Name-Pratham Asnani
RollNo-63
Batch-T13

ASSIGNMENT 6

**AIM:**To understand and implement arrow functions, DOM manipulation, and CSS manipulation using
JavaScript.

**LO-4**

**THEORY:**
1. Arrow Functions:
Arrow functions are a shorter syntax for writing functions in JavaScript. They are anonymous and use the => syntax.
Arrow functions do not have their own this binding; they inherit this from the parent scope, making them useful for callbacks and methods within classes.
Syntax example:
javascript
const add = (a, b) => a + b;

2. DOM Manipulation:
The Document Object Model (DOM) represents the structure of a web page. It allows JavaScript to access and modify HTML and CSS dynamically.

Common methods for DOM manipulation include getElementById, querySelector, appendChild, removeChild, and innerHTML.

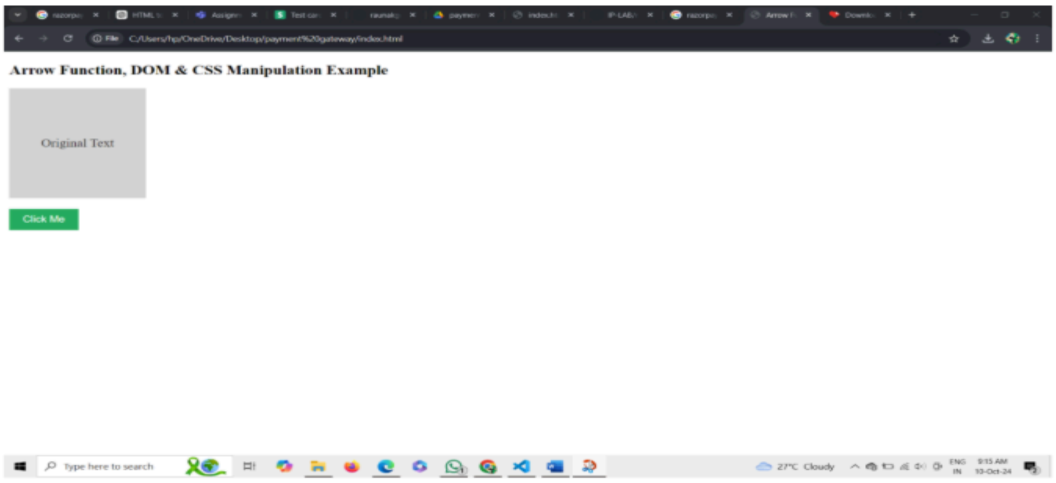
3. CSS Manipulation:
CSS styles can be manipulated via JavaScript to change the appearance of elements dynamically.
JavaScript provides ways to modify inline styles directly or to toggle CSS classes.
Example:
javascript
document.getElementById('myDiv').style.backgroundColor = 'blue';

PROGRAM:
This example program demonstrates the use of arrow functions, DOM manipulation, and CSS
manipulation to change the background color of a div element and update its text content on button click.

```
html
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF8">
```

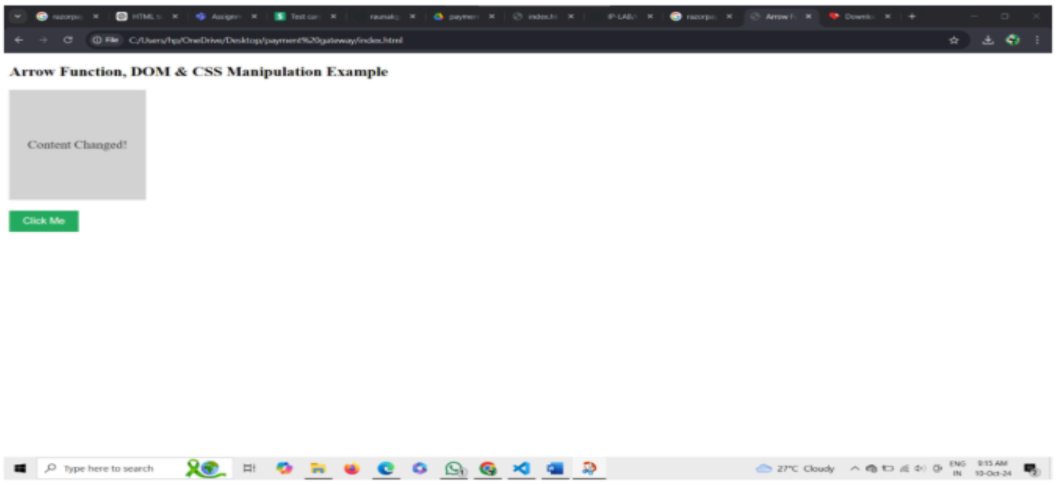
```
<meta name="viewport" content="width=devicewidth, initialscale=1.0">
<title>Arrow Functions, DOM & CSS Manipulation</title>
<style>
/ Initial CSS Styling /
contentBox {
width: 200px;
height: 200px;

backgroundcolor: lightgray;
display: flex;
justifycontent: center;
alignitems: center;
fontsize: 20px;
color: 333;
transition: backgroundcolor 0.3s ease;
margintop: 20px;
}
.highlight {
backgroundcolor: lightcoral;
color: white;
}
button {
padding: 10px 20px;
fontsize: 16px;
cursor: pointer;
margintop: 20px;
border: none;
backgroundcolor: 27ae60;
color: white;
}
button:hover {
backgroundcolor: 2ecc71;
}
</style>
</head>

<body>
<h2>Arrow Function, DOM & CSS Manipulation Example</h2>
<div id="contentBox">Original Text</div>
<button id="changeButton">Click Me</button>

<script>
// Arrow function for DOM and CSS Manipulation
const changeContentAndStyle = () => {
// DOM Manipulation: Change the text inside the div
const contentBox = document.getElementById('contentBox');
contentBox.innerHTML = "Content Changed!";
// CSS Manipulation: Toggle CSS class to change background color
contentBox.classList.toggle('highlight');
};

// Event listener with an arrow function
document.getElementById('changeButton').addEventListener('click',
changeContentAndStyle);
</script>
</body>
</html>
```

