**DAY-1**

**React JS :**

**Components:**

**Components are the building blocks of a React application. They are reusable, independent pieces of UI that can be composed together to create complex interfaces. Components can be either functional or class-based.**

* **Class Components**
* **Functional Components**
* **Higher Order Components(HOC)**
* **Nested Components**

**Functional Components:**

Functional components in React are simple JavaScript functions that accept props as input and return React elements to render UI. They are stateless, meaning they don't manage their own state, and are typically used for presenting UI based on props received from parent components. Functional components have become more powerful with the introduction of React hooks, allowing them to use state and other React features.

import React,{useState} from 'react';

const FunctionalComponent = () => {

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

    const incrementCount = () => {

      setCount(count + 1);

    };

    const decrementCount = () => {

      setCount(count - 1);

    };

    const reset = () => {

        setCount(0);

    }

  return (

    <div style={{backgroundColor:'rgb(199, 121, 121)', padding:'15px', margin:'10px 0'}}>

        <center>

        <h1 style={{fontWeight:'bold', fontSize : '30px', paddingBottom:'10px',textDecoration:'underline' }} >Functional Component Counter</h1>

        <h2><span style={{fontWeight:'bolder',fontSize:'30px',paddingRight:'5px'}}>Count</span>: <span style={{fontWeight:'bold',fontSize:'30px',color:'blueviolet'}}> {count}</span></h2>

    <button onClick={incrementCount}>Increment</button>

    <button onClick={decrementCount}>Decrement</button>

    <button onClick={reset}>Reset</button>

        </center>

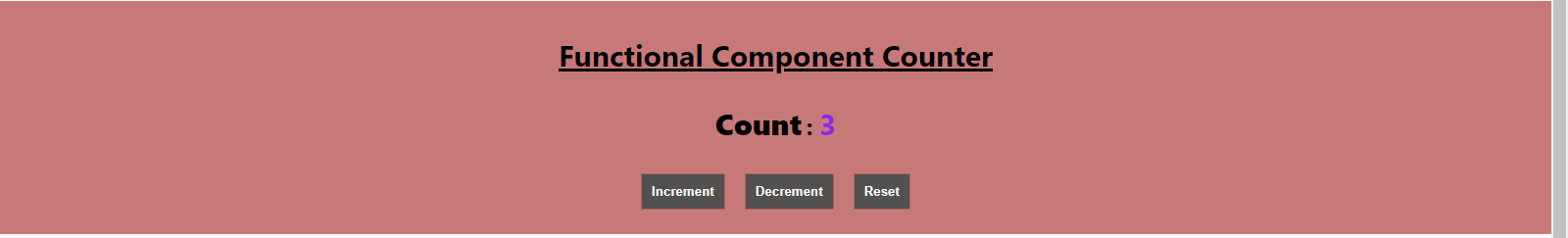
  </div>

  )

}

export default FunctionalComponent

This is an example of Functional Component in this example I have created a counter stating increment, decrement and reset of the count. It is an arrow function in which we have used “Use State” Hook and we have took count as set State and we wrote a increment and decrement and reset function for the Use State “Count”.



**Nested Components:**

Nested components in React refer to the practice of composing components within other components, forming a hierarchical structure. This approach enables developers to break down complex UIs into smaller, reusable, and manageable parts. Each nested component can encapsulate its own functionality, state, and rendering logic, promoting code reusability and maintainability. By nesting components, developers can create modular, well-organized, and easily understandable React applications.

import React, { useState } from 'react';

const NestedComponent = () => {

  const [isLoggedIn, setIsLoggedIn] = useState(false);

  const handleSubmit = () => {

    setIsLoggedIn(true);

  };

  return (

    <div style={{backgroundColor:'beige',padding:'40px', margin:'10px 0', border:'3px solid red' }}>

      {isLoggedIn ? (

        <div>

          <center>

          <h1 style={{color:'forestgreen'}}>Login Successful!</h1>

          </center>

        </div>

      ) : (

        <center>

        <form onSubmit={handleSubmit}>

          <input type="text" placeholder="Username" /> <br />

          <input type="password" placeholder="Password" /> <br />

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

        </form>

        </center>

      )}

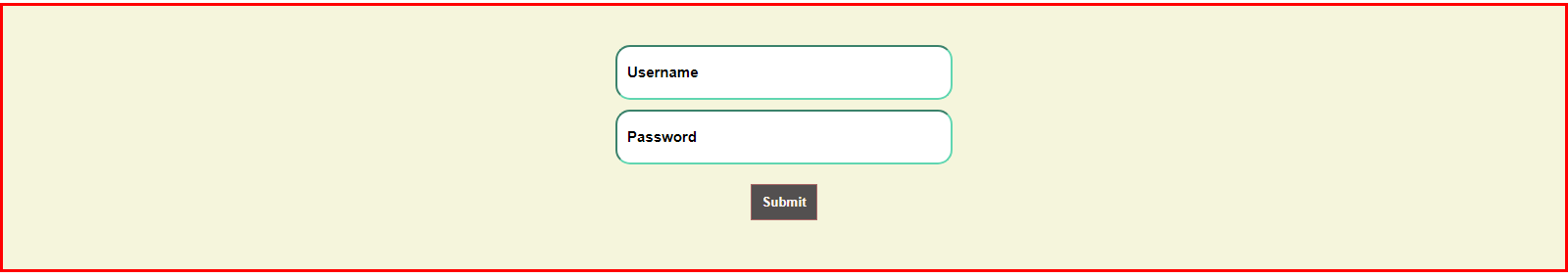
    </div>

  );

