1 A) Introduction to Modern JavaScript and DOM

Aim:

The aim of this experiment is to demonstrate how to link an external JavaScript file to an HTML page to enable interaction between the HTML and JavaScript code.

Description:

In this experiment, we will create a simple HTML page and link it to an external

JavaScript file. The JavaScript file will contain a basic script to interact with the HTML page. By linking the JavaScript file, we will enhance the functionality of the page, such as handling user interactions or performing dynamic content changes.

Source Code:

index.html:

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

<head>

<meta charset="UTF-8"> <meta name="viewport" content="width=device-width, initial-scale=l

<title>Linking JavaScript to

</head>

<body>

<h1>Welcome to the JavaScript Experiment</hl> <button id="clickButton">Click Me!</button>



<!-- Linking the external JavaScript file -->

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



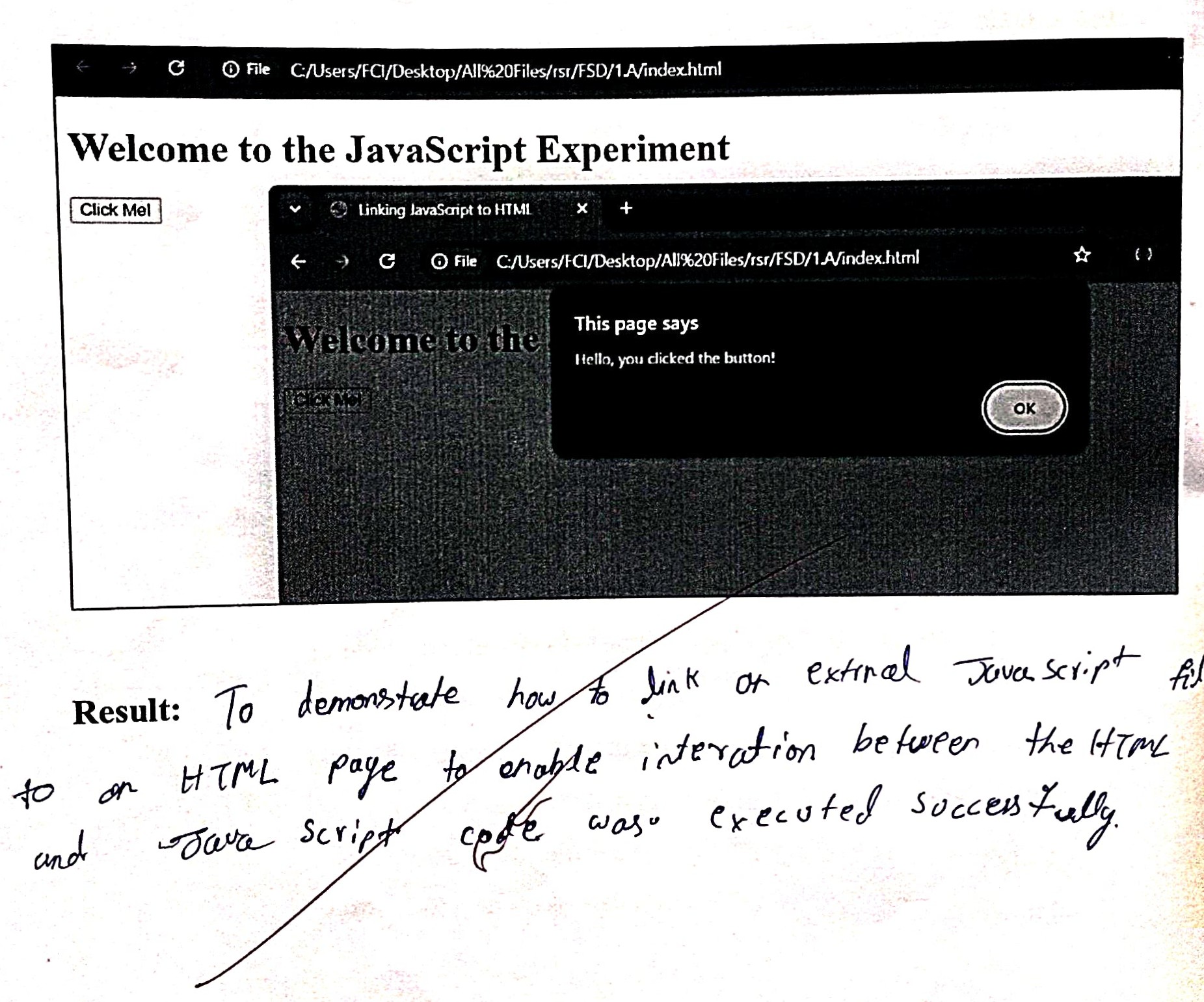
</html>

script.js:

// Accessing the button from the HTML let button = document.getElementById('clickButton'); // Adding an event listener to the button button.addEventListener('click', ftnction() { alert("Hello, you clicked the button!"); 

Output:

 When the user clicks on the "Click Me!" button, an alert message will appear that says "Hello, you clicked the button!" This demonstrates how JavaScript can be linked to HTML and interact with it dynamically.



1. Basics of React. js

Aim:

The aim of this experiment is to implement a counter button using React class components. The counter will increase each time the button is clicked.

Description:

In this experiment, we will create a React class component that manages a counter. The counter will be displayed on the page, and there will be a button that, when clicked, will increment the counter. Steps to follow:

1. Set up a React application: We will use create-react-app to set up a basic React application.

2. Create a class component: We will create a React class component to hold the counter state and handle the click event.

1. Render the counter: We will render the current counter value and a button to increase the counter.
2. Add event handler: We will add a method to handle the button click event, which will increment the counter.