**OUTPUT:**



Arrow Function, DOM & CSS Manipulation Example

Original Text



Arrow Function, DOM & CSS Manipulation Example

Content Changed!

**CONCLUSION:**
This program demonstrates the effectiveness of arrow functions for concise code, along with DOM manipulation to change the content of HTML elements dynamically, and CSS manipulation
to apply visual changes. Together, these techniques provide a foundation for creating interactive
and responsive web applications.

ASSIGNMENT 7

**Aim-**Design HTML5 form and validate in Javascript

**LO-5**

**Theory-**

## Validation in JavaScript

1. **Definition:**
   **Validation in JavaScript is the process of checking user input in web forms to ensure that it meets the required criteria before the data is sent to the server.**

2. **Purpose:**

   ○ **To prevent invalid or incomplete data from being submitted.**

   ○ **To enhance user experience by giving instant feedback.**

   ○ **To reduce load on the server by catching errors at the client side.**

3. **Types of Validation:**

   ○ **Client-Side Validation: Performed in the browser using JavaScript before sending data to the server.**

   ○ **Server-Side Validation: Performed on the server after submission (used as a backup for security).**

4. **Common Checks in JavaScript Validation:**

   ○ **Required fields are not left empty.**

   ○ **Email address is in correct format.**

   ○ **Password meets length and complexity requirements.**

   ○ **Numbers or dates fall within a valid range.**

   ○ **Data entered matches expected patterns (using Regular Expressions).**

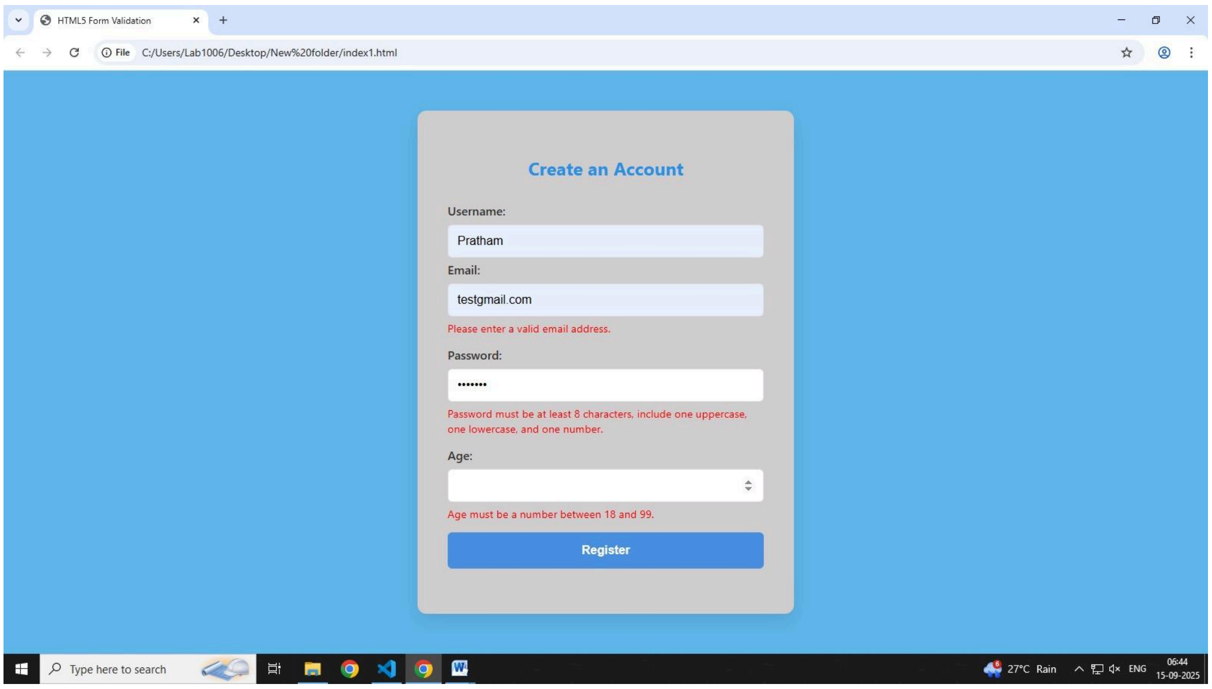
5. **How it Works:**

  - **JavaScript retrieves the value of form elements using methods like `document.getElementById()` or `document.forms[]`.**
  - **Conditions or regular expressions are applied to test the values.**
  - **If validation fails, error messages are displayed and form submission is stopped.**

