**Practical-1**

**AIM:** **Portfolio: Designing a Stunning Website with HTML ,CSS, and JavaScript.**

**Source Code:** [**https://trushang-patel.github.io/CV-WEBSITE/?authuser=0**](https://trushang-patel.github.io/CV-WEBSITE/?authuser=0)

**Output:**

**A screenshot of a computer

Description automatically generated**

**A screenshot of a computer

Description automatically generated**

**A screenshot of a computer

Description automatically generated**

**Practical-2**

**Aim:**

Create a Product List page (PLP) using html, css, javascript  
  
use fetch api to make api calls  
<https://fakestoreapi.com/docs>  
  
Fetch Api  
<https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API>

**Source Code:**

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

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

    <title>Product List Page</title>

    <style>

        body {

            font-family: Arial, sans-serif;

            margin: 0;

            padding: 0;

            background-color: #f4f4f4;

        }

        header {

            background-color: #333;

            color: #fff;

            padding: 1em;

            text-align: center;

        }

        .product-container {

            display: flex;

            flex-wrap: wrap;

            justify-content: space-around;

            padding: 20px;

        }

        .product-card {

            width: 300px;

            margin: 20px;

            padding: 10px;

            border: 1px solid #ddd;

            border-radius: 5px;

            background-color: #fff;

        }

        img {

            max-width: 100%;

            height: auto;

        }

    </style>

</head>

<body>

    <header>

        <h1>Product List Page</h1>

    </header>

    <div class="product-container" id="productContainer"></div>

    <script>

        document.addEventListener('DOMContentLoaded', () => {

            fetch('https://fakestoreapi.com/products')

                .then(response => response.json())

                .then(products => displayProducts(products))

                .catch(error => console.error('Error fetching products:', error));

            function displayProducts(products) {

                const container = document.getElementById('productContainer');

                products.forEach(product => {

                    const card = document.createElement('div');

                    card.className = 'product-card';

                    const image = document.createElement('img');

                    image.src = product.image;

                    image.alt = product.title;

                    const title = document.createElement('h2');

                    title.textContent = product.title;

                    const price = document.createElement('p');

                    price.textContent = `$${product.price}`;

                    card.appendChild(image);

                    card.appendChild(title);

                    card.appendChild(price);

                    container.appendChild(card);

                });

            }

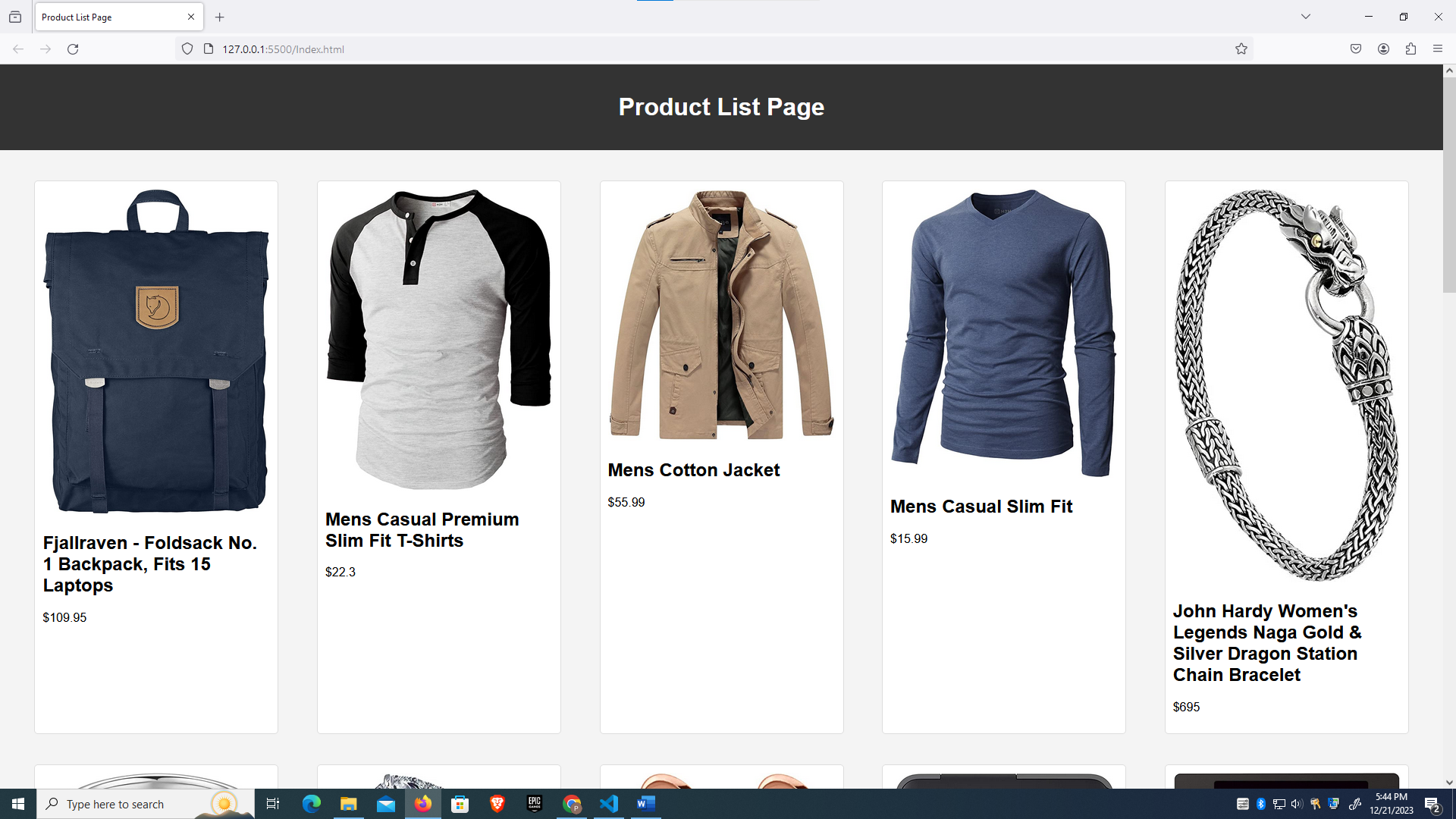
        });

    </script>

</body>

</html>

**Output:**



**Learning Outcome:**

Overall, the provided code examples cover various important aspects of web development, including HTML structure, event handling,CSS and asynchronous JavaScript.

**Practical-3**

**Task-1**

**Aim:**

Variables and Data Types

Declare a variable using var, let, and const. Assign different data types to each variable and print their values.

**Description:**

**var** variables can be redeclared and reassigned**, let** variables can be reassigned but not redeclared in the same scope, and **const** variables cannot be redeclared or reassigned.

**Source Code:**

let a = 10;

var n;

const m=5;

n=2;

console.log(a);

console.log(n);

console.log(m);

**Output:**



**Task-2**

**Aim:**

Operators and Expressions

Write a function that takes two numbers as arguments and returns their sum, difference, product, and quotient using arithmetic operators.

**Description:**

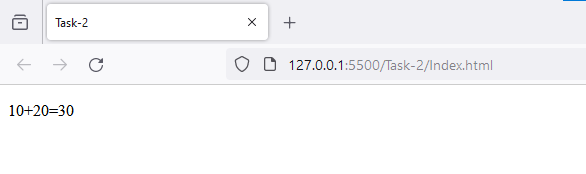
operators are symbols that are used to perform operations on operands. An **expression** is a block of code that evaluates to a value. A **statement** is any block of code that is performing some action.

**Source Code:**

let a = parseInt(prompt("Enter first number: "))+prompt("Enter the operator (+,-,\*,/)")+parseInt(prompt("Enter second number : "))

document.getElementById("demo").innerText=`${a}=${eval(a)}`

**Output:**



**Task-3**

**Aim:**

Control Flow

Write a program that prompts the user to enter their age. Based on their age, display different messages:

○ If the age is less than 18, display "You are a minor."

○ If the age is between 18 and 65, display "You are an adult."

○ If the age is 65 or older, display "You are a senior citizen."

**Description:**

* Use if to specify a block of code to be executed, if a specified condition is true
* Use else to specify a block of code to be executed, if the same condition is false
* Use else if to specify a new condition to test, if the first condition is false

**Source Code:**

let a = parseInt(prompt("Enter your age: "))

document.getElementById("demo").innerText+="Age is "+a;

if(a<18)

    document.getElementById("demo").innerText+="\nYou are a minor."

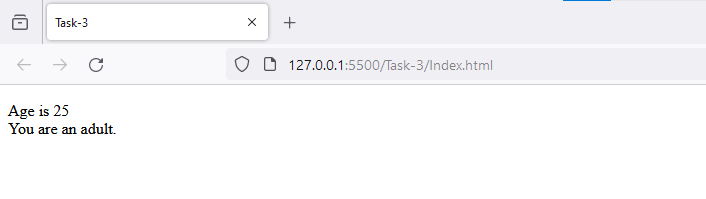
else if(a>18 && a<65)

    document.getElementById("demo").innerText+="\nYou are an adult."

else

    document.getElementById("demo").innerText+="\nYou are a senior citizen."

**Output:**



**Task-4**

**Aim:**

Functions

Write a function that takes an array of salary as an argument and returns the min/max salary in the array.

**Description:**

A JavaScript function is a block of code designed to perform a particular task.

**Source Code:**

function getMinMax(arr) {

    const minmax = {};

    arr.sort((a, b) => a - b);

    minmax.min = arr[0];

    minmax.max = arr[arr.length - 1];

    return minmax;

}

function main() {

    const arr = [1000, 11, 445, 1, 330, 3000];

    const minmax = getMinMax(arr);

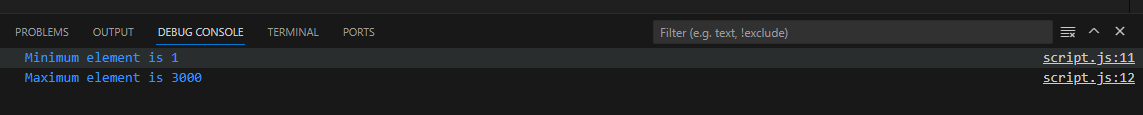
    console.log("Minimum element is " + minmax.min);

    console.log("Maximum element is " + minmax.max);

}

main();

**Output:**



**Task-5**

**Aim:**

Arrays and Objects

Create an array of your favorite books. Write a function that takes the array as an

argument and displays each book title on a separate line.

**Description:**

An array can hold many values under a single name, and you can access the values by referring to an index number.

An object is a collection of properties, and a property is an association between a name (or key) and a value. A property's value can be a function, in which case the property is known as a method.

