GEETHANJALI INSTITUTE OF SCIENCE & TECHNOLOGY *(Approved by AICTE, New Delhi & Affiliated to JNTUA, Anantapuramu)*
3rd Mile, Bombay Highway, Gangavaram (V), Kovur(M), SPSR Nellore (Dt),
Andhra Pradesh, India- 524137

Full Stack Development - II

LABORATORY MANUAL

Department of
COMPUTER SCIENCE & ENGINEERING (AI & ML)

Name	
Roll No.	
Class	III B.TECH I SEM
Branch	CSE (AI & ML)
Regulation	R23
Name of the Lab	Full Stack Development - II
Academic year	2025-26

	GEETHANJALI INSTITUTE OF SCIENCE & TECHNOLOGY
	A Unit of USHODAYA EDUCATIONAL SOCIETY
	An ISO 9001:2015 certified Institution: Recognized under Sec. 2(f)& 12(B) of UGC Act, 1956
	3rd Mile, Bombay Highway, Gangavaram (V), Kovur(M), SPSR Nellore (Dt), Andhra Pradesh, India- 524137
	Ph. No. 08622-212769, E-Mail: geethanjali@gist.edu.in, Website: www.gist.edu.in

DEPARTMENT OF CSE(AI&ML)

VISION

To become a learning-resource center producing system engineers suitable for career roles in Artificial Intelligence and Machine Learning domains.

MISSION

DM1. Imparting quality education through industry relevant curriculum and effective teaching-learning methodologies.

DM2. Organizing Professional skill development programmes to cater to the societal needs through Institute-Industry interface.

DM3. Incorporating life skill development programmes aimed at promoting social responsibility on ethical foundations.

DM4. Fostering employability skills by incorporating soft skills in the curriculum.

PROGRAMME SPECIFIC OUTCOMES (PSOs)

PSO1: **Professional Skills**: Proficiency in multiple skill sets which enable the learners to excel in divergent platforms in industries

PSO2: **Successful Career and Entrepreneurship:** Apply the Artificial Intelligence & Machine Learning concepts in wide ranging fields of engineering culminating in successful careers and entrepreneurs with a special focus on societal problem solving.

PROGRAMME EDUCATIONAL OBJECTIVES (PEOs)

PEO1: Carving successful careers in industry, academia with expertise in Artificial Intelligence & Machine Learning (AI & ML)

PEO2: Excelling as socially committed Artificial Intelligence & Machine Learning (AI & ML) Engineers.

PEO3: Engaged in the successful execution of multi-disciplinary projects for the welfare and progress of humankind.

PEO4: Pursuing higher studies, research activities and initiatives of entrepreneurship.

1. Introduction to Modern JavaScript and DOM

a. Write a JavaScript program to link JavaScript file with the HTML page
b. Write a JavaScript program to select the elements in HTML page using selectors
c. Write a JavaScript program to implement the event listeners
d. Write a JavaScript program to handle the click events for the HTML button elements
e. Write a JavaScript program with three types of functions:
i. Function declaration
ii. Function definition
iii. Arrow functions

Program 1(a): Write a JavaScript program to link JavaScript file with the HTML page

Aim

To demonstrate how to link an external JavaScript file to an HTML page and execute JavaScript code inside the browser.

Concepts Involved

- **HTML & JavaScript integration** → HTML provides structure while JS adds interactivity.
- **Script tag** → <script src="..."></script> is used to connect an external JS file.
- **Separation of concerns** → Keeping JS in a separate file improves code organization and reusability.
- **Execution in browser** → Browser interprets HTML first, then loads and executes the JS.

Code

index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Linking JavaScript</title>
  <style>
    body {
      font-family: 'Segoe UI', Tahoma, sans-serif;
```

```
  background: linear-gradient(135deg, #74ebd5, #ACB6E5);
  display: flex;
  justify-content: center;
  align-items: center;
  height: 100vh;
  margin: 0;
}
.container {
  background: white;
  padding: 30px;
  border-radius: 15px;
  box-shadow: 0 8px 20px rgba(0,0,0,0.1);
  text-align: center;
  max-width: 500px;
}

h1 {
  color: #2c3e50;
  margin-bottom: 20px;
}

button {
  margin-top: 20px;
  padding: 10px 20px;
  font-size: 16px;
  font-weight: bold;
  color: white;
  background: #2980b9;
  border: none;
  border-radius: 10px;
```

```
    cursor: pointer;
    transition: background 0.3s ease, transform 0.2s ease;
   }

   button:hover {
    background: #1f6690;
    transform: scale(1.05);
   }
  </style>
</head>
<body>
  <div class="container">
   <h1>JavaScript Linking Demo</h1>
   <button onclick="showMessage()">Click Me</button>
  </div>

  <script src="script.js"></script>
</body>
</html>
```

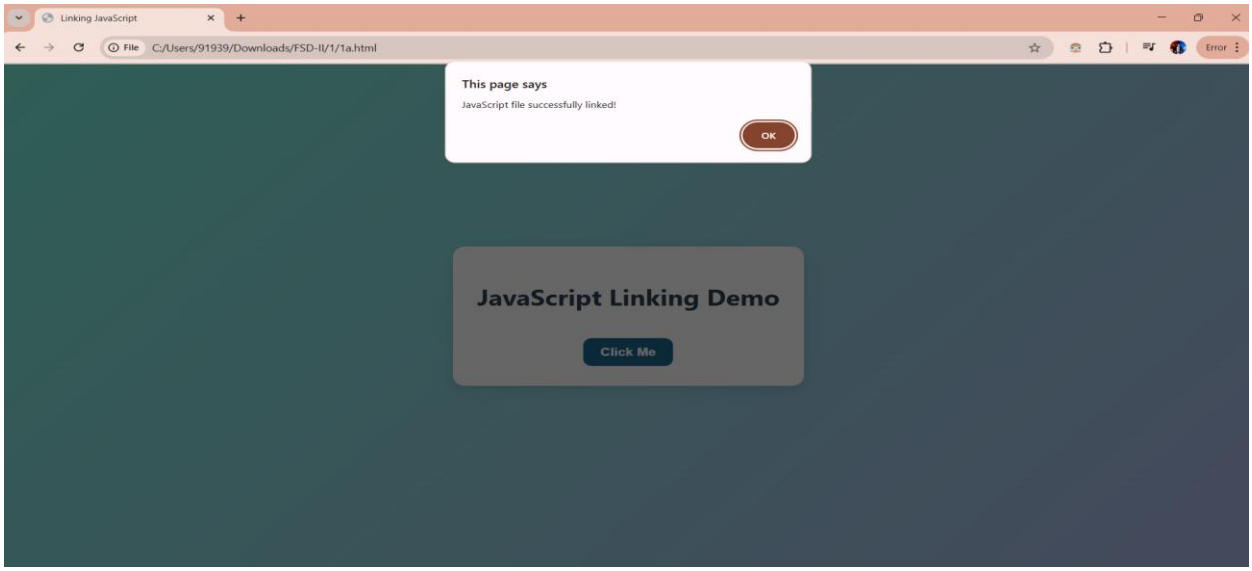
script.js

```
function showMessage() {
  alert("JavaScript file successfully linked!");
}
```

Output

When you open **index.html** in your browser and click the **button**, a popup alert will appear:

JavaScript file successfully linked!



Program 1(b): Write a JavaScript program to select the elements in HTML page using selectors

Aim

To demonstrate different ways of selecting HTML elements using JavaScript DOM selectors and display the results dynamically on the webpage.

Concepts Involved

- **getElementById** → Selects a single element by its unique ID.
- **getElementsByClassName** → Selects multiple elements with the same class.
- **getElementsByTagName** → Selects elements based on their tag name.
- **querySelector & querySelectorAll** → Modern CSS-style selectors for single and multiple elements.
- **DOM Manipulation** → Using innerHTML to show results directly inside the webpage.

Code

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
```

```
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>Selectors Demo</title>
<style>
  body {
    font-family: 'Segoe UI', Tahoma, sans-serif;
    background: linear-gradient(135deg, #d4fc79, #96e6a1);
    display: flex;
    justify-content: center;
    align-items: center;
    height: 100vh;
    margin: 0;
  }
  .container {
    background: white;
    padding: 30px;
    border-radius: 15px;
    box-shadow: 0 8px 20px rgba(0,0,0,0.1);
    text-align: center;
    max-width: 500px;
  }
  h1 {
    color: #2c3e50;
    margin-bottom: 20px;
  }
  p {
    color: #34495e;
    margin: 8px 0;
  }
  button {
    margin-top: 20px;
```

```
      padding: 10px 20px;
      font-size: 16px;
      font-weight: bold;
      color: white;
      background: #27ae60;
      border: none;
      border-radius: 10px;
      cursor: pointer;
      transition: background 0.3s ease, transform 0.2s ease;
    }
    button:hover {
      background: #219150;
      transform: scale(1.05);
    }
    .output {
      margin-top: 20px;
      padding: 15px;
      background: #ecf0f1;
      border-radius: 10px;
      text-align: left;
      font-size: 14px;
      color: #2c3e50;
    }
  </style>
</head>
<body>
  <div class="container">
    <h1 id="heading">JavaScript Selectors Example</h1>
    <p class="text">This is paragraph one.</p>
    <p class="text">This is paragraph two.</p>
```

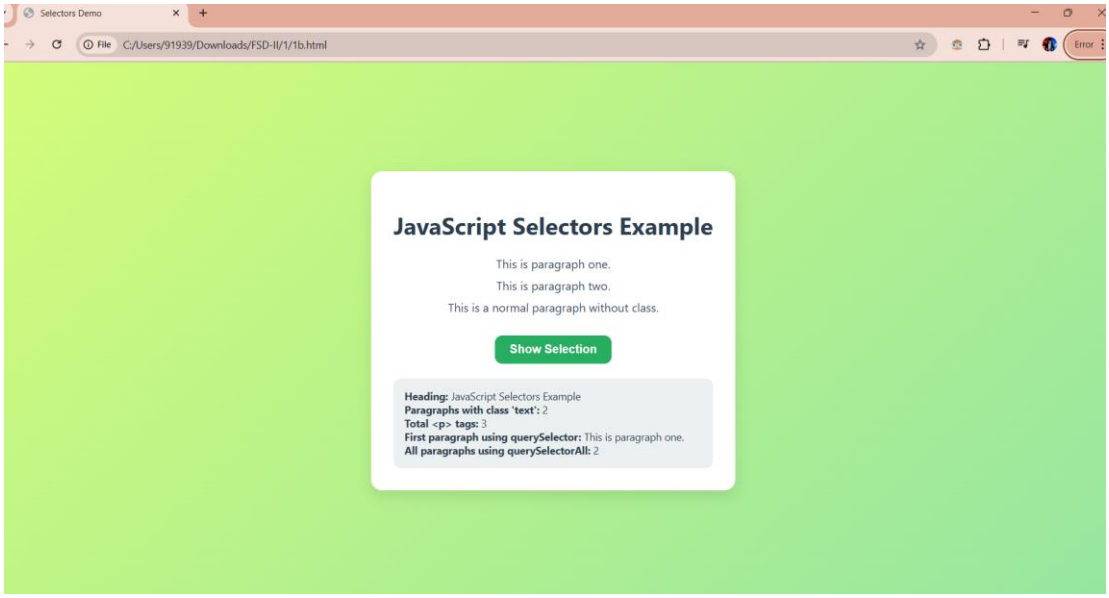
```
    <p>This is a normal paragraph without class.</p>
    <button id="btn">Show Selection</button>
    <div class="output" id="output"></div>
  </div>

  <script>
    document.getElementById("btn").addEventListener("click", () => {
      let heading = document.getElementById("heading").innerText;
      let paragraphs = document.getElementsByClassName("text");
      let tags = document.getElementsByTagName("p");
      let firstPara = document.querySelector(".text").innerText;
      let allParas = document.querySelectorAll(".text");

      document.getElementById("output").innerHTML = `
        <strong>Heading:</strong> ${heading} <br>
        <strong>Paragraphs with class 'text':</strong> ${paragraphs.length} <br>
        <strong>Total &lt;p&gt; tags:</strong> ${tags.length} <br>
        <strong>First paragraph using querySelector:</strong> ${firstPara} <br>
        <strong>All paragraphs using querySelectorAll:</strong> ${allParas.length}
      `;
    });
  </script>
</body>
</html>
```

Output

When the page loads, you see:

- A **beautiful gradient background**.
- A **white card** with heading, three paragraphs, and a styled button.