Steps to Integrate the Code in a React Project:

1. Install Dependencies: If you haven't set up a React project already, run the following commands in your terminal to create a new React app:

(npx create-react-app counter-app cd counter-app

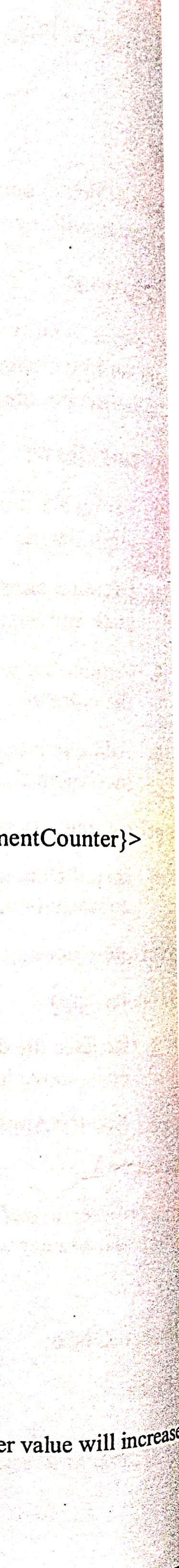
1. Replace the default App.js: Open the src/App.js file and replace its contents with the code above for the Counter class component.
2. Run the Application: Once the changes are made, save the file and run your app with: Cnpm start )

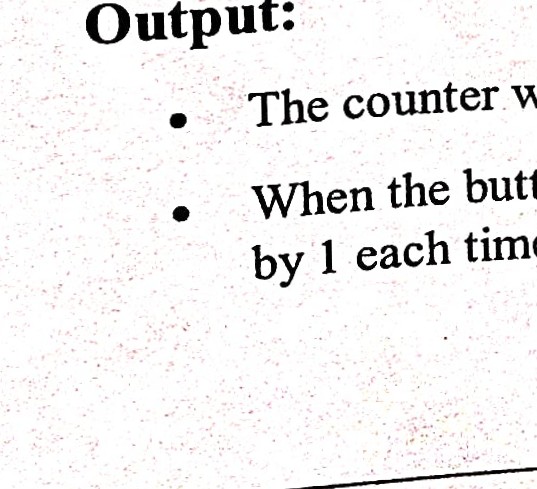
Your browser should open, showing the counter with a button. Each time you click the button, the counter will increase.

Source Code:

App.js:

import React, { Component } from 'react';

class Counter extends Component { constructor(props) { super(props); this.state = { count: 0 };



Output:

time.

incrementCounter = () {

this.setState(prev => ( { count: prev.count + 1 }));

render() { return (

<div className="counter-container">

<div className="counter-box">

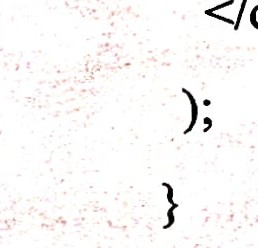
<h1>Counter: {this.state.count}</hl>

 <button className="counter-button" onClick={this.incrementCounter}>

 Increase Counter

</button>

</div>

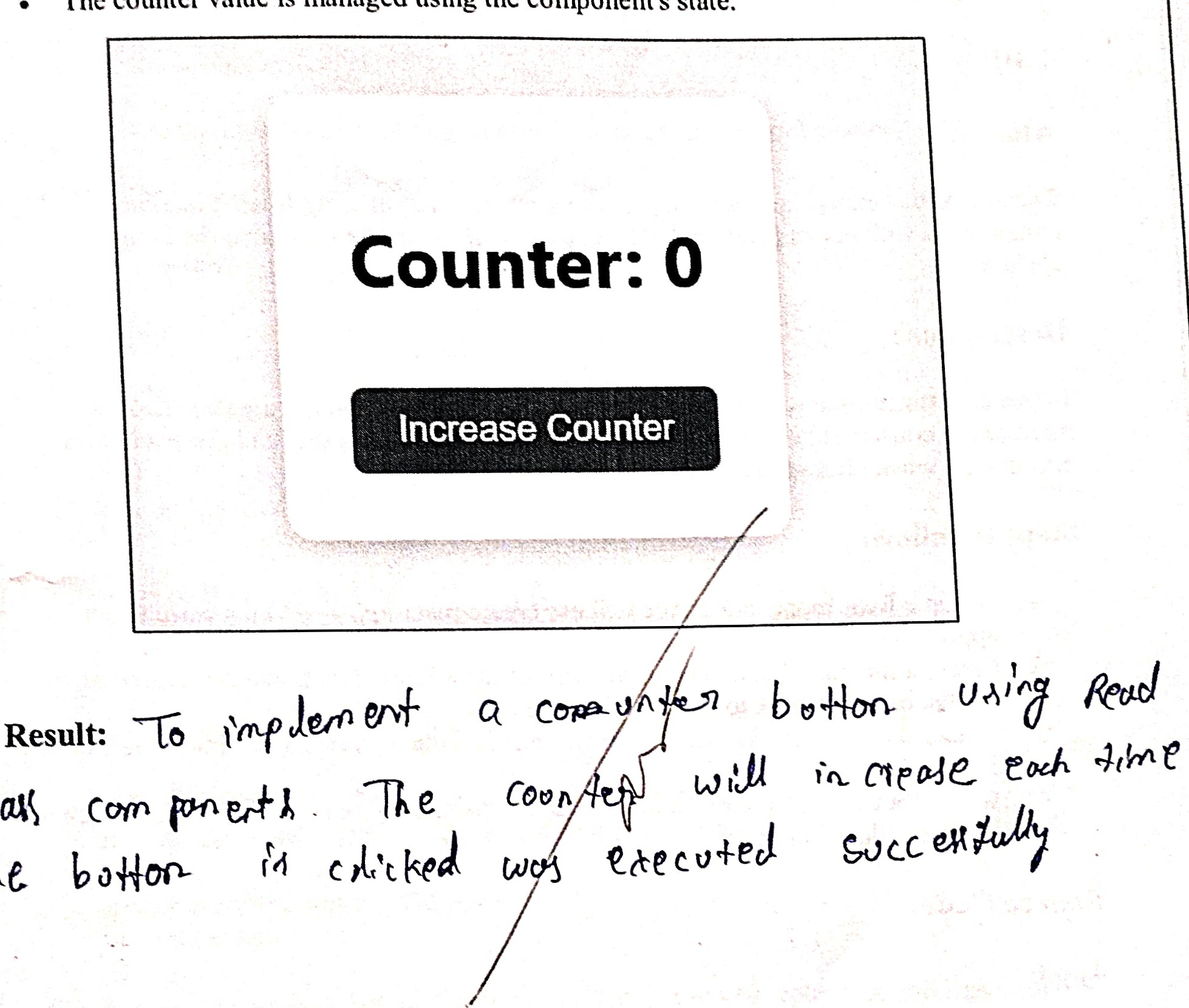
</div>

export default Counter;

will initially display 0.

When the button labeled "Increase Counter" is clicked, the counter value will increas

counter value is managed using the component's



The

state.

Aim:

The aim of this experiment is to demonstrate how to select elements in an HTML page using JavaScript selectors such as getElementById, getElementsByClassName, and querySelector.

Description:

In this experiment, we will create a simple HTML page containing various elements. We will then use JavaScript to select these elements using different selector methods and perform actions like modifying content or styles. This will help us understand how to target specific HTML elements dynamically using JavaScript.