**Source Code:**

const favoriteBooks = [

    "The Hitchhiker's Guide to the Galaxy",

    "1984",

    "To Kill a Mockingbird",

    "The Great Gatsby",

    "Harry Potter and the Sorcerer's Stone",

  ];

  function displayBooks(bookArray) {

    for (let i = 0; i < bookArray.length; i++) {

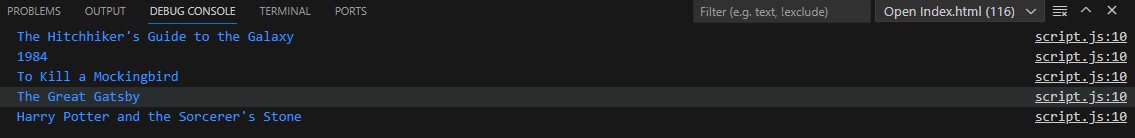
      console.log(bookArray[i]);

    }

  }

  displayBooks(favoriteBooks);

**Output:**



**Task-6**

**Aim:**

Scope and Hoisting

Declare a variable inside a function and try to access it outside the function. Observe the scope behavior and explain the results. [var vs let vs const]

**Description:**

There are two types of scopes.

* Global Scope: Scope outside the outermost function attached to the window.
* Local Scope: Inside the function being executed.

Hoisting: It is a concept that enables us to extract values of variables and functions even before initializing/assigning value without getting errors and this is happening due to the 1st phase (memory creation phase) of the Execution Context.

**Source Code:**

function testScope() {

      var variableInsideFunction = "I am inside the function.";

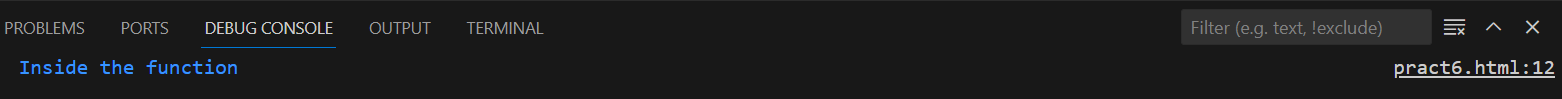
      console.log("Inside the function")

    }    testScope();

    // Uncommenting the line below will result in an error

    // console.log(variableInsideFunction);

**Output:**



**Task-7**

**Aim:**

DOM Manipulation

Create an HTML page with a button. Write JavaScript code that adds an event listener to the button and changes its text when clicked.

**Description:**

The HTML DOM is a standard **object** model and **programming interface** for HTML. It defines:

* The HTML elements as **objects**
* The **properties** of all HTML elements
* The **methods** to access all HTML elements
* The **events** for all HTML elements

In other words:**The HTML DOM is a standard for how to get, change, add, or delete HTML elements.**

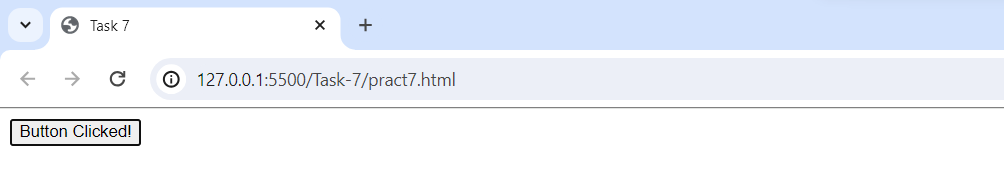
**Source Code:**

document.getElementById("myButton").addEventListener("click", function() {

      this.innerText = "Button Clicked!";

    });

**Output:**



**Task-8**

**Aim:**

Error Handling

Write a function that takes a number as an argument and throws an error if the number is negative. Handle the error and display a custom error message.

**Description:**

When executing JavaScript code, different errors can occur.

Errors can be coding errors made by the programmer, errors due to wrong input, and other unforeseeable things. To solve error we use following concepts:

The try statement defines a code block to run (to try).

The catch statement defines a code block to handle any error.

The finally statement defines a code block to run regardless of the result.

The throw statement defines a custom error.

**Source Code:**

 function checkNegativeNumber(number) {

      if (number < 0)

        throw new Error("Number cannot be negative");

      console.log("Number is positive.");

    }

    // Example usage

    try {

      checkNegativeNumber(-5);

    } catch (error) {

      console.error(error.message);

    }

**Output:**



**Task-9**

**Aim:**

Asynchronous JavaScript

Write a function that uses set Timeout to simulate an asynchronous operation. Use a callback function to handle the result.

**Description:**

asynchronous programming is an essential concept in JavaScript that allows your code to run in the background without blocking the execution of other code. Developers can create more efficient and responsive applications by using features like callbacks, async/await, and promises.

**Source Code:**

function simulateAsyncOperation(callback) {

      setTimeout(function() {

        let result = "Async operation completed.";

        callback(result);

      }, 2000);

    }// Example usage

    simulateAsyncOperation(function(result) {

      console.log(result);

    });

**Output:**



**Learning Outcome:**

Overall, the provided code examples cover various important aspects of web development, including HTML structure, event handling, error management, and asynchronous JavaScript.

**Practical-4**

**Task-1**

**Aim:** Basic Promises

● Create a function fetchData that simulates fetching data from an API using a Promise.

● Inside the function, use setTimeout to simulate a delay of 2 seconds.

● If the data is successfully fetched, resolve the Promise with the fetched data. If there is

an error, reject the Promise with an appropriate error message.

● Implement a function handleData that uses fetchData and handles both the resolved and

rejected cases. Log the data or error message to the console accordingly.

**Description:**

A Promise is a proxy for a value not necessarily known when the promise is created. It allows you to associate handlers with an asynchronous action's eventual success value or failure reason. This lets asynchronous methods return values like synchronous methods: instead of immediately returning the final value, the asynchronous method returns a promise to supply the value at some point in the future.

**Source Code:**

function fetchData() {

  return new Promise((resolve, reject) => {

    setTimeout(() => {

      const success = true;

      if (success) {

        resolve("Data successfully fetched");

      } else {

        reject("Error fetching data");

      }

    }, 2000);

  });

}

function handleData() {

  fetchData()

    .then((data) => {

      console.log("Resolved:", data);

    })

    .catch((error) => {

      console.error("Rejected:", error);

    });

}

 handleData();

**Output:**



**Task-2**

**Aim:** Chaining Promises

● Create three functions: fetchFirstData, fetchSecondData, and fetchThirdData, each

simulating fetching data with a 2-second delay.

● Use Promise chaining to fetch data sequentially: fetch the first data, then the second

data, and finally the third data.

● Log the combined result of all three data fetches.

**Description:**

A common need is to execute two or more asynchronous operations back-to-back, where each subsequent operation starts when the previous operation succeeds, with the result from the previous step.

**Source Code:**

let firstData;

let secondData;

function fetchFirstData() {

  return new Promise((resolve) => {

    setTimeout(() => {

      resolve("First Data");

    }, 2000);

  });

}

function fetchSecondData() {

  return new Promise((resolve) => {

    setTimeout(() => {

      resolve("Second Data");

    }, 2000);

  });

}

function fetchThirdData() {

  return new Promise((resolve) => {

    setTimeout(() => {

      resolve("Third Data");

    }, 2000);

  });

}

fetchFirstData()

  .then((data) => {

    firstData = data;

    return fetchSecondData();

  })

  .then((data) => {

    secondData=data

    return fetchThirdData()

})

  .then((thirdData) => {

    console.log("Combined Result:", firstData, secondData, thirdData);

  })

  .catch((error) => {

    console.error("Error:", error);

  });

**Output:**



**Task-3**

**Aim:** Promise.all

● Create three functions: fetchData1, fetchData2, and fetchData3, each simulating fetching

data with a 2-second delay.

● Use Promise.all to fetch all three sets of data concurrently.

● Log the combined result of all three data fetches.

**Description:**

The Promise.all() static method takes an iterable of promises as input and returns a single Promise. This returned promise fulfills when all of the input's promises fulfill (including when an empty iterable is passed), with an array of the fulfillment values. It rejects when any of the input's promises rejects, with this first rejection reason.

**Source Code:**

 function fetchData1() {

  return new Promise((resolve) => {

    setTimeout(() => {

      resolve("Data from FetchData1");

    }, 2000);

  });

}

function fetchData2() {

  return new Promise((resolve) => {

    setTimeout(() => {

      resolve("Data from FetchData2");

    }, 2000);

  });

}

function fetchData3() {

  return new Promise((resolve) => {

    setTimeout(() => {

      resolve("Data from FetchData3");

    }, 2000);

  });

}

Promise.all([fetchData1(), fetchData2(), fetchData3()])

  .then((results) => {

    console.log("Combined Result:", results);

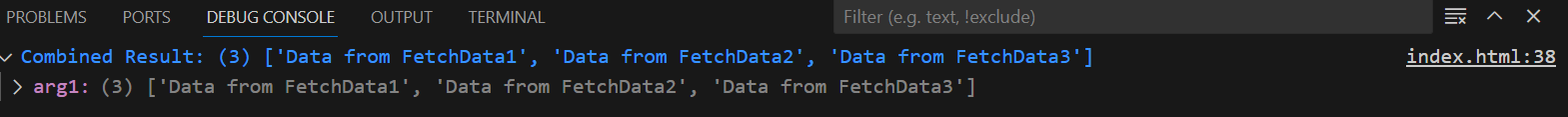
  })

  .catch((error) => {

    console.error("Error:", error);

  });

**Output:**



**Task-4**

**Aim:** Basic Async Function

● Create an asynchronous function fetchData that simulates fetching data from an API

using setTimeout with a delay of 2 seconds.

● Use async/await to handle the asynchronous operation.

● If the data is successfully fetched, return the data. If there is an error, throw an error with

an appropriate message.

● Implement a function handleData that calls fetchData and logs the data or error message

to the console.

**Description:**

The async function declaration creates a binding of a new async function to a given name. The await keyword is permitted within the function body, enabling asynchronous, promise-based behavior to be written in a cleaner style and avoiding the need to explicitly configure promise chains.