When you click **Show Selection**, the results are displayed neatly on the page itself:



Program 1(c): Write a JavaScript program to implement the event listeners

Aim

To demonstrate how to use JavaScript event listeners (addEventListener) to handle user interactions like click, mouseover, and keypress events.

Concepts Involved

- **Event Listeners** → Use addEventListener() instead of inline HTML events for clean and reusable code.
- **Click Events** → Triggered when a button or element is clicked.
- **Mouse Events** → Events such as mouseover and mouseout to make UI interactive.
- **Keyboard Events** → Using keydown or keyup to respond when a key is pressed.
- **DOM Manipulation** → Changing content or style dynamically when events occur.

Code

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
```

```
<title>Event Listeners Demo</title>
<style>
  body {
    font-family: 'Segoe UI', Tahoma, sans-serif;
    background: linear-gradient(135deg, #89f7fe, #66a6ff);
    display: flex;
    justify-content: center;
    align-items: center;
    height: 100vh;
    margin: 0;
  }
  .container {
    background: white;
    padding: 30px;
    border-radius: 15px;
    box-shadow: 0 8px 20px rgba(0,0,0,0.1);
    text-align: center;
    max-width: 500px;
  }
  h1 {
    color: #2c3e50;
    margin-bottom: 20px;
  }
  button {
    margin: 10px;
    padding: 10px 20px;
    font-size: 16px;
    font-weight: bold;
    color: white;
    background: #2980b9;
```

```
      border: none;
      border-radius: 10px;
      cursor: pointer;
      transition: background 0.3s ease, transform 0.2s ease;
    }
    button:hover {
      background: #1f6691;
      transform: scale(1.05);
    }
    input {
      margin-top: 20px;
      padding: 10px;
      border: 2px solid #2980b9;
      border-radius: 8px;
      font-size: 16px;
      width: 80%;
    }
    .output {
      margin-top: 20px;
      padding: 15px;
      background: #ecf0f1;
      border-radius: 10px;
      text-align: left;
      font-size: 14px;
      color: #2c3e50;
    }
  </style>
</head>
<body>
  <div class="container">
```

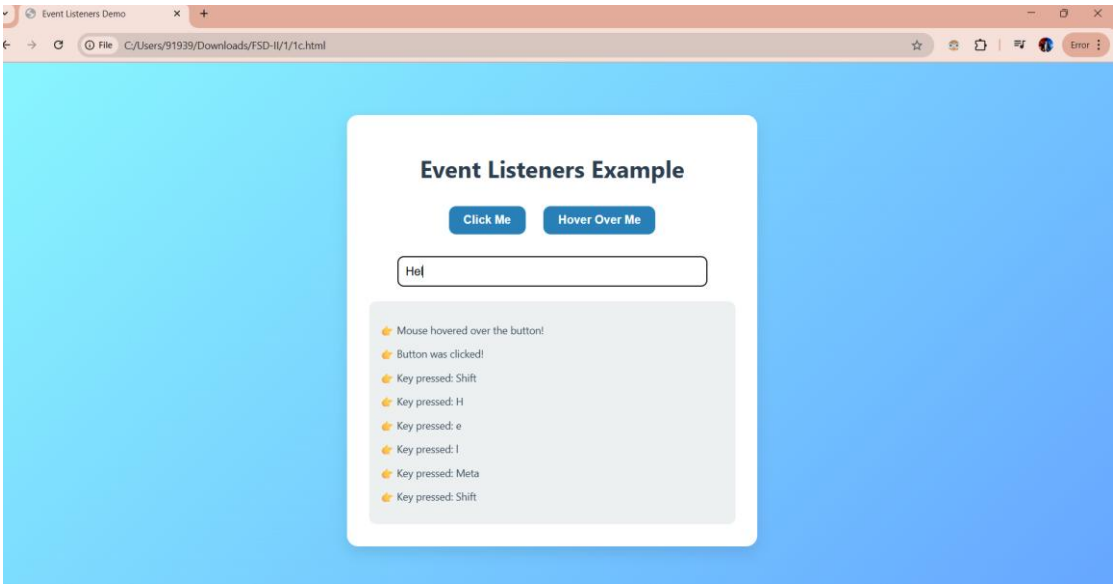
```
    <h1>Event Listeners Example</h1>
    <button id="clickBtn">Click Me</button>
    <button id="hoverBtn">Hover Over Me</button>
    <input type="text" id="inputBox" placeholder="Type something...">
    <div class="output" id="output"></div>
  </div>

  <script>
    const output = document.getElementById("output");

    // Click Event
    document.getElementById("clickBtn").addEventListener("click", () => {
      output.innerHTML += "<p>👉 Button was clicked!</p>";
    });

    // Mouseover Event
    document.getElementById("hoverBtn").addEventListener("mouseover", () => {
      output.innerHTML += "<p>👉 Mouse hovered over the button!</p>";
    });

    // Keyboard Event
    document.getElementById("inputBox").addEventListener("keydown", (event) => {
      output.innerHTML += `<p>👉 Key pressed: ${event.key}</p>`;
    });
  </script>
</body>
</html>
```

Output

1. When you open the page, you'll see:

- A gradient background.
- A white card with heading, two styled buttons, and an input box.

2. Actions:
   - **Clicking** the **Click Me** button adds a message → "👉 Button was clicked!".
   - **Hovering** over the **Hover Over Me** button adds → "👉 Mouse hovered over the button!".
   - **Typing** in the input box logs the keys pressed → e.g., "👉 Key pressed: A".
3. All messages are displayed inside the **Output Box** dynamically.



1(d) JavaScript Program to Handle Click Events for HTML Buttons

Aim

To demonstrate how to handle click events on multiple HTML button elements using JavaScript.

Concepts Involved

- **Click Event** → Triggered when a button is clicked.
- **Event Binding** → Attaching JavaScript functions to HTML buttons.
- **Multiple Buttons** → Each button can perform different actions.
- **Dynamic Feedback** → Updating page content when user interacts.

Code

```
<!DOCTYPE html>

<html lang="en">
```

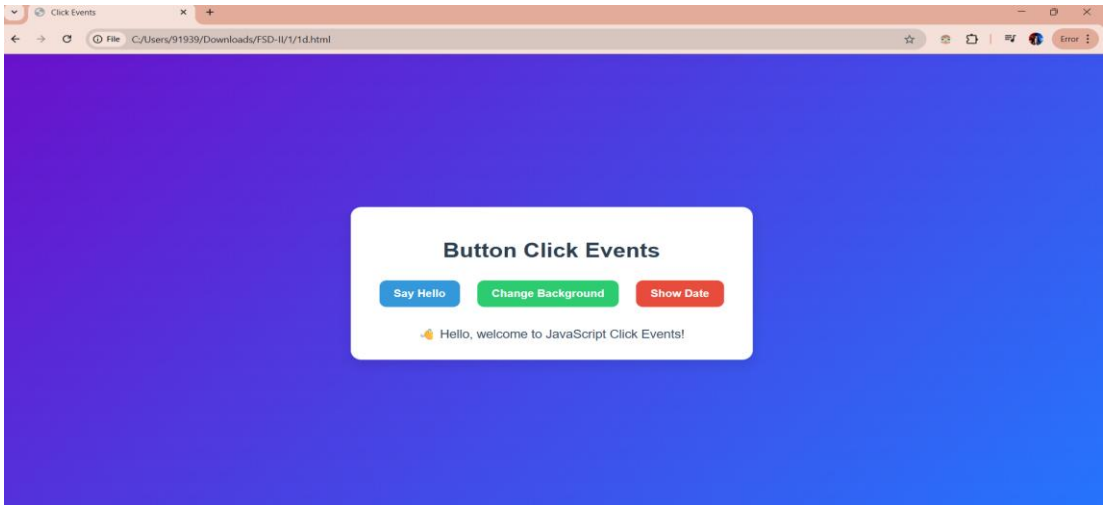
```
<head>
  <meta charset="UTF-8">
  <title>Click Events</title>
  <style>
    body {
      font-family: Arial, sans-serif;
      background: linear-gradient(135deg, #ff9a9e, #fad0c4);
      display: flex;
      justify-content: center;
      align-items: center;
      height: 100vh;
      margin: 0;
    }
    .container {
      background: white;
      padding: 30px;
      border-radius: 15px;
      box-shadow: 0 8px 20px rgba(0,0,0,0.1);
      text-align: center;
    }
    h1 {
      margin-bottom: 20px;
      color: #2c3e50;
    }
    button {
      margin: 10px;
      padding: 12px 20px;
      font-size: 16px;
      font-weight: bold;
      border: none;
```

```
      border-radius: 10px;
      cursor: pointer;
      transition: 0.3s;
    }
    #btn1 { background: #3498db; color: white; }
    #btn2 { background: #2ecc71; color: white; }
    #btn3 { background: #e74c3c; color: white; }
    button:hover {
      transform: scale(1.05);
      opacity: 0.9;
    }
    #output {
      margin-top: 20px;
      font-size: 18px;
      color: #34495e;
    }
  </style>
</head>
<body>
  <div class="container">
    <h1>Button Click Events</h1>
    <button id="btn1">Say Hello</button>
    <button id="btn2">Change Background</button>
    <button id="btn3">Show Date</button>
    <div id="output"></div>
  </div>
  <script>
    const output = document.getElementById("output");
    document.getElementById("btn1").addEventListener("click", () => {
      output.innerText = "👋 Hello, welcome to JavaScript Click Events!";
```

```
  });
  document.getElementById("btn2").addEventListener("click", () => {
    document.body.style.background = "linear-gradient(135deg, #6a11cb, #2575fc)";
    output.innerText = " 🎨 Background changed!";
  });
  document.getElementById("btn3").addEventListener("click", () => {
    output.innerText = " 📅 Current Date & Time: " + new Date().toLocaleString();
  });
 </script>
</body>
</html>
```

Output

- Clicking **Say Hello** → Displays *"Hello, welcome to JavaScript Click Events!"*
- Clicking **Change Background** → Changes background gradient.
- Clicking **Show Date** → Displays the current date and time



1(e) JavaScript Program with Three Types of Functions

Aim

To demonstrate different types of functions in JavaScript:

1. Function Declaration
2. Function Expression (Definition)

3. Arrow Function

Concepts Involved

- **Function Declaration** → Standard way to define reusable blocks of code.
- **Function Expression** → Assigning a function to a variable.
- **Arrow Function** → Concise syntax introduced in ES6, often used in callbacks.
- **Reusability** → Functions avoid repetition in programs.

Code

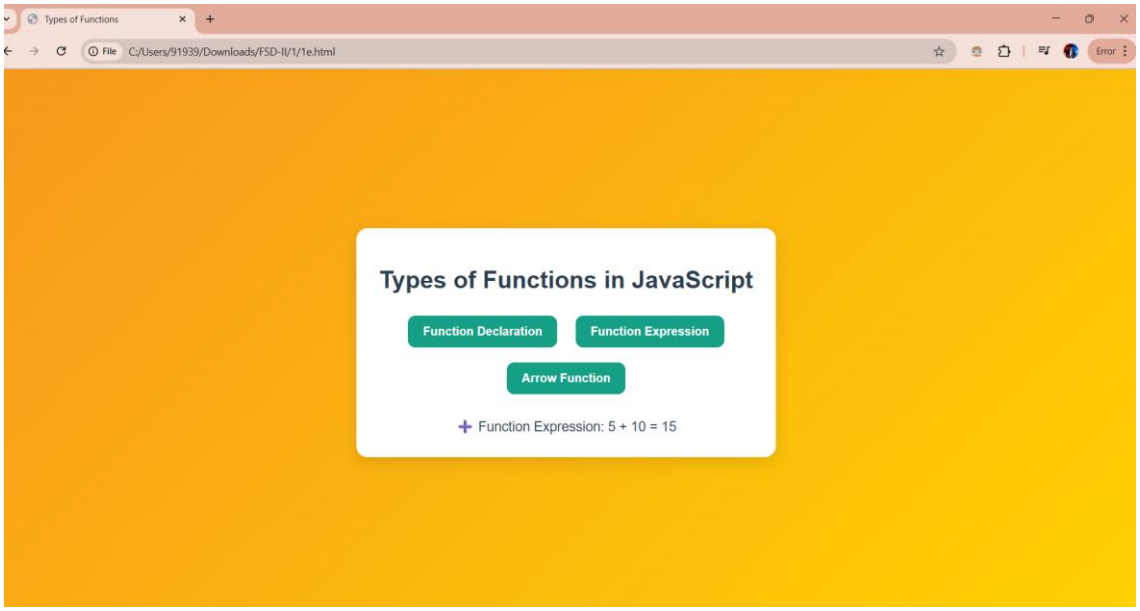
```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Types of Functions</title>
  <style>
    body {
      font-family: Arial, sans-serif;
      background: linear-gradient(135deg, #f7971e, #ffd200);
      display: flex;
      justify-content: center;
      align-items: center;
      height: 100vh;
      margin: 0;
    }
    .container {
      background: white;
      padding: 30px;
      border-radius: 15px;
      box-shadow: 0 8px 20px rgba(0,0,0,0.1);
      text-align: center;
      max-width: 500px;
    }
```

