**REACT JS**

React is developed by the Facebook.

React will work on two methods which are Class Based and Functional Based.

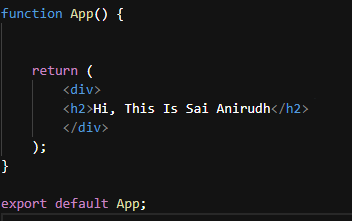
React is Very helpful to create SPA (Single Page Applications) by using the JavaScript Libraries.

React Creates the Virtual Dom in Memory.

**Components**

Components is a nothing a particular piece of code which can be reusable in the project.  
Components are two types which are class component and functional component.

Examples for the functional components



Here we can create multiple components  
  
**Nested Component**

In nested components, multiple functions are defined within a single file. These functions are then utilized within another function or component.



**Stateless Functional Component**

In a stateless functional component, we cannot manage internal state, but we can access data passed down from parent components through props.

const Number = ({ count }) => {

return <h2>The Count is {count}</h2>;

};

**State full Functional Component**

In a state full functional component, we can declare the state values and we can store and handle the state values

import React, { useState } from 'react';

const Counter = () => {

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

const incrementCount = () => {

setCount(prevCount => prevCount + 1);

};

const decrementCount = () => {

setCount(prevCount => prevCount - 1);

};

return (

<div>

<h2>Counter</h2>

<p>Count: {count}</p>

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

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

<Number count={count} />

</div>

);

};

export default Counter;

**Higher Order Component**

HOC is function that takes a component as an input and returns a new component with additional functionality.

Here I am taking that validation for login

HOC.Jsx

import React from 'react';

function HOC(Component) {

return class extends React.Component {

render() {

const { auth } = this.props;

if (auth) {

return <div>Please login</div>;

}

return (

<div>

<Component name='Anirudh' />

</div>

);

}

};

}

export default HOC;

in here if the auth is false its return please login if it the auth is true it returns as   
Hi Welcome Anirudh here I am send my name through props

App.jsx

import React from 'react'

import HOC from './hoc'

function App(props) {

return (

<div>

<h1>Hi Welcome {props.name}</h1>

</div>

)

}

export default HOC(App)

**Day -2**

**Props**

It is the concept of the passing the data from Parent Component to Child Component.

The data will be one-way process that means the data will receive to child component from that particular parent component.

**Passing Props:** Props are passed from parent components to child components. Parent components can pass data or functions as props to their children when they are being rendered.

**Receiving Props:** Child components receive props as arguments to their functional component or as properties of the this. Props object in class components.

**Immutable Data:** Props are immutable, meaning they cannot be modified by the child component. They are read-only within the component that receives them.

**Component Customization:** Props allow components to be customizable and reusable. By passing different props, you can render the same component in different ways with different data.

// FormComponent.js

import React from 'react';

const FormComponent = ({ formData, onInputChange, onSubmit }) => {

  return (

    <form onSubmit={onSubmit}>

      <div>

        <label>Username:</label>

        <input type="text" name="username" value={formData.username} onChange={onInputChange} />

      </div><div>

<label>Password:</label>

        <input type="password" name="password" value={formData.password} onChange={onInputChange} />

      </div>

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

    </form>

  );

}

export default FormComponent;

// ParentComponent.js

import React, { useState } from 'react';

import FormComponent from './FormComponent';

const ParentComponent = () => {

  const [formData, setFormData] = useState({

    username: '', password: ''   });

  const handleSubmit = (e) => {

    e.preventDefault();

    // Handle form submission logic

    console.log('Form submitted:', formData);

  };

  const handleInputChange = (event) => {

    // Update form data in parent component state

    setFormData({

      ...formData,

      [event.target.name]: event.target.value

    });