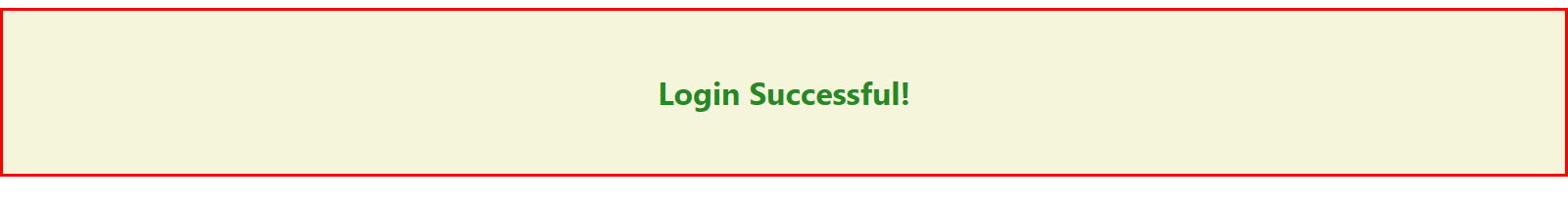
};

export default NestedComponent;

This an example of Nested Component Where we have Used Two Components where <h1>Login Successful</h1> is one Component and <form></form> is another component here if we fill the details in Form component and click on submit it will redirect to the other component and shows Login Successful



After Clicking Submit Button we will get Login Successful



We will get the Output like This.

**Higher Order Components (HOC’s):**

Higher-Order Components (HOCs) in React are functions that take a component and return a new component with enhanced functionality. They enable code reuse, abstraction of logic, and separation of concerns by allowing common functionality to be applied to multiple components. HOCs enhance composability and maintainability by promoting a modular and reusable design pattern.

import React, { Component } from 'react';

import { ToastContainer, toast } from 'react-toastify';

import 'react-toastify/dist/ReactToastify.css';

function withForm(Component) {

  return class extends Component {

    state = { formData: {} };

    handleChange = (e) => {

      const { name, value } = e.target;

      this.setState(prevState => ({

        formData: { ...prevState.formData, [name]: value }

      }));

    };

    handleSubmit = (e) => {

        e.preventDefault();

      const { formData } = this.state;

      toast.success('Username: ' + formData.username + '\n Password: ' + formData.password);

    };

    render() {

      return (

        <Component

          formData={this.state.formData}

          handleChange={this.handleChange}

          handleSubmit={this.handleSubmit}

          {...this.props}

        />

      );

    }

  };

}

class MyForm extends Component {

  render() {

    const { handleChange, handleSubmit } = this.props;

    return (

        <center>

            <form onSubmit={handleSubmit}>

                <ToastContainer />

                <input type="text" name="username" onChange={handleChange} placeholder="Username" /> <br />

                <input type="password" name="password" onChange={handleChange} placeholder="Password" /> <br />

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

            </form>

        </center>

    );

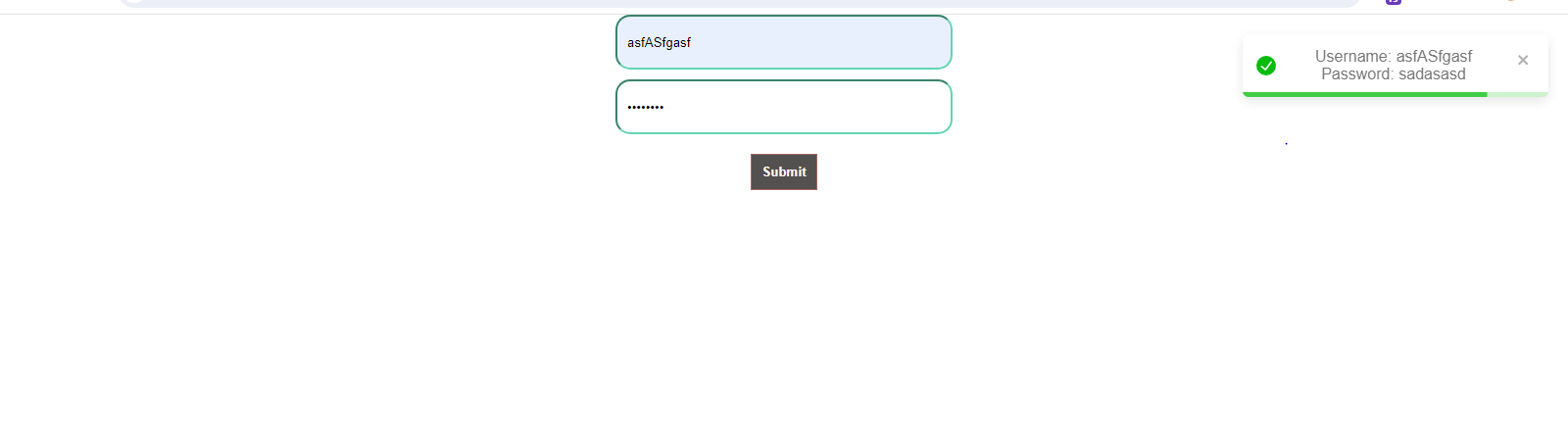
  }

}

const FormComponent = withForm(MyForm);

export default FormComponent;

In this HOC we have wrapped our form into HOC With Form here in this we have used Toast message to show the form inputs we have entered in toast after clicking the submit button.



