5a. Create an array of objects having movie details. The object should include the movie name, starring, language, and ratings. Render the details of movies on the page using the array.

DESCRIPTION:

Creating Arrays: Arrays can be created in JavaScript by enclosing a comma-separated list of values in square brackets []. An array can contain any type of data, including numbers, strings, objects, or other arrays.

Destructuring Arrays: Array destructuring is a feature in JavaScript that allows you to unpack values from an array into separate variables. You can use destructuring to assign array values to variables with a more concise syntax

Array Methods: JavaScript provides a number of built-in methods for working with arrays. Some of the most commonly used methods include:

push(): Adds one or more elements to the end of an array. pop(): Removes the last element from an array and returns it. shift(): Removes the first element from an array and returns it.

unshift(): Adds one or more elements to the beginning of an array.

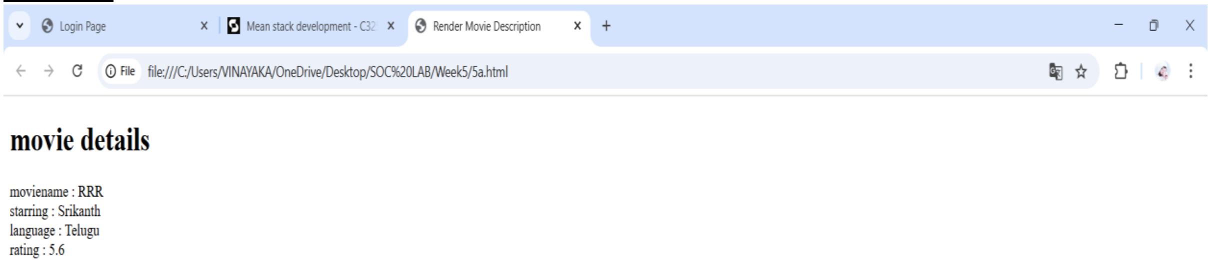
concat(): Joins two or more arrays together.

slice(): Returns a portion of an array as a new array

PROGRAM:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>Render Movie Description</title>
</head>
<body>
<h1>movie details</h1>
  <script>
    var moviedesc = {'moviename':'RRR', 'starring' : 'Srikanth', 'language' : 'Telugu', 'rating' : 5.6}
    for(var key in moviedesc){
      document.write(key+" : "+moviedesc[key]+"<br>")
    }
  </script>
</body>
</html>
```

OUTPUT:


Login Page
Mean stack development - C32
Render Movie Description
File
file:///C:/Users/VINAYAKA/OneDrive/Desktop/SOC%20LAB/Week5/5a.html

movie details

```
moviename : RRR
starring : Srikanth
language : Telugu
rating : 5.6
```

5b) Simulate a periodic stock price change and display on the console. Hints: (i) Create a method which returns a random number - use Math.random, floor and other methods to return a rounded value. (ii) Invoke the method for every three seconds and stop when

DESCRIPTION: Introduction to Asynchronous Programming: Asynchronous programming is a programming paradigm that allows multiple tasks to run concurrently. This means that a program can perform multiple tasks simultaneously without blocking the execution of other tasks. In JavaScript, asynchronous programming is essential for handling long-running operations, such as network requests or file operations, without freezing the user interface.

Callbacks: A callback is a function that is passed as an argument to another function and is executed after some operation is completed. In JavaScript, callbacks are often used in asynchronous programming to handle the results of asynchronous operations. For example, when making a network request, a callback function can be used to handle the response once it is received.

Promises: A promise is an object that represents the eventual completion or failure of an asynchronous operation and its resulting value. A promise has three states: pending, fulfilled, or rejected. Promises are used to handle asynchronous operations in a more readable and maintainable way than callbacks. Promises provide a chainable syntax that allows you to handle the result of an asynchronous operation without nesting callbacks.
Async and Await: Async/await is a newer syntax introduced in ES2017 that allows you to write asynchronous code that looks synchronous. It is built on top of promises and provides a simpler and more concise way to handle asynchronous operations. Async functions always return a promise, and the await keyword can be used to wait for the resolution of a promise.

PROGRAM:

```
<!DOCTYPE html>
<html>
<head>
<title>Stock Price Simulator</title>
<meta charset="UTF-8">
</head>
<body>
<h1>Stock Price Simulator</h1>
<p>Click the button to start simulating periodic stock price changes.</p>
<button onclick="startSimulation()">Start Simulation</button>
<script>
function getRandomPrice(min, max) {
return Math.floor(Math.random() * (max - min + 1) + min);
}

let intervalId;
function startSimulation() { intervalId = setInterval(() => {
const price = getRandomPrice(50, 150); console.log(`Stock price: $${price}`);
}, 3000);
}

function stopSimulation() { clearInterval(intervalId);
console.log("Simulationstopped.");
}

</script>
</body>
```

```
</html>
```

OUTPUT:

```
Stock price: $87
 Stock price: $112
Stock price: $64
Stock price: $93
```

The simulation will continue to generate and display stock prices every three seconds until you stop it by calling the stopSimulation( ) function or closing the browser tab.

5c) Validate the user by creating a login module. Hints: (i) Create a file login.js with a User class. (ii) Create a validate method with username and password as arguments. (iii) If the username and password are equal it will return "Login Successful" else w

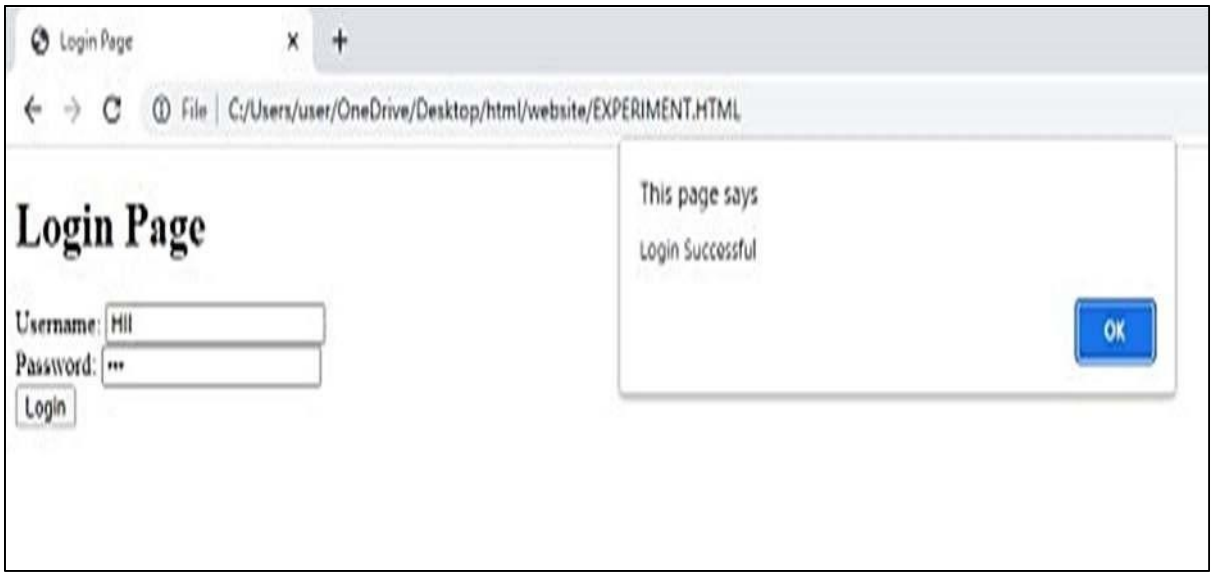
DESCRIPTION: In JavaScript, you can create a module by defining an object or function in a separate file and exporting it using the keyword. You can then import the module into another file using the keyword.

PROGRAM:

```
<!DOCTYPE html
<html>
<head>
<title>Login Page</title>
</head>
<body>
<h1>Login Page</h1>
<form>
<label for="username">Username:</label>
<input type="text" id="username" name="username" required><br>
<label for="password">Password:</label>
<input type="password" id="password" name="password" required><br>
<button type="button" onclick="validateLogin()">Login</button>
</form>
<script> class User {
constructor(username, password) { this.username = username; this.password
password;
}
validate() {

if (this.username === this.password) { return "Login Successful";
} else {

return "Invalid username or password";

}

}

}

function validateLogin() {

const username = document.getElementById("username").value; const password =
document.getElementById("password").value;

const user = new User(username, password); const validationMsg = user.validate();
alert(validationMsg);
}
```

```
</script>
</body>
</html>
```

OUTPUT:



This page says

Login Successful

6a).Verify how to execute different functions successfully in the Node.js platform.

DESCRIPTION: Node.js is an open-source, cross-platform, server-side JavaScript runtime environment that allows developers to build scalable and efficient web applications. Here are the basic steps to use Node.js:

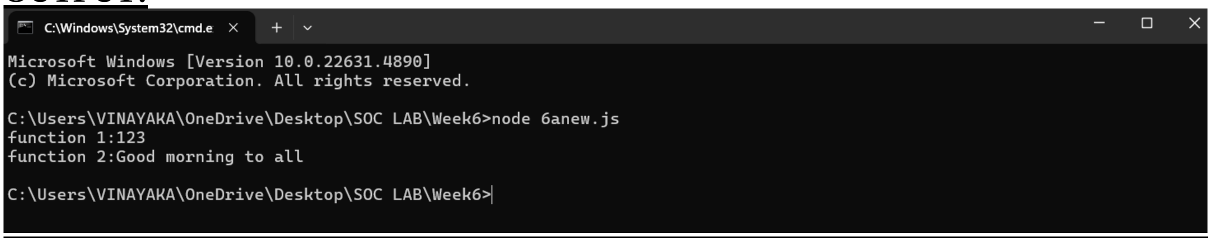
Install Node.js on your computer by downloading the installer from the official Node.js website. Create a new project directory for your Node.js application.

Initialize the project by running the **npm init** command in the terminal. This will create a **package.json** file for your project, which will contain the project's metadata and dependencies.

PROGRAM:

```
        function myData()
        {
        return 123;
        }
        console.log("function 1:"+myData());
        function myValue()
        {
        return "Good morning to all";
        }
console.log("function 2:"+myValue());
```

OUTPUT:



```
Microsoft Windows [Version 10.0.22631.4890]
(c) Microsoft Corporation. All rights reserved.

C:\Users\VINAYAKA\OneDrive\Desktop\SOC LAB\Week6>node 6anew.js
function 1:123
function 2:Good morning to all

C:\Users\VINAYAKA\OneDrive\Desktop\SOC LAB\Week6>
```

6b). Write a program to show the workflow of JavaScript code executable by creating web server in Node.js

DESCRIPTION:

To create a web server in Node.js, you can follow these steps:

1. Open your preferred code editor and create a new file with a .js extension. Give it a meaningful name, such as **server.js.**

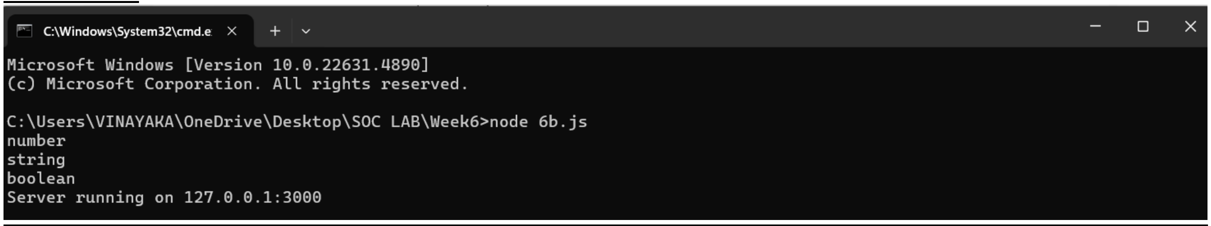
2. Require the built-in http module at the top of your file: const http = require('http');
3. Create a new server instance using the **http.createServer()** method:
4. const server = http.createServer((req, res) => { // Code to handle incoming requests });
5. Inside the server instance, define how your server should handle incoming requests. For example, you can use the res.writeHead() method to set the response header, and **res.end()** method to send the response body:
const server = http.createServer((req, res) => { res.writeHead(200, { 'Content-Type': 'text/plain'
}); res.end('Hello, world!'); });
This code will send a 200 OK status code and the message "Hello, world!" to any client that sends a request to the server.
6.Set the server to listen for incoming requests on a specific port using the server.listen() method:
const port = 3000; server.listen(port, () => { console.log(`Server running at http://localhost:${port}/`); });
This code will start the server on port 3000 and log a message to the console to confirm that the server is running.

PROGRAM:

```
var a = 8;
var b = "sandy";
var d = true;
console.log(typeof(a));
console.log(typeof(b));
console.log(typeof(d));

var http = require("http");
const host = "127.0.0.1";
const port = 3000;

var server = http.createServer(function(req, res) {
  res.writeHead(200, { 'Content-Type': 'text/plain' });
  res.end('Hello, World!\n');
});
```

```
        server.listen(port, host, function() {
          console.log("Server running on " + host + ":" + port);
});
```

OUTPUT:



```
C:\Windows\System32\cmd.e

Microsoft Windows [Version 10.0.22631.4890]
(c) Microsoft Corporation. All rights reserved.

C:\Users\VINAYAKA\OneDrive\Desktop\SOC LAB\Week6>node 6b.js
number
string
boolean
Server running on 127.0.0.1:3000
```

6c) Write a Node.js module to show the workflow of Modularization of Node application

DESCRIPTION: Modular programming is a programming paradigm in which software is composed of separate, reusable modules that can be combined to create complex applications. In Node.js, modular programming is particularly important due to the nature of JavaScript as an asynchronous, event- driven language.

Node.js makes it easy to create modular programs by providing a built-in **module** object that can be used to define and export modules. To define a module in Node.js, you simply create a new JavaScript file and define one or more functions or objects that you want to export using the **module.exports** object.

PROGRAM:

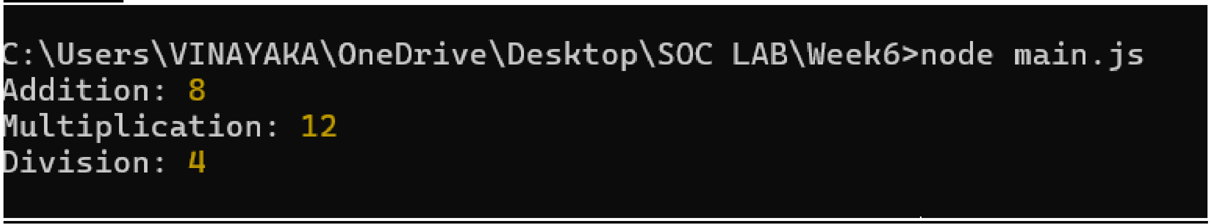
Main.js

```
//Importing the module
const calculator  = require('./calculator');
console.log("Addition:",calculator.add(5,3));
console.log("Multiplication:",calculator.multiply(2,6));
console.log("Division:", calculator.divide(20,5));
```

calculator.js

```
        function add(a,b)
        {
        return a+b;
        }
        function substract(a,b)
        {
        return a-b;
        }

