fs.appendFile` method to add new content to the file.
3. Log the contents of the file after appending to confirm the changes.

4. Reading a File Line by Line

Assignment:
Read a text file line by line.

Instructions:

1. Use the `fs.readFile` method to read the contents of a file.
2. Split the file contents into an array of lines and log each line to the console.
3. Use a loop to iterate through the lines and print them one by one.

5. Checking if a File Exists

Assignment:
Check if a specific file exists in the directory.

Instructions:

1. Use the `fs.existsSync` method to check for the existence of a file (e.g., `check.txt`).
2. Log a message indicating whether the file exists or not.

6. Renaming a File

Assignment:
Rename an existing file.

Instructions:

1. Create a file named `oldname.txt`.
2. Use the `fs.rename` method to change the file name to `newname.txt`.
3. Log a success message once the rename operation is complete.

7. Deleting a File

Assignment:
Delete a specific file from the directory.

Instructions:

1. Create a file named `deleteMe.txt`.
2. Use the `fs.unlink` method to delete the file.
3. Log a success message upon successful deletion.

8. Creating a Directory

Assignment:
Create a new directory.

Instructions:

1. Use the `fs.mkdir` method to create a new directory named `newDirectory`.
2. Log a success message once the directory is created.

9. Reading Files from a Directory

Assignment:
List all files in a specific directory.

Instructions:

1. Create a directory named `myFiles` and add several files to it.
2. Use the `fs.readdir` method to read the contents of the directory.
3. Log the names of all files to the console.

10. Copying a File

Assignment:
Copy a file from one location to another.

Instructions:

1. Create a file named `original.txt` with some sample content.
2. Use the `fs.copyFile` method to create a copy of the file named `copy.txt`.
3. Log a success message after the copy operation is complete.

11. Watching for File Changes

Assignment:
Watch a file for changes and log the changes.

Instructions:

1. Use the `fs.watch` method to monitor changes to a specific file (e.g., `watch.txt`).
2. Log a message to the console whenever the file is modified.

12. Reading a JSON File

Assignment:
Read a JSON file and parse its contents.

Instructions:

1. Create a JSON file named `data.json` with some sample data.
2. Use `fs.readFile` to read the file and parse its contents using `JSON.parse`.
3. Log the parsed object to the console.

13. Writing JSON Data to a File

Assignment:
Write a JavaScript object to a JSON file.

Instructions:

1. Create a JavaScript object with sample data (e.g., user details).
2. Use `fs.writeFile` to write the object to a file named `output.json` after converting it to a JSON string using `JSON.stringify`.
3. Log a success message once the write operation is complete.

14. Reading a File Asynchronously with Promises

Assignment:
Read a file using Promises and handle errors.

Instructions:

1. Create a text file named `promiseFile.txt` with some content.
2. Use the `fs.promises.readFile` method to read the file.
3. Handle any potential errors using `.catch()` and log the contents to the console.

15. Using Streams to Read and Write Files

Assignment:
Read and write files using streams.

Instructions:

1. Create a large text file named `largeFile.txt` with repetitive content.
2. Use `fs.createReadStream` to read the file in chunks and log each chunk to the console.
3. Use `fs.createWriteStream` to create a new file named `outputStream.txt` and write data to it.

6. HTTP and Web Servers

Theory Assignment:

- Describe how Node.js handles HTTP requests and responses. Discuss the request-response lifecycle.

Practical Assignment:

- Create a simple web server using the built-in http module. The server should respond with a "Hello World" message for GET requests.

1. Creating a Basic HTTP Server

Assignment:
Create a simple HTTP server using Node.js.

Instructions:

1. Use the `http` module to create a basic server that listens on a specified port (e.g., 3000).
2. Respond with a plain text message like "Hello, World!" when accessed in a browser.
3. Log a message to the console when the server starts.

2. Handling Different URL Routes

Assignment:
Implement routing to handle different URL paths.