**Code-**

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>HTML5 Form Validation</title>
  <style>
    body {
      font-family: 'Segoe UI', Tahoma, Geneva, Verdana, sans-serif;
      background-color: #60b7ea;
      display: flex;
      justify-content: center;
      align-items: center;
      min-height: 100vh;
      margin: 0;
      color: #333;
    }
    form {
      background-color: #d0cece;
      max-width: 500px;
      width: 100%;
      margin: 20px;
      padding: 40px;
      border-radius: 12px;
      box-shadow: 0 10px 25px rgba(0, 0, 0, 0.1);
      box-sizing: border-box;
    }
    h2 {
      text-align: center;
      color: #2c92e5;
      margin-bottom: 30px;
    }
    label {
      display: block;
      margin-bottom: 8px;
      font-weight: 600;
    }
    input, button {
      display: block;
      width: 100%;
      padding: 12px;
      margin-bottom: 20px;
      border: 1px solid #dcdfe6;
      border-radius: 6px;
      box-sizing: border-box;
      transition: all 0.3s ease;
      font-size: 16px;
    }
    input:focus {
      border-color: #4a90e2;
      box-shadow: 0 0 0 3px rgba(74, 144, 226, 0.2);
      outline: none;
    }
    button {
      background-color: #4a90e2;
      color: rgb(255, 255, 255);
      font-weight: bold;
      cursor: pointer;
      border: none;
      padding: 15px;
      margin-top: 10px;
      transition: background-color 0.3s ease;
    }
    button:hover {
      background-color: #357ABD;
    }
    .error-message {
      color: #ff0000;
      font-size: 0.9em;
      margin-top: -15px;
      margin-bottom: 15px;
    }
  </style>
</head>
<body>

  <form id="myForm" novalidate>
    <h2>Create an Account</h2>

    <label for="username">Username:</label>
    <input type="text" id="username" name="username" required minlength="3">
    <div class="error-message" id="username-error"></div>

    <label for="email">Email:</label>
    <input type="email" id="email" name="email" required>
    <div class="error-message" id="email-error"></div>
    <label for="password">Password:</label>
    <input type="password" id="password" name="password" required
pattern="(?=.*\d)(?=.*[a-z])(?=.*[A-Z]).{8,}">
    <div class="error-message" id="password-error"></div>

    <label for="age">Age:</label>
    <input type="number" id="age" name="age" min="18" max="99" required>
    <div class="error-message" id="age-error"></div>