**DAY-2**

**State in React:**

In React, "state" refers to an internal data storage mechanism that allows components to keep track of changing data. When the state of a component changes, react automatically re-renders the component to reflect the updated state. State is managed within a component using the useState hook (for functional components) or by extending the React.Component class and using this.state (for class components).

import React, { useState } from 'react';

function Form() {

  const [name, setName] = useState('');

  const [age, setAge] = useState('');

  const handleSubmit = (e) => {

    e.preventDefault();

    console.log(`Name: ${name}, Age: ${age}`);

    setName('');

    setAge('');

  };

  return (

    <div>

      <center>

      <h2>Submit Form</h2>

      <form onSubmit={handleSubmit}>

        <label>

          Name:

          <input

            type="text"

            value={name}

            onChange={(e) => setName(e.target.value)}

          />

        </label>

        <br />

        <label>

          Age:

          <input

            type="number"

            value={age}

            onChange={(e) => setAge(e.target.value)}

          />

        </label>

        <br />

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

      </form>

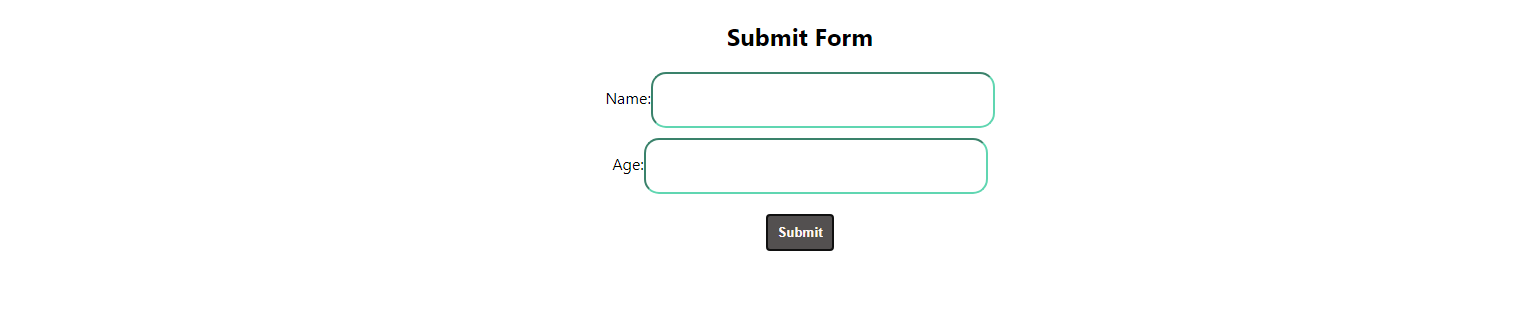
      </center>

    </div>

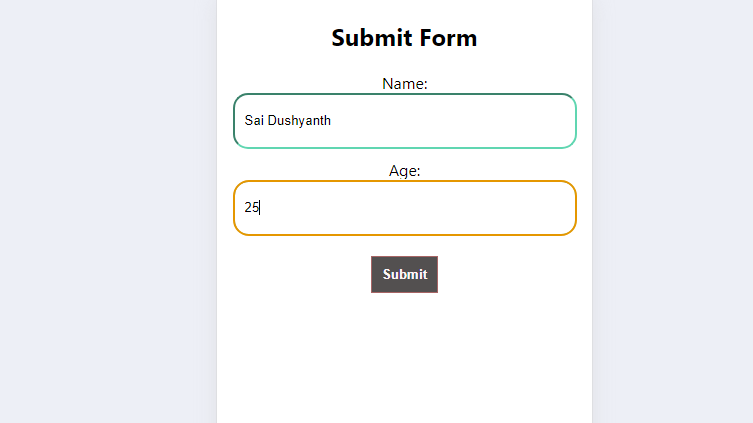
  );

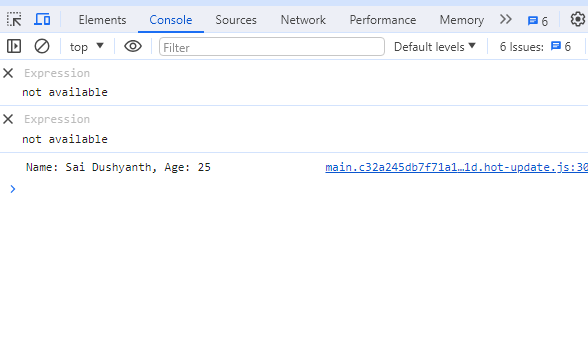
}

export default Form;



Here in this Component we have used useState Hook and we have Two state ‘Name’ and ‘Age’ Which store the value which we enter in the input field. The useState hook initializes name and age to empty strings.When the user types in the input fields, the corresponding state is updated using the onChange event handler. Here we are logging our data to the Console by using console.log action in handleSubmit function.