```
    h1 {
      margin-bottom: 20px;
      color: #2c3e50;
    }
    button {
      margin: 10px;
      padding: 12px 20px;
      font-size: 16px;
      font-weight: bold;
      background: #16a085;
      color: white;
      border: none;
      border-radius: 10px;
      cursor: pointer;
      transition: 0.3s;
    }
    button:hover {
      background: #138d75;
      transform: scale(1.05);
    }
    #result {
      margin-top: 20px;
      font-size: 18px;
      color: #34495e;
    }
  </style>
</head>
<body>
  <div class="container">
    <h1>Types of Functions in JavaScript</h1>
```

```
    <button onclick="sayHello()">Function Declaration</button>
    <button onclick="showSum()">Function Expression</button>
    <button onclick="arrowFunc()">Arrow Function</button>
    <div id="result"></div>
  </div>

  <script>
    const result = document.getElementById("result");
    // 1. Function Declaration
    function sayHello() {
      result.innerText = "👋 Hello! This is a Function Declaration.";
    }
    // 2. Function Expression
    const showSum = function() {
      let a = 5, b = 10;
      result.innerText = `➕ Function Expression: ${a} + ${b} = ${a+b}`;
    };
    // 3. Arrow Function
    const arrowFunc = () => {
      result.innerText = "⚡ This is an Arrow Function (ES6).";
    };
  </script>
</body>
</html>
```

Output

- Clicking **Function Declaration** → Shows *"Hello! This is a Function Declaration."*
- Clicking **Function Expression** → Shows *"5 + 10 = 15"*.
- Clicking **Arrow Function** → Shows *"This is an Arrow Function."*



2. Basics of React.js

a. Write a React program to implement a counter button using React class components
b. Write a React program to implement a counter button using React functional components
c. Write a React program to handle the button click events in functional component
d. Write a React program to conditionally render a component in the browser
e. Write a React program to display text using string literals

⚡ **Installing React.js on Linux**

1. Install Node.js & npm

React requires Node.js (JavaScript runtime) and npm (Node Package Manager).

sudo apt update

sudo apt install nodejs npm -y

✅ **Check versions:**

node -v

npm -v

(Recommended: Node.js ≥ 16, npm ≥ 8)

2. Install npx (if not already included with npm)

npx is needed to create React apps.

sudo npm install -g npx

3. Create a New React Project

Use create-react-app (official React scaffolding tool):

npx create-react-app myapp

This creates a folder myapp/ with everything needed.

4. Move into Project Folder

cd myapp

5. Start the Development Server

npm start

✅ **Now open http://localhost:3000/ in your browser → You'll see the React default page.**

6. Project Structure

Important files we will use:

- **src/App.js → main component**
- **src/index.js → entry point**
- **public/index.html → base HTML page**

We will write all our programs in App.js.

2(a) React Program – Counter Button using Class Components

1. Aim

To implement a **Counter Button** in React using **Class Components** and state management.

2. Description

This program demonstrates how to use **class-based components** in React.

- State (this.state) is used to store the counter value.

- Methods (this.setState) update the state.
- Buttons allow **increment, decrement, and reset** functionality.
- Inline CSS is used for styling to make the UI modern.

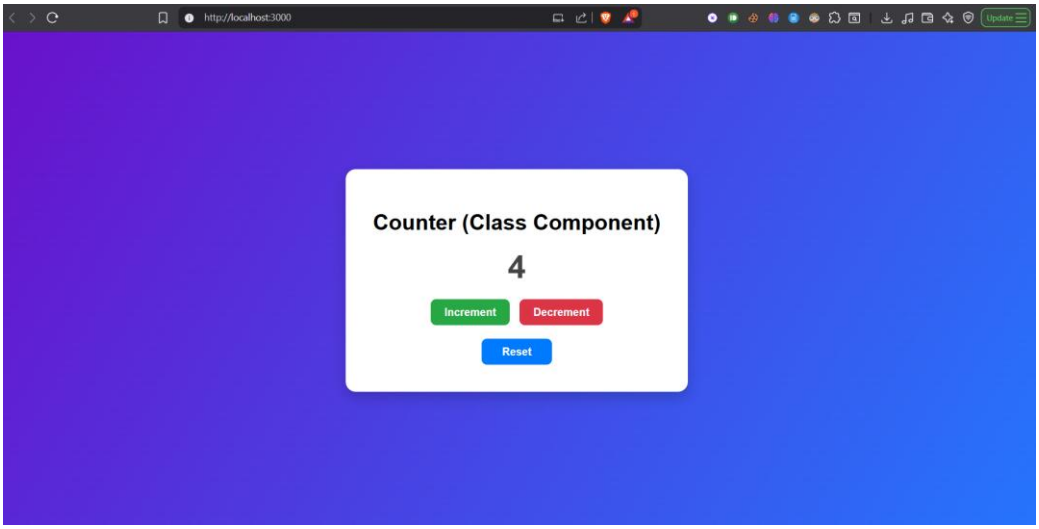
3. Source Code (App.js)

```
import React, { Component } from "react";

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

  render() {
    return (
      <div style={{
        display: "flex", justifyContent: "center", alignItems: "center",
        height: "100vh", background: "linear-gradient(120deg,#6a11cb,#2575fc)",
        fontFamily: "Arial"
      }}>
        <div style={{
          background: "#fff", padding: "40px", borderRadius: "15px",
          boxShadow: "0 8px 20px rgba(0,0,0,0.2)", textAlign: "center"
        }}>
          <h1 style={{ marginBottom: "20px" }}>Counter (Class Component)</h1>
          <h2 style={{ fontSize: "3rem", margin: "20px 0", color: "#444" }}>
            {this.state.count}
          </h2>

          <div style={{ display: "flex", justifyContent: "center", gap: "15px" }}>
```

```
          <button style={btnStyle("#28a745")}
            onClick={() => this.setState({ count: this.state.count + 1 })}>
            Increment
          </button>
          <button style={btnStyle("#dc3545")}
            onClick={() => this.setState({ count: this.state.count - 1 })}>
            Decrement
          </button>
        </div>

        <button style={{ ...btnStyle("#007bff"), marginTop: "20px", padding: "10px 30px" }}
          onClick={() => this.setState({ count: 0 })}>
          Reset
        </button>
      </div>
    </div>
  );
 }
}

function btnStyle(color) {
 return {
  padding: "10px 20px", fontSize: "1rem", background: color, color: "#fff",
  border: "none", borderRadius: "8px", cursor: "pointer",
  transition: "0.3s", fontWeight: "bold"
 };
}

export default App;
```

4. Output

- A styled counter app appears with:
    - Title: *Counter (Class Component)*
    - Large counter value in the center
    - Buttons: **Increment (Green)**, **Decrement (Red)**, and **Reset (Blue)**
- Counter updates dynamically when buttons are clicked.



2(b) React Program – Counter Button using Functional Components

1. Aim

To implement a **Counter Button** in React using **Functional Components** with the **useState Hook**.

2. Description

This program demonstrates how to:

- Use **React functional components**.
- Manage state using the **useState hook**.
- Create **Increment, Decrement, and Reset** functionality.
- Apply **modern styling** with gradients and shadows for better UI.

3. Source Code (App.js)

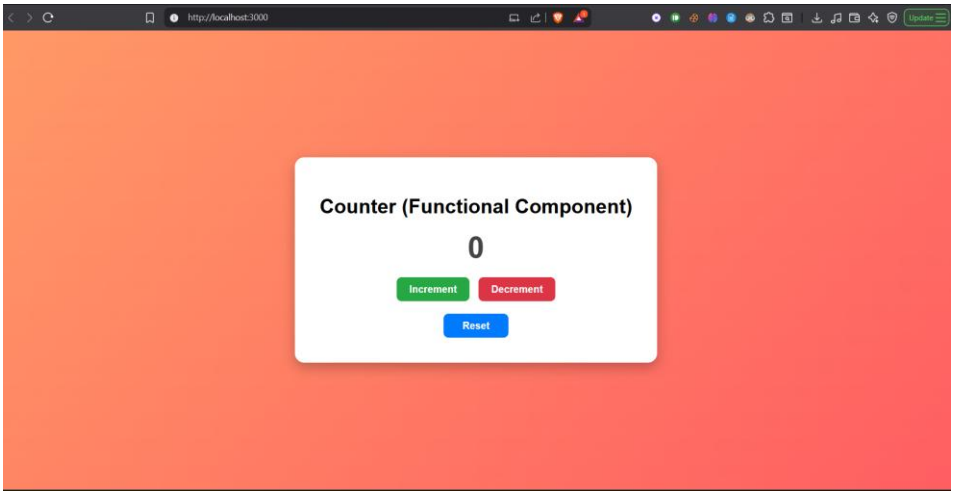
```
import React, { useState } from "react";

function App() {
  const [count, setCount] = useState(0);
  return (
```

```
    <div style={{
      display: "flex", justifyContent: "center", alignItems: "center",
      height: "100vh", background: "linear-gradient(135deg,#ff9966,#ff5e62)",
      fontFamily: "Arial"
    }}>
      <div style={{
        background: "#fff", padding: "40px", borderRadius: "15px",
        boxShadow: "0 8px 20px rgba(0,0,0,0.25)", textAlign: "center"
      }}>
        <h1 style={{ marginBottom: "20px" }}>Counter (Functional Component)</h1>
        <h2 style={{ fontSize: "3rem", margin: "20px 0", color: "#444" }}>
          {count}
        </h2>
        <div style={{ display: "flex", justifyContent: "center", gap: "15px" }}>
          <button style={btnStyle("#28a745")} onClick={() => setCount(count + 1)}>
            Increment
          </button>
          <button style={btnStyle("#dc3545")} onClick={() => setCount(count - 1)}>
            Decrement
          </button>
        </div>
        <button style={{ ...btnStyle("#007bff"), marginTop: "20px", padding: "10px 30px" }}
          onClick={() => setCount(0)}>
          Reset
        </button>
      </div>
    </div>
  );
}
```