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

  <script src="script.js"></script>

</body>
</html>
```

**Output-**



Create an Account

Username:

Pratham

Email:

testgmail.com

Please enter a valid email address.

Password:

•••••••

Password must be at least 8 characters, include one uppercase, one lowercase, and one number.

Age:

Age must be a number between 18 and 99.

Register

**CONCLUSION-**

Validation in JavaScript ensures that user inputs are correct and reliable before submission.
 It helps maintain data integrity and improves overall user experience.
 By combining client-side and server-side validation, applications become both efficient and secure.

## Assignment 8

**Aim-**Write  a program to implement concept of React Hooks (use states, use effect)

**LO-5**

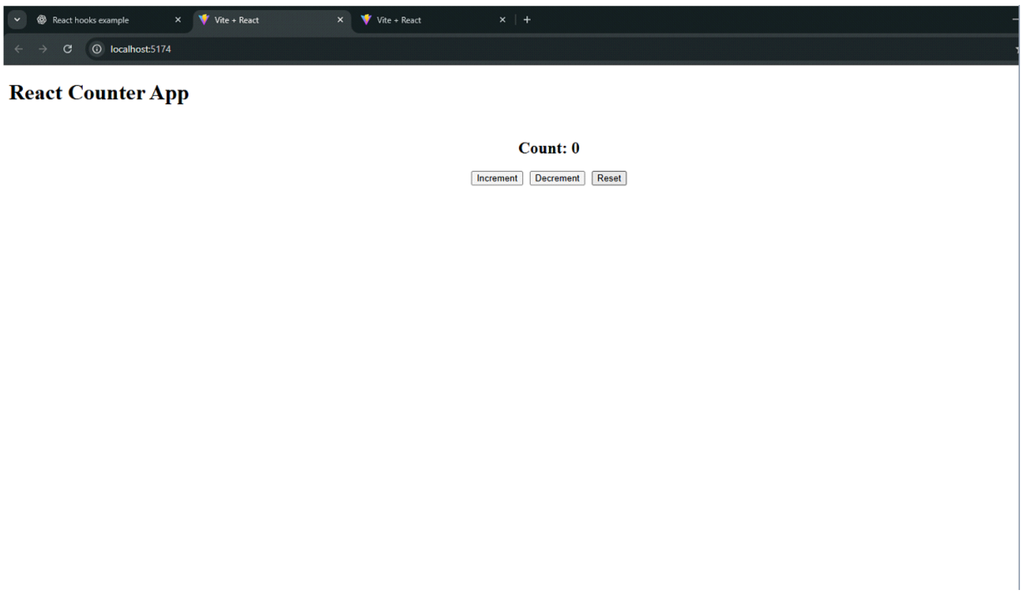
**Theory-**
In React, state represents the dynamic data of a component that determines how it renders and behaves. The useState hook allows functional components to have state variables. When the state changes, React re-renders the component to reflect the new state.
The useEffect hook lets you perform side effects in function components, such as logging to the console when the state changes.
In this example, we use useState to keep track of a count variable. The component has three main actions:

- Increment: Increase the count by 1
- Decrement: Decrease the count by 1
- Reset: Reset the count to 0

Each button updates the state using the setCount function. The component re-renders to display the updated count value, providing a dynamic and interactive user experience.

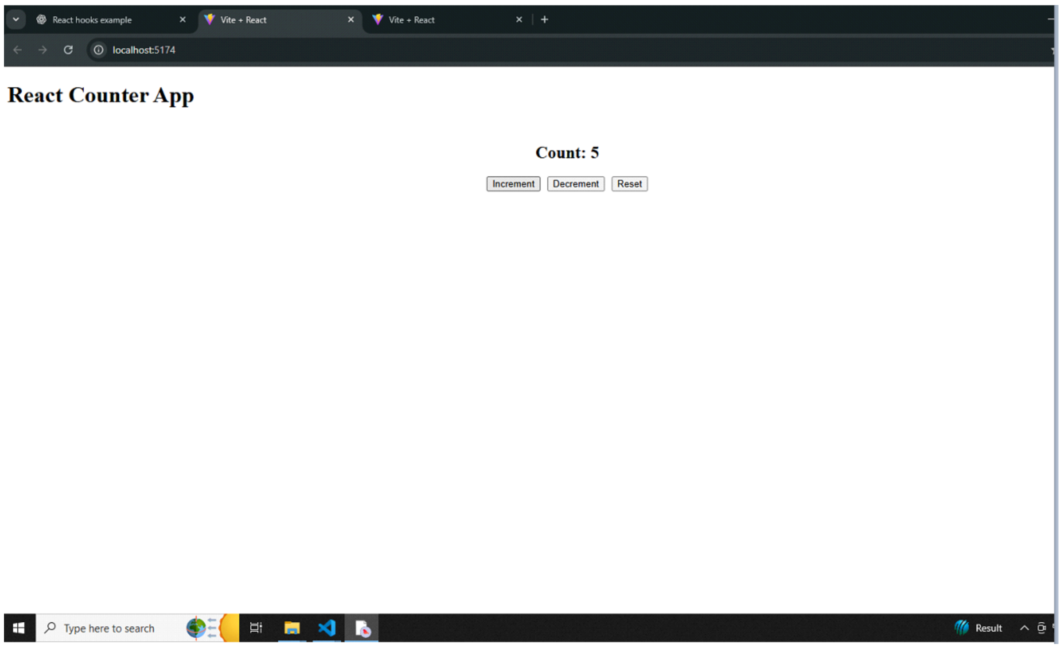
**Program-**
```
#counter
import React, { useState, useEffect } from 'react';

function Counter() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    console.log(`Count changed: ${count}`);
  }, [count]);

  const increment = () => {
    setCount(prev => prev + 1);
  };

  const decrement = () => {
    setCount(prev => prev - 1);
  };

  const reset = () => {
    setCount(0);
  };

  return (
    <div style={{ textAlign: 'center', marginTop: '50px' }}>
      <h2>Count: {count}</h2>
      <button onClick={increment}>Increment</button>
      <button onClick={decrement} style={{ margin: '0 10px' }}>Decrement</button>
      <button onClick={reset}>Reset</button>
    </div>
  );
}

export default Counter;

#app
import React from 'react';
import Counter from './Counter';

function App() {
  return (
    <div>
      <h1>React Counter App</h1>
      <Counter />
    </div>
  );
}

export default App;


#mainimport React from 'react';
import ReactDOM from 'react-dom/client';
import App from './App';

