V.E.S. Institute of Technology, Collector Colony, Chembur,
Mumbai, Maharashtra 400047
Department of M.C.A

INDEX

Sr. No.	Contents	Date Of Preparation	Date Of Submission	Marks	Faculty Sign
1.	Introduction to node.js and implementation of basic programs using JavaScript.	30-09-2024	08-10-2024		
2.	Implementation of Node.js modules.	07-10-2024	11-10-2024		
3.	Implementing Asynchronous Array Operations in Node.js Through Event Driven Programming.	14-10-2024	18-10-2024		
4.	Timer module functions and its explanation.	23-10-2024	31-10-2024		
5.	Using file handling to demonstrate all basic file operations.	23-10-2024	15-11-2024		
6.	Implementation of HTTP Module in Node.js	11-10-2024	16-11-2024		
7	MySQL Database Connectivity and Operations	13-11-2024	22-11-2024		
8	TypeScript installation, Environment setup	25-11-2024	29-11-2024		
9	Setting up an Angular Development Environment.	27-12-2024	07-12-2024		
10	Angular Forms and their types	27-11-2024	07-12-2024		

Final Grade	Instructor Signature

Name of Student: Shaikh Farhan			
Roll Number: 50		**LAB Assignment Number: 1**	
Title of LAB Assignment: Introduction to node.js and implementation of basic program using javascript.			
DOP: 30 – 09 – 2024		**DOS: 08 – 10 – 2024**	
CO Mapped: CO1	**PO Mapped: PO3, PO5**	**Faculty Signature:**	**Marks:**

Practical No: 1

Aim: Introduction to node.js and implementation of basic program using javascript.

Description:

1. Introduction to node.js:

Node.js is an open-source, server-side JavaScript runtime environment that allows developers to build scalable and efficient network applications. It was initially developed by Ryan Dahl in 2009 and has since gained immense popularity in the world of web development. Node.js is known for its event-driven, non-blocking I/O model, which makes it particularly well-suited for building real-time applications and handling a large number of concurrent connections.

Node.js is a JavaScript runtime that enables developers to build high-performance, scalable, and event- driven server-side applications. Its event-driven architecture, extensive package ecosystem, and cross-platform compatibility have made it a popular choice for modern web and network application development.

key aspects of Node.js

1] **JavaScript Everywhere**: It allows using JavaScript for both client and server-side development.

2] **Event-Driven, Non-Blocking**: Its event-driven, non-blocking architecture handles tasks asynchronously, making it efficient for handling concurrent operations.

3] **npm** Package Manager: Node.js comes with npm, simplifying the installation and management of third-party libraries.

4] **Extensive Ecosystem**: A vast ecosystem of open-source modules covers various functionalities, aiding rapid application development.

5] **Single-Threaded Event Loop**: Node.js efficiently manages many concurrent connections without creating new threads.

2. Advantages and Disadvantages of Node.js:

a) Advantages:

1] High Performance:
Node.js is known for its fast execution due to its non-blocking, event-driven architecture. It excels in handling multiple concurrent connections efficiently.

2] Unified Language:
Developers can use JavaScript on both the client and server sides, reducing the need to switch between languages
and improving code reusability.

3] Rich Ecosystem:
Node.js has a vast collection of open-source libraries and modules available through npm, making it easy to extend
functionality and speed up development.

4] Scalability:
It's well-suited for building scalable applications, including microservices, due to its lightweight nature and ability to
handle numerous connections simultaneously.

5] Real-Time Applications:
Node.js is ideal for real-time applications such as chat apps, online gaming servers, and streaming services, thanks to
its responsiveness.

b) Disadvantages:

1] Single-Threaded:
While Node.js's single-threaded model is efficient for I/O-bound tasks, it may not be the best choice for CPU-bound
tasks, as it can't take full advantage of multi-core processors in the system.

2] Callback Hell:
Managing callbacks for asynchronous operations can lead to callback hell or callback pyramid, making code complex
and hard to maintain. However, this can be mitigated using modern JavaScript features or libraries like Promises or
async/await.

3] Immaturity for Some Use Cases:
Node.js might not be the best choice for all use cases, particularly those that heavily rely on CPU-intensive operations.

4] Vulnerabilities in Packages:
The reliance on third-party packages increases the risk of security vulnerabilities. It is crucial for developers to regularly update dependencies and perform security audits to mitigate potential risks.

4] JavaScript Drawbacks:
As a dynamically typed language, JavaScript can lead to runtime errors that are hard to debug. While TypeScript can be used to add static typing, it introduces additional complexity to the development process.

3. REPL (Read-Eval-Print Loop)

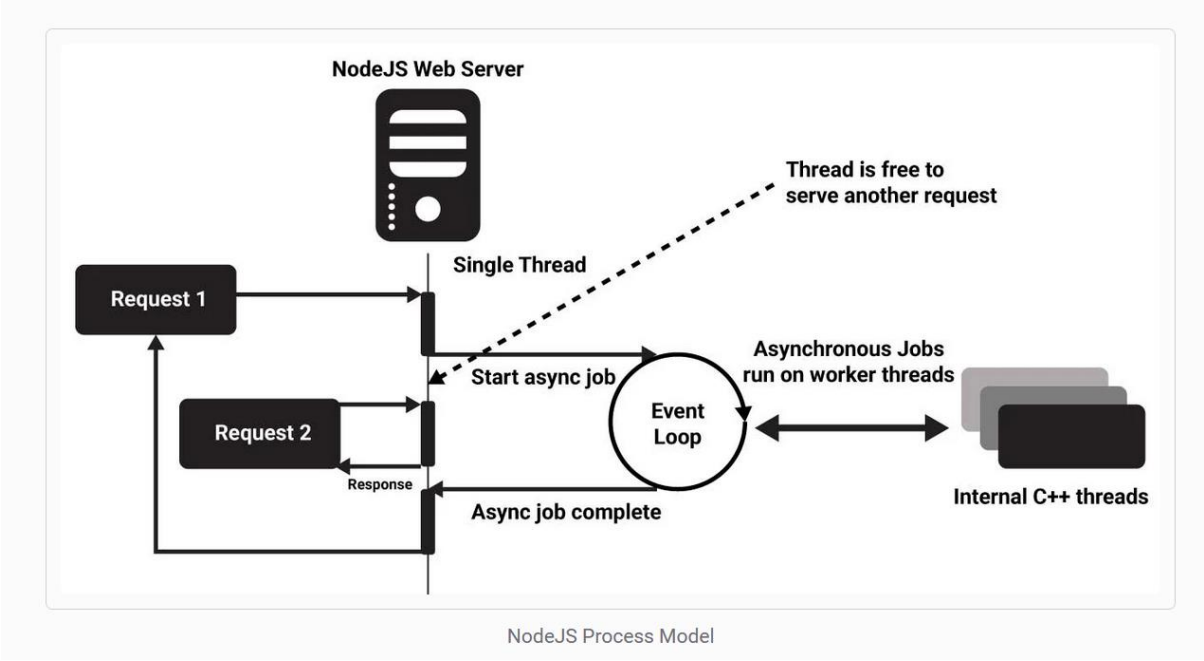
Node.js comes with a built-in REPL, which is a command-line environment for running JavaScript code interactively.

It's useful for quick testing and experimentation, allowing developers to test code snippets and understand behaviour without creating full-fledged programs.

4. Node.js Process Model:

The Node.js process model differs from traditional web servers in that Node.js runs in a single process with requests being processed on a single thread. One advantage of this is that Node.js requires far fewer resources. When a request comes in, it will be placed in an event queue. Node.js uses an event loop to listen for events to be raised for an asynchronous job. The event loop continuously runs, receiving requests from the event queue.

There are two scenarios that will occur depending on the nature of the request. If the request is non-blocking, it does not involve any long-running processes or data requests, the response will be immediately prepared and then sent back to the client. In the event the request is blocking, requiring I/O operations, the request will be sent to a worker thread pool. The request will have an associated call-back function that will fire when the request is finished and the worker thread can send the request to the event loop to be sent back to the client. In this way, when the single thread receives a blocking request, it hands it off so that the thread can process other requests in the meantime. In this way Node.js is inherently asynchronous.



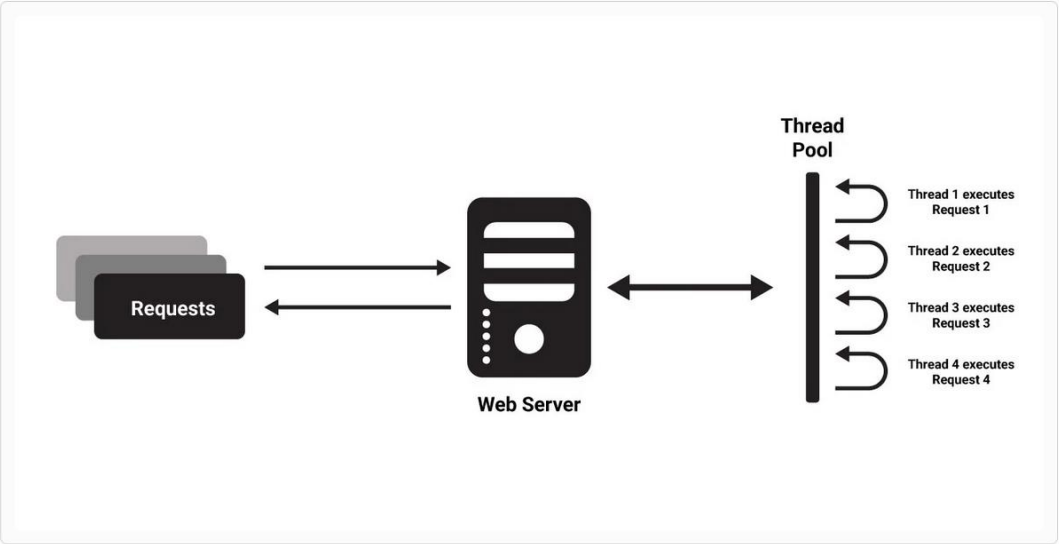
NodeJS Process Model

5. Traditional Web Server Model:

In traditional web server models, like those using Apache or Nginx, each client connection typically spawns a new thread or process. This can be resource-intensive when dealing with a large number of connections.

A thread is freed only when the request is processed, a response is returned, and the thread is returned back to the thread pool. This makes the traditional web server model synchronous and blocking.

Node.js, in contrast, handles connections on a single thread using non-blocking operations, making it more efficient for tasks involving many simultaneous connections.

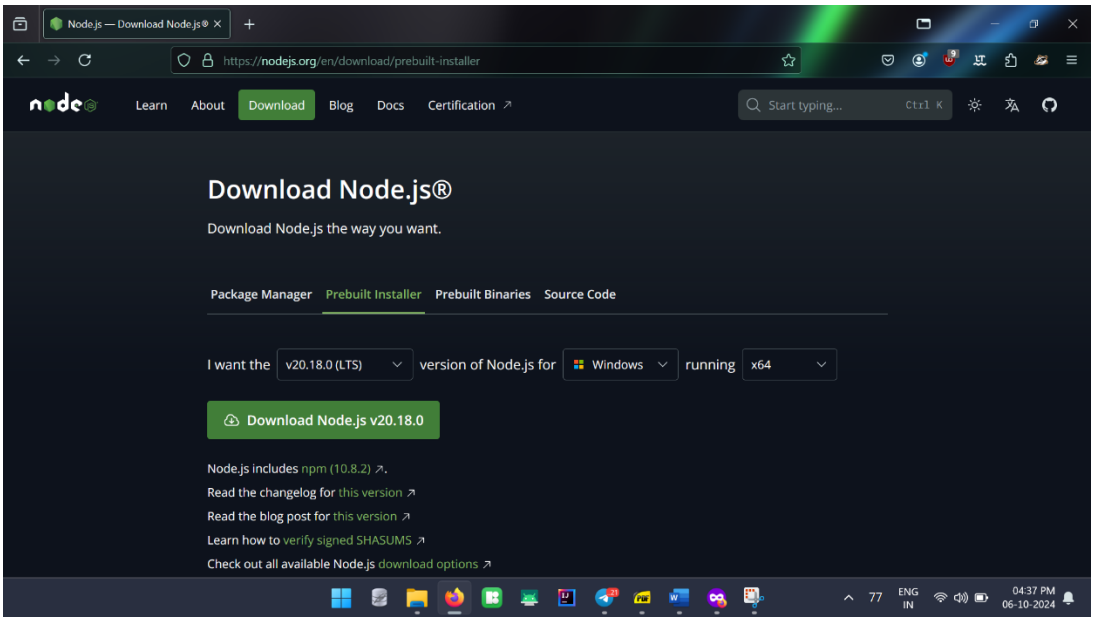


Traditional Web Server Model

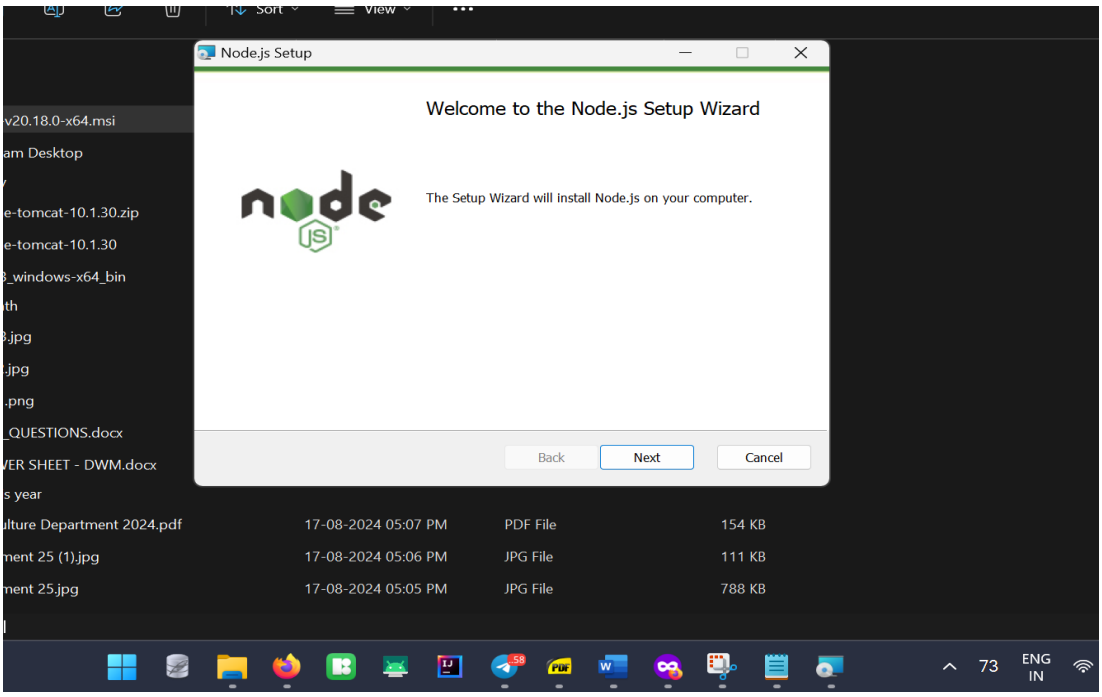
6. Installation of Node.js

Steps to install node.js

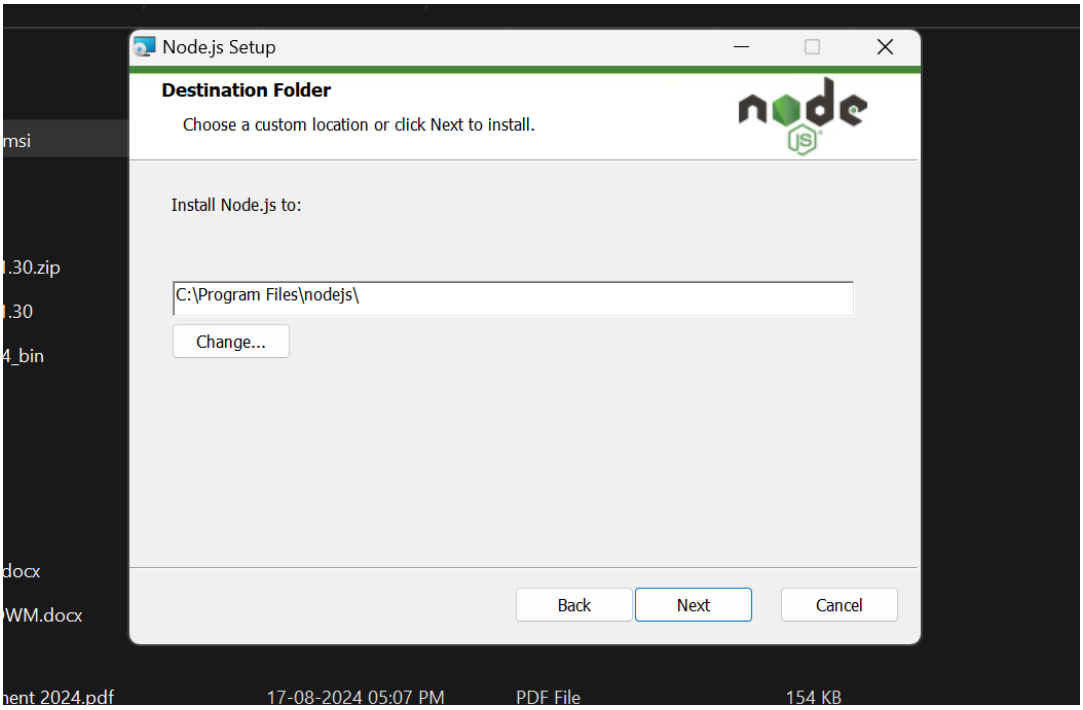
1. Downloading the Node.js '.msi' installer. Visit the official Node.js website. Visit the official Node.js website **https://nodejs.org/en/download/**



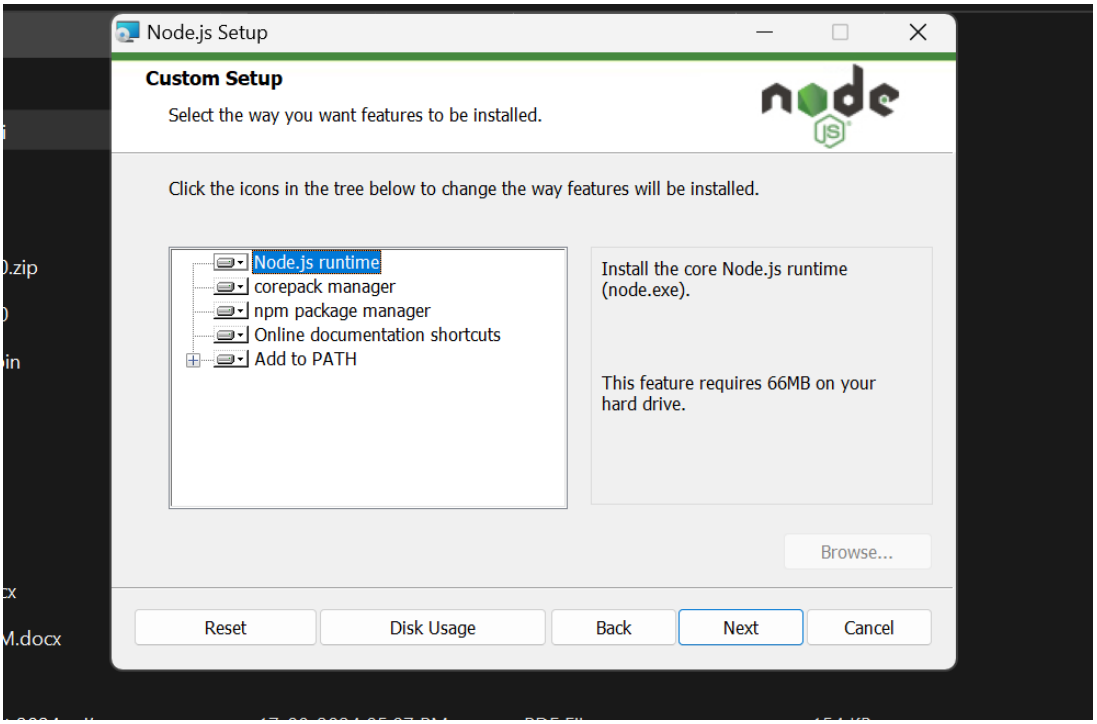
2. Double click on the .msi installer.The Node.js Setup wizard will open.



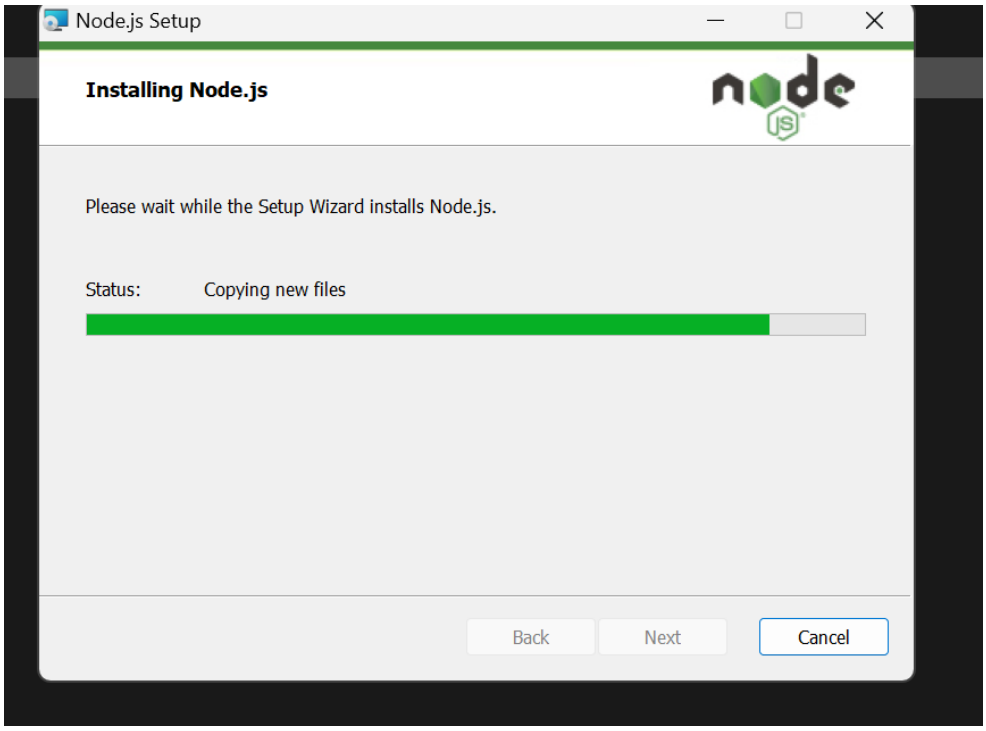
3. Select Destination folder for installation.



