FULL STACK DEVELOPMENT -2
(23E00)

B. TECH III YEAR - I SEM (2025-26)

DEPARTMENT COMPUTER SCIENCE AND ENGINEERINGOF

CHAITANYA BHARATHI INSTITUTE OF TECHNOLOGY
(Autonomous Institution – UGC, Govt. of India)
(Affiliated to AICTE, NEW DELHI, And Affiliated to JNTUA ,Anantapuramu Accredited by NBA & NAAC – 'A' Grade, and NBA and Recognized By UGC) VIDYA NAGAR, Pallavolu ,Proddatur – 516352, Andhra Pradesh, INDIA.

Question 1: Introduction to Modern JavaScript and DOM

a. **Write a JavaScript program to link a JavaScript file with an HTML page.**

b. **Write a JavaScript program to select elements in an HTML page using selectors.**

c. **Write a JavaScript program to implement event listeners.**

d. **Write a JavaScript program to handle click events for HTML button elements.**

e. **Write a JavaScript program demonstrating three types of functions:**

- **i. Function Declaration**
- **ii. Function Expression (Function Definition)**
- **iii. Arrow Function**

Experiment (a): Linking JavaScript to an HTML Page

Aim:

To create and demonstrate how to link a JavaScript file to an HTML page and perform basic interactions using JavaScript.

Procedure:

1. Create an HTML file named `index.html`.
2. Create a separate JavaScript file named `script.js`.
3. Link the JavaScript file to the HTML file using the `<script>` tag.
4. Add a button in the HTML page to trigger a JavaScript function.
5. Define the function in `script.js` to display an alert when the button is clicked.
6. Save both files and open the HTML file in a browser to test the output.

Code:

index.html

```
<!DOCTYPE html>
<html>
<head>
    <title>JavaScript Linking Example</title>
</head>
<body>

    <h1>Welcome to JavaScript Linking Example</h1>
    <button onclick="showMessage()">Click Me</button>

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

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

---

script.js

```
function showMessage() {

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

Output:

When the "Click Me" button is clicked, a pop-up alert box will display the message:
"**JavaScript file successfully linked!**"

Experiment (b): Selecting Elements in an HTML Page Using Selectors

---

Aim:

To demonstrate how to select HTML elements using different JavaScript selectors such as `getElementById`, `getElementsByClassName`, `getElementsByTagName`, `querySelector`, and `querySelectorAll`.

Procedure:

1. Create an HTML file named `selectors.html`.
2. Add a heading, paragraph, and a few elements with different IDs, classes, and tags.
3. Create a separate JavaScript file named `selectors.js`.
4. Use various JavaScript DOM selection methods to access and modify the content or style of the elements.
5. Link the JavaScript file in the HTML file.
6. Save and open the HTML file in a browser to observe the changes.

Code:

selectors.html

```
<!DOCTYPE html>
<html>
<head>
    <title>Element Selectors Example</title>
</head>
<body>

    <h1 id="main-heading">Hello!</h1>
    <p class="info">This is a paragraph with class "info".</p>
    <p>This is a normal paragraph.</p>
    <div class="info">Another div with class "info".</div>
    <button onclick="changeContent()">Click to Change Content</button>

    <!-- Link the JavaScript file -->
    <script src="selectors.js"></script>
</body>
</html>
```

---

selectors.js

```
function changeContent() {
    // Select element by ID
    const heading = document.getElementById("main-heading");
    heading.textContent = "Welcome to JavaScript DOM!";

    // Select elements by class name
    const infoElements = document.getElementsByClassName("info");
    for (let i = 0; i < infoElements.length; i++) {
        infoElements[i].style.color = "blue";
    }

    // Select elements by tag name
    const paragraphs = document.getElementsByTagName("p");
    paragraphs[1].style.fontWeight = "bold";

    // Select using querySelector
    const firstInfo = document.querySelector(".info");
    firstInfo.style.backgroundColor = "lightyellow";

    // Select using querySelectorAll
    const allInfo = document.querySelectorAll(".info");
    allInfo.forEach(el => el.style.border = "1px solid red");
}
```

Output:

- The heading text will change to “Welcome to JavaScript DOM!”
- All elements with class `"info"` will turn blue and get a red border.
- The second paragraph will become bold.
- The first element with class `"info"` will have a light yellow background.

Hello!

This is a paragraph with class "info".

This is a normal paragraph.

Another div with class "info".

Click to Change Content

Experiment (c): Implementing Event Listeners in JavaScript

Aim:

To implement and demonstrate the use of **event listeners** in JavaScript to respond to user interactions like clicks and mouse events.

Procedure:

1. Create an HTML file named `event_listener.html`.
2. Create a JavaScript file named `main.js`.
3. Create a CSS file named `styles.css` to style the elements.
4. In the HTML file, add a button and a paragraph.
5. In the JavaScript file, use `addEventListener()` to listen to events like `click` and `mouseover`.
6. On button click, update the paragraph text. On mouseover, change the style.
7. Link both the JavaScript and CSS files to the HTML file.
8. Save and run the file in a web browser.