ReactDOM.createRoot(document.getElementById('root')).render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
);
```

**Output-**


React hooks example
Vite + React
Vite + React
localhost:5174

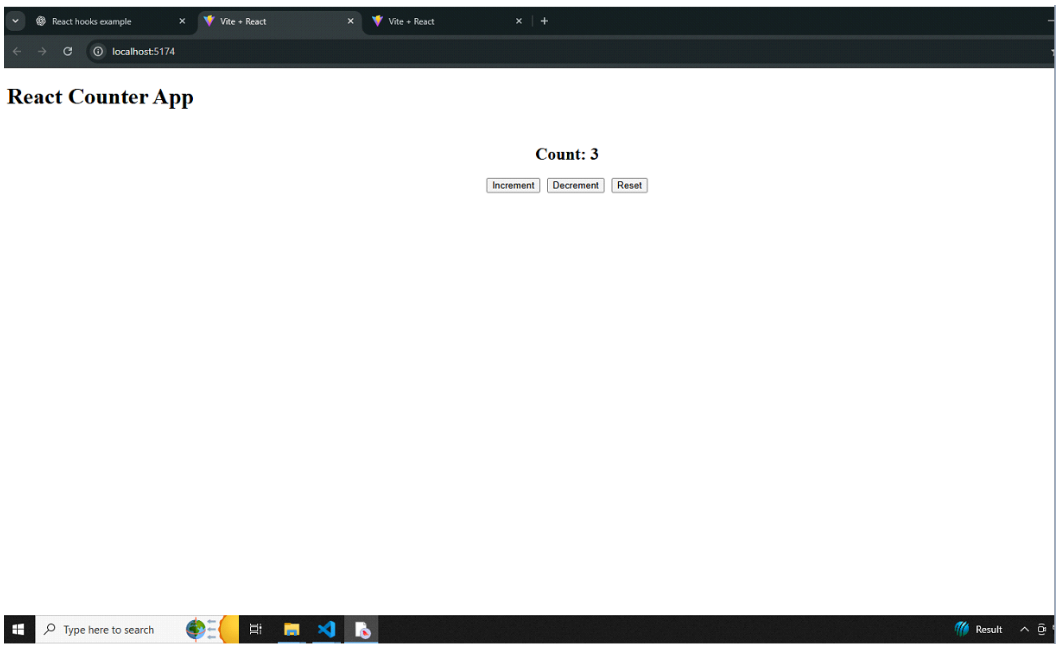
React Counter App

Count: 0


React hooks example
Vite + React
Vite + React
localhost:5174

React Counter App

Count: 5



**Conclusion-**
This React counter app demonstrates how to manage state using the useState hook and how to handle user interactions through event handlers in functional components. The addition of the useEffect hook to log state changes helps in understanding how side effects work in React.
By implementing increment, decrement, and reset functionalities, we create a simple but effective example of React's core concepts: state management, event handling, and component re-rendering. This knowledge forms the foundation for building more complex interactive applications using React.

**ASSIGNMENT-9**

**Aim-**Write a react program to implement the concept of Asynchronous programming using

**LO-5**

**Theory-**
**Definition of Asynchronous Programming**:
Asynchronous programming allows the program to execute tasks **without waiting for previous tasks to complete**, improving efficiency and responsiveness. In web development, this is commonly used for tasks like fetching data from a server, reading files, or waiting for user input.

**Promise in JavaScript**:
A **Promise** is a JavaScript object that represents the eventual completion (or failure) of an asynchronous operation.

- **States of a Promise**:
    - **Pending** – The operation is not yet completed.
    - **Fulfilled** – The operation completed successfully.
    - **Rejected** – The operation failed.

**How It Works in the Example**:

- The function `fetchUserData()` simulates fetching user data using a Promise.
- A **2-second delay** is simulated using `setTimeout()` to mimic a real API call.