Instructions:

1. Create an HTTP server that responds with different messages based on the requested URL path:
   - `/` should respond with "Welcome to the Home Page!"
   - `/about` should respond with "This is the About Page."
   - `/contact` should respond with "This is the Contact Page."
2. Log the requested URL to the console.

3. Handling HTTP GET Requests

Assignment:
Implement handling for GET requests.

Instructions:

1. Create a server that responds to a GET request to `/api/users` with a JSON array of user objects.
2. Use `res.setHeader` to set the response content type to `application/json`.
3. Include at least three sample users in the JSON response.

4. Handling HTTP POST Requests

Assignment:
Implement handling for POST requests.

Instructions:

1. Set up a POST route (`/api/users`) to accept user data (e.g., name and email).
2. Use the `body-parser` middleware to parse the incoming JSON data.
3. Respond with a success message and the received user data in JSON format.

5. Serving Static Files

Assignment:
Serve static files using Node.js.

Instructions:

1. Create a directory named `public` and add some HTML, CSS, and JavaScript files.
2. Use the `fs` module to serve the files in response to requests (e.g., `index.html` when accessing `/`).
3. Ensure that the correct content type is set for each file type (e.g., `text/html`, `text/css`, `application/javascript`).

6. Creating a Simple RESTful API

Assignment:
Create a RESTful API for managing a list of items.

Instructions:

1. Set up an HTTP server that handles the following routes:
   - `GET /api/items` to return a list of items (array of objects).
   - `POST /api/items` to add a new item.
   - `PUT /api/items/:id` to update an existing item by ID.
   - `DELETE /api/items/:id` to delete an item by ID.
2. Use a simple in-memory array to store the items.

7. Error Handling

Assignment:
Implement error handling for your HTTP server.

Instructions:

1. Add a middleware function to handle 404 errors for undefined routes.
2. Log the error message and respond with a 404 status code and a friendly message ("Page not found!").
3. Handle any potential server errors with appropriate responses.

8. Using Query Parameters

Assignment:
Handle query parameters in your HTTP requests.

Instructions:

1. Create an HTTP server that handles a GET request to `/api/search` with query parameters (e.g., `?q=node`).
2. Respond with a JSON object containing the search query and a message (e.g., "You searched for: node").
3. Log the received query parameters to the console.

9. Creating a Basic Form and Handling Form Submission

Assignment:
Create a basic HTML form and handle its submission.

Instructions:

1. Serve an HTML form with fields for a name and email.
2. Use the POST method to submit the form data to `/submit`.
3. Handle the form submission on the server side and respond with a success message including the submitted data.

10. Implementing Middleware

Assignment:
Create a middleware function for logging requests.

Instructions:

1. Create a middleware function that logs the request method and URL for every incoming request.
2. Use the middleware in your server to track all requests made to your API.
3. Log the request details to the console.

11. Basic Authentication

Assignment:
Implement basic authentication for a protected route.

Instructions:

1. Create a simple login route (`/login`) that accepts a username and password.
2. Validate the credentials and respond with a success message or an unauthorized error.
3. Protect a route (e.g., `/api/protected`) that requires successful login to access.

12. Redirecting Requests

Assignment:
Implement request redirection.

Instructions:

1. Set up a redirect from `/old-url` to `/new-url`.
2. Use the `res.writeHead` method to set the status code to 301 (Moved Permanently) and redirect users to the new URL.

13. Setting Custom Headers

Assignment:
Set custom HTTP headers in your responses.

Instructions:

1. Create a server that responds to requests with custom headers (e.g., `X-Powered-By`, `Content-Type`).
2. Log the headers sent in the response to the console.

14. Rate Limiting

Assignment:
Implement a simple rate-limiting mechanism.

Instructions:

1. Create a middleware that limits the number of requests a user can make to the server within a specific time frame (e.g., 5 requests per minute).
2. Use an in-memory store to track request counts and reset them after the time window.