**Source Code:**

 async function fetchData() {

  return new Promise((resolve, reject) => {

    setTimeout(() => {

      const success = true;

      if (success) {

        resolve("Data successfully fetched");

      } else {

        reject("Error fetching data");

      }

    }, 2000);

  });

}

async function handleData() {

  try {

    const data = await fetchData();

    console.log("Resolved:", data);

  } catch (error) {

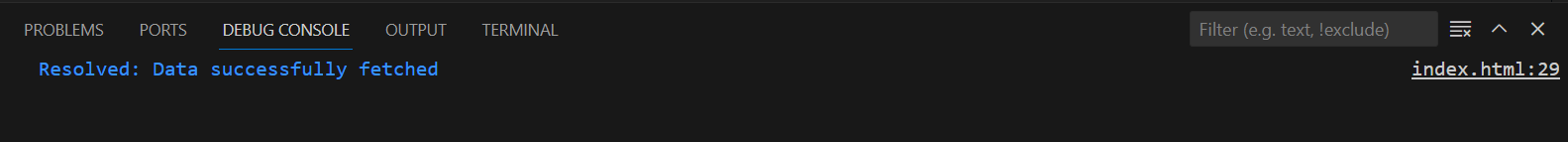
    console.error("Rejected:", error);

  }

}

handleData();

**Output:**

****

**Task-5**

**Aim:** Sequential Async Operations

● Create three asynchronous functions: fetchFirstData, fetchSecondData, and

fetchThirdData, each simulating fetching data with a 2-second delay.

● Use async/await to fetch data sequentially: fetch the first data, then the second data, and

Finally, the third data.

● Log the combined result of all three data fetches.

**Description:**

Sequential Async Operations refer to a scenario where asynchronous tasks are executed in a specific order, one after the other. In JavaScript, this often involves using the async/await syntax to handle promises in a more synchronous-looking manner.

**Source Code:**

async function fetchFirstData() {

  return new Promise((resolve) => {

    setTimeout(() => {

      resolve("First Data");

    }, 2000);

  });

}

async function fetchSecondData() {

  return new Promise((resolve) => {

    setTimeout(() => {

      resolve("Second Data");

    }, 2000);

  });

}

async function fetchThirdData() {

  return new Promise((resolve) => {

    setTimeout(() => {

      resolve("Third Data");

    }, 2000);

  });

}

async function sequentialAsyncOperations() {

  const firstData = await fetchFirstData();

  const secondData = await fetchSecondData();

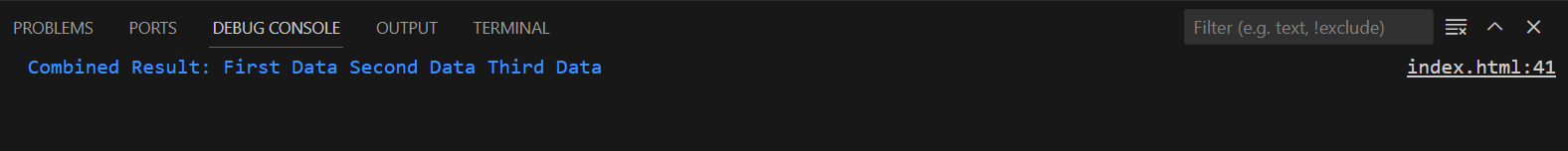
  const thirdData = await fetchThirdData();

  console.log("Combined Result:", firstData, secondData, thirdData);

}

 sequentialAsyncOperations();

**Output:**

****

**Task-6**

**Aim:** Parallel Async Operations

● Create three asynchronous functions: fetchData1, fetchData2, and fetchData3, each

simulating fetching data with a 2-second delay.

● Use Promise.all with async/await to fetch all three sets of data concurrently.

● Log the combined result of all three data fetches.

**Description:**

Parallel Async Operations refer to a scenario where multiple asynchronous tasks are executed concurrently or simultaneously, without waiting for the completion of one before starting the next. In JavaScript, this can be achieved using features such as Promise.all and asynchronous functions.

**Source Code:**

async function fetchData1() {

  return new Promise((resolve) => {

    setTimeout(() => {

      resolve("Data from FetchData1");

    }, 2000);

  });

}

async function fetchData2() {

  return new Promise((resolve) => {

    setTimeout(() => {

      resolve("Data from FetchData2");

    }, 2000);

  });

}

async function fetchData3() {

  return new Promise((resolve) => {

    setTimeout(() => {

      resolve("Data from FetchData3");

    }, 2000);

  });

}

async function parallelAsyncOperations() {

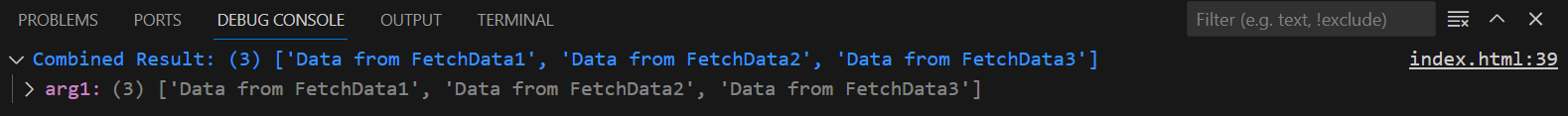
  const results = await Promise.all([fetchData1(), fetchData2(), fetchData3()]);

  console.log("Combined Result:", results);

}

 parallelAsyncOperations();

**Output:**

****

**Task-7**

**Aim:** Creating Modules

● Create a module mathOperations that exports functions for basic mathematical

operations (addition, subtraction, multiplication, division).

● Create another module stringOperations that exports functions for string manipulation

(concatenation, uppercase, lowercase).

● Import both modules in a main script and use functions from both modules to perform a

variety of operations.

**Description:**

Creating modules in JavaScript involves organizing code into separate files, each containing related functionality, variables, or classes. Modules help improve code maintainability, readability, and reusability. In JavaScript, there are different ways to create modules, and one common approach is using the ES6 module syntax.

**Source Code:**

//main.js

import \* as mathOps from "./mathOperations.js";

import \* as stringOps from "./stringOperations.js";

// Math Operations

const resultAdd = mathOps.add(5, 3);

const resultSubtract = mathOps.subtract(10, 4);

const resultMultiply = mathOps.multiply(2, 6);

const resultDivide = mathOps.divide(8, 2);

console.log("Math Operations:");

console.log("Addition:", resultAdd);

console.log("Subtraction:", resultSubtract);

console.log("Multiplication:", resultMultiply);

console.log("Division:", resultDivide);

// String Operations

const resultConcatenate = stringOps.concatenate("Hello", "World");

const resultUpperCase = stringOps.toUpperCase("hello");

const resultLowerCase = stringOps.toLowerCase("WORLD");

console.log("\nString Operations:");

console.log("Concatenation:", resultConcatenate);

console.log("Uppercase:", resultUpperCase);

console.log("Lowercase:", resultLowerCase);

//mathOperations.js

export function add(a, b) {

  return a + b;

}

export function subtract(a, b) {

  return a - b;

}

export function multiply(a, b) {

  return a \* b;

}

export function divide(a, b) {

  return a / b;

}

//stringOperations.js

export function concatenate(str1, str2) {

  return str1 + str2;

}

export function toUpperCase(str) {

  return str.toUpperCase();

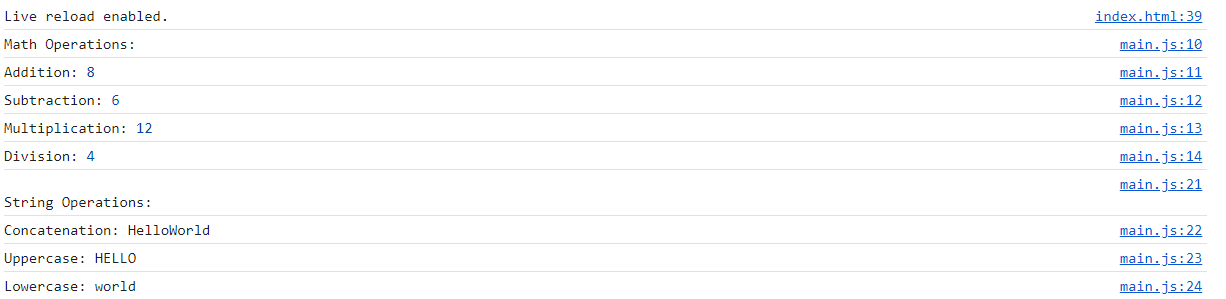
}

export function toLowerCase(str) {

  return str.toLowerCase();

}

**Output:**

****

**Task-8**

**Aim:** Asynchronous Module Loading

● Create a module asyncModule that exports an asynchronous function which simulates

fetching data with a 3-second delay.

● In the main script, use dynamic import to load the asyncModule asynchronously.

● Call the asynchronous function from the imported module and log the result.

**Description:**

Asynchronous Module Loading (AML) refers to the ability to load modules dynamically, on-demand, and asynchronously in a JavaScript application. This concept is closely related to the idea of lazy loading, where modules are only loaded when they are actually needed, rather than at the beginning when the application starts.

**Source Code:**

  //main.js

(async () => {

  const asyncModule = await import("./asyncModule.js");

  const result = await asyncModule.fetchDataAsync();

  console.log("Async Module Result:", result);

})();

//asyncModule.js

export async function fetchDataAsync() {

    return new Promise((resolve) => {

      setTimeout(() => {

        resolve("Data from Async Module");

      }, 3000);

    });

  }

**Output:**

****

**Task-9**

**Aim:** Default Export

● Create a module defaultExportModule that exports a default function to calculate the

square of a number.

● Import the module in a main script and use the default exported function to calculate the

square of a given number.

**Description:**

Default export in JavaScript allows a module to specify a single "default" export that can be imported without specifying a name. It provides a convenient way to export a primary entity or functionality from a module, and when importing, you can choose to give it any name you prefer.

**Source Code:**

//main.js

import square from "./defaultExportModule.js";

const numberToSquare = 5;

const result = square(numberToSquare);

console.log(`Square of ${numberToSquare} is:`, result);

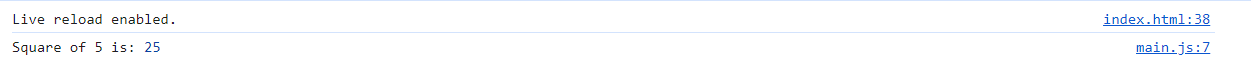
//defultExportModule.js

export default function square(number) {

    return number \* number;

  }

**Output:**

****

**Learning Outcome:**

Overall, the provided code examples cover various important aspects of web development, including HTML structure, Promises, Chaining Promises, Promise.all, Async Function, Sequential Async Operations, Parallel Async Operations, Creating Modules, Asynchronous Module Loading and Default Export JavaScript.

**Practical-5**

**Task-1**

**Aim:** custom Module

Create a Node.js module for basic arithmetic operations and demonstrate its utilization in a main script.

**Description:**

a custom module is a separate JavaScript file that encapsulates a set of related functionalities. The goal is to organize and modularize your code, making it easier to manage, maintain, and reuse.

