

SKILL-2 FSAD

2.A

Aim: To implement the working of React States

Description: **State** in ReactJS is an object that holds information that can change over the lifetime of a component. It allows React components to manage and respond to user interactions, input changes, or system events. When the state of a component changes, React re-renders the component to reflect the new state in the user interface.

Key Features of State

1. **Mutable:** State can be updated using the `setState` method in class components or the `useState` hook in functional components.
2. **Private:** State is local to the component and cannot be directly accessed or modified by other components.
3. **Triggers Re-render:** When state changes, React automatically re-renders the component to reflect the new state.

Types of State in React

1. **Local State:**
 - Managed within a single component.
 - Example: Form inputs, toggling a button, or a counter value.
2. **Global State:**
 - Shared across multiple components.
 - Managed using libraries like Context API, Redux, or Zustand.
3. **Server State:**
 - Data fetched from an external server (e.g., via APIs).
 - Typically handled using libraries like React Query, SWR, or directly with `useState`.
4. **Derived State:**
 - Computed based on existing state or props.
 - Example: Filtering a list of items based on a search query.

Using State in React

1. Class Component State (Legacy)

```
import React, { Component } from "react";
class Counter extends Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0, // Initial state
    };
  }
}
```

```

increment = () => {
  this.setState({ count: this.state.count + 1 }); // Updating state
};

render() {
  return (
    <div>
      <h1>Count: {this.state.count}</h1>
      <button onClick={this.increment}>Increment</button>
    </div>
  );
}
}

```

export default Counter;

- **this.setState** is used to update the state.
- React re-renders the component when the state changes.

2. Functional Component State (Modern Approach)

The modern way to manage state in functional components is using the **useState hook**.

```

import React, { useState } from "react";
const Counter = () => {
  const [count, setCount] = useState(0); // Declare state
  const increment = () => {
    setCount(count + 1); // Update state
  };
  return (
    <div>
      <h1>Count: {count}</h1>
      <button onClick={increment}>Increment</button>
    </div>
  );
};

```

export default Counter;

- **useState** returns an array with:
 - The current state value (count).
 - A function to update the state (setCount).

State in Action

Let's look at a typical state flow:

1. **Initial State:**
 - A state variable is initialized (e.g., `const [count, setCount] = useState(0)`).
 - The UI renders based on this state.
2. **State Update:**
 - A user action (e.g., button click) triggers an event handler.
 - The event handler calls the state updater function (e.g., `setCount`).

3. **Re-render:**

- React detects the state change and re-renders the component with the updated state.
- The new state is displayed in the UI.

Rules for Using State

1. **Do Not Modify State Directly:**

// WRONG

```
state.count = 5;
```

// RIGHT

```
this.setState({ count: 5 }); // Class Component
```

```
setCount(5); // Functional Component
```

2. **State Updates May Be Asynchronous:** Use the previous state when updating based on current state:

// Correct approach for dependent updates

```
setCount((prevCount) => prevCount + 1);
```

3. **State is Isolated:** Each component manages its own state, independent of other components.

Why Use State?

- To build interactive user interfaces.
- To handle dynamic behavior like toggling, updating lists, or managing user inputs.
- To enable React's **declarative programming style**, where the UI automatically updates in response to state changes.

CODE:

sum-n-numbers/

```
├── public/
│   └── index.html
├── src/
│   ├── components/
│   │   └── SumCalculator.jsx
│   ├── App.jsx
│   ├── index.jsx
│   └── styles.css
├── package.json
└── README.md
```

Index.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Sum of N Numbers</title>
  </head>
  <body><center>
    <div id="root"></div>
  </center>
</body>
</html>
```

SumCalculator.jsx

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

const SumCalculator = () => {
  const [n, setN] = useState(0); // State to store the value of 'n'
  const [sum, setSum] = useState(0); // State to store the calculated sum

  const calculateSum = () => {
    const number = parseInt(n, 10);
    if (isNaN(number) || number < 0) {
```

```

    alert("Please enter a valid non-negative number.");
    return;
  }
  const result = (number * (number + 1)) / 2; // Formula for the sum of the first 'n' numbers
  setSum(result);
};

return (
  <div className="sum-calculator">
    <div>
      <label htmlFor="numberInput">Enter a value for N: </label>
      <input
        id="numberInput"
        type="number"
        value={n}
        onChange={(e) => setN(e.target.value)}
        placeholder="Enter a number"
      />
    </div>
    <button onClick={calculateSum} className="calculate-button">
      Calculate Sum
    </button>
    <h2>Sum: {sum}</h2>
  </div>
);
};

export default SumCalculator;