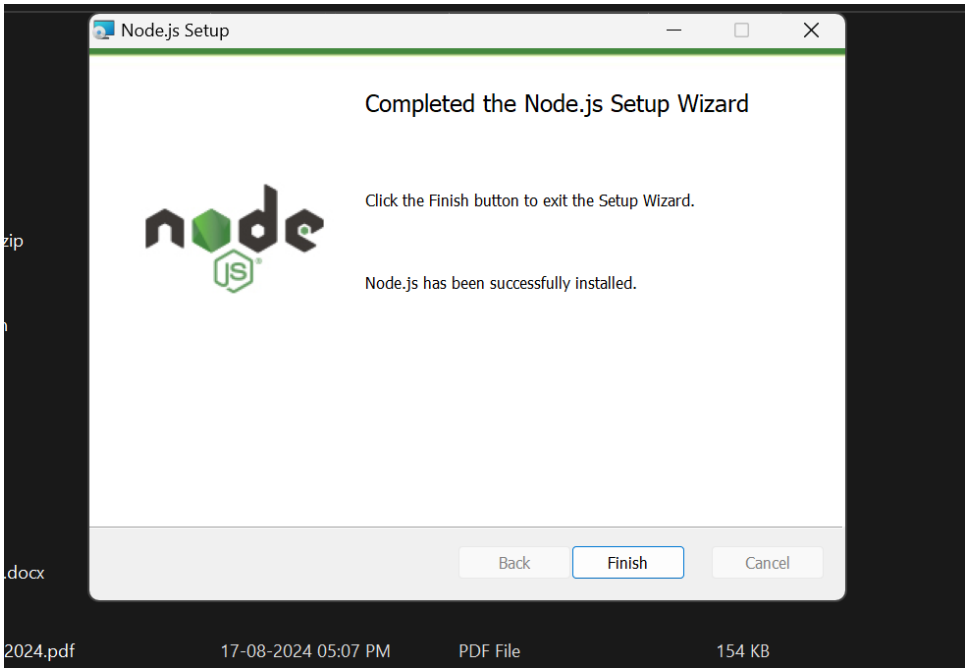
4. Select Custom Setup and Select “Next” The installer may prompt you to “install tools for native modules”



5. Installing Node.js. Do not close or cancel the installer until the install is complete



6. Complete the Node.js Setup Wizard. Click “Finish



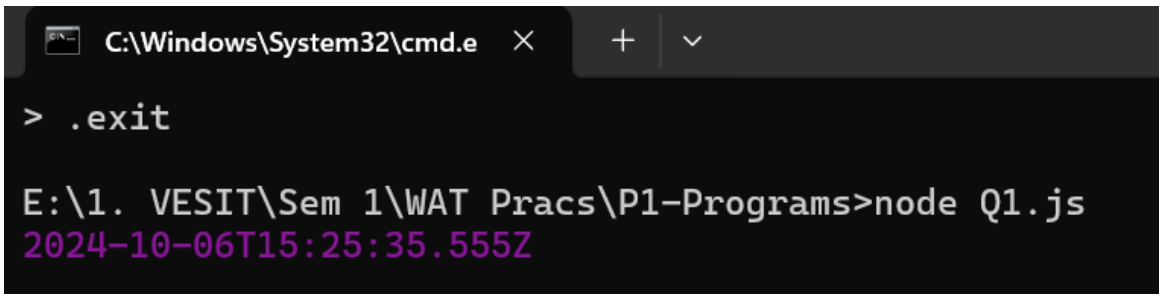
Program 1:

Print Today's date and time using REPL

Source Code:

```
const currentDate = new Date();

console.log(currentDate);
```

Output:



Program 2:

Write a program to Print the given pattern

12345

1234

123

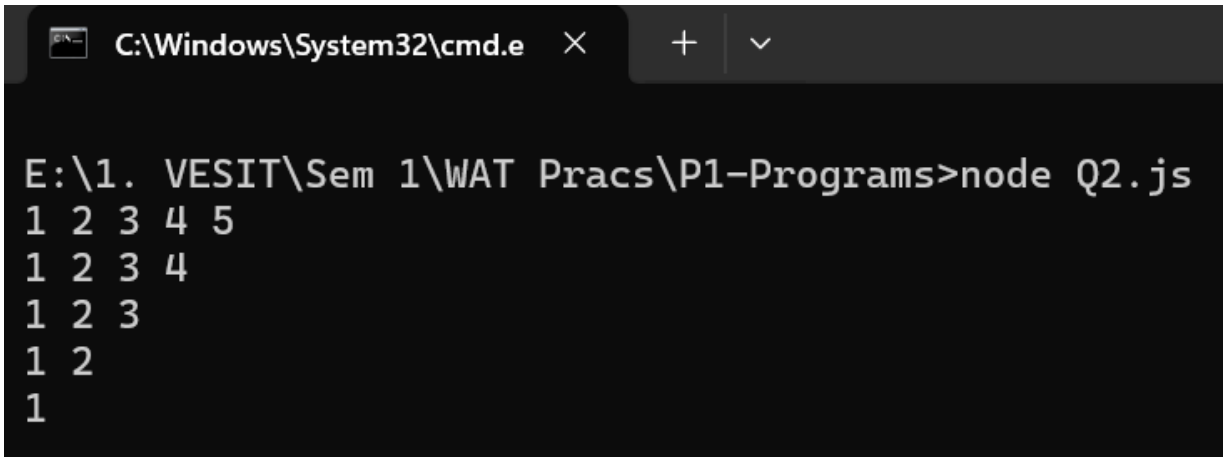
12

1

Source Code:

```
function printPattern(rows) {
for (let i = rows; i >= 1; i--) {
let rowPattern = '';
for (let j = 1; j <= i; j++) {
rowPattern += j + ' ';
}
console.log(rowPattern);
}
}
const numberOfRows = 5;
printPattern(numberOfRows);
```

Output:



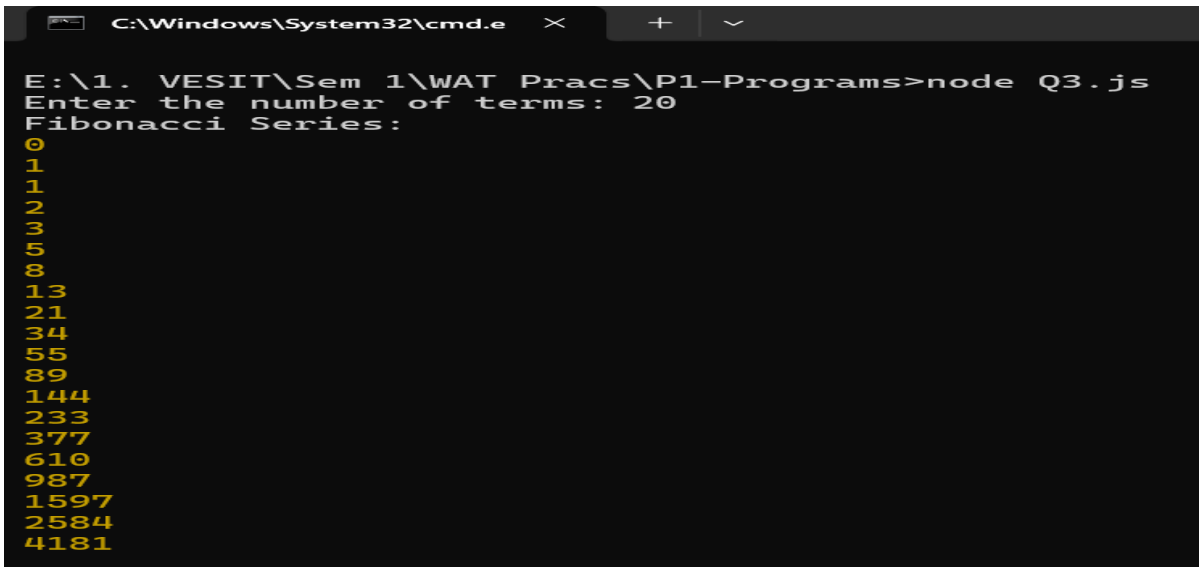
Program 3:

Write a Program to print first 20 Fibonacci numbers (take input through command line)

Source Code:

```
const prompt = require("prompt-sync")();

var number = prompt('Enter the number of terms: ');

let n1 = 0, n2 = 1, nextTerm;

console.log('Fibonacci Series:');

for (let i = 1; i <= number; i++) {

   console.log(n1);

   nextTerm = n1 + n2;

   n1 = n2;

   n2 = nextTerm;

}
```

Output:



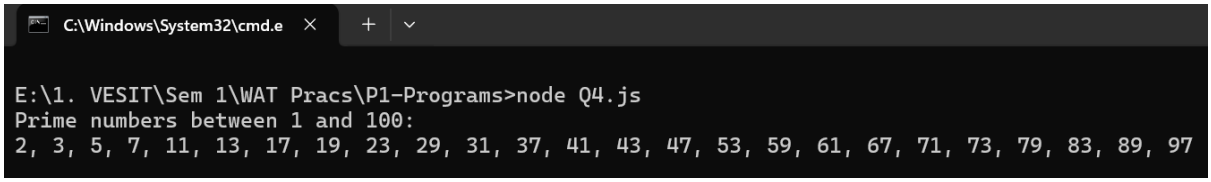
Program 4:

Write a program to print prime numbers between 1 to 100.

Source Code:

```
function isPrime(num) {
    if (num <= 1) return false;
    if (num <= 3) return true;
    for (let i = 2; i <= Math.sqrt(num); i++) {
        if (num % i === 0) {
            return false;
        }
    }
    return true;
}
console.log('Prime numbers between 1 and 100:');
const primeNumbers = [];
for (let i = 1; i <= 100; i++) {
    if (isPrime(i)) {
        primeNumbers.push(i);
    }
}
console.log(primeNumbers.join(', '));
```

Output:



Conclusion:

From this practical I have learned about the basics of Node.js and its key aspects, advantages and disadvantages. Also executed some basic javascript code using REPL aka (Node Shell).

Name of Student: Shaikh Farhan Ahmed			
Roll Number: 50		**LAB Assignment Number: 2**	
Title of LAB Assignment: Implementation of node.js modules			
DOP: 07 – 10 – 2024		**DOS: 11 – 10 – 2024**	
CO Mapped: CO1	**PO Mapped: PO3, PO5, PSO3, PSO3**	**Faculty Signature:**	**Marks:**

Practical: 2

AIM : Implementation of node js modules

Description:

1) What are modules in node.js:
In simple terms, a module is a piece of reusable JavaScript code. It could be a `.js` file or a directory containing `.js` files. You can export the content of these files and use them in other files.

Modules help developers adhere to the DRY (Don't Repeat Yourself) principle in programming. They also help to break down complex logic into small, simple, and manageable chunks.

2) Types of node modules:

There are three main types of Node modules that you will work with as a Node.js developer. They include the following.

- Built-in modules
- Local modules
- Third-party modules

Built-in Modules

Node.js comes with some modules out of the box. These modules are available for use when you install Node.js. Some common examples of built-in Node modules are the following:

- http
- url
- path
- fs
- os

you can use the built-in modules with the syntax below.

```
const someVariable = require('nameOfModule')
```

You load the module with the `require` function. You need to pass the name of the module you're loading as an argument to the `require` function.

Local Modules

When you work with Node.js, you create local modules that you load and use in your program. Let's see how to do that. Create a simple `sayHello` module. It takes a `userName` as a parameter and prints "hello" and the user's name.

```
function sayHello(userName) {

console.log(`Hello ${userName}!`) }

module.exports = sayHello
```

How to load your local modules

You can load your local modules and use them in other files. To do so, you use the **`require`** function as you did for the built-in modules.

But with your custom functions, you need to provide the path of the file as an argument. In this case, the path is `'./sayHello'` (which is referencing the `sayHello.js` file).

```
const sayHello = require('./sayHello')

sayHello("farhan") // Hello Farhan
```

First, you need to create the function. Then you export it using the syntax `module.exports`. It doesn't have to be a function, though. Your module can export an object, array, or any data type.

Third-Party Modules

A cool thing about using modules in Node.js is that you can share them with others. The Node Package Manager (NPM) makes that possible. When you install Node.js, NPM comes along with it.

With NPM, you can share your modules as packages via the NPM registry. And you can also use packages others have shared.

How to use third-party packages

To use a third-party package in your application, you first need to install it. You can run the command below to install a package.

```
npm install <name-of-package>
```

For example, there's a package called `capitalize`. It performs functions like capitalizing the first letter of a word.

Running the command below will install the capitalize package:

```
npm install capitalize
```

To use the installed package, you need to load it with the `require` function.

```
Const capitalize = require('capitalize)
```

And then you can use it in your code, like this for example:

```
const capitalize = require('capitalize')

console.log(capitalize("hello")) // Hello
```

Programs:

1. Built in Modules:

i) Write a program to print information about the computer's operating system using the OS module (use any 5 methods).

Source Code:

```
const os = require('os');

console.log('Operating System Information');

console.log('Platform : ${os.Platform()}');

console.log('Architecture : ${os.arch()}');

console.log('CPU Cores : ${os.cpus().length}');

console.log('Total memory (bytes) : ${os.totalmem()}');

console.log('Free memory (bytes) : ${os.freemem()}')
```

Output:

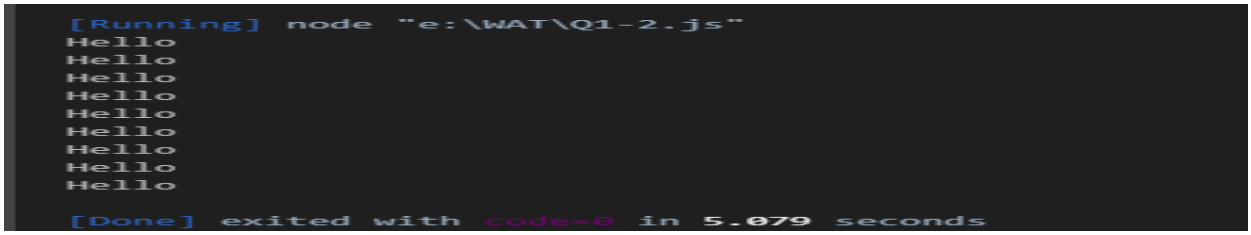
```
[Running] node "e:\WAT\Q1-1.js"
Operating System Information
Platform : ${os.Platform()}
Architecture : ${os.arch()}
CPU Cores : ${os.cpus().length}
Total memory (bytes) : ${os.totalmem()}
Free memory (bytes) : ${os.freemem()}
```

ii) Print "Hello" every 500 milliseconds using the Timer Module. The message should be printed exactly 10 times.
Use SetInterval ,ClearInterval and SetTimeout methods.

Source Code:

```
let count = 0;
const intervalId = setInterval(() => {
console.log();
count += 1;
if (count === 10) {
    clearInterval(intervalId);
  }
}, 500);
```

Output:



2. Custom Modules.

i) create a Calculator Node.js Module with functions add, subtract and multiply,Divide. And use the Calculator module in another Node.js file.

Source Code:

Creating a node js module with calculator operations.

```
// calculator.js

function add(a, b) {
```

```
    return a + b;
    }
    function subtract(a, b) {
    return a - b;
     }
    function multiply(a, b) {
    return a * b;
    }
    function divide(a, b) {
    if (b === 0) {
    throw new Error("Cannot divide by zero");
    }
    return a / b;
    }
module.exports = {addition,subtract,multiply,divide}
```

Creating separate Node.js file to use our first file with all arithmetic module:

```
//Q3.js
const calculator = require("./calculator");
const a = 20;
const b = 6;
console.log(`Add: ${calculator.add(a, b)}`);
console.log(`Subtract: ${calculator.subtract(a, b)}`);
console.log(`Multiply: ${calculator.multiply(a, b)}`);
console.log(`Divide: ${calculator.divide(a, b)}`)
```

Output:

```
PS E:\1. VESIT\Practicals\WAT\P2> node Q3.js
Subtract: 14
Multiply: 120
Divide: 3.3333333333333335
```

ii) Create a circle module with functions to find the area and perimeter of a circle and use it.

Source Code:

Creating Custom module to encapsulate perimeter and area logic.

area_perimeter.js

```
const PI = Math.PI;

function area(radius) {

return PI * radius * radius;

}

function perimeter(radius) {

return 2 * PI * radius;

}

module.exports = {area,perimeter}
```

Using the custom module inside another js file.

```
const calculate = require("./area_perimeter")

const radius = 12;

console.log(`Area of the circle: ${calculate.area(radius)}`);

console.log(`Perimeter of the circle: ${calculate.perimeter(radius)}`);
```

Output:

```
PS E:\1. VESIT\Practicals\WAT\P2> node Q4.js
Area of the circle: 452.3893421169302
Perimeter of the circle: 75.39822368615503
PS E:\1. VESIT\Practicals\WAT\P2>
```

Conclusion:

In this demonstration, I explored the fundamental concepts of Node.js modules, including core, local, and third-party modules. By utilizing the built-in os module, I retrieved essential system information, and through the Timer module, I implemented a simple interval-based greeting. Additionally, I created custom modules for a calculator and a circle, showcasing how to encapsulate functionalities in reusable components. This modular approach improves code organization and enhances maintainability and scalability, which are crucial for developing robust applications in Node.js.

<table>
<tr><td colspan="4">Name of Student: Shaikh Farhan Ahmed</td></tr>
<tr><td colspan="2">Roll Number: 50</td><td colspan="2">LAB Assignment Number: 3</td></tr>
<tr><td colspan="4">Title of LAB Assignment: Implementing Asynchronous Array Operations in Node.js Through Event-Driven Programming.</td></tr>
<tr><td colspan="2">DOP: 14 – 10 – 2024</td><td colspan="2">DOS: 18 – 10 – 2024</td></tr>
<tr><td>CO Mapped:
CO1</td><td>PO Mapped:
PO3, PO5,
PSO1,PSO2</td><td>Faculty Signature
:</td><td>Marks:</td></tr>
</table>

Practical No 3

AIM: Implementing Asynchronous Array Operations in Node.js Through Event-Driven Programming with EventEmitter.

Theory:

Events in Node.js

In Node.js, events are a core part of the asynchronous programming model. The event-driven architecture allows Node.js to handle multiple operations without blocking the execution of the program. Here's a breakdown of key concepts related to events in Node.js:

1. EventEmitter: This is a class provided by the `events` module. It allows you to create objects that can emit events and register listeners (callback functions) that respond to those events.
2. Emitting Events: You can emit an event using the `emit` method of an `EventEmitter` instance. This triggers all registered listeners for that event.
3. Listening to Events: You can register a listener for an event using the `on` method. When the event is emitted, the listener is executed.
4. Removing Listeners: You can remove listeners using the `removeListener` or `off` methods if you no longer want a listener to respond to an event.
5. Built-in Event Emitters: Many built-in Node.js modules, like `http`, `fs`, and `net`, are built around the event-driven model and utilize the `EventEmitter` class.

Events are crucial for building scalable applications with Node.js, enabling efficient handling of asynchronous operations.

EventEmitter Methods:

1. *emitter.addListener(event, listener)*

Adds a listener to the end of the listeners array for the specified event. No checks are made to see if the listener has already been added.

2. *emitter.on(event, listener)*

Adds a listener to the end of the listeners array for the specified event. No checks are made to see if the listener has already been added. It can also be called as an alias of emitter.addListener()

3. *emitter.once(event, listener)*

Adds a one-time listener for the event. This listener is invoked only the next time the event is fired, after which it is removed.

4. *emitter.removeListener(event, listener)*

Removes a listener from the listener array for the specified event. Caution: changes array indices in the listener array behind the listener.

5. *emitter.removeAllListeners([event])*

Removes all listeners, or those of the specified event.

6. *emitter.setMaxListeners(n)*

By default EventEmitters will print a warning if more than 10 listeners are added for a particular event.

7. *emitter.getMaxListeners()*

Returns the current maximum listener value for the emitter.

8. *emitter.listeners(event)*

Returns a copy of the array of listeners for the specified event.

9. *emitter.emit(event[, arg1][, arg2][, ...])*

Raise the specified events with the supplied arguments.

10. *emitter.listenerCount(type)*

Returns the number of listeners listening to the type of event.