```
function btnStyle(color) {
  return {
    padding: "10px 20px", fontSize: "1rem", background: color, color: "#fff",
    border: "none", borderRadius: "8px", cursor: "pointer",
    transition: "0.3s", fontWeight: "bold"
  };
}
export default App;
```

4. Output

- A **modern counter app** with:
  - Title: *Counter (Functional Component)*
  - A large number showing the counter value.
  - Buttons: **Increment (Green)**, **Decrement (Red)**, and **Reset (Blue)**.
- State updates instantly when buttons are clicked



2(c) React Program – Handling Button Click Events in Functional Component

1. Aim

To demonstrate how to **handle button click events** in a React functional component.

2. Description

This program shows:

- How to define **event handler functions** in a functional component.
- How to **bind button clicks** with those handlers.
- Displaying messages dynamically based on button click.
- Using **React useState hook** for updating UI on events.

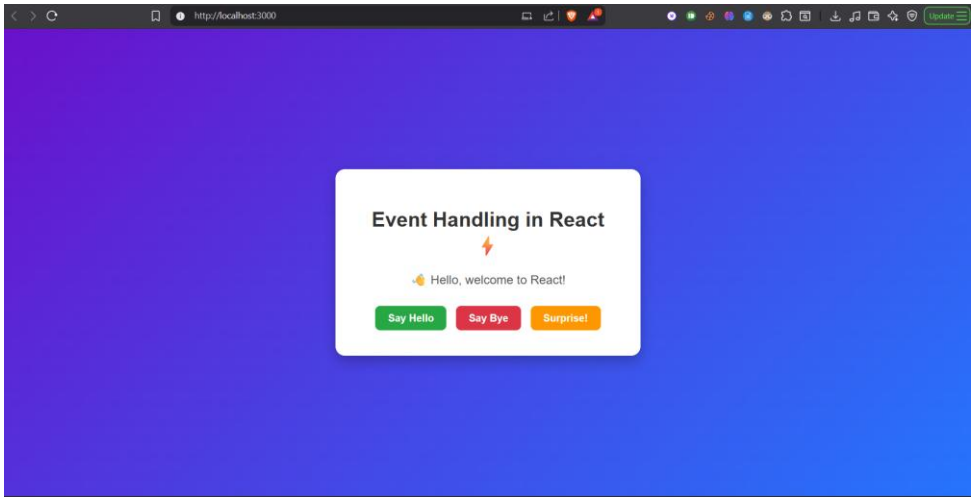
3. Source Code (App.js)

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

function App() {
  const [message, setMessage] = useState("Click a button to see the magic!");

  const handleHello = () => setMessage("👋 Hello, welcome to React!");
  const handleBye = () => setMessage("👋 Goodbye, see you soon!");
  const handleSurprise = () => setMessage("🎉 Surprise! You clicked the button!");

  return (
    <div style={{
      display: "flex", justifyContent: "center", alignItems: "center",
      height: "100vh", background: "linear-gradient(135deg,#6a11cb,#2575fc)",
      fontFamily: "Arial"
    }}>
      <div style={{
        background: "#fff", padding: "40px", borderRadius: "15px",
        boxShadow: "0 8px 20px rgba(0,0,0,0.25)", textAlign: "center",
        width: "400px"
      }}>
        <h1 style={{ marginBottom: "20px", color: "#333" }}>
          Event Handling in React ⚡
        </h1>
        <p style={{ marginBottom: "30px", fontSize: "1.2rem", color: "#555" }}>
```

```
        {message}
      </p>

      <div style={{ display: "flex", justifyContent: "center", gap: "15px", flexWrap: "wrap"
}}>
        <button style={btnStyle("#28a745")} onClick={handleHello}>Say Hello</button>
        <button style={btnStyle("#dc3545")} onClick={handleBye}>Say Bye</button>
        <button style={btnStyle("#ff9800")} onClick={handleSurprise}>Surprise!</button>
      </div>
    </div>
  </div>
 );
}

function btnStyle(color) {
 return {
  padding: "10px 20px", fontSize: "1rem", background: color, color: "#fff",
  border: "none", borderRadius: "8px", cursor: "pointer",
  transition: "0.3s", fontWeight: "bold"
 };
}

export default App;
```

4. Output

- The app displays:
    - Title: *Event Handling in React* ⚡
    - A message: *"Click a button to see the magic!"* initially.
    - Three buttons: **Say Hello (Green)**, **Say Bye (Red)**, **Surprise (Orange)**.
- On clicking each button, the message updates dynamically.



2(d) React Program – Conditionally Rendering a Component

1. Aim

To demonstrate how to **conditionally render components** in React using the useState hook.

2. Description

This program shows:

- How to **toggle between two components** dynamically.
- Using **conditional rendering (? : operator)** in React.
- Updating UI based on user interaction.

3. Source Code (App.js)

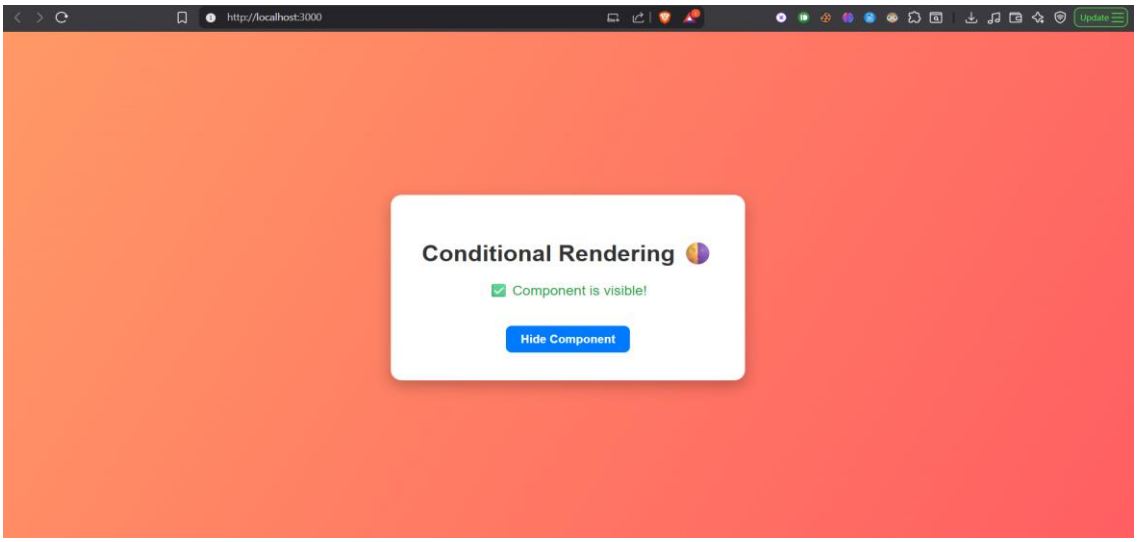
```
import React, { useState } from "react";

function App() {
  const [show, setShow] = useState(true);

  return (
    <div style={{
      display: "flex", justifyContent: "center", alignItems: "center",
      height: "100vh", background: "linear-gradient(120deg,#ff9966,#ff5e62)",
      fontFamily: "Arial"
```

```
    }}>
      <div style={{
        background: "#fff", padding: "40px", borderRadius: "15px",
        boxShadow: "0 8px 20px rgba(0,0,0,0.25)", textAlign: "center",
        width: "400px"
      }}>
        <h1 style={{ marginBottom: "20px", color: "#333" }}>
          Conditional Rendering 🌓
        </h1>

        {show ? (
          <p style={{ fontSize: "1.2rem", color: "#28a745" }}>
            ✅ Component is visible!
          </p>
        ) : (
          <p style={{ fontSize: "1.2rem", color: "#dc3545" }}>
            ❌ Component is hidden!
          </p>
        )}

        <button
          onClick={() => setShow(!show)}
          style={{
            marginTop: "20px", padding: "10px 20px", fontSize: "1rem",
            background: "#007bff", color: "#fff", border: "none",
            borderRadius: "8px", cursor: "pointer", fontWeight: "bold"
          }}
        >
          {show ? "Hide Component" : "Show Component"}
        </button>
```

```
    </div>
  </div>
 );
}

export default App;
```

4. Output

- The page displays:
  - Title: *Conditional Rendering* 🌓
  - A message showing either **"✅ Component is visible!"** or **"❌ Component is hidden!"**
  - A **toggle button** that switches between showing and hiding the component.



2(e) React Program – Displaying Text Using String Literals

1. Aim

To demonstrate how to display **dynamic text in React** using **JavaScript template literals (``)**.

2. Description

This program shows:

- Using **string interpolation** inside JSX.
- How to embed variables and expressions into a string using template literals.
- A simple user-friendly UI to demonstrate.

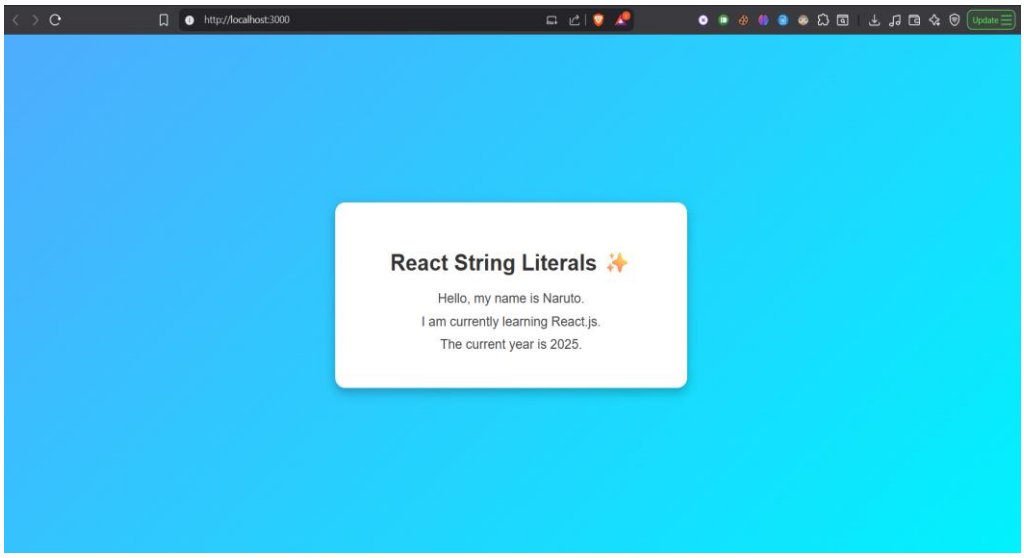
3. Source Code (App.js)

```
import React from "react";

function App() {
  const name = "Naruto";
  const course = "React.js";
  const year = new Date().getFullYear();

  return (
    <div style={{
      display: "flex", justifyContent: "center", alignItems: "center",
      height: "100vh", background: "linear-gradient(135deg,#4facfe,#00f2fe)",
      fontFamily: "Arial"
    }}>
      <div style={{
        background: "#fff", padding: "40px", borderRadius: "15px",
        boxShadow: "0 6px 15px rgba(0,0,0,0.25)", textAlign: "center",
        width: "450px"
      }}>
        <h1 style={{ color: "#333", marginBottom: "20px" }}>
          React String Literals ✨
        </h1>

        <p style={{ fontSize: "1.2rem", margin: "10px 0", color: "#444" }}>
          {`Hello, my name is ${name}.`}
```

```
      </p>

      <p style={{ fontSize: "1.2rem", margin: "10px 0", color: "#444" }}>
        {`I am currently learning ${course}.`}
      </p>

      <p style={{ fontSize: "1.2rem", margin: "10px 0", color: "#444" }}>
        {`The current year is ${year}.`}
      </p>
    </div>
  </div>
 );
}

export default App;
```

4. Output

The page will display:



3. Important Concepts of React.js

a. Write a React program to implement a counter button using React useState hook
b. Write a React program to fetch the data from an API using React useEffect hook
c. Write a React program with two React components sharing data using Props
d. Write a React program to implement forms in React
e. Write a React program to implement the iterative rendering using map() function

3(a) React Program – Counter Button using useState Hook

1. Aim

To implement a **counter button** in React using the **useState hook**.

2. Description

- The useState hook lets us add state variables in functional components.
- Here, we will use it to store and update the **count value**.
- Clicking the button increases the counter.