  };

  return (

    <div>

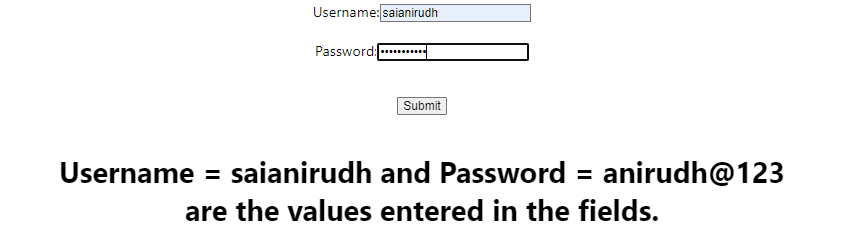
      <FormComponent formData={formData} onInputChange={handleInputChange} onSubmit={handleSubmit}/>

      <h1>Username = {formData.username} and Password = {formData.password} are the values entered in the fields.</h1>

    </div>

  ); }

export default ParentComponent;



In the Parent Component, we initialize a state variable formData using useState, providing default values for username and password. We define functions handleSubmit and handleInputChange to manage form submission and input changes, respectively. The FormComponent receives these state variables and functions as props to handle form rendering, input changes, and submission.

**Use State**

UseState is the Hook, that allows us to track state in a function component.

import React, { useState } from 'react';

const Counter = () => {

  // Define a state variable 'count' initialized to 0

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

  return (

    <div>

        <center>

            <h2>Count: {count}</h2>

            {/\* Button to increment the count \*/}

            <div style={{display:'flex', justifyContent:'space-around'}}>

                <button style={{padding:'10px 20px',fontSize:'20px'}} onClick={() => setCount(count + 1)}>Increment</button>

                <button  style={{padding:'10px 20px',fontSize:'20px'}} onClick={() => setCount(count - 1)}>Decrement</button>

                <button  style={{padding:'10px 20px',fontSize:'20px'}} onClick={() => setCount(0)}>Reset</button>

            </div>

        </center>

    </div>

  );

}

export default Counter;



const [Count, setCount] = useState ()

setCount is a function provided by the useState hook. It is used to update the state variable Count.

useState () is called without an initial value. This means that the initial state of Count will be undefined. It's generally recommended to provide an initial value to useState () for better predictability of your component's behavior.

**Use Effect**

useEffect is another React Hook used for managing side effects in functional components. Side effects are actions that occur outside the scope of the component, such as data fetching, DOM manipulation, or subscriptions.

The useEffect Hook allows you to perform side effects in your components.

useEffect accepts two arguments. The second argument is optional.

useEffect(<function>, <dependency>)

import { useState, useEffect } from "react";

function Counter1() {

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

  const [calculation, setCalculation] = useState(0);

  useEffect (() => {

    setCalculation(() => count \* 2);

  }, [count]); // <- add the count variable here

  return (

    <div style={{marginTop:'10%'}}>

        <center>

            <h1>Count: {count}</h1>

            <button style={{padding:'10px 20px',fontSize:'20px'}} onClick={() => setCount((c) => c + 1)}>Add Number</button>

            <h1>Calculation: {calculation}</h1>

        </center>

    </div>

  );

}

export default Counter1;

useEffect Hook that is dependent on a variable. If the count variable updates, the effect will run again.

**Use Context:**

useContext is a React Hook that provides a way to consume context within a functional component. Context in React is a mechanism for passing data through the component tree without having to pass props manually at every level.

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

const ThemeContext = React.createContext();

const ThemeProvider = ({ children }) => {

  const [theme, setTheme] = useState('light');

  const toggleTheme = () => {

    setTheme(prevTheme => (prevTheme === 'light' ? 'dark' : 'light'));

  };

  return (

    <ThemeContext.Provider value={{ theme, toggleTheme }}>

      {children}

    </ThemeContext.Provider>

  );

};

// Component consuming the context value

const ThemeConsumer = () => {

  const { theme } = useContext(ThemeContext);

  return (

    <div style={{ backgroundColor: theme === 'light' ? '#f0f0f0' : '#333', color: theme === 'light' ? '#333' : '#f0f0f0', padding: '20px', margin: '20px 0' }}>

