**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>

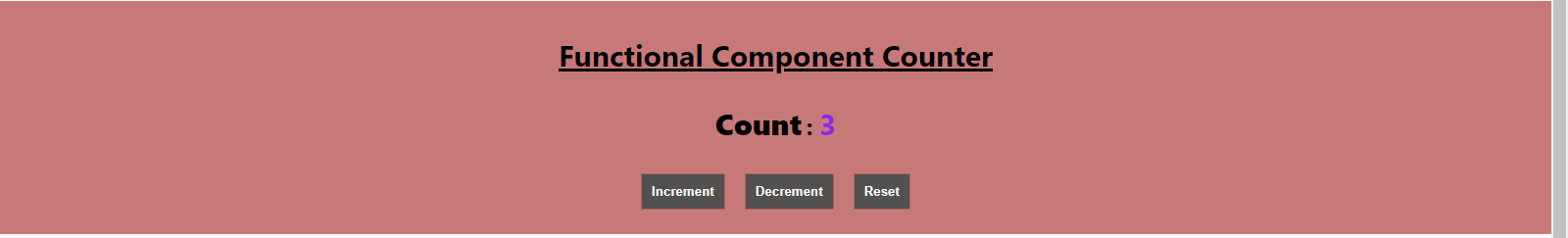
  </div>

  )

}

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>

      )}

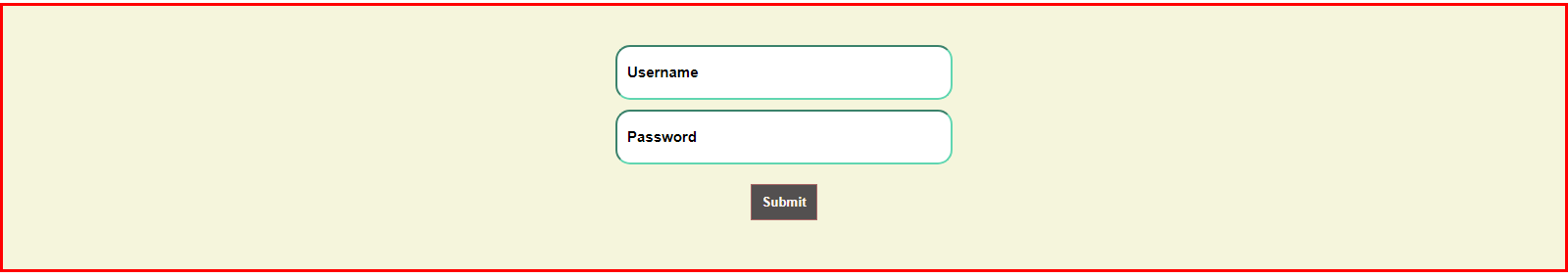
    </div>

  );

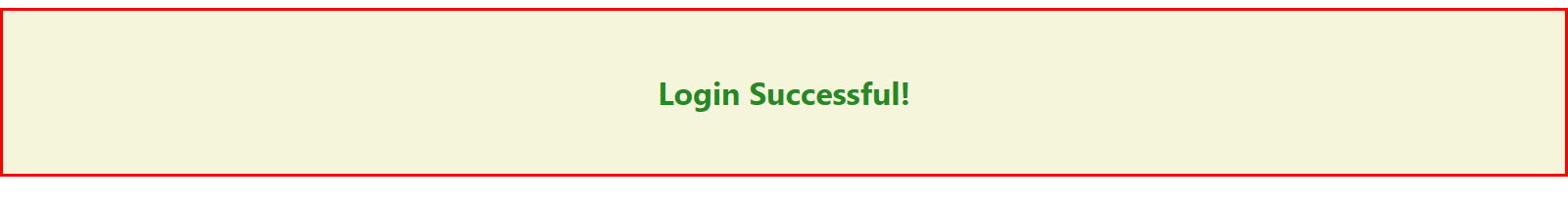
};

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>

    );