```

App.jsx

```

import React from "react";
import SumCalculator from "../components/SumCalculator";
function App() {
  return (
    <div className="App">
      <h1>Sum of N Numbers</h1>
      <SumCalculator />

    </div>
  );
}

export default App;

```

index.jsx

```

import React from "react";

```

```
import ReactDOM from "react-dom";
import App from "./App";
import "./styles.css"; // Importing global styles
```

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

styles.css

```
body {
  font-family: Arial, sans-serif;
  background-color: #f4f4f9;
  margin: 0;
  padding: 0;
}
```

```
.App {
  text-align: center;
  padding: 20px;
}
```

```
.sum-calculator {
  margin-top: 20px;
}
```

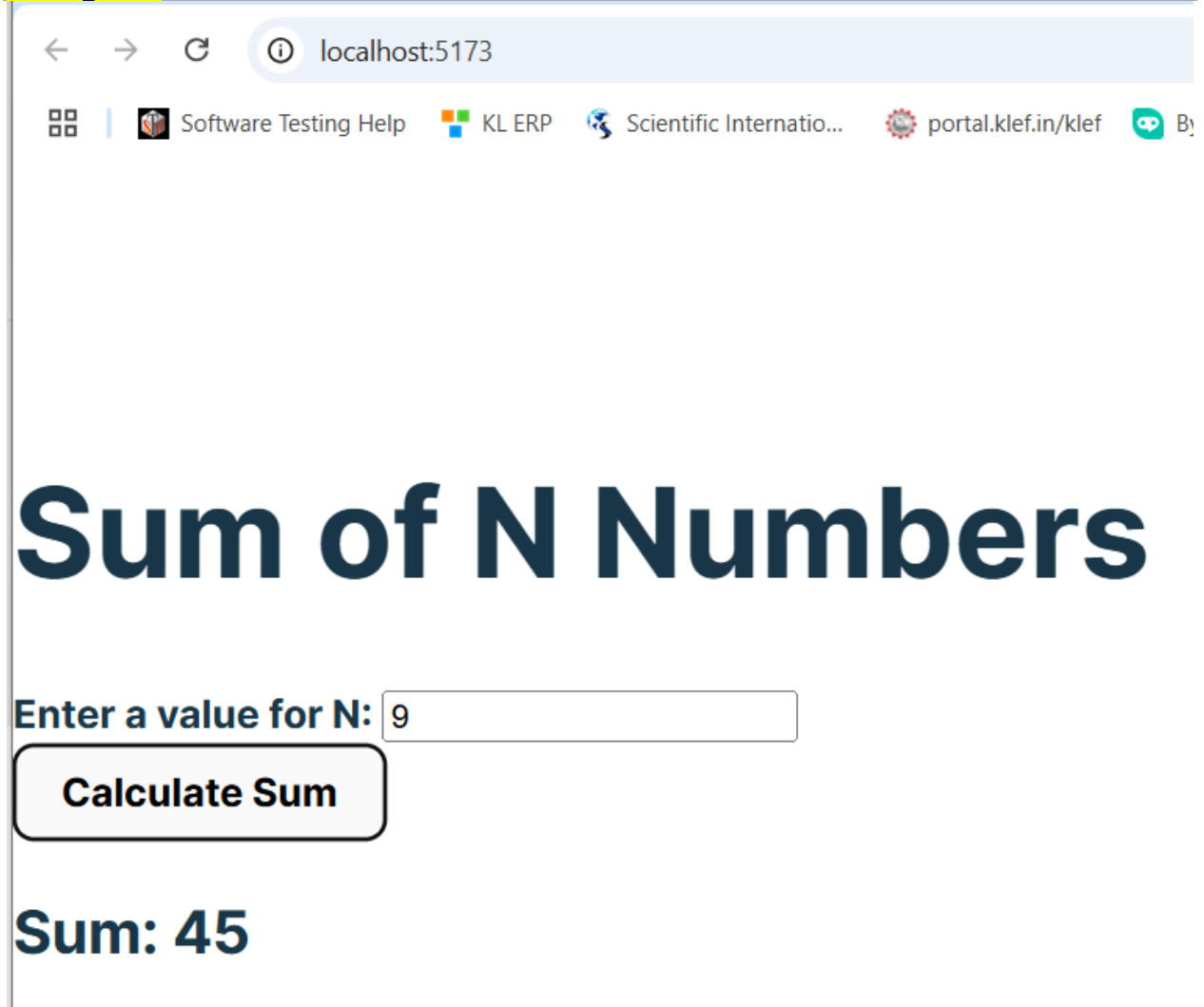
```
input {
  padding: 5px;
  font-size: 16px;
  margin-right: 10px;
}
```

```
.calculate-button {
  padding: 5px 10px;
  font-size: 16px;
  background-color: #4caf50;
  color: white;
  border: none;
  border-radius: 5px;
  cursor: pointer;
}
```

```
.calculate-button:hover {
  background-color: #45a049;
}
```

```
}  
  
h2 {  
  margin-top: 20px;  
}
```