      <p>Current Theme: {theme}</p></div>

  );

};

// App component

const App = () => {

  return (

    <div style={{marginTop:'20px',marginBottom: '40px'}}>

          <center>

              <ThemeProvider>

                  <div>

                      <h1>Theme Change Example</h1>

                      <ThemeConsumer />

                      <ThemeToggler />

                  </div></ThemeProvider></center>

    </div>

  );};

// Component to toggle theme

const ThemeToggler = () => {

  const { toggleTheme } = useContext(ThemeContext);

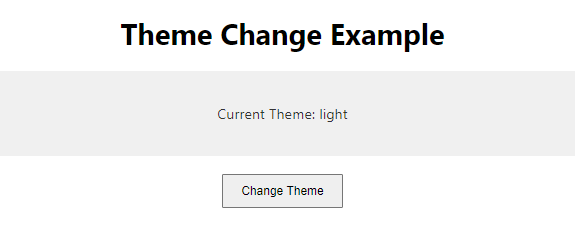
  return (

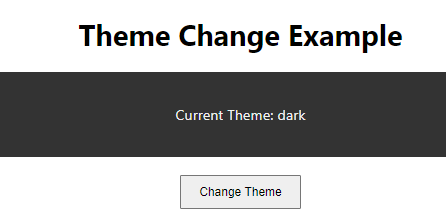
    <button style={{padding:'10px 20px'}} onClick={toggleTheme}>Change Theme</button>

  );

};

export default App;





The ThemeProvider component manages the theme state and provides it to its children via context. The ThemeConsumer component consumes the theme value and renders accordingly. Finally, the ThemeToggler component allows toggling between light and dark themes by invoking the toggleTheme function obtained from the context.

**USE REF**

useRef hook is a way to persistently store a reference to a DOM element or a value across renders without causing a re-render when it changes.

**Preserving Values Between Renders:** Normally, when a component re-renders, all its variables are reset. useRef allows you to create a reference that persists across renders.

**Accessing DOM Elements:** If you need to interact with a DOM element directly, like accessing its properties or methods, useRef provides a way to create a reference to that element.

import React, { useRef } from 'react';

function MyComponent() {

  const inputRef = useRef(null);

  function focusInput() {

    inputRef.current.focus();

  }

  return (

    <div style={{marginTop:'5%',marginBottom:'5%'}}>

        <center>

            <input style={{padding:'10px'}} ref={inputRef} type="text" />

            <br></br>

            <button style={{marginTop:25,padding:'20px 10px',fontSize:'20px'}} onClick={focusInput}>Focus the Input</button>

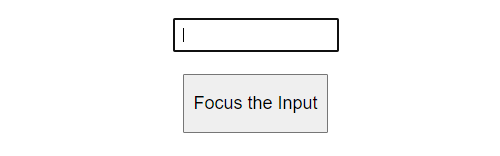
        </center>

    </div>

  );

}

export default MyComponent ;



inputRef is a reference created using useRef. We attach this reference to the input element using the ref attribute. Later, when the button is clicked, focusInput is called, which focuses on the input element using inputRef.current. focus(). This way, we can interact with the DOM element directly without triggering a re-render of the component.

**Use Reducer**

Use Reducer is a hook that is used for state management. It is alternative to useState. Internally useReducer using useState hook only.

The reducer function contains your custom state logic and the initial State can be a simple value but generally will contain an object. The useReducer Hook returns the current state and a dispatch method.

useReducer is another hook in React that helps manage state in a more complex and predictable way, especially when the state logic involves multiple sub-values or when the next state depends on the previous one.

**State Management:** Like useState, useReducer manages state within functional components. However, it's more suitable for managing state objects that have complex logic associated with them.

**Predictable State Updates:** With useReducer, state updates are more predictable and manageable, especially when dealing with complex state transitions. It's akin to using a reducer function in Redux.