Source Code:

 index.html:

html>

<html lang="en">

<head>

<meta

<meta name="viewport" content="width=device-width, 

<title>Selecting Elements with JavaScript</title>

<style>

.highlight { background-color: yellow;

</style>

</head>

<body>



is a simple page to demonstrate JavaScript selectors.</p>



<p

<button class="action-btn">Change Text</button>

<button id="highlightBtn">Highlight Title</button>

Linking the external JavaScript filc --> <script

</html>

script.js:

// Selecting elements using different JavaScript selectors

// 1. Selecting an element by ID let titleElement = document.getElementById('title');

// 2. Selecting elements by class name

let descriptionElement = document.getElementsByClassName('description')[O]; // First element with class 'description'

// 3. Selecting an element using querySelector (can select by any CSS selector) let actionButton = document.querySelector(t.action-btn'); // 4. Selecting multiple elements using querySelectorAll let allButtons = document.querySelectorAll('button');

// Function to change text content function changeText() { titleElement.textContent = "Text Changed!"'

 // Function to highlight the title function highlightTitle() { titleElement.classList.add(highlight');

// Event listeners for buttons  actionButton.addEventListener('click', changeText); document.getElementByIdChighlightBtn').addEventListener('click', highlightTitle)

Output:

3.B)

Aim:

The aim of this experiment is to fetch data from an external API using React's useEffect hook. We will display the fetched data in the component.

Description:

In this experiment, we will create a React functional component that fetches data from an API using the useEffect hook. The useEffect hook will ensure that the data is fetched when the component mounts. Once the data is fetched, it will be displayed in the component.

Steps to follow:

l. Set up a React application: We will use create-react-app to set up a basic React application.

1. Use useEffect hook: We will use the useEffect hook to make an API call when the component mounts.
2. Use fetch or axios for API requests: We will use the fetch API to fetch data from an external API.
3. Render the data: We will display the fetched data in the component.

Source Code:

App.js import React, { useState, useEffect } from 'react'; const FetchDataExample = () {

// State variables to store data and loading status const [data, setData] = useState([]); const [loading, setLoading] = useState(true); const [error, setError] = useState(null);

// useEffect hook to fetch data when the component mounts useEffect(() => {

// API endpoint for fetching data const fetchData = async () { try { const response = await fetch('https://jsonplaceholder.typicode.com/posts',);

thmxv new Ermt('Failed to fetch (lattit);

const result await response.json(); setData(result); // Update the state with fetched data

} catch (enor) { setEmr(error.message); // Update the error state if fetching fails

} finally { setLoading(false); // Set loading to false after data is fetched

fetchData();

[l); // Empty dependency array ensures this runs only once when component mounts

// Render the UI based on the loading and error states if (loading) { return <div>Loading...</div>;

if (error) { return <div>Error: {error}</div>;

return (

<div>

<h1>Fetched Data</hl>

{data.map(item => (

<li key= {item.

<h3>{item.title}</h3> {item.



</div>

export default FetchDataExample;

Steps to Integrate the Code in a React Project:

1. Install Dependencies: If you haven't set up a React project already, run the following commands in your terminal to create a new React app:

npx create-react-app fetch-data-app cd fetch-data-app npm start

1. Replace the default App.js: Open the src/App.js file and replace its contents with the code above for the FetchDataExample component.

* 3. Run the Application: Once the changes are made, save the file and run your app with:

npm start

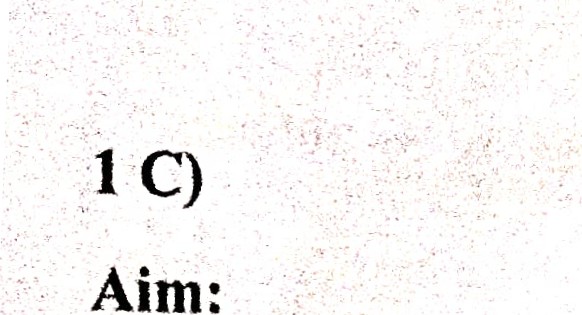
Your browser should open, and it will display the fetched data from the API.

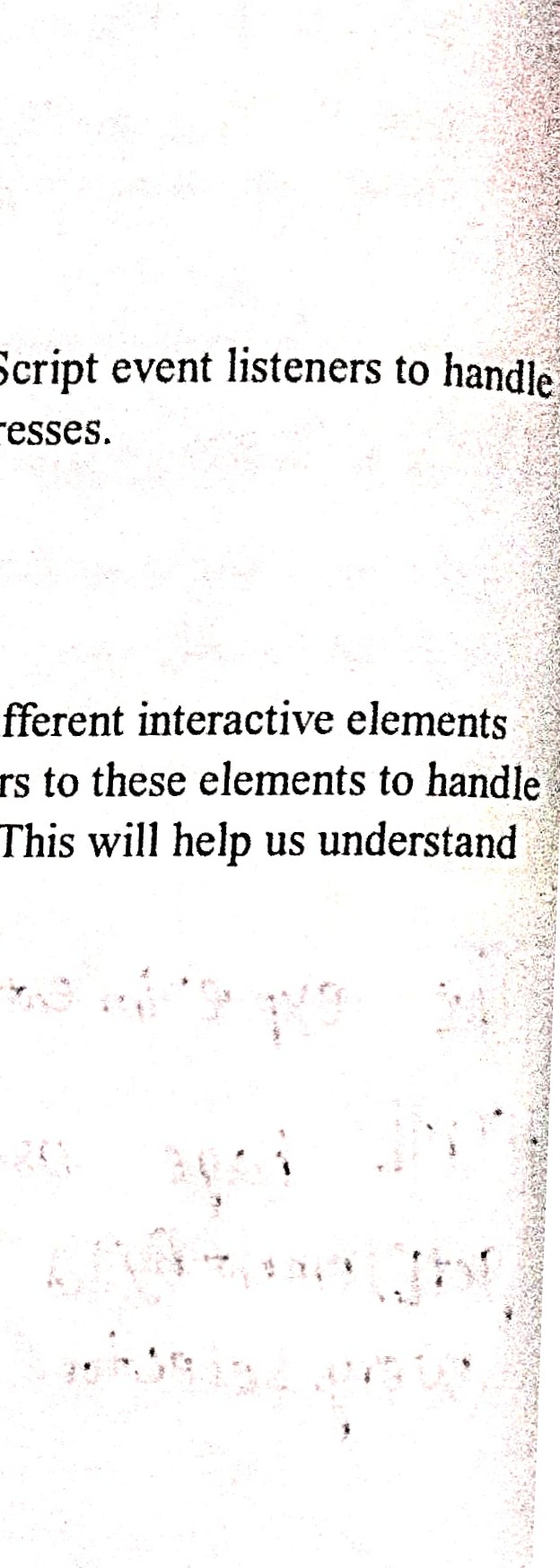
Output:

* Initially, the page will show a "Loading..." message while the data is being fetched.