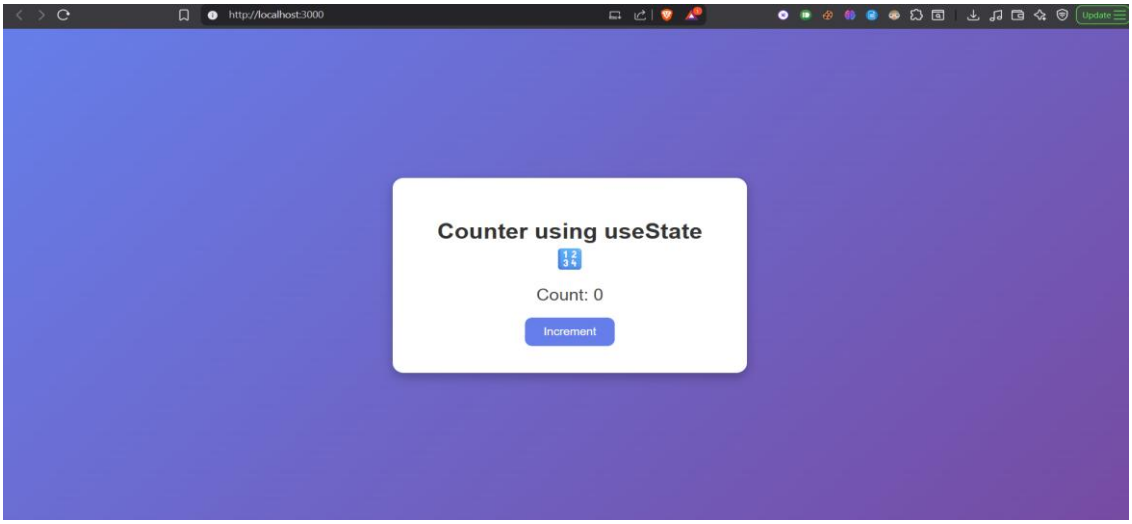
3. Source Code (App.js)

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


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


  return (
    <div style={{
      display: "flex", justifyContent: "center", alignItems: "center",
      height: "100vh", background: "linear-gradient(135deg,#667eea,#764ba2)",
      fontFamily: "Arial"
    }}>
      <div style={{
        background: "#fff", padding: "40px", borderRadius: "15px",
        boxShadow: "0 6px 15px rgba(0,0,0,0.25)", textAlign: "center",
```

```
        width: "400px"
      }}>
        <h1 style={{ color: "#333" }}>Counter using useState 🔢</h1>
        <p style={{ fontSize: "1.5rem", margin: "20px 0", color: "#444" }}>
          Count: {count}
        </p>
        <button
          onClick={() => setCount(count + 1)}
          style={{
            padding: "12px 25px", border: "none", borderRadius: "10px",
            background: "#667eea", color: "#fff", fontSize: "1rem",
            cursor: "pointer", transition: "0.3s"
          }}
        >
          Increment
        </button>
      </div>
    </div>
  );
}

export default App;
```

4. Output

- Initially displays: **Count: 0**.
- Each time you click the button, the count increases by **1**.



3(b) React Program – Fetch Data from API using useEffect

1. Aim

To fetch and display data from an external API in a React component using the **useEffect** hook.

2. Description

- The useEffect hook is used to perform side effects (like fetching data).
- We'll fetch user data from **JSONPlaceholder API** (https://jsonplaceholder.typicode.com/users).
- The fetched data will be stored in state and rendered dynamically.

3. Source Code (App.js)

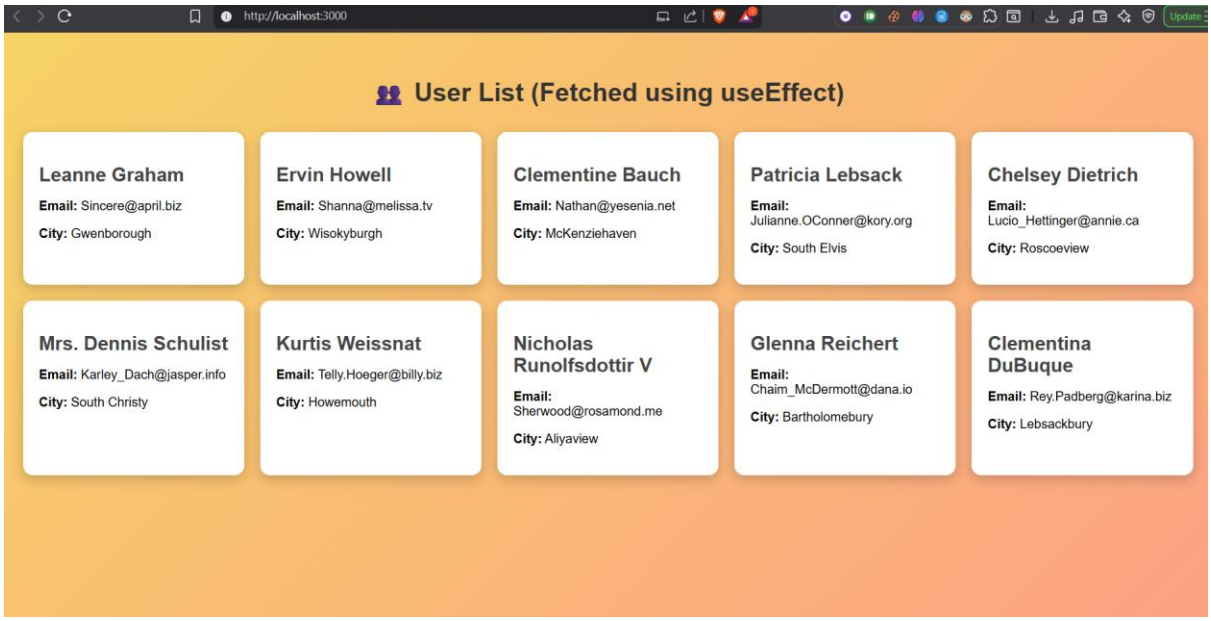
```
import React, { useState, useEffect } from "react";


function App() {
  const [users, setUsers] = useState([]);


  useEffect(() => {
    fetch("https://jsonplaceholder.typicode.com/users")
      .then(res => res.json())
      .then(data => setUsers(data))
      .catch(err => console.error("Error fetching users:", err));
  }, []);
```

```
  return (
    <div style={{
      padding: "30px",
      fontFamily: "Arial",
      background: "linear-gradient(135deg, #f6d365, #fda085)",
      minHeight: "100vh"
    }}>
      <h1 style={{ textAlign: "center", color: "#333" }}>👥 User List (Fetched using useEffect)</h1>

      <div style={{
        display: "grid", gridTemplateColumns: "repeat(auto-fit, minmax(250px, 1fr))",
        gap: "20px", marginTop: "30px"
      }}>
        {users.map(user => (
          <div key={user.id} style={{
            background: "#fff", padding: "20px", borderRadius: "12px",
            boxShadow: "0 5px 15px rgba(0,0,0,0.2)"
          }}>
            <h2 style={{ marginBottom: "10px", color: "#444" }}>{user.name}</h2>
            <p><b>Email:</b> {user.email}</p>
            <p><b>City:</b> {user.address.city}</p>
          </div>
        ))}
      </div>
    </div>
  );
}
```

export default App;

4. Output

- Displays a **list of users** fetched from the API.
- Each user's **name, email, and city** is shown in a styled card layout.



3(c) React Program – Sharing Data using Props

1. Aim

To demonstrate **data sharing between components** in React using **props**.

2. Description

- **Props** (short for *properties*) are used to pass data from a parent component to a child component.
- Here, the parent (App) will pass a **student object** to a child component (StudentCard).
- The child component will render the data it receives via props.

3. Source Code (App.js)

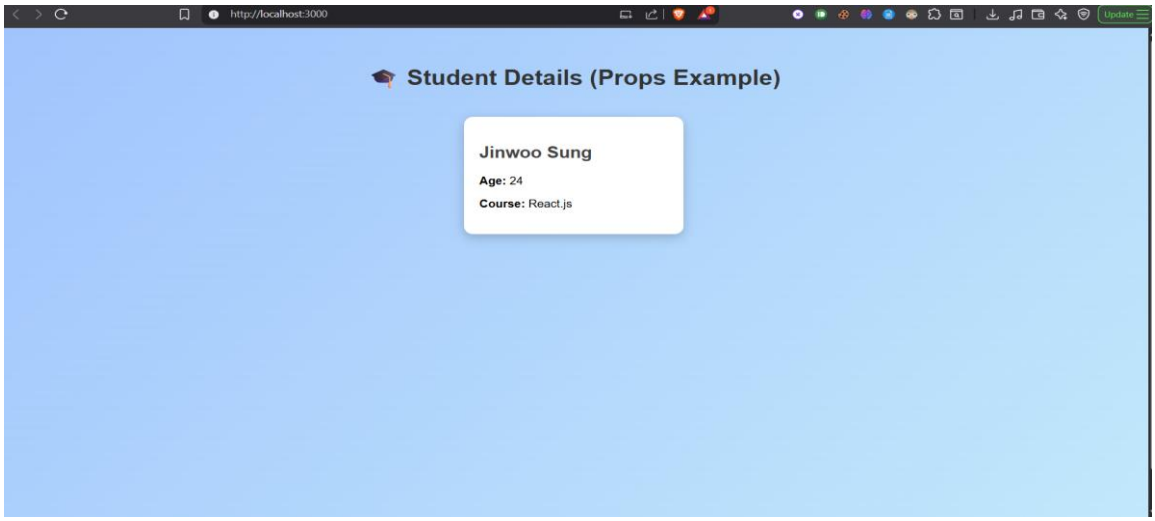
```
import React from "react";


// Child Component
function StudentCard({ name, age, course }) {
  return (
    <div style={{
```

```
      background: "#fff",
      borderRadius: "12px",
      padding: "20px",
      margin: "10px",
      width: "250px",
      boxShadow: "0 5px 15px rgba(0,0,0,0.2)"
    }}>
      <h2 style={{ color: "#444" }}>{name}</h2>
      <p><b>Age:</b> {age}</p>
      <p><b>Course:</b> {course}</p>
    </div>
  );
}

// Parent Component
function App() {
  const student = { name: "Jinwoo Sung", age: 24, course: "React.js" };

  return (
    <div style={{
      minHeight: "100vh",
      padding: "30px",
      fontFamily: "Arial",
      background: "linear-gradient(135deg, #a1c4fd, #c2e9fb)"
    }}>
      <h1 style={{ textAlign: "center", color: "#333" }}>🎓 Student Details (Props
Example)</h1>

      <div style={{
        display: "flex",
```

```
      justifyContent: "center",
      marginTop: "30px"
    }}>
      <StudentCard name={student.name} age={student.age} course={student.course} />
    </div>
  </div>
 );
}
export default App;
```

4. Output

- A **student card** is displayed with **name, age, and course**.
- The data comes from the **parent component** and is passed to the **child component** via props.



3(d) React Program – Implementing Forms

1. Aim

To implement a **form in React** and handle user input dynamically.

2. Description

- React handles forms using **state hooks** (useState).
- Each input field updates state values in real-time.
- On **form submission**, the entered data is displayed.

3. Source Code (App.js)

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

```
function App() {
  // State variables for form fields
  const [name, setName] = useState("");
  const [email, setEmail] = useState("");
  const [submitted, setSubmitted] = useState(false);

  // Handle form submission
  const handleSubmit = (e) => {
    e.preventDefault();
    setSubmitted(true);
  };
  return (
    <div style={{
      minHeight: "100vh",
      padding: "40px",
      background: "linear-gradient(135deg, #89f7fe, #66a6ff)",
      fontFamily: "Arial"
    }}>
      <h1 style={{ textAlign: "center", color: "#222" }}>📋 React Form Example</h1>