Here are the key components and concepts related to custom modules in Node.js:

1) Module Definition 2)Module Export 3)Module import 4)Path to Modules 5)Encapsulation 6) Reusability

**Source Code:**

const mathOperations = require('./mathOperations.js');

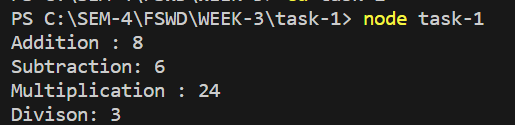
console.log(`Addition : ${mathOperations.add(5, 3)}`);

console.log(`Subtraction: ${mathOperations.subtract(8, 2)}`);

console.log(`Multiplication : ${mathOperations.multiply(4, 6)}`);

console.log(`Divison: ${mathOperations.divide(9, 3)}`);

**Output:**



**Task-2**

**Aim:** fs Module (File System)

Create a program that reads a directory and lists all files within it, displaying their names, sizes, and file types.

Implement functionalities to copy a file from one location to another, rename a file, and delete a file.

Provide an option to create a new directory and move specific files into it.

**Description:**

The fs module is not a standard module in Node.js, but it's commonly used as an abbreviation for the "file system" module, which is an integral part of Node.js core modules. The official name of the module is fs, which stands for "file system." It provides an API for interacting with the file system in a Node.js environment, allowing you to perform various file operations such as reading, writing, deleting, and manipulating files and directories.

**Source Code:**

const fs = require('fs');

const path = require('path');

// Reading a directory

const directoryPath = './Example';

fs.readdir(directoryPath, (err, files) => {

  if (err) {

    console.error('Error reading directory:', err);

    return;

  }

  // Displaying file details

  files.forEach((file) => {

    const filePath = path.join(directoryPath, file);

    const stats = fs.statSync(filePath);

    console.log(`${file} - Size: ${stats.size} bytes, Type: ${stats.isFile() ? 'File' : 'Directory'}`);

  });

  // Copying a file

  const sourceFile = path.join(directoryPath, 'example.txt');

  const destinationFile = path.join(directoryPath, 'example-copy.txt');

  fs.copyFileSync(sourceFile, destinationFile);

  console.log('File copied successfully.');

  // Renaming a file

  const oldFileName = path.join(directoryPath, 'example-copy.txt');

  const newFileName = path.join(directoryPath, 'renamed-file.txt');

  fs.renameSync(oldFileName, newFileName);

  console.log('File renamed successfully.');

  // Deleting a file

  const fileToDelete = path.join(directoryPath, 'renamed-file.txt');

  fs.unlinkSync(fileToDelete);

  console.log('File deleted successfully.');

  // Creating a new directory and moving files into it

  const newDirectoryName = path.join(directoryPath, 'new-directory');

  fs.mkdirSync(newDirectoryName);

   const filesToMove = ['index.html', 'style.css'];

   filesToMove.forEach((file) => {

     const sourcePath = path.join(directoryPath, file);

     const destinationPath = path.join(newDirectoryName, file);

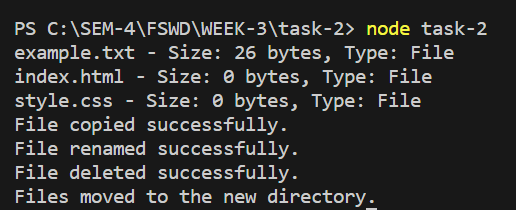
     fs.renameSync(sourcePath, destinationPath);

   });

   console.log('Files moved to the new directory.');

});

**Output:**



**Task-3**

**Aim:** **path Module (Path Handling)**

Design a tool that accepts a file path as input and extracts information like the file extension, filename, and directory path.

Normalize a given path and resolve a path by joining different segments.

Validate whether a path exists and whether it points to a file or a directory.

**Description:**

The path module is a core module in Node.js that provides utilities for working with file and directory paths. It helps in constructing, normalizing, and resolving file and directory paths in a platform-independent manner. This module is particularly useful for dealing with file paths in a way that is compatible across different operating systems, as the path formats can vary (e.g., backslashes in Windows vs. forward slashes in Unix-like systems).

**Source Code:**

const path = require('path')

const fs = require('fs')

const filepath = 'C:/SEM-4/FSWD/WEEK-3/TASK-2/Example/example.txt';

console.log('File Extension:', path.extname(filepath));

console.log('File Name:', path.basename(filepath));

console.log('Directory Path:', path.dirname(filepath));

const normalizedPath = path.normalize(filepath);

console.log('Normalized Path:', normalizedPath);

const resolvedPath = path.resolve(\_\_dirname, 'files', 'example.txt');

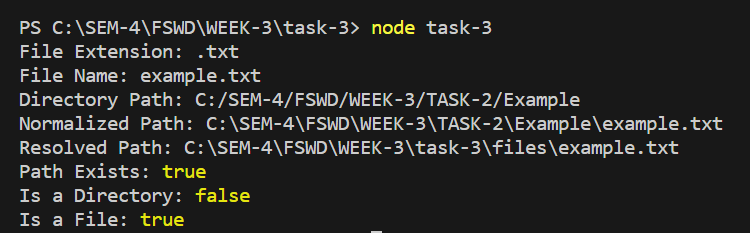
console.log('Resolved Path:', resolvedPath);

console.log('Path Exists:', fs.existsSync(filepath));

 console.log('Is a Directory:', fs.statSync(filepath).isDirectory());

 console.log('Is a File:', fs.statSync(filepath).isFile());

**Output:**

****

**Task-4**

**Aim:** **url Module (URL Handling)**

Create a program that parses a given URL into its components such as protocol, host, pathname, and query parameters.

Build a URL by combining its components (protocol, hostname, pathname, query) into a valid URL string.

Validate URLs and check if they are well-formed.

**Description:**

The url module in Node.js provides utilities for URL parsing and formatting. It is used to work with URLs (Uniform Resource Locators), allowing you to parse URL strings, construct URLs from components, and manipulate URL components.

**Source Code:**

const url = require('url')

// Parsing a URL

const urlstring = "https://mail.google.com/mail/u/0/?tab=rm&ogbl#inbox";

const parsedUrl = url.parse(urlstring, true);

console.log('Protocol:', parsedUrl.protocol);

console.log('Host:', parsedUrl.host);

console.log('Pathname:', parsedUrl.pathname);

console.log('Query Parameters:', parsedUrl.query);

// Building a URL

const builtUrl = url.format({

    protocol: 'https',

    host: 'www.example.com',

    pathname: '/newpath',

    query: { param1: 'value1', param2: 'value2' },

  });

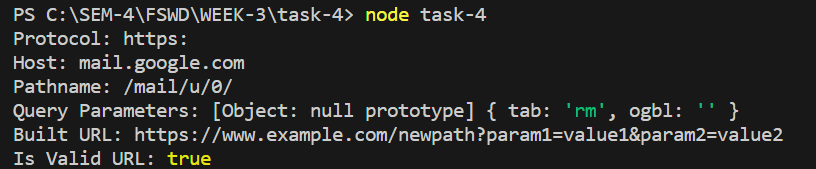
  console.log('Built URL:', builtUrl);

  // Validating a URL

const isValidUrl = url.parse(urlstring).protocol !== null;

console.log('Is Valid URL:', isValidUrl);

**Output:**

****

**Task-5**

**Aim:** **util Module (Utilities)**

Develop a script that uses the util module to create custom error objects with defined error codes and messages.

Utilize the util.inspect() function to log objects and nested structures for debugging purposes.

Demonstrate error handling techniques using the util.promisify() method for callback-based functions.

**Description:**

The util module in Node.js provides a set of utility functions that are commonly used in various applications. It includes functions that simplify common programming tasks, such as formatting and inspecting objects, working with asynchronous code, and more.

**Source Code:**

const util = require('util');

const fs = require('fs');

// Custom error objects

class CustomError extends Error {

  constructor(code, message) {

    super(message);

    this.code = code;

  }

}

  const myError = new CustomError(404,"Page Not Found");

  console.log('Custom Error:', myError);

// Logging objects with util.inspect()

const objToInspect = { name: 'John', age: 30, address: { city: 'Example City', zip: '12345' } };

console.log('Inspected Object:', util.inspect(objToInspect, { depth: null }));

// Error handling with util.promisify()

const readFileAsync = util.promisify(fs.readFile);

async function readAndHandleFile(filePath) {

  try {

    const content = await readFileAsync(filePath, 'utf8');

    console.log('File Content:', content);

  } catch (error) {

    if (error.code === 'ENOENT') {

      console.error('File not found:', error.message);

    } else {

      console.error('Error reading file:', error);

    }

  }

}

// Example usage

readAndHandleFile('./example.txt');

**Output:**

A computer screen shot of a program

Description automatically generated

**Task-6**

**Aim:** **buffer Module (Binary Data)**

Build a program that reads a text file and converts its content into a buffer.

Manipulate the buffer (e.g., split, slice, concatenate) and convert it back to text.

Demonstrate encoding and decoding functionalities (e.g., UTF-8, Base64) with buffers. **Description:**

The buffer module in Node.js provides a way to work with binary data directly, without having to first convert it to a string. It is a part of the core modules in Node.js and is commonly used in scenarios where handling binary data, such as reading from or writing to files, dealing with network protocols, or manipulating raw binary data, is necessary.

**Source Code:**

const fs = require('fs');

// Reading a text file and converting its content to a buffer

const textContent = 'Hello, this is a text file.';

const filePath = './textFile.txt';

fs.writeFileSync(filePath, textContent);

const buffer = fs.readFileSync(filePath);

console.log("Buffer: ",buffer)

console.log('Buffer in utf8:', buffer.toString('utf8'));

// Manipulating the buffer

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

console.log('Sliced Buffer:', slicedBuffer.toString());

const concatenatedBuffer = Buffer.concat([buffer, Buffer.from(' Appended text.')]);

console.log('Concatenated Buffer:', concatenatedBuffer.toString());

// Encoding and Decoding with buffers

const utf8Encoded = buffer.toString('utf8');

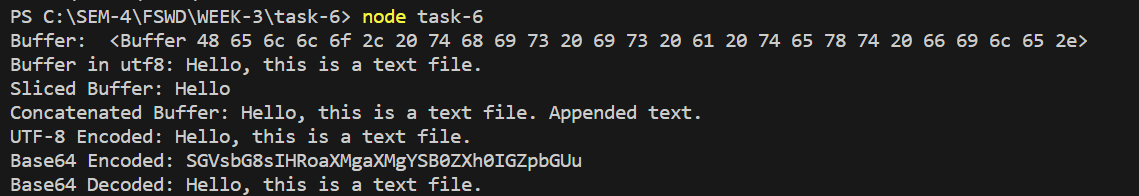
console.log('UTF-8 Encoded:', utf8Encoded);

const base64Encoded = buffer.toString('base64');

console.log('Base64 Encoded:', base64Encoded);

const base64Decoded = Buffer.from(base64Encoded, 'base64');

console.log('Base64 Decoded:', base64Decoded.toString('utf8'));

**Output:**

**Task-7**

**Aim: os Module (Operating System)**

Create a Node.js script that fetches and displays system information such as CPU architecture, total memory, operating system platform, and uptime.