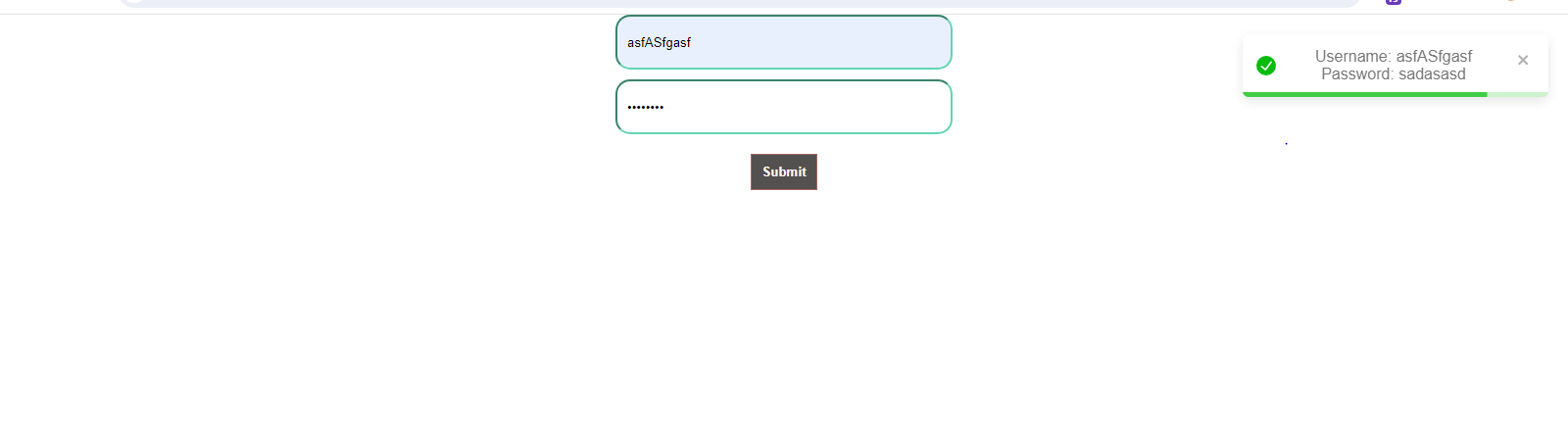
  }

}

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>

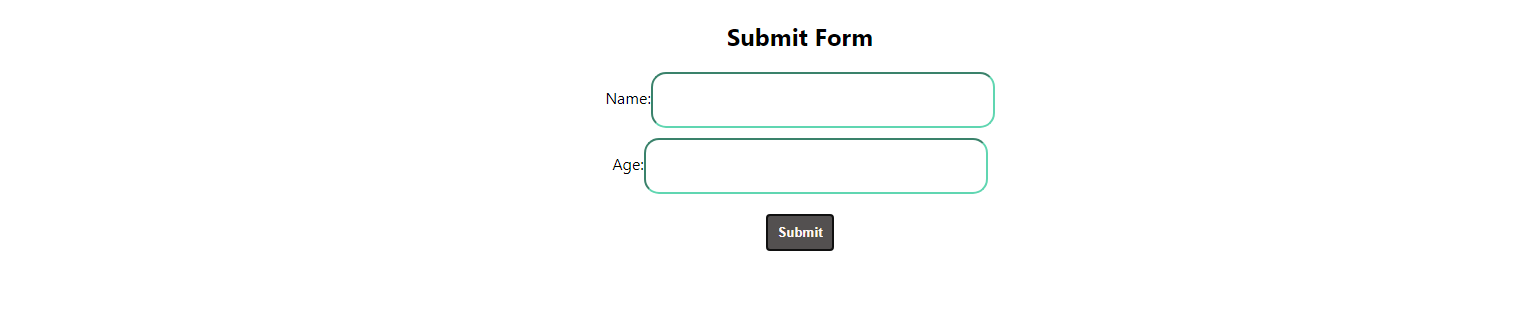
      </center>

    </div>

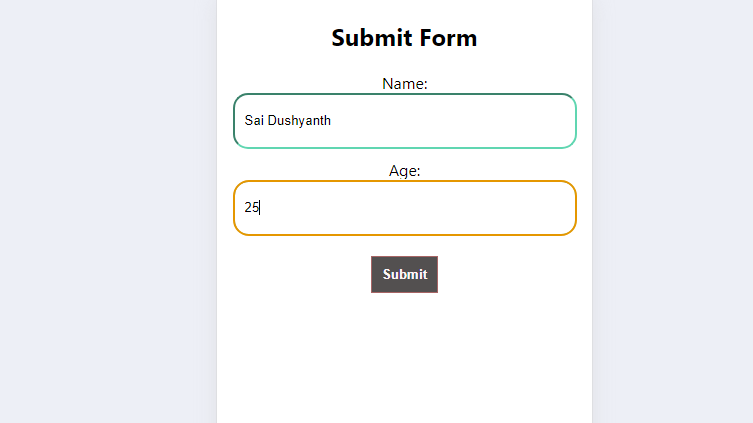
  );

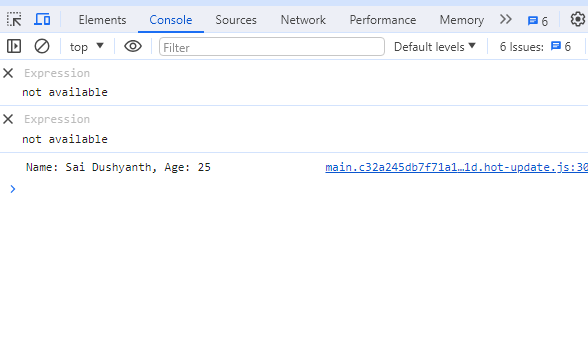
}

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>

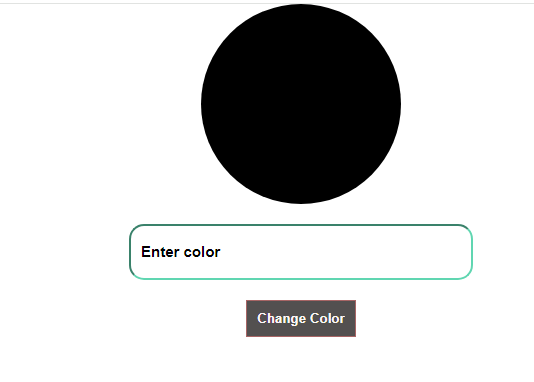
    </center>

  );

};

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.