Programs:

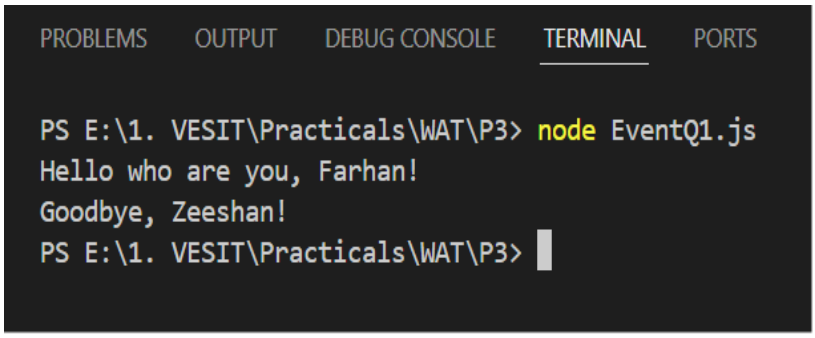
1. **Create an application to demonstrate various Node.js Events methods in event emitter class.**

   Source Code:

```
const EventEmitter = require('events');

const myEmitter = new EventEmitter();


// Define a listener for the 'greet' event

myEmitter.on('greet', (name) => {

     console.log(`Hello who are you, ${name}!`);

});


myEmitter.emit('greet', 'Farhan');

// Add a one-time listener for the 'farewell' event

myEmitter.once('farewell', (name) => {

     console.log(`Goodbye, ${name}!`);

});
```

```
// Emit the 'farewell' event

myEmitter.emit('farewell', 'Zeeshan');


// Emit the 'farewell' event again (no output, since it's a
one-time listener)

myEmitter.emit('farewell', 'Charlie');
```

Output:



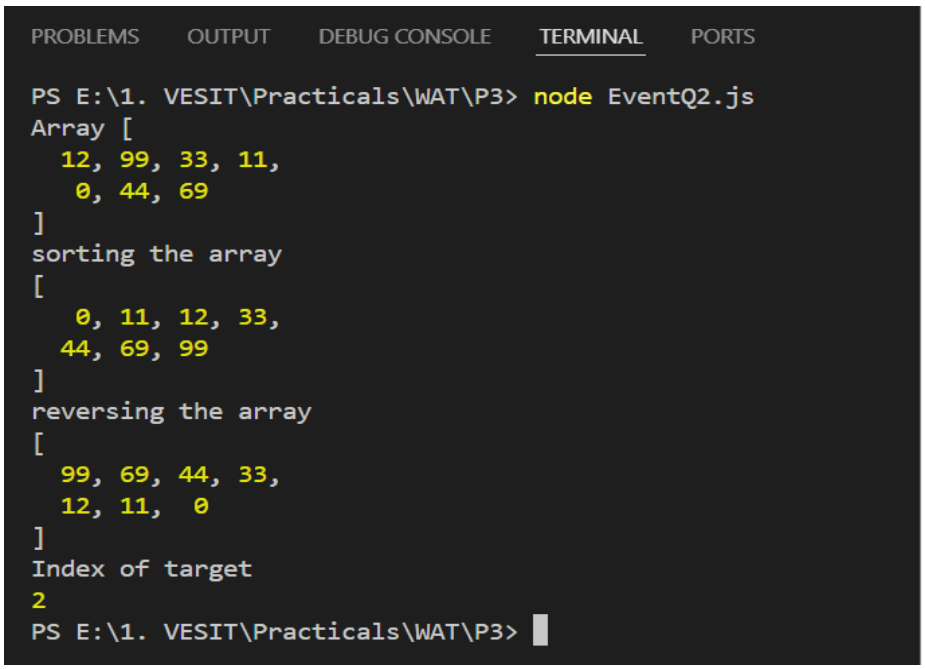
2. **Create functions to sort, reverse and search for an element in an array. Register and trigger these functions using events.**

Source Code:

```
const EventEmitter = require('events');

const myEmitter = new EventEmitter();

let nums = [12,99,33,11,0,44,69];

let sortArray = (nums) => console.log(nums.sort());

let reverseArray = (nums) => console.log(nums.reverse());

let searchArrayByIndex =
(nums, index) => console.log(nums.indexOf(index));
```

```
myEmitter.on('sort_event', sortArray);

myEmitter.on('reverse_event', reverseArray);

myEmitter.on('search_event', searchArrayByIndex);

console.log("Array", nums)

console.log("sorting the array")

myEmitter.emit('sort_event', nums);

console.log("reversing the array")

myEmitter.emit('reverse_event', nums);

console.log("Index of target")

myEmitter.emit('search_event', nums, 44);
```

Output:



Conclusion:

In Node.js, events are essential for managing asynchronous operations and enabling communication across various components of an application. The `EventEmitter` class is a fundamental part of this system, allowing objects to emit and respond to events.

Demonstration of EventEmitter:

We instantiated an `EventEmitter` (named `myEmitter`) and registered listeners for different events. We showcased the usage of methods such as `addListener`, `on`, `once`, `removeListener`, `removeAllListeners`, `setMaxListeners`, `getMaxListeners`, `listeners`, `emit`, and `listenerCount`.

Array Operations Using Events:

1. We defined functions to sort, reverse, and search for elements within an array.
2. These functions were registered as listeners for specific events.
3. We triggered the functions using `emit` to perform the array operations asynchronously.

<table>
<tr><td colspan="4">Name of Student: Shaikh Farhan Ahmed</td></tr>
<tr><td colspan="2">Roll Number: 50</td><td colspan="2">LAB Assignment Number: 04</td></tr>
<tr><td colspan="4">Title of LAB Assignment: To study and demonstrate Node.js timer functions.</td></tr>
<tr><td colspan="2">DOP: 23 – 10 – 2024</td><td colspan="2">DOS: 31 – 10 – 2024</td></tr>
<tr><td>CO Mapped:
CO1</td><td>PO Mapped:
PO3, PO5, PSO1,
PSO2</td><td>Faculty Signature :</td><td>Marks:</td></tr>
</table>

Practical No 4

Aim: To study and demonstrate Node.js timer functions.

Theory

Timer Modules functions:

In Node.js, the timer module provides functions to schedule the execution of code after a specified delay or at regular intervals. This is commonly used for tasks such as setting timeouts and intervals. The primary functions you'll encounter in this module are setTimeout(), setInterval(), and setImmediate().

1. **setTimeout():**
   setTimeout() can be used to schedule code execution after a designated amount of milliseconds.
   setTimeout() accepts a function to execute as its first argument and The millisecond delay defined as a number as the second argument. Additional arguments may also be included and these will be passed on to the function.

Example:

```
setTimeout(() => {
//run something
}, delaytime in ms)
function myFunc(arg) {
console.log(`arg was => ${arg}`);
}
setTimeout(myFunc, 1500, 'funky');
```

2. setInterval():

If there is a block of code that should execute multiple times, setInterval() can be used to execute that code.
setInterval() takes a function argument that will run an infinite number of times with a given millisecond delay as the second argument.

Example:

```
setInterval(() => {
//run something
}, delaytime in ms)

// Executed after every 1000 milliseconds
// from the start of the program
setInterval(function A() {
return console.log('Hello World!');
}, 1000);
// Executed right away
console.log('Executed before A...');
```

3. setImmediate():

When you want to execute some piece of code asynchronously, but as soon as possible, one option is to use the setImmediate() function provided by Node.js.

Example:

```
setImmediate(() => {
//run something
})
setImmediate(function(){
console.log('Hello world 4');
// It's like get to last and be take care of first
// but always after of .nextTick and before of setInterval(, 0)
});
```

clearImmediate(), clearInterval(), clearTimeout():

1] What can be done if a Timeout or Immediate object needs to be cancelled?

2] setTimeout(), setImmediate(), and setInterval() return a timer object that can be used to reference the set Timeout or Immediate object.

3] By passing said object into the respective clear function, execution of that object will be halted completely. The respective functions are clearTimeout(), clearImmediate(), and clearInterval().

Create an application to demonstrate Node.js timer functions.
Write functions to

1) Print multiplication table for a number

Source Code:

```
function printTable(number){
  for (let i = 1; i <= 10; i++) {
      console.log(number * i)
  }
}

 //Timer Function
  process.nextTick(() => printTable(12));
```

Output:

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS
PS D:\Farhan WATL> node Print_table_Timer.js
12
24
36
48
60
72
84
96
108
120
PS D:\Farhan WATL>
```

2. Find whether the entered year is leap year or not

Source Code:

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

 function isLeapYear(year){
  if(year %4 === 0 && year % 100 !== 0){
      return true;
  }
  else if (year % 400 === 0){
      return true;
  }
  else {
      return false;
  }
}

const year = prompt('Enter year to check whether it is leap year or not?
');

//SetImmediate
  setImmediate(() => {
   console.log(isLeapYear(year))
 });
```

Output:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS D:\Farhan WATL> node .\IsLeapYear_Timer.js
Enter year to check whether it is leap year or not? 2023
false
PS D:\Farhan WATL> node .\IsLeapYear_Timer.js
Enter year to check whether it is leap year or not? 2000
true
PS D:\Farhan WATL>
```

3. Check whether the given string is palindrome or not

Source Code:

```
const prompt = require('prompt-sync')();
const input = prompt('Enter String to check palindrome: ');

function isPalindrome(input){
originalString = input
const revString = input.split('').reverse().join('');
if(revString == originalString){
 console.log("String is palindrome");
}
else{
 console.log("String is not a palindrome");
 }
}

setTimeout(() => {
 isPalindrome(input)
},1000);
```

Output:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS D:\Farhan WATL> node .\isPalindrome_Timer.js
Enter String to check palindrome: farhan
String is not a palindrome
PS D:\Farhan WATL> node .\isPalindrome_Timer.js
Enter String to check palindrome: madam
String is palindrome
PS D:\Farhan WATL>
```

Conclusion:

In this practical application, we utilized various Node.js timer functions (process.nextTick, setImmediate, setInterval, and setTimeout) to perform different tasks concurrently. These timer functions are essential for managingasynchronous operations in Node.js, allowing for efficient handling of tasks without blocking the event loop.

By combining the functionalities of printing a multiplication table, checking leap years, and determining if a string is a palindrome, we demonstrated the versatility of Node.js in handling asynchronous tasks.

<table>
<tr><td colspan="4">Name of Student: Shaikh Farhan Ahmed</td></tr>
<tr><td colspan="2">Roll Number: 50</td><td colspan="2">LAB Assignment Number: 05</td></tr>
<tr><td colspan="4">Title of LAB Assignment: Using File Handling demonstrate all basic file operations (Open, Read, Write, Append, delete).write a menu driven program for the same.</td></tr>
<tr><td colspan="2">DOP: 23 – 10 – 2024</td><td colspan="2">DOS: 15 – 11 – 2024</td></tr>
<tr><td>CO Mapped:
CO1</td><td>PO Mapped:
PO3, PO5,
PSO1, PSO2</td><td>Faculty Signature:</td><td>Marks:</td></tr>
</table>

Practical No 5

Aim: Using File Handling demonstrate all basic file operations (Open, Read, Write, Append, delete).write a menu driven program for the same

Theory:

Synchronous and Asynchronous:

Synchronous is a blocking architecture and is best for programming reactive systems. As a single-thread model, it follows a strict set of sequences, which means that operations are performed one at a time, in perfect order. While one operation is being performed, other operations' instructions are blocked.
The completion of the first task triggers the next, and so on.

Asynchronous is non-blocking IO , which means it doesn't block further execution while one or more operations are in progress. With async programming, multiple related operations can run concurrently without waiting for other tasks to complete.
Multiple developers can work on projects simultaneously in a low-code platform, which helps accelerate the process of building apps.

fs Module in JS:

The fs (File System) module in Node.js provides an API for interacting with the file system. It allows you to perform operations such as reading, writing, updating, and deleting files and directories, which are essential for server-side applications and scripts.

Uses:

- Read Files
- Write Files
- Append Files
- Close Files
- Delete Files

Functions with explanation and syntax:

1. **fs.open() :**
   Explanation:
   The fs.open() method is used to create, read, or write a file.
   Syntax:
   fs.open(path, flags, mode, callback)

2. **fs.read() :**
   Explanation:
   The fs.read() method is used to read the file specified by fd. This method reads the entire file into the buffer.
   Syntax:
   fs.read(fd, buffer, offset, length, position, callback

3. **fs.writeFile() :**
   Explanation:
   This method will overwrite the file if the file already exists. The fs.writeFile() method is used to asynchronously write the specified data to a file. By default, the file would be replaced if it exists.
   Syntax:
   fs.writeFile(path, data, options, callback)

4. **fs.appendFile() :**
   Explanation:
   The fs.appendFile() method is used to synchronously append the data to the file.
   Syntax:
   fs.appendFile(filepath, data, options, callback);

5. **fs.unlink() :**
   Explanation:
   The fs.unlink() method is used to remove a file or symbolic link from the filesystem

   Syntax:
   fs.unlink(path, callback)

6. **fs.close() :**
   Explanation:
   The fs.close() method is used to asynchronously close the given file descriptor thereby clearing the file that is associated with it.
   Syntax:
   fs.close(fd, callback)

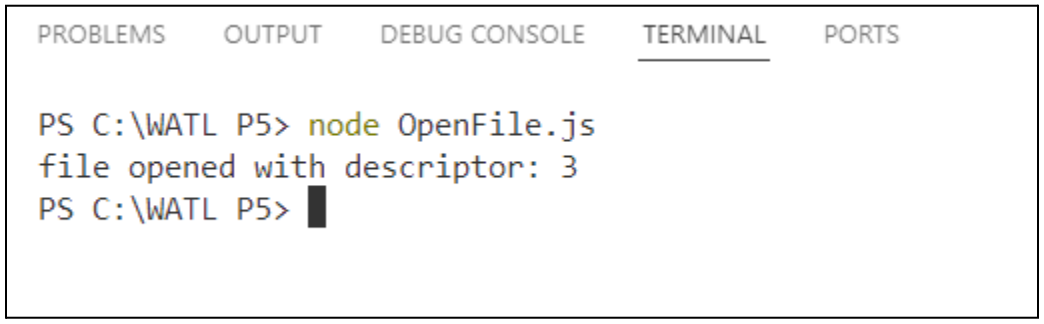
Program for each function with Output (Before and After Execution)

1. Open File.

Source Code:

```
const fs = require('fs');

const filename = "Data.txt"

fs.open(filename,'r',(err, fd) =>{
    if(err) throw err;

    console.log (`file opened with descriptor: ${fd}`)
})
```

Output:



2. Write File.

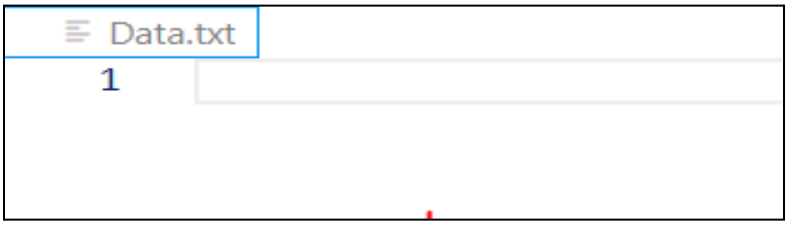
Source Code:

```
const fs = require('fs');
const filename = "Data.txt"

const content = "This line is getting written into the
file."
```

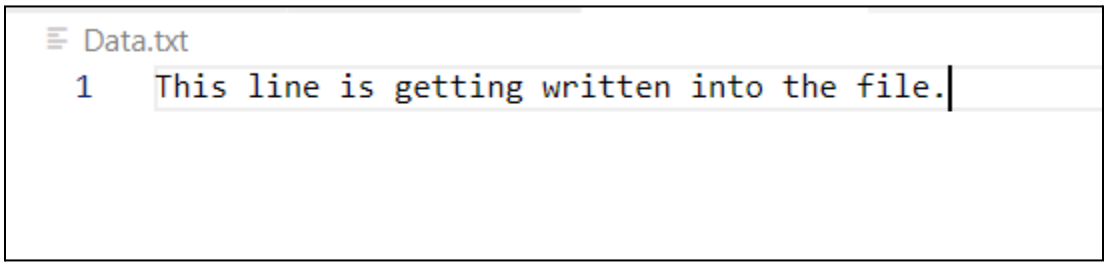
```
fs.writeFile(filename, content, (err, fd) =>{
    if(err) throw err;
    console.log (`Line written successfully`)
})
```

Output:

Before



After



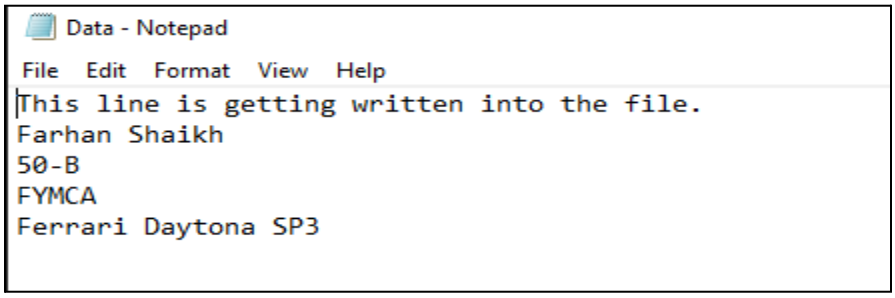
3. Read File

Source Code:

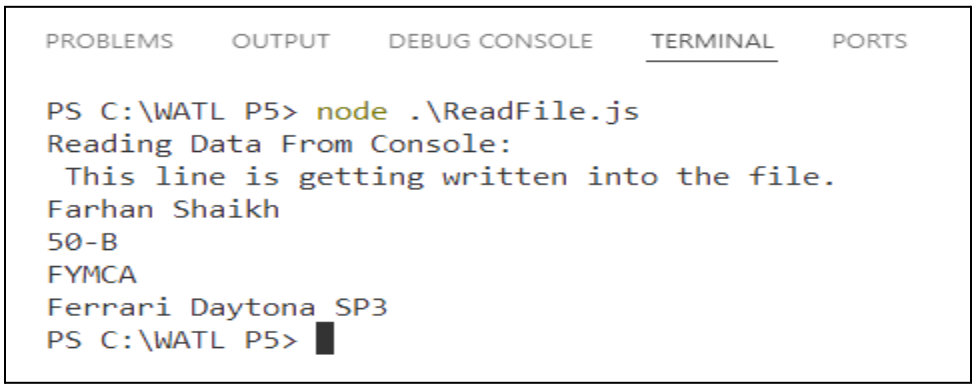
```
const fs = require('fs');
const filename = "Data.txt"

fs.readFile(filename, function (err, data) {
    if (err) {
        return console.error(err);
    }
    console.log("Reading Data From Console:\n " + data.toString());
});
```

Output:

Before



After



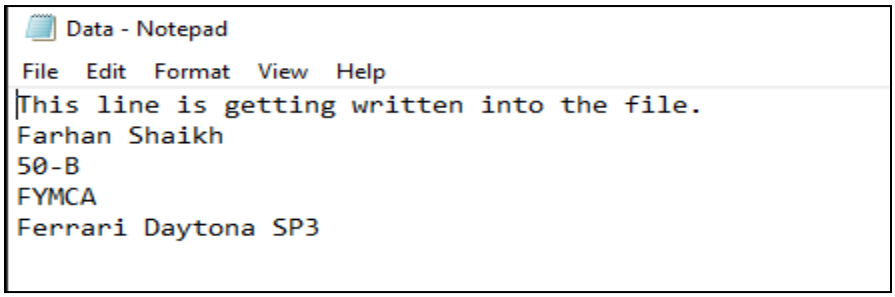
4. Append File

Source Code:

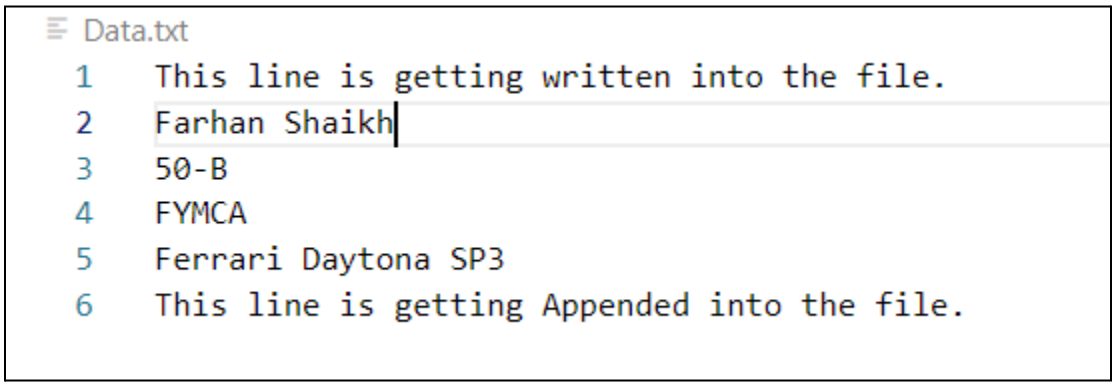
```
const fs = require('fs');

const filename = "Data.txt"
const content = "\nThis line is getting Appended into the file."

fs.appendFile(filename, content, (err, fd) =>{
    if(err) throw err;
    console.log (`Line appended successfully`)
})
```

Output:

Before



After
Appending a line to a file.



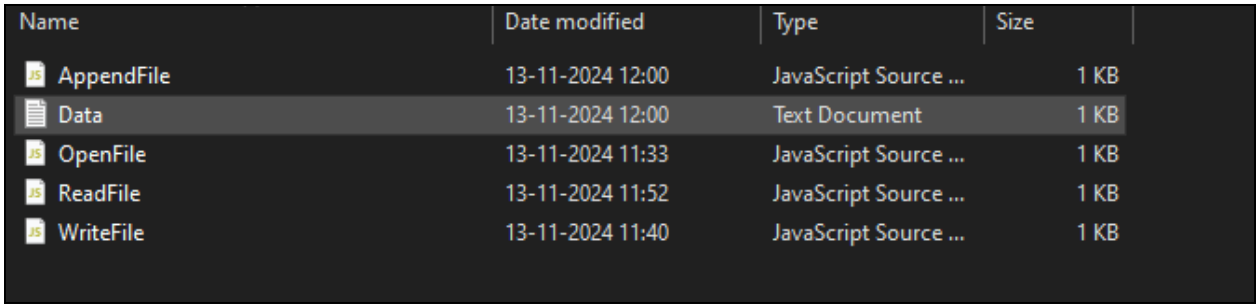
5. Delete File.