The form includes input fields for name and age, as well as a submit button. When the user clicks the submit button or presses Enter while focused on an input field, the form's onSubmit event is triggered, calling the handleSubmit function.

**JSX and JS:**

|  |  |
| --- | --- |
| **Differences** | |
| **JSX** | **JS** |
| JSX (JavaScript XML):  Syntax Extension: JSX is a syntax extension for JavaScript. It allows you to write HTML-like code within your JavaScript files, making it easier to define the structure of your UI components in React. | Regular JavaScript (JS):  Core Language: JavaScript is the core programming language of the web. It provides the logic and functionality for your React components, such as event handling, state management, and data manipulation. |
| Concise and Readable: JSX provides a more concise and readable way to define React elements compared to using React.createElement() calls directly. | Versatility: Regular JavaScript is used for a wide range of tasks beyond defining UI components, such as defining functions, working with arrays and objects, making HTTP requests, and more. |
| Familiarity: JSX syntax resembles HTML, which makes it easier for web developers, particularly those coming from an HTML/CSS background, to work with React code. Transformation: JSX code is transformed into regular JavaScript function calls (e.g., React.createElement()) by tools like Babel before being interpreted by the browser. | No Syntax Extension: Unlike JSX, regular JavaScript does not have built-in support for defining HTML-like elements. When working without JSX, you create React elements using React.createElement() function calls directly, which can be more verbose and less intuitive than JSX. |

**Life Cycle Methods:**

**In React, lifecycle methods are special methods that are automatically called at specific points in the lifecycle of a component. These methods allow you to hook into various stages of a component's existence, such as when it is first created, updated, or destroyed, and perform actions accordingly.**

**There are three main phases in the lifecycle of a React component:**

* **Mounting:** This phase occurs when a component is being inserted into the DOM for the first time.
* **Updating:** This phase occurs when a component is being re-rendered due to changes in its props or state.
* **Unmounting:** This phase occurs when a component is being removed from the DOM.

**Component Mounting**

* Class Component: componentDidMount()
* Functional Component: useEffect(() => {}, [])
* Component Updating:
* Class Component: componentDidUpdate()
* Functional Component: useEffect(() => {})
* Component Unmounting:
* Class Component: componentWillUnmount()
* Functional Component: useEffect(() => { return () => {} })
* import React, { useState, useEffect } from 'react';
* const LifeCycleFunctional = () => {
* const [time, setTime] = useState(0);
* const [isRunning, setIsRunning] = useState(false);
* useEffect(() => {
* let timerId;
* if (isRunning) {
* timerId = setInterval(() => {
* setTime((prevTime) => prevTime + 10); // Increment by 10 milliseconds
* }, 10); // Update every 10 milliseconds
* }
* return () => {
* clearInterval(timerId);
* };
* }, [isRunning]);
* const handleStart = () => {
* setIsRunning(true);
* };
* const handleStop = () => {
* setIsRunning(false);
* };
* const handleReset = () => {
* setTime(0);
* setIsRunning(false);
* };
* const formatTime = (milliseconds) => {
* const minutes = Math.floor(milliseconds / 60000);
* const remainingSeconds = Math.floor((milliseconds % 60000) / 1000);
* const remainingMilliseconds = Math.floor((milliseconds % 1000) / 10);
* return `${minutes.toString().padStart(2, '0')}:${remainingSeconds.toString().padStart(2, '0')}.${remainingMilliseconds.toString().padStart(2, '0')}`;
* };

* return (
* <div>
* <center style={{padding: "25px"}}>
* <h1>Stopwatch</h1>
* <p style={{color:'Forestgreen',fontWeight:'bold', fontSize:'20px'}}>Time : <span style={{color:'violet',fontStyle:'oblique'}}> {formatTime(time)} </span></p>
* <button onClick={handleStart} disabled={isRunning}>
* Start
* </button>
* <button onClick={handleStop} disabled={!isRunning}>
* Stop
* </button>
* <button onClick={handleReset}>
* Reset
* </button>
* </center>
* </div>
* )
* }
* export default LifeCycleFunctional

Here's an example of a stopwatch component using the useEffect hook to start and stop the stopwatch.

This example demonstrates how you can use the useEffect hook to implement complex behavior, such as starting and stopping a stopwatch, in a functional component.



The component renders the elapsed time formatted as minutes and seconds, along with buttons to start, stop, and reset the stopwatch.

Clicking the "Start" button sets isRunning to true, starting the stopwatch. Clicking the "Stop" button sets isRunning to false, stopping the stopwatch. Clicking the "Reset" button resets the elapsed time to 0 and stops the stopwatch.

**Hooks:**

**use Context Hook:** “Allows functional components to consume values from a React context”.

* Used for accessing values from a React context within functional components.
* Allows you to consume values provided by a Context. Provider higher in the component tree.