Once the data is successfully fetched, it will display a list of titles and bodies from the fetched posts.

If there is an error (e.g., network issue or API error), the error message will be displayed.

Aim:



JavaScript

The aim of this experiment is to demonstrate how to use user interactions such as clicks, mouse movements, and key presses.

Description:

In this experiment, we will create a simple HTML page with different like buttons. We will then use JavaScript to attach event listeners user actions such as clicking, mouseover, and keypress events. This will how JavaScript can respond to user interactions dynamically.

Source Code: 

index.html: 

html>

<html lang="en">

<head9

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=l.O">

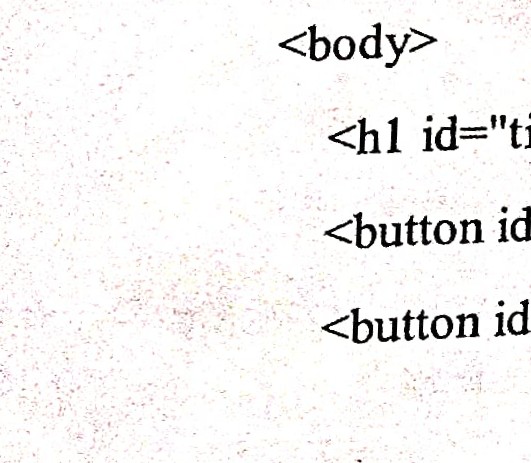
<title>Event Listeners in JavaScript</title>

 <style>

.highlight { background-color: rgb(255, 255, l);

</style>

</head>

id="title">Click the button to change text!</hl> id="changeTextBtn">Change Text</button> id="highlightBtn">Highlight Title</button> <input  placeholdct—'iTypc something" />



<!-- Linking the external JavaScript file

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

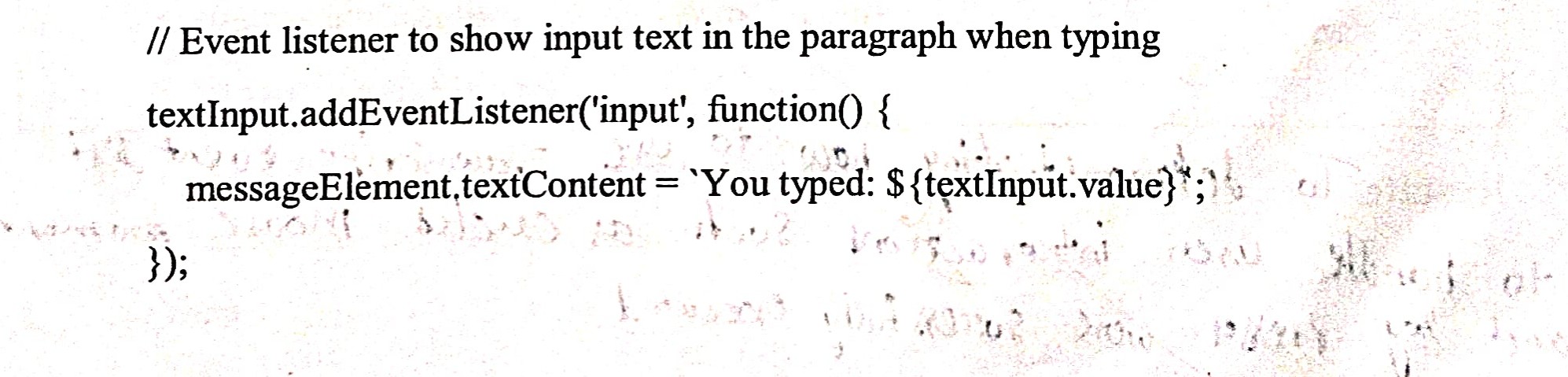
</html>

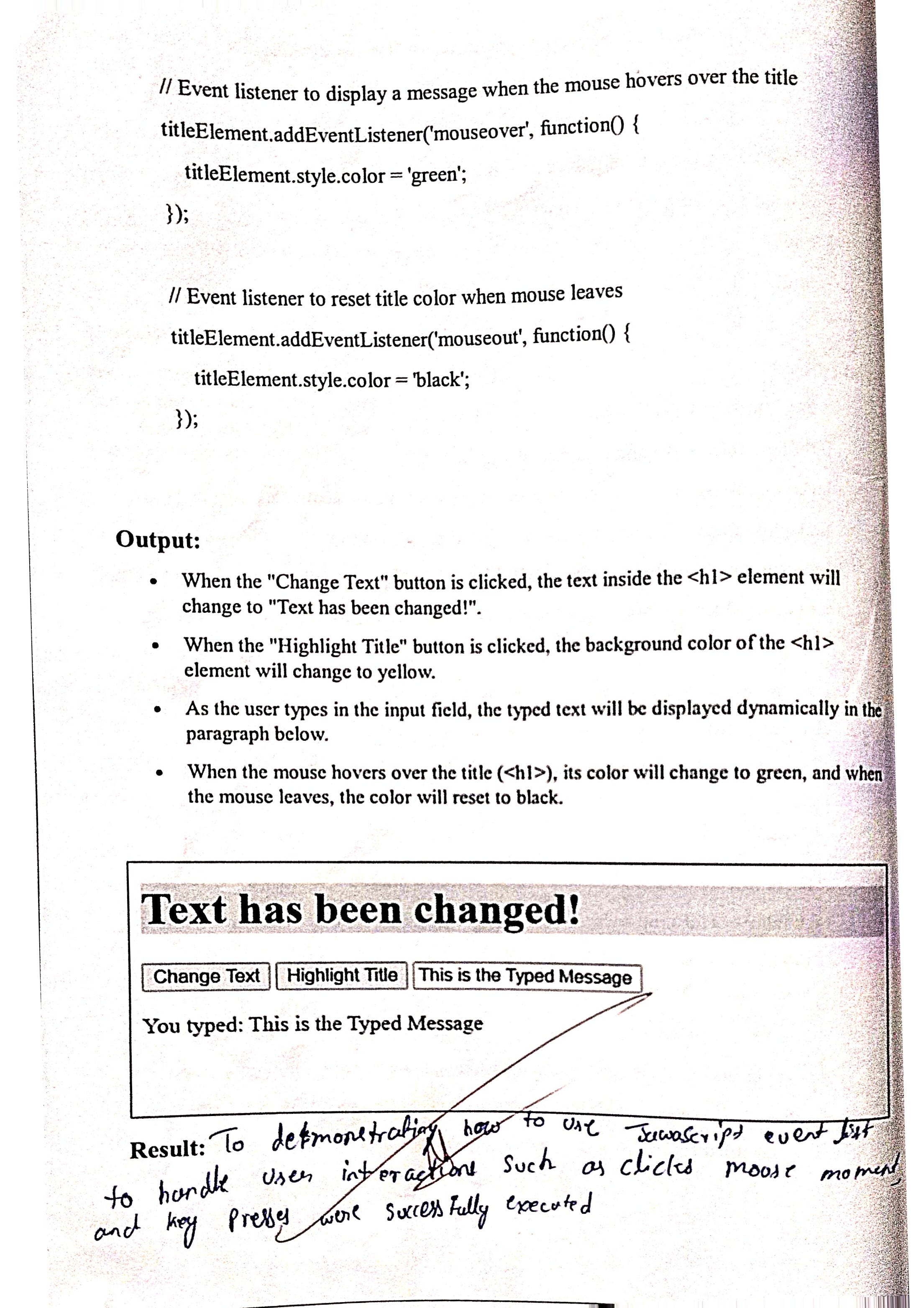
script.js:

// Selecting elements let titleElement = document.getElementById('title'); let changeTextBtn = document.getElementById('changeTextBtn'); let highlightBtn = document.getElementById('highlightBtn'); let textlnput = document.getElementById('textInput'); let messageElement = document.getElementById('message');

// Event listener to change text when the button is clicked changeTextBtn.addEventListener('click', function() { titleElement.textContent = "Text has been changed! "

// Event listener to highlight title when the button is clicked highlightBtn.addEventListener('click', function() { titleElement.classList.add(highlight');

// Event listener to show textlnput.addEventListener('input',



Aim:

 The aim of this experiment is to demonstrate how to share data between two React  components using props. One component will pass data to another component as a prop.

 Description:

In this experiment, we will create two React components: a Parent component and a Child component. The Parent component will pass data to the Child component via props. The Child component will receive the data and render it.

Steps to follow:

l. Set up a React application: We will use create-react-app to set up a basic React application.

2. Create two components: A Parent component that holds the data and a Child component that will receive and display the data.

3. Pass data via props: The Parent component will pass data to the Child component using props.

Source Codes:

parent.js:

import React from 'react'; import Child from './child';

// Parent component const Parent = O { const parentData = "Hello from Parent Component!"; // Data to pass to the Child component return (

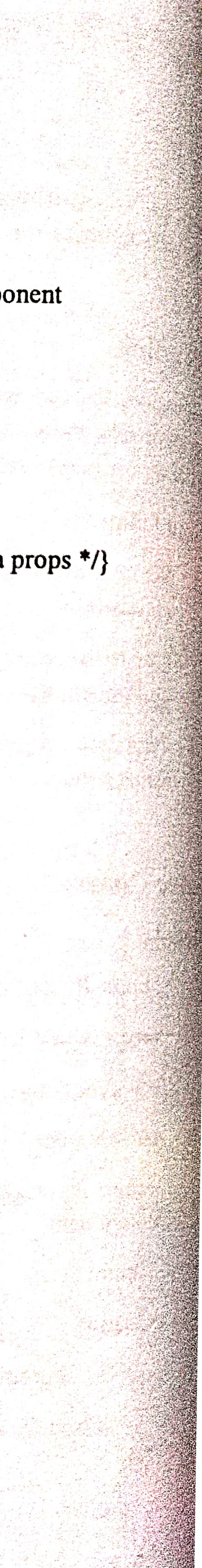
<div>

<h1>Parent Component</hl>

<Child data={parentData} Passing data to the Child component using props

</div>



export default Parent;

child.js:

import React from 'react';

// Child component that receives props from the Parent component const Child = (props) => {  return (

<div>

<h2>Child Component</h2>

<p>{props.data}</p> {l\* Displaying the data received via props

</div>

export default Child;

App.js:

import React from 'reactt; import Parent from './parcntt; // Import the Parent component const App = O { return (

<div>

<Parent /> Render the Parent component

</div>

export default App;

Steps to Integrate the Code in a React Project:

l. Install Dependencies: If you haven't set up Rcnct project tilrcndy, run the following cotntnands in your tertninnl to create n new React app:

npx create-react-app props-data-sharing-app cd props-data-sharing-app nptll statt

 2. Create the Components:

* Inside the src folder, create two new files: Parent.js and Child.js.
* Replace the code in App.js with the App.js code provided above.
* Add the Parent.js and Child.js components as shown.