Code:

`event_listener.html`

```
<!DOCTYPE html>
<html>
<head>
    <title>Event Listener Example</title>
    <link rel="stylesheet" href="styles.css">
</head>
<body>

    <h2>JavaScript Event Listener Demo</h2>
    <button id="changeBtn">Click Me</button>
    <p id="message">This is the original message.</p>
```

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

---

main.js

```
// Select the button and paragraph
const button = document.getElementById("changeBtn");
const paragraph = document.getElementById("message");

// Add click event listener
button.addEventListener("click", function () {
    paragraph.textContent = "The message has been updated!";
});

// Add mouseover event listener
paragraph.addEventListener("mouseover", function () {
    paragraph.style.color = "green";
});

// Add mouseout event to reset style
paragraph.addEventListener("mouseout", function () {
    paragraph.style.color = "black";
});
```

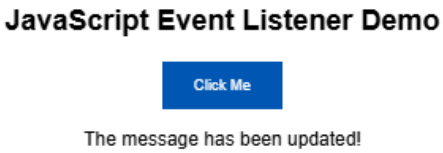
---

styles.css

```
body {
    font-family: Arial, sans-serif;
    text-align: center;
    margin-top: 50px;
}

button {
    padding: 10px 20px;
    background-color: #007bff;
    color: white;
    border: none;
    cursor: pointer;
}

button:hover {
    background-color: #0056b3;
}
```

Output:

- When the **button is clicked**, the paragraph content changes to:
  "The message has been updated!"
- When you **hover over the paragraph**, the text turns **green**.
- When you **move the mouse away**, it returns to **black**.



-

Experiment (d): Handling Click Events for HTML Button Elements

---

Aim:

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

Procedure:

1. Create an HTML file named `button_events.html`.
2. Create a JavaScript file named `main.js`.
3. Create a CSS file named `styles.css` to style the buttons.
4. Add multiple buttons in the HTML file.
5. In the JavaScript file, add event listeners to handle different actions when each button is clicked.
6. Link the JavaScript and CSS files to the HTML file.
7. Save and open the file in a browser to test the behavior.

Code:

button_events.html

```
html
CopyEdit
<!DOCTYPE html>
<html>
<head>
    <title>Button Click Events</title>
    <link rel="stylesheet" href="styles.css">
</head>
<body>

    <h2>Click Event Handler Example</h2>

    <button id="greetBtn">Greet</button>
    <button id="changeColorBtn">Change Background</button>
    <button id="hideTextBtn">Hide Text</button>

    <p id="displayText">Welcome! Click buttons to see changes.</p>

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

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

---

main.js

```
javascript
CopyEdit
// Select buttons and paragraph
const greetBtn = document.getElementById("greetBtn");
const colorBtn = document.getElementById("changeColorBtn");
const hideBtn = document.getElementById("hideTextBtn");
const text = document.getElementById("displayText");

// Greet button click
greetBtn.addEventListener("click", () => {
    alert("Hello! Have a great day!");
});

// Change background color
colorBtn.addEventListener("click", () => {
    document.body.style.backgroundColor = "#e0f7fa";
});

// Hide text on click
hideBtn.addEventListener("click", () => {
    text.style.display = "none";
});
```

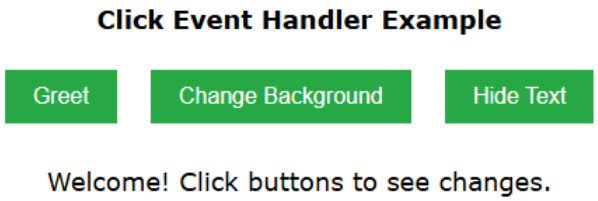
---

styles.css

```
css
CopyEdit
body {
    font-family: Verdana, sans-serif;
    text-align: center;
    margin-top: 60px;
}

button {
    margin: 10px;
    padding: 10px 20px;
    border: none;
    background-color: #28a745;
    color: white;
    font-size: 16px;
    cursor: pointer;
}

button:hover {
    background-color: #218838;
}

#displayText {
    margin-top: 20px;
    font-size: 18px;
}
```

Output:

- Clicking the **“Greet”** button shows an alert: *"Hello! Have a great day!"*
- Clicking the **“Change Background”** button changes the page background to a light blue color.
- Clicking the **“Hide Text”** button hides the paragraph from the page.



Experiment (e): Demonstrating Three Types of JavaScript Functions

---

Aim:

To demonstrate different types of function definitions in JavaScript:

- **Function Declaration**
- **Function Expression**
- **Arrow Function**

Procedure:

1. Create an HTML file named `function_types.html`.
2. Create a JavaScript file named `main.js`.
3. Create a CSS file named `styles.css` to style the page and buttons.
4. Add three buttons in the HTML file, each triggering a different function type.
5. In `main.js`, define the functions using:
    - Function Declaration
    - Function Expression
    - Arrow Function
6. Display output in a `<p>` element when each function is executed.
7. Link the JavaScript and CSS files to the HTML file.
8. Save and run the HTML file in a browser to see the result.

Code:

`function_types.html`

```
html
CopyEdit
<!DOCTYPE html>
<html>
```

```
<head>
    <title>Types of JavaScript Functions</title>
    <link rel="stylesheet" href="styles.css">
</head>
<body>

    <h2>Types of JavaScript Functions</h2>

    <button onclick="greet()">Function Declaration</button>
    <button id="expressionBtn">Function Expression</button>
    <button id="arrowBtn">Arrow Function</button>

    <p id="outputText">Click a button to invoke the function.</p>

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

---

main.js

```
javascript
CopyEdit
// i. Function Declaration
function greet() {
    document.getElementById("outputText").textContent = "Hello from Function
Declaration!";
}

// ii. Function Expression
const showMessage = function() {
    document.getElementById("outputText").textContent = "Hello from Function
Expression!";
};
document.getElementById("expressionBtn").addEventListener("click", showMessage);

// iii. Arrow Function
const arrowFunc = () => {
    document.getElementById("outputText").textContent = "Hello from Arrow
Function!";
};
document.getElementById("arrowBtn").addEventListener("click", arrowFunc);
```

