**REACT**

1. **Defination :**

* React is a JavaScript library developed by Facebook for building user interfaces (UIs) for web applications.
* It is used to create interactive UI components that can dynamically update based on data changes without reloading the entire page.
* React follows a component-based architecture, where UIs are broken down into reusable pieces called components.
* React uses a virtual DOM to improve performance by minimizing direct manipulation of the actual DOM. It compares the virtual DOM with the real DOM and only updates the parts that have changed, resulting in faster rendering.

1. **JSX :** JSX is a syntax extension for JavaScript that allows developers to write HTML-like code within JavaScript.
2. **Installation :** npx create-react-app app-name
3. **Version :** 18
4. **Package.json :** keep track the packages installed in you project and there related dependedcy.
5. **Package-lock.json :** keep track the package.json as well as the nternal tree structure of node\_modules.
6. **Public :** All the static file like images , files , css are placed here.
7. **Entry Point Of A React App :** index.js
8. **npm (Node Package Manager) :** npm is the default package manager for Node.js and JavaScript runtime environments. It comes pre-installed with Node.js, allowing developers to manage project dependencies, install packages, and execute scripts easil
9. **npx (Node Package Executer) :** npx is a tool that comes with npm version 5.2.0 and higher. It allows developers to execute npm packages without installing them globally or locally.
10. **Components :** Basically react have only two components : there is no specifc count for the components
11. Functional component
12. Class Component

But people have consider there own components

1. HOC (high ordered component)
2. Pure Component
3. Controlled Component
4. UnControlled Component

**Functional Component :**

|  |
| --- |
| const FunctionalComponent = (props) => {  return <div>{props.message}</div>;  }; |

**Class Component :**

|  |
| --- |
| import React, { Component } from "react";  class ClassComponent extends Component {      render() {          return (              <>                  <h1>Class Component</h1>              </>          )      }  }  export default ClassComponent; |

1. **State In Class Component :**

|  |
| --- |
| class InternalClassComponent extends Component {      constructor() {  *super*();  *this*.state = {              count: 0          }      }      increaseCounter() {  *this*.setState({ count: *this*.state.count + 1 })      }      decreseCounter() {  *this*.setState({ count: *this*.state.count - 1 });      }      render() {  *return* (              <>                  <h1>Internal Class Component</h1>                  <p>{*this*.state.count}</p>                  <button *type*="button"  *onClick*={() => *this*.decreseCounter()}>Decrement Count -- </button>                  <button *type*="button" *onClick*={() => *this*.increaseCounter()}>Increase Count ++</button>              </>  )      }  } |

1. **Props in class component :**

mechanism for passing data from a parent component to a child component. Props allow you to create reusable components and pass dynamic data or configuration settings to those components.

Note : We cant change the props only at the sender side not at the receiver side.

|  |
| --- |
| *import* React, { Component } *from* "react";  class ClassComponent extends Component {      constructor() {  *super*();  *this*.state = {              fname: "Shreyash",              lname: "Thaware"          };  *// Bind the updateName method to the class instance*  *this*.updateName = *this*.updateName.bind(*this*);      }      updateName(newName) {  *this*.setState({              fname: newName          });      }      render() {  *return* (              <>                  <h1>Class Component</h1>                  {*/\* Pass updateName function as myfunction prop \*/*}                  <InternalClassComponent  *fname*={*this*.state.fname}  *lname*={*this*.state.lname}  *myfunction*={*this*.updateName} *// Pass function reference, don't call it her* />              </>          );      }  }  class InternalClassComponent extends Component {      render() {  *return* (              <>                  <h1>Internal Class Component</h1>                  {*/\* Props \*/*}                  <h3>Props</h3>                  <p>                      {*this*.props.fname} {*this*.props.lname}                  </p>                  {*/\* Call updateName from props \*/*}                  <button  *type*="button"  *onClick*={() => *this*.props.myfunction("Json")}                  >                      Update Name                  </button>              </>          );      }  }  *export* *default* ClassComponent; |

1. **Class Based LifeCycle :**
2. **ComponentDidMount** : componentDidMount is a lifecycle method in React that is invoked immediately after a component is mounted (i.e., inserted into the DOM tree). It is commonly used to perform actions that need to happen only once when the component is first rendered.