**Action-based State Updates:** Instead of directly setting the state like useState, useReducer relies on a reducer function. This function takes the current state and an action, and returns a new state based on that action.

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

// Reducer function

function reducer(state, action) {

  switch (action.type) {

    case 'login':

      return { ...state, isLoggedIn: true };

    case 'logout':

      return { ...state, isLoggedIn: false };

    case 'deleteRow':

      return { ...state, rows: state.rows.filter(row => row.id !== action.payload) };

    default:

      throw new Error();

  }

}

// Dummy table data

const initialRows = [

  { id: 1, username: 'user1', password: 'pass1' },

  { id: 2, username: 'user2', password: 'pass2' },

  { id: 3, username: 'user3', password: 'pass3' }

];

function LoginTable() {

  const [state, dispatch] = useReducer(reducer, {

    isLoggedIn: false,

    rows: initialRows

  });

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

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

  const handleLogin = () => {

    // Dummy authentication logic, replace with your own

    const user = state.rows.find(row => row.username === username && row.password === password);

    if (user) {

      dispatch({ type: 'login' });

    } else {

      alert('Invalid username or password');

    }

  };

  const handleLogout = () => {

    dispatch({ type: 'logout' });

  };

  const handleDeleteRow = id => {

    dispatch({ type: 'deleteRow', payload: id });

  };

  return (

    <div>

      {state.isLoggedIn ? (

              <div style={{marginTop:'8%',marginBottom:'8%'}}>

                  <center>

                      <p>Welcome, {username}!</p>

                      <button onClick={handleLogout}>Logout</button>

                      <table>

                          <thead>

                              <tr>

                                  <th>ID</th>

                                  <th>Username</th>

                                  <th>Password</th>

                                  <th>Action</th>

                              </tr>

                          </thead>

                          <tbody>

                              {state.rows.map(row => (

                                  <tr key={row.id}>

                                      <td>{row.id}</td>

                                      <td>{row.username}</td>

                                      <td>{row.password}</td>

                                      <td><button onClick={() => handleDeleteRow(row.id)}>Delete</button></td>

                                  </tr>

                              ))}

                          </tbody>

                      </table>

                  </center>

              </div>

      ) : (

            <div style={{display:'flex',justifyContent:'center',marginTop:'8%',marginBottom:'8%'}}>

                    <div style={{display:'flex',flexDirection:'column',gap:'20px'}}>

                        <input style={{padding:'10px'}} type="text" placeholder="Username" value={username} onChange={e => setUsername(e.target.value)} />

                        <input style={{padding:'10px'}}   type="password" placeholder="Password" value={password} onChange={e => setPassword(e.target.value)} />

                        <button style={{padding:'10px 20px'}}  onClick={handleLogin}>Login</button>

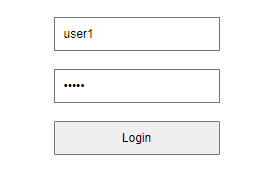
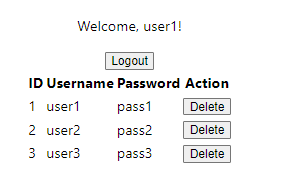
                    </div>

            </div>)}

    </div>

  );}

export default LoginTable;

The reducer function handles state updates for login, logout, and row deletion. The login form allows users to input their username and password. Upon successful login, the user is authenticated, and the table with rows is displayed. Each row in the table has a "Delete" button that, when clicked, dispatches an action to delete that row from the table. Authentication is done based on the username and password entered by the user.

**Use Callbacks**

use Callback and React. Memo are both performance optimization tools in React. They help to improve the efficiency of your components by reducing unnecessary re-renders.