---

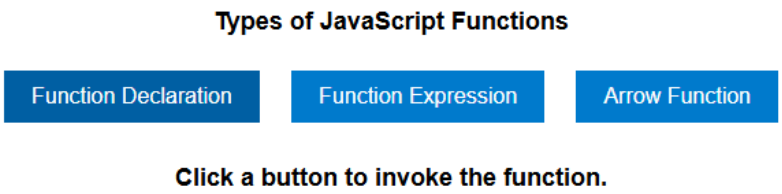
styles.css

```
css
CopyEdit
body {
    font-family: sans-serif;
    text-align: center;
    margin-top: 50px;
}

button {
    margin: 10px;
    padding: 10px 20px;
    font-size: 16px;
    background-color: #007acc;
    color: white;
    border: none;
    cursor: pointer;
```

```
}

button:hover {
    background-color: #005fa3;
}

#outputText {
    margin-top: 20px;
    font-size: 18px;
    font-weight: bold;
}
```

Output:

- Clicking the **"Function Declaration"** button displays:
  "Hello from Function Declaration!"
- Clicking the **"Function Expression"** button displays:
  "Hello from Function Expression!"
- Clicking the **"Arrow Function"** button displays:
  "Hello from Arrow Function!"



Question 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 button click events inside a 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.

Experiment (a): Counter Button using React Class Components

Aim:

To implement a simple counter application using **React Class Components**, where a button click increments the count value displayed on the screen.

Procedure:

1. Create a new React project using the command:

   ```
   npx create-react-app class-counter
   ```

2. Navigate to the project directory:

   ```
   cd class-counter
   ```

3. Open the `src/App.js` file.
4. Replace the existing content with a React class component named `Counter` that:
   - Initializes a `count` state.
   - Increments the count using `this.setState()` when the button is clicked.
5. Start the development server using:

   ```
   npm start
   ```

6. Observe the counter updating in the browser when the button is clicked.

Code:

src/App.js

```
import React, { Component } from 'react';
import './App.css';

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

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

  render() {
    return (
      <div className="App">
        <h2>Counter using Class Component</h2>
        <p>Count: {this.state.count}</p>
        <button onClick={this.incrementCount}>Increment</button>
      </div>
    );
  }
}

export default Counter;
```

src/index.js

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import './index.css';
import Counter from './App';

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Counter />);
```

Output:

- Initially displays:

  ```
  Count: 0
  [Increment]
  ```

- On each button click, the count value increases by 1.

Experiment (b): Counter Button using React Functional Components

---

Aim:

To create a counter application using **React Functional Components** and the **useState** hook to update and display the count when a button is clicked.

Procedure:

1. Create a new React app using the command:

   ```
   npx create-react-app functional-counter
   ```

2. Navigate to the project directory:

   ```
   cd functional-counter
   ```

3. Open the `src/App.js` file.
4. Define a **functional component** using the `useState` hook to manage the counter.
5. Use a `<button>` element to increment the count on click.
6. Start the React app using `npm start` and observe the output in the browser.

Code:

`src/App.js`

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

```
import './App.css';

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

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

  return (
    <div className="App">
      <h2>Counter using Functional Component</h2>
      <p>Count: {count}</p>
      <button onClick={increment}>Increment</button>
    </div>
  );
}

export default Counter;
```

---

src/index.js

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import './index.css';
import Counter from './App';

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Counter />);
```

Output:

- Initial display:

  ```
  Count: 0
  [Increment]
  ```

- Each click on the "Increment" button increases the count by 1.

Experiment (c): Handling Button Click Events in a Functional Component

---

Aim:

To demonstrate how to handle **button click events** inside a **React Functional Component** using the `onClick` event handler.

Procedure:

1. Create a new React app using:

```
npx create-react-app button-events
```

2. Navigate to the project folder:

```
cd button-events
```

3. Open `src/App.js`.
4. Define a functional component and create multiple button click event handlers.
5. Use the `onClick` attribute in buttons to trigger these functions.
6. Start the app using `npm start` and verify the behavior in the browser.

Code:

src/App.js

```
import React, { useState } from 'react';
import './App.css';

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

  const sayHello = () => {
    setMessage("Hello, User!");
  };

  const resetMessage = () => {
    setMessage("Click a button to see the action");
  };

  return (
    <div className="App">
      <h2>Handling Button Click Events</h2>
      <p>{message}</p>
      <button onClick={sayHello}>Say Hello</button>
      <button onClick={resetMessage}>Reset</button>
    </div>
  );
}

export default ButtonEvents;
```

---

src/index.js

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import './index.css';
import ButtonEvents from './App';

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<ButtonEvents />);
```

Output:

- Initially shows:

```
Click a button to see the action
[Say Hello] [Reset]
```

- Clicking **“Say Hello”** updates the message to:

```
Hello, User!
```

- Clicking **“Reset”** resets the message.

Experiment (d): Conditionally Rendering a Component in the Browser

---

Aim:

To demonstrate how to **conditionally render** a component in React using a **functional component** and the `useState` hook.

Procedure:

1. Create a new React app using the command:

```
npx create-react-app conditional-rendering
```

2. Navigate to the project directory:

```
cd conditional-rendering
```

3. Open `src/App.js`.
4. Create a functional component with state to control the visibility of another component.
5. Use a toggle button to show or hide a message component.
6. Start the React app using:

```
npm start
```

7. Observe the component being conditionally rendered based on state.

Code:

src/App.js

```
import React, { useState } from 'react';
import './App.css';

function Message() {
  return <p>This is a conditionally rendered component!</p>;
}
```

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

  const toggleMessage = () => {
    setShow(!show);
  };

  return (
    <div className="App">
      <h2>Conditional Rendering Example</h2>
      <button onClick={toggleMessage}>
        {show ? "Hide Message" : "Show Message"}
      </button>
      {show && <Message />}
    </div>
  );
}

export default App;
```

---

src/index.js

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import './index.css';
import App from './App';

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

Output:

- Initially, only the button appears with text **"Show Message"**.
- Clicking the button toggles the appearance of the message:
  - **Show Message** → Displays the message.
  - **Hide Message** → Hides the message.