        function multiply(a,b)
        {
        return a*b;
        }
    function divide(a,b)
        {
        if(b==0)
        {
        return "Cannot divide by zero";
        }
        return a/b;
        }
module.exports={add,multiply,divide};
```

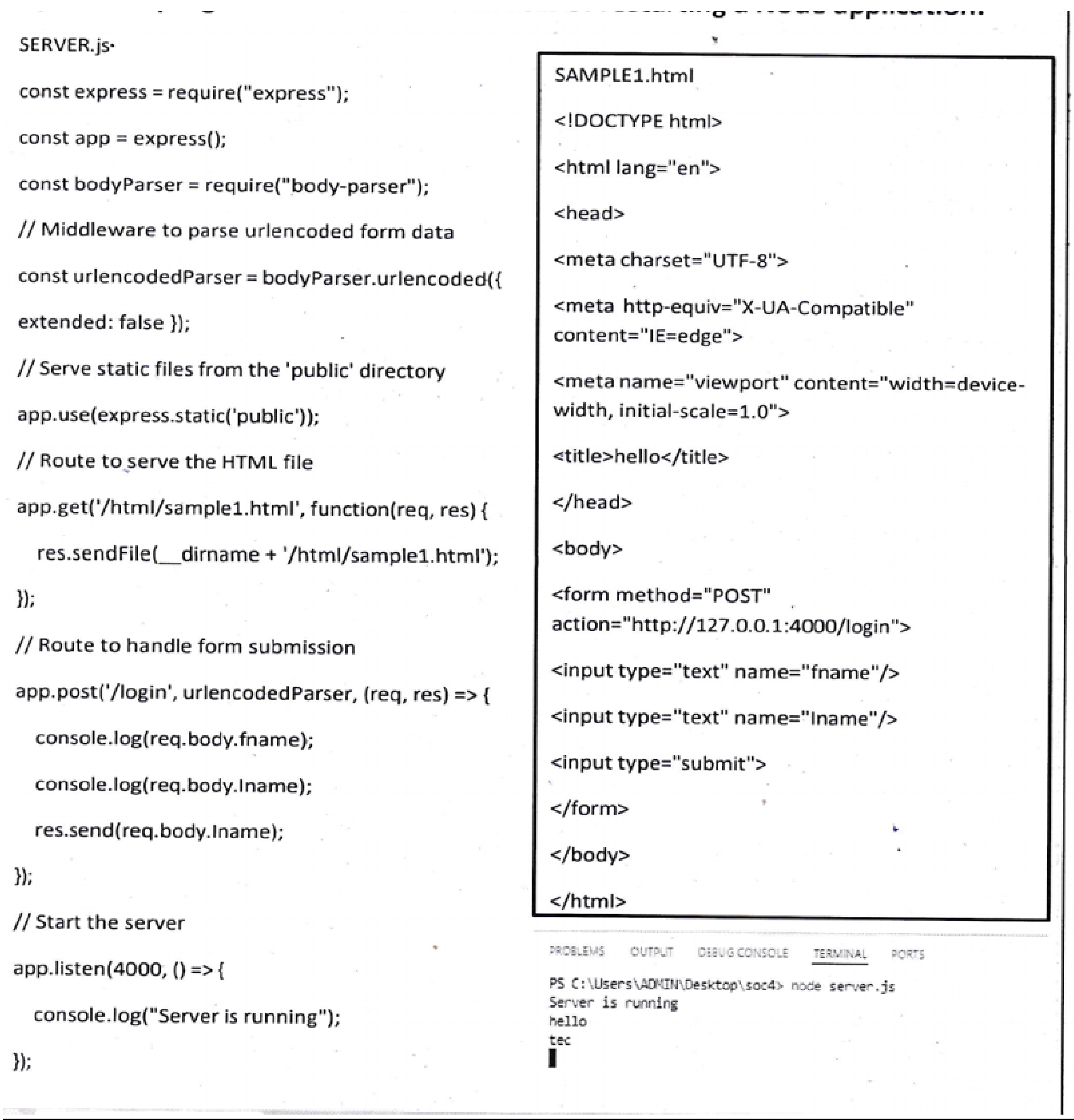
OUTPUT:



```
C:\Users\VINAYAKA\OneDrive\Desktop\SOC LAB\Week6>node main.js
Addition:  8
Multiplication:  12
Division:  4
```

6d). Write a program to show the workflow of restarting a Node application.

DESCRIPTION: There are several ways to restart a Node.js application, depending on the circumstances. If you are running your Node application using the **node** command in your terminal, you can simply stop the current instance of the application by pressing **ctrl + c** and then start a new instance by running the **node** command again.



SERVER.js·

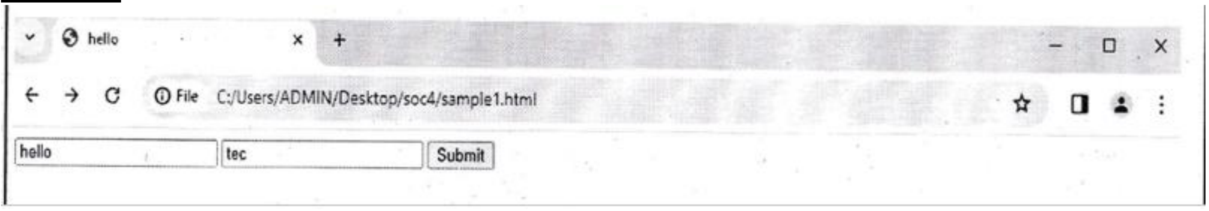
```
const express = require("express");
const app = express();
const bodyParser = require("body-parser");
// Middleware to parse urlencoded form data
const urlencodedParser = bodyParser.urlencoded({
extended: false });
// Serve static files from the 'public' directory
app.use(express.static('public'));
// Route to serve the HTML file
app.get('/html/sample1.html', function(req, res) {
   res.sendFile(__dirname + '/html/sample1.html');
});
// Route to handle form submission
app.post('/login', urlencodedParser, (req, res) => {
   console.log(req.body.fname);
   console.log(req.body.lname);
   res.send(req.body.lname);
});
// Start the server
app.listen(4000, () => {
   console.log("Server is running");
});
```

SAMPLE1.html

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta http-equiv="X-UA-Compatible"
content="IE=edge">
<meta name="viewport" content="width=device-
width, initial-scale=1.0">
<title>hello</title>
</head>
<body>
<form method="POST"
action="http://127.0.0.1:4000/login">
<input type="text" name="fname"/>
<input type="text" name="lname"/>
<input type="submit">
</form>
</body>
</html>
```

```
PS C:\Users\ADMIN\Desktop\soc4> node server.js
Server is running
hello
tec
```