Implement a function that monitors system resources (CPU, memory) and logs their usage periodically during script execution.

**Description:**

The os module in Node.js provides a set of utility functions to interact with the operating system. It allows you to access information about the operating system, such as platform, architecture, and network interfaces. The module is part of the core Node.js modules and is available for use without the need for additional installations.

**Source Code:**

const os = require('os');

console.log(`Hostname : ${os.hostname}`)

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

console.log(`Total Memory(GB) : ${os.totalmem/(1024\*1024\*1024)}`)

console.log(`Operating System Platform: ${os.platform}`)

console.log(`Uptime(second): ${os.uptime}`)

console.log(`Operating System type:${os.type}`)

//console.log(`CPU : ${os.cpus}`)

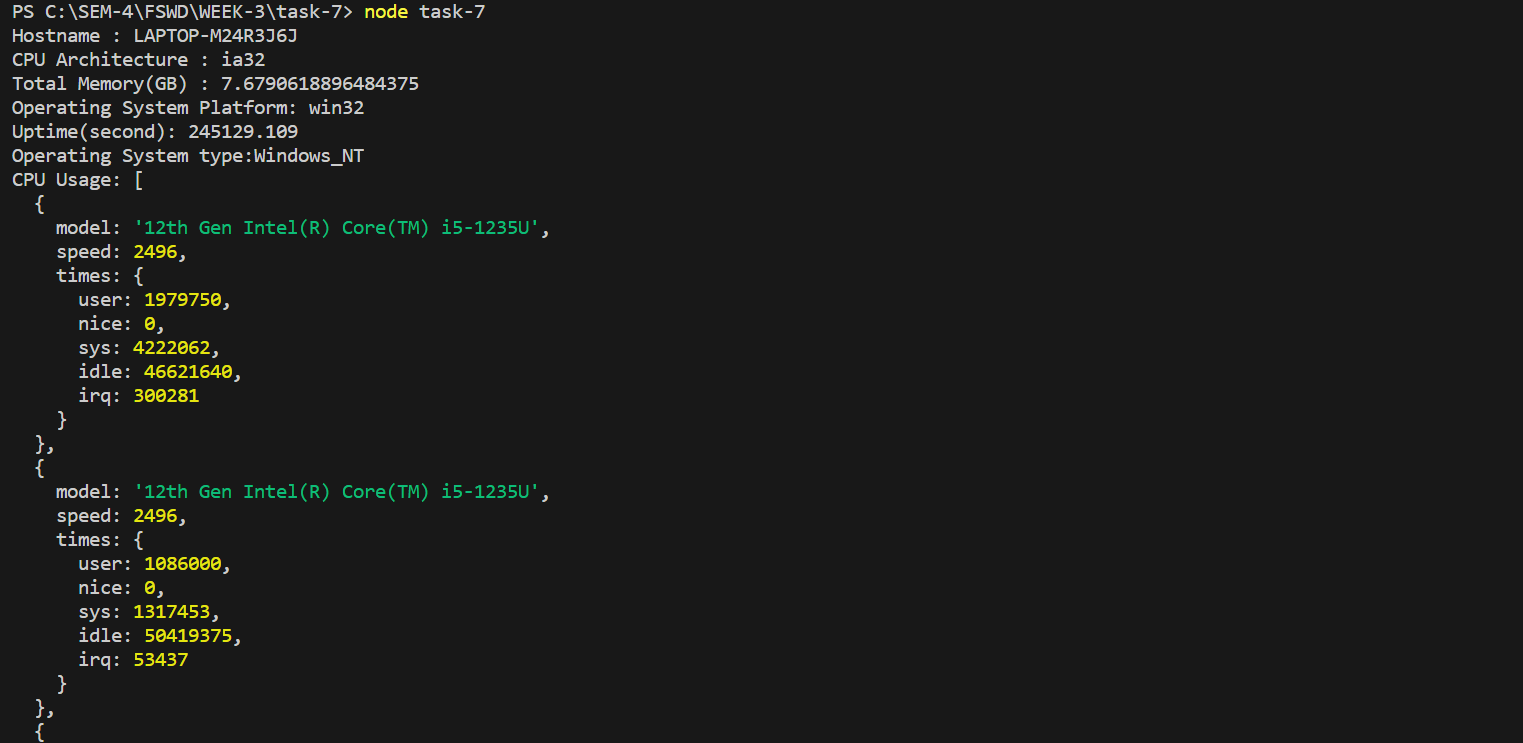
setInterval(() => {

    console.log('CPU Usage:', os.cpus());

    console.log('Free Memory (bytes):', os.freemem());

  }, 5000);

**Output:**

****

**A computer screen shot of a program

Description automatically generated**

**A screenshot of a computer program

Description automatically generated**

**Task-8**

**Aim:** **HTTP Module (HTTP Server)**

Set up an HTTP server that listens on a specified port 8080 and handles GET and POST requests.

Create endpoints for handling GET requests to retrieve data and POST requests to update data.

Implement error handling for different status codes and invalid routes.

**Description:**

The http module in Node.js is a core module that allows you to create and run HTTP servers, handle HTTP requests, and send HTTP responses. It provides the fundamental building blocks for creating web applications and handling network communication over the HTTP protocol.

**Source Code:**

const http = require('http');

const url = require('url');

const querystring = require('querystring');

// Sample data

let data = {

  name: 'John Doe',

  age: 25,

};

// Creating an HTTP server

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

  const parsedUrl = url.parse(req.url, true);

  const pathname = parsedUrl.pathname;

  if (req.method === 'GET') {

    if (pathname === '/getData') {

      res.writeHead(200, { 'Content-Type': 'application/json' });

      res.end(JSON.stringify(data));

    } else {

      res.writeHead(404, { 'Content-Type': 'text/plain' });

      res.end('Not Found');

    }

  } else if (req.method === 'POST') {

    if (pathname === '/updateData') {

      let body = '';

      req.on('data', (chunk) => {

        body += chunk.toString();

      });

      req.on('end', () => {

        const postData = querystring.parse(body);

        data = { ...data, ...postData };

        res.writeHead(200, { 'Content-Type': 'application/json' });

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

      });

    } else {

      res.writeHead(404, { 'Content-Type': 'text/plain' });

      res.end('Not Found');

    }

  } else {

    res.writeHead(405, { 'Content-Type': 'text/plain' });

    res.end('Method Not Allowed');

  }

});

// Listening on port 8080

const PORT = 8080;

server.listen(PORT, () => {

  console.log(`Server listening on port ${PORT}`);

});

**Output:**

**A screenshot of a computer program

Description automatically generated**

****

**Learning Outcome:**

Overall, the provided code examples cover various important aspects of web development, including custom Module, fs Module, path Module (Path Handling), url Module (URL Handling), util Module (Utilities), buffer Module (Binary Data), os Module (Operating System)and HTTP Module (HTTP Server)

**Practical-6**

**Task**

**Aim:** Event Emitter in Node js

Build a real-time file monitoring system using the fs module to watch a specific directory and emit custom events through an Event Emitter when new files are created, existing files are modified, or files are deleted.

**Description:**

In Node.js, the Event Emitter is a core module that facilitates communication between objects by allowing them to emit and listen for custom events. It plays a crucial role in implementing an asynchronous, event-driven paradigm. Objects that extend the Event Emitter class can emit named events, and other objects can register listeners to respond to those events, enabling effective decoupling and modularization in Node.js applications.

**Source Code:**

const fs = require('fs');

const path = require('path');

const EventEmitter = require('events');

class FileMonitor extends EventEmitter {

  constructor(directory) {

    super();this.directory = directory;

  }

  startMonitoring() {

    fs.watch(this.directory, (eventType, filename) => {

      const filePath = path.join(this.directory, filename);

      if (eventType === 'rename') {

        if (fs.existsSync(filePath)) {

          this.emit('fileCreated', filePath);

        } else {

          this.emit('fileDeleted', filePath);

        }

      } else if (eventType === 'change') {

        this.emit('fileModified', filePath);

      }

    });

    this.on('error', (err) => {

      console.error(`Error in FileMonitor: ${err.message}`);

    });

  }

}

// Usage example

const directoryToMonitor = 'C:\\SEM-4\\FSWD\\WEEK-4';  // Double backslashes for Windows

const fileMonitor = new FileMonitor(directoryToMonitor);

fileMonitor.on('fileCreated', (filePath) => {

  console.log(`File created: ${filePath}`);

});

fileMonitor.on('fileModified', (filePath) => {

  console.log(`File modified: ${filePath}`);

});

fileMonitor.on('fileDeleted', (filePath) => {

  console.log(`File deleted: ${filePath}`);

});

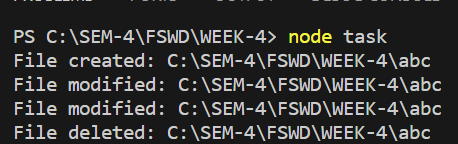
fileMonitor.on('error', (err) => {

  console.error(`Error in FileMonitor: ${err.message}`);

});

fileMonitor.startMonitoring();

**Output:**

****

**Learning Outcome:**

In this task we learn about event emitter.

**Practical-7**

**Task**

**Aim:** To implement CRUD operations for managing student details using MongoDB, an Express-based REST API will be developed, leveraging the Mongoose package for database interaction. Test REST-API using POSTMAN Tool. Send Email to admin email, Upon successful deletion of a record via the REST API, an email notification will be dispatched using the nodemailer package.

**Description:**  
A REST API for managing student details using MongoDB and Express, with Mongoose for database interaction, will be developed. CRUD operations can be tested using the POSTMAN tool. Additionally, upon successful deletion of a record via the API, an email notification will be sent to the admin email using the nodemailer package, enhancing the system's functionality.

**Source Code:**

**Exdemo.js**

const express = require('express')

const main = require('./Email.js');

const mongoose = require('mongoose');

const app = express()

const port=3000

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

app.use(express.json())

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

const UserModel = require('./models/Users')

mongoose.connect('mongodb://localhost:27017/mydb')

.then(()=>{console.log("Database connected successfully!");})

.catch((err)=>{console.log("Database does not connected!");})

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

{

   var mydata =

   {

      User\_name:'Trushang',

      User\_gender:'Male'

   }

   UserModel.create(mydata)

   .then(()=>{console.log("Record added")})

   .catch((err)=>{console.log("NO record Inserted"+err)})

   res.send("record added")

})

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

{

   console.log(req.body)

   var mydata = req.body;

   UserModel.create(mydata)

   .then(()=>{console.log("Record Added")})

   .catch((err)=>{console.log("Record not inserted"+err)})

   res.send(JSON.stringify("RECord added"))

})

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

      console.log(req.body);

      UserModel.find()

      .then(data=>res.json(data))

      .catch(err=>console.log(err))

})

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

    res.send('Hello World!')