Experiment (e): Displaying Text Using String Literals in React

---

Aim:

To display text on the browser using **JavaScript string literals** (including template literals) within a **React Functional Component**.

Procedure:

1. Create a new React app using the command:

   ```
   npx create-react-app string-literals
   ```

2. Navigate to the project folder:

   ```
   cd string-literals
   ```

3. Open `src/App.js`.
4. Define a functional component that:
   - Declares string variables.
   - Uses **template literals** to create a combined message.
   - Displays the message inside JSX using `{}`.
5. Start the development server using:

   ```
   npm start
   ```

6. Observe the text rendered on the browser.

Code:

src/App.js

```
import React from 'react';
import './App.css';

function App() {
  const name = "Sai Kiran";
  const course = "B.Tech CSE";
  const greeting = `Hello, my name is ${name} and I am studying ${course}.`;

  return (
    <div className="App">
      <h2>Displaying Text Using String Literals</h2>
      <p>{greeting}</p>
    </div>
  );
}

export default App;
```

---

src/index.js

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import './index.css';
import App from './App';

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

Output:

Displays on the browser:

```
Hello, my name is Sai Kiran and I am studying B.Tech CSE.
```

Question 3: Important Concepts of React.js

---

a. Write a React program to implement a counter button using the useState hook.

b. Write a React program to fetch data from an API using the useEffect hook.

c. Write a React program with two components where data is shared using Props.

d. Write a React program to implement Forms in React.

e. Write a React program to implement Iterative Rendering using the map() function.

Experiment (a): Counter Button using the `useState` Hook

Aim:

To implement a **counter button** in React using the **`useState` hook** for state management in a **functional component**.

Procedure:

1. Create a new React project:

   ```
   npx create-react-app use-state-counter
   ```

2. Navigate to the project directory:

   ```
   cd use-state-counter
   ```

3. Open `src/App.js` in a code editor.
4. Import the `useState` hook from React.
5. Create a functional component that:
   - Defines a `count` state using `useState`.
   - Updates the count value on button click using `setCount()`.
6. Display the count on the screen and render a button to increment it.
7. Run the app using:

   ```
   npm start
   ```

Code:

`src/App.js`

```
import React, { useState } from 'react';
import './App.css';
```

```
function App() {
  const [count, setCount] = useState(0); // Initial count value is 0

  const increment = () => {
    setCount(count + 1); // Update count state
  };

  return (
    <div className="App">
      <h2>React Counter using useState Hook</h2>
      <p>Current Count: {count}</p>
      <button onClick={increment}>Increment</button>
    </div>
  );
}

export default App;
```

---

src/index.js

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import './index.css';
import App from './App';

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

Output:

- Initially shows:

  ```
  Current Count: 0
  [Increment]
  ```

- On each click of the **Increment** button, the counter value increases by 1.

Experiment (b): Fetching Data from an API using the `useEffect` Hook

Aim:

To demonstrate how to **fetch data from an API** using the `useEffect` hook in a **React Functional Component** and display it in the browser.

Procedure:

1. Create a new React app using the command:

```
npx create-react-app api-fetch
```

2. Navigate to the project directory:

```
cd api-fetch
```

3. Open `src/App.js` in a code editor.
4. Use `useState` to store fetched data and `useEffect` to make the API call.
5. Fetch data using `fetch()` or `axios` from a sample API like `https://jsonplaceholder.typicode.com/users`.
6. Display the fetched data using JSX.
7. Start the development server with:

```
npm start
```

Code:

src/App.js

```
import React, { useState, useEffect } from 'react';
import './App.css';

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

  useEffect(() => {
    // Fetching user data from API
    fetch('https://jsonplaceholder.typicode.com/users')
      .then(response => response.json())
      .then(data => setUsers(data))
      .catch(error => console.error('Error fetching data:', error));
  }, []); // Empty dependency array means this runs once on component mount

  return (
    <div className="App">
      <h2>Users List from API</h2>
      <ul>
        {users.map(user => (
          <li key={user.id}>
            {user.name} ({user.email})
          </li>
        ))}
      </ul>
    </div>
  );
}

export default App;
```

---

src/index.js

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import './index.css';
import App from './App';

const root = ReactDOM.createRoot(document.getElementById('root'));
```

```
root.render(<App />);
```

Output:

- Displays a list of users fetched from the API:

```
Users List from API
- Leanne Graham (Sincere@april.biz)
- Ervin Howell (Shanna@melissa.tv)
- ...
```

- The data is loaded once when the component is mounted using `useEffect`.

Experiment (c): Sharing Data Between Components Using Props

Aim:

To demonstrate how to **pass data from one component to another** in React using **props**.

Procedure:

1. Create a new React project using:

```
npx create-react-app props-example
```

2. Navigate to the project directory:

```
cd props-example
```

3. Open `src/App.js` and define a **parent component** that passes data as props.
4. Create a **child component** (`Greeting.js`) that receives and displays the data.
5. Use JSX to render the child component and display the props.
6. Start the app with:

```
npm start
```

Code:

`src/App.js` (Parent Component)

```
import React from 'react';
import Greeting from './Greeting';
import './App.css';

function App() {
  const studentName = "Sai Kiran";
```

```
  return (
    <div className="App">
      <h2>Props Example</h2>
      <Greeting name={studentName} />
    </div>
  );
}

export default App;
```

`src/Greeting.js` (Child Component)

```
import React from 'react';

function Greeting(props) {
  return (
    <div>
      <p>Hello, {props.name}! Welcome to React Props Demo.</p>
    </div>
  );
}

export default Greeting;
```

`src/index.js`

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import './index.css';
import App from './App';

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

Output:

- The browser displays:

  ```
  Hello, Sai Kiran! Welcome to React Props Demo.
  ```

- The `App` component sends the `studentName` to the `Greeting` component through **props**.

Experiment (d): Implementing Forms in React

Aim:

To demonstrate how to implement **forms in React** using controlled components and the `useState` hook to manage form input values.

Procedure:

1. Create a new React app using:

   ```
   npx create-react-app react-form
   ```

2. Navigate to the project directory:

   ```
   cd react-form
   ```

3. Open `src/App.js` and define a form with input fields (Name and Email).
4. Use `useState` to manage the values of form inputs.
5. Handle form submission and display the entered data.
6. Start the development server using:

   ```
   npm start
   ```

Code:

src/App.js

```
import React, { useState } from 'react';
import './App.css';

function App() {
  const [name, setName] = useState('');
  const [email, setEmail] = useState('');
  const [submitted, setSubmitted] = useState(false);

  const handleSubmit = (event) => {
    event.preventDefault();
    setSubmitted(true);
  };

  return (
    <div className="App">
      <h2>React Form Example</h2>

      <form onSubmit={handleSubmit}>
        <div>
          <label>Name: </label>
          <input
            type="text"
            value={name}
            onChange={(e) => setName(e.target.value)}
            required
          />
        </div>
        <div>
          <label>Email: </label>
          <input
            type="email"
            value={email}
            onChange={(e) => setEmail(e.target.value)}
            required
          />
        </div>
        <button type="submit">Submit</button>
```

```
      </form>

      {submitted && (
        <div className="output">
          <h4>Form Submitted</h4>
          <p><strong>Name:</strong> {name}</p>
          <p><strong>Email:</strong> {email}</p>
        </div>
      )}
    </div>
  );
}

export default App;
```

---

src/index.js

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import './index.css';
import App from './App';

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

Output:

- Initially, the form is displayed with **Name** and **Email** input fields.
- On submitting the form:
  - The entered values are displayed below the form.
  - Example:

```
Form Submitted
Name: Sai Kiran
Email: saikiran@example.com
```

Question 4: Introduction to Node.js and Express.js

---

a. Write a program to implement the "Hello World" message in a route and display it 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" message in the browser console using Express.js.

d. Write a program to implement basic CRUD (Create, Read, Update, Delete) operations using Express.js.

e. Write a program to establish a connection between an API and a MySQL database using the Express–MySQL driver.

Experiment (a): Displaying "Hello World" in a Route using Express.js

---

Aim:

To create a simple Express.js application that sends a **"Hello World"** message when a route is accessed through the browser.

Procedure:

1. Ensure Node.js is installed on your system. Check by running:

   ```
   node -v
   npm -v
   ```

2. Create a new project folder and initialize it:

   ```
   mkdir express-hello
   cd express-hello
   npm init -y
   ```

3. Install the Express.js module:

   ```
   npm install express
   ```

4. Create a file named `app.js` and write the Express server code.
5. Run the server using:

   ```
   node app.js
   ```

6. Open your browser and visit:
   `http://localhost:3000`
7. The message **"Hello World from Express!"** should appear.

Code:

app.js

```
// Import the express module
const express = require('express');

// Create an Express application
const app = express();
```

```
// Define a route for the homepage
app.get('/', (req, res) => {
  res.send('Hello World from Express!');
});

// Start the server on port 3000
app.listen(3000, () => {
  console.log('Server is running at http://localhost:3000');
});
```

Output:

- When you open your browser and go to `http://localhost:3000`, you will see:

  ```
  Hello World from Express!
  ```

- The terminal will log:

  ```
  Server is running at http://localhost:3000
  ```

Experiment (b): Developing a Small Website with Multiple Routes using Express.js

---

Aim:

To create a small website with multiple routes like Home, About, and Contact using **Express.js**, and return appropriate messages for each route.

Procedure:

1. Open the terminal and create a new project:

   ```
   mkdir express-multiroute
   cd express-multiroute
   npm init -y
   ```

2. Install Express:

   ```
   npm install express
   ```

3. Create a file named `app.js`.
4. Define multiple routes (`/`, `/about`, `/contact`) using `app.get()`.
5. Run the server using:

   ```
   node app.js
   ```

6. Open your browser and test:
   - `http://localhost:3000/`
   - `http://localhost:3000/about`

- http://localhost:3000/contact

Code:

app.js

```
// Import express module
const express = require('express');

// Create express app
const app = express();

// Define PORT
const PORT = 3000;

// Home Route
app.get('/', (req, res) => {
  res.send('<h1>Welcome to My Website</h1><p>This is the Home Page.</p>');
});