**ColorContextComponent:**

import React, { createContext, useContext, useState } from 'react';

const ColorContext = createContext();

export const ColorProvider = ({ children }) => {

  const [color, setColor] = useState('black');

  return (

    <ColorContext.Provider value={{ color, setColor }}>

      {children}

    </ColorContext.Provider>

  );

};

export const useColor = () => useContext(ColorContext);

export default ColorContext;

createContext: Used to create a new context.

useContext: Hook used to consume a context's value.

useState: Hook used to create a state variable and its updater function. This code sets up a color-related context in a React application, providing a way to manage and share the color state and its setter function across components. The ColorProvider component acts as a provider for the color context, while the useColor hook allows components to consume the color-related context value.

**Circle.jsx:**

import React from 'react';

import { useColor } from './ColorContext';

const Circle = () => {

  const { color } = useColor();

  return (

    <center>

      <div

        style={{

          width: '200px',

          height: '200px',

          borderRadius: '50%',

          backgroundColor: color,

          marginBottom:'20px'

        }}

      ></div>

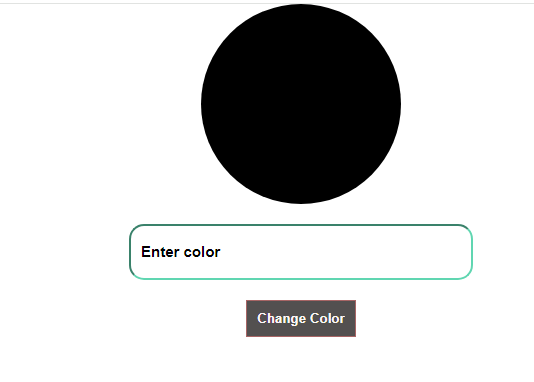
    </center>

  );

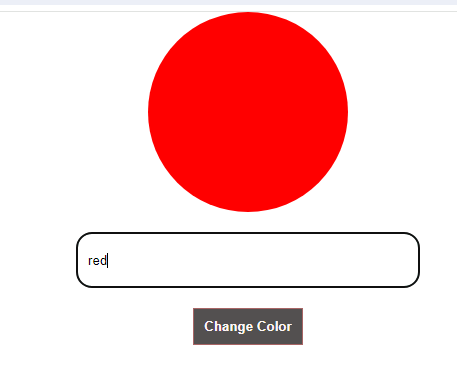
};

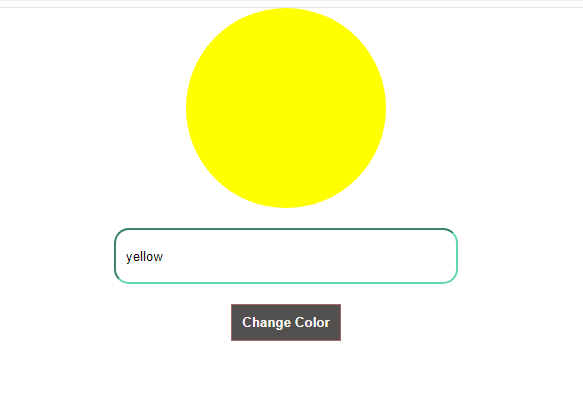
export default Circle;

Circle is a functional component that represents a circle element. Inside the component, we use the useColor hook to access the color state from the color context. We use object destructuring to extract the color from the object returned by the useColor hook. The Circle component is a simple functional component that renders a circle element. It uses the useColor hook to dynamically set the circle's background color based on the color obtained from the color context.



Here by default the circle will be in black colour when we type any colour in the input field it will changes its colour regarding the input given by user.





**DAY-3**

**‘useReducer()’ Hook:**

It is familiar to the REDUX.

* Redux:- Use ro store the state values in Redux store
* useReducer:- Use to store the state values to local const variable.

To install Redux we have dependencies

* npm install redux
* npm install react-redux
* npm install redux-thunk

these are the dependencies.

**useReudcer():-**

It is a built-in hook used for managing complex state logic in functional components. It is an alternative to useState hook when the state logic is more complex and involves multiple sub-values (or) when the next state depends on the prev one .

**Syntax:-**

Const[state , dispatch] = useReducer(reducer , initialState);

Here in reducer hook we have two arguments

* **Reducer:-** It specifies how the state should be updated based on the dispatched actions.
* **initialState:-** The initial state of the reducer.

import React, { useReducer } from 'react';

const initialState = {

  cart: []

};

const cartReducer = (state, action) => {

  switch (action.type) {

    case 'ADD\_ITEM':

      return { ...state, cart: [...state.cart, action.item] };

    case 'REMOVE\_ITEM':

      const indexToRemove = state.cart.findIndex(item => item.id === action.id);

      if (indexToRemove === -1) return state; // If item not found, return current state

      return {

        ...state,

        cart: [...state.cart.slice(0, indexToRemove), ...state.cart.slice(indexToRemove + 1)]

      };

    default:

      return state;

  }

};

const ShoppingCart = () => {

  const [state, dispatch] = useReducer(cartReducer, initialState);

  const calculateTotalPrice = () => {

    return state.cart.reduce((total, item) => total + item.price, 0);

  };

  const addItemToCart = (item) => {

    dispatch({ type: 'ADD\_ITEM', item });

  };

  const removeItemFromCart = (id) => {

    dispatch({ type: 'REMOVE\_ITEM', id });