Source Code:

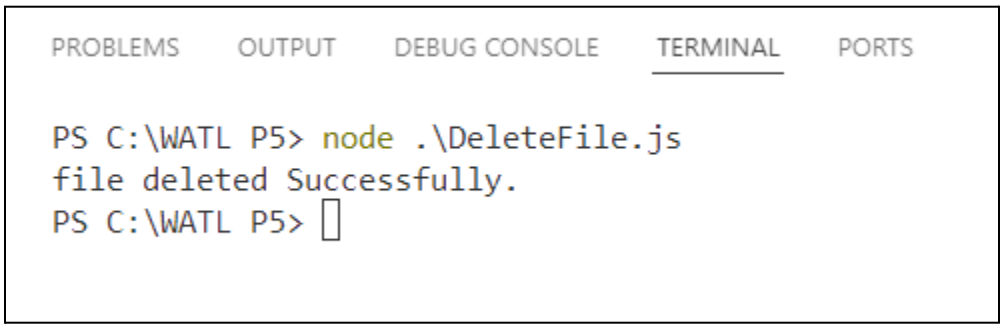
```
const fs = require('fs');
const filename = "Data.txt"

fs.unlink(filename, (err, fd) =>{
    if(err) throw err;

    console.log (`file deleted Successfully.`)
})
```

Output:

Before



Name	Date modified	Type	Size
AppendFile	13-11-2024 12:00	JavaScript Source ...	1 KB
Data	13-11-2024 12:00	Text Document	1 KB
OpenFile	13-11-2024 11:33	JavaScript Source ...	1 KB
ReadFile	13-11-2024 11:52	JavaScript Source ...	1 KB
WriteFile	13-11-2024 11:40	JavaScript Source ...	1 KB



```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS C:\WATL P5> node .\DeleteFile.js
file deleted Successfully.
PS C:\WATL P5>
```

After Deleting the file.

Conclusion:

After executing all operations(Open, Read, Write , Append, Delete) We find that the **fs** module is a fundamental tool for file operations in Node.js, offering a range of methods to handle files and directories effectively. Its versatility and robust error handling make it essential for building reliable server-side applications.

<table>
<tr><td colspan="4">Name of Student: Shaikh Farhan Ahmed</td></tr>
<tr><td colspan="2">Roll Number: 50</td><td colspan="2">LAB Assignment Number: 06</td></tr>
<tr><td colspan="4">Title of LAB Assignment: To study and Demonstrate programs based on HTTP module.</td></tr>
<tr><td colspan="2">DOP: 11 – 11 – 2024</td><td colspan="2">DOS: 16 – 11 – 2024</td></tr>
<tr><td>CO Mapped:
CO1</td><td>PO Mapped:
PO3, PO5,
PSO1, PSO2</td><td>Faculty Signature:</td><td>Marks:</td></tr>
</table>

Practical No 6

Programs:

Q1. Create a HTTP Server and serve a HTML, CSV, JSON and PDF Files.

Source Code:

Creating a http server

```
const http = require("http");
const port = 8081
const host = "localhost";

const requestListener = function (req, res) {
     res.writeHead(200);
     res.end("My first server is running!");
};

const server = http.createServer(requestListener);
server.listen(port, host, () => {
     console.log(`Server is running on http://${host}/${port}`);
});
```

Serving HTML

```
const http = require("http");
const fs = require('fs').promises;

const port = 8081
const host = "localhost";

const requestListener = function (req, res) {
     fs.readFile(__dirname + "/page.html")
     .then(contents => {
     res.setHeader("Content-Type", "text/html");
     res.writeHead(200);
     res.end(contents);
     })
```

```
     .catch(err => {
     res.writeHead(500);
     res.end(err);
     return;
     });

};

const server = http.createServer(requestListener);
server.listen(port, host, () => {
     console.log(`Server is running on http://${host}/${port}`);
});
```

Serving CSV

```
const http = require("http");
const port = 8081
const host = "localhost";

const requestListener = function (req, res) {
     res.setHeader("Content-Type", "text/csv");
     res.setHeader("Content-Disposition",
"attachment;filename=CSV_Data.csv");
     res.writeHead(200);
     res.end()
};

const server = http.createServer(requestListener);
server.listen(port, host, () => {
     console.log(`Server is running on http://${host}/${port}`);
});
```

Serving JSON

```
const http = require("http");
const port = 8081
const host = "localhost";

const requestListener = function (req, res) {
     res.setHeader("Content-Type", "text/csv");
     res.setHeader("Content-Disposition",
"attachment;filename=CSV_Data.csv");
     res.writeHead(200);
     res.end()
```

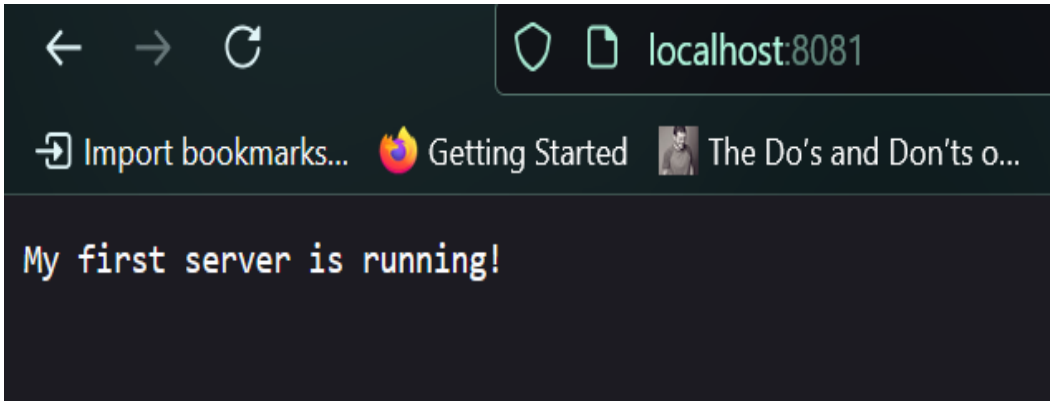
```
};
const server = http.createServer(requestListener);
server.listen(port, host, () => {
     console.log(`Server is running on http://${host}/${port}`);
});
```

Serving PDF

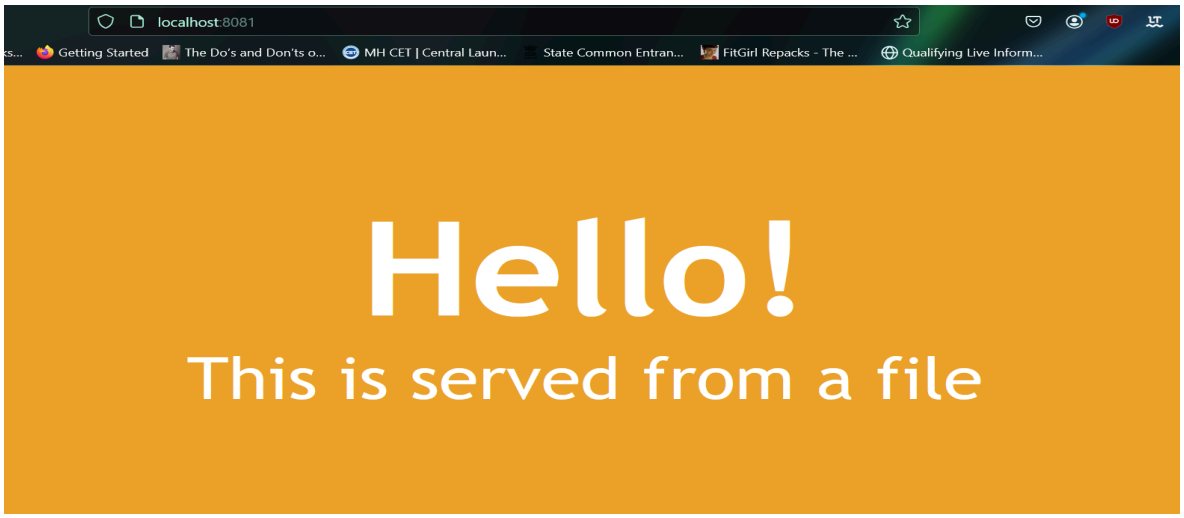
```
const http = require("http");
const port = 8081
const host = "localhost";
const fs = require("fs");

const requestListener = function (req, res) {
     res.setHeader("Content-Type", "application/pdf");
     const pdfStream = fs.createReadStream("./Sample_PDF.pdf");
     pdfStream.pipe(res);
};

const server = http.createServer(requestListener);
server.listen(port, host, () => {
     console.log(`Server is running on http://${host}/${port}`);
});
```

Output:

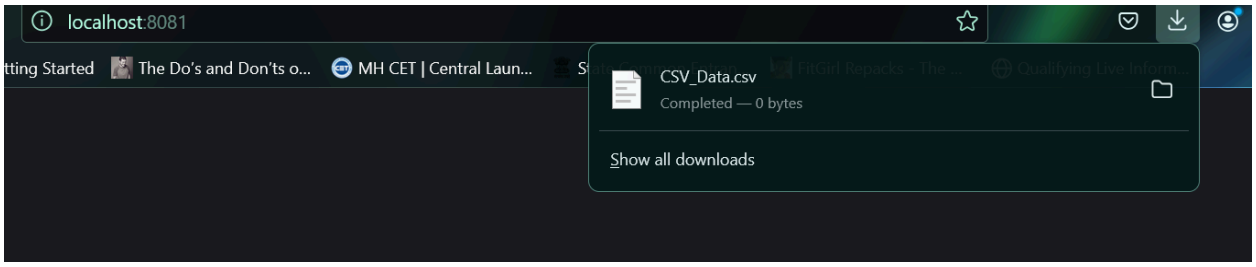
Initial Server:



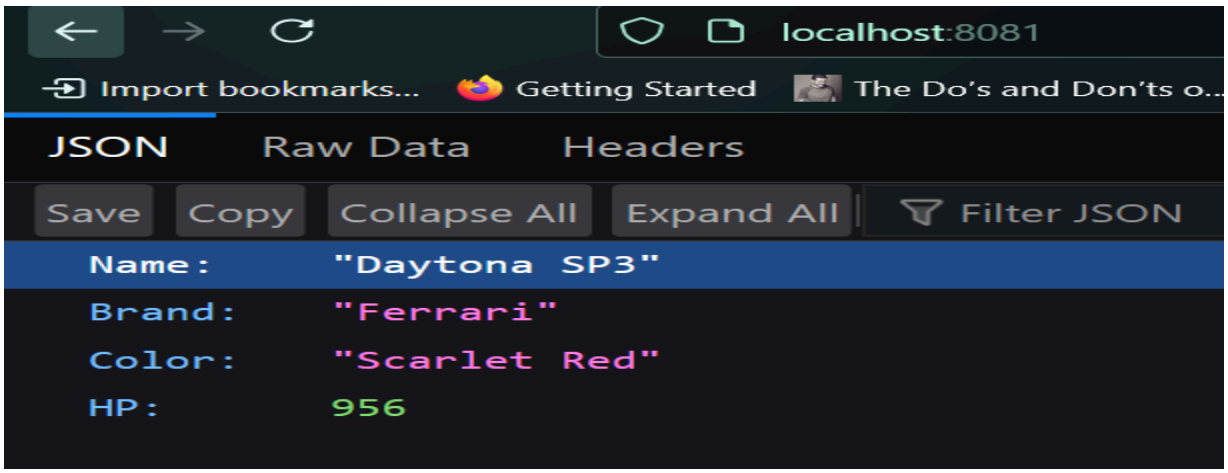
HTML Response:



CSV Response:



JSON Response:



Serving PDF

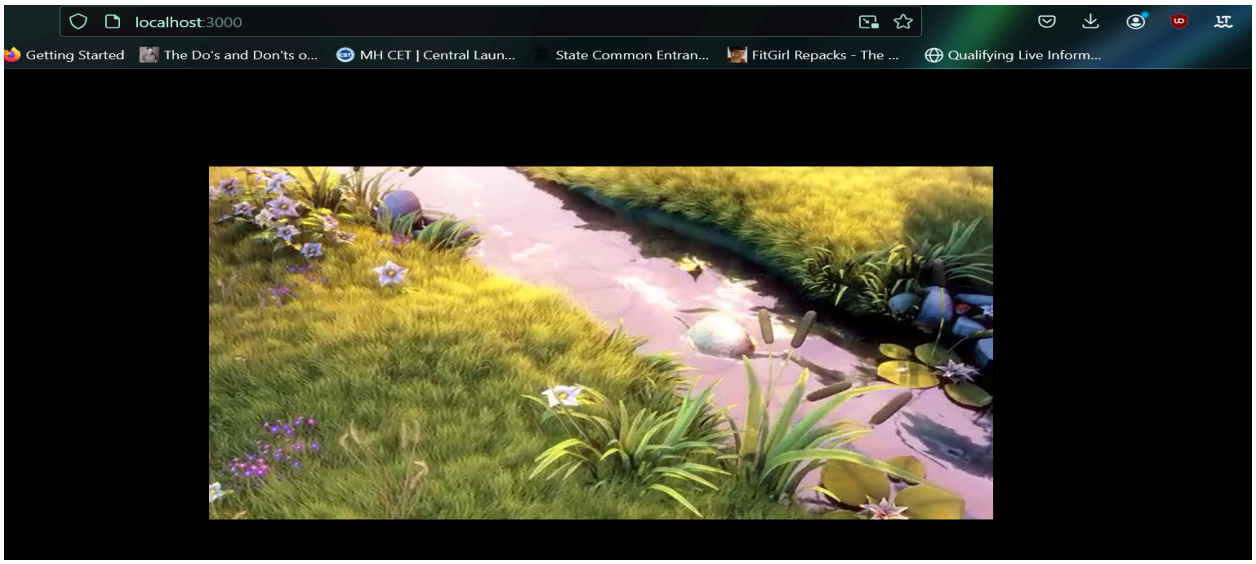

localhost:8081
Getting Started
The Do's and Don'ts o...
MH CET | Central Laun...
State Common Entran...
FitGirl Repacks - The ...
Qualifying Live Inform...
1 of 1
Automatic Zoom
Sample PDF
This is a simple PDF file. Fun fun fun.
Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Phasellus facilisis odio sed mi. Curabitur suscipit. Nullam vel nisi. Etiam semper ipsum ut lectus. Proin aliquam, erat eget pharetra commodo, eros mi condimentum quam, sed commodo justo quam ut velit. Integer a erat. Cras laoreet ligula cursus enim. Aenean scelerisque velit et tellus. Vestibulum dictum aliquet sem. Nulla facilisi. Vestibulum accumsan ante vitae elit. Nulla erat dolor, blandit in, rutrum quis, semper pulvinar, enim. Nullam varius congue risus. Vivamus sollicitudin, metus ut interdum eleifend, nisi tellus pellentesque elit, tristique accumsan eros quam et risus. Suspendisse libero odio, mattis sit amet, aliquet eget, hendrerit vel, nulla. Sed vitae augue. Aliquam erat volutpat. Aliquam feugiat vulputate nisl. Suspendisse quis nulla pretium ante pretium mollis. Proin velit ligula, sagittis at, egestas a, pulvinar quis, nisl.
Pellentesque sit amet lectus. Praesent pulvinar, nunc quis iaculis sagittis, justo quam lobortis tortor, sed vestibulum dui metus venenatis est. Nunc cursus ligula. Nulla facilisi.

Q2. Create a HTTP Server and stream a video file using piping

Source Code:

```
const http = require("http");
const fs = require("fs");
const path = require("path");

const server = http.createServer((req, res) => {
const videoPath = path.join(__dirname, "./demo.mp4");
const stat = fs.statSync(videoPath);
res.writeHead(200, {
"Content-Type": "video/mp4",
"Content-Length": stat.size,
});
const videoStream = fs.createReadStream(videoPath);
videoStream.pipe(res);
});

const port = 3000;
server.listen(port, () => {
console.log(`Server is running on http://localhost:${port}`);
});
```

Output:



Q3. Develop 3 HTML Web Pages for college home page(write about college & dept),about me and contact. Create a server and render these pages using Routing.

Source Code:

Index.html

```
<!DOCTYPE html>
<html>
<head>
<title>College Home Page</title>
</head>
<body>
   <h1>Magnum College</h1>
   <p>Welcome to our college homepage</p>
</body>
</html>
```

contactUs.html

```
<!DOCTYPE html>
<html>
<head>
<title>College Home Page</title>
</head>
<body>
   <h1>Contact Us</h1>
   <p>Feel free to connect with us regarding any queries</p>
</body>
</html>
```

about.html

```
<!DOCTYPE html>
<html>
<head>
<title>College Home Page</title>
</head>
<body>
   <h1>About!</h1>
   <p>Learn about our journey as a college</p>
</body>
</html>
```

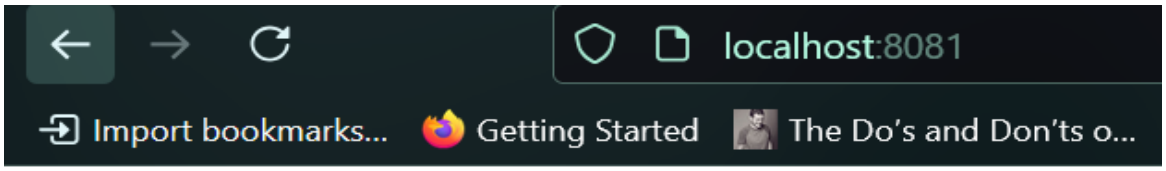
index.js

```
const http = require("http");
const fs = require('fs');
const port = 8081
const host = "localhost";

const requestListener = function (req, res) {
     const index = fs.createReadStream("./index.html")

   switch(req.url) {
     case "/index":
     res.setHeader("Content-Type", "text/html");
     res.writeHead(200);
     index.pipe(res)
     break;


     case "/about":
     const about = fs.createReadStream("./about.html")
     res.setHeader("Content-Type", "text/html");
     res.writeHead(200);
     about.pipe(res)
     break;
```

```
        case "/contactUs":
        const contactUs = fs.createReadStream("./contactUs.html")
        res.setHeader("Content-Type", "text/html");
        res.writeHead(200);
        contactUs.pipe(res)
        break;

        default:
        res.setHeader("Content-Type", "text/html");
        res.writeHead(200);
        index.pipe(res)
        }

};

const server = http.createServer(requestListener);
server.listen(port, host, () => {
        console.log(`Server is running on http://${host}/${port}`);
});
```

Output:

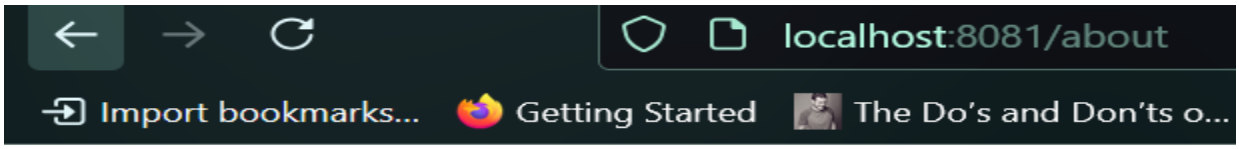
Home



Magnum College

Welcome to our college homepage

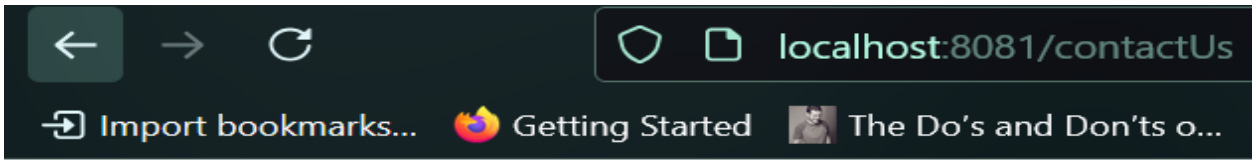
About



About!

Learn about our journey as a college

ContactUs



Contact Us

Feel free to connect with us regarding any queries

Conclusion:

We explored essential concepts in Node.js, including creating an HTTP server and serving various file types such as HTML, CSV, JSON, and PDF files, demonstrating the versatility of Node.js in handling diverse data formats. We also learned to create an HTTP server for streaming video files efficiently through piping, showcasing its potential for handling large media files. Additionally, we delved into web development, developing three HTML web pages for a college homepage, an "about me" page, and a contact page, implementing server-side routing to seamlessly serve and navigate between these pages, illustrating how Node.js can be used for building dynamic and interactive web applications.

<table>
<tr><td colspan="4">Name of Student: Shaikh Farhan Ahmed</td></tr>
<tr><td colspan="2">Roll Number: 50</td><td colspan="2">LAB Assignment Number: 07</td></tr>
<tr><td colspan="4">Title of LAB Assignment: Create http server with MySQL database connection and build CRUD application.</td></tr>
<tr><td colspan="2">DOP: 13 – 11 – 2024</td><td colspan="2">DOS: 22 – 11 – 2024</td></tr>
<tr><td>CO Mapped:
CO2</td><td>PO Mapped:
PO5,
PSO1, PSO2</td><td>Faculty Signature:</td><td>Marks:</td></tr>
</table>

Practical No 7

AIM: Create http server with MySQL database connection and build CRUD application.

Theory:

Steps to Connect and Use a Database in Node.js

1. Install a Database Driver or Library:
A driver is a software component that enables communication between Node.js and the database.
Examples:

- MySQL: mysql, mysql2
- PostgreSQL: pg
- MongoDB: mongodb

2. Establish a Connection:
Provide connection details such as host, port, username, password, and database name.
Connections can be managed as:

- Single Connection: Suitable for small applications or non-intensive tasks.
- Connection Pooling: Creates a pool of reusable connections for better performance in high-load scenarios.