15. Setting Up a Proxy

Assignment:
Set up a simple proxy for an external API.

Instructions:

1. Create a route that proxies requests to an external API (e.g., a public API like JSONPlaceholder).
2. Use the `http-proxy-middleware` package to handle the proxying.
3. Log the requests and responses to the console.

7. Express.js Framework

Theory Assignment:

- Explain the advantages of using the Express.js framework over the built-in HTTP module.

Practical Assignment:

- Build a basic Express.js application with at least three routes (e.g., GET, POST, DELETE) and demonstrate how to handle request parameters and query strings.

1. Setting Up a Basic Express Server

Assignment:
Create a basic Express server.

Instructions:

1. Set up an Express project using npm.
2. Create a server that listens on a specified port (e.g., 3000).
3. Respond with "Welcome to Express!" when accessed at the root route (`/`).
4. Log a message to the console indicating the server has started.

2. Creating Routes

Assignment:
Implement multiple routes in your Express application.

Instructions:

1. Create routes for:
   - `/` - respond with "Home Page"
   - `/about` - respond with "About Page"
   - `/contact` - respond with "Contact Page"
2. Log the requested route in the console.

3. Using Express Middleware

Assignment:
Implement custom middleware in your Express application.

Instructions:

1. Create a middleware function that logs the request method and URL.
2. Use this middleware in your Express app to log all incoming requests.
3. Ensure the middleware does not block the request from reaching the route handlers.

4. Handling Query Parameters

Assignment:
Handle query parameters in an Express route.

Instructions:

1. Create a route `/api/search` that accepts a query parameter (e.g., `?q=searchTerm`).
2. Respond with a JSON object that includes the search term and a message (e.g., `{"query": "searchTerm", "message": "Search received"}`).

3. Log the received query parameter to the console.

5. Handling Form Data with POST Requests

Assignment:
Set up a route to handle form submissions.

Instructions:

1. Create an HTML form with fields for name and email.
2. Set the form method to POST and action to `/submit`.
3. Create a POST route `/submit` that processes the form data and responds with a success message.

6. Serving Static Files

Assignment:
Serve static files using Express.

Instructions:

1. Create a directory called `public` and add HTML, CSS, and JavaScript files.
2. Use `express.static` to serve the static files from the `public` directory.
3. Access the files via URLs (e.g., `/index.html`).

7. Implementing Error Handling Middleware

Assignment:
Create a custom error handling middleware.

Instructions:

1. Implement an error handling middleware function that catches errors and sends a JSON response with the error message.
2. Ensure the middleware is used after all route handlers.
3. Simulate an error in one of your routes and verify the error handling works correctly.

8. Using Router for Modular Routing

Assignment:
Organize your routes using Express Router.

Instructions:

1. Create a new router for user-related routes (`/api/users`).
2. Implement routes to:
   - `GET /api/users` - respond with a list of users.
   - `POST /api/users` - add a new user.
3. Import the router into your main Express app and use it.

9. Connecting to a MongoDB Database

Assignment:
Connect your Express application to a MongoDB database using Mongoose.

Instructions:

1. Install and set up Mongoose in your project.
2. Create a User model with fields for name and email.
3. Set up a POST route (`/api/users`) to save user data to the MongoDB database.

10. Implementing Basic Authentication

Assignment:
Create a simple login system using basic authentication.

Instructions:

1. Set up a login route that accepts username and password.
2. Validate the credentials (hardcoded for simplicity).
3. Respond with a success or failure message based on the validation.

11. Implementing JSON Web Tokens (JWT)

Assignment:
Set up JWT authentication for protecting routes.

Instructions:

1. Create a registration route that generates a JWT for new users.
2. Protect a route (e.g., `/api/protected`) that requires a valid JWT to access.
3. Use middleware to verify the JWT on the protected route.

12. Using Environment Variables

Assignment:
Utilize environment variables in your Express application.