  };

  return (

    <div>

    <center>

      <h2>Shopping Cart</h2>

      <p>Total Price: ₹{calculateTotalPrice()}</p>

      <button onClick={() => addItemToCart({ id: 1, name: 'Product 1', price: 10 })}>

        Add Product 1

      </button>

      <button onClick={() => addItemToCart({ id: 2, name: 'Product 2', price: 20 })}>

        Add Product 2

      </button>

      <ul>

        {state.cart.map(item => (

          <li key={item.id}>

            {item.name} - ₹{item.price}

            <button onClick={() => removeItemFromCart(item.id)}>Remove</button>

          </li>

        ))}

      </ul>

    </center>

    </div>

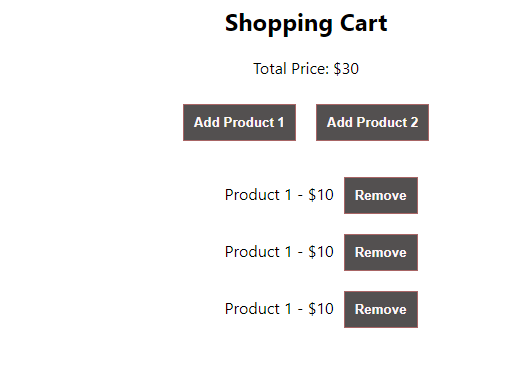
  );

};

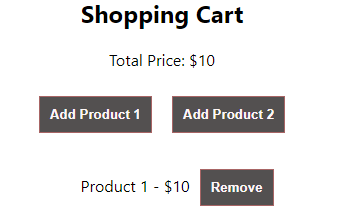
export default ShoppingCart;

This example demonstrates how to use useReducer to manage state in a shopping cart application. The reducer function handles state updates in a predictable and centralized manner, making it easier to manage complex state logic.

We define types for the state (State) and actions (Action) used in the reducer function. The reducer function cartReducer takes a State and an Action as arguments and returns a new State based on the action type. We create a functional component ShoppingCart that initializes the state using useReducer hook with an initial state containing an empty cart.Inside the component, we define functions calculateTotalPrice, addItemToCart, and removeItemFromCart to perform calculations and dispatch actions to update the cart state.We render the items in the cart as a list with a button to remove each item, as well as the total price of all items in the cart.We provide buttons to add two sample products to the cart, demonstrating the functionality of adding items.



Here when we are clicking the Add product button the items are getting added and the total prize was getting calculated and if we click on remove the item will get removed and the price will get updated automatically using useReducer().



Here I have clicked on remove button the items got removed and the total prize got updated.

**useRef() Hook:-**

useRef is used to take the values from the user and assign the value taken from the user to a variable properly. Both useRef() and useState() are similar both takes the values from user and assign it to a variable but the actual difference is.

* **useState():-** It will re-render when the content change and updated in UI.
* **useRef():-** It doesn’t notify you when its content changes. Mutating the (.current) property doesn’t cause a re-render. Using useRef can be beneficial when you need to access DOM elements imperatively, store mutable values across renders, or keep track of previous state values without causing re-renders

I have an example of simple calculator

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

const Calculator = () => {

    const [result, setResult] = useState('');

    const inputRef = useRef(null);

    const handleButtonClick = (value) => {

      setResult(prevResult => prevResult + value);

    };

    const calculateResult = () => {

      setResult(eval(result));

    };

    const clearInput = () => {

      setResult('');

    };

  return (

    <div className="calculator">

    <center>

        <h2>Calculator</h2>

        <input

        type="text"