// About Route
app.get('/about', (req, res) => {
  res.send('<h1>About Us</h1><p>This page provides information about us.</p>');
});

// Contact Route
app.get('/contact', (req, res) => {
  res.send('<h1>Contact Us</h1><p>You can contact us at
contact@example.com</p>');
});

// 404 Route - Catch All
app.get('*', (req, res) => {
  res.status(404).send('<h1>404 Page Not Found</h1>');
});

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

Output:

- Visiting `http://localhost:3000/` will show:

  ```
  Welcome to My Website
  This is the Home Page.
  ```

- Visiting `http://localhost:3000/about` will show:

  ```
  About Us
  This page provides information about us.
  ```

- Visiting `http://localhost:3000/contact` will show:

  ```
  Contact Us
  ```

```
You can contact us at contact@example.com
```

- Any undefined route like `http://localhost:3000/help` will show:

```
404 Page Not Found
```

Experiment (c): Printing "Hello World" in the Browser Console using Express.js

---

Aim:

To use **Express.js** to serve an HTML file containing JavaScript that prints **"Hello World"** in the **browser console** (not on the page).

Procedure:

1. Create a project folder and initialize:

```
mkdir express-browser-console
cd express-browser-console
npm init -y
```

2. Install Express:

```
npm install express
```

3. Create two files:
   - `app.js` (for Express server)
   - `public/index.html` (for HTML + JS code)
4. Use Express's `static` middleware to serve the `public` folder.
5. Run the server:

```
node app.js
```

6. Open browser at `http://localhost:3000` and open browser **Developer Tools → Console**.

Code:

`app.js`

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

// Serve static files from the "public" directory
app.use(express.static('public'));
```

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

---

public/index.html

```
<!DOCTYPE html>
<html>
<head>
  <title>Hello Console</title>
</head>
<body>
  <h1>Check the browser console</h1>

  <script>
    console.log("Hello World from Express.js!");
  </script>
</body>
</html>
```

Output:

- **Browser View**: Displays:

  ```
  Check the browser console
  ```

- **Console Output (Press F12 > Console tab)**:

  ```
  Hello World from Express.js!
  ```

Experiment (d): Implementing Basic CRUD Operations using Express.js

---

Aim:

To create a basic Express.js server that performs **CRUD** operations:
Create, **Read**, **Update**, and **Delete** on in-memory user data.

Procedure:

1. Create a new folder and initialize a Node.js project:

   ```
   mkdir express-crud
   cd express-crud
   npm init -y
   ```

2. Install Express:

```
npm install express
```

3. Create a file named `app.js`.
4. Define REST API routes for:
    - `GET` → Read all users
    - `POST` → Create a user
    - `PUT` → Update a user
    - `DELETE` → Delete a user
5. Use Postman or browser to test the endpoints.
6. Run the server:

```
node app.js
```

Code:

app.js

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

// Middleware to parse JSON
app.use(express.json());

// Sample in-memory user data
let users = [
  { id: 1, name: 'Alice' },
  { id: 2, name: 'Bob' }
];

// Read: GET all users
app.get('/users', (req, res) => {
  res.json(users);
});

// Create: POST a new user
app.post('/users', (req, res) => {
  const newUser = {
    id: users.length + 1,
    name: req.body.name
  };
  users.push(newUser);
  res.status(201).json(newUser);
});

// Update: PUT user by id
app.put('/users/:id', (req, res) => {
  const userId = parseInt(req.params.id);
  const updatedName = req.body.name;
  const user = users.find(u => u.id === userId);

  if (user) {
    user.name = updatedName;
    res.json(user);
  } else {
    res.status(404).json({ message: 'User not found' });
  }
});
```

```
// Delete: DELETE user by id
app.delete('/users/:id', (req, res) => {
  const userId = parseInt(req.params.id);
  users = users.filter(u => u.id !== userId);
  res.json({ message: 'User deleted successfully' });
});

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

Testing the API (Use Postman or cURL):

Operation	Method	Endpoint	Request Body (JSON)	Response
Read	GET	`/users`	—	List of users
Create	POST	`/users`	`{ "name": "Charlie" }`	New user created
Update	PUT	`/users/2`	`{ "name": "Bobby" }`	User 2 updated
Delete	DELETE	`/users/1`	—	User 1 deleted confirmation

Output Example:

```
// GET /users
[
  { "id": 1, "name": "Alice" },
  { "id": 2, "name": "Bob" }
]

// POST /users { "name": "Charlie" }
{ "id": 3, "name": "Charlie" }
```

Experiment (e): Connecting Express API to a MySQL Database

Aim:

To establish a connection between an **Express.js API** and a **MySQL database** using the **`mysql2`** driver and perform a sample `SELECT` query.

Procedure:

1. Ensure **MySQL server** is running on your system.
2. Create a database and table in MySQL:

```
CREATE DATABASE testdb;
USE testdb;

CREATE TABLE users (
```

```
  id INT AUTO_INCREMENT PRIMARY KEY,
  name VARCHAR(100),
  email VARCHAR(100)
);

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

3. Create a Node.js project:

```
mkdir express-mysql-api
cd express-mysql-api
npm init -y
```

4. Install necessary packages:

```
npm install express mysql2
```

5. Create `app.js` and write code to:
   - Connect to MySQL database
   - Handle GET `/users` route to fetch all user records
6. Run the server:

```
node app.js
```