Output:



The screenshot shows a web browser window with the address bar displaying 'localhost:5173'. The browser's tab bar includes several open tabs: 'Software Testing Help', 'KL ERP', 'Scientific Internatio...', 'portal.klef.in/klef', and 'By'. The main content area of the browser displays a web application titled 'Sum of N Numbers' in a large, bold, dark blue font. Below the title, there is a text input field with the placeholder text 'Enter a value for N:' and the number '9' entered. To the left of the input field is a rounded rectangular button with the text 'Calculate Sum'. Below the input field and button, the text 'Sum: 45' is displayed in a large, bold, dark blue font.

localhost:5173

Software Testing Help KL ERP Scientific Internatio... portal.klef.in/klef By

Sum of N Numbers

Enter a value for N:

Calculate Sum

Sum: 45

2.B

Aim: To implement the working of React Prop's

Description:

Props (short for properties) in ReactJS are used to pass data from a parent component to a child component. Props are immutable, meaning they cannot be modified by the child component that receives them. They are used to make components reusable and dynamic by allowing them to accept varying inputs.

Key Features of Props

1. **Immutable:** Props cannot be changed by the receiving component; they are read-only.
2. **Unidirectional Data Flow:** Props flow from the parent component to the child component in a one-way direction.
3. **Reusable Components:** Props enable the creation of dynamic components by passing different values for different instances.
4. **JavaScript Objects:** Props are passed as objects and can include any data type (strings, numbers, arrays, functions, etc.).

Props Syntax

Props are passed to a component as attributes within JSX.

```
<ComponentName propName="value" />
```

The child component receives props as an argument in its function (for functional components) or through this.props (in class components).

Example: Using Props

Parent Component:

```
import React from "react";
import Greeting from "../Greeting";
```

```
const App = () => {
  return (
    <div>
      <Greeting name="Alice" />
      <Greeting name="Bob" />
    </div>
  );
};
```

```
export default App;
```


Child Component:

```
import React from "react";

const Greeting = (props) => {
  return <h1>Hello, {props.name}!</h1>;
};

export default Greeting;
```

Output:

Copy code

Hello, Alice!

Hello, Bob!

How Props Work

1. Parent to Child Communication:

- The parent component passes data to the child via props.
- The child uses the props to render content or perform actions.

2. Immutable Nature:

- The child component cannot modify the props directly.
- This ensures a predictable data flow and avoids unintended side effects.

Props in Functional Components

Functional components receive props as a parameter.

```
const ComponentName = (props) => {
  return <div>{props.value}</div>;
};
```

To simplify, props can be destructured:

jsx

Copy code

```
const ComponentName = ({ value }) => {
  return <div>{value}</div>;
};
```

Props in Class Components

In class components, props are accessed using `this.props`.

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

class Greeting extends Component {
  render() {
    return <h1>Hello, {this.props.name}!</h1>;
  }
}

export default Greeting;
```

Passing Multiple Props

You can pass multiple props to a component:

```
<Greeting name="Alice" age={25} isStudent={true} />
```

jsx

Copy code

```
const Greeting = ({ name, age, isStudent }) => {  
  return (  
    <div>  
      <h1>Hello, {name}!</h1>  
      <p>Age: {age}</p>  
      <p>Student: {isStudent ? "Yes" : "No"}</p>  
    </div>  
  );  
};
```

Default Props

You can define default props for a component. These are used when a prop is not provided.