3. Run the Application: Once the changes are made, save the files and run your app with:

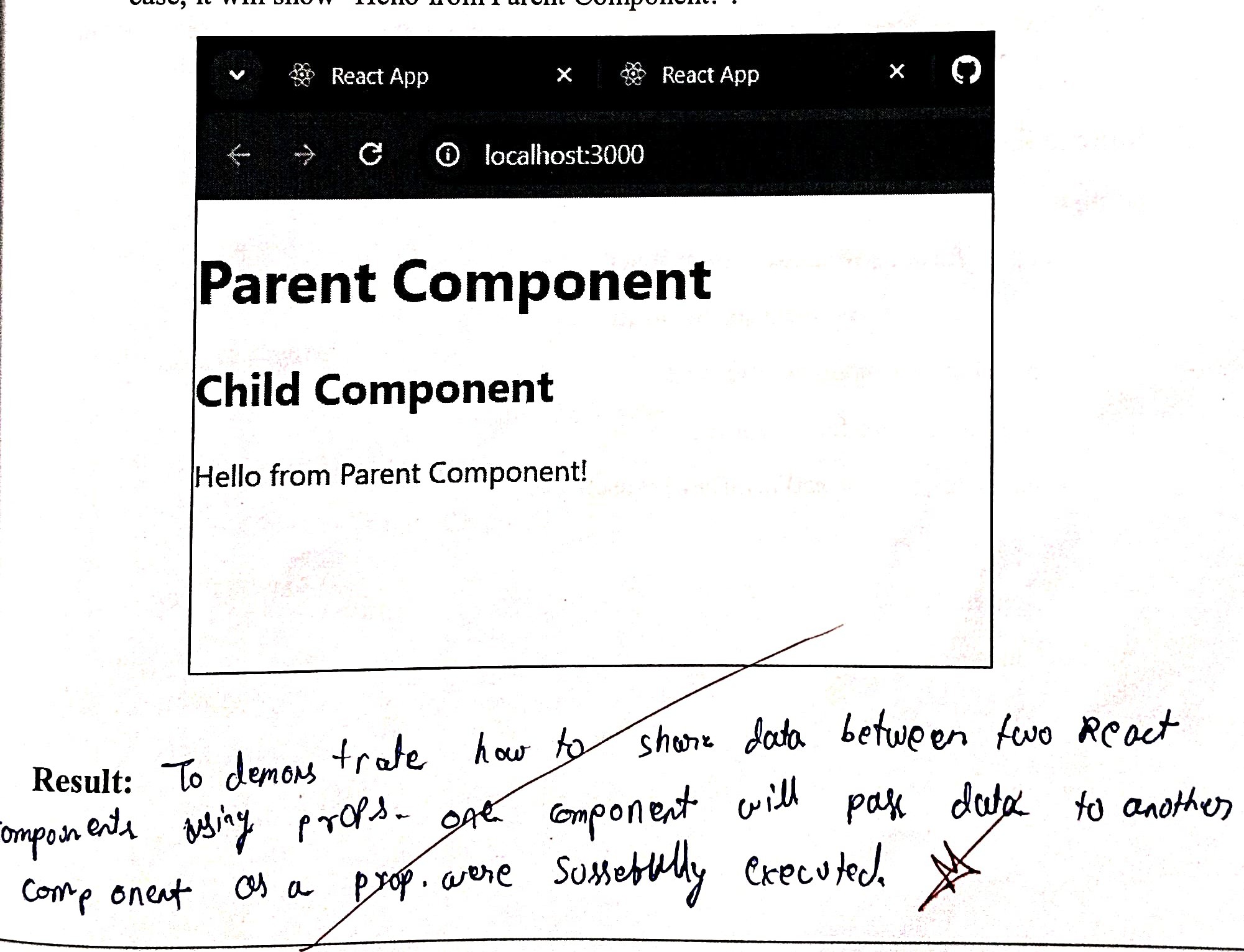
npm start

Your browser should open, and it will display the Parent component. The Parent component will pass data to the Child component, which will display it.

Output:

 The Parent component will display the text "Parent Component"

 The Child component will display the data passed from the Parent component. In this case, it will show "Hello from Parent Component!"



ID)

Aim:

The aim of this experiment is to demonstrate how to handle the click event for HTML button elements using JavaScript.

Description:

In this experiment, we will create a simple HTML page with multiple buttons. We will then use JavaScript to handle the click event for these buttons. The event listeners will execute specific actions such as changing text, displaying alerts, or modifying styles when the buttons are clicked.

Source Code: index.html:

html>

<html lang="en">

 <head>

<meta charset="UTF-8">

<meta  content="width=device-width, initial-scale=l

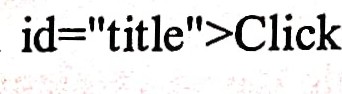
<title>Click Event Handling in JavaScript</title>

<style>

.highlight { background-color: yellow;

</head>

40

<hl  the buttons below to trigger actions</hl>

<button id="changeTextBtn">Change Text</button>

<button id="highlightBtn">Highlight Title</button>

<button id="alertBtn">Show Alert</button>

<p id="message"></p>

<!-- Linking the external JavaScript file -->

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

</body>

script.js:

// Selecting elements by ID let titleElement = document.getElementById('title'); let changeTextBtn = document.getElementById('changeTextBtn); let highlightBtn = document.getElementById('highlightBtn'); let alertBtn = document.getElementById('alertBtn'); let messageElement = document.getElementById('message');

// Event listener to change text content of the title changeTextBtn.addEventListener('click', function() { titleElement.textContent = "The text has been changed!"'

// Event listener to highlight the title by changing background color highlightBtn.addEventListener('click', ftnction() { titleElement.classList.toggle('highlight);

// Event listener to show an alert when the button is clicked alertBtn.addEventListener('click', function() { alert("This is an alert message!"); 

Output:

When the "Change Text" button is clicked, the text inside the element will change to "The text has been changed!".

When the "Highlight Title" button is clicked, the background color ofthe title will toggle between its original color and yellow (using the highlight class). 

When the "Show Alert" button is clicked, a JavaScript alert will pop up With the message "This is an alert message! ".

Aim:

The aim of this experiment is to demonstrate how to implement and handle forms in Reactt We will create a simple form to accept user input, and the form data will be managed and submitted.



collect

Description:

In this experiment, we will create a React functional component with a forrn that includes

input fields. We will use the useState hook to manage the form state. The form will user input and display the values once the form is submitted.

Steps to follow:

1. Set up a React application: We will use create-react-app to set up a basic React application.
2. Create a form: We will create a form component with various input fields.
3. Handle form input: We will use React's useState hook to capture and manage the input values.

4. Submit the form: When the form is submitted, we will display the captured input data.

Source Code:

App.js:

import React, { useState } from 'react';

// Functional component for the form const FormComponent = () {

// State to store form data const [formData, setFormData] = useState({ name:

email: ", message:

// Handle input change const handleChange = (e) { const { name, value } = e.target; setFormData({ ...formData,

[name]: value

// Handle form submission const handleSubmit = (e) => { 

e.preventDefault(); // Prevent page reload on form submission console.log('Form submitted:% formData); alert('Form submitted with name: $ {formData.name}, email: $ {formData.email} 1); 

return (

<h 1 >React Form Example</h 1 >

<form onSubmit= {handleSubmit}> 

<label htmlFor="name">Name: </label>

<input type="text"



name="name" value= {formData.name} onChange= {handleChange} required

<label </label>

<input type="email" id="email" name="email" value={formData.email} onChange={handleChange} required

</div>

<label htmlFor="message">Message: </label> <textarea id="message" name="message" value={formData.message} onChange={handleChange} required

></textarea>

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

</form>

<div>

<h2>Form Data:</h2> <p>Name: {formData.name}</p>

<p>Email: {formData.email}</p>

<p>Message: {formData.message}</p>

export default FormComponent;