- The Promise **resolves** if the data is fetched successfully, or **rejects** if an error occurs.
- The `getUserData()` function calls `fetchUserData()` and updates the HTML `<div>` based on the Promise result using `.then()` and `.catch()`.

**Advantages of Using Promises**:

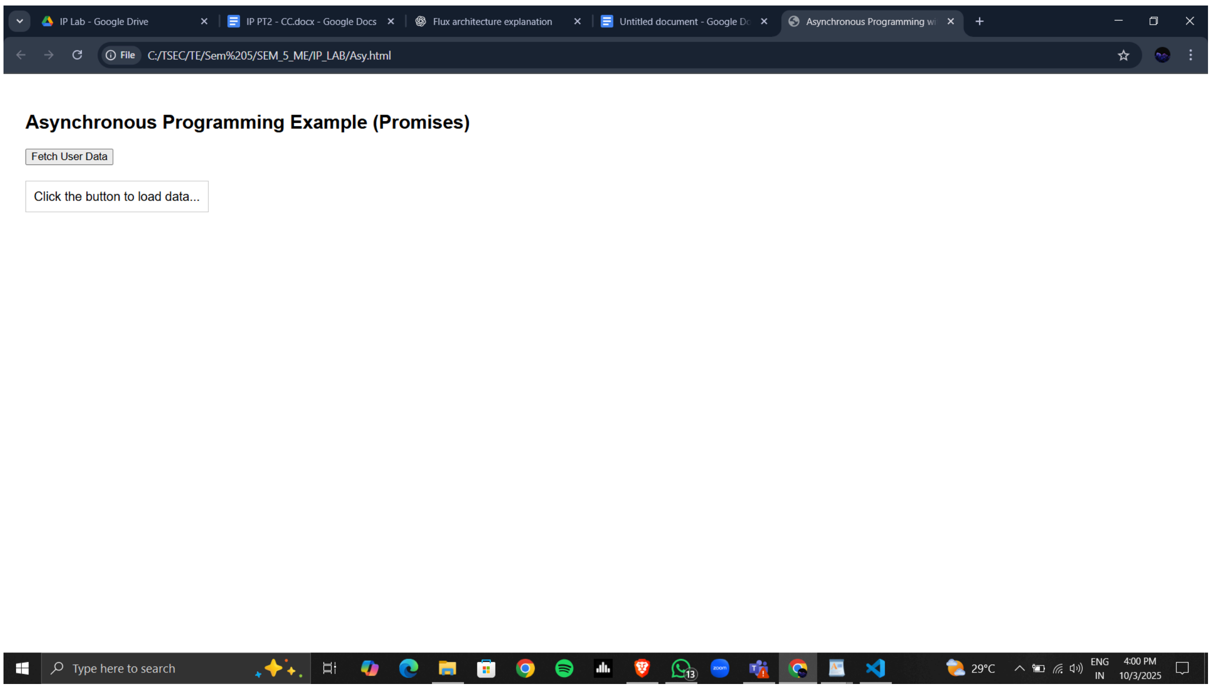
- Prevents the browser from freezing while waiting for long tasks.

- Makes asynchronous code easier to read and maintain than nested callbacks (“callback hell”).
- Provides a structured way to handle success and error cases separately.

**Program-**

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Asynchronous Programming with Promises</title>
  <style>
    body {
      font-family: Arial, sans-serif;
      padding: 20px;
    }
    #output {
      margin-top: 20px;
      padding: 10px;
      border: 1px solid #ccc;
      width: fit-content;
    }
  </style>
</head>
<body>
  <h2>Asynchronous Programming Example (Promises)</h2>
  <button onclick="getUserData()">Fetch User Data</button>

  <div id="output">Click the button to load data...</div>

  <script>
    // Function to simulate API call
    function fetchUserData() {
      return new Promise((resolve, reject) => {
        document.getElementById("output").innerText = "Fetching user
data...";

        setTimeout(() => {
          const success = true; // change to false to test rejection
          if (success) {
            resolve({ id: 1, name: "John Doe", age: 25 });
          } else {
            reject("❌ Error: Failed to fetch user data.");
          }
        }, 2000); // 2 second delay
      });
    }

    // Function called when button is clicked
    function getUserData() {
      fetchUserData()
        .then((data) => {
          document.getElementById("output").innerText =
            `✅ Data received:\nID: ${data.id}\nName:
${data.name}\nAge: ${data.age}`;
        })
        .catch((error) => {
          document.getElementById("output").innerText = error;
        });
    }
  </script>
</body>
</html>
```