```
const Greeting = ({ name }) => {  
  return <h1>Hello, {name}!</h1>;  
};
```

```
Greeting.defaultProps = {  
  name: "Guest",  
};
```

```
export default Greeting;
```

Output when no name prop is passed:

Copy code

Hello, Guest!

Prop Types

React allows you to validate the types of props using the prop-types library.

1. Install the library:

```
npm install prop-types
```

2. Define Prop Types:

```
import PropTypes from "prop-types";
```

```
const Greeting = ({ name, age }) => {  
  return (  
    <div>  
      <h1>Hello, {name}!</h1>  
      <p>Age: {age}</p>  
    </div>  
  );  
};
```

```
};

Greeting.propTypes = {
  name: PropTypes.string.isRequired,
  age: PropTypes.number,
};

Greeting.defaultProps = {
  age: 18,
};

export default Greeting;
```

- name is a required prop of type string.
- age is optional and defaults to 18 if not provided.

Passing Functions as Props

Props can also be used to pass functions for handling events in the child component.

Example:

```
const App = () => {
  const showMessage = (message) => {
    alert(message);
  };

  return <Button onClick={() => showMessage("Button clicked!")} />;
};
```

jsx

Copy code

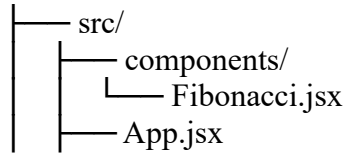
```
const Button = ({ onClick }) => {
  return <button onClick={onClick}>Click Me</button>;
};
```

Advantages of Props

1. **Reusable Components:** Components can be made dynamic and reusable with props.
2. **Unidirectional Data Flow:** Makes the app predictable and easier to debug.
3. **Customizable UI:** Props allow customizing components by passing different values.
4. **Scalable Architecture:** Encourages separation of concerns by managing data in the parent and logic in the child.

CODE:

fibonacci-series/



Fibonacci.jsx

```
import React from "react";

const Fibonacci = ({ n }) => {
  // Function to calculate Fibonacci series
  const calculateFibonacci = (num) => {
    if (num <= 0) return [];
    if (num === 1) return [0];
    if (num === 2) return [0, 1];

    const series = [0, 1];
    for (let i = 2; i < num; i++) {
      series.push(series[i - 1] + series[i - 2]);
    }
    return series;
  };

  const fibonacciSeries = calculateFibonacci(n);

  return (
    <div>
      <h2>First {n} Terms of Fibonacci Series:</h2>
      <p>{fibonacciSeries.join(", ")}</p>
    </div>
  );
};

export default Fibonacci;
```

App.jsx

```
import React, { useState } from "react";
import Fibonacci from "./components/Fibonacci";

function App() {
```

```

const [n, setN] = useState(10); // State to store the user input

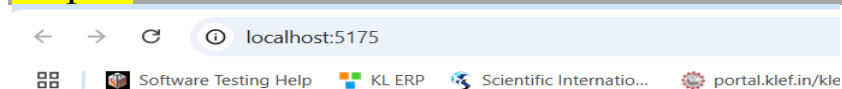
const handleChange = (e) => {
  const value = parseInt(e.target.value, 10);
  setN(isNaN(value) || value < 0 ? 0 : value); // Ensure the value is non-negative
};

return (
  <div>
    <h1>Fibonacci Series</h1>
    <label>
      Enter the number of terms:
      <input
        type="number"
        value={n}
        onChange={handleChange}
        min="0"
        placeholder="Enter a number" />
    </label>
    { /* Passing the number of terms as a prop */ }
    <Fibonacci n={n} />
  </div>
);
}

export default App;

```

output:



Fibonacci Series

Enter the number of terms:

First 10 Terms of Fibonacci Series:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34

Note: collect full project code from : https://github.com/subbu-neo/SpringBoot-Projects/upload/main/FSAD_Practical

Result:

Thus, in the above programs of ReactJS State and Prop's successfully executed without errors in VS Code Editor