})

 app.get('/home',(req,res)=>

 {

    res.send(\_\_dirname+'/home.html')

 })

 app.get('/display',(req,res)=>

 {

   UserModel.find()

   .then(data=>res.json(data))

   .catch(err=>console.log(err))

 });

 main();

 app.get('/update', (req, res) => {

   UserModel.updateOne({ User\_name: 'Trushang Patel' }, { User\_name: 'Trushang' })

       .then(() => { console.log("Record Updated") })

       .catch((err) => { console.log("Record not updated" + err) })

   res.send("Record Updated");

});

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

   UserModel.deleteOne({ User\_name: 'Trushang' })

       .then(() => { console.log("Record Deleted") })

       .catch((err) => { console.log("Record not Deleted" + err) })

   res.send("Record Deleted");

});

 app.listen(port,()=>{

    console.log(`Example app listening on port ${port}`)

 });

**Email.js:**

const nodemailer = require("nodemailer");

// Create a nodemailer transporter

const transporter = nodemailer.createTransport({

host: "smtp.gmail.com",

port: 465,

secure: true,

auth: {

user: "trushangpatel282@gmail.com",

pass: "aqfzngcdkrimhzbu",

},

});

// Define the email content

const emailOptions = {

from: 'trushangpatel282@gmail.com>', // sender address

to: "22it116@charusat.edu.in", // list of receivers

subject: "Hello ✔", // Subject line

text: "Hello world?", // plain text body

html: "<b>Hello world?</b>", // html body

};

// async..await is not allowed in global scope, must use a wrapper

const main=async ()=> {

try {

// Send mail with the defined transport object

const info = await transporter.sendMail(emailOptions);

console.log("Message sent: %s", info.messageId);

// Message sent: <b658f8ca-6296-ccf4-8306-87d57a0b4321@example.com

// NOTE: You can check your email delivery status and preview at

//https://forwardemail.net/my-account/emails

} catch (error) {

console.error("Error sending email:", error);

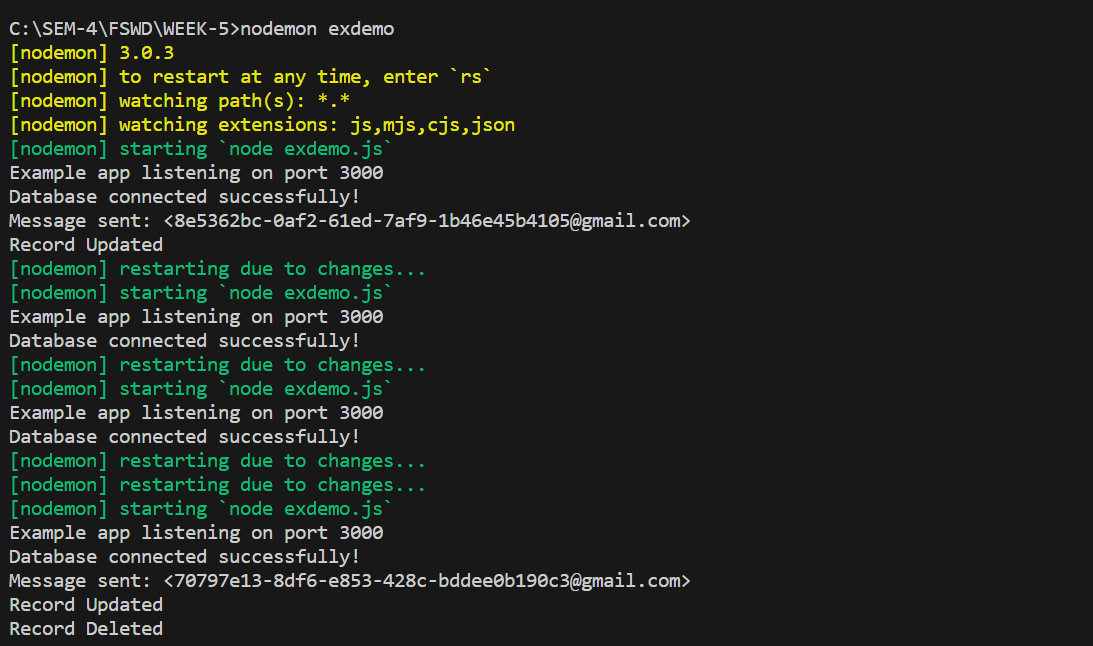
}

}

// Call the main function

module.exports = main;

**Output:**

****

**A screenshot of a computer

Description automatically generated**

**A screenshot of a computer

Description automatically generated**

**A screenshot of a computer

Description automatically generated**

**A screenshot of a computer

Description automatically generated**

**A screenshot of a computer

Description automatically generated**

**Learning Outcome:**

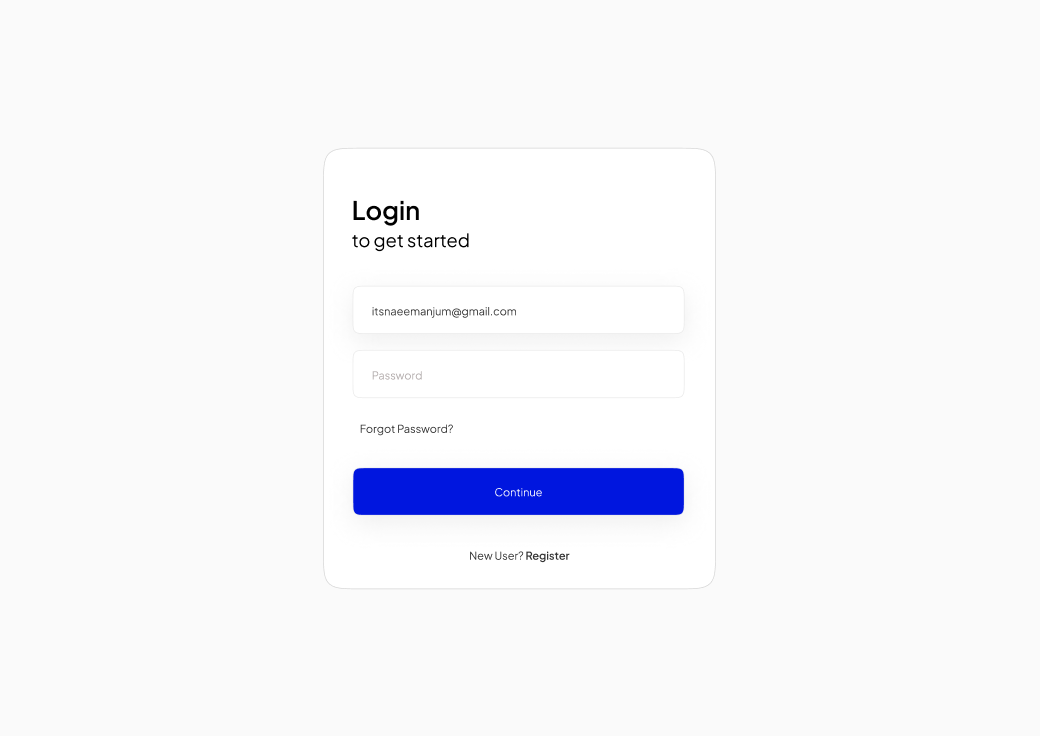
In this task we learn about how to connect MongoDB with node js and how to send email using node js.

**Practical-8**

**Task**

**Aim:** Create only frontend using React for below Images

**Login Page:**

****

**Registration Page:**

**A screen shot of a login form

Description automatically generated**

**Description:**  
React is a JavaScript library for building user interfaces, providing a declarative and efficient way to create interactive and dynamic UI components. Developed and maintained by Facebook, React simplifies the process of building scalable and reusable UI components, facilitating the creation of robust and efficient web applications.

**Source Code:**

**Login.js**

import React, { useState } from 'react';

import { useNavigate } from 'react-router-dom';

import './AuthForm.css'; // Import your CSS file

const Login = () => {

  const [formData, setFormData] = useState({

    email: '',

    password: '',

  });

  const [loginStatus, setLoginStatus] = useState(null);

  const navigate = useNavigate();

  const handleChange = (e) => {

    setFormData({ ...formData, [e.target.name]: e.target.value });

  };

  const handleSubmit = (e) => {

    e.preventDefault();

    // Simulate login success for demonstration purposes

    // In a real application, you would perform authentication with a backend

    console.log('Login Data:', formData);

    // Simulate a successful login for demonstration purposes

    // In a real application, you would check user credentials with a backend

    const isLoginSuccessful = true;

    if (isLoginSuccessful) {

      setLoginStatus('success');

      setTimeout(() => {

        navigate('/dashboard');

      }, 2000);

    }

  };

  const handleForgotPassword = () => {

    // Handle the "Forgot Password" functionality

    alert('Forgot Password: Implement your password recovery logic.');

  };

  const handleCreateAccount = () => {

    // Redirect to the registration page

    navigate('/register');

  };

  return (

    <div className="auth-container">

      <h2>Login</h2>

      {loginStatus === 'success' && (

        <p className="success-message">Login successful! Redirecting...</p>

      )}

      <form onSubmit={handleSubmit}>

        <label>

          Email:

          <input type="email" name="email" value={formData.email} onChange={handleChange} />

        </label>

        <label>

          Password:

          <input type="password" name="password" value={formData.password} onChange={handleChange} />

        </label>

        <div className="form-actions">

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

          <span className="forgot-password" onClick={handleForgotPassword}>

            Forgot Password?