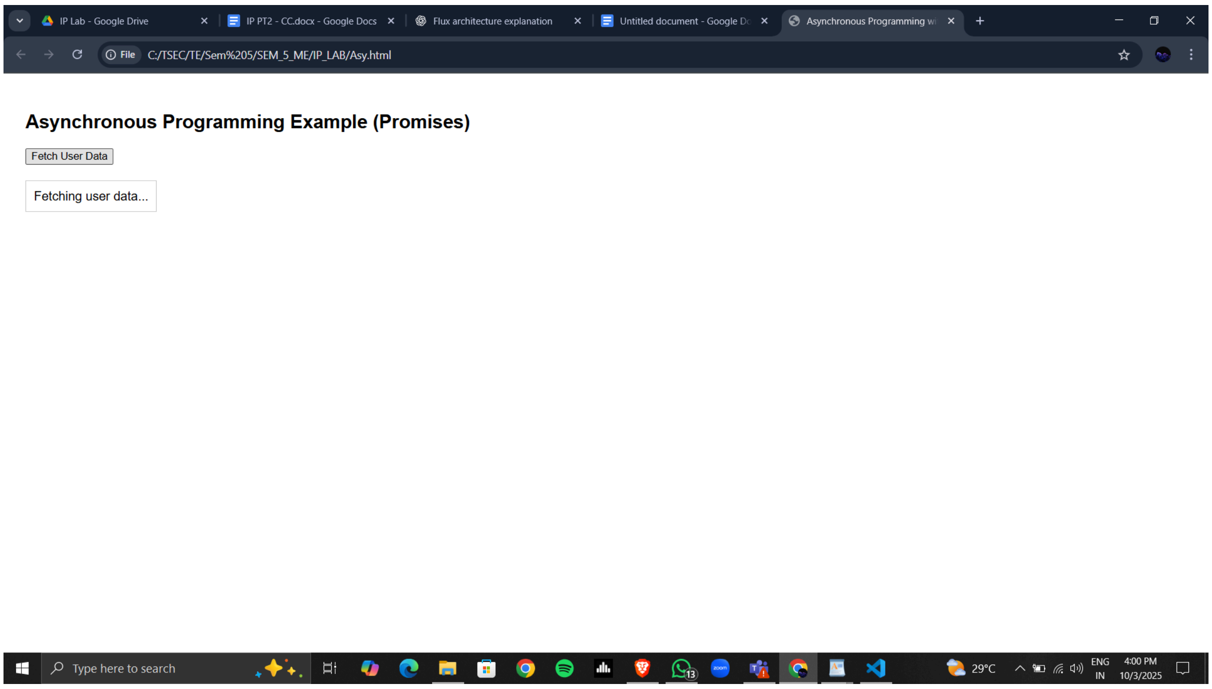
## Output-


IP Lab - Google Drive
IP PT2 - CC.docx - Google Docs
Flux architecture explanation
Untitled document - Google Do
Asynchronous Programming w
File C:/TSEC/TE/Sem%205/SEM_5_ME/IP_LAB/Asy.html
Type here to search
29°C
ENG IN
4:00 PM 10/3/2025

**Asynchronous Programming Example (Promises)**

Fetch User Data

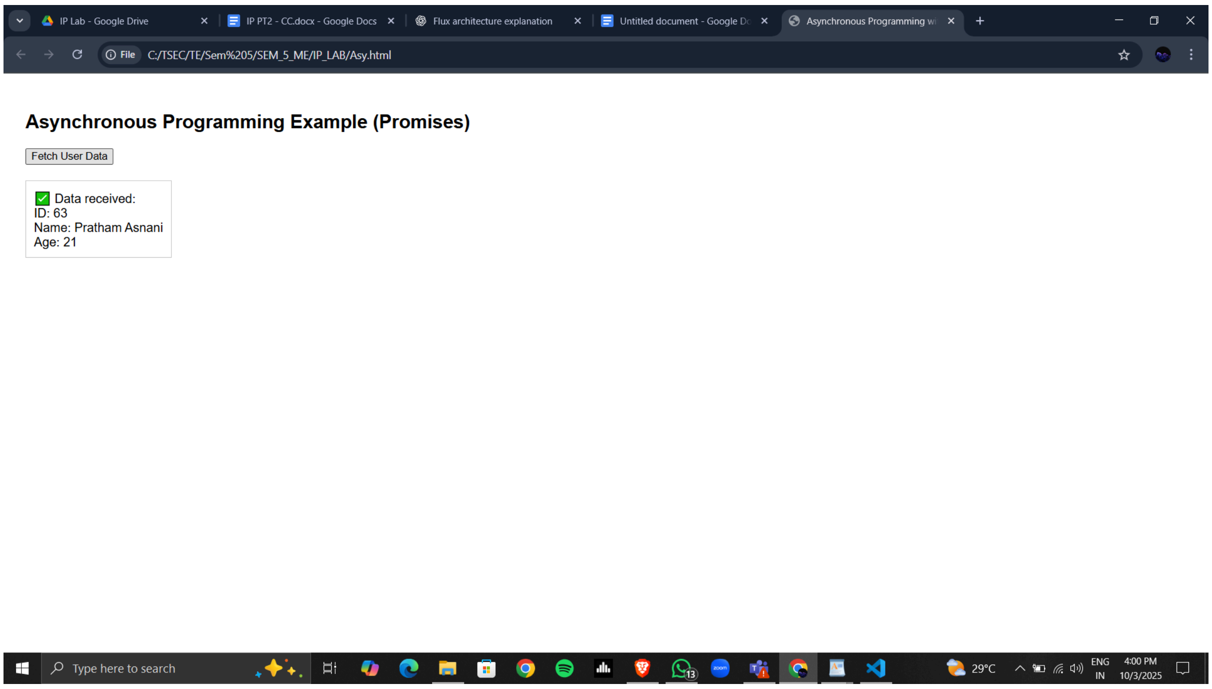
Click the button to load data...



Asynchronous Programming Example (Promises)

Fetch User Data

```
Fetching user data...
```



Asynchronous Programming Example (Promises)

Fetch User Data

```
✅ Data received:
ID: 63
Name: Pratham Asnani
Age: 21
```

**Conclusion-**

Asynchronous programming with Promises allows tasks to run in the background while the rest of the program continues executing.
It improves user experience by keeping the web page responsive during long-running tasks.
Using Promises in JavaScript provides a clean, readable way to handle asynchronous operations and errors.

**ASSIGNMENT-10**

**Aim-**Write a program Node js to

A. Create a file
B. Read the data
C. Write the data to a file
D. Rename a file
E. Delete a file

**LO-6**

**Theory-**

**File Handling in Node.js**

Node.js provides a powerful module called fs (File System) for interacting with files and directories. This module allows developers to perform various file operations such as creating,
reading, writing, renaming, and deleting files. File operations can be performed synchronously
(blocking) or asynchronously (non-blocking). Asynchronous operations are preferred in Node.js to avoid blocking the event loop, which is critical for handling multiple tasks efficiently.