3. Perform Database Operations:
Use SQL queries (for relational databases) or CRUD operations (for NoSQL databases) to interact with data.
Examples of operations:

- Create: Insert new records into a database.
- Read: Fetch data (e.g., using SELECT in SQL or find in
- MongoDB).
- Update: Modify existing records.
- Delete: Remove records.

4. Handle Results and Errors:

- Process query results returned by the database.
- Implement error handling to manage issues like connection failures,
- invalid queries, or transaction conflicts.

Benefits of Using Databases with Node.js

1. Unified Development:
Node.js allows developers to use JavaScript for both the server-side and database interactions, reducing the need for context switching.

2. Scalability:
Node.js's event-driven model and non-blocking I/O make it well-suited for high-performance database operations.

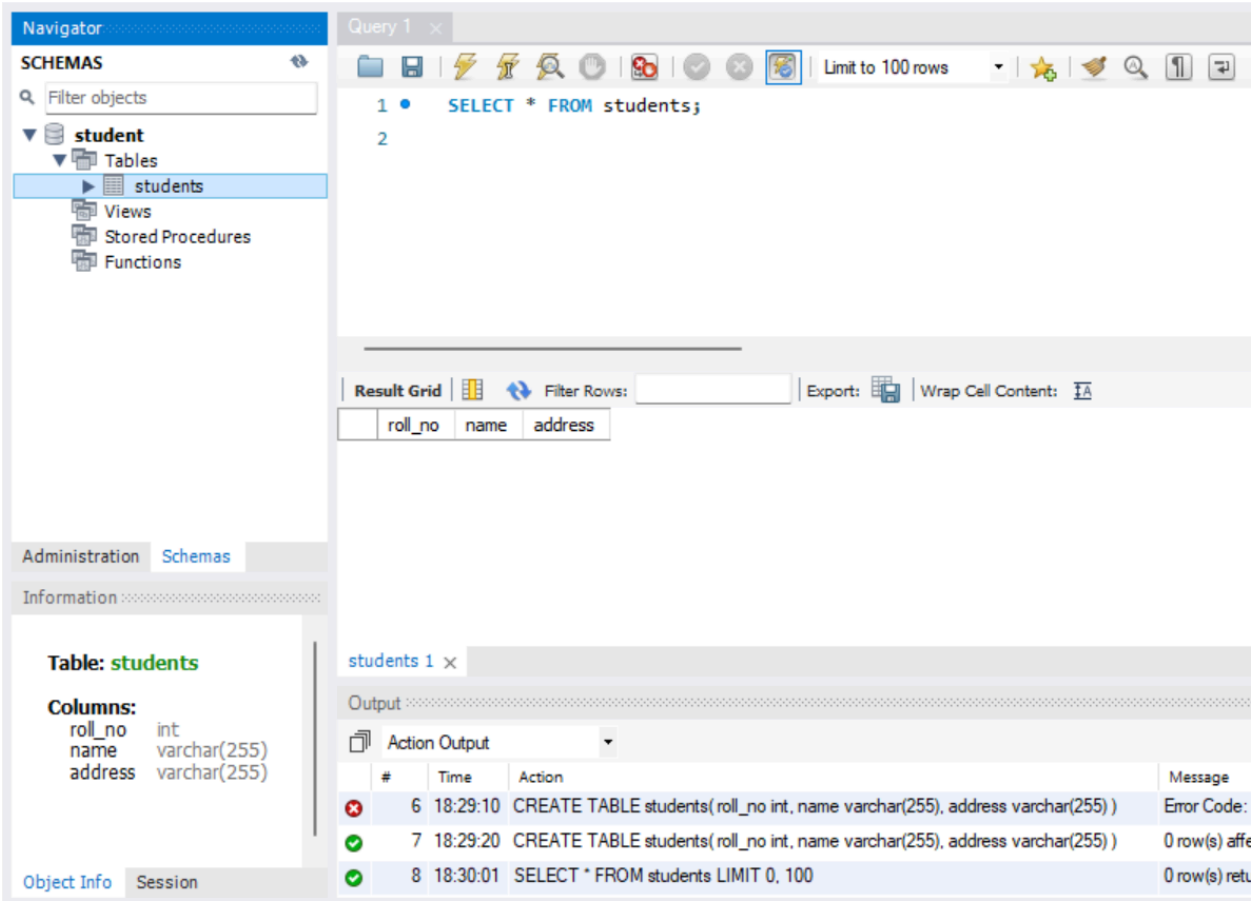
3. Flexibility:
Node.js supports multiple databases, enabling developers to choose the most suitable one for their application's needs.

4. Rich Ecosystem:
The Node.js ecosystem provides a variety of libraries and tools for seamless database integration and management.

Q) Create an application to establish a connection with the MySQL database and perform basic database operations on it(student db consisting rollno, name, address),insert 10 records, update a particular student's record, delete a record.

Database :
Create a database in mySQL and a table named students with columns

roll_no, name, address :



Source Code:

server.js

```
const express = require("express")
const PORT = 8081
const app = express()
const mysql = require('mysql2');

const pool = mysql.createPool({
    host: 'localhost',
    user: 'root',
    password: 'root',
    database: 'student'
});
app.set('view engine', 'ejs');
app.set('views', './views');

app.use(express.json());
app.use(express.urlencoded({ extended: true }))
app.get('/', (req, res) => {
    res.render('home');
})

app.get('/student/data', (req, res) => {
    let sql = 'SELECT * FROM students;';
    pool.execute(sql, (err, result) => {
        if (err) {
            console.log(`Error fetching students :
${err.message}`);
        }
        else {
            let string = JSON.stringify(result);
            console.log(`Result : ${string}`);
```

```
            let jsonOBJ = JSON.parse(string);
            console.log(`JSON OBJECT : ${jsonOBJ}`);
            res.render('viewStudent', { Students: jsonOBJ });
        }
    });
});

app.get('/student', (req, res) => {
    res.render('createStudent');
});
app.post('/student', (req, res) => {
    console.log(req.body);
    let sql = `INSERT INTO students(roll_no,name,address) VALUES

(${req.body.rollno},'${req.body.name}','${req.body.address}');`
    pool.execute(sql, (err, result) => {
        if (err) {
            console.log(`err ${err.message}`);
        }
        else {
            console.log(`Result : ${result}`)
        }

    })
    res.send('Student Added successfully in DB');
});

app.get('/student/update', (req, res) => {
    res.render('updateStudent');
});

app.post('/student/update', (req, res) => {
    let id = req.body.rollno;
    let updatedName = req.body.name;
    let updatedAddress = req.body.address;
```

```
    let sql = `UPDATE students

    SET name = '${updatedName}',address = '${updatedAddress}'
    WHERE roll_no = ${id};`;
    pool.execute(sql, (err) => {
        if (err) {
            console.log(`Error in updating database
err:${err.message}`);
        }
        res.send('Student updated Successfully!');
    })
});



app.get('/student/delete', (req, res) => {
    res.render('deleteStudent');
});
app.post('/student/delete/', (req, res) => {
    let id = req.body.rollno;
    sql = `DELETE FROM students WHERE roll_no = ${id};`;
    pool.execute(sql, (err) => {
        if (err) {
            console.log(`error in deleting student data from
database :
    ${err.message}`);
        }
        res.send('Student data deleted successfully');

    });
});


app.listen(PORT, () => {
    console.log(`Listening at port ${PORT}`)
```

```
})
```

home.ejs

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width,
initial-scale=1.0">
    <title>Practcal 7</title>
</head>

<body>
    <p>Homepage</p>
    <form action="/student" method="get">
        <input type="submit" value="Create Student">
    </form>
    <form action="/student/data" method="get">
        <input type="submit" value="View Students">
    </form>
    <form action="/student/update" method="get">
        <input type="submit" value="Update Students">
    </form>
    <form action="/student/delete" method="get">
        <input type="submit" value="Delete Students">
    </form>
</body>

</html>
```

viewStudent.ejs

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width,
initial-scale=1.0">
    <title>Practcal 7</title>
</head>

<body>
    <div>
        <% Students.forEach((student)=>{ %>
            <div>
                <p>Roll No : <%= student.roll_no %></p>
                <p>Name : <%= student.name %></p>
                <p>Address : <%= student.address %></p>
                <br>
            </div>
            <% }) %>
    </div>
    <form action="/student" method="get">
        <input type="submit" value="Create Student">
    </form>
    <form action="/" method="get">
        <input type="submit" value="Home">
    </form>
</body>

</html>
```

createStudent.ejs

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width,
initial-scale=1.0">
    <title>Practcal 7</title>
</head>

<body>
    <p>Enter details of a Students : </p>
    <form action="/student" method="post">
        <label>Roll No : </label>
        <input type="number" name="rollno"> <br>
        <label>Name : </label><input type="text"
name="name"><br>
        <label>Address
            : </label><input type="text" name="address"><br>
        <input type="submit" value="Enter Data">
    </form>
    <form action="/">
        <input type="submit" value="Home">
    </form>
    <form action="/student/data">
        <input type="submit" value="View Students">
    </form>
</body>

</html>
```

updateStudent.ejs

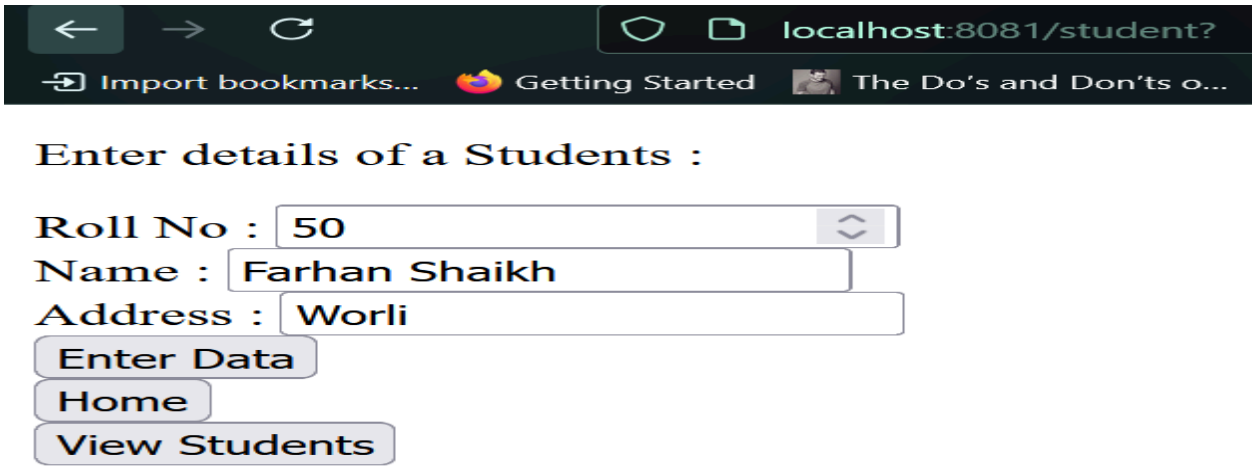
```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width,
initial-scale=1.0">
    <title>Practical 7</title>
</head>

<body>
    <p>Enter student details to update : </p>
    <form action="/student/update" method="post">
        <label>Enter Student ID : </label>
        <input type="number" name="rollno"> <br>
        <label>Enter updated Name : </label>
        <input type="text" name="name"><br>
        <label>Enter updated Address : </label>
        <input type="text" name="address">
        <input type="submit" value="Update!">
    </form>
</body>

</html>
```

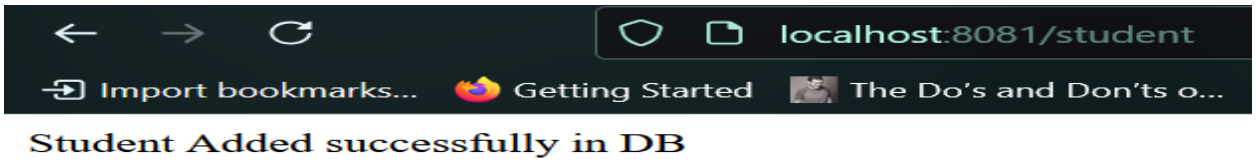
deleteStudent.ejs

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width,
initial-scale=1.0">
    <title>Practical 7</title>
</head>

<body>
    <p>Delete the data of a student : </p>
    <form action="/student/delete/" method="post">
        <label>Enter the roll_no of student : </label>
        <input type="number" name="rollno" id="rollno"> <br>
        <input type="submit" value="Delete!">
    </form>
</body>

</html>
```

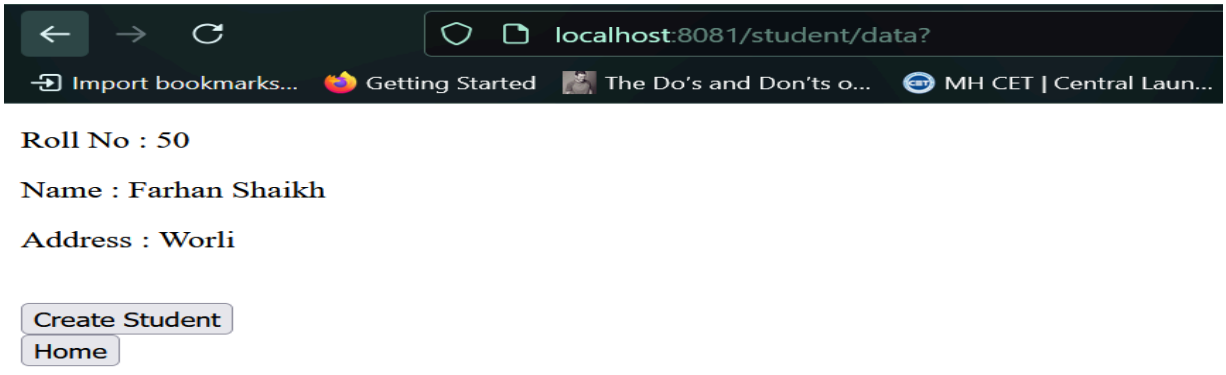
Output:

Creating a student:

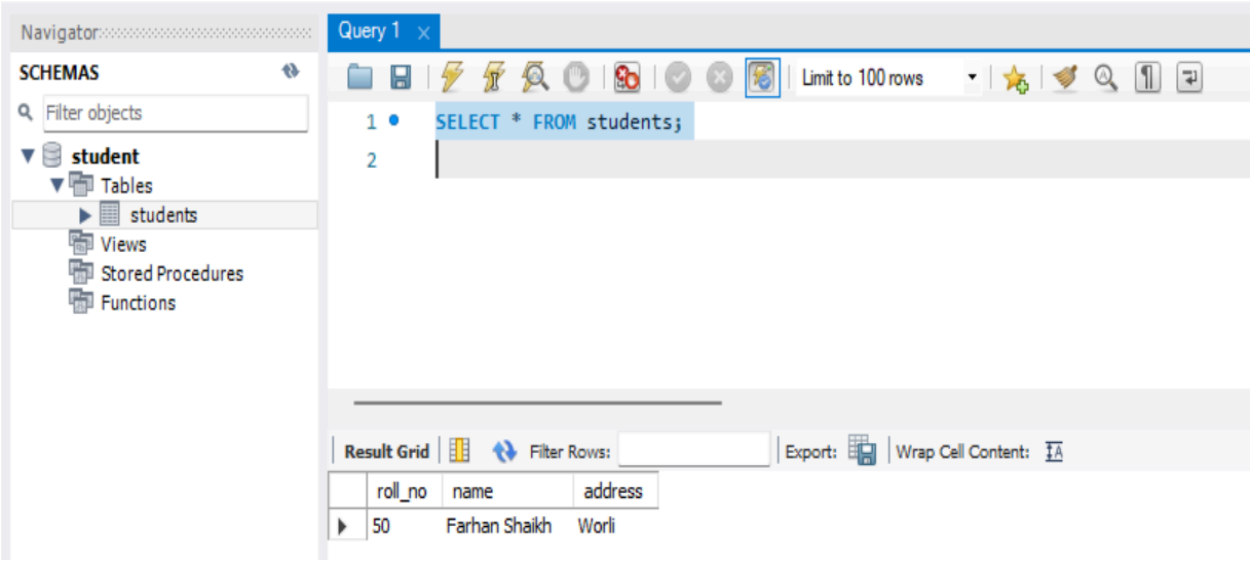




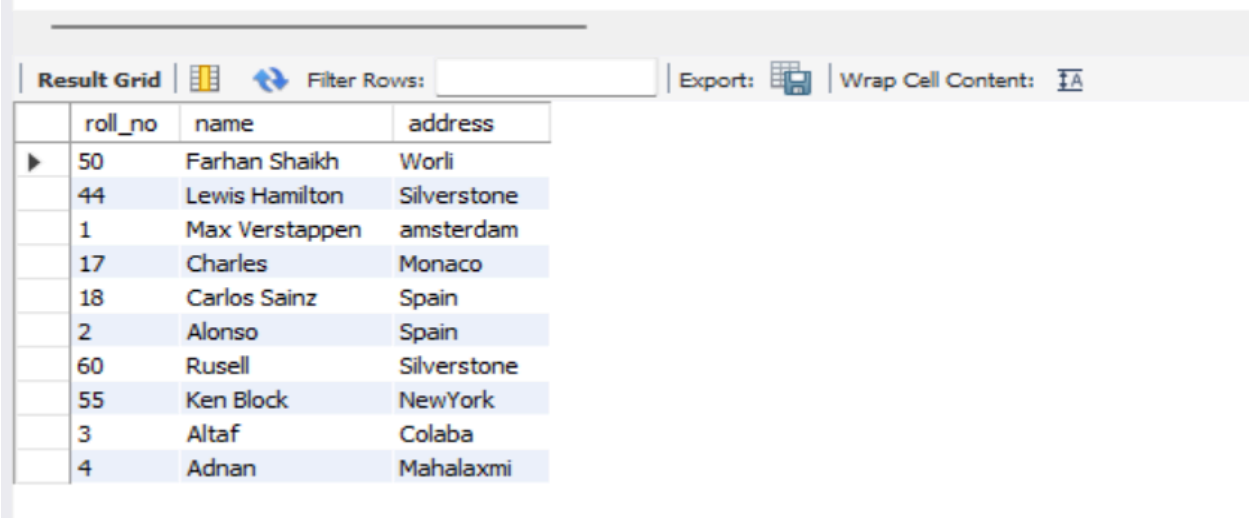
Viewing Created Student:



Checking Database to ensure student is saved.



Storing 9 more students in DB:



roll_no	name	address
50	Farhan Shaikh	Worli
44	Lewis Hamilton	Silverstone
1	Max Verstappen	amsterdam
17	Charles	Monaco
18	Carlos Sainz	Spain
2	Alonso	Spain
60	Rusell	Silverstone
55	Ken Block	NewYork
3	Altaf	Colaba
4	Adnan	Mahalaxmi

Updating a student record:

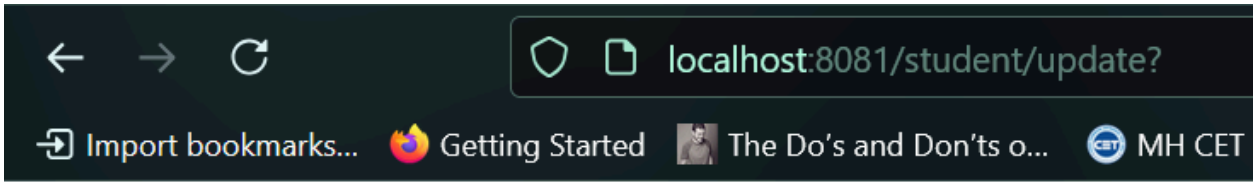
Before:

Roll No : 50

Name : Farhan Shaikh

Address : Worli

Updating:

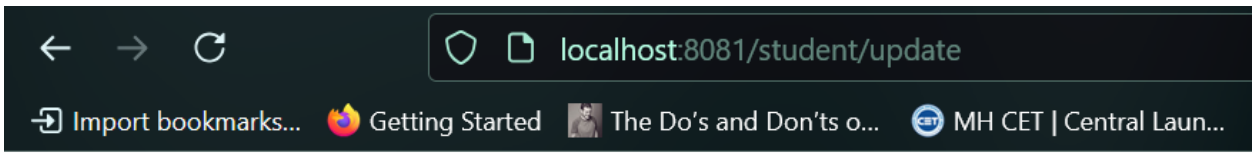


Enter student details to update :

Enter Student ID : 50

Enter updated Name : Farhan Shaikh

Enter updated Address : Byculla Update!



Student updated Successfully!