      <form
        onSubmit={handleSubmit}
        style={{
          background: "#fff",
          padding: "30px",
          borderRadius: "12px",
          width: "350px",
          margin: "30px auto",
          boxShadow: "0 6px 20px rgba(0,0,0,0.2)"
```

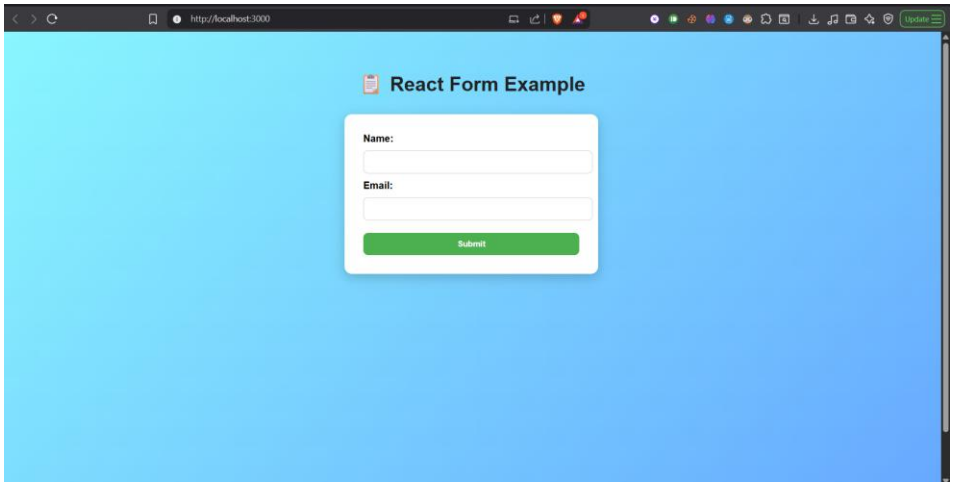
```
    }}
  >
    <label style={{ fontWeight: "bold" }}>Name:</label>
    <input
      type="text"
      value={name}
      onChange={(e) => setName(e.target.value)}
      required
      style={{
        width: "100%",
        padding: "10px",
        margin: "10px 0",
        borderRadius: "8px",
        border: "1px solid #ccc"
      }}
    /
    <label style={{ fontWeight: "bold" }}>Email:</label>
    <input
      type="email"
      value={email}
      onChange={(e) => setEmail(e.target.value)}
      required
      style={{
        width: "100%",
        padding: "10px",
        margin: "10px 0",
        borderRadius: "8px",
        border: "1px solid #ccc"
      }}
    />
```

```
        <button
          type="submit"
          style={{
            background: "#4CAF50",
            color: "#fff",
            padding: "10px 15px",
            border: "none",
            borderRadius: "8px",
            cursor: "pointer",
            width: "100%",
            fontWeight: "bold",
            marginTop: "10px"
          }}
        >
          Submit
        </button>
      </form>

      {submitted && (
        <div style={{
          textAlign: "center",
          marginTop: "20px",
          background: "#fff",
          padding: "15px",
          borderRadius: "10px",
          width: "350px",
          margin: "auto",
          boxShadow: "0 4px 15px rgba(0,0,0,0.1)"
        }}>
```

```
        <h2>✅ Submitted Data</h2>
        <p><b>Name:</b> {name}</p>
        <p><b>Email:</b> {email}</p>
      </div>
    )}
  </div>
 );
}
export default App;
```

4. Output

- A **form** with Name and Email fields.
- On clicking **Submit**, the entered data is shown below the form.



3(e) React Program – Iterative Rendering using map()

1. Aim

To implement **iterative rendering** in React using the map() function.

2. Description

- In React, the map() function allows rendering **lists of elements dynamically**.
- It improves **scalability** as we can render multiple items from an array easily.
- Each list item requires a unique **key** for React's reconciliation.

3. Source Code (App.js)

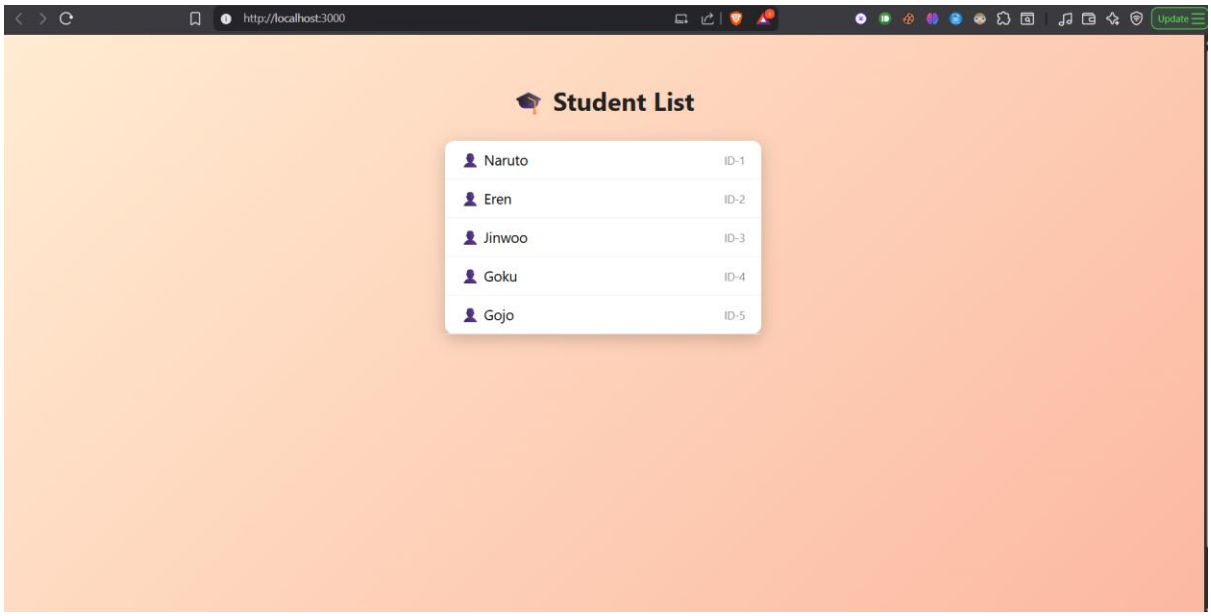
```
import React from "react";
```

```
function App() {
  // Sample list of students
  const students = ["Naruto", "Eren", "Jinwoo", "Goku", "Gojo"];
x
  return (
    <div style={{
      minHeight: "100vh",
      background: "linear-gradient(135deg, #ffecd2, #fcb69f)",
      fontFamily: "Segoe UI, sans-serif",
      padding: "40px"
    }}>
      <h1 style={{ textAlign: "center", color: "#222" }}>🎓 Student List</h1>

      <ul style={{
        listStyle: "none",
        padding: 0,
        maxWidth: "400px",
        margin: "30px auto",
        background: "#fff",
        borderRadius: "12px",
        boxShadow: "0 6px 20px rgba(0,0,0,0.2)"
      }}>
        {students.map((student, index) => (
          <li
            key={index}
            style={{
              padding: "12px 20px",
              borderBottom: "1px solid #eee",
              fontSize: "18px",
```

```
            display: "flex",
            justifyContent: "space-between",
            alignItems: "center"
          }}
        >
          <span>👤 {student}</span>
          <span style={{ color: "#888", fontSize: "14px" }}>ID-{index + 1}</span>
        </li>
      ))}
    </ul>
  </div>
 );
}

export default App;
```

4. Output

- A **styled list of students** is displayed.
- Each student's name is shown along with an automatically generated ID.



4. Introduction to Git and GitHub

a. Setup

- Install Git on local machine
- Configure Git (user name, email)
- Create GitHub account and generate a personal access token

b. Basic Git Workflow

- Create a local repository using git init
- Create and add files → git add .
- Commit files → git commit -m "Initial commit"
- Connect to GitHub remote → git remote add origin <repo_url>
- Push to GitHub → git push -u origin main

c. Branching and Collaboration

- Create a branch → git checkout -b feature1
- Merge branch to main → git merge feature1
- Resolve merge conflicts (guided)

4. Introduction to Git and GitHub

4.a Setup

Aim:

To install Git on a Linux system, configure user information, and connect Git to GitHub using a personal access token (PAT).

Description:

Git is a version control system that tracks changes in source code, enabling collaboration among developers. GitHub is a cloud-based platform for hosting Git repositories. Setting up Git involves installing it, configuring global user details, and connecting to GitHub using a personal access token.

Commands / Steps:

1.Install Git on Linux

sudo apt update # Update package lists

sudo apt install git -y # Install Git

git --version # Verify installation

2. Configure Git (username and email)

git config --global user.name "Your Name"

git config --global user.email "youremail@example.com"

git config --list # Check configuration

3.Create GitHub account and generate Personal Access Token

- Go to GitHub and create an account if you don't have one.
- Generate a PAT for authentication:
  Settings → Developer Settings → Personal access tokens → Tokens (classic) → Generate new token → select scopes → copy token
  (This token will be used instead of password for HTTPS Git operations.)

Output / Expected Result:

- Git installed successfully (git --version shows version)
- Username and email configured globally
- Personal Access Token ready for authenticating GitHub operations

4.b Basic Git Workflow

Aim:

To create a local repository, track changes, commit files, and push them to GitHub.

Description:

The basic Git workflow involves initializing a repository, staging files, committing changes, connecting to a remote repository, and pushing the commits. This ensures the project is version-controlled and backed up on GitHub.

Commands / Steps:

1. **Navigate to your project directory**

cd ~/projects/myproject # Replace with your project path

2. **Initialize Git repository**

git init

(Creates a .git folder for version control)

3. **Add files to staging area**

git add .

(Stages all files in the project folder)

4. **Commit changes**

git commit -m "Initial commit"

(Saves the staged changes with a descriptive message)

5. **Connect to GitHub remote repository**

git remote add origin https://github.com/username/repo.git

(Replace username/repo.git with your GitHub repository URL)

6. **Push local commits to GitHub**

git push -u origin main

(Pushes local main branch to remote repository and sets upstream)

Output / Expected Result:

- Local repository initialized (.git/ folder created)
- Files committed to local repository
- Files pushed to GitHub and visible in the repository online

4.c Branching and Collaboration

Aim:

To create a branch, implement features in parallel, merge it with the main branch, and resolve conflicts if any.

Description:

Branching allows multiple developers to work on different features without affecting the main branch. After development, branches can be merged. Merge conflicts can occur if changes overlap, and Linux provides simple commands to resolve them.

Commands / Steps:

1. **Create a new branch**

git checkout -b feature1

(Creates a branch called feature1 and switches to it)

2. **Make changes and commit**

Edit files using your preferred editor (nano, vim, VS Code)

nano index.html

Stage and commit changes

git add index.html

git commit -m "Added feature1 changes"

3. **Switch back to main branch**

git checkout main

4. **Merge feature branch into main**

git merge feature1

5. **Resolve merge conflicts (if any)**

- Open the conflicted file, e.g.:

nano index.html

- Conflicts are marked like this:

<<<<<<< HEAD

Main branch content

=======

Feature branch content

>>>>>>> feature1

- Keep the desired content, delete conflict markers, save file
- Mark conflict resolved:

git add index.html

git commit -m "Resolved merge conflict in index.html"

6. **Push merged changes to GitHub**

git push origin main

Output / Expected Result:

- Branch feature1 created and changes committed
- Branch successfully merged into main
- Merge conflicts resolved if present
- Updated main branch visible on GitHub

5. Upload React Project to GitHub

- Create a new React app using npx create-react-app myapp
- Initialize a git repo and push to GitHub
- Use .gitignore to exclude node_modules
- Create multiple branches: feature/navbar, feature/form
- Practice merge and pull requests (can use GitHub GUI)

5. Upload React Project to GitHub

5.a Create a New React App

Aim:

To create a new React application using npx create-react-app on a Linux system.

Description:

React is a JavaScript library for building user interfaces. The create-react-app command sets up a ready-to-use React project with necessary folder structure and configuration.

Commands / Steps:

Navigate to projects directory

cd ~/projects

Create a new React app named "myapp"

npx create-react-app myapp

Navigate into the project folder

cd myapp

List the project structure

ls -l

Output / Expected Result:

- A folder named myapp is created
- Subfolders src, public, node_modules and files package.json, package-lock.json are present
- Ready-to-run React project

5.b Initialize Git Repository and Push to GitHub

Aim:

To initialize Git, commit project files, and push the React project to GitHub.

Description:

React projects include a large node_modules folder that should not be uploaded. A Git repository is initialized locally and connected to a remote GitHub repository for version control.

Commands / Steps:

Initialize Git repository

git init

Add all files to staging area

git add .

Commit files

git commit -m "Initial React app commit"

Connect local repo to GitHub

git remote add origin https://github.com/username/myapp.git

Push to GitHub

git push -u origin main

Output / Expected Result:

- .git folder created for version control
- Files committed locally
- Project visible on GitHub

5.c Use .gitignore to Exclude node_modules

Aim:

To prevent the node_modules folder from being tracked and pushed to GitHub.

Description:

.gitignore specifies files and folders Git should ignore. React projects already include node_modules/ in .gitignore, but this step ensures it is properly configured.

Commands / Steps:

Check existing .gitignore

ls -a

cat .gitignore

Add node_modules to .gitignore if not present

echo "node_modules/" >> .gitignore

Stage and commit .gitignore

git add .gitignore

git commit -m "Add .gitignore to exclude node_modules"

Verify node_modules is ignored

git status

Output / Expected Result:

- .gitignore contains node_modules/
- node_modules folder is ignored by Git
- Keeps GitHub repository clean and lightweight

5.d Create Multiple Branches

Aim:

To create branches for parallel development of React components.

Description:

Branches allow development of new features independently, avoiding conflicts with the main branch. For example, creating branches for Navbar and Form components.

Commands / Steps:

Create branch for Navbar component

git checkout -b feature/navbar

Create branch for Form component

git checkout -b feature/form

Switch back to main branch

git checkout main

Output / Expected Result:

- Branches feature/navbar and feature/form created
- Main branch remains unaffected
- Can switch between branches freely

5.e Practice Merge and Pull Requests

Aim:

To merge feature branches into the main branch and practice Pull Requests on GitHub.

Description:

After feature development, branches are merged into main. Pull Requests allow code review and approval before merging, ensuring safe collaboration.

Commands / Steps (Linux Git):

Merge feature/navbar into main

git checkout main

git merge feature/navbar

Push merged changes to GitHub

git push origin main

Using GitHub GUI:

1. Open repository on GitHub
2. Click **Compare & Pull Request** for a branch
3. Review changes and click **Merge Pull Request**

4. Optionally, delete the merged branch

Output / Expected Result:

- Feature branch changes merged into main
- Pull Request reviewed and merged on GitHub
- Updated main branch reflects new features

6. Introduction to Node.js and Express.js

a. Write a program to implement the "Hello World" message in the route through the browser using Express.js
b. Write a program to develop a small website with multiple routes using Express.js
c. Write a program to print the "Hello World" in the browser console using Express.js
d. Write a program to implement CRUD operations using Express.js
e. Write a program to establish the connection between API and Database using Express + MySQL driver

Common steps:

1.Initialize Node.js project

mkdir hello-express

cd hello-express

npm init -y

2. **Install Express**

npm install express

3. **Create app.js**

6. Introduction to Node.js and Express.js

6.a "Hello World" in Browser

Aim:

To Write a program to implement the "Hello World" message in the route through the browser using Express.js

Description:

This program demonstrates a basic Express server with a single route (/). Using HTML and inline CSS, the message is styled for better visualization.

Code:

```
// app-hello.js
const express = require('express');
const app = express();
const port = 3000;

app.get('/', (req, res) => {
  res.send(`
    <h1 style="color:blue; text-align:center;">Hello World!</h1>
    <p style="font-size:16px; text-align:center;">This message is served by Express.js</p>
  `);
});

app.listen(port, () => {
  console.log(`Server running at http://localhost:${port}`);
});
```

Output:

- Browser: Blue, centered "Hello World!" message
- Terminal: Server running at http://localhost:3000

6.b Small Website with Multiple Routes

Aim:

To Write a program to develop a small website with multiple routes using Express.js

Description:

This program demonstrates routing in Express with /, /about, and /contact. Inline CSS is used for styling and navigation links allow easy access to each route.

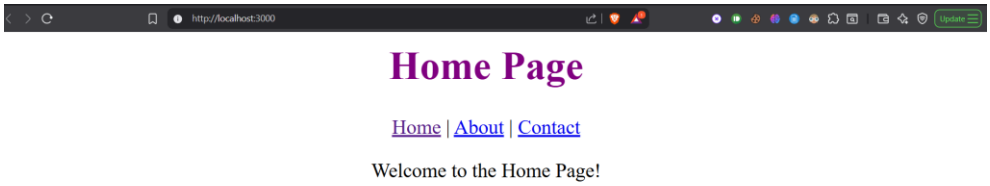
Code:

```
// app-multiroutes.js

const express = require('express');

const app = express();

const port = 3000;


function renderPage(title, content) {
   return `
      <h1 style="color:purple; text-align:center;">${title}</h1>
      <nav style="text-align:center;">
         <a href="/">Home</a> |
         <a href="/about">About</a> |
         <a href="/contact">Contact</a>
      </nav>
      <p style="text-align:center;">${content}</p>
   `;
}


app.get('/', (req, res) => res.send(renderPage('Home Page', 'Welcome to the Home Page!')));

app.get('/about', (req, res) => res.send(renderPage('About Page', 'This is the About Page.')));

app.get('/contact', (req, res) => res.send(renderPage('Contact Page', 'Contact us at
contact@example.com.')));


app.listen(port, () => console.log(`Server running at http://localhost:${port}`));
```

Output:

- Browser: Styled pages with navigation links

- Terminal: Server running at http://localhost:3000



6.c "Hello World" in Browser Console

Aim:

To print "Hello World" in the browser console using Express.js.

Description:

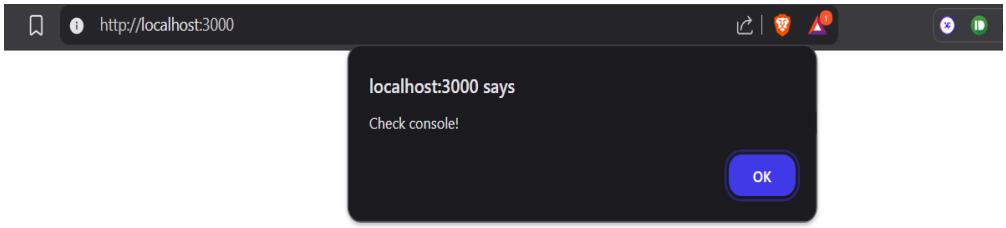
By sending a <script> tag in the response, JavaScript executes in the browser console.

Code (app.js):

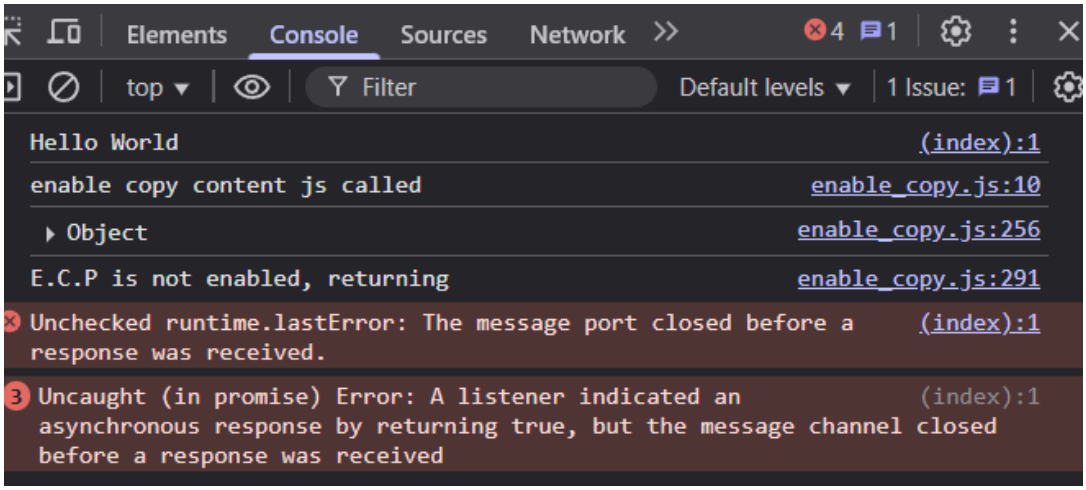
```
const express = require('express');
const app = express();
const port = 3000;

app.get('/', (req, res) => {
   res.send('<script>console.log("Hello World"); alert("Check console!");</script>');
});

app.listen(port, () => console.log(`Server running at http://localhost:${port}`));
```

Output / Expected Result:

- Browser displays an alert **"Check console!"**
- Browser console shows **Hello World**



✓ Right click on the mouse and then click on inspect option, navigate to Console section



6.d CRUD Operations Using Express.js (Interactive & Colorful Webpage)

Aim:

To write a program to implement CRUD operations using Express.js

Description:

This program demonstrates:

1. An **in-memory array** to store users.
2. A **dynamic HTML table** showing all users.
3. **Compact, color-coded forms** for Create, Update, and Delete operations.

4. **JavaScript Fetch API** to send POST, PUT, DELETE requests from the browser.
5. **Immediate reflection** of CRUD operations on the webpage without using Postman.
6. Alternating row colors and styled headings/buttons for better visual appeal.

Code:

```
// app-crud-final-colorful.js
const express = require('express');
const app = express();
const port = 3000;

app.use(express.json());

let users = [
  { name: "Tony", age: 25 },
  { name: "Pepper", age: 22 }
];

// Redirect root to /users
app.get('/', (req, res) => res.redirect('/users'));

// GET route: display users + interactive forms
app.get('/users', (req, res) => {
  let table = `<table border="1" cellpadding="10" style="margin:auto; text-align:center; border-collapse: collapse;">
    <tr style="background-color:#4CAF50; color:white;"><th>ID</th><th>Name</th><th>Age</th></tr>`;
  users.forEach((u, i) => {
    table += `<tr style="background-color:${i%2===0?'#f2f2f2':'#e6f7ff'};"><td>${i}</td><td>${u.name}</td><td>${u.age}</td></tr>`;
  });
  table += `</table>`;
```

```
table += `
<h2 style="text-align:center; color:#ff5722;">Add User</h2>
<form id="createForm" style="text-align:center;">
<input type="text" id="name" placeholder="Name" required>
<input type="number" id="age" placeholder="Age" required>
<button type="submit" style="background-color:#4CAF50;color:white;padding:5px
15px;border:none;border-radius:4px;">Add</button>
</form>

<h2 style="text-align:center; color:#3f51b5;">Update User</h2>
<form id="updateForm" style="text-align:center;">
<input type="number" id="updateId" placeholder="ID" required>
<input type="text" id="updateName" placeholder="Name" required>
<input type="number" id="updateAge" placeholder="Age" required>
<button type="submit" style="background-color:#3f51b5;color:white;padding:5px
15px;border:none;border-radius:4px;">Update</button>
</form>

<h2 style="text-align:center; color:#f44336;">Delete User</h2>
<form id="deleteForm" style="text-align:center;">
<input type="number" id="deleteId" placeholder="ID" required>
<button type="submit" style="background-color:#f44336;color:white;padding:5px
15px;border:none;border-radius:4px;">Delete</button>
</form>

<script>
document.getElementById('createForm').addEventListener('submit', async (e) => {
e.preventDefault();
await fetch('/users', {
method:'POST',
```

```
            headers:{'Content-Type':'application/json'},
            body:JSON.stringify({name: document.getElementById('name').value, age: parseInt(document.getElementById('age').value)})
        });
        location.reload();
    });

    document.getElementById('updateForm').addEventListener('submit', async (e) => {
        e.preventDefault();
        await fetch('/users/' + document.getElementById('updateId').value, {
            method:'PUT',
            headers:{'Content-Type':'application/json'},
            body:JSON.stringify({name: document.getElementById('updateName').value, age: parseInt(document.getElementById('updateAge').value)})
        });
        location.reload();
    });

    document.getElementById('deleteForm').addEventListener('submit', async (e) => {
        e.preventDefault();
        await fetch('/users/' + document.getElementById('deleteId').value, { method:'DELETE' });
        location.reload();
    });
  </script>
  `;

  res.send(`<h1 style="text-align:center; color:#009688;">User Management System</h1>${table}`);
});