**a. Creating a File**

Creating a file in Node.js involves specifying the file name and content. If the file does not exist, Node.js creates it; if it exists, the file can either be overwritten or appended. Creation can be done using methods that either write data immediately to a new file or create an empty file to be filled later.
Key points:
• Node.js automatically handles the creation if the specified file path does not exist.
• You can create a file with initial content or leave it empty.
• Asynchronous creation ensures that other processes are not blocked while the file is being created.

**b. Reading Data from a File**

Reading a file means retrieving its content and using it in the program. Node.js provides methods that can read files either as text (UTF-8 encoding) or as binary data.
Key points:
• Reading can be synchronous (blocking) or asynchronous (non-blocking).
Asynchronous reading is better for performance.
• You can read the entire file at once or stream it in chunks for large files.
• Proper error handling is crucial, as reading a non-existent file or a file with restricted permissions can throw errors.

**c. Writing Data to a File**

Writing to a file means storing data into the file system. Node.js allows writing either by replacing the existing content or by appending new data to the end of the file.
Key points:
• Writing can be synchronous or asynchronous.

• Overwriting will erase the existing content, whereas appending adds new data without removing the existing content.
• Writing operations are important for saving user data, logs, or configuration files in server-side applications.

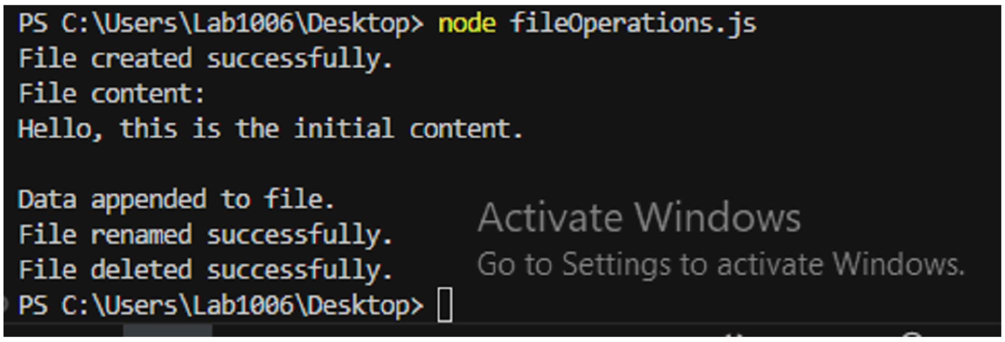
**d. Renaming a File**
Renaming a file changes its name or moves it to a different directory without altering its content. This operation is performed by specifying the current name/path and the new name/path.
Key points:
• Renaming can be used to organize files or implement version control.
• Node.js provides both synchronous and asynchronous methods to rename files.
• It is crucial to check that the target name does not conflict with an existing file unless overwriting is intended.

**e. Deleting a File**
Deleting a file removes it from the file system permanently. Node.js provides methods to safely delete files while handling potential errors like attempting to delete a non-existent file.
Key points:
• Asynchronous deletion is preferred to prevent blocking the event loop.
• Deleting files is often part of cleanup operations or temporary file handling.
• Proper error handling ensures that attempting to delete a file that does not exist does not crash the application.

**Program-**
```
const fs = require('fs');
const path = require('path');

// File names
const originalFile = path.join(__dirname, 'sample.txt');
const renamedFile = path.join(__dirname, 'renamed_sample.txt');

// a. Create a file
fs.writeFile(originalFile, 'Hello, this is the initial content.\n', (err) => {
   if (err) throw err;
   console.log('File created successfully.');

   // b. Read data from file
   fs.readFile(originalFile, 'utf8', (err, data) => {
      if (err) throw err;
      console.log('File content:');
      console.log(data);

      // c. Write (append) data to the file
      fs.appendFile(originalFile, 'Appending more content...\n', (err) => {
         if (err) throw err;
         console.log('Data appended to file.');
```

```
        // d. Rename the file
        fs.rename(originalFile, renamedFile, (err) => {
          if (err) throw err;
          console.log('File renamed successfully.');

          // e. Delete the file
          fs.unlink(renamedFile, (err) => {
            if (err) throw err;
            console.log('File deleted successfully.');
          });
        });
      });
    });
});
```

**Output-**



```
PS C:\Users\Lab1006\Desktop> node fileOperations.js
File created successfully.
File content:
Hello, this is the initial content.

Data appended to file.
File renamed successfully.
File deleted successfully.
PS C:\Users\Lab1006\Desktop>
```

**Conclusion-**
The fs module in Node.js provides a comprehensive set of functions for file handling, making server-side applications capable of performing all essential file operations. Key considerations while performing these operations include:
• Choosing asynchronous methods for better performance.
• Implementing error handling to manage exceptions gracefully.
• Ensuring correct file paths and permissions to prevent runtime issues.
These operations are fundamental in Node.js for tasks like data storage, log management, configuration handling, and dynamic file management in web applications.