          </span>

        </div>

      </form>

      <p className="create-account-link">

        Don't have an account? <span onClick={handleCreateAccount}>Create an Account</span>

      </p>

    </div>

  );

};

export default Login;

**Register.js**

import React, { useState } from 'react';

import { useNavigate } from 'react-router-dom';

import './AuthForm.css'; // Import your CSS file

const Register = () => {

  const [formData, setFormData] = useState({

    username: '',

    email: '',

    password: '',

    confirmPassword: '',

    agreeTerms: false,

  });

  const [registrationStatus, setRegistrationStatus] = useState(null);

  const navigate = useNavigate();

  const handleChange = (e) => {

    setFormData({ ...formData, [e.target.name]: e.target.type === 'checkbox' ? e.target.checked : e.target.value });

  };

  const handleSubmit = (e) => {

    e.preventDefault();

    if (!formData.agreeTerms) {

      alert("Please agree to the terms and conditions.");

      return;

    }

    console.log('Registration Data:', formData);

    const isRegistrationSuccessful = true;

    if (isRegistrationSuccessful) {

      setRegistrationStatus('success');

      setTimeout(() => {

        navigate('/login');

      }, 2000);

    }

  };

  return (

    <div className="auth-container">

      <h2>Register</h2>

      {registrationStatus === 'success' && (

        <p className="success-message">You have registered successfully!</p>

      )}

      <form onSubmit={handleSubmit}>

        <label>

          Username:

          <input type="text" name="username" value={formData.username} onChange={handleChange} />

        </label>

        <label>

          Email:

          <input type="email" name="email" value={formData.email} onChange={handleChange} />

        </label>

        <label>

          Password:

          <input type="password" name="password" value={formData.password} onChange={handleChange} />

        </label>

        <label>

          Confirm Password:

          <input

            type="password"

            name="confirmPassword"

            value={formData.confirmPassword}

            onChange={handleChange}

          />

        </label>

        <label className="checkbox-label">

          <input

            type="checkbox"

            name="agreeTerms"

            checked={formData.agreeTerms}

            onChange={handleChange}

          />

          {' '}

          I agree to the terms and conditions

        </label>

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

      </form>

      <p className="login-link">

        Already Registered? <span onClick={() => navigate('/login')}>Login</span>

      </p>

    </div>

  );

};

export default Register;

**Dashboard.css**

/\* src/Dashboard.css \*/

.dashboard-container {

    text-align: center;

    margin-top: 50px;

  }

  .success-message {

    font-size: 24px; /\* Increase the text size of the success message \*/

    color: green; /\* Change the text color if desired \*/

  }

**AuthForm.css**

body {

  margin: 0;

  font-family: 'Segoe UI', Tahoma, Geneva, Verdana, sans-serif;

  background-color: #f4f4f4;

}

.auth-container {

  max-width: 300px;

  margin: auto;

  padding: 20px;

  border: 1px solid #ccc;

  border-radius: 5px;

  box-shadow: 0 0 10px rgba(0, 0, 0, 0.1);

  margin-top: 50px;

  background-color: #fff;

}

form {

  display: flex;

  flex-direction: column;

  gap: 10px;

}

label {

  display: block;

  margin-bottom: 5px;

}

input {

  padding: 8px;

  font-size: 16px;

}

/\* Add the following styles for aligning email and password input fields \*/

label {

  display: flex;

  flex-direction: column;

}

input {

  margin-top: 5px;

}

button {

  padding: 10px;

  background-color: #007bff;

  color: #fff;

  border: none;

  cursor: pointer;

  font-size: 16px;

}

button:hover {

  background-color: #0056b3;

}

p {

  margin-top: 10px;

  font-size: 14px;

}

/\* src/AuthForm.css \*/

body {

  margin: 0;

  font-family: 'Segoe UI', Tahoma, Geneva, Verdana, sans-serif;

  background-color: #f4f4f4;

}

.auth-container {

  max-width: 300px;

  margin: auto;

  padding: 20px;

  border: 1px solid #ccc;

  border-radius: 5px;

  box-shadow: 0 0 10px rgba(0, 0, 0, 0.1);

  margin-top: 50px;

  background-color: #fff;

}

form {

  display: flex;

  flex-direction: column;

  gap: 10px;

}

label {

  display: flex;

  flex-direction: column;

}

input {

  padding: 8px;

  font-size: 16px;

}

/\* Add the following styles for the checkbox \*/

label.checkbox-label {

  display: flex;

  align-items: center;

}

.checkbox-label input {

  margin-top: 0;

  margin-right: 5px;

}

/\* src/AuthForm.css \*/

/\* ... (your existing styles) \*/

.login-link {

  margin-top: 10px;

  font-size: 14px;

  text-align: center;

}

.login-link span {

  color: #007bff;

  cursor: pointer;

}

/\* src/AuthForm.css \*/

/\* ... (your existing styles) \*/

.forgot-password {

  margin-top: 10px;

  font-size: 14px;

  color: #007bff;

  cursor: pointer;

}

.form-actions {

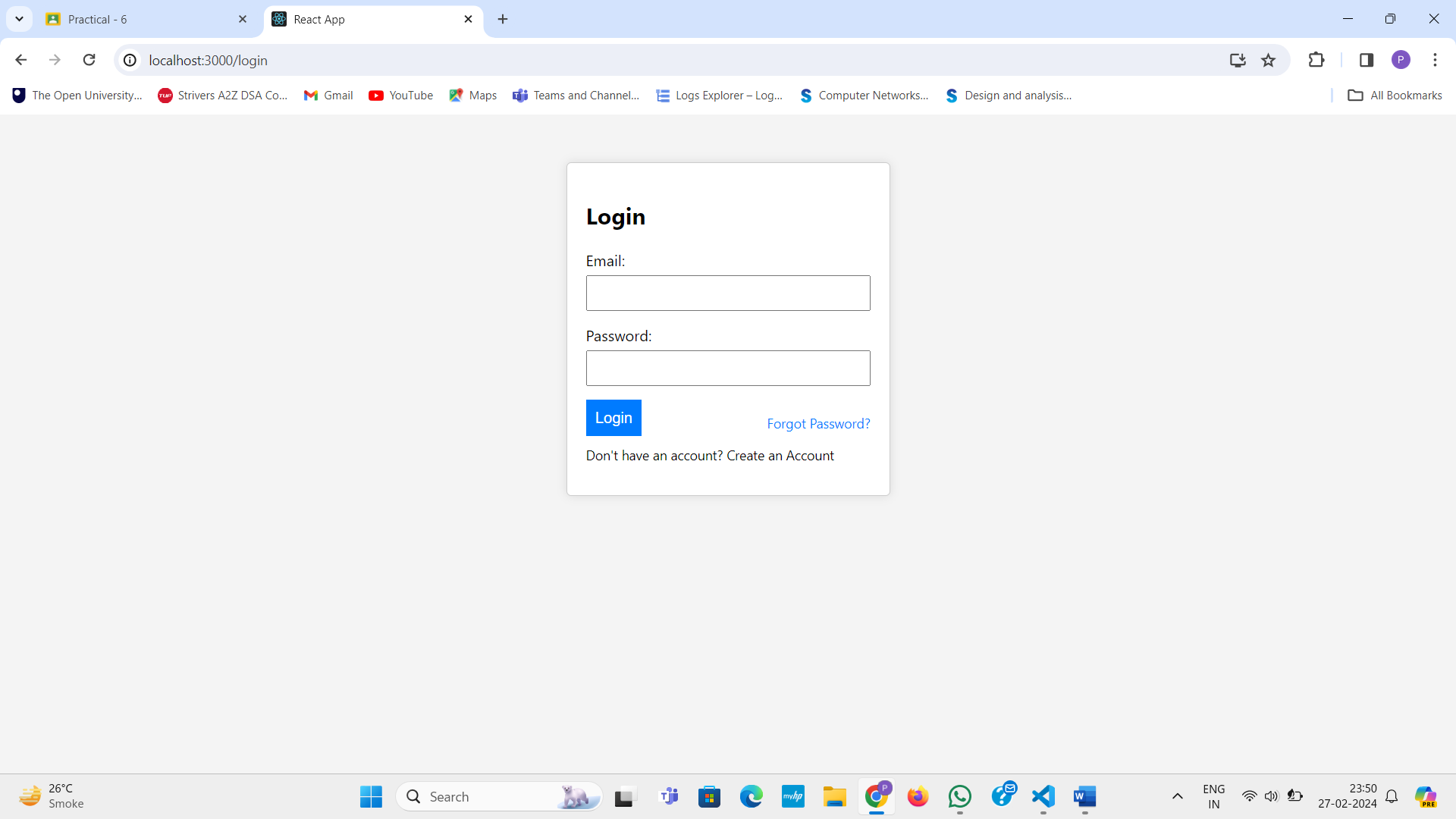
  display: flex;

  justify-content: space-between;

  align-items: center;

}

**Output:**

****

**A screenshot of a computer

Description automatically generated**

**A screenshot of a computer

Description automatically generated**

**A screenshot of a computer

Description automatically generated**

**Learning Outcome:**

In this task we learn about how React js.

**Practical-9**

**Aim:**

Create Following pages using react

1. Product List Page (PLP)

2. Product Detail Page (PDP)

**Source Code:**

**App.js**

import React ,{useState,useEffect}from "react";

import './App.css';

import ProductCard from './component/Card';

import Nav from './component/nav';

function App() {

  const [products, setProducts] = useState(null);

  useEffect(() => {

      fetch('https://fakestoreapi.com/products')

          .then(res => res.json())

          .then(json => setProducts(json));

  }, []);

  if (!products) {

      return <div>Loading...</div>;

  }

  return (

    <>

    <Nav/>

    <div className="card-container">

      {products.map((product) => (<ProductCard {...product}/>))}

      </div>

    </>

   );

}

export default App;

**card.js**

import React,{useState} from 'react';

import './styles.css';

function ProductCard({id, title, description, price, image}) {

    const [view, setView] = useState(false);

    function viewmore(){

        setView(!view);

    }

    const dec = {

         id : id,

         title : title,

         description:description,

         price : price

    }

    return (

        <div key={id} className='card'>

            <img src={image} alt={title} style={{ width: '100%' }} onClick={viewmore}/>

            {view && <Viewcont {...dec}/> }

        </div>

    );