You can use use Callback to memorize callbacks and React. Memo to memorize entire components, ensuring that only the necessary parts of your application re-render when state or props change.

import React, { useCallback } from 'react';

const ChildComponent = React.memo(function ChildComponent({ onClick }) {

console.log('Child component rendered');

return <button onClick={onClick}>Click me</button>;

});

function ParentComponent() {

const handleClick = useCallback(() => {

console.log('Button clicked');

}, []);

return <ChildComponent onClick={handleClick} />;

}

export default ParentComponent;

ChildComponent is wrapped with react. Memo, ensuring that it only re-renders when its props change. The handleClick function in ParentComponent is memorized using useCallback, preventing it from being recreated on every render of ParentComponent. This combination optimizes performance by reducing unnecessary re-renders.

**Day-4**

**Event handling**

React provides a synthetic event system that wraps the native browser events and provides a consistent interface across different browsers.

**onClick:**

 function handleClick() {

    console.log('Button clicked');

  }

  <button onClick={handleClick}>Click me</button>

This example demonstrates a click event listener attached to a button. When the button is clicked, the handleClick function is called.

**onChange:**

function handleChange(event) {

    console.log('Input value changed:', event.target.value);

  }

  <input type="text" onChange={handleChange} />

Here, an onChange event listener is attached to an input field. The handleChange function is called whenever the value of the input changes.

**onSubmit:**

function handleSubmit(event) {

    event.preventDefault();

    console.log('Form submitted');

  }

  <form onSubmit={handleSubmit}>

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

  </form>

In this example, an onSubmit event listener is added to a form. The handleSubmit function is invoked when the form is submitted. event.preventDefault() is called to prevent the default form submission behavior.

**onMouseOver and onMouseOut:**

 function handleMouseOver() {

    console.log('Mouse over');

  }

  function handleMouseOut() {

    console.log('Mouse out');

  }

  <div onMouseOver={handleMouseOver} onMouseOut={handleMouseOut}>

    Hover over me

  </div>

These events are triggered when the mouse pointer enters or leaves the element, respectively. In this example, handleMouseOver is called when the mouse enters the div, and handleMouseOut is called when the mouse leaves the div.

**onKeyDown:**

 function handleKeyDown(event) {

    console.log('Key pressed:', event.key);

  }

  <input type="text" onKeyDown={handleKeyDown} />

This event is triggered when a key is pressed while the input field is focused. The handleKeyDown function is called with the event object containing information about the key that was pressed.

<button onDoubleClick={handleDoubleClick}>Double click me</button>

<div onMouseOver={handleMouseOver} onMouseOut={handleMouseOut}>Hover over me</div>

<div onMouseEnter={handleMouseEnter} onMouseLeave={handleMouseLeave}>Hover over me</div>

<button onMouseDown={handleMouseDown} onMouseUp={handleMouseUp}>Click and hold me</button>

<input onKeyDown={handleKeyDown} onKeyUp={handleKeyUp} onKeyPress={handleKeyPress} />

<input onFocus={handleFocus} onBlur={handleBlur} />

<input type="text" onChange={handleChange} />

**Click:** Fires when the user clicks an element.

**DoubleClick:** Fires when the user double-clicks an element.

**MouseOver:** Fires when the mouse pointer enters an element.

**MouseOut:** Fires when the mouse pointer leaves an element.

**MouseEnter:** Fires when the mouse pointer enters an element, but doesn't bubble up.

**MouseLeave:** Fires when the mouse pointer leaves an element, but doesn't bubble up.

**MouseDown:** Fires when a mouse button is pressed down over an element.

**MouseUp:** Fires when a mouse button is released over an element.

**KeyDown:** Fires when a key is pressed down while the element is focused.

**KeyUp:** Fires when a key is released while the element is focused.

**KeyPress:** Fires when a key is pressed while the element is focused, but provides character information.

**Submit:** Fires when a form is submitted.

**Reset:** Fires when a form is reset.

**Focus:** Fires when an element gains focus.

**Blur:** Fires when an element loses focus.

**Change:** Fires when the value of an input element changes (e.g., text input, checkbox, radio button).

**Input:** Fires every time the value of an input element changes, regardless of the input method (e.g., typing, pasting).