Instructions:

1. Use the `dotenv` package to load environment variables from a `.env` file.
2. Store sensitive information like the database connection string in the `.env` file.
3. Access the environment variables in your code and connect to the database using these variables.

13. Implementing Rate Limiting

Assignment:
Create a rate-limiting middleware.

Instructions:

1. Implement a middleware function that limits the number of requests from a single IP address within a time frame (e.g., 100 requests per hour).
2. Use an in-memory store or a package like `express-rate-limit` to handle rate limiting.

14. Sending Emails with Nodemailer

Assignment:
Set up email functionality using Nodemailer.

Instructions:

1. Install and configure Nodemailer in your Express application.
2. Create a route `/send-email` that sends an email when accessed.
3. Customize the email content (subject, body) and log the result of the sending operation.

15. Implementing CORS

Assignment:
Enable Cross-Origin Resource Sharing (CORS) in your Express application.

Instructions:

1. Use the `cors` middleware to allow requests from different origins.
2. Configure CORS to only allow specific origins if desired.
3. Test your setup by making requests from a different domain.

16. Implementing File Uploads

Assignment:
Set up file uploads in your Express application.

Instructions:

1. Use the `multer` middleware to handle file uploads.
2. Create a form for uploading files and set the form's `enctype` to `multipart/form-data`.
3. Create a POST route to handle the file upload and respond with the uploaded file's details.

17. Creating a Simple Todo Application

Assignment:
Build a basic todo application with CRUD operations.

Instructions:

1. Set up routes to:
   - `GET /api/todos` - retrieve the list of todos.
   - `POST /api/todos` - create a new todo.
   - `PUT /api/todos/:id` - update a todo by ID.
   - `DELETE /api/todos/:id` - delete a todo by ID.
2. Store the todos in an in-memory array or a MongoDB database.

18. Handling Cookies and Sessions

Assignment:
Implement cookie and session handling in your Express application.

Instructions:

1. Use `express-session` to manage user sessions.
2. Create a login route that sets a session upon successful login.
3. Protect a route that requires the user to be logged in.

19. Setting Up Unit Tests with Mocha and Chai

Assignment:
Write unit tests for your Express routes.

Instructions:

1. Set up a testing environment using Mocha and Chai.
2. Write tests for your routes to verify correct responses for various scenarios (e.g., success, error).
3. Run the tests and ensure they pass.

20. Deploying Your Express Application

Assignment:
Deploy your Express application to a cloud service.

Instructions:

1. Choose a cloud platform (e.g., Heroku, Vercel, or AWS) to deploy your application.
2. Follow the platform's guidelines to deploy your Express app.
3. Ensure that the application runs correctly in the production environment.

8. Restful APIs

Theory Assignment:

- Define RESTful API principles and discuss their importance in web application development.

Practical Assignment:

- Create a RESTful API for a simple resource (e.g., books). Implement CRUD (Create, Read, Update, Delete) operations using Express.js.

1. Setting Up a Basic RESTful API

Assignment:
Create a basic RESTful API using Node.js and Express.

Instructions:

1. Initialize a new Node.js project with `npm init`.
2. Install Express using `npm install express`.
3. Create a server that listens on a specified port (e.g., 3000).
4. Implement a simple GET endpoint (`/api`) that returns a JSON response (e.g., `{ message: "Welcome to the API!" }`).

2. Implementing CRUD Operations

Assignment:
Create a RESTful API for managing a list of items (e.g., tasks, products).

Instructions:

1. Set up routes for the following CRUD operations:
    - `GET /api/items` - Retrieve all items.
    - `GET /api/items/:id` - Retrieve a single item by ID.
    - `POST /api/items` - Create a new item.
    - `PUT /api/items/:id` - Update an existing item by ID.
    - `DELETE /api/items/:id` - Delete an item by ID.
2. Store items in an in-memory array (or use a simple database).

3. Handling Request and Response

Assignment:
Implement detailed request and response handling for your API.