}

const Viewcont = ({id, title, description, price})=>{

    return(

    <>

    <h1>{title}</h1>

    <p>{description}</p>

    <p>Price: {price}</p>

    </>

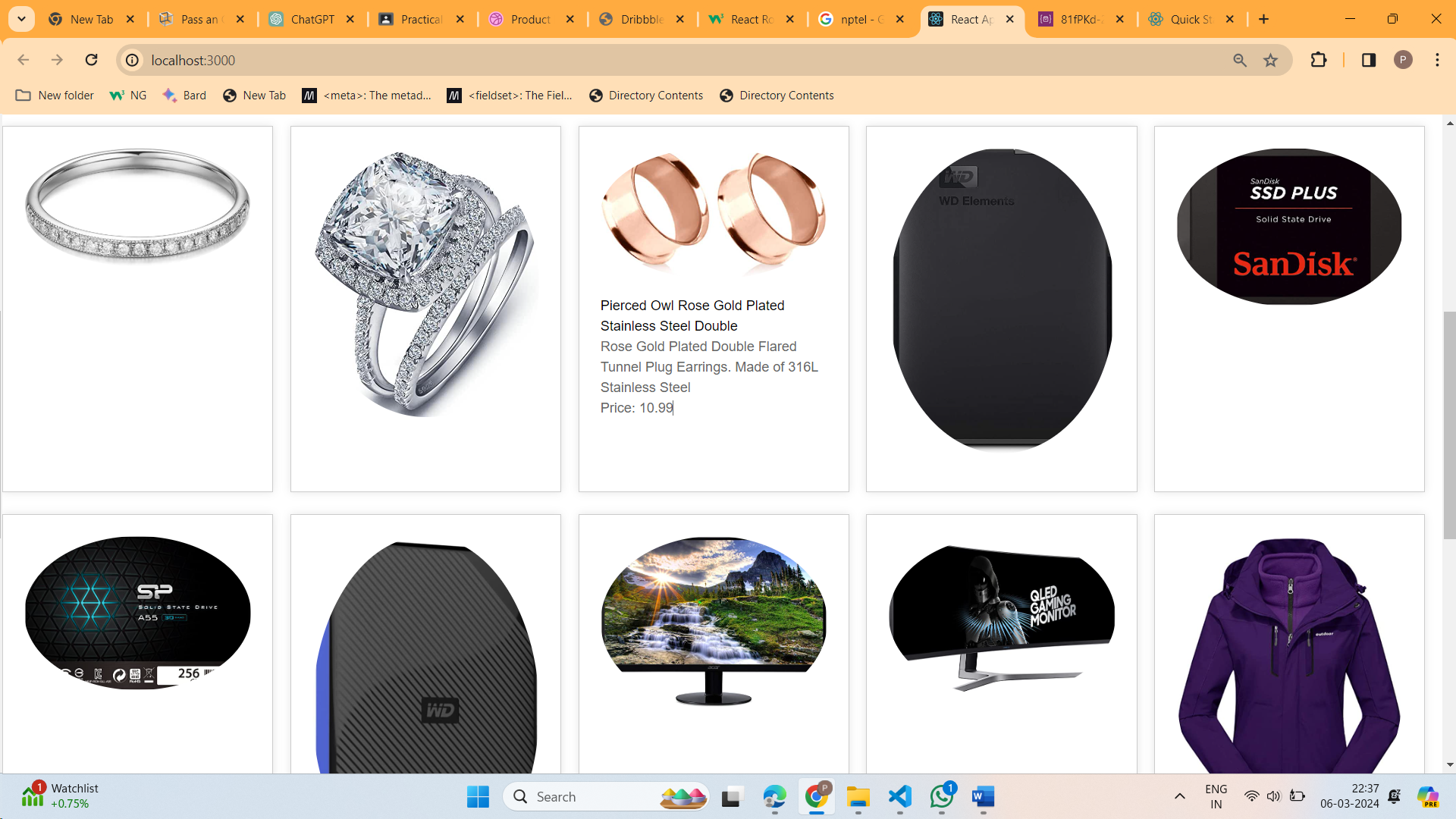
    );

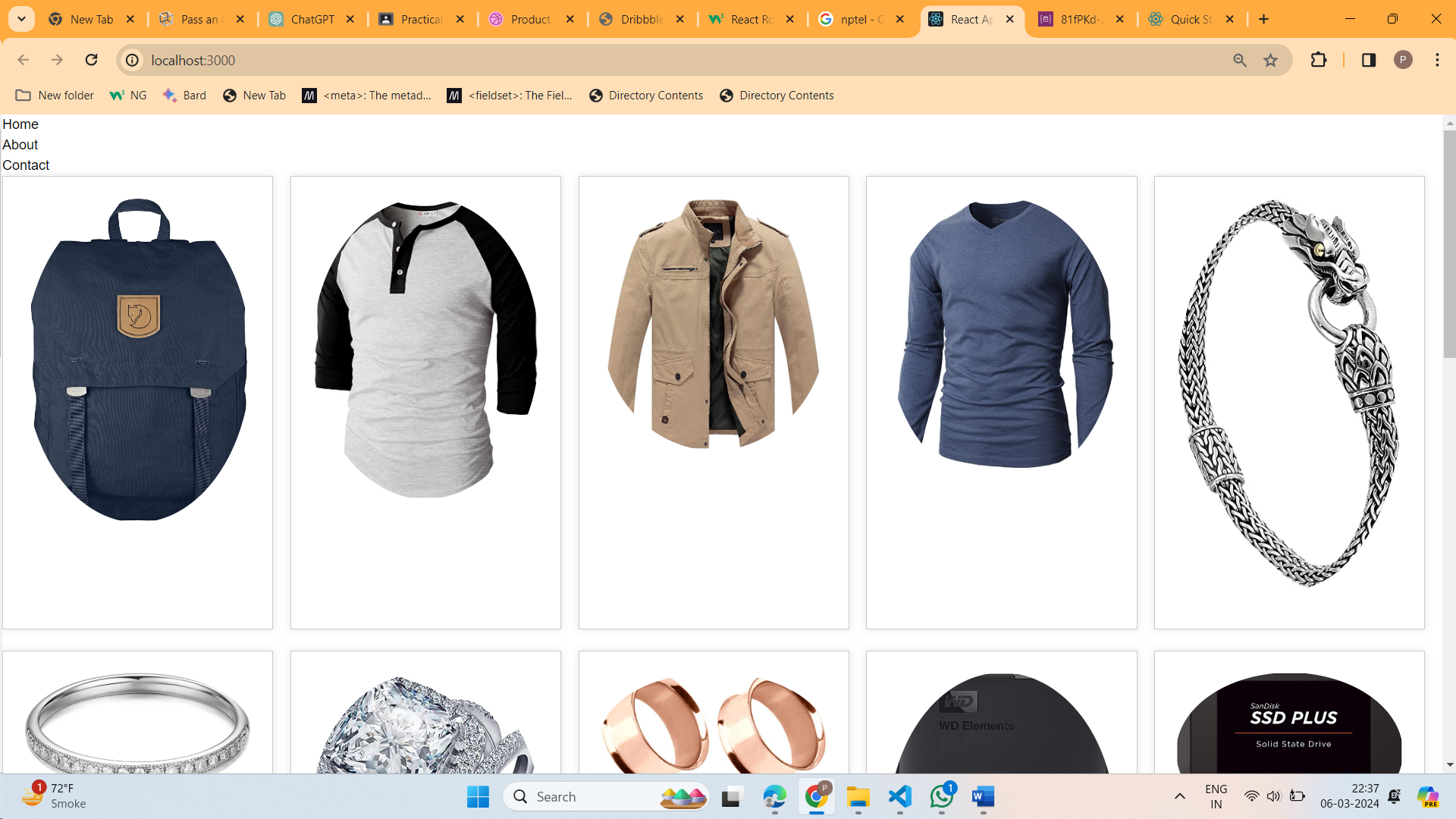
}

export default ProductCard;

**Output:**



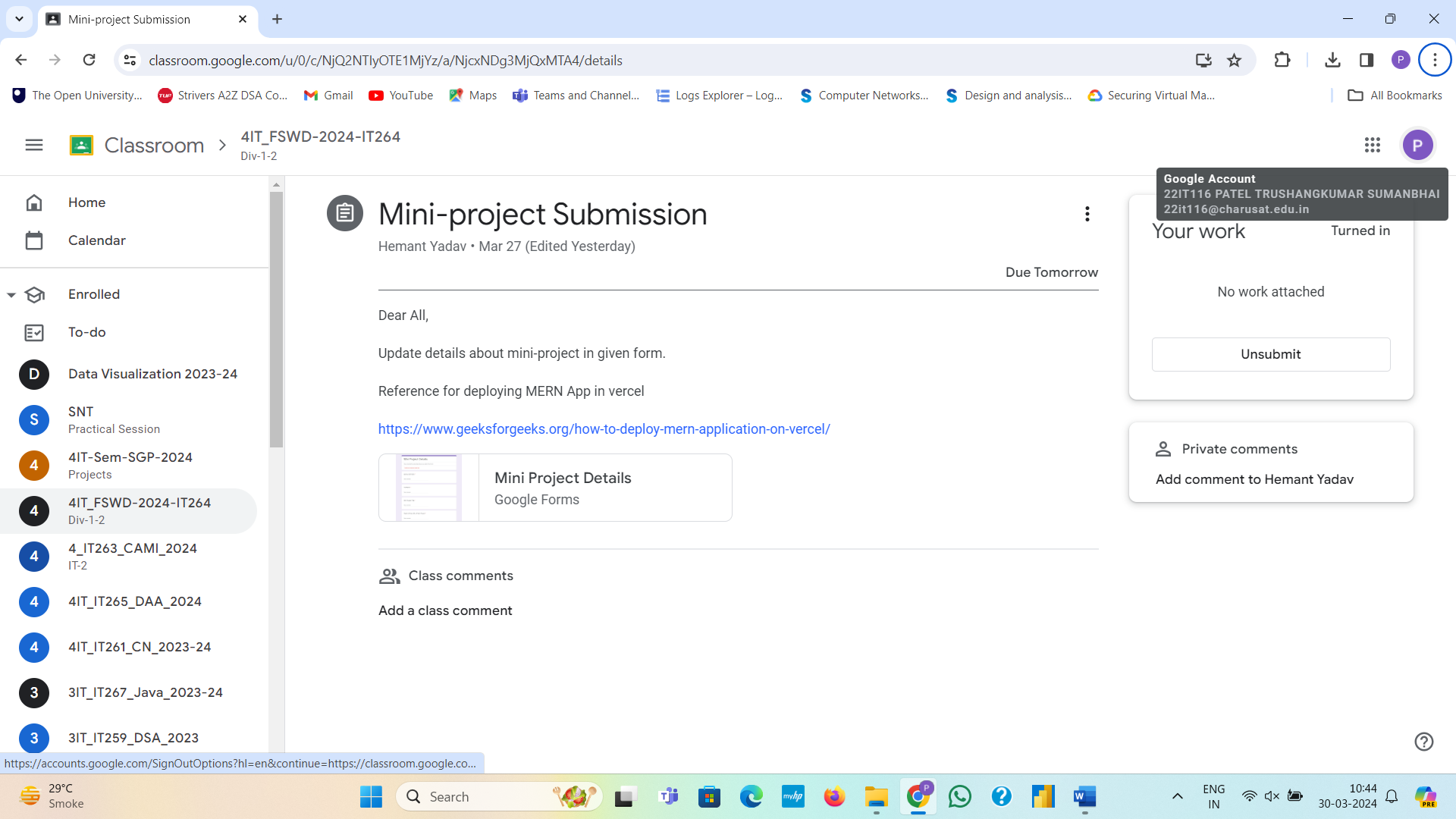




**Learning Outcome:**

Overall, the provided code explains how to make a Product List Page using React Js.

**MINI PROJECT:**

****