**Note** : Render run first and then componentDidMount.But before render the constructor is executed.

|  |
| --- |
| import React, { Component } from "react";  class ClassLifeCycle extends Component {      constructor() {          super();          console.warn("I am Constructor");          this.state = {              count: 0          }      }      componentDidMount() {          console.log("I am ComponentDidMount");      }      render() {          console.log("I am Render");          return (              <div>                  <h2>Class Life Cycle</h2>                  <button type="button" onClick={() => this.setState({ count: this.state.count + 1 })}>Update Count {this.state.count}</button>              </div>          )      }  }  export default ClassLifeCycle; |

1. **ComponentDidUpdate :** componentDidUpdate is a lifecycle method in React that is invoked immediately after updating occurs, but not for the initial render. It is called every time the component re-renders due to changes in props or state. This method is commonly used to perform actions in response to component updates, such as fetching new data when props change or updating the DOM based on state changes.

**Note :** in this render run first and the componentDidUpdate.

|  |
| --- |
| import React, { Component } from "react";  class ClassLifeCycle extends Component {      constructor() {          super();          console.warn("I am Constructor");          this.state = {              count: 0          }      }      componentDidMount() {          console.log("I am ComponentDidMount");      }      componentDidUpdate(prevProps, prevState, snapShot) {          console.log("I am ComponentDidUpdate");          console.log("This will return the previous state : ", prevState, "current state : ", this.state.count);          // this.setState({count:this.state.count+1});  // this leads to infinite loop          if (this.state.count < 10) {              this.setState({                  count: this.state.count + 1              })          }          console.log("snapShot : ", snapShot);  // this will always return undefined until you implement getSnapShot Method      }      render() {          console.log("I am Render");          return (              <div>                  <h2>Class Life Cycle</h2>                  <button type="button" onClick={() => this.setState({ count: this.state.count + 1 })}>Update Count {this.state.count}</button>              </div>          )      }  }  export default ClassLifeCycle; |

1. **shouldComponentUpdate() :**

shouldComponentUpdate is a lifecycle method in React class components that allows you to control whether a component should re-render or not. It is invoked before rendering when new props or state are being received. By default, shouldComponentUpdate returns true, meaning the component will re-render when its props or state change.

|  |
| --- |
| import React, { Component } from "react";  class ClassLifeCycle extends Component {      constructor() {          super();          console.warn("I am Constructor");          this.state = {              count: 0          }      }      componentDidMount() {          console.log("I am ComponentDidMount");      }      componentDidUpdate(prevProps, prevState, snapShot) {          console.log("I am ComponentDidUpdate");          console.log("This will return the previous state : ", prevState, "current state : ", this.state.count);          // this.setState({count:this.state.count+1});  // this leads to infinite loop          if (this.state.count < 10) {              this.setState({                  count: this.state.count + 1              })          }          console.log("snapShot : ", snapShot);  // this will always return undefined until you implement getSnapShot Method      }      // We can stop the execution of componentDid update if we have implement shouldComponentUpdate() return false;  **shouldComponentUpdate() {  // block the rendering**  **console.warn("I am shouldComponentUpdate");**  **if (this.state.count === 5) {**  **return false;**  **}**  **return true;**  **}**      render() {          console.log("I am Render");          return (              <div>                  <h2>Class Life Cycle</h2>                  <button type="button" onClick={() => this.setState({ count: this.state.count + 1 })}>Update Count {this.state.count}</button>              </div>          )      }  }  export default ClassLifeCycle; |

1. **ComponentWillUnmount :** componentWillUnmount is a lifecycle method in React class components that is invoked immediately before a component is unmounted and destroyed. This method is commonly used to perform cleanup tasks such as clearing timers, canceling network requests, or unsubscribing from external subscriptions to prevent memory leaks and ensure proper cleanup when a component is no longer needed.

**Note :** It runs befor the component removed

|  |
| --- |
| import React, { Component } from "react";  class ClassLifeCycle extends Component {      constructor() {          super();          console.warn("I am Constructor");          this.state = {              show: true          }      }        render() {          console.log("I am Render");          return (              <div>                  {                      this.state.show ?                          <Child />                          :                          "I am Parent Component"                  }                  <button type="button" onClick={() => this.setState({ show: !this.state.show })}>  Toggle To Child  </button>              </div>          )      }  }  export default ClassLifeCycle;  export class Child extends Component {      componentWillUnmount() {          console.error("I am ComponentWillUnmount");      }      render() {          console.log("i am child");          return (              <div>                  <h1>I am Child</h1>              </div>          )      }  } |

1. **Hooks :** Hooks in React.js are a feature introduced in React version **16.8** to allow functional components to have stateful logic and side effects that were previously only available in class components. Before the introduction of hooks, functional components were stateless and couldn't manage state or lifecycle methods without using class components.