Instructions:

1. For the `POST /api/items` route, accept JSON data for creating a new item.
2. Validate incoming data and send appropriate responses for success or error (e.g., 400 for bad requests).
3. Use the `res.status()` method to set the HTTP status code based on the operation's result.

4. Implementing Middleware for Logging

Assignment:
Create middleware to log incoming requests to your API.

Instructions:

1. Implement a logging middleware that logs the request method, URL, and timestamp.
2. Apply this middleware to all incoming requests.
3. Use `console.log` to output the log information to the terminal.

5. Using Query Parameters

Assignment:
Enhance your API to handle query parameters for filtering.

Instructions:

1. Modify the `GET /api/items` route to accept optional query parameters (e.g., `?search=keyword`).
2. Filter the list of items based on the search query and return the filtered results.
3. Respond with a message if no items match the search criteria.

6. Connecting to a MongoDB Database

Assignment:
Integrate MongoDB with your RESTful API.

Instructions:

1. Install Mongoose using `npm install mongoose`.
2. Set up a connection to a MongoDB database.
3. Replace the in-memory array with a MongoDB collection to store items.
4. Implement the CRUD operations to interact with the MongoDB database.

7. Implementing Error Handling

Assignment:
Create a centralized error handling mechanism for your API.

Instructions:

1. Implement an error handling middleware that catches errors and responds with a JSON error message.
2. Ensure that the error handling middleware is the last middleware in your app.
3. Simulate an error in one of your routes and verify that the error handling works correctly.

8. Adding Authentication (Basic or JWT)

Assignment:
Secure your API with basic authentication or JWT (JSON Web Tokens).

Instructions:

1. Choose between basic authentication or JWT.
2. For JWT:
   - Set up user registration and login routes to issue tokens.
   - Protect certain routes by requiring a valid token.
3. For basic authentication:
   - Implement a simple check for username and password in the headers.

9. Implementing Pagination

Assignment:
Add pagination to your `GET /api/items` route.

Instructions:

1. Allow clients to request specific pages of items by accepting `page` and `limit` query parameters.
2. Calculate the appropriate items to return based on the requested page and limit.
3. Respond with the paginated list of items and include metadata (e.g., total items, current page).

10. Testing Your API with Postman

Assignment:
Use Postman to test your RESTful API.

Instructions:

1. Install Postman and create a new collection for your API.
2. Create requests for each of the CRUD operations and test their functionality.
3. Document the expected responses for each request and check that they match your implementation.

11. Implementing CORS

Assignment:
Enable Cross-Origin Resource Sharing (CORS) for your API.

Instructions:

1. Install the CORS middleware using `npm install cors`.
2. Configure CORS to allow requests from specific origins or all origins.
3. Test your API from a frontend application to ensure CORS is working correctly.

12. Versioning Your API

Assignment:
Implement versioning for your RESTful API.

Instructions:

1. Set up your API to support multiple versions (e.g., `/api/v1/items` and `/api/v2/items`).
2. Differentiate the implementations between versions, allowing for changes and improvements in newer versions.
3. Test both versions to ensure they function correctly.

13. Implementing Rate Limiting

Assignment:
Add rate limiting to your RESTful API.

Instructions:

1. Use the `express-rate-limit` package to limit the number of requests to your API.
2. Configure the rate limiter for specific routes (e.g., limit requests to `/api/items`).
3. Test the rate limiting by making repeated requests.

14. Creating a Simple Client to Consume Your API

Assignment:
Create a simple frontend application to interact with your RESTful API.

Instructions:

1. Use HTML and JavaScript (or a library like Axios) to create a basic user interface.
2. Implement functions to make API calls to your backend for CRUD operations.
3. Display the results (e.g., list of items, responses) in the UI.

15. Deploying Your RESTful API

Assignment:
Deploy your RESTful API to a cloud service.

Instructions:

1. Choose a cloud platform (e.g., Heroku, Vercel, AWS) for deployment.
2. Follow the platform's guidelines to deploy your Node.js application.
3. Ensure that the API runs correctly in the production environment and can be accessed publicly.