Steps to Integrate the Code in a React Project:

l. Install Dependencies: If you haven't set up a React project already, run the following commands in your terminal to create a new React app:

npx create-react-app form-handling-app cd form-handling-app npm start

1. Replace the default App.js: Open the src/App.js file and replace its contents with the code above for the FormComponent.
2. Run the Application: Once the changes are made, save the file and run your app with:

npm start

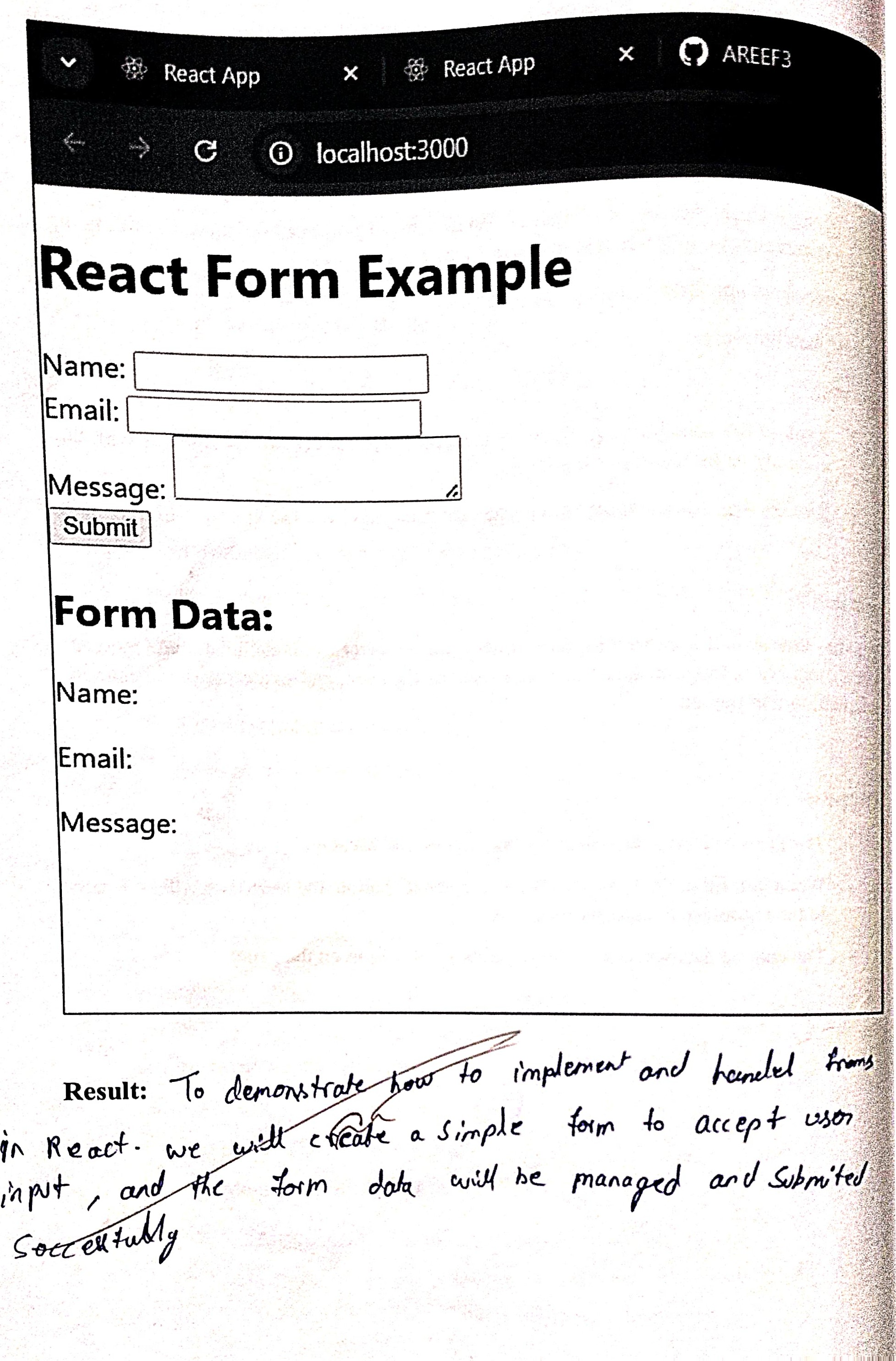
Your browser should open with the form, where you can enter the name, email, and message. Upon submission, the form data will be displayed on the page, and an alert with the entered information will pop up.

Output:

* + The form will have three fields: Name, Email, and Message.

 When you fill in the form and click the "Submit" button, the form data will be logged to the console and displayed as an alert.

* + The entered data will also be displayed below the form on the page.

acc % + AIM:

The aim of this experiment is to demonstrate and compare the following types of functions in JavaScript:

1. Function Declaration

2. Function Expression (Definition)

3. Arrow Function

Description:

In this experitfient, we Will create a simple JavaScript program that functions. These functions will perform a basic task, such as adding demonstrate how each function type works.

Source Code:

index.html:

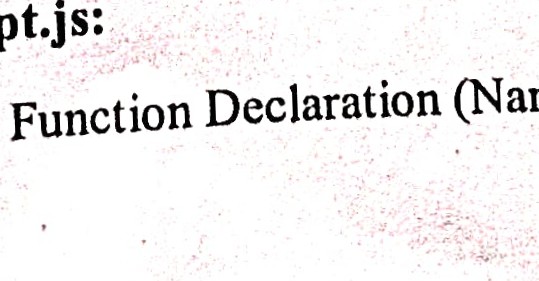
html> <html lang="en">

<head>

<meta charset="UTF-8"> 

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

<title>Different Types of Functions in JavaScript</title>



script.js:

//

1.

(Named

Function)

</head>

<body>

<h1>JavaScript Function Types</hl>

<p id="result"></p>

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

</body>

</html>

function addNumbersDeclaration(a, b) { return a + b;

// 2. Function Expression (Definition) (Anonymous Function)  const addNumbersDefinition = function(a, b) { return a + b;

// 3. Arrow Function (Concise function) const addNumbersArrow = (a, b) => a + b;

// Using all three ftnctions and displaying results let resultDeclaration addNumbersDeclaration(5, 10); // Using mction declaration



let resultDeKnition = addNumbersDefinition(5, 10); // Using functiori definition  let resultArrow = addNumbersArrow(5, 10); // Using arrow function

// Displaying the results in HTML document.getElementById('result').innerHTML = ' 