**Notes :** hooks allow to use react life cycle methods like class component.

**We can not use the hooks in class components.**

1. **useState()** : useState is a React Hook that allows functional components to manage state. Before the introduction of hooks, state management was primarily done in class components using the setState method. However, with the useState hook, functional components can now have their own stateful logic.

|  |
| --- |
| import React, { useState } from 'react'  export default function UseStateHook01() {      const [state, setState] = useState(0);      return (          <div>              <button type='button' onClick={() => setState(state + 1)}>count {state}</button>          </div>      )  } |

1. **useEffect() :** The useEffect hook in React is used to perform side effects in functional components. Side effects can include data fetching, DOM manipulation, subscriptions, or any code that needs to run after the component renders. useEffect is similar to lifecycle methods like componentDidMount, componentDidUpdate, and componentWillUnmount in class components, but it's more flexible and can be used in functional components.

**Note :** It is used as a life cycle method.

It is the combination of componentDidMount,componentDidUpdate & componentWillUnmount

|  |
| --- |
| import React, { useEffect, useState } from 'react'  export default function UseEffectHook02() {      const [state, setState] = useState({ count: 0 });      const [name, setName] = useState("Piyush")  **1. No Dependency Passed : Runs on every render.**      useEffect(() => {          console.log("I am useEffect - 1");      })  **2. An Empty Array Of Dependency : Runs only on the first render**      useEffect(() => {          console.log("I am useEffect - 2");      }, [])  **3. Array of Dependency : Runs on the first render & any time any dependency value changes**      useEffect(() => {          console.log("I am useEffect - 3");      }, [name])      return (          <div>  <button type='button' onClick={() => setState({ count: state.count + 1 })}>  Update Count : {state.count}  </button>              <button type='button' onClick={() => setName("Shreyash")}>Update Name : {name}</button>          </div>      )  } |

1. **useLayoutEffect() :** The useLayoutEffect hook is similar to the useEffect hook, but it runs synchronously after all DOM mutations.

useLayoutEffect is a React hook that allows you to perform immediate DOM manipulation or read layout information from the DOM after a component has been rendered or updated, but before the browser paints those changes on the screen.

|  |
| --- |
| import React, { useRef } from 'react'  import { useLayoutEffect } from 'react';  import { useEffect } from 'react'  export default function UseLayoutEffect03() {      const ref = useRef("red");      useEffect(() => {          ref.current = "blue";          console.log("I am useEffect",ref);      })      useLayoutEffect(() => {          console.log("I am useLayoutEffect",ref);      })      return (          <div>          </div>      )  }  **OUTPUT :**  I am useLayoutEffect {current: 'red'}  I am useEffect {current: 'blue'} |

1. **useRef :**

The useRef Hook allows you to persist values between renders.

It can be used to store a mutable value that does not cause a re-render when updated.

It can be used to access a DOM element directly.

It consist of object with current.

If we tried to count how many times our application renders using the useState Hook, we would be caught in an infinite loop since this Hook itself causes a re-render.

To avoid this, we can use the useRef Hook.

|  |
| --- |
| import React, { useRef, useState } from 'react'  export default function UseRefHook04() {      const [input, setInput] = useState('');      const ref = useRef("red");      function handleSubmit(e) {          e.preventDefault();          console.log(input);          console.log("Input Value Using Ref : ", ref.current.value);          setInput("");          ref.current.focus();          ref.current.style.backgroundColor = 'red';          ref.current.style.color = "white";      }      return (          <div>              <form action="" onSubmit={handleSubmit}>                  <input type="text" ref={ref} value={input} onChange={(e) => setInput(e.target.value)} placeholder='Enter Fullname' />                  <button type='submit'>submit</button>              </form>          </div>      )  }  **Example : 02 : count re-render using useRef**  import React, { useEffect, useRef, useState } from 'react'  export default function UseRefHook04() {      const [state, setState] = useState(0);      const render = useRef(0);      useEffect(() => {          render.current = render.current + 1;      })      return (          <div>              <h3><b>Render : {render.current}</b></h3>              <button type='button' onClick={() => setState(state + 1)} >Update State : {state}</button>          </div>      )  } |

1. **useContext() :** the useContext hook in React allows you to access and use shared data or state (called context) from higher-level components without having to pass props through every level of the component tree.