        value={result}

        ref={inputRef}

        onChange={() => {}}

readOnly

      />

      <div>

        <button onClick={() => handleButtonClick('7')}>7</button>

        <button onClick={() => handleButtonClick('8')}>8</button>

        <button onClick={() => handleButtonClick('9')}>9</button>

        <button onClick={() => handleButtonClick('+')}>+</button>

      </div>

      <div>

        <button onClick={() => handleButtonClick('4')}>4</button>

        <button onClick={() => handleButtonClick('5')}>5</button>

        <button onClick={() => handleButtonClick('6')}>6</button>

        <button onClick={() => handleButtonClick('-')}>-</button>

      </div>

      <div>

        <button onClick={() => handleButtonClick('1')}>1</button>

        <button onClick={() => handleButtonClick('2')}>2</button>

        <button onClick={() => handleButtonClick('3')}>3</button>

        <button onClick={() => handleButtonClick('\*')}>\*</button>

      </div>

      <div>

        <button onClick={() => handleButtonClick('0')}>0</button>

        <button onClick={() => handleButtonClick('.')}>.</button>

        <button onClick={calculateResult}>=</button>

        <button onClick={() => handleButtonClick('/')}>/</button>

      </div>

      <div>

        <button onClick={clearInput}>Clear</button>

      </div>

    </center>

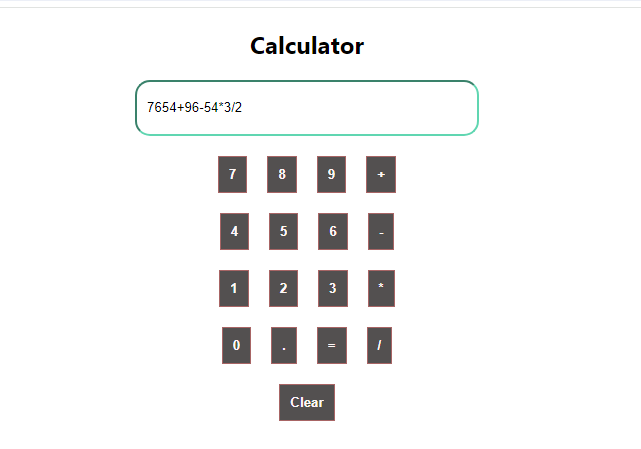
    </div>

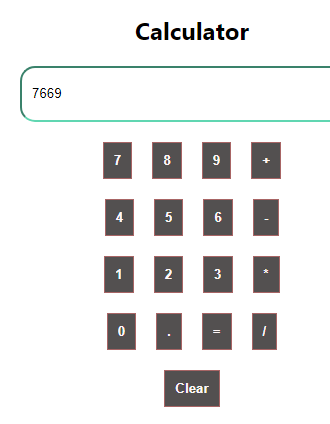
  );

};

export default Calculator;

Here the user can only enter the value by pressing the buttons cannot directly enter the number in input field.





We maintain the current input value and result in the component state using useState.We create a ref using useRef and assign it to the input element to access its value imperatively.We define functions to handle button clicks (handleButtonClick), calculate the result (calculateResult), and clear the input (clearInput).Each button click updates the input value accordingly.The "=" button calculates the result using JavaScript's eval function (for simplicity, but it's not recommended for production).The "Clear" button clears the input field.We use readOnly attribute to make the input field read-only so that users cannot directly type into it.

**Conditional Rendering:-**

Conditional rendering in React refers to the practice of rendering different components or UI elements based on certain conditions. It allows you to control what gets displayed to the user dynamically, depending on the state of your application or user interactions.

There are several ways to implement conditional rendering in React:

* **Using If-Else Statements:** You can use regular JavaScript if-else statements within your JSX to conditionally render components.
* **Using the Ternary Operator:** The ternary operator condition ? trueCase : falseCase can be used for concise conditional rendering.
* **Using Logical && Operato**r**:** When you want to render a component only if a certain condition is true, you can use the logical AND (&&) operator.
* **Using Conditional Rendering with Logical Operators:** You can also combine conditions using logical operators (&&, ||) to render components conditionally based on multiple conditions.
* **Using Switch Statements:** For more complex conditional rendering, you might use switch statements.

Each of these methods allows you to conditionally render components based on the state of your application, user input, or other factors. React re-renders components automatically when their state or props change, so any changes to the conditions will trigger the appropriate rendering updates.

import React, { useState } from 'react';

const LoginComponent = ({ onLogin }) => {

  const [username, setUsername] = useState('');

  const [password, setPassword] = useState('');

  const handleLogin = () => {

    // Perform authentication logic here, for simplicity let's just check if username and password are not empty

    if (username.trim() !== '' && password.trim() !== '') {

      onLogin();

    } else {

      alert('Please enter username and password');

    }

  };

  return (

    <div>

      <h2>Login</h2>

      <input type="text" placeholder="Username" value={username} onChange={(e) => setUsername(e.target.value)} />

      <input type="password" placeholder="Password" value={password} onChange={(e) => setPassword(e.target.value)} />

      <button onClick={handleLogin}>Login</button>

    </div>

  );

};

const RandomGame = ({ onNext }) => {

    const [randomNumber, setRandomNumber] = useState(generateRandomNumber());

    const [guess, setGuess] = useState('');

    const [message, setMessage] = useState('');

    const [attempts, setAttempts] = useState(0);

    function generateRandomNumber() {

      return Math.floor(Math.random() \* 100) + 1;

    }

    const handleInputChange = (event) => {

      setGuess(event.target.value);

    };

    const handleSubmit = (event) => {

      event.preventDefault();

      const userGuess = parseInt(guess);

      if (isNaN(userGuess)) {

        setMessage('Please enter a valid number.');

      } else if (userGuess === randomNumber) {

        setMessage(`Congratulations! You guessed the correct number in ${attempts + 1} attempts.`);

        onNext();

      } else if (userGuess < randomNumber) {

        setMessage('Too low. Try again.');

      } else {

        setMessage('Too high. Try again.');

      }

      setAttempts(attempts + 1);

      setGuess('');

    };

    const handleRestart = () => {

      setRandomNumber(generateRandomNumber());

      setGuess('');

      setMessage('');

      setAttempts(0);

    };

    return (

      <div>

        <center>

          <h1>Random Number Guessing Game</h1>

          <p>Guess a number between 1 and 100:</p>

          <form onSubmit={handleSubmit}>

            <input

              type="number"

              value={guess}

              onChange={handleInputChange}

              min="1"

              max="100"

              required

            />

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

          </form>

          <p>{message}</p>

          {message && (

            <button onClick={handleRestart}>Restart Game</button>

          )}

        </center>

      </div>

    );