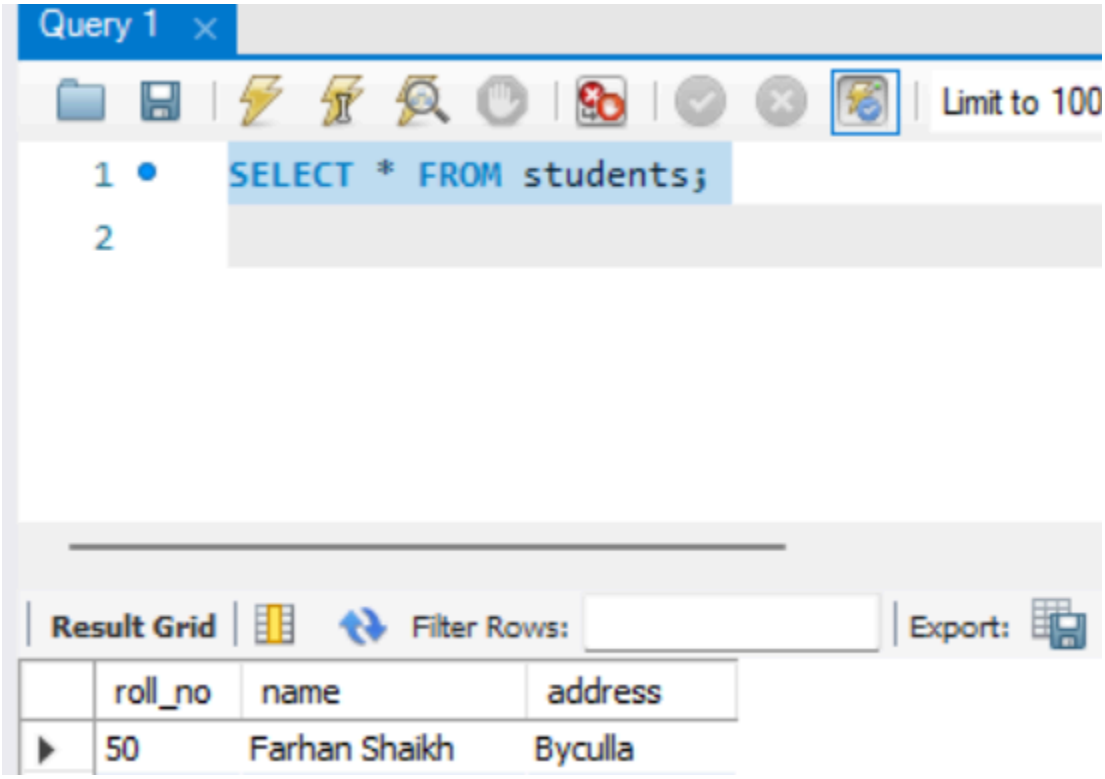
After:

Roll No : 50

Name : Farhan Shaikh

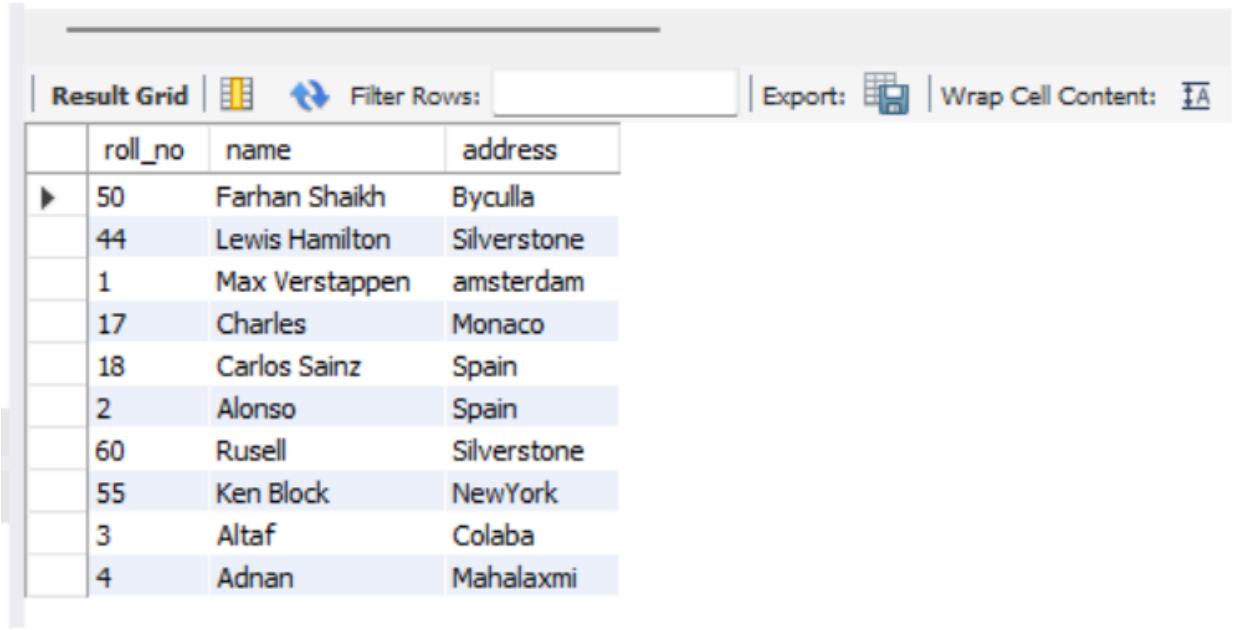
Address : Byculla

Database after Updating:



Deleting a student from the database:

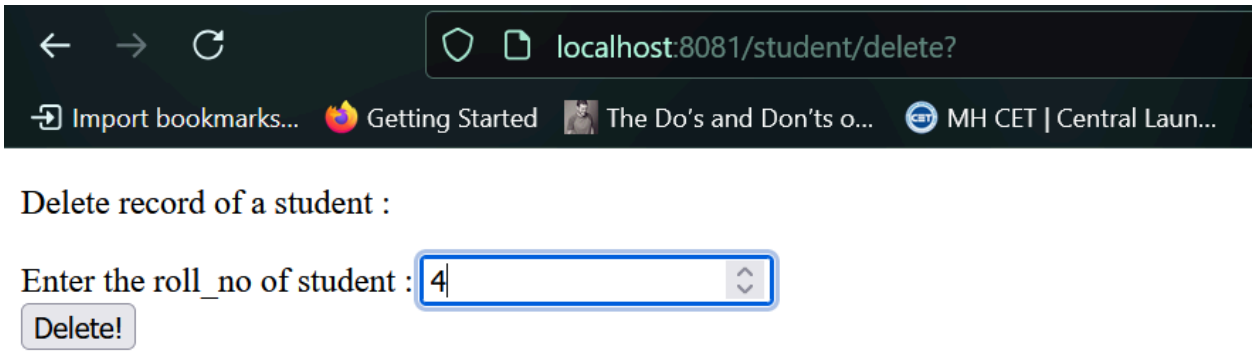
Initial Database:

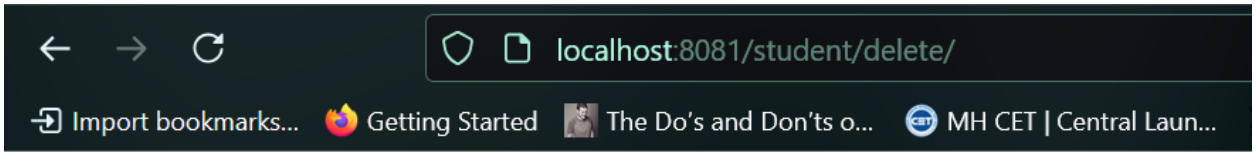


Result Grid | Filter Rows: | Export: | Wrap Cell Content:

roll_no	name	address
50	Farhan Shaikh	Byculla
44	Lewis Hamilton	Silverstone
1	Max Verstappen	amsterdam
17	Charles	Monaco
18	Carlos Sainz	Spain
2	Alonso	Spain
60	Rusell	Silverstone
55	Ken Block	NewYork
3	Altaf	Colaba
4	Adnan	Mahalaxmi

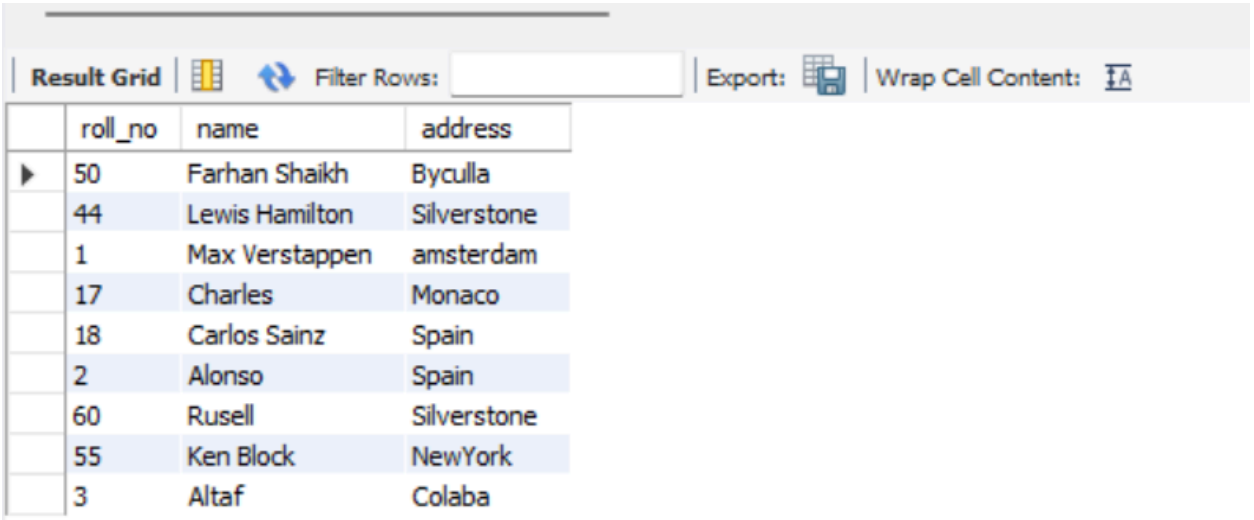
Deleting a record:





Student data deleted successfully

After deleting :



roll_no	name	address
50	Farhan Shaikh	Byculla
44	Lewis Hamilton	Silverstone
1	Max Verstappen	amsterdam
17	Charles	Monaco
18	Carlos Sainz	Spain
2	Alonso	Spain
60	Rusell	Silverstone
55	Ken Block	NewYork
3	Altaf	Colaba

Student adnan is deleted from the database.

Conclusion:
From this practical I have learned to connect mysql database with node.js and create a simple CRUD application.

<table>
<tr><td colspan="4">Name of Student: Shaikh Farhan Ahmed</td></tr>
<tr><td colspan="2">Roll Number: 50</td><td colspan="2">LAB Assignment Number: 08</td></tr>
<tr><td colspan="4">Title of LAB Assignment: TypeScript installation, Environment Setup.</td></tr>
<tr><td colspan="2">DOP: 25 – 11 – 2024</td><td colspan="2">DOS: 29 – 11 – 2024</td></tr>
<tr><td>CO Mapped:
CO4</td><td>PO Mapped:
PO3, PO5,
PSO1, PSO2</td><td>Faculty Signature:</td><td>Marks:</td></tr>
</table>

Practical No 8

AIM: TypeScript installation, Environment Setup.

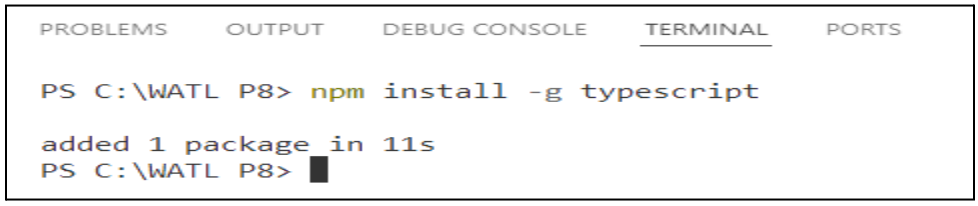
Theory:

TypeScript is a free and open-source high-level programming language developed by Microsoft that adds static typing with optional type annotations to JavaScript. It is designed for the development of large applications and transpiled to JavaScript.
TypeScript may be used to develop JavaScript applications for both client-side and server-side execution (as with Node.js, Deno or Bun). Multiple options are available for transpilation. The default TypeScript Compiler can be used or the Babel compiler can be invoked to convert TypeScript to JavaScript.

Installation and Configuration of TypeScript:

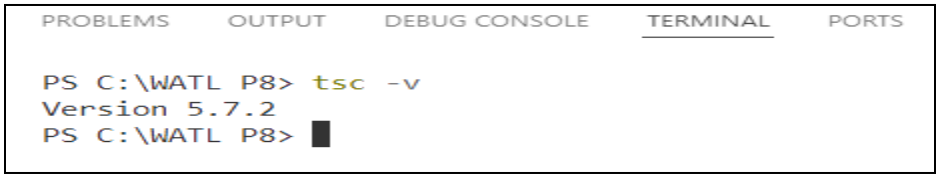
Step 1: Within your npm project, run the following command to install the compiler:

```
npm install -g typescript
```



Step 2: Verify the installation.

```
tsc -v
```



Step 3: Configuring TypeScript We need to configure TypeScript for our project now that a new project has been created and TypeScript has been installed. To accomplish this, a tsconfig.json file must be created in the root of our project directory. The TypeScript configuration options are located in this file.

```
tsc --init
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
PS C:\WATL P8> tsc --init

Created a new tsconfig.json with:

  target: es2016
  module: commonjs
  strict: true
  esModuleInterop: true
  skipLibCheck: true
  forceConsistentCasingInFileNames: true
```

Step 4: Converting TypeScript file into JavaScript and running it with node.

test.ts

```
function add(a:number,b:number){
    return a + b
}
console.log(add(10,20))
```

Converted JavaScript File.

test.js

```
function add(a,b){
    return a + b
}
console.log(add(10,20))
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
PS C:\WATL P8> tsc test.ts
PS C:\WATL P8> node test.js
30
PS C:\WATL P8>
```

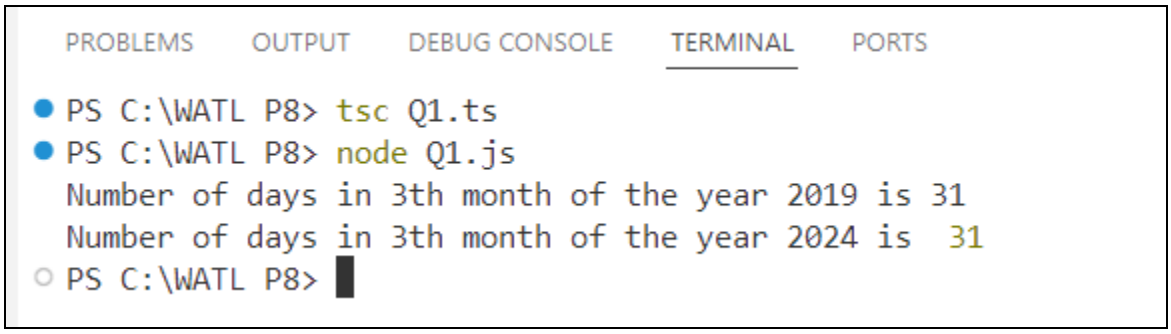
1. Write a function in typescript that returns the number of days in a specified month and year [Hint: use switch case] (input will be year and number of the month .Make the year as a default parameter
input:
getDay(2019,3)
getDay(undefined,3)

Source Code:

```
function getDay(year : number = new Date().getFullYear(), month:
number) {
    switch (month) {
        case 1: // January
        case 3: // March
        case 5: // May
        case 7: // July
        case 8: // August
        case 10: // October
        case 12: // December
            return 31;
        case 4: // April
        case 6: // June
        case 9: // September
        case 11: // November
            return 30;
        case 2: // February
            return (year % 4 === 0 && (year % 100 !== 0 || year
                    % 400 === 0)) ? 29 : 28;
        default:
            return -1;
    }
}

let month = 3;
let year = 2019;

console.log("Number of days in " + month + "th month of the year
"
    + year + " is " + getDay(year, month));
```

```
console.log("Number of days in " + month + "th month of the year
2024 is ", getDay(undefined, month));
```

Output:



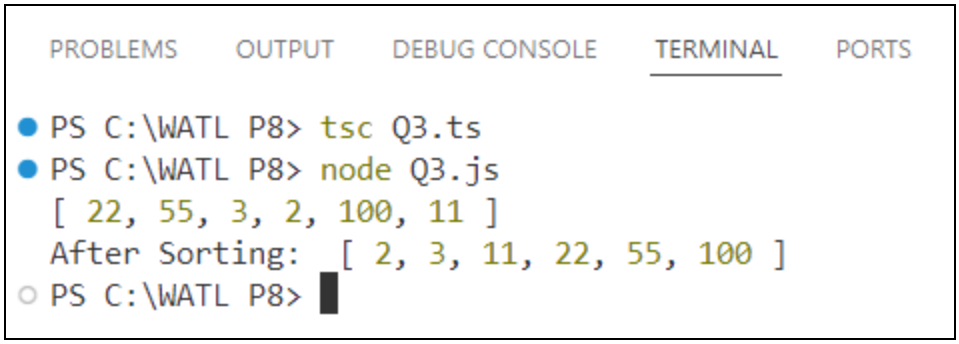
2. Write a program to sort a given array using bubble sort.

Source Code:

Q3.ts

```
function bubbleSort(arr: number[]): number[] {
    const n = arr.length;

    for (let i = 0; i < n - 1; i++) {
      for (let j = 0; j < n - i - 1; j++) {
        if (arr[j] > arr[j + 1]) {
          [arr[j], arr[j + 1]] = [arr[j + 1], arr[j]];
        }
      }
    }
    return arr;
  }


let arr : number[] = new Array(22,55,3,2,100,11)
console.log(arr)
console.log("After Sorting: ", bubbleSort(arr))
```

Output:



3. Create a class Book with the following class variables-Book ID, Book Name, Author,ISBN No,Publisher Use a constructor to initialize values.Write 2 methods addBook and displayBook. Add information about min. 5 books. Also provided with a search function to search a book using Book ID.

Source Code:

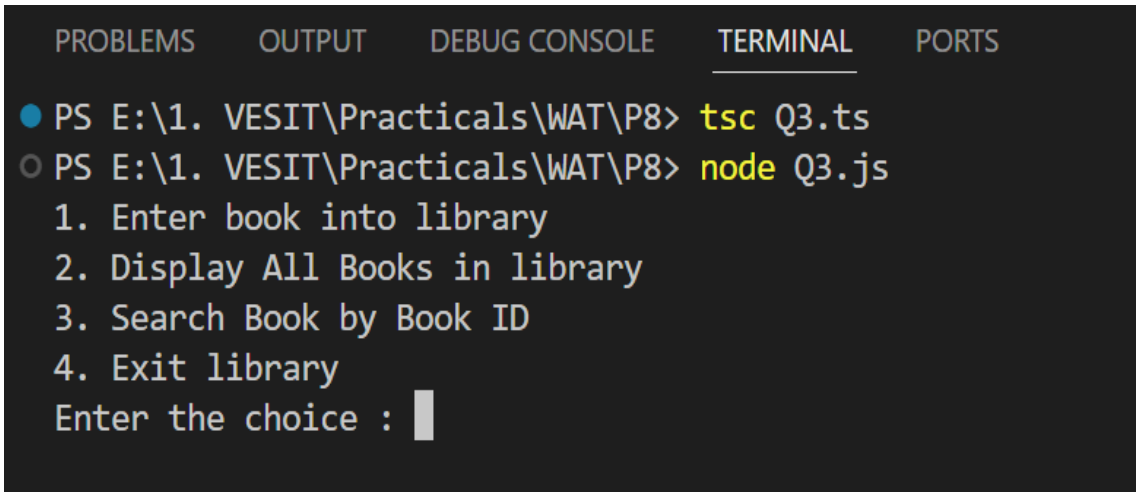
```
import promptSync = require('prompt-sync');
const prompt = promptSync();
class Book {
     bookId: number;
     bookName: string;
     author: string;
     ISBN: number;
     publisher: string;
     constructor(id: number, name: string, auth: string, isbn:
                 number, publisher: string) {
     this.bookId = id;
     this.bookName = name;
     this.author = auth;
     this.ISBN = isbn;
     this.publisher = publisher;
     }

     displayBook() {
     console.log('---------- Displaying Book Details---------');
     console.log(`Book ID : ${this.bookId}`);
     console.log(`Book Name : ${this.bookName}`);
     console.log(`Book author : ${this.author}`);
     console.log(`Book ISBN : ${this.ISBN}`);
```

```
        console.log(`Book Publisher : ${this.publisher}`);
        console.log('-----------------------------------');
        }
}

class Library {
        self: Book[] = [];
        addBook(book: Book) {
        this.self.push(book);
        }

        displayBooks() {
        this.self.forEach((book) => {
                console.log('-------------------------------');
                console.log(`Book ID : ${book.bookId}`);
                console.log(`Book Name : ${book.bookName}`);
                console.log(`Book author : ${book.author}`);
                console.log(`Book ISBN : ${book.ISBN}`);
                console.log(`Book Publisher : ${book.publisher}`);
                console.log('-------------------------------');
        });
        }

        searchBook(id: number) {
        this.self.forEach((book) => {
                if (book.bookId === id) {
                        book.displayBook();
                }
                else {
                        console.log(`No book found for id : ${id}`);
                }
        });
        }
}

let loop: boolean = true;
let library: Library = new Library();



while (loop) {
```