9. Asynchronous Programming

Theory Assignment:

- Discuss the importance of asynchronous programming in Node.js. Explain the concepts of callbacks, promises, and async/await.

Practical Assignment:

- Refactor a callback-based code snippet to use promises and then implement async/await for the same functionality.

1. Understanding Callbacks

Assignment:
Create a simple Node.js application that demonstrates the use of callbacks.

Instructions:

1. Write a function `fetchData` that simulates fetching data with a delay (using `setTimeout`).
2. Use a callback to return the fetched data after the delay.
3. Create another function that calls `fetchData` and logs the result to the console.

Example:

```
javascript
Copy code
function fetchData(callback) {
    setTimeout(() => {
        const data = { message: "Data fetched successfully!" };
        callback(data);
    }, 2000);
}

fetchData((result) => {
    console.log(result);
});
```

2. Promises

Assignment:
Convert the previous callback-based implementation to use promises.

Instructions:

1. Modify the `fetchData` function to return a promise.
2. Resolve the promise with the fetched data after the delay.
3. Create a function that calls `fetchData` and uses `.then()` to log the result.

Example:

```
javascript
Copy code
function fetchData() {
    return new Promise((resolve) => {
        setTimeout(() => {
            const data = { message: "Data fetched successfully!" };
            resolve(data);
        }, 2000);
    });
}

fetchData().then((result) => {
    console.log(result);
});
```

3. Chaining Promises

Assignment:
Create a sequence of asynchronous operations using promise chaining.

Instructions:

1. Create a function `processData` that takes data as input and returns a promise.
2. Chain two calls to `fetchData` and `processData` to demonstrate how to work with multiple asynchronous operations.
3. Log the final result after both operations are complete.

Example:

```
javascript
Copy code
function processData(data) {
    return new Promise((resolve) => {
        setTimeout(() => {
            const processed = { message: data.message.toUpperCase() };
            resolve(processed);
        }, 1000);
    });
}

fetchData()
    .then((result) => processData(result))
    .then((processedResult) => {
        console.log(processedResult);
    });
```

4. Async/Await

Assignment:
Refactor the promise-based implementation to use async/await syntax.

Instructions:

1. Create an `async` function that fetches data and processes it using `await`.
2. Use `try/catch` to handle any errors that may occur during the asynchronous operations.
3. Log the final result.

Example:

```
javascript
Copy code
async function fetchAndProcessData() {
    try {
        const result = await fetchData();
        const processedResult = await processData(result);
        console.log(processedResult);
    } catch (error) {
        console.error("Error:", error);
    }
}

fetchAndProcessData();
```

5. Error Handling in Promises

Assignment:
Implement error handling in your promise-based functions.

Instructions:

1. Modify the `fetchData` function to randomly throw an error (e.g., using `Math.random()`).
2. Use `.catch()` to handle errors when calling the function.
3. Log the error message to the console.

Example:

```
javascript
Copy code
function fetchData() {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            if (Math.random() < 0.5) {
                const data = { message: "Data fetched successfully!" };
                resolve(data);
            } else {
                reject(new Error("Failed to fetch data."));
            }
        }, 2000);
    });
}

fetchData()
    .then((result) => console.log(result))
    .catch((error) => console.error("Error:", error.message));
```

6. Using `Promise.all()`

Assignment:
Demonstrate the use of `Promise.all()` to handle multiple asynchronous operations.

Instructions:

1. Create multiple functions that return promises (e.g., `fetchData1`, `fetchData2`).
2. Use `Promise.all()` to wait for all promises to resolve.
3. Log the results of all resolved promises.