  };

const RockPaperScissors = () => {

  const [playerChoice, setPlayerChoice] = useState('');

  const [computerChoice, setComputerChoice] = useState('');

  const [result, setResult] = useState('');

  const handlePlayerChoice = (choice) => {

    setPlayerChoice(choice);

    const choices = ['rock', 'paper', 'scissors'];

    const randomIndex = Math.floor(Math.random() \* choices.length);

    const computerChoice = choices[randomIndex];

    setComputerChoice(computerChoice);

    determineWinner(choice, computerChoice);

  };

  const determineWinner = (playerChoice, computerChoice) => {

    if (playerChoice === computerChoice) {

      setResult('It\'s a tie!');

    } else if (

      (playerChoice === 'rock' && computerChoice === 'scissors') ||

      (playerChoice === 'paper' && computerChoice === 'rock') ||

      (playerChoice === 'scissors' && computerChoice === 'paper')

    ) {

      setResult('You win!');

    } else {

      setResult('Computer wins!');

    }

  };

  return (

    <div>

      <center>

        <h1>Rock, Paper, Scissors</h1>

        <div>

          <button onClick={() => handlePlayerChoice('rock')}>Rock</button>

          <button onClick={() => handlePlayerChoice('paper')}>Paper</button>

          <button onClick={() => handlePlayerChoice('scissors')}>Scissors</button>

        </div>

        <div>

          <p>Your choice: {playerChoice}</p>

          <p>Computer's choice: {computerChoice}</p>

          <p>{result}</p>

        </div>

      </center>

    </div>

  );

};

const App = () => {

  const [loggedIn, setLoggedIn] = useState(false);

  const [showRandomNumberGame, setShowRandomNumberGame] = useState(false);

  const handleLogin = () => {

    setLoggedIn(true);

    setShowRandomNumberGame(true);

  };

  const handleNext = () => {

    setShowRandomNumberGame(false);

  };

  return (

    <div>

      {!loggedIn && <LoginComponent onLogin={handleLogin} />}

      {loggedIn && showRandomNumberGame && <RandomGame onNext={handleNext} />}

      {loggedIn && !showRandomNumberGame && <RockPaperScissors />}

    </div>

  );

};

export default App;

In this example it ensures that different components are displayed to the user based on their interactions and the state of the application.

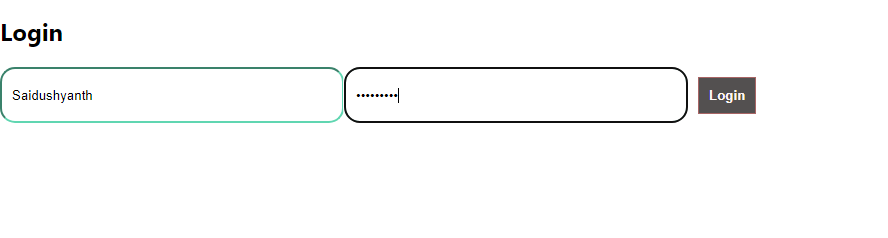
**LoginComponent:**This component renders a login form with inputs for username and password.It uses the useState hook to manage the state of username and password.When the user clicks the "Login" button, the handleLogin function is called.handleLogin checks if both username and password are not empty. If they are not empty, it calls the onLogin prop function, which is passed from the parent component (in this case, App). Otherwise, it shows an alert asking the user to enter both username and password.

**RandomNumberGameComponent:**This component implements a simple game where the user guesses a random number between 1 and 10.It uses useState to manage the state of guess, randomNumber, and message.When the user clicks the "Guess" button, the handleGuess function is called.handleGuess checks if the entered guess is equal to the randomly generated number. If it is, it displays a success message and calls the onNext prop function, which is used to move to the next game. Otherwise, it displays a message asking the user to try again or to enter a valid number.

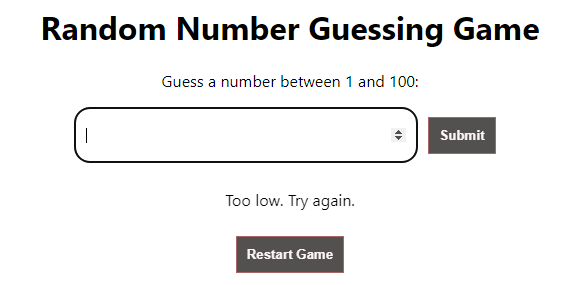
**RockPaperScissors:**This component implements the Rock, Paper, Scissors game.It uses useState to manage the state of playerChoice, computerChoice, and result.When the user clicks one of the buttons (Rock, Paper, or Scissors), the handlePlayerChoice function is called with the corresponding choice.handlePlayerChoice sets the player's choice, generates a random choice for the computer, and determines the winner based on the choices.The result is displayed below the buttons.

**App:**This is the main component that manages the application state and renders child components conditionally based on that state.It uses useState to manage the state of loggedIn and showRandomNumberGame.When the user logs in successfully (handleLogin), loggedIn is set to true and showRandomNumberGame is set to true, which displays the RandomNumberGameComponent.When the user completes the random number game (handleNext), showRandomNumberGame is set to false, which hides the RandomNumberGameComponent and shows the RockPaperScissors component instead.

**LoginComponent:-**



**RandomNumberGame:-**



**RockPaperScisors:-**