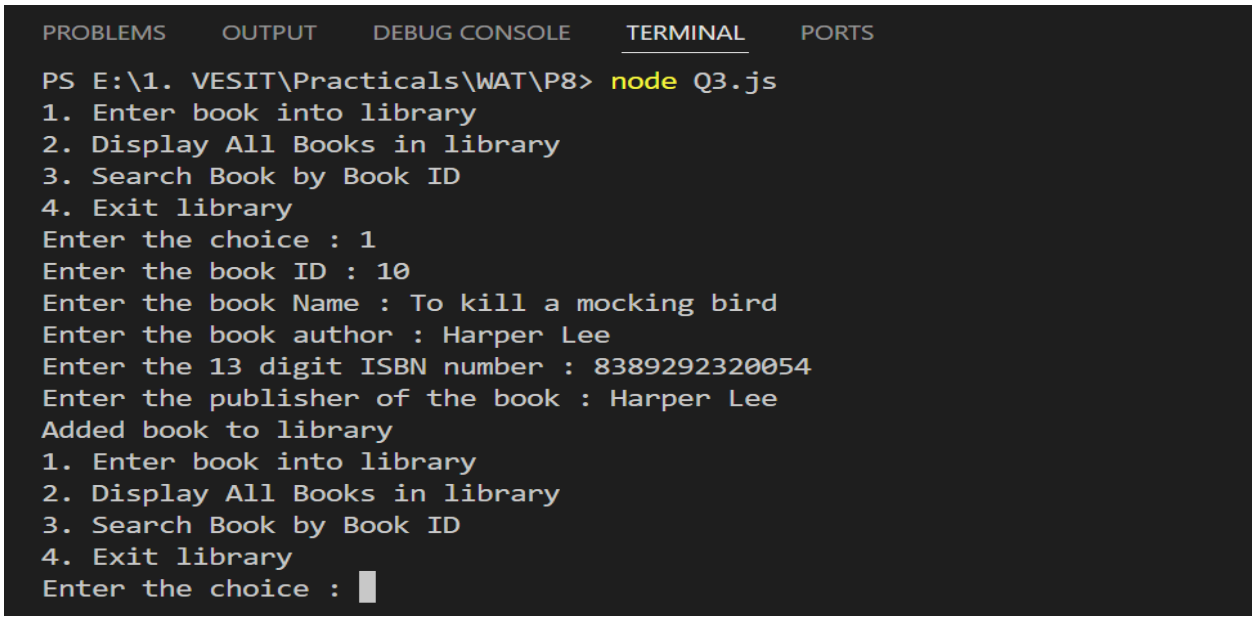
```
    let ch: number;
    console.log('1. Enter book into library');
    console.log('2. Display All Books in library');
    console.log('3. Search Book by Book ID');
    console.log('4. Exit library');
    ch = parseInt(prompt('Enter the choice : '));

    switch (ch) {
    case 1:
        let id: number = parseInt(prompt('Enter the
bookID:'));
        let name: string = prompt('Enter the book Name : ');
        let author: string = prompt('Enter the book author
:');
        let isbn: number =
         parseInt(prompt('Enter the 13 digit ISBN number :
'));
        let publisher: string
           = prompt('Enter the publisher of the book : ');
        let newBook: Book =
            new Book(id, name, author, isbn, publisher);
        library.addBook(newBook);
        console.log('Added book to library');
        break;
    case 2:
        library.displayBooks();
        break;
    case 3:
        let searchId: number =
             parseInt(prompt('Enter the book ID : '));
        library.searchBook(searchId);
        break;
    case 4:
        loop = false;
        break;
    }

}
```

Output:

List of Options:



Adding a single Book and Displaying its information:



Adding 4 more books to the library and displaying:

```
Enter the choice : 2
--------------------------------
Book ID : 10
Book Name : To Kill a Mocking Bird
Book author : Harper Lee
Book ISBN : 9780061120084
Book Publisher : Harper Lee
--------------------------------
--------------------------------
Book ID : 11
Book Name : Its Not About the Bike
Book author : Lance Armstrong
Book ISBN : 9780061120022
Book Publisher : Picador
--------------------------------
--------------------------------
Book ID : 12
Book Name : Blood Meridian
Book author : Cormac McCarthy
Book ISBN : 9780061130011
Book Publisher : Picador
--------------------------------
--------------------------------
Book ID : 13
Book Name : 1984
Book author : George Orwell
Book ISBN : 9780451524935
Book Publisher : Harcourt
--------------------------------
```

```
--------------------------------
Book ID : 14
Book Name : Introductions to Algorithms
Book author : Cormen
Book ISBN : 9780451524999
Book Publisher : PHI
--------------------------------
```

Searching a Book by ID:

```
1. Enter book into library
2. Display All Books in library
3. Search Book by Book ID
4. Exit library
Enter the choice : 3
Enter the book ID : 14
No book found for id : 14
No book found for id : 14
No book found for id : 14
No book found for id : 14
---------------- Displaying Book Details ----------------
Book ID : 14
Book Name : Introductions to Algorithms
Book author : Cormen
Book ISBN : 9780451524999
Book Publisher : PHI
------------------------------------------------------------
```

CONCLUSION :

From this practical I learned to install and use typescript with Node JS.

Name of Student: Shaikh Farhan		
Roll Number: 50	**Lab Assignment Number: 09**	
Title of Lab Assignment: Introduction to Angular,Setup for local development environment, Angular Architecture Create an application to demonstrate directives and pipes		
DOP: 27-11-2024	**DOS: 07-12-2024**	
CO Mapped: **CO5**	**PO Mapped:** **PO3, PO5, PSO1, PSO2**	**Signature:**

Practical 9

AIM : Introduction to Angular,Setup for local development environment, Angular Architecture Create an application to demonstrate directives and pipes

Theory :

Angular is a powerful, open-source web application framework developed and maintained by Google. It is built on TypeScript and enables developers to create dynamic, single-page applications (SPAs) with ease. Angular follows a component-based architecture, where each part of the application is encapsulated into reusable components, promoting modularity and separation of concerns. The framework leverages declarative programming using HTML templates, which are enhanced with directives to create dynamic views. This approach enhances productivity and maintainability, allowing for better code organization and scalability.

At its core, Angular relies on a bidirectional data binding mechanism, which ensures that changes to the application model automatically update the view, and vice versa. This eliminates the need for developers to manually manage the DOM, leading to faster development cycles and a more responsive user experience. Angular's dependency injection (DI) system further simplifies development by allowing components to request services, such as data handling or authentication, without tightly coupling components to specific service implementations.

Additionally, Angular incorporates powerful tools for routing, state management, form handling, and HTTP client integration, making it a comprehensive solution for modern web development. The framework supports the creation of progressive web applications (PWAs) and offers robust testing tools for unit, integration, and end-to-end tests, promoting high-quality, maintainable code. With its rich ecosystem and large community support, Angular continues to be a popular choice for building complex, enterprise-grade applications.

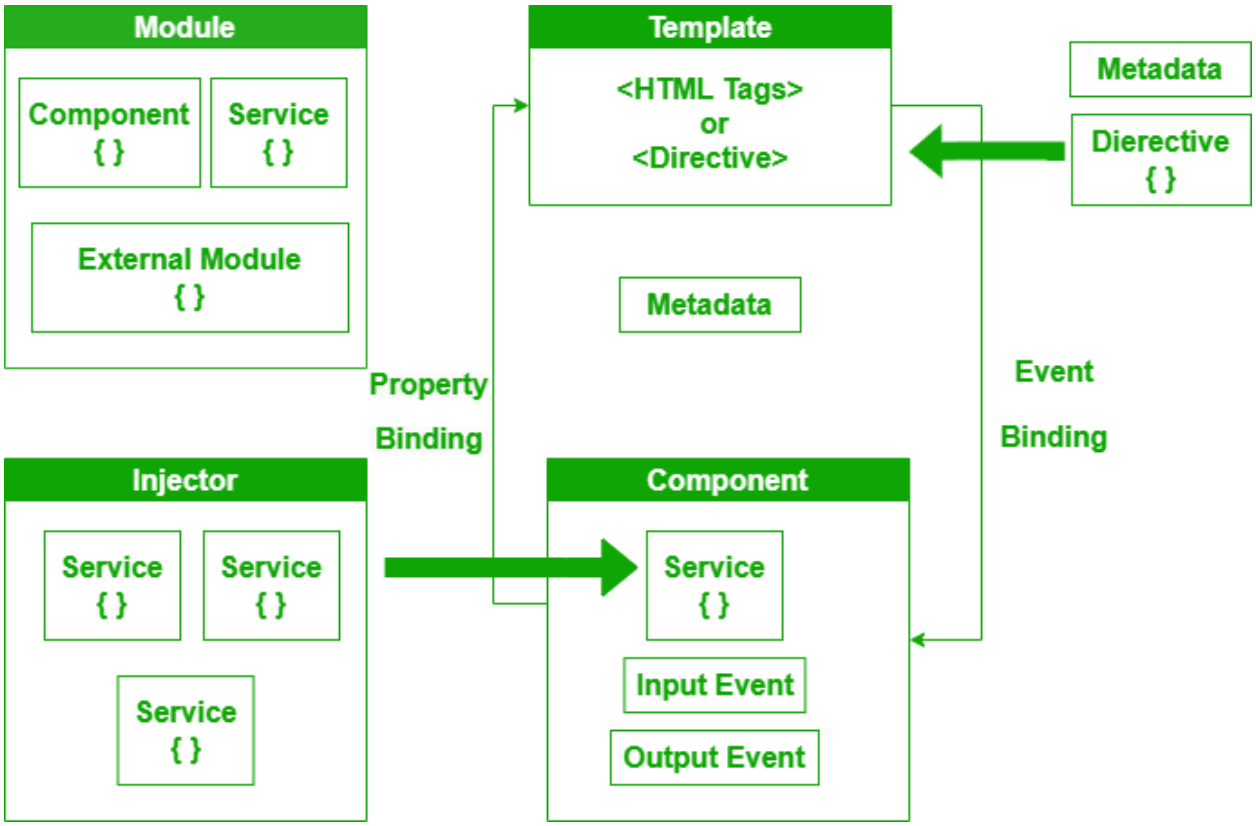
One of Angular's standout features is its comprehensive CLI (Command Line Interface), which simplifies and accelerates the development process. The Angular CLI provides a range of powerful commands for generating components, services, modules, and even entire projects, reducing the need for repetitive boilerplate code. It also supports tasks like testing, linting, building, and deploying applications, all from the command line. With built-in optimization features like Ahead-of-Time (AOT) compilation and tree-shaking, the Angular CLI ensures that applications are performant and lightweight, making it easier for developers to focus on building features rather than managing infrastructure. This robust tooling ecosystem significantly enhances the overall developer experience and productivity, contributing to Angular's reputation as a full-featured, end-to-end framework.

1. **Introduction to Angular, Setup for local development environment, Angular Architecture**

Ans:

Angular architecture :

To develop any web application, Angular follows the MVC (Model-View-Controller) and MVVM (Model-View-ViewModel) design patterns, which facilitates a structured and organized approach to designing the application, along with making it easier to manage code, improve maintainability, & etc. These types of design patterns usually maintain a clear distinction between data (Model), user interface (View), and logic (Controller/ViewModel), which results in more scalable and testable applications. It also provides code reusability and ensures a smoother development process for large & complex projects.



Setting up Angular :

Run npm init to initialise node package manager

```
PS E:\13B pushkar WTL practical 9> npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help init` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg>` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
package name: (13b-pushkar-wtl-practical-9)
version: (1.0.0)
description:
entry point: (index.js)
test command:
git repository:
keywords:
author:
license: (ISC)
About to write to E:\13B pushkar WTL practical 9\package.json:
```

Run npm install -g @angular/cli

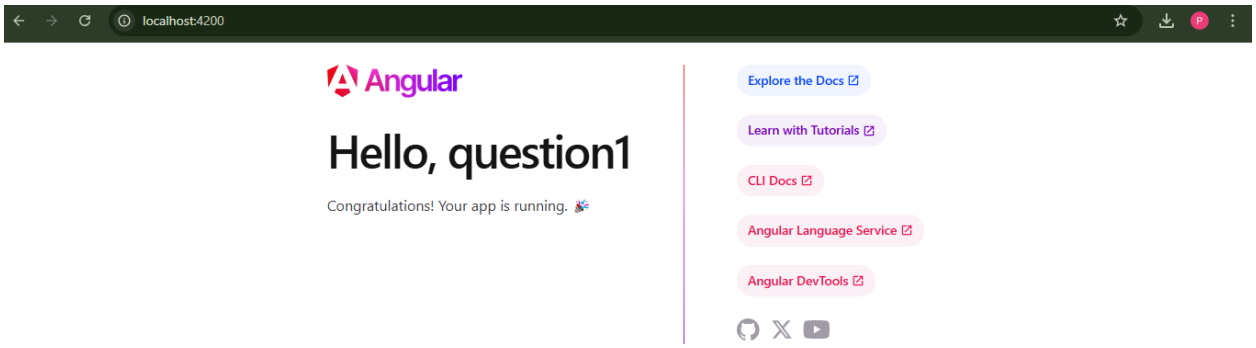
```
PS E:\13B pushkar WTL practical 9> npm install -g @angular/cli@12.0.0
npm warn EBADENGINE Unsupported engine {
npm warn EBADENGINE   package: '@angular/cli@12.0.0',
npm warn EBADENGINE   required: {
npm warn EBADENGINE     npm: '^6.11.0 || ^7.5.6',
npm warn EBADENGINE     node: '^12.14.1 || ^14.0.0',
npm warn EBADENGINE     yarn: '>= 1.13.0'
npm warn EBADENGINE   },
npm warn EBADENGINE   current: { node: 'v22.11.0', npm: '10.9.0' }
npm warn EBADENGINE }
npm warn EBADENGINE Unsupported engine {
npm warn EBADENGINE   package: '@angular-devkit/architect@0.1200.0',
npm warn EBADENGINE   required: {
npm warn EBADENGINE     npm: '^6.11.0 || ^7.5.6',
npm warn EBADENGINE     node: '^12.14.1 || ^14.0.0',
npm warn EBADENGINE     yarn: '>= 1.13.0'
npm warn EBADENGINE   },
npm warn EBADENGINE   current: { node: 'v22.11.0', npm: '10.9.0' }
npm warn EBADENGINE }
npm warn EBADENGINE Unsupported engine {
npm warn EBADENGINE   package: '@angular-devkit/core@12.0.0',
npm warn EBADENGINE   required: {
```

Run ng new question1 To create a angular project

```
REATE question1/public/favicon.ico (15086 bytes)
/ Packages installed successfully.
'git' is not recognized as an internal or external command,
perable program or batch file.
S E:\13B pushkar WTL practical 9> cd question1
S E:\13B pushkar WTL practical 9\question1>
```

Run npm start to run the project

```
Browser bundles
Initial chunk files   | Names           |  Raw size
polyfills.js          | polyfills       |  90.20 kB |
main.js               | main            |  18.31 kB |
styles.css            | styles          |  95 bytes |

                      | Initial total   | 108.60 kB

Server bundles
Initial chunk files   | Names           |  Raw size
polyfills.server.mjs  | polyfills.server| 572.91 kB |
main.server.mjs       | main.server     |  19.54 kB |
server.mjs            | server          |   1.86 kB |

Application bundle generation complete. [1.811 seconds]

Watch mode enabled. Watching for file changes...
NOTE: Raw file sizes do not reflect development server per-request transformations.
  → Local:   http://localhost:4200/
  → press h + enter to show help
```



2. **Create an Angular application to accept a number from user and display its multiplication table.Also show or hide the multiplication table on click of a button (Use nglf and ngFor)**

Ans:

Create an angular project

Create a component using ng generate component multiplication-table

Source code :

Multiplication-table-component.html :

```
<div>

    <h1>Multiplication Table</h1>

    <label for="numberInput">Enter a number: </label>

    <input id="numberInput" type="number" [(ngModel)]="number"
/>



    <button (click)="toggleTable()">{{ showTable ? 'Hide' :
'Show' }} Table</button>



    <div *ngIf="showTable && number !== null">

      <h3>Multiplication Table for {{ number }}</h3>

      <ul>

        <li *ngFor="let result of getMultiplicationTable(); let
i = index">

          {{ number }} x {{ i + 1 }} = {{ result }}
```

```
        </li>
      </ul>
    </div>
  </div>
```

multiplication-table-component.ts :

```
import { Component } from '@angular/core';
import { CommonModule } from '@angular/common';
import { FormsModule } from '@angular/forms';


@Component({
  selector: 'app-multiplication-table',
  standalone: true,
  imports: [CommonModule, FormsModule],
  templateUrl: './multiplication-table.component.html',
  styleUrls: ['./multiplication-table.component.css']
})
export class MultiplicationTableComponent {
  number: number =0;
  showTable = false;


  toggleTable(): void {
```

```
    this.showTable = !this.showTable;

  }



  getMultiplicationTable(): number[] {

    return this.number ? Array.from({ length: 10 }, (_, i) => (i
+ 1) * this.number) : [];

  }

}
```

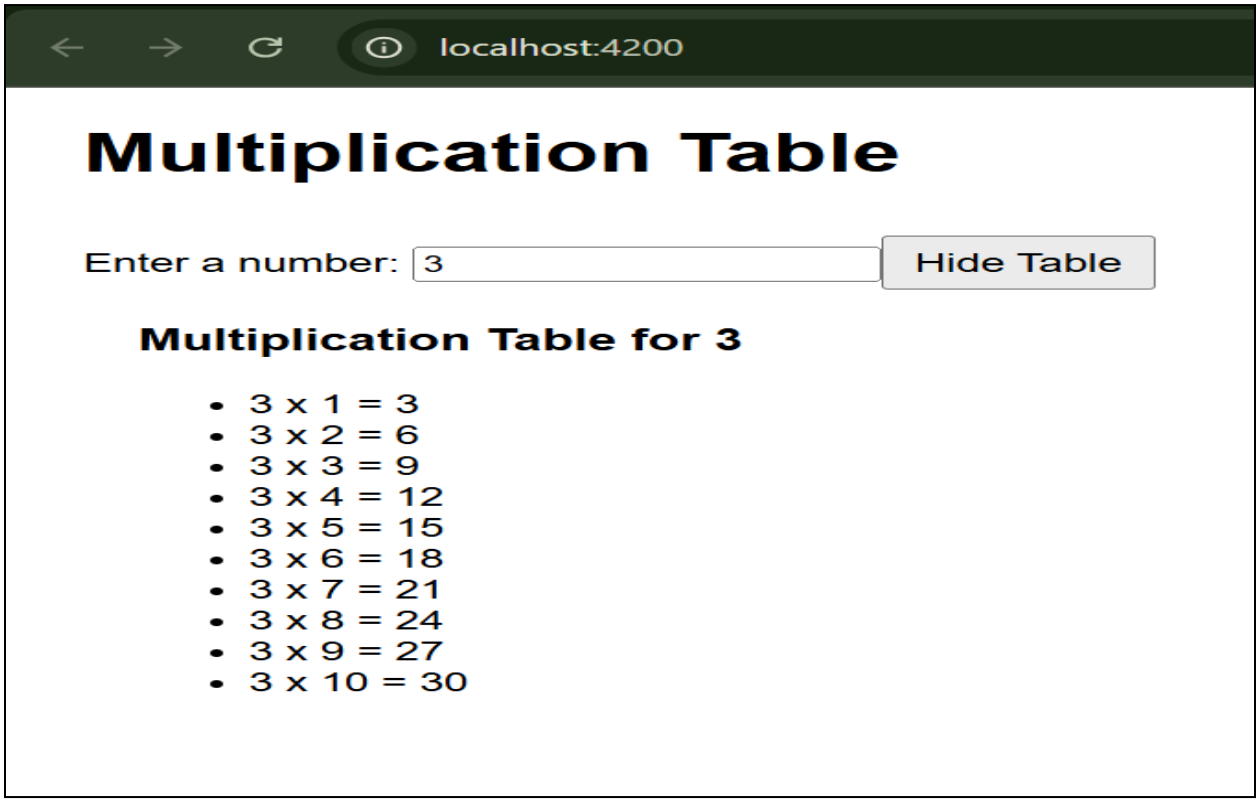
app.component.ts :

```
import { Component } from '@angular/core';

import { MultiplicationTableComponent } from
'./multiplication-table/multiplication-table.component';



@Component({

  selector: 'app-root',

  standalone: true,

  imports: [MultiplicationTableComponent],

  template:
'<app-multiplication-table></app-multiplication-table>',

  styleUrls: ['./app.component.css']

})

export class AppComponent { }
```

App.component.html :

```
<app-multiplication-table>



</app-multiplication-table>
```

Output :



3. **Create an Angular application to calculate Simple Interest Accept a principal amount, Rate of interest and year from user.**

Ans :

Source code :

Simple-interest.component.html :

```
<div>

    <h1>Simple Interest Calculator</h1>



    <label for="principal">Principal Amount:</label>

    <input id="principal" type="number" [(ngModel)]="principal"
/>



    <label for="rate">Rate of Interest (%):</label>

    <input id="rate" type="number" [(ngModel)]="rate" />



    <label for="years">Number of Years:</label>

    <input id="years" type="number" [(ngModel)]="years" />



    <button (click)="calculateInterest()">Calculate
Interest</button>
```

```
    <div *ngIf="simpleInterest !== null">
      <h3>Calculated Simple Interest: {{ simpleInterest }}</h3>
    </div>
  </div>
```

simple-interest.component.ts :

```
import { Component } from '@angular/core';
import { CommonModule } from '@angular/common';
import { FormsModule } from '@angular/forms';

@Component({
  selector: 'app-simple-interest',
  standalone: true,
  imports: [CommonModule, FormsModule],
  templateUrl: './simple-interest.component.html',
  styleUrls: ['./simple-interest.component.css']
})
export class SimpleInterestComponent {
  principal: number | null = null;
  rate: number | null = null;
  years: number | null = null;
```

```
  simpleInterest: number | null = null;



  calculateInterest(): void {

    if (this.principal !== null && this.rate !== null &&
this.years !== null) {

      this.simpleInterest = (this.principal * this.rate *
this.years) / 100;

    } else {

      this.simpleInterest = null;

    }

  }

}
```

App.component.html :