**React Conditional Rendering**

import React from 'react';

function ConditionalRenderingExample({ isLoggedIn, status, items }) {

  let content1 = isLoggedIn ? <p>User is logged in</p> : <p>User is not logged in</p>;

  let content2 = isLoggedIn && <p>User is logged in</p>;

  let content4;

  switch (status) {

    case 'loading':

      content4 = <p>Loading...</p>;

      break;

    case 'success':

      content4 = <p>Data loaded successfully</p>;

      break;

    case 'error':

      content4 = <p>Error occurred while loading data</p>;

      break;

    default:

      content4 = null;

  }

  let listItems = null;

  if (items && items.length > 0) {

    listItems = items.map(item => <li key={item.id}>{item.name}</li>);

  }

  return (

    <div>

      {content1}

      {content2}

      {content4}

      {listItems && <ul>{listItems}</ul>}

    </div>

  );

}

export default ConditionalRenderingExample;

This React component, ConditionalRenderingExample, demonstrates various techniques for conditional rendering based on the provided props: isLoggedIn, status, and items.

isLoggedIn:

It determines whether the user is logged in or not.

If isLoggedIn is true, content1 displays "User is logged in". Otherwise, it displays "User is not logged in".

content2 uses the logical AND (&&) operator to conditionally render "User is logged in" if isLoggedIn is true.

status:

It represents the status of data loading, which can be 'loading', 'success', or 'error'.

Using a switch statement, content4 is assigned a JSX element based on the value of status.

If status is 'loading', it displays "Loading...". If 'success', it displays "Data loaded successfully". If 'error', it displays "Error occurred while loading data". If none of these, content4 is set to null.

items:

It is an array of items to be rendered as a list.

listItems is initially set to null.

If items exists and has elements (items.length > 0), it maps over each item, generating a list of <li> elements with unique keys based on the id of each item.

Finally, within the component's JSX return statement:

content1, content2, and content4 are rendered directly.

listItems is conditionally rendered within a <ul> element using the logical AND (&&) operator to prevent rendering an empty list when listItems is null.

This component effectively demonstrates conditional rendering based on different states or props, ensuring that the UI adapts dynamically to the changing conditions.

**List**

In React, a "list" typically refers to an array of data that you want to render as a series of components or elements in your user interface. Lists are a common and important concept in web development, as they allow you to dynamically display data in a structured and organized manner.

In React, rendering lists involves using the map() method of JavaScript arrays to iterate over the elements of the array and generate React elements for each item. This allows you to create a series of components or elements based on the data in the array.

import React from 'react';

function ListRenderingExample() {

  // Array for rendering list of numbers

  const numbers = [1, 2, 3, 4, 5];

  // Array of users for rendering list of components

  const users = [

    { id: 1, name: 'Alice' },

    { id: 2, name: 'Bob' },

    { id: 3, name: 'Charlie' }

  ];

  return (

    <div>

      {/\* Rendering list using Array.map() \*/}

      <h2>Rendering list of numbers:</h2>

      <ul>

        {numbers.map(number => (

          <li key={number.toString()}>{number}</li>

        ))}

      </ul>

      {/\* Rendering list of components \*/}

      <h2>Rendering list of users:</h2>

      <ul>

        {users.map(user => (

          <li key={user.id}>

            {user.name} (ID: {user.id})

          </li>

        ))}

      </ul>

    </div>

  );

}

export default ListRenderingExample;

ListRenderingExample renders both a list of numbers and a list of user components.

The list of numbers is generated by mapping over the numbers array and creating a <li> element for each number.

The list of users is created by mapping over the users array and rendering a custom user component (<li>) for each user, displaying their name and ID.

Each list item has a unique key prop assigned to it (number.toString() for the numbers list and user.id for the users list) to help React identify each item uniquely for efficient rendering and updating.