<strong>Function Declaration Result:</strong> $ {resultDeclaration} <br>

 <strong>Function Definition Result:</strong> $ {resultDefinition} <br>

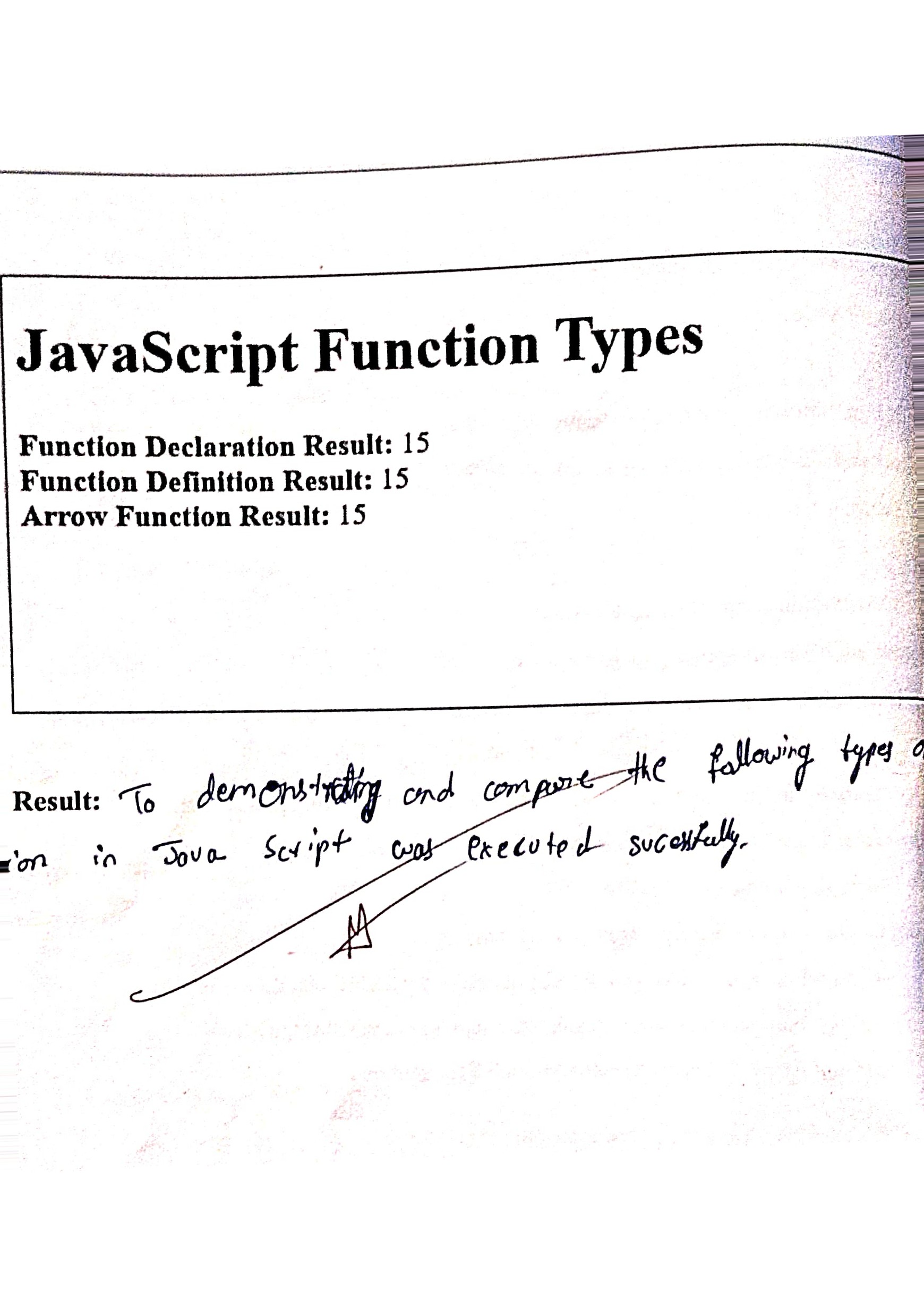
<strong>Arrow Function Result:</strong> $ {resultArrow}

Output:

The program demonstrates three types of functions:

* Function Declaration: The function addNumbersDeclaration is defined using the traditional function declaration syntax. It returns the sum of two numbers.
* Function Definition: The function addNumbersDefinition is an anonymous ftnction assigned to a variable. It works the same as the åmction declaration.
* Arrow Function: The function addNumbersArrow is defined using the arrow mnction syntax. It is a more concise version of the other two.

When the page is loaded, the sum of the numbers 5 and 10 will be calculated using all three types of functions and displayed in the browser.



Aim:

The aim of this experment is to demonstrate how to perform iterative rendering of elements in React using the map() function. We will display a list of items by iterating over an array using map().

Description:

In this experiment, we will create a React functional component that renders a list of items dynamically by iterating over an array using the map() function. The map() function will allow us to iterate over the data array and render each item as a component or list item.

Steps to follow:

1. Set up a React application: We will use create-react-app to set up a basic React application.
2. Create an array of data: We will create an array that holds data to be rendered in the component.
3. Use map() function: We will use the map() function to iterate over the array and render each item.
4. Render the list: The map() function will return a new array ofJSX elements, which will be rendered on the screen. Source Code:

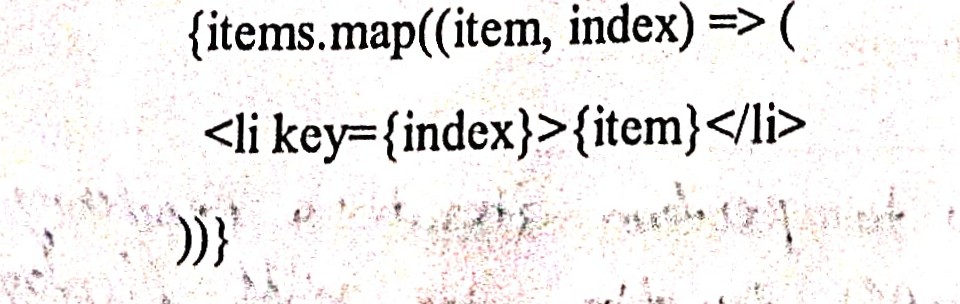
App.js:

import React from 'react'; const IterativeRenderingExample = () {

// Array of items to display const items = ['Apple', 'Banana', 'Cherry', Date', 'Elderberry]; return (

<hl>lterative Rendering Example</hl>

{/\* Using map() to iterate over the items array and render each item \*l}

{items.map((item,

<li key=

</ul>

</div>

export default IterativeRenderingExample;

Steps to Integrate the Code in a React Project:

l. Install Dependencies: If you haven't set up a React project already, run the follo commands in your terminal to create a new React app:

npx create-react-app iterative-rendering-app cd iterative-rendering-app npm start

1. Replace the default App.js: Open the src/App.js file and replace its contents with code above for the IterativeRenderingExample component.
2. Run the Application: Once the changes are made, save the file and run your app npm start

Your browser should open, displaying the list of fruits (Apple, Banana, Cherry, Date, Elderberry).

Output:

The app will display a list of fruits (Apple, Banana, Cherry, Date, Elderberry) as elements inside an unordered list (<ul>).

The list is generated dynamically using the map() function, which iterates over items array and renders each element.

|  |
| --- |
|  |
| Iterative Rendering Example   * Apple * Banana * Date * Eld |

Result: (o 

t..,e 4.JJ.Å

eoUkJ