|  |
| --- |
| **Index.js** |
| import React, { createContext } from 'react'  import ReactDOM from 'react-dom/client'  import App from './App.jsx'  import './index.css'  // Create a new context  export const MyContext = createContext()  const profile = {    fname: "Piyush",    lname: "Thaware",    email: "Piyush@gmail.com",    age: 23  }  ReactDOM.createRoot(document.getElementById('root')).render(    <MyContext.Provider value={profile}>      <App />    </MyContext.Provider>  ) |

|  |
| --- |
| **File where you want to use** |
| import React, { useContext } from 'react'  import { MyContext } from '../main'  export default function UseContextHook05() {      const mydata = useContext(MyContext);      return (          <div>              <ul>                  <li>Fname : {mydata?.fname}</li>                  <li>Lname : {mydata?.lname}</li>                  <li>Email : {mydata?.email}</li>                  <li>Age : {mydata?.age}</li>              </ul>          </div>      )  } |

1. **useReducer() :**

The useReducer Hook is similar to the useState Hook.

The useReducer hook in React is a powerful alternative to using the useState hook for managing complex state logic in functional components. It is particularly useful when you have state logic that involves multiple sub-values or when the next state depends on the previous state.

|  |
| --- |
| import React, { useReducer } from 'react'  export default function UseReducerHook06() {      function MyReducer(state, action) {          if (action.type === "INCREMENT") {              return state + 1;          }          else if (action.type === "DECREMENT") {              return state - 1;          }          else if (action.type === "INCREMENTBYPAYLOAD") {              return state + action.payload;          }          else if (action.type === "DECREMENTBYPAYLOAD") {              return state - action.payload;          }          else {              return state;          }      }      const initialValue = 100;      const [state, dispatch] = useReducer(MyReducer, initialValue);      return (          <div>              <h3><b>Count : {state}</b></h3>              <button type='button' onClick={() => dispatch({ type: "INCREMENT" })}>  Increment + 1  </button>              <button type='button' onClick={() => dispatch({ type: "DECREMENT" })}>  Decrement – 1  </button>              <button type='button' onClick={() => dispatch({ type: "INCREMENTBYPAYLOAD", payload: 10 })}>Increment + Payload</button>              <button type='button' onClick={() => dispatch({ type: "DECREMENTBYPAYLOAD", payload: 10 })}>Decrement - Payload</button>          </div>      )  } |

1. **useMemo() :** useMemo hook returns the memorize values which means it has already cache or stored the value somewhere.

**Kabhi kabhi kya hita hain a jo react app mein kuch function unnecessary call hote hai jiska koi need nhi hain and this leads to performance issue so to prevent the performance useMemo hook is used.It memoized and cache.**

**Note :** there is a very large difference between useMemo and useCallback even though both are used for the enhancing the performance of our application.

**While useMemo return a memomized value while useCallback return a memorized function.**

|  |
| --- |
| import React, { useMemo, useState } from 'react'  export default function UseMemoHook07() {      const [add, setAdd] = useState(0);      const [sub, setSub] = useState(100);      const multiple = useMemo(() => {          console.log("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");          return add \* 10;      }, [add])      return (          <div>              {multiple}              <br />              <button onClick={() => setAdd(add + 1)}>add {add}</button>              <button onClick={() => setSub(sub - 1)}>sub {sub}</button>          </div>      )  } |

1. **useCallback() :**

useCallback is a React hook used for memoizing functions in functional components.

It is particularly useful for optimizing performance by preventing unnecessary re-renders of child components that rely on these functions.

useCallback memoizes the provided function instance so that it is only re-created if its dependencies change.

This can be beneficial in scenarios where passing callbacks to child components and optimizing their renders is important.

|  |
| --- |
| **Parent.js** |
| import React, { useCallback, useState } from 'react'  import ChildComponent from './ChildComponent';  export default function UseCallback08() {      const [count, setCount] = useState(0);      const [state, setState] = useState(10);      const Learning = useCallback(function Learning() {          // some operations      }, [state])  // Dependency array, re-create the function only if count changes      return (          <div>              <button onClick={() => setCount(count + 1)}>Count : {count}</button>              <button onClick={() => setState(state + 2)}>state : {state}</button>              <ChildComponent learning={Learning} />          </div>      )  }  // Note : when you click any button the ChildComponent will re-render again to prevent unnecessary re-render we can use useMemo as  well as useCallback |

|  |
| --- |
| **Child.js** |
| import { React, memo } from "react";  // Since i have wrapped the child-component with memo but still it re-render this is due to referential equality  function ChildComponent({ learning }) {      console.log(">> I am child component");      return (          <>          </>      )  }  // here i have cover childcomponent with memo  export default memo(ChildComponent); |