Example:

```
javascript
Copy code
```

```
function fetchData1() {
    return new Promise((resolve) => {
        setTimeout(() => resolve("Data 1"), 1000);
    });
}

function fetchData2() {
    return new Promise((resolve) => {
        setTimeout(() => resolve("Data 2"), 1500);
    });
}

Promise.all([fetchData1(), fetchData2()])
    .then((results) => {
        console.log("All data:", results);
    });
```

7. Using `Promise.race()`

Assignment:
Use `Promise.race()` to handle the first promise that resolves.

Instructions:

1. Create two functions that return promises with different timeouts.
2. Use `Promise.race()` to get the result of the promise that resolves first.
3. Log the result of the winning promise.

Example:

```
javascript
Copy code
function fetchData1() {
    return new Promise((resolve) => {
        setTimeout(() => resolve("Data 1"), 2000);
    });
}

function fetchData2() {
    return new Promise((resolve) => {
        setTimeout(() => resolve("Data 2"), 1000);
    });
}

Promise.race([fetchData1(), fetchData2()])
    .then((result) => {
        console.log("First resolved:", result);
    });
```

8. Handling Asynchronous File Operations

Assignment:
Demonstrate asynchronous file operations using the `fs` module with promises.

Instructions:

1. Use the `fs.promises` API to read and write files.
2. Create a function that reads a file, processes its content, and writes the result to a new file.
3. Log the success or error message.

Example:

```
javascript
Copy code
const fs = require('fs').promises;

async function readAndWriteFile() {
    try {
        const data = await fs.readFile('input.txt', 'utf8');
        const processedData = data.toUpperCase();
        await fs.writeFile('output.txt', processedData);
        console.log("File written successfully!");
    } catch (error) {
        console.error("Error:", error.message);
    }
}

readAndWriteFile();
```

9. Asynchronous HTTP Requests

Assignment:
Make asynchronous HTTP requests using the `axios` library.

Instructions:

1. Install the `axios` library using `npm install axios`.
2. Create an async function that fetches data from a public API (e.g., JSONPlaceholder).
3. Log the fetched data to the console.

Example:

```
javascript
Copy code
const axios = require('axios');

async function fetchPosts() {
    try {
        const response = await
axios.get('https://jsonplaceholder.typicode.com/posts');
        console.log(response.data);
    } catch (error) {
        console.error("Error:", error.message);
    }
}

fetchPosts();
```

10. Combining Async/Await with Express

Assignment:
Create an Express application that uses async/await for route handlers.

Instructions:

1. Set up an Express server with a route that fetches data from an external API.
2. Use async/await in the route handler to manage asynchronous requests.
3. Return the fetched data as a JSON response.

Example:

```
javascript
Copy code
const express = require('express');
const axios = require('axios');
const app = express();

app.get('/posts', async (req, res) => {
    try {
        const response = await
axios.get('https://jsonplaceholder.typicode.com/posts');
        res.json(response.data);
    } catch (error) {
        res.status(500).send("Error fetching data.");
    }
});

app.listen(3000, () => {
    console.log("Server is running on port 3000");
});
```

10. Database Integration

Theory Assignment:

- Compare SQL and NoSQL databases. Discuss how to connect Node.js applications to databases.

Practical Assignment:

1. Setting Up a Node.js Project

Assignment:
Create a basic Node.js application.

Instructions:

1. Initialize a new Node.js project using `npm init -y`.
2. Install the required packages: `express`, `mongoose`, and `nodemon`.

3. Create a simple Express server that listens on a specified port.

2. Connecting to MongoDB

Assignment:
Connect your Node.js application to a MongoDB database.

Instructions:

1. Create a MongoDB cluster using MongoDB Atlas.
2. Use Mongoose to connect to the MongoDB database in your Node.js application.
3. Log a message to the console to confirm the connection.

3. Creating a Model

Assignment:
Define a Mongoose model for a collection.

Instructions:

1. Create a Mongoose schema and model for a simple `User` collection with fields like `name`, `email`, and `password`.
2. Export the model for use in other parts of your application.

4. Creating and Saving Documents

Assignment:
Implement a route to create a new document in the MongoDB database.