// POST: Add new user
```

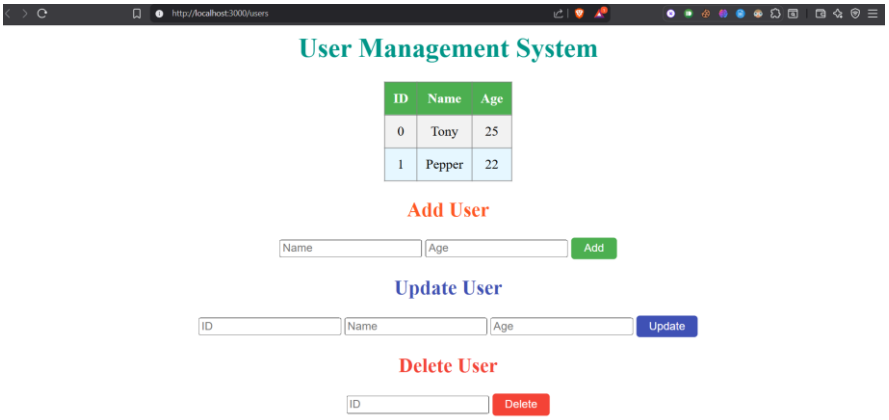
```
app.post('/users', (req, res) => { users.push(req.body); res.send({message:"User added", users}); });

// PUT: Update user
app.put('/users/:id', (req, res) => {
  const id = parseInt(req.params.id);
  if(users[id]) { users[id] = req.body; res.send({message:"User updated", users}); }
  else { res.status(404).send({message:"User not found"}); }
});

// DELETE: Remove user
app.delete('/users/:id', (req, res) => {
  const id = parseInt(req.params.id);
  if(users[id]) { users.splice(id,1); res.send({message:"User deleted", users}); }
  else { res.status(404).send({message:"User not found"}); }
});

// Start server
app.listen(port, () => console.log(`Server running at http://localhost:${port}`));
```

Output:

1. **Browser (http://localhost:3000/)**

User Table:

ID	Name	Age
0	Tony	25
1	Pepper	22



6.e Establish Connection Between API and Database Using Express + MySQL

Aim:

To connect an Express.js application to a MySQL database and perform basic **CRUD operations**, demonstrating how the API interacts with the database and returns data to the browser.

Description:

This program demonstrates:

1. Setting up a **MySQL database** and a table to store data (e.g., users).
2. Using the **mysql2 Node.js package** to connect Express.js to MySQL.
3. Implementing **API routes** to fetch and insert data into the database.
4. Returning data in **JSON format** to the browser or API clients.

Key points:

- MySQL must be installed locally.
- Node.js connects using mysql2 driver.
- The program can be extended for update and delete operations, but this example focuses on **connectivity and fetching data**.

Code:

```
// app-mysql.js

const express = require('express');

const mysql = require('mysql2');
```

```
const app = express();
const port = 3000;

app.use(express.json());

// MySQL connection
const db = mysql.createConnection({
    host: 'localhost',
    user: 'root',        // replace with your MySQL username
    password: 'password', // replace with your MySQL password
    database: 'testdb'    // database name
});

// Connect to MySQL
db.connect((err) => {
    if(err) {
        console.error('Error connecting to MySQL:', err.message);
    } else {
        console.log('Connected to MySQL database.');
    }
});

// Create table if not exists
const createTableQuery = `
CREATE TABLE IF NOT EXISTS users (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(50),
    age INT
)
`;
```

```
db.query(createTableQuery, (err) => {
  if(err) console.error('Error creating table:', err.message);
});

// GET route: fetch all users
app.get('/users', (req, res) => {
  db.query('SELECT * FROM users', (err, results) => {
    if(err) return res.status(500).send({ error: err.message });
    res.json(results);
  });
});

// POST route: add new user
app.post('/users', (req, res) => {
  const { name, age } = req.body;
  db.query('INSERT INTO users (name, age) VALUES (?, ?)', [name, age], (err, result) => {
    if(err) return res.status(500).send({ error: err.message });
    res.send({ message: 'User added', id: result.insertId });
  });
});

// Start server
app.listen(port, () => console.log(`Server running at http://localhost:${port}`));
```

Setup Instructions:

1. Install MySQL on your machine (Linux/Windows).
2. Create database testdb:

```
CREATE DATABASE testdb;
```

3. Ensure MySQL username/password match your connection settings.
4. Install dependencies:

npm install express mysql2

5. Run the server:

node app-mysql.js

6. Open browser or Postman:
    - GET: http://localhost:3000/users → List of users
    - POST: http://localhost:3000/users with JSON body:

```
{
  "name": "Alice",
  "age": 28
}
```

Output:

Terminal:

Connected to MySQL database.

Server running at http://localhost:3000

Browser or API Client (GET /users):

```
[
  { "id": 1, "name": "John", "age": 25 },
  { "id": 2, "name": "Jane", "age": 22 }
]
```

After POST /users with Alice, 28:

```
{
  "message": "User added",
  "id": 3
}
```

GET /users again:

```
[
  { "id": 1, "name": "John", "age": 25 },
  { "id": 2, "name": "Jane", "age": 22 },
  { "id": 3, "name": "Alice", "age": 28 }
```

]

7. Introduction to MySQL

a. Write a program to create a Database and table inside that database using MySQL command line client
b. Write a MySQL query to create table, insert data, update data in the table
c. Write a MySQL query to implement the subqueries in the MySQL command line client
d. Write a MySQL program to create the script files in the MySQL workbench
e. Write a MySQL program to create a database directory in Project and initialize a .sql file to integrate the database into API

7. Introduction to MySQL

7.a Create a Database and Table Using MySQL Command Line

Aim:

To create a database and a table in MySQL using the command line client.

Description:

This program demonstrates how to create a MySQL database and a table within it using the MySQL CLI.

Code / Commands:

```
-- Login to MySQL

mysql -u root -p


-- Create database

CREATE DATABASE mydb;


-- Use the database

USE mydb;


-- Create table

CREATE TABLE users (

    id INT AUTO_INCREMENT PRIMARY KEY,

    name VARCHAR(50),

    age INT

);
```

Output:

Query OK, 1 row affected (0.01 sec)

Query OK, 0 rows affected (0.00 sec)

Query OK, 0 rows affected (0.01 sec)

7.b Create Table, Insert Data, Update Data

Aim:

To perform basic operations: create table, insert data, and update data in MySQL.

Description:

This program demonstrates table creation and manipulation using SQL queries.

Code / Queries:

```
-- Create table
CREATE TABLE students (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(50),
    grade INT
);

-- Insert data
INSERT INTO students (name, grade) VALUES ('Alice', 90);
INSERT INTO students (name, grade) VALUES ('Bob', 85);

-- Update data
UPDATE students SET grade = 95 WHERE name = 'Bob';

-- Select to verify
SELECT * FROM students;
```

Output:

id name grade

1 Alice 90

2 Bob 95

7.c Implement Subqueries

Aim:

To demonstrate the use of subqueries in MySQL.

Description:

A subquery is a query nested inside another query. This example finds students whose grades are above the average.

Code / Queries:

-- Find students with grade above average

SELECT name, grade

FROM students

WHERE grade > (SELECT AVG(grade) FROM students);

Output:

name grade

Bob 95

7.d Create Script Files in MySQL Workbench

Aim:

To create and execute SQL scripts in MySQL Workbench.

Description:

SQL scripts allow you to store multiple SQL commands in a .sql file and execute them in one go.

Code / Example (script.sql):

-- script.sql

CREATE DATABASE projectDB;

USE projectDB;

```
CREATE TABLE employees (
   id INT AUTO_INCREMENT PRIMARY KEY,
   name VARCHAR(50),
   position VARCHAR(50)
);

INSERT INTO employees (name, position) VALUES ('John Doe', 'Manager');
INSERT INTO employees (name, position) VALUES ('Jane Smith', 'Developer');
```

Output in Workbench:

Database projectDB created successfully

Table employees created successfully

2 rows inserted

7.e Create Database Directory and .sql File for API Integration

Aim:

To integrate a MySQL database into a Node.js/Express project by creating a directory and SQL initialization file.

Description:

This step shows how to prepare a database script that can be used by the API for initialization.

Steps:

1. **Create project directory:**

```
mkdir myproject/database
cd myproject/database
```

2. **Initialize SQL file (init.sql):**

```
-- init.sql
CREATE DATABASE myprojectDB;
USE myprojectDB;

CREATE TABLE users (
   id INT AUTO_INCREMENT PRIMARY KEY,
```

```
    name VARCHAR(50),
    email VARCHAR(50)
);


INSERT INTO users (name, email) VALUES ('Alice', 'alice@example.com');
INSERT INTO users (name, email) VALUES ('Bob', 'bob@example.com');
```

3. **Integrate into API**

- Your Express.js app can read this .sql file and execute queries on startup to initialize the database using mysql2 driver.

Output Example:

Database myprojectDB created

Table users created

2 rows inserted

8. Team Collaboration Using GitHub

- Form groups of 2–3 students
- Create a shared GitHub repo
- Assign tasks and work in branches
- Use Issues, Pull Requests, and Code Reviews
- Document code with README.md

8. Team Collaboration Using GitHub

Aim:

To demonstrate collaborative development using GitHub by forming teams, working in branches, using issues, pull requests, and code reviews, and documenting the project.

Description:

This exercise shows how a team can work on a shared repository efficiently:

- Each member works in their **own branch**.
- Tasks are tracked using **GitHub Issues**.
- Code is reviewed via **Pull Requests (PRs)**.
- Project documentation is maintained using **README.md**.
- Collaboration ensures smooth integration of features and proper version control.

Example Team Members:

- Tony Stark
- Steve Rogers
- Thor Odinson

Steps / Workflow:

1. Form a Team

- Group of 2–3 students.
- Assign roles:
    - Developer: Tony Stark
    - Reviewer: Steve Rogers
    - Documenter: Thor Odinson

2. Create a Shared Repository

On GitHub, create repository: TeamProject

Clone locally

git clone https://github.com/<username>/TeamProject.git

cd TeamProject

3. Create and Work in Branches

Create a feature branch

git checkout -b feature/navbar

Add code or make changes

git add .

git commit -m "Implemented navbar component"

Push branch to remote

git push -u origin feature/navbar

- Each team member works in their own branch (e.g., feature/form, feature/login).

4. Use Issues

- Create **Issues** on GitHub for tasks, bugs, or enhancements.
- Assign issues to team members.
- Link commits and pull requests to relevant issues.

5. Create Pull Requests

- Open a **Pull Request (PR)** on GitHub to merge feature branch into main.
- Add reviewers for code review (e.g., Steve Rogers).
- Resolve any **merge conflicts** before merging.

6. Code Review & Merge

- Reviewer checks code, suggests changes, or approves.
- Merge feature branch into main once approved.

7. Document Code

- Add **README.md** in repository root:

TeamProject

Description

This project demonstrates collaborative development using GitHub.

Team Members

- Tony Stark (Developer)

- Steve Rogers (Reviewer)

- Thor Odinson (Documenter)

Setup Instructions

1. Clone the repository

2. Install dependencies

3. Run project

Output / Results:

1. **GitHub Repository:**

- Multiple branches visible: main, feature/navbar, feature/form.
- Issues and PRs track tasks.

2. **Merged Code:**

- Feature branches successfully merged into main.
- No conflicts remain after review.

3. **Documentation:**

- README.md contains project description, team members, and setup instructions.

4. **Collaborative Workflow:**

- Team members work on different features simultaneously.
- GitHub tracks all changes, merges, and history.

Notes / Best Practices:

- Pull latest main before creating a new branch.
- Commit frequently with descriptive messages.
- Use clear branch names: feature/login, bugfix/navbar.
- Resolve conflicts quickly and communicate with teammates.
- Keep README.md updated with project changes.