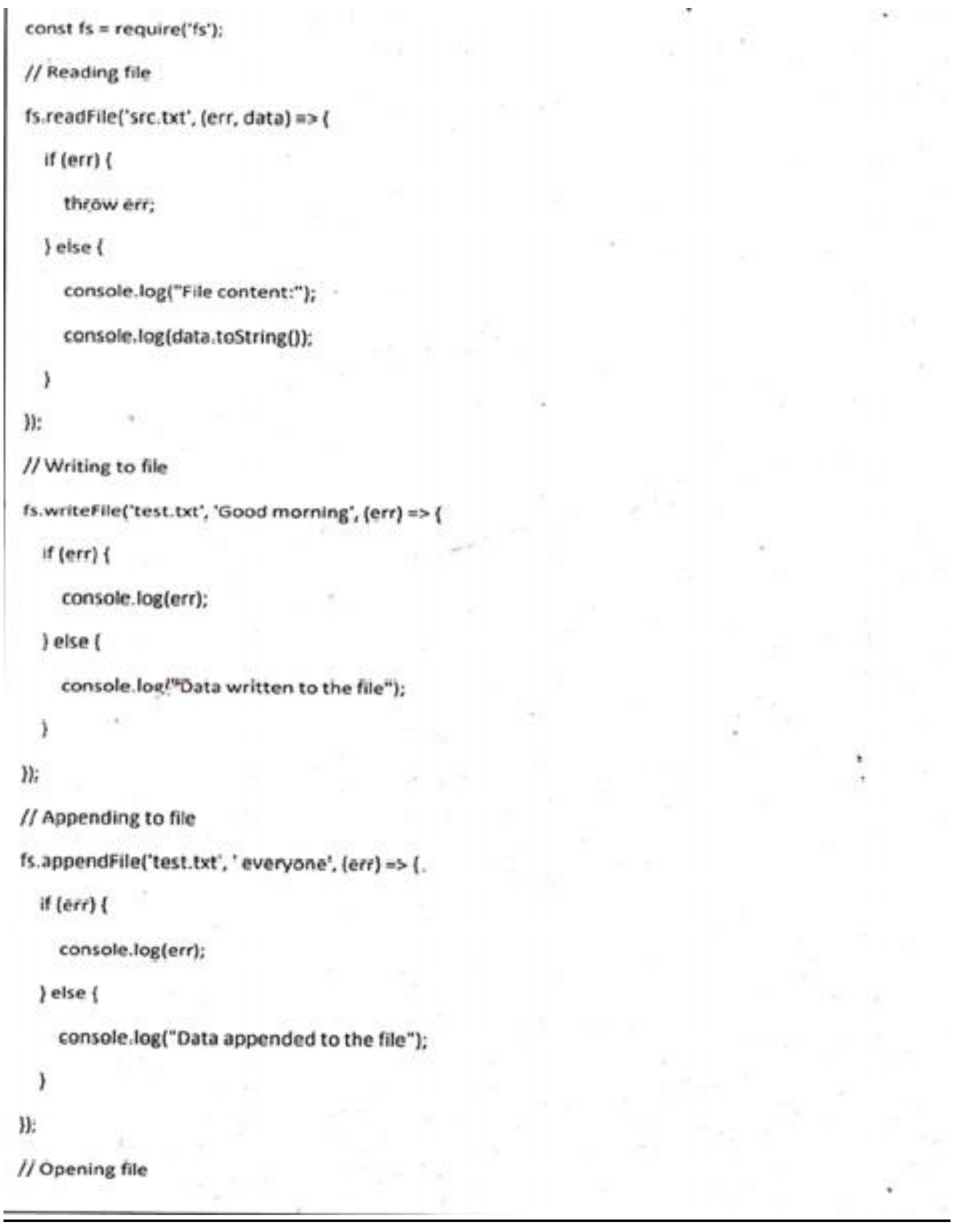
OUTPUT:



```
C:/Users/ADMIN/Desktop/soc4/sample1.html
```

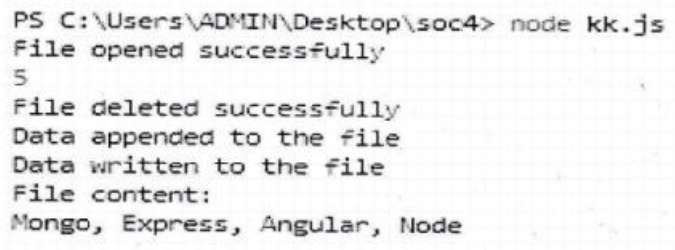
hello	tec	Submit

6e) Create a text file src.txt and add the following data to it. Mongo, Express, Angular, Node.



```
const fs = require('fs');
// Reading file
fs.readFile('src.txt', (err, data) => {
  if (err) {
    throw err;
  } else {
    console.log("File content:");
    console.log(data.toString());
  }
});
// Writing to file
fs.writeFile('test.txt', 'Good morning', (err) => {
  if (err) {
    console.log(err);
  } else {
    console.log("Data written to the file");
  }
});
// Appending to file
fs.appendFile('test.txt', ' everyone', (err) => {
  if (err) {
    console.log(err);
  } else {
    console.log("Data appended to the file");
  }
});
// Opening file
```

```
fs.open('test.txt', 'r+', (err, f) => {

  if (err) {

    console.log(err);

  } else {

    console.log("File opened successfully");

    console.log(f);

  }

});

// Deleting file

fs.unlink('test.txt', (err) => {

  if (err) {

    console.log(err);

  } else {

    console.log("File deleted successfully");

  }

});
```



```
PS C:\Users\ADMIN\Desktop\soc4> node kk.js
File opened successfully
5
File deleted successfully
Data appended to the file
Data written to the file
File content:
Mongo, Express, Angular, Node
```