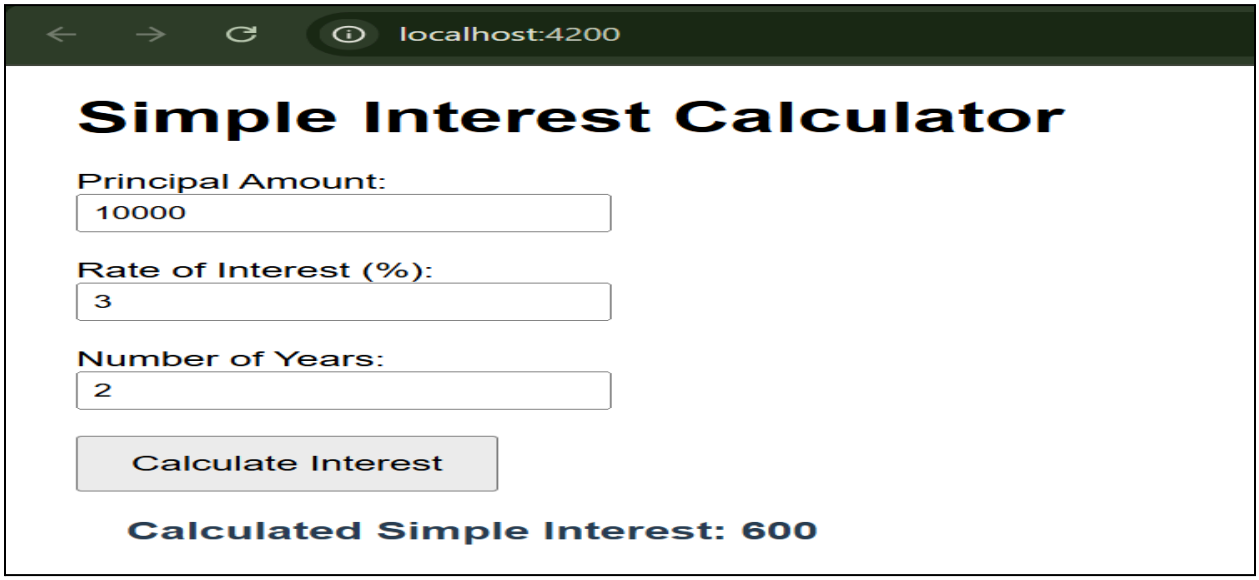
```
<app-simple-interest>



</app-simple-interest>
```

app.component.ts:

```
import { Component } from '@angular/core';

import { SimpleInterestComponent } from
'./simple-interest/simple-interest.component';
```

```
@Component({
  selector: 'app-root',
  standalone: true,
  imports: [SimpleInterestComponent], // Import the standalone SimpleInterestComponent here
  template: '<app-simple-interest></app-simple-interest>', // Use the selector for SimpleInterestComponent
  styleUrls: ['./app.component.css']
})
export class AppComponent { }
```

Output :



4. **Create an application to demonstrate mouse and keyboard and blur events on webpage**

Ans :

Source code :

Event-demo.component.html :

```
<div>
    <h1>Event Demonstration</h1>


    <!-- Mouse Events -->
    <div
      (mouseover)="onMouseEvent('Mouse Over')"
      (mouseleave)="onMouseEvent('Mouse Leave')"
      (click)="onMouseEvent('Mouse Click')"
      class="mouse-area"
    >
      Hover, Click, or Leave this box
    </div>
    <p>{{ mouseEvent }}</p>


    <!-- Keyboard Events -->
```

```
    <label for="keyInput">Type something:</label>

    <input id="keyInput" type="text"
(keyup)="onKeyPress($event)" />

    <p>{{ message }}</p>



    <!-- Blur Event -->

    <label for="blurInput">Blur Example:</label>

    <input id="blurInput" type="text" (blur)="onBlur()" />

    <p>{{ blurMessage }}</p>

  </div>
```

event-demo.component.ts :

```
import { Component } from '@angular/core';

import { CommonModule } from '@angular/common';



@Component({

  selector: 'app-event-demo',

  standalone: true,

  imports: [CommonModule],

  templateUrl: './event-demo.component.html',

  styleUrls: ['./event-demo.component.css']
```

```
})
export class EventDemoComponent {
  message: string = '';
  mouseEvent: string = '';
  blurMessage: string = ''; // Ensure this property is defined

  onKeyPress(event: KeyboardEvent): void {
    const key = event.key;
    this.message = `You pressed: ${key}`;
  }

  onMouseEvent(eventType: string): void {
    this.mouseEvent = `Mouse Event: ${eventType}`;
  }

  onBlur(): void {
    this.blurMessage = 'Input field lost focus';
  }
}
```

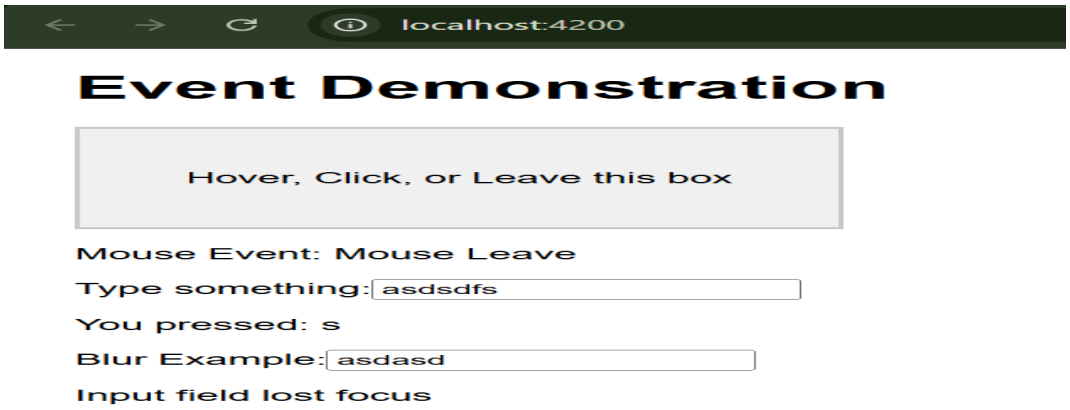
app.component.ts:

```
import { Component } from '@angular/core';

import { EventDemoComponent } from
'./event-demo/event-demo.component';

@Component({

  selector: 'app-root',

  standalone: true,

  imports: [EventDemoComponent],

  template: '<app-event-demo></app-event-demo>',

  styleUrls: ['./app.component.css']

})

export class AppComponent { }
```

App.component.html :

```
<app-event-demo></app-event-demo>
```

Output :



5. **Create an application using pipes-**

   Create an Array to store the information of 5 employees (empid, name, salary, doj) and apply relevant pipes to

   i) convert name to Title Case

   ii) apply currency and number pipe

   iii) apply date pipe

Ans :

Source code :

Employee-list.component.html :

```
<h2>Employee List</h2>

<table border="1">

  <thead>

    <tr>

      <th>Emp ID</th>

      <th>Name</th>

      <th>Salary</th>

      <th>Date of Joining</th>

    </tr>

  </thead>

  <tbody>

    <tr *ngFor="let employee of employees">
```

```
      <td>{{ employee.empid }}</td>

      <td>{{ employee.name | titlecase }}</td>

      <td>{{ employee.salary | currency: 'USD':'symbol':'1.2-2'
}}</td>

      <td>{{ employee.doj | date: 'MMMM d, y' }}</td>

    </tr>

  </tbody>

</table>
```

employee-list.component.ts:

```
import { Component } from '@angular/core';

import { CommonModule } from '@angular/common'; // Import
CommonModule



@Component({

  selector: 'app-employee-list',

  standalone: true,

  imports: [CommonModule], // Include CommonModule to enable
built-in pipes like 'date'

  templateUrl: './employee-list.component.html',

  styleUrls: ['./employee-list.component.css']

})
```

```
export class EmployeeListComponent {

  employees = [

    { empid: 1, name: 'john doe', salary: 50000, doj: new
Date('2020-01-15') },

    { empid: 2, name: 'jane smith', salary: 60000, doj: new
Date('2019-06-20') },

    { empid: 3, name: 'michael johnson', salary: 55000, doj: new
Date('2021-08-10') },

    { empid: 4, name: 'sarah connor', salary: 75000, doj: new
Date('2018-03-25') },

    { empid: 5, name: 'alexander hamilton', salary: 65000, doj:
new Date('2022-12-01') },

  ];

}
```

App.component.html :

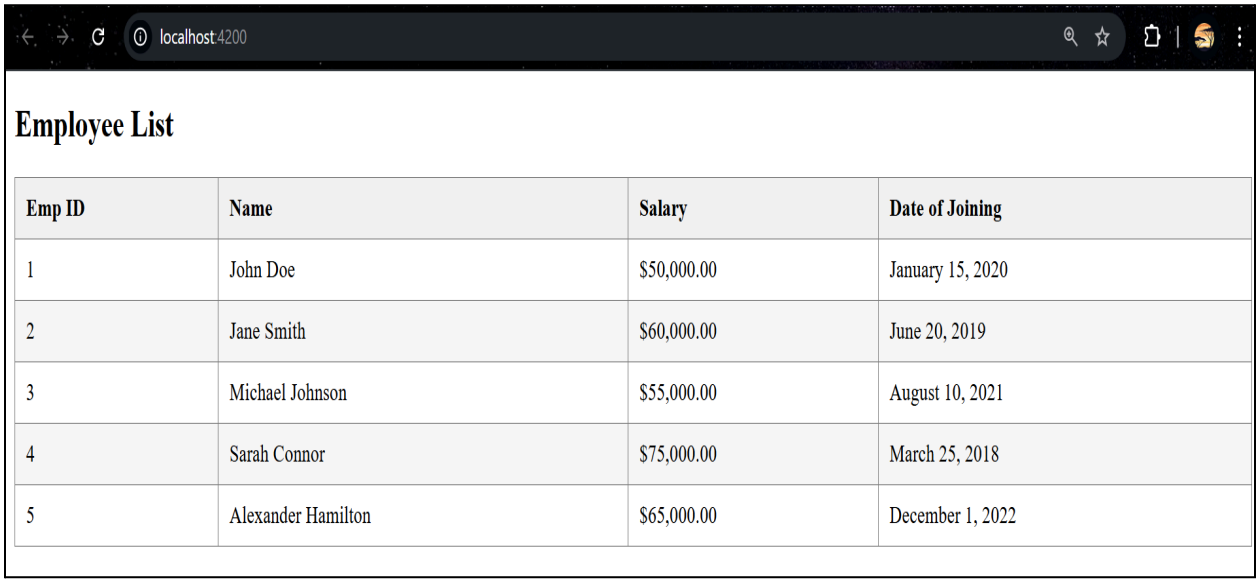
```
<app-employee-list></app-employee-list>
```

app.component.ts :

```
import { Component } from '@angular/core';

import { EmployeeListComponent } from
'./employee-list/employee-list.component'; // Import the
standalone component
```

```
@Component({

  selector: 'app-root',

  standalone: true,

  imports: [EmployeeListComponent], // Import the standalone
component here

  templateUrl: './app.component.html',

  styleUrls: ['./app.component.css']

})

export class AppComponent {

  title = 'employee-app';

}
```

Output :



6. **Create a list of grocery items. Add the items entered by the user into the same list by calling the "AddItem" method on the button.**

Ans :

Source code :

grocery-list.component.ts :

```
import { Component } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { NgFor, NgIf } from '@angular/common';


@Component({
  selector: 'app-grocery-list',
  standalone: true,
  imports: [FormsModule, NgFor, NgIf],
  template: `
    <div class="grocery-list">
      <h1>Grocery List</h1>
      <input
        type="text"
        placeholder="Enter grocery item"
```

```
      [(ngModel)]="newItem"
      (keyup.enter)="addItem()"

    />
    <button (click)="addItem()">Add Item</button>

    <ul>
      <li *ngFor="let item of items">{{ item }}</li>
    </ul>
  </div>
`,
styles: [`
  .grocery-list {
    max-width: 400px;
    margin: 0 auto;
    text-align: center;
  }

  input {
    width: 70%;
    padding: 8px;
```

```
    margin: 8px 0;
  }

  button {
    padding: 8px 16px;
    margin: 8px 0;
    cursor: pointer;
  }

  ul {
    list-style-type: none;
    padding: 0;
  }

  li {
    background-color: #f4f4f4;
    margin: 4px 0;
    padding: 8px;
    border-radius: 4px;
  }
`]
```

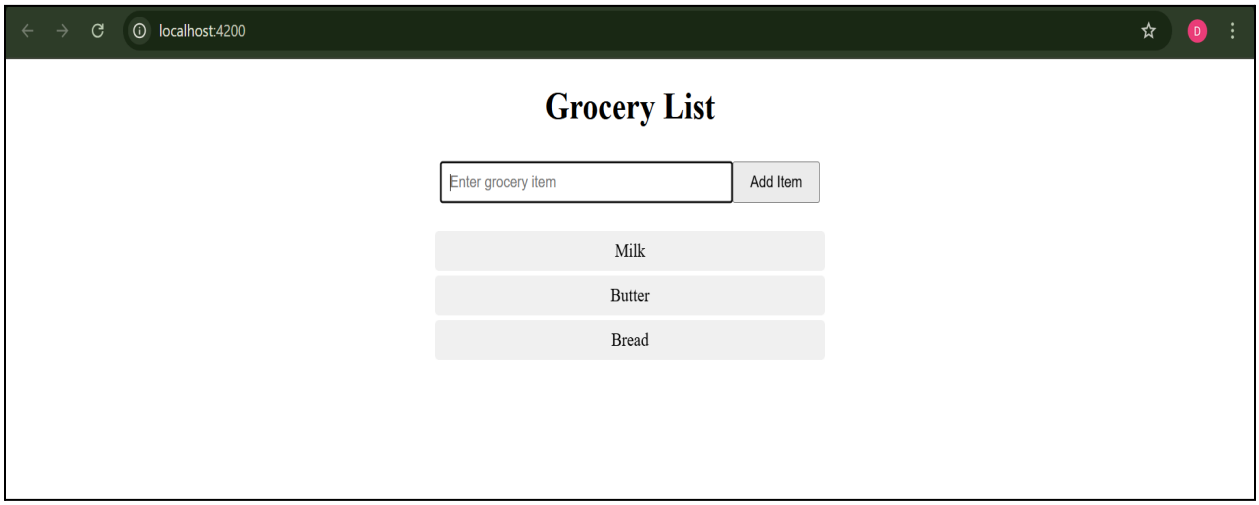
```
})


export class GroceryListComponent {
  items: string[] = [];
  newItem: string = '';

  addItem() {
    if (this.newItem.trim()) {
      this.items.push(this.newItem.trim());
      this.newItem = '';
    }
  }
}
```

app.component.ts :

```
import { Component } from '@angular/core';
import { RouterOutlet } from '@angular/router';
import { GroceryListComponent } from
'./grocery-list/grocery-list.component';
@Component({
  selector: 'app-root',
```

```
  imports: [RouterOutlet,GroceryListComponent],
  templateUrl: './app.component.html',
  styleUrl: './app.component.css'
})
export class AppComponent {
  title = 'grocery-app';
}
```

Output :



Conclusion : Learned to create angular applications.

Name of Student: Shaikh Farhan Ahmed		
Roll Number: 50	**Lab Assignment Number: 10**	
Title of Lab Assignment: Create a registration form for a college event containing fname, lName,gender, class, dept, type of event, contact, email. Put validation, on button click display the entered data		
DOP: 27-11-2024	**DOS: 07-12-2024**	
CO Mapped: **CO5**	**PO Mapped:** **PO3, PO5, PSO1,PSO2**	**Signature:**

Practical 10

AIM : Create a registration form for a college event containing fname, lName,gender, class, dept, type of event, contact, email. Put validation, on button click display the entered data

Theory :

Form Handling in Angular

In Angular, form handling is crucial for capturing user input and interacting with backend systems. Angular provides two primary ways to handle forms: **Template-driven forms** and **Reactive forms**. Template-driven forms are simple to set up and are ideal for basic forms, relying heavily on Angular's template syntax and directives like `ngModel`, `ngForm`, and `ngModelGroup`. This approach is best suited for simple forms with fewer validation requirements and minimal logic in the component.

Reactive forms, on the other hand, offer more control and are suited for complex forms that require advanced validation, dynamic form controls, or custom logic. They are built using the `FormGroup`, `FormControl`, and `FormArray` classes, allowing developers to manage form state explicitly in the component. This approach enables better scalability and maintainability, especially in larger applications, as form logic is encapsulated in the component rather than the template.

Angular's form handling also includes built-in form validation, both synchronous and asynchronous, allowing developers to enforce rules like required fields, minimum/maximum values, and pattern matching. The `Validators` class provides a set of predefined validators, and custom validators can be created to meet specific business requirements. Additionally, Angular supports form submission, resetting forms, and handling form state (valid/invalid, touched/untouched), making it a powerful tool for building interactive and dynamic web applications.

Create a registration form for a college event containing fname, lName,gender, class, dept, type of event, contact, email. Put validation, on button click display the entered data

Ans :
Source Code :

registration-form.component.ts :

```
import { Component } from '@angular/core';
import { FormGroup, FormBuilder, Validators } from '@angular/forms';
import { CommonModule } from '@angular/common';  // Import CommonModule
import { ReactiveFormsModule } from '@angular/forms';  // Correct
ReactiveFormsModule import

@Component({
  selector: 'app-registration-form',
  standalone: true,
  imports: [CommonModule, ReactiveFormsModule],  // Add
ReactiveFormsModule here
  template: `
    <div class="form-container">
      <h2>Event Registration Form</h2>
      <form [formGroup]="registrationForm" (ngSubmit)="onSubmit()">
        <!-- Form fields go here -->
        <label for="fname">First Name</label>
        <input type="text" id="fname" formControlName="fname" />

        <label for="lname">Last Name</label>
        <input type="text" id="lname" formControlName="lname" />

        <label for="gender">Gender</label>
        <select id="gender" formControlName="gender">
          <option value="male">Male</option>
          <option value="female">Female</option>
        </select>

        <label for="class">Class</label>
        <input type="text" id="class" formControlName="class" />

        <label for="dept">Department</label>
```

```
        <input type="text" id="dept" formControlName="dept" />

        <label for="eventType">Type of Event</label>
        <input type="text" id="eventType" formControlName="eventType" />

        <label for="contact">Contact Number</label>
        <input type="text" id="contact" formControlName="contact" />

        <label for="email">Email</label>
        <input type="email" id="email" formControlName="email" />

        <button type="submit"
[disabled]="registrationForm.invalid">Submit</button>
      </form>

      <div *ngIf="submittedData">
        <h3>Entered Data:</h3>
        <pre>{{ submittedData | json }}</pre>  <!-- The 'json' pipe will
work now -->
      </div>
    </div>
  `,
  styles: [`
    .form-container {
      width: 400px;
      margin: 0 auto;
      padding: 20px;
      border: 1px solid #ccc;
      border-radius: 5px;
      background: #f9f9f9;
    }
    label {
      display: block;
      margin: 10px 0 5px;
    }
    input, select {
      width: 100%;
      padding: 8px;
      margin: 5px 0 10px;
      border: 1px solid #ccc;
```

```
      border-radius: 4px;
    }
    small {
      color: red;
    }
  `]
})
export class RegistrationFormComponent {
  registrationForm: FormGroup;
  submittedData: any;

  constructor(private fb: FormBuilder) {
    this.registrationForm = this.fb.group({
      fname: ['', Validators.required],
      lname: ['', Validators.required],
      gender: ['', Validators.required],
      class: ['', Validators.required],
      dept: ['', Validators.required],
      eventType: ['', Validators.required],
      contact: ['', [Validators.required,
Validators.pattern('^[0-9]{10}$')]],
      email: ['', [Validators.required, Validators.email]],
    });
  }

  onSubmit() {
    if (this.registrationForm.valid) {
      this.submittedData = this.registrationForm.value;
    }
  }
}
```

app.component.ts :

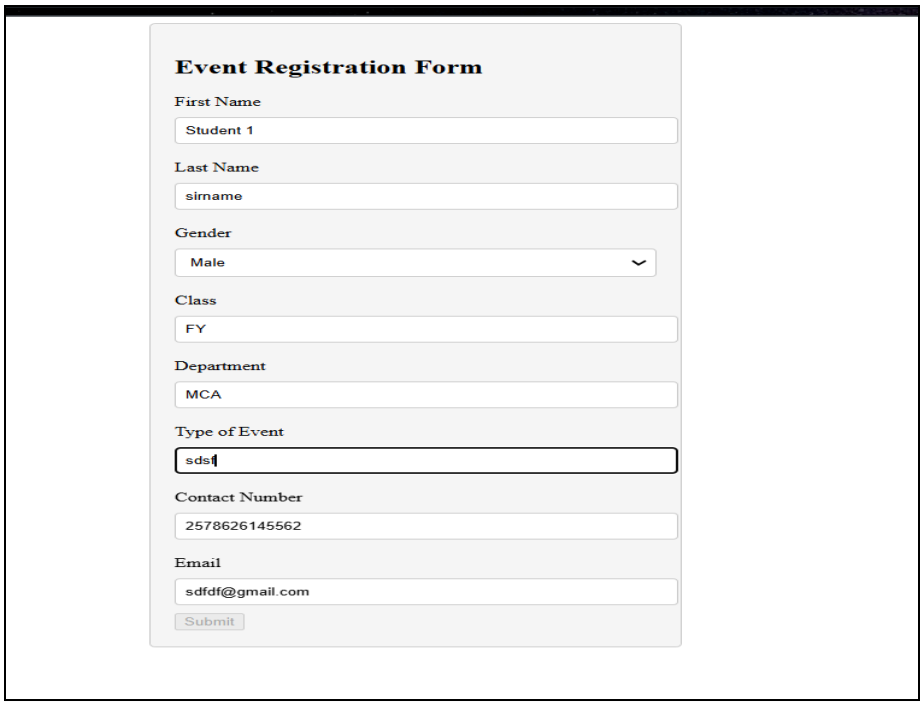
```
import { Component } from '@angular/core';
import { RouterOutlet } from '@angular/router';
import { RegistrationFormComponent } from
'./registration-form/registration-form.component';

@Component({
```

```
  selector: 'app-root',
  imports: [RouterOutlet,RegistrationFormComponent],
  templateUrl: './app.component.html',
  styleUrl: './app.component.css'
})
export class AppComponent {
  title = 'college-event';
}
```

main.ts :

```
import { bootstrapApplication } from '@angular/platform-browser';
import { appConfig } from './app/app.config';
import { AppComponent } from './app/app.component';

bootstrapApplication(AppComponent, appConfig)
  .catch((err) => console.error(err));
```

Output :



Entered Data:


```
{
  "fname": "Student 1",
  "lname": "sirname",
  "gender": "male",
  "class": "FY",
  "dept": "MCA",
  "eventType": "sdsf",
  "contact": "1234567892",
  "email": "sdfdf@gmail.com"
}
```


Conclusion : I learned to create and Handle Forms using angular.js