Instructions:

1. Create a POST route (`/users`) in your Express app to accept user data.
2. Use the Mongoose model to save the data to the database.
3. Return a success message or the created user data in the response.

5. Retrieving Documents

Assignment:
Implement a route to fetch documents from the MongoDB database.

Instructions:

1. Create a GET route (`/users`) to retrieve all users from the database.
2. Use the Mongoose model to find all documents and return them as JSON.

6. Updating Documents

Assignment:
Implement a route to update an existing document.

Instructions:

1. Create a PUT route (`/users/:id`) that accepts user ID as a URL parameter.
2. Use Mongoose to find the user by ID and update their information.
3. Return the updated user data in the response.

7. Deleting Documents

Assignment:
Implement a route to delete a document from the MongoDB database.

Instructions:

1. Create a DELETE route (`/users/:id`) that accepts user ID as a URL parameter.
2. Use Mongoose to delete the user by ID.
3. Return a confirmation message upon successful deletion.

8. Using Query Parameters

Assignment:
Implement filtering of users using query parameters.

Instructions:

1. Modify the GET route (`/users`) to accept optional query parameters (e.g., `name` or `email`).
2. Use these parameters to filter the results when retrieving users from the database.

9. Validating User Input

Assignment:
Add validation for user input before saving to the database.

Instructions:

1. Use the `express-validator` package to validate user input in the POST route.
2. Return appropriate error messages if validation fails.

10. Implementing Pagination

Assignment:
Implement pagination for user retrieval.

Instructions:

1. Modify the GET route (`/users`) to accept `page` and `limit` query parameters.
2. Use Mongoose's `.skip()` and `.limit()` methods to paginate the results.

11. Setting Up a Simple Authentication System

Assignment:
Create a simple user authentication system.

Instructions:

1. Implement a registration route to create new users and hash passwords using `bcrypt`.
2. Implement a login route to authenticate users and return a JSON Web Token (JWT).

12. Protecting Routes with Middleware

Assignment:
Implement middleware to protect certain routes.

Instructions:

1. Create a middleware function that checks for a valid JWT in the authorization header.
2. Protect the `/users` route so that only authenticated users can access it.

13. Aggregating Data

Assignment:
Use MongoDB aggregation to perform data analysis.

Instructions:

1. Create a route that returns the total number of users and any other relevant statistics (e.g., count by email domain).
2. Use Mongoose's `.aggregate()` method to perform the aggregation.

14. Handling Errors

Assignment:
Implement error handling for your application.

Instructions:

1. Create a centralized error-handling middleware in your Express application.
2. Use this middleware to handle errors from your routes and return appropriate responses.

15. Deployment

Assignment:
Deploy your Node.js application with MongoDB to a cloud platform.

Instructions:

1. Deploy your application to platforms like Heroku, Vercel, or Render.
2. Ensure your MongoDB connection string is set up properly for the deployed environment.

11. Error Handling

Theory Assignment:

- Explain the importance of error handling in Node.js applications. Discuss the various error handling techniques.

Practical Assignment:

- Write a Node.js application that includes error handling for asynchronous operations. Use try/catch blocks and promise.catch to manage errors gracefully.

12. Testing and Debugging

Theory Assignment:

- Discuss the importance of testing in software development. Describe popular testing frameworks for Node.js.

Practical Assignment:

- Write unit tests for your Express.js application using Mocha and Chai. Test at least two routes of your API to ensure they return the expected results.

13. Middleware in Express.js

Theory Assignment:

- Define middleware in the context of Express.js and discuss its role in request handling.

Practical Assignment:

- Create custom middleware for logging request details (method, URL, timestamp) and apply it to your Express.js application.

14. Authentication and Security

Theory Assignment:

- Discuss the importance of authentication and security in web applications. Explain common strategies for user authentication in Node.js.

Practical Assignment:

- Implement user authentication in your Express.js application using Passport.js or JWT (JSON Web Tokens). Create registration and login routes.