7. Test the API using a browser or Postman:

```
http://localhost:3000/users
```

Code:

app.js

```
const express = require('express');
const mysql = require('mysql2');
const app = express();
const PORT = 3000;

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

// Connect to the database
connection.connect((err) => {
  if (err) {
    console.error('Database connection failed:', err.stack);
    return;
  }
  console.log('Connected to MySQL database.');
});

// API endpoint to fetch all users
app.get('/users', (req, res) => {
```

```
  const query = 'SELECT * FROM users';

  connection.query(query, (error, results) => {
    if (error) {
      res.status(500).json({ error: 'Failed to fetch users' });
    } else {
      res.json(results);
    }
  });
});

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

Output:

When visiting `http://localhost:3000/users`, the browser or Postman displays:

```
[
  { "id": 1, "name": "Alice", "email": "alice@example.com" },
  { "id": 2, "name": "Bob", "email": "bob@example.com" }
]
```

Question 5: Introduction to MySQL

---

a. Write a MySQL program to create a database and a table inside that database using the MySQL Command Line Client.

b. Write MySQL queries to create a table, insert data into the table, and update data in the table.

c. Write MySQL queries to implement subqueries using the MySQL Command Line Client.

d. Write a MySQL program to create script files using the MySQL Workbench.

e. Write a MySQL program to create a database directory inside a project and initialize a `database.sql` file to integrate the database into an API.

Experiment (a): Creating a Database and Table using MySQL Command Line Client

---

Aim:

To create a new **MySQL database** and a **table** within that database using SQL commands in the **MySQL Command Line Client**.

Procedure:

1. Open the **MySQL Command Line Client** and log in using your username and password.
2. Use the `CREATE DATABASE` command to create a new database.
3. Select the database using the `USE` command.
4. Create a table using the `CREATE TABLE` command with appropriate column definitions.
5. Verify the creation using the `SHOW` and `DESC` commands.

SQL Code:

```
-- Step 1: Create a database
CREATE DATABASE CollegeDB;

-- Step 2: Use the created database
USE CollegeDB;

-- Step 3: Create a table named 'students'
CREATE TABLE students (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(100),
    age INT,
    department VARCHAR(50)
);

-- Step 4: Show available tables
SHOW TABLES;

-- Step 5: Describe the structure of the 'students' table
DESC students;
```

Expected Output:

After executing the above commands:

- `SHOW TABLES;` will output:

```
+------------------+
| Tables_in_CollegeDB |
+------------------+
| students         |
+------------------+
```

- `DESC students;` will output:

```
+------------+--------------+------+-----+---------+----------------+
| Field      | Type         | Null | Key | Default | Extra          |
+------------+--------------+------+-----+---------+----------------+
| id         | int          | NO   | PRI | NULL    | auto_increment |
| name       | varchar(100) | YES  |     | NULL    |                |
| age        | int          | YES  |     | NULL    |                |
| department | varchar(50)  | YES  |     | NULL    |                |
```

```
+-----------+-------------+------+-----+---------+---------------+
```

Experiment (b): Creating a Table, Inserting Data, and Updating Data using MySQL

Aim:

To write SQL queries to:

- Create a table
- Insert records into the table
- Update records in the table
  using the **MySQL Command Line Client**.

Procedure:

1. Open **MySQL Command Line Client** and select the desired database using `USE`.
2. Create a table using `CREATE TABLE`.
3. Insert records using `INSERT INTO`.
4. Modify data using `UPDATE`.
5. Use `SELECT * FROM` to verify changes.

SQL Code:

```
-- Step 1: Use the database
USE CollegeDB;

-- Step 2: Create a table called 'courses'
CREATE TABLE courses (
    course_id INT AUTO_INCREMENT PRIMARY KEY,
    course_name VARCHAR(100),
    instructor VARCHAR(100),
    duration INT -- duration in weeks
);

-- Step 3: Insert data into the 'courses' table
INSERT INTO courses (course_name, instructor, duration)
VALUES
('Database Systems', 'Dr. Smith', 10),
('Operating Systems', 'Prof. Johnson', 8),
('Web Technologies', 'Ms. Linda', 12);

-- Step 4: View inserted records
SELECT * FROM courses;

-- Step 5: Update a course's duration
UPDATE courses
SET duration = 14
```

```
WHERE course_name = 'Web Technologies';

-- Step 6: View updated records
SELECT * FROM courses;
```

Expected Output:

After Step 4:

```
+-----------+--------------------+----------------+----------+
| course_id | course_name        | instructor     | duration |
+-----------+--------------------+----------------+----------+
|     1     | Database Systems   | Dr. Smith      |    10    |
|     2     | Operating Systems  | Prof. Johnson  |    8     |
|     3     | Web Technologies   | Ms. Linda      |    12    |
+-----------+--------------------+----------------+----------+
```

After Step 6:

```
+-----------+--------------------+----------------+----------+
| course_id | course_name        | instructor     | duration |
+-----------+--------------------+----------------+----------+
|     1     | Database Systems   | Dr. Smith      |    10    |
|     2     | Operating Systems  | Prof. Johnson  |    8     |
|     3     | Web Technologies   | Ms. Linda      |    14    |
+-----------+--------------------+----------------+----------+
```

Experiment (c): Implementing Subqueries using MySQL Command Line Client

---

Aim:

To demonstrate how to use **subqueries** in MySQL for extracting and filtering data based on results from another query.

Procedure:

1. Open **MySQL Command Line Client** and use an existing database (e.g., `CollegeDB`).
2. Create two related tables: `students` and `departments`.
3. Populate both tables with sample data.
4. Write **subqueries** inside `SELECT`, `WHERE`, and `FROM` clauses.
5. View results and verify accuracy.

SQL Code:

```
-- Step 1: Use the database
USE CollegeDB;

-- Step 2: Create 'departments' table
CREATE TABLE departments (
    dept_id INT PRIMARY KEY,
    dept_name VARCHAR(50)
);

-- Step 3: Create 'students' table
CREATE TABLE students (
    id INT PRIMARY KEY,
    name VARCHAR(100),
    age INT,
    dept_id INT,
    FOREIGN KEY (dept_id) REFERENCES departments(dept_id)
);

-- Step 4: Insert data into departments
INSERT INTO departments VALUES
(1, 'CSE'),
(2, 'ECE'),
(3, 'MECH');

-- Step 5: Insert data into students
INSERT INTO students VALUES
(101, 'Alice', 20, 1),
(102, 'Bob', 21, 2),
(103, 'Charlie', 22, 1),
(104, 'David', 23, 3),
(105, 'Eve', 19, 2);

-- Step 6: Subquery Example 1: List students who belong to the 'CSE' department
SELECT name
FROM students
WHERE dept_id = (
  SELECT dept_id
  FROM departments
  WHERE dept_name = 'CSE'
);

-- Step 7: Subquery Example 2: Get names of departments where students are older
than 21
SELECT DISTINCT dept_name
FROM departments
WHERE dept_id IN (
  SELECT dept_id
  FROM students
  WHERE age > 21
);
```

Expected Output:

Subquery Example 1 Output:

```
+--------+
| name   |
+--------+
| Alice  |
| Charlie|
+--------+
```

Subquery Example 2 Output:

```
+-----------+
| dept_name |
+-----------+
| CSE       |
| MECH      |
+-----------+
```

Experiment (d): Creating Script Files using MySQL Workbench

Aim:

To demonstrate how to **create**, **save**, and **execute SQL script files** in **MySQL Workbench** to create a database, tables, and insert data.

Procedure:

1. **Open MySQL Workbench.**
2. Click on the **"+" icon** to start a new SQL tab (or use `File > New Query Tab`).
3. In the new tab, write SQL code to create a database and table.
4. Save the script by selecting `File > Save Script As` (e.g., `college_script.sql`).
5. Click the **Execute (lightning bolt icon)** to run the script.
6. Refresh the **SCHEMAS** section to see the newly created database and table.

Script File Content (`college_script.sql`):

```
-- Step 1: Create the database
CREATE DATABASE IF NOT EXISTS CollegeDB;

-- Step 2: Use the database
USE CollegeDB;

-- Step 3: Create the 'students' table
CREATE TABLE IF NOT EXISTS students (
    id INT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(100) NOT NULL,
    age INT,
    department VARCHAR(50)
);

-- Step 4: Insert data into the 'students' table
INSERT INTO students (name, age, department) VALUES
('Alice', 20, 'CSE'),
('Bob', 21, 'ECE'),
('Charlie', 22, 'CSE'),
('David', 23, 'MECH');
```

How to Save Script in MySQL Workbench:

1. Go to `File` → `Save Script As`.
2. Name the file (e.g., `college_script.sql`) and choose the location.
3. Click **Save**.

Output:

After executing the script:

- A database named `CollegeDB` is created.
- A table named `students` is added.
- 4 records are inserted into the table.

You can verify using:

```
USE CollegeDB;
SELECT * FROM students;
```

Experiment (e): Creating a Database Directory and Initializing a `database.sql` File for API Integration

---

Aim:

To create a **project database directory** and initialize a `database.sql` file that can be used to set up the database structure and sample data for integration with a backend API.

Procedure:

1. Create a project folder (e.g., `student-api`).
2. Inside the project, create a folder named `database/`.
3. Inside the `database/` folder, create a file named `database.sql`.
4. Write SQL code to:
    - Create the database
    - Create tables
    - Insert initial data
5. Use this SQL file to initialize the database from **MySQL CLI** or **Workbench**.
6. Connect this database to an API using backend code (e.g., Express.js + MySQL).

Directory Structure:

```
student-api/
│
```

```
├── database/
│   └── database.sql
├── app.js
├── package.json
```

SQL Code in `database/database.sql`:

```
-- Create database
CREATE DATABASE IF NOT EXISTS StudentDB;

-- Use the created database
USE StudentDB;

-- Create 'students' table
CREATE TABLE IF NOT EXISTS students (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    age INT,
    department VARCHAR(50)
);

-- Insert sample data
INSERT INTO students (name, age, department) VALUES
('Alice', 20, 'CSE'),
('Bob', 22, 'ECE'),
('Charlie', 21, 'MECH');
```

How to Run the SQL File:

✅ From MySQL Command Line Client:

```
mysql -u root -p < database/database.sql
```

✅ From MySQL Workbench:

1. Open `database.sql`.
2. Click **Execute** (⚡ lightning bolt).

Integration Example (Optional): Connecting API in `app.js`):

```
const express = require('express');
const mysql = require('mysql2');
const app = express();
const PORT = 3000;

const connection = mysql.createConnection({
  host: 'localhost',
  user: 'root',
  password: '',
  database: 'StudentDB'
});

connection.connect((err) => {
  if (err) throw err;
```

```
  console.log('Connected to MySQL Database');
});

app.get('/students', (req, res) => {
  connection.query('SELECT * FROM students', (err, results) => {
    if (err) throw err;
    res.json(results);
  });
});

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

Output:

- The database and table are created successfully.
- The `database.sql` file can be reused for API projects.
- The API can fetch data from the MySQL database using a RESTful endpoint.